Spring

IOC

Inverse of control（控制）

控制反转

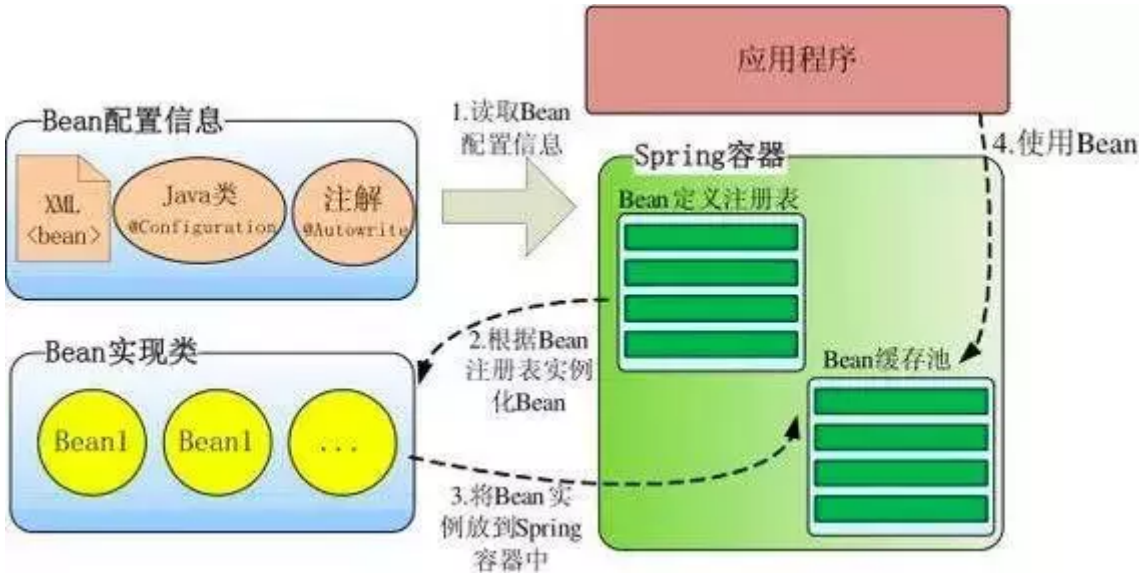
- 控制：实例的生成权
- 反转：将实例的控制权由应用程序反转给Spring

IOC的好处

1. 不用自己组装，拿来就用
2. 享受单例的好处，效率高，不浪费空间
3. 便于单元测试，方便切换mock组件
4. 便于进行AOP操作，对于使用者是透明的
5. 统一配置，便于修改

IOC的原理

- 原理就是通过Java的反射技术来实现的！通过反射我们可以获取类的所有信息(成员变量、类名等等等)！
- 再通过配置文件(xml)或者注解来描述类与类之间的关系
- 我们就可以通过这些配置信息和反射技术来构建出对应的对象和依赖关系了



1. 根据Bean配置信息在容器内部创建Bean定义注册表
2. 根据注册表加载、实例化bean、建立Bean与Bean之间的依赖关系
3. 将这些准备就绪的Bean放到Map缓存池中，等待应用程序调用

DI

Dependency Injection

依赖注入

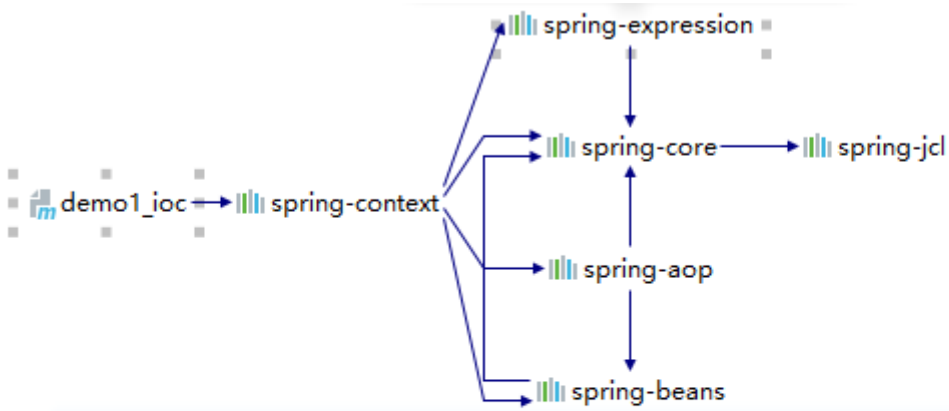
- 依赖：组件中所包含的必须的资源(成员变量)
- 注入：Spring 注入给应用程序

谁依赖谁，为什么依赖，谁注入谁，注入了什么

入门案例

引入依赖

5 Spring+1logging




```
<!--spring 5+1-->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.1.9.RELEASE</version>
</dependency>
```

Spring 配置文件

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

    <!-- bean definitions here -->
    <!--管理userDao的实例-->
    <bean id="userDao" class="com.cskaoyan.dao.UserDao"/>

</beans>
```

project→springframework→learn→core→appendix

class是必须的

单元测试

```
/*使用spring的形式写代码
* 控制反转
* 使用spring的配置文件（xml）来管理组件
* */
@Test
public void mytest2(){
    ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext( configLocation: "conf/application.xml");
    UserDao userDao1 = (UserDao) applicationContext.getBean( name: "userDao");
    UserDao userDao2 = applicationContext.getBean(UserDao.class);
    boolean b = userDao1.addUser( username: "songge",  password: "songgezhenshuai");
    System.out.println(b);
    Assert.assertEquals(userDao1,userDao2);
}
```

resources
conf
application.xml

加载的是classpath目录（sources和resources）

根据id获得实例

id是唯一的

根据类型

这个类型的实例在spring容器中只有一个

true通过

通过不同方式取出，是同一个实例

组件之间关系的维护

通过service和dao组件之间的关系，来维护组件之间的关系

```java
public class UserServiceImpl implements UserService {

    UserDao userDao;

    String name;

    String filePath;


    @Override
    public boolean register(String username, String password) {
        return userDao.addUser(username, password);
    }

    public UserDao getUserDao() {
        return userDao;
    }

    public void setUserDaozzz(UserDao userDao) {
        this.userDao = userDao;
    }
```

service的类和接口

```java
public interface UserService {

    public boolean register(String username,String password);
}
```

```xml
<!-- bean definitions here -->
<!--管理userDao的实例-->
<bean id="userDao" class="com.cskaoyan.dao.UserDaoImpl"/>

<bean class="com.cskaoyan.service.UserServiceImpl">
    <!--我们通常写的是成员变量名，实际上是setter方法-->
    <!--ref里的值是ioc容器中已经注册过的组件的id-->
    <property name="userDaozzz" ref="userDao"/>
    <property name="name" value="userServiceheihei"/>
</bean>
```

分别注册dao和service的实例

维护组件之间的关系

注册 bean

从容器中取出组件之前已经完成了实例化(生命周期)

默认是使用无参构造,也是最常用的

- 无参构造

```xml
<bean class="com.cskaoyan.dao.UserDaoImpl">
    <property name="filePath" value="d:/spring"/>
</bean>
```

默认使用无参构造

通过property标签和set方法搭配给成员变量赋值

```java
public class UserDaoImpl implements UserDao {

    String filePath;

    @Override
    public String hello(String content) {

        return "hello " + content;
    }

    public UserDaoImpl() {
        System.out.println("无参构造方法");
    }

    public String getFilePath() {
        return filePath;
    }

    public void setFilePath(String filePath) {
        this.filePath = filePath;
    }
}
```

- 有参构造

```xml
<bean class="com.cskaoyan.bean.ContructorBean">
    <constructor-arg name="usernamez" value="songge"/>
    <constructor-arg name="password" value="songgezhenshuai"/>
</bean>
```

constructor-arg的name属性是和有参构造方法的形参名对应的

```java
public class ContructorBean {

    String username;
    String password;

    public ContructorBean(String usernamez, String password) {
        this.username = usernamez;
        this.password = password;
    }
}
```

工厂

主要是整合其他组件

```
<!--实例工厂-->
<!--先将实例工厂注册在容器中-->
<bean id="instanceFactory" class="com.cskaoyan.factory.InstanceFactory"/>
<!--通过实例工厂的组件调用工厂方法去注册组件-->
<bean id="carFromInstance" factory-bean="instanceFactory" factory-method="create"/>
```

```
/**
 * 实例工厂
 */
public class InstanceFactory {
    public Car create() { return new Car(); }
}
```

```
//实例工厂获得car
InstanceFactory instanceFactory = new InstanceFactory();
Car car1 = instanceFactory.create();
```

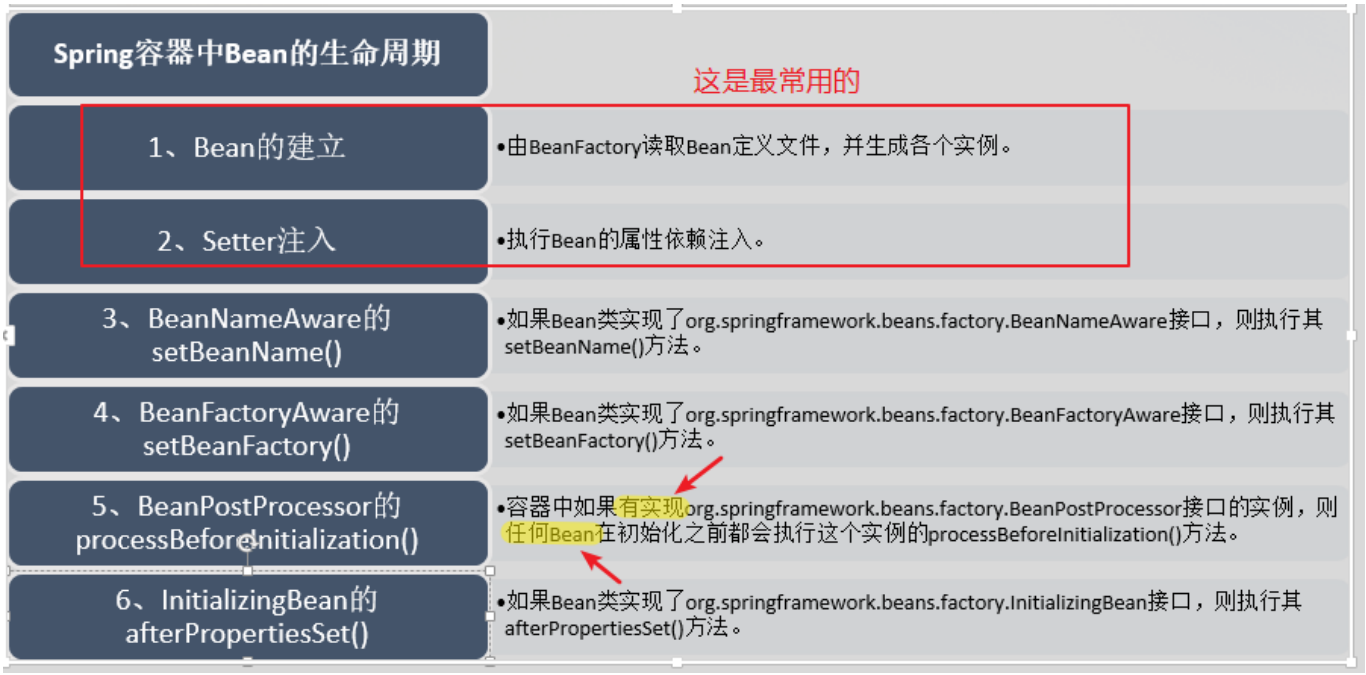
之前方式的单元测试

Scope

- Singleton：每一次从容器中取出的组件都是同一个组件(默认是单例的形式)
- Prototype：每一次从容器中取出的组件都是新的组件(每一次都去new了一下)

lifeCycle

重点部分：1、2、5、7、8、10



Demo：

```java
public class LifeCycleBean implements BeanNameAware, BeanFactoryAware, InitializingBean, DisposableBean {
    String fieldName;
    public LifeCycleBean() {
        System.out.println("1 构造方法");
    }
    public void setFieldName(String fieldName) {
        System.out.println("2 set方法");
        this.fieldName = fieldName;
    }
    @Override
    public void setBeanName(String s) {
        System.out.println("3 bean name aware " + s);
    }
    @Override
    public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
        System.out.println("4 bean factory aware" + beanFactory);
    }
    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("6 initializingbean ");
    }
    public void myinit(){
        System.out.println("7 init method");
    }
    @Override
    public void destroy() throws Exception {
        System.out.println("9 disposable bean");
    }
    public void mydestory(){
        System.out.println("10 destory method");
    }
```

容器中的BeanPostProcessor

```java
public class CustomBeanPostProcessor implements BeanPostProcessor {
    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
        System.out.println("5 beanpostprocessor 的before");
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        System.out.println("8 beanpostprocessor 的after");

        return bean;
    }
}
```

```
<bean class="com.cskaoyan.bean.CustomBeanPostProcessor"/>
```

将这个组件注册在容器中，全部组件初始化时均会执行到

Init-method和destory-method的使用

```
<bean class="com.cskaoyan.bean.LifeCycleBean" init-method="myinit" destroy-method="mydestory">
    <property name="fieldName" value="abc"/>
</bean>
```

方法名为class中包含的方法名

CollectionData

字符串

```
<bean class="com.cskaoyan.bean.CollectionBean">
    <property name="arrayData">
        <array>
            <value>array1</value>
            <value>array2</value>
            <value>array3</value>
        </array>
    </property>
    <property name="listData">
        <list>
            <value>list1</value>
            <value>list2</value>
            <value>list3</value>
        </list>
    </property>
    <property name="setData">
        <set>
            <value>set1</value>
            <value>set2</value>
            <value>set3</value>
        </set>
    </property>
    <property name="mapData">
        <map>
            <entry key="key1" value="value1"/>
            <entry key="key2" value="value2"/>
            <entry key="key3" value="value3"/>
        </map>
    </property>
```

```
@Data
public class CollectionBean {
    String[] arrayData;
    List<String> listData;
    Set<String> setData;
    Map<String,String> mapData;
    Properties properties;
}
```

使用value标签和value属性，封装基本类型或字符串

map使用了entry标签

Javabean类型

注解使用Spring

打开注解

```
<!-- com.cskaoyan这个包以及这个包下全部目录 -->
<context:component-scan base-package="com.cskaoyan"/>
```

组件注册类

Component

- @Component
- @Service
- @Repository

注入类

- @Autowired
- @Qualifier
- @Resource（name）
- @Value

@Scope

生命周期

- @PostConstruct
- @PreDestroy

单元测试

- @Runwith（SpringJUnit4ClassRunner.class）
- @ContextConfiguration(“classpath:application.xml”)
- @ContextConfiguration(classes={XXX.class})