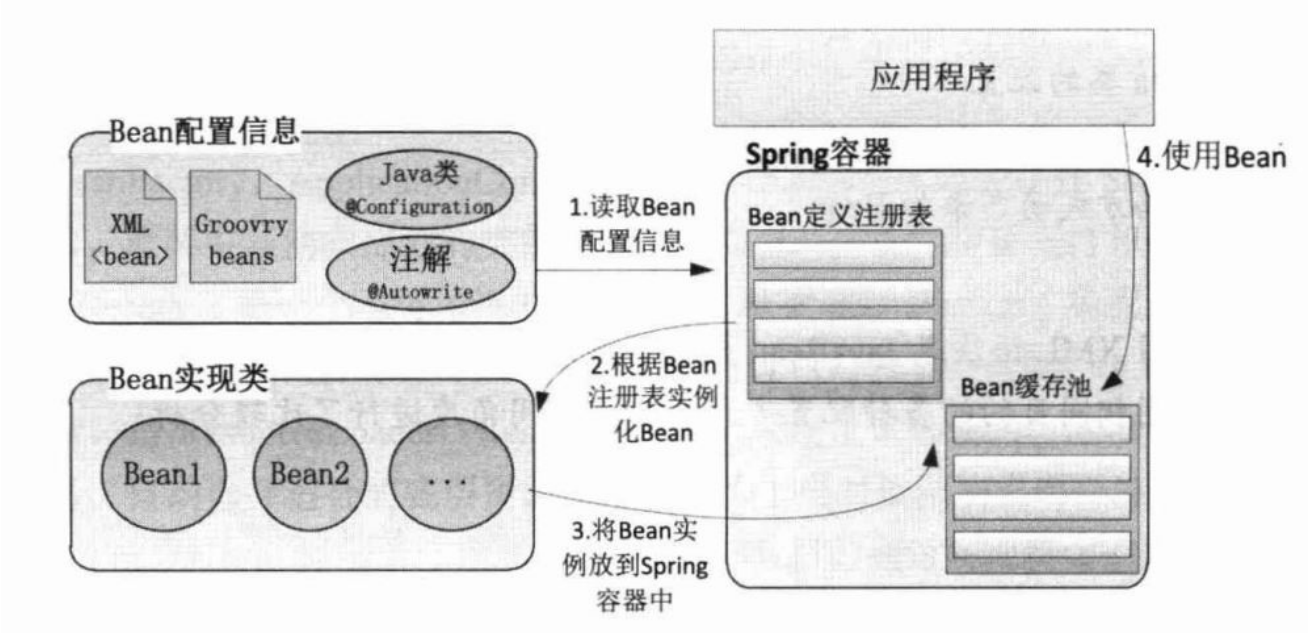
Spring IoC容器（重点）

> ## 本节目标
>
> 1.了解IoC容器的构建 2.掌握Bean的初始化 3.掌握Bean的依赖装配 4.了解Bean的作用域 5.了解Bean的生命周期

1. IoC容器构建

我们了解了Spring框架之后，就需要开启我们的Spring的学习之旅。我们知道Spring框架实际上是IoC的容器，同时也是实现了DI的功能。 接下来先了解如何创建一个SpringIoC容器。

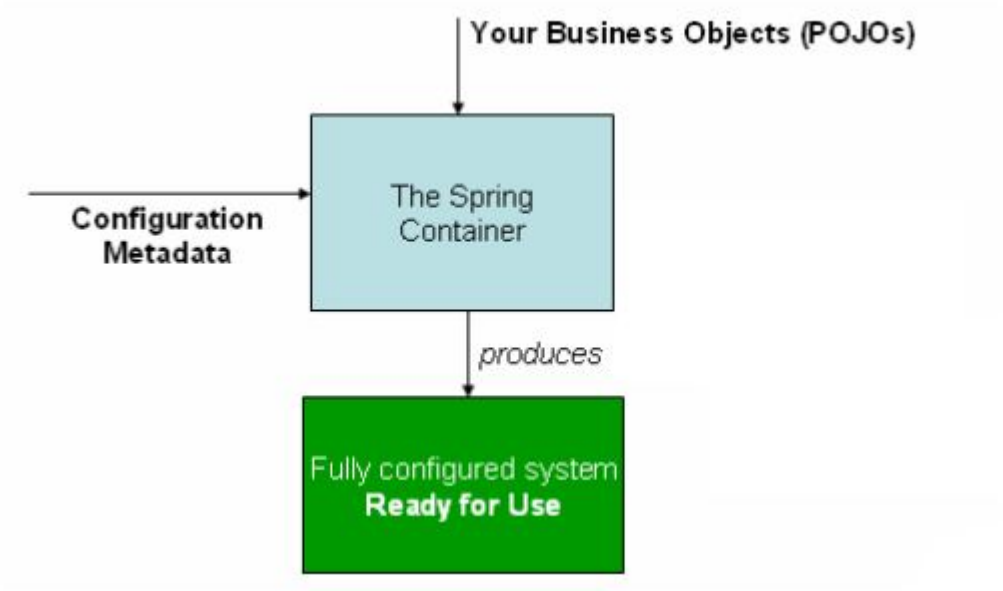
开始学习SpringIoC之前，我们先看一张图，这张图是关于Spring容器，Bean的配置信息，Bean的实现类以及应用程序之间的关系：



SpringIoC的构建方式有：

- **A** 基于XML配置的方式 （可以开启注解配置，通过注解配置Bean和装配工作）
- B 基于Goovy脚本配置的方式（也称：Spring's Groovy Bean Definition DSL ）
- **C** 基于Java Config的配置注解方式 （主要通过Configuration和Bean注解，以及其它注解 SpringBoot课程会涉及到）

目前使用较为广泛的是A和C两种方式，并且随着Spring生态的发展，方式C越来越流行，已知SpringBoot项目就是通过方式C构建完成的。



如上图所示：无论是哪种方式配置都属于**Configuration Metadata**配置，通过配置来完成容器的组装。

下面是使用这三种方式的直观示例：

方式A：

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

    <bean id="accountDao"
class="org.springframework.samples.jpetstore.dao.jpa.JpaAccountDao">
    </bean>

    <bean id="itemDao"
class="org.springframework.samples.jpetstore.dao.jpa.JpaItemDao">
    </bean>
</beans>
```

```
ApplicationContext context = new ClassPathXmlApplicationContext("services.xml",
"daos.xml");
```

方式B（目前使用很少，不作为学习的点）：

```
beans {
    dataSource(BasicDataSource) {
            driverClassName = "org.hsqldb.jdbcDriver"
            url = "jdbc:hsqldb:mem:grailsDB"
             username = "sa"
            password = ""
            settings = [mynew:"setting"]
```

```
    }

    sessionFactory(SessionFactory) {
            dataSource = dataSource
    }

    myService(MyService) {
            nestedBean = { AnotherBean bean ->
            dataSource = dataSource
        }
    }
}
```

```
ApplicationContext context = new GenericGroovyApplicationContext("services.groovy",
"daos.groovy");
```

方式C:

```
package com.bittech;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import java.util.HashMap;
import java.util.Map;

@Configuration
public class BlogConfigApplication {

    @Bean
    public String hello() {
        return "hello";
    }

    @Bean(name = "pointMap")
    public Map<String, String> javaEEPoint() {
        Map<String, String> points = new HashMap<>();
        points.put("1", "mybatis");
        points.put("2", "spring core");
        points.put("3", "spring mvc");
        points.put("4", "spring boot");
        return points;
    }

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(BlogConfigApplication.class);
        String hello = context.getBean(String.class);
        System.out.println("类型为: String 的Bean " + hello);
        Map<String,String> map = (Map<String, String>) context.getBean("pointMap");
        System.out.println(map);
```

```
    }
}
```

我们先学习XML的方式，再学习配置注解的方式。 先学习XML的方式主要有以下几个原因的考虑:

- XML是Spring最早支持的方式
- XML天生的严谨性，可以确保我们使用上避免一些错误
- XML可视化比较好，通过XML可是较为直观的看到整个容器中的Bean的关联关系
- XML的使用也是为了更好的理解Spring框架中的一些概念

2. Bean的实例化

在JavaSE中我们创建一个Bean的实例时，通常都是通过关键字`new`来完成，那么我们要在XML配置中怎么完成Bean的实例化呢？ Spring框架为我们提供了三种方式。

2.1 构造方法实例化

```
<bean id="exampleBean" class="examples.ExampleBean"/>
<bean name="anotherExample" class="examples.ExampleBeanTwo"/>
```

上面就是通过构造方法的方式实例化Bean,默认这个类是需要一个无参的构造方法。其中id和name都是Bean配置的属性，用来标识一个Bean。

2.2 静态工厂方法实例化

静态工厂方法顾名思义：就是通过一个类的静态（工厂）方法来创建Bean

```
public class ClientService {

    private static ClientService clientService = new ClientService();

    private ClientService() {}

    public static ClientService createInstance() {
        return clientService;
    }
}
```

```
<bean id="clientService" class="examples.ClientService"
factory-method="createInstance"/>
```

从上面的静态方法中我们就可以在对象返回之前来设置其属性，这样就完成了对象的创建。

2.3 实例工厂方法实例化

实例工厂方法和静态工厂方法比较相似，区别在于实例工厂方法是通过**已经在容器中的Bean**通过其方法实例化一个新的Bean。

```
<bean id="serviceLocator" class="examples.DefaultServiceLocator"></bean>

<bean id="clientService" factory-bean="serviceLocator"
factory-method="createClientServiceInstance"/>

public class DefaultServiceLocator {
    private static ClientService clientService = new ClientServiceImpl();

    public ClientService createClientServiceInstance() {
        return clientService;
    }
}
```

当然一个实例中可以有多个实例工厂方法的，我们可以在XML中以同样的方式配置更多的Bean。

3.Bean的依赖装配

本节是学习SpringCore的核心内容，掌握了各种Bean的依赖注入方法，就可以在实际项目中灵活选择合适的方式，而不至于失去`new`的机会，面对Bean的属性值设置而不知措施。

3.1 依赖装配方式

3.1.1 构造方法参数装配

在Spring中，可以使用"通过构造方法自动装配"，实际上是按构造函数的参数类型以及构造方法的参数顺序自动装配。 这意味着，如果一个bean的数据类型与其它bean的构造器参数的数据类型是相同的，那么将自动装配。当然如果装配过程中构造方法的参数具有歧义的时候，就需要我们指定类型，位置或者参数名的方式来告知Spring如何装配。

下面示例展示了通过构造方法参数装配的4种方式：

第一种：Bean的引用

```
<beans>
    <bean id="foo" class="com.bittech.springcore.assemble.Foo">
        <constructor-arg ref="bar"/>
        <constructor-arg ref="baz"/>
    </bean>
    <bean id="bar" class="com.bittech.springcore.assemble.Bar"/>
    <bean id="baz" class="com.bittech.springcore.assemble.Baz"/>
</beans>
```

第二种：根据参数类型

```java
package com.bittech.springcore.assemble;
public class ExampleBean {
    private int years;
    private String ultimateAnswer;
    public ExampleBean(int years, String ultimateAnswer) {
        this.years = years;
        this.ultimateAnswer = ultimateAnswer;
    }
}
```

```xml
<bean id="exampleBean" class="com.bittech.springcore.assemble">
    <constructor-arg type="int" value="7500000"/>
    <constructor-arg type="java.lang.String" value="42"/>
</bean>
```

第三种：根据参数位置

```xml
<bean id="exampleBean" class="com.bittech.springcore.assemble.ExampleBean">
    <constructor-arg index="0" value="7500000"/>
    <constructor-arg index="1" value="42"/>
</bean>
```

> 说明：如果构造方法有两个相同类型的参数时，可以使用index来设置依赖顺序，index是基于0开始的。

第四种：根据参数名称

```xml
<bean id="exampleBean" class="com.bittech.springcore.assemble.ExampleBean">
    <constructor-arg name="years" value="7500000"/>
    <constructor-arg name="ultimateAnswer" value="42"/>
</bean>
```

使用参数名称来查找构造方法中的参数。这种方式使用场景比较少，具有特殊限制，需要开启编译时参数名称保留。在JDK1.8之前采用`@ConstructorProperties`注解， JDK1.8之后我们可以在编译的时候添加`-parameters`来确保编译之后保留参数名。

3.1.2 Setter方法装配

容器在调用无参数构造方法或者无参工厂方法实例化Bean之后，调用Setter方法完成属性的装配。

```java
public class XmlShapeCompute {
    private Shape circular;
    public void setCircular(Shape circular) {
        this.circular = circular;
    }
}
```

```xml
<bean id="shapeCompute" class="com.bittech.springcore.xml.XmlShapeCompute">
     <property name="circular" ref="circular"/>
</bean>
```

3.2 依赖和配置详解

3.2.1 直接赋值

直接赋值主要针对`基本类型和String类型`，我们可以配置`properties`元素的`value`属性用来指定属性或者构造方法参数赋值。

开始之前，我们在maven中引入`commons-dbcp2`开源的数据库连接池库，在配置文件中描述创建数据源`BasicDataSource`对象的方式。

```
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-dbcp2</artifactId>
    <version>2.1.1</version>
</dependency>
```

```
<bean id="myDataSource" class="org.apache.commons.dbcp2.BasicDataSource" destroy-
method="close">
    <!-- results in a setDriverClassName(String) call -->
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/mydb"/>
    <property name="username" value="root"/>
    <property name="password" value="masterkaoli"/>
</bean>
```

3.2.2 idref和ref元素

在Spring中，idref属性和ref属性都可以用在constructor-arg元素和property元素中完成装配。

```
 <!-- Bean的引用-->
<bean id="bar" class="com.bittech.springcore.Bar"/>
<bean id="baz" class="com.bittech.springcore.Baz"/>
<bean id="foo" class="com.bittech.springcore.Foo">
    <constructor-arg ref="bar"/>
    <constructor-arg ref="baz"/>
</bean>

<!-- idref -->
<bean id="idRefExample" class="com.bittech.springcore.IdRefExample">
    <constructor-arg>
        <idref bean="bar"/>
    </constructor-arg>
</bean>
```

`idref`和`ref`的区别：

- idref装配的是目标bean的id而不是目标bean的实例，同时使用idref容器在部署的时候还会验证这个名称的bean是否真实存在。其实idref就跟value一样，**只是将某个字符串装配到属性或者构造函数中**，只不过装配的是某个Bean定义的id属性值。
- ref则是是将目标Bean定义的实例装配到属性或构造函数中。

3.2.3 内部bean

一个 `<bean/>` 元素定义在 `<property/>` 和 `<constructor-arg/>` 元素中我们称之为 `inner Bean` 。

```
<bean id="shapeCompute" class="com.bittech.springcore.xml.XmlShapeCompute">
        <property name="circular" ref="circular"/>
        <property name="rectangle" ref="rectangle"/>
        <property name="triangle">
            <!-- Inner Bean -->
            <bean class="com.bittech.springcore.common.impl.Triangle">
                <constructor-arg name="a" value="10"/>
                <constructor-arg name="b" value="24"/>
                <constructor-arg name="c" value="30"/>
            </bean>
        </property>
</bean>
```

3.3 其它

3.3.1 Collection

使用 `<list/>, <set/>, <map/>, <props/>` 元素可以设置Java集合类型（比如：List,Set,Map,Properties）的属性和参数。

```
<bean id="moreComplexObject" class="example.ComplexObject">
    <!-- results in a setAdminEmails(java.util.Properties) call -->
    <property name="adminEmails">
        <props>
                <prop key="administrator">administrator@example.org</prop>
                <prop key="support">support@example.org</prop>
                <prop key="development">development@example.org</prop>
        </props>
    </property>

    <!-- results in a setSomeList(java.util.List) call -->
    <property name="someList">
        <list>
            <value>a list element followed by a reference</value>
            <ref bean="myDataSource" />
        </list>
    </property>

    <!-- results in a setSomeMap(java.util.Map) call -->
    <property name="someMap">
        <map>
            <entry key="an entry" value="just some string"/>
            <entry key ="a ref" value-ref="myDataSource"/>
        </map>
    </property>

    <!-- results in a setSomeSet(java.util.Set) call -->
    <property name="someSet">
```

```
        <set>
            <value>just some string</value>
            <ref bean="myDataSource" />
        </set>
    </property>
</bean>
```

在使用集合元素时如Map的key和value,集合的value可以使用以下元素：

```
bean | ref | idref | list | set | map | props | value | null
```

3.3.2 Null和空字符串值

- 空字符串赋值

```
<bean class="ExampleBean">
    <property name="email" value=""/>
</bean>
```

```
//上述配置等价与Java代码中的方法赋值参数为""
exampleBean.setEmail("");
```

- `<null/>`元素处理`null`值

```
<bean class="ExampleBean">
    <property name="email">
        <null/>
    </property>
</bean>
```

```
//上述配置等价与Java代码中的方法赋值参数为null
exampleBean.setEmail(null);
```

3.3.3 延迟初始化

Spring默认时容器启动时将所有的Bean都初始化完成，设置延迟初始化时告诉容器在第一次使用Bean的时候，完成初始化操作，而不是启动时。

```
<!-- Lazy-initialized -->
<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true"/>
```

3.3.3 自动装配（了解）

在课程前面我们都讲的是手工完成Bean之间的依赖关系的建立（装配），Spring还提供了自动装配的能力。Spring的自动装配功能的定义：无须在Spring配置文件中描述Bean之间的依赖关系（如配置`<property>`、`<constructor-arg>`），IoC容器会自动建立Bean之间的关联关系。

手动注入：

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

  <bean id="customerDAO" class="com.hebeu.customer.dao.JdbcCustomerDAO">
        <property name="dataSource" ref="dataSource" />
  </bean>
</beans>
```

自动装配:

案例一:

```
public class Bar {
}
public class Customer {
    private Bar bar;
    public Customer(Bar bar) {
        this.bar = bar;
    }
    public Bar getBar() {
        return bar;
    }
    @Override
    public String toString() {
        return "Customer{" +
                "bar=" + bar +
                '}';
    }
}
```

```
<bean id="bar" class="com.bittech.springcore.assemble.Bar"></bean>
<bean id="customer" class="com.bittech.springcore.scope.Customer"
autowire="constructor"/>
```

案例二:

```
public class Bar {
}
public class Customer {
    private Bar bar;
    public void setBar(Bar bar) {
        this.bar = bar;
    }
    public Bar getBar() {
        return bar;
    }
    @Override
    public String toString() {
        return "Customer{" +
```

```
                "bar=" + bar +
                '}';
    }
}
```

```
<bean id="bar" class="com.bittech.springcore.assemble.Bar"></bean>
<bean id="customer" class="com.bittech.springcore.scope.Customer" autowire="byType"/>
```

案例三：

```
<bean id="bar" class="com.bittech.springcore.assemble.Bar"></bean>
<bean id="customer" class="com.bittech.springcore.scope.Customer" autowire="byName"/>
```

在Spring中，支持4种自动装配模，（3.x版本之前是5种模式）：

- no：缺省情况下，自动配置是通过“ref”属性手动设定
- byName： 根据属性名称自动装配。如果一个bean的名称和其它bean属性的名称是一样的，将会自装配它
- byType： 按数据类型自动装配。如果一个bean的数据类型是用其它bean属性的数据类型，兼容并自动装配它（存在问题：如果同一个类型bean有多个，就无法进行自动装配）
- constructor：构造函数参数的通过byType方式自动装配

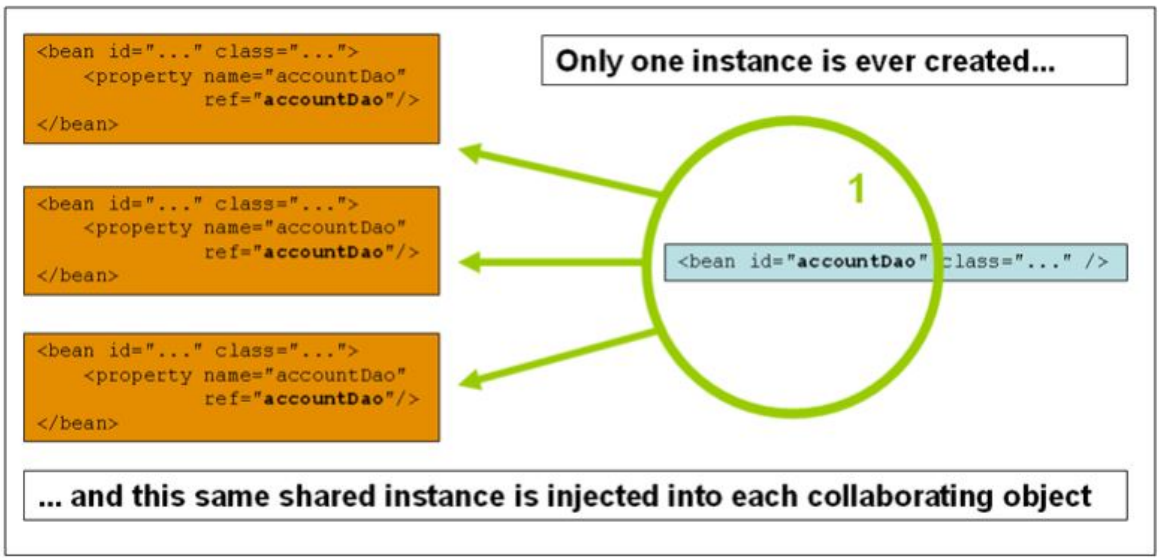
> 说明：自动装配虽好，但不利于阅读和直观展示依赖关系。

4. Bean的作用域

Bean的作用域主要是描述一个Bean实例在Spring IoC容器中的存在状态，比如根据不同请求，线程而存在多个。

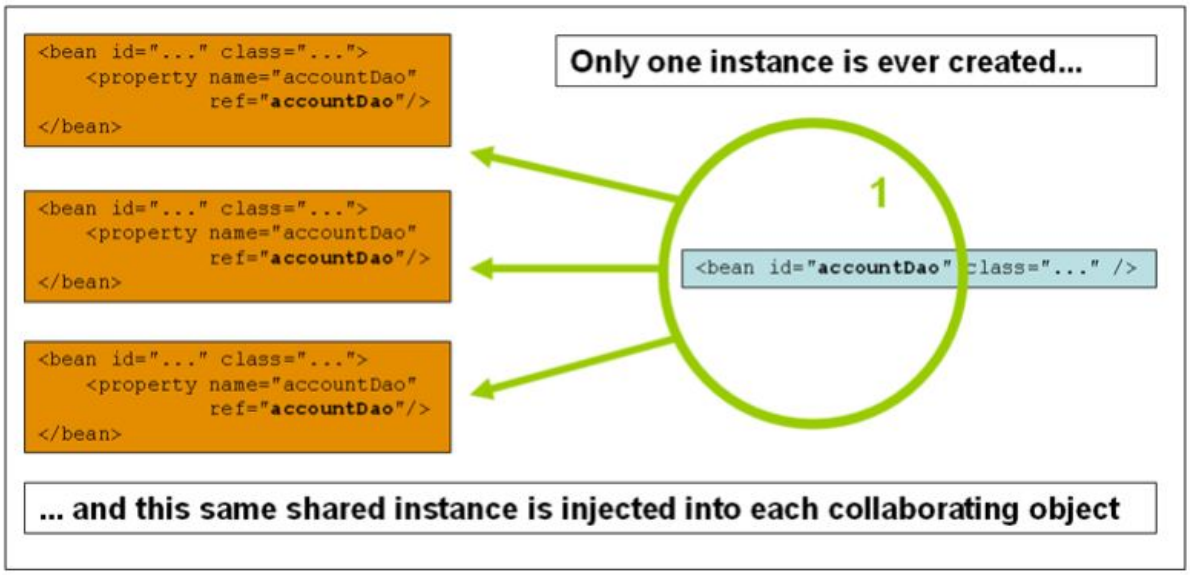
singleton:

- 描述：该作用域下的Bean在IoC容器中只存在一个实例，所有对象对其的引用都返回同一个
- 场景：通常**无状态**的Bean使用该作用域
- 备注：**Spring默认选择该作用域**



prototype:

- 描述：每次对该作用域下的Bean的请求都会创建新的实例
- 场景：通常**有状态**的Bean使用该作用域



request

- 描述：每次http请求会创建新的Bean实例，类似于prototype
- 场景：一次http的请求和响应的共享Bean
- 备注：限定SpringMVC中使用

session

- 描述：在一个http session中，定义一个Bean实例
- 场景：用户回话的共享Bean, 比如：记录一个用户的登陆信息
- 备注：限定SpringMVC中使用

global session

- 描述：类似与http session，但限于protlet web应用可用
- 场景：所有构成某个portlet web应用的各种不同的portlet所共享
- 备注：限定protlet web应用，你在编写一个标准的基于Servlet的web应用，并且定义了一个或多个具有global session作用域的bean，http session作用域，并且不会引起任何错误

application session

- 描述：在一个http servlet Context中，定义一个Bean实例
- 场景：Web应用的上下文信息，比如：记录一个应用的共享信息
- 备注：限定SpringMVC中使用

5. 基于注解配置

无论是通过XML还是注解配置，它们都是表达Bean定义的载体，其本质都是为Spring容器提供Bean定义的信息。在表现形式上都是将XML定义的内容通过注解进行描述。

Spring容器启动成功的三大要素：

- Bean定义信息

- Bean实现类
- Spring本身

5.1 使用注解定义Bean

采用基于XML的配置，则Bean定义信息和Bean实现类本身事分离的，而采用基于注解的配置文件，则Bean的定义信息事通过在Bean实现类上标注注解实现的。

基于注解：

```
package com.bittech.common.component;
import org.springframework.stereotype.Component;
@Component
public class ChineseCurrency {
    //more code
}
```

基于XML配置：

```
<bean class="com.bittech.common.component.ChineseCurrency"/>
```

上面基于注解和下面基于XML配置是等价的。使用@Component注解标识的ChineseCurrency类会被Spring容器识别，自动转为被容器管理的Bean。

> **注意：**基于注解配置我们需要在XML配置文件中添加：`<context:component-scan/>`来告诉Spring容器开启注解Bean的扫描

除了@Component注解外，Spring还提供了等价与@Component的几个注解：

- @Repository: 用户DAO实现了进行标识
- @Serivce: 用于Service实现类进行标识
- @Controller: 用于Contorller实现类进行标识

之所以在@Component之外提供这3个注解，主要是为了标识类的用途，使我们可以一眼看出来Bean的用途。建议使用特定的注解类标识类。

5.2 使用注解装配

5.2.1 @Autowired自动装配

```
@Component
public class ShapeService {
    @Autowired
    private Shape shape;
}
```

`@Service`将ShapeService标注为一个Bean,通过@Autowired注入Shape的Bean。@Autowired默认是按类型(byType)匹配的方式在容器中查找匹配的Bean，当有且仅有一个匹配的Bean时，Spring将其注入@Autowired标注的变量中。

`@Autowired` 的required属性值默认为 `true` ,如果容器中没有找到和标注变量类型匹配的Bean,那么Spring容器启动时将报 `NoSuchBeanDefinitionException` 异常。如果希望Spring即使找不到匹配的Bean完成注入时也不要抛出异常，那么可以使用 `@Autowired(required=false)` 进行标注。

5.2.2 @Qualifier注入指定Bean

```
@Component
public class ShapeService {
    @Autowired
    @Qualifier(value = "circular")
    private Shape shape;
}
```

假设容器中有三个类型为Shape的Bean，分别名为：circular，rectangle，triangle，则上面代码中会注入circular的Bean。

5.2.3 @Scope, @PostConstruct, @PreDestroy

```
@Component
@Scope(scopeName = "prototype")
public class CardComponent {

    @PostConstruct
    public void init() {
        System.out.println("Bean初始化之后调用");
    }

    @PreDestroy
    public void destroy() {
        System.out.println("Bean销毁之前调用");
    }
}
```

上面的注解方式等价与XML中的配置

```
<bean class="com.bittech.common.component.CardComponent" scope="prototype" init-
method="init" destroy-method="destroy"/>
```

> **注意：** `@PostConstruct @PreDestroy` 时 `JSR-250(Java Specification Requests)` 中定义的属于Java的类。

6. Bean的配置比较

6.1 配置的方式比较

	基于 XML 配置	基于注解配置	基于 Java 类配置	基于 Groovy DSL 配置
Bean 定义	在 XML 文件中通过 <bean> 元素定义 Bean，如： `<bean class="com.smart.UserDao"/>`	在 Bean 实现类处通过标注@Compoent 或衍型类（@Repository、@Service 及 @Controller ）定义 Bean	在标注了@Configuration 的 Java 类中，通过在类方法上标注@Bean 定义一个 Bean。方法必须提供 Bean 的实例化逻辑	在 Groovy 文件中通过 DSL 定义 Bean，如： `userDao(UserDao)`

续表

	基于 XML 配置	基于注解配置	基于 Java 类配置	基于 Groovy DSL 配置
Bean 名称	通过<bean>的 id 或 name 属性定义，如： `<bean id="userDao" class="com.smart.UserDao "/>` 默认名称为 com.smart.UserDao#0	通过注解的 value 属性定义，如@Component ("userDao")。默认名称为小写字母开头的类名（不带包名）userDao	通过@Bean 的 name 属性定义，如@Bean("userDao")。默认名称为方法名	通过 Groovy 的 DSL 定义 Bean 的名称（Bean 的类型，Bean 构建函数参数），如： `logonService(LogonService, userDao)`
Bean 注入	通过<property>子元素或通过 p 命名空间的动态属性，如 p:userDao-ref="userDao" 进行注入	通过在成员变更或方法入参处标注@Autowired，按类型匹配自动注入。还可以配合使用@Qualifier 按名称匹配方式注入	比较灵活，可以在方法处通过@Autowired 使方法入参绑定 Bean，然后在方法中通过代码进行注入；还可通过调用配置类的@Bean 方法进行注入	比较灵活，可以在方法处通过 ref()方法进行注入，如 ref("logDao")
Bean 生命过程方法	通过<bean>的 init-method 和 destory-method 属性指定 Bean 实现类的方法名。最多只能指定一个初始化方法和一个销毁方法	通过在目标方法上标注 @PostConstruct 和 @PreDestroy 注解指定初始化或销毁方法，可以定义任意多个	通过@Bean 的 initMethod 或 destoryMethod 指定一个初始化或销毁方法。 对于初始化方法来说，可以直接在方法内部通过代码的方式灵活定义初始化逻辑	通过 bean-> bean.initMethod 或 bean.destoryMethod 指定一个初始化或销毁方法
Bean 作用范围	通过<bean>的 scope 属性指定，如： `<bean class=" com.smart.UserDao " scope=" prototype"/>`	通过在类定义处标注@Scope 指定，如@Scope ("prototype")	通过在 Bean 方法定义处标注@Scope 指定	通过 bean-> bean.scope = "prototype"指定
Bean 延迟初始化	通过 <bean> 的 lazy-init 属性指定，默认为 default，继承于 <beans> 的 default-lazy-init 设置，该值默认为 false	通过在类定义处标注 @Lazy 指定，如@Lazy (true)	通过在 Beam 方法定义处标注@Lazy 指定	通过 bean-> bean.lazyInit = true 指定

6.2 配置的应用场景比较

	基于 XML 配置	基于注解配置	基于 Java 类配置	基于 Groovy DSL 配置
适用场景	（1）Bean 实现类来源于第三方类库，如 DataSource、JdbcTemplate 等，因无法在类中标注注解，所以通过 XML 配置方式较好。 （2）命名空间的配置，如 aop、context 等，只能采用基于 XML 的配置	Bean 的实现类是当前项目开发的，可以直接在 Java 类中使用基于注解的配置	基于 Java 类配置的优势在于可以通过代码方式控制 Bean 初始化的整体逻辑。如果实例化 Bean 的逻辑比较复杂，则比较适合基于 Java 类配置的方式	基于 Groovy DSL 配置的优势在于可以通过 Groovy 脚本灵活控制 Bean 初始化的过程。如果实例化 Bean 的逻辑比较复杂，则比较适合基于 Groovy DSL 配置的方式

总结

知识块	知识点	分类	掌握程度
IoC容器构建	1.通过XML方式构建 2. 通过Java Config方式构建	实战型	掌握
Bean的实例化	1. 构造方法实例化 2. 静态工厂实例化 3. 实例工程方法实例化	实战型	掌握
Bean的依赖装配	1. 构造方法装配 2.Setter方法装配 3.特殊类型装配	实战型	掌握
Bean的作用域	1.Bean作用域配置和使用	理解性	掌握
Bean的注解配置	1.注解定义Bean 2.注解装配Bean 3.注解声明Bean的作用域 4.注解声明Bean的生命周期	实战型	掌握

实践

案例：改造JDBC项目成为Spring项目 功能： 1.便签组的查询 2.便签组的删除 3.便签组的修改 4.便签组的创建 步骤： 1.项目复制，修改名称 2.添加pom依赖 3.调整代码的包 4.添加Spring的配置文件 5.编写相应的代码 6.验证方法