

```

    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse
resp) throws ServletException, IOException {

        // 设置请求体的字符编码
        req.setCharacterEncoding("UTF-8");

        String fromCardNo = req.getParameter("fromCardNo");
        String toCardNo = req.getParameter("toCardNo");
        String moneyStr = req.getParameter("money");
        int money = Integer.parseInt(moneyStr);

        Result result = new Result();

        try {

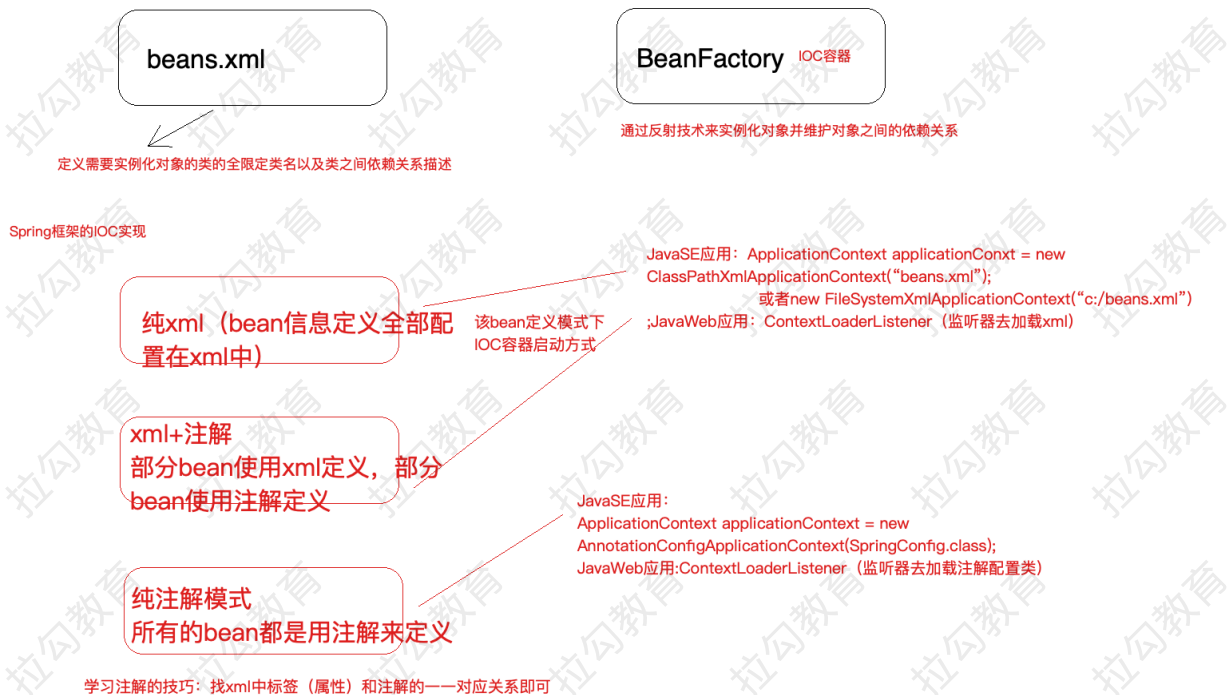
            // 2. 调用service层方法
            transferService.transfer(fromCardNo,toCardNo,money);
            result.setStatus("200");
        } catch (Exception e) {
            e.printStackTrace();
            result.setStatus("201");
            result.setMessage(e.toString());
        }

        // 响应
        resp.setContentType("application/json;charset=utf-8");
        resp.getWriter().print(JsonUtils.object2Json(result));
    }
}

```

第四部分 Spring IOC 应用

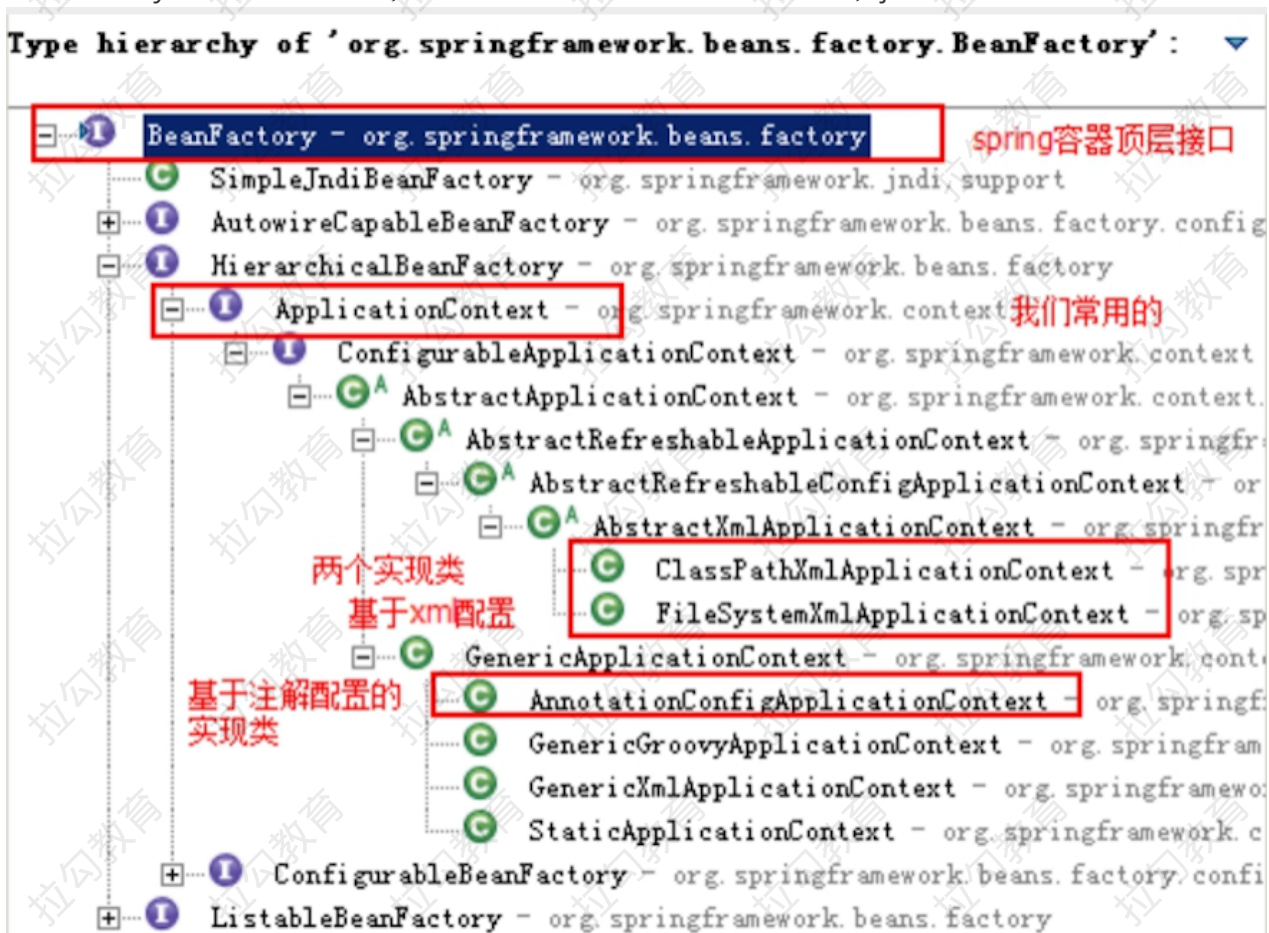
第1节 Spring IoC基础



1.1 BeanFactory与ApplicationContext区别

BeanFactory是Spring框架中IoC容器的顶层接口,它只是用来定义一些基础功能,定义一些基础规范,而ApplicationContext是它的一个子接口,所以ApplicationContext是具备BeanFactory提供的全部功能的。

通常,我们称BeanFactory为SpringIOC的基础容器,ApplicationContext是容器的高级接口,比BeanFactory要拥有更多的功能,比如说国际化支持和资源访问(xml, java配置类)等等



启动 IoC 容器的方式

- Java环境下启动IoC容器
 - ClassPathXmlApplicationContext：从类的根路径下加载配置文件（推荐使用）
 - FileSystemXmlApplicationContext：从磁盘路径上加载配置文件
 - AnnotationConfigApplicationContext：纯注解模式下启动Spring容器
- Web环境下启动IoC容器
 - 从xml启动容器

```
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
  <display-name>Archetype Created Web Application</display-name>

  <!--配置Spring ioc容器的配置文件-->
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext.xml</param-value>
  </context-param>
  <!--使用监听器启动Spring的IOC容器-->
  <listener>
    <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-
class>
    </listener>
  </web-app>
```

- 从配置类启动容器

```
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
  <display-name>Archetype Created Web Application</display-name>

  <!--告诉ContextloaderListener知道我们使用注解的方式启动ioc容器-->
  <context-param>
    <param-name>contextClass</param-name>
    <param-
value>org.springframework.web.context.support.AnnotationConfigWebAppli
cationContext</param-value>
  </context-param>
```

```

<!--配置启动类的全限定类名-->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>com.lagou.edu.SpringConfig</param-value>
</context-param>
<!--使用监听器启动Spring的IOC容器-->
<listener>
    <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-
class>
    </listener>
</web-app>

```

1.2 纯xml模式

本部分内容我们不采用——讲解知识点的方式，而是采用Spring IoC 纯 xml 模式改造我们前面手写的IoC 和 AOP 实现，在改造的过程中，把各个知识点串起来。

- xml 文件头

```

<?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">

```

- 实例化Bean的三种方式

- 方式一：使用无参构造函数

在默认情况下，它会通过反射调用无参构造函数来创建对象。如果类中没有无参构造函数，将创建失败。

```

<!--配置service对象-->
<bean id="userService" class="com.lagou.service.impl.TransferServiceImpl">
</bean>

```

- 方式二：使用静态方法创建

在实际开发中，我们使用的对象有些时候并不是直接通过构造函数就可以创建出来的，它可能在创建的过程中会做很多额外的操作。此时会提供一个创建对象的方法，恰好这个方法是static修饰的方法，即是此种情况。

例如，我们在做jdbc操作时，会用到java.sql.Connection接口的实现类，如果是mysql数据库，那么用的就是JDBC4Connection，但是我们会去写 `JDBC4Connection connection = new JDBC4Connection()`，因为我们要注册驱动，还要提供URL和凭证信息，用 `DriverManager.getConnection` 方法来获取连接。

那么在实际开发中，尤其早期的项目没有使用Spring框架来管理对象的创建，但是在设计时使用了工厂模式解耦，那么当接入spring之后，工厂类创建对象就具有和上述例子相同特征，即可采用此种方式配置。

<!--使用静态方法创建对象的配置方式-->

```
<bean id="userService" class="com.lagou.factory.BeanFactory"
      factory-method="getTransferService"></bean>
```

- 方式三：使用实例化方法创建

此种方式和上面静态方法创建其实类似，区别是用于获取对象的方法不再是static修饰的了，而是类中的一个普通方法。此种方式比静态方法创建的使用几率要高一些。

在早期开发的项目中，工厂类中的方法有可能是静态的，也有可能是非静态方法，当是非静态方法时，即可采用下面的配置方式：

<!--使用实例方法创建对象的配置方式-->

```
<bean id="beanFactory"
      class="com.lagou.factoryinstancemethod.BeanFactory"></bean>
<bean id="transferService" factory-bean="beanFactory" factory-
      method="getTransferService"></bean>
```

- Bean的X及生命周期

- 作用范围的改变

在spring框架管理Bean对象的创建时，Bean对象默认都是单例的，但是它支持配置的方式改变作用范围。作用范围官方提供的说明如下图：

Scope	Description
singleton	(Default) Scopes a single bean definition to a single object instance for each Spring IoC container.
prototype	Scopes a single bean definition to any number of object instances.
request	Scopes a single bean definition to the lifecycle of a single HTTP request. That is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext.
session	Scopes a single bean definition to the lifecycle of an HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.
application	Scopes a single bean definition to the lifecycle of a ServletContext. Only valid in the context of a web-aware Spring ApplicationContext.
websocket	Scopes a single bean definition to the lifecycle of a WebSocket. Only valid in the context of a web-aware Spring ApplicationContext.

在上图中提供的这些选项中，我们实际开发中用到最多的作用范围就是singleton（单例模式）和prototype（原型模式，也叫多例模式）。配置方式参考下面的代码：

<!--配置service对象-->

```
<bean id="transferService"
      class="com.lagou.service.impl.TransferServiceImpl" scope="singleton">
</bean>
```

- 不同作用范围的生命周期

单例模式：singleton

对象出生：当创建容器时，对象就被创建了。

对象活着：只要容器在，对象一直活着。

对象死亡：当销毁容器时，对象就被销毁了。

一句话总结：单例模式的bean对象生命周期与容器相同。

多例模式：prototype

对象出生：当使用对象时，创建新的对象实例。

对象活着：只要对象在使用中，就一直活着。

对象死亡：当对象长时间不用时，被java的垃圾回收器回收了。

一句话总结：多例模式的bean对象，spring框架只负责创建，不负责销毁。

● Bean标签属性

在基于xml的IoC配置中，bean标签是最基础的标签。它表示了IoC容器中的一个对象。换句话说，如果一个对象想让spring管理，在XML的配置中都需要使用此标签配置，Bean标签的属性如下：

id属性：用于给bean提供一个唯一标识。在一个标签内部，标识必须唯一。

class属性：用于指定创建Bean对象的全限定类名。

name属性：用于给bean提供一个或多个名称。多个名称用空格分隔。

factory-bean属性：用于指定创建当前bean对象的工厂bean的唯一标识。当指定了此属性之后，class属性失效。

factory-method属性：用于指定创建当前bean对象的工厂方法，如配合factory-bean属性使用，则class属性失效。如配合class属性使用，则方法必须是static的。

scope属性：用于指定bean对象的作用范围。通常情况下就是singleton。当要用到多例模式时，可以配置为prototype。

init-method属性：用于指定bean对象的初始化方法，此方法会在bean对象装配后调用。必须是一个无参方法。

destroy-method属性：用于指定bean对象的销毁方法，此方法会在bean对象销毁前执行。它只能为scope是singleton时起作用。

● DI 依赖注入的xml配置

- 依赖注入分类

■ 按照注入的方式分类

构造函数注入：顾名思义，就是利用带参构造函数实现对类成员的数据赋值。

set方法注入：它是通过类成员的set方法实现数据的注入。（使用最多的）

■ 按照注入的数据类型分类

基本类型和String

注入的数据类型是基本类型或者是字符串类型的数据。

其他Bean类型

注入的数据类型是对象类型，称为其他Bean的原因是，这个对象是要求出现在IoC容器中的。那么针对当前Bean来说，就是其他Bean了。

复杂类型（集合类型）

注入的数据类型是Aarry, List, Set, Map, Properties中的一种类型。

- 依赖注入的配置实现之构造函数注入 顾名思义，就是利用构造函数实现对类成员的赋值。它的使用要求是，类中提供的构造函数参数个数必须和配置的参数个数一致，且数据类型匹配。同时需要注意的是，当没有无参构造时，则必须提供构造函数参数的注入，否则Spring框架会报错。

```
/**
 * @author 应癡
 */
public class JdbcAccountDaoImpl implements AccountDao {

    private ConnectionUtils connectionUtils;

    private String name;
    private int sex;
    private float money;

    public JdbcAccountDaoImpl(ConnectionUtils connectionUtils, String name, int sex, float money) {
        this.connectionUtils = connectionUtils;
        this.name = name;
        this.sex = sex;
        this.money = money;
    }
}

<bean id="accountDao" class="com.lagou.edu.dao.impl.JdbcAccountDaoImpl" scope="singleton" init-method="init" destroy-method="destory">

    <!--set注入使用property标签，如果注入的是另外一个bean那么使用ref属性，如果注入的是普通值那么使用的是value属性-->
    <!--<property name="ConnectionUtils" ref="connectionUtils"/>
    <property name="name" value="zhangsan"/>
    <property name="sex" value="1"/>
    <property name="money" value="100.3"/>-->

    <!--<constructor-arg index="0" ref="connectionUtils"/>
    <constructor-arg index="1" value="zhangsan"/>
    <constructor-arg index="2" value="1"/>
    <constructor-arg index="3" value="100.5"/>-->

    <!--name: 按照参数名称注入，index按照参数索引位置注入-->
    <constructor-arg name="connectionUtils" ref="connectionUtils"/>
    <constructor-arg name="name" value="zhangsan"/>
    <constructor-arg name="sex" value="1"/>
    <constructor-arg name="money" value="100.6"/>
</bean>
```

在使用构造函数注入时，涉及的标签是 **constructor-arg**，该标签有如下属性：

name：用于给构造函数中指定名称的参数赋值。

index：用于给构造函数中指定索引位置的参数赋值。

value：用于指定基本类型或者String类型的数据。

ref：用于指定其他Bean类型的数据。写的是其他bean的唯一标识。

- 依赖注入的配置实现之set方法注入

顾名思义，就是利用字段的set方法实现赋值的注入方式。此种方式在实际开发中是使用最多的注入方式。

```

/**
 * @author 应癡
 */
public class JdbcAccountDaoImpl implements AccountDao {

    private ConnectionUtils connectionUtils;
    private String name;
    private int sex;
    private float money;

    public void setConnectionUtils(ConnectionUtils connectionUtils) {
        this.connectionUtils = connectionUtils;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setSex(int sex) {
        this.sex = sex;
    }

    public void setMoney(float money) {
        this.money = money;
    }
}

<bean id="accountDao" class="com.lagou.edu.dao.impl.JdbcAccountDaoImpl" scope="singleton" init-method="init" destroy-method="destory">

    <!--set注入使用property标签, 如果注入的是另外一个bean那么使用ref属性, 如果注入的是普通值那么使用的是value属性-->
    <!--<property name="ConnectionUtils" ref="connectionUtils"/>
    <property name="name" value="zhangsan"/>
    <property name="sex" value="1"/>
    <property name="money" value="100.3"/>-->

    <!--<constructor-arg index="0" ref="connectionUtils"/>
    <constructor-arg index="1" value="zhangsan"/>
    <constructor-arg index="2" value="1"/>
    <constructor-arg index="3" value="100.5"/>-->

    <!--name: 按照参数名称注入, index按照参数索引位置注入-->
    <constructor-arg name="connectionUtils" ref="connectionUtils"/>
    <constructor-arg name="name" value="zhangsan"/>
    <constructor-arg name="sex" value="1"/>
    <constructor-arg name="money" value="100.6"/>

```

在使用set方法注入时, 需要使用 **property** 标签, 该标签属性如下:

name: 指定注入时调用的set方法名称。(注: 不包含set这三个字母,druid连接池指定属性名称)

value: 指定注入的数据。它支持基本类型和String类型。

ref: 指定注入的数据。它支持其他bean类型。写的是其他bean的唯一标识。

- 复杂数据类型注入 首先, 解释一下复杂类型数据, 它指的是集合类型数据。集合分为两类, 一类是List结构(数组结构), 一类是Map接口(键值对)。

接下来就是注入的方式的选择, 只能在构造函数和set方法中选择, 我们的示例选用set方法注入。


```
private String[] myArray;  
private Map<String,String> myMap;  
private Set<String> mySet;  
private Properties myProperties;  
  
public void setMyArray(String[] myArray) {  
    this.myArray = myArray;  
}  
  
public void setMyMap(Map<String, String> myMap) {  
    this.myMap = myMap;  
}  
  
public void setMySet(Set<String> mySet) {  
    this.mySet = mySet;  
}  
  
public void setMyProperties(Properties myProperties) {  
    this.myProperties = myProperties;  
}
```

<!--set注入注入复杂数据类型-->

```
<property name="myArray">
  <array>
    <value>array1</value>
    <value>array2</value>
    <value>array3</value>
  </array>
</property>
```

```
<property name="myMap">
  <map>
    <entry key="key1" value="value1"/>
    <entry key="key2" value="value2"/>
  </map>
</property>
```

```
<property name="mySet">
  <set>
    <value>set1</value>
    <value>set2</value>
  </set>
</property>
```

```
<property name="myProperties">
  <props>
    <prop key="prop1">value1</prop>
    <prop key="prop2">value2</prop>
  </props>
</property>
```

在List结构的集合数据注入时，`array`、`list`、`set` 这三个标签通用，另外注值的 `value` 标签内部可以直接写值，也可以使用 `bean` 标签配置一个对象，或者用 `ref` 标签引用一个已经配合的bean的唯一标识。

在Map结构的集合数据注入时，`map` 标签使用 `entry` 子标签实现数据注入，`entry` 标签可以使用 `key` 和 `value` 属性指定存入map中的数据。使用 `value-ref` 属性指定已经配置好的bean的引用。同时 `entry` 标签中也可以使用 `ref` 标签，但是不能使用 `bean` 标签。而 `property` 标签中不能使用 `ref` 或者 `bean` 标签引用对象

1.3 xml与注解相结合模式

注意：

- 1) 实际企业开发中，纯xml模式使用已经很少了
- 2) 引入注解功能，不需要引入额外的jar
- 3) xml+注解结合模式，xml文件依然存在，所以，spring IOC容器的启动仍然从加载xml开始

4) 哪些bean的定义写在xml中, 哪些bean的定义使用注解

第三方jar中的bean定义在xml, 比如德鲁伊数据库连接池

自己开发的bean定义使用注解

- xml中标签与注解的对应 (IoC)

xml形式	对应的注解形式
标签	@Component("accountDao"), 注解加在类上 bean的id属性内容直接配置在注解后面如果不配置, 默认定义个这个bean的id为类的类名首字母小写; 另外, 针对分层代码开发提供了@Component的三种别名@Controller、@Service、@Repository分别用于控制层类、服务层类、dao层类的bean定义, 这四个注解的用法完全一样, 只是为了更清晰的区分而已
标签的scope属性	@Scope("prototype"), 默认单例, 注解加在类上
标签的init-method属性	@PostConstruct, 注解加在方法上, 该方法就是初始化后调用的方法
标签的destroy-method属性	@PreDestroy, 注解加在方法上, 该方法就是销毁前调用的方法

- DI 依赖注入的注解实现方式

@Autowired (推荐使用)

@Autowired为Spring提供的注解, 需要导入包
org.springframework.beans.factory.annotation.Autowired。

@Autowired采取的策略为按照类型注入。

```
public class TransferServiceImpl {  
    @Autowired  
    private AccountDao accountDao;  
}
```

如上代码所示, 这样装配回去spring容器中找到类型为AccountDao的类, 然后将其注入进来。这样会产生一个问题, 当一个类型有多个bean值的时候, 会造成无法选择具体注入哪一个的情况, 这个时候我们需要配合着@Qualifier使用。

@Qualifier告诉Spring具体去装配哪个对象。

```
public class TransferServiceImpl {
    @Autowired
    @Qualifier(name="jdbcAccountDaoImpl")
    private AccountDao accountDao;
}
```

这个时候我们就可以通过类型和名称定位到我们想注入的对象。

@Resource

@Resource 注解由 J2EE 提供，需要导入包 javax.annotation.Resource。

@Resource 默认按照 ByName 自动注入。

```
public class TransferService {
    @Resource
    private AccountDao accountDao;
    @Resource(name="studentDao")
    private StudentDao studentDao;
    @Resource(type="TeacherDao")
    private TeacherDao teacherDao;
    @Resource(name="manDao", type="ManDao")
    private ManDao manDao;
}
```

- 如果同时指定了 name 和 type，则从Spring上下文中找到唯一匹配的bean进行装配，找不到则抛出异常。
- 如果指定了 name，则从上下文中查找名称（id）匹配的bean进行装配，找不到则抛出异常。
- 如果指定了 type，则从上下文中找到类似匹配的唯一bean进行装配，找不到或是找到多个，都会抛出异常。
- 如果既没有指定name，又没有指定type，则自动按照byName方式进行装配；

注意：

@Resource 在 Jdk 11 中已经移除，如果要使用，需要单独引入jar包

```
<dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId>
    <version>1.3.2</version>
</dependency>
```

1.4 纯注解模式

改造xm+注解模式，将xml中遗留的内容全部以注解的形式迁移出去，最终删除xml，从Java配置类启动

对应注解

@Configuration 注解，表明当前类是一个配置类

@ComponentScan 注解，替代 context:component-scan

@PropertySource, 引入外部属性配置文件

@Import 引入其他配置类

@Value 对变量赋值, 可以直接赋值, 也可以使用 \${} 读取资源配置文件中的信息

@Bean 将方法返回对象加入 SpringIOC 容器

第2节 Spring IOC高级特性

2.1 lazy-Init 延迟加载

Bean的延迟加载 (延迟创建)

ApplicationContext 容器的默认行为是在启动服务器时将所有 singleton bean 提前进行实例化。提前实例化意味着作为初始化过程的一部分, ApplicationContext 实例会创建并配置所有的singleton bean。

比如:

```
<bean id="testBean" class="cn.lagou.LazyBean" />
```

该bean默认的设置为:

```
<bean id="testBean" class="cn.lagou.LazyBean" lazy-init="false" />
```

lazy-init="false", 立即加载, 表示在spring启动时, 立刻进行实例化。

如果不想让一个singleton bean 在 ApplicationContext实现初始化时被提前实例化, 那么可以将bean 设置为延迟实例化。

```
<bean id="testBean" class="cn.lagou.LazyBean" lazy-init="true" />
```

设置 lazy-init 为 true 的 bean 将不会在 ApplicationContext 启动时提前被实例化, 而是第一次向容器通过 getBean 索取 bean 时实例化的。

如果一个设置了立即加载的 bean1, 引用了一个延迟加载的 bean2, 那么 bean1 在容器启动时被实例化, 而 bean2 由于被 bean1 引用, 所以也被实例化, 这种情况也符合延时加载的 bean 在第一次调用时才被实例化的规则。

也可以在容器层次中通过在 元素上使用 "default-lazy-init" 属性来控制延时初始化。如下面配置:

```
<beans default-lazy-init="true">
  <!-- no beans will be eagerly pre-instantiated... -->
</beans>
```

如果一个 bean 的 scope 属性为 scope="prototype" 时, 即使设置了 lazy-init="false", 容器启动时也不会实例化bean, 而是调用 getBean 方法实例化的。

应用场景

- (1) 开启延迟加载一定程度提高容器启动和运转性能

(2) 对于不常使用的 Bean 设置延迟加载，这样偶尔使用的时候再加载，不必要从一开始该 Bean 就占用资源

(3) 父Bean应用泛型，当类实例化时通过反射来确定具体类型：需要设置父Bean为延迟加载

2.2 FactoryBean 和 BeanFactory

BeanFactory接口是容器的顶级接口，定义了容器的一些基础行为，负责生产和管理Bean的一个工厂，具体使用它下面的子接口类型，比如ApplicationContext；此处我们重点分析FactoryBean

Spring中Bean有两种，一种是普通Bean，一种是工厂Bean（FactoryBean），FactoryBean可以生成某一个类型的Bean实例（返回给我们），也就是说我们可以借助于它自定义Bean的创建过程。

Bean创建的三种方式中的静态方法和实例化方法和FactoryBean作用类似，FactoryBean使用较多，尤其在Spring框架一些组件中会使用，还有其他框架和Spring框架整合时使用

```
// 可以让我们自定义Bean的创建过程（完成复杂Bean的定义）
public interface FactoryBean<T> {

    @Nullable
    // 返回FactoryBean创建的Bean实例，如果isSingleton返回true，则该实例会放到Spring容器的单例对象缓存池中Map
    T getObject() throws Exception;

    @Nullable
    // 返回FactoryBean创建的Bean类型
    Class<?> getObjectType();

    // 返回作用域是否单例
    default boolean isSingleton() {
        return true;
    }
}
```

Company类

```
package com.lagou.edu.pojo;

/**
 * @author 应癡
 */
public class Company {

    private String name;
    private String address;
    private int scale;

    public String getName() {
```

```

        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public int getScale() {
        return scale;
    }

    public void setScale(int scale) {
        this.scale = scale;
    }

    @Override
    public String toString() {
        return "Company{" +
            "name='" + name + '\'' +
            ", address='" + address + '\'' +
            ", scale=" + scale +
            '}';
    }
}

```

CompanyFactoryBean类

```

package com.lagou.edu.factory;

import com.lagou.edu.pojo.Company;
import org.springframework.beans.factory.FactoryBean;

/**
 * @author 应癩
 */
public class CompanyFactoryBean implements FactoryBean<Company> {

    private String companyInfo; // 公司名称,地址,规模

    public void setCompanyInfo(String companyInfo) {

```

```

        this.companyInfo = companyInfo;
    }

    @Override
    public Company getObject() throws Exception {

        // 模拟创建复杂对象Company
        Company company = new Company();
        String[] strings = companyInfo.split(",");
        company.setName(strings[0]);
        company.setAddress(strings[1]);
        company.setScale(Integer.parseInt(strings[2]));

        return company;
    }

    @Override
    public Class<?> getObjectType() {
        return Company.class;
    }

    @Override
    public boolean isSingleton() {
        return true;
    }
}

```

xml配置

```

<bean id="companyBean" class="com.lagou.edu.factory.CompanyFactoryBean">
    <property name="companyInfo" value="拉勾,中关村,500"/>
</bean>

```

测试，获取FactoryBean产生的对象

```

Object companyBean = applicationContext.getBean("companyBean");
System.out.println("bean:" + companyBean);

// 结果如下
bean:Company{name='拉勾', address='中关村', scale=500}

```

测试，获取FactoryBean，需要在id之前添加"&"


```
Object companyBean = applicationContext.getBean("&companyBean");
System.out.println("bean:" + companyBean);
```

// 结果如下

```
bean:com.lagou.edu.factory.CompanyFactoryBean@53f6fd09
```

2.3 后置处理器

Spring提供了两种后处理bean的扩展接口，分别为 BeanPostProcessor 和 BeanFactoryPostProcessor，两者在使用上是有所区别的。

工厂初始化（BeanFactory）—> Bean对象

在BeanFactory初始化之后可以使用BeanFactoryPostProcessor进行后置处理做一些事情

在Bean对象实例化（并不是Bean的整个生命周期完成）之后可以使用BeanPostProcessor进行后置处理做一些事情

注意：对象不一定是springbean，而springbean一定是个对象

SpringBean的生命周期

2.3.1 BeanPostProcessor

BeanPostProcessor是针对Bean级别的处理，可以针对某个具体的Bean。

```
public interface BeanPostProcessor {
    @Nullable
    default Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
        return bean;
    }

    @Nullable
    default Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        return bean;
    }
}
```

该接口提供了两个方法，分别在Bean的初始化方法前和初始化方法后执行，具体这个初始化方法指的是什么方法，类似我们在定义bean时，定义了init-method所指定的方法

定义一个类实现了BeanPostProcessor，默认是会对整个Spring容器中所有的bean进行处理。如果要对具体的某个bean处理，可以通过方法参数判断，两个类型参数分别为Object和String，第一个参数是每个bean的实例，第二个参数是每个bean的name或者id属性的值。所以我们可以通过第二个参数，来判断我们将要处理的具体的bean。

注意：处理是发生在Spring容器的实例化和依赖注入之后。





















2.3.2 BeanFactoryPostProcessor

BeanFactory级别的处理，是针对整个Bean的工厂进行处理，典型应用:PropertyPlaceholderConfigurer

```
@FunctionalInterface
public interface BeanFactoryPostProcessor {
    void postProcessBeanFactory(ConfigurableListableBeanFactory var1) throws BeansException;
}
```

此接口只提供了一个方法，方法参数为ConfigurableListableBeanFactory，该参数类型定义了一些方法

ConfigurableListableBeanFactory

-   `clearMetadataCache(): void`
-   `freezeConfiguration(): void`
-   `getBeanDefinition(String): BeanDefinition`
-   `getBeanNamesIterator(): Iterator<String>`
-   `ignoreDependencyInterface(Class<?>): void`
-   `ignoreDependencyType(Class<?>): void`
-   `isAutowiredCandidate(String, DependencyDescriptor): boolean`
-   `isConfigurationFrozen(): boolean`
-   `preInstantiateSingletons(): void`
-   `registerResolvableDependency(Class<?>, Object): void`

其中有个方法名为`getBeanDefinition`的方法，我们可以根据此方法，找到我们定义bean的 `BeanDefinition`对象。然后我们可以对定义的属性进行修改，以下是`BeanDefinition`中的方法