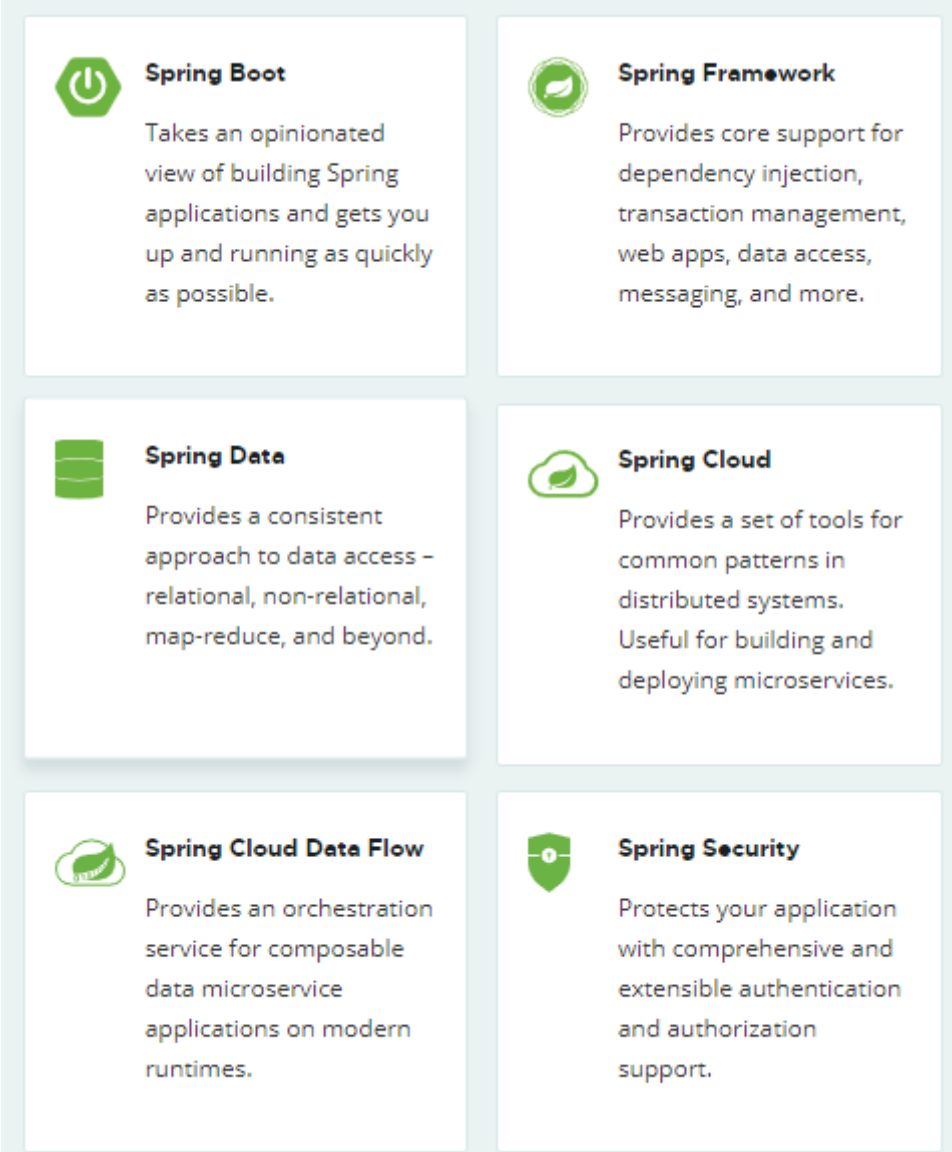
spring-day01

第一章 Spring的简介

第一节 Spring公司简介

该公司的创建者Rod Johnson被称之为Spring之父，他领导的Spring研发团队下有众多的优秀开发者，Spring公司旗下有非常多的优秀框架。例如:Spring FrameWork、Spring Boot、Spring Cloud、Spring Data、Spring Security等等，几乎涉及了Java开发的每一个领域。 官网地址：https://spring.io/



第二节 Spring Framework的介绍

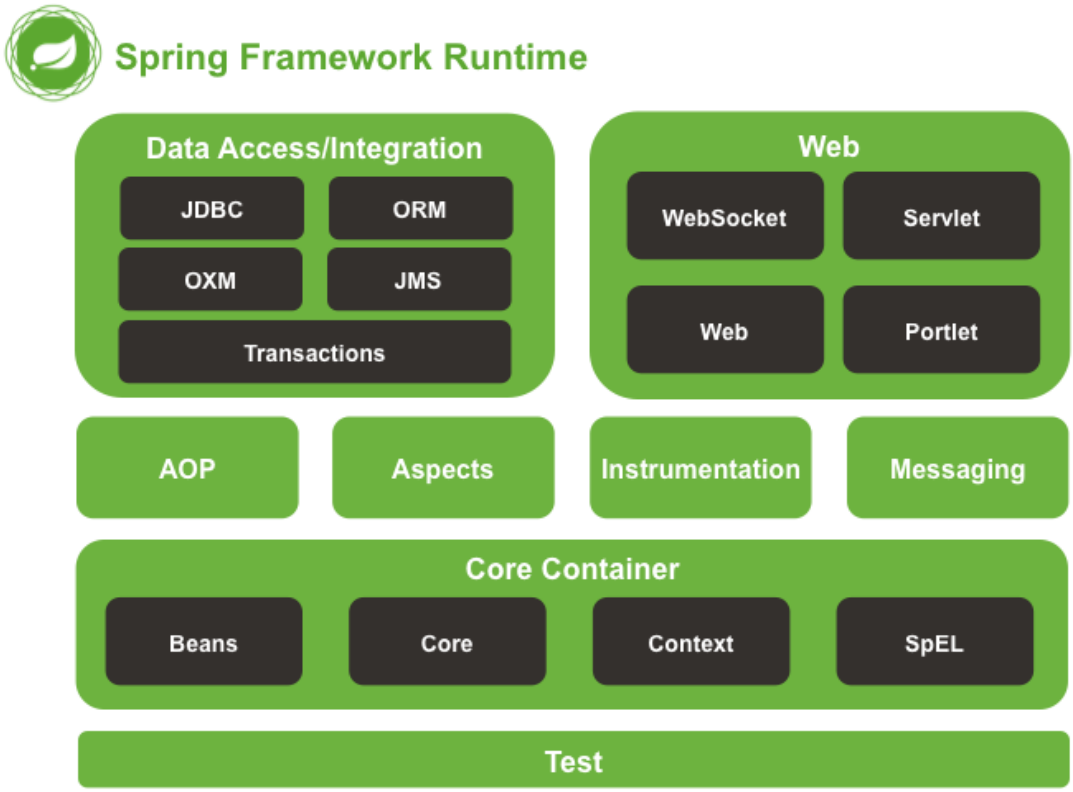
1. 概念

Spring Framework是Spring 基础框架，可以视为 Spring 基础设施，基本上任何其他 Spring 项目都是以 Spring Framework 为基础的。是一个分层的JavaSE/EE full-stack(一站式) 轻量级开源框架

2. 特征

- 非侵入式：使用 Spring Framework 开发应用程序时，Spring 对应用程序本身的结构影响非常小。对领域模型(domain)可以做到零污染；对功能性组件也只需要使用几个简单的注解进行标记，完全不会破坏原有结构，反而能将组件结构进一步简化。这就使得基于 Spring Framework 开发应用程序时结构清晰、简洁优雅。
- 控制反转：IOC——Inversion of Control，反转资源获取方向。把自己创建资源变成环境将资源准备好，我们享受资源注入。
- 面向切面编程：AOP——Aspect Oriented Programming，在不修改源代码的基础上增强代码功能。
- 容器：Spring IOC 是一个容器，因为它包含并且管理组件对象的生命周期。组件享受到了容器化的管理，替程序员屏蔽了组件创建过程中的大量细节，极大的降低了使用门槛，大幅度提高了开发效率。
- 组件化：Spring 实现了使用简单的组件配置组合成一个复杂的应用。在 Spring 中可以使用 XML 和 Java 注解组合这些对象。这使得我们可以基于一个个功能明确、边界清晰的组件有条不紊的搭建超大型复杂应用系统。
- 声明式：很多以前需要编写代码才能实现的功能，现在只需要声明需求即可由框架代为实现。
- 一站式：在 IOC 和 AOP 的基础上可以整合各种企业应用的开源框架和优秀的第三方类库。而且 Spring 旗下的项目已经覆盖了广泛领域，很多方面的功能性需求可以在 Spring Framework 的基础上全部使用 Spring 来实现。

2. Spring Framework五大功能模块



功能模块	功能介绍
Core Container	核心容器，在 Spring 环境下使用任何功能都必须基于 IOC 容器。
AOP&Aspects	面向切面编程
Test	提供了对 junit 或 TestNG 测试框架的整合。
Data Access/Integration	提供了对数据访问/集成的功能。
Spring MVC	提供了面向Web应用程序的集成功能。

第二章 IOC容器概念

第一节 容器的概念

1. 普通容器

普通容器只是负责存储数据(对象)，例如我们在JavaSE中学习的数组、List、Map等等，可以让我们使用它存储数据、获取数据，不具备其它复杂的功能

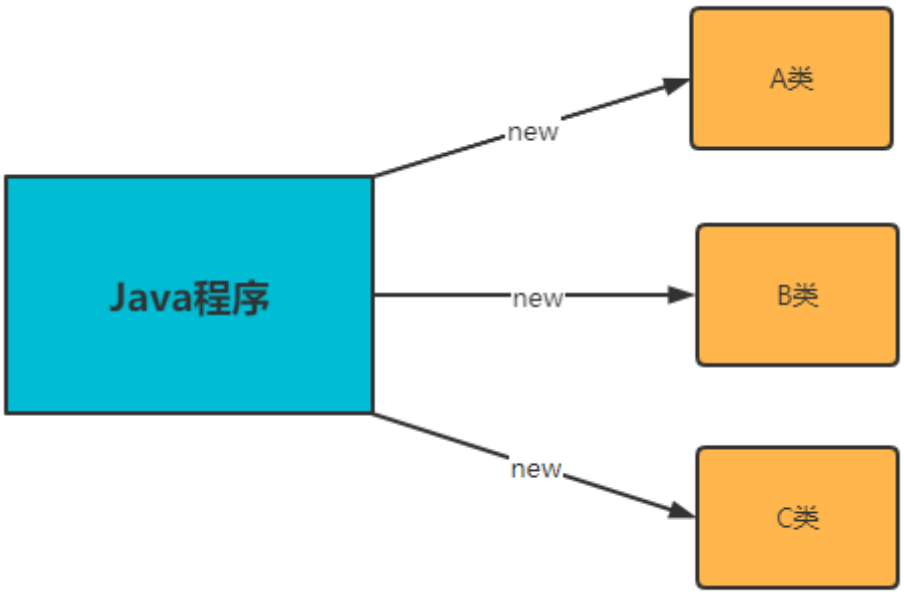
2. 复杂容器

复杂容器不仅要负责存储对象，还需要具备创建对象、调用对象方法、管理对象生命周期、并且在一定情况下负责销毁对象。例如我们之前学习的Tomcat就是一个复杂容器，它能够负责创建Servlet、Filter、Listener等等对象，并且管理他们的生命周期，在生命周期的不同阶段调用他们的不同方法。而我们后续要学习的IOC容器也是一个复杂容器

第二节 IOC的概念

1. 传统方式创建对象

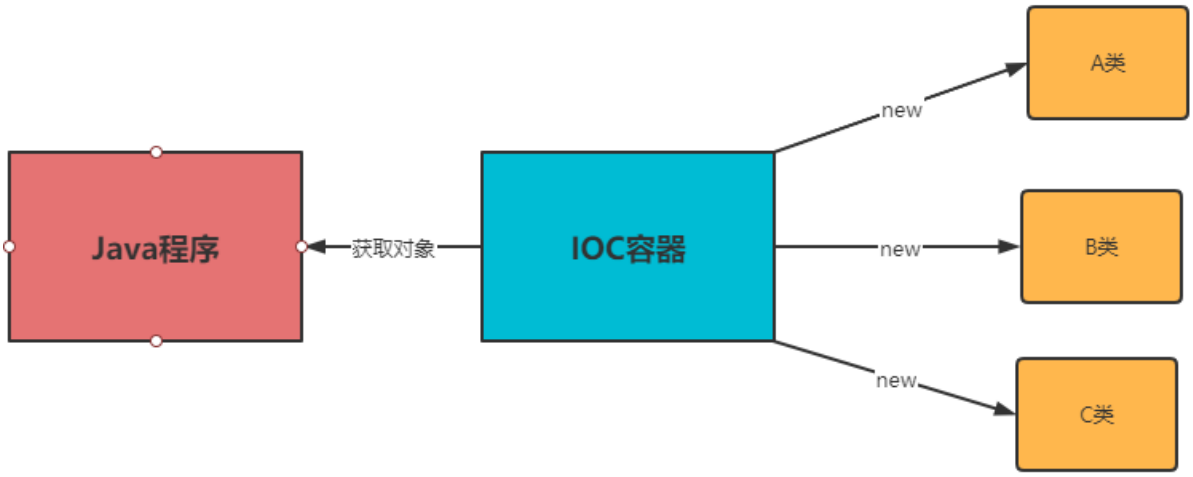
传统方式创建对象的方式是: 需要哪个类的对象，就直接在项目中new哪个类的对象，这样就会导致各个类之间的耦合度非常高



2. IOC方式创建对象

IOC(inversion of control)的中文解释是“控制反转”，对象的使用者不是创建者. 作用是将对象的创建反转给spring框架来创建和管理。控制反转怎么去理解呢。 其实它反转的是什么呢，是对象的创建工作。举个例子:平常我们在servlet或者service里面创建对象，都是使用new 的方式来直接创建对象，现在有了spring之后，我们就再也不new对象了，而是把对象创建的工作交给spring容器去维护。我们只需要告诉spring容器我们需要什么对象即可

IOC的作用：削减计算机程序的耦合(解除我们代码中的依赖关系)。



第三节 IOC容器在Spring中的实现

Spring 的 IOC 容器就是 IOC 思想的一个落地的产品实现。IOC 容器中管理的组件也叫做 bean。在创建 bean 之前，首先需要创建 IOC 容器。Spring 提供了 IOC 容器的两种实现方式:

1. BeanFactory

这是 IOC 容器的基本实现，是 Spring 内部使用的接口。面向 Spring 框架本身，供Spring框架内部功能使用，不建议开发人员使用。

2. ApplicationContext

BeanFactory 的子接口，提供了更多高级特性。面向 Spring 框架的使用者，几乎**所有**场合都使用 ApplicationContext 而不是底层的 BeanFactory。

> 以后在 Spring 环境下看到一个类或接口的名称中包含 ApplicationContext，那基本就可以断定，这个类或接口与 IOC 容器有关。

3. ApplicationContext的主要实现类

类型名	简介
ClassPathXmlApplicationContext	通过读取类路径下的 XML 格式的配置文件创建 IOC 容器对象
FileSystemXmlApplicationContext	通过文件系统路径读取 XML 格式的配置文件创建 IOC 容器对象
ConfigurableApplicationContext	ApplicationContext 的子接口，包含一些扩展方法 refresh() 和 close() ，让 ApplicationContext 具有启动、关闭和刷新上下文的能力。
AnnotationConfigApplicationContext	可以实现基于Java的配置类加载Spring的应用上下文，创建IOC容器对象
WebApplicationContext	专门为 Web 应用准备，基于 Web 环境创建 IOC 容器对象，并将对象引用存入 ServletContext 域中。

第三章 Spring IOC

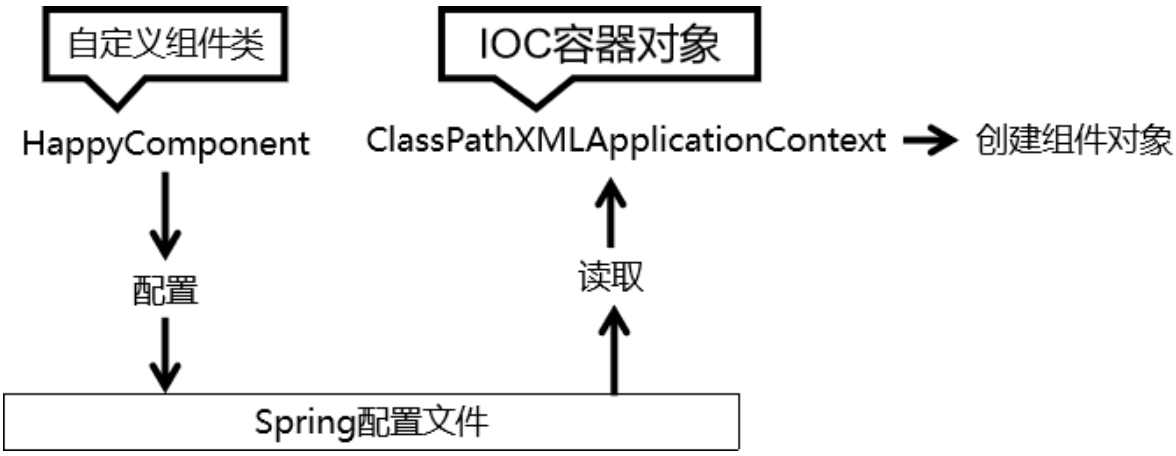
第一节 快速入门

1. 目标

1.1 让Spring IOC容器创建类的对象

1.2 从Spring IOC容器中获取对象

2. 思路



3. 具体实现

3.1 Maven依赖

```
<dependencies>
    <!-- 基于Maven依赖传递性，导入spring-context依赖即可导入当前所需所有jar包 -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.3.1</version>
    </dependency>
    <!-- junit测试 -->
    <dependency>
        <groupId>junit</groupId>
```

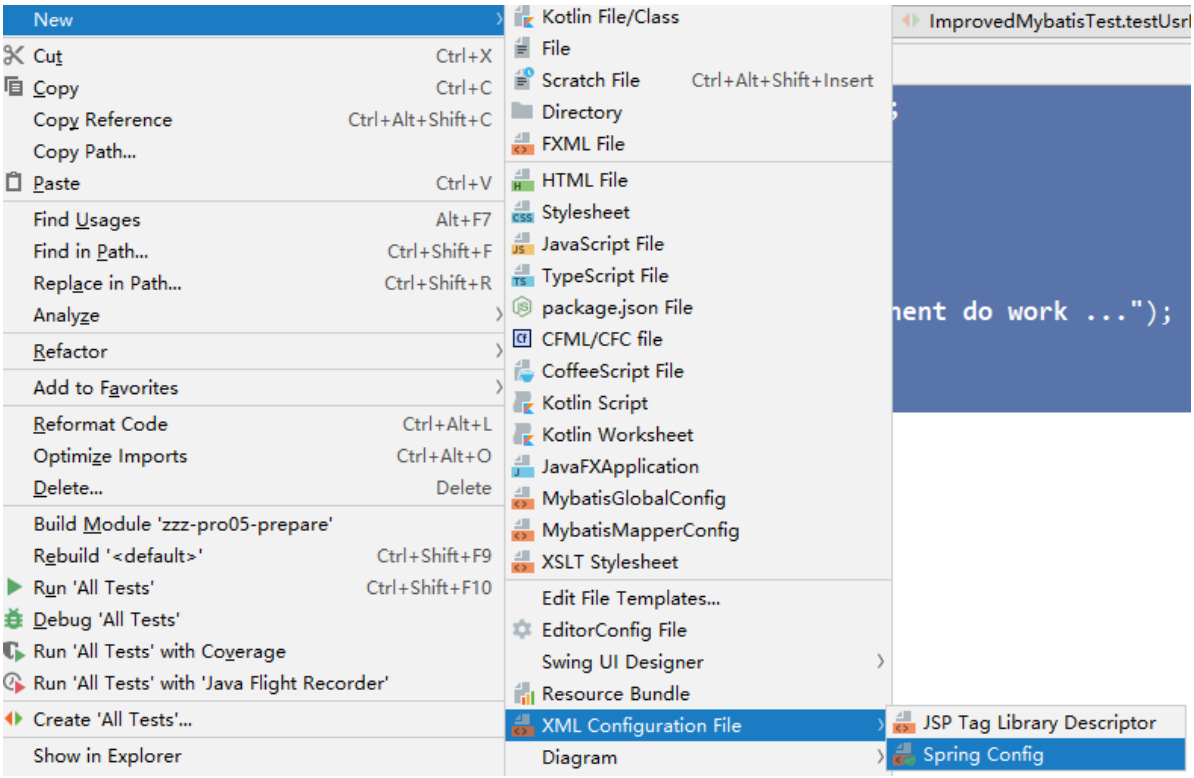
```
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

3.2 创建类

```
package com.atguigu.ioc.component;

public class HappyComponent {

    public void doWork() {
        System.out.println("component do work ...");
    }

}
```

3.3 创建Spring 配置文件并且配置组件

配置文件的存放路径建议放在resources根路径下，配置文件名字随意




```
<?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">
    <!--
        每一个实现类就对应一个bean标签
            id属性：对象的唯一标识，根据这个唯一标识，就可以从核心容器中获取对象
            class属性：对象所属的实现类的全限定名
    -->
    <bean id="happyComponent" class="com.atguigu.ioc.component.HappyComponent"/>
</beans>
```

- bean标签：通过配置bean标签告诉IOC容器需要创建对象的组件是什么
- id属性：bean的唯一标识
- class属性：组件类的全类名

3.4 从核心容器中获取对象

方式一：根据id获取

```
public class IOCTest {

    // 创建 IOC 容器对象，为便于其他实验方法使用声明为成员变量
    private ApplicationContext iocContainer = new
ClassPathXmlApplicationContext("applicationContext.xml");

    @Test
    public void testExperiment01() {

        // 从 IOC 容器对象中获取bean，也就是组件对象
        HappyComponent happyComponent = (HappyComponent)
iocContainer.getBean("happyComponent");
        happyComponent.doWork();
    }
}
```

方式二：根据类型获取

如果该类型在核心容器中只有一个对象：

```
@Test
public void testExperiment02() {

    HappyComponent component = iocContainer.getBean(HappyComponent.class);

    component.doWork();

}
```

如果该类型在核心容器中有多个对象：那么根据类型获取时会抛出异常，具体异常信息如下

> org.springframework.beans.factory.**NoUniqueBeanDefinitionException**: No qualifying bean of type 'com.atguigu.ioc.component.HappyComponent' available: expected single matching bean but found 2: happyComponent,happyComponent2

思考

如果组件类实现了接口，根据接口类型可以获取 bean对象 吗?

> 可以，前提是bean对象唯一

如果一个接口有多个实现类，这些实现类都配置了 bean，根据接口类型可以获取 bean 吗?

> 不行，因为bean对象不唯一

结论

根据类型来获取bean时，在满足bean唯一性的前提下，其实只是看：『对象 **instanceof** 指定的类型』的返回结果，只要返回的是true就可以认定为和类型匹配，能够获取到。

第二节 依赖注入

依赖注入全称是 dependency Injection 翻译过来是依赖注入.其实就是如果spring核心容器管理的某一个类中存在属性，需要spring核心容器在创建该类实例的时候，顺便给这个对象里面的属性进行赋值。

1. setter方法注入

如果某个Bean对象的属性有对应的setter方法，那我们可以在配置文件中使用setter方法对属性进行依赖注入

1.1 注入简单类型数据

1.1.1 给组件类添加一个简单类型属性

```
package com.atguigu.component;

/**
 * 包名:com.atguigu.component
 *
 * @author Leevi
 * 日期2021-08-29  10:10
 * 给一个对象的成员变量赋值的方式:
 * 1. 调用set方法
 * 2. 通过构造器
 * 3. 通过暴力反射
 */
public class HappyComponent {
    private String username;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public void doWork() {
        System.out.println("component do work ...");
    }
}
```

1.1.2 在配置时给属性指定值

通过property标签配置的属性值会通过setXxx()方法注入，大家可以通过debug方式验证一下

```
<!--
        依赖注入:给核心容器中的Bean对象的成员变量赋值
        setter方法进行依赖注入:
            在要进行依赖注入的bean标签中添加<property>子标签,该子标签的name属性就是要赋值的成员变量名
```

```
            前提是这个属性一定要有set方法,name属性的值应该是"setXXX"后面的"XXX"首字母改小
写
            1. 注入简单类型数据: 那么我们使用property标签的value属性给简单类型的成员变量赋
值


            2. 注入Bean类型数据: 那么我们使用property标签的ref属性给Bean类型的成员变量赋
值,
                                ref属性的值就是要赋值的Bean类型的对象在核心容器中的id
    -->
<bean id="happyComponent" class="com.atguigu.component.HappyComponent">
    <property name="username" value="奥巴马"></property>
</bean>
```

1.1.3 测试代码

```
@Test
public void testGetHappyComponent(){
    //2. 使用核心容器对象获取HappyComponent对象
    //根据id获取:获取ioc容器中id为"happyComponent"的对象
    HappyComponent happyComponent1 = (HappyComponent)
act.getBean("happyComponent");

    //3. 使用HappyComponent对象获取username属性
    System.out.println(happyComponent1.getUsername());
}
```

1.2 注入Bean类型数据

1.2.1 声明新的组件类UserServlet

```
package com.atguigu.servlet;

import com.atguigu.service.UserService;

/**
 * 包名:com.atguigu.servlet
 *
 * @author Leevi
 * 日期2021-08-29  10:28
 * 1. IOC : 由核心容器创建Bean对象
 * 2. DI(依赖注入) : 给核心容器中的Bean对象的成员变量赋值
 */
public class UserServlet {
    private UserService userService;

    public void setUserService(UserService userService) {
        this.userService = userService;
    }

    public void sayHello(){
        userService.sayHello();
    }
}
```

1.2.2 声明新的组件接口UserService和实现类UserServiceImpl

UserService接口

```
package com.atguigu.service;

/**
 * 包名:com.atguigu.service
 *
 * @author Leevi
 * 日期2021-08-29  10:27
 */
public interface UserService {
    void sayHello();
}
```

UserServiceImpl实现类

```
package com.atguigu.service.impl;

import com.atguigu.component.HappyComponent;
import com.atguigu.service.UserService;

/**
 * 包名:com.atguigu.service
 *
 * @author Leevi
 * 日期2021-08-29  10:33
 *
 */
public class UserServiceImpl implements UserService {
    private HappyComponent happyComponent;

    public void setHappyComponent(HappyComponent happyComponent) {
        this.happyComponent = happyComponent;
    }

    @Override
    public void sayHello() {
        System.out.println("hello,"+happyComponent.getUsername());
    }
}
```

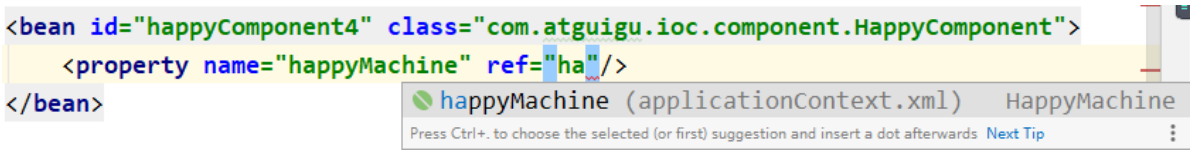
1.2.3 在UserService对象中注入HappyComponent对象

```
<!--
        使用依赖注入给HappyComponent属性赋值
    -->
<bean id="userService" class="com.atguigu.service.impl.UserServiceImpl">
    <property name="happyComponent" ref="happyComponent"></property>
</bean>
```

1.2.4 在UserServlet对象中注入UserService对象

```
<!--
        你想让ioc容器创建什么对象，就将那个类配置到bean标签中

        使用依赖注入给UserService属性赋值
    -->
<bean id="userServlet" class="com.atguigu.servlet.UserServlet">
    <property name="userService" ref="userService"></property>
</bean>
```

这个操作在 IDEA 中有提示：



1.2.5 测试

```
@Test
public void testSayHello(){
    //通过ioc容器获取UserServlet的对象
    UserServlet userServlet = (UserServlet) act.getBean("userServlet");
    //调用UserServlet的sayHello()方法
    userServlet.sayHello();
}
```

1.2.6 易错点

> 如果错把ref属性写成了value属性，会抛出异常： Caused by: java.lang.IllegalStateException: Cannot convert value of type 'java.lang.String' to required type 'com.atguigu.ioc.component.HappyMachine' for property 'happyMachine': no matching editors or conversion strategy found 意思是不能把String类型转换成我们要的HappyMachine类型 说明我们使用value属性时，Spring只把这个属性看做一个普通的字符串，不会认为这是一个bean的id，更不会根据它去找到bean来赋值

1.3 注入内部Bean类型数据(了解)

1.3.1 重新配置原组件

在bean里面配置的bean就是内部bean，内部bean只能在当前bean内部使用，在其他地方不能使用。

```
<bean id="userService" class="com.atguigu.service.impl.UserServiceImpl">
    <property name="happyComponent">
        <!--这个bean标签创建的HappyComponent对象只能用于给UserServiceImpl的
happyComponent属性赋值,别的地方不能使用-->
        <bean class="com.atguigu.component.HappyComponent">
            <property name="username" value="aobama"></property>
        </bean>
    </property>
</bean>
```

1.3.2 测试

```
@Test
public void testExperiment04() {
    //通过核心容器获取UserService的对象:我为什么要使用接口类型接收实现类的对象,为了解耦
    UserService userService = (UserService) act.getBean("userService");
    userService.sayHello();
}
```

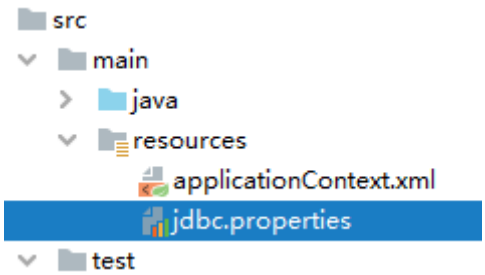
1.4 引入外部属性文件用于给Bean注入属性

1.4.1 添加Maven依赖

这个依赖只是为了使用Druid连接池，而不是引入外部属性文件所必须的

```
<!-- MySQL驱动 -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.3</version>
</dependency>
<!-- 数据源 -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.0.31</version>
</dependency>
```

1.4.2 创建外部属性文件




```
jdbc.user=root
jdbc.password=123456
jdbc.url=jdbc:mysql://localhost:3306/mybatis-example
jdbc.driver=com.mysql.jdbc.Driver
```

1.4.3 在spring的配置文件中引入jdbc.properties文件

```
<!-- 引入外部属性文件 -->
<context:property-placeholder location="classpath:jdbc.properties"/>
```

1.4.4 在spring的配置文件中使用引入的jdbc.properties文件中的数据

```
<!--[重要]给bean的属性赋值：引入外部属性文件 -->
<bean id="druidDataSource" class="com.alibaba.druid.pool.DruidDataSource">
    <property name="url" value="${jdbc.url}"/>
    <property name="driverClassName" value="${jdbc.driver}"/>
    <property name="username" value="${jdbc.user}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>
```

1.4.5 测试

```
@Test
public void testExperiment06() throws SQLException {
    DataSource dataSource = iocContainer.getBean(DataSource.class);
    Connection connection = dataSource.getConnection();
    System.out.println("connection = " + connection);
}
```

1.4.6 结论

标签的value属性: 注入简单类型数据

标签的ref属性:用于引入IOC容器中的Bean对象的id，注入Bean对象类型的数据

1.5 注入集合类型属性(了解)

1.5.1 给组件类添加集合类型属性

```
package com.atguigu.component;

import java.util.List;

/**
 * 包名:com.atguigu.component
 *
 * @author Leevi
 * 日期2021-08-29  10:10
 * 给一个对象的成员变量赋值的方式:
 * 1. 调用set方法
 * 2. 通过构造器
 * 3. 通过暴力反射
 */
public class HappyComponent {
    private String username;
    private List<String> memberList;

    public List<String> getMemberList() {
        return memberList;
    }

    public void setMemberList(List<String> memberList) {
        this.memberList = memberList;
    }

    public String getUsername() {
        return username;
    }
```

```
    public void setUsername(String username) {
        this.username = username;
    }

    public void doWork() {
        System.out.println("component do work ...");
    }
}
```

1.5.2 配置

```
<bean id="happyComponent" class="com.atguigu.component.HappyComponent">
    <property name="username" value="奥巴马"></property>
    <!--注入集合类型的数据-->
    <property name="memberList">
        <!--<list>
                <value>张三</value>
                <value>李四</value>
                <value>王五</value>
                <value>赵六</value>
            </list>-->

        <!--
                使用set标签注入集合可以去重
            -->
        <!--<set>
                <value>张三</value>
                <value>李四</value>
                <value>王五</value>
                <value>赵六</value>
                <value>王五</value>
            </set>-->

        <array>
            <value>张三</value>
            <value>李四</value>
            <value>王五</value>
            <value>赵六</value>
            <value>王五</value>
        </array>
    </property>
</bean>
```

1.6 注入Map类型属性(了解)

1.6.1 给组件类添加Map类型属性

```
package com.atguigu.component;

import java.util.List;
import java.util.Map;

/**
 * 包名:com.atguigu.component
 *
 * @author Leevi
 * 日期2021-08-29  10:10
```

```
 * 给一个对象的成员变量赋值的方式:
 * 1. 调用set方法
 * 2. 通过构造器
 * 3. 通过暴力反射
 */
public class HappyComponent {
    private String username;
    private List<String> memberList;
    private Map<String,String> managerMap;

    public Map<String, String> getManagerMap() {
        return managerMap;
    }

    public void setManagerMap(Map<String, String> managerMap) {
        this.managerMap = managerMap;
    }

    public List<String> getMemberList() {
        return memberList;
    }

    public void setMemberList(List<String> memberList) {
        this.memberList = memberList;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public void doWork() {
        System.out.println("component do work ...");
    }
}
```

1.6.2 配置

```
<bean id="happyComponent" class="com.atguigu.component.HappyComponent">
    <!--注入Map类型的数据-->
    <property name="managerMap">
        <!--<map>
                <entry key="k1" value="v1" ></entry>
                <entry key="k2" value="v2"></entry>
                <entry key="k3" value="v3"></entry>
                <entry key="k4" value="v4"></entry>
            </map>-->

        <props>
            <prop key="k1">v1</prop>
            <prop key="k2">v2</prop>
            <prop key="k3">v3</prop>
            <prop key="k4">v4</prop>
        </props>
```

```
    </property>
</bean>
```

1.7 注入Bean的集合类型(了解)

1.7.1 给组件添加Bean的集合类型属性

```
package com.atguigu.component;

import java.util.List;
import java.util.Map;

/**
 * 包名:com.atguigu.component
 *
 * @author Leevi
 * 日期2021-08-29  10:10
 * 给一个对象的成员变量赋值的方式:
 * 1. 调用set方法
 * 2. 通过构造器
 * 3. 通过暴力反射
 */
public class HappyComponent {
    private String username;
    private List<String> memberList;
    private Map<String,String> managerMap;
    private List<User> userList;

    public List<User> getUserList() {
        return userList;
    }

    public void setUserList(List<User> userList) {
        this.userList = userList;
    }

    public Map<String, String> getManagerMap() {
        return managerMap;
    }

    public void setManagerMap(Map<String, String> managerMap) {
        this.managerMap = managerMap;
    }

    public List<String> getMemberList() {
        return memberList;
    }

    public void setMemberList(List<String> memberList) {
        this.memberList = memberList;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
```

```
        this.username = username;
    }

    public void doWork() {
        System.out.println("component do work ...");
    }
}
```

User类

```
package com.atguigu.component;

/**
 * 包名:com.atguigu.component
 *
 * @author Leevi
 * 日期2021-08-29  14:57
 */
public class User {
    private String name;
    private String address;

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

    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;
    }
}
```

1.7.2 配置

```
<bean id="happyComponent" class="com.atguigu.component.HappyComponent">
    <property name="username" value="奥巴马"></property>
    <!--注入集合类型的数据-->
    <property name="memberList">
        <!--<list>
                <value>张三</value>
                <value>李四</value>
```

```
                <value>王五</value>
                <value>赵六</value>
            </list>-->

        <!--
                使用set标签注入集合可以去重
            -->
        <!--<set>
                <value>张三</value>
                <value>李四</value>
                <value>王五</value>
                <value>赵六</value>
                <value>王五</value>
            </set>-->

        <array>
            <value>张三</value>
            <value>李四</value>
            <value>王五</value>
            <value>赵六</value>
            <value>王五</value>
        </array>
    </property>

    <!--注入Map类型的数据-->
    <property name="managerMap">
        <!--<map>
                <entry key="k1" value="v1" ></entry>
                <entry key="k2" value="v2"></entry>
                <entry key="k3" value="v3"></entry>
                <entry key="k4" value="v4"></entry>
            </map>-->

        <props>
            <prop key="k1">v1</prop>
            <prop key="k2">v2</prop>
            <prop key="k3">v3</prop>
            <prop key="k4">v4</prop>
        </props>
    </property>

    <!--注入Bean的集合类型-->
    <property name="userList">
        <list>
            <bean class="com.atguigu.component.User">
                <property name="name" value="张三"></property>
                <property name="address" value="深圳"></property>
            </bean>
            <bean class="com.atguigu.component.User">
                <property name="name" value="李四"></property>
                <property name="address" value="广州"></property>
            </bean>
            <bean class="com.atguigu.component.User">
                <property name="name" value="王五"></property>
                <property name="address" value="北京"></property>
            </bean>
        </list>
    </property>
```

```
</bean>
```

2. 构造器注入(了解)

在前面我们通过 <bean> 标签配置Bean对象，其实是执行Bean类的**无参构造**函数创建的对象，当Bean类包含有参构造函数的时候，我们在配置文件中可以通过有参构造函数进行配置注入

2.1 声明组件类

```
package com.atguigu.ioc.component;

public class HappyTeam {

    private String teamName;
    private Integer memberCount;
    private Double memberSalary;

    public String getTeamName() {
        return teamName;
    }

    public void setTeamName(String teamName) {
        this.teamName = teamName;
    }

    public Integer getMemberCount() {
        return memberCount;
    }

    public void setMemberCount(Integer memberCount) {
        this.memberCount = memberCount;
    }

    public Double getMemberSalary() {
        return memberSalary;
    }

    public void setMemberSalary(Double memberSalary) {
        this.memberSalary = memberSalary;
    }

    @Override
    public String toString() {
        return "HappyTeam{" +
                "teamName='" + teamName + '\'' +
                ", memberCount=" + memberCount +
                ", memberSalary=" + memberSalary +
                '}';
    }

    public HappyTeam(String teamName, Integer memberCount, Double memberSalary)
{
        this.teamName = teamName;
        this.memberCount = memberCount;
        this.memberSalary = memberSalary;
    }
```

```
    public HappyTeam() {
    }
}
```

2.2 配置构造器注入

```
<!-- 给bean的属性赋值：构造器注入 -->
<bean id="happyTeam" class="com.atguigu.ioc.component.HappyTeam">
    <constructor-arg value="happyCorps"/>
    <constructor-arg value="10"/>
    <constructor-arg value="1000.55"/>
</bean>
```

2.3 测试

```
@Test
public void testExperiment08() {

    HappyTeam happyTeam = iocContainer.getBean(HappyTeam.class);

    System.out.println("happyTeam = " + happyTeam);

}
```

2.4 补充

constructor-arg标签还有两个属性可以进一步描述构造器参数：

- index属性：指定参数所在位置的索引（从0开始）
- name属性：指定参数名

3. 特殊值处理(了解)

3.1 声明一个类用于测试

```
package com.atguigu.ioc.component;

public class PropValue {
    private String commonValue;
    private String expression;

    public String getCommonValue() {
        return commonValue;
    }

    public void setCommonValue(String commonValue) {
        this.commonValue = commonValue;
    }

    public String getExpression() {
        return expression;
    }

    public void setExpression(String expression) {
        this.expression = expression;
    }
```

```
    @Override
    public String toString() {
        return "PropValue{" +
                "commonValue='" + commonValue + '\'' +
                ", expression='" + expression + '\'' +
                '}';
    }
    public PropValue(String commonValue, String expression) {
        this.commonValue = commonValue;
        this.expression = expression;
    }
    public PropValue() {
    }
}
```

3.2 null值

```
<property name="commonValue">
    <!-- null标签：将一个属性值明确设置为null -->
    <null/>
</property>
```

3.3 当value值中有特殊字符时

3.3.1 使用XML实体字符(转义符)解决

```
<bean id="propValue" class="com.atguigu.ioc.component.PropValue">
    <!-- 小于号在XML文档中用来定义标签的开始，不能随便使用 -->
    <!-- 解决方案一：使用XML实体来代替 -->
    <property name="expression" value="a &lt; b"/>
</bean>
```

3.3.2 使用CDATA解决

```
<bean id="propValue" class="com.atguigu.ioc.component.PropValue">
    <property name="expression">
        <!-- 解决方案二：使用CDATA节 -->
        <!-- CDATA中的C代表Character，是文本、字符的含义，CDATA就表示纯文本数据 -->
        <!-- XML解析器看到CDATA节就知道这里是纯文本，就不会当作XML标签或属性来解析 -->
        <!-- 所以CDATA节中写什么符号都随意 -->
        <value><![CDATA[a < b]]></value>
    </property>
</bean>
```

4. p命名空间方式注入(了解)

4.1 引入p命名空间的约束

使用 p 名称空间需要导入相关的 XML 约束，在 IDEA 的协助下导入即可：

```
<?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"
xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
https://www.springframework.org/schema/context/spring-context.xsd">
```

4.2 使用p命名空间注入

```
<!--注入简单类型数据-->
<bean id="happyComponent" class="com.atguigu.component.HappyComponent"
p:username="奥拉夫">
</bean>

<!--注入Bean类型数据-->
<bean id="userServlet" class="com.atguigu.servlet.UserServlet" p:userService-
ref="userService">
</bean>

<bean id="userService" class="com.atguigu.service.impl.UserServiceImpl"
p:happyComponent-ref="happyComponent">
</bean>
```

4.3 测试

```
@Test
public void testSayHello(){
    //通过ioc容器获取UserServlet的对象
    UserServlet userServlet = (UserServlet) act.getBean("userServlet");
    //调用UserServlet的sayHello()方法
    userServlet.sayHello();
}
```

5. 自动装配(理解)

所谓自动装配就是一个组件需要其他组件时，由 IOC 容器负责找到那个需要的组件，并装配进去。

5.1 配置

```
<bean id="happyComponent2" class="com.atguigu.component.HappyComponent"
p:username="奥拉夫">
</bean>
<bean id="happyComponent" class="com.atguigu.component.HappyComponent"
p:username="奥巴马">
</bean>
<!--
        你想让ioc容器创建什么对象，就将那个类配置到bean标签中

        使用依赖注入给UserService属性赋值

        自动装配:autowire属性表示自动装配，就是不需要你去管依赖注入，IOC容器会自动进行依赖注
入。它的取值有如下两个
```

```
            1. byName:根据要注入的属性名和Bean对象的id的对应关系去注入
            2. byType:表示核心容器会自动在自身容器中查找一个该类型的对象，给成员变量赋值



    -->
<bean id="userServlet" class="com.atguigu.servlet.UserServlet"
autowire="byType">
</bean>

<bean id="userService" class="com.atguigu.service.impl.UserServiceImpl"
autowire="byName">
</bean>
```

5.2 测试

```
@Test
public void testSayHello(){
    //通过ioc容器获取UserServlet的对象
    UserServlet userServlet = (UserServlet) act.getBean("userServlet");
    //调用UserServlet的sayHello()方法
    userServlet.sayHello();
}
```

第三节 Bean的作用域和生命周期

1. Bean的作用域

1.1 概念

在Spring中可以通过配置bean标签的scope属性来指定bean的作用域范围，各取值含义参加下表：

取值	含义	创建对象的时机
singleton	在IOC容器中，这个bean的对象始终为单实例	IOC容器初始化时
prototype	这个bean在IOC容器中有多个实例	获取bean时

如果是在WebApplicationContext环境下还会有另外两个作用域（但几乎不用）：

取值	含义
request	在一个请求范围内有效
session	在一个会话范围内有效

1.2 配置

```
<!-- scope属性：取值singleton（默认值），bean在IOC容器中只有一个实例，IOC容器初始化时创建对象 -->
<!-- scope属性：取值prototype，bean在IOC容器中可以有多个实例，getBean()时创建对象 -->
<bean id="happyComment" scope="prototype"
class="com.atguigu.component.HappyComment">
</bean>
```

1.3 测试

```
@Test
public void testGetBean() {
    HappyComment happyComment01 = (HappyComment) act.getBean("happyComment");
    HappyComment happyComment02 = (HappyComment) act.getBean("happyComment");

    System.out.println(happyComment01 == happyComment02);
}
```

2. Bean的生命周期(了解)

2.1 bean的生命周期清单

- bean对象创建（调用无参构造器/有参构造器）
- 给bean对象设置属性（依赖注入）
- bean对象初始化之前操作（由bean的后置处理器前置方法负责）
- bean对象初始化（需在配置bean时指定初始化方法）
- bean对象初始化之后操作（由bean的后置处理器后置方法负责）
- bean对象就绪可以使用
- bean对象销毁（需在配置bean时指定销毁方法）
- IOC容器关闭

2.2 指定bean的初始化方法和销毁方法

2.2.1 创建两个方法作为初始化和销毁方法

用com.atguigu.component.HappyComponent类测试，在类中加俩方法：

```
package com.atguigu.component;

/**
 * 包名:PACKAGE_NAME
 *
 * @author Leevi
 * 日期2021-08-29  16:02
 * 目标:让HappyComponent对象创建的时候，就执行initLifeCircle()方法，在HappyComponent对象销毁之前就执行destroyLifeCircle()
 */
public class HappyComponent {
    public void initLifeCircle(){
        System.out.println("HappyComponent对象创建了，我可以做一些初始化操作...");
    }

    public void destroyLifeCircle(){
        System.out.println("HappyComponent对象销毁了，我可以做一些数据备份工作...");
    }
    public void sayHello(){
        System.out.println("hello world");
    }
}
```

2.2.2 配置bean时指定初始化和销毁方法

```
<!--
        bean标签的scope属性表示这个Bean对象的范围:
            1. singleton(默认取值): 单例
            2. prototype: 多例
        bean标签的init-method属性是用于配置这个Bean对象的初始化方法,
        bean标签的destroy-method属性是用于配置这个Bean对象的销毁方法
    -->
<bean id="happyComponent" class="com.atguigu.component.HappyComponent"
      scope="prototype"
      init-method="initLifeCircle"
      destroy-method="destroyLifeCircle"></bean>
```

2.3 bean的后置处理器

2.3.1 创建后置处理器类

```
package com.atguigu.ioc.process;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;

// 声明一个自定义的bean后置处理器
// 注意：bean后置处理器不是单独针对某一个bean生效，而是针对IOC容器中所有bean都会执行
public class MyHappyBeanProcessor implements BeanPostProcessor {

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName)
throws BeansException {

        System.out.println("☆☆☆" + beanName + " = " + bean);

        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName)
throws BeansException {

        System.out.println("★★★" + beanName + " = " + bean);

        return bean;
    }
}
```

2.3.2 把bean的后置处理器放入IOC容器

```
<!-- bean的后置处理器要放入IOC容器才能生效 -->
<bean id="myHappyBeanProcessor"
class="com.atguigu.ioc.process.MyHappyBeanProcessor"/>
```

2.3.3 执行效果示例

> HappyComponent创建对象
>
> HappyComponent要设置属性了
>
> ☆☆☆happyComponent = com.atguigu.ioc.component.HappyComponent@ca263c2
>
> HappyComponent初始化
>
> ★★★happyComponent = com.atguigu.ioc.component.HappyComponent@ca263c2
> HappyComponent销毁

第四节 FactoryBean机制(了解)

1. 简介

FactoryBean是Spring提供的一种整合第三方框架的常用机制。和普通的bean不同，配置一个FactoryBean类型的bean，在获取bean的时候得到的并不是class属性中配置的这个类的对象，而是getObject()方法的返回值。通过这种机制，Spring可以帮我们把复杂组件创建的详细过程和繁琐细节都屏蔽起来，只把最简洁的使用界面展示给我们。

将来我们整合Mybatis时，Spring就是通过FactoryBean机制来帮我们创建SqlSessionFactory对象的。

源码:

```
/*
 * Copyright 2002-2020 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package org.springframework.beans.factory;
import org.springframework.lang.Nullable;

/**
 * Interface to be implemented by objects used within a {@link BeanFactory}
which
 * are themselves factories for individual objects. If a bean implements this
 * interface, it is used as a factory for an object to expose, not directly as a
 * bean instance that will be exposed itself.
 *
 * <p><b>NB: A bean that implements this interface cannot be used as a normal
bean.</b>
 * A FactoryBean is defined in a bean style, but the object exposed for bean
 * references ({@link #getObject()}) is always the object that it creates.
 *
 * <p>FactoryBeans can support singletons and prototypes, and can either create
 * objects lazily on demand or eagerly on startup. The {@link SmartFactoryBean}
 * interface allows for exposing more fine-grained behavioral metadata.
```

```
 *
 * <p>This interface is heavily used within the framework itself, for example
for
 * the AOP {@link org.springframework.aop.framework.ProxyFactoryBean} or the
 * {@link org.springframework.jndi.JndiObjectFactoryBean}. It can be used for
 * custom components as well; however, this is only common for infrastructure
code.
 *
 * <p><b>{@code FactoryBean} is a programmatic contract. Implementations are not
 * supposed to rely on annotation-driven injection or other reflective
facilities.</b>
 * {@link #getObjectType()} {@link #getObject()} invocations may arrive early in
the
 * bootstrap process, even ahead of any post-processor setup. If you need access
to
 * other beans, implement {@link BeanFactoryAware} and obtain them
programmatically.
 *
 * <p><b>The container is only responsible for managing the lifecycle of the
FactoryBean
 * instance, not the lifecycle of the objects created by the FactoryBean.</b>
Therefore,
 * a destroy method on an exposed bean object (such as {@link
java.io.Closeable#close()}
 * will <i>not</i> be called automatically. Instead, a FactoryBean should
implement
 * {@link DisposableBean} and delegate any such close call to the underlying
object.
 *
 * <p>Finally, FactoryBean objects participate in the containing BeanFactory's
 * synchronization of bean creation. There is usually no need for internal
 * synchronization other than for purposes of lazy initialization within the
 * FactoryBean itself (or the like).
 *
 * @author Rod Johnson
 * @author Juergen Hoeller
 * @since 08.03.2003
 * @param <T> the bean type
 * @see org.springframework.beans.factory.BeanFactory
 * @see org.springframework.aop.framework.ProxyFactoryBean
 * @see org.springframework.jndi.JndiObjectFactoryBean
 */
public interface FactoryBean<T> {

    /**
     * The name of an attribute that can be
     * {@link org.springframework.core.AttributeAccessor#setAttribute set} on a
     * {@link org.springframework.beans.factory.config.BeanDefinition} so that
     * factory beans can signal their object type when it can't be deduced from
     * the factory bean class.
     * @since 5.2
     */
    String OBJECT_TYPE_ATTRIBUTE = "factoryBeanObjectType";


    /**
     * Return an instance (possibly shared or independent) of the object
     * managed by this factory.
```

```
     * <p>As with a {@link BeanFactory}, this allows support for both the
     * Singleton and Prototype design pattern.
     * <p>If this FactoryBean is not fully initialized yet at the time of
     * the call (for example because it is involved in a circular reference),
     * throw a corresponding {@link FactoryBeanNotInitializedException}.
     * <p>As of Spring 2.0, FactoryBeans are allowed to return {@code null}
     * objects. The factory will consider this as normal value to be used; it
     * will not throw a FactoryBeanNotInitializedException in this case anymore.
     * FactoryBean implementations are encouraged to throw
     * FactoryBeanNotInitializedException themselves now, as appropriate.
     * @return an instance of the bean (can be {@code null})
     * @throws Exception in case of creation errors
     * @see FactoryBeanNotInitializedException
     */
    @Nullable
    T getObject() throws Exception;

    /**
     * Return the type of object that this FactoryBean creates,
     * or {@code null} if not known in advance.
     * <p>This allows one to check for specific types of beans without
     * instantiating objects, for example on autowiring.
     * <p>In the case of implementations that are creating a singleton object,
     * this method should try to avoid singleton creation as far as possible;
     * it should rather estimate the type in advance.
     * For prototypes, returning a meaningful type here is advisable too.
     * <p>This method can be called <i>before</i> this FactoryBean has
     * been fully initialized. It must not rely on state created during
     * initialization; of course, it can still use such state if available.
     * <p><b>NOTE:</b> Autowiring will simply ignore FactoryBeans that return
     * {@code null} here. Therefore it is highly recommended to implement
     * this method properly, using the current state of the FactoryBean.
     * @return the type of object that this FactoryBean creates,
     * or {@code null} if not known at the time of the call
     * @see ListableBeanFactory#getBeansOfType
     */
    @Nullable
    Class<?> getObjectType();

    /**
     * Is the object managed by this factory a singleton? That is,
     * will {@link #getObject()} always return the same object
     * (a reference that can be cached)?
     * <p><b>NOTE:</b> If a FactoryBean indicates to hold a singleton object,
     * the object returned from {@code getObject()} might get cached
     * by the owning BeanFactory. Hence, do not return {@code true}
     * unless the FactoryBean always exposes the same reference.
     * <p>The singleton status of the FactoryBean itself will generally
     * be provided by the owning BeanFactory; usually, it has to be
     * defined as singleton there.
     * <p><b>NOTE:</b> This method returning {@code false} does not
     * necessarily indicate that returned objects are independent instances.
     * An implementation of the extended {@link SmartFactoryBean} interface
     * may explicitly indicate independent instances through its
     * {@link SmartFactoryBean#isPrototype()} method. Plain {@link FactoryBean}
     * implementations which do not implement this extended interface are
     * simply assumed to always return independent instances if the
     * {@code isSingleton()} implementation returns {@code false}.
```

```
     * <p>The default implementation returns {@code true}, since a
     * {@code FactoryBean} typically manages a singleton instance.
     * @return whether the exposed object is a singleton
     * @see #getObject()
     * @see SmartFactoryBean#isPrototype()
     */
    default boolean isSingleton() {
        return true;
    }
}
```

2. 实现FactoryBean接口

```
package com.atguigu.component;

import org.springframework.beans.factory.FactoryBean;

/**
 * 包名:com.atguigu.component
 *
 * @author Leevi
 * 日期2021-08-31  09:03
 * 要使用FactoryBean机制就得写一个类实现FactoryBean接口,接口的泛型表示你想通过这个
FactoryBean创建什么对象
 *
 * FactoryBean机制:你在spring的配置文件中进行IOC配置的是HappyComponentFactoryBean类，但
是真正创建出来存储在核心容器中的对象是
 *                HappyComponentFactoryBean对象调用getObject()方法所获取的对象
(HappyComponent)
 */
public class HappyComponentFactoryBean implements FactoryBean<HappyComponent> {
    private String componentName;

    public void setComponentName(String componentName) {
        this.componentName = componentName;
    }

    @Override
    public HappyComponent getObject() throws Exception {
        HappyComponent happyComponent = new HappyComponent();
        happyComponent.setComponentName(componentName);
        return happyComponent;
    }

    @Override
    public Class<?> getObjectType() {
        return null;
    }
}
```

3. 配置bean

```
    <?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="happyComponent"
class="com.atguigu.component.HappyComponentFactoryBean">
            <property name="componentName" value="奥巴马"></property>
        </bean>
    </beans>
```

4. 测试获取bean

```
package com.atguigu;

import com.atguigu.component.HappyComponent;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * 包名:com.atguigu
 *
 * @author Leevi
 * 日期2021-08-31  09:08
 */
public class TestFactoryBean {
    @Test
    public void testGetBean(){
        //1. 创建核心容器
        ApplicationContext act = new ClassPathXmlApplicationContext("spring-
application.xml");
        //2.从核心容器中获取对象
        HappyComponent happyComponent = (HappyComponent)
act.getBean("happyComponent");
    }
}
```