

# 韩顺平 Spring(轻量级容器框架) - 2022 版

## 1 Spring 基本介绍

### 1.1 官方资料

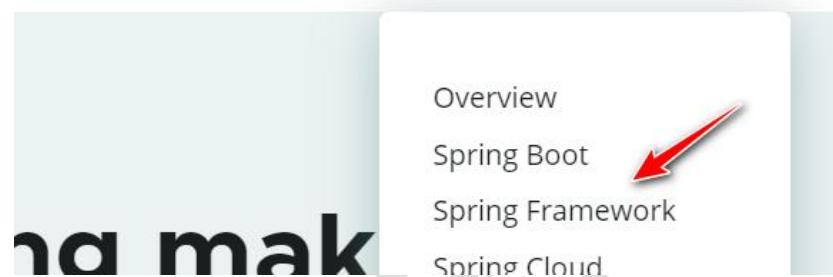
1.1.1 官网 <https://spring.io/>

1.1.2 Spring5 下载

1. 进入官网 <https://spring.io/>

2. 进入 Spring5

Why Spring ▾ Learn ▾ Projects ▾ Training Su



3. 进入 Spring5 的 github

Spring Framework 5.3.15



4. 进入 Spring5 的 github

下拉 Access to Binaries，进入 Spring Framework Artifacts

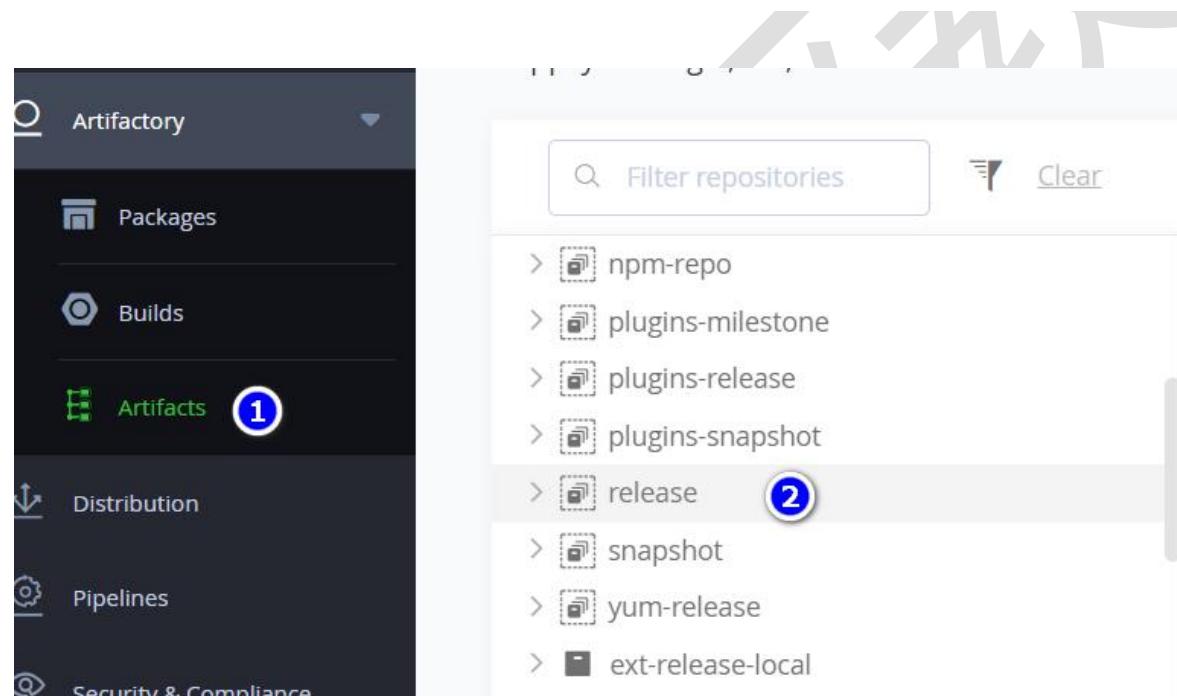
## Access to Binaries

For access to artifacts or a distribution zip, see the [Spring Framework Artifacts](#) wiki page.

## Downloading a Distribution

If for whatever reason you are not using a build system with dependency management capability, you can download Spring Framework distribution zips from the Spring repository at <https://repo.spring.io>. These distributions contain binary jar files, as well as Javadoc and reference documentation, but do not contain external dependencies.

To create a distribution with all dependencies locally you can build from source. See [Build Zip via Maven](#).



release

- > com
- > io
- org
  - > apache
  - > cloudfoundry
  - > projectreactor
  - springframework
    - > amqp
    - > analytics
    - > android

Filter repositories Clear My Favorites (0)

General Properties

Info

Name: spring [Download Address](#)

Repository Path: release/org/springframework/spring/

URL to file: <https://repo.spring.io/artifactory/release/org/springframework/spring/>

Created: 30-09-19 09:06:32 +00:00

Package Information

-----拷贝下载地址，打开---

← → C 🔒 repo.spring.io/ui/native/release/org/springframework/spring/

## Index of release/org/springframework/spring

Name	Last Modified
<a href="#">..</a>	
<a href="#">1.0/</a>	27-04-17 10:22:05 +0800
<a href="#">1.0-m4/</a>	27-04-17 20:56:53 +0800
<a href="#">1.0-rc1/</a>	27-04-17 21:31:58 +0800
<a href="#">1.0.1/</a>	28-04-17 02:06:15 +0800
<a href="#">1.1/</a>	30-04-17 15:54:13 +0800
<a href="#">1.1-rc1/</a>	26-01-17 09:14:00 +0800

-----选择 5.3.8，点击进入，即可下载----- 机制和原理

← → C 🔒 repo.spring.io/ui/native/release/org/springframework/spring/5.3.8/

## Index of release/org/springframework/sp

Name	Last Modified
<a href="#">..</a>	
<a href="#">spring-5.3.8-dist.zip</a>	09-06-21 15:50:58 +0800
<a href="#">spring-5.3.8-docs.zip</a>	09-06-21 15:50:54 +0800
<a href="#">spring-5.3.8-schema.zip</a>	09-06-21 15:50:53 +0800
<a href="#">spring-5.3.8.pom</a>	09-06-21 15:50:49 +0800
<a href="#">spring-5.3.8.pom.asc</a>	09-06-21 16:05:28 +0800

Artifactory Online Server at localhost Port 8081

1.1.3 在线文档: <https://docs.spring.io/spring-framework/docs/current/reference/html/>

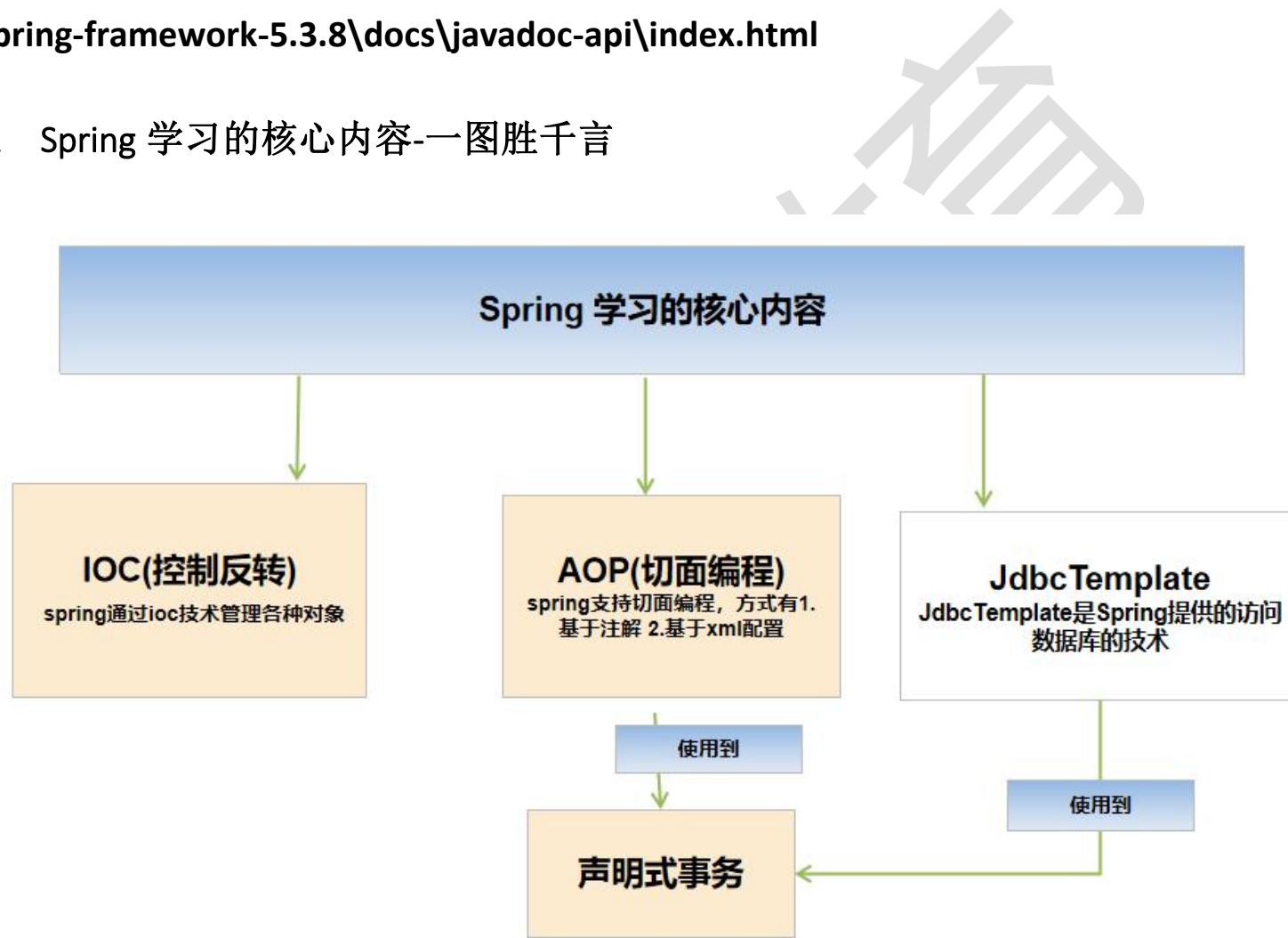
1.1.4 离线文档：解压 [spring-5.3.8-dist.zip](#)

[spring-framework-5.3.8\docs\reference\html\index.html](#)

1.1.5 离线 API：解压 [spring-5.3.8-dist.zip](#)

[spring-framework-5.3.8\docs\javadoc-api\index.html](#)

## 1.2 Spring 学习的核心内容-一图胜千言



老韩解读上图：

1、Spring 核心学习内容 IOC、AOP, jdbcTemplate, 声明式事务

2. IOC: 控制反转，可以管理 java 对象

3. AOP：切面编程

4. JDBCTemplate：是 spring 提供一套访问数据库的技术，应用性强，相对好理解

5. 声明式事务：基于 ioc/aop 实现事务管理，理解有需要小伙伴花时间

6. IOC, AOP 是重点同时难点

### 1.3 Spring 几个重要概念

1. Spring 可以整合其他的框架(老韩解读: Spring 是管理框架的框架)

2. Spring 有两个核心的概念: IOC 和 AOP

3. IOC [Inversion Of Control 反转控制]

• 传统的开发模式[JdbcUtils / 反射]

程序----->环境 //程序读取环境配置，然后自己创建对象.

## 传统的开发模式



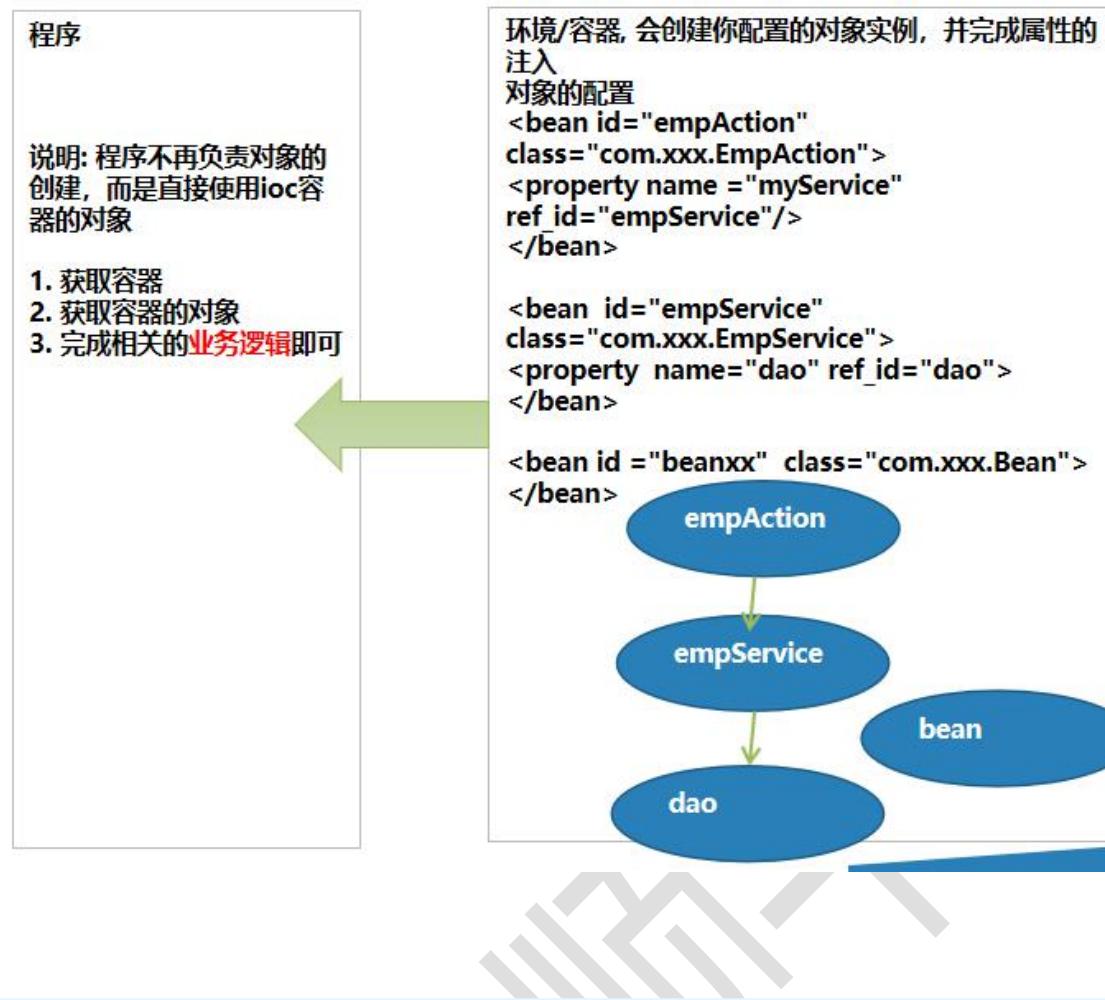
老韩解读上图(以连接到数据库为例说明)

1. 程序员编写程序，在程序中读取配置信息
2. 创建对象，`new Object???( ) // 反射方式`
3. 使用对象完成任务

- IOC 的开发模式 [EmpAction EmpService EmpDao Emp]

程序<-----容器 //容器创建好对象，程序直接使用.

## ioc(控制反转 )



### 老韩解读上图

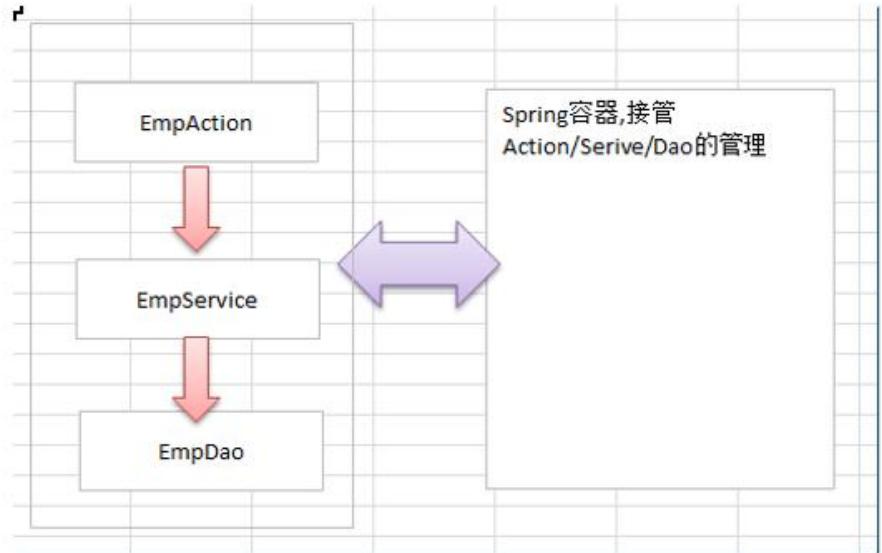
- 1、Spring 根据配置文件 xml/注解, 创建对象, 并放入到容器(ConcurrentHashMap)中, 并且可以完成对象之间的依赖
- 2、当需要使用某个对象实例的时候, 就直接从容器中获取即可
- 3、程序员可以更加关注如何使用对象完成相应的业务, (以前是 new ... ==> 注解/配置方式)

4. DI—Dependency Injection 依赖注入，可以理解成是 IOC 的另外叫法.

5. Spring 最大的价值，通过配置，给程序提供需要使用的

web 层[Servlet(Action/Controller)]/Service/Dao/[JavaBean/entity]对象，

这个是核心价值所在，也是 ioc 的具体体现，实现解耦.



## 1.4 Spring 快速入门

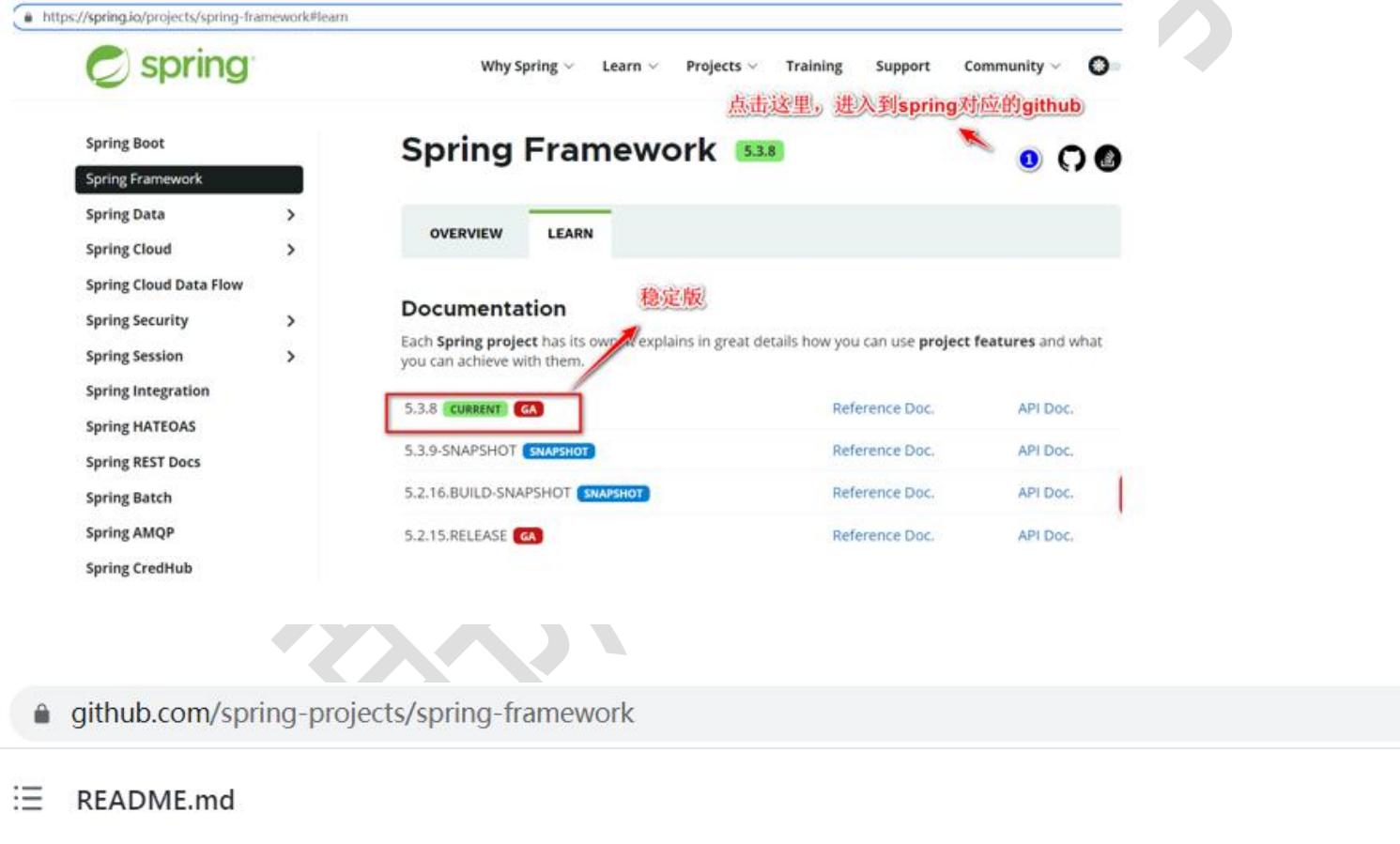
### 1.4.1 需求说明

1. 通过 Spring 的方式[配置文件]，获取 JavaBean: Monster 的对象，并给该的对象属性赋值，输出该对象信息。

```
Monster{monsterId=1, name='牛魔王', skill='牛魔王拳'}
```

### 1.4.2 完成步骤

1. 下载 spring5 开发包(老韩说明：下载的位置不太好找，小伙伴注意看.). spring 官网  
<https://spring.io/projects/spring-framework#learn>



The screenshot shows the official Spring Framework documentation page. At the top, there's a navigation bar with links for Why Spring, Learn, Projects, Training, Support, and Community. Below the navigation is a large banner for "Spring Framework 5.3.8". The banner has tabs for "OVERVIEW" and "LEARN", with "LEARN" being the active tab. A red arrow points from the text "点击这里，进入到spring对应的github" to the GitHub icon in the top right corner of the banner. Another red arrow points to the "5.3.8 CURRENT GA" link under the Documentation section, which is highlighted with a red box. The Documentation section also contains text about Spring projects and links to Reference Doc. and API Doc. for various versions: 5.3.8, 5.3.9-SNAPSHOT, 5.2.16.BUILD-SNAPSHOT, and 5.2.15.RELEASE. At the bottom of the page, there's a link to "README.md" and a section titled "Access to Binaries" with a note about artifacts and a distribution zip.

https://spring.io/projects/spring-framework#learn

Why Spring ▾ Learn ▾ Projects ▾ Training Support Community ▾

点击这里，进入到spring对应的github

Spring Framework 5.3.8

OVERVIEW LEARN

Documentation 稳定版

Each Spring project has its own documentation that explains in great details how you can use project features and what you can achieve with them.

5.3.8 CURRENT GA

5.3.9-SNAPSHOT SNAPSHOT

5.2.16.BUILD-SNAPSHOT SNAPSHOT

5.2.15.RELEASE GA

Reference Doc. API Doc.

Reference Doc. API Doc.

Reference Doc. API Doc.

Reference Doc. API Doc.

github.com/spring-projects/spring-framework

README.md

## Access to Binaries

For access to artifacts or a distribution zip, see the [Spring Framework Artifacts](#) wiki page.

C [github.com/spring-projects/spring-framework/wiki/Spring-Framework-Artifacts](https://github.com/spring-projects/spring-framework/wiki/Spring-Framework-Artifacts)

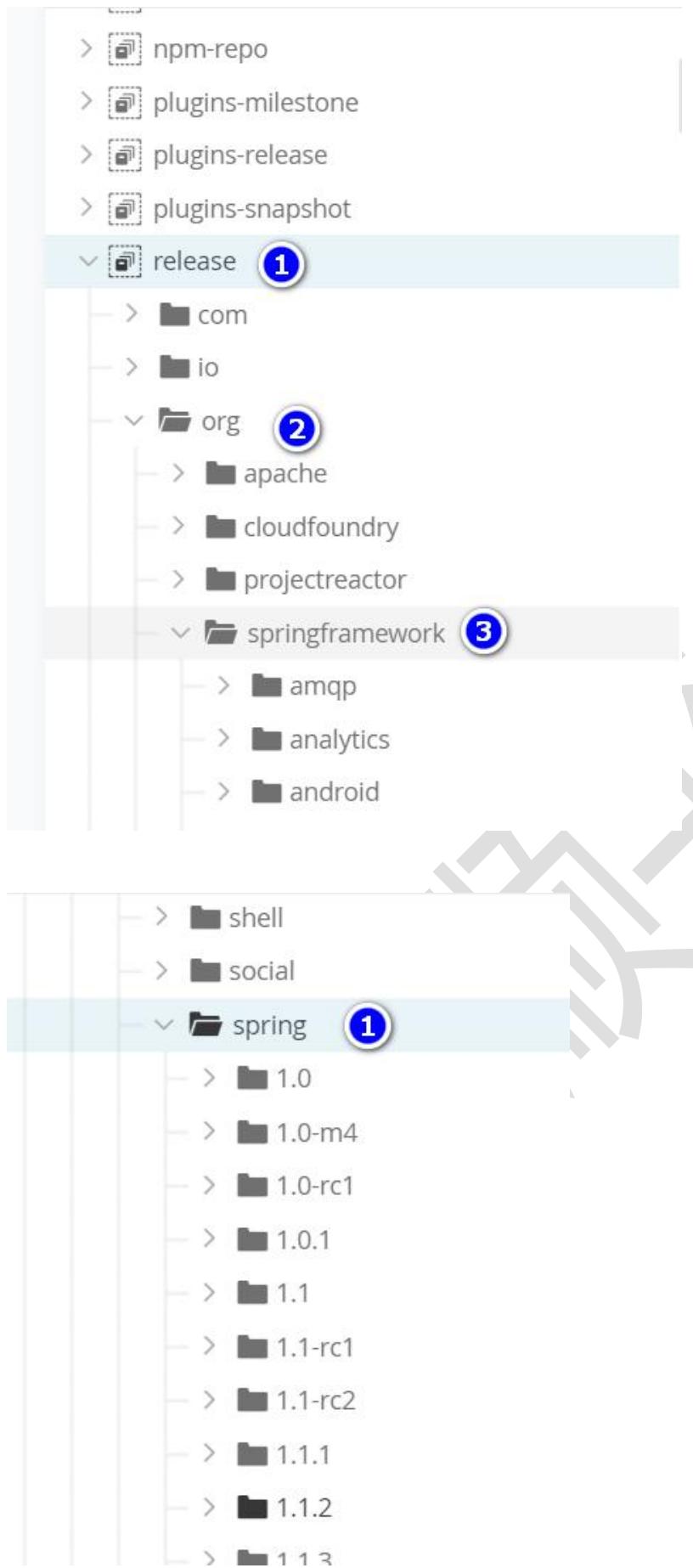
```
</dependency>
```

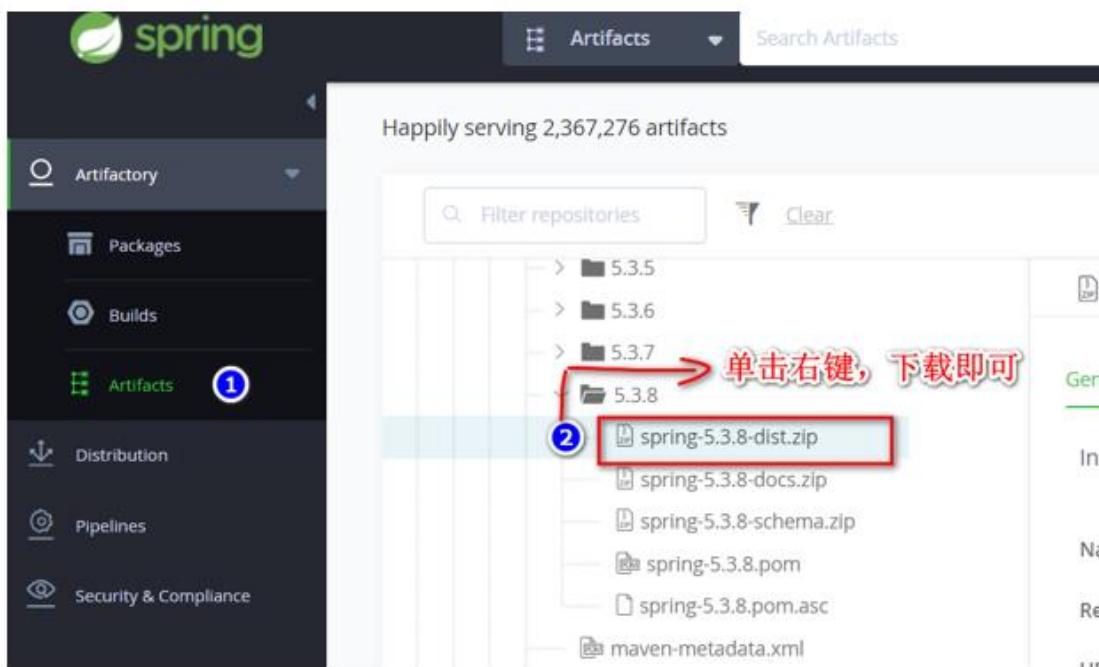
## Spring Repositories

Snapshot, milestone, and release candidate versions are published to an Artifactory in the Web UI at <https://repo.spring.io> to browse the Spring Artifactory, or go directly to <https://repo.spring.io/ui/repos/tree/General/release>.

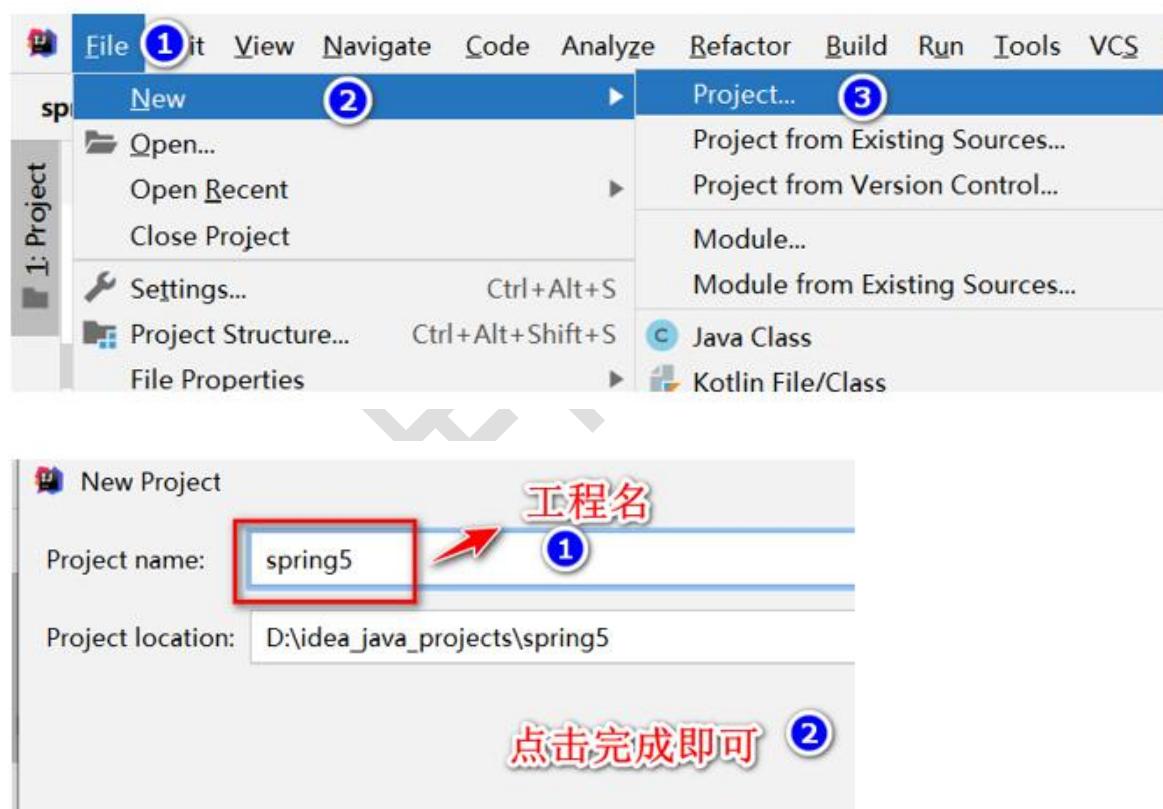
The screenshot shows the Spring Artifactory web interface. The top navigation bar includes a back/forward button, a refresh icon, and a URL field showing [repo.spring.io/ui/repos/tree/General/release](https://repo.spring.io/ui/repos/tree/General/release). The header features the Spring logo and tabs for 'Artifacts' and 'Search Artifact'. A message at the top states 'Happily serving 2,372,856 artifacts'. On the left, a sidebar menu lists 'Artifactory' (selected), 'Packages', 'Builds', 'Artifacts' (with a red circle containing '1'), 'Distribution', 'Pipelines', and 'Security & Compliance'. The main content area displays a list of repositories under the heading 'Filter repositories' with a 'Clear' button. The list includes: libs-release, libs-release-remote, libs-snapshot, libs-spring-dataflow-private-release, libs-spring-dataflow-private-snapshot, libs-spring-framework-build, milestone, npm-repo, plugins-milestone, plugins-release, plugins-snapshot, release (highlighted with a red circle containing '2'), and snapshot.

- > libs-release
- > libs-release-remote
- > libs-snapshot
- > libs-spring-dataflow-private-release
- > libs-spring-dataflow-private-snapshot
- > libs-spring-framework-build
- > milestone
- > npm-repo
- > plugins-milestone
- > plugins-release
- > plugins-snapshot
- > release **2**
- > snapshot

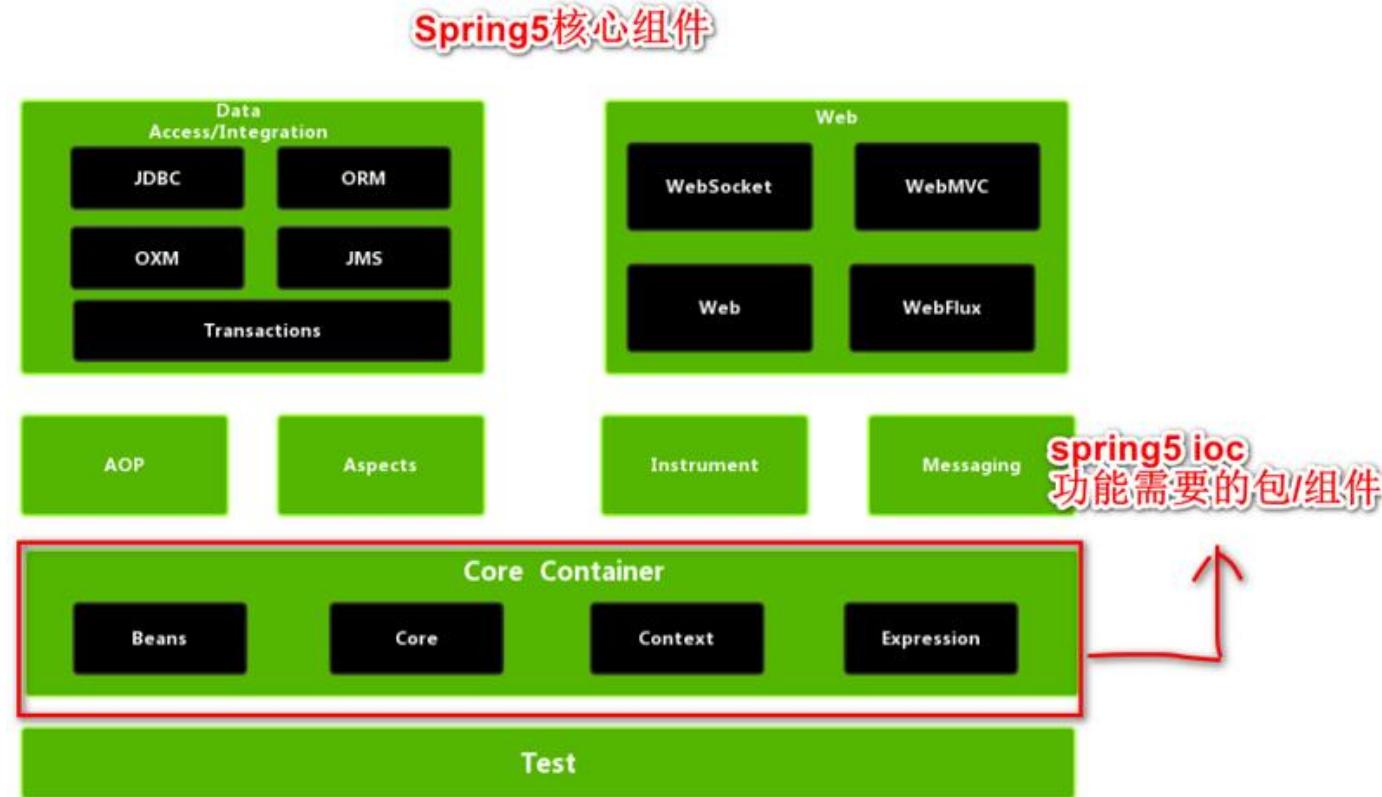




## 2. 创建 Java 工程：spring5，为了清晰 Spring5 的各 jar 包作用，老师使用 Java 工程



### 3. 引入开发 spring5 的基本包



#### 4. 创建代码实现【参考老师视频讲解和项目源码】

##### 1.4.3 注意事项和细节

###### 1、说明

```
ClassPathXmlApplicationContext ioc =
```

```
    new ClassPathXmlApplicationContext("beans.xml");
```

为什么读取到 beans.xml

###### 2、解释一下类加载路径。可以给学员输出一下：

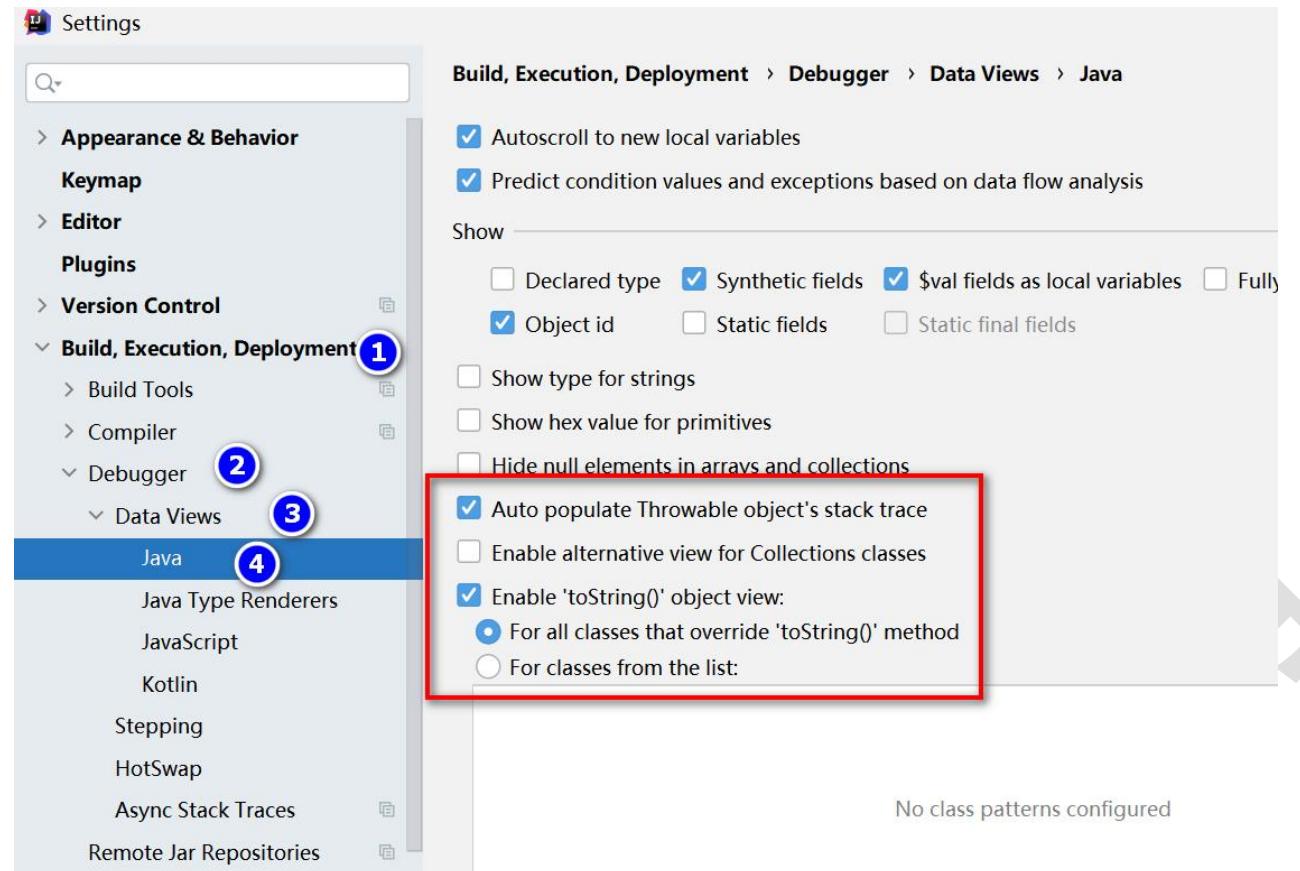
```
// 获取类加载的路径
```

```
// File f = new File(this.getClass().getResource("/").getPath());
```

```
// System.out.println(f);
```

###### 3、debug 看看 spring 容器结构/机制，记住你是 OOP 程序员，重要！截图

- 老韩说明：注意配置 debugger，否则你看到的 debug 视图和老师不一样



```

> this = {SpringBeanTest@2293}
< ioc = {ClassPathXmlApplicationContext@2292} "org.springframework.context.support.ClassPath
  f configResources = null
  f validating = true
> f configLocations = {String[1]@2321}
  f setIdCalled = false
  f allowBeanDefinitionOverriding = null
  f allowCircularReferences = null
2 f beanFactory = {DefaultListableBeanFactory@2322} "org.springframework.beans.factory.suppo
  > f serializationId = "org.springframework.context.support.ClassPathXmlApplicationContext@2
    f allowBeanDefinitionOverriding = true
    f allowEagerClassLoading = true
    f dependencyComparator = null
  > f autowireCandidateResolver = {SimpleAutowireCandidateResolver@2649}
  > f resolvableDependencies = {ConcurrentHashMap@2650} "{interface org.springframework.c
3 > f beanDefinitionMap = {ConcurrentHashMap@2651} "{monster01=Generic bean: class [com
  > f mergedBeanDefinitionHolders = {ConcurrentHashMap@2652} "{}"
  > f allBeanNamesByType = {ConcurrentHashMap@2653} "{}"
  > f singletonBeanNamesByType = {ConcurrentHashMap@2654} "{}"

```

1. beanDefinitionMap 类型是 ConcurrentHashMap 集合  
 2. 存放beanx.xml 中的bean节点配置的bean对象的信息

```

> f resolvableDependencies = {ConcurrentHashMap@2650} "interface org.springframework.cc
< f beanDefinitionMap = {ConcurrentHashMap@2651} "{monster01=Generic bean: class [com.
  < f table = {ConcurrentHashMap$Node[512]@2706} ①
    └ 0 = null
    └ 1 = null
    └ 2 = null
    └ 3 = null
    └ 4 = null
    └ 5 = null
    └ 6 = null
    └ 7 = null
    └ 8 = null
    └ 9 = null
    └ 10 = null
    └ 11 = null

```

1. 在beanDefinitionMap 中有属性table  
 2. table是数组 类型是 ConcurrentHashMap\$Node  
 3. 因为是数组，所以可以存放很多的bean对象信息，就是 beans.xml 配置  
 4. 初始化是512，当超过时，会自动扩容

```

    └ 215 = null
    └ 216 = null
    < f 217 = {ConcurrentHashMap$Node@2711} "monster01=Generic bean: class [com.hspedu.spring.bean.Monster]; scope=
      < f hash = 830607065
      < f key = "monster01"
        > f value = {char[9]@2767}
          < f hash = -1316831397
        < f val = {GenericBeanDefinition@2698} "Generic bean: class [com.hspedu.spring.bean.Monster]; scope=; abstract=false
          < f parentName = null
          > f beanClass = "com.hspedu.spring.bean.Monster"
          > f scope = ""
          < f abstractFlag = false
          > f lazyInit = {Boolean@2716} false
          < f autowireMode = 0

```

1. 通过hash算法我们的 Monster01 对象的信息就保存在index=217位置  
 2. 保存是以 ConcurrentHashMap\$Node 类型保存  
 3. key 就是 beans.xml 中配置的 monster01  
 4. value 就是 monster01 对象的信息【比如属性/属性值/类信息/是不是懒加载】

```

f factoryMethodName = null
f constructorArgaumentValues = null
✓ f propertyValues = {MutablePropertyValues@2718} "PropertyValues: length=3; bean property 'monsterId'
  ✓ f propertyValueList = {ArrayList@2724} "[bean property 'monsterId', bean property 'name', bean prop
    ✓ f elementData = {Object[3]@2726}
      ✓ 0 = {PropertyValue@2727} "bean property 'monsterId'"
        > f name = "monsterId"
        ✓ f value = {TypedStringValue@2734} "TypedStringValue: value [100], target type [null]"
          > f value = "100"
          f targetType = null
          f source = null
          f specifiedTypeName = null
          f dynamic = false
          f optional = false
          f converted = false
          f convertedValue = null

```

这里就是记录beans.xml中配置的  
monster01对象的属性名/属性值

1.

```

> f applicationStartup = {DefaultApplicationStartup@2335}
> f factoryBeanObjectCache = {ConcurrentHashMap@2676} "{}"
✓ f singletonObjects = {ConcurrentHashMap@2677} "{applicationStartup=org.springframework.core.metrics.Default
  ✓ f table = {ConcurrentHashMap$Node[512]@2750}
    0 = null
    1 = null
    2 = null
    3 = null
    4 = null
    5 = null
    6 = null
    7 = null
    8 = null

```

1. 在beanfactory中，属性 singletonObjects  
 2. 类型是 ConcurrentHashMap  
 3. 还有一个属性 table 类型是 ConcurrentHashMap\$Node  
 4. 如果你在beans.xml文件中配置的对象是 单例的 就会初始化防在table中

215 = null  
216 = null  
217 = {ConcurrentHashMap\$Node@2757} "monster01=Monster{monsterId=100, name='牛魔王'  
    f hash = 830607065  
    f key = "monster01"  
    f val = {Monster@2763} "Monster{monsterId=100, name='牛魔王', skill='芭蕉扇'"  
        f monsterId = {Integer@2765} 100  
        f name = "牛魔王"  
        f skill = "芭蕉扇"  
        f next = null  
218 = null  
219 = null  
220 = null  
    f beanDefinitionNames = {ArrayList@2655} "[monster01]"  
        f elementData = {Object[256]@2710}  
            0 = "monster01"  
            1 = null  
            2 = null  
            3 = null  
            4 = null  
            5 = null  
            6 = null  
            7 = null

1. 在beanfactory中，还有一个属性 **beanDefinitionNames**
2. 记录了我们在**beans.xml** 中配置的 bean 的名称，方便查找

#### 4、查看容器注入了哪些 bean 对象，会输出 bean 的 id

```
// String[] str = ioc.getBeanDefinitionNames();
```

```
// for (String string : str) {
```

```
//     System.out.println("..." + string);
```

```
//}
```

## 1.5 手动开发- 简单的 Spring 基于 XML 配置的程序

### 1.5.1 需求说明

1. 自己写一个简单的 Spring 容器，通过读取 beans.xml，获取第 1 个 JavaBean: Monster 的对象，并给该的对象属性赋值，放入到容器中，输出该对象信息。

```
6. <property name="monsterId" value="100"> 用于给该对象的属性  
-->  
<bean class="com.hspedu.spring.bean.Monster" id="monster01">  
    <property name="monsterId" value="100"/>  
    <property name="name" value="牛魔王"/>  
    <property name="skill" value="芭蕉扇"/>  
</bean>
```

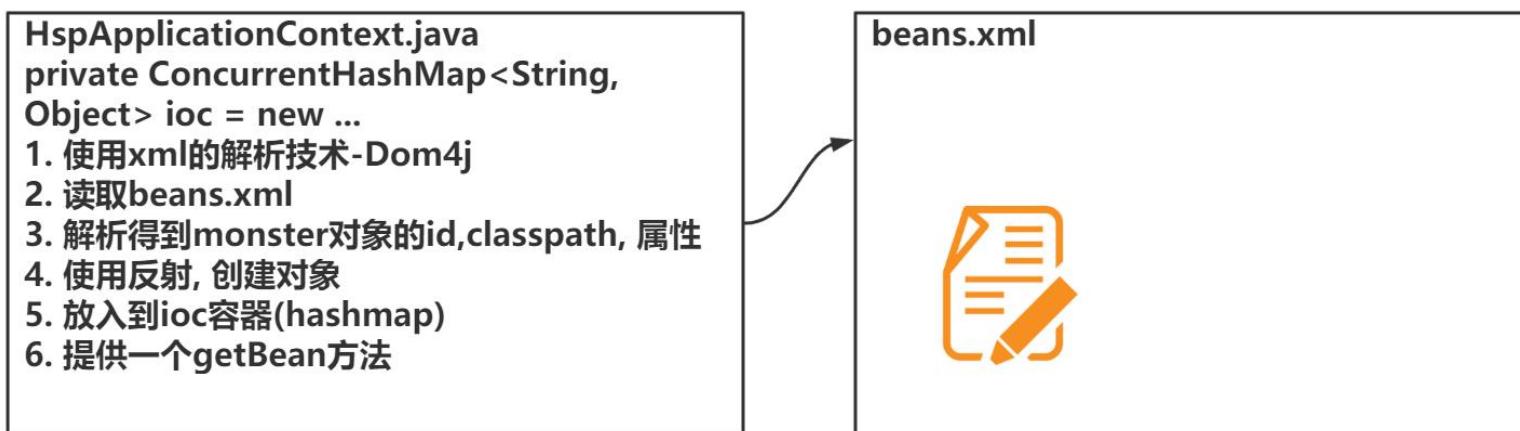
```
Monster{monsterId=1, name='牛魔王', skill='牛魔王拳'}
```

2. 也就是说，不使用 Spring 原生框架，我们自己简单模拟实现

3. 可以让小伙伴了解 Spring 容器的简单机制

### 1.5.2 思路分析

#### 1. 思路分析/图解



### 1.5.3 完成步骤 【参看老师的视频讲解和项目源码】

1. 导入 Dom4j jar 包

2.

D:\hsppedu\_ssm\_temp\spring5\src\com\hsppedu\spring\test\HspApplicationContext.java

package com.hsppedu.spring.test;

import com.hsppedu.spring.beans.Monster;

import org.dom4j.Document;

import org.dom4j.DocumentException;

import org.dom4j.Element;

```
import org.dom4j.io.SAXReader;

import java.io.File;
import java.util.List;
import java.util.concurrent.ConcurrentHashMap;

/**
 * @author 韩顺平
 * @version 1.0
 */

public class HspApplicationContext {

    private ConcurrentHashMap<String, Object> ioc =
        new ConcurrentHashMap<>();

    /**
     * 后面老师还会非常详细的写机制，这里简写，体会即可
     * @param iocBeanXmlFile
     * @throws DocumentException
     * @throws ClassNotFoundException
     * @throws IllegalAccessException
     */
}
```

```
* @throws InstantiationException
```

```
*/
```

```
public HspApplicationContext(String iocBeanXmlFile) throws DocumentException,  
ClassNotFoundException, IllegalAccessException, InstantiationException {
```

```
//反射的方式得到->使用 dom4j
```

```
//得到一个解析器
```

```
SAXReader reader = new SAXReader();
```

```
Document document = reader.read(new File(iocBeanXmlFile));
```

```
//1. 得到rootElement
```

```
Element rootElement = document.getRootElement();
```

```
//2. 获取第 1 个 bean, 如有所有, 就遍历-老师就是一个示意
```

```
Element bean = (Element) rootElement.elements("bean").get(0);
```

```
String id = bean.attributeValue("id");
```

```
String classFullPath = bean.attributeValue("class");
```

```
List<Element> property = bean.elements("property");
```

```
Integer monsterId = Integer.parseInt(property.get(0).attributeValue("value"));
```

```
String name = property.get(1).attributeValue("value");
```

```
String skill = property.get(2).attributeValue("value");
```

```
//3. 使用反射创建 bean 实例, 并放入到 ioc 中
```

```
Class cls = Class.forName(classFullPath);
```

```
Monster instance = (Monster)cls.newInstance();

instance.setMonsterId(monsterId);

instance.setName(name);

instance.setSkill(skill);

ioc.put(id, instance);

}

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

### 3. 创建 D:\hsedu\_ssm\_temp\spring5\src\com\hsedu\spring\test\HspSpringIOTest.java

```
package com.hsedu.spring.test;

import com.hsedu.spring.beans.Monster;
import org.dom4j.DocumentException;

/**
 * @author 韩顺平
 */
```

```
* @version 1.0  
*/  
  
public class HspSpringIOTest {  
    public static void main(String[] args) throws DocumentException,  
IllegalAccessException, InstantiationException, ClassNotFoundException {  
    //得到beans.xml 文件.  
    HspApplicationContext ioc = new HspApplicationContext("src/beans.xml");  
    Monster monster01 = (Monster)ioc.getBean("monster01");  
    System.out.println("ioc 的 monster01= " + monster01);  
}  
}
```

#### 4. 完成测试

```
D:\program\hspjdk8\bin\java.exe ...  
ioc的monster01= Monster{monsterId=1, name='牛魔王', skill='牛魔  
Process finished with exit code 0
```

## 1.6 Spring 原生容器底层结构

### 1.6.1 一图胜千言

1. Debug 看一下 Spring 容器的处理机制，使用快速入门代码 Debug 即可

```
✓  ioc = {ClassPathXmlApplicationContext@1714} "org.springframework.context.support.ClassPathXmlApplicationConte
  f configResources = null          ①
  f validating = true
  > f configLocations = {String[1]@1766}
    f setIdCalled = false
    f allowBeanDefinitionOverriding = null
    f allowCircularReferences = null      ②
  > f beanFactory = {DefaultListableBeanFactory@1767} "org.springframework.beans.factory.support.DefaultListableBeanFactory@1767
    f serializationId = "org.springframework.context.support.ClassPathXmlApplicationContext@1714"
    f allowBeanDefinitionOverriding = true
    f allowEagerClassLoading = true
    f dependencyComparator = null
    > f autowireCandidateResolver = {SimpleAutowireCandidateResolver@1996}
    > f resolvableDependencies = {ConcurrentHashMap@1997} "{interface org.springframework.aop.TargetSource@1998}"
  ③ f beanDefinitionMap = {ConcurrentHashMap@1998} "{monster01=Generic bean: class [..]}
```

```
✓ f beanDefinitionMap = {ConcurrentHashMap@1998} "{monster01=Generi
  ✓ f table = {ConcurrentHashMap$Node[512]@2038}
    └─ 0 = null
    └─ 1 = null
    └─ 2 = null
    └─ 3 = null
    └─ 4 = null
    └─ 5 = null
    └─ 6 = null
```

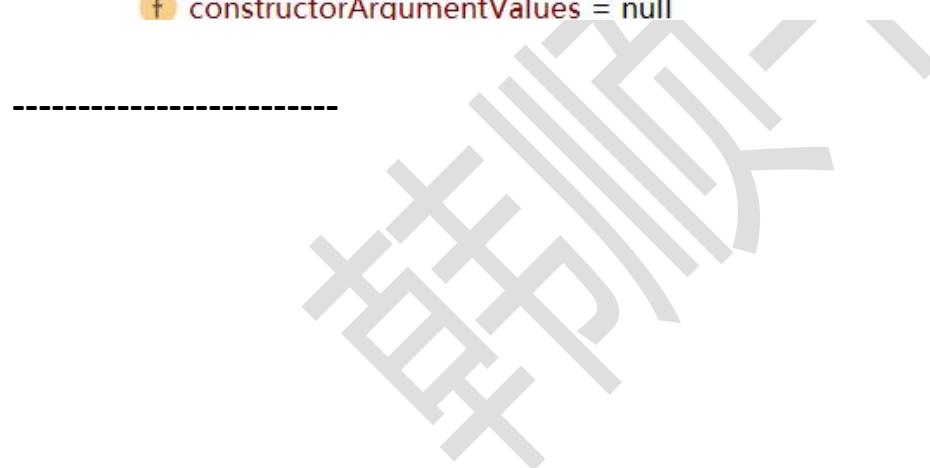
4

```
└─ 216 = null
  ✓ 217 = {ConcurrentHashMap$Node@2039} "monster01=Generic bean: class [com.hspedu.spring.beans.Monster]; scope=; a
    f hash = 830607065
    > f key = "monster01"
    > f val = {GenericBeanDefinition@2042} "Generic bean: class [com.hspedu.spring.beans.Monster]; scope=; abstract=false;
      f next = null
    └─ 218 = null
    └─ 219 = null
    └─ 220 = null
```

1

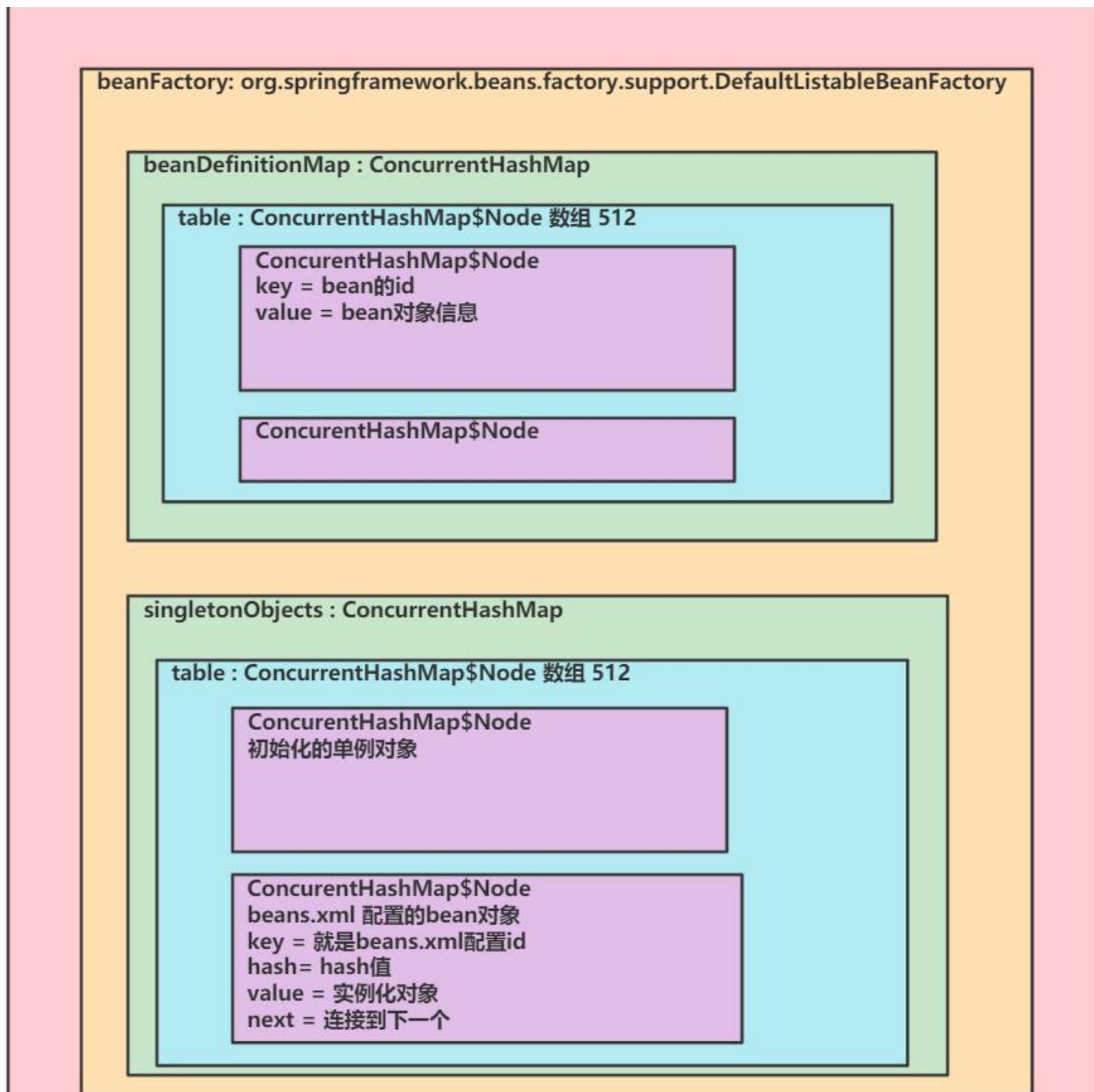
2

```
> t key = monsteru1
< f val = {GenericBeanDefinition@2042} "Generic bean: class [com.hspedu.spring.beans.Monster]; scope=; i
  < f parentName = null
  > f beanClass = "com.hspedu.spring.beans.Monster" ①
  > f scope = ""
  < f abstractFlag = false
  > f lazyInit = {Boolean@2046} false ②
    < f autowireMode = 0
    < f dependencyCheck = 0
    < f dependsOn = null
    < f autowireCandidate = true
    < f primary = false
  > f qualifiers = {LinkedHashMap@2047} "{}"
    < f instanceSupplier = null
    < f nonPublicAccessAllowed = true
    < f lenientConstructorResolution = true
    < f factoryBeanName = null
    < f factoryMethodName = null
    < f constructorArgumentValues = null
```



```
✓ f propertyValues = {MutablePropertyValues@2048} "PropertyValues: length=3; bean property 'monsterId'; bear
  ✓ f PropertyValueList = {ArrayList@2058} "[bean property 'monsterId', bean property 'name', bean property 'skill']"
    ✓ f elementData = {Object[3]@2060}
      > 0 = {PropertyValue@2061} "bean property 'monsterId'" ①
      < 1 = {PropertyValue@2062} "bean property 'name'" ②
        > f name = "name"
        > f value = {TypedStringValue@2072} "TypedStringValue: value [牛魔王], target type [null]"
          f optional = false
          f converted = false
          f convertedValue = null
          f conversionNecessary = null
          f resolvedTokens = null
          f source = null
        > f attributes = {LinkedHashMap@2073} "{}"
      > 2 = {PropertyValue@2063} "bean property 'skill'" ③
```

## 2. 老韩解图



### 1.6.2 课后作业题

- 在 `beans.xml` 中，我们注入 2 个 `Monster` 对象，但是不指定 `id`,如下

```
<bean class="com.hspedu.spring.beans.Monster">  
    <property name="monsterId" value="1010"/>  
    <property name="name" value="牛魔王~"/>  
    <property name="skill" value="芭蕉扇~"/>  
</bean>
```

```
<bean class="com.hspedu.spring.beans.Monster">  
    <property name="monsterId" value="666"/>  
    <property name="name" value="牛魔王~~!!"/>  
    <property name="skill" value="芭蕉扇~~"/>  
</bean>
```

## 2. 问题 1：运行会不会报错

答：不会报错，会正常运行

## 3. 问题 2：如果不报错，你能否找到分配的 id，并获得到该对象.

答：系统会默认分配 id，分配 id 的规则是 全类名#0，全类名#1 这样的规则来分配 id  
，我们可以通过 debug 方式来查看。

## 1.7 Spring 课堂练习

- 课堂练习 (10-15min): 创建一个 Car 类(id , name , price ), 具体要求如下:

1. 创建 ioc 容器文件(配置文件), 并配置一个 Car 对象(bean) .

2. 通过 java 程序到 ioc 容器获取该 bean 对象, 输出

提示: 比较简单, 随堂

## 2 Spring 管理 Bean-IOC

### 2.1 Spring 配置/管理 bean 介绍

#### 2.1.1 Bean 管理包括两方面

2.1.1.1 创建 bean 对象

2.1.1.2 给 bean 注入属性

#### 2.1.2 Bean 配置方式

2.1.2.1 基于 xml 文件配置方式

2.1.2.2 基于注解方式

### 2.2 基于 XML 配置 bean

#### 2.2.1 通过类型来获取 bean

2.2.1.1 应用案例

• 案例说明：

1. 通过 spring 的 ioc 容器，获取一个 bean 对象

2. 说明：获取 bean 的方式：按类型

- 完成步骤

1. 创建 Monster.java，前面已经有
2. 在 beans.xml 配置(说明前面已经配置好了)

```
<bean id="monster01" class="com.hspedu.spring.beans.Monster">  
    <property name="monsterId" value="1"/>  
    <property name="name" value="牛魔王"/>  
    <property name="skill" value="牛魔王拳"/>  
</bean>
```

3. 完成 测试 : 修 改
- D:\idea\_java\_projects\spring5\src\com\hspedu\spring\test\Spring5BeanTest.java, 通过类型来获取 bean 对象。

```
/**  
 * 通过类型来获取容器的bean对象  
 */  
  
@Test  
public void getMonsterByType() {
```

```
ApplicationContext ioc = new ClassPathXmlApplicationContext("beans.xml");  
Monster monster = ioc.getBean(Monster.class);  
System.out.println("monster=" + monster);
```

```
Monster monster2 = ioc.getBean(Monster.class);
System.out.println("monster == monster2 的值= " + (monster == monster2));
}
```

### 2.2.1.2 细节说明

- 按类型来获取 bean，要求 ioc 容器中的同一个类的 bean 只能有一个，否则会抛出异常 **NoUniqueBeanDefinitionException**

```
factory.NoUniqueBeanDefinitionException: No qualifying bean of type [com.itheima.spring.ioc.core.IocTest$1] found for dependency: expected at least one bean which qualifies as autowire candidate for that dependency. Dependency annotations: @org.springframework.beans.factory.annotation.Autowired(required=true)
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.resolveNamedBeans(DefaultListableBeanFactory.java:420)
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.resolveBean(DefaultListableBeanFactory.java:360)
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBean(DefaultListableBeanFactory.java:320)
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBean(DefaultListableBeanFactory.java:313)
```

- 这种方式的应用场景：比如 XxxAction/Servlet/Controller，或 XxxService 在一个线程中只需要一个对象实例(单例)的情况
- 老师这里在说明一下：在容器配置文件(比如 beans.xml)中给属性赋值，底层是通过 setter 方法完成的，这也是为什么我们需要提供 setter 方法的原因[演示，举例说明]

### 2.2.2 通过构造器配置 bean

#### 2.2.2.1 应用案例

- 案例说明：

在 spring 的 ioc 容器，可以通过构造器来配置 bean 对象

- 完成步骤

- 在 beans.xml 配置

```
<!--  
    在 spring 的 ioc 容器，可以通过构造器来配置 bean 对象  
-->  
<bean id="monster02" class="com.hspedu.spring.beans.Monster">  
    <constructor-arg value="2" index="0"/>  
    <constructor-arg value="蜘蛛精" index="1"/>  
    <constructor-arg value="吐口水" index="2"/>  
</bean>  
  
<!--  
    数据类型就是对应的 Java 数据类型，按构造器参数顺序  
-->  
<bean id="monster03" class="com.hspedu.spring.beans.Monster">  
    <constructor-arg value="3" type="java.lang.Integer"/>  
    <constructor-arg value="白骨精" type="java.lang.String"/>  
    <constructor-arg value="白骨鞭" type="java.lang.String"/>  
</bean>
```

## 2. 完成 测试：修改

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\test\Spring5BeanTest.java, 增加测试方法.

`@Test`

```
public void getMonsterByConstructor() {  
  
    ApplicationContext ioc = new ClassPathXmlApplicationContext("beans.xml");  
    Object monster02 = ioc.getBean("monster02");  
    Object monster03 = ioc.getBean("monster03");  
    System.out.println("monster02= " + monster02);  
    System.out.println("monster03= " + monster03);  
  
}
```

### 2.2.2.2 使用细节

1. 通过 `index` 属性来区分是第几个参数
2. 通过 `type` 属性来区分是什么类型(按照顺序)

### 2.2.3 通过 p 名称空间配置 bean

#### 2.2.3.1 应用实例

- 案例说明：

在 spring 的 ioc 容器，可以通过 p 名称空间来配置 bean 对象

- 完成步骤

1. 在 beans.xml 配置，增加命名空间配置

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

<!--

在 spring 的 ioc 容器，可以通过 p 名称空间来配置 bean 对象 -->

```
<bean id="monster04" class="com.hspedu.spring.beans.Monster"
      p:monsterId="4"
      p:name="红孩儿"
```

*p:skill="吐火~"*

/>

## 2. 完成 测 试 : 修 改

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\test\Spring5BeanTest.java, 增加测试方法.

*@Test*

```
public void getMonsterByP() {  
    ApplicationContext ioc = new ClassPathXmlApplicationContext("beans.xml");  
    Monster monster04 = ioc.getBean("monster04", Monster.class);  
    System.out.println("monster04=" + monster04);  
}
```

### 2.2.4 引用/注入其它 bean 对象

#### 2.2.4.1 实例演示

- 案例说明:

在 spring 的 ioc 容器, 可以通过 ref 来实现 bean 对象的相互引用

- 完成步骤

1. 创建 D:\idea\_java\_projects\spring5\src\com\hspedu\spring\dao\MemberDAOImpl.java

```
package com.hspedu.spring.dao;
```

```
/*
 * @author 韩顺平
 * @version 1.0
 */

public class MemberDAOImpl {

    public MemberDAOImpl() {
        System.out.println("MemberDAOImpl 构造器...");
    }

    public void add() {
        System.out.println("MemberDAOImpl add()方法");
    }
}
```

- 2.

创

建

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\service\MemberServiceImpl.java

```
package com.hspedu.spring.service;

import com.hspedu.spring.dao.MemberDAOImpl;

/**
 * @author 韩顺平
 * @version 1.0
 */

public class MemberServiceImpl {
    private MemberDAOImpl memberDAO;

    public MemberServiceImpl() {
        System.out.println("MemberServiceImpl 构造器~");
    }

    public void add() {
        System.out.println("MemberServiceImpl add()... ");
        memberDAO.add();
    }

    public void setMemberDAO(MemberDAOImpl memberDAO) {
        this.memberDAO = memberDAO;
    }
}
```

```
    }  
  
    public MemberDAOImpl getMemberDAO() {  
        return memberDAO;  
    }  
}
```

### 3. 在 beans.xml 配置

<!-- bean 对象的相互引用 -->

1. 其它含义和前面一样

2. ref 表示 memberDAO 这个属性将引用/指向 id = memberDAOImpl 对象

-->

```
<bean id="memberServiceImpl" class="com.hspedu.spring.service.MemberServiceImpl">  
    <property name="memberDAO" ref="memberDAOImpl"/>  
</bean>  
  
<bean id="memberDAOImpl" class="com.hspedu.spring.dao.MemberDAOImpl"/>
```

### 4. 完成 测试 : 修 改

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\test\Spring5BeanTest.java, 增加测试方法.

@Test

```
public void setBeanByRef() {  
    ApplicationContext ioc = new ClassPathXmlApplicationContext("beans.xml");  
    MemberServiceImpl memberServiceImpl = ioc.getBean("memberServiceImpl",  
        MemberServiceImpl.class);  
    memberServiceImpl.add();  
}
```

## 2.2.5 引用/注入内部 bean 对象

### 2.2.5.1 应用实例

- 案例说明

在 spring 的 ioc 容器，可以直接配置内部 bean 对象

- 完成步骤

1. 创建 MemberDAOImpl.java，前面有了
2. 创建 MemberServiceImpl.java，前面有了。
3. 修改 D:\idea\_java\_projects\spring5\src\beans.xml，增加配置

<!-- 引用/注入内部 bean 对象，直接在配置 bean 时注入 -->

```
<bean id="memberServiceImpl02">  
    class="com.hspedu.spring.service.MemberServiceImpl">  
        <property name="memberDAO">  
            <bean class="com.hspedu.spring.dao.MemberDAOImpl"/>  
        </property>  
    </bean>
```

#### 4. 完成测试：修改

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\test\Spring5BeanTest.java, 增加测试方法。

```
@Test  
public void setBeanByPro() {  
    ApplicationContext ioc = new ClassPathXmlApplicationContext("beans.xml");  
    MemberServiceImpl memberServiceImpl02 = ioc.getBean("memberServiceImpl02",  
        MemberServiceImpl.class);  
    memberServiceImpl02.add();  
}
```

### 2.2.6 引用/注入集合/数组类型

#### 2.2.6.1 应用实例

- 案例说明

## 在 spring 的 ioc 容器，看看如何给 bean 对象的集合/数组类型属性赋值



太上老君两个童子下界为妖，老君将其收伏

- 完成步骤

1. 创建 Monster.java，前面有了

2. 创建 D:\idea\_java\_projects\spring5\src\com\hspedu\spring\beans\Master.java

```
package com.hspedu.spring.beans;
```

```
import java.util.List;  
import java.util.Map;  
import java.util.Properties;  
import java.util.Set;
```

```
/**
```

```
* @author 韩顺平
```

```
* @version 1.0
```

```
*/
```

```
public class Master {
```

```
private String name;  
private List<Monster> monsterList;  
private Map<String, Monster> monsterMap;  
private Set<Monster> monsterSet;  
private String[] monsterName;  
//这个Properties 是 Hashtable 的子类，是key-value 的形式  
//这里Properties key 和value 都是 String  
private Properties pros;  
  
public Master() {  
}  
  
public Master(String name) {  
    this.name = name;  
}  
  
public Set<Monster> getMonsterSet() {  
    return monsterSet;  
}  
  
public void setMonsterSet(Set<Monster> monsterSet) {
```

```
this.monsterSet = monsterSet;  
}  
  
public String[] getMonsterName() {  
    return monsterName;  
}  
  
public void setMonsterName(String[] monsterName) {  
    this.monsterName = monsterName;  
}  
  
public String getName() {  
    return name;  
}  
  
public void setName(String name) {  
    this.name = name;  
}  
  
public List<Monster> getMonsterList() {  
    return monsterList;  
}
```

```
public void setMonsterList(List<Monster> monsterList) {  
    this.monsterList = monsterList;  
}  
  
public Map<String, Monster> getMonsterMap() {  
    return monsterMap;  
}  
  
public void setMonsterMap(Map<String, Monster> monsterMap) {  
    this.monsterMap = monsterMap;  
}  
  
public Properties getPros() {  
    return pros;  
}  
  
public void setPros(Properties pros) {  
    this.pros = pros;  
}
```

### 3. 配置 beans.xml

```
<!-- 给集合属性注入值  
-->  
  
<bean id="master01" class="com.hspedu.spring.beans.Master">  
    <property name="name" value="太上老君"/>  
    <!-- 给 bean 对象的 list 集合赋值 -->  
    <property name="monsterList">  
        <list>  
            <ref bean="monster03"/>  
            <ref bean="monster02"/>  
        </list>  
    </property>  
    <!-- 给 bean 对象的 map 集合赋值 -->  
    <property name="monsterMap">  
        <map>  
            <entry>  
                <key>  
                    <value>monsterKey01</value>  
                </key>  
                <ref bean="monster01"/>  
            </entry>  
            <entry>  
                <key>
```

```
<key>
    <value>monsterKey02</value>
</key>
<ref bean="monster02"/>
</entry>
</map>
</property>
<!-- 给 bean 对象的 properties 集合赋值 -->
<property name="pros">
    <props>
        <prop key="k1">Java 工程师</prop>
        <prop key="k2">前端工程师</prop>
        <prop key="k3">大数据工程师</prop>
    </props>
</property>
<!-- 给 bean 对象的数组属性注入值 -->
<property name="monsterName">
    <array>
        <value>银角大王</value>
        <value>金角大王</value>
    </array>
</property>
```

```
<!-- 给 bean 对象的 set 属性注入值 -->
<property name="monsterSet">
    <set>
        <ref bean="monster01"/>
        <bean class="com.hspedu.spring.beans.Monster">
            <property name="monsterId" value="10"/>
            <property name="name" value="玉兔精"/>
            <property name="skill" value="钻地洞"/>
        </bean>
    </set>
</property>
</bean>
```

#### 4. 完成测试：修改

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\test\Spring5BeanTest.java, 增加测试方法。

```
/*
 * 测试 引用/注入集合/数组类型
 */
@Test
public void setCollectionByPro() {
    ApplicationContext ioc = new ClassPathXmlApplicationContext("beans.xml");
}
```

```
Master master01 = ioc.getBean("master01", Master.class);
```

```
//获取list 集合
```

```
System.out.println("=====list=====");
```

```
List<Monster> monster_list = master01.getMonsterList();
```

```
for (Monster monster : monster_list) {
```

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

```
}
```

```
//获取map 集合
```

```
System.out.println("=====map=====");
```

```
Map<String, Monster> monster_map = master01.getMonsterMap();
```

```
Set<Map.Entry<String, Monster>> entrySet = monster_map.entrySet();
```

```
for (Map.Entry<String, Monster> entry : entrySet) {
```

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

```
}
```

```
//获取properties 集合
```

```
System.out.println("=====properties=====");
```

```
Properties pros = master01.getPros();
```

```
String property1 = pros.getProperty("k1");
```

```
String property2 = pros.getProperty("k2");
```

```
String property3 = pros.getProperty("k3");
System.out.println(property1 + "\t" + property2 + "\t" + property3);
```

```
//获取数组
System.out.println("=====数组=====");
String[] monsterName = master01.getMonsterName();
for (String s : monsterName) {
    System.out.println("妖怪名= " + s);
}
```

```
//获取set
System.out.println("=====set=====");
Set<Monster> monsterSet = master01.getMonsterSet();
for (Monster monster : monsterSet) {
    System.out.println(monster);
}
```

### 2.2.6.2 使用细节

1. 主要掌握 List/Map/Properties 三种集合的使用.
2. Properties 集合的特点

1) 这个 Properties 是 Hashtable 的子类，是 key-value 的形式

2) key 是 string 而 value 也是 string

## 2.2.7 通过 util 名称空间创建 list

### 2.2.7.1 实例演示

- 案例说明

spring 的 ioc 容器，可以通过 util 名称空间创建 list 集合

- 完成步骤

1. 创建 D:\idea\_java\_projects\spring5\src\com\hspedu\spring\beans\BookStore.java

```
package com.hspedu.spring.beans;
```

```
import java.util.List;
```

```
/**
```

```
 * @author 韩顺平
```

```
 * @version 1.0
```

```
 */
```

```
public class BookStore {//书店
```

```
    private List<String> bookList;
```

```
public BookStore() {  
}  
  
public List<String> getBookList() {  
    return bookList;  
}  
  
public void setBookList(List<String> bookList) {  
    this.bookList = bookList;  
}  
}
```

## 2. 修改 beans.xml , 增加配置

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

```
<!--
```

通过 util 名称空间来创建 list 集合, 可以当做创建 bean 对象的工具来使用

-->

```
<util:list id="myListBook">  
    <value>三国演义</value>  
    <value>西游记</value>  
    <value>红楼梦</value>  
    <value>水浒传</value>  
</util:list>
```

```
<bean id="bookStore" class="com.hspedu.spring.beans.BookStore">  
    <property name="bookList" ref="myListBook"/>  
</bean>
```

### 3. 完成测试：修改

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\test\Spring5BeanTest.java, 增加测试方法.

```
@Test
```

```
public void getListByUtil() {  
    ApplicationContext ioc = new ClassPathXmlApplicationContext("beans.xml");  
    BookStore bookStore = ioc.getBean("bookStore", BookStore.class);  
    List<String> bookList = bookStore.getBookList();  
    for (String s : bookList) {
```

```
System.out.println(s);  
}  
}
```

## 2.2.8 级联属性赋值

### 2.2.8.1 实例演示

- 案例说明

spring 的 ioc 容器，可以直接给对象属性的属性赋值，即级联属性赋值

- 完成步骤

#### 1. 创建 D:\idea\_java\_projects\spring5\src\com\hspedu\spring\beans\Dept.java

```
package com.hspedu.spring.beans;
```

```
/**
```

```
* @author 韩顺平
```

```
* @version 1.0
```

```
*/
```

```
public class Dept {
```

```
    private String name;
```

```
    public Dept() {
```

```
}

public String getName() {

    return name;

}

public void setName(String name) {

    this.name = name;

}

}
```

## 2. 创建 D:\idea\_java\_projects\spring5\src\com\hspedu\spring\beans\Emp.java

```
package com.hspedu.spring.beans;
```

```
/*
 * @author 韩顺平
 * @version 1.0
 */

public class Emp {

    private String name;

    private Dept dept;

    public Emp() {

    }

    public String getName() {
```

```
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public Dept getDept() {  
        return dept;  
    }  
  
    public void setDept(Dept dept) {  
        this.dept = dept;  
    }  
}
```

### 3. 修改 beans.xml，增加配置

```
<!--  
    级联属性赋值  
-->  
  
<bean id="emp" class="com.hspedu.spring.beans.Emp">  
    <property name="name" value="jack"/>  
    <property name="dept" ref="dept"/>  
    <property name="dept.name" value="Java 开发部"/>
```

```
</bean>

<bean id="dept" class="com.hspedu.spring.beans.Dept"/>
```

#### 4. 完成 测试：修改

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\test\Spring5BeanTest.java, 增加测试方法.

```
/*
 * 测试 级联属性赋值
 */

@Test
public void setProByRelation() {
    ApplicationContext ioc = new ClassPathXmlApplicationContext("beans.xml");
    Emp emp = ioc.getBean("emp", Emp.class);
    System.out.println(emp.getDept().getName());
}
```

### 2.2.9 通过静态工厂获取对象

#### 2.2.9.1 实例演示

- 案例说明

在 spring 的 ioc 容器，可以通过静态工厂获取 bean 对象

- 完成步骤

1.

创

建

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\factory\MyStaticFactory.java

```
package com.hspedu.spring.factory;
```

```
import com.hspedu.spring.beans.Monster;
```

```
import java.util.HashMap;
```

```
import java.util.Map;
```

```
/**
```

```
 * @author 韩顺平
```

```
 * @version 1.0
```

```
 */
```

```
public class MyStaticFactory {
```

```
    private static Map<String, Monster> monsterMap;
```

```
    static {
```

```
        monsterMap = new HashMap<String, Monster>();
```

```
    monsterMap.put("monster_01", new Monster(100, "黄袍怪", "一阳指"));
    monsterMap.put("monster_02", new Monster(200, "九头金雕", "如来神掌"));
}
```

```
public static Monster getMonster(String key) {
    return monsterMap.get(key);
}
```

## 2. 修改 beans.xml，增加配置

```
<!-- 通过静态工厂来获取 bean 对象 -->
<bean id="my_monster" class="com.hspedu.spring.factory.MyStaticFactory"
      factory-method="getMonster">
    <!-- constructor-arg 标签提供 key -->
    <constructor-arg value="monster_01"/>
</bean>
```

3. 完成测试：修改  
D:\idea\_java\_projects\spring5\src\com\hspedu\spring\test\Spring5BeanTest.java，增加测试方法。

```
@Test
public void getBeanByStaticFactory() {
```

```
ApplicationContext ioc = new ClassPathXmlApplicationContext("beans.xml");
Monster my_monster = ioc.getBean("my_monster", Monster.class);
System.out.println(my_monster);
}
```

## 2.2.10 通过实例工厂获取对象

### 2.2.10.1 实例演示

- 案例说明

在 spring 的 ioc 容器，可以通过实例工厂获取 bean 对象

- 完成步骤

1.

创

建

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\factory\MyInstanceFactory.java

```
package com.hspedu.spring.factory;
```

```
import com.hspedu.spring.beans.Monster;
```

```
import java.util.HashMap;
```

```
import java.util.Map;
```

```
/*
 * @author 韩顺平
 * @version 1.0
 */

public class MyInstanceFactory {
    private Map<String, Monster> monster_map;

    //非静态代码块
    {
        monster_map = new HashMap<String, Monster>();
        monster_map.put("monster_01", new Monster(100, "猴子精", "吃人"));
        monster_map.put("monster_02", new Monster(200, "九头金雕", "如来神掌"));
    }

    public Monster getMonster(String key) {
        return monster_map.get(key);
    }
}
```

## 2. 配置 beans.xml

```
<!-- 通过实例工厂来获取 bean 对象 -->
```

```
<bean id="myInstanceFactory" class="com.hspedu.spring.factory.MyInstanceFactory"/>

<bean id="my_monster2" factory-bean="myInstanceFactory"

    factory-method="getMonster">

    <constructor-arg value="monster_02"/>

</bean>
```

### 3. 完成 测试：修改

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\test\Spring5BeanTest.java, 增加测试方法。

```
@Test

public void getBeanByInstanceFactory() {

    ApplicationContext ioc = new ClassPathXmlApplicationContext("beans.xml");

    Monster my_monster = ioc.getBean("my_monster2", Monster.class);

    System.out.println(my_monster);

}
```

#### 2.2.11 课后作业练习，把老师前面写的代码自己走一遍

#### 2.2.12 通过 FactoryBean 获取对象(重点)

##### 2.2.12.1 应用实例

- 案例说明

## 在 spring 的 ioc 容器,通过 FactoryBean 获取 bean 对象(重点)

### • 完成步骤

1.

创

建

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\factory\MyFactoryBean.java

```
package com.hspedu.spring.factory;
```

```
import com.hspedu.spring.beans.Monster;
```

```
import org.springframework.beans.factory.FactoryBean;
```

```
import java.util.HashMap;
```

```
import java.util.Map;
```

```
/**
```

```
* @author 韩顺平
```

```
* @version 1.0
```

```
*/
```

```
public class MyFactoryBean implements FactoryBean<Monster> {
```

```
    private String keyVal;
```

```
    private Map<String, Monster> monster_map;
```

```
{
```

```
monster_map = new HashMap<String, Monster>();  
monster_map.put("monster_01", new Monster(100, "黄袍怪", "一阳指"));  
monster_map.put("monster_02", new Monster(200, "九头金雕", "如来神掌"));  
}  
  
public void setKeyVal(String keyVal) {  
    this.keyVal = keyVal;  
}  
  
@Override  
public Monster getObject() throws Exception {  
    // TODO Auto-generated method stub  
    return this.monster_map.get(keyVal);  
}  
  
@Override  
public Class<?> getObjectType() {  
    // TODO Auto-generated method stub  
    return Monster.class;  
}  
  
@Override  
public boolean isSingleton() {  
    // TODO Auto-generated method stub  
    return true;  
}
```

```
}
```

## 2. 配置 beans.xml

```
<!--
```

老韩解读

1. 通过 FactoryBean 来获取 bean 对象
2. name="keyVal" 就是 MyFactoryBean 定义的 setKeyVal 方法
3. value="monster\_01" , 就是给 keyVal 的值

```
-->
```

```
<bean id="myFactoryBean" class="com.hspedu.spring.factory.MyFactoryBean">  
    <property name="keyVal" value="monster_01"/>  
</bean>
```

3. 完成测试：修改  
D:\idea\_java\_projects\spring5\src\com\hspedu\spring\test\Spring5BeanTest.java, 增加测试方法。

```
@Test
```

```
public void getBeanByFactoryBean() {  
    ApplicationContext ioc = new ClassPathXmlApplicationContext("beans.xml");  
    Monster monster = ioc.getBean("myFactoryBean", Monster.class);  
    System.out.println(monster);
```

{}

### 2.2.13 bean 配置信息重用(继承)

#### 2.2.13.1 应用实例

- 说明

在 spring 的 ioc 容器，提供了一种继承的方式来实现 bean 配置信息的重用

- 应用实例演示

##### 1. 配置 beans.xml

```
<!-- 继承的方式来实现 bean 配置信息的重用 -->
<bean id="monster10" class="com.hspedu.spring.beans.Monster">
    <property name="monsterId" value="10"/>
    <property name="name" value="蜈蚣精"/>
    <property name="skill" value="蛰人"/>
</bean>

<!-- parent="monster10" 就是继承使用了 monster10 的配置信息 -->
<bean id="monster11" class="com.hspedu.spring.beans.Monster"
      parent="monster10"/>

<!-- 当我们把某个bean 设置为 abstract="true" 这个bean 只能被继承, 而不能实例化了 -->
```

```
<bean id="monster12" class="com.hspedu.spring.beans.Monster" abstract="true">  
    <property name="monsterId" value="12"/>  
    <property name="name" value="美女蛇"/>  
    <property name="skill" value="吃人"/>  
</bean>
```

*<!-- parent="monster12" 就是继承使用了 monster12 的配置信息 -->*

```
<bean id="monster13" class="com.hspedu.spring.beans.Monster"  
parent="monster12"/>
```

2. 完成测试：修改  
D:\idea\_java\_projects\spring5\src\com\hspedu\spring\test\Spring5BeanTest.java, 增加测试方法.

```
@Test  
public void getBeanByExtends() {  
  
    ApplicationContext ioc = new ClassPathXmlApplicationContext("beans.xml");  
  
    Monster monster1 = ioc.getBean("monster11", Monster.class);  
  
    System.out.println(monster1);  
  
    Monster monster2 = (Monster) ioc.getBean("monster13", Monster.class);  
  
    System.out.println(monster2);
```

}

## 2.2.14 bean 创建顺序

### 2.2.14.1 实例演示

- 说明

1. 在 spring 的 ioc 容器，默认是按照配置的顺序创建 bean 对象

```
<bean id="student01" class="com.hspedu.bean.Student" />
```

```
<bean id="department01" class="com.hspedu.bean.Department" />
```

会先创建 student01 这个 bean 对象，然后创建 department01 这个 bean 对象

2. 如果这样配置

```
<bean id="student01" class="com.hspedu.bean.Student" depends-on="department01"/>
```

```
<bean id="department01" class="com.hspedu.bean.Department" />
```

会先创建 department01 对象，再创建 student01 对象。

### 3. 小伙伴可以通过输出信息来验证.

#### 2.2.14.2 一个问题?

- 问题说明

1. 先看下面的配置, 请问两个 bean 创建的顺序是什么? 并分析执行流程

1) 先创建 id=memberDAOImpl

2) 再创建 id = memberServiceImpl

3) 调用 memberServiceImpl.setMemberDAO() 完成引用

```
<!-- 配置MemeberDao 对象-->
<bean class="com.hspedu.spring.dao.MemberDAOImpl" id="memberDAOImpl"/>

<bean class="com.hspedu.spring.service.MemberServiceImpl" id="memberServiceImpl">
    <property name="memberDAO" ref="memberDAOImpl"/>
</bean>
```

2. 先看下面的配置, 请问两个 bean 创建的顺序是什么, 并分析执行流程

1) 先创建 id = memberServiceImpl

2) 再创建 id=memberDAOImpl

### 3) 用 memberServiceImpl.setMemberDAO() 完成引用

```
<bean class="com.hspedu.spring.service.MemberServiceImpl"
      id="memberServiceImpl">
    <property name="memberDAO" ref="memberDAOImpl"/>
</bean>

<!-- 配置MemberDao 对象-->
<bean class="com.hspedu.spring.dao.MemberDAOImpl" id="memberDAOImpl"/>
```

## 2.2.15 bean 对象的单例和多例

### 2.2.15.1 应用实例

- 说明

在 spring 的 ioc 容器，在默认是按照单例创建的，即配置一个 **bean** 对象后，ioc 容器只会创建一个 **bean** 实例。

如果，我们希望 ioc 容器配置的某个 **bean** 对象，是以多个实例形式创建的则可以通过配置 **scope="prototype"** 来指定

- 应用实例演示

1. 创建 D:\idea\_java\_projects\spring5\src\com\hspedu\spring\beans\Car.java

```
package com.hspedu.spring.beans;
```

```
/**  
 * @author 韩顺平  
 * @version 1.0  
 */  
  
public class Car {  
  
    public Car() {  
        System.out.println("car 构造器");  
    }  
}
```

## 2. 配置 beans.xml

```
<!--
```

老韩解读

1. 在 **spring** 的 **ioc** 容器，在默认情况下是安装单例创建的，即配置一个 **bean** 对象后，  
**ioc** 容器只会创建一个 **bean** 实例。

2. 如果，我们希望 **ioc** 容器配置的某个 **bean** 对象，  
是以多个实例形式创建的则可以通过配置 **scope="prototype"** 来指定

-->

```
<bean name="car" scope="prototype" class="com.hspedu.spring.beans.Car"/>
```

### 3. 完成测试：修改

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\test\Spring5BeanTest.java, 增加测试方法.

```
@Test
```

```
public void getBeanByPrototype() {  
    ApplicationContext ioc = new ClassPathXmlApplicationContext("beans.xml");  
    for (int i = 0; i < 3; i++) {  
        Car car = ioc.getBean("car", Car.class);  
        System.out.println(car);  
    }  
}
```

#### 2.2.15.2 使用细节

1. 默认是单例 singleton, 在启动容器时, 默认就会创建, 并放入到 singletonObjects 集合
2. 当 `<bean scope="prototype" >` 设置为多实例机制后, 该 bean 是在 `getBean()` 时才创建
3. 如果是单例 singleton, 同时希望在 `getBean` 时才创建, 可以 指定懒加载 `lazy-init="true"` (注意默认是 false)

4. 通常情况下, lazy-init 就使用默认值 false , 在开发看来, 用空间换时间是值得的, 除非有特殊的要求.
5. 如果 scope="prototype" 这时你的 lazy-init 属性的值不管是 true, 还是 false 都是在 getBean 时候, 才创建对象.

## 2.2.16 bean 的生命周期

### 2.2.16.1 应用实例

- 说明: bean 对象创建是由 JVM 完成的, 然后执行如下方法
  1. 执行构造器
  2. 执行 set 相关方法
  3. 调用 bean 的初始化的方法 (需要配置)
  4. 使用 bean
  5. 当容器关闭时候, 调用 bean 的销毁方法 (需要配置)
- 应用实例演示
  1. 创建 D:\idea\_java\_projects\spring5\src\com\hspedu\spring\beans\House.java

```
package com.hspedu.spring.beans;
```

```
/*
 * @author 韩顺平
 * @version 1.0
 */

public class House {

    private String name;

    public House() {
        System.out.println("House() 构造器");
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        System.out.println("House setName(..)");
        this.name = name;
    }

    public void init() {
        System.out.println("House init(..)");
    }
}
```

```
public void destory() {  
    System.out.println("House destory()..");  
}  
}
```

## 2. 配置 beans.xml

```
<!-- 配置 bean 的初始化方法和销毁方法 -->  
<bean id="house" class="com.hspedu.spring.beans.House"  
      init-method="init"  
      destroy-method="destory">  
    <property name="name" value="北京豪宅"/>  
</bean>
```

## 3. 完成测试：修改

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\test\Spring5BeanTest.java, 增加测试方法。

```
@Test
```

```
public void beanLife() {  
    ApplicationContext ioc = new ClassPathXmlApplicationContext("beans.xml");  
    House house = ioc.getBean("house", House.class);  
    System.out.println(house);  
    //关闭容器
```

```
((ConfigurableApplicationContext) ioc).close();
```

{}

#### 2.2.16.2 使用细节

1. 初始化 `init` 方法和 `destroy` 方法，是程序员来指定
2. 销毁方法就是当关闭容器时，才会被调用。

#### 2.2.17 配置 bean 的后置处理器 【这个比较难】

##### 2.2.17.1 应用实例

- 说明
  1. 在 spring 的 ioc 容器,可以配置 bean 的后置处理器
  2. 该处理器/对象会在 **bean 初始化方法**调用前和初始化方法调用后被调用
  3. 程序员可以在后置处理器中编写自己的代码
- 应用实例演示

##### 1. 创建 House.java 前面有

##### 2. 创建后置处理器

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\beans\MyBeanPostProcessor.java

```
package com.hspedu.spring.beans;

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

/**
 * @author 韩顺平
 * @version 1.0
 */

public class MyBeanPostProcessor implements BeanPostProcessor {

    /**
     * 在 bean 初始化之前完成某些任务
     * @param bean      : 就是 ioc 容器返回的 bean 对象, 如果这里被替换会修改, 则返回的 bean 对象也会被修改
     * @param beanName: 就是 ioc 容器配置的 bean 的名称
     * @return Object: 就是返回的 bean 对象
     */

    public Object postProcessBeforeInitialization(Object bean, String beanName)
            throws BeansException {
        // TODO Auto-generated method stub
        System.out.println("postProcessBeforeInitialization 被调用 " + beanName + " "
                + bean= + bean.getClass());
    }
}
```

```
        return bean;  
    }  
  
    /**  
     * 在 bean 初始化之后完成某些任务  
     * @param bean      : 就是 ioc 容器返回的 bean 对象, 如果这里被替换会修改, 则返  
     * 回的 bean 对象也会被修改  
     * @param beanName: 就是 ioc 容器配置的 bean 的名称  
     * @return Object: 就是返回的 bean 对象  
     */  
  
    public Object postProcessAfterInitialization(Object bean, String beanName)  
        throws BeansException {  
        System.out.println("postProcessAfterInitialization 被调用 " + beanName + " bean=" +  
            + bean.getClass());  
        return bean;  
    }  
}
```

### 3. 老师新创建 src\beans02.xml，方便讲解

```
<?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">](http://www.springframework.org/schema/beans/spring-beans.xsd)

```
<!-- 配置 bean 的初始化方法和销毁方法 -->  
<bean id="house" class="com.hspedu.spring.beans.House"  
    init-method="init"  
    destroy-method="destory">  
    <property name="name" value="北京豪宅"/>  
</bean>
```

```
<!-- bean 后置处理器的配置 -->  
<bean  
    class="com.hspedu.spring.beans.MyBeanPostProcessor" />  
</beans>
```

4. 完成测试：修改  
D:\idea\_java\_projects\spring5\src\com\hspedu\spring\test\Spring5BeanTest.java, 增加测试方法。

@Test

```
public void testBeanPostProcessor() {  
    ApplicationContext ioc = new ClassPathXmlApplicationContext("beans02.xml");  
    House house = ioc.getBean("house", House.class);
```

```
System.out.println(house);
//关闭容器
((ConfigurableApplicationContext) ioc).close();
}
```

#### 2.2.17.2 其它说明

- 1、怎么执行到这个方法?=> 使用 AOP(反射+动态代理+IO+容器+注解)
  - 2、有什么用? => 可以对 IOC 容器中所有的对象进行统一处理 ,比如 日志处理/权限的校验/安全的验证/事务管理.
- 初步体验案例: 如果类型是 House 的统一改成 上海豪宅
- 3、针对容器的所有对象吗? 是的=>切面编程特点
  - 4、后面我们会自己实现这个底层机制, 这个是一个比较难理解的知识点, 现在老韩不做过多的纠结, 后面我会带小伙伴实现这个机制

#### 2.2.18 通过属性文件给 bean 注入值

##### 2.2.18.1 应用实例

- 说明

在 spring 的 ioc 容器,通过属性文件给 bean 注入值

- 应用实例演示

## 1. src/ 创建 my.properties

```
name=\u9EC4\u888D\u602A
```

```
id=10
```

```
skill=\u72EE\u5B50\u543C
```

## 2. 修改 src\beans.xml，继续完成配置

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

```
<!--
```

老韩解读：

1. 通过属性文件给 bean 注入值,
2. 需要导入: xmlns:context 名字空间, 并指定属性文件路径

-->

```
<context:property-placeholder location="classpath:my.properties"/>  
<bean id="monster100" class="com.hspedu.spring.beans.Monster">  
    <property name="monsterId" value="${id}" />  
    <property name="name" value="${name}" />  
    <property name="skill" value="${skill}" />  
</bean>
```

### 3. 完成 测试 : 修 改

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\test\Spring5BeanTest.java, 增加测试方法.

```
/**  
 * 通过属性文件给 bean 注入值  
 */  
  
@Test  
  
public void setProByProFile() {  
    ApplicationContext ioc = new ClassPathXmlApplicationContext("beans.xml");  
    Monster monster100 = ioc.getBean("monster100", Monster.class);  
    System.out.println(monster100);  
}
```

## 2.2.19 基于 XML 的 bean 的自动装配

### 2.2.19.1 应用实例

- 说明

在 spring 的 ioc 容器,可以实现自动装配 bean

- 应用实例演示

这里说的 Action 就是我们前面讲过的 Servlet->充当 Controller

1. 创建 D:\idea\_java\_projects\spring5\src\com\hspedu\spring\dao\OrderDao.java

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\service\OrderService.java

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\action\OrderAction.java

```
package com.hspedu.spring.dao;
```

```
/**
```

```
* @author 韩顺平
```

```
* @version 1.0
```

```
*/
```

```
public class OrderDao {
```

```
    public void saveOrder() {
```

```
System.out.println("保存 一个订单...");  
}  
}  
  
package com.hspedu.spring.service;  
  
import com.hspedu.spring.dao.OrderDao;  
  
/**  
 * @author 韩顺平  
 * @version 1.0  
 */  
public class OrderService {  
    private OrderDao orderDao;  
    public OrderDao getOrderDao() {  
        return orderDao;  
    }  
    public void setOrderDao(OrderDao orderDao) {  
        this.orderDao = orderDao;  
    }  
}  
package com.hspedu.spring.action;
```

```
import com.hspedu.spring.service.OrderService;

/*
 * @author 韩顺平
 * @version 1.0
 */

public class OrderAction {

    private OrderService orderService;

    public OrderService getOrderService() {
        return orderService;
    }

    public void setOrderService(OrderService orderService) {
        this.orderService = orderService;
    }
}
```

## 2. 配置 beans.xml

```
<!-- 基于xml的bean的自动装配演示
      autowire="byType" 表示根据类型进行自动组装-->
```

-->

```
<!--          <bean      id="orderAction"      autowire="byType"  
class="com.hspedu.spring.action.OrderAction" />-->  
  
<!--          <bean      id="orderService"      autowire="byType"  
class="com.hspedu.spring.service.OrderService"/>-->  
  
<!--    <bean id="orderDao" class="com.hspedu.spring.dao.OrderDao"/>-->  
  
<!--
```

### 基于 xml 的 bean 的自动装配 演示

- 特别说明: `autowire = "byName"` 会自动去找 `id` 为 `setXXXX` 后面 `XXXX` 的 bean 自动组装.  
,如果找到就装配, 如果找不到就报错, 比如这里的  
`OrderService`
- `<bean id="orderAction" autowire="byName" class="com.hspedu.bean.OrderAction" />`  
就会去找 `OrderAction` 类中定义的 `setOrderService` 的 `id` 为 `orderService` 的  
`OrderService`  
`bean` 组装, 找到就阻装, 找不到就组装失败

-->

```
<bean      id="orderAction"      autowire="byName"  
class="com.hspedu.spring.action.OrderAction"/>  
  
<bean      id="orderService"      autowire="byName"  
class="com.hspedu.spring.service.OrderService"/>  
  
<bean id="orderDao" class="com.hspedu.spring.dao.OrderDao"/>
```

### 3. 完成 测试：修改

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\test\Spring5BeanTest.java, 增加测试方法。

```
@Test
```

```
public void setProByAutowire() {  
    ApplicationContext ioc = new ClassPathXmlApplicationContext("beans.xml");  
    OrderAction orderAction = ioc.getBean("orderAction", OrderAction.class);  
    orderAction.getOrderService().getOrderDao().saveOrder();  
}
```

#### 2.2.19.2 其它说明

1. 这个知识点作为了解即可，后面我们主要还是使用基于注解的方式(重点.)

2. 但是机制和原理类似

#### 2.2.20 spring el 表达式[知道即可，老师给大家演示一把]

##### 2.2.20.1 使用实例

- 说明

1. Spring Expression Language, Spring 表达式语言，简称 SpEL。支持运行时查询并可以操作对象。

2. 和 EL 表达式一样， SpEL 根据 JavaBean 风格的 `getXxx()`、`setXxx()`方法定义的属性访问对象
3. SpEL 使用`#{...}`作为定界符，所有在大括号中的字符都将被认为是 SpEL 表达式。
4. 不是重点，如果看到有人这样使用，能看懂即可

- 应用实例演示

1. 创建 `D:\idea_java_projects\spring5\src\com\hspedu\spring\beans\SpELBean.java`

```
package com.hspedu.spring.beans;
```

```
/**
```

```
* @author 韩顺平
```

```
* @version 1.0
```

```
*/
```

```
public class SpELBean {
```

```
    private String name;
```

```
    private Monster monster;
```

```
    private String monsterName;
```

```
    private String crySound;
```

```
    private String bookName;
```

```
    private Double result;
```

```
public SpELBean() {  
}  
  
public String getName() {  
    return name;  
}  
  
public void setName(String name) {  
    this.name = name;  
}  
  
public Monster getMonster() {  
    return monster;  
}  
  
public void setMonster(Monster monster) {  
    this.monster = monster;  
}  
  
public String getMonsterName() {  
    return monsterName;
```

```
}
```

```
public void setMonsterName(String monsterName) {  
    this.monsterName = monsterName;  
}
```

```
public String getCrySound() {  
    return crySound;  
}
```

```
public void setCrySound(String crySound) {  
    this.crySound = crySound;  
}
```

```
public String getBookName() {  
    return bookName;  
}
```

```
public void setBookName(String bookName) {  
    this.bookName = bookName;  
}
```

```
public Double getResult() {  
    return result;  
}  
  
public void setResult(Double result) {  
    this.result = result;  
}  
  
public String cry(String sound) {  
    return "发出 " + sound + "叫声...";  
}  
  
public static String read(String bookName) {  
    return "正在看 " + bookName;  
}
```

## 2. 配置 beans.xml

```
<!-- spring el 表达式 -->  
<bean id="spELBean" class="com.hspedu.spring.beans.SpELBean">  
    <!-- sp el 给变量 -->  
    <property name="name" value="#{'韩顺平教育'}"/>
```

```
<!-- sp el 引用其它 bean -->
<property name="monster" value="#{monster01}"/>

<!-- sp el 引用其它 bean 的属性值 -->
<property name="monsterName" value="#{monster02.name}"/>

<!-- sp el 调用普通方法 赋值 -->
<property name="crySound" value="#{spELBean.cry('喵喵的..')}"/>

<!-- sp el 调用静态方法 赋值 -->
<property name="bookName" value="#{T(com.hspedu.spring.beans.SpELBean).read('
天龙八部')}">

<!-- sp el 通过运算赋值 -->
<property name="result" value="#{89*1.2}"/>

</bean>
```

### 3. 完成 测 试 : 修 改

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\test\Spring5BeanTest.java, 增加测试方法.

```
@Test
```

```
public void setProBySpel() {
    ApplicationContext ioc = new ClassPathXmlApplicationContext("beans.xml");
    SpELBean spELBean = ioc.getBean("spELBean", SpELBean.class);
    System.out.println(spELBean.getName());
```

```
System.out.println(spELBean.getMonster());  
System.out.println(spELBean.getMonsterName());  
System.out.println(spELBean.getcrySound());  
System.out.println(spELBean.getBookName());  
System.out.println(spELBean.getResult());  
  
}
```

## 2. 2. 21 课后作业练习，把老师前面写的代码自己走一遍

### 2. 3 基于注解配置 bean

#### 2. 3. 18 基本使用

##### 2.3.18.1 说明

- 基本介绍

基于注解的方式配置 **bean**, 主要是项目开发中的组件, 比如 **Controller**、**Service**、和 **Dao**.

- 组件注解的形式有

1. **@Component** 表示当前注解标识的是一个组件
2. **@Controller** 表示当前注解标识的是一个控制器, 通常用于 **Servlet**
3. **@Service** 表示当前注解标识的是一个处理业务逻辑的类, 通常用于 **Service** 类

4. **@Repository** 表示当前注解标识的是一个持久化层的类，通常用于 Dao 类

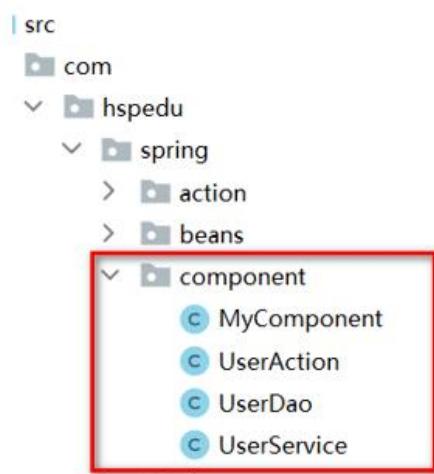
### 2.3.18.2 快速入门

- 应用实例

#### 使用注解的方式来配置 Controller / Service / Repository / Component

- 代码实现

1. 引入 spring-aop-5.3.8.jar，在 spring/libs 下拷贝即可
2. 创建 UserAction.java UserService.java, UserDao.java MyComponent.java



```
package com.hspedu.spring.component;
import org.springframework.stereotype.Repository;
```

```
/**
```

```
* @author 韩顺平
```

```
* @version 1.0
```

```
*/
```

```
@Repository
```

```
public class UserDao {
```

```
}
```

```
package com.hspedu.spring.component;
```

```
import org.springframework.stereotype.Service;
```

```
/**
```

```
* @author 韩顺平
```

```
* @version 1.0
```

```
*/
```

```
@Service
```

```
public class UserService {
```

```
}
```

```
package com.hspedu.spring.component;
```

```
import org.springframework.stereotype.Controller;
```

```
/**  
 * @author 韩顺平  
 * @version 1.0  
 */  
  
@Controller  
  
public class UserAction {  
}  
  
package com.hspedu.spring.component;  
  
import org.springframework.stereotype.Component;  
  
/**  
 * @author 韩顺平  
 * @version 1.0  
 */  
  
@Component  
  
public class MyComponent {  
}
```

### 3. 配置 beans.xml

```
<!-- 配置自动扫描的包，注意需要加入 context 名称空间 -->  
<context:component-scan base-package="com.hspedu.spring.component" />
```

4. 完成测试，可以通过 Debug 来看（非常清楚.），修改 D:\idea\_java\_projects\spring5\src\com\hspedu\spring\test\Spring5BeanTest.java，增加测试方法

```
@Test  
public void getBeanByAnnotation() {  
    ApplicationContext ioc = new ClassPathXmlApplicationContext("beans.xml");  
    UserAction userAction = ioc.getBean(UserAction.class);  
    System.out.println(userAction);  
    UserDao userDao = ioc.getBean(UserDao.class);  
    System.out.println(userDao);  
    MyComponent myComponent = ioc.getBean(MyComponent.class);  
    System.out.println(myComponent);  
    UserService userService = ioc.getBean(UserService.class);  
    System.out.println(userService);  
}
```

### 2.3.18.3 注意事项和细节说明

1. 需要导入 spring-aop-5.3.8.jar，别忘了
2. 必须在 Spring 配置文件中指定“自动扫描的包”，IOC 容器才能够检测到当前项目中哪

些类被标识了注解， 注意到导入 context 名称空间

<!-- 配置自动扫描的包 -->

<context:component-scan base-package="com.hspedu.spring.component" />

可以使用通配符 \* 来指定， 比如 com.hspedu.spring.\* 表示

--老韩提问: com.hspedu.spring.component 会不会去扫描它的子包?

答：会的

--测试完毕，恢复原来的状态

3. Spring 的 IOC 容器不能检测一个使用了@Controller 注解的类到底是不是一个真正的控制器。注解的名称是用于程序员自己识别当前标识的是什么组件。其它的@Service @Repository 也是一样的道理 [也就是说 spring 的 IOC 容器只要检查到注解就会生成对象，但是这个注解的含义 spring 不会识别，注解是给程序员编程方便看的]

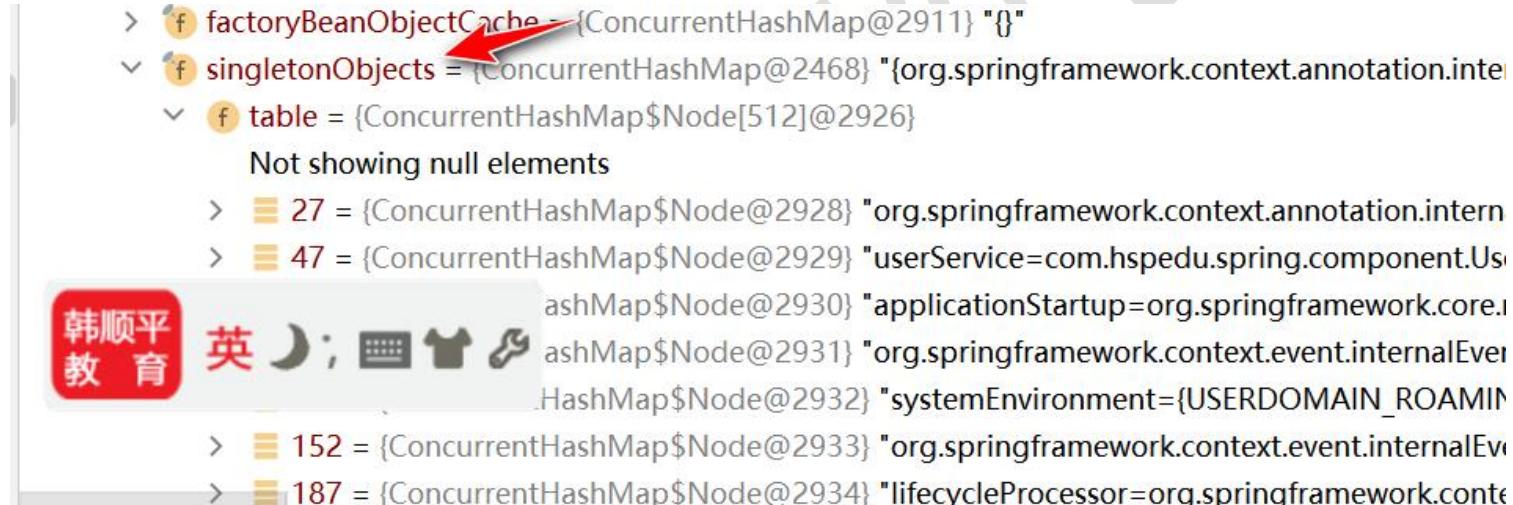
4. <context:component-scan base-package="com.hspedu.spring.component" resource-pattern="User\*.class" />resource-pattern="User\*.class": 表示只扫描满足要求的类.[使用的少，不想扫描，不写注解就可以，知道这个知识点即可]

5. <context:component-scan base-package="com.hspedu.spring.component" >

<!-- 排除哪些类，以 annotation 注解为例 -->

```
<context:exclude-filter type="annotation"  
expression="org.springframework.stereotype.Service"/>  
  
</context>
```

--通过 Debug IOC 容器结构，可以一目了然.



```
<!-- package="com.hspedu.spring.component" /-->
<!-- component-scan -->
<!-- component-scan -->
<!-- component-scan -->
```

Tag name: **expression**

Description : Indicates the filter expression, the type of which is indicated by "type".

 commons-logging-1.1.1

鼠标放在expression  
即可看到说明

## 老韩解读

- 1) <context:exclude-filter> 放在<context:component-scan>内，表示扫描过滤掉当前包的某些类
- 2) type="annotation" 按照注解类型进行过滤.
- 3) expression :就是注解的全类名，比如 org.springframework.stereotype.Service 就是@Service 注解的全类名,其它比@Controller @Repository 等 依次类推
- 4) 上面表示过滤掉 com.hspedu.spring.component 包下，加入了@Service 注解的类
- 5) 完成测试，修改 beans.xml，增加 exclude-filter，发现 UserService，不会注入到容器.

<!--

## 老韩解读

1. <context:exclude-filter> 放在<context:component-scan>内，表示扫描过滤掉当前包的某些类
2. type="annotation" 按照注解方式进行过滤.
3. expression :就是注解的全类名，比如 org.springframework.stereotype.Service 就是@Service 注解的全类名,其它比@Controller @Repository 等 依次类推
4. 上面表示过滤掉 com.hspedu.spring.component 包下，加入了@Service 注解的类

-->

```
<context:component-scan base-package="com.hspedu.spring.component">  
<context:exclude-filter type="annotation">
```

```
expression="org.springframework.stereotype.Service"/>  
</context:component-scan>
```

## 6. 指定自动扫描哪些注解类

### 1) 修改 beans.xml 完成演示

```
<!--
```

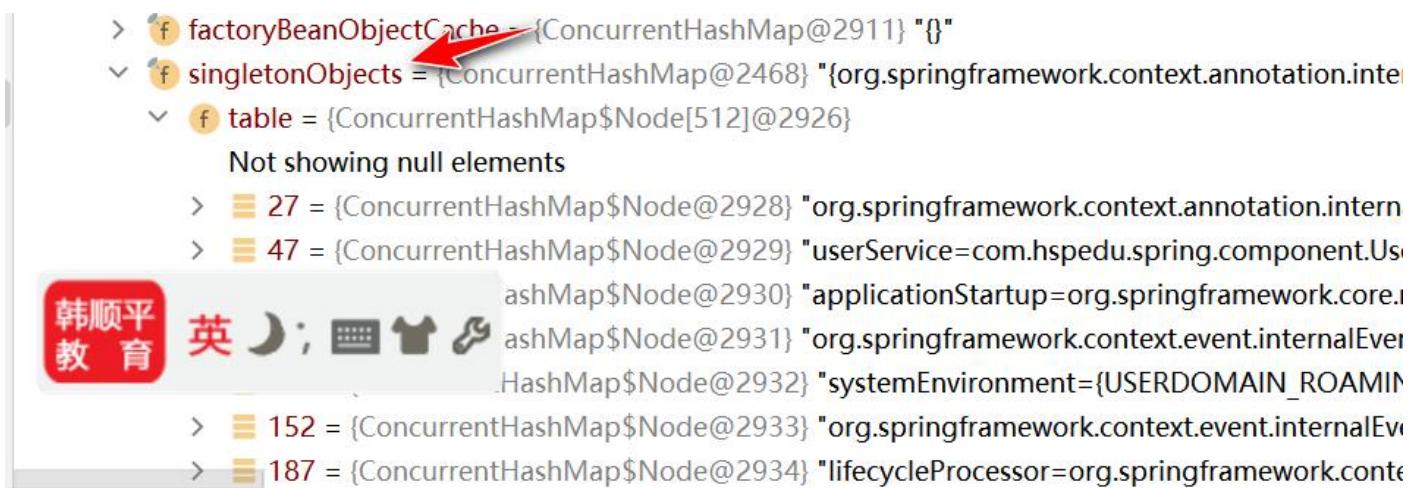
老韩解读

1. use-default-filters="false": 不再使用默认的过滤机制
2. context:include-filter: 表示只是扫描指定的注解的类
3. expression="org.springframework.stereotype.Controller": 注解的全类名

```
-->
```

```
<context:component-scan  
use-default-filters="false">  
    <context:include-filter  
        type="annotation"  
        expression="org.springframework.stereotype.Service"/>  
    <context:include-filter  
        type="annotation"  
        expression="org.springframework.stereotype.Controller"/>  
</context:component-scan>
```

---通过 Debug IOC 容器 可以一目了然



```
> f factoryBeanObjectCache = {ConcurrentHashMap@2911} "0"
< f singletonObjects = {ConcurrentHashMap@2468} "{org.springframework.context.annotation.inte
  < f table = {ConcurrentHashMap$Node[512]@2926}
    Not showing null elements
  > 27 = {ConcurrentHashMap$Node@2928} "org.springframework.context.annotation.intern
  > 47 = {ConcurrentHashMap$Node@2929} "userService=com.hspedu.spring.component.UserService
    ashMap$Node@2930} "applicationStartup=org.springframework.core.i
    ashMap$Node@2931} "org.springframework.context.event.internalEven
    .HashMap$Node@2932} "systemEnvironment={USERDOMAIN_ROAMIN
  > 152 = {ConcurrentHashMap$Node@2933} "org.springframework.context.event.internalEven
  > 187 = {ConcurrentHashMap$Node@2934} "lifecycleProcessor=org.springframework.context.event.interna
```

## 2) 完成测试，修改

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\test\Spring5BeanTest.java, 增加测试方法

@Test

```
public void testBeanByIncludefilter() {
    ApplicationContext ioc = new ClassPathXmlApplicationContext("beans.xml");
    System.out.println("-----");
    UserAction userAction = ioc.getBean(UserAction.class);
    System.out.println(userAction);
    UserService userService = ioc.getBean(UserService.class);
    System.out.println(userService);
    //下面这个就会报错
    UserDao userDao = ioc.getBean(UserDao.class);
    System.out.println(userDao);
```

}

7. 默认情况：标记注解后，类名首字母小写作为 id 的值。也可以使用注解的 value 属性指定 id 值，并且 value 可以省略。[代码演示]

```
@Controller(value="userAction01")
```

```
@Controller("userAction01")
```

1) 修改 UserAction.java

```
/*
 * @author 韩顺平
 * @version 1.0
 */
// @Controller
// @Controller(value = "userAction01")
@Controller("userAction01")
public class UserAction {
```

2) 完成测试，修改  
D:\idea\_java\_projects\spring5\src\com\hspedu\spring\test\Spring5BeanTest.java, 增加测试方法

```
@Test
```

```
public void getBeanByAnnotationId() {
```

```
ApplicationContext ioc = new ClassPathXmlApplicationContext("beans.xml");

//默认 id 获取

// UserAction userAction = ioc.getBean("userAction",UserAction.class);

// System.out.println(userAction);

//指定 id 获取

UserAction userAction01 = ioc.getBean("userAction01", UserAction.class);

System.out.println(userAction01);

}
```

## 8. 扩展-@Controller 、 @Service、 @Component 区别 : (回去看看一下老师的讲解的注解基础) <https://zhuanlan.zhihu.com/p/454638478>

### 2.3.19 手动开发-【老韩新增】 简单的 Spring 基于注解配置的程序 -老韩要求：小伙伴要至少独立写 2 遍

#### 2.3.19.1 需求说明

1. 自己写一个简单的 Spring 容器，通过读取类的注解 (@Component @Controller @Service @Repository)，将对象注入到 IOC 容器

=====

bean id= MyComponent bean 对象= com.hspedu.spring.component.MyComponent@13221655

bean id= UserDao bean 对象= com.hspedu.spring.component.UserDao@2f2c9b19

bean id= UserService bean 对象= com.hspedu.spring.component.UserService@31befd9f

```
bean id= UserController bean 对象= com.hspedu.spring.component.UserController@1c20c684
```

```
i am user dao
```

## 2. 也就是说，不使用 Spring 原生框架，我们自己使用 IO+Annotation+反射+集合 技术实现，打通 Spring 注解方式开发的技术痛点

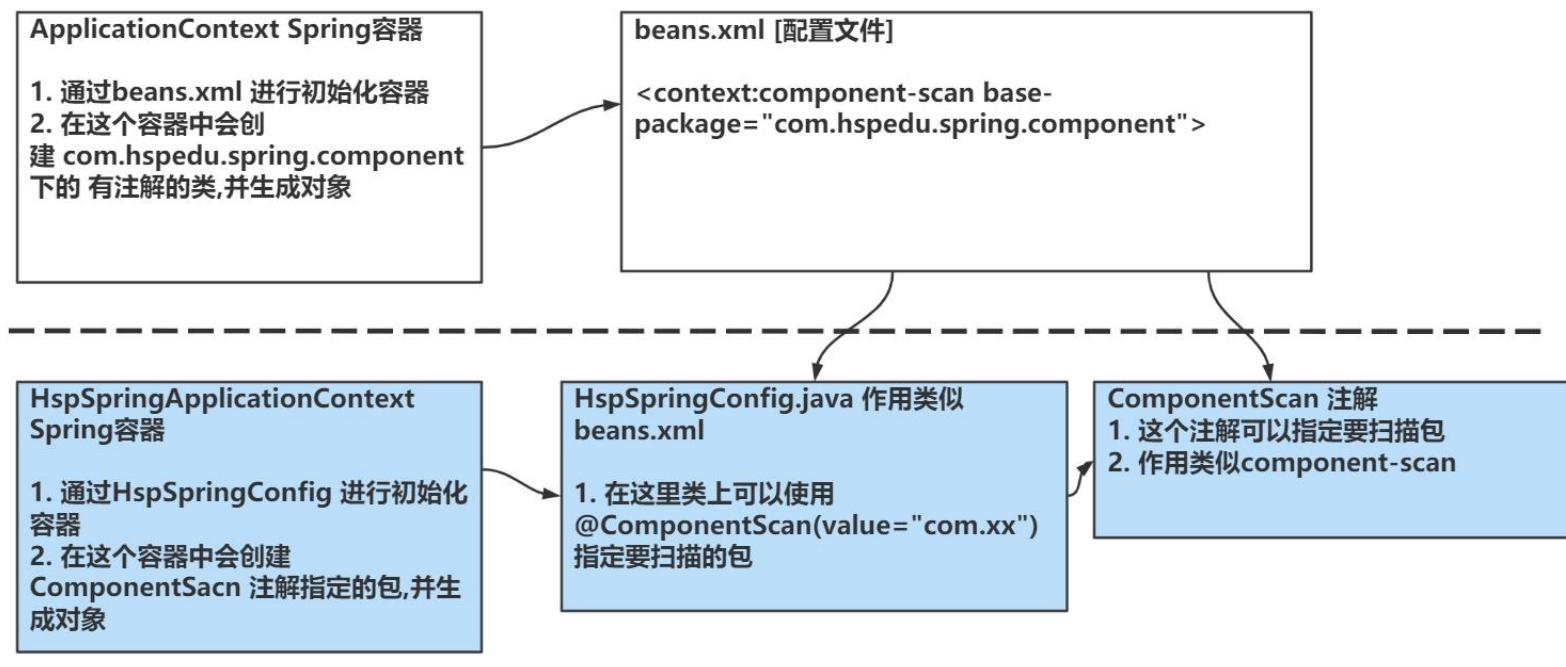


### 2.3.19.2 思路分析

#### 1. 思路分析+程序结构

1) 我们使用注解方式完成，这里老韩不用 xml 来配置

2) 程序框架图



### 2.3.19.3 代码实现

- 应用实例

- 手动实现注解的方式来配置 Controller / Service / Repository / Component

- 我们使用自定义注解来完成。

- 代码实现

- 仍然使用前面的 \com\hspedu\spring\component\ 包下的类

2. 创建

D:\hspedu\_ssm\_temp\spring5\src\com\hspedu\spring\annotation\ComponentScan.java

```
package com.hspedu.spring.annotation;
```

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

```
/*
 * @author 韩顺平
 * @version 1.0
 * 定义我们的 ComponentScan 注解
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface ComponentScan {
```

```
    String value();
```

```
}
```

3.

创

建

D:\hspedu\_ssm\_temp\spring5\src\com\hspedu\spring\annotation\HspSpringConfig.java

```
package com.hspedu.spring.annotation;
```

```
/*
```

```
* @author 韩顺平  
* @version 1.0  
* 作用类似我们的beans.xml 文件，用于对spring 容器指定配置信息  
*/
```

//指定要扫描的包

```
@ComponentScan("com.hspedu.spring.component")  
public class HspSpringConfig {  
}
```

4.

创

建

D:\hspedu\_ssm\_temp\spring5\src\com\hspedu\spring\annotation\HspSpringApplicationCo  
nfig.java

```
package com.hspedu.spring.annotation;  
  
import org.springframework.stereotype.Component;  
import org.springframework.stereotype.Controller;  
import org.springframework.stereotype.Repository;  
import org.springframework.stereotype.Service;
```

```
import java.io.File;  
import java.lang.annotation.Annotation;  
import java.net.URL;
```

```
import java.util.concurrent.ConcurrentHashMap;
```

```
/**  
 * @author 韩顺平  
 * @version 1.0  
 * 充当容器类,类似 Spring 原生的 ApplicationContext  
 */  
  
public class HspSpringApplicationContext {  
    private Class configClass;  
    private final ConcurrentHashMap<String, Object> ioc = new  
ConcurrentHashMap<>();  
  
    public ConcurrentHashMap<String, Object> getioc() {  
        return ioc;  
    }  
  
    public HspSpringApplicationContext(Class configClass) {  
        this.configClass = configClass;  
  
        //1. 解析配置类  
    }  
}
```

```
//2. 获取到配置类的 @ComponentScan("com.hspedu.spring.component")  
ComponentScan componentScan = (ComponentScan)  
this.configClass.getDeclaredAnnotation(ComponentScan.class);  
  
String path = componentScan.value();  
System.out.println("扫描路径 = " + path);  
  
//3. 获取扫描路径下所有的类文件  
//(1) 先得到类加载器，使用 App 方式来加载。  
ClassLoader classLoader =  
HspSpringApplicationContext.class.getClassLoader();  
  
//在获取某个包的对应的 URL 时，要求是 com/hspedu/spring/component  
//URL resource = classLoader.getResource("com/hspedu/spring/component");  
  
//(2) 将 path 转成 形式为 com/hspedu/spring/component  
path = path.replace(".", "/");  
  
//(3) 这里是获取到我们要加载的资源(类)的路径(工作路径：  
file:/D:/hspedu_ssm_temp/spring5/out/production/spring5/com/hspedu/spring/component)  
  
//     ，目的是为了扫描该路径下的类  
URL resource = classLoader.getResource(path);  
File file = new File(resource.getFile());  
if (file.isDirectory()) {
```

```
File[] files = file.listFiles();

for (File f : files) {

    String fileAbsolutePath = f.getAbsolutePath();

    System.out.println("=====");
    System.out.println("文件绝对路径 = " + fileAbsolutePath);

    if (fileAbsolutePath.endsWith(".class")) { //说明是类文件

        //通过类加载器获取来类文件的Clazz 对象
        // 先得到类的完整类路径 形式为
        com.hspedu.spring.component.MonsterService

        String className =
            fileAbsolutePath.substring(fileAbsolutePath.lastIndexOf("\\") + 1,
            fileAbsolutePath.indexOf(".class"));

        String classFullPath = path.replace("/", ".") + "." + className;
        System.out.println("类名 = " + className);
        System.out.println("类的全路径 = " + classFullPath);

        try {
            //获取到扫描包下的类的 clazz 对象
            Class<?> clazz = classLoader.loadClass(classFullPath);

            if (clazz.isAnnotationPresent(Component.class)) ||
```

```
clazz.isAnnotationPresent(Controller.class) ||  
clazz.isAnnotationPresent(Service.class) ||  
clazz.isAnnotationPresent(Repository.class)) {
```

//老师这里只是测试一个，对Component，取出指定的值  
// if(clazz.isAnnotationPresent(Component.class)) {  
// Component annotation =  
clazz.getAnnotation(Component.class);

// if(!"".equals(annotation.value())) {  
// className = annotation.value();  
// }

// }  
// }

//如果这个类有@Component，说明是一个 spring

bean

System.out.println("是一个 bean = " + clazz);

//将其反射生成到ioc 中。

Class<?> aClass = Class.forName(classFullPath);

try {

Object instance = aClass.newInstance();

ioc.put(className, instance); //这里可以将类名的

首字母小写



```
* 返回ioc注入的指定bean  
* @param name  
* @return  
*/  
  
public Object getBean(String name) {  
    return ioc.get(name);  
}  
}
```

5.

创

建

D:\hspedu\_ssm\_temp\spring5\src\com\hspedu\spring\annotation\HspSpringAnnotationMa  
in.java

```
package com.hspedu.spring.annotation;  
  
import com.hspedu.spring.component.UserDao;
```

```
import java.util.Enumeration;  
import java.util.concurrent.ConcurrentHashMap;
```

```
/**  
 * @author 韩顺平  
 * @version 1.0
```

\*/

```
public class HspSpringAnnoationMain {  
    public static void main(String[] args) {  
        //创建我们的spring 容器对象  
        HspSpringApplicationContext hspSpringApplicationContext =  
            new HspSpringApplicationContext(HspSpringConfig.class);  
  
        //可以看看注入了哪些bean.  
        ConcurrentHashMap<String, Object> ioc =  
            hspSpringApplicationContext.getIoc();  
        //遍历一把  
        Enumeration<String> keys = ioc.keys();  
        for (String key : ioc.keySet()) {  
            System.out.println("bean id= " + key + " bean 对象= " + ioc.get(key));  
        }  
        //指定获取bean  
        UserDao userDao =  
            (UserDao) hspSpringApplicationContext.getBean("UserDao");  
        //调用方法。  
        userDao.hi();  
    }  
}
```

## 6. 完成测试

```
=====
bean      id=      MyComponent      bean      对      象      =
com.hspedu.spring.component.MyComponent@13221655
```

```
bean id= UserDao bean 对象= com.hspedu.spring.component.UserDao@2f2c9b19
```

```
bean      id=      UserService      bean      对      象      =
com.hspedu.spring.component.UserService@31befd9f
```

```
bean      id=      UserController      bean      对      象      =
com.hspedu.spring.component.UserController@1c20c684
```

```
i am user dao
```

### 2.3.19.3.1 1.搭建基本结构并获取的扫描包

### 2.3.19.3.2 2.获取扫描包下所有.class 文件

### 2.3.19.3.3 3.获取全类名 反射对象 放入容器

### 2.3.19.4 注意事项和细节说明

1. 还可以通过@Component(value = "xx") @Controller(value = "yy") @Service(value = "zz") 中指定的 value, 给 bean 分配 id

1)

修

改

D:\hspedu\_ssm\_temp\spring5\src\com\hspedu\spring\component\MyComponent.java

```
package com.hspedu.spring.component;
```

```
import org.springframework.stereotype.Component;
```

```
/**
```

```
* @author 韩顺平
```

```
* @version 1.0
```

```
*/
```

```
@Component(value = "hsp1")
```

```
// @Component
```

```
public class MyComponent {
```

```
}
```

2)

修

改

D:\hspedu\_ssm\_temp\spring5\src\com\hspedu\spring\annotation\HspSpringApplicationContext.java

```
Class<?> clazz = classLoader.loadClass(classFullPath);
```

```
if (clazz.isAnnotationPresent(Component.class) ||
```

```
    clazz.isAnnotationPresent(Controller.class) ||
```

```
    clazz.isAnnotationPresent(Service.class) ||
```

```
clazz.isAnnotationPresent(Repository.class)) {  
  
    //老师这里只是测试一个，对 Component，取出指定的值  
  
    if(clazz.isAnnotationPresent(Component.class)) {  
  
        Component annotation = clazz.getAnnotation(Component.class);  
  
        if(!"".equals(annotation.value())) {  
  
            className = annotation.value();  
  
        }  
  
    }  
  
....
```

3) 完成测试，这时 @Component(value = "xx") bean 的 id 和 value 就一致了。

2.3.19.5 课后作业布置：把老师写的 Spring 基于注解的程序，写两遍

2.3.20 自动装配

2.3.20.1 应用实例

- 基本说明

1. 基于注解配置 bean，也可实现自动装配，使用的注解是：@AutoWired 或者 @Resource

2. @AutoWired 的规则说明

- 1) 在 IOC 容器中查找待装配的组件的**类型**,如果有唯一的 bean 匹配,则使用该 bean 装配
- 2) 如待装配的类型对应的 bean 在 IOC 容器中有多个,则使用待装配的属性的属性名作为 id 值再进行查找,找到就装配,找不到就抛异常

### 3. @Resource 的规则说明

- 1) @Resource 有两个属性是比较重要的,分是 name 和 type, Spring 将@Resource 注解的 name 属性解析为 bean 的名字,而 type 属性则解析为 bean 的类型.所以如果使用 name 属性,则使用 byName 的自动注入策略,而使用 type 属性时则使用 byType 自动注入策略
- 2) 如果@Resource 没有指定 name 和 type ,则先使用 byName 注入策略, 如果匹配不上,再使用 byType 策略, 如果都不成功, 就会报错

### 4. 老韩建议, 不管是@Autowired 还是 @Resource 都保证属性名是规范的写法就可以注入.

- 应用实例需求

1. 以 Action/Service/Dao 几个组件来进行演示
2. 这里我就演示 UserAction 和 UserService 的两级自动组装

- 应用实例-代码实现

## 1. 修改 UserService.java 和 UserAction.java 增加相关代码

```
@Service
```

```
public class UserService {
```

```
    public void hi(){
```

```
        System.out.println("UserService hi()~");
```

```
}
```

```
}
```

```
@Controller("userAction01")
```

```
public class UserAction {
```

```
    // @Autowired //自动装配 UserService, 这时是以 id=userService 的 UserService 对象进行组装.
```

```
    @Autowired
```

```
    private UserService userService;
```

```
    public void sayOk(){
```

```
        System.out.println("UserAction.userService= " + userService);
```

```
        userService.hi();
```

```
}
```

```
//不写这个方法，也可以完成组装
```

```
    public void setUserService(UserService userService) {
```

```
    this.userService = userService;  
}  
}
```

## 2. 完成测试，修改

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\test\Spring5BeanTest.java, 增加测试方法

```
/**  
 * 基于注解的方式配置 bean-自动装配  
 */  
  
@Test  
public void setProByAnnotationAutowired() {  
    ApplicationContext ioc = new ClassPathXmlApplicationContext("beans.xml");  
    UserAction userAction01 = ioc.getBean(UserAction.class);  
    userAction01.sayOk();  
}
```

### 2.3.20.2 注意事项和细节说明

1. 如待装配的类型对应的 bean 在 IOC 容器中有多个，则使用待装配的属性的属性名作为 id 值再进行查找，找到就装配，找不到就抛异常 [代码演示]

1) 修改 beans.xml 增加一个 bean

&lt;!--

增加一个 UserService 对象

--&gt;

```
<bean id="userService02" class="com.hspedu.spring.component.UserService"/>
```

2) 修改 D:\java\_projects\spring5\src\com\hspedu\spring\component\UserAction.java

```
/*
 * @author 韩顺平
 * @version 1.0
 */
//@Controller
@Controller(value="userAction01")
// @Controller("userAction01")
public class UserAction {
    //@@Autowired //自动装配 UserService, 这时是以 id=userService 的 UserService 对象进
行组装.
    //@@Autowired
    //指定 id 进行组装, 这时, 是装配的 id=userService02, 需要两个注解都需要写上
    @Autowired
    @Qualifier(value = "userService02")//指定 id=userService02 的 UserService 对象进
```

行组装

```
private UserService userService;  
  
public void sayOk(){  
  
    System.out.println("UserAction.userService= " + userService);  
  
    userService.hi();  
}  
  
public void setUserService(UserService userService) {  
  
    this.userService = userService;  
}  
}
```

3) 修改 D:\java\_projects\spring5\src\com\hspepu\spring\test\Spring5BeanTest.java, 观察,  
这时自动装配的 bean 是 id 为 *userService02* 的 bean

```
@Test  
  
public void setProByAnnotationAutowired() {  
  
    ApplicationContext ioc = new ClassPathXmlApplicationContext("beans.xml");  
  
    UserAction userAction01 = ioc.getBean(UserAction.class);  
  
    userAction01.sayOk();
```

```
UserService userService = ioc.getBean("userService", UserService.class);  
  
System.out.println("userService= " + userService);
```

```
UserService userService02 = ioc.getBean("userService02", UserService.class);  
System.out.println("userService02= " + userService02);  
}
```

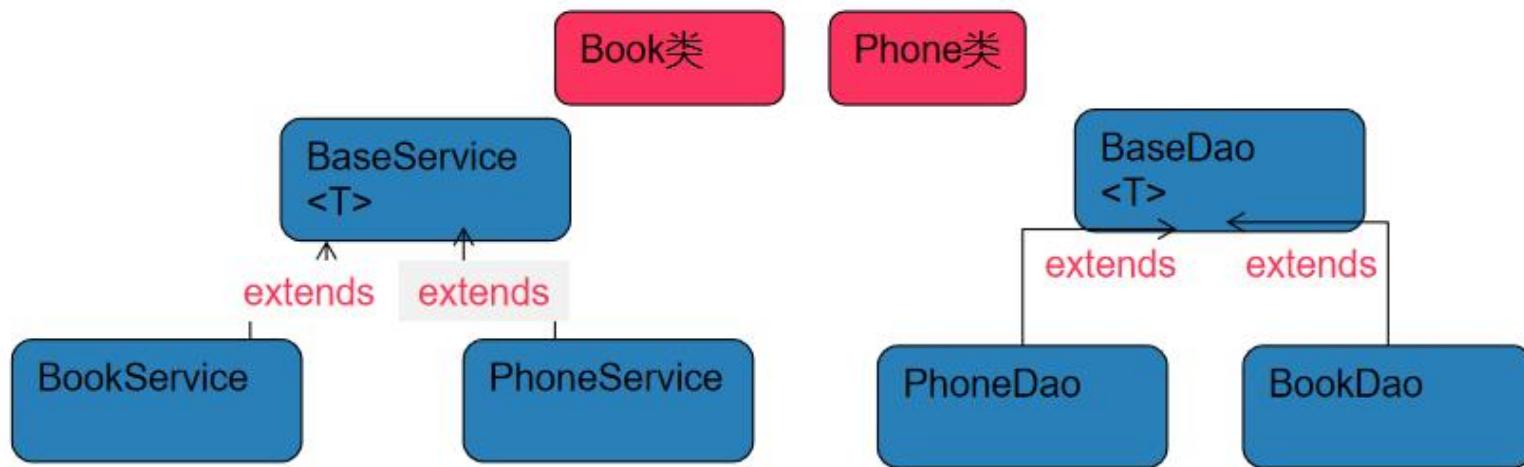
## 2.3.21 泛型依赖注入

### 2.3.21.1 泛型依赖解释

- 基本说明
  - 1. 为了更好的管理有继承和相互依赖的 bean 的自动装配, spring 还提供基于泛型依赖的注入机制
  - 2. 在继承关系复杂情况下, 泛型依赖注入就会有很大的优越性

### 2.3.21.2 应用实例

- 应用实例需求
  - 1. 各个类关系图



2. 传统方法是将 PhoneDao /BookDao 自动装配到 BookService/PhoneService 中，当这种继承关系多时，就比较麻烦，可以使用 spring 提供的泛型依赖注入

- 应用实例-代码实现

1. 创建包 com.hspedu.spring.depinjection，在包下创建一系列的 Java 类(这些类都比较简单，**老师事先已经准备好了**)： Book.java Phone.java..

```
package com.hspedu.spring.depinjection;
```

```
/**
```

```
* @author 韩顺平
```

```
* @version 1.0
```

```
*/
```

```
public class Book {
```

```
}
```

```
=====
```

```
package com.hspedu.spring.depinjection;
```

```
/**
```

```
* @author 韩顺平
```

```
* @version 1.0
```

```
*/
```

```
public class Phone {
```

```
}
```

```
=====
```

```
package com.hspedu.spring.depinjection;
```

```
/**
```

```
* @author 韩顺平
```

```
* @version 1.0
```

```
*/
```

```
public abstract class BaseDao<T> {
```

```
    public abstract void save();
```

```
}
```

```
=====

package com.hspedu.spring.depinjection;

import org.springframework.stereotype.Repository;

/**
 * @author 韩顺平
 * @version 1.0
 */
@Repository
public class BookDao extends BaseDao<Book> {

    @Override
    public void save() {
        System.out.println("BookDao 的 save()");
    }
}
=====
```

```
package com.hspedu.spring.depinjection;

import org.springframework.stereotype.Repository;
```

```
/*
 * @author 韩顺平
 * @version 1.0
 */

@Repository
public class PhoneDao extends BaseDao<Phone> {
    @Override
    public void save() {
        System.out.println("PhoneDao 的 save()");
    }
}

=====
package com.hspedu.spring.depinjection;

import org.springframework.beans.factory.annotation.Autowired;

/*
 * @author 韩顺平
 * @version 1.0
 */
```

```
public class BaseService<T> {  
  
    @Autowired  
    private BaseDao<T> baseDao;  
  
    public void save() {  
        baseDao.save();  
    }  
}  
=====  
package com.hspedu.spring.depinjection;  
  
import org.springframework.stereotype.Service;  
  
/**  
 * @author 韩顺平  
 * @version 1.0  
 */  
@Service  
public class BookService extends BaseService<Book> {  
}
```

```
=====
```

```
package com.hspedu.spring.depinjection;

import org.springframework.stereotype.Service;

/**
 * @author 韩顺平
 * @version 1.0
 */
@Service
public class PhoneService extends BaseService<Phone> {

}
```

## 2. 修改 beans.xml , 增加配置

```
<context:component-scan base-package="com.hspedu.spring.depinjection"/>
```

## 3. 完成测试，修改

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\test\Spring5BeanTest.java , 增加测试方法.

```
/*
```

\* 测试 spring 基于泛型依赖的 bean 的自动装配

\*/

@Test

```
public void setProByDepinjectionAutowired() {  
    ApplicationContext ioc = new ClassPathXmlApplicationContext("beans.xml");  
    BookService bookService = ioc.getBean(BookService.class);  
    bookService.save();  
    PhoneService phoneService = ioc.getBean(PhoneService.class);  
    phoneService.save();  
}
```

### 3 AOP

#### 3.1 官方文档

3.1.1 AOP 讲解: <spring-framework-5.3.8/docs/reference/html/core.html#aop>

3.1.2 AOP APIs : <spring-framework-5.3.8/docs/reference/html/core.html#aop-api>

#### 3.2 动态代理-精致小案例

##### 3.2.1 需求说明

- 需求说明

1. 有 Vehicle(交通工具接口, 有一个 run 方法), 下面有两个实现类 Car 和 Ship
2. 当运行 Car 对象 的 run 方法和 Ship 对象的 run 方法时, 输入如下内容, 注意观察前后有统一的输出.

交通工具开始运行了...

大轮船在水上 running...

交通工具停止运行了...

-----

交通工具开始运行了...

小汽车在公路 running..

交通工具停止运行了...|

### 3. 请思考如何完成.

#### 3.2.2 解决方案-传统方式

- 解决方案 1-代码实现

1. 传统的解决思路, 在各个方法的[前, 执行过程, 后]输出日志提示信息
2. 创建 D:\hspedu\_ssm\_temp\spring5\src\com\hspedu\spring\Proxy2\Vehicle.java

```
package com.hspedu.spring.aop.proxy2;
```

```
/**  
 * @author 韩顺平  
 * @version 1.0  
 * Vehicle 交通工具  
 */  
  
public interface Vehicle {  
    public void run();  
}
```

### 3. 创建 D:\hspedu\_ssm\_temp\spring5\src\com\hspedu\spring\aoP\proxy2\Car.java

```
package com.hspedu.spring.aoP.proxy2;  
  
/**  
 * @author 韩顺平  
 * @version 1.0  
 */  
  
public class Car implements Vehicle {  
    @Override  
    public void run() {  
        System.out.println("交通工具开始运行了...");  
        System.out.println("小汽车在公路 running..");  
        System.out.println("交通工具停止运行了...");  
    }  
}
```

```
}
```

#### 4. 创建 D:\hspedu\_ssm\_temp\spring5\src\com\hspedu\spring\aoP\proxy2\Ship.java

```
package com.hspedu.spring.aoP.proxy2;
```

```
/**  
 * @author 韩顺平  
 * @version 1.0  
 */
```

```
public class Ship implements Vehicle {  
    @Override  
    public void run() {  
        System.out.println("交通工具开始运行了...");  
        System.out.println("大轮船在水上 running...");  
        System.out.println("交通工具停止运行了...");  
    }  
}
```

#### 5. 创建 D:\hspedu\_ssm\_temp\spring5\src\com\hspedu\spring\aoP\proxy2\Test.java

```
Vehicle vehicle = new Car(); //可以切换成 new Ship()
```

```
vehicle.run();
```

6. 来思考一下，解决方案好吗? ==> 代码冗余，其实就是一个对象的调用，并没有很好的解决

### 3.2.3 解决方案-动态代理方式 !!!!!!

- 解决方案 2-代码实现

1. 动态代理解决思路，在调用方法时，使用反射机制，根据方法去决定调用哪个对象方法

2. D:\hspedu\_ssm\_temp\spring5\src\com\hspedu\spring\aoP\proxy2\Vehicle.java 不变

```
package com.hspedu.spring.aop.proxy2;

/**
 * @author 韩顺平
 * @version 1.0
 * Vehicle 交通工具
 */
public interface Vehicle {
    public void run();
}
```

3. 修改 D:\hspedu\_ssm\_temp\spring5\src\com\hspedu\spring\aoP\proxy2\Car.java

```
package com.hspedu.spring.aop.proxy2;
```

```
/**  
 * @author 韩顺平  
 * @version 1.0  
 */  
  
public class Car implements Vehicle {  
    @Override  
    public void run() {  
        //System.out.println("交通工具开始运行了...");  
        System.out.println("小汽车在公路 running..");  
        //System.out.println("交通工具停止运行了...");  
    }  
}
```

#### 4. 修改 D:\hspedu\_ssm\_temp\spring5\src\com\hspedu\spring\Proxy2\Ship.java

```
package com.hspedu.spring.aop.proxy2;
```

```
/**  
 * @author 韩顺平  
 * @version 1.0  
 */  
  
public class Ship implements Vehicle {  
    @Override
```

```
public void run() {  
    //System.out.println("交通工具开始运行了...");  
    System.out.println("大轮船在水上 running...");  
    //System.out.println("交通工具停止运行了...");  
}  
}
```

5.

创

建

D:\hsedu\_ssm\_temp\spring5\src\com\hsedu\spring\Proxy2\VehicleProxyProvider.java

```
package com.hsedu.spring.aop.proxy2;
```

```
import com.hsedu.spring.aop.proxy.SmartAnimalable;
```

```
import java.lang.reflect.InvocationHandler;
```

```
import java.lang.reflect.Method;
```

```
import java.lang.reflect.Proxy;
```

```
/**
```

```
 * @author 韩顺平
```

```
 * @version 1.0
```

```
 */
```

```
public class VehicleProxyProvider {  
    //设置一个将来要运行的对象，只要是实现了 Vehicle 接口的就 ok  
    private Vehicle target_vehicle;  
  
    public VehicleProxyProvider(Vehicle target_vehicle) {  
        this.target_vehicle = target_vehicle;  
    }  
  
    //编写代码，返回一个 Vehicle 的代理对象  
    public Vehicle getProxy() {  
        // 1. 获取类加载对象  
        ClassLoader loader = target_vehicle.getClass().getClassLoader();  
        // 2. 获取接口类型数组，为什么是接口信息，而不是方法，是因为我们都是走接口  
        // 的。  
        Class<?>[] interfaces = target_vehicle.getClass().getInterfaces();  
        // 3. 创建 InvocationHandler 对象 - 以匿名内部类的方式来获取  
        // InvocationHandler  
        // 这个对象有一个方法是 invoke 到时可以通过反射，动态调用目标对象的方  
        // 法  
        InvocationHandler invocationHandler = new InvocationHandler() {  
            @Override
```

```
public Object invoke(Object proxy, Method method, Object[] args) throws  
Throwable {  
    System.out.println("交通工具开始运行了...");  
    //这个地方的方法,就是你调用时, 动态传入的可能是run, 可能是hi等  
    Object result = method.invoke(target_vehicle, args);  
    System.out.println("交通工具停止运行了...");  
    return result;  
}  
;  
  
//将上面的 loader, interfaces, invocationHandler 构建一个 Vehicle  
//的代理对象.  
Vehicle proxy = (Vehicle)Proxy.newProxyInstance(loader, interfaces,  
invocationHandler);  
return proxy;  
}  
}
```

## 6. 创建 D:\hspedu\_ssm\_temp\spring5\src\com\hspedu\spring\aoP\proxy2\Test.java

```
package com.hspedu.spring.aoP.proxy2;
```

```
import com.hspedu.spring.aop.proxy.MyProxyProvider;

/**
 * @author 韩顺平
 * @version 1.0
 */

public class Test {
    public static void main(String[] args) {
        //这里可以切换 Vehicle 的 实现类(对象)
        Vehicle vehicle = new Car();
        VehicleProxyProvider vehicleProxyProvider =
            new VehicleProxyProvider(vehicle);

        //给学员看一下 proxy 的结构.
        Vehicle proxy = vehicleProxyProvider.getProxy();
        System.out.println("proxy 编译类型是 Vehicle");
        System.out.println("proxy 运行类型" + proxy.getClass());
        //动态代理的 动态怎么体现
        //老韩认为
        //1. proxy 运行类型是 com.sun.proxy.$Proxy0 该类型被转型成 Vehicle
```

```
// 因此可以调用 Vehicle 的接口方法  
//2. 当执行 run() 的时候会调用, 根据 Java 的动态绑定机制, 这时直接调用 Car  
的 run(), 而是 proxy 对象的 invocationHandler 的 invoke 方法!!!!!!  
//3. invoke 方法使用反射机制来调用 run() 方法注意这个 run 方法也可以是  
Vehicle 的其它方法)  
  
// 这时就可以在调用 run() 方法前, 进行前置处理和后置处理  
  
//4. 也就是说 proxy 的 target_vehicle 运行类型只要是实现了 Vehicle 接口  
// , 就可以去调用不同的方法, 是动态的, 变化的, 底层就是 使用反射完成的.  
proxy.run();  
}  
}
```

### 3.2.4 把老师写的代码，自己独立写 2 遍

## 3.3 课后作业-动态代理深入

### 3.3.1 需求说明

- 需求说明

1. 有一个 SmartAnimal 接口，可以完成简单的加减法，要求在执行 getSum() 和 getSub() 时，输出执行前，执行过程，执行后的日志输出，请思考如何实现。

日志-方法名-getSum-参数 10.0 2.0

方法内部打印 **result = 12.0**

日志-方法名-getSum-结果 **result= 12.0**

=====

日志-方法名-getSub-参数 **10.0 2.0**

方法内部打印 **result = 8.0**

日志-方法名-getSub-结果 **result= 8.0**

2. 请使用传统方法完成

3. 请使用动态代理方式完成，**并要求考虑代理对象调用方法(底层是反射调用)时， 可能出现的异常-**

方法执行开始-日志-方法名-getSub-参数 **[10.0, 2.0]**

方法内部打印 **result = 8.0**

方法执行正常结束-日志-方法名-getSub-结果 **result= 8.0**

方法最终结束-日志-方法名-getSub

=====

方法执行开始-日志-方法名-getSum-参数 **[10.0, 2.0]**

方法内部打印 **result = 12.0**

方法执行正常结束-日志-方法名-getSum-结果 result= 12.0

方法最终结束-日志-方法名-getSum

### 3.3.2 解决方案- 传统方式和动态代理方式

- 解决方案 1-代码实现

1. 传统的解决思路，在各个方法的[前，执行过程，后]输出日志

2.

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\Proxy\SmartAnimalable.java

```
package com.hspedu.spring.proxy;

/**
 * @author 韩顺平
 * @version 1.0
 */

public interface SmartAnimalable {
    float getSum(float i, float j);
    float getSub(float i, float j);
}
```

3. 创建 D:\idea\_java\_projects\spring5\src\com\hspedu\spring\Proxy\SmartDog.java

```
package com.hspedu.spring.aop.proxy;

/**
 * @author 韩顺平
 * @version 1.0
 */

public class SmartDog implements SmartAnimalable {

    @Override
    public float getSum(float i, float j) {
        System.out.println("日志--方法名--getSum 方法开始--参数: " + i + "," + j);
        float result = i + j;
        System.out.println("方法内部打印: result=" + result);
        System.out.println("日志--方法名--getSum 方法结束--结果: result=" + result);
        return result;
    }

    @Override
    public float getSub(float i, float j) {
        System.out.println("日志--方法名--getSub 方法开始--参数: " + i + "," + j);
        float result = i - j;
        System.out.println("方法内部打印: result=" + result);
        System.out.println("日志--方法名--getSub 方法结束--结果: result=" + result);
        return result;
    }
}
```

}

#### 4. 完成测试，创建

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\AopTest.java

日志--方法名--getSum方法开始--参数: 10.78,89.7

方法内部打印: result=100.479996

日志--方法名--getSum方法结束--结果: result=100.479996

日志--方法名--getSub方法开始--参数: 10.78,89.7

方法内部打印: result=-78.92

日志--方法名--getSub方法结束--结果: result=-78.92

```
package com.hspedu.spring.aop.proxy;
```

```
import org.junit.jupiter.api.Test;
```

```
/**
```

```
 * @author 韩顺平
```

```
 * @version 1.0
```

```
 */
```

```
public class AopTest {
```

```
    @Test
```

```
    public void smartDogTest() {
```

```
SmartDog smartDog = new SmartDog();  
smartDog.getSum(10.78f, 89.7f);  
smartDog.getSub(10.78f, 89.7f);  
}  
}
```

- 思考解决方案 1 的缺陷



1. 优点：实现简单直接
2. 缺点：日志代码维护不方便，代码复用性差

- 解决思路

1. 使用动态代理来更好的处理日志记录问题
2. 其它比如封装函数，或者类的继承在这里都不是特别合适

- 解决方案 2-使用动态代理

1.

开

发

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\oop\proxy\MyProxyProvider.java， 使用动态代理来完成

```
package com.hspedu.spring.aop.proxy;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.Arrays;

/**
 * @author 韩顺平
 * @version 1.0
 */
public class MyProxyProvider {

    private SmartAnimalable target_obj;
    // 构造器

    public MyProxyProvider(SmartAnimalable target_obj) {
        this.target_obj = target_obj;
    }
}
```

```
}

public SmartAnimalable getProxy() {

    // 1. 获取类加载对象

    ClassLoader loader = target_obj.getClass().getClassLoader();

    // 2. 获取接口类型数组

    Class<?>[] interfaces = target_obj.getClass().getInterfaces();

    // 3. 获取 InvocationHandler 以匿名内部类的方式来获取 InvocationHandler

    InvocationHandler h = new InvocationHandler() {

        // 4. 以动态代理的方式调用目标对象的目标方法

        public Object invoke(Object proxy, Method method, Object[] args)

            throws Throwable {

            Object result = null;

            String methodName = method.getName();

            try {

                // 1. 在调用目标方法之前打印“方法开始”日志

                System.out.println("日志--方法名：" + methodName + "--方法开始--" + Arrays.asList(args));

                // 2. 调用目标方法并接收返回值

                result = method.invoke(target_obj, args);

                // 3. 在目标方法结束后打印“方法结束”日志

                System.out.println("日志--方法名：" + methodName + "--方法结束--");

            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    };
}
```

```
+ "--方法正常结束--结果: result=" + result);

} catch (Exception e) {

    // 4.如果目标方法抛出异常，打印“方法异常”日志

    e.printStackTrace();

    System.out.println("日志--方法名: " + methodName

        + "--方法抛出异常--异常类型: " + e.getClass().getName());

} finally {

    // 5.在finally 中打印“方法最终结束”日志

    System.out.println("日志--方法名: " + methodName + "--方法最终结

束");

}

// 6. 返回目标方法的返回值

return result;

};

//生成 SmartAnimaleable 的代理对象

//需要三个参数，

//1.就是 loader(获取类加载对象)

//2.接口数组

//3.InvocationHandler 对象 [这个相对麻烦..]

SmartAnimaleable proxy = (SmartAnimaleable) Proxy.newProxyInstance(

    loader, interfaces, h);
```

```
    return proxy;  
}  
}
```

## 2. 修改 D:\idea\_java\_projects\spring5\src\com\hspedu\spring\Proxy\SmartDog.java

```
public class SmartDog implements SmartAnimalable {  
    @Override  
    public float getSum(float i, float j) {  
        // System.out.println("日志--方法名--getSum 方法开始--参数: " + i + "," + j);  
        float result = i + j;  
        // System.out.println("方法内部打印: result=" + result);  
        // System.out.println("日志--方法名--getSum 方法结束--结果: result=" + result);  
        return result;  
    }  
    @Override  
    public float getSub(float i, float j) {  
        // System.out.println("日志--方法名--getSub 方法开始--参数: " + i + "," + j);  
        float result = i - j;  
        // System.out.println("方法内部打印: result=" + result);  
        // System.out.println("日志--方法名--getSub 方法结束--结果: result=" + result);  
        return result;  
    }  
}
```

}

### 3. 完成测试：修改

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\aspect\proxy\AopTest.java, 增加新的测试方法

```
package com.hspedu.spring.aspect;
```

```
import org.junit.jupiter.api.Test;
```

```
/**
```

```
* @author 韩顺平
```

```
* @version 1.0
```

```
*/
```

```
public class AopTest {
```

```
    @Test
```

```
    public void smartDogTest() {
```

```
        SmartDog smartDog = new SmartDog();
```

```
        smartDog.getSum(10.78f, 89.7f);
```

```
        smartDog.getSub(10.78f, 89.7f);
```

```
}
```

```
/**
```

\* 动态代理方式完成日志输出

\*/

@Test

```
public void smartDogTestByProxy() {
```

```
    SmartAnimalable smartDog = new SmartDog();
```

//老韩解读

//1. 使用动态代理的方式来调用

//2. 当使用动态代理调用是 SmartDog 的各个函数的日志输出就可以不写了，有代理对象帮我们完成。

```
    MyProxyProvider provider = new MyProxyProvider(smartDog);
```

```
    smartDog = provider.getProxy();
```

```
    smartDog.getSum(10.78f, 89.7f);
```

```
    smartDog.getSub(10.78f, 89.7f);
```

```
}
```

```
}
```

### 3.4 老韩分析：问题再次出现

#### 3.4.1 问题提出

- 问题提出

1. 在 MyProxyProvider.java 中，我们的输出语句功能比较弱，在实际开发中，我们希望是

以一个方法的形式，嵌入到真正执行的目标方法前，怎么办？

## 2. 也就是如图分析

```

public Object invoke(Object proxy, Method method, Object[] args)
    throws Throwable {
    Object result = null;
    String methodName = method.getName();
    try {
        // 1. 在调用目标方法之前打印“方法开始”日志
        System.out.println("日志--方法名: " + methodName + "--方法开始--参数: "
            + Arrays.asList(args));
        // 2. 调用目标方法并接收返回值
        result = method.invoke(target_obj, args);
        // 3. 在目标方法结束后打印“方法结束”日志
        System.out.println("日志--方法名: " + methodName
            + "--方法正常结束--结果: result=" + result);
    } catch (Exception e) {
        // 4. 如果目标方法抛出异常，打印“方法异常”日志
        e.printStackTrace();
        System.out.println("日志--方法名: " + methodName
    }
}

```

### 3.4.2 用老韩的土方法解决

1. 需求分析：使用老韩的土方法解决前面的问题 => 后面使用 Spring 的 AOP 组件完成，先过苦日子，再过甜日子

2. 先新建一个包，把相关文件拷贝过来，进行修改完成，思路更加清晰。老师学习小技巧：新建一个包，保留原来的代码

3. 修 改

D:\hsedu\_ssm\_temp\spring5\src\com\hsedu\spring\proxy3\MyProxyProvider.java

```
package com.hspedu.spring.aop.proxy3;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.Arrays;

/**
 * @author 韩顺平
 * @version 1.0
 */
public class MyProxyProvider {

    private SmartAnimalable target_obj;

    // 构造器
    public MyProxyProvider(SmartAnimalable target_obj) {
        this.target_obj = target_obj;
    }

    public void before(Object proxy, Method method, Object[] args) {
        // 1. 在调用目标方法之前打印“方法开始”日志
    }
}
```

```
System.out.println("before 日志--方法名: " + method.getName() + "--方法开始--参数: "
+ Arrays.asList(args));
}

public void after(Method method, Object result) {
    // 3. 在目标方法结束后打印“方法结束”日志
    System.out.println("after 日志--方法名: " + method.getName()
        + "--方法正常结束--结果: result=" + result);
}

//可以更多方法

public SmartAnimalable getProxy() {
    // 1. 获取类加载对象
    ClassLoader loader = target_obj.getClass().getClassLoader();
    // 2. 获取接口类型数组，为什么是接口信息，而不是方法，是因为我们都是走接口的
    Class<?>[] interfaces = target_obj.getClass().getInterfaces();
    // 3. 创建 InvocationHandler 对象 - 以匿名内部类的方式来获取 InvocationHandler
    //
    InvocationHandler h = new InvocationHandler() {
        // 4. 以动态代理的方式调用目标对象的目标方法
    }
}
```

```
public Object invoke(Object proxy, Method method, Object[] args)
    throws Throwable {
    Object result = null;
    String methodName = method.getName();
    try {
        before(proxy,method,args);//切入到目标方法前
        // 2. 调用目标方法并接收返回值
        result = method.invoke(target_obj, args);
        after(method, result);//切入到目标方法后
    } catch (Exception e) {
        // 4. 如果目标方法抛出异常，打印“方法异常”日志
        e.printStackTrace();
        System.out.println("日志--方法名：" + methodName
            + " -- 方法抛出异常 -- 异常类型：" + e
            .getClass().getName());
    } finally {
        // 5. 在finally中打印“方法最终结束”日志
        System.out.println("日志--方法名：" + methodName + "--方法最终
结束");
    }
    // 6. 返回目标方法的返回值
}
```

```
        return result;  
    }  
};  
//生成 SmartAnimaleable 的代理对象  
//需要三个参数 ,  
//1.就是 loader(获取类加载对象)  
//2.接口数组  
//3.InvocationHandler 对象 [这个相对麻烦..]  
SmartAnimalable proxy = (SmartAnimalable) Proxy.newProxyInstance(  
    loader, interfaces, h);  
return proxy;  
}  
}
```

#### 4. 该方法问题分析：耦合度高

##### 3.4.3 对土方法解耦-开发简易的 AOP 类

###### 1. 创建 D:\hspedu\_ssm\_temp\spring5\src\com\hspedu\spring\aoP\proxy3\HspAO.java

```
package com.hspedu.spring.aoP.proxy;
```

```
import java.lang.reflect.Method;
```

```
import java.util.Arrays;
```

```
/**  
 * @author 韩顺平  
 * @version 1.0  
 * 自己写的一个切入类  
 */  
  
public class HspAOP {  
  
    public static void before(Object proxy, Method method, Object[] args) {  
        // 1. 在调用目标方法之前打印“方法开始”日志  
        System.out.println("自己的切入类的 before 日志--方法名：" + method.getName()  
            + "--方法开始--参数："  
            + Arrays.asList(args));  
    }  
  
    public static void after(Method method, Object result) {  
        // 3. 在目标方法结束后打印“方法结束”日志  
        System.out.println("自己的切入类的 after 日志--方法名：" + method.getName()  
            + "--方法正常结束--结果：" + result);  
    }  
}
```

2.

修

改

D:\hspepu\_ssm\_temp\spring5\src\com\hspepu\spring\Proxy3\MyProxyProvider.java

```
public Object invoke(Object proxy, Method method, Object[] args)
    throws Throwable {
    Object result = null;
    String methodName = method.getName();
    try {
        //before(proxy,method,args); //切入到目标方法前
        HspAOP.before(proxy,method,args);
        // 2. 调用目标方法并接收返回值
        result = method.invoke(target_obj, args);
        //after(method, result); //切入到目标方法后
        HspAOP.after(method,result);
    } catch (Exception e) {
        // 4. 如果目标方法抛出异常，打印“方法异常”日志
        e.printStackTrace();
        System.out.println("日志--方法名：" + methodName
            + "--方法抛出异常--异常类型：" + e.getClass().getName());
    } finally {
        // 5. 在finally 中打印“方法最终结束”日志
        System.out.println("日志--方法名：" + methodName + "--方法最终结束");
    }
}
```

```
// 6. 返回目标方法的返回值  
return result;  
}
```

### 3. 完成测试

自己的切入类的 before 日志--方法名: getSum--方法开始--参数: [10.78, 89.7]

执行目标方法 getSum

自己的切入类的 after 日志--方法名: getSum--方法正常结束--结果: result=100.479996

日志--方法名: getSum--方法最终结束

自己的切入类的 before 日志--方法名: getSub--方法开始--参数: [10.78, 89.7]

执行目标方法 getSub

自己的切入类的 after 日志--方法名: getSub--方法正常结束--结果: result=-78.92

日志--方法名: getSub--方法最终结束



### 3.4.4 再次分析-提出 Spring AOP

3.4.4.1 土方法 不够灵活

3.4.4.2 土方法 复用性差

3.4.4.3 土方法 还是一种硬编码(因为没有注解和反射支撑)

3.4.4.4 Spring AOP 闪亮登场-底层是 ASPECTJ

3.4.4.5 有了前面的技术引导，理解 Spring AOP 就水到渠成

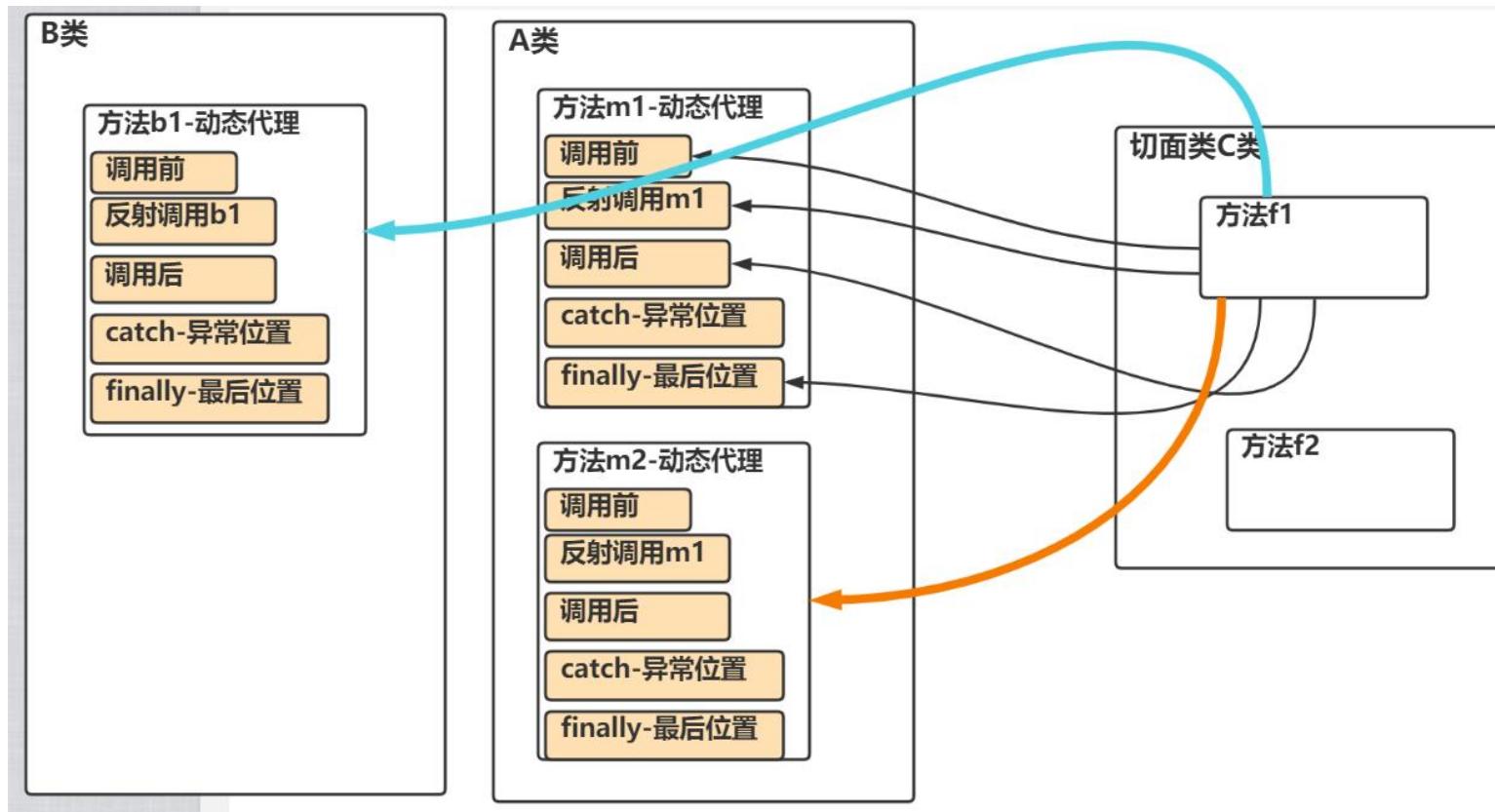
## 3.5 AOP 的基本介绍

- 什么是 AOP

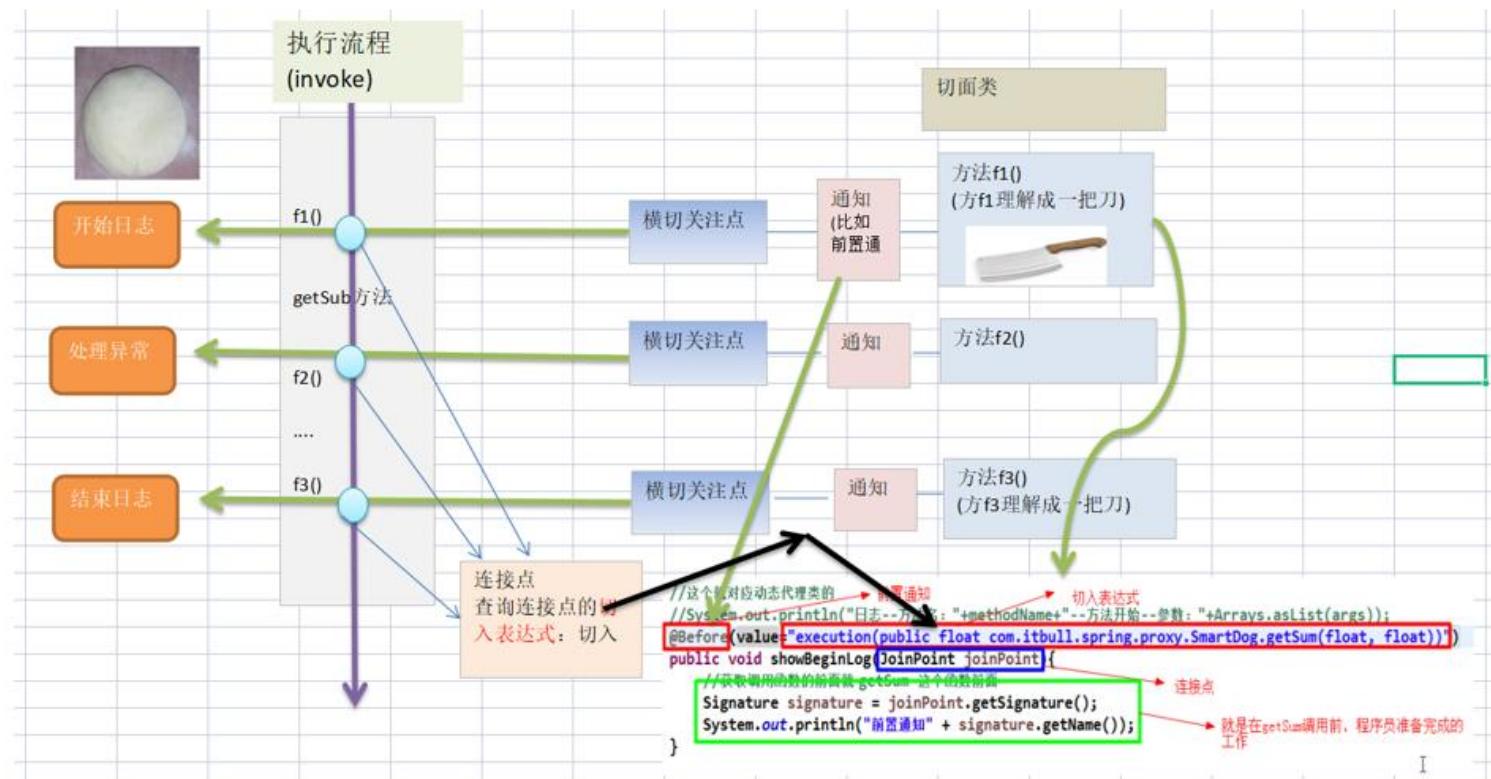
AOP 的全称(aspect oriented programming) , 面向切面编程

- 2 张示意图说明 AOP 的相关概念

1) 一张简易图说明 AOP



## 2) 一张详细图说明 AOP



- AOP 实现方式

1. 基于动态代理的方式[内置 aop 实现]

2. 使用框架 aspectj 来实现

## 3.6 AOP 编程快速入门

### 3.6.1 基本说明

- 说明

1. 需要引入核心的 aspect 包

2. 在切面类中声明通知方法

- 1) 前置通知：@Before

- 2) 返回通知：@AfterReturning

- 3) 异常通知：@AfterThrowing

- 4) 后置通知：@After

- 5) 环绕通知：@Around

```

try {
    //在调用目标方法之前打印“方法开始”日志
    System.out.println("日志--方法名: "+methodName+"--方法开始--参数: "+Arrays.asList(args));
    //调用目标方法并接收返回值
    result = method.invoke(target_obj, args);
    //在目标方法结束后打印“方法结束”日志
    System.out.println("日志--方法名: "+methodName+"--方法正常结束--结果: result="+result);
} catch (Exception e) {
    //如果目标方法抛出异常，打印“方法异常”日志
    e.printStackTrace();
    System.out.println("日志--方法名: "+methodName+"--方法抛出异常--异常类型: "+e.getClass().getName());
} finally {
    //在finally中打印“方法最终结束”日志
    System.out.println("日志--方法名: "+methodName+"--方法最终结束");
}

```

### 3. 五种通知和前面写的动态代理类方法的对应关系

#### 3.6.2 快速入门实例

- 需求说明

我们使用 aop 编程的方式，来实现手写的动态代理案例效果，就以上一个案例为例来讲解（如图）

前置通知  
方法内部打印: result=21.2  
最终通知  
返回通知

- 代码实现步骤

## 1. 导入 AOP 编程需要的包

```
com.springsource.net.sf.cglib-2.2.0.jar  
com.springsource.org.aopalliance-1.0.0.jar  
com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar  
commons-logging-1.1.3.jar  
spring-aop-5.3.8.jar  
spring-aspects-5.3.8.jar  
spring-beans-5.3.8.jar  
spring-context-5.3.8.jar  
spring-core-5.3.8.jar  
spring-expression-5.3.8.jar
```

## 2.

创

建

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\AOP\AspectJ\SmartAnimalable.java

前面有，拷贝即可

```
package com.hspedu.spring.aop.aspectj;  
  
/**  
 * @author 韩顺平  
 * @version 1.0  
 */  
  
public interface SmartAnimalable {  
    float getSum(float i, float j);  
    float getSub(float i, float j);  
}
```

### 3. 创建 D:\idea\_java\_projects\spring5\src\com\hspedu\spring\aspectj\SmartDog.java 前面有，拷贝并做修改，加入注解@Component

```
package com.hspedu.spring.aop.aspectj;
```

```
import org.springframework.stereotype.Component;
```

```
/**
```

```
* @author 韩顺平
```

```
* @version 1.0
```

```
*/
```

```
@Component
```

```
public class SmartDog implements SmartAnimalable {
```

```
    @Override
```

```
    public float getSum(float i, float j) {
```

```
        float result = i + j;
```

```
        System.out.println("getSum() 方法内部打印 result= " + result);
```

```
        return result;
```

```
}
```

```
    @Override
```

```
    public float getSub(float i, float j) {
```

```
        float result = i - j;
```

```
System.out.println("getSub() 方法内部打印 result= " + result);

return result;

}

}
```

4.

创

建

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\aspectj\SmartAnimalAspect.java

**韩老师提醒**：SmartAnimalAspect 作用就是去接管切面编程，此时原来的 MyProxyProvider 类就可以拿掉了。

```
package com.hspedu.spring.aop.springaop;

/**

 * @author 韩顺平
 * @version 1.0
 */

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.Signature;
import org.aspectj.lang.annotation.*;
import org.springframework.stereotype.Component;

import java.util.Arrays;
```

```
/**  
 * 使用切面编程来替代原来的动态代理类，机制是一样的。  
 * @author Administrator  
 */  
  
@Aspect //表示这个类是一个切面类  
@Component //需要加入到IOC 容器  
  
public class SmartAnimalAspect {  
    //这个就对应动态代理类的  
    //System.out.println(" 日志--方法名：" + methodName + "--方法开始--参数：  
    "+Arrays.asList(args));  
    @Before(value = "execution(public  
com.hspedu.spring.aop.springaop.SmartDog.getSum(float ,float))")  
    public void showBeginLog(JoinPoint joinPoint) {  
        System.out.println("前置通知");  
        Signature signature = joinPoint.getSignature();  
        // 1. 在调用目标方法之前打印“方法开始”日志  
        System.out.println("日志--方法名：" + signature.getName() + "--方法开始--参数：  
"  
            + Arrays.asList(joinPoint.getArgs()));  
    }  
}
```

```
//这个就对应动态代理类的  
//System.out.println("日志--方法名：" + methodName + "--方法正常结束--结果：  
result = "+result);  
  
@AfterReturning(value = "execution(public float  
com.hspedu.spring.aop.springaop.SmartDog.getSum(float ,float))")  
  
public void showSuccessEndLog(JoinPoint joinPoint) {  
  
    System.out.println("返回通知");  
  
    Signature signature = joinPoint.getSignature();  
  
    // 3. 在目标方法结束后打印“方法结束”日志  
  
    System.out.println("日志--方法名：" + signature.getName() + "--方法正常结束  
--~");  
}  
  
//这个就对应动态代理类的  
//System.out.println("日志--方法名：" + methodName + "--方法抛出异常--异常类型：  
"+e.getClass().getName());  
  
@AfterThrowing(value = "execution(public float  
com.hspedu.spring.aop.springaop.SmartDog.getSum(float ,float))")  
  
public void showExceptionLog() {  
  
    System.out.println("异常通知");  
}  
//
```

```
//这个就对应动态代理类的  
//System.out.println("日志--方法名: "+methodName+"--方法最终结束");  
@After(value = "execution(public float  
com.hspedu.spring.aop.springaop.SmartDog.getSum(float ,float))")  
public void showFinallyEndLog() {  
    System.out.println("最终通知");  
}  
}
```

## 5. 创建 src\beans6.xml 并配置

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

```
<!-- 配置自动扫描的包,根据实际情况配置即可 -->
<context:component-scan base-package="com.hspedu.spring.aop.aspectj"/>
<!-- 开启基于注解的AOP 功能 -->
<aop:aspectj-autoproxy/>

</beans>
```

## 6. 测 试 , 创 建

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\aspectj\AopAspectjTest.java

```
package com.hspedu.spring.aop.springaop;

import org.junit.jupiter.api.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * @author 韩顺平
 * @version 1.0
 */

public class AopAspectjTest {
```

```
/*
 * spring aop 方式切入前置before 和 后置after 方法
 */

@Test
public void smartDogTestByProxy() {

    ApplicationContext ioc = new ClassPathXmlApplicationContext("beans6.xml");
    //通过接口来
    SmartAnimalable bean = ioc.getBean(SmartAnimalable.class);
    float sum = bean.getSum(101.0f, 11.2f);
    System.out.println("sum= " + sum);
    //getSub 没有AOP 注解，所以就是普通调用
    //bean.getSub(30.6f, 43.2f);
    System.out.println("-----");
}

}
```

## 7. 输出效果图

前置通知

日志--方法名: `getSum`--方法开始--参数: [101.0, 11.2]

目标方法`getSum()` 被执行

返回通知

日志--方法名: `getSum`--方法正常结束---~

最终通知

`SUM= 112.2`

### 3.6.3 细节说明

- 关于切面类方法命名可以自己规范一下，比如 `showBeginLog()` . `showSuccessEndLog()` `showExceptionLog()` , `showFinallyEndLog()`
- 切入表达式的更多配置，比如使用模糊配置

```
@Before(value="execution(* com.hspedu.aop.proxy.SmartDog.*(..))")
```

- 表示所有访问权限，所有包的下所有有类的所有方法，都会被执行该前置通知方法

```
@Before(value="execution(* *.*(..))")
```

- 当 spring 容器开启了 `<!-- 开启基于注解的AOP 功能 --> <aop:aspectj-autoproxy/>`，我们获取注入的对象，需要以接口的类型来获取，因为你注入的对象`.getClass()` 已经是代理类型了！

5. 当 spring 容器开启了 `<!-- 开启基于注解的AOP 功能 --> <aop:aspectj-autoproxy/>`，我们获取注入的对象，也可以通过 id 来获取，但是也要转成接口类型。

#### 3.6.4 课后作业

1. 有接口 `UsbInterface` (方法 `work`)
2. 实现子类 `Phone` 和 `Camera` 实现 `UsbInterface`
3. 请在 `SmartAnimalAspect` 切面类，写一个方法(可输出日志信息) 等作为前置通知，在 `Phone` 和 `Camera` 对象执行 `work` 方法前调用
4. 其它返回通知，异常通知，后置通知，也可以加入

## 3.7 AOP-切入表达式

### 3.7.1 具体使用

#### 切入点表达式

##### 1.1 作用

通过**表达式的方式**定位一个或多个具体的连接点。

##### 1.2 语法细节

①切入点表达式的语法格式

```
execution([权限修饰符] [返回值类型] [简单类名/全类名] [方法名]([参数列表]))
```

##### ②举例说明

表达式	execution(* com.sina.spring.ArithmeticCalculator.*(..))
含义	ArithmeticCalculator 接口中声明的所有方法。 第一个“*”代表任意修饰符及任意返回值。 第二个“*”代表任意方法。 “..”匹配任意数量、任意类型的参数。 若目标类、接口与该切面类在同一个包中可以省略包名。

表达式	execution(public * ArithmeticCalculator.*(..))
含义	ArithmeticCalculator 接口的所有公有方法

表达式	execution(public double ArithmeticCalculator.*(..))
含义	ArithmeticCalculator 接口中返回 double 类型数值的方法

表达式	execution(public double ArithmeticCalculator.*(double, ..))
含义	第一个参数为 double 类型的方法。 “..”匹配任意数量、任意类型的参数。

表达式	execution(public double ArithmeticCalculator.*(double, double))
含义	参数类型为 double, double 类型的方法

③在 AspectJ 中，切入点表达式可以通过“`&&`”、“`||`”、“`!`”等操作符结合起来。

表达式	<code>execution(* *.add(int,...))    execution(* *.sub(int,...))</code>
含义	任意类中第一个参数为 int 类型的 add 方法或 sub 方法

### 3.7.2 注意事项和细节

1. 切入表达式也可以指向类的方法，这时切入表达式会对该类/对象生效
2. 切入表达式也可以指向接口的方法，这时切入表达式会对实现了接口的类/对象生效
3. 切入表达式也可以对没有实现接口的类，进行切入【举例说明】

```
class Car {  
  
    public void run() {  
  
        System.out.println("car run");  
  
    }  
  
}
```

4. 老师补充：动态代理 jdk 的 Proxy 与 Spring 的 CGLib

<https://www.cnblogs.com/threeAgePie/p/15832586.html>

### 3.8 AOP-JoinPoint

#### 3.8.1 应用实例

- 通过 JoinPoint 可以获取到调用方法的签名

- 应用实例需求

说明：在调用前置通知获取到调用方法的签名，和其它相关信息

- 应用实例-代码实现

前面我们已经举例说明过了

- 其它常用方法一览

```
public void beforeMethod(JoinPoint joinPoint){  
    joinPoint.getSignature().getName(); // 获取目标方法名  
    joinPoint.getSignature().getDeclaringType().getSimpleName(); // 获取目标方法所属  
    类的简单类名  
    joinPoint.getSignature().getDeclaringTypeName(); // 获取目标方法所属类的类名  
    joinPoint.getSignature().getModifiers(); // 获取目标方法声明类型(public、private、  
protected)
```

```
Object[] args = joinPoint.getArgs(); // 获取传入目标方法的参数，返回一个数组  
joinPoint.getTarget(); // 获取被代理的对象  
joinPoint.getThis(); // 获取代理对象自己  
}
```

## 3.9 AOP-返回通知获取结果

### 3.9.1 应用实例

- 如何在返回通知方法获取返回结果

看一个需求：

在返回通知方法获取返回的结果。

- 案例演示

1.

修

改

D:\java\_projects\spring5\src\com\hspedu\spring\aspect\joinpoint\SmartAnimalAspect.java

```
/*
 * returning = "res", Object res 名称保持一致
 * @param joinPoint
 * @param res 调用getSum() 返回的结果
 *
 */
@AfterReturning(value = "execution(public float com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float, float))",
    returning = "res")
public void showSuccessEndLog(JoinPoint joinPoint, Object res) {
    System.out.println("返回通知" + "--结果是--" + res );
}
```

**SmartAnimalAspect.java**

//注销原来的showSuccessEndLog()

//这个就对应动态代理类的

//System.out.println(" 日志-- 方法名：" +methodName+ "-- 方法正常结束-- 结果：

result = "+result);

// @AfterReturning(value = "execution(public float com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float ,float))")

// public void showSuccessEndLog() {

// System.out.println("返回通知");

// }

/\*\*

\* returning = "res", Object res 名称保持一致

\* @param joinPoint

\* @param res 调用 getSum() 返回的结果

\*

```
*/  
  
@AfterReturning(value = "execution(public float  
com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float, float))",  
returning = "res")  
  
public void showSuccessEndLog(JoinPoint joinPoint, Object res) {  
    System.out.println("返回通知" + "--结果是--" + res);  
}  
  
}
```

## 2. 完成测试

前置通知--调用的方法是 getSum--参数是 --[10.0, 11.2]

getSum() 方法内部打印 result= 21.2

返回通知--结果是--21.2

最终通知

getSub() 方法内部打印 result= -1.1999998

### 3.10 AOP-异常通知中获取异常

#### 3.10.1 应用实例

- 异常通知方法中获取异常

看一个需求：

## 如何在异常通知方法中获取异常信息。

- 案例演示

### 1. 在原来的代码上修改即可

```
@Component
public class SmartDog implements SmartAnimalable {
    @Override
    public float getSum(float i, float j) {
        float result = i + j;
        System.out.println("getSum() 方法内部打印 result= " + result);
        result = 1 / 0; → 故意抛出异常
        return result;
    }
    @Override
    public float getSub(float i, float j) {
        float result = i - j;
        System.out.println("getSub() 方法内部打印 result= " + result);
        return result;
    }
}
```

2.

修

改

D:\java\_projects\spring5\src\com\hspedu\spring\aspect\joinpoint\SmartAnimalAspect.java

//注销 showException()

//这个就对应动态代理类的

//System.out.println(" 日志-- 方法名：" + methodName + "-- 方法抛出异常-- 异常类型：

" + e.getClass().getName());

// @AfterThrowing(value = "execution(public float

com.hspedu.spring.aspect.joinpoint.SmartDog.getSum(float ,float))")

```
// public void showExceptionLog() {  
//     System.out.println("异常通知");  
// }  
  
@AfterThrowing(value = "execution(public  
com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float, float))",  
throwing = "Throwable")  
  
public void showExceptionLog(JoinPoint joinPoint, Throwable throwable) {  
    System.out.println("异常通知 -- 异常信息--" + throwable);  
}
```

### 3. 完成测试

```
D:\program\hspjdk8\bin\java.exe ...  
  
前置通知--调用的方法是 getSum--参数是 --[10.0, 11.2]  
getSum() 方法内部打印 result= 21.2  
异常通知 -- 异常信息--java.lang.ArithmetricException: / by zero  
最终通知
```

## 3.11 AOP-环绕通知【了解】

### 3.11.1 应用实例

- 环绕通知可以完成其它四个通知要做的事情

看一个需求：如何使用环绕通知完成其它四个通知的功能。

- 案例演示：在原来的代码上修改即可

### 1. 修改 SmartAnimalAspect.java

```
public class SmartAnimalAspect {  
    //=====环绕通知 start=====  
    //@Around(value = "execution(* *.*(..))")  
    @Around(value="execution(public float com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float, float))")  
    public Object doAround(ProceedingJoinPoint joinPoint) {  
        Object result = null;  
        String methodName = joinPoint.getSignature().getName();  
        try {  
            //1.相当于前置通知完成的事情  
            Object[] args = joinPoint.getArgs();  
            List<Object> argList = Arrays.asList(args);  
            System.out.println("AOP环绕通知--" + methodName + "方法开始了--参数有：" + argList);  
            //在环绕通知中一定要调用joinPoint.proceed()来执行目标方法  
            result = joinPoint.proceed();  
            //2.相当于返回通知完成的事情  
            System.out.println("AOP环绕通知" + methodName + "方法结束了--结果是：" + result);  
        } catch (Throwable throwable) {  
            //3.相当于异常通知完成的事情  
            System.out.println("AOP环绕通知" + methodName + "方法抛异常了--异常对象：" + throwable);  
        } finally {  
            //4.相当于最终通知完成的事情  
            System.out.println("AOP后置通知" + methodName + "方法最终结束了...");  
        }  
        return result;  
    }  
    //=====环绕通知 end=====
```

1. 包 com.hspedu.spring.aop.joinpoint  
2. 增加环绕通知处理  
3. 其它四个通知代码注销

---

```
package com.hspedu.spring.aop.joinpoint;
```

```
import org.aspectj.lang.JoinPoint;  
import org.aspectj.lang.ProceedingJoinPoint;
```

```
import org.aspectj.lang.Signature;
import org.aspectj.lang.annotation.*;
import org.springframework.stereotype.Component;

import java.util.Arrays;
import java.util.List;

/**
 * 使用切面编程来替代原来的动态代理类，机制是一样的。
 * @author Administrator
 */

@Aspect //表示这个类是一个切面类
@Component //需要加入到IOC 容器
public class SmartAnimalAspect {

    //=====环绕通知 start=====
    //@Around(value = "execution(* *.*(..))")
    @Around(value="execution(public
com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float, float))")
    public Object doAround(ProceedingJoinPoint joinPoint) {
        Object result = null;
        String methodName = joinPoint.getSignature().getName();
        float sum = 0;
        if ("getSum".equals(methodName)) {
            float a = (float) joinPoint.getArgs()[0];
            float b = (float) joinPoint.getArgs()[1];
            sum = a + b;
        }
        return result;
    }
}
```

```
try {  
    //1.相当于前置通知完成的事情  
    Object[] args = joinPoint.getArgs();  
    List<Object> argList = Arrays.asList(args);  
    System.out.println("AOP 环绕通知--" + methodName + "方法开始了--参数有：  
" + argList);  
  
    //在环绕通知中一定要调用joinPoint.proceed()来执行目标方法  
    result = joinPoint.proceed();  
  
    //2.相当于返回通知完成的事情  
    System.out.println("AOP 环绕通知" + methodName + "方法结束了--结果是："  
+ result);  
}  
catch (Throwable throwable) {  
    //3.相当于异常通知完成的事情  
    System.out.println("AOP 环绕通知" + methodName + "方法抛异常了--异常对  
象：" + throwable);  
}  
finally {  
    //4.相当于最终通知完成的事情  
    System.out.println("AOP 后置通知" + methodName + "方法最终结束了...");  
}  
return result;  
}  
  
=====环绕通知 end=====
```

```
//===== 注释 start ======
```

```
//这个就对应动态代理类的
```

```
//@Before(value="execution(public
```

```
com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float, float))")
```

```
//@Before(value="execution(* com.hspedu.spring.aop.joinpoint.SmartDog.*(..))")
```

```
// @Before(value = "execution(* *.*(..))")
```

```
// public void showBeginLog(JoinPoint joinPoint) { //前置方法
```

```
//    //得到方法的签名
```

```
//        // 调用前置通知对应的方法签名 : float
```

```
com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float, float)
```

```
//    Signature signature = joinPoint.getSignature();
```

```
//    //得到方法名.
```

```
//    String method_name = signature.getName();
```

```
//    //得到参数
```

```
//    Object[] args = joinPoint.getArgs();
```

```
//    System.out.println("前置通知 JoinPoint joinPoint 火狐" + "--调用的方法是 " +
```

```
method_name + "--参数是 --" + Arrays.asList(args));
```

```
// }
```

```
//这个就对应动态代理类的

//System.out.println(" 日志-- 方法名 : "+methodName+"-- 方法正常结束-- 结果 :
result="+result);

// @AfterReturning(value      =      "execution(public           float
com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float ,float))"

// public void showSuccessEndLog() {
//     System.out.println("返回通知");
// }

/** * returning = "res", Object res 名称保持一致
 * @param joinPoint
 * @param res 调用getSum() 返回的结果
 *
 */
// @AfterReturning(value      =      "execution(public           float
com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float, float)),
//     returning = "res")

// public void showSuccessEndLog(JoinPoint joinPoint, Object res) {
//     System.out.println("返回通知" + "--结果是--" + res );
// }

// }
```

```
//这个就对应动态代理类的

//System.out.println("日志--方法名：" + methodName + "--方法抛出异常--异常类型：
"+e.getClass().getName());

//      @AfterThrowing(value          = "execution(public           float
com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float ,float))")

// public void showExceptionLog() {

//     System.out.println("异常通知");

// }

//      @AfterThrowing(value          = "execution(public           float
com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float, float))",
//                  throwing = "Throwable")
// public void showExceptionLog(JoinPoint joinPoint, Throwable throwable) {
//     System.out.println("异常通知 -- 异常信息--" + throwable);
// }

//这个就对应动态代理类的

//System.out.println("日志--方法名：" + methodName + "--方法最终结束");

//      @After(value          = "execution(public           float
com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float ,float))")

// public void showFinallyEndLog() {

//     System.out.println("最终通知");
}
```

```
//}
```

```
//===== 注释 end =====
```

```
}
```

## 2. 测试 AopJoinPointTest.java

```
AOP环绕通知--getSum方法开始了--参数有: [10.0, 11.2]
```

```
getSum() 方法内部打印 result= 21.2
```

```
AOP环绕通知getSum方法结束了--结果是: 21.2
```

```
AOP后置通知getSum方法最终结束了...
```

## 3. 老韩小结：环绕通知和动态代理完成的事情相似

### 3.12 AOP-切入点表达式重用

#### 3.12.1 应用实例

- 切入点表达式重用

为了统一管理切入点表达式，可以使用切入点表达式重用技术。

- 应用案例：在原来的代码上修改即可

#### 1. 修改 SmartAnimalAspect.java

```
public class SmartAnimalAspect {  
  
    //=====AOP-切入点表达式重用 start ======  
    /*...*/  
    @Pointcut(value="execution(public float com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float, float))")  
    public void myPointCut(){  
    }  
    // @Before(value="execution(public float com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float, float))")  
    @Before(value = "myPointCut()")  
    public void showBeginLog(JoinPoint joinPoint) { //前置方法  
        //得到方法的签名  
        //调用前置通知对应的方法签名: float com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float, float)  
        Signature signature = joinPoint.getSignature();  
        //得到方法名。  
        String method_name = signature.getName();  
        //得到参数  
        Object[] args = joinPoint.getArgs();  
        System.out.println("前置通知" + "--调用的方法是 " + method_name + "--参数是 --" + Arrays.asList(args));  
    }  
    //@After(value = "execution(public float com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float, float))")  
    @After(value = "myPointCut()")  
    public void showFinallyEndLog() {  
        System.out.println("最终通知 -- AOP-切入点表达式重用");  
    }  
}
```

包: com.hspedu.spring.aop.joinpoint  
注销环绕通知，打开4个通知，并做修改

```
package com.hspedu.spring.aop.joinpoint;  
  
import org.aspectj.lang.JoinPoint;  
import org.aspectj.lang.ProceedingJoinPoint;  
import org.aspectj.lang.Signature;  
import org.aspectj.lang.annotation.*;  
import org.springframework.stereotype.Component;  
  
  
import java.util.Arrays;  
import java.util.List;  
  
/**
```

\* 使用切面编程来替代原来的动态代理类，机制是一样的。

\* @author Administrator

\*/

@Aspect //表示这个类是一个切面类

@Component //需要加入到IOC 容器

```
public class SmartAnimalAspect {
```

//=====AOP-切入点表达式重用 start =====

/\*

\* 这样定义的一个切入点表达式，就可以在其它地方直接使用

\*/

@Pointcut(value

=

"execution(public

float

```
com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float, float))")
```

```
public void myPointCut() {
```

```
}
```

//

@Before(value="execution(public

float

```
com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float, float))")
```

@Before(value = "myPointCut()")

```
public void showBeginLog(JoinPoint joinPoint) { //前置方法
```

//得到方法的签名

// 调 用 前 置 通 知 对 应 的 方 法 签 名 : float

```
com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float, float)
```

```
Signature signature = joinPoint.getSignature();  
//得到方法名。  
String method_name = signature.getName();  
//得到参数  
Object[] args = joinPoint.getArgs();  
System.out.println("前置通知" + "--调用的方法是 " + method_name + "--参数是  
--" + Arrays.asList(args));  
}  
  
//@After(value = "execution(public float  
com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float, float))")  
@After(value = "myPointCut()")  
public void showFinallyEndLog() {  
    System.out.println("最终通知 -- AOP-切入点表达式重用");  
}  
  
/**  
 * returning = "res", Object res 名称保持一致  
 * @param joinPoint  
 * @param res 调用 getSum() 返回的结果  
 */
```

```
@AfterReturning(value = "execution(public float com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float, float))", returning = "res")  
  
public void showSuccessEndLog(JoinPoint joinPoint, Object res) {  
    System.out.println("返回通知" + "--结果是--" + res);  
}  
  
@AfterThrowing(value = "execution(public float com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float, float))", throwing = "throwable")  
  
public void showExceptionLog(JoinPoint joinPoint, Throwable throwable) {  
    System.out.println("异常通知 -- 异常信息--" + throwable);  
}  
  
=====AOP-切入点表达式重用 end =====  
  
=====环绕通知 start=====  
  
//@Around(value = "execution(* *.*(..))")  
// @Around(value="execution(public float com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float, float))")  
// public Object doAround(ProceedingJoinPoint joinPoint) {
```

```
//      Object result = null;  
  
//      String methodName = joinPoint.getSignature().getName();  
  
//      try {  
  
//          //1.相当于前置通知完成的事情  
  
//          Object[] args = joinPoint.getArgs();  
  
//          List<Object> argList = Arrays.asList(args);  
  
//          System.out.println("AOP 环绕通知--" + methodName + "方法开始了--参数有："  
+ argList);  
  
//          //在环绕通知中一定要调用joinPoint.proceed()来执行目标方法  
  
//          result = joinPoint.proceed();  
  
//          //2.相当于返回通知完成的事情  
  
//          System.out.println("AOP 环绕通知" + methodName + "方法结束了--结果是：" +  
result);  
  
//      } catch (Throwable throwable) {  
  
//          //3.相当于异常通知完成的事情  
  
//          System.out.println("AOP 环绕通知" + methodName + "方法抛异常了--异常对象："  
+ throwable);  
  
//      } finally {  
  
//          //4.相当于最终通知完成的事情  
  
//          System.out.println("AOP 后置通知" + methodName + "方法最终结束了...");  
  
//      }  
  
//      return result;
```

```
//}  
//=====环绕通知 end=====
```

```
===== 注释 start =====
```

```
//这个就对应动态代理类的  
  
//@Before(value="execution(public  
com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float, float))")  
  
//@Before(value="execution(* com.hspedu.spring.aop.joinpoint.SmartDog.*(..))")  
  
// @Before(value = "execution(* *.*(..))")  
  
// public void showBeginLog(JoinPoint joinPoint) { //前置方法  
  
//     //得到方法的签名  
  
//         // 调用前置通知对应的方法签名 : float  
com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float, float)  
  
//     Signature signature = joinPoint.getSignature();  
  
//     //得到方法名。  
  
//     String method_name = signature.getName();  
  
//     //得到参数  
  
//     Object[] args = joinPoint.getArgs();  
  
//     System.out.println("前置通知 JoinPoint joinPoint 火狐" + "--调用的方法是 " +  
method_name + "--参数是 --" + Arrays.asList(args));
```

```
//}
```

//这个就对应动态代理类的

```
//System.out.println(" 日志-- 方法名：" +methodName+"-- 方法正常结束-- 结果：  
result)+"+result);
```

```
// @AfterReturning(value) = "execution(public  
com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float ,float))"
```

```
// public void showSuccessEndLog() {
```

```
//     System.out.println("返回通知");
```

```
//}
```

```
/**
```

\* returning = "res", Object res 名称保持一致

\* @param joinPoint

\* @param res 调用getSum() 返回的结果

\*

\*/

```
// @AfterReturning(value) = "execution(public  
com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float ,float))",
```

```
//     returning = "res")
```

```
// public void showSuccessEndLog(JoinPoint joinPoint, Object res) {
```

```
//     System.out.println("返回通知" + "--结果是--" + res );
```

```
//  
//}  
  
//这个就对应动态代理类的  
  
//System.out.println("日志--方法名：" + methodName + "--方法抛出异常--异常类型：  
"+e.getClass().getName());  
  
// @AfterThrowing(value = "execution(public float  
com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float ,float))")  
  
// public void showExceptionLog() {  
//     System.out.println("异常通知");  
// }  
  
// @AfterThrowing(value = "execution(public float  
com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float, float)),  
//         throwing = "Throwable")  
  
// public void showExceptionLog(JoinPoint joinPoint, Throwable throwable) {  
//     System.out.println("异常通知 -- 异常信息--" + throwable);  
// }  
  
//这个就对应动态代理类的  
  
//System.out.println("日志--方法名：" + methodName + "--方法最终结束");  
  
// @After(value = "execution(public float  
com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float ,float))")
```

```
// public void showFinallyEndLog() {  
//     System.out.println("最终通知");  
// }  
  
//===== 注释 end =====
```

## 2. 测试 AopJoinPointTest.java

前置通知--调用的方法是 `getSum`--参数是 --[101.0, 11.2]

`getSum()` 方法内部打印 `result= 112.2`

返回通知--结果是--112.2

最终通知 -- AOP-切入点表达式重用

### 3.13 AOP-切面优先级问题

#### 3.13.1 应用实例

- 切面优先级问题:

如果同一个方法，有多个切面在同一个切入点切入，那么执行的优先级如何控制。

- 基本语法:

@order(value=n) 来控制 n 值越小，优先级越高。

- 案例说明

1. 创建  
D:\idea\_java\_projects\spring5\src\com\hspedu\spring\AOP\joinpoint\SmartAnimalAspect2.java

**@Aspect** // 表示这个类是一个切面类  
**@Order(value = 2)**  
**@Component** // 需要加入 IOC 容器

```
public class SmartAnimalAspect2 {  
    /*  
     * 这样定义的一个切入点表达式，就可以在其它地方直接使用  
     */  
    @Pointcut(value = "execution(public float com.hspedu.spring.aop.joinpoint.SmartDog.getSum(float, float))")  
    public void myPointCut() {}  
  
    @Before(value = "myPointCut()")  
    public void showBeginLog(JoinPoint joinPoint) {  
        System.out.println("前置通知 SmartAnimalAspect2 showBeginLog");  
    }  
  
    @AfterReturning(value = "myPointCut()", returning = "res")
```

```
public void showSuccessEndLog(JoinPoint joinPoint, Object res) {  
    System.out.println("返回通知 SmartAnimalAspect2 showSuccessEndLog");  
}  
  
@AfterThrowing(value = "myPointCut()",  
    throwing = "Throwable")  
  
public void showExceptionLog(JoinPoint joinPoint, Throwable throwable) {  
    System.out.println("异常通知 SmartAnimalAspect2 showExceptionLog");  
}  
  
@After(value = "myPointCut")  
  
public void showFinallyEndLog() {  
    System.out.println("最终通知 SmartAnimalAspect2 showFinallyEndLog");  
}  
}
```

2. 修 改

D:\java\_projects\spring5\src\com\hspedu\spring\AOP\joinpoint\SmartAnimalAspect.java

```
@Aspect // 表示这个类是一个切面类  
 @Order(value = 1)  
 @Component // 需要加入IOC容器  
 public class SmartAnimalAspect {
```

3. 测试

```
public class AopJoinPointTest {  
    /** 切面编程 aop:aspectj */  
    @Test  
    public void testAopJoinPoint() {  
        ApplicationContext ioc = new ClassPathXmlApplicationContext("beans04.xml");  
        SmartAnimalable bean = ioc.getBean(SmartAnimalable.class);  
        bean.getSum(10.0f, 11.2f);  
        //bean.getSub(10.0f, 11.2f);  
        System.out.println("-----");  
    }  
}
```

在原来的JUnit测试即可

前置通知--调用的方法是 getSum--参数是 --[10.0, 11.2]

前置通知 SmartAnimalAspect2 showBeginLog

getSum() 方法内部打印 result= 21.2

返回通知 SmartAnimalAspect2 showSuccessEndLog

最终通知 SmartAnimalAspect2 showFinallyEndLog

返回通知--结果是--21.2

最终通知 -- AOP-切入点表达式重用

#### 4. 如何理解输出的信息顺序，类似 Filter 的过滤链式调用机制。（示意图-就比较清楚了。）

##### 3.13.2 注意事项和细节说明

- 不能理解成：优先级高的每个消息通知都先执行，这个和方法调用机制（和 Filter 过滤器链式调用类似）

前置通知--调用的方法是 getSum--参数是 --[10.0, 11.2]

前置通知 SmartAnimalAspect2 showBeginLog

getSum() 方法内部打印 result= 21.2

返回通知 SmartAnimalAspect2 showSuccessEndLog

最终通知 SmartAnimalAspect2 showFinallyEndLog

返回通知--结果是 21.2

最终通知 -- AOP-切入点表达式重用

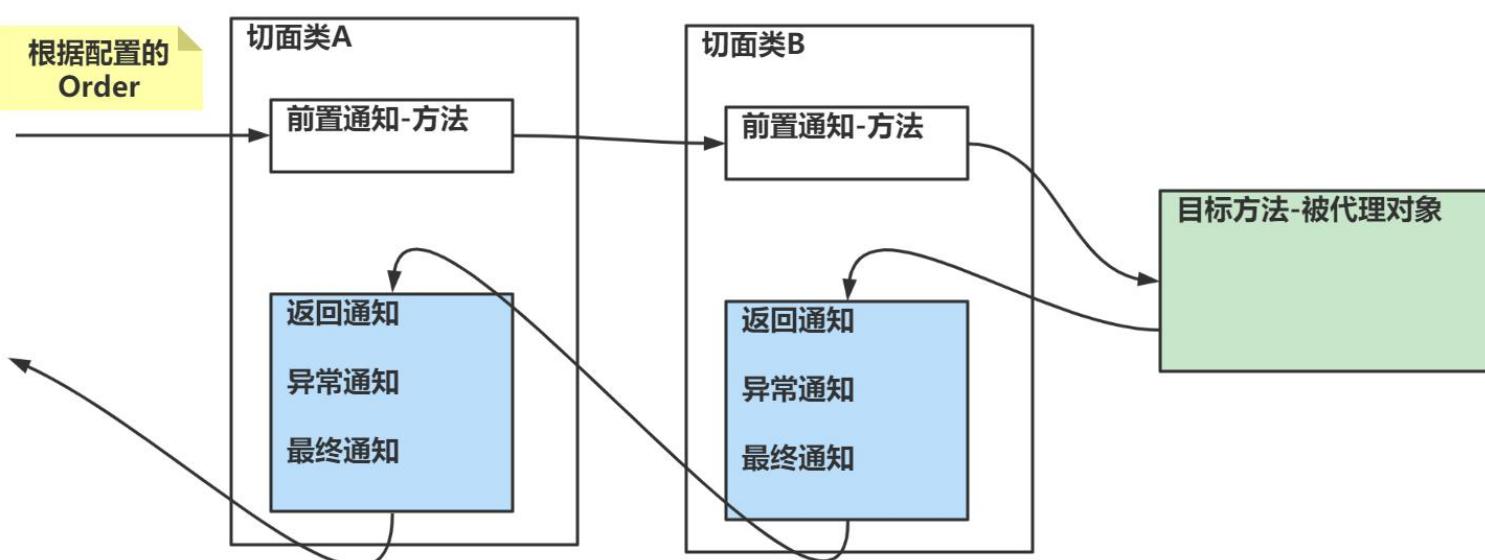
后执行的，  
会被先执行的包裹起来

## 2. 如何理解执行顺序



1. 类似前面学习过的 Filter 链式调用

2. 示意图(画一下)



### 3.14 AOP-基于 XML 配置 AOP

#### 3.14.1 应用实例

- 基本说明:

前面我们是通过注解来配置 aop 的，在 spring 中，我们也可以通过 xml 的方式来配置 AOP

- 应用实例

1. 创建包 `com.hspedu.spring.aop.xml` , `SmartAnimalable.java` 和 `SmartDog.java` 从其它包拷贝即可

2. 创建 `SmartAnimalAspect.java` , 从 `com.hspedu.spring.aop` 包拷贝，并去掉所有注解

```
package com.hspedu.spring.aop.xml;
```

```
import org.aspectj.lang.JoinPoint;  
import org.aspectj.lang.Signature;  
import java.util.Arrays;
```

```
public class SmartAnimalAspect {
```

```
    public void showBeginLog(JoinPoint joinPoint) { //前置通知  
        //得到方法的签名
```

```
Signature signature = joinPoint.getSignature();
//得到方法名。
String method_name = signature.getName();
//得到参数
Object[] args = joinPoint.getArgs();
System.out.println("XML 方式: 前置通知" + "--调用的方法是 " + method_name +
"--参数是 --" + Arrays.asList(args));
}

public void showFinallyEndLog() {
    System.out.println("XML 方式: 最终通知 -- AOP-切入点表达式重用");
}

public void showSuccessEndLog(JoinPoint joinPoint, Object res) {
    System.out.println("XML 方式: 返回通知" + "--结果是--" + res);
}

public void showExceptionLog(JoinPoint joinPoint, Throwable throwable) {
    System.out.println("XML 方式: 异常通知 -- 异常信息--" + throwable);
}
```

### 3. 创建 src\ioc.xml 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"

    xmlns:aop="http://www.springframework.org/schema/aop"

    xsi:schemaLocation="http://www.springframework.org/schema/beans

        http://www.springframework.org/schema/beans/spring-beans.xsd

        http://www.springframework.org/schema/context

        http://www.springframework.org/schema/context/spring-context.xsd

        http://www.springframework.org/schema/aop

        http://www.springframework.org/schema/aop/spring-aop.xsd">

    <!-- 配置自动扫描的包 -->

    <context:component-scan base-package="com.hspedu.spring.aop.xml"/>

    <!-- 开启基于注解的 AOP 功能 -->

    <aop:aspectj-autoproxy/>

    <!-- 配置 SmartAnimalAspect bean -->

    <bean id="smartAnimalAspect"

        class="com.hspedu.spring.aop.xml.SmartAnimalAspect"/>

    <!-- 配置 SmartDog-->

    <bean class="com.hspedu.spring.aop.xml.SmartDog" id="smartDog"/>
```

```
<aop:config>

    <!-- 配置统一切入点 -->

    <aop:pointcut          expression="execution(public           float
com.hspedu.spring.aop.xml.SmartDog.getSum(float, float))"
                                id="myPointCut"/>

    <aop:aspect ref="smartAnimalAspect" order="1">
        <!-- 配置各个通知对应的切入点 -->

        <aop:before method="showBeginLog" pointcut-ref="myPointCut"/>
        <aop:after-returning           method="showSuccessEndLog"
pointcut-ref="myPointCut" returning="res"/>
        <aop:after-throwing           method="showExceptionLog"
pointcut-ref="myPointCut" throwing="Throwable"/>
        <aop:after method="showFinallyEndLog" pointcut-ref="myPointCut"/>
        <!-- 还可以配置环绕通知 -->
        <!-- <aop:around method="" /> -->
    </aop:aspect>

</aop:config>

</beans>
```

## 4. 测 试 , 创 建

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\AopXMLTest.java

```
package com.hspedu.spring.aop.xml;
```

```
import org.junit.jupiter.api.Test;  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
/**  
 * @author 韩顺平  
 * @version 1.0  
 */  
  
public class AopXMLTest {  
    /**  
     * 切面编程 aop:aspectj  
     */  
    @Test  
    public void testAopJoinPoint() {  
        ApplicationContext ioc = new ClassPathXmlApplicationContext("aop_ioc.xml");  
        SmartAnimalable bean = ioc.getBean(SmartAnimalable.class);  
        bean.getSum(10.0f, 11.2f);  
        System.out.println("-----");  
    }  
}
```

### 3.15 AOP-课后作业

#### 3.15.1 作业布置

1. 请编写一个 Cal 接口

方法 cal1(int n) 计算  $1+2+\dots+n$

方法 cal2(int n) 计算  $1 * 2 * \dots * n$

2. 实现类 MyCal implements Cal

3. 请分别使用注解方式 / XML 配置方式 完成 AOP 编程

(1) 在执行 cal1 前打印开始执行的时间，在 执行完后打印时间

(2) 在执行 cal2 前打印开始执行的时间，在 执行完后打印时间

开始执行计算 1644658610720

cal2 res~= 120

结束执行计算 1644658610720

```
开始执行计算 1644658635167  
cal1 res= 10  
结束执行计算 1644658635167|
```

### 3.15.2 作业评讲

#### 1. 参考答案：

##### (1) 创建 D:\hspedu\_ssm\_temp\spring5\src\com\hspedu\spring\homework1\Cal.java

```
package com.hspedu.spring.aop.homework1;
```

```
/**
```

```
* @author 韩顺平
```

```
* @version 1.0
```

```
*/
```

```
public interface Cal {
```

```
    public void cal1(int n);
```

```
    public void cal2(int n);
```

```
}
```

(2)

创

建

```
D:\hspedu_ssm_temp\spring5\src\com\hspedu\spring\homework1\MyCal.java
```

```
package com.hspedu.spring.aop.homework1;
```

```
import org.springframework.stereotype.Component;
```

```
/**  
 * @author 韩顺平  
 * @version 1.0  
 */  
  
@Component  
public class MyCal implements Cal{
```

```
    @Override  
    public void cal1(int n) {  
        int res = 0;  
        for (int i = 0; i <= n; i++) {  
            res += i;  
        }  
        System.out.println("cal1 res= " + res);  
    }
```

```
    @Override  
    public void cal2(int n) {  
        int res = 1;  
        for (int i = 1; i <= n; i++) {
```

```
        res *= i;  
    }  
  
    System.out.println("cal2 res~= " + res);  
}  
}
```

(3)

创

建

D:\hsedu\_ssm\_temp\spring5\src\com\hsedu\spring\homework1\CalAspect.java

```
package com.hsedu.spring.aop.homework1;  
  
import org.aspectj.lang.annotation.AfterReturning;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Before;  
import org.aspectj.lang.annotation.Pointcut;  
import org.springframework.core.annotation.Order;  
import org.springframework.stereotype.Component;  
  
import java.util.Date;  
  
/**  
 * @author 韩顺平  
 * @version 1.0  
 */
```

@Aspect //表示这个类是一个切面类

@Component //需要加入到IOC 容器

```
public class CalAspect {
```

@Pointcut(value = "execution(public void com.hspedu.spring.aop.homework1.MyCal.cal1(int))")

```
    public void myPointCut() {
```

```
}
```

@Before(value = "myPointCut()")

```
    public void startCal() {
```

```
        System.out.println("开始执行计算 " + System.currentTimeMillis());
```

```
}
```

@AfterReturning(value = "myPointCut()")

```
    public void endCal() {
```

```
        System.out.println("结束执行计算 " + System.currentTimeMillis());
```

```
}
```

@Pointcut(value = "execution(public void com.hspedu.spring.aop.homework1.MyCal.cal2(int))")

```
public void myPointCut2() {  
}  
  
@Before(value = "myPointCut2())")  
public void startCal2() {  
    System.out.println("开始执行计算 " + System.currentTimeMillis());  
}  
  
@AfterReturning(value = "myPointCut2())")  
public void endCal2() {  
    System.out.println("结束执行计算 " + System.currentTimeMillis());  
}  
}
```

## (4) D:\hsedu\_ssm\_temp\spring5\src\beans8.xml

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

<https://www.springframework.org/schema/context/spring-context.xsd>

<http://www.springframework.org/schema/aop>

[>](https://www.springframework.org/schema/aop/spring-aop.xsd)

```
<!-- 配置自动扫描的包 -->
<context:component-scan base-package="com.hspedu.spring.aop.homework1"/>
<!-- 开启基于注解的AOP 功能 -->
<aop:aspectj-autoproxy/>
</beans>
```

(5)

创

建

D:\hspedu\_ssm\_temp\spring5\src\com\hspedu\spring\homework1\CalTest.java

```
package com.hspedu.spring.aop.homework1;
```

```
import com.hspedu.spring.aop.springaop.SmartAnimalable;
import org.junit.jupiter.api.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
```

```
/**
```

```
* @author 韩顺平
```

```
* @version 1.0
```

```
*/  
public class CalTest {  
    @Test  
    public void t1() {
```

```
        ApplicationContext ioc = new ClassPathXmlApplicationContext("beans8.xml");  
        //通过结果来  
        Cal bean = ioc.getBean(Cal.class);
```

```
        bean.cal1(5);  
    }  
}
```

## (6) 完成测试

```
开始执行计算 1644658635167  
cal1 res= 10  
结束执行计算 1644658635167
```

## 4 韩顺平 手动实现 Spring 底层机制 【老韩新增】 【初始化 IOC 容器+依赖注入+BeanPostProcessor 机制+AOP】 - 2022 版

### 4.1 先看一个案例，引出对 Spring 底层实现再思考

#### 4.1.1 引言：前面我们实际上已经用代码简单实现了

##### 4.1.1.1 Spring XML 注入 bean

##### 4.1.1.2 Spring 注解方式注入 bean

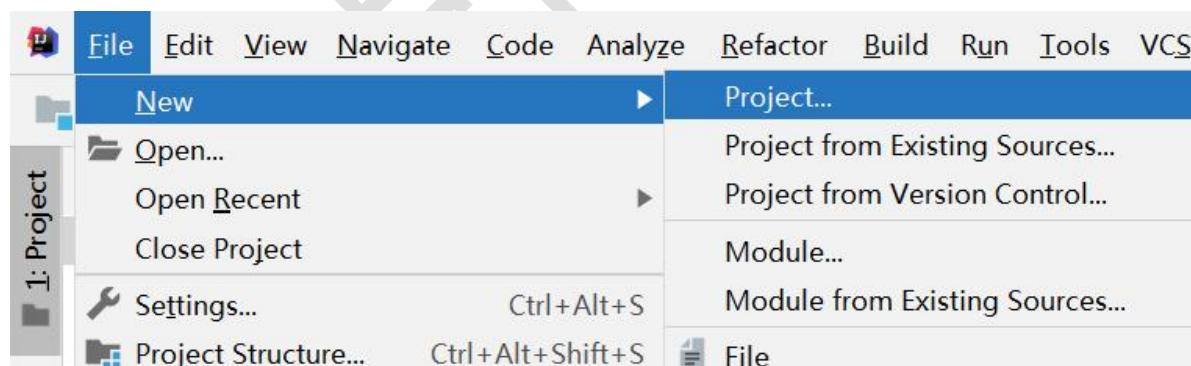
##### 4.1.1.3 Spring AOP 动态代理实现

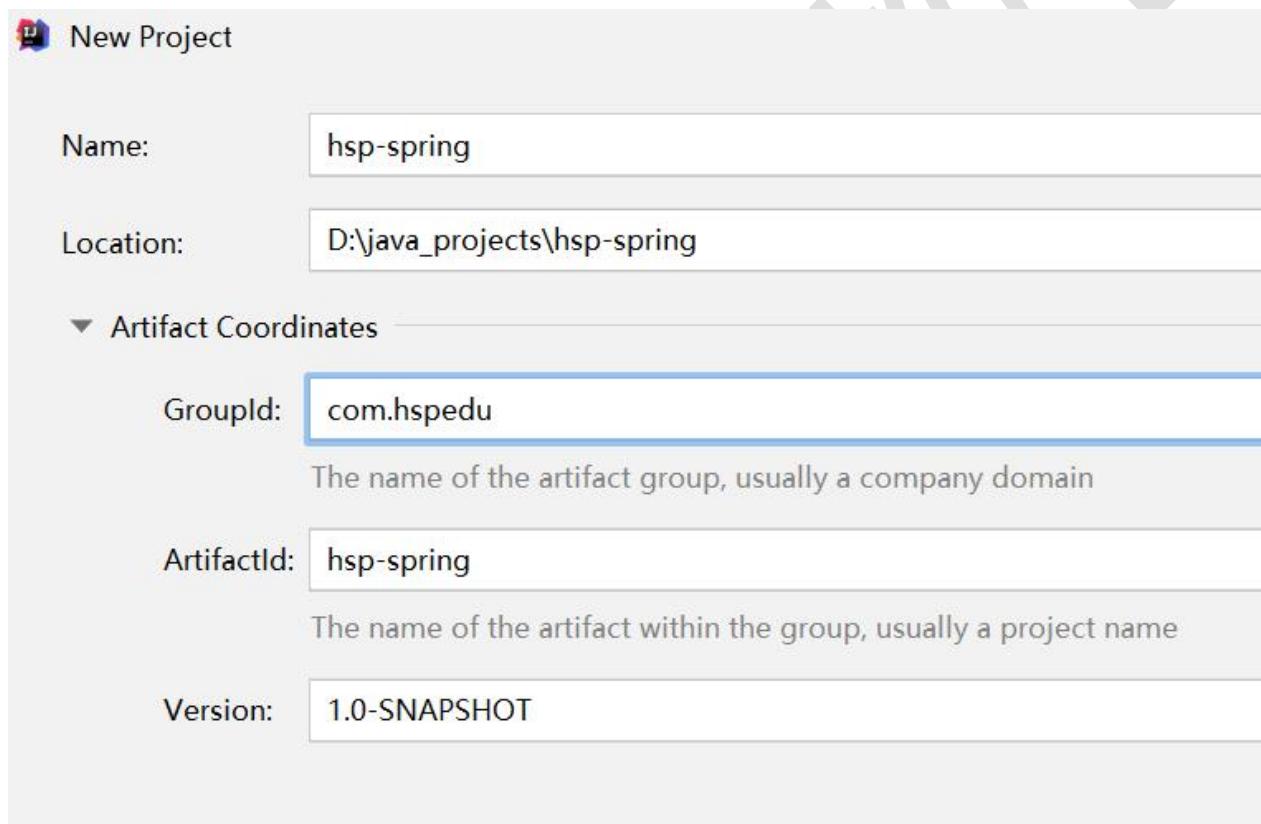
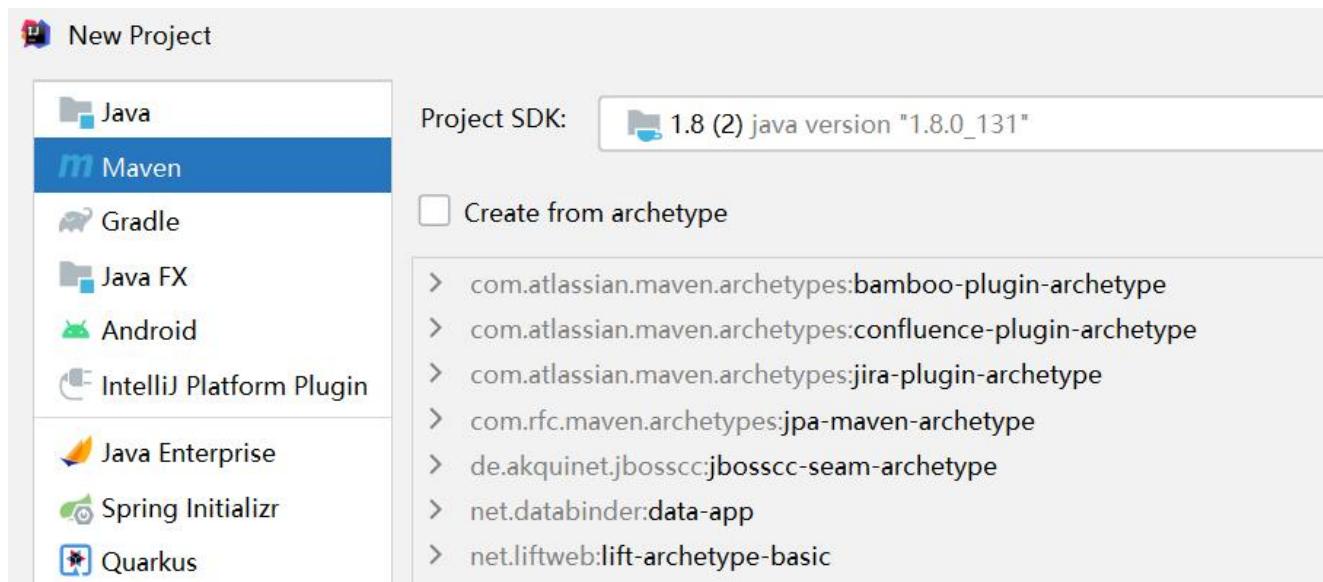
#### 4.1.2 继续思考-原生 Spring 如何实现依赖注入和 singleton、prototype

##### 4.1.2.1 实例演示-Maven 项目

- 快速给小伙伴完成这个小案例(提示：创建新项目，不和原来的 Spring 混在一起)

##### 1. 先创建一个 Maven 的 Java 项目 hsp-spring





## 2. 修改 D:\java\_projects\hsp-spring\pom.xml，引入需要的 jar 包

```
<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.hspedu</groupId>
<artifactId>hsp-spring</artifactId>
<version>1.0-SNAPSHOT</version>

<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.3.8</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-aspects</artifactId>
        <version>5.3.8</version>
    </dependency>
</dependencies>
```

```
</project>
```

3.

创

建

D:\java\_projects\hsp-spring\src\main\java\com\hspedu\spring\component\UserAction.java

a

```
package com.hspedu.spring.component;
```

```
import org.springframework.stereotype.Component;
```

```
import org.springframework.stereotype.Controller;
```

```
/**
```

```
* @author 韩顺平
```

```
* @version 1.0
```

```
*/
```

```
//也可以用 @Controller
```

```
@Component
```

```
//@Scope(value = "prototype")
```

```
public class UserAction {
```

```
}
```

4.

创

建

D:\java\_projects\hsp-spring\src\main\java\com\hspedu\spring\component\UserDao.java

```
package com.hspedu.spring.component;

import org.springframework.stereotype.Component;
import org.springframework.stereotype.Repository;

/**
 * @author 韩顺平
 * @version 1.0
 */
// 也可以用 @Repository
@Component
public class UserDao {
    public void hi() {
        System.out.println("UserDao hi() 被调用...");
    }
}
```

## 5. 建造者模式

D:\java\_projects\hsp-spring\src\main\java\com\hspedu\spring\component\UserService.java

```
package com.hspedu.spring.component;
```

```
import org.springframework.beans.factory.InitializingBean;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Service;

import javax.annotation.PostConstruct;

/**
 * @author 韩顺平
 * @version 1.0
 */
// 也可以用 @Service
@Component
public class UserService {
    // 依赖注入
    @Autowired
    private UserDao userDao;

    public void m1() {
        userDao.hi();
    }
}
```

}

## 6. 创建 D:\java\_projects\hsp-spring\src\main\resources\beans.xml

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

    <!-- 配置自动扫描的包, 注意需要加入 context 名称空间 -->
    <context:component-scan base-package="com.hspedu.spring.component"/>

</beans>
```

## 7. 创建 D:\java\_projects\hsp-spring\src\main\java\com\hspedu\spring\AppMain.java

```
package com.hspedu.spring;

import com.hspedu.spring.aop.SmartAnimalable;
import com.hspedu.spring.aop.SmartDog;
import com.hspedu.spring.component.UserAction;
import com.hspedu.spring.component.UserDao;
import com.hspedu.spring.component.UserService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * @author 韩顺平
 * @version 1.0
 */
public class AppMain {
    public static void main(String[] args) {
        //获取bean 容器/ioc 容器
        ApplicationContext ioc = new ClassPathXmlApplicationContext("beans.xml");
        System.out.println(ioc.getBean("userAction"));
        System.out.println(ioc.getBean("userAction"));
    }
}
```

```
UserDao userDao = (UserDao) ioc.getBean("userDao");
System.out.println(userDao);

UserService userService = (UserService) ioc.getBean("userService");
System.out.println(userService);

//====测试依赖注入=====
userService.m1();

//关闭容器
((ConfigurableApplicationContext) ioc).close();
}

}
```

## 8. 运行效果 如图



```
D:\program\hspjdk8\bin\java.exe ...
com.hspedu.spring.component.UserAction@11c20519
com.hspedu.spring.component.UserAction@11c20519
com.hspedu.spring.component.UserDao@70beb599
com.hspedu.spring.component.UserService@4e41089d
UserDao hi() 被调用...
```

#### 4.1.2.2 思考问题

4.1.2.2.1 Spring 底层实现，如何实现 IOC 容器创建和初始化【前面我们实现过，现在要再深入】

4.1.2.2.2 Spring 底层实现，如何实现 getBean，根据 singleton 和 prototype 来返回 bean 实例

#### 4.1.3 继续思考-原生 Spring 如何实现 BeanPostProcessor

##### 4.1.3.1 实例演示

1. 创建 BeanPostProcessor

D:\java\_projects\hsp-spring\src\main\java\com\hspedu\spring\process\MyBeanPostProcessor.java

```
package com.hspedu.spring.process;

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

/**
 * @author 韩顺平
 * @version 1.0
 */
public class MyBeanPostProcessor implements BeanPostProcessor {
```

```
/**  
 * 在bean 初始化之前完成某些任务  
 * @param bean      : 就是ioc 容器返回的bean 对象, 如果这里被替换会修改, 则返  
 * @param beanName: 就是ioc 容器配置的bean 的名称  
 * @return Object: 就是返回的bean 对象  
 */  
  
public Object postProcessBeforeInitialization(Object bean, String beanName)  
    throws BeansException {  
    // TODO Auto-generated method stub  
    System.out.println("postProcessBeforeInitialization 被调用 " + beanName + "  
bean= " + bean.getClass());  
    return bean;  
}  
  
/**  
 * 在bean 初始化之后完成某些任务  
 * @param bean      : 就是ioc 容器返回的bean 对象, 如果这里被替换会修改, 则返  
 * @param beanName: 就是ioc 容器配置的bean 的名称  
 * @return Object: 就是返回的bean 对象  
 */  
  
public Object postProcessAfterInitialization(Object bean, String beanName)
```

```
    throws BeansException {  
  
        System.out.println("postProcessAfterInitialization 被调用 " + beanName + " bean= "  
        + bean.getClass());  
  
        return bean;  
    }  
}
```

## 2. 修改 D:\java\_projects\hsp-spring\src\main\resources\beans.xml

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

```
<!-- 配置自动扫描的包，注意需要加入 context 名称空间 -->
```

```
<context:component-scan base-package="com.hspedu.spring.component"/>
```

```
<!-- bean 后置处理器的配置 -->  
<bean id="myBeanPostProcessor"  
      class="com.hspedu.spring.process.MyBeanPostProcessor"/>  
  
</beans>
```

3. 修 改  
D:\java\_projects\hsp-spring\src\main\java\com\hspedu\spring\component\UserService.java

```
package com.hspedu.spring.component;  
  
import org.springframework.beans.factory.InitializingBean;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Component;  
import org.springframework.stereotype.Service;  
  
import javax.annotation.PostConstruct;  
  
/**  
 * @author 韩顺平
```

```
* @version 1.0  
*/  
  
// 也可以用 @Service  
  
@Component  
  
public class UserService {  
  
    //依赖注入  
  
    @Autowired  
  
    private UserDao userDao;  
  
  
  
  
    public void m1() {  
        userDao.hi();  
    }  
  
    //通过注解指定在构造器完成后执行的方法，即完成初始化任务  
  
    @PostConstruct  
  
    public void init() {  
        System.out.println("初始化业务...");  
    }  
  
}
```

#### 4. 完成 测试 运行

D:\java\_projects\hsp-spring\src\main\java\com\hspedu\spring\AppMain.java



```
D:\program\hspjdk8\bin\java.exe ...
postProcessBeforeInitialization 被调用 userAction bean= class com.hspedu.spring.component.UserAction@1176dcec
postProcessAfterInitialization 被调用 userAction bean= class com.hspedu.spring.component.UserAction@1176dcec
postProcessBeforeInitialization 被调用 userDao bean= class com.hspedu.spring.component.UserDao@120d6fe6
postProcessAfterInitialization 被调用 userDao bean= class com.hspedu.spring.component.UserDao@120d6fe6
postProcessBeforeInitialization 被调用 userService bean= class com.hspedu.spring.component.UserService@4ba2ca36
初始化业务...
postProcessAfterInitialization 被调用 userService bean= class com.hspedu.spring.component.UserService@4ba2ca36
com.hspedu.spring.component.UserAction@1176dcec
com.hspedu.spring.component.UserAction@1176dcec
com.hspedu.spring.component.UserDao@120d6fe6
com.hspedu.spring.component.UserService@4ba2ca36
UserDao hi() 被调用...
```

#### 4.1.3.2 思考问题

##### 4.1.3.2.1 Spring 底层实现，如何实现 Bean 后置处理器机制

#### 4.1.4 继续思考-原生 Spring 是如何实现 AOP

##### 4.1.4.1 实例演示

1.

创

建

D:\java\_projects\hsp-spring\src\main\java\com\hspedu\spring\AOP\SmartAnimalable.java

```
package com.hspedu.spring.aop;
```

```
/**
```

```
* @author 韩顺平
```

```
* @version 1.0
```

```
 */
```

```
public interface SmartAnimalable {  
    float getSum(float i, float j);  
    float getSub(float i, float j);  
}
```

2.

创

建

D:\java\_projects\hsp-spring\src\main\java\com\hspedu\spring\AOP\SmartDog.java

```
package com.hspedu.spring.aop;
```

```
import org.springframework.stereotype.Component;
```

```
/**
```

```
* @author 韩顺平
```

```
* @version 1.0
```

```
*/
```

```
@Component
```

```
public class SmartDog implements SmartAnimalable {
```

```
    @Override
```

```
    public float getSum(float i, float j) {
```

```
        float result = i + j;
```

```
System.out.println("getSum() 方法内部打印 result= " + result);  
return result;  
}
```

```
@Override  
public float getSub(float i, float j) {  
    float result = i - j;  
    System.out.println("getSub() 方法内部打印 result= " + result);  
    return result;  
}
```

### 3. 创建切面类

D:\java\_projects\hsp-spring\src\main\java\com\hspedu\spring\aspect\SmartAnimalAspect.java

```
package com.hspedu.spring.aspect;  
  
import org.aspectj.lang.annotation.*;  
import org.springframework.stereotype.Component;  
  
/**
```

\* 使用切面编程来替代原来的动态代理类，机制是一样的。

\* @author Administrator

\*/

@Aspect //表示这个类是一个切面类

@Component //需要加入到IOC 容器

public class SmartAnimalAspect {

//这个就对应动态代理类的

//System.out.println(" 日志--方法名：" +methodName+"--方法开始--参数：" +Arrays.asList(args));

@Before(value = "execution(public com.hspedu.spring.aop.SmartDog.getSum(float ,float))") float

public static void showBeginLog() {

System.out.println("前置通知");

}

//这个就对应动态代理类的

//System.out.println(" 日志--方法名：" +methodName+"--方法正常结束--结果：" +result);

@AfterReturning(value = "execution(public com.hspedu.spring.aop.SmartDog.getSum(float ,float))") float

public void showSuccessEndLog() {

```
System.out.println("返回通知");  
}  
  
//这个就对应动态代理类的  
//System.out.println("日志--方法名：" + methodName + "--方法抛出异常--异常类型：  
//+e.getClass().getName());  
  
@AfterThrowing(value = "execution(public  
com.hspedu.spring.aop.SmartDog.getSum(float ,float))")  
  
public void showExceptionLog() {  
    System.out.println("异常通知");  
}  
  
//这个就对应动态代理类的  
//System.out.println("日志--方法名：" + methodName + "--方法最终结束");  
  
@After(value = "execution(public  
com.hspedu.spring.aop.SmartDog.getSum(float ,float))")  
  
public void showFinallyEndLog() {  
    System.out.println("最终通知");  
}  
}
```

#### 4. 修改 D:\java\_projects\hsp-spring\src\main\java\com\hspedu\spring\AppMain.java

```
package com.hspedu.spring;
```

```
import com.hspedu.spring.aop.SmartAnimalable;
import com.hspedu.spring.aop.SmartDog;
import com.hspedu.spring.component.UserAction;
import com.hspedu.spring.component.UserDao;
import com.hspedu.spring.component.UserService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.annotation.EnableAspectJAutoProxy;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * @author 韩顺平
 * @version 1.0
 */

public class AppMain {
    public static void main(String[] args) {
        //获取bean 容器/ioc 容器
        ApplicationContext ioc = new ClassPathXmlApplicationContext("beans.xml");
        System.out.println(ioc.getBean("userAction"));
    }
}
```

```
System.out.println(ioc.getBean("userAction"));
```

```
UserDao userDao = (UserDao) ioc.getBean("userDao");
```

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

```
UserService userService = (UserService) ioc.getBean("userService");
```

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

```
=====测试依赖注入=====
```

```
userService.m1();
```

```
=====测试 aop=====
```

```
SmartAnimalable smartDog = (SmartAnimalable) ioc.getBean("smartDog");
```

```
smartDog.getSum(10, 30);
```

```
//关闭容器
```

```
((ConfigurableApplicationContext) ioc).close();
```

```
}
```

```
}
```

## 5. 修改 D:\java\_projects\hsp-spring\src\main\resources\beans.xml

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

    <!-- 配置自动扫描的包，注意需要加入 context 名称空间 -->
    <context:component-scan base-package="com.hspedu.spring.component"/>
    <context:component-scan base-package="com.hspedu.spring.aop"/>

    <!-- bean 后置处理器的配置 -->
    <bean id="myBeanPostProcessor"
          class="com.hspedu.spring.process.MyBeanPostProcessor"/>

    <!-- 开启基于注解的 AOP 功能 -->
    <aop:aspectj-autoproxy/>
```

```
</beans>
```

## 6. 完成测试，看输出效果

```
postProcessAfterInitialization 仅仅对单例bean生效  
com.hspedu.spring.component.UserAction@776aec5c  
com.hspedu.spring.component.UserAction@776aec5c  
com.hspedu.spring.component.UserDao@1d296da  
com.hspedu.spring.component.UserService@7c7a06ec  
UserDao hi() 被调用...
```

前置通知

getSum() 方法内部打印 result= 40.0

返回通知

最终通知

### 4.1.4.2 简单分析 AOP 和 BeanPostProcessor 关系

1. AOP 实现 Spring 可以通过给一个类，加入注解 `@EnableAspectJAutoProxy` 来指定，比如

```
*/  
@EnableAspectJAutoProxy  
public class Test {  
}
```

2. 我们来追一下`@EnableAspectJAutoProxy`

```
/*
 * Target(ElementType.TYPE)
 * Retention(RetentionPolicy.RUNTIME)
 * Documented
 * Import(AspectJAutoProxyRegistrar.class)
 * interface EnableAspectJAutoProxy {
 *
 *     /**
 *      * Indicate whether subclass-based (CGLIB) proxies
 *      * should be used for proxying target objects.
 *      */
 *
 *     class AspectJAutoProxyRegistrar implements ImportBeanDefinitionRegistrar {
 *
 *         /**
 *          * Register, escalate, and configure the AspectJ auto proxy creator based on the
 *          * of the {@link EnableAspectJAutoProxy#proxyTargetClass()} attribute on the import
 *          * {@code @Configuration} class.
 *         */
 *
 *         @Override
 *         public void registerBeanDefinitions(
 *             AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry
 *         ) {
 *             AopConfigUtils.registerAspectJAnnotationAutoProxyCreatorIfNecessary(registry);
 *
 *             AnnotationAttributes enableAspectJAutoProxy =
 *
 *             @Nullable
 *             public static BeanDefinition registerAspectJAnnotationAutoProxyCreatorIfNecessary(Bea
 *                 return registerAspectJAnnotationAutoProxyCreatorIfNecessary(registry, null);
 *             }
 *         }
 *     }
 * }
```

```
/*
@Override
public void registerBeanDefinitions(
    AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry)
    AopConfigUtils.registerAspectJAnnotationAutoProxyCreatorIfNecessary(registry);

AnnotationAttributes enableAspectJAutoProxy =
    AnnotationConfigUtils.attributesFor(importingClassMetadata, EnableAspectJAutoProxy.class);
if (enableAspectJAutoProxy != null) {
    if (enableAspectJAutoProxy.getBoolean("proxyTargetClass")) {
        AopConfigUtils.forceAutoProxyCreatorToUseClassProxying(registry);
    }
    if (enableAspectJAutoProxy.getBoolean("exposeProxy")) {
        AopConfigUtils.forceAutoProxyCreatorToExposeProxy(registry);
    }
}
}
```



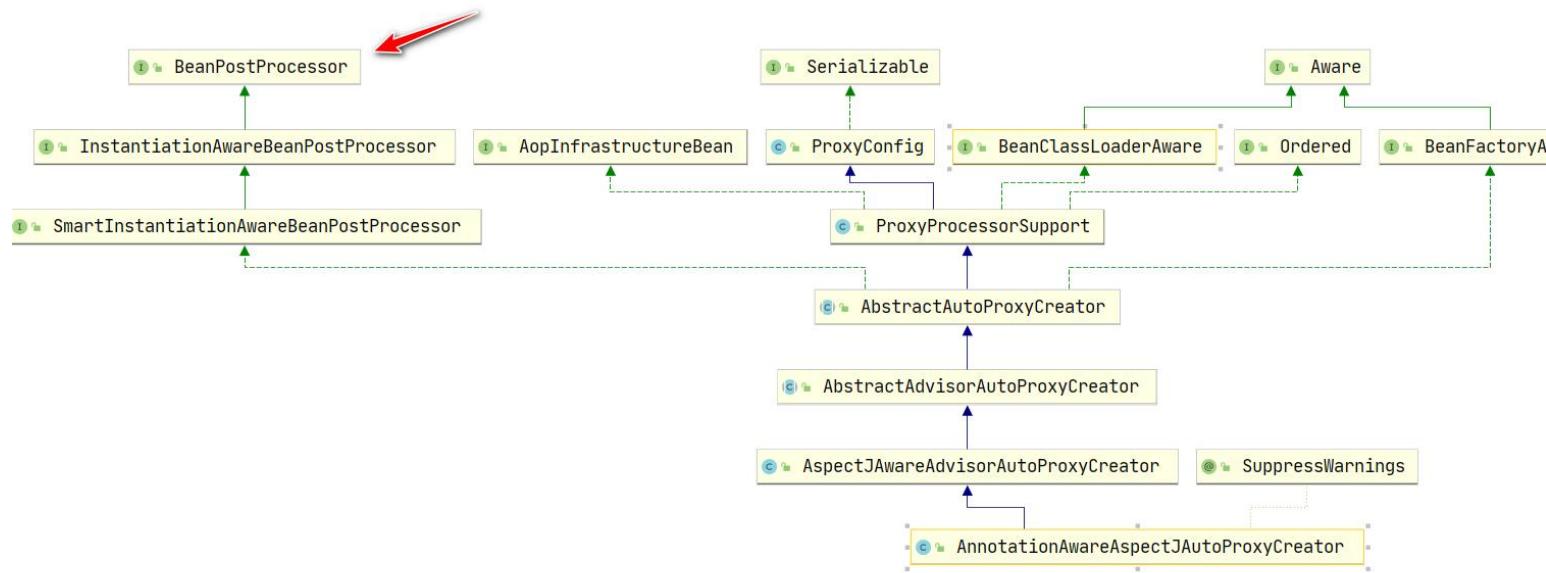
```
@Nullable
public static BeanDefinition registerAspectJAnnotationAutoProxyCreatorIfNecessary(BeanDefinitionRegistry registry) {
    return registerAspectJAnnotationAutoProxyCreatorIfNecessary(registry, null);
}
```



```
@Nullable
public static BeanDefinition registerAspectJAnnotationAutoProxyCreatorIfNecessary(
    BeanDefinitionRegistry registry, @Nullable Object source) {
    return registerOrEscalateApcAsRequired(AnnotationAwareAspectJAutoProxyCreator.class);
}
```

最后返回的是 AnnotationAwareAspectJAutoProxyCreator

### 3. 看一下 AnnotationAwareAspectJAutoProxyCreator 的类图



### 4. 老韩解读

- 1) AOP 底层是基于 `BeanPostProcessor` 机制的.
- 2) 即在 Bean 创建好后，根据是否需要 AOP 处理，决定返回代理对象，还是原生 Bean
- 3) 在返回代理对象时，就可以根据要代理的类和方法来返回
- 4) 其实这个机制并不难，本质就是在 `BeanPostProcessor` 机制 + 动态代理技术
- 5) 下面我们就准备自己来实现 AOP 机制，这样小伙伴们就不在觉得 AOP 神秘，通透很多了.

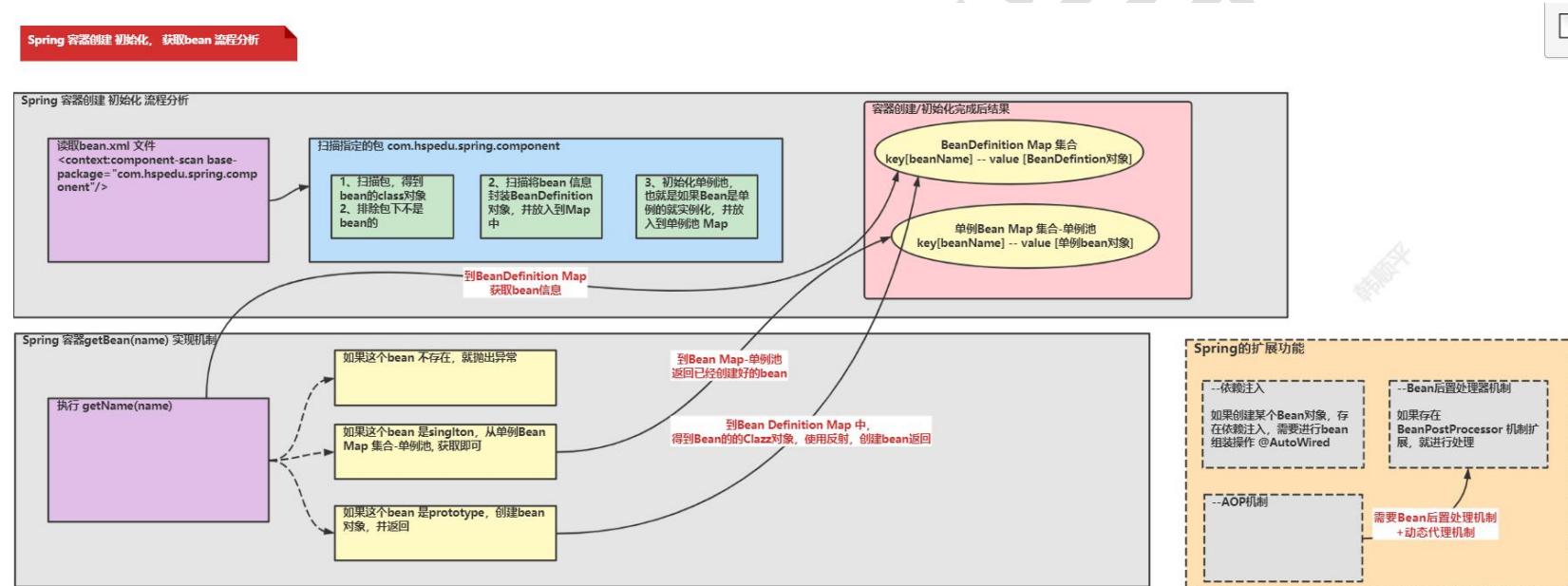
#### 4.1.4.3 思考问题

##### 4.1.4.3.1 Spring 底层实现，如何实现 AOP 编程

#### 4.1.5 我们的目标：不用 Spring 框架，模拟 Spring 底层实现，也能完成相同的功能

### 4.2 Spring 整体架构分析

#### 4.2.1 一图胜千言



#### 4.3 手动实现 Spring 底层机制【初始化 IOC 容器+依赖注入+BeanPostProcessor 机制+AOP】

##### 4.3.1 实现任务阶段 1- 编写自己 Spring 容器，实现扫描包，得到 bean 的 class 对象

###### 4.3.1.1 知识扩展：类加载器

- java 的类加载器 3 种

Bootstrap 类加载器-----对应路径 jre/lib

Ext 类加载器-----对应路径 jre/lib/ext

App 类加载器-----对应路径 classpath

- classpath 类路径，就是 java.exe 执行时，指定的路径，比如

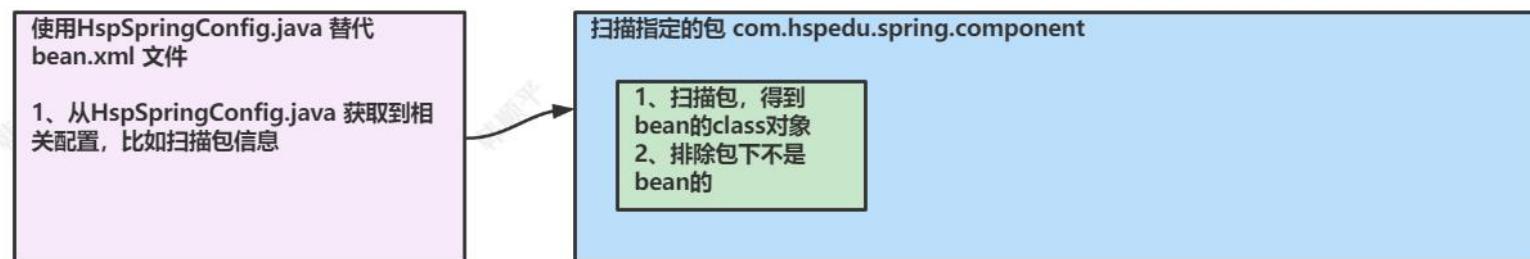
```
e\lib\jfxswt.jar;D:\program\hspjdk8\jre\lib\jsse.jar;D:\program\hspjdk8\jre\lib\management-agent.jar;D:\p  
rogram\hspjdk8\jre\lib\plugin.jar;D:\program\hspjdk8\jre\lib\resources.jar;D:\program\hspjdk8\jre\lib\rt.j  
ar;D:\java_projects\hsp-myspring\target\classes com.hspedu.spring.AppMain
```

#### 4.3.1.2 说明：编写自己 Spring 容器，实现扫描包，得到 bean 的 class 对象

#### 4.3.1.3 分析+代码实现+测试

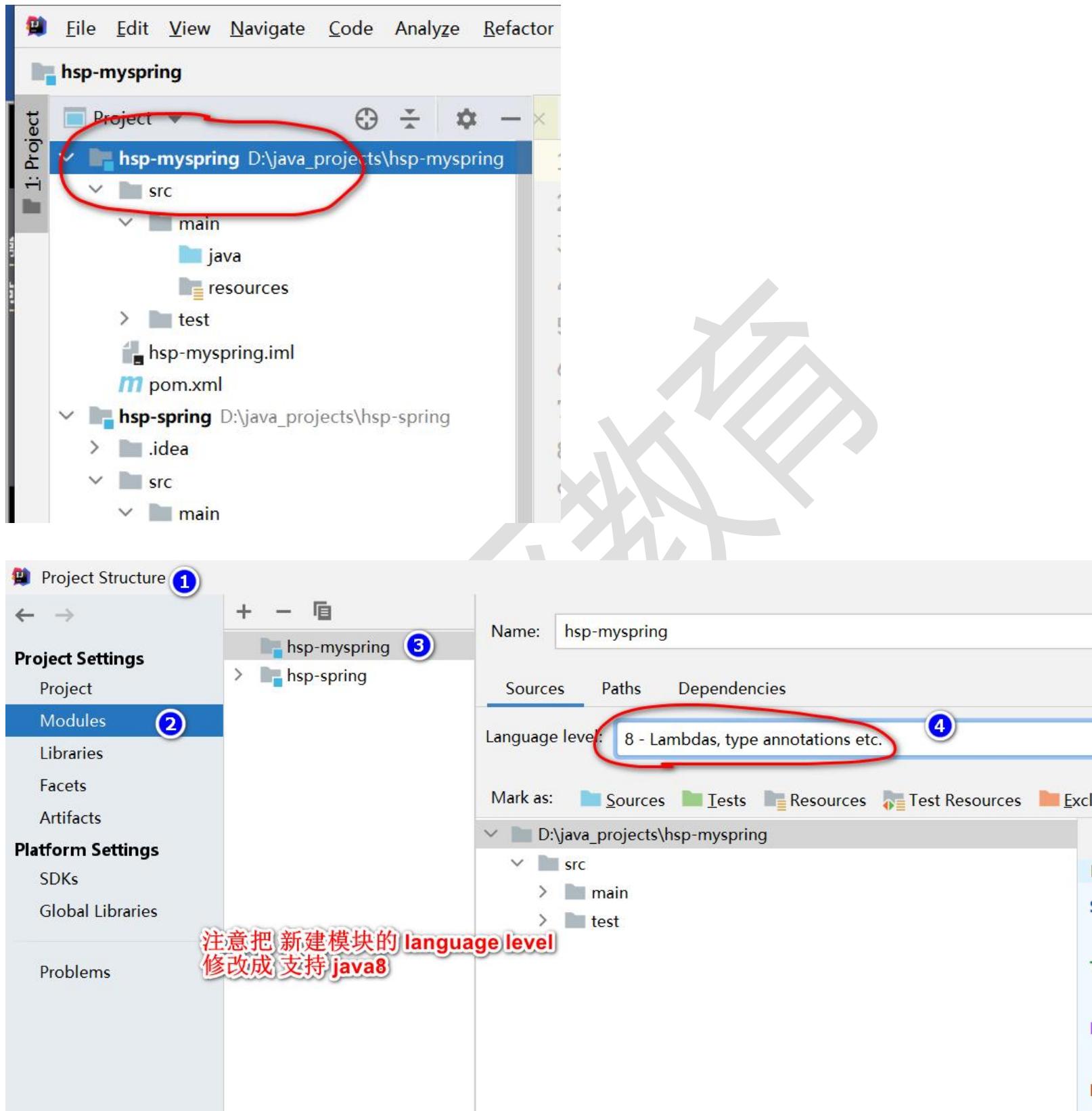
- 分析示意图

老韩的思路，化繁为简-任务拆解  
实现任务阶段1- 编写自己Spring容器，实现扫描包，得到bean的class对象



- 代码实现

- 为了方便讲解, 创建 子模块 (java maven module), 这个具体步骤, 我们讲解 Tomcat 时, 讲过, 只是这里注意创建的是 java maven 模块即可



2.

创

建

D:\java\_projects2\hsp-spring\src\main\java\com\hsperedu\spring\annotation\ComponentSc

**an.java**

```
package com.hspedu.spring.annotation;

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

/**
 * @author 韩顺平
 * @version 1.0
 * 定义我们的 ComponentScan 注解
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface ComponentScan {
    String value();
}
```

3.

创

建

D:\java\_projects2\hsp-spring\src\main\java\com\hspedu\spring\annotation\Component.java

```
package com.hspedu.spring.annotation;

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

/**
 * @author 韩顺平
 * @version 1.0
 * 定义我们的 Component 注解
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Component {
    // @Component 可以传入属性值，这里我们默认为 ""
    String value() default "";
}
```

4.

创

建

D:\java\_projects2\hsp-spring\src\main\java\com\hspedu\spring\ioc\HspSpringConfig.java

```
package com.hspedu.spring.ioc;

import com.hspedu.spring.annotation.ComponentScan;

/**
 * @author 韩顺平
 * @version 1.0
 * 作用类似我们的beans.xml 文件，用于对spring 容器指定配置信息
 */

//指定要扫描的包
@ComponentScan("com.hspedu.spring.component")
public class HspSpringConfig {
```

5.

创

建

D:\java\_projects2\hsp-spring\src\main\java\com\hspedu\spring\component\MonsterService.java

```
package com.hspedu.spring.component;
```

```
import com.hspedu.spring.annotation.Autowired;
```

```
import com.hspedu.spring.annotation.Before;
import com.hspedu.spring.annotation.Component;
import com.hspedu.spring.annotation.Scope;
import com.hspedu.spring.processe.InitializingBean;

/**
 * @author 韩顺平
 * @version 1.0
 */

/**
 * 1. 使用@Component("monsterService") 修饰
 * 2. 给该 MonsterService 注入到容器设置 beanName 为 monsterService
 * 3. 如果没有设置，默认可以以类名 首字母小写来玩(底层还有很多细节)
 */
@Component("monsterService")
public class MonsterService {

}
```

6.

创

建

D:\java\_projects2\hsp-spring\src\main\java\com\hspedu\spring\component\MonsterDao.java

```
package com.hspedu.spring.component;

import com.hspedu.spring.annotation.Component;

/**
 * @author 韩顺平
 * @version 1.0
 */
@Component("monsterDao")
public class MonsterDao {
```

7.

建

D:\java\_projects2\hsp-spring\src\main\java\com\hspedu\spring\ioc\HspSpringApplication

**Context.java 充当容器类**

```
package com.hspedu.spring.ioc;
```

```
import com.hspedu.spring.annotation.Component;
```

```
import com.hspedu.spring.annotation.ComponentScan;
```

```
import java.io.File;
```

```
import java.lang.annotation.Annotation;  
  
import java.net.URL;  
  
/**  
 * @author 韩顺平  
 * @version 1.0  
 */  
  
public class HspSpringApplicationContext {  
  
    private Class configClass;  
  
    public HspSpringApplicationContext(Class configClass) {  
  
        this.configClass = configClass;  
  
        //1. 解析配置类  
  
        //2. 获取到配置类的 @ComponentScan("com.hspedu.spring.component")  
  
        ComponentScan componentScan = (ComponentScan)  
            this.configClass.getDeclaredAnnotation(ComponentScan.class);
```

```
String path = componentScan.value();
```

```
System.out.println("扫描路径 = " + path);
```

//3. 获取扫描路径下所有的类文件

//(1) 先得到类加载器, 使用 App 方式来加载.

```
ClassLoader classLoader = HspSpringApplicationContext.class.getClassLoader();
```

//在获取某个包的 d 对应的 URL 时, 要求是 com/hspedu/spring/component

```
//URL resource = classLoader.getResource("com/hspedu/spring/component");
```

//(2) 将 path 转成 形式为 com/hspedu/spring/component

```
path = path.replace(".", "/");
```

```
URL resource = classLoader.getResource(path);
```

```
File file = new File(resource.getFile());
```

```
if(file.isDirectory()) {
```

```
    File[] files = file.listFiles();
```

```
    for (File f : files) {
```

```
        String fileAbsolutePath = f.getAbsolutePath();
```

```
System.out.println("=====");  
  
System.out.println("文件绝对路径 = " + fileAbsolutePath);  
  
if(fileAbsolutePath.endsWith(".class")) {//说明是类文件  
  
    //通过类加载器获取来类文件的 Clazz 对象  
  
    // 先 得 到 类 的 完 整 类 路 径      形 式 为  
com.hspedu.spring.component.MonsterService  
  
    String className =  
  
        fileAbsolutePath.substring(fileAbsolutePath.lastIndexOf("\\")  
+ 1, fileAbsolutePath.indexOf(".class"));  
  
    String classFullPath = path.replace("/", ".") + "." + className;  
  
    System.out.println("类名 = " + className);  
  
    System.out.println("类的全路径 = " + classFullPath);  
  
    try {  
  
        //获取到扫描包下的类的 clazz 对象  
  
        Class<?> clazz = classLoader.loadClass(classFullPath);  
    }  
}
```

```
if(clazz.isAnnotationPresent(Component.class)) {  
  
    //如果这个类有@Component, 说明是一个 spring bean  
  
    System.out.println("是一个 bean = " + clazz);  
  
} else {  
  
    //如果这个类没有@Component, 说明不是一个 spring bean  
  
    System.out.println("不是一个 bean = " + clazz);  
  
}  
  
} catch (ClassNotFoundException e) {  
  
    e.printStackTrace();  
  
}  
  
System.out.println("=====");  
  
}  
  
}  
  
}
```

```
public Object getBean(String name) {  
  
    return null;  
  
}  
  
}
```

## 8. 创建 D:\java\_projects2\hsp-spring\src\main\java\com\hspedu\spring\AppMain.java

```
package com.hspedu.spring;  
  
  
import com.hspedu.spring.component.MonsterService;  
import com.hspedu.spring.component.SmartAnimalable;  
import com.hspedu.spring.component.SmartDog;  
import com.hspedu.spring.ioc.HspSpringApplicationContext;  
import com.hspedu.spring.ioc.HspSpringConfig;  
  
/**  
 * @author 韩顺平  
 * @version 1.0  
 */  
  
public class AppMain {  
    public static void main(String[] args) {
```

```
//创建我们的spring 容器对象  
HspSpringApplicationContext hspSpringApplicationContext =  
    new HspSpringApplicationContext(HspSpringConfig.class);  
}  
}
```

## 9. 完成测试，输出效果

```
D:\program\hspjdk8\bin\java.exe ...  
扫描路径 = com.hspedu.spring.component  
=====  
文件绝对路径 = D:\java_projects\hsp-myspring\target\classes\com\hspedu\spring\component  
类名 = MonsterDao  
类的全路径 = com.hspedu.spring.component.MonsterDao  
是一个bean = class com.hspedu.spring.component.MonsterDao  
=====  
=====  
文件绝对路径 = D:\java_projects\hsp-myspring\target\classes\com\hspedu\spring\component  
类名 = MonsterService  
类的全路径 = com.hspedu.spring.component.MonsterService  
是一个bean = class com.hspedu.spring.component.MonsterService  
=====
```

### 4.3.2 实现任务阶段 2- 扫描将 bean 信息封装到 BeanDefinition 对象，并放入到 Map

#### 4.3.2.1 说明：扫描将 bean 信息封装到 BeanDefinition 对象，并放入到 Map

#### 4.3.2.2 分析+代码实现+测试

- 分析示意图



- 代码实现

1.

创

建

D:\java\_projects2\hsp-spring\src\main\java\com\hsppedu\spring\annotation\Scope.java

```
package com.hsppedu.spring.annotation;
```

```
import java.lang.annotation.ElementType;
```

```
import java.lang.annotation.Retention;
```

```
import java.lang.annotation.RetentionPolicy;
```

```
import java.lang.annotation.Target;
```

```
/*
 * @author 韩顺平
 * @version 1.0
 * 定义我们的 Scope 注解
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Scope {
    String value(); //不需要给默认值
}
```

## 2. 修 改 , 增 加 @Scope , 测 试 用

D:\java\_projects2\hsp-spring\src\main\java\com\hspedu\spring\component\MonsterService.java

```
@Component("monsterService")
@Scope("prototype")
public class MonsterService{}
```

## 3. 创 建

D:\java\_projects2\hsp-spring\src\main\java\com\hspedu\spring\ioc\BeanDefinition.java

```
package com.hspedu.spring.ioc;
```

```
/*
 * @author 韩顺平
 * @version 1.0
 */

/**
 * 1. 在扫描时，将bean 信息，封装到beandefinition 对象中
 * 2. 然后将beandefinition 对象，放入到spring 容器的集合中
 */

public class BeanDefinition {
    private Class clazz; //bean 对应的 Class 对象
    private String scope; //bean 的作用域 singleton/prototype

    public Class getClazz() {
        return clazz;
    }

    public void setClazz(Class clazz) {
        this.clazz = clazz;
    }

    public String getScope() {
```

```
        return scope;  
    }  
  
    public void setScope(String scope) {  
        this.scope = scope;  
    }  
  
    @Override  
    public String toString() {  
        return "BeanDefinition{" +  
            "clazz=" + clazz +  
            ", scope='" + scope + '\'' +  
            '}';  
    }  
}
```

4. 修 改  
D:\java\_projects2\hsp-spring\src\main\java\com\hspedu\spring\ioc\HspSpringApplicationContext.java, 将扫描到的 Bean 信息封装到 BeanDefinition 对象中，并保存到 Map

```
package com.hspedu.spring.ioc;  
  
import com.hspedu.spring.annotation.Component;
```

```
import com.hspedu.spring.annotation.ComponentScan;  
  
import com.hspedu.spring.annotation.Scope;  
  
  
import java.io.File;  
  
import java.lang.annotation.Annotation;  
  
import java.net.URL;  
  
import java.util.concurrent.ConcurrentHashMap;  
  
/**  
 * @author 韩顺平  
 * @version 1.0  
 */  
  
public class HspSpringApplicationContext {  
  
    private Class configClass;  
  
    //如果 bean 是单例的，就直接放在这个 单例 bean 对象池  
  
    private ConcurrentHashMap<String, Object> singletonObjects =
```

```
new ConcurrentHashMap<>();  
  
//将 bean 的定义，放在这个 beanDefinitionMap 集合  
  
private ConcurrentHashMap<String,BeanDefinition> beanDefinitionMap =  
  
    new ConcurrentHashMap<>();  
  
public HspSpringApplicationContext(Class configClass) {  
  
    //通过扫描，得到 beanDefinition 的 map  
  
    beanDefinitionsByScan(configClass);  
  
    System.out.println(beanDefinitionMap);  
}  
  
private void beanDefinitionsByScan(Class configClass) {  
  
    this.configClass = configClass;  
  
    //1. 解析配置类  
  
    //2. 获取到配置类的 @ComponentScan("com.hspedu.spring.component")
```

ComponentScan                    componentScan                    =                            (ComponentScan)

```
this.configClass.getDeclaredAnnotation(ComponentScan.class);
```

```
String path = componentScan.value();
```

```
System.out.println("扫描路径 = " + path);
```

//3. 获取扫描路径下所有的类文件

//(1) 先得到类加载器，使用 App 方式来加载。

```
ClassLoader classLoader = HspSpringApplicationContext.class.getClassLoader();
```

//在获取某个包的 d 对应的 URL 时，要求是 com/hspedu/spring/component

```
//URL resource = classLoader.getResource("com/hspedu/spring/component");
```

//(2) 将 path 转成 形式为 com/hspedu/spring/component

```
path = path.replace(".", "/");
```

```
URL resource = classLoader.getResource(path);
```

```
File file = new File(resource.getFile());
```

```
if(file.isDirectory()) {
```

```
    File[] files = file.listFiles();
```

```
for (File f : files) {  
  
    String fileAbsolutePath = f.getAbsolutePath();  
  
    System.out.println("=====");  
  
    System.out.println("文件绝对路径 = " + fileAbsolutePath);  
  
    if(fileAbsolutePath.endsWith(".class")) {//说明是类文件  
  
        //通过类加载器获取来类文件的 Clazz 对象  
  
        // 先 得 到 类 的 完 整 类 路 径      形 式 为  
com.hspedu.spring.component.MonsterService  
  
        String className =  
            fileAbsolutePath.substring(fileAbsolutePath.lastIndexOf("\\")  
+ 1, fileAbsolutePath.indexOf(".class"));  
  
        String classFullPath = path.replace("/", ".") + "." + className;  
  
        System.out.println("类名 = " + className);  
  
        System.out.println("类的全路径 = " + classFullPath);  
  
        try {
```

```
//获取到扫描包下的类的 clazz 对象  
  
Class<?> clazz = classLoader.loadClass(classFullPath);  
  
if(clazz.isAnnotationPresent(Component.class)) {  
  
    //如果这个类有@Component, 说明是一个 spring bean
```

System.out.println("是一个 bean = " + clazz);

### //老韩解读

- //1. 因为这里不能直接将 bean 实例放入 singletonObjects
- //2. 原因是如果 bean 是 prototype 是需要每次创建新的 bean
- 对象
- //3. 所以, Spring 底层是这样设计的: 将 bean 信息封装到 BeanDefinition 对象中, 便于 getBean 的操作

BeanDefinition beanDefinition = new BeanDefinition();

beanDefinition.setClazz(clazz);

//获取 bean 的 name

Component

componentAnnotation

=

clazz.getDeclaredAnnotation(Component.class);

```
String beanName = componentAnnotation.value();  
  
//获取 bean 的 scope  
  
if(clazz.isAnnotationPresent(Scope.class)) { // 如果有  
@Scope  
  
    Scope scope = clazz.getDeclaredAnnotation(Scope.class);  
  
    scopeAnnotation = scope;  
  
    beanDefinition.setScope(scopeAnnotation.value());  
  
} else { //如果没有@Scope, 默认是 singleton  
  
    beanDefinition.setScope("singleton");  
  
}  
  
//放入到 beanDefinitionMap  
  
beanDefinitionMap.put(beanName, beanDefinition);  
  
}  
  
}  
  
//如果这个类没有@Component, 说明不是一个 spring bean  
  
System.out.println("不是一个 bean = " + clazz);
```

```
    }
```

```
        }
```

```
    } catch (ClassNotFoundException e) {
```

```
        e.printStackTrace();
```

```
    }
```

```
    System.out.println("=====");
```

```
    }
```

```
}
```

```
}
```

```
public Object getBean(String name) {
```

```
    return null;
```

```
}
```

```
}
```

## 5. 运行 D:\java\_projects\hsp-myspring\src\main\java\com\hspedu\spring\AppMain.java, 完成测试

```
AppMain (1) ×
D:\program\hspjdk8\bin\java.exe ...
扫描路径 = com.hspedu.spring.component
=====
文件绝对路径 = D:\java_projects\hsp-myspring\target\classes\com\hspedu\spring\component\M
类名 = MonsterDao
类的全路径 = com.hspedu.spring.component.MonsterDao
是一个bean = class com.hspedu.spring.component.MonsterDao
=====
文件绝对路径 = D:\java_projects\hsp-myspring\target\classes\com\hspedu\spring\component\M
类名 = MonsterService
类的全路径 = com.hspedu.spring.component.MonsterService
是一个bean = class com.hspedu.spring.component.MonsterService
=====
{monsterService=BeanDefinition{clazz=class com.hspedu.spring.component.MonsterService,
```

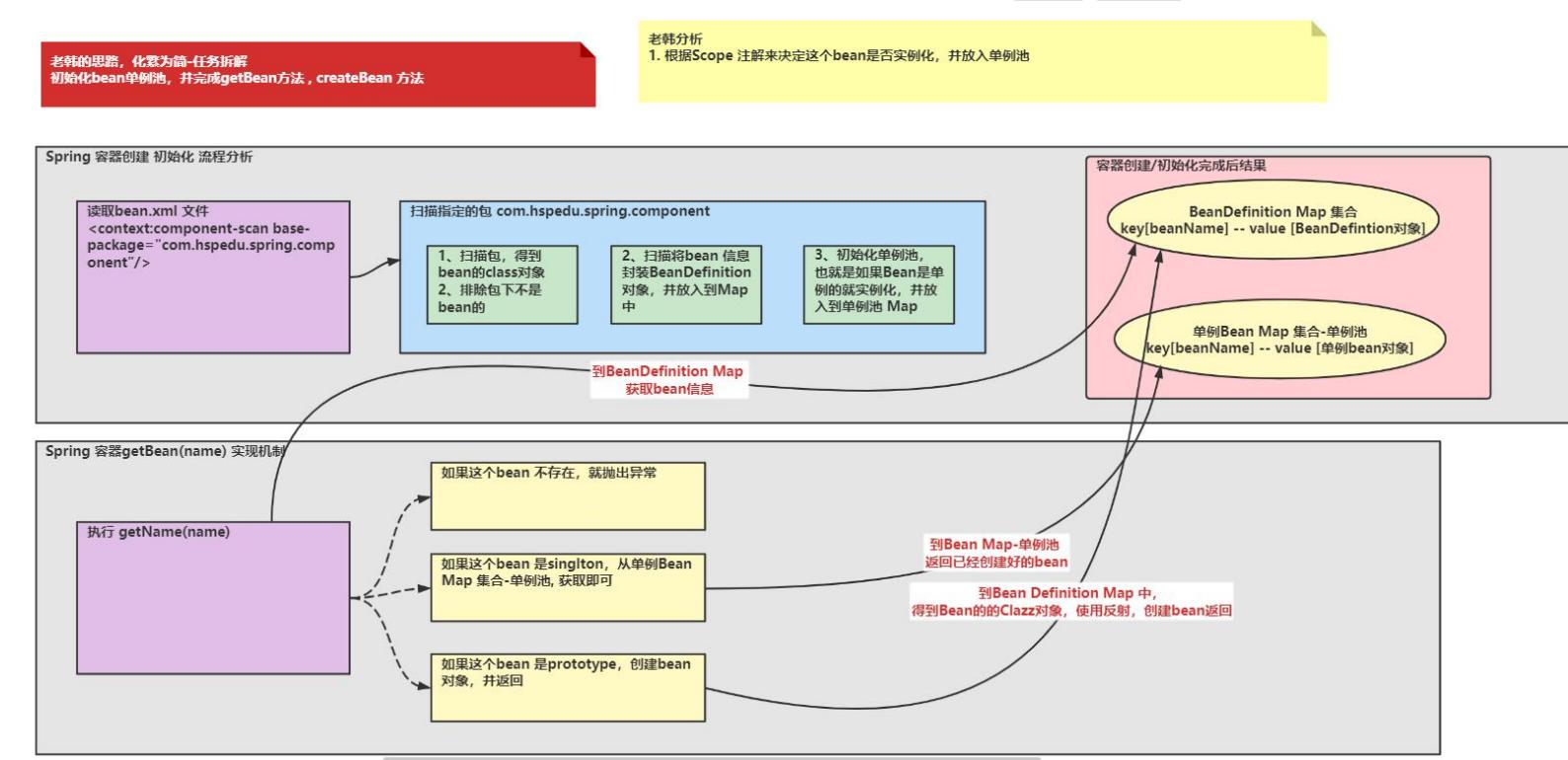


### 4.3.3 实现任务阶段 3- 初始化 bean 单例池，并完成 getBean 方法，createBean 方法

#### 4.3.3.1 说明：初始化 bean 单例池，并完成 getBean 方法，createBean 方法

#### 4.3.3.2 分析+代码实现+测试

- 分析示意图



- 代码实现，老师说明，整个实现思路，就是参考 Spring 规范

1. 修 改

D:\java\_projects2\hsp-spring\src\main\java\com\hspedu\spring\ioc\HspSpringApplicationContext.java, 增加相应的业务代码.

```
public HspSpringApplicationContext(Class configClass) {  
    //通过扫描，得到beanDefinition 的 map  
    beanDefinitionsByScan(configClass);  
    System.out.println(beanDefinitionMap);  
  
    //通过 beanDefinitionMap , 初始化 singletonObjects bean 单列池  
    Enumeration<String> keys = beanDefinitionMap.keys();  
    while (keys.hasMoreElements()) {  
        //得到 beanName  
        String beanName = keys.nextElement();  
        //通过 beanName 得到 beanDefinition  
        BeanDefinition beanDefinition = beanDefinitionMap.get(beanName);  
        if ("singleton".equalsIgnoreCase(beanDefinition.getScope())) {  
            //将该 bean 实例放入 singletonObjects  
            Object bean = createBean(beanDefinition);  
            singletonObjects.put(beanName, bean);  
        }  
    }  
    System.out.println("singletonObjects 单例池 = " + singletonObjects);  
}  
  
//先简单实现实现，后面在完善.
```

```
private Object createBean(BeanDefinition beanDefinition) {  
    //得到bean 的类型  
    Class clazz = beanDefinition.getClazz();  
  
    try {  
        //使用反射得到实例  
        Object instance = clazz.getDeclaredConstructor().newInstance();  
        return instance;  
    } catch (InstantiationException e) {  
        e.printStackTrace();  
    } catch (IllegalAccessException e) {  
        e.printStackTrace();  
    } catch (InvocationTargetException e) {  
        e.printStackTrace();  
    } catch (NoSuchMethodException e) {  
        e.printStackTrace();  
    }  
    //如果没有创建成功，返回null  
    return null;  
}  
  
public Object getBean(String name) {  
    if(beanDefinitionMap.containsKey(name)) {
```

```
BeanDefinition beanDefinition = beanDefinitionMap.get(name);

//得到bean 的 scope , 分别处理

if("singleton".equalsIgnoreCase(beanDefinition.getScope())) {

    //单例, 直接从 bean 单例池获取

    return singletonObjects.get(name);

} else { //不是单例, 则没有返回新的实例

    return createBean(beanDefinition);

}

} else {

    throw new NullPointerException("没有该 bean");

}

}
```

## 2. 修改 D:\java\_projects\hsp-myspring\src\main\java\com\hspedu\spring\AppMain.java

```
package com.hspedu.spring;

import com.hspedu.spring.ioc.HspSpringApplicationContext;
import com.hspedu.spring.ioc.HspSpringConfig;

/**
 * @author 韩顺平
 *
```

```
* @version 1.0  
*/  
  
public class AppMain {  
    public static void main(String[] args) {  
        //创建我们的spring 容器对象  
        HspSpringApplicationContext hspSpringApplicationContext =  
            new HspSpringApplicationContext(HspSpringConfig.class);  
  
        //通过spring 容器对象，获取bean 对象  
        System.out.println(hspSpringApplicationContext.getBean("monsterService"));  
        System.out.println(hspSpringApplicationContext.getBean("monsterService"));  
        System.out.println(hspSpringApplicationContext.getBean("monsterService"));  
    }  
}
```

### 3. 完成测试：输出内容

```
文件绝对路径 = D:\java_projects\hsp-myspring\target\classes\com\hspedu\spring\component\MonsterService  
类名 = MonsterService  
类的全路径 = com.hspedu.spring.component.MonsterService  
是一个bean = class com.hspedu.spring.component.MonsterService  
=====  
{monsterService=BeanDefinition{clazz=class com.hspedu.spring.component.MonsterService, scope=singleton}  
singletonObjects 单例池 = {monsterService=com.hspedu.spring.component.MonsterService@266474c2  
com.hspedu.spring.component.MonsterService@266474c2  
com.hspedu.spring.component.MonsterService@266474c2  
com.hspedu.spring.component.MonsterService@266474c2  
=====
```

```

    扫描 -> class com.hspedu.spring.component.ComponentScanner
=====
{monsterService=BeanDefinition{clazz=class com.hspedu.spring.component.MonsterService
singletonObjects 单例池 = {monsterDao=com.hspedu.spring.component.MonsterDao@6f9
com.hspedu.spring.component.MonsterService@5e481248
com.hspedu.spring.component.MonsterService@66d3c617
com.hspedu.spring.component.MonsterService@63947c6b
}
}

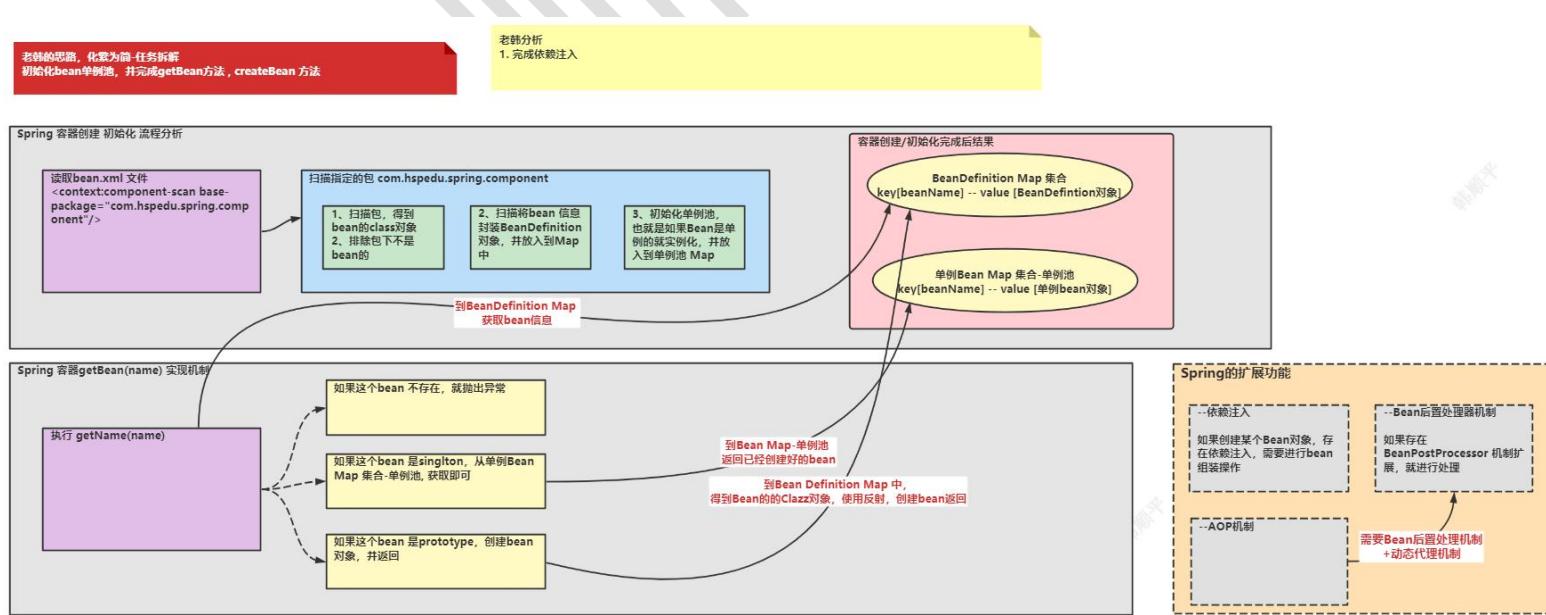
```

#### 4.3.4 实现任务阶段 4- 完成依赖注入

##### 4.3.4.1 说明：完成依赖注入

##### 4.3.4.2 分析+代码实现+测试

- 分析示意图



- 代码实现，老师说明，整个实现思路，就是参考 Spring 规范

1.

创

建

D:\java\_projects2\hsp-spring\src\main\java\com\hspedu\spring\annotation\Autowired.java

a

```
package com.hspedu.spring.annotation;
```

```
import java.lang.annotation.ElementType;
```

```
import java.lang.annotation.Retention;
```

```
import java.lang.annotation.RetentionPolicy;
```

```
import java.lang.annotation.Target;
```

```
/**
```

```
* @author 韩顺平
```

```
* @version 1.0
```

```
* 定义我们的 Autowired 注解
```

```
*/
```

```
@Retention(RetentionPolicy.RUNTIME)
```

```
@Target({ElementType.METHOD, ElementType.FIELD})
```

```
public @interface Autowired {
```

```
    //一个属性 required , 这里我们就不讲了, 也比较简单, 有兴趣同学们作为课后加入
```

```
    //String required() default "true";
```

```
}
```

2.

修

改

D:\java\_projects\hsp-myspring\src\main\java\com\hspedu\spring\component\MonsterDa  
o.java

```
package com.hspedu.spring.component;
```

```
import com.hspedu.spring.annotation.Component;
```

```
/**
```

```
* @author 韩顺平
```

```
* @version 1.0
```

```
*/
```

```
@Component("monsterDao")
```

```
public class MonsterDao {
```

```
    public void hi() {
```

```
        System.out.println("hi 我是 monster Dao, select * from ....");
```

```
}
```

```
}
```

3.

修

改

D:\java\_projects2\hsp-spring\src\main\java\com\hspedu\spring\component\MonsterServi  
ce.java

```
package com.hspedu.spring.component;
```

```
import com.hspedu.spring.annotation.Autowired;
import com.hspedu.spring.annotation.Before;
import com.hspedu.spring.annotation.Component;
import com.hspedu.spring.annotation.Scope;
import com.hspedu.spring.processe.InitializingBean;

/**
 * @author 韩顺平
 * @version 1.0
 */

/**
 * 1. 使用@Component("monsterService") 修饰
 * 2. 给该MonsterService 注入到容器设置beanName 为 monsterService
 * 3. 如果没有设置，默认可以以类名 首字母小写来玩(底层还有很多细节)
 */
@Component("monsterService")
@Scope("prototype")

public class MonsterService implements InitializingBean {

    @Autowired
    private MonsterDao monsterDao;
```

```
public void m1() {  
    //调用 monsterDao 的 hi()  
    monsterDao.hi();  
}  
}
```

4. 修 改  
D:\java\_projects\hsp-myspring\src\main\java\com\hspedu\spring\ioc\HspSpringApplicatio  
nContext.java

```
//先简单实现实现，后面在完善。  
private Object createBean(BeanDefinition beanDefinition) {  
    //得到bean 的类型  
    Class clazz = beanDefinition.getClazz();  
  
    try {  
        //使用反射得到实例  
        Object instance = clazz.getDeclaredConstructor().newInstance();  
    }
```

```
//完成依赖注入  
for (Field declaredField : clazz.getDeclaredFields()) {
```

```
if(declaredField.isAnnotationPresent(Autowired.class)) {  
    // 处理@.Autowired 注解的属性 required, 很简单, 自己完成  
    //          Autowired           annotation      =  
  
    declaredField.getAnnotation(Autowired.class);  
  
    // System.out.println(annotation.required());  
  
    //如果该属性有@Autowired, 就进行组装  
    Object bean = getBean(declaredField.getName());  
  
    declaredField.setAccessible(true); //因为属性是 private, 需要暴破  
  
    declaredField.set(instance, bean);  
  
}  
}  
  
return instance;  
}  
catch (InstantiationException e) {  
    e.printStackTrace();  
}  
catch (IllegalAccessException e) {  
    e.printStackTrace();  
}  
catch (InvocationTargetException e) {  
    e.printStackTrace();  
}  
catch (NoSuchMethodException e) {  
    e.printStackTrace();  
}
```

```
    }

    //如果没有创建成功，返回null
    return null;
}
```

## 5. 修改 D:\java\_projects\hsp-myspring\src\main\java\com\hspedu\spring\AppMain.java

```
package com.hspedu.spring;

import com.hspedu.spring.component.MonsterService;
import com.hspedu.spring.ioc.HspSpringApplicationContext;
import com.hspedu.spring.ioc.HspSpringConfig;

/**
 * @author 韩顺平
 * @version 1.0
 */
public class AppMain {
    public static void main(String[] args) {
        //创建我们的spring 容器对象
        HspSpringApplicationContext hspSpringApplicationContext =
            new HspSpringApplicationContext(HspSpringConfig.class);
    }
}
```

```
//通过spring 容器对象，获取bean 对象  
//  
System.out.println(hspSpringApplicationContext.getBean("monsterService"));  
//  
System.out.println(hspSpringApplicationContext.getBean("monsterService"));  
//  
System.out.println(hspSpringApplicationContext.getBean("monsterService"));
```

MonsterService      monsterService      =      (MonsterService)

```
hspSpringApplicationContext.getBean("monsterService");  
    monsterService.m1();  
}  
}
```

6. 运行完成测试。

```
=====
文件绝对路径 = D:\java_projects\hsp-myspring\target\classes\com.hspedu.spring.component.MonsterService
类名 = MonsterService
类的全路径 = com.hspedu.spring.component.MonsterService
是一个bean = class com.hspedu.spring.component.MonsterService
=====
{monsterService=BeanDefinition{clazz=class com.hspedu.spring.component.MonsterService
singletonObjects 单例池 = {monsterDao=com.hspedu.spring.compor
hi 我是monster Dao, select * from ....
Process finished with exit code 0
```

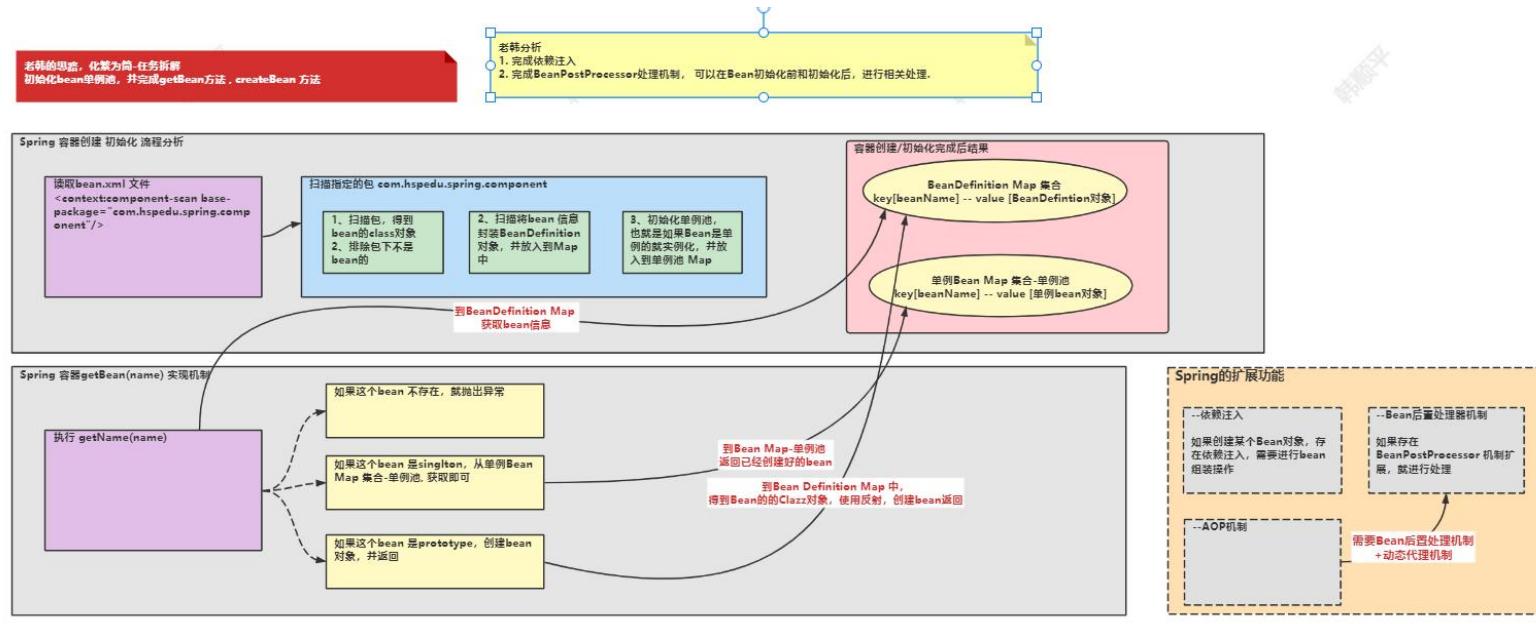
#### 4.3.5 实现任务阶段 5-bean 后置处理器实现

##### 4.3.5.1 先完成原生 Spring 使用 Bean 后置处理器的案例

##### 4.3.5.2 说明：实现自己的 bean 后置处理器

##### 4.3.5.3 分析+代码实现+测试

- 分析示意图



## • 代码实现，老师说明，整个实现思路，就是参考 Spring 规范

1.

创

建

D:\java\_projects2\hsp-spring\src\main\java\com\hspedu\spring\processe\InitializingBean.java，实现该接口的 Bean，需要实现 Bean 初始化方法，可以参考 原生 Spring 规范来定义这个接口

```
package com.hspedu.spring.processe;
```

```
/**
```

```
* @author 韩顺平
```

```
* @version 1.0
```

```
*/
```

```
public interface InitializingBean {
```

```
    void afterPropertiesSet() throws Exception;
```

```
}
```

2.

修

改

D:\java\_projects2\hsp-spring\src\main\java\com\hsppedu\spring\component\MonsterService.java, 实现 InitializingBean 接口

```
package com.hsppedu.spring.component;

import com.hsppedu.spring.annotation.Autowired;
import com.hsppedu.spring.annotation.Before;
import com.hsppedu.spring.annotation.Component;
import com.hsppedu.spring.annotation.Scope;
import com.hsppedu.spring.processe.InitializingBean;

/**
 * @author 韩顺平
 * @version 1.0
 */
/***
 * 1. 使用@Component("monsterService") 修饰
 * 2. 给该MonsterService 注入到容器设置 beanName 为 monsterService
 * 3. 如果没有设置，默认可以以类名 首字母小写来玩(底层还有很多细节)
 */
```

```
*/  
  
@Component("monsterService")  
@Scope("prototype")  
public class MonsterService implements InitializingBean {  
    @Autowired  
    private MonsterDao monsterDao;  
  
    public void m1() {  
        //调用 monsterDao 的 hi()  
        monsterDao.hi();  
    }  
  
    @Override  
    public void afterPropertiesSet() throws Exception {  
        System.out.println("MonsterService 进行初始化, 具体业务由程序员来搞定....");  
    }  
}
```

### 3. 修 改

D:\java\_projects2\hsp-spring\src\main\java\com\hspedu\spring\ioc\HspSpringApplicationContext.java，在创建好 Bean 实例后，判断是否需要进行初始化 【老师心得：容器中常用的一个方法是，根据该类是否实现了某个接口，来判断是否要执行某个业务逻辑，这里

## 【其实就是 java 基础的接口编程实际运用】

//先简单实现实现，后面在完善.

```
private Object createBean(String beanName, BeanDefinition beanDefinition) {
```

//得到 bean 的类型

```
Class clazz = beanDefinition.getClazz();
```

```
try {
```

//使用反射得到实例

```
Object instance = clazz.getDeclaredConstructor().newInstance();
```

//完成依赖注入

```
for (Field declaredField : clazz.getDeclaredFields()) {
```

```
if(declaredField.isAnnotationPresent(Autowired.class)) {
```

// 处理@Autowired 注解的属性 required, 很简单, 自己完成

```
//          Autowired           annotation      =
```

```
declaredField.getAnnotation(Autowired.class);
```

```
// System.out.println(annotation.required());  
  
//如果该属性有@.Autowired, 就进行组装  
  
Object bean = getBean(declaredField.getName());  
  
declaredField.setAccessible(true); //因为属性是 private,需要暴破  
  
declaredField.set(instance, bean);  
  
}  
  
}  
  
//这里还有其他, 比如 Aware 回调. 老师不写了  
  
//这里调用初始化,如果 bean 实现了 InitializingBean  
  
System.out.println("=====创建好了====" + instance);  
  
if(instance instanceof InitializingBean) {  
  
    try {  
  
        ((InitializingBean) instance).afterPropertiesSet();  
  
    } catch (Exception e) {  
  
        e.printStackTrace();  
    }  
}
```

```
    }

}

return instance;

} catch (InstantiationException e) {

    e.printStackTrace();

} catch (IllegalAccessException e) {

    e.printStackTrace();

} catch (InvocationTargetException e) {

    e.printStackTrace();

} catch (NoSuchMethodException e) {

    e.printStackTrace();

}

//如果没有创建成功，返回 null

return null;

}
```

## 4. 完成 测 试 , 运 行

D:\java\_projects2\hsp-spring\src\main\java\com\hspedu\spring\AppMain.java , 看看实现了 InitializingBean 的 Bean 在创建好后，有没有触发初始化方法

```

类名 = MonsterService
类的全路径 = com.hspedu.spring.component.MonsterService
是一个bean = class com.hspedu.spring.component.MonsterService
=====
{monsterService=BeanDefinition{clazz=class com.hspedu.spring.component.MonsterService, scope='r
=====创建好了====com.hspedu.spring.component.MonsterDao@5e481248
singletonObjects 单例池 = {monsterDao=com.hspedu.spring.component.MonsterDao@5e481248}
=====创建好了====com.hspedu.spring.component.MonsterService@355da254
MonsterService 进行初始化，具体业务由程序员来搞定.... ○ ←
如果Bean实现了InitializingBean,
就会在创建好后，调用它的初始化方法

```

## 5. 创 建

D:\java\_projects2\hsp-spring\src\main\java\com\hspedu\spring\processe\BeanPostProces  
sor.java , 该接口可以参考原生 Spring 规范 , 注意体会切面编程

```

package com.hspedu.spring.processe;

import com.sun.istack.internal.Nullable;

/**
 * @author 韩顺平
 * @version 1.0
 */
public interface BeanPostProcessor {

```

//bean 初始化前执行的业务

```
Object postProcessBeforeInitialization(Object bean, String beanName);
```

//bean 初始化后执行的业务

```
Object postProcessAfterInitialization(Object bean, String beanName);
```

```
}
```

6.

创

建

D:\java\_projects2\hsp-spring\src\main\java\com\hspedu\spring\component\HspBeanPostProcessor.java

```
package com.hspedu.spring.component;
```

```
import com.hspedu.spring.annotation.Component;
```

```
import com.hspedu.spring.processe.BeanPostProcessor;
```

```
import java.lang.reflect.InvocationHandler;
```

```
import java.lang.reflect.Method;
```

```
import java.lang.reflect.Proxy;
```

```
/**
```

```
* @author 韩顺平
```

```
* @version 1.0
```

```
*/  
  
/**  
 * 老韩说明  
 * 1. HspBeanPostProcessor Bean 处理器本身也是一个 bean  
 */  
  
@Component  
  
public class HspBeanPostProcessor implements BeanPostProcessor {  
  
    /**  
     * 该方法时在 bean 创建好后，进行初始化前调用  
     * @param bean : 创建好的 bean 对象  
     * @param beanName 创建好的 bean 的名字  
     * @return  
     */  
  
    @Override  
  
    public Object postProcessBeforeInitialization(Object bean, String beanName) {  
        //这里程序员来决定业务逻辑，spring 只是提供处理机制  
        System.out.println("postProcessBeforeInitialization 被调用 "  
                           + beanName + " bean= " + bean.getClass());  
  
        return bean;  
    }  
}
```

```
/*
 * 该方法时在 bean 创建好后，初始化完成后调用
 * @param bean : 创建好的 bean 对象
 * @param beanName : 创建好的 bean 的名字
 * @return
 */
@Override
public Object postProcessAfterInitialization(Object bean, String beanName) {
    //这里程序员来决定业务逻辑，spring 只是提供处理机制
    System.out.println("postProcessAfterInitialization 被调用 "
        + beanName + " bean= " + bean.getClass());
    return bean;
}
```

7. 修 改  
D:\java\_projects\hsp-myspring\src\main\java\com\hspedu\spring\ioc\HspSpringApplicationContext.java

**注意：** 这里 createBean(String beanName, BeanDefinition beanDefinition) 需要增加入参 beanName，就会导致好几个位置错误，需要根据错误提示，对应解决即可。

//将bean 处理器对象，放在ArrayList 即可

```
private List<BeanPostProcessor> beanPostProcessorList = new ArrayList<>();
```

```
private void beanDefinitionsByScan(Class configClass) {
```

```
    this.configClass = configClass;
```

//1. 解析配置类

//2. 获取到配置类的 @ComponentScan("com.hspedu.spring.component")

ComponentScan componentScan = (ComponentScan)

```
this.configClass.getDeclaredAnnotation(ComponentScan.class);
```

```
String path = componentScan.value();
```

```
System.out.println("扫描路径 = " + path);
```

//3. 获取扫描路径下所有的类文件

//(1) 先得到类加载器，使用App 方式来加载

```
ClassLoader classLoader = HspSpringApplicationContext.class.getClassLoader();
```

//在获取某个包的d 对应的URL 时，要求是 com/hspedu/spring/component

```
//URL resource = classLoader.getResource("com/hspedu/spring/component");
```

//(2) 将path 转成 形式为 com/hspedu/spring/component

```
path = path.replace(".", "/");
```

```
URL resource = classLoader.getResource(path);
```

```
File file = new File(resource.getFile());  
  
if (file.isDirectory()) {  
  
    File[] files = file.listFiles();  
  
    for (File f : files) {  
  
        String fileAbsolutePath = f.getAbsolutePath();  
  
        System.out.println("=====");  
  
        System.out.println("文件绝对路径 = " + fileAbsolutePath);  
  
        if (fileAbsolutePath.endsWith(".class")) {//说明是类文件  
  
            //通过类加载器获取来类文件的Clazz 对象  
  
            // 先 得 到 类 的 完 整 类 路 径 形 式 为  
com.hspedu.spring.component.MonsterService  
  
            String className =  
  
                fileAbsolutePath.substring(fileAbsolutePath.lastIndexOf("\\") + 1,  
fileAbsolutePath.indexOf(".class"));  
  
            String classFullPath = path.replace("/", ".") + "." + className;  
  
            System.out.println("类名 = " + className);  
            System.out.println("类的全路径 = " + classFullPath);  
  
            try {  
  
                //获取到扫描包下的类的 clazz 对象  
                Class<?> clazz = classLoader.loadClass(classFullPath);  
  
                if (clazz.isAnnotationPresent(Component.class)) {  
                    //...  
                }  
            } catch (Exception e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

//如果这个类有@Componen, 说明是一个 spring bean

System.out.println("是一个 bean = " + clazz);

//老韩解读

//1. 增加一个逻辑，如果这个 clazz 类型是实现了 BeanPostProcessor 接口,说明是一个 bean 处理器，特殊处理

//2. 注意不能使用 clazz instanceof BeanPostProcessor 判断  
因为 clazz 并不是一个实例对象,而是一个类对象

//3. 老师这里实现是为了方便获取 bean 处理器对象，所以放在一个 beanPostProcessorList, spring 底层源码,

// 还是走的 createBean(),getBean(), 只是需要在 singletonObjects,增加代码处理,我这里主要讲的是 bean 处理器的工作

// 机制,就不处理了,小伙伴知道即可

**if (BeanPostProcessor.class.isAssignableFrom(clazz)) {**

//创建一个实例对象

    BeanPostProcessor                  beanPostProcessor         =

    (BeanPostProcessor) clazz.newInstance();

//放入到 beanPostProcessorList

    beanPostProcessorList.add(beanPostProcessor);

**continue;**

}

//老韩解读

//1. 因为这里不能直接将 bean 实例放入 singletonObjects  
//2. 原因是如果 bean 是 prototype 是需要每次创建新的 bean 对象  
//3. 所以， Spring 底层是这样设计的：将 bean 信息封装到  
BeanDefinition 对象中，便于 getBean 的操作

```
BeanDefinition beanDefinition = new BeanDefinition();
beanDefinition.setClazz(clazz);
//获取 bean 的 name
Component componentAnnotation = clazz.getDeclaredAnnotation(Component.class);
String beanName = componentAnnotation.value();
//获取 bean 的 scope
if (clazz.isAnnotationPresent(Scope.class)) { //如果有@Scope
    Scope scopeAnnotation = clazz.getDeclaredAnnotation(Scope.class);
    beanDefinition.setScope(scopeAnnotation.value());
} else { //如果没有@Scope, 默认是 singleton
    beanDefinition.setScope("singleton");
}

//放入到 beanDefinitionMap
beanDefinitionMap.put(beanName, beanDefinition);
```

```
        } else {  
            //如果这个类没有@Componenent, 说明不是一个 spring bean  
            System.out.println("不是一个 bean = " + clazz);  
        }  
  
    } catch (ClassNotFoundException e) {  
        e.printStackTrace();  
    } catch (IllegalAccessException e) {  
        e.printStackTrace();  
    } catch (InstantiationException e) {  
        e.printStackTrace();  
    }  
    System.out.println("====");  
}  
}  
}
```

//先简单实现实现，后面在完善。

```
private Object createBean(String beanName, BeanDefinition beanDefinition) {  
    //得到bean 的类型
```

```
Class clazz = beanDefinition.getClazz();

try {
    // 使用反射得到实例
    Object instance = clazz.getDeclaredConstructor().newInstance();
    // 完成依赖注入
    for (Field declaredField : clazz.getDeclaredFields()) {
        if (declaredField.isAnnotationPresent(Autowired.class)) {
            // 处理@.Autowired 注解的属性 required, 很简单, 自己完成
            // Autowired annotation = declaredField.getAnnotation(Autowired.class);
            // System.out.println(annotation.required());
            // 如果该属性有@Autowired, 就进行组装
            Object bean = getBean(declaredField.getName());
            declaredField.setAccessible(true); // 因为属性是 private, 需要暴破
            declaredField.set(instance, bean);
        }
    }
    // 这里还有其他, 比如 Aware 回调. 老师不写了

    // 老韩说明
    // 1. 在 bean 初始化前调用所有 bean 处理器的 postProcessBeforeInitialization
    // 2. 调用时, 不能保证顺序
}
```

//3. 可以通过加入@Order("值"), 来指定bean 处理器调用顺序, 同学们可以自行完成, 不难

//4. 如果希望指定对哪些 bean 进行初始化前处理, 可以在 处理器的 postProcessBeforeInitialization()

// 加入相关业务判断即可. 比如:

```
/*
 * @Override
 *      public Object postProcessBeforeInitialization(Object bean, String
beanName) {
        if("monsterService".equalsIgnoreCase(beanName)) {
            //这里程序员来决定业务逻辑, spring 只是提供处理机制
            System.out.println("postProcessBeforeInitialization 被调用 "
+ beanName + " bean= " + bean.getClass());
            return bean;
        } else {
            return bean;
        }
    }
*/
```

**for (BeanPostProcessor beanPostProcessor : beanPostProcessorList) {**

//4. 也会返回一个对象, 这个返回的对象是什么, 由程序员在编写 bean 处理器决定, 可能是原来的 bean, 也可能被改变了

```
instance = beanPostProcessor.postProcessBeforeInitialization(instance,  
beanName);  
}
```

//这里调用初始化,如果 bean 实现了 InitializingBean

```
if (instance instanceof InitializingBean) {  
    try {  
        ((InitializingBean) instance).afterPropertiesSet();  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

//老韩说明

//1. 在 bean 初始化后调用所有 bean 处理器的 postProcessAfterInitialization

//2. 调用时，不能保证顺序

//3. 可以通过加入@Order("值"), 来指定 bean 处理器调用顺序, 同学们可以自行完成,

不难

//4. 如果希望指定对哪些 bean 进行初始化后处理, 可以在 处理器的  
postProcessAfterInitialization()

// 加入相关业务判断即可

```
for (BeanPostProcessor beanPostProcessor : beanPostProcessorList) {
```

//4. 也会返回一个对象，这个返回的对象是什么，由程序员在编写 bean 处理器决定，可能是原来的 bean,也可能被改变了

```
instance = beanPostProcessor.postProcessAfterInitialization(instance,  
beanName);  
  
}  
  
return instance;  
} catch (InstantiationException e) {  
    e.printStackTrace();  
} catch (IllegalAccessException e) {  
    e.printStackTrace();  
} catch (InvocationTargetException e) {  
    e.printStackTrace();  
} catch (NoSuchMethodException e) {  
    e.printStackTrace();  
}  
//如果没有创建成功，返回 null  
return null;  
}
```

8. 运行 D:\java\_projects2\hsp-spring\src\main\java\com\hspedu\spring\AppMain.java , 完成测试。

```
类的全路径 = com.hspedu.spring.component.MonsterService  
是一个bean = class com.hspedu.spring.component.MonsterService  
=====  
{monsterService=BeanDefinition{clazz=class com.hspedu.spring.component.MonsterService, scope='prototy  
=====创建好了====com.hspedu.spring.component.MonsterDao@5e481248  
postProcessBeforeInitialization 被调用 monsterDao bean= class com.hspedu.spring.component.MonsterDa  
postProcessAfterInitialization 被调用 monsterDao bean= class com.hspedu.spring.component.MonsterDa  
singletonObjects 单例池 = {monsterDao=com.hspedu.spring.component.MonsterDao@5e481248}  
=====创建好了====com.hspedu.spring.component.MonsterService@355da254  
postProcessBeforeInitialization 被调用 monsterService bean= class com.hspedu.spring.component.MonsterS  
MonsterService 进行初始化，具体业务由程序员来搞定....  
postProcessAfterInitialization 被调用 monsterService bean= class com.hspedu.spring.component.MonsterSe  
Process finished with exit code 0
```

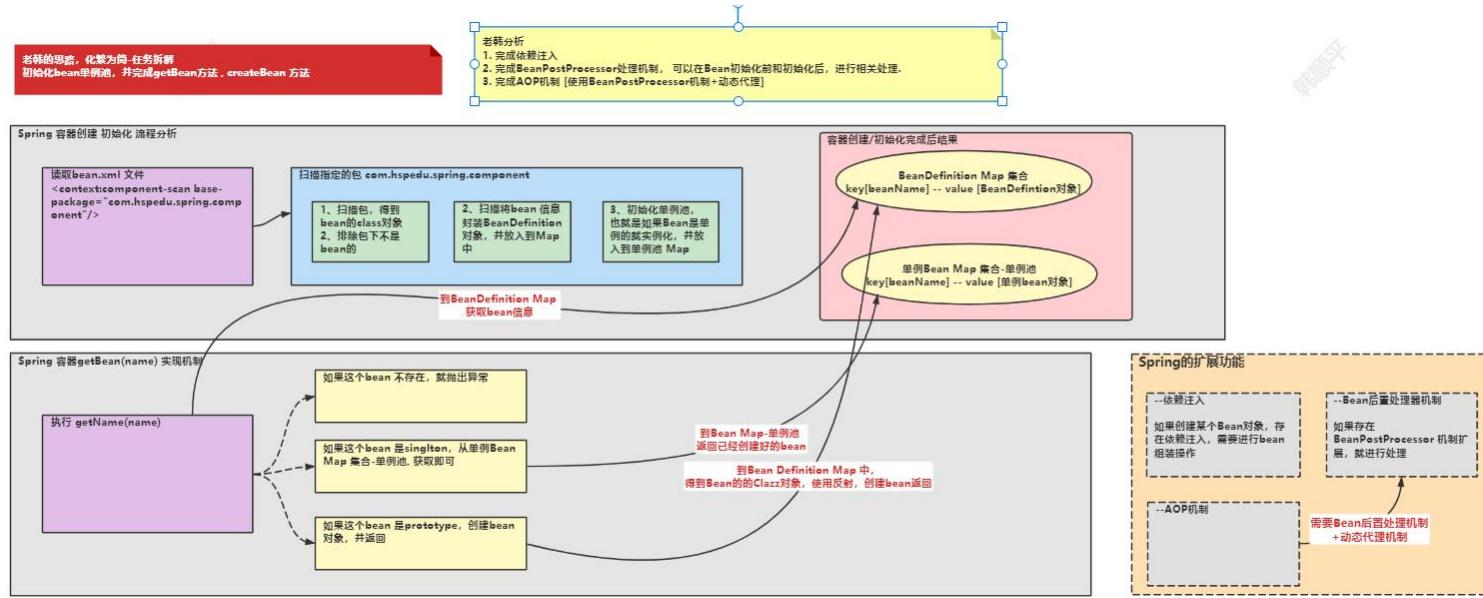
#### 4.3.6 实现任务阶段 6- AOP 机制实现

##### 4.3.6.1 先完成原生 Spring 使用 AOP 的案例

##### 4.3.6.2 说明：实现自己的 AOP 机制

##### 4.3.6.3 分析+代码实现+测试

- 分析示意图



- 代码实现，老师说明，整个实现思路，就是参考 Spring 规范，这里老师先写死，然后再分析如何做的更加灵活

1.

创

建

D:\java\_projects2\hsp-spring\src\main\java\com\hspedu\spring\component\SmartAnimalable.java

```
package com.hspedu.spring.component;
```

```
/**
```

```
* @author 韩顺平
```

```
* @version 1.0
```

```
*/
```

```
public interface SmartAnimalable {
```

```
    float getSum(float i, float j);
```

```
    float getSub(float i, float j);  
}
```

2.

创

建

D:\java\_projects2\hsp-spring\src\main\java\com\hspedu\spring\component\SmartDog.java

a

```
package com.hspedu.spring.component;
```

```
import com.hspedu.spring.annotation.Component;
```

```
/**
```

```
* @author 韩顺平
```

```
* @version 1.0
```

```
* 实现接口是为了方便返回代理对象.
```

```
*/
```

```
@Component("smartDog")
```

```
public class SmartDog implements SmartAnimalable {
```

```
    @Override
```

```
    public float getSum(float i, float j) {
```

```
        float result = i + j;
```

```
        System.out.println("getSum() 方法内部打印 result= " + result);
```

```
        return result;
```

```
}

@Override

public float getSub(float i, float j) {

    float result = i - j;

    System.out.println("getSub() 方法内部打印 result= " + result);

    return result;

}

}

3. 创建
```

D:\java\_projects2\hsp-spring\src\main\java\com\hspedu\spring\component\SmartAnimal

Aspect.java

```
package com.hspedu.spring.component;

import com.hspedu.spring.annotation.Before;
import com.hspedu.spring.annotation.Component;

/**
 * @author 韩顺平
 * @version 1.0
 */

public class SmartAnimalAspect {
```

```
public static void showBeginLog() {  
    System.out.println("前置通知");  
}  
  
public static void showSuccessEndLog() {  
    System.out.println("返回通知");  
}  
}
```

4.

修

改

D:\java\_projects2\hsp-spring\src\main\java\com\hsppedu\spring\component\HspBeanPostProcessor.java，根据情况返回代理对象还是 Bean 原生对象

```
package com.hsppedu.spring.component;  
  
import com.hsppedu.spring.annotation.Component;  
import com.hsppedu.spring.processe.BeanPostProcessor;  
  
import java.lang.reflect.InvocationHandler;  
import java.lang.reflect.Method;  
import java.lang.reflect.Proxy;  
  
/**  
 * @author 韩顺平  
 * @version 1.0
```

```
*/
```

```
/**
```

```
* 老韩说明
```

```
* 1. HspBeanPostProcessor Bean 处理器本身也是一个 bean
```

```
*/
```

```
@Component
```

```
public class HspBeanPostProcessor implements BeanPostProcessor {
```

```
/**
```

```
* 该方法时在 bean 创建好后，进行初始化前调用
```

```
* @param bean : 创建好的 bean 对象
```

```
* @param beanName 创建好的 bean 的名字
```

```
* @return
```

```
*/
```

```
@Override
```

```
public Object postProcessBeforeInitialization(Object bean, String beanName) {
```

```
//这里程序员来决定业务逻辑，spring 只是提供处理机制
```

```
System.out.println("postProcessBeforeInitialization 被调用 "
```

```
    + beanName + " bean= " + bean.getClass());
```

```
return bean;
```

```
}
```

```
/**  
 * 该方法时在bean 创建好后，初始化完成后调用  
 * @param bean : 创建好的bean 对象  
 * @param beanName : 创建好的bean 的名字  
 * @return  
 */  
  
@Override  
public Object postProcessAfterInitialization(Object bean, String beanName) {  
  
    //这里程序员来决定业务逻辑，spring 只是提供处理机制  
    System.out.println("postProcessAfterInitialization 被调用 "  
        + beanName + " bean= " + bean.getClass());  
  
    //实现AOP, 返回代理对象，即对bean 对象进行包装，不在返回原生的bean  
    //1. 先死活后活 --> 后面使用注解就可非常灵活  
    if("smartDog".equals(beanName)) {  
  
        //2. 使用JDK 的动态代理，代理bean, 返回的代理对象  
        Object proxyInstance =  
            Proxy.newProxyInstance(HspBeanPostProcessor.class.getClassLoader(),  
                bean.getClass().getInterfaces(), new InvocationHandler() {  
  
                    @Override  
                    public Object invoke(Object proxy, Method method,
```

```
Object[] args) throws Throwable {  
    System.out.println("method      =      "+"  
    method.getName());  
  
    //if(method.getName().equals(""))  
    //System.out.println("代理方法，也就是在执行目标方  
法前希望切入的逻辑代码");  
  
    Object invoke = null;  
    if("getSum".equals(method.getName())) {  
        SmartAnimalAspect.showBeginLog();  
        //return method.invoke(bean, args);  
        invoke = method.invoke(bean, args);  
        SmartAnimalAspect.showSuccessEndLog();  
    } else {  
        invoke = method.invoke(bean, args);  
    }  
    return invoke;  
}  
});  
  
return proxyInstance;  
}  
  
//不需要实现 AOP, 仍然返回原生的 bean  
return bean;
```

```
    }  
}
```

## 5. 修改 D:\java\_projects2\hsp-spring\src\main\java\com\hspedu\spring\AppMain.java, 完成测试

```
package com.hspedu.spring;  
  
import com.hspedu.spring.component.MonsterService;  
import com.hspedu.spring.component.SmartAnimalable;  
import com.hspedu.spring.component.SmartDog;  
import com.hspedu.spring.ioc.HspSpringApplicationContext;  
import com.hspedu.spring.ioc.HspSpringConfig;  
  
/**  
 * @author 韩顺平  
 * @version 1.0  
 */  
  
public class AppMain {  
    public static void main(String[] args) {  
        //创建我们的 spring 容器对象  
        HspSpringApplicationContext hspSpringApplicationContext =
```

```
new HspSpringApplicationContext(HspSpringConfig.class);

//通过spring 容器对象, 获取bean 对象

MonsterService      monsterService      =      (MonsterService)
hspSpringApplicationContext.getBean("monsterService");

//      MonsterService      monsterService2      =      (MonsterService)
hspSpringApplicationContext.getBean("monsterService");

//      MonsterService      monsterService3      =      (MonsterService)
hspSpringApplicationContext.getBean("monsterService");

//System.out.println(monsterService);
// System.out.println(monsterService2);
// System.out.println(monsterService3);

//monsterService.m1();

//如果某个Bean 是被代理了, 则返回的是代理对象, 而不是原生 Bean, 从而实现
AOP

SmartAnimalable smartDog =
(SmartAnimalable)

hspSpringApplicationContext.getBean("smartDog");

smartDog.getSub(10, 20);
System.out.println("=====");
```

```
    smartDog.getSum(10, 40);  
}  
}
```

```
AppMain (1) ×  
singletonObjects 单例池 = {smartDog=com.hspedu.spring.com...  
=====创建好了====com.hspedu.spring.component.MonsterService  
postProcessBeforeInitialization 被调用 monsterService bear  
MonsterService 进行初始化，具体业务由程序员来搞定....  
postProcessAfterInitialization 被调用 monsterService bean=  
method = getSub  
getSub() 方法内部打印 result= -10.0  
=====  
method = getSum  
前置通知  
getSum() 方法内部打印 result= 50.0  
返回通知
```

SmartDog被代理-  
getSub() 没有AOP切入

SmartDog被代理-  
getSum() 有AOP切入

#### 4.3.6.4 课后思考扩展：如何做的更加灵活

1. 前面我们使用的硬编码，不灵活，但是已经把 AOP 核心机制说清楚了
2. 小伙伴可以思考如何把 AOP 做的更加灵活，核心知识点(注解+数据结构/map/少许算法+业务处理)，其实和 AOP 机制关系不大了。
3. 老师给出思路，同学们课后愿意做的，就玩玩。

1)

修

改

D:\java\_projects2\hsp-spring\src\main\java\com\hspedu\spring\component\SmartAnimalAspect.java, 老师这里简化了

```
package com.hspedu.spring.component;
```

```
import com.hspedu.spring.annotation.Before;
```

```
import com.hspedu.spring.annotation.Component;
```

```
/**
```

```
* @author 韩顺平
```

```
* @version 1.0
```

```
*/
```

**@Aspect** //表示这个类是一个切面类

**@Component** //需要加入到IOC 容器

```
public class SmartAnimalAspect {
```

```
    @Before(value = "execution com.hspedu.spring.aop.aspectj.SmartDog getSum")
```

```
    public static void showBeginLog() {
```

```
        System.out.println("前置通知");
```

```
}
```

```
    @After(value = "execution com.hspedu.spring.aop.aspectj.SmartDog getSum")
```

```
public static void showSuccessEndLog() {  
    System.out.println("返回通知");  
}  
}
```

2)

修

改

D:\java\_projects2\hsp-spring\src\main\java\com\hspedu\spring\component\HspBeanPostProcessor.java

```
package com.hspedu.spring.component;
```

```
/**
```

```
* @author 韩顺平
```

```
* @version 1.0
```

```
*/
```

```
import com.hspedu.spring.annotation.Component;
```

```
import com.hspedu.spring.processe.BeanPostProcessor;
```

```
import java.lang.reflect.InvocationHandler;
```

```
import java.lang.reflect.Method;
```

```
import java.lang.reflect.Proxy;
```

```
/**  
 * 老韩说明  
 * 1. HspBeanPostProcessor Bean 处理器本身也是一个 bean  
 */
```

@Component

```
public class HspBeanPostProcessor implements BeanPostProcessor {
```

```
/**
```

```
 * 该方法时在 bean 创建好后，进行初始化前调用
```

```
 * @param bean : 创建好的 bean 对象
```

```
 * @param beanName 创建好的 bean 的名字
```

```
 * @return
```

```
*/
```

@Override

```
public Object postProcessBeforeInitialization(Object bean, String beanName) {
```

```
//这里程序员来决定业务逻辑，spring 只是提供处理机制
```

```
System.out.println("postProcessBeforeInitialization 被调用 "
```

```
    + beanName + " bean= " + bean.getClass());
```

```
return bean;
```

```
}
```

```
/**
```

```
 * 该方法时在 bean 创建好后，初始化完成后调用
```

```
* @param bean : 创建好的bean 对象  
* @param beanName : 创建好的bean 的名字  
* @return  
*/
```

@Override

```
public Object postProcessAfterInitialization(Object bean, String beanName) {
```

//这里程序员来决定业务逻辑，spring 只是提供处理机制

```
System.out.println("postProcessAfterInitialization 被调用 "  
    + beanName + " bean= " + bean.getClass());
```

//实现AOP, 返回代理对象, 即对bean 对象进行包装, 不在返回原生的bean

//1. 先死活 --> 后面使用注解就可非常灵活

```
if("smartDog".equals(beanName)) { //1. 通过注解看看当前这个类是否已经被代理
```

**【需要事先就在 Map 中代理保存关系】**

//2. 使用JDK 的动态代理, 代理bean, 返回的代理对象

Object proxyInstance =

```
Proxy.newProxyInstance(HspBeanPostProcessor.class.getClassLoader(),
```

```
bean.getClass().getInterfaces(), new InvocationHandler() {
```

@Override

```
public Object invoke(Object proxy, Method method, Object[] args)
```

```
throws Throwable {
```

```
System.out.println("method = " + method.getName());  
//if(method.getName().equals(""))  
//System.out.println("代理方法，也就是在执行目标方法前希望切入的逻辑代码");
```

```
Object invoke = null;
```

**//2. 通过注解看看当前这个类的哪个方法已经被代理 【需要事先就在 Map 中代理保存关系】**

```
SmartAnimalAspect.showBeginLog();
```

**@Before 看看当前这个类的这个方法已经被切面类的哪个方法进行@Before 切入 【需要事先就在 Map 中代理保存关系】**

```
//return method.invoke(bean, args);
```

```
invoke = method.invoke(bean, args);
```

```
SmartAnimalAspect.showSuccessEndLog();
```

**//4. 通过注解@AfterReturing 看看当前这个类的这个方法已经被切面类的哪个方法进行@AfterReturing 切入 【需要事先就在 Map 中代理保存关系】**

```
} else {
```

```
invoke = method.invoke(bean, args);
```

```
}
```

```
return invoke;
```

```
}
```

```
});
```

```
return proxyInstance;
```

```
    }

    //不需要实现AOP, 仍然返回原生的bean

    return bean;

}

}
```

#### 4. 如何获取到方法的注解和值，给一个简单案例

1)

创

建

D:\java\_projects\hsp-myspring\src\main\java\com\hspedu\spring\annotation\Aspect.java

```
package com.hspedu.spring.annotation;

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

/**
 * @author 韩顺平
 * @version 1.0
 */

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
```

```
public @interface Aspect {  
    // @Component 可以传入属性值，这里我们默认为 ""  
    String value() default "";  
}
```

2)

创

建

D:\java\_projects\hsp-myspring\src\main\java\com\hspedu\spring\annotation\Before.java

```
package com.hspedu.spring.annotation;
```

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

```
/**
```

```
* @author 韩顺平
```

```
* @version 1.0
```

```
*/
```

```
@Retention(RetentionPolicy.RUNTIME)
```

```
@Target({ElementType.METHOD})
```

```
public @interface Before {
```

```
    String value();
```

}

3)

创

建

D:\java\_projects\hsp-myspring\src\main\java\com\hspedu\spring\annotation\After.java

```
package com.hspedu.spring.annotation;
```

```
import java.lang.annotation.ElementType;
```

```
import java.lang.annotation.Retention;
```

```
import java.lang.annotation.RetentionPolicy;
```

```
import java.lang.annotation.Target;
```

```
/**
```

```
* @author 韩顺平
```

```
* @version 1.0
```

```
*/
```

```
@Retention(RetentionPolicy.RUNTIME)
```

```
@Target({ElementType.METHOD})
```

```
public @interface After {
```

```
    String value();
```

}

4)

修

改

D:\java\_projects\hsp-myspring\src\main\java\com\hspedu\spring\component\SmartAnimalAspect.java

```
package com.hspedu.spring.component;

import com.hspedu.spring.annotation.After;
import com.hspedu.spring.annotation.Aспект;
import com.hspedu.spring.annotation.Before;
import com.hspedu.spring.annotation.Component;

/**
 * @author 韩顺平
 * @version 1.0
 */
@Aспект //表示这个类是一个切面类
@Component //需要加入到IOC 容器
public class SmartAnimalAspect {

    @Before(value = "execution com.hspedu.spring.aop.aspectj.SmartDog getSum")
    public static void showBeginLog() {
        System.out.println("前置通知");
    }

    @After(value = "execution com.hspedu.spring.aop.aspectj.SmartDog getSum")
```

```
public static void showSuccessEndLog() {  
    System.out.println("返回通知");  
}  
}
```

## 5) 创建 测 试 文 件 :

D:\java\_projects\hsp-myspring\src\main\java\com\hspedu\spring\component\Test.java

```
package com.hspedu.spring.component;  
  
import com.hspedu.spring.annotation.After;  
import com.hspedu.spring.annotation.Before;  
  
import java.lang.annotation.Annotation;  
import java.lang.reflect.AnnotatedType;  
import java.lang.reflect.InvocationTargetException;  
import java.lang.reflect.Method;  
  
/**  
 * @author 韩顺平  
 * @version 1.0  
 */  
  
public class Test {
```

```
public static void main(String[] args) throws NoSuchMethodException,  
InvocationTargetException, IllegalAccessException, InstantiationException {
```

```
Class<SmartAnimalAspect> smartAnimalAspectClass = SmartAnimalAspect.class;
```

```
for (Method declaredMethod : smartAnimalAspectClass.getDeclaredMethods()) {
```

```
    if (declaredMethod.isAnnotationPresent(Before.class)) {
```

```
        System.out.println("m:= " + declaredMethod.getName());
```

```
        Before annotation = declaredMethod.getAnnotation(Before.class);
```

```
        System.out.println("value:= " + annotation.value());
```

//得到切入要执行的方法

Method

declaredMethod1

=

```
smartAnimalAspectClass.getDeclaredMethod(declaredMethod.getName());
```

//调用切入方法

```
declaredMethod1.invoke(smartAnimalAspectClass.newInstance(),null);
```

```
} else if(declaredMethod.isAnnotationPresent(After.class)) {
```

```
    System.out.println("m:= " + declaredMethod.getName());
```

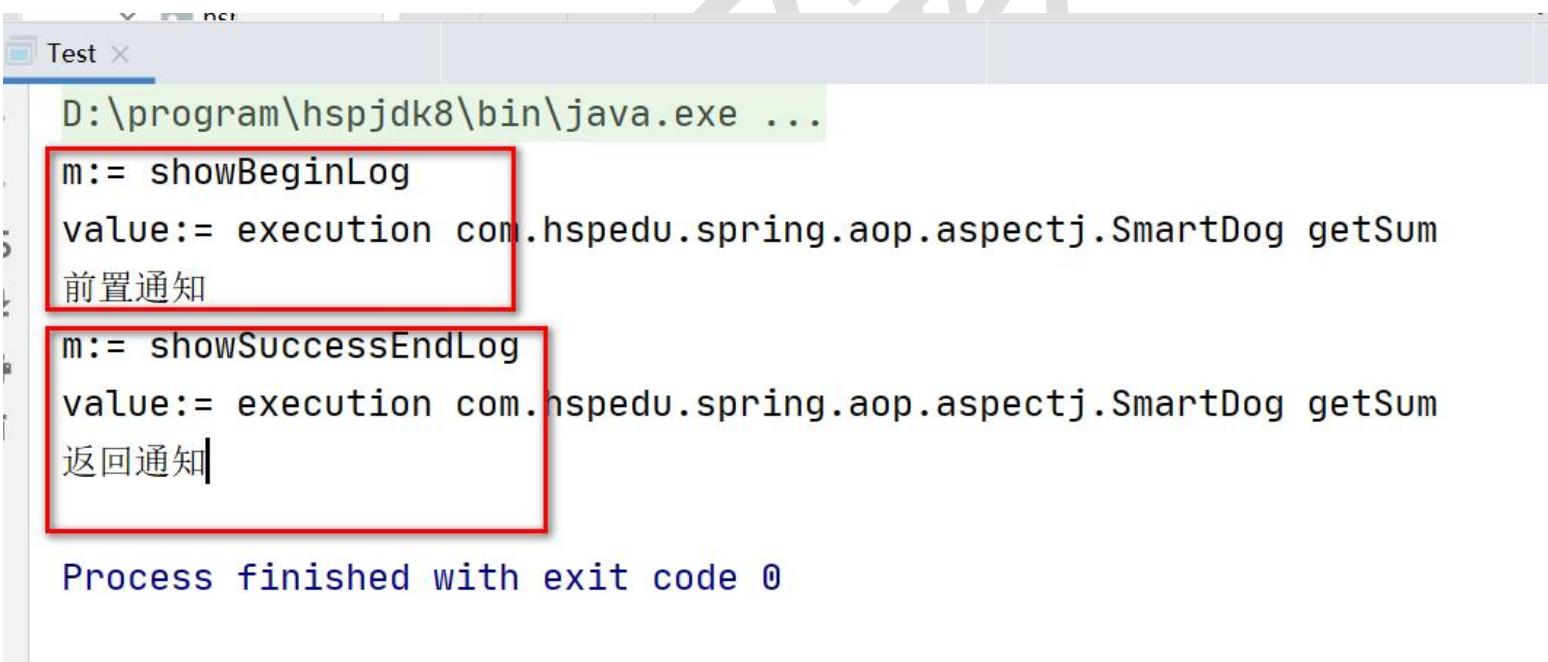
```
    After annotation = declaredMethod.getAnnotation(After.class);
```

```
    System.out.println("value:= " + annotation.value());
```

//得到切入要执行的方法

```
Method declaredMethod1 =  
smartAnimalAspectClass.getDeclaredMethod(declaredMethod.getName());  
//调用切入方法  
declaredMethod1.invoke(smartAnimalAspectClass.newInstance(),null);  
}  
}  
}  
}  
}
```

## 6) 测试效果



The screenshot shows a Java terminal window titled "Test". The command entered is "D:\program\hspjdk8\bin\java.exe ...". The output displays two annotated methods from the SmartDog class:

```
m:= showBeginLog  
value:= execution com.hspedu.spring.aop.aspectj.SmartDog getSum  
前置通知  
m:= showSuccessEndLog  
value:= execution com.hspedu.spring.aop.aspectj.SmartDog getSum  
返回通知
```

The first method, "showBeginLog", is annotated with `@Before("execution com.hspedu.spring.aop.aspectj.SmartDog getSum")`. The second method, "showSuccessEndLog", is annotated with `@After("execution com.hspedu.spring.aop.aspectj.SmartDog getSum")`. Both annotations are highlighted with red boxes.

At the bottom of the terminal, the message "Process finished with exit code 0" is displayed.

## 4.4 小结

### 4.4.2 回顾 Spring 工作流程

## 4.5 课后作业题

4.5.2 不看老师代码，完成手动实现 Spring 底层机制【初始化 IOC 容器+依赖注入 +BeanPostProcessor 机制+AOP】，2-3 遍

## 5 JdbcTemplate

### 5.1 看一个实际需求

- 实际需求：如果程序员就希望使用 spring 框架来做项目，spring 框架如何处理对数据库的操作呢？

1. 方案 1. 使用前面做项目开发的 JdbcUtils 类

2. 方案 2. 其实 spring 提供了一个操作数据库(表)功能强大的类 JdbcTemplate。我们可以同 ioc 容器来配置一个 jdbcTemplate 对象，使用它来完成对数据库表的各种操作。

## 5.2 官方文档

### 5.2.1 JdbcTemplate APIs : /spring-framework-5.3.8/docs/javadoc-api/index.html

The screenshot shows the official Java API documentation for the `JdbcTemplate` class. The top navigation bar includes links for Overview, Package, Class (which is highlighted in orange), Use, Tree, Deprecated, Index, and Help. Below the navigation bar, there are links for Prev Class, Next Class, Frames, and No Frames. Summary and Detail sections provide links for Nested, Field, Constr, Method. The package name `org.springframework.jdbc.core` is listed. The class name `JdbcTemplate` is bolded. The inheritance chain is shown as `java.lang.Object` → `org.springframework.jdbc.support.JdbcAccessor` → `org.springframework.jdbc.core.JdbcTemplate`. The `All Implemented Interfaces:` section lists `InitializingBean` and `JdbcOperations`.

```
public class JdbcTemplate  
extends JdbcAccessor  
implements JdbcOperations
```

## 5.3 JdbcTemplate-基本介绍

- 基本介绍

1. 通过 Spring 可以配置数据源，从而完成对数据表的操作
2. `JdbcTemplate` 是 Spring 提供的访问数据库的技术。可以将 JDBC 的常用操作封装为模板方法。`[JdbcTemplate 类图]`。

JdbcTemplate		
m	JdbcTemplate()	
m	JdbcTemplate(DataSource)	
m	JdbcTemplate(DataSource, boolean)	
m	setIgnoreWarnings(boolean)	void
m	isIgnoreWarnings()	boolean
m	setFetchSize(int)	void
m	getFetchSize()	int
m	setMaxRows(int)	void
m	getMaxRows()	int
m	setQueryTimeout(int)	void
m	getQueryTimeout()	int
m	setSkipResultsProcessing(boolean)	void
m	isSkipResultsProcessing()	boolean
m	setSkipUndeclaredResults(boolean)	void
m	isSkipUndeclaredResults()	boolean
m	setResultsMapCaseInsensitive(boolean)	void
m	isResultsMapCaseInsensitive()	boolean
m	execute(ConnectionCallback<T>)	T
m	createConnectionProxy(Connection)	Connection
m	execute(StatementCallback<T>, boolean)	T
m	execute(StatementCallback<T>)	T

功能强大，彪悍的JdbcTemplate

## 5.4 JdbcTemplate 使用实例

### 5.4.1 需求说明

我们使用 spring 的方式来完成 JdbcTemplate 配置和使用

### 5.4.2 JdbcTemplate 使用-代码演示

#### 1. 引入使用 JdbcTemplate 需要的 jar 包



#### 2. 创建数据库 spring 和表 monster

-- 创建数据库

```
CREATE DATABASE spring
```

```
USE spring
```

-- 创建表 monster

```
CREATE TABLE monster(
```

```
id INT PRIMARY KEY,
```

```
`name` VARCHAR(64) NOT NULL DEFAULT "",
```

```
skill VARCHAR(64) NOT NULL DEFAULT ''  
 )CHARSET=utf8  
  
INSERT INTO monster VALUES(100, '青牛怪', '吐火');  
  
INSERT INTO monster VALUES(200, '黄袍怪', '吐烟');  
  
INSERT INTO monster VALUES(300, '蜘蛛怪', '吐丝');
```

### 3. 创建配置文件 src/jdbc.properties

```
jdbc.userName=root  
jdbc.password=hsp  
jdbc.driverClass=com.mysql.jdbc.Driver  
jdbc.url=jdbc:mysql://localhost:3306/spring
```

### 4. 创建配置文件 src/JdbcTemplate\_ioc.xml

```
<?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  
                           http://www.springframework.org/schema/beans/spring-beans.xsd  
                           http://www.springframework.org/schema/context  
                           http://www.springframework.org/schema/context"
```

```
http://www.springframework.org/schema/context/spring-context.xsd">

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

<!-- 配置数据源 -->
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <property name="user" value="${jdbc.userName}"></property>
    <property name="password" value="${jdbc.password}"></property>
    <property name="driverClass" value="${jdbc.driverClass}"></property>
    <property name="jdbcUrl" value="${jdbc.url}"></property>
</bean>
</beans>
```

5.

创

建

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\jdbctemplate\JdbcTemplateTest.java ,  
测试是否可以正确得到数据源

```
package com.hspedu.spring.jdbctemplate;

import com.hspedu.spring.beans.Monster;
import com.hspedu.spring.jdbctemplate.dao.MonsterDao;
import org.junit.jupiter.api.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
```

```
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 * @author 韩顺平
 * @version 1.0
 */

public class JdbcTemplateTest {
```

```
* 通过spring 提供的 JdbcTemplate 获取连接  
* @throws SQLException  
*/  
  
@Test  
  
public void testDataSourceByJdbcTemplate() throws SQLException {  
    ApplicationContext ioc = new ClassPathXmlApplicationContext("JdbcTemplate_ioc.xml");  
    DataSource dataSource = ioc.getBean(DataSource.class);  
    Connection connection = dataSource.getConnection();  
    System.out.println("connection= " + connection);  
    connection.close();  
}  
}
```

## 6. 配置 JdbcTemplate\_ioc.xml， 将数据源分配给 JdbcTemplate bean

```
<!-- 配置 JdbcTemplate -->  
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">  
    <!-- 将上面的数据源分配给 jdbcTemplate -->  
    <property name="dataSource" ref="dataSource"/>  
</bean>
```

## 7. 修改 JdbcTemplateTest.java， 添加一个新的 monster

```
/**  
 * 添加数据  
 * 添加一个新的 monster  
 */  
  
@Test  
public void addDataByJdbcTemplate() {  
    ApplicationContext ioc = new ClassPathXmlApplicationContext("JdbcTemplate_ioc.xml");  
    //得到 JdbcTemplate bean  
    JdbcTemplate bean = ioc.getBean(JdbcTemplate.class);  
    // 1. 添加方式 1  
    // String sql = "INSERT INTO monster VALUES(400, '红孩儿', '枪法厉害");  
    // bean.execute(sql);  
    //2. 添加方式 2, 绑定参数  
    String sql = "INSERT INTO monster VALUES(?, ?, ?)";  
    int affected = bean.update(sql, 700, "红孩儿 2", "枪法厉害 2");  
    System.out.println("add ok affected= " + affected);  
}
```

## 8. 修改 JdbcTemplateTest.java，更新一个 monster 的 skill

```
/**  
 * 修改数据
```

```
*/  
  
@Test  
  
public void updateDataByJdbcTemplate() {  
  
    ApplicationContext ioc = new ClassPathXmlApplicationContext("JdbcTemplate_ioc.xml");  
  
    //得到JdbcTemplate bean  
  
    JdbcTemplate bean = ioc.getBean(JdbcTemplate.class);  
  
    String sql = "UPDATE monster SET skill = ? WHERE id=?";  
  
    int affected = bean.update(sql, "美女计", 300);  
  
    System.out.println("affected= " + affected);  
  
    System.out.println("update data ok~");  
  
}  
  
9. 修改 JdbcTemplateTest.java，批量添加二个 monster 白蛇精和青蛇精
```

```
/**  
 * batch add data  
 * 批量添加二个 monster 白蛇精和青蛇精  
 */  
  
@Test  
  
public void addBatchDataByJdbcTemplate() {  
  
    ApplicationContext ioc = new ClassPathXmlApplicationContext("JdbcTemplate_ioc.xml");  
}
```

```
//得到JdbcTemplate bean  
JdbcTemplate bean = ioc.getBean(JdbcTemplate.class);  
  
String sql = "INSERT INTO monster VALUES(?, ?, ?);  
  
List<Object[]> param_list = new ArrayList<Object[]>();  
  
param_list.add(new Object[]{500, "白蛇精", "吃人"});  
param_list.add(new Object[]{600, "青蛇精", "吃小孩"});  
  
bean.batchUpdate(sql, param_list);  
  
System.out.println("batch add ok");  
  
}
```

## 10. 查询 id=100 的 monster 并封装到 Monster 实体对象

```
/**  
 * 查询 id=100 的 monster 并封装到 Monster 实体对象  
 */  
  
@Test  
  
public void selectDataByJdbcTemplate() {  
  
    ApplicationContext ioc = new ClassPathXmlApplicationContext("JdbcTemplate_ioc.xml");  
  
    //得到JdbcTemplate bean  
  
    JdbcTemplate bean = ioc.getBean(JdbcTemplate.class);  
  
    String sql = "SELECT id as monsterId, name, skill FROM monster WHERE id =?";  
  
    //下面这个rowmapper 是一个接口，可以将查询的结果，封装到你指定的 Monster 对象中。  
    RowMapper<Monster> rowMapper =
```

```
new BeanPropertyRowMapper<Monster>(Monster.class);

Monster monster = bean.queryForObject(sql, rowMapper, 100);

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

}
```

## 11. 查询 id>=200 的 monster 并封装到 Monster 实体对象

```
/**

 * 查询多条记录

 */

@Test

public void selectMulDataByJdbcTemplate() {

    ApplicationContext ioc = new ClassPathXmlApplicationContext("JdbcTemplate_ioc.xml");

    //得到JdbcTemplate bean

    JdbcTemplate bean = ioc.getBean(JdbcTemplate.class);

    String sql = "SELECT id as monsterId, name, skill FROM monster WHERE id >=?";

    //下面这个rowmapper 是一个接口，可以将查询的结果，封装到你指定的Monster 对象中

    RowMapper<Monster>           rowMapper           =           new

    BeanPropertyRowMapper<Monster>(Monster.class);

    List<Monster> monster_list = bean.query(sql, rowMapper, 200);

    for (Monster monster : monster_list) {

        System.out.println(monster);

    }

}
```

```
    }  
}
```

## 12. 查询返回结果只有一行一列的值，比如查询 id=100 的怪物名

```
/**  
 * 查询返回结果只有一行一列的值  
 */  
  
@Test  
public void selectScalarByJdbcTemplate() {  
    ApplicationContext ioc = new ClassPathXmlApplicationContext("JdbcTemplate_ioc.xml");  
    //得到JdbcTemplate bean  
    JdbcTemplate bean = ioc.getBean(JdbcTemplate.class);  
    String sql = "SELECT name FROM monster WHERE id =100";  
    String name = bean.queryForObject(sql, String.class);  
    System.out.println(name);  
}
```

## 13. 使用 Map 传入具名参数完成操作，比如添加 螃蟹精.:name 就是具名参数形式需要使用 NamedParameterJdbcTemplate 类

```
//修改 D:\idea_java_projects\spring5\src\JdbcTemplate_ioc.xml, 增加配置.
```

```
<!-- 配置 NamedParameterJdbcTemplate, 支持具名参数 -->
```

```
<bean id="namedParameterJdbcTemplate"
      class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate">
    <!-- 这里需要使用构造器关联数据源 -->
    <constructor-arg name="dataSource" ref="dataSource"/>
</bean>

/**
 * 使用 Map 传入具名参数完成操作，比如添加
 */
@Test
public void testDataByNamedParameterJdbcTemplate() {
    ApplicationContext ioc =
        new ClassPathXmlApplicationContext("JdbcTemplate_ioc.xml");
    //得到 NamedParameterJdbcTemplate bean
    NamedParameterJdbcTemplate namedParameterJdbcTemplate =
        ioc.getBean(NamedParameterJdbcTemplate.class);
    String sql = "INSERT INTO monster VALUES(:my_id, :name, :skill)";
    Map<String, Object> map_parameter = new HashMap<String, Object>();
    map_parameter.put("my_id", 800);
    map_parameter.put("name", "螃蟹精");
    map_parameter.put("skill", "钳子无敌大法");
    namedParameterJdbcTemplate.update(sql, map_parameter);
```

```
System.out.println("add data ok~");  
}
```

## 14. 使用 Sqlparametersource 来封装具名参数，还是添加一个 Monster 狐狸精

```
/**  
 * 使用 SqlParameterSource 传入具名参数完成操作，比如添加  
 */  
  
@Test  
public void operDataBySqlparametersorce() {  
    ApplicationContext ioc = new ClassPathXmlApplicationContext("JdbcTemplate_ioc.xml");  
    //得到 NamedParameterJdbcTemplate bean  
    NamedParameterJdbcTemplate namedParameterJdbcTemplate =  
        ioc.getBean(NamedParameterJdbcTemplate.class);  
    String sql = "INSERT INTO monster VALUES(:monsterId, :name, :skill)";  
    Monster monster = new Monster(900, "狐狸精", "媚狐之术");  
    SqlParameterSource source = new BeanPropertySqlParameterSource(monster);  
    namedParameterJdbcTemplate.update(sql, source);  
    System.out.println("add ok~");  
}
```

## 15. Dao 对象中使用 JdbcTemplate 完成对数据的操作

1)

创

建

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\jdbctemplate\dao\MonsterDao.java

```
package com.hspedu.spring.jdbctemplate.dao;

import com.hspedu.spring.beans.Monster;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

/**
 * @author 韩顺平
 * @version 1.0
 */
@Repository
public class MonsterDao {
    @Autowired
    private JdbcTemplate jdbcTemplate;
    //添加monster
    public void save(Monster monster) {
        String sql = "INSERT INTO monster VALUES(?, ?, ?)";
        jdbcTemplate.update(sql, monster.getMonsterId(),
            monster.getName(), monster.getSkill());
    }
}
```

}

## 2) 修改 D:\idea\_java\_projects\spring5\src\JdbcTemplate\_ioc.xml 增加扫描配置

```
<!-- 加入自动扫描包 -->  
<context:component-scan base-package="com.hspedu.spring.jdbcTemplate.dao"/>
```

3)

修

改

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\jdbcTemplate\JdbcTemplateTest.java,  
增加测试方法

```
/**  
 * 通过 dao 使用 jdbcTemplate 完成操作  
 */  
  
@Test  
public void operDataByDao() {  
    ApplicationContext ioc =  
        new ClassPathXmlApplicationContext("JdbcTemplate_ioc.xml");  
    MonsterDao bean = ioc.getBean(MonsterDao.class);  
    Monster monster = new Monster(1000, "大虾精", "夹子功");  
    bean.save(monster);  
}
```

## 5 声明式事务

### 5.1 事务分类

- 分类：

1. 编程式事务：

示意代码，传统方式

```
Connection connection = JdbcUtils.getConnection();
```

```
try {
```

```
    //1. 先设置事务不要自动提交
```

```
    connection.setAutoCommit(false);
```

```
    //2. 进行各种 crud
```

```
    //多个表的修改，添加，删除
```

```
    //3. 提交
```

```
    connection.commit();
```

```
} catch (Exception e) {
```

```
    //4. 回滚
```

```
conection.rollback();  
}
```

## 2. 声明式事务:

我们后面 以一个购买商品的系统为例来讲解

### 5.2 声明式事务-使用实例

#### 5.2.1 需求说明-用户购买商品

我们需要去处理用户购买商品的业务逻辑:分析: 当一个用户要去购买商品应该包含三个步骤

1. 通过商品 id 获取价格.
2. 购买商品(某人购买商品, 修改用户的余额.)
3. 修改库存量
4. 其实大家可以看到, 这时, 我们需要涉及到三张表商品表, 用户表, 商品存量表。 应该使用事务处理

#### 5.2.2 解决方案分析

1. 使用传统的编程事务来处理, 将代码写到一起[缺点: 代码冗余, 效率低, 不利于扩展, 优点是简单, 好理解]

```
Connection connection = JdbcUtils.getConnection();
```

```
try {
```

```
    //1. 先设置事务不要自动提交
```

```
    connection.setAutoCommit(false);
```

```
    //2. 进行各种 crud
```

```
    //多个表的修改，添加，删除
```

```
    select from 商品表 => 获取价格
```

```
    修改用户余额 update ...
```

```
    修改库存量 update
```

```
    //3. 提交
```

```
    connection.commit();
```

```
} catch (Exception e) {
```

```
    //4. 回滚
```

```
    connection.rollback();
```

```
}
```

2. 使用 Spring 的声明式事务处理， 可以将上面三个子步骤分别写成一个方法， 然后统一管理.

[这个是 Spring 很牛的地方，在开发使用的很多，优点是无代码冗余，效率高，扩展方便，缺点是理解较困难]

--> 底层使用 AOP (动态代理+动态绑定+反射+注解) => 看 Debug 源码..

### 5.2.3 声明式事务使用-代码实现

1. 先创建商品系统的数据库和表

-- 演示声明式事务创建的表

```
CREATE TABLE `user_account`(  
    user_id INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,  
    user_name VARCHAR(32) NOT NULL DEFAULT "",  
    money DOUBLE NOT NULL DEFAULT 0.0  
)CHARSET=utf8;
```

```
INSERT INTO `user_account` VALUES(NULL,'张三', 1000);
```

```
INSERT INTO `user_account` VALUES(NULL,'李四', 2000);
```

```
CREATE TABLE `goods`(  
    ...
```

```
goods_id INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,  
goods_name VARCHAR(32) NOT NULL DEFAULT "",  
price DOUBLE NOT NULL DEFAULT 0.0  
)CHARSET=utf8 ;  
  
INSERT INTO `goods` VALUES(NULL,'小风扇', 10.00);  
  
INSERT INTO `goods` VALUES(NULL,'小台灯', 12.00);  
  
INSERT INTO `goods` VALUES(NULL,'可口可乐', 3.00);  
  
CREATE TABLE `goods_amount`(  
goods_id INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,  
goods_num INT UNSIGNED DEFAULT 0  
)CHARSET=utf8 ;  
  
INSERT INTO `goods_amount` VALUES(1,200);  
  
INSERT INTO `goods_amount` VALUES(2,20);  
  
INSERT INTO `goods_amount` VALUES(3,15);
```

## 2. 创建 D:\idea\_java\_projects\spring5\src\com\hspedu\spring\tx\dao\GoodsDao.java

```
package com.hspedu.spring.tx.dao;
```

```
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.jdbc.core.JdbcTemplate;  
import org.springframework.stereotype.Repository;  
  
/**  
 * @author 韩顺平  
 * @version 1.0  
 */  
@Repository  
public class GoodsDao {  
    @Autowired  
    private JdbcTemplate jdbcTemplate;  
    public Float queryPriceById(Integer id) {  
        String sql = "SELECT price From goods Where goods_id=?";  
        Float price = jdbcTemplate.queryForObject(sql, Float.class, id);  
        return price;  
    }  
    public void updateBalance(Integer user_id, Float money) {  
        String sql = "UPDATE user_account SET money=money-? Where user_id=?";  
        jdbcTemplate.update(sql, money, user_id);  
    }  
}
```

```
public void updateAmount(Integer goods_id, int amount){  
    String sql = "UPDATE goods_amount SET goods_num=goods_num-? Where  
    goods_id=?";  
    jdbcTemplate.update(sql, amount , goods_id);  
}  
}
```

### 3. 创建 src\tx\_ioc.xml 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"  
       xmlns:tx="http://www.springframework.org/schema/tx"  
       xmlns:aop="http://www.springframework.org/schema/aop"  
       xsi:schemaLocation="http://www.springframework.org/schema/beans  
                           http://www.springframework.org/schema/beans/spring-beans.xsd  
                           http://www.springframework.org/schema/context  
                           http://www.springframework.org/schema/context/spring-context.xsd  
                           http://www.springframework.org/schema/tx  
                           http://www.springframework.org/schema/tx/spring-tx.xsd
```

<http://www.springframework.org/schema/aop>

[>](http://www.springframework.org/schema/aop/spring-aop.xsd)

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

<!-- 配置数据源 -->
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <property name="user" value="${jdbc.userName}"></property>
    <property name="password" value="${jdbc.password}"></property>
    <property name="driverClass" value="${jdbc.driverClass}"></property>
    <property name="jdbcUrl" value="${jdbc.url}"></property>
</bean>

<!-- 配置 JdbcTemplate -->
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <!-- 将上面的数据源分配给 jdbcTemplate -->
    <property name="dataSource" ref="dataSource"/>
</bean>

<!-- 配置事务管理器 -->
<bean id="dataSourceTransactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>

<!-- 开启基于注解的声明式事务功能 -->
```

```
<tx:annotation-driven transaction-manager="dataSourceTransactionManager"/>

<!-- 加入自动扫描包 dao -->

<context:component-scan base-package="com.hspedu.spring.tx.dao"/>

</beans>
```

#### 4. 创建 D:\idea\_java\_projects\spring5\src\com\hspedu\spring\tx\TxTest.java, 完成对 GoodsDao 的测试

```
package com.hspedu.spring.tx;

import com.hspedu.spring.beans.Monster;
import com.hspedu.spring.jdbcTemplate.dao.MonsterDao;
import com.hspedu.spring.tx.dao.GoodsDao;
import com.hspedu.spring.tx.service.GoodsService;
import com.hspedu.spring.tx.service.MultiplyTxService;
import org.junit.jupiter.api.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
```

```
import org.springframework.jdbc.core.namedparam.SqlParameterSource;  
  
import javax.sql.DataSource;  
import java.sql.Connection;  
import java.sql.SQLException;  
import java.util.ArrayList;  
import java.util.HashMap;  
import java.util.List;  
import java.util.Map;  
  
/**  
 * @author 韩顺平  
 * @version 1.0  
 */  
public class TxTest {  
    /**  
     * 查询商品的价格  
     */  
    @Test  
    public void queryPriceByIdTest() {  
        ApplicationContext ioc = new ClassPathXmlApplicationContext("tx_ioc.xml");  
        System.out.println("查询价格");
```

```
GoodsDao bean = ioc.getBean(GoodsDao.class);

    Float price = bean.queryPriceById(1);

    System.out.println(" 1 号商品的价格= " + price);

}

/** 
 * 修改用户余额(购买商品后)
 */
@Test
public void updateBalanceTest() {
    ApplicationContext ioc = new ClassPathXmlApplicationContext("tx_ioc.xml");

    GoodsDao bean = ioc.getBean(GoodsDao.class);

    bean.updateBalance(1, 30.5f);

    System.out.println("====修改余额成功====");

}

/** 
 * 测试修改商品的库存量
 */
@Test
public void updateAmountTest() {
    ApplicationContext ioc = new ClassPathXmlApplicationContext("tx_ioc.xml");

    GoodsDao bean = ioc.getBean(GoodsDao.class);
```

```
        bean.updateAmount(1, 2);

        System.out.println("====修改商品库存量成功====");

    }

}
```

5. 创建 D:\idea\_java\_projects\spring5\src\com\hspedu\spring\tx\service\GoodsService.java, 编写方法，验证不使用事务就会出现数据不一致现象.

### 1) 创建 GoodsService.java

```
package com.hspedu.spring.tx.service;

import com.hspedu.spring.tx.dao.GoodsDao;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.TransactionDefinition;
import org.springframework.transaction.annotation.Isolation;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

/**
 * @author 韩顺平
 * @version 1.0
 */
```

```
@Service
```

```
public class GoodsService {
```

```
@Autowired
```

```
private GoodsDao goodsDao;
```

```
/**
```

```
* 购买商品[没有使用事务]
```

```
* @param user_id
```

```
* @param goods_id
```

```
* @param num
```

```
*/
```

```
public void buyGoods(int user_id, int goods_id, int num) {
```

```
//查询到商品价格
```

```
Float goods_price = goodsDao.queryPriceById(goods_id);
```

```
//购买商品，减去余额
```

```
goodsDao.updateBalance(user_id, goods_price * num);
```

```
///老韩：模拟一个异常，会发生数据库数据不一致现象
```

```
// int i = 10 / 0;
```

```
//更新库存
```

```
goodsDao.updateAmount(goods_id, num);
```

```
    }  
}
```

## 2) 修改 D:\idea\_java\_projects\spring5\src\tx\_ioc.xml, 加入对 Service 的扫描

```
<!-- 加入自动扫描包 service -->  
<context:component-scan base-package="com.hspedu.spring.tx.service"/>
```

## 6. 测试 TxTest.java, 增加方法, 这时没有加入事务控制

```
/**  
 * 测试购买商品  
 */  
  
@Test  
public void buyGoodsTest() {  
  
    ApplicationContext ioc = new ClassPathXmlApplicationContext("tx_ioc.xml");  
    GoodsService bean = ioc.getBean(GoodsService.class);  
    bean.buyGoods(1, 2, 1);  
    System.out.println("====购买商品成功====");  
}
```

## 7. 修改 GoodsService.java 并测试

### 1) 修改 GoodsService.java, 增加测试方法, 加入声明式事务注解

```
/*
 * 购买商品(使用spring 的声明式事务)
 * 默认事务是传播属性:REQUIRED
 * @param user_id
 * @param goods_id
 * @param num
 */
@Transactional
public void buyGoodsByTx(int user_id, int goods_id, int num) {
    //查询到商品价格
    Float goods_price = goodsDao.queryPriceById(goods_id);
    //购买商品, 减去余额
    goodsDao.updateBalance(user_id, goods_price * num);
    ///老韩: 模拟一个异常, 会发生数据库数据不一致现象
    //int i = 10 / 0;
    //更新库存
    goodsDao.updateAmount(goods_id, num);
}
```

2) 修改 TxTest.java, 增加测试方法, 对声明式事务进行测试, 看看是否保证了数据一致性

```
/*
 * 测试购买商品(使用了声明式事务)
 */

@Test
public void buyGoodsByTxTest() {

    ApplicationContext ioc = new ClassPathXmlApplicationContext("tx_ioc.xml");
    GoodsService bean = ioc.getBean(GoodsService.class);
    //使用buyGoodsByTx()
    bean.buyGoodsByTx(1, 2, 1);
    System.out.println("====购买商品成功====");
}
```

#### 5.2.4 声明式事务机制-Debug

### 5.3 事务的传播机制

#### 5.3.1 事务的传播机制说明

1. 当有多个事务处理并存时, 如何控制?
2. 比如用户去购买两次商品(使用不同的方法), 每个方法都是一个事务, 那么如何控制呢?

### 3. 这个就是事务的传播机制，看一个具体的案例(如图)

```
1 /**
 * 多个事务控制的机制-事务传播机制
 */
@Test
public void test06(){
    MultiTxService bean = ioc.getBean(MultiTxService.class);
    bean.multiTxTest();
}
```

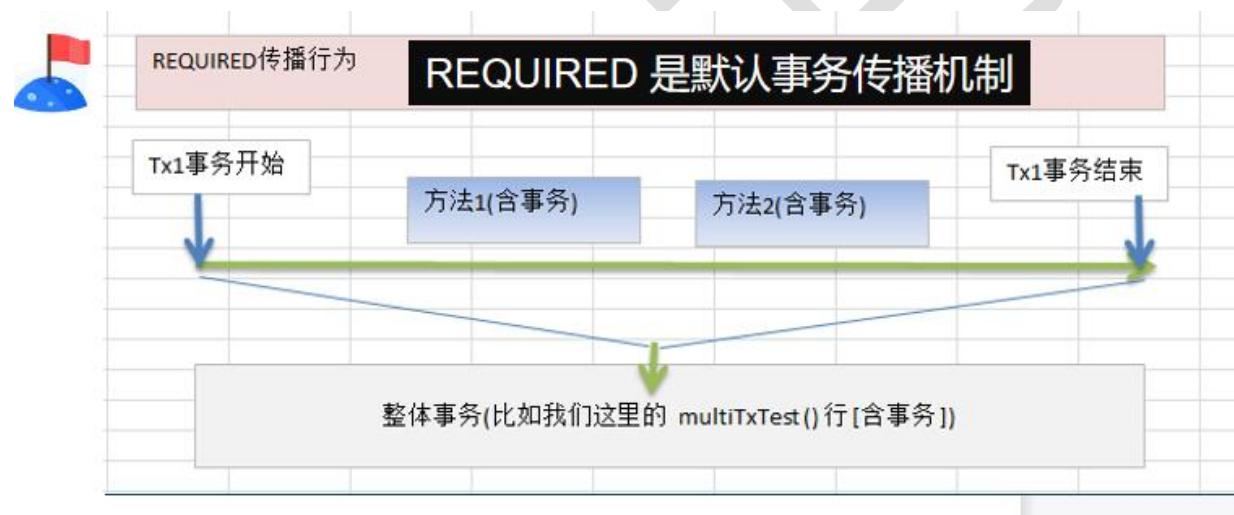
```
@Transactional
public void multiTxTest(){
    //2号用户购买2号商品，购买数量是1个。
    goodService.buyGoods(2, 2, 1);
    //1号用用户1号商品，购买数量是1个。
    goodService.buyGoods2(1, 1, 1);
}
```

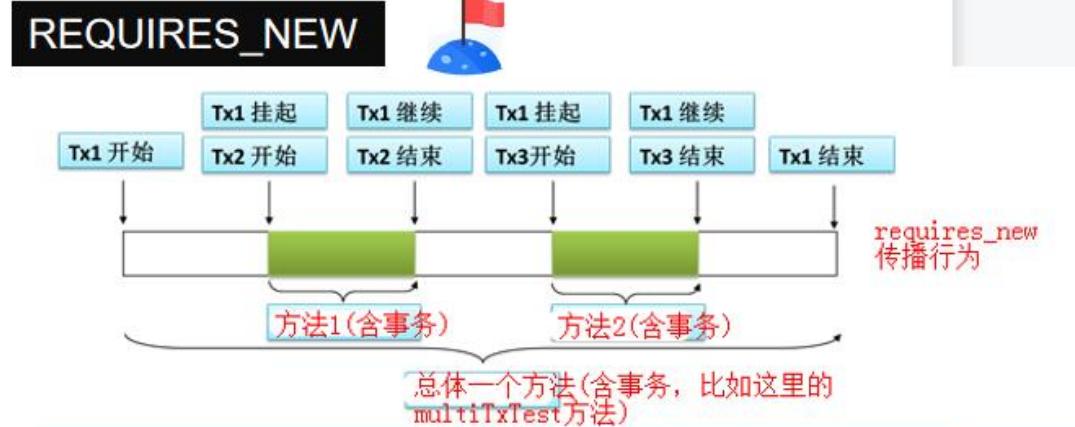
#### 5.3.2 事务传播机制种类

- 事务传播的属性/种类一览图

传播属性	描述
REQUIRED	如果有事务在运行，当前的方法就在这个事务内运行，否则，就启动一个新的事务，并在自己的事务内运行
REQUIRES_NEW	当前的方法必须启动新事务，并在它自己的事务内运行。如果有事务正在运行，应该将它挂起
SUPPORTS	如果有事务在运行，当前的方法就在这个事务内运行。否则它可以不运行在事务中。
NOT_SUPPORTED	当前的方法不应该运行在事务中。如果有运行的事务，将它挂起
MANDATORY	当前的方法必须运行在事务内部，如果没有正在运行的事务，就抛出异常
NEVER	当前的方法不应该运行在事务中。如果有运行的事务，就抛出异常
NESTED	如果有事务在运行，当前的方法就应该在这个事务的嵌套事务内运行。否则，就启动一个新的事务，并在它自己的事务内运行。

- 事务传播的属性/种类机制分析，重点分析了 REQUIRED 和 REQUIRES\_NEW 两种事务传播属性，其它知道即可(看上图)





- 事务的传播机制的设置方法

```
@Transactional(propagation=Propagation.REQUIRES_NEW)
[ public void buyGoods2(int user_id, int goods_id, int num){
```

- REQUIRES\_NEW 和 REQUIRED 在处理事务的策略

```
@Transactional
public void multiTxTest(){
    //2号用户购买2号商品，购买数量是1个。
    goodService.buyGoods(2, 2, 1);
    //1号用户购买1号商品，购买数量是1个。
    goodService.buyGoods2(1, 1, 1);
}
```

1. 如果设置为 REQUIRES\_NEW

buyGoods2 如果错误，不会影响到 buyGoods() 反之亦然，即它们的事务是独立的。

2. 如果设置为 REQUIRED

buyGoods2 和 buyGoods 是一个整体，只要有方法的事务错误，那么两个方法都不

会执行成功。!

### 5.3.3 事务的传播机制-应用实例

- 事务的传播机制需要说明

1. 比如用户去购买两次商品(使用不同的方法), 每个方法都是一个事务, 那么如何控制呢?  
=>这个就是事务的传播机制

2. 看一个具体的案例(用 required/requires\_new 来测试):

- 事务的传播机制-代码实现

1. 修改 GoodsDao.java, 增加方法

//为了测试事务的传播机制, 再准备一套方法.

```
public Float queryPriceById02(Integer id) {  
  
    String sql = "SELECT price From goods Where goods_id=?";  
  
    Float price = jdbcTemplate.queryForObject(sql, Float.class, id);  
  
    return price;  
}  
  
public void updateBalance02(Integer user_id, Float money) {
```

//老韩说明：这里，故意写错sql语句

```
String sql = "UPDATE user_account SET money=money-? Where user_id=?";  
jdbcTemplate.update(sql, money, user_id);
```

```
}
```

```
/**
```

```
* 修改商品库存量
```

```
* @param goods_id
```

```
* @param amount
```

```
*/
```

```
public void updateAmount02(Integer goods_id, int amount){
```

```
    String sql = "UPDATE goods_amount SET goods_num=goods_num-? Where  
    goods_id=?";
```

```
    jdbcTemplate.update(sql, amount, goods_id);
```

```
}
```

## 2. 修改 GoodsService.java 增加 buyGoodsByTx02(), 使用默认的传播机制

```
/**
```

```
* 购买商品(使用spring 的声明式事务[讲解事务])
```

```
* 默认事务是传播属性:REQUIRED
```

```
* @param user_id  
* @param goods_id  
* @param num  
*/  
  
@Transactional  
  
public void buyGoodsByTx02(int user_id, int goods_id, int num) {  
  
    //查询到商品价格  
    Float goods_price = goodsDao.queryPriceById02(goods_id);  
  
    //购买商品，减去余额  
    goodsDao.updateBalance02(user_id, goods_price * num);  
  
    //更新库存  
    goodsDao.updateAmount02(goods_id, num);  
  
}
```

3.

创

建

D:\idea\_java\_projects\spring5\src\com\hspedu\spring\tx\service\MultiplyTxService.java

```
package com.hspedu.spring.tx.service;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;
```

```
import org.springframework.transaction.annotation.Transactional;
```

```
/**  
 * @author 韩顺平  
 * @version 1.0  
 */  
  
@Service  
  
public class MultiplyTxService {  
  
    @Autowired  
    private GoodsService goodService;  
  
    @Transactional  
    public void multiTxTest() {  
  
        //2号用户购买2号商品，购买数量是1个。  
        goodService.buyGoodsByTx(2, 2, 1);  
  
        //1号用用户1号商品，购买数量是1个。  
        goodService.buyGoodsByTx02(1, 1, 1);  
    }  
}
```

4. 测试 TxTest.java，可以验证：为 REQUIRED buyGoodsByTx 和 buyGoodsByTx02 是整体，

只要有方法的事务错误，那么两个方法都不会执行成功

```
/*
 * 测试购买商品(使用了声明式事务， 测试事务的传播机制)
 */

@Test
public void buyGoodsByMulTxTest() {

    ApplicationContext ioc = new ClassPathXmlApplicationContext("tx_ioc.xml");
    MultiplyTxService bean = ioc.getBean(MultiplyTxService.class);
    bean.multiTxTest();
    System.out.println("-----ok-----");

}

public void updateBalance02(Integer user_id, Float money) {
    //老韩说明：这里，故意写错sql语句
    String sql = "UPDATEXX user_account SET money=money-? Where user_id=?";
    jdbcTemplate.update(sql, money, user_id);
}

```

**GoodsDao.java**

5. 修改 GoodsService.java， 将传播机制改成 REQUIRES\_NEW 可以验证：设置为 REQUIRES\_NEW buyGoodsByTx 如果错误，不会影响到 buyGoodsByTx02() 反之亦然，也就是说它们的事务是独立的，完成测试。

```
@Transactional(propagation = Propagation.REQUIRES_NEW)
public void buyGoodsByTx(int user_id, int goods_id, int num){

    // 查询到商品价格
    Float goods_price = goodsDao.queryPriceById(goods_id);

    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void buyGoodsByTx02(int user_id, int goods_id, int num){

        // 查询到商品价格
    }
}
```

## 5.4 事务的隔离级别

### 5.4.1 事务隔离级别说明

- 事务隔离级别的概念在 mysql 讲过，就不再说了，在 Mysql 的 799 讲

隔离级别	脏读	不可重 复读	幻读	加锁读
读未提交 (Read uncommitted)	V	V	V	不加锁
读已提交 (Read committed)	X	V	V	不加锁
可重复读 (Repeatable read)	X	X	X 【SQL92 标准有，mysql数据库改进了，解决了这个级别的幻读问题】	不加锁
可串行化 (Serializable) [演示重开客户端]	X	X	X	加锁

- 事务隔离级别说明

1. 默认的隔离级别，就是 mysql 数据库默认的隔离级别 一般为 REPEATABLE\_READ

2. 看源码可知 `Isolation.DEFAULT` 是 : Use the default isolation level of the underlying datastore

3. 查看数据库默认的隔离级别 `SELECT @@global.tx_isolation`

#### 5.4.2 事务隔离级别的设置和测试

1. 修改 `GoodsService.java`, 先测默认隔离级别, 增加方法 `buyGoodsByTxISOLATION()`

```
@Transactional(propagation = Propagation.REQUIRES_NEW)
public void buyGoodsByTxISOLATION(int user_id, int goods_id, int num) {
    // 查询到商品价格
    Float goods_price = goodsDao.queryPriceById(goods_id);
    System.out.println("第一次读取的价格 = " + goods_price);
    // 测试一下隔离级别, 在同一个事务中, 查询一下价格
    goods_price = goodsDao.queryPriceById(goods_id);
    System.out.println("第二次读取的价格 = " + goods_price)
}
```

测试默认的隔离级别:  
`REPEATABLE_READ` 可重复读

1. 下一个断点  
2. 执行到这里时, 在 `SQLyog` 修改对应商品价格  
3. 然后看下第三次读取到的价格情况

```
/*
 * 测试事务的隔离级别
 *
 * 1. 默认的隔离级别, 就是 mysql 数据库默认的隔离级别 一般为 REPEATABLE_READ
 * 2. 看源码可知 Use the default isolation level of the underlying datastore
 *
 * public enum Isolation {
 *
 * <p>
 *
 * /**
 *
 * * * Use the default isolation level of the underlying datastore.
 *
 * * * All other levels correspond to the JDBC isolation levels.
 *
```

```
* * @see java.sql.Connection
* DEFAULT(TransactionDefinition.ISOLATION_DEFAULT)}
* 3. 查看数据库默认的隔离级别 SELECT @@global.tx_isolation
* @param user_id
* @param goods_id
* @param num
*/
@Transactional(propagation = Propagation.REQUIRES_NEW)
public void buyGoodsByTxISOLATION(int user_id, int goods_id, int num) {
    //查询到商品价格
    Float goods_price = goodsDao.queryPriceById(goods_id);
    System.out.println("第一次读取的价格 = " + goods_price);
    //测试一下隔离级别，在同一个事务中，查询一下价格
    goods_price = goodsDao.queryPriceById(goods_id);
    System.out.println("第二次读取的价格 = " + goods_price);
}
```

56  
57 UPDATE goods SET price=7 WHERE goods\_id=1  
58 |  
59

goods_id	goods_name	price
1	小风扇	7
2	小台灯	12
3	可口可乐	3

2. 完成测试，修改 D:\idea\_java\_projects\spring5\src\com\hspedu\spring\tx\TxTest.java，增加测试方法，默认隔离级别下，两次读取到的价格是一样的，不会受到 SQLyog 修改影响

```
/*
 * 测试购买商品(使用了声明式事务， 测试事务隔离级别)
 */

@Test
public void buyGoodsByTxISOLATIONTest() {
    ApplicationContext ioc = new ClassPathXmlApplicationContext("tx_ioc.xml");
    GoodsService bean = ioc.getBean(GoodsService.class);
    bean.buyGoodsByTxISOLATION(1, 1, 1);
    System.out.println("-----ok-----");
}
```

3. 修改 GoodsService.java，测试 READ\_COMMITTED 隔离级别情况

```

@Transactional(propagation = Propagation.REQUIRES_NEW, isolation = Isolation.READ_COMMITTED)
public void buyGoodsByTxISOLATION(int user_id, int goods_id, int num) {
    // 查询到商品价格
    Float goods_price = goodsDao.queryPriceById(goods_id);
    System.out.println("第一次读取的价格 = " + goods_price);
    // 测试一下隔离级别，在同一个事务中，查询一下价格
    goods_price = goodsDao.queryPriceById(goods_id);
    System.out.println("第二次读取的价格 = " + goods_price);
}

```

1. 下断点  
2. 执行到这里时，使用SQLyog修改对应商品价格  
3. 看看第二次读取到的商品价格是什么情况

```

56
57 UPDATE goods SET price=7 WHERE goods_id=1
58
59

```

注意：1. 在SQLyog中，默认是自动提交

goods_id	goods_name	price
1	小风扇	7
2	小台灯	12
3	可口可乐	3

4. 完成测试，使用前面已经创建好的测试方法，在 READ\_COMMITTED 隔离级别下，两次读取到的价格会受到 SQLyog 修改影响

```

/**
 * 测试购买商品(使用了声明式事务， 测试事务隔离级别)
 */

```

**@Test**

```

public void buyGoodsByTxISOLATIONTest() {

```

```

ApplicationContext ioc = new ClassPathXmlApplicationContext("tx_ioc.xml");

```

```
GoodsService bean = ioc.getBean(GoodsService.class);
bean.buyGoodsByTxISOLATION(1, 1, 1);
System.out.println("-----ok-----");
}
```

### 5.4.3 事务的超时回滚

- 基本介绍
  - 1. 如果一个事务执行的时间超过某个时间限制，就让该事务回滚。
  - 2. 可以通过设置事务超时回顾来实现
- 基本语法

```
@Transactional(
    propagation=Propagation.REQUIRES_NEW,
    isolation=Isolation.READ_COMMITTED,
    timeout=2)
public void buyGoods(int user_id, int goods_id, int num){
```

- 超时回滚-代码实现

#### 1. 修改 GoodsService.java ， 增加 buyGoodsByTxTimeout()

```
/**
 * 测试事务的超时回滚(超时时间，我们设置为2秒)
 * @param user_id
```

```
* @param goods_id  
* @param num  
*/  
  
@Transactional(timeout = 2)  
public void buyGoodsByTxTimeout(int user_id, int goods_id, int num) {  
    //查询到商品价格  
    Float goods_price = goodsDao.queryPriceById02(goods_id);  
    //购买商品，减去余额  
    goodsDao.updateBalance02(user_id, goods_price * num);  
    System.out.println("====超时 start====");  
    try {  
        Thread.sleep(4000);  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
    System.out.println("====超时 end====");  
    //更新库存  
    goodsDao.updateAmount02(goods_id, num);  
}
```

## 2. 测试 TxTest.java, 增加测试方法

会抛出异常，事务滚回，原来的操作将会撤销

Transaction timed out: deadline was Sat Jul 10 22:20:51

```
/*
 * 测试购买商品(使用了声明式事务， 测试事务超时回滚)
 */
@Test
public void buyGoodsByTxTimeoutTest() {
    ApplicationContext ioc = new ClassPathXmlApplicationContext("tx_ioc.xml");
    GoodsService bean = ioc.getBean(GoodsService.class);
    bean.buyGoodsByTxTimeout(1, 1, 1);
    System.out.println("-----ok-----");
}
```

## 5.5 声明式事务-课后练习

- 要求

模拟一个用户银行转账购买淘宝商品的的业务模块，数据表/dao/service 自己设计()

seller[卖家]

主讲人 : 韩顺平老师

韩顺平 Java 工程师

buyer[买家]

goods[商品表[库存量]]

taoBao[提取入账成交额的 10%]

要求. 使用注解完成即可

