第一部分：Spring的IOC

一、设计模式-工厂模式

工厂模式是我们最常用的实例化对象模式了，它是用工厂中的方法代替new创建对象的一种设计模式。

我们以Mybatis的SqlSession接口为例，它有一个实现类DefaultSqlSession，如果我们要创建该接口的实例对象:SqlSession sqlSession=new DefaultSqlSession();

可是，实际情况是，通常我们都要在创建SqlSession实例时做点初始化的工作,比如解析XML,封装连接数据库的信息等等。

在创建对象时，如果有一些不得不做的初始化操作时，我们首先到的是，可以使用构造函数，这样生成实例就写成:SqlSession sqlSession=new DefaultSqlSession(传入配置文件的路径);

但是，如果创建sqlSession实例时所做的初始化工作不是像赋值这样简单的事，可能是很长一段代码，如果也写入构造函数中，那你的代码很难看了（就需要Refactor重构）。

为什么说代码很难看，初学者可能没有这种感觉，我们分析如下，初始化工作如果是很长一段代码，说明要做的工作很多，将很多工作装入一个方法中，相当于将很多鸡蛋放在一个篮子里，是很危险的，这也是有悖于Java面向对象的原则，面向对象的封装(Encapsulation)和分派(Delegation)告诉我们，尽量将长的代码分派“切割”成每段，将每段再“封装”起来(减少段和段之间耦合联系性)，这样，就会将风险分散，以后如果需要修改，只要更改每段，不会再发生牵一动百的事情。

所以，Mybatis框架在使用时为我们提供了SqlSessionFactory工厂类，通过方法获取到SqlSession对象。同时方法有很多重载，用于实现不同的需求。这个方法就是openSession()，它支持传入Connection参数来保证连接的一致性；支持传入true|false来保证事务的提交时机等等。

二、IOC和DI

1、IOC-Inversion of Control

它的含义为：控制反转。它不是一个技术，而是一种思想。其作用是用于削减代码间的耦合。它的实现思想就是利用了工厂设计模式，把创建对象代码从具体类中剥离出去，交由工厂来完成，从而降低代码间的依赖关系。

耦合有如下分类：

（1） 内容耦合。当一个模块直接修改或操作另一个模块的数据时，或一个模块不通过正常入口而转入另一个模块时，这样的耦合被称为内容耦合。内容耦合是最高程度的耦合，应该避免使用之。

（2） 公共耦合。两个或两个以上的模块共同引用一个全局数据项，这种耦合被称为公共耦合。在具有大量公共耦合的结构中，确定究竟是哪个模块给全局变量赋了一个特定的值是十分困难的。

（3） 外部耦合 。一组模块都访问同一全局简单变量而不是同一全局数据结构，而且不是通过参数表传递该全局变量的信息，则称之为外部耦合。

（4） 控制耦合 。一个模块通过接口向另一个模块传递一个控制信号，接受信号的模块根据信号值而进行适当的动作，这种耦合被称为控制耦合。

（5） 标记耦合 。若一个模块A通过接口向两个模块B和C传递一个公共参数，那么称模块B和C之间存在一个标记耦合。

（6） 数据耦合。模块之间通过参数来传递数据，那么被称为数据耦合。数据耦合是最低的一种耦合形式，系统中一般都存在这种类型的耦合，因为为了完成一些有意义的功能，往往需要将某些模块的输出数据作为另一些模块的输入数据。

（7） 非直接耦合 。两个模块之间没有直接关系，它们之间的联系完全是通过主模块的控制和调用来实现的。

为什么要解耦：

耦合是影响软件复杂程度和设计质量的一个重要因素，在设计上我们应采用以下原则：如果模块间必须存在耦合，就尽量使用数据耦合，少用控制耦合，限制公共耦合的范围，尽量避免使用内容耦合。

2、DI-Dependency Injection

没有依赖的情况。ioc解耦只是降低他们的依赖关系，但不会消除。例如：我们的业务层仍会调用持久层的方法。
那这种业务层和持久层的依赖关系，在使用spring之后，就让spring来维护了。
简单的说，就是坐等框架把持久层对象传入业务层，而不用我们自己去获取。

三、Spring注解驱动开发入门

1、写在最前

spring在2.5版本引入了注解配置的支持，同时从Spring 3版本开始，Spring JavaConfig项目提供的许多特性成为核心Spring框架的一部分。因此，可以使用Java而不是XML文件来定义应用程序类外部的bean。在这里面官方文档为我们提供了四个基本注解@Configuration,@Bean,@Import,@DependsOn

- Annotation-based configuration: Spring 2.5 introduced support for annotation-based configuration metadata.
- Java-based configuration: Starting with Spring 3.0, many features provided by the Spring JavaConfig project became part of the core Spring Framework. Thus, you can define beans external to your application classes by using Java rather than XML files. To use these new features, see the `@Configuration`, `@Bean`, `@Import`, and `@DependsOn` annotations.

2、注解驱动入门案例介绍

1.需求：
实现保存一条数据到数据库。
2.表结构：

```
create table account(
    id int primary key auto_increment,
    name varchar(50),
    money double(7,2)
);
```

3.要求：
使用spring框架中的JdbcTemplate和DriverManagerDataSource
使用纯注解配置spring的ioc

3、案例实现

3.1、导入坐标

```
<dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.1.6.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>5.1.6.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.45</version>
```

3.2、编写配置类

```
/**
 * spring的配置类
 * 用于替代xml配置
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
@Configuration
@Import(JdbcConfig.class)
@PropertySource("classpath:jdbc.properties")
public class SpringConfiguration {
}


/**
 * 连接数据库的配置
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
public class JdbcConfig {

    @Value("${jdbc.driver}")
    private String driver;
    @Value("${jdbc.url}")
    private String url;
    @Value("${jdbc.username}")
    private String username;
    @Value("${jdbc.password}")
    private String password;


    @Bean("jdbcTemplate")
    public JdbcTemplate createJdbcTemplate(DataSource dataSource){
        return new JdbcTemplate(dataSource);
    }


    @Bean("dataSource")
    public DataSource createDataSource(){
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName(driver);
        dataSource.setUrl(url);
        dataSource.setUsername(username);
        dataSource.setPassword(password);
        return dataSource;
    }
}
```

3.3、编写配置文件

```
jdbc.username=root
jdbc.password=1234
```

3.4、编写测试类

```
/**
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
public class SpringAnnotationDrivenTest {

    /**
     * 测试
     * @param args
     */
    public static void main(String[] args) {
        //1.获取容器
        ApplicationContext ac = new
AnnotationConfigApplicationContext(SpringConfiguration.class);
        //2.根据id获取对象
        JdbcTemplate jdbcTemplate =
ac.getBean("jdbcTemplate",JdbcTemplate.class);
        //3.执行操作
        jdbcTemplate.update("insert into
account(name,money)values(?,?)","test",12345);
    }
}
```

四、IOC的常用注解分析

1、用于注解驱动的注解

1.1、@Configuration

1.1.1、源码

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Configuration {

    /**
     * Explicitly specify the name of the Spring bean definition associated with
the
     * {@code @Configuration} class. If left unspecified (the common case), a
bean
     * name will be automatically generated.
     * <p>The custom name applies only if the {@code @Configuration} class is
picked
     * up via component scanning or supplied directly to an
     * {@link AnnotationConfigApplicationContext}. If the {@code @Configuration}
class
```

```
     * element will take precedence.
     * @return the explicit component name, if any (or empty String otherwise)
     * @see org.springframework.beans.factory.support.DefaultBeanNameGenerator
     */
    @AliasFor(annotation = Component.class)
    String value() default "";
}
```

1.1.2、说明

作用：

它是在spring3.0版本之后加入的。此注解是spring支持注解驱动开发的一个标志。表明当前类是spring的一个配置类，作用是替代spring的applicationContext.xml。但其本质就是@Component注解，被此注解修饰的类，也会被存入spring的ioc容器。

属性：

value:

用于存入spring的Ioc容器中Bean的id。

使用场景：

在注解驱动开发时，用于编写配置的类，通常可以使用此注解。一般情况下，我们的配置也会分为主从配置，@Configuration一般出现在主配置类上。例如，入门案例中的SpringConfiguration类上。值得注意的是，如果我们在注解驱动开发时，构建ioc容器使用的是传入字节码的构造函数，此注解可以省略。但是如果传入的是一个包，此注解则不能省略。

1.1.3、示例

在注解驱动的入门案例中，由于没有了applicationContext.xml，就没法在xml中配置spring创建容器要扫描的包了。那么，我们自己写的一些类，通过注解配置到ioc容器中也无法实现了。此时就可以使用此注解来代替spring的配置文件。

```
/**
 * spring的配置类
 * 用于替代xml配置
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
@Configuration
@Import(JdbcConfig.class)
@PropertySource("classpath:jdbc.properties")
@ComponentScan("com.itheima")
public class SpringConfiguration {
}
```

1.2、@ComponentScan

1.2.1、源码

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
@Repeatable(ComponentScans.class)
public @interface ComponentScan {
    /**
     */
    @AliasFor("basePackages")
    String[] value() default {};
```

```
     */
    @AliasFor("value")
    String[] basePackages() default {};

    /**
     */
    Class<?>[] basePackageClasses() default {};

    /**
     */
    Class<? extends BeanNameGenerator> nameGenerator() default
BeanNameGenerator.class;

    /**
     */
    Class<? extends ScopeMetadataResolver> scopeResolver() default
AnnotationScopeMetadataResolver.class;

    /**
     */
    ScopedProxyMode scopedProxy() default ScopedProxyMode.DEFAULT;

    /**
     */
    String resourcePattern() default
ClassPathScanningCandidateComponentProvider.DEFAULT_RESOURCE_PATTERN;

    /**
     */
    boolean useDefaultFilters() default true;

    /**
     */
    Filter[] includeFilters() default {};

    /**
     */
    Filter[] excludeFilters() default {};

    /**
     */
    boolean lazyInit() default false;


    /**
     * Declares the type filter to be used as an {@linkplain
ComponentScan#includeFilters
     * include filter} or {@linkplain ComponentScan#excludeFilters exclude
filter}.
     */
    @Retention(RetentionPolicy.RUNTIME)
    @Target({})
    @interface Filter {
        /**
         */
        FilterType type() default FilterType.ANNOTATION;
```

```
         */
        @AliasFor("classes")
        Class<?>[] value() default {};

        /**
         */
        @AliasFor("value")
        Class<?>[] classes() default {};

        /**
         */
        String[] pattern() default {};
    }
}
```

1.2.2、说明

作用：

用于指定创建容器时要扫描的包。该注解在指定扫描的位置时，可以指定包名，也可以指定扫描的类。同时支持定义扫描规则，例如包含哪些或者排除哪些。同时，它还支持自定义Bean的命名规则

属性：

value:

用于指定要扫描的包。当指定了包的名称之后，spring会扫描指定的包及其子包下的所有类。

basePackages:

它和value作用是一样的。

basePackageClasses:

指定具体要扫描的类的字节码。

nameGenrator:

指定扫描bean对象存入容器时的命名规则。详情请参考第五章第4小节的BeanNameGenerator及其实现类。

scopeResolver:

用于处理并转换检测到的Bean的作用范围。

soperdProxy:

用于指定bean生成时的代理方式。默认是Default，则不使用代理。详情请参考第五章第5小节ScopedProxyMode枚举。

resourcePattern:

用于指定符合组件检测条件的类文件，默认是包扫描下的 **/*.class

useDefaultFilters:

是否对带有@Component @Repository @Service @Controller注解的类开启检测,默认是开启的。

includeFilters:

自定义组件扫描的过滤规则，用以扫描组件。

FilterType有5种类型：

ANNOTATION, 注解类型 默认

ASSIGNABLE_TYPE,指定固定类

ASPECTJ， ASPECTJ类型

REGEX,正则表达式

CUSTOM,自定义类型

详细用法请参考第五章第6小节自定义组件扫描过滤规则

excludeFilters:

自定义组件扫描的排除规则。

lazyInit:

组件扫描时是否采用懒加载 ，默认不开启。

使用场景：

细节：
在spring4.3版本之后还加入了一个@ComponentScans的注解，该注解就是支持配置多个@ComponentScan。

1.2.3、示例

在入门案例中，如果我们加入了dao或者记录日志的工具类，这些使用了@Component或其衍生注解配置的bean，要想让他们进入ioc容器，就少不了使用@ComponentScan

```
package com.itheima.dao.impl;
import com.itheima.dao.AccountDao;
/**
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
@Repository("accountDao")
public class AccountDaoImpl implements AccountDao{
    //持久层开发(此处没有考虑mybatis代理dao方式或者其他持久层技术，因为不希望和其他框架技术关联)
}

/**
 * spring的配置类
 * 用于替代xml配置
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
@Configuration
@Import(JdbcConfig.class)
@PropertySource("classpath:jdbc.properties")
@ComponentScan("com.itheima")
public class SpringConfiguration {
}
```

1.3、@Bean

1.3.1、源码

```
package org.springframework.context.annotation;

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

import org.springframework.beans.factory.annotation.Autowire;
import org.springframework.beans.factory.support.AbstractBeanDefinition;
import org.springframework.core.annotation.AliasFor;

@Target({ElementType.METHOD, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Bean {
    /**
```

```
    String[] value() default {};

    /**
     */
    @AliasFor("value")
    String[] name() default {};

    /**
     */
    @Deprecated
    Autowire autowire() default Autowire.NO;

    /**
     */
    boolean autowireCandidate() default true;

    /**
     */
    String initMethod() default "";

    /**
     */
    String destroyMethod() default AbstractBeanDefinition.INFER_METHOD;
}
```

1.3.2、说明

作用：

它写在方法上，表示把当前方法的返回值存入spring的ioc容器。

同时还可以出现在注解上，作为元注解来使用。

属性：

name:

用于指定存入spring容器中bean的标识。支持指定多个标识。当不指定该属性时，默认值是当前方法的名称。

value:

此属性是在4.3.3版本之后加入的。它和name属性的作用是一样的。

autowireCandidate:

用于指定是否支持自动按类型注入到其他bean中。只影响@Autowired注解的使用。不影响@Resource注解注入。默认值为true，意为允许使用自动按类型注入。

initMethod:

用于指定初始化方法。

destroyMethod:

用于指定销毁方法。

使用场景：

通常情况下，在基于注解的配置中，我们用于把一个类存入spring的ioc容器中，首先考虑的是使用@Component以及他的衍生注解。但是如果遇到要存入容器的Bean对象不是我们写的类，此时无法在类上添加@Component注解，这时就需要此注解了。

示例：

例如，在我们配置JdbcTemplate使用Spring内置数据源DriverManagerDataSource时，数据源类是spring-jdbc这个jar包中类，此时我们无法编辑，在上面加注解，此时就可以使用@Bean注解配置。

1.3.3、示例代码

```
    return new JdbcTemplate(dataSource);
}
```

1.4、@Import

1.4.1、源码

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Import {
    /**
     */
    Class<?>[] value();
}
```

1.4.2、说明

作用：

该注解是写在类上的，通常都是和注解驱动的配置类一起使用的。其作用是引入其他的配置类。使用了此注解之后，可以使我们的注解驱动开发和早期xml配置一样，分别配置不同的内容，使配置更加清晰。同时指定了此注解之后，被引入的类上可以不再使用@Configuration,@Component等注解。

属性：

value：

用于指定其他配置类的字节码。它支持指定多个配置类。

关于ImportSelector和ImportBeanDefinitionRegistrar请参考第五章第7小节@Import注解的高级分析。

使用场景：

当我们在使用注解驱动开发时，由于配置项过多，如果都写在一个类里面，配置结构和内容将杂乱不堪，此时使用此注解可以把配置项进行分门别类进行配置。

1.4.3、示例

在入门案例中，我们使用了SpringConfiguration做为主配置类，而连接数据库相关的配置被分配到了JdbcConfig配置类中，此时使用在SpringConfiguration类上使用@Import注解把JdbcConfig导入进来就可以了。

```
 /**
 * spring的配置类
 * 用于替代xml配置
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
@Configuration
@Import(JdbcConfig.class)
@PropertySource("classpath:jdbc.properties")
@ComponentScan("com.itheima")
public class SpringConfiguration {
}

/**
 * 连接数据库的配置
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
```

```
    @Value("${jdbc.driver}")
    private String driver;
    @Value("${jdbc.url}")
    private String url;
    @Value("${jdbc.username}")
    private String username;
    @Value("${jdbc.password}")
    private String password;


    @Bean("jdbcTemplate")
    public JdbcTemplate createJdbcTemplate(DataSource dataSource){
        return new JdbcTemplate(dataSource);
    }


    @Bean("dataSource")
    public DataSource createDataSource(){
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName(driver);
        dataSource.setUrl(url);
        dataSource.setUsername(username);
        dataSource.setPassword(password);
        return dataSource;
    }
}
```

1.5、@PropertySource

1.5.1、源码

```
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Repeatable(PropertySources.class)
public @interface PropertySource {

    /**
     */
    String name() default "";

    /**
     */
    String[] value();

    /**
     */
    boolean ignoreResourceNotFound() default false;

    /**
     */
    String encoding() default "";

    /**
     */
```

```
}
```

1.5.2、说明

作用：

用于指定读取资源文件的位置。注意，它不仅支持properties，也支持xml文件，并且通过YAML解析器，配合自定义PropertySourceFactory实现解析yml配置文件（详情请参考第五章第8小节自定义PropertySourceFactory实现YAML文件解析）。

属性：

name:

指定资源的名称。如果没有指定，将根据基础资源描述生成。

value:

指定资源的位置。可以是类路径，也可以是文件路径。

ignoreResourceNotFound:

指定是否忽略资源文件有没有，默认是false,也就是说当资源文件不存在时spring启动将会报错。

encoding:

指定解析资源文件使用的字符集。当有中文的时候，需要指定中文的字符集。

factory:

指定读取对应资源文件的工厂类，默认的是PropertySourceFactory。

使用场景：

我们实际开发中，使用注解驱动后，xml配置文件就没有了，此时一些配置如果直接写在类中，会造成和java源码的紧密耦合，修改起来不方法。此时一些配置可以使用properties或者yml来配置就变得很灵活方便。

1.5.3、示例

在入门案例中，我们连接数据库的信息如果直接写到JdbcConfig类中，当需要修改时，就面临修改源码的问题，此时使用@PropertySource和SpringEL表达式，就可以把配置放到properties文件中了。

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/spring_day01
jdbc.username=root
jdbc.password=1234
```

```
/**
 * 连接数据库的配置
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
public class JdbcConfig {

    @Value("${jdbc.driver}")
    private String driver;
    @Value("${jdbc.url}")
    private String url;
    @Value("${jdbc.username}")
    private String username;
    @Value("${jdbc.password}")
    private String password;
}

/**
 * spring的配置类
```

```
 * @Company http://www.itheima.com
 */
@Configuration
@Import(JdbcConfig.class)
@PropertySource(value = "classpath:jdbc.properties")
@ComponentScan("com.itheima")
public class SpringConfiguration {
}
```

2、注解驱动开发之注入时机和设定注入条件的注解

2.1、@DependsOn

2.1.1、源码

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface DependsOn {
    String[] value() default {};
}
```

2.1.2、说明

作用：
　　用于指定某个类的创建依赖的bean对象先创建。spring中没有特定bean的加载顺序，使用此注解则可指定bean的加载顺序。(在基于注解配置中，是按照类中方法的书写顺序决定的)
属性：
　　value：
　　　　用于指定bean的唯一标识。被指定的bean会在当前bean创建之前加载。
使用场景：
　　在观察者模式中，分为事件，事件源和监听器。一般情况下，我们的监听器负责监听事件源，当事件源触发了事件之后，监听器就要捕获，并且做出相应的处理。以此为前提，我们肯定希望监听器的创建时间在事件源之前，此时就可以使用此注解。

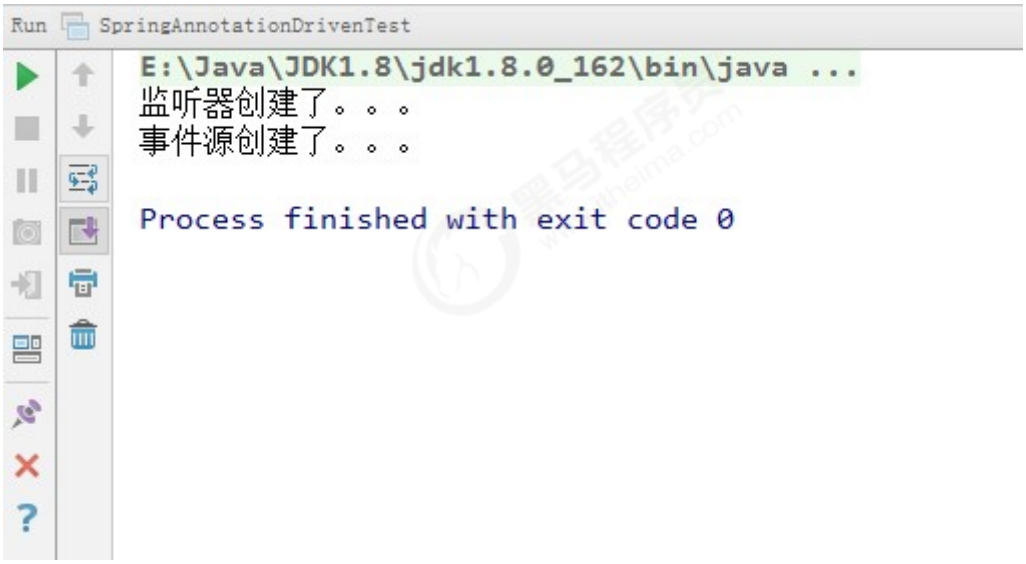
2.1.3、示例

```
/**
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
@Component
public class CustomerListener {

    public CustomerListener(){
        System.out.println("监听器创建了。。。");
    }
}

/**
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
@Component
```

```
    public Customer(){
        System.out.println("事件源创建了。。。");
    }
}
```



2.2、@Lazy

2.2.1、源码

```
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.CONSTRUCTOR,
ElementType.PARAMETER, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Lazy {
    boolean value() default true;
}
```

2.2.2、说明

作用：

用于指定单例bean对象的创建时机。在没有使用此注解时，单例bean的生命周期与容器相同。但是当使用了此注解之后，单例对象的创建时机变成了第一次使用时创建。注意：这不是延迟加载思想（因为不是每次使用时都创建，只是第一次创建的时机改变了）。

属性：

value:

指定是否采用延迟加载。默认值为true，表示开启。

使用场景：

在实际开发中，当我们的Bean是单例对象时，并不是每个都需要一开始都加载到ioc容器之中，有些对象可以在真正使用的时候再加载，当有此需求时，即可使用此注解。值得注意的是，此注解只对单例bean对象起作用，当指定了@Scope注解的prototype取值后，此注解不起作用。

2.2.3、示例

```
/**
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
```

```
//@Lazy(value = false)
//@Scope("prototype")
public class LazyBean {

    public LazyBean(){
        System.out.println("LazyBean对象创建了");
    }
}

/**
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
public class SpringAnnotationDrivenTest {

    /**
     * 测试
     * @param args
     */
    public static void main(String[] args) {
      //1.获取容器
      ApplicationContext ac = new AnnotationConfigApplicationContext("config");
      //2.根据id获取对象
      LazyBean lazyBean = (LazyBean)ac.getBean("lazyBean");
      System.out.println(lazyBean);
    }

}
```

```
30          //@Lazy
31          ApplicationContext ac = new AnnotationConfigApplicationContext( ...base
32          LazyBean lazyBean = (LazyBean)ac.getBean( name: "lazyBean");  ac: "org.
33          System.out.println(lazyBean);
34      }
35
36  }
37
```

LazyBean.java

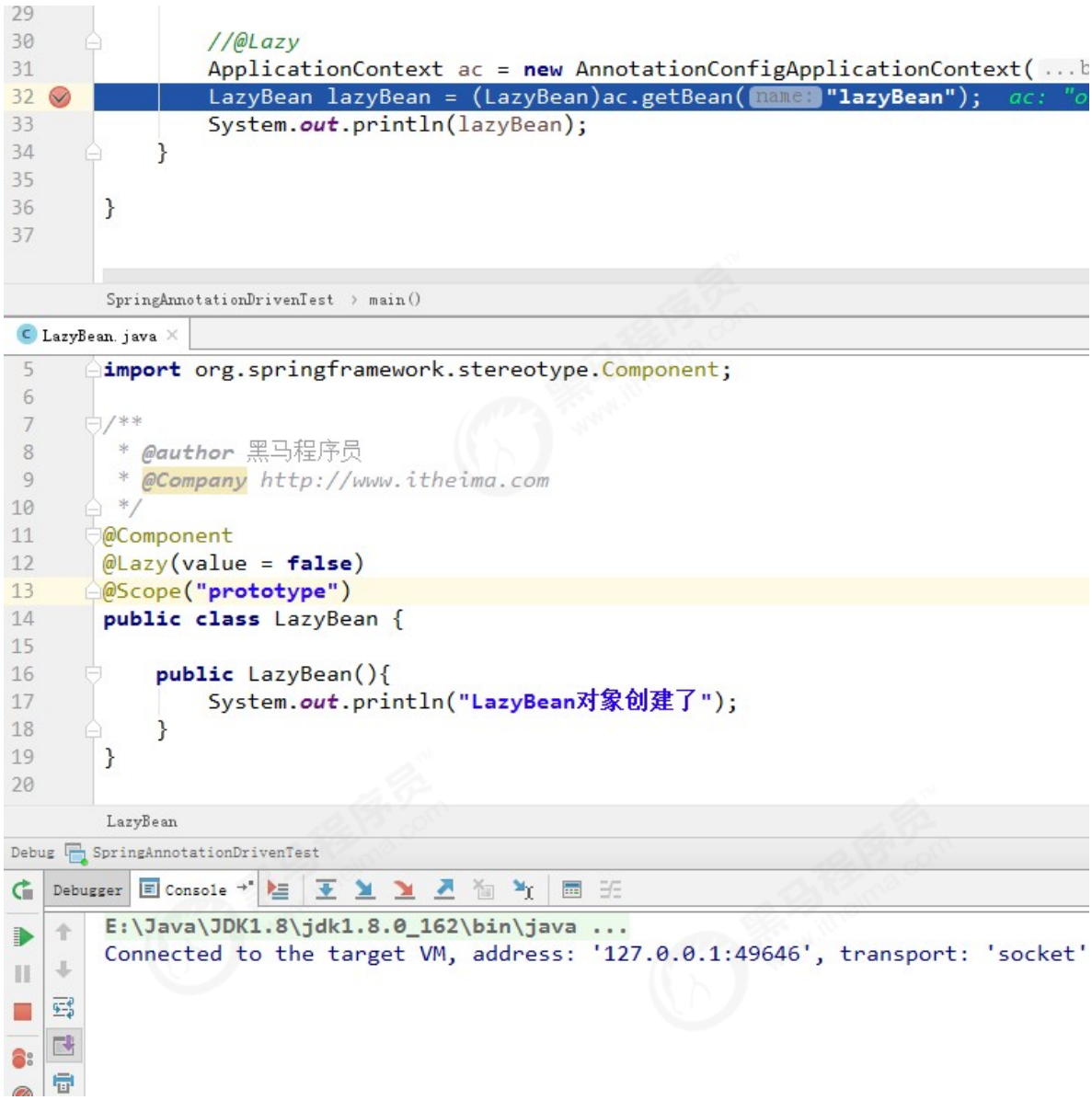
```
4   import org.springframework.stereotype.Component;
5
6   /**
7    * @author 黑马程序员
8    * @Company http://www.itheima.com
9    */
10  @Component
11  @Lazy
12  public class LazyBean {
13
14      public LazyBean(){
15          System.out.println("LazyBean对象创建了");
16      }
17  }
18
```

LazyBean

Debug SpringAnnotationDrivenTest

Debugger Console

```
E:\Java\JDK1.8\jdk1.8.0_162\bin\java ...
Connected to the target VM, address: '127.0.0.1:49610', transport: 'socket'
```



2.3、@Conditional

2.3.1、源码

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Conditional {

    /**
     * All {@link Condition Conditions} that must {@linkplain Condition#matches
match}
     * in order for the component to be registered.
     */
    Class<? extends Condition>[] value();
}
```

2.3.2、说明

它的作用是根据条件选择注入的bean对象。

属性：

value：

用于提供一个Condition接口的实现类，实现类中需要编写具体代码实现注入的条件。

使用场景：

当我们在开发时，可能会使用多平台来测试，例如我们的测试数据库分别部署到了linux和windows两个操作系统上面，现在根据我们的工程运行环境选择连接的数据库。此时就可以使用此注解。同时基于此注解引出的@Profile注解，就是根据不同的环境，加载不同的配置信息，详情请参考第五章第9小节@Profile的使用。

2.3.3、示例

```
/**
 * 连接数据库的配置
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
public class JdbcConfig {

    @Value("${jdbc.driver}")
    private String driver;
    @Value("${jdbc.url}")
    private String url;
    @Value("${jdbc.username}")
    private String username;
    @Value("${jdbc.password}")
    private String password;



    /**
     * linux系统注入的数据源
     * @param lDriver
     * @param lUrl
     * @param lUsername
     * @param lPassword
     * @return
     */
    @Bean("dataSource")
    @Conditional(LinuxCondition.class)
    public DataSource createLinuxDataSource(@Value("${linux.driver}") String
lDriver,
                                            @Value("${linux.url}")String lUrl,
                                            @Value("${linux.username}")String
lUsername,
                                            @Value("${linux.password}")String
lPassword){
        DriverManagerDataSource dataSource = new
DriverManagerDataSource(lUrl,lUsername,lPassword);
        dataSource.setDriverClassName(lDriver);
        System.out.println(lUrl);
        return dataSource;
    }

    /**
```

```
     */
    @Bean("dataSource")
    @Conditional(WindowsCondition.class)
    public DataSource createWindowsDataSource(){
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName(driver);
        dataSource.setUrl(url);
        dataSource.setUsername(username);
        dataSource.setPassword(password);
        System.out.println(url);
        return dataSource;
    }
}

/**
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
public class LinuxCondition implements Condition {

    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata
metadata) {
        //获取ioc使用的beanFactory
        ConfigurableListableBeanFactory beanFactory = context.getBeanFactory();
        //获取类加载器
        ClassLoader classLoader = context.getClassLoader();
        //获取当前环境信息
        Environment environment = context.getEnvironment();
        //获取bean定义的注册类
        BeanDefinitionRegistry registry = context.getRegistry();

        //获得当前系统名
        String property = environment.getProperty("os.name");
        //包含Windows则说明是windows系统，返回true
        if (property.contains("Linux")){
            return true;
        }
        return false;
    }
}


/**
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
public class WindowsCondition implements Condition {

    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata
metadata) {
        //获取ioc使用的beanFactory
        ConfigurableListableBeanFactory beanFactory = context.getBeanFactory();
        //获取类加载器
        ClassLoader classLoader = context.getClassLoader();
```

```
        /**
         * 获取所有系统环境变量
         */
        if(environment instanceof StandardEnvironment){
            StandardEnvironment standardEnvironment =
(StandardEnvironment)environment;
            Map<String,Object> map = standardEnvironment.getSystemProperties();
            for(Map.Entry<String,Object> me : map.entrySet()){
                System.out.println(me.getKey()+","+me.getValue());
            }
        }

        //获取bean定义的注册类
        BeanDefinitionRegistry registry = context.getRegistry();

        //获得当前系统名
        String property = environment.getProperty("os.name");
        //包含Windows则说明是windows系统，返回true
        if (property.contains("Windows")){
            return true;
        }
        return false;
    }
}
```

```
linux.driver=com.mysql.jdbc.Driver
linux.url=jdbc:mysql://localhost:3306/ssm
linux.username=root
linux.password=1234
--------------------------------------------------------------------------------
-----
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/spring_day01
jdbc.username=root
jdbc.password=1234
```

3、用于创建对象的注解

3.1、@Component和三个衍生注解

3.1.1、源码

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Indexed
public @interface Component {

    /**
     * The value may indicate a suggestion for a logical component name,
     * to be turned into a Spring bean in case of an autodetected component.
     * @return the suggested component name, if any (or empty String otherwise)
     */
```

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Controller {

    /**
     * The value may indicate a suggestion for a logical component name,
     * to be turned into a Spring bean in case of an autodetected component.
     * @return the suggested component name, if any (or empty String otherwise)
     */
    @AliasFor(annotation = Component.class)
    String value() default "";

}

@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Service {

    /**
     * The value may indicate a suggestion for a logical component name,
     * to be turned into a Spring bean in case of an autodetected component.
     * @return the suggested component name, if any (or empty String otherwise)
     */
    @AliasFor(annotation = Component.class)
    String value() default "";

}

@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Repository {

    /**
     * The value may indicate a suggestion for a logical component name,
     * to be turned into a Spring bean in case of an autodetected component.
     * @return the suggested component name, if any (or empty String otherwise)
     */
    @AliasFor(annotation = Component.class)
    String value() default "";

}
```

3.1.2、说明

时，首选默认无参构造函数。同时支持带参构造，前提是构造函数的参数依赖必须要有值。否则抛异常

属性：

value:

用于指定存入容器时bean的id。当不指定时，默认值为当前类的名称。

使用场景：

当我们需要把自己编写的类注入到Ioc容器中，就可以使用以上四个注解实现。以上四个注解中@Component注解通常用在非三层对象中。而@Controller，@Service，@Repository三个注解一般是针对三层对象使用的，提供更加精确的语义化配置。

需要注意的是，spring在注解驱动开发时，要求必须先接管类对象，然后会处理类中的属性和方法。如果类没有被spring接管，那么里面的属性和方法上的注解都不会被解析。

3.1.3、示例

正确的方式1：使用默认构造函数

```
/**
 * 用于记录系统日志
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
@Component
public class LogUtil {
    /**
     * 默认无参构造函数
     */
    public LogUtil(){
    }
  //可以使用aop思想实现系统日志的记录
}
```

正确的方式2：在构造函数中注入一个已经在容器中的bean对象。

```
/**
 * 此处只是举例：使用JdbcTemplate作为持久层中的操作数据库对象
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
@Repository("userDao")
public class UserDaoImpl implements UserDao{

    private JdbcTemplate jdbcTemplate ;
    /**
     * 此时要求容器中必须有JdbcTemplate对象
     * @param jdbcTemplate
     */
    public UserDaoImpl(JdbcTemplate jdbcTemplate){
        this.jdbcTemplate = jdbcTemplate;
    }
}
```

正确的方式3:在构造函数中注入一个读取配置文件获取到的值。

```
/**
 * 用于记录系统日志
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
@Component
```

```
         * 构造时，注入日志级别
         * @param logLevel
         */
        public LogUtil(@Value("${log.level}")String logLevel){
            System.out.println(logLevel);
        }
        //可以使用aop思想实现系统日志的记录
    }
```

错误的方式：由于logLevel没有值，所以运行会报错。

```
   /**
     * 用于记录系统日志
     * @author 黑马程序员
     * @Company http://www.itheima.com
     */
    @Component
    public class LogUtil {
        /**
         * 构造时，注入日志级别
         * @param logLevel
         */
        public LogUtil(String logLevel){
            System.out.println(logLevel);
        }
        //可以使用aop思想实现系统日志的记录
    }
```

4、用于注入数据的注解

4.1、@Autowired

4.1.1、源码

```
@Target({ElementType.CONSTRUCTOR, ElementType.METHOD, ElementType.PARAMETER,
ElementType.FIELD, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Autowired {

    /**
     * Declares whether the annotated dependency is required.
     * <p>Defaults to {@code true}.
     */
    boolean required() default true;

}
```

4.1.2、说明

使用变量名称（写在方法上就是方法名称）作为bean的id，在符合类型的bean中再次匹配，能匹配上就可以注入成功。当匹配不上时，是否报错要看required属性的取值。

属性：

required：

是否必须注入成功。默认值是true，表示必须注入成功。当取值为true的时候，注入不成功会报错。

使用场景：

此注解的使用场景非常之多，在实际开发中应用广泛。通常情况下我们自己写的类中注入依赖bean对象时，都可以采用此注解。

4.1.3、示例

```
/**
 * 此处只是举例：使用Jdbc作为持久层中的操作数据库对象
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
@Repository("accountDao")
public class AccountDaoImpl implements AccountDao{
    @Autowired
    private JdbcTemplate jdbcTemplate ;
}
```

4.2、@Qualifier

4.2.1、源码

```
@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER,
ElementType.TYPE, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface Qualifier {

    String value() default "";

}
```

4.2.2、说明

作用：

当使用自动按类型注入时，遇到有多个类型匹配的时候，就可以使用此注解来明确注入哪个bean对象。注意它通常情况下都必须配置@Autowired注解一起使用

属性：

value：

用于指定bean的唯一标识。

使用场景：

在我们的项目开发中，很多时候都会用到消息队列，我们一ActiveMQ为例。当和spring整合之后，Spring框架提供了一个JmsTemplate对象，它既可以用于发送点对点模型消息也可以发送主题模型消息。如果项目中两种消息模型都用上了，那么针对不同的代码，将会注入不同的JmsTemplate，而容器中出现两个之后，就可以使用此注解注入。当然不用也可以，我们只需要把要注入的变量名称改为和要注入的bean的id一致即可。

略

4.3、@Resource

4.3.1、源码

```
@Target({TYPE, FIELD, METHOD})
@Retention(RUNTIME)
public @interface Resource {
    String name() default "";
    String lookup() default "";
    Class<?> type() default java.lang.Object.class;
    enum AuthenticationType {
            CONTAINER,
            APPLICATION
    }
    AuthenticationType authenticationType() default
AuthenticationType.CONTAINER;
    boolean shareable() default true;
    String mappedName() default "";
    String description() default "";
}
```

4.3.2、说明

作用：

此注解来源于JSR规范（Java Specification Requests）,其作用是找到依赖的组件注入到应用来，它利用了JNDI（Java Naming and Directory Interface Java命名目录接口 J2EE规范之一）技术查找所需的资源。

默认情况下，即所有属性都不指定，它默认按照byType的方式装配bean对象。如果指定了name，没有指定type，则采用byName。如果没有指定name，而是指定了type，则按照byType装配bean对象。当byName和byType都指定了，两个都会校验，有任何一个不符合条件就会报错。

属性：

name：

资源的JNDI名称。在spring的注入时，指定bean的唯一标识。

type：

指定bean的类型。

lookup:

引用指向的资源的名称。它可以使用全局JNDI名称链接到任何兼容的资源。

authenticationType:

指定资源的身份验证类型。它只能为任何受支持类型的连接工厂的资源指定此选项，而不能为其他类型的资源指定此选项。

shareable：

指定此资源是否可以在此组件和其他组件之间共享。

mappedName：

指定资源的映射名称。

description：

指定资源的描述。

使用场景：

当我们某个类的依赖bean在ioc容器中存在多个的时候，可以使用此注解指定特定的bean对象注入。当然我们也可以使用@Autowired配合@Qualifier注入。

4.3.3、示例

4.4、@Value

4.4.1、源码

```
@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER,
ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Value {

    /**
     * The actual value expression: for example {@code #
{systemProperties.myProp}}.
     */
    String value();

}
```

4.4.2、说明

作用：
    用于注入基本类型和String类型的数据。它支持spring的EL表达式，可以通过${} 的方式获取配置文件中的数据。配置文件支持properties,xml和yml文件。
属性：
    value:
        指定注入的数据或者spring的el表达式。
使用场景：
    在实际开发中，像连接数据库的配置，发送邮件的配置等等，都可以使用配置文件配置起来。此时读取配置文件就可以借助spring的el表达式读取。

4.4.3、示例

```
/**
 * 连接数据库的配置
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
public class JdbcConfig {

    @Value("${jdbc.driver}")
    private String driver;
    @Value("${jdbc.url}")
    private String url;
    @Value("${jdbc.username}")
    private String username;
    @Value("${jdbc.password}")
    private String password;


    /**
     * windows系统注入的数据源
     * @return
     */
    @Bean("dataSource")
```

```
        dataSource.setDriverClassName(driver);
        dataSource.setUrl(url);
        dataSource.setUsername(username);
        dataSource.setPassword(password);
        return dataSource;
    }

}
```

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/spring_day01
jdbc.username=root
jdbc.password=1234
```

4.5、@Inject

4.5.1、源码

```
@Target({ METHOD, CONSTRUCTOR, FIELD })
@Retention(RUNTIME)
@Documented
public @interface Inject {
}
```

4.5.2、说明

作用：

它也是用于建立依赖关系的。和@Resource和@Autowired的作用是一样。在使用之前需要先导入坐标：

```
<!-- https://mvnrepository.com/artifact/javax.inject/javax.inject -->
<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>1</version>
</dependency>
```

但是他们之前也有区别：

@Autowired：来源于spring框架自身。

默认是byType自动装配，当配合了@Qualifier注解之后，由@Qualifier实现byName装配。它有一个required属性，用于指定是否必须注入成功。

@Resource：来源于JSR-250规范。

在没有指定name属性时是byType自动装配，当指定了name属性之后，采用byName方式自动装配。

@Inject：来源于JSR-330规范。（JSR330是Jcp给出的官方标准反向依赖注入规范。）

它不支持任何属性，但是可以配合@Qualifier或者@Primary注解使用。

同时，它默认是采用byType装配，当指定了JSR-330规范中的@Named注解之后，变成byName装配。

属性：

无

使用场景：

在使用@Autowired注解的地方，都可以替换成@Inject。它也可以出现在方法上，构造函数上和字段上，但是需要注意的是：因为JRE无法决定构造方法注入的优先级，所以规范中规定类中只能有一个构造方法带@Inject注解。

```
/**
 * 第一种写法：  写在字段上
 * 此处只是举例：使用Jdbc作为持久层中的操作数据库对象
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
@Repository("userDao")
public class UserDaoImpl implements UserDao{

    @Inject
    private JdbcTemplate jdbcTemplate ;

}

/**
 * 第二种写法：写在构造函数上
 * 此处只是举例：使用Jdbc作为持久层中的操作数据库对象
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
@Repository("accountDao")
public class AccountDaoImpl implements AccountDao{

    private JdbcTemplate jdbcTemplate ;
    /**
     * 此时要求容器中必须有JdbcTemplate对象
     * @param jdbcTemplate
     */
    @Inject
    public AccountDaoImpl(JdbcTemplate jdbcTemplate){
        this.jdbcTemplate = jdbcTemplate;
    }
}

/**
 * 第三种写法：配合@Named注解使用
 * 此处只是举例：使用Jdbc作为持久层中的操作数据库对象
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
@Repository("accountDao")
public class AccountDaoImpl implements AccountDao{

    @Inject
    @Named("jdbcTemplate")
    private JdbcTemplate jdbcTemplate ;
}
```

4.6、@Primary

4.6.1、源码

```
@Documented
public @interface Primary {

}
```

4.6.2、说明

作用：
　　用于指定bean的注入优先级。被@Primary修饰的bean对象优先注入
属性：
　　无
使用场景：
　　当我们的依赖对象，有多个存在时，@Autowired注解已经无法完成功能，此时我们首先想到的是@Qualifier注解指定依赖bean的id。但是此时就产生了，无论有多少个bean，每次都会使用指定的bean注入。但是当我们使用@Primary，表示优先使用被@Primary注解的bean，但是当不存在时还会使用其他的。

4.6.3、示例

```
/**
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
@Service("accountService")
public class AccountServiceImpl implements AccountService {

    @Autowired
//    @Qualifier("accountImpl1")
    private AccountDao accountDao;

    public void save(){
        System.out.println(accountDao);
    }
}


/**
 * 此处只是举例：使用Jdbc作为持久层中的操作数据库对象
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
@Repository("accountDaoImpl1")
public class AccountDaoImpl implements AccountDao{

    @Override
    public String toString() {
        return "accountDaoImpl1";
    }
}

/**
 * 此处只是举例：使用Jdbc作为持久层中的操作数据库对象
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
```

```
public class AccountDaoImpl2 implements AccountDao{

    @Override
    public String toString() {
        return "accountDaoImpl2";
    }
}
```

5、和生命周期以及作用范围相关的注解

5.1、@Scope

5.1.1、源码

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Scope {

    /**
     * Alias for {@link #scopeName}.
     * @see #scopeName
     */
    @AliasFor("scopeName")
    String value() default "";

    /**
     * Specifies the name of the scope to use for the annotated component/bean.
     * <p>Defaults to an empty string ({@code ""}) which implies
     * {@link ConfigurableBeanFactory#SCOPE_SINGLETON SCOPE_SINGLETON}.
     * @since 4.2
     * @see ConfigurableBeanFactory#SCOPE_PROTOTYPE
     * @see ConfigurableBeanFactory#SCOPE_SINGLETON
     * @see org.springframework.web.context.WebApplicationContext#SCOPE_REQUEST
     * @see org.springframework.web.context.WebApplicationContext#SCOPE_SESSION
     * @see #value
     */
    @AliasFor("value")
    String scopeName() default "";

    /**
     * Specifies whether a component should be configured as a scoped proxy
     * and if so, whether the proxy should be interface-based or subclass-based.
     * <p>Defaults to {@link ScopedProxyMode#DEFAULT}, which typically indicates
     * that no scoped proxy should be created unless a different default
     * has been configured at the component-scan instruction level.
     * <p>Analogous to {@code <aop:scoped-proxy/>} support in Spring XML.
     * @see ScopedProxyMode
     */
    ScopedProxyMode proxyMode() default ScopedProxyMode.DEFAULT;

}
```

5.1.2、说明

属性：
value:
指定作用范围的取值。在注解中默认值是""。
但是在spring初始化容器时，会借助ConfigurableBeanFactory接口中的类成员：
String SCOPE_SINGLETON = "singleton";
scopeName:
它和value的作用是一样的。
proxyMode:
它是指定bean对象的代理方式的。指定的是ScopedProxyMode枚举的值
DEFAULT：默认值。（就是NO）
NO：不使用代理。
INTERFACES：使用JDK官方的基于接口的代理。
TARGET_CLASS：使用CGLIB基于目标类的子类创建代理对象。
使用场景：
在实际开发中，我们的bean对象默认都是单例的。通常情况下，被spring管理的bean都使用单例模式来创建。但是也有例外，例如Struts2框架中的Action，由于有模型驱动和OGNL表达式的原因，就必须配置成多例的。

5.1.3、示例

略

5.2、@PostConstruct

5.2.1、源码

```
@Documented
@Retention (RUNTIME)
@Target(METHOD)
public @interface PostConstruct {
}
```

5.2.2、说明

作用：
用于指定bean对象的初始化方法。
属性：
无
使用场景：
在bean对象创建完成后，需要对bean中的成员进行一些初始化的操作时，就可以使用此注解配置一个初始化方法，完成一些初始化的操作。

5.2.3、示例

略

5.3、@PreDestroy

5.3.1、源码

```
@Target(METHOD)
public @interface PreDestroy {
}
```

5.3.2、说明

作用：
　　用于指定bean对象的销毁方法。
属性：
　　无
使用场景：
　　在bean对象销毁之前，可以进行一些清理操作。

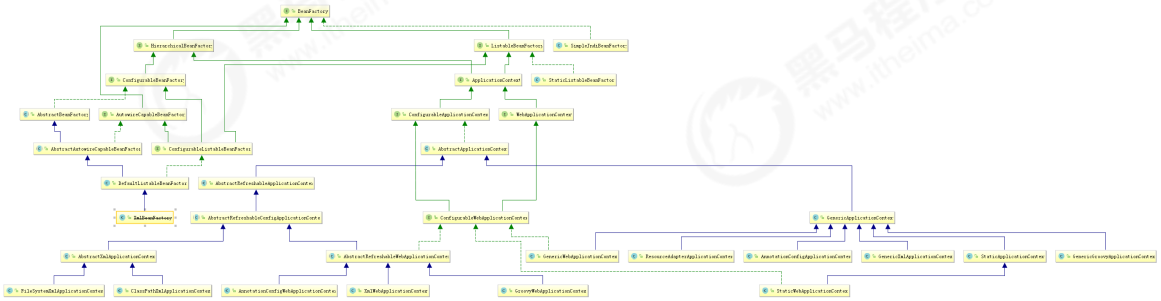
5.3.3、示例

略

五、Spring高级-IOC的深入剖析

1、Spring中的BeanFactory

1.1、BeanFactory类视图



1.2、工厂详解

1.2.1、BeanFactory

BeanFactory 中定义的各种方法如上面方法注释，整个设计还是比较简洁、直观的，其中将近一半是获取 bean 对象的各种方法，另外就是对 bean 属性的获取和判定，该接口仅仅是定义了 IoC 容器的最基本基本形式，具体实现都交由子类来实现。

1.2.2、HierarchicalBeanFactory

HierarchicalBeanFactory 译为中文是“分层的”，它相对于 BeanFactory 增加了对父 BeanFactory 的获取，子容器可以通过接口方法访问父容器，让容器的设计具备了层次性。这种层次性增强了容器的扩展性和灵活性，我们可以通过编程的方式为一个已有的容器添加一个或多个子容器，从而实现一些特殊功能。层次容器有一个特点就是子容器对于父容器来说是透明的，而子容器则能感知到父容器的存在。典型的应用场景就是 Spring MVC，控制层的 bean 位于子容器中，并将业务层和持久层的 bean 所在的容器设置为父容器，这样的设计可以让控制层的 bean 访问业务层和持久层的 bean，反之则不行，从而在容器层面对三层软件结构设计提供支持。

1.2.3、ListableBeanFactory

数，获取容器中所有 bean 的名称列表，按照名称类型获取 bean 名称，以及检查容器中是否包含指定名称的 bean 等等。Listable 中文译为“可列举的”，对于容器而言，bean 的定义和属性是可以列举的对象。

1.2.4、AutowireCapableBeanFactory

AutowireCapableBeanFactory 提供了创建 bean、自动注入，初始化以及应用 bean 的后置处理器等功能。自动注入让配置变得更加简单，也让注解配置成为可能，Spring 提供了四种自动注入类型：

byName：

根据名称自动装配。假设 bean A 有一个名为 b 的属性，如果容器中刚好存在一个 bean 的名称为 b，则将该 bean 装配给 bean A 的 b 属性。

byType：

根据类型自动匹配。假设 bean A 有一个类型为 B 的属性，如果容器中刚好有一个 B 类型的 bean，则使用该 bean 装配 A 的对应属性。

constructor：

仅针对构造方法注入而言，类似于 byType。如果 bean A 有一个构造方法，构造方法包含一个 B 类型的入参，如果容器中有一个 B 类型的 bean，则使用该 bean 作为入参，如果找不到，则抛出异常。

autodetect：

根据 bean 的自省机制决定采用 byType 还是 constructor 进行自动装配。如果 bean 提供了默认的构造函数，则采用 byType，否则采用 constructor。

总结：

<beans/> 元素标签中的 default-autowire 属性可以配置全局自动匹配，default-autowire 默认值为 no，表示不启用自动装配。在实际开发中，XML 配置方式很少启用自动装配功能，而基于注解的配置方式默认采用 byType 自动装配策略。

1.2.5、ConfigurableBeanFactory

ConfigurableBeanFactory 提供配置 Factory 的各种方法，增强了容器的可定制性，定义了设置类装载器、属性编辑器、容器初始化后置处理器等方法。

1.2.6、DefaultListableBeanFactory

DefaultListableBeanFactory 是一个非常重要的类，它包含了 IoC 容器所应该具备的重要功能，是容器完整功能的一个基本实现，XmlBeanFactory 是一个典型的由该类派生出来的 Factory，并且只是增加了加载 XML 配置资源的逻辑，而容器相关的特性则全部由 DefaultListableBeanFactory 来实现。

1.2.7、ApplicationContext

ApplicationContext 是 Spring 为开发者提供的高级容器形式，也是我们初始化 Spring 容器的常用方式，除了简单容器所具备的功能外，ApplicationContext 还提供了许多额外功能来降低开发人员的开发量，提升框架的使用效率。这些额外的功能主要包括：

国际化支持：ApplicationContext 实现了 org.springframework.context.MessageSource 接口，该接口为容器提供国际化消息访问功能，支持具备多语言版本需求的应用开发，并提供了多种实现来简化国际化资源文件的装载和获取。

发布应用上下文事件：ApplicationContext 实现了 org.springframework.context.ApplicationEventPublisher 接口，该接口让容器拥有发布应用上下文事件的功能，包括容器启动、关闭事件等，如果一个 bean 需要接收容器事件，则只需要实现 ApplicationListener 接口即可，Spring 会自动扫描对应的监听器配置，并注册成为主题的观察者。

丰富的资源获取的方式：ApplicationContext 实现了 org.springframework.core.io.support.ResourcePatternResolver 接口，ResourcePatternResolver 的实现类 PathMatchingResourcePatternResolver 让我们可以采用 Ant 风格的资源路径去加载配置文件。

1.2.8、ConfigurableApplicationContext

的主要流程（包含简单的容器的全部功能，以及高级容器特有的扩展功能）

1.2.9、WebApplicationContext

WebApplicationContext 是为 WEB 应用定制的上下文，可以基于 WEB 容器来实现配置文件的加载，以及初始化工作。对于非 WEB 应用而言，bean 只有 singleton 和 prototype 两种作用域，而在 WebApplicationContext 中则新增了 request、session、globalSession，以及 application 四种作用域。
WebApplicationContext 将整个应用上下文对象以属性的形式放置到 ServletContext 中，所以在 WEB 应用中，我们可以通过 WebApplicationContextUtils 的 getWebApplicationContext(ServletContext sc) 方法，从 ServletContext 中获取到 ApplicationContext 实例。为了支持这一特性，WebApplicationContext 定义了一个常量：

```
ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE = WebApplicationContext.class.getName() +
".ROOT"
```

并在初始化应用上下文时以该常量为 key，将 WebApplicationContext 实例存放到 ServletContext 的属性列表中，当我们在调用 WebApplicationContextUtils 的 getWebApplicationContext(ServletContext sc) 方法时，本质上是在调用 ServletContext 的 getAttribute(String name) 方法，只不过 Spring 会对获取的结果做一些校验。

1.2.10、高级容器的一些具体实现类型

AnnotationConfigApplicationContext：
是基于注解驱动开发的高级容器类，该类中提供了AnnotatedBeanDefinitionReader和ClassPathBeanDefinitionScanner两个成员，AnnotatedBeanDefinitionReader用于读取注解创建Bean的定义信息，ClassPathBeanDefinitionScanner负责扫描指定包获取Bean的定义信息。

ClasspathXmlApplicationContext:
是基于xml配置的高级容器类，它用于加载类路径下配置文件。

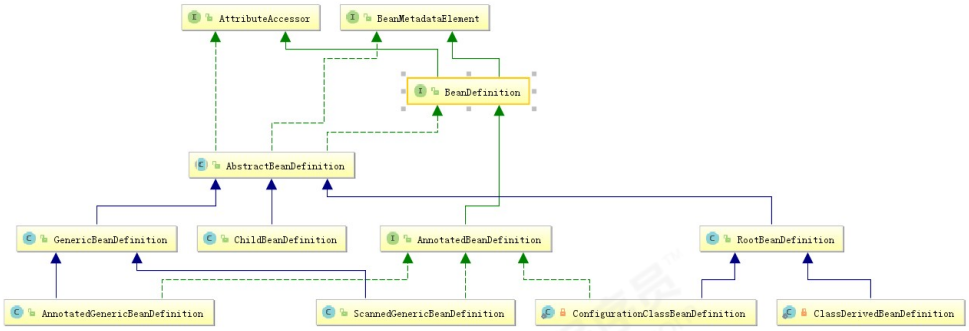
FileSystemXmlApplicationContext:
是基于xml配置的高级容器类，它用于加载文件系统中的配置文件。

AnnotationConfigWebApplicationContext:
是注解驱动开发web应用的高级容器类。

2、Spring中的BeanDefinition

2.1、BeanDefinition类视图

2.2.1、类视图



2.2.2、说明

现实中的容器都是用来装物品的，Spring 的容器也不例外，这里的物品就是 bean。我通常对于 bean 的印象是一个个躺在配置文件中的 <bean/> 标签，或者是被注解的类，但是这些都是 bean 的静态表示，是还没有放入容器的物料，最终（加载完配置，且在 getBean 之前）加载到容器中的是一个个 BeanDefinition 实例。BeanDefinition 的继承关系如下图，RootBeanDefinition、ChildBeanDefinition，以及 GenericBeanDefinition 是三个主要的实现。有时候我们需要在配置时，通过 parent 属性指定 bean 的父子关系，这个时候父 bean 则用 RootBeanDefinition 表示，而子 bean 则用 ChildBeanDefinition 表示。GenericBeanDefinition 自 2.5 版本引入，是对于一般的 bean 定义的一站式服务中心。

2.2、Bean的定义信息详解

2.2.1、源码分析

```
/**
 * 在上一小节我们介绍了RootBeanDefinition，ChildBeanDefinition，
GenericBeanDefinition三个类
 * 他们都是由AbstractBeanDefinition派生而来，该抽象类中包含了bean的所有配置项和一些支持程
序运
 * 行的属性。以下是类中属性的说明。
 */
public abstract class AbstractBeanDefinition extends
BeanMetadataAttributeAccessor implements BeanDefinition, Cloneable {
    // 常量定义略

    /** bean 对应的类实例 */
    private volatile Object beanClass;
    /** bean的作用域，对应scope属性 */
    private String scope = SCOPE_DEFAULT;
    /** 是否是抽象类，对应abstract属性 */
    private boolean abstractFlag = false;
    /** 是否延迟加载，对应lazy-init属性 */
    private boolean lazyInit = false;
    /** 自动装配模式，对应autowire属性 */
    private int autowireMode = AUTOWIRE_NO;
    /** 依赖检查，对应dependency-check属性 */
    private int dependencyCheck = DEPENDENCY_CHECK_NONE;
    /** 对应depends-on，表示一个bean实例化前置依赖另一个bean */
    private String[] dependsOn;
    /** 对应autowire-candidate属性，设置为false时表示取消当前bean作为自动装配候选者的资格
*/
```

```
    private boolean primary = false;
    /** 对应qualifier属性 */
    private final Map<String, AutowireCandidateQualifier> qualifiers = new
LinkedHashMap<String, AutowireCandidateQualifier>(0);
    /** 非配置项：表示允许访问非公开的构造器和方法，由程序设置 */
    private boolean nonPublicAccessAllowed = true;
    /**
     * 非配置项：表示是否允许以宽松的模式解析构造函数，由程序设置
     *
     * 例如：如果设置为true，则在下列情况时不会抛出异常（示例来源于《Spring源码深度解析》）
     * interface ITest{}
     * class ITestImpl implements ITest {}
     * class Main {
     * Main(ITest i){}
     * Main(ITestImpl i){}
     * }
     */
    private boolean lenientConstructorResolution = true;
    /** 对应factory-bean属性 */
    private String factoryBeanName;
    /** 对应factory-method属性 */
    private String factoryMethodName;
    /** 记录构造函数注入属性，对应<construct-arg/>标签 */
    private ConstructorArgumentValues constructorArgumentValues;
    /** 记录<property/>属性集合 */
    private MutablePropertyValues propertyValues;
    /** 记录<lookup-method/>和<replaced-method/>标签配置 */
    private MethodOverrides methodOverrides = new MethodOverrides();
    /** 对应init-method属性 */
    private String initMethodName;
    /** 对应destroy-method属性 */
    private String destroyMethodName;
    /** 非配置项：是否执行init-method，由程序设置 */
    private boolean enforceInitMethod = true;
    /** 非配置项：是否执行destroy-method，由程序设置 */
    private boolean enforceDestroyMethod = true;
    /** 非配置项：表示是否是用户定义，而不是程序定义的，创建AOP时为true,由程序设置 */
    private boolean synthetic = false;
    /**
     * 非配置项：定义bean的应用场景，由程序设置，角色如下：
     * ROLE_APPLICATION：用户
     * ROLE_INFRASTRUCTURE：完全内部使用
     * ROLE_SUPPORT：某些复杂配置的一部分
     */
    private int role = BeanDefinition.ROLE_APPLICATION;
    /** bean的描述信息，对应description标签 */
    private String description;
    /** bean定义的资源 */
    private Resource resource;

    // 方法定义略
}
```

2.2.2、总结

似于内存数据库，其实现类 SimpleBeanDefinitionRegistry 主要以 Map 作为存储标的。

3、注解驱动执行过程分析

3.1、使用配置类字节码的构造函数

3.1.1、构造函数源码

```
/**
 * Create a new AnnotationConfigApplicationContext that needs to be populated
 * through {@link #register} calls and then manually {@linkplain #refresh
refreshed}.
 */
public AnnotationConfigApplicationContext() {
    this.reader = new AnnotatedBeanDefinitionReader(this);
    this.scanner = new ClassPathBeanDefinitionScanner(this);
}
/**
 * Create a new AnnotationConfigApplicationContext, deriving bean definitions
 * from the given annotated classes and automatically refreshing the context.
 * @param annotatedClasses one or more annotated classes,
 * e.g. {@link Configuration @Configuration} classes
 */
public AnnotationConfigApplicationContext(Class<?>... annotatedClasses) {
    this();
    register(annotatedClasses);
    refresh();
}
```

3.1.2、register方法说明

它是根据传入的配置类字节码解析Bean对象中注解的（包括类上的和类中方法和字段上的注解。如果类没有被注解，那么类中方法和字段上的注解不会被扫描）。使用的是AnnotatedGenericBeanDefinition,里面包含了BeanDefinition和Scope两部分信息，其中BeanDefinition是传入注解类的信息,即SpringConfiguration;scope是指定bean的作用范围，默认情况下为单例。

同时，借助AnnotationConfigUtils类中processCommonDefinitionAnnotations方法判断是否使用了Primary，Lazy，DependsOn等注解来决定Bean的加载时机。

在ConfigurationClassBeanDefinitionReader类中的registerBeanDefinitionForImportedConfigurationClass方法会把导入的JdbcConfig类注册到容器中。而loadBeanDefinitionsForBeanMethod方法会解析Bean注解，把被Bean注解修饰的方法返回值存入容器。

3.1.3、执行过程分析图


ioc-annotation-execute1

3.2、使用包扫描的构造函数

3.2.1、构造函数源码

```
/**
 * Create a new AnnotationConfigApplicationContext that needs to be populated
```

```
     */
    public AnnotationConfigApplicationContext() {
        this.reader = new AnnotatedBeanDefinitionReader(this);
        this.scanner = new ClassPathBeanDefinitionScanner(this);
    }
    /**
     * Create a new AnnotationConfigApplicationContext, scanning for bean
    definitions
     * in the given packages and automatically refreshing the context.
     * @param basePackages the packages to check for annotated classes
     */
    public AnnotationConfigApplicationContext(String... basePackages) {
        this();
        scan(basePackages);
        refresh();
    }
```

3.2.2、scan方法说明

它是根据传入的类路径下(classpath*)的包解析Bean对象中注解的（包括类上以及类成员的），使用的是ClassPathBeanDefinitionScanner类中的doScan方法，该方法最终将得到的BeanDefinitionHolder信息存储到LinkedHashSet中，为后面初始化容器做准备。

doScan中的findCandidateComponents方法调用ClassPathScanningCandidateComponentProvider类中的scanCandidateComponents方法，而此方法又去执行了PathMatchingResourcePatternResolver类中的doFindAllClassPathResources方法，找到指定扫描包的URL(是URL，不是路径。因为是带有file协议的)，然后根据磁盘路径读取当前目录及其子目录下的所有类。接下来执行AnnotationConfigUtils类中的processCommonDefinitionAnnotations方法，剩余就和本章节第一小节后面的过程一样了。

3.2.3、执行过程分析图

3.3、注册注解类型过滤器

3.3.1、ClassPathScanningCandidateComponentProvider的registerDefaultFilters方法说明

```
/**
     * Register the default filter for {@link Component @Component}.
     * <p>This will implicitly register all annotations that have the
     * {@link Component @Component} meta-annotation including the
     * {@link Repository @Repository}, {@link Service @Service}, and
     * {@link Controller @Controller} stereotype annotations.
     * <p>Also supports Java EE 6's {@link javax.annotation.ManagedBean} and
     * JSR-330's {@link javax.inject.Named} annotations, if available.
     *
     */
    @SuppressWarnings("unchecked")
    protected void registerDefaultFilters() {
        this.includeFilters.add(new AnnotationTypeFilter(Component.class));
        ClassLoader cl =
ClassPathScanningCandidateComponentProvider.class.getClassLoader();
        try {
            this.includeFilters.add(new AnnotationTypeFilter(
```

```
            logger.trace("JSR-250 'javax.annotation.ManagedBean' found and
supported for component scanning");
        }
        catch (ClassNotFoundException ex) {
            // JSR-250 1.1 API (as included in Java EE 6) not available - simply
skip.
        }
        try {
            this.includeFilters.add(new AnnotationTypeFilter(
                    ((Class<? extends Annotation>)
ClassUtils.forName("javax.inject.Named", cl)), false));
            logger.trace("JSR-330 'javax.inject.Named' annotation found and
supported for component scanning");
        }
        catch (ClassNotFoundException ex) {
            // JSR-330 API not available - simply skip.
        }
    }
```

3.4、准备和初始化容器

3.4.1、AbstractApplicationContext的refresh方法说明

```
@Override
public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
        //Prepare this context for refreshing.
        //1.准备容器，设置一些初始化信息，例如启动时间。验证必须要的属性等等。
        prepareRefresh();

        // Tell the subclass to refresh the internal bean factory.
        //2.告诉子类刷新内部bean工厂。实际就是重新创建一个Bean工厂
        ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

        // Prepare the bean factory for use in this context.
        //3.准备使用创建的这个BeanFactory，添加或者注册到当前Bean工厂一些必要对象。
        prepareBeanFactory(beanFactory);

        try {
            // Allows post-processing of the bean factory in context subclasses.
            //4.允许子容器对BeanFactory进行后处理。例如，在web环境中bean的作用范围等等。
            postProcessBeanFactory(beanFactory);

            // Invoke factory processors registered as beans in the context.
            //5.在Singleton的Bean对象初始化前，对Bean工厂进行一些处理
            invokeBeanFactoryPostProcessors(beanFactory);

            // Register bean processors that intercept bean creation.
            //6.注册拦截bean创建的处理器
            registerBeanPostProcessors(beanFactory);

            // Initialize message source for this context.
            //7.初始化消息资源接口的实现类。主要用于处理国际化（i18n）
            initMessageSource();
```

```
            initApplicationEventMulticaster();

            // Initialize other special beans in specific context subclasses.
            //9.在AbstractApplicationContext的子类中初始化其他特殊的bean
            onRefresh();

            // Check for listener beans and register them.
            //10.注册应用的监听器。就是注册实现了ApplicationListener接口的监听器bean
            registerListeners();

            // Instantiate all remaining (non-lazy-init) singletons.
            //11.实例化所有剩余的（非lazy init）单例。（就是没有被@Lazy修饰的单例Bean）
            finishBeanFactoryInitialization(beanFactory);//十一、

            // Last step: publish corresponding event.
            //12.完成context的刷新。主要是调用LifecycleProcessor的onRefresh()方法，并且发布事件（ContextRefreshedEvent）。
            finishRefresh();
        }

        catch (BeansException ex) {
            if (logger.isWarnEnabled()) {
                logger.warn("Exception encountered during context initialization - " +
                        "cancelling refresh attempt: " + ex);
            }

            // Destroy already created singletons to avoid dangling resources.
            destroyBeans();//如果刷新失败那么就会将已经创建好的单例Bean销毁掉

            // Reset 'active' flag.
            cancelRefresh(ex);//重置context的活动状态

            // Propagate exception to caller.
            throw ex;//抛出异常
        }

        finally {
            // Reset common introspection caches in Spring's core, since we
            // might not ever need metadata for singleton beans anymore...
            resetCommonCaches();//重置的Spring内核的缓存。因为可能不再需要metadata给单例Bean了。
        }
    }
}
```

3.5、实例化和获取Bean对象

3.5.1、AbstractBeanFactory的doGetBean方法说明

```
protected <T> T doGetBean(
            final String name, final Class<T> requiredType, final Object[] args,
boolean typeCheckOnly) throws BeansException {
    /*
     * 获取name对应的真正beanName
```

```
容：
     * 1. 如果是FactoryBean，则去掉修饰符“&”
     * 2. 沿着引用链获取alias对应的最终name
     */
    final String beanName = this.transformedBeanName(name);

    Object bean;

    /*
     * 检查缓存或者实例工厂中是否有对应的单例
     *
     * 在创建单例bean的时候会存在依赖注入的情况，而在创建依赖的时候为了避免循环依赖
     * Spring创建bean的原则是不等bean创建完成就会将创建bean的ObjectFactory提前曝光（将对应的ObjectFactory加入到缓存）
     * 一旦下一个bean创建需要依赖上一个bean，则直接使用ObjectFactory对象
     */
    Object sharedInstance = this.getSingleton(beanName); // 获取单例
    if (sharedInstance != null && args == null) {
        // 实例已经存在
        if (logger.isDebugEnabled()) {
            if (this.isSingletonCurrentlyInCreation(beanName)) {
                logger.debug("Returning eagerly cached instance of singleton bean '" + beanName + "' that is not fully initialized yet - a consequence of a circular reference");
            } else {
                logger.debug("Returning cached instance of singleton bean '" + beanName + "'");
            }
        }
        // 返回对应的实例
        bean = this.getObjectForBeanInstance(sharedInstance, name, beanName, null);
    } else {
        // 单例实例不存在
        if (this.isPrototypeCurrentlyInCreation(beanName)) {
            /*
             * 只有在单例模式下才会尝试解决循环依赖问题
             * 对于原型模式，如果存在循环依赖，也就是满足this.isPrototypeCurrentlyInCreation(beanName)，抛出异常
             */
            throw new BeanCurrentlyInCreationException(beanName);
        }

        // 获取parentBeanFactory实例
        BeanFactory parentBeanFactory = this.getParentBeanFactory();
        // 如果在beanDefinitionMap中（即所有已经加载的类中）不包含目标bean，则尝试从parentBeanFactory中获取
        if (parentBeanFactory != null && !this.containsBeanDefinition(beanName)) {
            String nameToLookup = this.originalBeanName(name);  // 获取name对应的真正beanName，如果是factoryBean，则加上“&”前缀
            if (args != null) {
                // 递归到BeanFactory中寻找
                return (T) parentBeanFactory.getBean(nameToLookup, args);
            } else {
                return parentBeanFactory.getBean(nameToLookup, requiredType);
```

```
        // 如果不仅仅是做类型检查，标记bean的状态已经创建，即将beanName加入alreadyCreated
集合中
        if (!typeCheckOnly) {
            this.markBeanAsCreated(beanName);
        }

        try {
            /*
             * 将存储XML配置的GenericBeanDefinition实例转换成RootBeanDefinition实例，
方便后续处理
             * 如果存在父bean，则同时合并父bean的相关属性
             */
            final RootBeanDefinition mbd =
this.getMergedLocalBeanDefinition(beanName);
            // 检查bean是否是抽象的，如果是则抛出异常
            this.checkMergedBeanDefinition(mbd, beanName, args);

            // 加载当前bean依赖的bean
            String[] dependsOn = mbd.getDependsOn();
            if (dependsOn != null) {
                // 存在依赖，递归实例化依赖的bean
                for (String dep : dependsOn) {
                    if (this.isDependent(beanName, dep)) {
                        // 检查dep是否依赖beanName，从而导致循环依赖
                        throw new
BeanCreationException(mbd.getResourceDescription(), beanName, "Circular depends-
on relationship between '" + beanName + "' and '" + dep + "'");
                    }
                    // 缓存依赖调用
                    this.registerDependentBean(dep, beanName);
                    this.getBean(dep);
                }
            }

            // 完成加载依赖的bean后，实例化mbd自身
            if (mbd.isSingleton()) {
                // scope == singleton
                sharedInstance = this.getSingleton(beanName, new
ObjectFactory<Object>() {
                    @Override
                    public Object getObject() throws BeansException {
                        try {
                            return createBean(beanName, mbd, args);
                        } catch (BeansException ex) {
                            // 清理工作，从单例缓存中移除
                            destroySingleton(beanName);
                            throw ex;
                        }
                    }
                });
                bean = this.getObjectForBeanInstance(sharedInstance, name,
beanName, mbd);
            } else if (mbd.isPrototype()) {
                // scope == prototype
                Object prototypeInstance;
```

```
                    this.beforePrototypeCreation(beanName);
                    // 创建bean
                    prototypeInstance = this.createBean(beanName, mbd, args);
                } finally {
                    this.afterPrototypeCreation(beanName);
                }
                // 返回对应的实例
                bean = this.getObjectForBeanInstance(prototypeInstance, name,
beanName, mbd);
            } else {
                // 其它scope
                String scopeName = mbd.getScope();
                final Scope scope = this.scopes.get(scopeName);
                if (scope == null) {
                    throw new IllegalStateException("No Scope registered for
scope name '" + scopeName + "'");
                }
                try {
                    Object scopedInstance = scope.get(beanName, new
ObjectFactory<Object>() {
                        @Override
                        public Object getObject() throws BeansException {
                            beforePrototypeCreation(beanName);
                            try {
                                return createBean(beanName, mbd, args);
                            } finally {
                                afterPrototypeCreation(beanName);
                            }
                        }
                    });
                    // 返回对应的实例
                    bean = this.getObjectForBeanInstance(scopedInstance, name,
beanName, mbd);
                } catch (IllegalStateException ex) {
                    throw new BeanCreationException(beanName, "Scope '" +
scopeName + "' is not active for the current thread; consider defining a scoped
proxy for this bean if you intend to refer to it from a singleton", ex);
                }
            }
        } catch (BeansException ex) {
            cleanupAfterBeanCreationFailure(beanName);
            throw ex;
        }
    }

    // 检查需要的类型是否符合bean的实际类型，对应getBean时指定的requireType
    if (requiredType != null && bean != null &&
!requiredType.isAssignableFrom(bean.getClass())) {
        try {
            // 执行类型转换，转换成期望的类型
            return this.getTypeConverter().convertIfNecessary(bean,
requiredType);
        } catch (TypeMismatchException ex) {
            if (logger.isDebugEnabled()) {
                logger.debug("Failed to convert bean '" + name + "' to required
type '" + ClassUtils.getQualifiedName(requiredType) + "'", ex);
```

```
bean.getClass());
            }
        }
        return (T) bean;
    }
```

4、BeanNameGenerator及其实现类

4.1、BeanNameGenerator

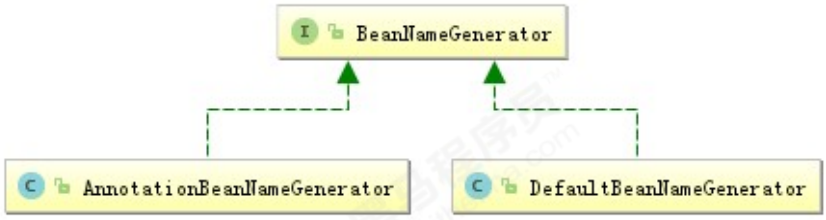
BeanNameGenerator接口位于 org.springframework.beans.factory.support 包下面:

```
/**
 * Strategy interface for generating bean names for bean definitions.
 *
 * @author Juergen Hoeller
 * @since 2.0.3
 */
public interface BeanNameGenerator {

    /**
     * Generate a bean name for the given bean definition.
     * @param definition the bean definition to generate a name for
     * @param registry the bean definition registry that the given definition
     * is supposed to be registered with
     * @return the generated bean name
     */
    String generateBeanName(BeanDefinition definition, BeanDefinitionRegistry
registry);

}
```

它有两个实现类：分别是:



其中DefaultBeanNameGenerator是给资源文件加载bean时使用（BeanDefinitionReader中使用）；AnnotationBeanNameGenerator是为了处理注解生成bean name的情况。

4.2、AnnotationBeanNameGenerator

```
/**
 * 此方法是接口中抽象方法的实现。
 * 该方法分为两个部分：
 *     第一个部分：当指定了bean的名称，则直接使用指定的名称。
```

```
@Override
public String generateBeanName(BeanDefinition definition, BeanDefinitionRegistry
registry) {
    //判断bean的定义信息是否为基于注解的
    if (definition instanceof AnnotatedBeanDefinition) {
        //解析注解中的属性，看看有没有指定的bean的唯一标识
        String beanName =
determineBeanNameFromAnnotation((AnnotatedBeanDefinition) definition);
        if (StringUtils.hasText(beanName)) {
            //返回注解的属性指定的bean的唯一标识
            return beanName;
        }
    }
    // 调用方法，使用注解bean名称的命名规则，生成bean的唯一标识
    return buildDefaultBeanName(definition, registry);
}

/**
 * Derive a default bean name from the given bean definition.
 * <p>The default implementation delegates to {@link
#buildDefaultBeanName(BeanDefinition)}.
 * @param definition the bean definition to build a bean name for
 * @param registry the registry that the given bean definition is being
registered with
 * @return the default bean name (never {@code null})
 */
protected String buildDefaultBeanName(BeanDefinition definition,
BeanDefinitionRegistry registry) {
  return buildDefaultBeanName(definition);
}

/**
 * Derive a default bean name from the given bean definition.
 * <p>The default implementation simply builds a decapitalized version
 * of the short class name: e.g. "mypackage.MyJdbcDao" -> "myJdbcDao".
 * <p>Note that inner classes will thus have names of the form
 * "outerClassName.InnerClassName", which because of the period in the
 * name may be an issue if you are autowiring by name.
 * @param definition the bean definition to build a bean name for
 * @return the default bean name (never {@code null})
 */
protected String buildDefaultBeanName(BeanDefinition definition) {
  String beanClassName = definition.getBeanClassName();
  Assert.state(beanClassName != null, "No bean class name set");
  String shortClassName = ClassUtils.getShortName(beanClassName);
  return Introspector.decapitalize(shortClassName);
}
```

ClassUtils的代码节选：

```
/**
 * Miscellaneous class utility methods.
 * Mainly for internal use within the framework.
 *
 * @author Juergen Hoeller
 * @author Keith Donald
```

```
 * @since 1.1
 * @see TypeUtils
 * @see ReflectionUtils
 */
public abstract class ClassUtils {

    /** Suffix for array class names: {@code "[]"}. */
    public static final String ARRAY_SUFFIX = "[]";

    /** Prefix for internal array class names: {@code "["}. */
    private static final String INTERNAL_ARRAY_PREFIX = "[";

    /** Prefix for internal non-primitive array class names: {@code "[L"}. */
    private static final String NON_PRIMITIVE_ARRAY_PREFIX = "[L";

    /** The package separator character: {@code '.'}. */
    private static final char PACKAGE_SEPARATOR = '.';

    /** The path separator character: {@code '/'}. */
    private static final char PATH_SEPARATOR = '/';

    /** The inner class separator character: {@code '$'}. */
    private static final char INNER_CLASS_SEPARATOR = '$';

    /** The CGLIB class separator: {@code "$$"}. */
    public static final String CGLIB_CLASS_SEPARATOR = "$$";

    /** The ".class" file suffix. */
    public static final String CLASS_FILE_SUFFIX = ".class";

    /**
     * Get the class name without the qualified package name.
     * @param className the className to get the short name for
     * @return the class name of the class without the package name
     * @throws IllegalArgumentException if the className is empty
     */
    public static String getShortName(String className) {
        Assert.hasLength(className, "Class name must not be empty");
        int lastDotIndex = className.lastIndexOf(PACKAGE_SEPARATOR);
        int nameEndIndex = className.indexOf(CGLIB_CLASS_SEPARATOR);
        if (nameEndIndex == -1) {
            nameEndIndex = className.length();
        }
        String shortName = className.substring(lastDotIndex + 1, nameEndIndex);
        shortName = shortName.replace(INNER_CLASS_SEPARATOR, PACKAGE_SEPARATOR);
        return shortName;
    }

    //其余代码略
}
```

4.3、DefaultBeanNameGenerator

```
/**
```

```
 * {@link BeanDefinitionReaderUtils#generateBeanName(BeanDefinition,
BeanDefinitionRegistry)}.
 *
 * @author Juergen Hoeller
 * @since 2.0.3
 */
public class DefaultBeanNameGenerator implements BeanNameGenerator {

    @Override
    public String generateBeanName(BeanDefinition definition,
BeanDefinitionRegistry registry) {
        return BeanDefinitionReaderUtils.generateBeanName(definition, registry);
    }
}
```

DefaultBeanNameGenerator类将具体的处理方式委托给了，BeanDefinitionReaderUtils#generateBeanName(BeanDefinition, BeanDefinitionRegistry)方法处理。

以下是代码节选：

```
public abstract class BeanDefinitionReaderUtils {

    public static String generateBeanName(BeanDefinition beanDefinition,
BeanDefinitionRegistry registry)
            throws BeanDefinitionStoreException {
        //此方法除了bean的定义信息和定义注册之外，还有一个布尔类型的值，用于确定是内部bean还
是顶层bean
        return generateBeanName(beanDefinition, registry, false);
    }

    /**
     * 生成bean的唯一标识（默认规则）
     */
    public static String generateBeanName(
            BeanDefinition definition, BeanDefinitionRegistry registry, boolean
isInnerBean)
            throws BeanDefinitionStoreException {

        String generatedBeanName = definition.getBeanClassName();
        if (generatedBeanName == null) {
            if (definition.getParentName() != null) {
                generatedBeanName = definition.getParentName() + "$child";
            }
            else if (definition.getFactoryBeanName() != null) {
                generatedBeanName = definition.getFactoryBeanName() +
"$created";
            }
        }
        if (!StringUtils.hasText(generatedBeanName)) {
            throw new BeanDefinitionStoreException("Unnamed bean definition
specifies neither " +
                    "'class' nor 'parent' nor 'factory-bean' - can't generate
bean name");
        }
```

```
            // Inner bean: generate identity hashcode suffix.
            id = generatedBeanName + GENERATED_BEAN_NAME_SEPARATOR +
ObjectUtils.getIdentityHexString(definition);
        }
        else {
            // Top-level bean: use plain class name with unique suffix if
necessary.
            return uniqueBeanName(generatedBeanName, registry);
        }
        return id;
    }

    //其他代码略
}
```

5、ScopedProxyMode枚举

```
/**
 * Enumerates the various scoped-proxy options.
 *
 * <p>For a more complete discussion of exactly what a scoped proxy is, see the
 * section of the Spring reference documentation entitled '<em>Scoped beans as
 * dependencies</em>'.
 *
 * @author Mark Fisher
 * @since 2.5
 * @see ScopeMetadata
 */
public enum ScopedProxyMode {

    /**
     * Default typically equals {@link #NO}, unless a different default
     * has been configured at the component-scan instruction level.
     */
    DEFAULT,

    /**
     * Do not create a scoped proxy.
     * <p>This proxy-mode is not typically useful when used with a
     * non-singleton scoped instance, which should favor the use of the
     * {@link #INTERFACES} or {@link #TARGET_CLASS} proxy-modes instead if it
     * is to be used as a dependency.
     */
    NO,

    /**
     * Create a JDK dynamic proxy implementing <i>all</i> interfaces exposed by
     * the class of the target object.
     */
    INTERFACES,

    /**
     * Create a class-based proxy (uses CGLIB).
     */
    TARGET_CLASS;
```

6、自定义组件扫描过滤规则

6.1、FilterType枚举

```
public enum FilterType {

    /**
     * Filter candidates marked with a given annotation.
     * @see org.springframework.core.type.filter.AnnotationTypeFilter
     */
    ANNOTATION,

    /**
     * Filter candidates assignable to a given type.
     * @see org.springframework.core.type.filter.AssignableTypeFilter
     */
    ASSIGNABLE_TYPE,

    /**
     * Filter candidates matching a given AspectJ type pattern expression.
     * @see org.springframework.core.type.filter.AspectJTypeFilter
     */
    ASPECTJ,

    /**
     * Filter candidates matching a given regex pattern.
     * @see org.springframework.core.type.filter.RegexPatternTypeFilter
     */
    REGEX,

    /** Filter candidates using a given custom
     * {@link org.springframework.core.type.filter.TypeFilter} implementation.
     */
    CUSTOM

}
```

6.2、TypeFilter接口

```
public interface TypeFilter {

    /**
     * 此方法返回一个boolean类型的值。
     * 当返回true时，表示加入到spring的容器中。返回false时，不加入容器。
     * 参数metadataReader：表示读取到的当前正在扫描的类的信息
     * 参数metadataReaderFactory：表示可以获得到其他任何类的信息
     */
    boolean match(MetadataReader metadataReader, MetadataReaderFactory
metadataReaderFactory)
            throws IOException;

}
```

6.3、使用Spring提供的过滤规则-AnnotationTypeFilter

```
/**
 * @author 黑马程序员
 * @Company http://www.itheima.com
 * 当我们使用注解驱动开发JavaEE项目时，spring提供的容器分为RootApplicationContext和
 * ServletApplicationContext。此时我们不希望Root容器创建时把Controller加入到容器中，
 * 就可以使用过滤规则排除@Controller注解配置的Bean对象。
 */
@Configuration
@ComponentScan(value = "com.itheima",excludeFilters = @ComponentScan.Filter(type
= FilterType.ANNOTATION,classes = Controller.class))
public class SpringConfiguration {
}
```

6.4、自定义过滤规则

6.4.1、场景分析

在实际开发中，有很多下面这种业务场景：一个业务需求根据环境的不同可能会有很多种实现。针对不同的环境，要加载不同的实现。我们看下面这个案例：

我们现在是一个汽车销售集团，在成立之初，只是在北京销售汽车，我们的项目研发完成后只在北京部署上线。但随着公司的业务发展，现在全国各地均有销售大区，总部设在北京。各大区有独立的项目部署，但是每个大区的业绩计算和绩效提成的计算方式并不相同。

例如：

在华北区销售一台豪华级轿车绩效算5，提成销售额1%，销售豪华级SUV绩效算3，提成是0.5%。

在西南区销售一台豪华级轿车绩效算3，提成销售额0.5%，销售豪华级SUV绩效算5，提成是1.5%。

这时，我们如果针对不同大区对项目源码进行删减替换，会带来很多不必要的麻烦。而如果加入一些if/else的判断，显然过于简单粗暴。此时应该考虑采用桥接设计模式，把将涉及到区域性差异的模块功能单独抽取到代表区域功能的接口中。针对不同区域进行实现。并且在扫描组件注册到容器中时，采用哪个区域的具体实现，应该采用配置文件配置起来。而自定义TypeFilter就可以实现注册指定区域的组件到容器中。

6.4.2、代码实现

```
/**
 * 区域的注解
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
```

```
public @interface District {

    /**
     * 指定区域的名称
     * @return
     */
    String value()  default "";
}

/**
 * 销售分成的桥接接口
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
public interface DistrictPercentage {

    /**
     * 不同车型提成
     * @param carType
     */
    void salePercentage(String carType);
}

/**
 * 绩效计算桥接接口
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
public interface DistrictPerformance {

    /**
     * 计算绩效
     * @param carType
     */
    void calcPerformance(String carType);
}

/**
 * 华北区销售分成具体实现
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
@Component("districtPercentage")
@District("north")
public class NorthDistrictPercentage implements DistrictPercentage {


    @Override
    public void salePercentage(String carType) {
        if("SUV".equalsIgnoreCase(carType)) {
            System.out.println("华北区"+carType+"提成1%");
        }else if("car".equalsIgnoreCase(carType)){
            System.out.println("华北区"+carType+"提成0.5%");
        }
    }
}
```

```
 * 华北区销售绩效具体实现
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
@Component("districtPerformance")
@District("north")
public class NorthDistrictPerformance implements DistrictPerformance {

    @Override
    public void calcPerformance(String carType) {
        if("SUV".equalsIgnoreCase(carType)) {
            System.out.println("华北区"+carType+"绩效3");
        }else if("car".equalsIgnoreCase(carType)){
            System.out.println("华北区"+carType+"绩效5");
        }
    }
}

/**
 * 西南区销售分成具体实现
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
@Component("districtPercentage")
@District("southwest")
public class SouthwestDistrictPercentage implements DistrictPercentage {

    @Override
    public void salePercentage(String carType) {
        if("SUV".equalsIgnoreCase(carType)) {
            System.out.println("西南区"+carType+"提成1.5%");
        }else if("car".equalsIgnoreCase(carType)){
            System.out.println("西南区"+carType+"提成0.5%");
        }
    }
}


/**
 * 西南区绩效计算具体实现
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
@Component("districtPerformance")
@District("southwest")
public class SouthwestDistrictPerformance implements DistrictPerformance {

    @Override
    public void calcPerformance(String carType) {
        if("SUV".equalsIgnoreCase(carType)) {
            System.out.println("西南区"+carType+"绩效5");
        }else if("car".equalsIgnoreCase(carType)){
            System.out.println("西南区"+carType+"绩效3");
        }
    }
}
```

```
 * spring的配置类
 * 用于替代xml配置
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
@Configuration
@PropertySource(value = "classpath:district.properties")
@ComponentScan(value = "com.itheima",
            excludeFilters =
@ComponentScan.Filter(type=FilterType.CUSTOM,classes =
DistrictTypeFilter.class))
public class SpringConfiguration {
}
```

```
/**
 * spring的自定义扫描规则
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
public class DistrictTypeFilter extends AbstractTypeHierarchyTraversingFilter {

    //定义路径校验类对象
    private PathMatcher pathMatcher;

    //注意:使用@Value注解的方式是获取不到配置值的。
    //因为Spring的生命周期里，负责填充属性值的InstantiationAwareBeanPostProcessor 与
TypeFilter的实例化过程压根搭不上边。
//    @Value("${common.district.name}")
    private String districtName;

    /**
     * 默认构造函数
     */
    public DistrictTypeFilter() {
        //1.第一个参数：不考虑基类。2.第二个参数：不考虑接口上的信息
        super(false, false);


        //借助Spring默认的Resource通配符路径方式
        pathMatcher = new AntPathMatcher();

        //硬编码读取配置信息
        try {
            Properties loadAllProperties =
PropertiesLoaderUtils.loadAllProperties("district.properties");
            districtName =
loadAllProperties.getProperty("common.district.name");
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    //注意本类将注册为Exclude，返回true代表拒绝
    @Override
```

```
            if (!isPotentialPackageClass(className)) {
                return false;
            }

            // 判断当前区域是否和所配置的区域一致，不一致则阻止载入Spring容器
            Class<?> clazz = ClassUtils.forName(className,
DistrictTypeFilter.class.getClassLoader());
            District districtAnnotation = clazz.getAnnotation(District.class);
            if(null == districtAnnotation){
                return false;
            }
            final String districtValue = districtAnnotation.value();
            return (!districtName.equalsIgnoreCase(districtValue));
        }catch (Exception e){
            throw new RuntimeException(e);
        }
    }

    // 潜在的满足条件的类的类名，指定package下
    private static final String PATTERN_STANDARD = ClassUtils
            .convertClassNameToResourcePath("com.itheima.service.impl.*");

    // 本类逻辑中可以处理的类 -- 指定package下的才会进行逻辑判断，
    private boolean isPotentialPackageClass(String className) {
        // 将类名转换为资源路径，以进行匹配测试
        final String path =
ClassUtils.convertClassNameToResourcePath(className);
        return pathMatcher.match(PATTERN_STANDARD, path);
    }

}
```

```
/**
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
public class SpringAnnotationTypeFilterTest {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext ac = new
AnnotationConfigApplicationContext("config");
        DistrictPerformance districtPerformance =
ac.getBean("districtPerformance", DistrictPerformance.class);
        districtPerformance.calcPerformance("SUV");

        DistrictPercentage districtPercentage =
ac.getBean("districtPercentage",DistrictPercentage.class);
        districtPercentage.salePercentage("car");

    }
}
```

7、@Import注解的高级分析

特别说明：

我们在注入bean对象时，可选的方式有很多种。

例如：

我们自己写的类，可以使用@Component,@Service,@Repository,@Controller等等。

我们导入的第三方库中的类，可以使用@Bean(当需要做一些初始化操作时，比如DataSource)，也可以使用@Import注解，直接指定要引入的类的字节码。

但是当我们的类很多时，在每个类上加注解会很繁琐，同时使用@Bean或者@Import写起来也很麻烦。此时我们就可以采用自定义ImportSelector或者ImportBeanDefinitionRegistrar来实现。顺便说一句，在SpringBoot中，@EnableXXX这样的注解，绝大多数都借助了ImportSelector或者ImportBeanDefinitionRegistrar。在我们的spring中，@EnableTransactionManagement就是借助了ImportSelector,而@EnableAspectJAutoporxy就是借助了ImportBeanDefinitionRegistrar。

共同点：

他们都是用于动态注册bean对象到容器中的。并且支持大批量的bean导入。

区别：

ImportSelector是一个接口，我们在使用时需要自己提供实现类。实现类中返回要注册的bean的全限定类名数组，然后执行ConfigurationClassParser类中中的processImports方法注册bean对象的。

ImportBeanDefinitionRegistrar也是一个接口，需要我们自己编写实现类，在实现类中手动注册bean到容器中。

注意事项：

实现了ImportSelector接口或者ImportBeanDefinitionRegistrar接口的类不会被解析成一个Bean注册到容器中。

同时，在注册到容器中时bean的唯一标识是全限定类名，而非短类名。

7.2、自定义ImportSelector

```
/**
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
public interface UserService {
    void saveUser();
}

/**
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
public class UserServiceImpl implements UserService {

    @Override
    public void saveUser() {
        System.out.println("保存用户");
    }
}

/**
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
@Configuration
@ComponentScan("com.itheima")
```

```
}
```

```
/**
 * customeimport.properties配置文件中的内容：
 *      custome.importselector.expression= com.itheima.service.impl.*
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
public class CustomeImportSelector implements ImportSelector {

    private String expression;

    public CustomeImportSelector(){
        try {
            Properties loadAllProperties =
PropertiesLoaderUtils.loadAllProperties("customeimport.properties");
            expression =
loadAllProperties.getProperty("custome.importselector.expression");
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    /**
     * 生成要导入的bean全限定类名数组
     * @param importingClassMetadata
     * @return
     */
    @Override
    public String[] selectImports(AnnotationMetadata importingClassMetadata) {
        //1.定义扫描包的名称
        String[] basePackages = null;
        //2.判断有@Import注解的类上是否有@ComponentScan注解
        if (importingClassMetadata.hasAnnotation(ComponentScan.class.getName()))
{
            //3.取出@ComponentScan注解的属性
            Map<String, Object> annotationAttributes =
importingClassMetadata.getAnnotationAttributes(ComponentScan.class.getName());
            //4.取出属性名称为basePackages属性的值
            basePackages = (String[]) annotationAttributes.get("basePackages");
        }
        //5.判断是否有此属性（如果没有ComponentScan注解则属性值为null，如果有
ComponentScan注解，则basePackages默认为空数组）
        if (basePackages == null || basePackages.length == 0) {
            String basePackage = null;
            try {
                //6.取出包含@Import注解类的包名
                basePackage =
Class.forName(importingClassMetadata.getClassName()).getPackage().getName();
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            }
            //7.存入数组中
            basePackages = new String[] {basePackage};
```

```
        ClassPathScanningCandidateComponentProvider scanner = new
ClassPathScanningCandidateComponentProvider(false);
        //9.创建类型过滤器(此处使用切入点表达式类型过滤器)
        TypeFilter typeFilter = new
AspectJTypeFilter(expression,this.getClass().getClassLoader());
        //10.给扫描器加入类型过滤器
        scanner.addIncludeFilter(typeFilter);
        //11.创建存放全限定类名的集合
        Set<String> classes = new HashSet<>();
        //12.填充集合数据
        for (String basePackage : basePackages) {
            scanner.findCandidateComponents(basePackage).forEach(beanDefinition
-> classes.add(beanDefinition.getBeanClassName()));
        }
        //13.按照规则返回
        return classes.toArray(new String[classes.size()]);
    }
}
```

```
/**
 * 测试类
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
public class SpringCustomeImportSelectorTest {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext ac = new
AnnotationConfigApplicationContext("config");
        String[] names = ac.getBeanDefinitionNames();
        for(String beanName : names){
            Object obj = ac.getBean(beanName);
            System.out.println(beanName+"============"+obj);
        }
    }
}
```

```
E:\Java\JDK1.8\jdk1.8.0_162\bin\java ...
org.springframework.context.annotation.internalConfigurationAnnotationProcessor============org.springframework.context.annotation.ConfigurationClassPostProcessor@243c4f91
org.springframework.context.annotation.internalAutowiredAnnotationProcessor============org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor@291ae
org.springframework.context.annotation.internalCommonAnnotationProcessor============org.springframework.context.annotation.CommonAnnotationBeanPostProcessor@61df66b6
org.springframework.context.event.internalEventListenerProcessor============org.springframework.context.event.EventListenerMethodProcessor@50eac852
org.springframework.context.event.internalEventListenerFactory============org.springframework.context.event.DefaultEventListenerFactory@16ec5519
springConfiguration==========config.SpringConfiguration$$EnhancerBySpringCGLIB$$52f1bea8@2f7298b
com.itheima.service.impl.UserServiceImpl==========com.itheima.service.impl.UserServiceImpl@188715b5

Process finished with exit code 0
```

7.3、自定义ImportBeanDefinitionRegistrar

借助7.2小节的案例代码，只需要把配置改一下：

```
/**
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
@Configuration
@ComponentScan("com.itheima")
@Import(CustomeImportDefinitionRegistrar.class)
public class SpringConfiguration {
```

```
/**
 * 自定义bean导入注册器
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
public class CustomeImportDefinitionRegistrar implements
ImportBeanDefinitionRegistrar {

    private String expression;

    public CustomeImportDefinitionRegistrar(){
        try {
            Properties loadAllProperties =
PropertiesLoaderUtils.loadAllProperties("customeimport.properties");
            expression =
loadAllProperties.getProperty("custome.importselector.expression");
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }


    @Override
    public void registerBeanDefinitions(AnnotationMetadata
importingClassMetadata, BeanDefinitionRegistry registry) {
        //1.定义扫描包的名称
        String[] basePackages = null;
        //2.判断有@Import注解的类上是否有@ComponentScan注解
        if (importingClassMetadata.hasAnnotation(ComponentScan.class.getName()))
{
            //3.取出@ComponentScan注解的属性
            Map<String, Object> annotationAttributes =
importingClassMetadata.getAnnotationAttributes(ComponentScan.class.getName());
            //4.取出属性名称为basePackages属性的值
            basePackages = (String[]) annotationAttributes.get("basePackages");
        }
        //5.判断是否有此属性（如果没有ComponentScan注解则属性值为null，如果有
ComponentScan注解，则basePackages默认为空数组）
        if (basePackages == null || basePackages.length == 0) {
            String basePackage = null;
            try {
                //6.取出包含@Import注解类的包名
                basePackage =
Class.forName(importingClassMetadata.getClassName()).getPackage().getName();
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            }
            //7.存入数组中
            basePackages = new String[] {basePackage};
        }
        //8.创建类路径扫描器
        ClassPathBeanDefinitionScanner scanner = new
ClassPathBeanDefinitionScanner(registry, false);
        //9.创建类型过滤器(此处使用切入点表达式类型过滤器)
```

```
        //10.给扫描器加入类型过滤器
        scanner.addIncludeFilter(typeFilter);
        //11.扫描指定包
        scanner.scan(basePackages);
    }
}
```

7.4、原理分析

我们写的自定义导入器的解析写在了ConfigurationClassParser类中的processImports方法，以下是源码节选：

```
private void processImports(ConfigurationClass configClass, SourceClass
currentSourceClass,
            Collection<SourceClass> importCandidates, boolean
checkForCircularImports) {

        if (importCandidates.isEmpty()) {
            return;
        }

        if (checkForCircularImports && isChainedImportOnStack(configClass)) {
            this.problemReporter.error(new CircularImportProblem(configClass,
this.importStack));
        }
        else {
            this.importStack.push(configClass);
            try {
                for (SourceClass candidate : importCandidates) {
                    //对ImportSelector的处理
                    if (candidate.isAssignable(ImportSelector.class)) {
                        // Candidate class is an ImportSelector -> delegate to
it to determine imports
                        Class<?> candidateClass = candidate.loadClass();
                        ImportSelector selector =
BeanUtils.instantiateClass(candidateClass, ImportSelector.class);
                        ParserStrategyUtils.invokeAwareMethods(
                                selector, this.environment, this.resourceLoader,
this.registry);
                        if (this.deferredImportSelectors != null && selector
instanceof DeferredImportSelector) {
                            //如果为延迟导入处理则加入集合当中
                            this.deferredImportSelectors.add(
                                    new
DeferredImportSelectorHolder(configClass, (DeferredImportSelector) selector));
                        }
                        else {
                            //根据ImportSelector方法的返回值来进行递归操作
                            String[] importClassNames =
selector.selectImports(currentSourceClass.getMetadata());
                            Collection<SourceClass> importSourceClasses =
asSourceClasses(importClassNames);
                            processImports(configClass, currentSourceClass,
importSourceClasses, false);
                        }
                    }
```

```
                        // Candidate class is an ImportBeanDefinitionRegistrar -
>
                        // delegate to it to register additional bean
definitions
                        Class<?> candidateClass = candidate.loadClass();
                        ImportBeanDefinitionRegistrar registrar =
                                BeanUtils.instantiateClass(candidateClass,
ImportBeanDefinitionRegistrar.class);
                        ParserStrategyUtils.invokeAwareMethods(
                                registrar, this.environment,
this.resourceLoader, this.registry);
                        configClass.addImportBeanDefinitionRegistrar(registrar,
currentSourceClass.getMetadata());
                    }
                    else {
                       // 如果当前的类既不是ImportSelector也不是
ImportBeanDefinitionRegistar就进行@Configuration的解析处理
                        // Candidate class not an ImportSelector or
ImportBeanDefinitionRegistrar ->
                        // process it as an @Configuration class
                        this.importStack.registerImport(
                                currentSourceClass.getMetadata(),
candidate.getMetadata().getClassName());

 processConfigurationClass(candidate.asConfigClass(configClass));
                    }
                }
            }
            catch (BeanDefinitionStoreException ex) {
                throw ex;
            }
            catch (Throwable ex) {
                throw new BeanDefinitionStoreException(
                        "Failed to process import candidates for configuration
class [" +
                        configClass.getMetadata().getClassName() + "]", ex);
            }
            finally {
                this.importStack.pop();
            }
        }
    }
```

8、自定义PropertySourceFactory实现YAML文件解析

8.1、PropertySourceFactory及DefaultPropertySourceFactory

```
/**
 * Strategy interface for creating resource-based {@link PropertySource}
wrappers.
 *
 * @author Juergen Hoeller
 * @since 4.3
 * @see DefaultPropertySourceFactory
 */
```

```
    /**
     * Create a {@link PropertySource} that wraps the given resource.
     * @param name the name of the property source
     * @param resource the resource (potentially encoded) to wrap
     * @return the new {@link PropertySource} (never {@code null})
     * @throws IOException if resource resolution failed
     */
    PropertySource<?> createPropertySource(@Nullable String name,
EncodedResource resource) throws IOException;

}
```

```
/**
 * The default implementation for {@link PropertySourceFactory},
 * wrapping every resource in a {@link ResourcePropertySource}.
 *
 * @author Juergen Hoeller
 * @since 4.3
 * @see PropertySourceFactory
 * @see ResourcePropertySource
 */
public class DefaultPropertySourceFactory implements PropertySourceFactory {

    @Override
    public PropertySource<?> createPropertySource(@Nullable String name,
EncodedResource resource) throws IOException {
        return (name != null ? new ResourcePropertySource(name, resource) : new
ResourcePropertySource(resource));
    }

}
```

8.2、执行过程分析

8.2.1、ResourcePropertySource

```
/**
 *  DefaultPropertySourceFactory在创建PropertySource对象时使用的是此类的构造函数
 *  在构造时，调用的
 */
public class ResourcePropertySource extends PropertiesPropertySource {

    /**
     * 当我们没有指定名称时，执行的是此构造函数，此构造函数调用的是父类的构造函数
     * 通过读取父类的构造函数，得知第二个参数是一个properties文件
     * spring使用了一个工具类获取properties文件
     */
    public ResourcePropertySource(EncodedResource resource) throws IOException {
        super(getNameForResource(resource.getResource()),
PropertiesLoaderUtils.loadProperties(resource));
        this.resourceName = null;
    }

    //其他方法略
```

8.2.2、PropertiesPropertySource

```
/**
 * 此类是ResourcePropertySource的父类
 */
public class PropertiesPropertySource extends MapPropertySource {

    /**
     * 此构造函数中包含两个参数：第一个是名称。第二个是解析好的properties文件
     */
    @SuppressWarnings({"unchecked", "rawtypes"})
    public PropertiesPropertySource(String name, Properties source) {
        super(name, (Map) source);
    }

    protected PropertiesPropertySource(String name, Map<String, Object> source)
{
        super(name, source);
    }

}
```

8.2.3、PropertiesLoaderUtils

```
/**
 * 获取properties的工具类
 */
public abstract class PropertiesLoaderUtils {

    private static final String XML_FILE_EXTENSION = ".xml";


    /**
     * ResourcePropertySource类中的构造函数就是执行了此方法
     */
    public static Properties loadProperties(EncodedResource resource) throws
IOException {
        Properties props = new Properties();
        fillProperties(props, resource);
        return props;
    }

    /**
     *
     */
    public static void fillProperties(Properties props, EncodedResource
resource)
            throws IOException {

        fillProperties(props, resource, new DefaultPropertiesPersister());
    }

    /**
```

```
     */
    static void fillProperties(Properties props, EncodedResource resource,
PropertiesPersister persister)
            throws IOException {

        InputStream stream = null;
        Reader reader = null;
        try {
            String filename = resource.getResource().getFilename();
            if (filename != null && filename.endsWith(XML_FILE_EXTENSION)) {
                stream = resource.getInputStream();
                //读取xml文件
                persister.loadFromXml(props, stream);
            }
            else if (resource.requiresReader()) {
                reader = resource.getReader();
                //读取properties文件
                persister.load(props, reader);
            }
            else {
                stream = resource.getInputStream();
                persister.load(props, stream);
            }
        }
        finally {
            if (stream != null) {
                stream.close();
            }
            if (reader != null) {
                reader.close();
            }
        }
    }
    //其他代码略
}
```

8.2.4、PropertiesPersister

```
/**
 * spring中声明的解析xml和properties文件的接口
 */
public interface PropertiesPersister {


    /**
     * Load properties from the given Reader into the given
     * Properties object.
     * @param props the Properties object to load into
     * @param reader the Reader to load from
     * @throws IOException in case of I/O errors
     */
    void load(Properties props, Reader reader) throws IOException;

    /**
```

```
     * @param props the Properties object to load into
     * @param is the InputStream to load from
     * @throws IOException in case of I/O errors
     * @see java.util.Properties#loadFromXML(java.io.InputStream)
     */
    void loadFromXml(Properties props, InputStream is) throws IOException;

    //其他代码略
}
```

```
/**
 *  PropertiesPersister接口的实现类
 */
public class DefaultPropertiesPersister implements PropertiesPersister {


    @Override
    public void load(Properties props, Reader reader) throws IOException {
        props.load(reader);

    @Override
    public void loadFromXml(Properties props, InputStream is) throws IOException
{
        props.loadFromXML(is);
    }

    //其他代码略
}
```

8.3、自定义PropertySourceFactory

我们通过分析@PropertySource源码，得知默认情况下此注解只能解析properties文件和xml文件，而遇到yaml（yml）文件，解析就会报错。此时就需要我们自己编写一个PropertySourceFactory的实现类，借助yaml解析器，实现yml文件的解析。

8.3.1、编写yml配置文件

```
jdbc:
 driver: com.mysql.jdbc.Driver
 url: jdbc:mysql://localhost:3306/spring_day01
 username: root
 password: 1234
```

8.3.2、导入yaml解析器的坐标

```
<!-- yaml解析器 https://mvnrepository.com/artifact/org.yaml/snakeyaml -->
<dependency>
  <groupId>org.yaml</groupId>
  <artifactId>snakeyaml</artifactId>
  <version>1.23</version>
</dependency>
```

8.3.3、编写自定义PropertySourceFactory

```
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
public class CustomerPropertySourceFactory implements PropertySourceFactory {

    /**
     * 重写接口中的方法
     * @param name
     * @param resource
     * @return
     * @throws IOException
     */
    @Override
    public PropertySource<?> createPropertySource(String name, EncodedResource
resource) throws IOException {
        //1.创建yaml文件解析工厂
        YamlPropertiesFactoryBean yaml = new YamlPropertiesFactoryBean();
        //2.设置资源内容
        yaml.setResources(resource.getResource());
        //3.解析成properties文件
        Properties properties = yaml.getObject();
        //4.返回符合spring的PropertySource对象
        return name != null ? new PropertiesPropertySource(name,properties) :
new PropertiesPropertySource(resource.getResource().getFilename(), properties);

    }
}
```

8.3.4、使用@PropertyeSource的factory属性配置自定义工厂

```
/**
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
@Configuration
@Import(JdbcConfig.class)
@PropertySource(value = "classpath:jdbc.yml",factory =
CustomerPropertySourceFactory.class)
@ComponentScan("com.itheima")
public class SpringConfiguration {
}
```

```
/**
 * 连接数据库的配置
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
public class JdbcConfig {

    @Value("${jdbc.driver}")
    private String driver;
    @Value("${jdbc.url}")
    private String url;
    @Value("${jdbc.username}")
```

```
    private String password;


    @Bean("jdbcTemplate")
    public JdbcTemplate createJdbcTemplate(DataSource dataSource){
        return new JdbcTemplate(dataSource);
    }


    @Bean("dataSource")
    public DataSource createDataSource(){
        System.out.println(driver);
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName(driver);
        dataSource.setUrl(url);
        dataSource.setUsername(username);
        dataSource.setPassword(password);
        return dataSource;
    }
}
```

8.3.5、测试运行结果

```
/**
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
public class SpringAnnotationDrivenTest {

    /**
     * 测试
     * @param args
     */
    public static void main(String[] args) {
        ApplicationContext ac = new
AnnotationConfigApplicationContext("config");
    }
}
```

9、@Profile注解的使用

9.1、使用场景分析

@Profile注解是spring提供的一个用来标明当前运行环境的注解。我们正常开发的过程中经常遇到的问题是，开发环境是一套环境，测试是一套环境，线上部署又是一套环境。这样从开发到测试再到部署，会对程序中的配置修改多次，尤其是从测试到上线这个环节，让测试的也不敢保证改了哪个配置之后能不能在线上运行。为了解决上面的问题，我们一般会使用一种方法，就是针对不同的环境进行不同的配置，从而在不同的场景中跑我们的程序。

而spring中的@Profile注解的作用就体现在这里。在spring使用DI来注入的时候，能够根据当前制定的运行环境来注入相应的bean。最常见的就是使用不同的DataSource了。

9.2、代码实现

9.2.1、自定义不同环境的数据源

```
package config;

import com.alibaba.druid.pool.DruidDataSource;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Profile;

/**
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
public class JdbcConfig {

    @Value("${jdbc.driver}")
    private String driver;
    @Value("${jdbc.url}")
    private String url;
    @Value("${jdbc.username}")
    private String username;
    @Value("${jdbc.password}")
    private String password;
```

```
    /**
     * @return
     */
    @Bean("dataSource")
    @Profile("dev")
    public DruidDataSource createDevDataSource(){
        DruidDataSource dataSource = new DruidDataSource();
        dataSource.setDriverClassName(driver);
        dataSource.setUrl(url);
        dataSource.setUsername(username);
        dataSource.setPassword(password);
        dataSource.setMaxActive(10);
        return dataSource;
    }

    /**
     * @return
     */
    @Bean("dataSource")
    @Profile("test")
    public DruidDataSource createTestDataSource(){
        DruidDataSource dataSource = new DruidDataSource();
        dataSource.setDriverClassName(driver);
        dataSource.setUrl(url);
        dataSource.setUsername(username);
        dataSource.setPassword(password);
        dataSource.setMaxActive(50);
        return dataSource;
    }

    /**
     * @return
     */
    @Bean("dataSource")
    @Profile("produce")
    public DruidDataSource createProduceDataSource(){
        DruidDataSource dataSource = new DruidDataSource();
        dataSource.setDriverClassName(driver);
        dataSource.setUrl(url);
        dataSource.setUsername(username);
        dataSource.setPassword(password);
        dataSource.setMaxActive(100);
        return dataSource;
    }
}
```

9.2.2、编写配置类

```
 * @Company http://www.itheima.com
 */
@Configuration
@Import(JdbcConfig.class)
public class SpringConfiguration {
}
```

9.2.3、编写测试类

```
/**
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = SpringConfiguration.class)
@ActiveProfiles("test")
public class SpringProfileTest {

    @Autowired
    private DruidDataSource druidDataSource;

    @Test
    public void testProfile(){
        System.out.println(druidDataSource.getMaxActive());
    }
}
```

9.2.4、测试结果