

5. Spring IoC&DI

本节目标

1. 了解Spring, Spring MVC, Spring Boot 之间的联系及区别
2. 掌握IoC&DI的概念以及写法

1. IoC & DI 入门

在前面的章节中, 我们学习了Spring Boot和Spring MVC的开发, 可以完成一些基本功能的开发了, 但是什么是Spring呢? Spring, Spring Boot 和SpringMVC又有什么关系呢? 咱们还是带着问题去学习.

我们先看什么是Spring

1.1 Spring 是什么?

通过前面的学习, 我们知道了Spring是一个开源框架, 他让我们的开发更加简单. 他支持广泛的应用场景, 有着活跃而庞大的社区, 这也是Spring能够长久不衰的原因.

但是这个概念相对来说, 还是比较抽象.

我们用一句更具体的话来概括Spring, 那就是: **Spring 是包含了众多工具方法的 IoC 容器**

那问题来了, 什么是容器? 什么是 IoC 容器? 接下来我们一起来看看

1.1.1 什么是容器?

容器是用来容纳某种物品的（基本）装置。——来自：百度百科

生活中的水杯, 垃圾桶, 冰箱等等这些都是容器.

我们想想, 之前课程我们接触的容器有哪些?

- List/Map -> 数据存储容器
- Tomcat -> Web 容器

1.1.2 什么是 IoC?

IoC 是Spring的核心思想, 也是常见的面试题, 那什么是IoC呢?

其实IoC我们在前面已经使用了, 我们在前面讲到, 在类上面添加 `@RestController` 和 `@Controller` 注解, 就是把这个对象交给Spring管理, Spring 框架启动时就会加载该类. 把对象交给Spring管理, 就是IoC思想.

IoC: Inversion of Control (控制反转), 也就是说 Spring 是一个"控制反转"的容器。

什么是控制反转呢? 也就是**控制权**反转. 什么的控制权发生了反转? 获得依赖对象的过程被反转了
也就是说, 当需要某个对象时, 传统开发模式中需要自己通过 new 创建对象, 现在不需要再进行创建, 把创建对象的任务交给容器, 程序中只需要依赖注入 (Dependency Injection,DI)就可以了.
这个容器称为: IoC容器. Spring是一个IoC容器, 所以有时Spring 也称为Spring 容器.

控制反转是一种思想, 在生活中也是处处体现.

比如自动驾驶, 传统驾驶方式, 车辆的横向和纵向驾驶控制权由驾驶员来控制, 现在交给了驾驶自动化系统来控制, 这也是控制反转思想在生活中的实现.

比如招聘, 企业的员工招聘,入职, 解雇等控制权, 由老板转交给给HR(人力资源)来处理

1.2 IoC 介绍

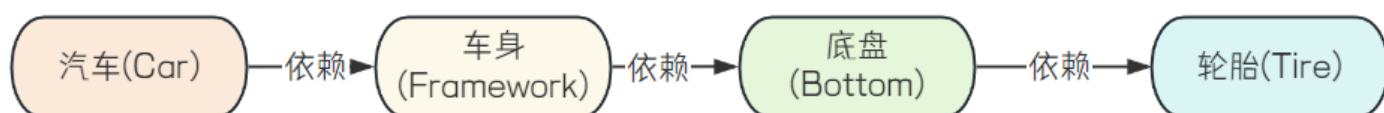
接下来我们通过案例来了解一下什么是IoC

需求: 造一辆车

1.2.1 传统程序开发

我们的实现思路是这样的:

先设计轮子(Tire), 然后根据轮子的大小设计底盘(Bottom), 接着根据底盘设计车身(Framework), 最后根据车身设计好整个汽车(Car)。这里就出现了一个"依赖"关系: 汽车依赖车身, 车身依赖底盘, 底盘依赖轮子.



最终程序的实现代码如下:

```
1 public class NewCarExample {
2     public static void main(String[] args) {
3         Car car = new Car();
4         car.run();
5     }
6
7     /**
8      * 汽车对象
9      */
10    static class Car {
11        private Framework framework;
12    }
```

```
13     public Car() {
14         framework = new Framework();
15         System.out.println("Car init....");
16     }
17     public void run(){
18         System.out.println("Car run...");
19     }
20 }
21
22 /**
23  * 车身类
24  */
25 static class Framework {
26     private Bottom bottom;
27
28     public Framework() {
29         bottom = new Bottom();
30         System.out.println("Framework init...");
31     }
32 }
33
34 /**
35  * 底盘类
36  */
37 static class Bottom {
38     private Tire tire;
39
40     public Bottom() {
41         this.tire = new Tire();
42         System.out.println("Bottom init...");
43     }
44 }
45
46 /**
47  * 轮胎类
48  */
49 static class Tire {
50     // 尺寸
51     private int size;
52
53     public Tire(){
54         this.size = 17;
55         System.out.println("轮胎尺寸: " + size);
56     }
57 }
58 }
```

1.2.2 问题分析

这样的设计看起来没问题，但是可维护性却很低。

接下来需求有了变更：随着对的车的需求量越来越大, 个性化需求也会越来越多，我们需要加工多种尺寸的轮胎。

那这个时候就要对上面的程序进行修改了，修改后的代码如下所示：

```
static class Tire {  
    // 尺寸  
    private int size;  
  
    public Tire(){  
        this.size = 17;  
        System.out.println("轮胎尺寸: " + size);  
    }  
}
```



```
static class Tire {  
    // 尺寸  
    private int size;  
  
    public Tire(int size){  
        this.size = size;  
        System.out.println("轮胎尺寸: " + size);  
    }  
}
```

修改之后, 其他调用程序也会报错, 我们需要继续修改

```
static class Bottom {  
    private Tire tire;  
  
    public Bottom() {  
        this.tire = new Tire();  
        System.out.println("Bottom init...");  
    }  
}
```



```
static class Bottom {  
    private Tire tire;  
  
    public Bottom(int size) {  
        this.tire = new Tire(size);  
        System.out.println("Bottom init...");  
    }  
}
```

```
static class Framework {  
    private Bottom bottom;  
  
    public Framework() {  
        bottom = new Bottom();  
        System.out.println("Framework init...");  
    }  
}
```



```
static class Framework {  
    private Bottom bottom;  
  
    public Framework(int size) {  
        bottom = new Bottom(size);  
        System.out.println("Framework init...");  
    }  
}
```

```

static class Car {
    private Framework framework;

    public Car() {
        framework = new Framework();
        System.out.println("Car init....");
    }
    public void run(){
        System.out.println("Car run...");
    }
}

```



```

static class Car {
    private Framework framework;

    public Car(int size) {
        framework = new Framework(size);
        System.out.println("Car init....");
    }
    public void run(){
        System.out.println("Car run...");
    }
}

```

完整代码如下:

```

1 public class NewCarExample {
2     public static void main(String[] args) {
3         Car car = new Car(20);
4         car.run();
5     }
6
7     /**
8      * 汽车对象
9      */
10    static class Car {
11        private Framework framework;
12
13        public Car(int size) {
14            framework = new Framework(size);
15            System.out.println("Car init....");
16        }
17        public void run(){
18            System.out.println("Car run...");
19        }
20    }
21
22    /**
23     * 车身类
24     */
25    static class Framework {
26        private Bottom bottom;
27
28        public Framework(int size) {
29            bottom = new Bottom(size);
30            System.out.println("Framework init...");
31        }
32    }
33

```

```

34     /**
35      * 底盘类
36      */
37     static class Bottom {
38         private Tire tire;
39
40         public Bottom(int size) {
41             this.tire = new Tire(size);
42             System.out.println("Bottom init...");
43         }
44     }
45
46     /**
47      * 轮胎类
48      */
49     static class Tire {
50         // 尺寸
51         private int size;
52
53         public Tire(int size){
54             this.size = size;
55             System.out.println("轮胎尺寸: " + size);
56         }
57     }
58 }

```

从以上代码可以看出，以上程序的问题是：当最底层代码改动之后，整个调用链上的所有代码都需要修改。

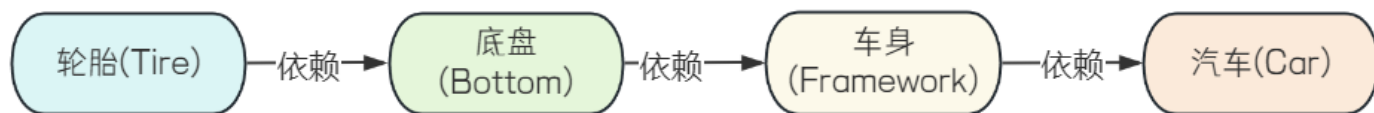
程序的耦合度非常高(修改一处代码, 影响其他处的代码修改)

1.2.3 解决方案

在上面的程序中, 我们是根据轮子的尺寸设计的底盘, 轮子的尺寸一改, 底盘的设计就得修改. 同样因为我们是根据底盘设计的车身, 那么车身也得改, 同理汽车设计也得改, 也就是整个设计几乎都得改

我们尝试换一种思路, 我们先设计汽车的大概样子, 然后根据汽车的样子来设计车身, 根据车身来设计底盘, 最后根据底盘来设计轮子. 这时候, 依赖关系就倒置过来了: 轮子依赖底盘, 底盘依赖车身, 车身依赖汽车

这就类似我们打造一辆完整的汽车, 如果所有的配件都是自己造, 那么当客户需求发生改变的时候, 比如轮胎的尺寸不再是原来的尺寸了, 那我们要自己动手来改了, 但如果我们是把轮胎外包出去, 那么即使是轮胎的尺寸发生变变了, 我们只需要向代理工厂下订单就行了, 我们自身是不需要出力的。



如何实现呢:

我们可以尝试不在每个类中自己创建下级类，如果自己创建下级类就会出现当下级类发生改变操作，自己也要跟着修改。

此时，我们只需要将原来由自己创建的下级类，改为传递的方式（也就是注入的方式），因为我们不需要在当前类中创建下级类了，所以下级类即使发生变化（创建或减少参数），当前类本身也无需修改任何代码，这样就完成了程序的解耦。

1.2.4 IoC程序开发

基于以上思路，我们把调用汽车的程序示例改造一下，把创建子类的方式，改为注入传递的方式。

具体实现代码如下：

```
1 public class IocCarExample {
2     public static void main(String[] args) {
3         Tire tire = new Tire(20);
4         Bottom bottom = new Bottom(tire);
5         Framework framework = new Framework(bottom);
6         Car car = new Car(framework);
7         car.run();
8     }
9
10    static class Car {
11        private Framework framework;
12
13        public Car(Framework framework) {
14            this.framework = framework;
15            System.out.println("Car init....");
16        }
17        public void run() {
18            System.out.println("Car run...");
19        }
20    }
21
22    static class Framework {
23        private Bottom bottom;
24
25        public Framework(Bottom bottom) {
26            this.bottom = bottom;
27            System.out.println("Framework init...");
```

```

28     }
29 }
30
31 static class Bottom {
32     private Tire tire;
33
34     public Bottom(Tire tire) {
35         this.tire = tire;
36         System.out.println("Bottom init...");
37     }
38 }
39
40 static class Tire {
41     private int size;
42
43     public Tire(int size) {
44         this.size = size;
45         System.out.println("轮胎尺寸: " + size);
46     }
47 }
48 }

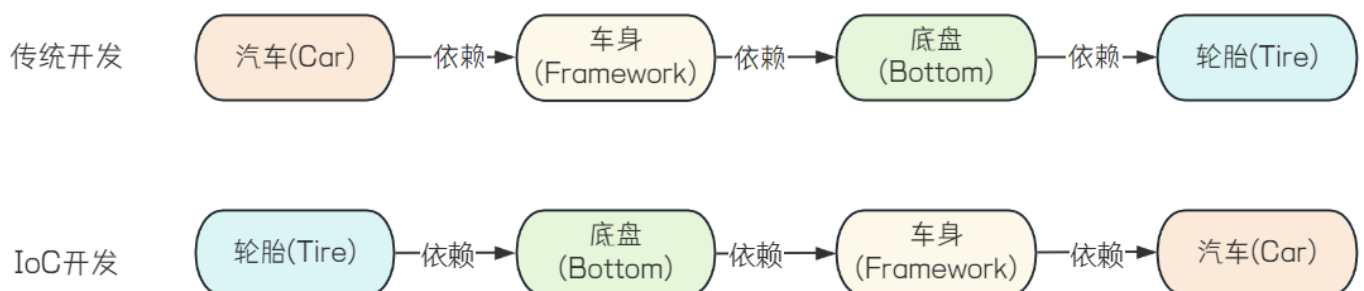
```

代码经过以上调整，无论底层类如何变化，整个调用链是不用做任何改变的，这样就完成了代码之间的**解耦**，从而实现了更加灵活、通用的程序设计了。

1.2.5 IoC 优势

在传统的代码中对象创建顺序是：Car -> Framework -> Bottom -> Tire

改进之后解耦的代码的对象创建顺序是：Tire -> Bottom -> Framework -> Car



我们发现了一个规律，通用程序的实现代码，类的创建顺序是反的，传统代码是 Car 控制并创建了 Framework，Framework 创建并创建了 Bottom，依次往下，而**改进之后的控制权发生的反转**，不再是使用方对象创建并控制依赖对象了，而是把依赖对象注入到当前对象中，依赖对象的控制权不再由当前类控制了。

这样的话，即使依赖类发生任何改变，当前类都是不受影响的，这就是典型的控制反转，也就是 IoC 的实现思想。

学到这里, 我们大概就知道了什么是控制反转了, 那什么是控制反转容器呢, 也就是IoC容器



这部分代码, 就是IoC容器做的工作.

从上面也可以看出来, IoC容器具备以下优点:

资源不由使用资源的双方管理, 而由不使用资源的第三方管理, 这可以带来很多好处。第一, 资源集中管理, 实现资源的可配置和易管理。第二, 降低了使用资源双方的依赖程度, 也就是我们说的耦合度。

1. 资源集中管理: IoC容器会帮我们管理一些资源(对象等), 我们需要使用时, 只需要从IoC容器中去取就可以了
2. 我们在创建实例的时候不需要了解其中的细节, 降低了使用资源双方的依赖程度, 也就是耦合度。

Spring 就是一种IoC容器, 帮助我们来做了这些资源管理.

1.3 DI 介绍

上面学习了IoC, 什么是DI呢?

DI: Dependency Injection(依赖注入)

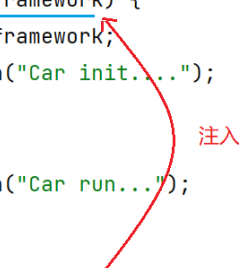
容器在运行期间, 动态的为应用程序提供运行时所依赖的资源, 称之为依赖注入。

程序运行时需要某个资源, 此时容器就为其提供这个资源.

从这点来看，依赖注入（DI）和控制反转（IoC）是从不同的角度的描述的同件事情，就是指通过引入 IoC 容器，利用依赖关系注入的方式，实现对象之间的解耦。

上述代码中，是通过构造函数的方式，把依赖对象注入到需要使用的对象中的

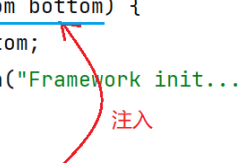
```
static class Car {  
    private Framework framework;  
  
    public Car(Framework framework) {  
        this.framework = framework;  
        System.out.println("Car init...");  
    }  
    public void run() {  
        System.out.println("Car run...");  
    }  
}
```



注入

The diagram shows a red arrow pointing from the `bottom` parameter in the `Framework` constructor to the `framework` parameter in the `Car` constructor, indicating the flow of dependency injection.

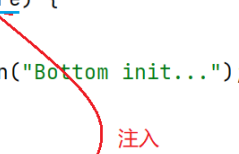
```
static class Framework {  
    private Bottom bottom;  
  
    public Framework(Bottom bottom) {  
        this.bottom = bottom;  
        System.out.println("Framework init...");  
    }  
}
```



注入

The diagram shows a red arrow pointing from the `tire` parameter in the `Bottom` constructor to the `bottom` parameter in the `Framework` constructor, indicating the flow of dependency injection.

```
static class Bottom {  
    private Tire tire;  
  
    public Bottom(Tire tire) {  
        this.tire = tire;  
        System.out.println("Bottom init...");  
    }  
}
```



注入

The diagram shows a red arrow pointing from the `size` parameter in the `Tire` constructor to the `tire` parameter in the `Bottom` constructor, indicating the flow of dependency injection.

```
static class Tire {  
    private int size;  
  
    public Tire(int size) {  
        this.size = size;  
        System.out.println("轮胎尺寸: " + size);  
    }  
}
```

IoC 是一种思想，也是"目标"，而思想只是一种指导原则，最终还是要要有可行的落地方案，而 DI 就属于具体的实现。所以也可以说，DI 是IoC的一种实现。

比如说我今天心情比较好，吃一顿好的犒劳犒劳自己，那么"吃一顿好的"是思想和目标（是 IoC），但最后我是吃海底捞还是杨国福？这就是具体的实现，就是 DI。

2. IoC & DI 使用

对IoC和DI有了初步的了解, 我们接下来具体学习Spring IoC和DI的代码实现.

依然是**先使用, 再学习**

既然 Spring 是一个 IoC（控制反转）容器, 作为容器, 那么它就具备两个最基础的功能:

- 存
- 取

Spring 容器 管理的主要是对象, 这些对象, 我们称之为"Bean". 我们把这些对象交由Spring管理, 由Spring来负责对象的创建和销毁. 我们程序只需要告诉Spring, 哪些需要存, 以及如何从Spring中取出对象

目标: 把BookDao, BookService 交给Spring管理, 完成Controller层, Service层, Dao层的解耦

步骤:

1. Service层及Dao层的实现类, 交给Spring管理: 使用注解: `@Component`
2. 在Controller层 和Service层 注入运行时依赖的对象: 使用注解 `@Autowired`

实现:

1. 把BookDao 交给Spring管理, 由Spring来管理对象

```
1 @Component
2 public class BookDao {
3     /**
4      * 数据Mock 获取图书信息
5      *
6      * @return
7      */
8     public List<BookInfo> mockData() {
9         List<BookInfo> books = new ArrayList<>();
10        for (int i = 0; i < 5; i++) {
11            BookInfo book = new BookInfo();
12            book.setId(i);
13            book.setBookName("书籍" + i);
14            book.setAuthor("作者" + i);
15            book.setCount(i * 5 + 3);
16            book.setPrice(new BigDecimal(new Random().nextInt(100)));
17            book.setPublish("出版社" + i);
18            book.setStatus(1);
19            books.add(book);
20        }
21        return books;
```

```
22     }  
23 }
```

2. 把BookService 交给Spring管理, 由Spring来管理对象

```
1 @Component  
2 public class BookService {  
3     private BookDao bookDao = new BookDao();  
4  
5     public List<BookInfo> getBookList() {  
6         List<BookInfo> books = bookDao.mockData();  
7         for (BookInfo book : books) {  
8             if (book.getStatus() == 1) {  
9                 book.setStatusCN("可借阅");  
10            } else {  
11                book.setStatusCN("不可借阅");  
12            }  
13        }  
14        return books;  
15    }  
16 }
```

3. 删除创建BookDao的代码, 从Spring中获取对象

```
1 @Component  
2 public class BookService {  
3     @Autowired  
4     private BookDao bookDao;  
5  
6     public List<BookInfo> getBookList() {  
7         List<BookInfo> books = bookDao.mockData();  
8         for (BookInfo book : books) {  
9             if (book.getStatus() == 1) {  
10                book.setStatusCN("可借阅");  
11            } else {  
12                book.setStatusCN("不可借阅");  
13            }  
14        }  
15        return books;  
16    }  
17 }
```

4. 删除创建BookService的代码, 从Spring中获取对象

```
1 @RequestMapping("/book")
2 @RestController
3 public class BookController {
4     @Autowired
5     private BookService bookService;
6
7     @RequestMapping("/getList")
8     public List<BookInfo> getList(){
9         //获取数据
10         List<BookInfo> books = bookService.getBookList();
11         return books;
12     }
13 }
```

5. 重新运行程序, http://127.0.0.1:8080/book_list.html

图书列表展示

添加图书

批量删除

| 选择 | 图书ID | 书名 | 作者 | 数量 | 定价 | 出版社 | 状态 | 操作 |
|--------------------------|------|-----|-----|----|----|------|-----|---------------------------------------|
| <input type="checkbox"/> | 0 | 书籍0 | 作者0 | 3 | 85 | 出版社0 | 可借阅 | 修改 删除 |
| <input type="checkbox"/> | 1 | 书籍1 | 作者1 | 8 | 77 | 出版社1 | 可借阅 | 修改 删除 |
| <input type="checkbox"/> | 2 | 书籍2 | 作者2 | 13 | 97 | 出版社2 | 可借阅 | 修改 删除 |
| <input type="checkbox"/> | 3 | 书籍3 | 作者3 | 18 | 6 | 出版社3 | 可借阅 | 修改 删除 |
| <input type="checkbox"/> | 4 | 书籍4 | 作者4 | 23 | 25 | 出版社4 | 可借阅 | 修改 删除 |

首页

上一页

1

2

3

4

5

下一页

最后一页

3. IoC 详解

通过上面的案例, 我们已经知道了Spring IoC 和DI的基本操作, 接下来我们来系统的学习Spring IoC和DI的操作.

前面我们提到IoC控制反转，就是将对象的控制权交给Spring的IOC容器，由IOC容器创建及管理对象。

也就是bean的存储。

3.1 Bean的存储

在之前的入门案例中，要把某个对象交给IOC容器管理，需要在类上添加一个注解：`@Component` 而Spring框架为了更好的服务web应用程序，提供了更丰富的注解。

共有两类注解类型可以实现：

1. 类注解：`@Controller`、`@Service`、`@Repository`、`@Component`、`@Configuration`。
2. 方法注解：`@Bean`。

接下来我们分别来看

3.1.1 @Controller（控制器存储）

使用 `@Controller` 存储 bean 的代码如下所示：

```
1 @Controller // 将对象存储到 Spring 中
2 public class UserController {
3     public void sayHi(){
4         System.out.println("hi,UserController...");
5     }
6 }
```

如何观察这个对象已经存在Spring容器当中了呢？

接下来我们学习如何从Spring容器中获取对象

```
1 @SpringBootApplication
2 public class SpringIocDemoApplication {
3
4     public static void main(String[] args) {
5         //获取Spring上下文对象
6         ApplicationContext context = SpringApplication.run(SpringIocDemoApplicatio
7         //从Spring上下文中获取对象
8         UserController userController = context.getBean(UserController.class);
9         //使用对象
10        userController.sayHi();
11    }
12 }
```

因为对象都交给 Spring 管理了，所以获取对象要从 Spring 中获取，那么就得先得到 Spring 的上下文

关于上下文的概念

上学时, 阅读理解经常会这样问: 根据上下文, 说一下你对XX的理解

在计算机领域,上下文这个概念,咱们最早是在学习线程时了解到过,比如我们应用进行线程切换的时候,切换前都会把线程的状态信息暂时储存起来,这里的上下文就包括了当前线程的信息,等下次该线程又得到CPU时间的时候,从上下文中拿到线程上次运行的信息

这个上下文,就是指当前的运行环境,也可以看作是一个容器,容器里存了很多内容,这些内容是当前运行的环境

观察运行结果,发现成功从Spring中获取到Controller对象,并执行Controller的sayHi方法

```

      .      _ _ _ _ _      _      _ _ _ _ _
/\ \ / _ _ ' _ _ _ _ _ ( _ ) _ _ _ _ _ \ \ \ \ \
( ( ) \ _ _ | ' _ | ' _ | | ' _ \ / _ ` | \ \ \ \ \
\ \ / _ _ _ ) | | _ ) | | | | | | | | ( _ | | ) ) ) )
' | _ _ _ | . _ | | | | _ | | | | _ \ _ , | / / / /
=====|_|=====| _ _ / = / _ / _ /
:: Spring Boot ::                                (v2.7.14)

```

```
hi,UserController...
```

如果把@Controller删掉,再观察运行结果

```
Exception in thread "main" org.springframework.beans.factory.NoSuchBeanDefinitionException: Create breakpoint : No qualifying bean of type 'com.example.demo.controller
.UserController' available
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBean(DefaultListableBeanFactory.java:351)
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBean(DefaultListableBeanFactory.java:342)
    at org.springframework.context.support.AbstractApplicationContext.getBean(AbstractApplicationContext.java:1175)
    at com.example.demo.SpringIoCDemoApplication.main(SpringIoCDemoApplication.java:17)
```

报错信息显示: 找不到类型是: com.example.demo.controller.UserController的bean

获取bean对象的其他方式

上述代码是根据类型来查找对象,如果Spring容器中,同一个类型存在多个bean的话,怎么来获取呢?

ApplicationContext 也提供了其他获取bean的方式, ApplicationContext 获取bean对象的功能, 是父类BeanFactory提供的功能.

```

1 public interface BeanFactory {
2
3     //以上省略...
4
5     // 1. 根据bean名称获取bean
6     Object getBean(String var1) throws BeansException;
7     // 2. 根据bean名称和类型获取bean
8     <T> T getBean(String var1, Class<T> var2) throws BeansException;
9     // 3. 按bean名称和构造函数参数动态创建bean,只适用于具有原型(prototype)作用域的bean
10    Object getBean(String var1, Object... var2) throws BeansException;
11    // 4. 根据类型获取bean
12    <T> T getBean(Class<T> var1) throws BeansException;
13    // 5. 按bean类型和构造函数参数动态创建bean,只适用于具有原型(prototype)作用域的bean
14    <T> T getBean(Class<T> var1, Object... var2) throws BeansException;
15
16    //以下省略...
17 }

```

常用的是上述1,2,4种,这三种方式,获取到的bean是一样的

其中1,2种都涉及到根据名称来获取对象. bean的名称是什么呢?

Spring bean是Spring框架在运行时管理的对象, Spring会给管理的对象起一个名字.

比如学校管理学生, 会给每个学生分配一个学号, 根据学号, 就可以找到对应的学生.

Spring也是如此, 给每个对象起一个名字, 根据Bean的名称(BeanId)就可以获取到对应的对象.

Bean 命名约定

我们看下官方文档的说明: [Bean Overview :: Spring Framework](#)

Bean Naming Conventions

The convention is to use the standard Java convention for instance field names when naming beans. That is, bean names start with a lowercase letter and are camel-cased from there. Examples of such names include `accountManager`, `accountService`, `userDao`, `loginController`, and so forth.

Naming beans consistently makes your configuration easier to read and understand. Also, if you use Spring AOP, it helps a lot when applying advice to a set of beans related by name.

Note

With component scanning in the classpath, Spring generates bean names for unnamed components, following the rules described earlier: essentially, taking the simple class name and turning its initial character to lower-case. However, in the (unusual) special case when there is more than one character and both the first and second characters are upper case, the original casing gets preserved. These are the same rules as defined by `java.beans.Introspector.decapitalize` (which Spring uses here).

程序开发人员不需要为bean指定名称(BeanId), 如果没有显式的提供名称(BeanId), Spring容器将为该bean生成唯一的名称.

命名约定使用Java标准约定作为实例字段名. 也就是说, bean名称以小写字母开头, 然后使用驼峰式大小写.

比如

类名: UserController, Bean的名称为: userController

类名: AccountManager, Bean的名称为: accountManager

类名: AccountService, Bean的名称为: accountService

也有一些特殊情况, 当有多个字符并且第一个和第二个字符都是大写时, 将保留原始的大小写. 这些规则与java.beans.Introspector.decapitalize (Spring在这里使用的)定义的规则相同.

比如

类名: UController, Bean的名称为: UController

类名: AManager, Bean的名称为: AManager

根据这个命名规则, 我们来获取Bean.

```
1 @SpringBootApplication
2 public class SpringIocDemoApplication {
3
4     public static void main(String[] args) {
5         //获取Spring上下文对象
6         ApplicationContext context = SpringApplication.run(SpringIocDemoApplicatio
7         //从Spring上下文中获取对象
8         //根据bean类型, 从Spring上下文中获取对象
9         UserController userController1 = context.getBean(UserController.class);
10        //根据bean名称, 从Spring上下文中获取对象
11        UserController userController2 = (UserController) context.getBean("userCon
12        //根据bean类型+名称, 从Spring上下文中获取对象
13        UserController userController3 = context.getBean("userController",UserCont
14
15        System.out.println(userController1);
16        System.out.println(userController2);
17        System.out.println(userController3);
18    }
19 }
```

运行结果:

```
com.example.demo.controller.UserController@5226e402
com.example.demo.controller.UserController@5226e402|
com.example.demo.controller.UserController@5226e402
```

地址一样, 说明对象是一个

获取bean对象, 是父类BeanFactory提供的功能

ApplicationContext VS BeanFactory (常见面试题)

- 继承关系和功能方面来说: Spring 容器有两个顶级的接口: BeanFactory 和 ApplicationContext。其中 BeanFactory 提供了基础的访问容器的能力, 而 ApplicationContext 属于 BeanFactory 的子类, 它除了继承了 BeanFactory 的所有功能之外, 它还拥有独特的特性, 还添加了对国际化支持、资源访问支持、以及事件传播等方面的支持。
- 从性能方面来说: ApplicationContext 是一次性加载并初始化所有的 Bean 对象, 而 BeanFactory 是需要那个才去加载那个, 因此更加轻量。(空间换时间)

3.1.2 @Service (服务存储)

使用 @Service 存储 bean 的代码如下所示:

```
1 @Service
2 public class UserService {
3     public void sayHi(String name) {
4         System.out.println("Hi," + name);
5     }
6 }
```

读取 bean 的代码:

```
1 @SpringBootApplication
2 public class SpringIocDemoApplication {
3
4     public static void main(String[] args) {
5         //获取Spring上下文对象
6         ApplicationContext context = SpringApplication.run(SpringIocDemoApplicatio
7         //从Spring中获取UserService对象
8         UserService userService = context.getBean(UserService.class);
9         //使用对象
10        userService.sayHi();
11    }
```

观察运行结果,发现成功从Spring中获取到UserService对象,并执行UserService的sayHi方法

```
( _ ) \_ _ _ | _ - | _ - V _ - | \ \ \ \ \
\\ / _ _ _ ) | | _ ) | | | | | | | ( _ | | ) ) ) )
' | _ _ _ | . _ _ | _ | _ | _ | _ \_ _ , | / / / /
=====|_|=====|___/=/_/_/_/
:: Spring Boot ::                (v2.7.14)
```

Hi, UserService

同样的, 把注解@Service删掉, 再观察运行结果

3.1.3 @Repository（仓库存储）

使用 `@Repository` 存储 bean 的代码如下所示:

```
1 @Repository
2 public class UserRepository {
3     public void sayHi() {
4         System.out.println("Hi, UserRepository~");
5     }
6 }
```

读取 bean 的代码:

```
1 @SpringBootApplication
2 public class SpringIocDemoApplication {
3
4     public static void main(String[] args) {
5         //获取Spring上下文对象
6         ApplicationContext context = SpringApplication.run(SpringIocDemoApplicatio
7         //从Spring上下文中获取对象
8         UserRepository userRepository = context.getBean(UserRepository.class);
9         //使用对象
10        userRepository.sayHi();
11    }
12 }
```

观察运行结果, 发现成功从Spring中获取到UserRepository 对象, 并执行UserRepository 的sayHi方法

```
( ( ) \_ _ _ | ' _ | ' | | ' _ \ / _ ` | \ \ \ \
\\ / _ _ _ ) | | _ | | | | | | | ( | | ) ) ) )
' | _ _ _ | . _ _ | | | _ | | \ _ _ , | / / / /
===== | _ | ===== | _ _ / = / _ / _ / _ /
:: Spring Boot ::                (v2.7.14)
```

Hi, UserRepository~

同样的, 把注解@Repository删掉, 再观察运行结果

3.1.4 @Component (组件存储)

使用 @Component 存储 bean 的代码如下所示:

```
1 @Component
2 public class UserComponent {
3     public void sayHi() {
4         System.out.println("Hi, UserComponent~");
5     }
6 }
```

读取 bean 的代码:

```
1 @SpringBootApplication
2 public class SpringIocDemoApplication {
3
4     public static void main(String[] args) {
5         //获取Spring上下文对象
6         ApplicationContext context = SpringApplication.run(SpringIocDemoApplicatio
7         //从Spring上下文中获取对象
8         UserComponent userComponent = context.getBean(UserComponent.class);
9         //使用对象
10        userComponent.sayHi();
11    }
12 }
```

观察运行结果, 发现成功从Spring中获取到UserComponent 对象, 并执行UserComponent 的sayHi方法

```
( ( ) \_ _ _ | ' _ | ' _ | | ' _ \ / _ ` | \ \ \ \
\ \ / _ _ _ ) | | _ ) | | | | | | | ( _ | | ) ) ) )
' | _ _ _ | . _ _ | | | _ | | _ \ _ , | / / / /
=====|_|=====|_ _ _ / = / _ / _ / _ /
:: Spring Boot ::                (v2.7.14)
```

```
Hi, UserComponent~
```

同样的, 把注解@Component删掉, 再观察运行结果

3.1.5 @Configuration (配置存储)

使用 @Configuration 存储 bean 的代码如下所示:

```
1 @Configuration
2 public class UserConfiguration {
3     public void sayHi() {
4         System.out.println("Hi,UserConfiguration~");
5     }
6 }
```

读取 bean 的代码:

```
1 @SpringBootApplication
2 public class SpringIocDemoApplication {
3
4     public static void main(String[] args) {
5         //获取Spring上下文对象
6         ApplicationContext context = SpringApplication.run(SpringIocDemoApplicatio
7         //从Spring上下文中获取对象
8         UserConfiguration userConfiguration = context.getBean(UserConfiguration.cl
9         //使用对象
10        userConfiguration.sayHi();
11    }
12 }
```

观察运行结果, 发现成功从Spring中获取到UserConfiguration对象, 并执行UserConfiguration的sayHi方法

```
( ( ) \_ _ _ | ' _ | ' _ | | ' _ \ / _ ' | \ \ \ \
\ \ / _ _ _ ) | | _ ) | | | | | | | ( _ | | ) ) ) )
' | _ _ _ | . _ _ | _ | | _ | | _ \ _ , | / / / /
=====|_|=====|_ _ _/=/_/_/_/_/
:: Spring Boot ::                (v2.7.14)
```

```
Hi,UserConfiguration~
```

同样的, 把注解@Configuration删掉, 再观察运行结果

3.2 为什么要这么多类注解?

这个也是和咱们前面讲的应用分层是呼应的. 让程序员看到类注解之后, 就能直接了解当前类的用途.

- @Controller: 控制层, 接收请求, 对请求进行处理, 并进行响应.
- @Service: 业务逻辑层, 处理具体的业务逻辑.
- @Repository: 数据访问层, 也称为持久层. 负责数据访问操作
- @Configuration: 配置层. 处理项目中的一些配置信息.

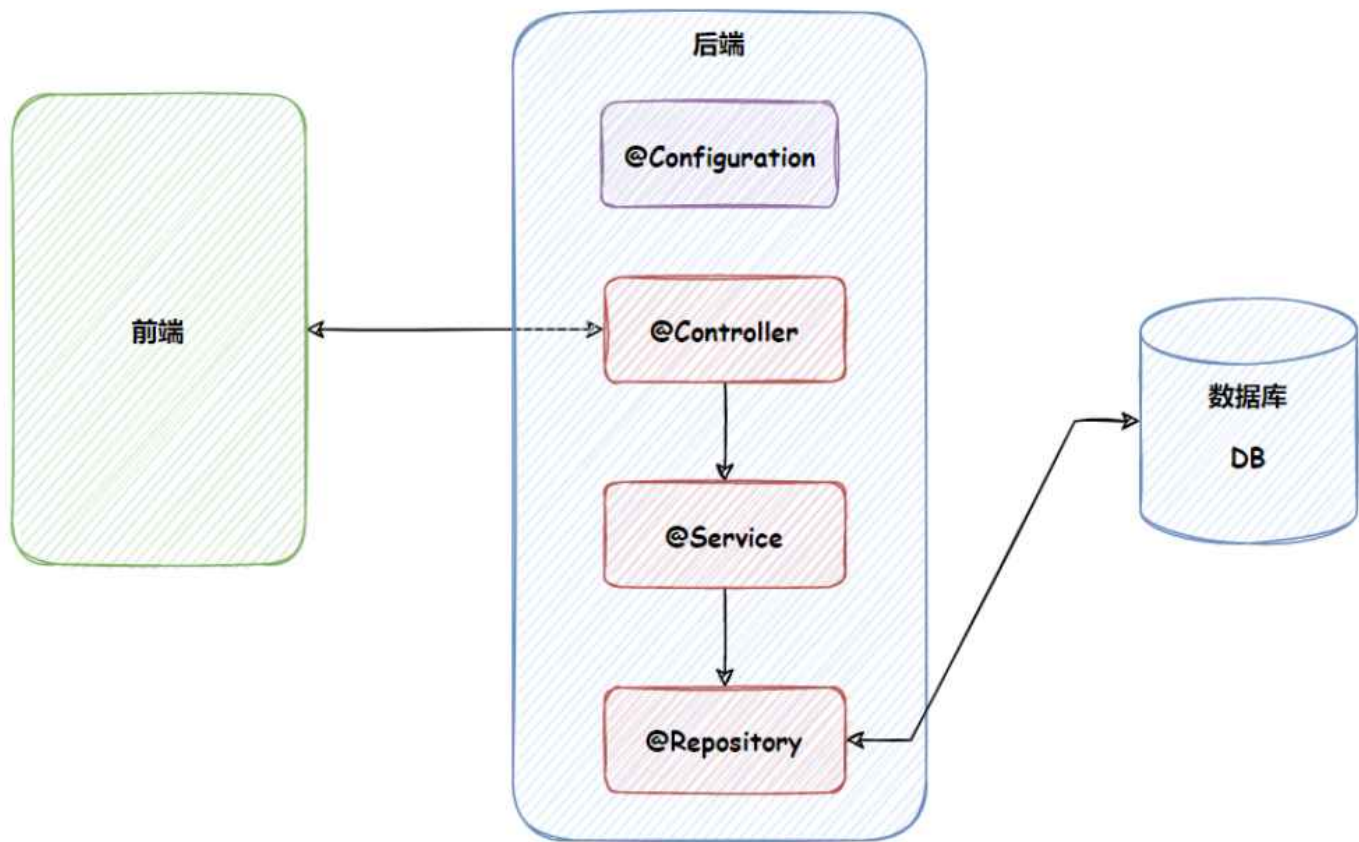
这和每个省/市都有自己的车牌号是一样的.

车牌号都是唯一的, 标识一个车辆的. 但是为什么还需要设置不同的车牌开头呢.

比如陕西的车牌号就是: 陕X: XXXXXX, 北京的车牌号: 京X: XXXXXX, 甚至一个省不同的县区也是不同的, 比如西安就是, 陕A: XXXXXX, 咸阳: 陕B: XXXXXX, 宝鸡, 陕C: XXXXXX, 一样.

这样做的好处除了可以节约号码之外, 更重要的作用是可以直观的标识一辆车的归属地.

程序的应用分层, 调用流程如下:



类注解之间的关系

查看 `@Controller` / `@Service` / `@Repository` / `@Configuration` 等注解的源码发现：

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Controller {
    @AliasFor(
        annotation = Component.class
    )
    String value() default "";
}
```

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Service {
    @AliasFor(
        annotation = Component.class
    )
    String value() default "";
}
```

其实这些注解里面都有一个注解 `@Component`，说明它们本身就是属于 `@Component` 的"子类".

`@Component` 是一个元注解，也就是说可以注解其他类注解，如 `@Controller`，`@Service`，`@Repository` 等. 这些注解被称为 `@Component` 的衍生注解.

`@Controller` , `@Service` 和 `@Repository` 用于更具体的用例(分别在控制层, 业务逻辑层, 持久化层), 在开发过程中, 如果你要在业务逻辑层使用 `@Component` 或 `@Service`, 显然 `@Service` 是更好的选择

比如杯子有喝水杯, 刷牙杯等, 但是我们更倾向于在日常喝水时使用水杯, 洗漱时使用刷牙杯.

更多资料参考:

<https://docs.spring.io/spring-framework/reference/core/beans/classpath-scanning.html#beans-stereotype-annotations>

3.3 方法注解 @Bean

类注解是添加到某个类上的, 但是存在两个问题:

1. 使用外部包里的类, 没办法添加类注解
2. 一个类, 需要多个对象, 比如多个数据源

这种场景, 我们就需要使用方法注解 `@Bean`

我们先来看看方法注解如何使用:

```
1 public class BeanConfig {
2     @Bean
3     public User user(){
4         User user = new User();
5         user.setName("zhangsan");
6         user.setAge(18);
7         return user;
8     }
9 }
```

然而, 当我们写完以上代码, 尝试获取 bean 对象中的 user 时却发现, 根本获取不到:

```
1 @SpringBootApplication
2 public class SpringIocDemoApplication {
3
4     public static void main(String[] args) {
5         //获取Spring上下文对象
6         ApplicationContext context = SpringApplication.run(SpringIocDemoApplicatio
7         //从Spring上下文中获取对象
8         User user = context.getBean(User.class);
9         //使用对象
10        System.out.println(user);
11    }
12 }
```



```
11     }
12 }
```

以上程序的执行结果如下：

```
Exception in thread "main" org.springframework.beans.factory.NoSuchBeanDefinitionException: Create breakpoint: No qualifying bean of type 'com.example.demo.model.User'
available
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBean(DefaultListableBeanFactory.java:351)
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBean(DefaultListableBeanFactory.java:342)
    at org.springframework.context.support.AbstractApplicationContext.getBean(AbstractApplicationContext.java:1175)
    at com.example.demo.SpringIocDemoApplication.main(SpringIocDemoApplication.java:21)
```

这是为什么呢？

3.3.1 方法注解要配合类注解使用

在 Spring 框架的设计中，方法注解 `@Bean` 要配合类注解才能将对象正常的存储到 Spring 容器中，如下代码所示：

```
1 @Component
2 public class BeanConfig {
3     @Bean
4     public User user(){
5         User user = new User();
6         user.setName("zhangsan");
7         user.setAge(18);
8         return user;
9     }
10 }
```

再次执行以上代码，运行结果如下：

```
"D:\Program Files\Java\jdk1.8.0_261\bin\java.exe" ...
```

```
.   ____          _            _ _
/\ /  ___'  _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
( ( )\___ | '_ | '_ | | '_ \ / _ `| \ \ \ \
\\ /  ___)| |_)| | | | |___| | ) ) ) )
'  |____| .__| | | | |___| | \ \ \ \
=====|_|=====|___/=/_/_/_/_/
:: Spring Boot ::                (v2.7.14)
```

```
User(name=zhangsan, age=18)
```

3.3.2 定义多个对象

对于同一个类, 如何定义多个对象呢?

比如多数据源的场景, 类是同一个, 但是配置不同, 指向不同的数据源.

我们看下@Bean的使用

```
1 @Component
2 public class BeanConfig {
3     @Bean
4     public User user1(){
5         User user = new User();
6         user.setName("zhangsan");
7         user.setAge(18);
8         return user;
9     }
10
11     @Bean
12     public User user2(){
13         User user = new User();
14         user.setName("lisi");
15         user.setAge(19);
16         return user;
17     }
18 }
```

定义了多个对象的话, 我们根据类型获取对象, 获取的是哪个对象呢?

```
1 @SpringBootApplication
2 public class SpringIocDemoApplication {
3
4     public static void main(String[] args) {
5         //获取Spring上下文对象
6         ApplicationContext context = SpringApplication.run(SpringIocDemoApplicatio
7         //从Spring上下文中获取对象
8         User user = context.getBean(User.class);
9         //使用对象
10        System.out.println(user);
11    }
12 }
```

运行结果:

```
Exception in thread "main" org.springframework.beans.factory.NoUniqueBeanDefinitionException: Create breakpoint : No qualifying bean of type 'com.example.demo.model.User'
available: expected single matching bean but found 2: user1,user2
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.resolveNamedBean(DefaultListableBeanFactory.java:1273)
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.resolveBean(DefaultListableBeanFactory.java:494)
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBean(DefaultListableBeanFactory.java:349)
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBean(DefaultListableBeanFactory.java:342)
    at org.springframework.context.support.AbstractApplicationContext.getBean(AbstractApplicationContext.java:1175)
    at com.example.demo.SpringIocDemoApplication.main(SpringIocDemoApplication.java:21)
```

报错信息显示: 期望只有一个匹配, 结果发现了两个, user1, user2

从报错信息中, 可以看出来, @Bean 注解的bean, bean的名称就是它的方法名

接下来我们根据名称来获取bean对象

```
1 @SpringBootApplication
2 public class SpringIocDemoApplication {
3
4     public static void main(String[] args) {
5         //获取Spring上下文对象
6         ApplicationContext context = SpringApplication.run(SpringIocDemoApplicatio
7         //根据bean名称, 从Spring上下文中获取对象
8         User user1 = (User) context.getBean("user1");
9         User user2 = (User) context.getBean("user2");
10        System.out.println(user1);
11        System.out.println(user2);
12    }
13 }
```

运行结果:

User(name=zhangsan, age=18)

User(name=lisi, age=19)

可以看到, @Bean 可以针对同一个类, 定义多个对象.

3.3.3 重命名 Bean

可以通过设置 name 属性给 Bean 对象进行重命名操作, 如下代码所示:

```
1 @Bean(name = {"u1","user1"})
2 public User user1(){
3     User user = new User();
4     user.setName("zhangsan");
5     user.setAge(18);
}
```

```
6     return user;
7 }
```

此时我们使用 u1 就可以获取到 User 对象了，如下代码所示：

```
1 @SpringBootApplication
2 public class SpringIocDemoApplication {
3
4     public static void main(String[] args) {
5         //获取Spring上下文对象
6         ApplicationContext context = SpringApplication.run(SpringIocDemoApplicatio
7         //从Spring上下文中获取对象
8         User u1 = (User) context.getBean("u1");
9         //使用对象
10        System.out.println(u1);
11    }
12 }
```

name={} 可以省略，如下代码所示：

```
1 @Bean({"u1","user1"})
2 public User user1(){
3     User user = new User();
4     user.setName("zhangsan");
5     user.setAge(18);
6     return user;
7 }
```

只有一个名称时, {}也可以省略, 如:

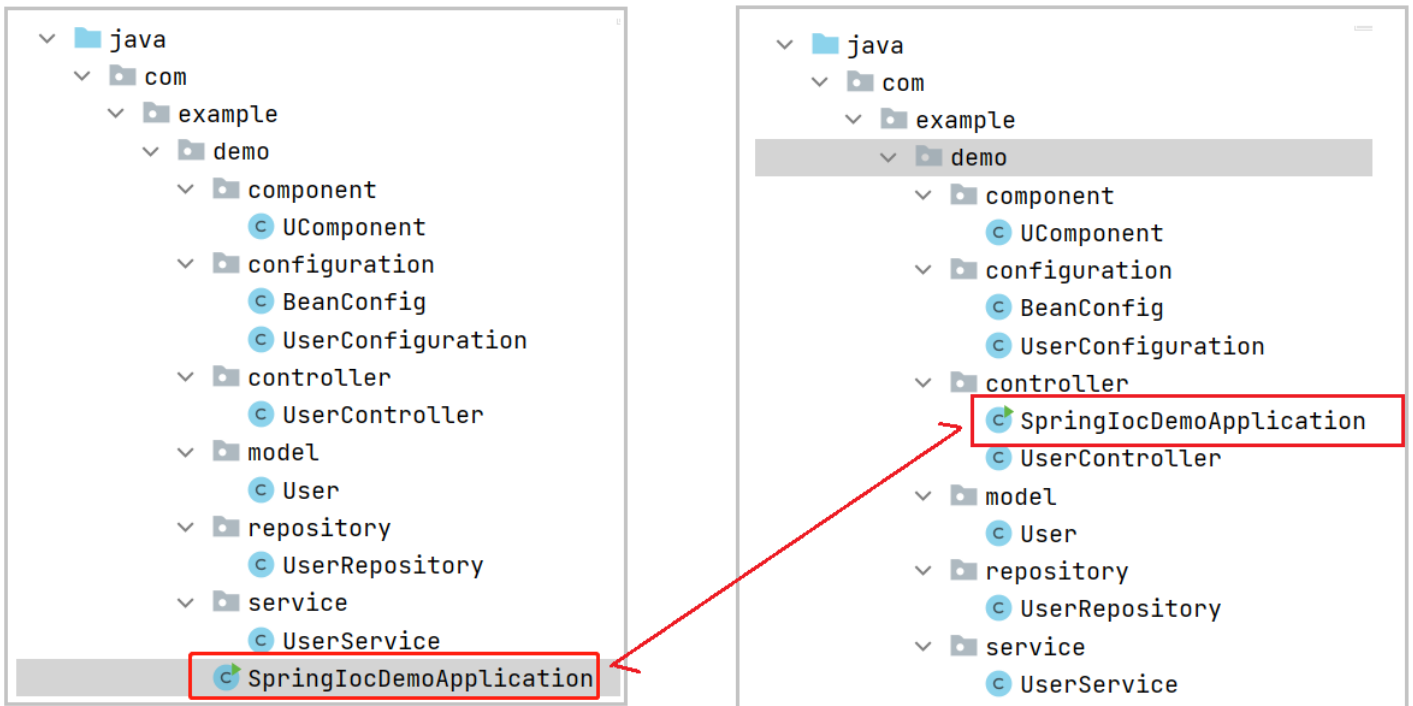
```
1 @Bean("u1")
2 public User user1(){
3     User user = new User();
4     user.setName("zhangsan");
5     user.setAge(18);
6     return user;
7 }
```

3.4 扫描路径

Q: 使用前面学习的四个注解声明的bean，一定会生效吗？

A: 不一定（原因：bean想要生效，还需要被Spring扫描）

下面我们通过修改项目工程的目录结构，来测试bean对象是否生效：



再运行代码：

```
1 @SpringBootApplication
2 public class SpringIocDemoApplication {
3
4     public static void main(String[] args) {
5         //获取Spring上下文对象
6         ApplicationContext context = SpringApplication.run(SpringIocDemoApplicatio
7         //从Spring上下文中获取对象
8         User u1 = (User) context.getBean("u1");
9         //使用对象
10        System.out.println(u1);
11    }
12 }
```

运行结果：

```
Exception in thread "main" org.springframework.beans.factory.NoSuchBeanDefinitionException: No bean named 'u1' available
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBeanDefinition(DefaultListableBeanFactory.java:874)
    at org.springframework.beans.factory.support.AbstractBeanFactory.getMergedLocalBeanDefinition(AbstractBeanFactory.java:1358)
    at org.springframework.beans.factory.support.AbstractBeanFactory.doGetBean(AbstractBeanFactory.java:309)
    at org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:208)
    at org.springframework.context.support.AbstractApplicationContext.getBean(AbstractApplicationContext.java:1157)
    at com.example.demo.controller.SpringIocDemoApplication.main(SpringIocDemoApplication.java:15)
```

解释: 没有bean的名称为u1

为什么没有找到bean对象呢?

使用五大注解声明的bean, 要想生效, 还需要配置扫描路径, 让Spring扫描到这些注解

也就是通过 `@ComponentScan` 来配置扫描路径.

```
1 @ComponentScan({"com.example.demo"})
2 @SpringBootApplication
3 public class SpringIocDemoApplication {
4
5     public static void main(String[] args) {
6         //获取Spring上下文对象
7         ApplicationContext context = SpringApplication.run(SpringIocDemoApplicatio
8         //从Spring上下文中获取对象
9         User u1 = (User) context.getBean("u1");
10        //使用对象
11        System.out.println(u1);
12    }
13 }
```

`{}` 里可以配置多个包路径

这种做法仅做了解, 不做推荐使用

那为什么前面没有配置 `@ComponentScan` 注解也可以呢?

`@ComponentScan` 注解虽然没有显式配置, 但是实际上已经包含在了启动类声明注解 `@SpringBootApplication` 中了

默认扫描的范围是SpringBoot启动类所在包及其子包

在配置类上添加 `@ComponentScan` 注解, 该注解默认会扫描该类所在的包下所有的配置类

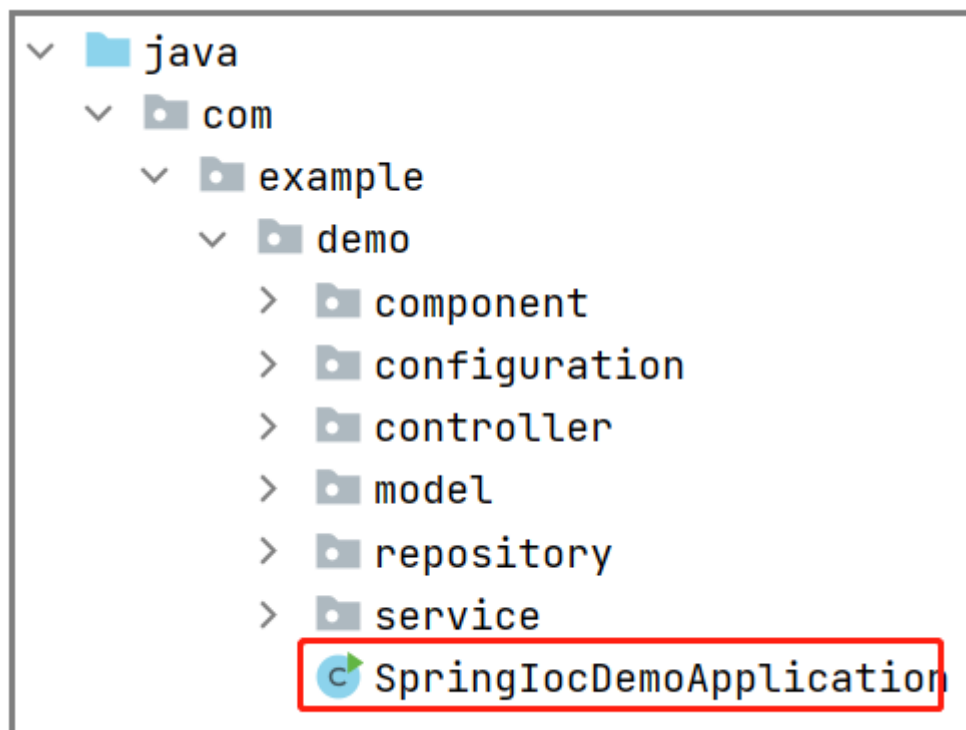
```

@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(
    excludeFilters = {@Filter(
        type = FilterType.CUSTOM,
        classes = {TypeExcludeFilter.class}
    )}, @Filter(
        type = FilterType.CUSTOM,
        classes = {AutoConfigurationExcludeFilter.class}
    )}
)
public @interface SpringBootApplication {
    @AliasFor(
        annotation = EnableAutoConfiguration.class
    )
    Class<?>[] exclude() default {};
}

```

推荐做法:

把启动类放在我们希望扫描的包的路径下, 这样我们定义的bean就都可以被扫描到



4. DI 详解

上面我们讲解了控制反转IoC的细节, 接下来呢, 我们学习依赖注入DI的细节。

依赖注入是一个过程，是指IoC容器在创建Bean时，去提供运行时所依赖的资源，而资源指的就是对象。

在上面程序案例中，我们使用了 `@Autowired` 这个注解，完成了依赖注入的操作。

简单来说，就是把对象取出来放到某个类的属性中。

在一些文章中，依赖注入也被称之为 "对象注入", "属性装配", 具体含义需要结合文章的上下文来理解

关于依赖注入, Spring也给我们提供了三种方式:

1. 属性注入(Field Injection)
2. 构造方法注入(Constructor Injection)
3. Setter 注入(Setter Injection)

接下来，我们分别来看。

下面我们按照实际开发中的模式，将 Service 类注入到 Controller 类中。

4.1 属性注入

属性注入是使用 `@Autowired` 实现的，将 Service 类注入到 Controller 类中。

Service 类的实现代码如下：

```
1 import org.springframework.stereotype.Service;
2
3 @Service
4 public class UserService {
5     public void sayHi() {
6         System.out.println("Hi,UserService");
7     }
8 }
```

Controller 类的实现代码如下：

```
1 @Controller
2 public class UserController {
3     //注入方法1: 属性注入
4     @Autowired
5     private UserService userService;
6
7     public void sayHi(){
8         System.out.println("hi,UserController...");
9         userService.sayHi();
10    }
```



```
10     }
11 }
```

获取 Controller 中的 sayHi方法：

```
1 @SpringBootApplication
2 public class SpringIocDemoApplication {
3
4     public static void main(String[] args) {
5         //获取Spring上下文对象
6         ApplicationContext context = SpringApplication.run(SpringIocDemoApplicatio
7         //从Spring上下文中获取对象
8         UserController userController = (UserController) context.getBean("userCont
9         //使用对象
10        userController.sayHi();
11    }
12 }
```

最终结果如下：

```
( ( ) \_ _ _ | ' _ | ' _ | | ' _ \ / _ | \ \ \ \
\\ / _ _ _ ) | | _ | | | | | | | ( _ | | ) ) ) )
' | _ _ _ | . _ _ | | | | | | | _ \ _ , | / / / /
=====|_|=====|_ _ _ / = / _ / _ /
:: Spring Boot ::                (v2.7.14)
```

```
hi,UserController...
Hi,UserService
```

去掉@Autowired , 再运行一下程序看看结果

```
hi,UserController...
```

```
Exception in thread "main" java.lang.NullPointerException Create breakpoint |
    at com.example.demo.controller.UserController.sayHi(UserController.java:15)
    at com.example.demo.SpringIocDemoApplication.main(SpringIocDemoApplication.java:18)
```

4.2 构造方法注入

构造方法注入是在类的构造方法中实现注入，如下代码所示：

```

1 @Controller
2 public class UserController2 {
3     //注入方法2: 构造方法
4     private UserService userService;
5
6     @Autowired
7     public UserController2(UserService userService) {
8         this.userService = userService;
9     }
10
11     public void sayHi(){
12         System.out.println("hi,UserController2...");
13         userService.sayHi();
14     }
15 }

```

注意事项：如果类只有一个构造方法，那么 `@Autowired` 注解可以省略；如果类中有多个构造方法，那么需要添加上 `@Autowired` 来明确指定到底使用哪个构造方法。

4.3 Setter 注入

Setter 注入和属性的 Setter 方法实现类似，只不过在设置 set 方法的时候需要加上 `@Autowired` 注解，如下代码所示：

```

1 @Controller
2 public class UserController3 {
3     //注入方法3: Setter方法注入
4     private UserService userService;
5
6     @Autowired
7     public void setUserService(UserService userService) {
8         this.userService = userService;
9     }
10
11     public void sayHi(){
12         System.out.println("hi,UserController3...");
13         userService.sayHi();
14     }
15 }

```

练习一下：尝试一下 set 方法如果不加 `@Autowired` 注解能注入成功吗？

4.4 三种注入优缺点分析

- 属性注入
 - 优点: 简洁, 使用方便;
 - 缺点:
 - 只能用于 IoC 容器, 如果是非 IoC 容器不可用, 并且只有在使用的时候才会出现 NPE (空指针异常)
 - 不能注入一个Final修饰的属性
- 构造函数注入(Spring 4.X推荐)
 - 优点:
 - 可以注入final修饰的属性
 - 注入的对象不会被修改
 - 依赖对象在使用前一定会被完全初始化, 因为依赖是在类的构造方法中执行的, 而构造方法是在类加载阶段就会执行的方法.
 - 通用性好, 构造方法是JDK支持的, 所以更换任何框架, 他都是适用的
 - 缺点:
 - 注入多个对象时, 代码会比较繁琐
- Setter注入(Spring 3.X推荐)
 - 优点: 方便在类实例之后, 重新对该对象进行配置或者注入
 - 缺点:
 - 不能注入一个Final修饰的属性
 - 注入对象可能会被改变, 因为setter方法可能会被多次调用, 就有被修改的风险.

更多DI相关内容参考: [Dependencies :: Spring Framework](#)

4.5 @Autowired存在问题

当同一类型存在多个bean时, 使用@Autowired会存在问题

```
1 @Component
2 public class BeanConfig {
3
4     @Bean("u1")
5     public User user1(){
6         User user = new User();
7         user.setName("zhangsan");
8         user.setAge(18);
9     }
10 }
```

```

9         return user;
10    }
11
12    @Bean
13    public User user2() {
14        User user = new User();
15        user.setName("lisi");
16        user.setAge(19);
17        return user;
18    }
19 }

```

```

1 @Controller
2 public class UserController {
3
4     @Autowired
5     private UserService userService;
6     //注入user
7     @Autowired
8     private User user;
9
10    public void sayHi(){
11        System.out.println("hi,UserController...");
12        userService.sayHi();
13        System.out.println(user);
14    }
15 }

```

运行结果:

APPLICATION FAILED TO START

Description:

Field user in com.example.demo.controller.UserController required a single bean, but 2 were found:

- **u1:** defined by method 'user1' in class path resource [com/example/demo/configuration/BeanConfig.class]
- **user2:** defined by method 'user2' in class path resource [com/example/demo/configuration/BeanConfig.class]

报错的原因是，非唯一的 Bean 对象。

如何解决上述问题呢？Spring提供了以下几种解决方案：

- @Primary

- @Qualifier
- @Resource

使用@Primary注解：当存在多个相同类型的Bean注入时，加上@Primary注解，来确定默认的实现。

```
1 @Component
2 public class BeanConfig {
3
4     @Primary //指定该bean为默认bean的实现
5     @Bean("u1")
6     public User user1(){
7         User user = new User();
8         user.setName("zhangsan");
9         user.setAge(18);
10        return user;
11    }
12
13    @Bean
14    public User user2() {
15        User user = new User();
16        user.setName("lisi");
17        user.setAge(19);
18        return user;
19    }
20 }
```

使用@Qualifier注解：指定当前要注入的bean对象。在@Qualifier的value属性中，指定注入的bean的名称。

- @Qualifier注解不能单独使用，必须配合@Autowired使用

```
1 @Controller
2 public class UserController {
3     @Qualifier("user2") //指定bean名称
4     @Autowired
5     private User user;
6
7     public void sayHi(){
8         System.out.println("hi,UserController...");
9         System.out.println(user);
10    }
11 }
```

使用@Resource注解：是按照bean的名称进行注入。通过name属性指定要注入的bean的名称。

```
1 @Controller
2 public class UserController {
3     @Resource(name = "user2")
4     private User user;
5
6     public void sayHi(){
7         System.out.println("hi,UserController...");
8         System.out.println(user);
9     }
10 }
```

常见面试题:

@Autowired 与 @Resource的区别

- @Autowired 是spring框架提供的注解，而@Resource是JDK提供的注解
- @Autowired 默认是按照类型注入，而@Resource是按照名称注入. 相比于 @Autowired 来说，@Resource 支持更多的参数设置，例如 name 设置，根据名称获取 Bean。

5. 练习

通过上面的学习, 我们把前面的图书管理系统代码进行调整

Service层的注解, 改成 @Service

Dao层的注解, 改成 @Repository

重新运行代码, 验证程序访问正常

图书列表展示

书名: 作者:

| 选择 | 图书ID | 书名 | 作者 | 数量 | 定价 | 出版社 | 状态 | 操作 |
|--------------------------|------|-----|-----|----|----|------|-----|---------------------------------------|
| <input type="checkbox"/> | 0 | 书籍0 | 作者0 | 3 | 55 | 出版社0 | 可借阅 | 修改 删除 |
| <input type="checkbox"/> | 1 | 书籍1 | 作者1 | 8 | 39 | 出版社1 | 可借阅 | 修改 删除 |
| <input type="checkbox"/> | 2 | 书籍2 | 作者2 | 13 | 22 | 出版社2 | 可借阅 | 修改 删除 |
| <input type="checkbox"/> | 3 | 书籍3 | 作者3 | 18 | 31 | 出版社3 | 可借阅 | 修改 删除 |
| <input type="checkbox"/> | 4 | 书籍4 | 作者4 | 23 | 96 | 出版社4 | 可借阅 | 修改 删除 |

[首页](#) [上一页](#) [1](#) [2](#) [3](#) [4](#) [5](#) [下一页](#) [最后一页](#)

6. 总结

6.1 Spring, Spring Boot 和Spring MVC的关系以及区别

Spring: 简单来说, Spring 是一个开发应用框架, 什么样的框架呢, 有这么几个标签: 轻量级、一站式、模块化, 其目的是用于简化企业级应用程序开发.

Spring的主要功能: 管理对象, 以及对象之间的依赖关系, 面向切面编程, 数据库事务管理, 数据访问, web框架支持等.

但是Spring具备高度可开放性, 并不强制依赖Spring, 开发者可以自由选择Spring的部分或者全部, Spring可以无缝继承第三方框架, 比如数据访问框架(Hibernate、JPA), web框架(如Struts、JSF)

Spring MVC: Spring MVC是Spring的一个子框架, Spring诞生之后, 大家觉得很好用, 于是按照MVC模式设计了一个 MVC框架(一些用Spring 解耦的组件), 主要用于开发WEB应用和网络接口, 所以, Spring MVC 是一个Web框架.

Spring MVC基于Spring进行开发的, 天生的与Spring框架集成. 可以让我们更简洁的进行Web层开发, 支持灵活的 URL 到页面控制器的映射, 提供了强大的约定大于配置的契约式编程支持, 非常容易与其他视图框架集成, 如 Velocity、FreeMarker等

Spring Boot: Spring Boot是对Spring的一个封装, 为了简化Spring应用的开发而出现的, 中小型企业, 没有成本研究自己的框架, 使用Spring Boot 可以更加快速的搭建框架, 降级开发成本, 让开发人员更加专注于Spring应用的开发, 而无需过多关注XML的配置和一些底层的实现.

Spring Boot 是个脚手架, 插拔式搭建项目, 可以快速的集成其他框架进来.

比如想使用SpringBoot开发Web项目, 只需要引入Spring MVC框架即可, Web开发的工作是SpringMVC完成的, 而不是SpringBoot, 想完成数据访问, 只需要引入Mybatis框架即可.

Spring Boot只是辅助简化项目开发的, 让开发变得更加简单, 甚至不需要额外的web服务器, 直接生成jar包执行即可.

最后一句话总结: **Spring MVC和Spring Boot都属于Spring, Spring MVC 是基于Spring的一个MVC 框架, 而Spring Boot 是基于Spring的一套快速开发整合包.**

比如我们的图书系统代码中

整体框架是通过SpringBoot搭建的

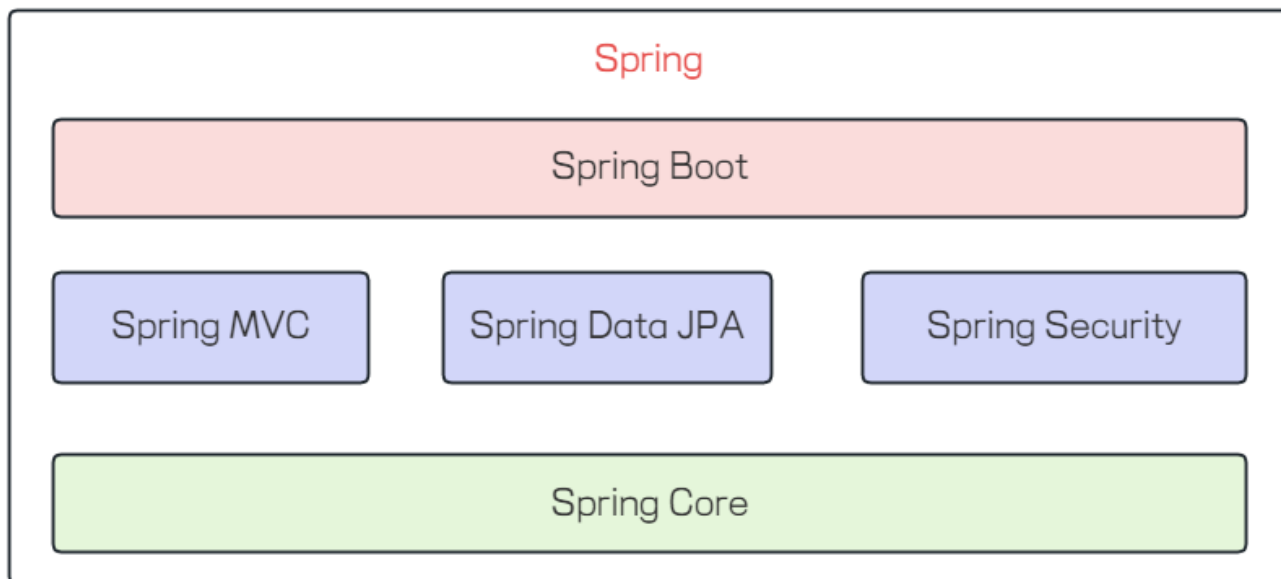
IoC, DI功能是Spring的提供的,

web相关功能是Spring MVC提供的

```
@RequestMapping("/book") Spring MVC
@RestController
public class BookController {
    @Autowired
    private BookService bookService; Spring

    @RequestMapping("/getList")
    public List<BookInfo> getList() {
        // 获取数据
        List<BookInfo> books = bookService.getBookList();
        return books;
    }
}
```

这三者专注的领域不同, 解决的问题也不一样, 总的来说, Spring 就像一个大家族, 有众多衍生产品, 但他们的基础都是Spring, 用一张图来表示他们三个的关系:



6.2 bean 的命名

1) 五大注解存储的bean

- ① 前两位字母均为大写, bean名称为类名
- ② 其他的为类名首字母小写
- ③ 通过 value属性设置 `@Controller(value = "user")`

2) @Bean 注解存储的bean

- ① bean名称为方法名
- ② 通过name属性设置 `@Bean(name = {"u1", "user1"})`

6.3 常见面试题

1) 三种注入方式的优缺点(参考上面课件内容)

2) 常见注解有哪些? 分别是什么作用?

web url映射: `@RequestMapping`

参数接收和接口响应: `@RequestParam`, `@RequestBody`, `@ResponseBody`

bean的存储: `@Controller`, `@Service`, `@Repository`, `@Component`, `@Configuration`, `@Bean`

bean的获取: `@Autowired`, `@Qualifier`, `@Resource`

3) @Autowired 和@Resource 区别

4) 说下你对Spring, SpringMVC, Springboot的理解(参考上面课件)

