IOC

一，IoC的理解

- IOC就是控制反转，是指创建的对象控制权的转移，以前创建对象的主动权和时机是由自己把控的，而现在这种权力转移到了Spring容器中，并由容器根据配置文件去创建实例和管理各个实例之间的依赖关系。
- DI：依赖注入，是指应用程序在运行是依赖IOC容器来动态注入需要的外部资源。

二，Bean的装配

- 自动化装配bean
  - 声明一个@Component->组件自动扫描(@ComponentScan/基于XML：<context:compontent-scan base-pase-package="包名">)
- 通过java代码装配bean
  - 通过@Bean注解来声明一个bean->通过一个@Configuration创建一个配置类->通过@ContexConfiguration来引入配置类中配置的信息。
- 通过XML来装配bean
  - <bean id=""class="类路径">
  - 如果要注入bean的话，通过标签
- DI
  - 如果要注入bean的话，通过标签
  - 注入字段值
    - 通过构造器

```
public class Test{
    String name;
    String password;
    public Test(String name,String password){
        this.name=name;
        this.password=password;
    }
}
<!-- 通过constructor标签来注入属性 -->
<bean id="" class="">
<construcor-arg value="Tom"/>
<construcor-arg value="123"/>
</bean>
<!-- 通过c命名空间 -->
<bean id="" class="" c:_name="Tom" c:_password="123"/>
或者
<bean id="" class="" c:_0="Tom" c:_1="123"/>

<!--装配集合-->
//假如说构造函数中存在集合类
<bean id="" class="">
<constructor-arg>
```

```
    <list>
        <value>a</value>
        <value>b</value>
        <value>c</value>
    </list>
</constructor-arg>
</bean>
```

- 通过Setter方法

```
public class Test{
    String name;
    String password;

    public void setName(String name){
        this.name = name;
    }
    public void setPassword(String password){
        this.password = password;
    }
}
<!-- 通过property标签 -->
<bean id="" class="">
    <property name="name" value="Tom"></property>
    <property name="password" value="123"></perproty>
</bean>

//如果类中含有List等类属性要初始化的时候，则：
<property name="list数组名(setter参数中)">
    <list>
        <value>1</value>
        <value>2</value>
        <value>3<value>
    </list>
</property>

<!-- 通过p命名空间 -->
<bean id="" class="" p:name="Tom" p:password="123"/>
```

- 导入和混合配置
  - 在javaconfig类中，引用其他配置类的配置信息

    @Import(Otherconfig.class),如果要引入多个类的配置信息则：
    @Import(Otherconfig1.class,Otherconfig2.class)
  - 在配置类中引入XML配置信息

    @ImportResource("classpath:Bean.xml");
  - 在XML中引入config类的配置信息

```
<bean class="配置类的路径"/>
```

- 在XML中引入其他XML信息

```
<import resource="XML的路径"/>
```

三，Bean的高级装配

- 条件化的bean
  - 如果想要一个Bean在另一个Bean初始化之后去加载，则使用@Conditional(other.class);
- 处理自动装配的歧义性
  - 在@Compotent标签下加上@Primary即可

```
@Autowired
@Qualifier("iceCream")    //第二种方式，参数是Bean的ID
public void setDessert(Dessert dessert){
    this.dessert = dessert;
}
@Compotent
@Primary     //第一种方式
public class Cake implements Dessert{

}
@Compotent
public class Cookies implements Dessert{

}
@Compotent
public class IceCream implements Dessert{

}
```

- Bean的作用域
  - 单例(Singleton)：在整个应用中，只创建bean的一个实例。(默认)
  - 原型(Property)：每次注入或者通过Spring应用上下文获取的时候，都会创建一个新的bean实例。
  - 会话(Session)：在web应用中，为每个会话创建一个bean实例。
  - 请求(Request)：在web应用中，为每个请求创建一个bean实例。
  - 设置：

    通过XML配置：

    ```
    <bean id="" class="" scope="Prototype">
    ```

    通过注解配置

    ```
    @Component
    @Scope(ConfigurableBeanFactory.SCOPE_PROPERTY)
    ```

- 运行时注入

引入
```
@PropertySource("classpath/app.properties")   //声明属性源

app.properties中的内容:
disc.title=A
disc.artist=B

类中
{
    @Autiwired
    Enviroment env;

    获取值:
    env.getProperty("disc.title");
    env.getProperty("disc.artist");
}
```