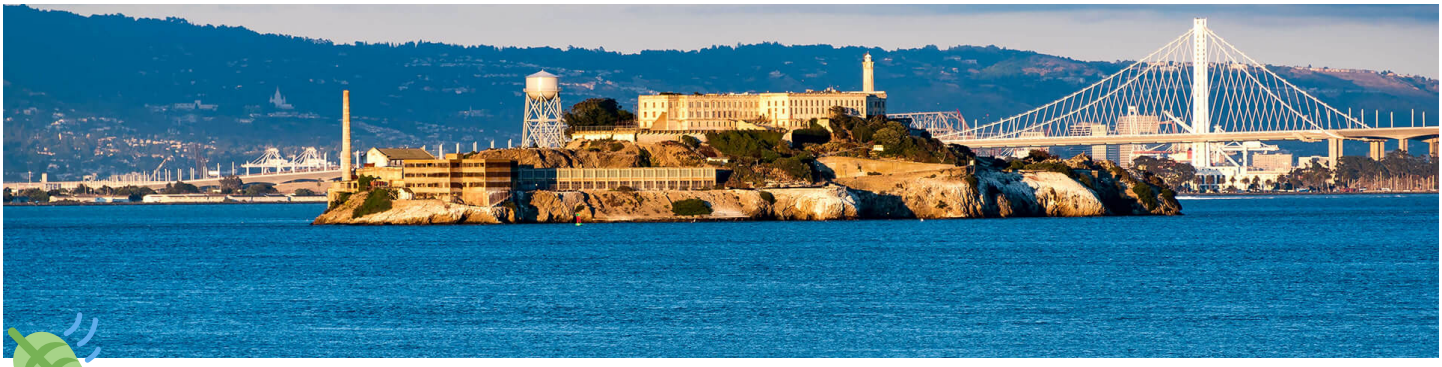

Spring框架

Spring框架的核心是ioc与aop，即控制反转和面向切面。

Spring Ioc 依赖注入

Spring中可以通过@Component 等注解、applicationContext.xml文件配置等方式指定哪些类会在程序启动时被实例化为bean对象，这些bean对象会被存放在ioc容器（一个map）中，因此我们使用这些类的对象时，不需要再使用new等方式创建。

IOC 介绍与搭建

在java代码中，类之间互相调用方式为调用者在自身内部new一个被调用者对象。使用Spring之后，其思想是被调用者主动将自身对象交给调用者，即依赖注入（DI），此过程中主动方发生反转，即控制反转（IOC）。

传统对象调用

Java

```java
// 被调用者类
public class Product {
    public String name = "被调用者";
}

// 调用者
public class Consumer {
    public Product product = new Product(); // new一个对象
}
```

使用IOC调用

Java

```java
// 被调用者类
@Component
public class Product {
    public String name = "被调用者";
}

// 调用者
@Component
public class Consumer {
    @Autowired
    public Product product;
}
```

想使用IOC我们需要先使用Spring的jar包，以提供SpringIOC支持。

原理：程序启动时，将指定的类实例化为bean并注册到IOC容器中，并在bean初始化时利用反射将对象赋值给调用者bean。

Spring的maven依赖如下（根据需要引入）：

```xml
<dependencies>
    <!-- core: spring核心工具类-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>4.3.7.RELEASE</version>
    </dependency>
    <!-- beans: 核心容器，基本的ioc注入-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-beans</artifactId>
        <version>4.3.7.RELEASE</version>
    </dependency>
    <!-- context: 扩展,JNDI、校验 | context-support: 缓存、邮件、任务等支持-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>4.3.7.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context-support</artifactId>
        <version>4.3.7.RELEASE</version>
    </dependency>
    <!-- tx事务: 声明式、编程式事务管理-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-tx</artifactId>
        <version>4.3.7.RELEASE</version>
    </dependency>
    <!-- expression：一些表达式支持-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-expression</artifactId>
        <version>4.3.7.RELEASE</version>
    </dependency>
</dependencies>
```

Spring配置：在resources目录下创建applicationContext.xml文件，bean可以在这里配置，也可以用注解指定，编译后resources和类在同一个目录下。

```xml
<!-- beans|tx|aop|context 按需指定，beans必须有 -->
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://www.springframework.org/schema/beans"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
            http://www.springframework.org/schema/beans   http://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/tx      http://www.springframework.org/schema/tx/spring-tx.xsd
            http://www.springframework.org/schema/aop     http://www.springframework.org/schema/aop/spring-aop.xsd
            http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd
       ">

    <!-- 扫描com.test.service包下所有能注册bean的注解 -->
    <context:component-scan base-package="com.test.service">
        <!-- 被@Controller注解的类被排除 -->
        <context:exclude-filter type="annotation" expression="org.springframework.stereotype.Controller"/>
    </context:component-scan>

    <!-- 其他操作：事务处理、aop、注册bean、注入bean等操作 -->

</beans>
```

Spring的各种配置信息都在上面的文件中。

测试功能：加载容器、获取bean

```java
public class Application {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("classpath:applicationContext.xml");
```

```
        UserService service = (UserService)context.getBean("beanName");
        System.out.println(service.getName());
    }
}
```

bean 注册与注入

- Spring 是不会将所有类都注册为 bean 的，需要通过某种方式指定 bean 的注册、注入方式。
- 循环依赖：即两个类互相注入，构造函数注入时将抛出异常，因此使用 setter 方法进行注入。
- Spring 容器创建时，会验证每个 bean 的配置，并创建具有单例作用域并设置为预实例化（默认）的 Bean，但在实际创建 bean 之前不会设置属性。

bean 注册：运行时，将指定的类实例化并存放在容器内。

bean 注入：运行时，通过容器将一个 bean 或另一个值，通过反射赋值给另一个 bean 的成员变量。

1.注解方式

Spring 中可以使用注解方式标记一个类，并将此类实例化为 bean 并注册到 IOC 容器中，也可以使用注解将 bean 注入到指定位置。使用注解的优势在于代码简洁，因此这是最常见的方式。

Spring 使用注解，需要在配置文件中声明使用注解：

XML
```xml
<context:component-scan base-package="包名"/>
```

注册 bean

Java
```java
// 类上标记,除此之外还有：@Controller、@Service、@Repository等
@Component
public class Product {
    public String name = "被调用者";
}
```

bean 注册配置:

- @Scope：设置 bean 的作用域。
- @PostConstruct：指定初始化回调方法。
- @PreDestroy：指定销毁回调方法。

注入 bean

Java
```java
@Component // 调用者也必须注册为bean，否则无法注入
public class Consumer {

    @Autowired // 注入bean：先使用类型，后使用bean名
    public Product product;

    @Value     // 注入值
    public Integer value;

    @Resource  // 注入bean：先使用名称、后使用类型匹配
    public Source source;

}
```

bean 注入配置:

- @Qualifier("bean 名")：为 @Autowired 指定注入的 bean 名，构造注入时放在参数中，set 方法、属性注入时放在方法、属性上。
- @Required：属性必需已经配置了注入的 bean，用于 set 方法。
- @Primary：按类型自动装配时，给某个同类型 bean 优先权。

2.配置类方式

Java
```java
@Configuration                          // 表明这个类为bean配置类(必选注解)
@ComponentScan(basePackages = "包名")  //扫描一个包，自动注册bean
```

```java
@Import(ConfigA.class)                  //引入另一个配置类
@Profile("development")                 //设置此配置类的使用环境(开发环境)
public class AppConfig {

    @Bean
    public MyBean myBean(){
      return new MyBean();
    }

    @Bean                               //注册一个bean：myBean2
    @Scope("prototype")                 //设置bean的作用域。
    @PostConstruct()
    public MyBean2 myBean2(){
      return new MyBean2();
    }

}
```

3.xml 配置方式

xml 配置方式即在 Spring 配置文件中添加 bean 的配置信息，或者在 Spring 配置文件中引入其它的 xml 文件，并在那些 xml 中配置 bean 信息：

XML

```xml
<!-- 引入一个bean配置文件 -->
<import resource="themeSource.xml"/>
```

利用命名空间实现注册、注入

beans 中添加：xmlns:p="http://www.springframework.org/schema/p"

XML

```xml
<bean id="bean名" class="包名.类名" p:属性名="注入的值"/>
```

利用 Setter 方法参数注册、注入

XML

```xml
<!--属性支持级联-->
<bean id="bean名" class="包名.类名">
    <property name="属性名" value="注入值"/>
    <property name="属性名" ref="注入bean"/>
</bean>
```

利用构造方法参数注册、注入

XML

```xml
<!--注意：property-arg只能出现一次-->
<bean id="bean名" class="包名.类名">
  <property-arg name="参数"  value="注入的值"/>
  <property-arg  name="参数"  ref="注入的bean名"/>
</bean>
```

批量注入

XML

```xml
<!--property、property-arg都行，对应的参数为集合类型-->
<property name="数组参数">
  <list>
    <ref bean="bean1">
    <ref bean="bean2">
    <ref bean="bean3">
  </list>
</property>
```

空值注入

XML

```xml
<!--如下两种方式-->
<property name="un"value=""></property>
<property name="un"><null/></property>
```

注入配置

自动装配即根据某一规则进行装配，符合条件就会自动装配，规则有三种：

- byName：属性名=bean名时装配，setter方法注入。
- byType：属性类型=bean类型时装配，setter方法注入。
- constructor：参数类型=bean类型时装配，构造方法注入。

```xml
<!--装配的bean-->
<bean id="user" class="com.wen.User" autowire="byName" >
<!--被装配的bean: 内含User类型属性-->
<bean id="name" class="com.wen.Name"/>
```

自动装配不支持原始类型数据，并且装配时，重复的bean会被最新的覆盖。

配置属性大全

```xml
<bean
   id="bean名" class="类路径" scope="bean的作用域" parent="继承的父bean"

   init-method="bean的初始化方法 "  destory-method=" bean的销毁方法 "

   autowire="自动装配方式"  autowire-candidate="是否允许自动装配，默认为true"

   lazy-init="是否懒加载，容器在bean被使用时才创建其实例，默认true"

   scope="bean的作用域：
       单例（singleton：容器仅一个bean）、
       原型（prototype：每次获取bean时生成一个新的实例）、
       请求（request：单个http请求仅一个bean，请求完成时销毁）、
       会话（session：单个会话仅一个bean，请求完成时销毁）、
       全局会话（global-session：Portlet应用程序中使用，每个全局会话只有一个bean实例）、
       自定义范围（Application、WebScoket）
   "
/>
```

Spring 容器

构建容器对象：注意ApplicationContext是它们的接口父类，定义为此类型会限制实现类的方法。

```java
// 类路径的xml：初始化Spring容器，并加载applicationcontext.xml文件，参数可指定多个xml
ApplicationContext con1 = new ClassPathXmlApplicationContext("applicationContext.xml");

// 指定文件的xml
ApplicationContext con2 = new FileSystemXmlApplicationContext(new String[]{"D:/source.xml"});

// 指定的配置类
ApplicationContext con3 = new AnnotationConfigAPPlicationContext(SpringConfig.class);

// 获取web应用默认配置文件
ApplicationContext con4 = new XmlWebApplicationContext();

// 根据配置类初始化容器
ApplicationContext con4 = new AnnotationConfigApplicationContext(BeanConfig.class);
```

获取bean对象：

```java
ApplicationContext con = new XmlWebApplicationContext();
// 获取bean
User user1 = (User) con.getBean("user");
User user2 = con.getBean("user",User.class);
```

关闭-销毁容器：

```java
con.close();                // web应用关闭容器，通常会自动关闭
con.registerShutdownHook(); // 非web应用关闭容器
```

Spring Aop 面向切面

面向方面的编程（AOP）通过提供另一种思考程序结构的方式来补充面向对象的编程（OOP）。OOP 中模块化的关键单元是类，而在 AOP 中模块化是方面。

IOC 不依赖 AOP，AOP 强化了 IOC。

AOP 依赖：

```XML
<!-- SpringAOP包 -->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aop</artifactId>
  <version>5.2.0.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjweaver</artifactId>
  <version>1.9.5</version>
</dependency>
```

AOP 中的几个概念：

- **切面**(Aspect)： 一些横跨多个类的公共模块。
- **连接点**(Joint Point)： 目标类中插入代码的地方。连接点可以是方法、异常、字段，连接点处的切面代码会在方法执行、异常抛出、字段修改时触发执行。
- **通知**(Advice)： 在连接点插入的实际代码(即切面的方法)，有 5 种不同类型：
  - before ： 前置通知，在方法之前运行。
  - after：后置通知，在方法之后运行（不管方法是否成功执行）。
  - after-returning：返回通知，在方法之后运行（当方法执行成功时）。
  - after-throwing：异常通知，方法抛出异常后运行。
  - around：环绕通知，方法被调用之前和之后运行。
- **切点**(Pointcut)：定义了连接点的条件，一般通过正则表达式。例如，可以定义所有以 loadUser 开头的方法作为连接点，插入日志代码。

1.xml 方式配置

applicationContext.xml 头文件：

- 1.引入空间 AOP：xmlns:aop="http://www.springframework.org/schema/aop"
- 2.xsi:schemaLocation 加入：
  http://www.springframework.org/schema/aop
  http://www.springframework.org/schema/aop/spring-aop.xsd

```XML
<!-- 配置aop -->
<aop:config>

  <!-- 配置切面： -->
  <aop:aspect id="切面id" ref="切面类的bean">
      <!-- 定义切点：用正则匹配方法，然后拦截此方法。(此方法为被插入切面的方法) -->
      <aop:pointcut id="切点id" expression="execution(public void 方法(String,String))"/>

      <!-- 前置通知：方法调用前执行。 -->
      <aop:before method="前置方法" pointcut-ref="切点id"/>

      <!-- 返回通知：方法调用成功后执行。 -->
      <aop:after-returning method="返回方法" returning="返回值" pointcut-ref="切点id"/>

      <!-- 后置通知：方法调用后执行。 -->
      <aop:after method="后置方法" pointcut-ref="切点id"/>

      <!-- 异常通知：抛出异常时执行。 -->
      <aop:after-throwing method="异常方法" pointcut-ref="切点id" throwing="e"/>

      <!-- 环绕通知：方法调用前后执行。 -->
      <aop:around method="环绕方法" pointcut-ref="切点id" />
  </aop:aspect>

</aop:config>
```

2.注解方式配置

> 需要开启注解扫描

- @Aspect：定义一个切面（对切面类注释）。
- @Pointcut：定义一个切点（对切面内一个空方法注释，参数为匹配表达式）。
- @Before：定义一个前置通知，相当于BeforeAdvice（对切面方法注释，参数为”空方法名()”）。
- @After：定义一个后置通知（注释同上）。
- @AfterReturning：定义一个返回通知，相当于AfterReturningAdvice（注释同上）。
- @AfterThrowing：定义一个异常通知，相当于ThrowingAdvice（注释同上）。
- @Around：定义一个环绕通知，相当于MethodInterceptor（注释同上）。

示例：

```Java
//定义切面：
@Aspect("aop-AA")//定义切面
@Component("AA")//定义bean
public class AllLogAdvice {...}

//定义切点：
@Pointcut("execution(public void browse(String,String))")
public void allfun() {//使用一个返回值为void，方法体为空的方法来命名切点}

//定义通知：
@Before("allfun()")
public void aop_before(JoinPoint jp) {...}
```

切点匹配规则：

- execution()：方法信息

```Java
execution(可见性 返回值类型 类路径.类目名(参数类型) 异常类型)
// *表示通配符1、..表示通配符多个
execution(public * com.test.Controller(java.lang.String,..))
```

- within(): 类信息

```Java
within(类路径)
// *表示通配符1、..表示通配符多个
within(com.test.Controller)
```

- args：参数

```Java
args(java.lang.String)
```

- this和target：类或接口

```Java
this(类路径) // 如果类为接口实现类，实际会代理cglib生成的子类
target(类路径)
```

- @within： 类注解

```Java
@within(org.springframework.web.bind.annotation.RestController)
```

- @annotation：方法注解

```Java
@annotation(org.springframework.web.bind.annotation.RequestMapping)
```

- @args：参数注解

```java
@annotation(org.springframework.web.bind.annotation.Param)
```

SpringBoot 注解 AOP

引入 aop 依赖：

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

切面类：

```java
@Aspect
@Component
public class LogAspect {

    // 定义切点：控制层所有方法
    @Pointcut("@within(org.springframework.web.bind.annotation.RestController)")
    public void cutPoint(){}


    // 定义环绕通知：ProceedingJoinPoint 是连接点，JoinPoint的子类
    @Around("cutPoint()")
    public Object doAround(ProceedingJoinPoint point) throws Throwable {
      // 使用proceed 调用目标方法，否则请求无法继续
      Object proceed = point.proceed();
      return proceed;
    }

    // 定义前置通知
    @Before("cutPoint()")
    public void doBefore(JoinPoint  point) throws Throwable {}

    // 定义后置通知：returning将目标方法返回值绑定到参数responseBody上
    @AfterReturning(value = "cutPoint()",returning = "responseBody")
    public void doAfterReturning(JoinPoint point,Object responseBody){}

    // 定义异常通知：throwing 将目标方法异常绑定到参数throwable上
    @AfterThrowing(value = "cutPoint()",throwing = "throwable")
    public void doAfterThrowing(Throwable throwable){}



}
```

也可以不单独定义切点：

```java
// 定义前置通知
@Before("@annotation(org.springframework.web.bind.annotation.RestController)")
public void doBefore(JoinPoint  point) throws Throwable {}
```

常用方法：

```java
// 获取请求对象
HttpServletRequest request = ((ServletRequestAttributes)RequestContextHolder.getRequestAttributes()).getRequest();

// 获取目标方法
Method method = ((MethodSignature) point.getSignature()).getMethod();
```

线程池常量：如果获取日志，日志对象需要跨过不同的方法，因此需要将对象存储起来

```
// 类中定义一个线程池变量存放log对象
private final ThreadLocal<Log> threadLocal = new ThreadLocal<>();

// 定义log对象并存放
Log log = new Log();
threadLocal.set(log);
```

```
7
8   // 获取log对象
9   Log log = threadLocal.get();
10
11  // 移除log对象：log对象使用完成要记得移除
12  threadLocal.remove();
```

Spring 附加

Spring 国际化

MessageSource 接口提供国际化（i18n）功能。

Spring 还提 HierarchicalMessageSource 接口，可以分层解析消息。

接口方法如下：

Java

```java
//从MessageSource中检索消息，找不到使用默认值
String getMessage(String code, Object[] args, String default, Locale loc);
//和上面一样，但不提供默认值
String getMessage(String code, Object[] args, Locale loc);
//属性都包装在类中
String getMessage(MessageSourceResolvable resolvable, Locale locale);
```

使用的bean：ResourceBundleMessageSource，名称设为 messageSource，因为 xml 加载时会搜索 messageSource。

查看

官网

Spring 标签

> 除了使用命名空间，还可以使用 value 标签来指定注入的值，Spring 容器使用 JavaBeans 机制将 value 元素内的文本转换为 `java.util.Properties` 实例 `PropertyEditor`

XML

```xml
<property name="properties">
    <value>
        jdbc.driver.className=com.mysql.jdbc.Driver
        jdbc.url=jdbc:mysql://localhost:3306/mydb
    </value>
</property>
```

< idref >

使用标签指定注入的值，会在容器创建时验证 bean 是否存在，而且在你使用一些拦截器时，可以验证你的 beanID 是否正确。

使用此标签：

XML

```xml
<bean id="bean" class="...">
  <property name="属性">
      <idref bean="bean2"/>
  </property>
</bean>
```

不使用此标签：

XML

```xml
<bean id="bean" class="...">
    <property name="属性" value="bean2"/>
</bean>
```

< ref >

XML

```xml
<bean id="bean" class="...">
  <property name="属性">
      <ref bean="bean2"/>
  </property>
</bean>
```

ref的两个属性：

- bean属性：指定注入的bean。
- parent属性：指定引用父容器的bean，允许和当前bean重名。

内部bean

> 内部bean不需要设置id，因为它是匿名的，设置id也会被容器忽略。

XML

```xml
<bean id="outer" class="...">
    <property name="target">
        <bean class="com.example.Person"> <!-- this is the inner bean -->
            <property name="name" value="Fiona Apple"/>
            <property name="age" value="25"/>
        </bean>
    </property>
</bean>
```

< list/>< set/>< map/>< props/>

> 分别对应Java中：List、Set、Map、Properties。

XML

```xml
<bean id="beanID" class="bean对应的类">

    <property name="Properties属性">
        <props>
            <prop key="name">wenyijun</prop>
            <prop key="email">2476545423@qq.com</prop>
            <prop key="age">22</prop>
        </props>
    </property>

    <property name="List属性">
        <list>
            <value>后跟引用的列表元素</value>
            <ref bean="user" />
        </list>
    </property>
    <property name="Map属性">
        <map>
            <entry key="name" value="下面是个user"/>
            <entry key ="val" value-ref="user"/>
        </map>
    </property>
    <property name="Set属性">
        <set>
            <value>后跟set元素</value>
            <ref bean="user" />
        </set>
    </property>
</bean>
```

bean继承

父bean：

XML

```xml
<bean id="parent" abstract="true" class="com.FUser">
  <property name="user">
    <props>
      <prop key="name">xxx</prop>
      <prop key="age">100</prop>
    </props>
  </property>
</bean>
```

子bean：继承并覆盖父bean默认值(key相同则覆盖，key不同则继承)。

```xml
<bean id="child" parent="parent">
  <property name="user">
    <props merge="true">
      <prop key="name">wenyijun</prop>
      <prop key="age2">22</prop>
    </props>
  </property>
</bean>
```

> 上面将会有3个键值: name、age、age2（merge="true"进行合并）。
> 注意：不能合并不同类型集合、父bean不需要指定merge。

生命周期回调

执行顺序：注解>继承接口>命名指定

1.初始化回调

容器在bean上设置了所有必需的属性后，该接口可让bean执行初始化工作。

bean：

```xml
<bean id="app" class="com.App" init-method="init"/>
```

回调方法：

```java
public class App {
    public void init() {
       //回调方法
    }
}
```

> 也可以直接继承InitializingBean 接口，但这样会耦合到spring，两种方法效果一样。
> 继承接口的初始化回调方法：**afterPropertiesSet( )**。
> 也可以使用注解指定一个方法： **@PostConstruct**

2.销毁回调

包含该接口的容器被销毁时。

bean:

```xml
<bean id="app" class="com.App" destroy-method="cleanup"/>
```

回调方法：

```java
public class ExampleBean {
    public void cleanup() {
        //回调方法
    }
}
```

> 也可以直接继承DisposableBean接口，但这样会耦合到spring，两种方法效果一样。
> 继承接口的销毁回调方法：destroy() 。
> 也可以使用注解指定一个方法： @PreDestroy

3.启动、停止回调

任何被spring管理的对象都可以实现接口

Lifecycle

```java
public interface Lifecycle {
```

```
    void start();

    void stop();

    boolean isRunning();
}
```

4.创建自定义的@Qualifier

定义注解：

```Java
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Genre {
    String value();
}
```

使用：

```Java
 @Autowired
 @Genre("Action")
 private MovieCatalog actionCatalog;
```

限定符自动装配

注解：

```Java
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Offline {

}
```

使用：

```Java
@Autowired
@Offline
private MovieCatalog offlineCatalog;
```

xml 注入：

```XML
<bean class="example.SimpleMovieCatalog">
    <qualifier type="Offline"/>
</bean>
```

> 可以使用组合注解，也可以修改注解的默认值。

Spring 资源加载

资源接口 Resource 提供了资源的访问，接口信息如下：

```Java
public interface Resource extends InputStreamSource {
    //资源是否存在
    boolean exists();
    //资源是否具有打开流的句柄，是则不能多次读取，读完关闭
    boolean isOpen();

    URL getURL() throws IOException;

    File getFile() throws IOException;

    Resource createRelative(String relativePath) throws IOException;
```

```
    String getFilename();

    String getDescription();
}
```

- getInputStream()：找到并打开资源，返回一个资源 InputStream 读取。预计每次调用都会返回一个新的 `InputStream` 。呼叫者有责任关闭流。
- getDescription()：获取资源描述信息，通常是 url、file
- UrlResource：访问 url 资源
- ClassPathResource：访问类路径资源
- FileSystemResource：访问 url、file 资源
- ServletContextResource：流访问和 URL 访问，取决于 Servlet 容器。

资源加载器

Java
```
public interface ResourceLoader {
    Resource getResource(String location);
}
```

使用：

Java
```
Resource template = ctx.getResource("classpath:some/resource/path/myTemplate.txt");
Resource template = ctx.getResource("file:///some/resource/path/myTemplate.txt");
Resource template = ctx.getResource("https://myhost.com/resource/path/myTemplate.txt");
```

Spring 整合

Spring 整合 JDBC

1.maven 依赖：处理 SpringIOC 的依赖，还需要引入 sql 驱动、jdbc

XML
```
<!-- MySql驱动 -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.8</version>
</dependency>
<!-- JDBC支持 -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>4.3.7.RELEASE</version>
</dependency>
```

2.配置 jsbc 数据源、jdbc 操作 jdbcTemplate

execute（sql）：执行 sql，如创建、删除数据表等。

update()：执行插入、更新、删除

- int update(sql)：返回影响行数
- int update(PreparedStatementCreator pp)：执行 pp 返回的语句，返回影响行数
- int update(sql,PreparedStatementsetter ps)：设置 SQL 语句参数，返回影响行数
- int update(sql,Object oo)：使用 oo 对象设置 SQL 语句参数（参数不能为空），返回影响行数

query（）：执行查询语句

- List query（sql,PreparedStatementsetter ps，RowMapper rm）：sql 语句创建 ps 对象，通过 rm 返回到 list 中
- List query（sql,Object oo，RowMapper rm）：使用 oo[]设置 SQL 参数，通过 rm 返回到 list 中
- queryForObject（sql,Object oo，RowMapper rm）：使用 oo[]设置 SQL 参数，通过 rm 返回单行记录(Object)，rm 建议使用匿名内部类，来设置 user 对象的属性值

- queryForList（sql,Object oo，RowMapper rm，class ct）：返回多行数据列表，ct参数返回list元素类型

XML

```xml
<!-- beans|tx|aop|context 按需指定，beans必须有 -->
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://www.springframework.org/schema/beans"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
            http://www.springframework.org/schema/beans   http://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/tx      http://www.springframework.org/schema/tx/spring-tx.xsd
            http://www.springframework.org/schema/aop     http://www.springframework.org/schema/aop/spring-aop.xsd
            http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd
       ">

    <!-- 扫描com.test.service包下所有能注册bean的注解 -->
    <context:component-scan base-package="com.test.service">
        <!-- 被@Controller注解的类被排除 -->
        <context:exclude-filter type="annotation" expression="org.springframework.stereotype.Controller"/>
    </context:component-scan>

    <!-- 配置解析配置文件：获取application.properties中配置的变量，用${变量名}获取 -->
    <context:property-placeholder location="classpath:db.properties"/>
    <!-- bean-dataSource 配置数据源：值被注入到jdbcTemplate的属性中 -->
    <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="${jdbc.driver}"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>

    <!-- bean-jdbcTemplate 配置JdbcTemplate: spring的jdbc操作 -->
    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
         <property name="dataSource" ref="dataSource"/>
    </bean>

</beans>
```

连接池配置：db.properties

.properties

```properties
jdbc.driver=com.mysql.cj.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/test?allowMultiQueries=true&serverTimezone=UTC&useUnicode=true&characterEncoding=utf8
jdbc.username=root
jdbc.password=root
```

3.构建数据访问层代码：jdbcTemplate的具体操作此处不过多介绍。

Java

```java
/**
 * 使用注解方式注册bean：userDao，注意本类的要被注解扫描扫到才能注册
 */
@Repository
public class UserDao {

    /**
     * 使用注解方式注入bean：jdbcTemplate
     */
    @Resource
    private JdbcTemplate jdbcTemplate;

    /**
     * 根据id查询单个用户
     * @param id 用户id
     * @return 用户Map
     */
    public Map<String,Object> selectUser(Integer id){
        return jdbcTemplate.queryForMap("select * from t_user where id = ?",id);
    }

}
```

4.测试jdbc操作

Java

```java
public class Application {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
        UserDao dao = (UserDao) context.getBean("userDao");
        Map<String, Object> userMap = dao.selectUser(1);
        System.out.println(userMap.get("username"));
    }

}
```

Spring 整合 Servlet

1.引入 servlet 依赖

XML

```xml
<!-- spring web包 -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>4.3.7.RELEASE</version>
</dependency>
<!-- sevlet包 -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>2.5</version>
    <scope>provided</scope>
</dependency>
```

2.在 servlet 的 web.xml 配置(web.xml 可建在 java 同目录 webapp/WEB-INF 下，打包时指定即可)

XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">

    <display-name>spring_servlet</display-name>
    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
        <welcome-file>index.htm</welcome-file>
        <welcome-file>index.jsp</welcome-file>
        <welcome-file>default.html</welcome-file>
        <welcome-file>default.htm</welcome-file>
        <welcome-file>default.jsp</welcome-file>
    </welcome-file-list>

    <!-- 加载Spring 上下文，即加载applicationContext.xml配置 -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:applicationContext.xml</param-value>
    </context-param>
    <!-- 监听Spring 上下文 -->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

    <!-- 这个servlet只用于创建ApplicationContext对象，并不被人访问所以无须配置servlet-mapping -->
    <servlet>
        <servlet-name>servletManager</servlet-name>
        <servlet-class>com.test.controller.ServletManager</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <!-- end -->

    <servlet>
        <servlet-name>userController</servlet-name>
        <servlet-class>com.test.controller.UserController</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>userController</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>

</web-app>
```

3.创建ApplicationContext对象：用于初始化容器，否则无法注册bean

```java
public class ServletManager extends HttpServlet {

    private static ApplicationContext context = null;

    /**
     * 初始化Servlet方法 ： 加载容器
     */
    @Override
    public void init() throws ServletException {
        if(context == null){
            context = WebApplicationContextUtils.getWebApplicationContext(this.getServletContext());
        }
    }

    /**
     * 获取bean： 不能通过注解在servlet中直接注入，因此需要获取bean
     * @param beanName ben名称
     * @return bean对象
     */
    public static Object getBean(String beanName){
        return context.getBean(beanName);
    }

    /**
     * 销毁Servlet
     */
    @Override
    public void destroy() {
        super.destroy();
    }

}
```

4.处理请求的Servlet类

```java
public class UserController extends HttpServlet {

    @Override
    protected void service(HttpServletRequest request, HttpServletResponse response) throws IOException {
        UserDao userDao = (UserDao) ServletManager.getBean("userDao");
        String username = "";
        if ("/selectUserName".equals(request.getServletPath()) && "GET".equals(request.getMethod())) {
            Map<String, Object> user = userDao.selectUser(1);
            username = (String) user.get("username");
            System.out.println(username);
        }
        // 响应json数据给客户端
        response.setContentType("application/json; charset=utf-8");
        String userJson = "{\"name\":\"" + username + "\"}";
        OutputStream out = response.getOutputStream();
        out.write(userJson.getBytes(StandardCharsets.UTF_8));
        out.flush();
    }
}
```

5.项目配置问题

- 引用配置文件时最好要加上classpath:，否则会出现找不到文件的错误。
- 项目结构-模块：项目名下有Spring、Web模块。Spring指定上下文文件；Web描述符指向webxml、资源目录指向WEB-INF父目录。
- 项目结构-工件：基于模块新建一个server_web:war exploded工件，再基于该工件新建一个server:war工件，并使用tomcat运行server:war。
- HttpServlet子类不能直接进行注解注入，只能手动获取bean

编译后的目录结构：

Spring 整合 MyBatis

Spring 整合 SpringMVC