第3章 Spring技术和Spring IoC

学习目的与学习要求

学习目的：深入了解 Spring 框架及模块，掌握基本配置及 IoC 机制。

学习要求：扎实掌握 Spring 的核心配置及 IoC 机制，熟练搭建 Spring 环境的流程，学会配置和管理 bean。

本章主要内容

本章介绍 Spring 框架的结构概述，重点讲解 Spring IoC 原理、bean 的各种配置方式及作用域、使用 BeanFactory 和 ApplicationContext 加载 Spring 配置和管理 bean。

接下来讨论 Spring 框架，它是连接 Spring MVC 与 MyBatis 的桥梁，同时很好地处理了业务逻辑层。

3.1 Spring 简介

Spring 框架是一个分层架构，由 7 个定义好的模块组成。Spring 模块构建在核心容器之上，核心容器定义了创建、配置和管理 bean 的方式，如图 3-1 所示。

组成 Spring 框架的每个模块（或组件）都可以单独存在，或者与其他一个或多个模块联合实现。每个模块的功能如下。

核心容器：提供 Spring 框架的基本功能。核心容器的主要组件是 BeanFactory，它是工厂模式的实现。BeanFactory 使用 IoC 模式将应用程序的配置和依赖性规范与实际的应用程序代码分开。

Spring 上下文：是一个配置文件，向 Spring 框架提供上下文信息。Spring 上下文包括企业服务，如 JNDI、EJB、电子邮件、国际化、校验和调度功能。

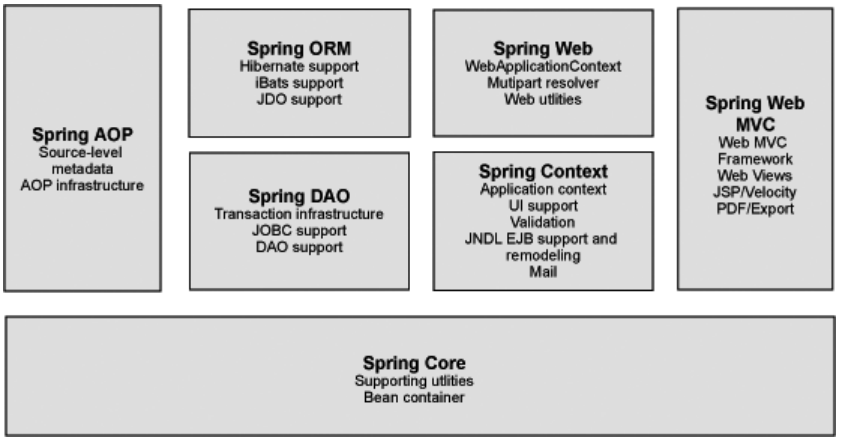

图 3-1 Spring 框架的模块

Spring AOP：通过配置管理特性，Spring AOP 模块直接将面向方面的编程功能集成到 Spring 框架中，所以可以很容易地使 Spring 框架管理的任何对象支持 AOP。Spring AOP 模块为基于 Spring 的应用程序中的对象提供了事务管理服务。使用 Spring AOP，不依赖 EJB 组件，就可以将声明性事务管理集成到应用程序中。

Spring DAO：JDBC DAO（Data Access Object）抽象层提供了有意义的异常层次结构，可用该结构管理异常处理和不同数据库供应商抛出的错误消息。异常层次结构简化了错误处理，并且极大地降低了需要编写的异常代码数量。Spring DAO 的面向 JDBC 的异常遵从通用的 DAO 异常层次结构。

Spring ORM：Spring 框架插入了若干个 Object/Relation Mapping 框架，从而提供了 ORM 的对象关系映射工具，其中包括 JDO、MyBatis 和 iBatis SQL Map。所有这些都遵从 Spring 的通用事务和 DAO 异常层次结构。

Spring Web 模块：Web 上下文模块建立在应用程序上下文模块之上，为基于 Web 的应用程序提供了上下文。所以，Spring 框架支持与 Jakarta Spring MVC 的集成。Web 模块还简化了处理大部分请求以及将请求参数绑定到域对象的工作。

Spring MVC 框架：MVC 框架是一个全功能的构建 Web 应用程序的 MVC 实现。通过策略接口，MVC 框架变成高度可配置的，MVC 容纳了大量视图技术，其中包括 JSP、Velocity、Tiles、iText 和 POI。

Spring 框架的功能可以用在任何的 J2EE 服务器中，大多数功能也适用于不受管理的环境。Spring 的核心要点是：支持不绑定到特定 J2EE 服务的可重用业务和数据访问对象。毫无疑问，这样的对象可以在不同 J2EE 环境（Web 或 EJB）、独立应用程序、测试环境之间重用。

3.2 Spring IoC

首先介绍 Spring IoC 这个最核心、最重要的概念。

3.2.1 IoC 的原理

IoC，直观地讲，就是由容器控制程序之间的关系，而非传统实现中由程序代码直接操控。这也就是所谓“控制反转”的概念所在：控制权由应用代码中转到外部容器，控制权的转移是所谓的反转。IoC 还有另外一个名字：“依赖注入(Dependency Injection)”。从名字上理解，所谓依赖注入，即组件之间的依赖关系由容器在运行期决定，形象地说，即由容器动态地将某种依赖关系注入组件中。

下面通过一个生动形象的例子介绍 IoC。

例如，一个女孩希望找到合适的男朋友，如图 3-2 所示。

可以有 3 种方式：

(1) 青梅竹马；

(2) 亲友介绍；

(3) 父母包办。

第一种方式是青梅竹马，如图 3-3 所示。

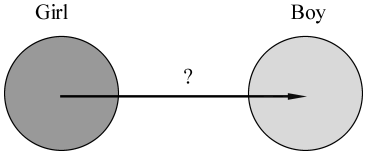

图 3-2 一个女孩希望找到合适的男朋友

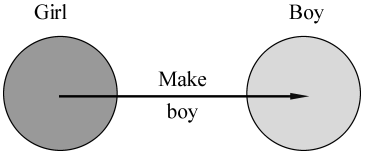

图 3-3 青梅竹马

通过代码表示如下：

```
public class Girl {
  void kiss(){
    Boy boy=new Boy();
  }
}
```

第二种方式是亲友介绍，如图 3-4 所示。

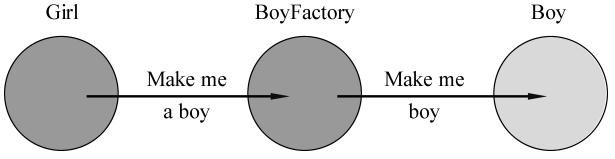

图 3-4 亲友介绍

通过代码表示如下：

```
public class Girl {
  void kiss(){
    Boy boy=BoyFactory.createBoy();
  }
}
```

第三种方式是父母包办，如图 3-5 所示。

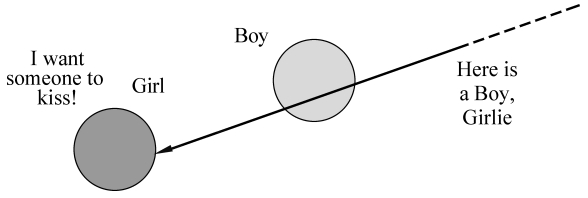

图 3-5 父母包办

通过代码表示如下：

```
public class Girl {
  void kiss(Boy boy){
    //kiss boy
    boy.kiss();
  }
}
```

哪种方式为 IoC 呢？虽然在现实生活中我们都希望青梅竹马，但在 Spring 世界里选择的却是父母包办，它就是 IoC，而这里具有控制力的父母就是 Spring 的所谓的容器概念。

典型的 IoC 如图 3-6 所示。

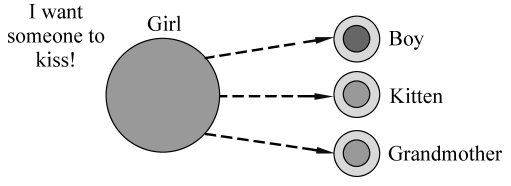

图 3-6 典型的 IoC

IoC 的 3 种依赖注入类型：

第一种是通过接口注射，这种方式要求类必须实现容器给定的一个接口，然后容器会利用这个接口给这个类注射它所依赖的类。

```
public class Girl implements Servicable{
  Kissable kissable;
  public void service(ServiceManager mgr) {
    kissable=(Kissable) mgr.lookup("kissable");
  }
  public void kissYourKissable() {
    kissable.kiss();
  }
}

<container>
  <component name="kissable" class="Boy">
    <configuration>…</configuration>
```

```
  </component><component name="girl" class="Girl" />
</container>
```

第二种是通过 setter()方法注射，这种方式也是 Spring 推荐的方式。

```
public class Girl {
  private Kissable kissable;
  public void setKissable(Kissable kissable) {
    this.kissable=kissable;
  }
  public void kissYourKissable() {
    kissable.kiss();
  }
}

<beans>
  <bean id="boy" class="Boy"/>
  <bean id="girl" class="Girl">
    <property name="kissable">
      <ref bean="boy"/>
    </property>
  </bean>
</beans>
```

第三种是通过构造方法注射类，这种方式 Spring 同样给予了实现，它和通过 setter()方法一样，都在类里无任何侵入性，但是不是没有侵入性，只是把侵入性转移了。显然，第一种方式要求实现特定的接口，侵入性非常强，不方便以后移植。

```
public class Girl {
  private Kissable kissable;
  public Girl(Kissable kissable) {
    this.kissable=kissable;
  }
  public void kissYourKissable() {
    kissable.kiss();
  }
}

PicoContainer container=new DefaultPicoContainer();
container.registerComponentImplementation(Boy.class);
container.registerComponentImplementation(Girl.class);
Girl girl=(Girl) container.getComponentInstance(Girl.class);
girl.kissYourKissable();
```

3.2.2 Bean Factory

Spring IoC 设计的核心是 org.springframework.beans 包，它的设计目标是与 JavaBean

组件一起使用。这个包通常不是由用户直接使用，而是由服务器将其用作其他多数功能的底层中介。下一个最高级抽象是 BeanFactory 接口，它是工厂设计模式的实现，允许通过名称创建和检索对象。BeanFactory 也可以管理对象之间的关系。

BeanFactory 支持两个对象模型。

- 单态模型：它提供了具有特定名称的对象的共享实例，可以在查询时对其进行检索。Singleton 是默认的，也是最常用的对象模型，对于无状态服务对象很理想。
- 原型模型：它确保每次检索都会创建单独的对象。在每个用户都需要自己的对象时，原型模型最适合。

bean 工厂的概念是 Spring 作为 IoC 容器的基础，IoC 将处理事情的责任从应用程序代码转移到框架。Spring 框架使用 JavaBean 属性和配置数据指出必须设置的依赖关系。

1. BeanFactory

BeanFactory 实际上是实例化、配置和管理众多 bean 的容器。这些 bean 通常彼此合作，因而它们之间会产生依赖。BeanFactory 使用的配置数据可以反映这些依赖关系（一些依赖可能不像配置数据一样可见，而是在运行期作为 bean 之间程序交互的函数）。

一个 BeanFactory 可以用接口 org.springframework.beans.factory.BeanFactory 表示，这个接口有多个实现。最常使用的简单的 BeanFactory 实现是 org.springframework.beans.factory.xml.XmlBeanFactory。（这里提醒一下：ApplicationContext 是 BeanFactory 的子类，所以大多数用户更喜欢使用 ApplicationContext 的 XML 形式。）

虽然大多数情况下，几乎所有被 BeanFactory 管理的用户代码都不需要知道 BeanFactory，但是 BeanFactory 还是以某种方式实例化。可以使用下面的代码实例化 BeanFactory：

```
InputStream is=new FileInputStream("beans.xml");
XmlBeanFactory factory=new XmlBeanFactory(is);
```

或者

```
ClassPathResource res=new ClassPathResource("beans.xml");
XmlBeanFactory factory=new XmlBeanFactory(res);
```

或者

```
ClassPathXmlApplicationContext appContext=new ClassPathXmlApplicationContext(
     new String[] {"applicationContext.xml", "applicationContext-part2.xml"});
//of course, an ApplicationContext is just a BeanFactory
BeanFactory factory=(BeanFactory) appContext;
```

很多情况下，用户代码不需要实例化 BeanFactory，因为 Spring 框架代码会做这件事。例如，Web 层提供支持代码，在 J2EE Web 应用启动过程中自动载入一个 Spring ApplicationContext。这个声明过程在这里描述：

编程操作 BeanFactory 将会在后面提到，下面集中描述 BeanFactory 的配置。

一个最基本的 BeanFactory 配置由一个或多个它所管理的 Bean 定义组成。在一个 XmlBeanFactory 中，根节点 Beans 中包含一个或多个 Bean 元素。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.
org/dtd/spring-beans.dtd">

<beans>

  <bean id="..." class="...">
    ...
  </bean>
  <bean id="..." class="...">
    ...
  </bean>

  ...

</beans>
```

2. Bean Definition

一个 XmlBeanFactory 中的 Bean 定义包括的内容有：

(1) classname，这通常是 Bean 的真正的实现类。但是，如果一个 Bean 使用一个静态工厂方法创建，而不是被普通的构造函数创建，那么这实际上就是工厂类的 classname。

(2) Bean 行为配置元素，它声明这个 Bean 在容器的行为方式(如 prototype 或 singleton，自动装配模式，依赖检查模式，初始化和析构方法)。

构造函数的参数和新创建 Bean 需要的属性：举一个例子，一个管理连接池的 Bean 使用的连接数目(既可以指定为一个属性，也可以作为一个构造函数参数)，或者池的大小限制和这个 Bean 工作相关的其他 Bean：比如它的合作者(同样可以作为属性或者构造函数的参数)。这也被叫作依赖。

上面列出的概念直接转化为组成 Bean 定义的一组元素。这些元素(见表 3-1)都有更详细的说明的链接。

表 3-1　Bean 定义的解释

特　性	说　明	特　性	说　明
class	Bean 的类	自动装配模式	自动装配协作对象
id 和 name	Bean 的标识符(id 与 name)	依赖检查模式	依赖检查
singleton 或 prototype	Singleton 的使用与否	初始化模式	生命周期接口
构造函数参数	设置 Bean 的属性和合作者	析构方法	生命周期接口
Bean 的属性	设置 Bean 的属性和合作者		

注意：Bean 定义可以表示为真正的接口 org.springframework.beans.factory.config.BeanDefinition 以及它的各种子接口和实现。然而，绝大多数的用户代码不需要与 BeanDefinition 直接接触。

3. Bean 类

class 属性通常是强制性的，有两种用法。在绝大多数情况下，BeanFactory 直接调用 Bean 的构造函数创建一个 Bean(相当于调用 new 的 Java 代码)。class 属性指定了需要创建的 Bean 的类。在比较少的情况下，BeanFactory 调用某个类的静态的工厂方法创建 Bean。class 属性指定了实际包含静态工厂方法的那个类。(至于静态工厂方法返回的 Bean 的类型是同一个类，还是完全不同的另一个类，这并不重要。)

1) 通过构造函数创建 Bean

当使用构造函数创建 Bean 时，所有普通的类都可以被 Spring 使用并且和 Spring 兼容。这就是说，被创建的类不需要实现任何特定的接口或者按照特定的样式进行编写，仅指定 Bean 的类就足够了。然而，根据 Bean 使用的 IoC 类型，你可能需要一个默认的(空的)构造函数。

另外，BeanFactory 并不局限于管理真正的 JavaBean，它也能管理任何你想让它管理的类。虽然很多使用 Spring 的人喜欢在 BeanFactory 中用真正的 JavaBean (仅包含一个默认的(无参数的)构造函数，在属性后面定义相对应的 setter()和 getter()方法)，但是在 BeanFactory 中也可以使用特殊的非 Bean 样式的类。例如，如果需要使用一个遗留下来的完全没有遵守 JavaBean 规范的连接池，不要担心，Spring 同样能够管理它。

使用 XmlBeanFactory 可以像下面这样定义 Bean class：

```
<bean id="exampleBean"
      class="examples.ExampleBean"/>
<bean name="anotherExample"
      class="examples.ExampleBeanTwo"/>
```

至于为构造函数提供(可选的)参数，以及在对象实例创建后设置实例属性，将会在后面叙述。

2) 通过静态工厂方法创建 Bean

当定义一个使用静态工厂方法创建的 Bean，同时使用 class 属性指定包含静态工厂方法的类，这时需要 factory-method 属性指定工厂方法名。Spring 调用这个方法(包含一组可选的参数)并返回一个有效的对象，之后这个对象就完全和构造方法创建的对象一样。用户可以使用这样的 Bean 定义在遗留代码中调用静态工厂。

下面是一个 Bean 定义的例子，声明这个 Bean 要通过 factory-method 指定的方法创建。注意，这个 Bean 定义并没有指定返回对象的类型，只指定包含工厂方法的类。在这个例子中，createInstance 必须是 static()方法。

```
<bean id="exampleBean"
      class="examples.ExampleBean2"
      factory-method="createInstance"/>
```

至于为工厂方法提供(可选的)参数，以及在对象实例被工厂方法创建后设置实例属性，将会在后面叙述。

3) 通过实例工厂方法创建 Bean

使用一个实例工厂方法(非静态的)创建 Bean 和使用静态工厂方法非常类似，调用一

个已存在的 Bean(这个 Bean 应该是工厂类型)的工厂方法创建新的 Bean。

使用这种机制,class 属性必须为空,而且 factory-bean 属性必须指定一个 Bean 的名字,这个 Bean 一定要在当前的 Bean 工厂或者父 Bean 工厂中,并包含工厂方法。而工厂方法本身仍然要通过 factory-method 属性设置。

下面是一个例子:

```
<!--The factory bean, which contains a method called
    createInstance -->
<bean id="myFactoryBean"
      class="...">
...
</bean>
<!--The bean to be created via the factory bean -->
<bean id="exampleBean"
      factory-bean="myFactoryBean"
      factory-method="createInstance"/>
```

虽然我们要在后面讨论设置 Bean 的属性,但是这个方法意味着工厂 Bean 本身能够被容器通过依赖注射管理和配置。

4. Bean 的标识符(id 与 name)

每个 bean 都有一个或多个 id(也叫作标识符,或名字;这些名词说的是一回事)。这些 id 在管理 Bean 的 BeanFactory 或 ApplicationContext 中必须是唯一的。一个 Bean 差不多总是只有一个 id,但是如果一个 Bean 有超过一个的 id,那么另外的那些本质上可以认为是别名。

在一个 XmlBeanFactory 中(包括 ApplicationContext 的形式),可以用 id 或者 name 属性指定 Bean 的 id(s),并且在这两个或其中一个属性中至少指定一个 id。id 属性允许指定一个 id,并且它在 XML DTD(定义文档)中作为一个真正的 XML 元素的 ID 属性被标记,所以 XML 解析器能够在其他元素指向它的时候做一些额外的校验。正因如此,用 id 属性指定 Bean 的 id 是一个比较好的方式。然而,XML 规范严格限定了在 XML ID 中合法的字符。通常这并不是真正限制你,但是如果有必要使用这些字符(在 ID 中的非法字符),或者想给 Bean 增加其他的别名,可以通过 name 属性指定一个或多个 id(用逗号,或者分号;分隔)。

5. Singleton 的使用与否

Beans 被定义为两种部署模式中的一种: singleton 或 non-singleton(后一种也叫作 prototype,尽管这个名词用得不精确)。如果一个 Bean 是 singleton 形态的,就只有一个共享的实例存在,所有和这个 Bean 定义的 id 符合的 Bean 请求都会返回这个唯一的、特定的实例。

如果 Bean 以 non-singleton、prototype 模式部署,对这个 Bean 的每次请求都会创建一个新的 Bean 实例。这对于每个 user 需要一个独立的 user 对象这样的情况是非常理想的。

Beans 默认被部署为 singleton 模式,除非有指定。把部署模式变为 non-singletion (prototype)后,每次对这个 Bean 的请求都会导致一个新创建的 Bean,而这可能并不是你

真正想要的。所以，仅在绝对需要的时候才把模式改成 prototype。

在下面这个例子中，两个 Bean 中的一个被定义为 singleton，另一个被定义为 non-singleton(prototype)。客户端每次向 BeanFactory 请求都会创建新的 exampleBean，而 AnotherExample 仅被创建一次；每次对它请求都会返回这个实例的引用。

```
<bean id="exampleBean"
      class="examples.ExampleBean" singleton="false"/>
<bean name="yetAnotherExample"
      class="examples.ExampleBeanTwo" singleton="true"/>
```

注意：当部署一个 Bean 为 prototype 模式，这个 Bean 的生命周期就会有稍许改变。通过定义，Spring 无法管理一个 non-singleton/prototype Bean 的整个生命周期，因为当它创建之后，它被交给客户端，而且容器根本不再跟踪它了。当说起 non-singleton/prototype Bean 的时候，可以把 Spring 的角色想象成 new 操作符的替代品。之后的任何生命周期方面的事情都由客户端处理。

3.2.3 ApplicationContext

Beans 包提供了以编程的方式管理和操控 Bean 的基本功能，而 context 包增加了 ApplicationContext，它以一种更加面向框架的方式增强了 BeanFactory 的功能。多数用户可以以一种完全的声明式方式使用 ApplicationContext，甚至不用手工创建它，但是却依赖像 ContextLoader 的支持类，在 J2EE 的 Web 应用的启动进程中用它启动 ApplicationContext。当然，这种情况下还是可以以编程的方式创建一个 ApplicationContext。

context 包的基础是位于 org.springframework.context 包中的 ApplicationContext 接口。它由 BeanFactory 接口集成而来，提供 BeanFactory 所有的功能。为了以一种更像面向框架的方式工作，context 包使用分层和有继承关系的上下文类，包括：

- MessageSource，提供对 i18n 消息的访问。
- 资源访问，如 URL 和文件。
- 事件传递给实现了 ApplicationListener 接口的 Bean。
- 载入多个(有继承关系)上下文类，使得每个上下文类都专注于一个特定的层次，如应用的 Web 层。

因为 ApplicationContext 包括 BeanFactory 所有的功能，所以通常建议先于 BeanFactory 使用，除有限的一些场合(如在一个 Applet 中，内存的消耗是关键的)，每千字节都很重要。接下来介绍 ApplicationContext 在 BeanFactory 的基本能力上增加的功能。

1. 使用 MessageSource

ApplicationContext 接口继承 MessageSource 接口，所以提供了 messaging 功能(i18n 或者国际化)。它同 NestingMessageSource 一起使用，就能处理分级的信息，这些是 Spring 提供的处理信息的基本接口。这里定义的方法有如下 3 个。

StringgetMessage (String code，Object[] args，String default，Locale loc)：这个方法是从 MessageSource 取得信息的基本方法。如果对于指定的 locale 没有找到信息，则使用

默认的信息。传入的参数 args 用来代替信息中的占位符，这是通过 Java 标准类库的 MessageFormat 实现的。

StringgetMessage（String code，Object[] args，Locale loc）：本质上和上一个方法一样，区别只是没有默认值可以指定；如果找不到信息，就会抛出一个 NoSuchMessage Exception。

StringgetMessage（MessageSourceResolvable resolvable，Locale locale）：上面两个方法使用的所有属性都封装到一个叫作 MessageSourceResolvable 的类中，可以通过这个方法直接使用它。

当 ApplicationContext 被加载的时候，它会自动查找在 context 中定义的 MessageSource Bean。这个 Bean 必须叫作 messageSource。如果找到了这样一个 Bean，所有对上述方法的调用将会被委托给找到的 message source。如果没有找到 message source，ApplicationContext 将会尝试查它的父亲是否包含这个名字的 Bean。如果有，它将会把找到的 Bean 作为 Message Source。如果它最终没有找到任何信息源，一个空的 StaticMessageSource 将会被实例化，使它能够接受上述方法的调用。

Spring 目前提供了两个 MessageSource 的实现，分别是 ResourceBundleMessageSource 和 StaticMessageSource。两个都实现了 NestingMessageSource，以便能够嵌套地解析信息。StaticMessageSource 很少被使用，但是它提供以编程的方式向 source 增加信息。Resource BundleMessageSource 用得更多一些，下面是它的一个例子：

```
<beans>
    <bean id="messageSource"
    class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basenames">
            <list>
                <value>format</value>
                <value>exceptions</value>
                <value>windows</value>
            </list>
        </property>
    </bean>
</beans>
```

这段配置假定你在 classpath 有 3 个 resource bundle，分别为 format、exceptions 和 windows。使用 JDK 通过 ResourceBundle 解析信息的标准方式，任何解析信息的请求都会被处理。

2. 事件传递

ApplicationContext 中的事件处理是通过 ApplicationEvent 类和 ApplicationListener 接口提供的。如果上下文中部署了一个实现了 ApplicationListener 接口的 Bean，每次一个 ApplicationEvent 发布到 ApplicationContext 时，那个 Bean 就会被通知。实质上，这是标准的 Observer 设计模式。Spring 提供了 3 个标准事件，见表 3-2。

表 3-2 内置事件

事　　件	解　　释
ContextRefreshedEvent	当 ApplicationContext 已经初始化或刷新后发送的事件。这里的初始化意味着所有的 Bean 被装载，singleton 被预实例化，以及 ApplicationContext 已准备好
ContextClosedEvent	当使用 ApplicationContext 的 close()方法结束上下文的时候发送的事件。这里的结束意味着 singleton 被销毁
RequestHandledEvent	一个与 Web 相关的事件，告诉所有的 Bean 一个 HTTP 请求已经被响应了(这个事件将会在一个请求结束后被发送)。注意，这个事件只能应用于使用了 Spring 的 DispatcherServlet 的 Web 应用

同样，也可以实现自定义的事件。通过调用 ApplicationContext 的 publishEvent()方法，并且指定一个参数，这个参数是你自定义的事件类的一个实例。下面看一个例子。

首先是 ApplicationContext:

```
<bean id="emailer" class="example.EmailBean">
    <property name="blackList">
        <list>
            <value>black@list.org</value>
            <value>white@list.org</value>
            <value>john@doe.org</value>
        </list>
    </property>
</bean>

<bean id="blackListListener" class="example.BlackListNotifier">
    <property name="notificationAddress">
        <value>spam@list.org</value>
    </property>
  </bean>
```

然后是实际的 Bean:

```
public class EmailBean implements ApplicationContextAware {

    /** the blacklist * /
    private List blackList;

    public void setBlackList(List blackList) {
        this.blackList=blackList;
}

    public void setApplicationContext(ApplicationContext ctx) {
        this.ctx=ctx;
}

    public void sendEmail(String address, String text) {
        if (blackList.contains(address)) {
```

```
            BlackListEventevt=new BlackListEvent(address, text);
            ctx.publishEvent(evt);
            return;
        }
        //send email
    }
}

public class BlackListNotifier implement ApplicationListener{

    /** notification address * /
    private String notificationAddress;

    public void setNotificationAddress(String notificationAddress) {
        this.notificationAddress=notificationAddress;
    }

    public void onApplicationEvent(ApplicationEvent evt) {
        if (evtinstanceofBlackListEvent) {
            //notify appropriate person
        }
    }
}
```

3. 在 Spring 中使用资源

很多应用程序都需要访问资源。Spring 提供了一个清晰透明的方案，以一种协议无关的方式访问资源。ApplicationContext 接口包含一个方法（getResource(String)）负责这项工作。

Resource 类定义了几个方法，见表 3-3。这几个方法被所有的 Resource 实现所共享。

表 3-3　资源功能

方　法	解　释
getInputStream()	用 InputStream 打开资源，并返回这个 InputStream
exists()	检查资源是否存在，如果不存在，则返回 false
isOpen()	如果这个资源不能打开多个流将会返回 true。常见的资源实现一般返回 false
getDescription()	返回资源的描述，通常是全限定文件名或者实际的 URL

Spring 提供了几个 Resource 的实现。它们都需要一个 String 表示的资源的实际位置。依据这个 String，Spring 将会自动为你选择正确的 Resource 实现。当向 ApplicationContext 请求一个资源时，Spring 首先检查你指定的资源位置，之后寻找任何前缀。根据不同的 Application Context 的实现，不同的 Resource 实现可被使用的 Resource 最好使用 ResourceEditor 配置，如 XmlBeanFactory。

接下来看一个 Spring IoC 实例。

（1）新建 Java 项目，如图 3-7 所示。

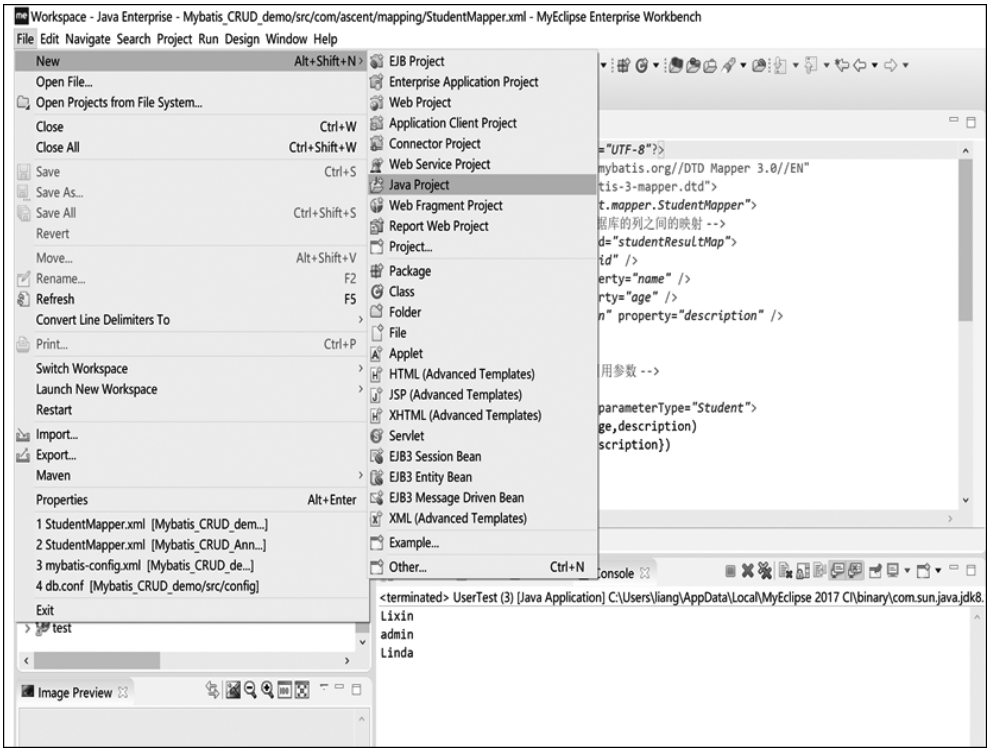

图 3-7 新建 Java 项目

在 Project name 中输入 Spring_IoC_demo，如图 3-8 所示。

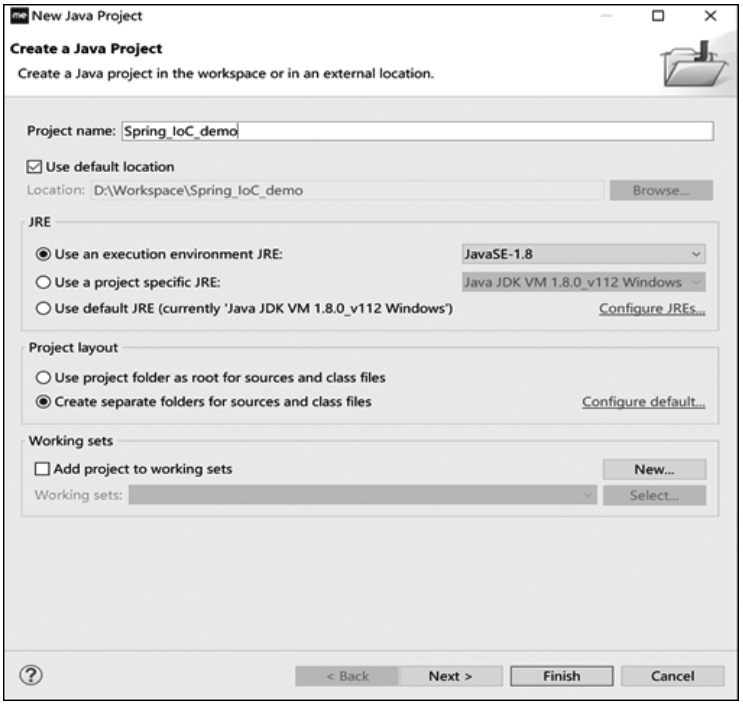

图 3-8 命名项目

（2）导入 Spring 项目依赖的 jar 包，右击 Spring_IoC_demo 项目，从弹出的快捷菜单中选择 Configure Facets→Install Spring Facet，如图 3-9 所示。

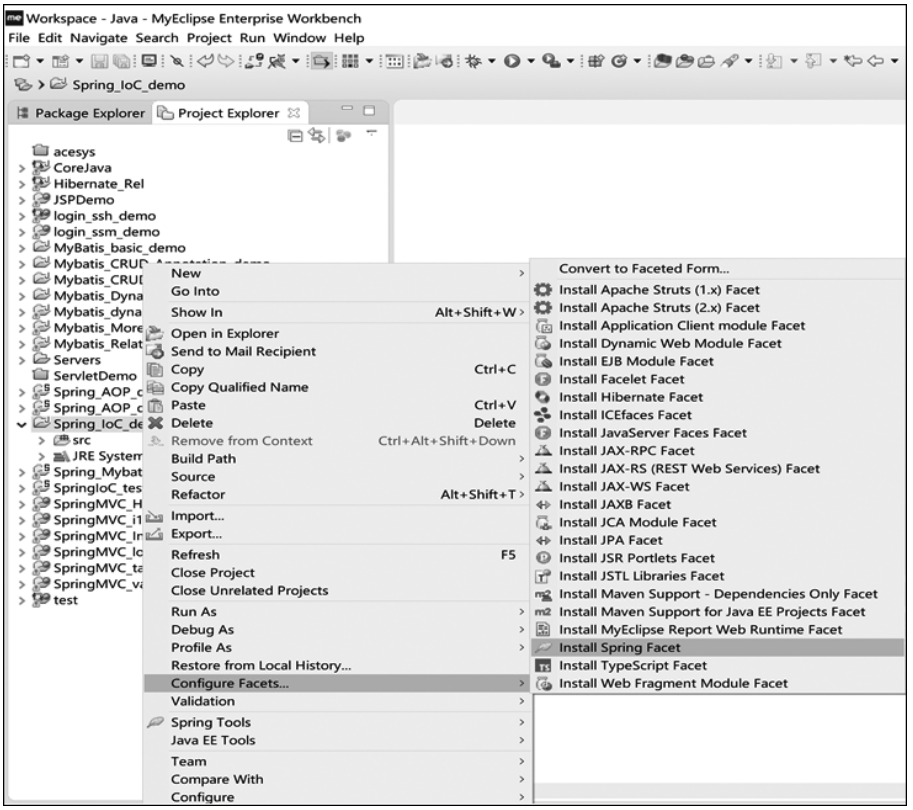

图 3-9　Install Spring Facet 界面 1

单击 Next 按钮，不需修改，直到单击 Finish 按钮，如图 3-10 所示。

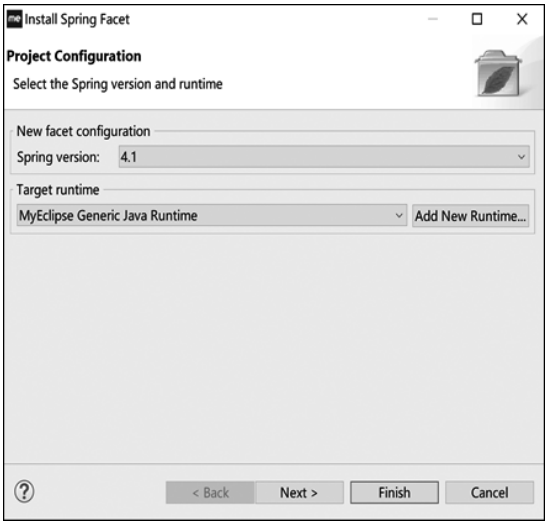

图 3-10　Install Spring Facet 界面 2

(3) 建立包结构。右击 src,从弹出的快捷菜单中选择 New→Package,如图 3-11 所示。

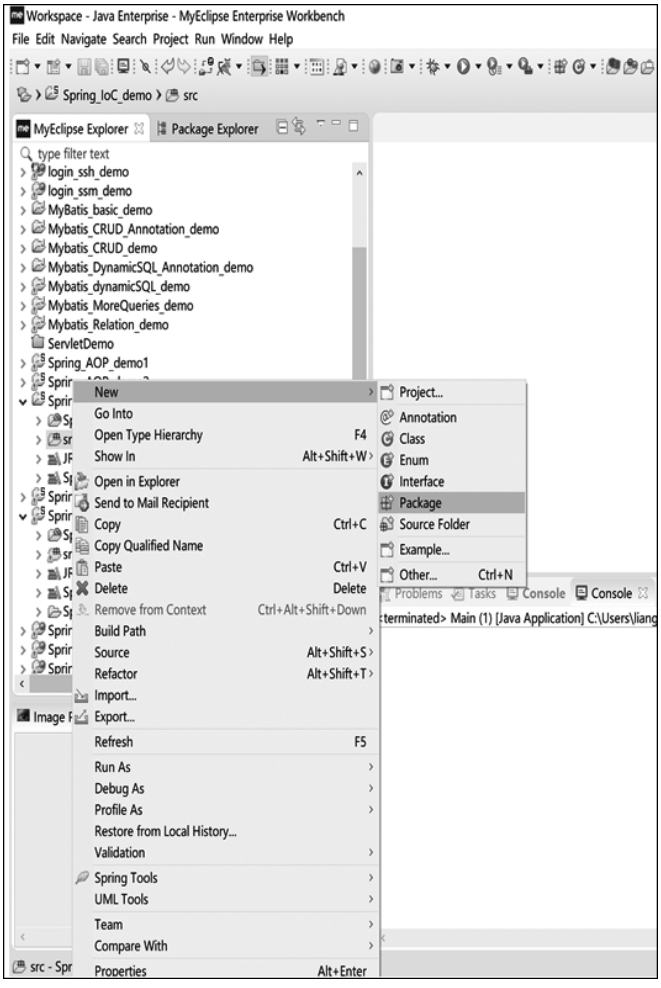

图 3-11 新建 Package

在 Name 处输入 com.ascent,之后单击 Finish 按钮,如图 3-12 所示。

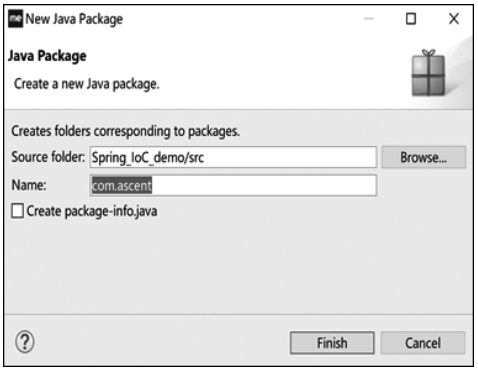

图 3-12 命名 Package

（4）在 src 目录下编写配置文件 applicationContext.xml，内容如下：

```
<?xmlversion="1.0"encoding="UTF-8"?>
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.1.xsd">

  <bean id="boy"class="com.ascent.Boy"/>
  <bean id="girl"class="com.ascent.Girl">
  <propertyname="kissable">
    <ref bean="boy"/>
      </property>
    </bean>
  </beans>
```

（5）在 com.ascent 目录下分别编写 Boy、Girl.java、Kissable.java 和 Test.java，代码如下：

```
//Boy.java
package com.ascent;

public class Boy implements Kissable{
  public void kiss(){
      System.out.println("This is Kiss Boy");
  }
}

//Girl.java
package com.ascent;

public class Girl {
    private Kissable kissable;
    public void setKissable(Kissable kissable) {
    this.kissable=kissable;
    }
    public void kissYourKissable() {
    kissable.kiss();
    }
}

//Kissable.java
package com.ascent;
```

```
public interface Kissable {
  public void kiss();
}

  //Test.java
  package com.ascent;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {

    public static void main(String[] args) {
        ApplicationContext apc=new ClassPathXmlApplicationContext(
            "applicationContext.xml");
        Girl g=(Girl) apc.getBean("girl");
        g.kissYourKissable();
    }

}
```

(6) 右击 Test 类,从弹出的快捷菜单中选择 Run As→Java Application,如图 3-13 所示。

得到如下结果:

```
This is Kiss Boy
```

3.3 项目案例

3.3.1 学习目标

使用 Spring 框架,为 eGov 项目提供功能组件对象,并装配功能组件为应用上层提供封装对象,在实践中学习如何使用 ApplicationContext 对象,获取 Spring 配置文件中声明的功能组件。如果是在 Web 环境中,请使用 WebApplicationContext 对象替代 ApplicatiionContext 对象。

3.3.2 案例描述

本章案例为 eGov 中一般用户浏览新闻信息中的两部分内容:头版头条新闻和综合新闻。其中,头版头条需要取一条新闻数据,而综合新闻则需要按一定条件获取 6 条新闻数据。

获取头版头条的一条新闻数据和获取综合新闻的 6 条新闻数据为一个业务功能,用 Java 中的一个类的一个方法实现,将两种结果合并到 HashMap 中。

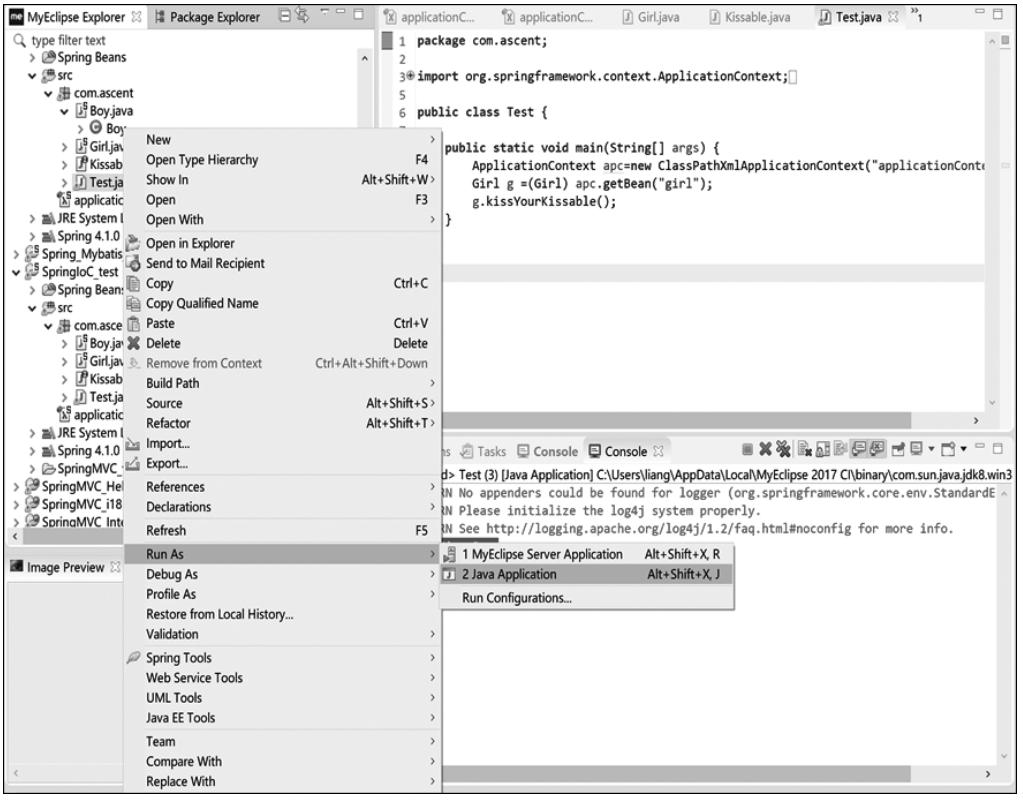

图 3-13　运行 class

3.3.3　案例要点

用户输入浏览器的 Web 项目地址，请求到达 Spring MVC 控制器，控制器再转向 index.jsp 页面。在 index.jsp 中像调用普通 Java 功能类一样，使用 JSTL 标签调用控制器中的相应业务控制方法，获取头条新闻和综合新闻的数据集合，然后再传给 index.jsp，在 index.jsp 中完成头版头条新闻和综合新闻数据的处理。由于没有使用到 Spring MVC 和 MyBatis，所以在这个案例中重点是 Spring 框架的搭建，以及如何使用 Spring DI/IoC 的依赖注入和控制反转配置业务类，使用 main()方法模拟 Spring MVC 控制器，借助 ApplicationContext 对象获取业务对象，执行业务方法，将执行结果输出到控制台终端。

3.3.4　案例实施

（1）在 MyEclipse 中新建 electrone Java Web 项目。

（2）导入 mysql-connector-java-5.1.47-bin.jar 到项目 buildpath 中，如图 3-14 所示。

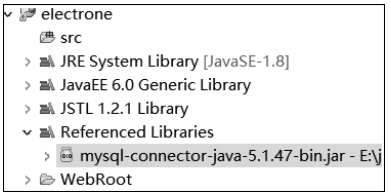

图 3-14　导入 mysql 驱动包

（3）为项目添加 Spring 的支持。

右击工程，从弹出的快捷菜单中选择 Spring Tools→Add Spring Runtime Dependencies，如图 3-15 所示。

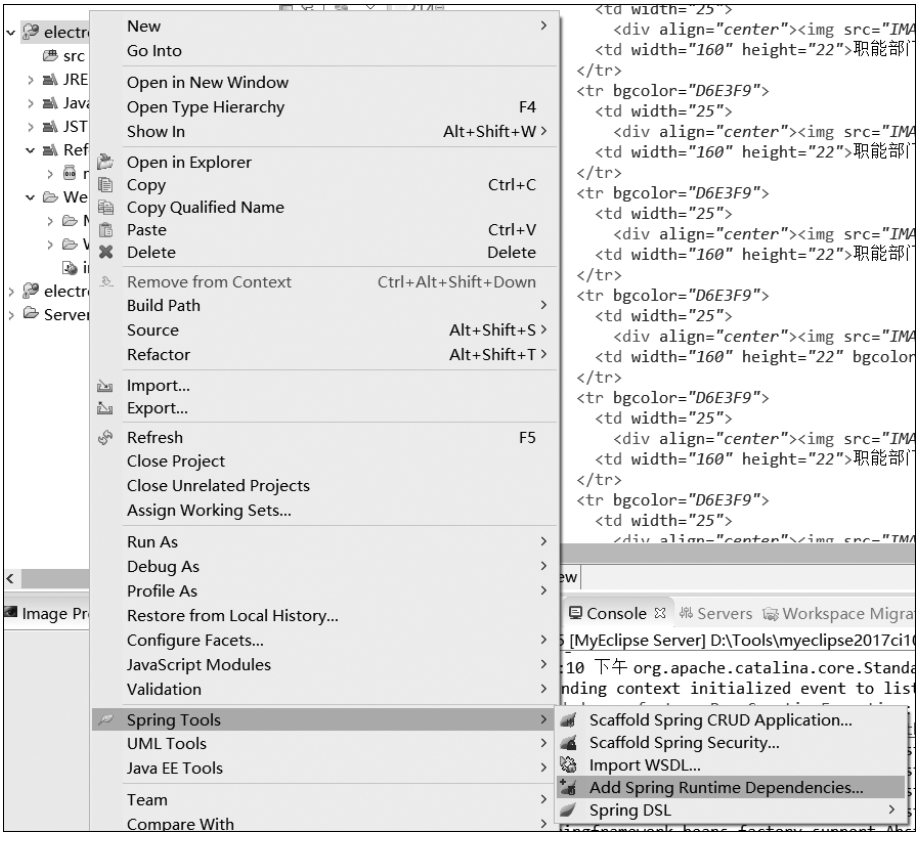

图 3-15 Add Spring Runtime Dependencies 界面 1

单击 Next 按钮，无须修改，Spring Version 为 4.1，再单击 Next 按钮，如图 3-16 所示。

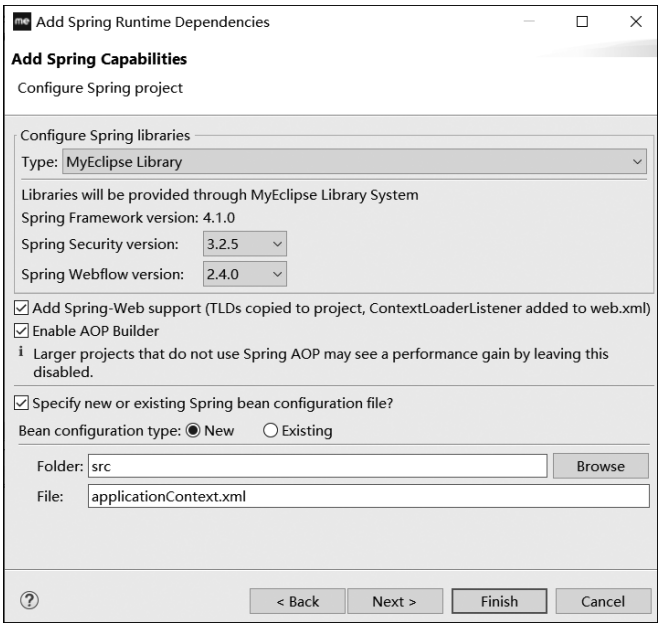

图 3-16 Add Spring Runtime Dependencies 界面 2