Spring Principle

Pionpill [1]

本文档为作者学习 Spring 理论时的笔记。

2022 年 10 月 31 日

[1]笔名：北岸，电子邮件：673486387@qq.com，Github：https://github.com/Pionpill

前言：

笔者为软件工程系在校本科生，有计算机学科理论基础 (操作系统，数据结构，计算机网络，编译原理等)，本人在撰写此笔记时已有 Java 开发经验，基础知识不再赘述。

本书各部分内容及主要参考文献如下:

部分	内容	主要参考
Spring FrameWork	IoC(beans, context), AOP	《Spring 揭秘》：王福强，2009 版
Spring MVC	Servlet, MVC	《看透 Spring MVC：源代码分析与实践》：韩路彪，2015 版

本文重点是 Spring 原理，入门及实战可以参考网上的一些视频教程。

本人的编写及开发环境如下：

- Java: Java11
- SpringBoot: 2.7.3
- OS: Windows11
- MySQL: 8.0.3

2022 年 10 月 31 日

目录

第一部分

Spring FrameWork

I Spring 的 IoC 容器

1 IoC 的基本概念

1.1 Spring 框架总体结构

Spring 本质是始终不变的，都是为了提供各种服务，以帮助我们简化基于 POJO 的 Java 应用程序开发。Spring 框架为 POJO 提供的各种服务共同组成了 Spring 框架的总体结构:

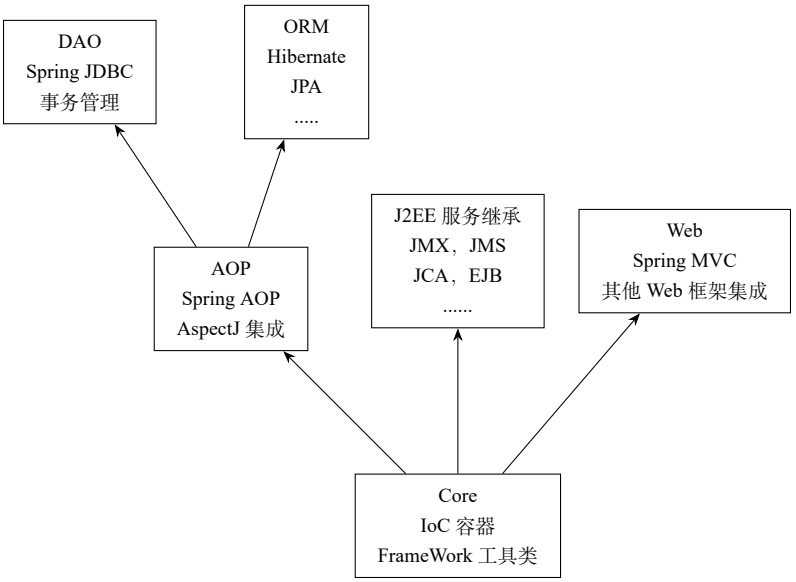


图 1.1 Spring **框架总体结构**

整个框架结构最核心的部分为 IoC 容器与 AOP 框架:

- **Core**: 是整个框架的基础，Spring 为我们提供了一个 IoC 容器，用于帮助我们以**依赖注入**的方式管理对象之间的依赖关系。除此之外，Core 核心模块中还包括框架内部使用的各种工具类。
- **AOP**: 提供了一个轻便但功能强大的 AOP 框架，让我们可以以 AOP 的形式增强各 POJO（Plain Old Java Object，简单 Java 对象）的能力。

其余的 DAO, JDBC 是 AOP 提供的数据库操作，事务管理服务。在 IoC 容器的基础上可以使用各种 J2EE 服务和 Web 框架。

1.2 IoC 理念

IoC 全称 Inversion of Control，中文通常翻译为“控制反转”。IoC 中有一个重要的机制叫依赖注入 DI(Dependency Injection)，依赖注入是 IoC 的具体实现方式。

要理解依赖注入，首选需要理解依赖的含义，看下面一段代码:

```
public class Provider {
    private Listener newsListener = new NewsListener(); // 依赖对象
    private Persister newsPersistener = new NewsPersistener(); // 依赖对象
    public void getAndPersistNews () {
        ...
    }
}
```

其中，Provider 需要依赖 newsListener 来抓取内容，并依赖 newPersistener 存储抓取的内容。这两个类被称为 Provider 的依赖类。

如果我们依赖于某个类或服务，最简单而有效的方式就是直接在类的构造函数中 new 相应的依赖类。这个过程是我们自己主动去获取依赖的对象。

可是回头想想，我们自己每次用到什么依赖对象都要主动地去获取，这是否真的必要？如果有人能够在我们需要时将某个依赖对象送过来，为什么还要大费周折地自己去折腾？

实际上，IoC 就是为了帮助我们避免之前的 “大费周折”，而提供了更加轻松简洁的方式。它的反转，就反转在让你从原来的事必躬亲，转变为现在的享受服务。简单点儿说，IoC 的理念就是，让别人为你服务！

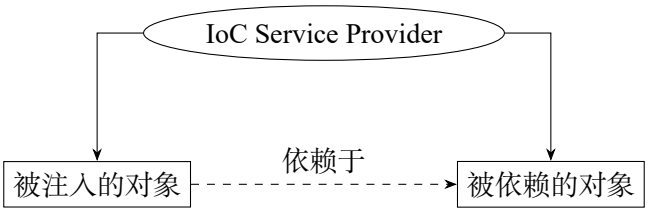


图 1.2 IoC 的角色

通常情况下，被注入对象会直接依赖于被依赖对象。但是，在 IoC 的场景中，二者之间通过 IoC Service Provider 来打交道，所有的被注入对象和依赖对象现在由 IoC Service Provider 统一管理。被注入对象需要什么，直接跟 IoC Service Provider 招呼一声，后者就会把相应的被依赖对象注入到被注入对象中，从而达到 IoC Service Provider 为被注入对象服务的目的。IoC Service Provider 在这里就是通常的 IoC 容器所充当的角色。从被注入对象的角度看，与之前直接寻求依赖对象相比，依赖对象的取得方式发生了反转，控制也从被注入对象转到了 IoC Service Provider 那里。

IoC 容器负责创建的对象，初始化等一系列工作，被创建或被管理的对象在 IoC 容器中统称为 Bean。

1.3 依赖注入的方式

一般的，依赖注入有三种方式:

- 构造方法注入: constructor injection

- setter 方法注入: setter injection
- 接口注入: interface injection

构造方法注入

顾名思义，通过构造方法中声明依赖对象的参数列表，让外部 (通常是 IoC 容器) 知道它需要哪些依赖对象。

```
public FXNewsProvider(IFXNewsListener newsListner,IFXNewsPersister newsPersister) {
    this.newsListener = newsListner;
    this.newPersistener = newsPersister;
}
```

IoC Service Provider 会检查被注入对象的构造方法，取得它所需要的依赖对象列表，进而为其注入相应的对象。同一个对象是不可能被构造两次的，因此，被注入对象的构造乃至其整个生命周期，应该是由 IoC Service Provider 来管理的。

构造方法注入方式比较直观，对象被构造完成后，即进入就绪状态，可以马上使用。

setter 方法注入

对于 JavaBean 对象来说，通常会通过 setXXX() 和 getXXX() 方法来访问对应属性。这些 setXXX() 方法统称为 setter 方法，getXXX() 当然就称为 getter 方法。通过 setter 方法，可以更改相应的对象属性，通过 getter 方法，可以获得相应属性的状态。所以，当前对象只要为其依赖对象所对应的属性添加 setter 方法，就可以通过 setter 方法将相应的依赖对象设置到被注入对象中。

```
public class FXNewsProvider {
    private IFXNewsListener newsListener;
    private IFXNewsPersister newPersistener;
    public IFXNewsListener getNewsListener() {
        return newsListener;
    }
    public void setNewsListener(IFXNewsListener newsListener) {
        this.newsListener = newsListener;
    }
    public IFXNewsPersister getNewPersistener() {
        return newPersistener;
    }
    public void setNewPersistener(IFXNewsPersister newPersistener) {
        this.newPersistener = newPersistener;
    }
}
```

这样，外界就可以通过调用 setNewsListener 和 setNewPersistener 方法为 FXNewsProvider 对象注入所依赖的对象了。

setter 方法注入虽不像构造方法注入那样，让对象构造完成后即可使用，但相对来说更宽

松一些，可以在对象构造完成后再注入。

接口注入

相对于前两种注入方式来说，接口注入没有那么简单明了。被注入对象如果想要 IoC Service Provider 为其注入依赖对象，就必须实现某个接口。这个接口提供一个方法，用来为其注入依赖对象。IoC Service Provider 最终通过这些接口来了解应该为被注入对象注入什么依赖对象。

接口注入的方式比较死板，应用比较少，这里不做过多说明。

三种注入方式的比较

- **接口注入**: 接口注入是现在不甚提倡的一种方式，基本处于 “退役状态”。因为它强制被注入对象实现不必要的接口，带有侵入性。
- **构造方法注入**: 这种注入方式的优点就是，对象在构造完成之后，即已进入就绪状态，可以马上使用。缺点如下:
  - 当依赖对象比较多的时候，构造方法的参数列表会比较长。而通过反射构造对象的时候，对相同类型的参数的处理会比较困难，维护和使用上也比较麻烦。
  - 构造方法无法被继承，无法设置默认值。对于非必须的依赖处理，可能需要引入多个构造方法，而参数数量的变动可能造成维护上的不便。
- **setter 方法注入**: 因为方法可以命名，所以 setter 方法注入在描述性上要比构造方法注入好一些。另外，setter 方法可以被继承，允许设置默认值，而且有良好的 IDE 支持。缺点当然就是对象无法在构造完成后马上进入就绪状态。

2 IoC Service Provider

2.1 IoC Service Provider 简介与职责

虽然业务对象可以通过 IoC 方式声明相应的依赖，但是最终仍然需要通过某种角色或者服务将这些相互依赖的对象绑定到一起，而 IoC Service Provider 就对应 IoC 场景中的这一角色。简单地说，IoC Service Provider 负责实现 IoC。

IoC Service Provider 在这里是一个抽象出来的概念，它可以指代任何将 IoC 场景中的业务对象绑定到一起的实现方式。它可以是一段代码，也可以是一组相关的类，甚至可以是比较通用的 IoC 框架或者 IoC 容器实现。比如，可以通过以下代码绑定与新闻相关的对象。

```
IFXNewsListener newsListener = new DowJonesNewsListener();
IFXNewsPersister newsPersister = new DowJonesNewsPersister();
FXNewsProvider newsProvider = new FXNewsProvider(newsListener,newsPersister);
newsProvider.getAndPersistNews();
```

这段代码就可以认为是这个场景中的 IoC Service Provider，只不过比较简单，而且弊端很多。现在许多开源产品通过各种方式为我们做了这部分工作。所以，目前来看，我们只需要使用这些产品提供的服务就可以了。Spring 的 IoC 容器就是一个提供依赖注入服务的 IoC Service Provider。

IoC Service Provider 的职责主要有两个：业务对象的构建管理和业务对象间的依赖绑定。

- **构建管理**: 在 IoC 场景中，业务对象无需关心所依赖的对象如何构建如何取得，但这部分工作始终需要有人来做。所以，IoC Service Provider 需要将对象的构建逻辑从客户端对象那里剥离出来，以免这部分逻辑污染业务对象的实现。
- **依赖绑定**: 这个职责是最艰巨也是最重要的，这是它的最终使命之所在。IoC Service Provider 通过结合之前构建和管理的所有业务对象，以及各个业务对象间可以识别的依赖关系，将这些对象所依赖的对象注入绑定，从而保证每个业务对象在使用的时候，可以处于就绪状态。

2.2 IoC Service Provider 管理对象间的依赖关系

IoC Service Provider 需要通过各种方式来记录诸多对象之间的对应关系，比如:

- 通过最基本的文本文件来记录被注入对象和其依赖对象之间的对应关系;
- 通过描述性较强的 XML 文件格式来记录对应信息;
- 通过编写代码的方式来注册这些对应信息;
- 通过语音方式来记录对象间的依赖注入关系。

直接编码方式

在容器启动之前，我们就可以通过程序编码的方式将被注入对象和依赖对象注册到容器中，并明确它们相互之间的依赖注入关系。

```
IoContainer container = ...;
container.register(FXNewsProvider.class,new FXNewsProvider());
container.register(IFXNewsListener.class,new DowJonesNewsListener());
...
FXNewsProvider newsProvider = (FXNewsProvider)container.get(FXNewsProvider.class);
newProvider.getAndPersistNews();
```

通过为相应的类指定对应的具体实例，可以告知 IoC 容器，当我们要这种类型的对象实例时，请将容器中注册的、对应的那个具体实例返回给我们。

配置文件方式

这是一种较为普遍的依赖注入关系管理方式。像普通文本文件、properties 文件、XML 文件等，都可以成为管理依赖注入关系的载体。不过，最为常见的，还是通过 XML 文件来管理对象注册和对象间依赖关系。

```
<bean id="newsProvider" class="..FXNewsProvider">
    <property name="newsListener">
        <ref bean="djNewsListener"/>
    </property>
    <property name="newPersistener">
        <ref bean="djNewsPersister"/>
    </property>
</bean>
<bean id="djNewsListener" class="..impl.DowJonesNewsListener">
</bean>
<bean id="djNewsPersister" class="..impl.DowJonesNewsPersister">
</bean>
```

最后，我们就可以通过“newsProvider”这个名字，从容器中取得已经组装好的 FXNewsProvider 并直接使用。

```
...
container.readConfigurationFiles(...);
FXNewsProvider newsProvider = (FXNewsProvider)container.getBean("newsProvider");
newsProvider.getAndPersistNews();
```

元数据方式

我们可以直接在类中使用元数据信息来标注各个对象之间的依赖关系，然后由框架根据这些注解所提供的信息将这些对象组装后，交给客户端对象使用。这是现在主流的方式，也是 SpringBoot 提倡的方式:

```
public class FXNewsProvider {
    private IFXNewsListener newsListener;
    private IFXNewsPersister newPersistener;
    @Inject
    public FXNewsProvider(IFXNewsListener listener,IFXNewsPersister persister) {
        this.newsListener = listener;
        this.newPersistener = persister;
    }
    ...
}
```

通过 @Inject，我们指明需要 IoC Service Provider 通过构造方法注入方式，为 FXNewsProvider 注入其所依赖的对象。至于余下的依赖相关信息，由框架进行处理。

一个例子

装入 IoC 容器

假如没有 Spring，我们编写如下的代码:

```
// 一个 Service 类
public class ServiceImpl implement Service {
    private bookDao = new BookDaoImpl();
    public void save() { ... }
}

// 一个 Dao 类
public class DaoImpl implement Dao {
    public void save() { ... }
}
```

此时我们要调用 Service 层必须这样做:

```
// App 主程序
public class App {
    public static void main(String[] args) {
        Service bookService = new ServiceImpl();
        bookService.save();
    }
}
```

有了 Spring，我们可以将类放入到容器中成为 bean，再利用容器获取这些依赖。首先，为了导入 spring 我们需要在 pom 文件中加入如下依赖 (XML 形式):

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>自己选</version>
</dependency>
```

然后，我们需要在 resources 文件下新建 spring 的 xml 文件，这里命名为 applicationContext.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">
<bean id="dao" class="xxx.impl.DaoImpl"/>
<bean id="service" class="xxx.impl.ServiceImpl"/>
</beans>
```

这样，我们就将容器的信息配置在了一个 XML 文件中，下面我们通过 IoC 容器实现 App:

```
public class App2 {
    public static void main(String[] args) {
        // 获取 IoC 容器
        ApplicationContext ctx = new ClassPathXmlApplication("applicationContext");
        // 通过容器获取 bean
        BookDao bookDao = (BookDao) ctx.getBean("bookDao");
        bookDao.save();
    }
}
```

这样，我们就实现了容器的基本作用: 将对象注入容器，并通过容器获得。

实现控制反转

上一小节实现了 IoC 容器的装载与获取 bean 功能，这节实现 DI 功能。

```
// 取消 new 的依赖方式，改用 setter
public class ServiceImpl implement Service {
    private bookDao;
    public void save() { ... }

    public void setBookDao(BookDao bookDao) {
        this.bookDao = bookDao;
    }
}
```

依赖关系写入 XML 文件中，这样也能运行:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">
<bean id="dao" class="xxx.impl.DaoImpl"/>
<bean id="service" class="xxx.impl.ServiceImpl">
    <!--配置当前的属性，name 表示属性，ref 表示属于的值-->
    <property name = "bookDao", ref = "dao">
</bean>
```

```
</beans>
```

3 BeanFactory

Spring 的 IoC 容器是一个提供 IoC 支持的轻量级容器，除了基本的 IoC 支持，它作为轻量级容器还提供了 IoC 之外的支持。如在 Spring 的 IoC 容器之上，Spring 还提供了相应的 AOP 框架支持、企业级服务集成等服务。

Spring 提供了两种容器类型: BeanFactory 和 ApplicationContext。

- **BeanFactory**: 基础类型 IoC 容器，提供完整的 IoC 服务支持。如果没有特殊指定，默认采用延迟初始化策略（lazy-load）。只有当客户端对象需要访问容器中的某个受管对象的时候，才对该受管对象进行初始化以及依赖注入操作。所以，相对来说，容器启动初期速度较快，所需要的资源有限。对于资源有限，并且功能要求不是很严格的场景，BeanFactory 是比较合适的 IoC 容器选择。
- **ApplicationContext**: ApplicationContext 在 BeanFactory 的基础上构建，是相对比较高级的容器实现。ApplicationContext 所管理的对象，在该类型容器启动之后，默认全部初始化并绑定完成。所以，相对于 BeanFactory 来说，ApplicationContext 要求更多的系统资源，同时，因为在启动时就完成所有初始化，容器启动时间较之 BeanFactory 也会长一些。在那些系统资源充足，并且要求更多功能的场景中，ApplicationContext 类型的容器是比较合适的选择。

作为 Spring 提供的基本的 IoC 容器，BeanFactory 可以完成作为 IoC Service Provider 的所有职责，包括业务对象的注册和对象间依赖关系的绑定。将应用所需的所有业务对象交给 BeanFactory 之后，剩下要做的，就是直接从 BeanFactory 取得最终组装完成并且可用的对象。至于这个最终业务对象如何组装，你不需要关心，BeanFactory 会帮你搞定。

3.1 BeanFactory 作用

对于应用程序的开发来说，不管是否引入 BeanFactory 之类的轻量级容器，应用的设计和开发流程实际上没有太大改变。换句话说，针对系统和业务逻辑，该如何设计和实现当前系统不受是否引入轻量级容器的影响。

之前我们的系统业务对象需要自己去“拉”（Pull）所依赖的业务对象，有了 BeanFactory 之类的 IoC 容器之后，需要依赖什么让 BeanFactory 为我们推过来（Push）就行了。所以，简单点儿说，拥有 BeanFactory 之后，要使用 IoC 模式进行系统业务对象的开发。

FX 新闻系统初期的设计和实现框架代码如下:

```
// 设计FXNewsProvider类用于普遍的新闻处理
public class FXNewsProvider { ... }
// 设计IFXNewsListener接口抽象各个新闻社不同的新闻获取方式，并给出相应实现类
public interface IFXNewsListener { ... }
public class DowJonesNewsListener implements IFXNewsListener { ... }
// 设计IFXNewsPersister接口抽象不同数据访问方式，并实现相应的实现类
public interface IFXNewsPersister { ... }
public class DowJonesNewsPersister implements IFXNewsPersister { ... }
```

BeanFactory 会说，这些让我来干吧。既然使用 IoC 模式开发的业务对象现在不用自己操心如何解决相互之间的依赖关系，那么肯定得找人来做这个工作。通常情况下，BeanFactory 会通过常用的 XML 文件来注册并管理各个业务对象之间的依赖关系。

```
<beans>
    <bean id="djNewsProvider" class="..FXNewsProvider">
        <constructor-arg index="0">
            <ref bean="djNewsListener"/>
        </constructor-arg>
        <constructor-arg index="1">
            <ref bean="djNewsPersister"/>
        </constructor-arg>
    </bean>
    ...
</beans>
```

在 BeanFactory 出现之前，我们通常会直接在应用程序的入口类的 main 方法中，自己实例化相应的对象并调用之，如以下代码所示：

```
FXNewsProvider newsProvider = new FXNewsProvider();
newsProvider.getAndPersistNews();
```

现在既然有了 BeanFactory，我们通常只需将“生产线图纸”交给 BeanFactory，让 BeanFactory 为我们生产一个 FXNewsProvider，如以下代码所示：

```
BeanFactory container = new XmlBeanFactory(new ClassPathResource("配置文件路径"));
FXNewsProvider newsProvider = (FXNewsProvider)container.getBean("djNewsProvider");
newsProvider.getAndPersistNews();
```

版本 3.1. *在早期的 Spring 中，用文件进行 IoC 管理比较多，但现在 (2022 年) 更多的使用 SpringBoot 注解。由于前几章的主要参考书籍《Spring 揭秘》比较远古 (2009)，本人会省略掉很多具体的实现代码，更注重于原理。下文类似。*

3.2 BeanFactory 内幕

BeanFactory 只是一个接口，我们最终需要一个该接口的实现来进行实际的 Bean 的管理，DefaultListableBeanFactory 就是这么一个比较通用的 BeanFactory 实现类。DefaultListableBeanFactory 除了间接地实现了 BeanFactory 接口，还实现了 BeanDefinitionRegistry 接口，该接口才是在 BeanFactory 的实现中担当 Bean 注册管理的角色。

基本上，BeanFactory 接口只定义如何**访问**容器内管理的 Bean 的方法，各个 BeanFactory 的具体实现类负责具体 Bean 的注册以及管理工作。BeanDefinitionRegistry 接口定义抽象了 Bean 的**注册逻辑**。通常情况下，具体的 BeanFactory 实现类会实现这个接口来管理 Bean 的注册。

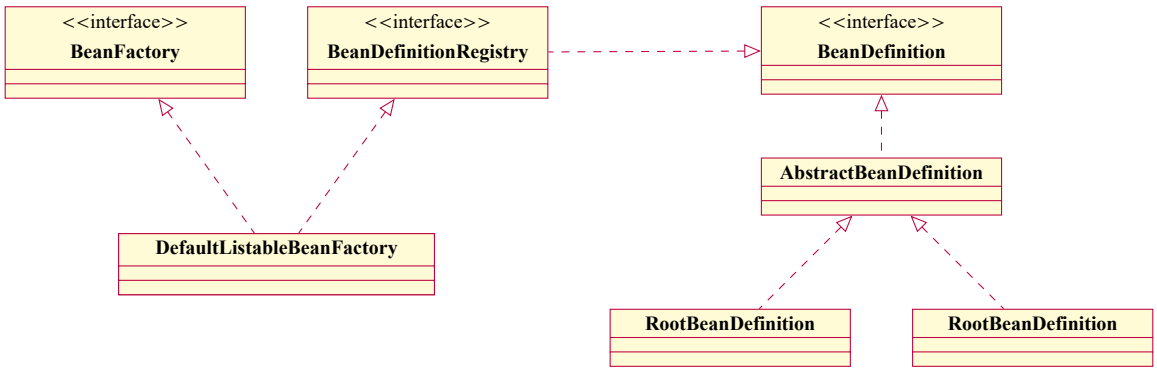


图 3.1 BeanFactory 依赖关系

每一个受管的对象，在容器中都会有一个 BeanDefinition 的实例（instance）与之相对应，该 BeanDefinition 的实例负责保存对象的所有必要信息，包括其对应的对象的 class 类型、是否是抽象类、构造方法参数以及其他属性等。

当客户端向 BeanFactory 请求相应对象的时候，BeanFactory 会通过这些信息为客户端返回一个完备可用的对象实例。RootBeanDefinition 和 ChildBeanDefinition 是 BeanDefinition 的两个主要实现类。

这其中各个接口的作用如下:

- **BeanFactory**: 定义如何访问容器内管理的 Bean 的方法，如果我们查看原码，会发现大部分方法都是 get,is 方法。
- **BeanDefinitionRegistry**: 定义抽象了 Bean 的注册逻辑，即如何注册进容器。
- **BeanDefinition**: 存储具体 Bean 的信息。

BeanFactory

BeanFactory 是定义了如何管理 Bean，它的部分源码如下:

```
public interface BeanFactory {
    Object getBean(String name) throws BeansException;
    <T> ObjectProvider<T> getBeanProvider(Class<T> requiredType);
    boolean containsBean(String name);
    boolean isSingleton(String name) throws NoSuchBeanDefinitionException;
    boolean isPrototype(String name) throws NoSuchBeanDefinitionException;
    boolean isTypeMatch(String name, ResolvableType typeToMatch) throws
        NoSuchBeanDefinitionException;
    Class<?> getType(String name) throws NoSuchBeanDefinitionException;
    String[] getAliases(String name);
```

可以看到，BeanFactory 最主要的只有一个 getBean 方法，其他方法都是对 getBean 返回值的进一步操作。也就是说 BeanFactory 实现了 IoC 的核心功能，获取 Bean。

虽然仅有 getBean 一个主要方法，但该方法非常关键，首次调用 getBean 方法会出发 Bean 实例化阶段的活动，这将开启 Bean 的生命周期。

BeanDefinition

BeanDefinition 存储了 Bean 的具体信息，他的绝大多数方法都是 getXXX 与 setXXX 方法，部分源代码如下:

```
public interface BeanDefinition extends AttributeAccessor, BeanMetadataElement {
    void setBeanClassName(@Nullable String beanClassName);
    String getBeanClassName();
    void setLazyInit(boolean lazyInit);
    boolean isLazyInit();
    ......
}
```

BeanDefinitionRegistry

BeanDefinitionRegistry 定义了 Bean 的注册逻辑，它的部分源代码如下:

```
public interface BeanDefinitionRegistry extends AliasRegistry {
    void registerBeanDefinition(String beanName, BeanDefinition beanDefinition)
      throws BeanDefinitionStoreException;
    void removeBeanDefinition(String beanName) throws NoSuchBeanDefinitionException;
    BeanDefinition getBeanDefinition(String beanName) throws NoSuchBeanDefinitionException;
}
```

3.3 Bean 详解

Spring 的 IoC 容器所起的作用，就像下图所展示的那样，它会以某种方式加载 Configuration Metadata（通常也就是注解），然后根据这些信息绑定整个系统的对象，最终组装成一个可用的基于轻量级容器的应用系统。

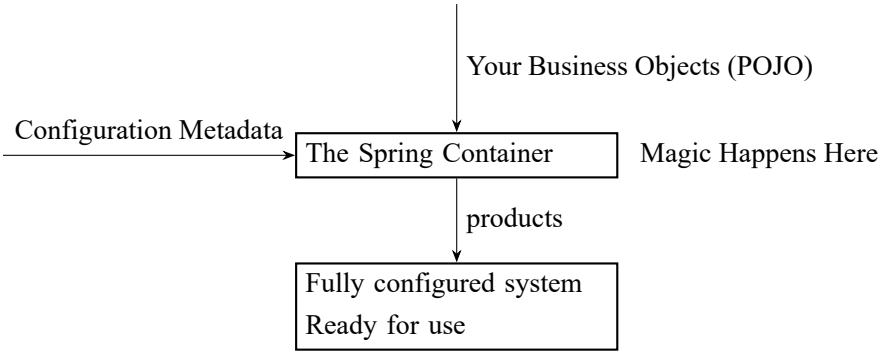


图 3.2 Spring 容器处理过程

本节用来讨论 The Spring Container 里的 Magic Happens。

3.3.1 装配 Bean

Spring 的 IoC 容器所起的作用，以某种方式加载 Configuration Metadata（通常是注解），然后根据这些信息绑定整个系统的对象，最终组装成一个可用的基于轻量级容器的应用系统。

Spring 的 IoC 容器实现以上功能的过程，基本上可以按照类似的流程划分为两个阶段，即容器启动阶段和 Bean 实例化阶段。

容器启动阶段

1. 首先会通过某种途径加载 Configuration MetaData。除了代码方式比较直接，在大部分情况下，容器需要依赖某些工具类（BeanDefinitionReader）对加载的 Configuration MetaData 进行解析和分析。
2. 将分析后的信息编组为相应的 BeanDefinition。
3. 最后把这些保存了 bean 定义必要信息的 BeanDefinition，注册到相应的 BeanDefinitionRegistry。

总地来说，该阶段所做的工作可以认为是准备性的，重点更加侧重于对象管理信息的收集。当然，一些验证性或者辅助性的工作也可以在这个阶段完成。

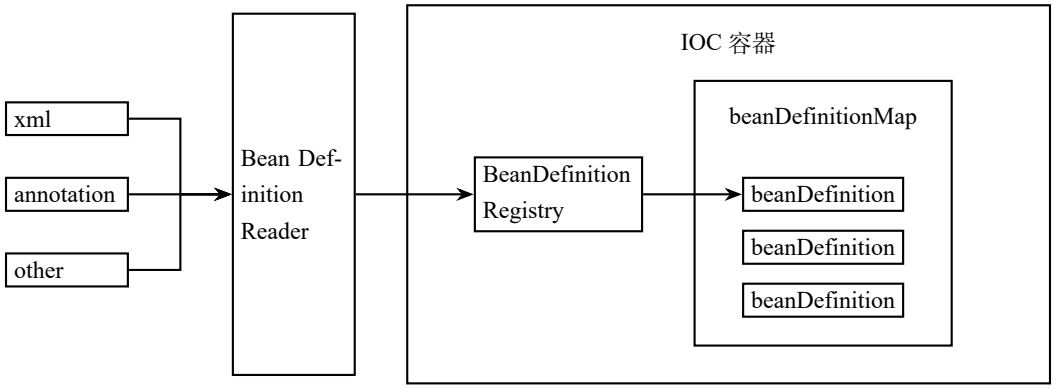


图 3.3 容器启动阶段

Bean 实例化阶段

1. 首先检查所请求的对象之前是否已经初始化。如果没有，则会根据注册的 BeanDefinition 所提供的信息实例化被请求对象，并为其注入依赖。如果该对象实现了某些回调接口，也会根据回调接口的要求来装配它。
2. 当该对象装配完毕之后，容器会立即将其返回请求方使用。

当某个请求方通过容器的 getBean 方法明确地请求某个对象，或者因依赖关系容器需要隐式地调用 getBean 方法时，就会触发第二阶段的活动。

如果说第一阶段只是根据图纸装配生产线的话，那么第二阶段就是使用装配好的生产线来生产具体的产品了。

我们知道，只有当请求方通过 BeanFactory 的 getBean() 方法来请求某个对象实例的时候，才有可能触发 Bean 实例化阶段的活动。BeanFactory 的 getBean 方法可以被客户端对象显式

调用，也可以在容器内部隐式地被调用。隐式调用有如下两种情况:

- **BeanFactory 延迟实例化**: 通常情况下，当对象 A 被请求而需要第一次实例化的时候，如果它所依赖的对象 B 之前同样没有被实例化，那么容器会先实例化对象 A 所依赖的对象。这种情况是容器内部调用 getBean()，对于本次请求的请求方是隐式的。
- **ApplicationContext 实例化所有 bean 定义**: ApplicationContext 在实现的过程中遵循 Spring 容器实现流程的两个阶段，只不过它会在启动阶段的活动完成之后，紧接着调用注册到该容器的所有 bean 定义的实例化方法 getBean()。这就是为什么当你得到 ApplicationContext 类型的容器引用时，容器内所有对象已经被全部实例化完成。

之所以说 getBean() 方法是有可能触发 Bean 实例化阶段的活动，是因为只有当对应某个 bean 定义的 getBean() 方法第一次被调用时，不管是显式的还是隐式的，Bean 实例化阶段的活动才会被触发，第二次被调用则会直接返回容器缓存的第一次实例化完的对象实例（prototype 类型 bean 除外）。当 getBean() 方法内部发现该 bean 定义之前还没有被实例化之后，会通过其他方法来进行具体的对象实例化，也就开启了 Bean 的生命周期。

了解 Bean 生命周期之前，需要知道 Bean 的作用域，作用域用于确定 Spring 创建 Bean 的实例个数:

- Singleton: 单例，在 IoC 容器中仅存在一个 Bean 实例，默认值。
- Prototype: 每次调取 Bean 时们都返回一个新实例，相当于每次 new 一个新的对象。
- Request: 每次 Http 请求都会创建一个新的 Bean，该作用域仅适用于 WebApplicationContext 环境。
- Session: 同一个 Http Session 共享一个 Bean，不同 Session 使用不同 Bean，仅适用于 WebApplicationContext 环境。
- GlobalSession: 一般用于 Portlet 应用环境，该作用域仅适用于 WebApplicationContext 环境。

3.3.2 Bean 生命周期

这小节参考文献:

- 三分恶: `https://blog.csdn.net/sinat_40770656/article/details/123498761`
- 老周聊架构: `https://blog.csdn.net/riemann_/article/details/118500805`

总的来说，Bean 的生命周期分为四个阶段: 实例化 -> 属性赋值 -> 初始化 -> 销毁。

表 1.1 Bean 生命周期

阶段	过程
实例化	实例化 Bean
设置属性	设置对象属性
初始化	检查 Aware 的相关接口并设置相关依赖 BeanPostProcessor 前置处理 是否实现 InitializationBean 接口 是否配置自定义的 init-method BeanPostProcessor 后置处理
-	注册 Destruction 相关回调接口 使用
销毁	是否实现 DisposableBean 接口 是否配置自定义的 destroy-method

1. Bean 实例化与 BeanWrapper

容器在内部实现的时候，采用“策略模式（Strategy Pattern）”来决定采用何种方式初始化 bean 实例。通常，可以通过反射或者 CGLIB 动态字节码生成来初始化相应的 bean 实例或者动态生成其子类。

默认情况下，容器内部采用的是 CglibSubclassingInstantiationStrategy: 通过 CGLIB 的动态字节码生成功能，该策略实现类可以动态生成某个类的子类，进而满足了方法注入所需的对象实例化需求。容器只要根据相应 bean 定义的 BeanDefintion 取得实例化信息，结合 CglibSubclassingIns-tantiationStrategy 以及不同的 bean 定义类型，就可以返回实例化完成的对象实例。

返回方式上有些“点缀”。不是直接返回构造完成的对象实例，而是以 BeanWrapper 对构造完成的对象实例进行包裹，返回相应的 BeanWrapper 实例。

2. 设置对象属性

BeanWrapper 接口通常在 Spring 框架内部使用，它有一个实现类 org.springframework.beans.BeanWrapperImpl。其作用是对某个 bean 进行“包裹”，然后对这个“包裹”的 bean 进行操作，比如设置或者获取 bean 的相应属性值。而在第一步结束后返回 BeanWrapper 实例而不是原先的对象实例，就是为了“设置对象属性”。

3. Aware 接口

当对象实例化完成并且相关属性以及依赖设置完成之后，Spring 容器会检查当前对象实例是否实现了一系列的以 Aware 命名结尾的接口定义。如果是，则将这些 Aware 接口定义中规定的依赖注入给当前对象实例。

- **BeanNameAware**: 将该对象实例的 bean 定义对应的 beanName 设置到当前对象实例。可以理解为 Map 中的 key 为 beanName, value 为 bean 实例。[1]

[1]扩展文献: https://blog.csdn.net/zyn170605/article/details/80339499

- **BeanClassLoaderAware**: 将对应加载当前 bean 的 Classloader 注入当前对象实例。
- **BeanFactoryAware**: BeanFactory 容器会将自身设置到当前对象实例。这样，当前对象实例就拥有了一个 BeanFactory 容器的引用，并且可以对这个容器内允许访问的对象按照需要进行访问。

以上几个 Aware 接口只是针对 BeanFactory 类型的容器而言，对于 ApplicationContext 类型的容器，也存在几个 Aware 相关接口。不过在检测这些接口并设置相关依赖的实现机理上，与以上几个接口处理方式有所不同，使用的是下面将要说到的 BeanPostProcessor 方式。

对于 ApplicationContext 类型容器，容器在这一步还会检查以下几个 Aware 接口并根据接口定义设置相关依赖。

- **ResourceLoaderAware**: 将当前 ApplicationContext 自身设置到对象实例，这样当前对象实例就拥有了其所在 ApplicationContext 容器的一个引用。
- **ApplicationEventPublisherAware**: ApplicationContext 作为一个容器，同时还实现了 ApplicationEventPublisher 接口，将自身注入当前对象。
- **MessageSourceAware**: ApplicationContext 通过 Message- Source 接口提供国际化的信息支持，将自身注入当前对象实例。
- **ApplicationContextAware**: 将自身注入当前对象实例。

4. BeanPostProcessor

> **注意 3.1.** *BeanPostProcessor 的概念容易与 BeanFactoryPostProcessor 的概念混淆。但只要记住 Bean- PostProcessor 是存在于对象实例化阶段，而 BeanFactoryPostProcessor 则是存在于容器启动阶段，这两个概念就比较容易区分了。*

BeanPostProcessor 会处理容器内所有符合条件的实例化后的对象实例。该接口声明了两个方法，分别在两个不同的时机执行:

```
1 public interface BeanPostProcessor {
2     Object postProcessBeforeInitialization(Object bean, String beanName) throws
          BeansException;
3     Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException;
4 }
```

顾名思义，一个是前置处理，一个是后置处理。通常比较常见的使用 BeanPostProcessor 的场景，是处理标记接口实现类，或者为当前对象提供代理实现。ApplicationContext 对应的那些 Aware 接口实际上就是通过 BeanPostProcessor 的方式进行处理的。

当 ApplicationContext 中每个对象的实例化过程走到 BeanPostProcessor 前置处理这一步时，ApplicationContext 容器会检测到之前注册到容器的 ApplicationContextAwareProcessor 这个 BeanPostProcessor 的实现类，然后就会调用其 postProcessBeforeInitialization() 方法，检查并设置 Aware 相关依赖。

5. InitializingBean 和 init-method

InitializingBean 是容器内部广泛使用的一个对象生命周期标识接口，其定义如下:

```
1 public interface InitializingBean {
2     void afterPropertiesSet() throws Exception;
```

```
}
```

其作用在于，在对象实例化过程调用过“BeanPostProcessor 的前置处理”之后，会接着检测当前对象是否实现了 InitializingBean 接口，如果是，则会调用其 afterPropertiesSet() 方法进一步调整对象实例的状态。比如，在有些情况下，某个业务对象实例化完成后，还不能处于可以使用状态。这个时候就可以让该业务对象实现该接口，并在方法 afterPropertiesSet() 中完成对该业务对象的后续处理。

虽然该接口在 Spring 容器内部广泛使用，但如果真的让我们的业务对象实现这个接口，则显得 Spring 容器比较具有侵入性。所以，Spring 还提供了另一种方式来指定自定义的对象初始化操作，那就是通过 init-method[2]。

6. DisposableBean 与 destroy-method

当所有的一切，该设置的设置，该注入的注入，该调用的调用完成之后，容器将检查 singleton 类型的 bean 实例，看其是否实现了 DisposableBean 接口。或者其对应的 bean 是否有 destroy-method 属性指定了自定义的对象销毁方法。如果是，就会为该实例注册一个用于对象销毁的回调（Callback），以便在这些 singleton 类型的对象实例销毁之前，执行销毁逻辑。

与 InitializingBean 和 init-method 用于对象的自定义初始化相对应，DisposableBean 和 destroy-method 为对象提供了执行自定义销毁逻辑的机会。

不过，这些自定义的对象销毁逻辑，在对象实例初始化完成并注册了相关的回调方法之后，并不会马上执行。回调方法注册后，返回的对象实例即处于使用状态，只有该对象实例不再被使用的时候，才会执行相关的自定义销毁逻辑，此时通常也就是 Spring 容器关闭的时候。

但 Spring 容器在关闭之前，不会聪明到自动调用这些回调方法。所以，需要我们告知容器，在哪个时间点来执行对象的自定义销毁方法。

对于 BeanFactory 容器来说: 我们需要在独立应用程序的主程序退出之前，或者其他被认为是合适的情况下（依照应用场景而定），调用 ConfigurableBeanFactory 提供的 destroySingletons() 方法销毁容器中管理的所有 singleton 类型的对象实例。

[2]参考: https://blog.csdn.net/tuzongxun/article/details/53580695

4 ApplicationContext

ApplicationContext 除了拥有 BeanFactory 支持的所有功能之外，还进一步扩展了基本容器的功能，包括 BeanFactoryPostProcessor、BeanPostProcessor 以及其他特殊类型 bean 的自动识别、容器启动后 bean 实例的自动初始化、国际化的信息支持、容器内事件发布等。

4.1 统一资源加载策略

4.1.1 Spring 中的 Resource

Spring 框架内部使用 Resource 接口作为所有资源的抽象和访问接口。

Resource 接口可以根据资源的不同类型，或者资源所处的不同场合，给出相应的具体实现。Spring 框架在这个理念的基础上，提供了一些实现类:

- **ByteArrayResource**: 将字节（byte）数组提供的数据作为一种资源进行封装，如果通过 InputStream 形式访问该类型的资源，该实现会根据字节数组的数据，构造相应的 ByteArrayInputStream 并返回。
- **ClassPathResource**: 该实现从 Java 应用程序的 ClassPath 中加载具体资源并进行封装，可以使用指定的类加载器（ClassLoader）或者给定的类进行资源加载。
- **FileSystemResource**: 对 java.io.File 类型的封装，所以，我们可以以文件或者 URL 的形式对该类型资源进行访问，只要能跟 File 打的交道，基本上跟 FileSystemResource 也可以。
- **UrlResource**: 通过 java.net.URL 进行的具体资源查找定位的实现类，内部委派 URL 进行具体的资源操作。
- **InputStreamResource**: 将给定的 InputStream 视为一种资源的 Resource 实现类，较为少用。

如果以上资源实现还不能满足要求，那么我们可以根据相应场景给出自己的实现，只需要实现 `Resource` 接口，通常继承 `AbstractResource` 即可。

4.1.2 `ResourceLoader`

资源是有了，但如何去查找和定位这些资源，则应该是 `ResourceLoader` 的职责所在。`ResourceLoader` 接口是资源查找定位策略的统一抽象。

```
public interface ResourceLoader {
    String CLASSPATH_URL_PREFIX = ResourceUtils.CLASSPATH_URL_PREFIX;
    Resource getResource(String location);
    ClassLoader getClassLoader();
}
```

其中最主要的就是 `getResource` 方法，通过它，我们就可以根据指定的资源位置，定位到具体的资源实例。

Spring 提供了几个 getResource 的实现类，关系如下:

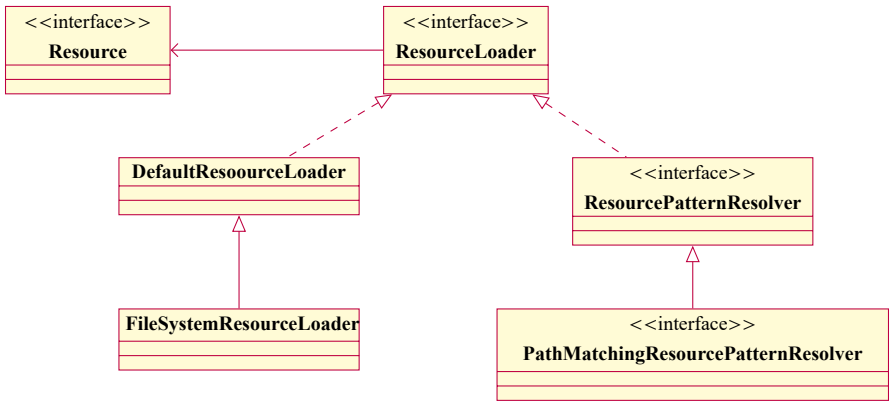


图 4.1 ResourceLoader 类层次图

DefaultResourceLoader

DefaultResourceLoader 顾名思义，是 ResourceLoader 是默认实现类，该类的默认查找处理逻辑如下:

- 检查资源路径是否以 classpath: 前缀打头，如果是，则尝试构造 ClassPathResource 类型资源并返回。
- 否则，(a) 尝试通过 URL，根据资源路径来定位资源，如果没有抛出 MalformedURLException，有则会构造 UrlResource 类型的资源并返回；(b) 如果还是无法根据资源路径定位指定的资源，则委派 getResourceByPath(String) 方法来定位，DefaultResourceLoader 的 getResourceByPath(String) 方法默认实现逻辑是，构造 ClassPathResource 类型的资源并返回。

FileSystemResourceLoader 覆写了 getResourceByPath 方法，使之从文件系统加载资源并以 FileSystemResource 类型返回。这样，我们就可以取得预想的资源类型。

ResourcePatternResolver

ResourceLoader 每次只能根据资源路径返回确定的单个 Resource 实例，而 ResourcePatternResolver 则可以根据指定的资源路径匹配模式，每次返回多个 Resource 实例。

ResourceLoader 最常用的实现是 PathMatchingResourcePatternResolver。

4.1.3 ApplicationContext 与 ResourceLoader

ApplicationContext 继承了 ResourcePatternResolver 下面是其继承逻辑，有了前面的铺垫，结合下图可以很清楚地对 ApplicationContext 有一个更完善的认知。

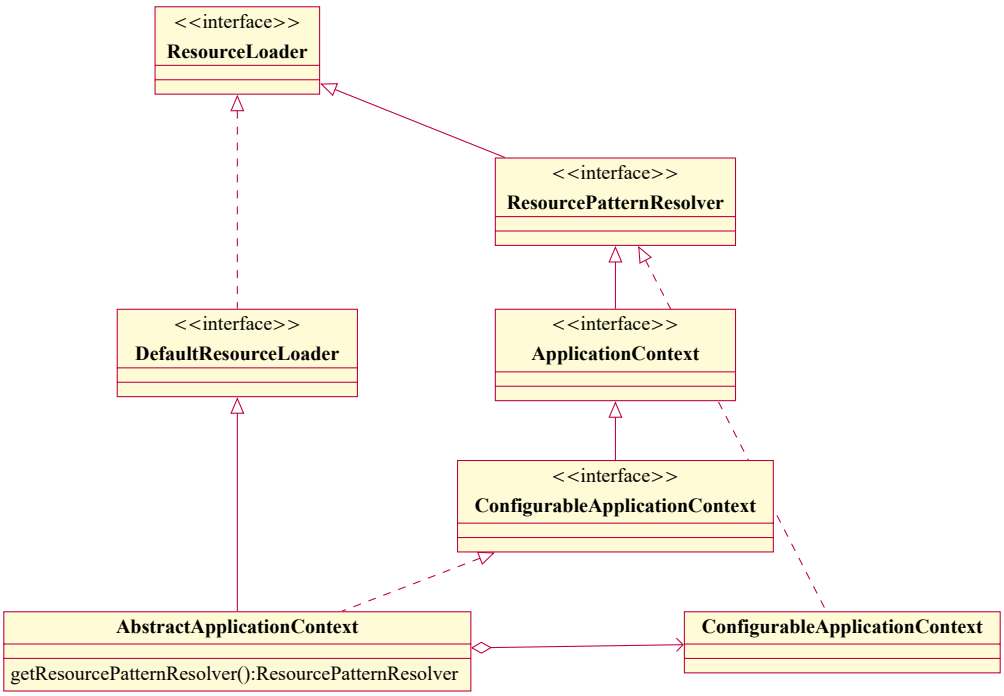


图 4.2 ApplicationContext 继承关系

4.2 容器内部事件发布

Spring 的 ApplicationContext 容器提供的容器内事件发布功能，是通过提供一套基于 Java SE 标准自定义事件类而实现的。为了更好地了解这组自定义事件类，我们可以先从 Java SE 的标准自定义事件类实现的推荐流程说起。

4.2.1 自定义事件发布

Java SE 提供了实现自定义事件发布（Custom Event publication）功能的基础类，即 java.util.EventObject 类和 java.util.EventListener 接口。所有的自定义事件类型可以通过扩展 EventObject 来实现，而事件的监听器则扩展自 EventListener。

我们可以通过扩展 EventObject 类来实现自定义的事件类型:

```
public class MethodExecutionEvent extends EventObject {
    private static final long serialVersionUID = -71960369269303337L;
    private String methodName;
    public MethodExecutionEvent(Object source) {
        super(source);
    }
    public MethodExecutionEvent(Object source,String methodName) {
        super(source);
        this.methodName = methodName;
    }
    public String getMethodName() {
        return methodName;
    }
```

```
    public void setMethodName(String methodName) {
        this.methodName = methodName;
    }
}
```

当该类型的事件发布之后，相应的监听器即可对该类型的事件进行处理。如果需要，自定义事件类可以根据情况提供更多信息，不用担心自定义事件类的“承受力”。

实现针对自定义事件类的事件监听器接口。自定义的事件监听器需要在合适的时机监听自定义的事件，我们可以在方法开始执行的时候发布该事件，也可以在方法执行即将结束之际发布该事件。

```
public interface MethodExecutionEventListener extends EventListener {
    // 处理方法开始执行的时候发布的MethodExecutionEvent事件
    void onMethodBegin(MethodExecutionEvent evt);
    // 处理方法执行将结束时候发布的MethodExecutionEvent事件
    void onMethodEnd(MethodExecutionEvent evt);
}
```

组合事件类和监听器，发布事件。有了自定义事件和自定义事件监听器，剩下的就是发布事件，然后让相应的监听器监听并处理事件了。通常情况下，我们会有一个事件发布者（EventPublisher），它本身作为事件源，会在合适的时间点，将相应事件发布给对应的事件监听器。

```
public class MethodExeuctionEventPublisher {
    private List<MethodExecutionEventListener> listeners = new
        ArrayList<MethodExecutionEventListener>();

    public void methodToMonitor() {
        MethodExecutionEvent event2Publish = new MethodExecutionEvent(this,"methodToMonitor");
        publishEvent(MethodExecutionStatus.BEGIN,event2Publish);
        //  执行实际的方法逻辑
        //  ...
        publishEvent(MethodExecutionStatus.END,event2Publish);
    }

    protected void publishEvent(MethodExecutionStatus status, MethodExecutionEvent
        methodExecutionEvent) {
        List<MethodExecutionEventListener> copyListeners = new
            ArrayList<MethodExecutionEventListener>(listeners);
        for(MethodExecutionEventListener listener:copyListeners) {
            if(MethodExecutionStatus.BEGIN.equals(status))
                listener.onMethodBegin(methodExecutionEvent);
            else
                listener.onMethodEnd(methodExecutionEvent);
        }
    }

    ......

    public static void main(String[] args) {
        MethodExeuctionEventPublisher eventPublisher = new MethodExeuctionEventPublisher();
        eventPublisher.addMethodExecutionEventListener(new
```

```
            SimpleMethodExecutionEventListener());
        eventPublisher.methodToMonitor();
    }
}
```

4.2.2 Spring 的容器内部事件发布类结构分析

Spring 的 ApplicationContext 容器内部允许以 ApplicationEvent 的形式发布事件，容器内注册的 ApplicationListener 类型的 bean 定义会被 ApplicationContext 容器自动识别，它们负责监听容器内发布的所有 ApplicationEvent 类型的事件。也就是说，一旦容器内发布 ApplicationEvent 及其子类型的事件，注册到容器的 ApplicationListener 就会对这些事件进行处理。

- ApplicationEvent: Spring 容器内自定义事件类型，继承自 EventObject，它是一个抽象类，需要根据情况提供相应子类以区分不同情况。默认情况下，Spring 提供了三个实现。
  - ContextClosedEvent: 容器在即将关闭的时候发布的事件类型。
  - ContextRefreshedEvent: 容器在初始化或者刷新的时候发布的事件类型。
  - RequestHandledEvent: Web 请求处理后发布的事件。
- ApplicationListener: 继承自 Eve ntListener。容器启动时，会自动识别并加载 EventListener 类型 bean 定义，一旦容器内有事件发布，将通知这些注册到容器的 EventListener。
- ApplicationContext: 它继承了 ApplicationEventPublisher 接口，担当的就是事件发布者的角色。

ApplicationContext 内部实现比较复杂,这里单独讲一下。ApplicationContext 容器的具体实现类在实现事件的发布和事件监听器的注册方面,并没有亲自处理,而是把这些活儿转包给了一个称作 ApplicationEventMulticaster 的接口。该接口定义了具体事件监听器的注册管理以及事件发布的方法。他有一个抽象子类: AbstractApplicationEventMulyicaster, Spring 提供了一个子类实现: SimpleApplicationEventMulticaster。

因为 ApplicationContext 容器的事件发布功能全部委托给了 ApplicationEventMulticaster 来做,所以,容器启动伊始,就会检查容器内是否存在名称为 applicationEventMulticaster 的 ApplicationEventMulticaster 对象实例。有的话就使用提供的实现，没有则默认初始化一个 SimpleApplicationEventMulticaster 作为将会使用的 ApplicationEventMulticaster 。

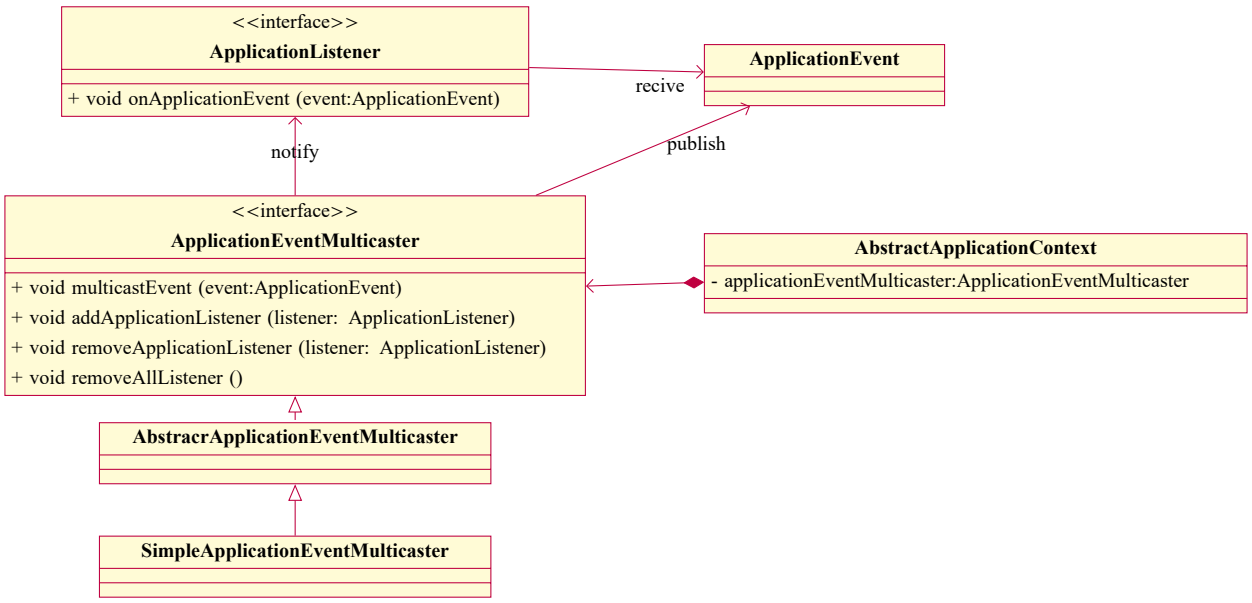


图 4.3 Spring 容器内事件发布实现类图

4.2.3 Spring 容器内事件发布的应用

Spring 的 ApplicationContext 容器内的事件发布机制，主要用于单一容器内的简单消息通知和处理，并不适合分布式、多进程、多容器之间的事件通知。我们应该在合适的地点、合适的需求分析的前提下，合理地使用 Spring 提供的 ApplicationContext 容器内的事件发布机制。

要让我们的业务类支持容器内的事件发布，需要它拥有 ApplicationEventPublisher 的事件发布支持。所以，需要为其注入 ApplicationEventPublisher 实例。可以通过如下两种方式为我们的业务对象注入 ApplicationEventPublisher 的依赖。

- ApplicationEventPublisherAware: 在 ApplicationContext 类型的容器启动时，会自动识别该类型的 bean 定义并将 ApplicationContext 容器本身作为 ApplicationEventPublisher 注入当前对象，而 ApplicationContext 容器本身就是一个 ApplicationEventPublisher。
- ApplicationContextAware: 既然 ApplicationContext 本身就是一个 ApplicationEventPublisher，那么通过 ApplicationContextAware 几乎达到第一种方式相同的效果。

II Spring 的 AOP 框架

5 Spring AOP 基础

AOP 全称 Aspect-Oriented Programming，即面向切面编程，AOP 是基于 OOP 的。Spring AOP 是 Spring 核心框架的重要组成部分，通常认为它与 Spring 的 IoC 容器以及 Spring 框架的其他 JavaEE 服务的集成共同组成了 Spring 框架的质量三角。

5.1 Spring AOP 概述

AOP 可以在不惊动原始设计的基础上为其进行功能增强。

日志记录、安全检查、事务管理等系统需求就像一把把刀横切到我们组织良好的各个业务功能模块之上，我面的所有功能模块都必须经过这几个系统。以 AOP 的行话来说，这些系统需求是系统中的横切关注点 (cross-cuting concern)。

举个例子，我对某一块代码运行速度进行检测:

```
public void save() {
    Long startTime = System.currentTimeMillis();
    for (int i=0; i<10000; i++) {
        System.out.println("book dao save...");
    }
    Long endTime = System.currentTimeMillis();
    Long totalTime = endTime - startTime;
    System.out.println("万次耗时: " + totalTime + "ms");
}
public void update() {
    System.out.println("book dao update...");
}
public void delete() {
    System.out.println("book dao delete...");
}
public void select() {
    System.out.println("book dao select...");
}
```

实际上上，只有 for 循环中的那一句语句是需要重新设计的，其他语句完全可以复用。我们可以将这些可复用的部分抽取出来。

```
public void method() {
    Long startTime = System.currentTimeMillis();
    for (int i=0; i<10000; i++) {
        // 原始操作
    }
```

```
    Long endTime = System.currentTimeMillis();
    Long totalTime = endTime - startTime;
    System.out.println("万次耗时: " + totalTime + "ms");
}
```

其中，这些被抽取出来的原始方法被称为**连接点** (JoinPoint)。对于要追加功能的方法，我们称其为**切入点** (Pointcut)。切入点是连接点，但连接点不一定是切入点，只有需要追加功能的连接点，才是切入点。

可复用的共性功能被称为**通知** (advice)。通知所在的类被称为**通知类**。通知和切入点需要进行绑定，好让切入点进入通知执行，通知与切入点的绑定关系称为 **切面 (Aspect)**。通俗一点说，切面就是让切入点知道他去哪里执行，产生一个一一对应关系。

5.2 Spring AOP 的实现方式

Spring AOP 属于第二代 AOP[1]，采用动态代理机制和字节码生成技术实现。

5.2.1 动态代理

SpringAOP 本质上就是采用代理机制实现的，但一般的静态代理模式有一定的缺陷，往往需要手动设置很多个对象。为了解决这个问题，Spring AOP 使用了动态代理。

JDK1.3 之后引入了一种称之为动态代理 (Dynamic Proxy) 的机制。使用该机制，我们可以为指定的接口在系统运行期间动态地生成代理对象。

代理模式的实现机制如下:

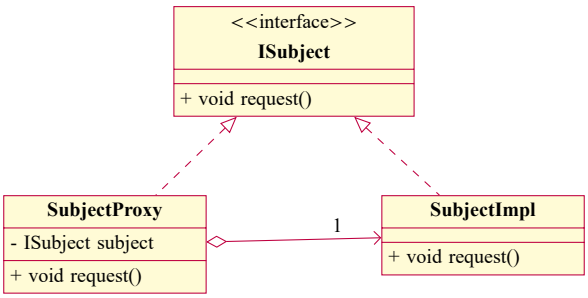


图 5.1 代理模式

代理类在程序运行时创建的代理方式被成为动态代理。在静态代理中，代理类（Renter-Proxy）是自己已经定义好了的，在程序运行之前就已经编译完成。而动态代理是在运行时根据我们在 Java 代码中的“指示”动态生成的。动态代理相较于静态代理的优势在于可以很方便的对代理类的所有函数进行统一管理，如果我们想在每个代理方法前都加一个方法，如果代理方法很多，我们需要在每个代理方法都要写一遍，很麻烦。而动态代理则不需要。[2]

Java 实现动态代理需要依靠 `reflect` 包下提供的 `Proxy` 类与 `InvocationHandler` 接口。

[1]有兴趣自己查一下 AOP 历史，本文只讲 Spring AOP

[2]参考: https://blog.csdn.net/qq_34609889/article/details/85317582

首先我们需要定义接口与对应的实现类:

```
public interface Person {
    public void rentHouse();
}

public class Renter implements Person {
    @Override
    public void rentHouse() {
        System.out.println("租客租房成功！");
    }
}
```

然后我们需要定义一个实现了 InvocationHandler 接口的类并持有一个被代理的对象，InvocationHandler 中有一个 invoke 方法，所有执行代理对象的方法都会被替换成该方法。然后通过反射在 invoke 方法中执行代理类的方法。在代理过程中，在执行代理类的方法前或后可以执行自己的操作，这就是 Spring AOP 动态代理的主要原理。

```
public class RenterInvocationHandler<T> implements InvocationHandler {
    private T target;

    public RenterInvocationHandler(T target) {
        this.target = target;
    }

    @Override
    public Object invoke (Object proxy, Method method, Object[] args) throws Throwable {
        // 具体的代理方法，这里只是给出一种示例
        System.out.println("租客和中介交流");
        Object result = method.invoke(this.target, args);
        return result;
    }
}
```

最后我们通过 Proxy 创建动态代理并运行:

```
public static void main(String[] args) {
    Person renter = new Renter();
    InvocationHandler renterHandler = new RenterInvocationHandler<Person>(renter);
    Person rentProxy = (Person) Proxy.newProxyInstance(Person.class.getClassLoader(),
            new Class<?>[]{Person.class}, renterHandler);
    rentProxy.rentHouse();
}
```

深入了解我们会发现，java 自动生成了一个前缀为 $Proxy 的代理类，这个类通过反射获取构造方法，然后创建代理类实例对象。分析源代码我们会发合适呢个调用 rentHouse 方法时的大概流程: 调用 RentInvocationHandler 类的 invoke 方法，invoke 方法又用过反射调用被代理类的 rentHouse 方法。

5.2.2 动态字节码

使用动态字节码生成技术扩展对象行为的原理是: 可以对目标对象进行继承扩展，为其生成相应的子类，而子类可以通过覆写来扩展父类的行为，只要将横切逻辑的实现放到子类中，然后让系统使用扩展后的目标对象的子类，就可以达到与代理模式相同的效果。

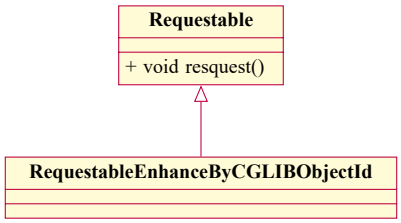


图 5.2 CGLIB 的继承扩展

使用继承方式来扩展对象定义，也不能像静态代理模式那样，为每个不同类型的目标对象都单独创建相应的扩展子类。所以，我们要借助 CGLIB 这样的动态字节码生成库，在系统运行期间动态地为目标对象生成相应的扩展子类。

动态字节码可对实现了某种接口的类，或者没有实现任何接口的类进行扩展。

6 Spring AOP 一世

在动态代理和 CGLIB 的支持下，SpringAOP 框架的实现经过了两代。从 Spring 的 AOP 框架第一次发布，到 Spring2.0 发布之前的 AOP 实现，是 Spring 第一代 AOP 实现。Spring2.0 发布后的 AOP 实现是第二代。

不过划分归划分，SpringAOP 的底层实现机制却一直没变。唯一改变的，是各种 AOP 概念实体的表现形式以及 Spring AOP 的使用方式。

6.1 Spring AOP 中的 Joinpoint

AOP 的 JoinPoint 可以有许多种类型，如构造方法调用，字段的设置及获取，方法调用，方法执行等。但在 SpringAOP 中，仅支持方法级别的 Joinpoint。但在实际的开发过程中，这已经可以满足 80% 的需求了。

Sring AOP 这样做的原因有以下几点:

- Spring AOP 提供一个强大而简单的 AOP 框架，但不会因为追求强大而变得臃肿。
- 对于类中属性 (Field) 级别的 Joinpoint，如果提供这个级别的拦截支持，那么久破坏了面向对象的封装，而且完全可以通过 setter 和 getter 方法的拦截达到同样的目的。
- 如果应用要求非常特殊，不妨求助于其他 AOP 产品。如 AspectJ。

6.2 Spring AOP 中的 PointCut

Spring 中以接口定义 `Pointcut` 作为其 AOP 框架中所有 Pointcut 的最顶层抽象，该接口定义了两个方法用来帮助捕捉系统中的相应 Joinpoint，并提供了一个 `TruePointcut` 类型实例。

如果 `Pointcut` 类型为 `TruePointcut` ，默认会对系统中的所有对象，以及对象上所有被支持的 Joinpoint 进行匹配。

`Pointcut` 接口定义如下所示:

```
public interface Pointcut {
    ClassFilter getClassFilter();
    MethodMatcher getMethodMatcher();
    Pointcut TRUE = TruePointcut.INSTANCE;
}
```

其中 `ClassFilter` 和 `MethodMatcher` 分别用于匹配将执行织入操作的对象以及相应的方法。之所以将这两个分开，是为了重用不同级别的匹配定义。并且可以在不同的级别或者相同的级别上进行组合操作，或者强制让某个子类覆写相应的方法定义等。

`ClassFilter` 接口的作用是对 `Joinpoint` 所处的对象进行 `Class` 级别的类型匹配，定义如下:

```java
public interface ClassFilter {
    boolean matches(Class clazz);
    ClassFilter TRUE = TrueClassFilter.INSTANCE;
}
```

当织入的目标对象的 Class 类型与 Pointcut 所规定的类型相符时，matches 方法将会返回 true，否则，返回 false，即意味着不会对这个类型的目标对象进行织入操作。比如，如果我们仅希望对系统中 Foo 类型的类执行织入，则可以如下这样定义 ClassFilter：

```java
public boolean matches(Class clazz) {
    return Foo.class.equals(clazz);
}
```

相对于 `ClassFilter` 的简单定义，`MethodMatcher` 则要复杂得多。毕竟，Spring 主要支持的就是方法级别的拦截——“重头戏”可不能单薄啊！`MethodMatcher` 定义如下：

```java
public interface MethodMatcher {
    boolean matches(Method method, Class targetClass);
    boolean isRuntime();
    boolean matches(Method method, Class targetClass, Object[] args);
    MethodMatcher TRUE = TrueMethodMathcer.INSTANCE;
}
```

`MethodMatcher` 定义了两个 `matches` 方法，而这两个方法的分界线就是 `isRuntime` 方法。在对对象具体方法进行拦截的时候，可以忽略每次方法执行的时候调用者传入的参数，也可以每次都检查这些方法调用参数，以强化拦截条件。假设对以下方法进行拦截:

```java
public boolean login(String username, String password);
```

如果只想在 login 方法之前插入计数功能，那么 login 方法的参数对于 Joinpoint 捕捉就是可以忽略的。而如果想在用户登录的时候对某个用户做单独处理，如不让其登录或者给予特殊权限，那么这个方法的参数就是在匹配 Joinpoint 的时候必须要考虑的。

- 前一种情况下，`isRuntime` 返回 `false`，表示不会考虑具体 `Joinpoint` 的方法参数，这种类型的 `MethodMatcher` 称之为 `StaticMethodMatcher` 。

  因为不用每次都检查参数，那么对于同样类型的方法匹配结果，就可以在框架内部缓存以提高性能。`isRuntime` 方法返回 `false` 表明当前的 `MethodMatcher` 为 `StaticMethodMatcher` 的时候，只有 `boolean matches(Method method, Class targetClass);` 方法将被执行，它的匹配结果将会成为其所属的 `Pointcut` 主要依据。
- 当 `isRuntime` 方法返回 `true` 时，表明该 `MethodMatcher` 将会每次都对方法调用的参数进行匹配检查，这种类型的 `MethodMatcher` 称之为 `DynamicMethodMatcher` 。

  因为每次都要对方法参数进行检查，无法对匹配的结果进行缓存，所以，匹配效率相对于 `staticMethodMatcher` 来说要差。而且大部分情况下，`staticMethodMatcher` 已经可以满足需要，最好避免使用 `DynamicMethodMatcher` 类型。

6.3 Spring AOP 中的 Advice

Spring 中各种 Advice 类型实现与 AOPAlliance 中标准接口之间的关系如下图:

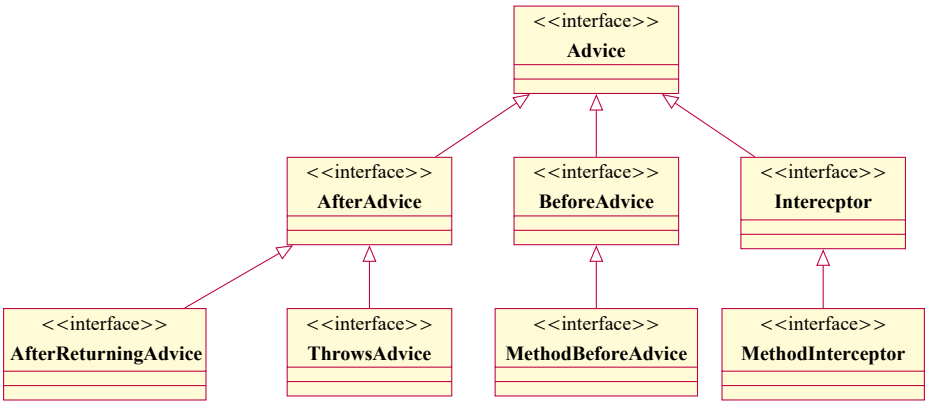


图 6.1 Spring 中的 Advice 缩略图

Advice 实现了被织入到 Pointcut 规定的 Joinpoint 处的横切逻辑。

在 Spring 中，Advice 按照其自身实例能否在目标对象类的所有实例中共享这一标准，可以划分为两大类，即 `per-class` 类型的 Advice 和 `per-instance` 类型的 Advice。

per-class 类型的 Advice 该类型的 Advice 的实例可以在目标对象类的所有实例之间共享。这种类型的 Advice 通常只是提供方法拦截的功能，不会为目标对象类保存任何状态或者添加新的特性。上图中没有列出的 `Introduction` 类型的 Advice 不属于 `perclass` 类型的 Advice 之外，图中的所有 Advice 均属此列。

per-instance 类型的 Advice 该类型的 Advice 不会在目标类所有对象实例之间共享，而是会为不同的实例对象保存它们各自的状态以及相关逻辑。在 SpringAOP 中，`Introduction` 就是唯一的一种 `per-instance` 型 Advice。

6.4 Spring AOP 中的 Aspect

Advisor 代表 Spring 中的 Aspect，但是，与正常的 Aspect 不同，Advisor 通常只持有一个 Pointcut 和一个 Advice。而理论上，Aspect 定义中可以有多个 Pointcut 和多个 Advice，所以，我们可以认为 Advisor 是一种特殊的 Aspect。

我们可以将 Advisor 简单划分为两个分支，一个分支以 `PointcutAdvisor` 为首，另一个分支则以 `IntroductionAdvisor` 为头儿。

系统中只存在单一的横切关注点的情况很少，大多数时候，都会有多个横切关注点需要处理，那么，系统实现中就会有多个 Advisor 存在。当其中的某些 Advisor 的 Pointcut 匹配了同一个 Joinpoint 的时候，就会在这同一个 Joinpoint 处执行多个 Advice 的横切逻辑。

如果这些 Advisor 所关联的 Advice 之间没有很强的优先级依赖关系，那么谁先执行，谁后执行都不会造成任何影响。而一旦这几个需要在同一 Joinpoint 处执行的 Advice 逻辑存在优先顺序依赖的话，就需要我们来干预了，否则，系统的行为就会偏离我们的预想。

Spring 在处理同一 Joinpoint 处的多个 Advisor 的时候，实际上会按照指定的顺序和优先级来执行它们，顺序号决定优先级，顺序号越小，优先级越高，优先级排在前面的，将被优先执行。我们可以从 0 或者 1 开始指定，因为小于 0 的顺序号原则上由 SpringAOP 框架内部使用。默认情况下，如果我们不明确指定各个 Advisor 的执行顺序，那么 Spring 会按照它们的声明顺序来应用它们，最先声明的顺序号最小但优先级最大，其次次之。

6.5 Spring AOP 织入

Spring AOP 使用 `ProxyFactory` 作为织入器。`ProxyFactory` 并非 SpringAOP 中唯一可用的织入器，而是最基本的一个织入器实现。`ProxyFactory` 织入非常简单:

```
ProxyFactory weaver = new ProxyFactory(yourTargetObject);
// 或者
// ProxyFactory weaver = new ProxyFactory();
// weaver.setTarget(task);

Advisor advisor = ...;
weaver.addAdvisor(advisor);
Object proxyObject = weaver.getProxy();
```

第二部分

Spring MVC

III Spring MVC 核心组件

7 Servlet

7.1 MVC Servlet 简介

Spring MVC 是 Spring Framework 提供的 Web 组件，全称是 Spring Web MVC, 是目前主流的实现 MVC 设计模式的框架，提供前端路由映射、视图解析等功能[1]。

MVC 是一种软件架构思想，把软件按照模型，视图，控制器来划分：

- 模型 (Model): 指工程中的 JavaBean，用来处理数据。JavaBean 分成两类：
  - 实体类 Bean: 专门用来存储业务数据，比如 Student,User。
  - 业务处理 Bean: 指 Servlet 或 Dao 对象，专门用来处理业务逻和数据访问。
- 视图 (View): 指工程中的 html,jsp 等页面，作用是和用户进行交互，展示数据
- 控制 (Controller): 指工程中的 Servlet, 作用是接收请求和响应浏览器

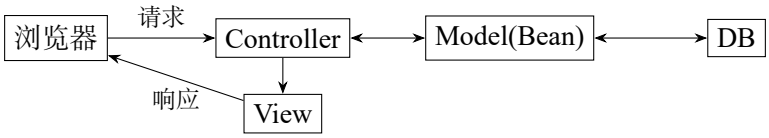


图 7.1 MVC 设计模式

Spring MVC 包含以下核心组件:

- DispatcherServlet: 前端控制器，负责调度其他组件的执行，可以降低不同组件之间的耦合性，是整个 Spring MVC 的核心模块
- Handler: 处理器，完成具体的业务逻辑，相当于 Servlet。
- HandlerMapping: DispatcherServlet 通过它把请求映射到不同的 Handler。
- HandlerInterceptor: 处理拦截器，用于进行一些拦截处理，是一个接口。
- HandlerExecutionChain: 处理器执行链, 包括两部分内容:Handler 和 HandlerInterceptor。
- HandlerAdapter: 处理器适配器，Handler 执行业务方法之前，需要进行一系列的操作。
- ModelAndView: 封装了模型数据和视图信息，作为 Handler 的处理结果，返回给 DispatcherServlet。
- ViewResolver: 视图解析器,DispatcherServlet 通过它把逻辑视图解析为物理视图，最终把渲染的结果响应给客户端。

[1]这节参考文献: https://blog.csdn.net/qq_52797170/article/details/125591705

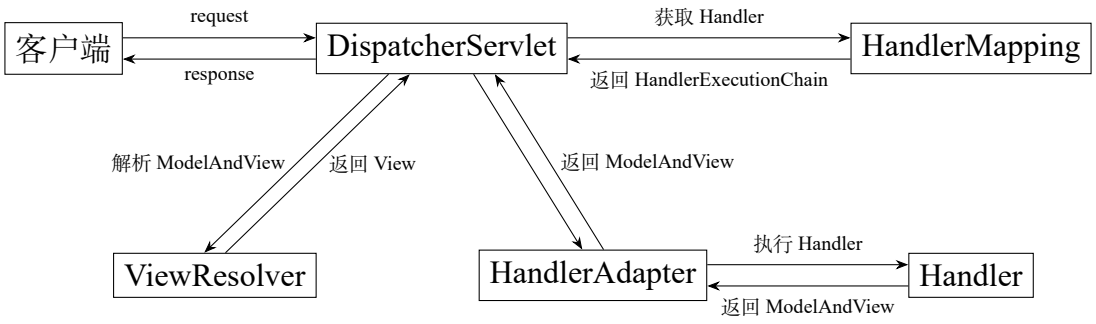


图 7.2 MVC 工作流程

假设有下面代码，分析其工作流程:

```
@Controller
public class HelloHandler {
    @RequestMapping("/index")
    public String index() {
        System.out.println("接收到 index 请求");
        return "index";
    }
}
```

此时 SpringMVC 的处理流程如下:

- 客户端尝试获取 “/index” 页面时
- DispatcherServlet 接收到 URL 请求 index，尝试通过 HandlerMapping 获取对应的 Handler
- 结合 @RequestMapping，HandlerMapping 返回 HelloHandler。
- 调用 HelloHandler 中对应的方法，返回 “index” 字符串。
- ViewResolver 解析 “index” 字符串，找到目标资源: /index.html(也可能是其他类型的资源，如 jsp) 并将该资源返回。

这段代码实际中的响应过程如下:

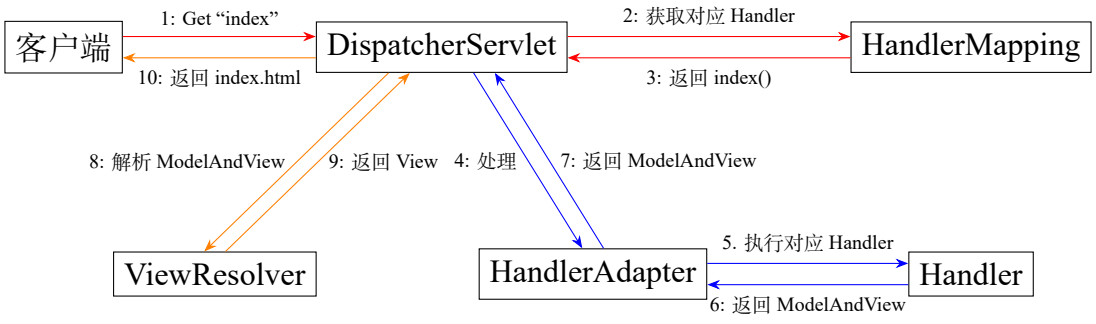
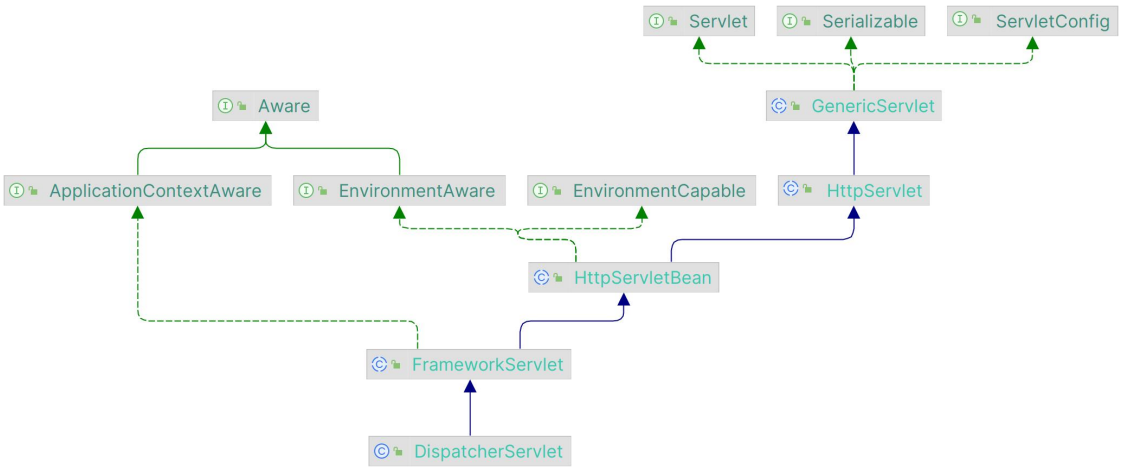


图 7.3 例子工作流程

7.2 MVC Servlet 整体结构

DispatcherServlet 的继承结构如下，其中 HttpServlet 为 Java 实现，其他为 Spring 内容。



7.2.1 HttpServlet

Servlet 是 Server + Applet 的缩写，表示一个服务器应用。Java 中最基础的 Servlet 接口有如下规范:

```
public interface Servlet {
    void init(ServletConfig var1) throws ServletException;
    ServletConfig getServletConfig();
    void service(ServletRequest var1, ServletResponse var2) throws ServletException,
        IOException;
    String getServletInfo();
    void destroy();
}
```

其中，init 与 destroy 分别在容器启动和销毁时调用，这两个方法都只会被调用一次。getServletConfig 方法用于获取 ServletConfig，getServletInfo 方法可以获取一些 Servlet 相关的信息，如作者、版权等，这个方法需要自己实现，默认返回空字符串。

比较重要的 service 方法用于具体处理一个请求; 是 Servlet 的主要功能。

Init 方法被调用时会接收到一个 ServletConfig 类型的参数，是容器传进去的。配置一般来源于 xml 文件。ServletConfig 接口定义如下:

```
public interface ServletConfig {
    public String getServletName();
    public ServletContext getServletContext();
    public String getInitParameter(String name);
    public Enumeration<String> getInitParameterNames();
}
```

这几个方法都比较简单，必要重要的 getServletContext 返回值 ServletContext 代表的是我们这个应用本身。既然 ServletContext 代表应用本身，那么 ServletContext 里边设置的参数就

可以被当前应用的所有 Servlet 共享了。我们做项目的时候都知道参数可以保存在 Session 中，也可以保存在 Application 中，而后者很多时候就是保存在了 ServletContext 中。

我们可以这么理解，ServletConfig 是 Servlet 级的，而 ServletContext 是 Context（也就是 Application）级的。当然，ServletContext 的功能要强大很多，并不只是保存一下配置参数，否则就叫 ServletContextConfig 了。

GenericServlet 是 Servlet 的默认实现，源码非常简单不贴了，主要做了三件事：

- 实现了 ServletConfig 接口，我们可以直接调用 ServletConfig 里面的方法;
- 提供了无参的 init 方法;
- 提供了 log 方法，可以记录日志或异常。

HttpServlet 是用 HTTP 协议实现的 Servlet 的基类，写 Servlet 时直接继承它就可以了，不需要再从头实现 Servlet 接口。

既然 HttpServlet 是跟协议相关的，当然主要关心的是如何处理请求了，所以 HttpServlet 主要重写了 service 方法。在 service 方法中首先将 ServletRequest 和 ServletResponse 转换为了 HttpServletRequest 和 HttpServletResponse，然后根据 Http 请求的类型不同将请求路由到了不同的处理方法。

具体的处理方法是 doXXX 结构 (doGet, doHead)。doGet, doPost, doPut, doDelete 方法都是模板方法，需要子类实现，否额会抛异常。

- **doGet**: 针对 Get 方法做了优化，调用前会做过期检查，如果没有过期则直接返回 304 状态码使用缓存。
- **doHead**: 调用 deGet 方法，返回空 body 的 Response。
- **doOptions, doTrace**: 功能非常固定，HttpServlet 提供了默认实现。

7.2.2 HttpServletBean

在 DispatcherServlet 集成结构中，HttpServletBean 直接继承了 HttpServlet，同时实现了 EnvironmentCapable 和 EnvironmentAware 接口。首先我们需要知道有 Aware 和 Capable 后缀接口的意义:

- Aware: Aware 接口本身是一个标识接口，XXXAware 在 spring 里表示对 XXX 可感知，如果在某个类里面想要使用 spring 的一些东西，就可以通过实现 XXXAware 接口告诉 spring 给你送过来，而接收的方式是通过实现接口唯一的方法 set-XXX。
- Capable: 具备某些能力，如具有 Environment 的能力，当 spring 需要 Environment 的时候就会调用其 getEnvironment 方法跟它要。

Environment 是环境的意思，具体功能和 ServletContext 类似。在 HttpServletBean 中 Environment 使用的是 Standard-Servlet-Environment，它封装了 ServletContext，同时还封装了 ServletConfig、JndiProperty、系统环境变量和系统属性，这些都封装到了其 propertySources 属性下。总而言之，很多环境相关的配置都存放在了 Environment 中。

在 HttpServletBean 的 init 中，首先将 Servlet 中配置的参数使用 BeanWrapper 设置到 Dis-

patcherServle 的相关属性，然后调用模板方法 initServletBean，子类就通过这个方法初始化。

7.2.3 FrameworkServlet

FrameworkServlet 实现了 ApplicationContextAware, 也即获得了获取 ApplicationContext 的能力。

Framework 实现了 initServletBean 方法，其中有两句核心代码:

```
this.webApplicationContext = initWebApplicationContext();
initFrameworkServlet();
```

initFrameworkServlet 方法是模板方法，子类可以覆盖然后在里面做一些初始化的工作。FrameworkServlet 在构建的过程中的主要作用就是初始化了 WebApplicationContext。initWebApplicationContext 方法做了三件事:

- 获取 spring 的根容器 rootContext。
- 设置 webApplicationContext 并根据情况调用 onRefresh 方法。
- 将 webApplicationContext 设置到 ServletContext 中。

7.2.4 DispathchServlet

onRefresh 方法是 DispatcherServlet 的入口方法。onRefresh 中简单地调用了 initStrategies，在 initStrategies 中调用了 9 个初始化方法:

```
@Override
protected void onRefresh(ApplicationContext context) {
  initStrategies(context);
}

protected void initStrategies(ApplicationContext context) {
  initMultipartResolver(context);
  initLocaleResolver(context);
  initThemeResolver(context);
  initHandlerMappings(context);
  initHandlerAdapters(context);
  initHandlerExceptionResolvers(context);
  initRequestToViewNameTranslator(context);
  initViewResolvers(context);
  initFlashMapManager(context);
}
```

initStrategies 的具体内容非常简单，就是初始化的 9 个组件。下面看一下 initLocaleResolver 的具体实现:

```
private void initLocaleResolver(ApplicationContext context) {
  try {
    this.localeResolver = context.getBean(LOCALE_RESOLVER_BEAN_NAME, LocaleResolver.class);
    if (logger.isTraceEnabled()) {
```

```
      logger.trace("Detected " + this.localeResolver);
    }
    else if (logger.isDebugEnabled()) {
      logger.debug("Detected " + this.localeResolver.getClass().getSimpleName());
    }
  }
  catch (NoSuchBeanDefinitionException ex) {
    // We need to use the default.
    this.localeResolver = getDefaultStrategy(context, LocaleResolver.class);
    if (logger.isTraceEnabled()) {
      logger.trace("No LocaleResolver '" + LOCALE_RESOLVER_BEAN_NAME +
          "': using default [" + this.localeResolver.getClass().getSimpleName() + "]");
    }
  }
}
```

初始化方式分两步：首先通过 context.getBean 在容器里面按注册时的名称或类型（这里指 “localeResolver” 名称或者 LocaleResolver.class 类型）进行查找，如果找不到就调用 getDefaultStrategy 按照类型获取默认的组件。

7.3 MVC Servlet 处理过程

7.3.1 FrameworkServlet

前面讲过 Servlet 的处理过程：首先是从 Servlet 接口的 service 方法开始，然后在 HttpServlet 的 service 方法中根据请求的类型不同将请求路由到了对应的 doXXX 方法中。

在 FrameworkServlet 中重写了 service 与大部分 doXXX 方法，(doHead 没重写)。在 service 方法中增加了对 PATCH 类型请求的处理，其他类型的请求直接交给了父类进行处理; 其中:

- doOptions 和 doTrace 方法可以通过设置 dispatchOptionsRequest 和 dispatchTraceRequest 参数决定是自己处理还是交给父类处理（默认都是交给父类处理，doOptions 会在父类的处理结果中增加 PATCH 类型）;
- doGet、doPost、doPut 和 doDelete 都是自己处理。所有需要自己处理的请求都交给了 processRequest 方法进行统一处理。

```
@Override
protected void service(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException {
    HttpMethod httpMethod = HttpMethod.resolve(request.getMethod());
    if (httpMethod == HttpMethod.PATCH || httpMethod == null) {
        processRequest(request, response);
    }
    else {
        super.service(request, response);
    }
}

```

```
@Override
protected final void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    processRequest(request, response);
}
```

看到这可能会比较迷惑，为什么 service 调用父类 service 方法处理 request，又要自己重写 doXXX 方法，直接覆盖了 service 不是就可以了吗？原书的解释如下:

> **问题 7.1.** 可能有的读者会想，直接覆盖了 *service* 不是就可以了吗？*HttpServlet* 是在 *service* 方法中将请求路由到不同的方法的，如果在 *service* 中不再调用 *super.service()*，而是直接将请求交给 *processRequest* 处理不是更简单吗？从现在的结构来看确实如此，不过那么做其实存在着一些问题。比如，我们为了某种特殊需求需要在 *Post* 请求处理前对 *request* 做一些处理，这时可能会新建一个继承自 *DispatcherServlet* 的类，然后覆盖 *doPost* 方法，在里面先对 *request* 做处理，然后再调用 *supper.doPost()*，但是父类根本就没调用 *doPost*，所以这时候就会出问题了。虽然这个问题的解决方法也很简单，但是按正常的逻辑，调用 *doPost* 应该可以完成才合理，而且一般情况下开发者并不需要对 *Spring MVC* 内部的结构非常了解，所以 *Spring MVC* 的这种做法虽然看起来有点笨拙但是是必要的。

那么，为什么自己处理的 doXXX 方法又要统一到 processRequest 中处理。下面看一下最核心的方法: processRequest。

processRequest 方法中的核心语句是 doService（request，response），这是一个模板方法，在 DispatcherServlet 中具体实现。在 doService 前后还做了一些事情 (装饰器模式?)，配置了一些信息做了一些判断。具体是如何实现的讲起来比较复杂，请看原文或者源码。

7.3.2 DispatcherServlet

通过之前的分析我们知道，DispatcherServlet 里面执行处理的入口方法应该是 doService，不过 doService 并没有直接进行处理，而是交给了 doDispatch 进行具体的处理，在 doDispatch 处理前 doService 做了一些事情: 首先判断是不是 include 请求，如果是则对 request 的 Attribute 做个快照备份，等 doDispatch 处理完之后进行还原。在做完快照后又对 request 设置了一些属性。比较复杂，看懂了再回来写。

doDispatch 方法非常简洁，从顶层设计了整个请求处理的过程。doDispatch 中最核心的代码只要 4 句，它们的任务分别是:

- 根据 request 找到 Handler。
- 根据 Handler 找到对应的 HandlerAdapter。
- 用 HandlerAdapter 处理 Handler。
- 调用 processDispatchResult 方法处理上面处理之后的结果 (包裹找到 View 并渲染输出给用户)。

```
mappedHandler = getHandler(processedRequest);
HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());
```

```
mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);
```

先解释三个概念: HandlerMapping、Handler 和 HandlerAdapter:

- Handler: 处理器，直接对应 MVC 中的 Controller 层，具体表现形式有很多，可以是类，方法。@RequestMapping 的所有方法都可以看成一个 Handler。只要可以实际处理请求就可以是 Handler。
- HandlerMapping: 是用来查找 Handler 的。
- HandlerAdapter: 因为 Spring MVC 中的 Handler 可以是任意的形式，只要能处理请求就 OK，但是 Servlet 需要的处理方法的结构却是固定的，都是以 request 和 response 为参数的方法（如 doService 方法）。HandlerAdapter 负责进行适配，是最复杂的组件。

另外 View 和 ViewResolver 的原理与 Handler 和 HandlerMapping 的原理类似。View 是用来展示数据的，而 ViewResolver 用来查找 View。View 里面的内容就是 Model。

7.3.3 doDispatch 结构

doDispatch 大体可分为两部分: 处理请求 (Handler) 和渲染页面 (View)。大致的实现逻辑如下:

- 检查是不是上传请求，如果是上传请求，则将 request 转换为 MultipartHttpServletRequest，并将 multipartRequestParsed 标志设置为 true。其中使用到了 MultipartResolver。
- 通过 getHandler 方法获取 Handler 处理器链，其中使用到了 HandlerMapping，返回值为 HandlerExecutionChain 类型，其中包含着与当前 request 相匹配的 Interceptor 和 Handler。
- 接下来是处理 GET、HEAD 请求的 Last-Modified。浏览器第一次请求资源（GET、Head 请求）时，返回的请求头里面会包含一个 Last-Modified 的属性，在浏览器以后发送请求时会同时发送之前接收到的 Last-Modified，服务器接收到带 Last-Modified 的请求后会用其值和自己实际资源的最后修改时间做对比，判断是否需要更新。
- 接下来依次调用相应 Interceptor 的 preHandle。
- 处理完成后，让 HandlerAdapter 使用 Handler 处理请求，Controller 就是在这个地方执行的。这里主要使用了 HandlerAdapter。
- Handler 处理完请求后，如果需要异步处理，则直接返回，如果不需要异步处理，当 view 为空时（如 Handler 返回值为 void），设置默认 view，然后执行相应 Interceptor 的 postHandle。设置默认 view 的过程中使用到了 ViewNameTranslator。
- 到这里请求处理的内容就完成了，接下来使用 processDispatchResult 方法处理前面返回的结果，其中包括处理异常、渲染页面、触发 Interceptor 的 afterCompletion 方法三部分内容。

doDispatch 方法的具体实现比较复杂，涉及到很多组件，现在知道他是干什么的就行了。

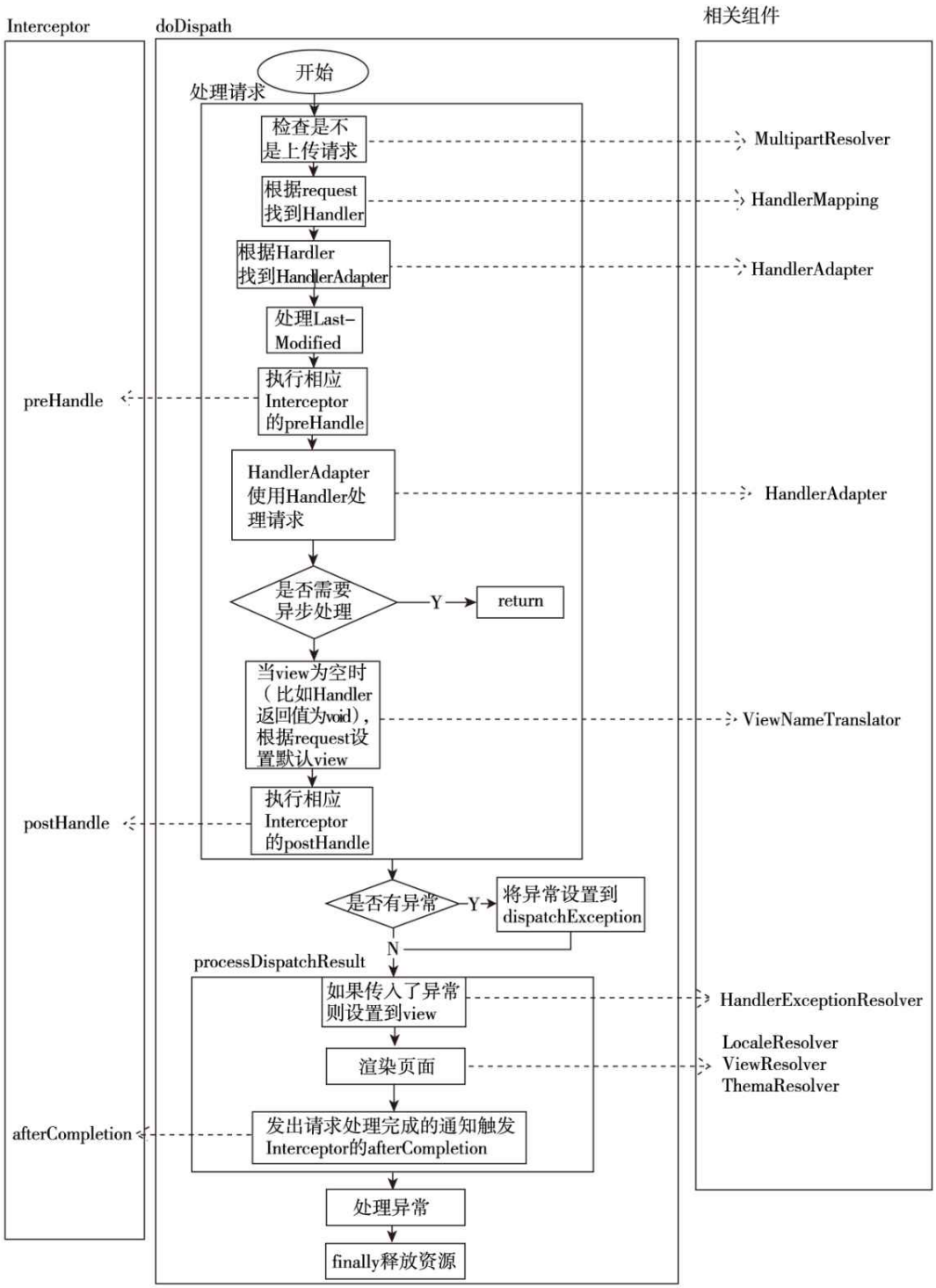


图 7.4 doDispatch 方法处理流程 (源自：« 看透 Spring MVC: 源代码分析与实践 »)

8 Module

8.1 Module 简介

这里主要介绍在 DispatchServlet 中初始化的九大组件，这些组件内部还可能使用到一些子组件。

HandlerMapping

HandlerMapping 的作用是根据 request 找到对应的处理器 Handler 和 Interceptors，HandlerMapping 接口里只有一个公有方法:

```
@Nullable
HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception;
```

在 HandlerExecutionChain 里只包含两个主要元素:

```
private final Object handler;
private final List<HandlerInterceptor> interceptorList = new ArrayList<>();
```

在纯 Spring MVC 中，多个 HandlerMapping 的搜索顺序需要手动配置。

HandlerAdapter

HandlerAdapter 是用来处理 Handler 的，他有三个主要方法:

- supports: 判断是否可以使用某个 Handler;
- handle: 使用 handler 干活;
- getLastModified: 获取资源的 Last-Modified。(被废弃！)

```
public interface HandlerAdapter {
    boolean supports(Object handler);
    @Nullable
    ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object
        handler) throws Exception;
    @Deprecated
    long getLastModified(HttpServletRequest request, Object handler);
}
```

之所以要使用 HandlerAdapter 是因为 Spring MVC 中并没有对处理器做任何限制，处理器可以以任意合理的方式来表现，可以是一个类，也可以是一个方法，还可以是别的合理的方式，从 handle 方法可以看出它是 Object 的类型。这种模式就给开发者提供了极大的自由。

使用哪个 HandlerAdapter 的过程在 DispatchServlet 的 getHandlerAdapter 方法中，通过遍历素有 Adapter 检查哪个可以处理当前的 Handler：

```
protected HandlerAdapter getHandlerAdapter(Object handler) throws ServletException {
    if (this.handlerAdapters != null) {
```

```
        for (HandlerAdapter adapter : this.handlerAdapters) {
            if (adapter.supports(handler)) {
                return adapter;
            }
        }
    }
    throw new ServletException(XXX);
}
```

HandlerExceptionResolver

HandlerExceptionResolver 根据异常设置 ModelAndView，之后再交给 render 方法进行渲染。HandlerExceptionResolver 是在 Render 之前工作的，只是用于解析对请求做处理的过程中产生的异常。

```
public interface HandlerExceptionResolver {
    @Nullable
    ModelAndView resolveException(HttpServletRequest request, HttpServletResponse response,
        @Nullable Object handler, Exception ex);
}
```

HandlerExceptionResolver 结构非常简单，只有一个方法，只需要从异常解析出 ModelAndView 就可以了。具体实现可以维护一个异常为 key、View 为 value 的 Map，解析时直接从 Map 里获取 View，如果在 Map 里没有相应的异常可以返回默认的 View。

ViewResolver

ViewResolver 用来将 String 类型的视图名和 Locale 解析为 View 类型的视图，ViewResolve 接口也非常简单，只有一个方法，定义如下：

```
public interface ViewResolver {
    @Nullable
    View resolveViewName(String viewName, Locale locale) throws Exception;
}
```

View 是用来渲染页面的，通俗点说就是要将程序返回的参数填入模板里，生成 html（也可能是其他类型）文件。这里有两个关键的问题：使用哪个模板？用什么技术填入参数？

ViewResolver 需要找到渲染所用的模板和所用的技术（也就是视图的类型）进行渲染，具体的渲染过程则交给不同的视图自己完成。我们最常使用的 UrlBasedViewResolver 系列的解析器都是针对单一视图类型进行解析的，只需要找到使用的模板就可以了，比如，InternalResourceViewResolver 只针对 jsp 类型的视图。

RequestToViewNameTranslator

ViewResolver 是根据 ViewName 查找 View，但有的 Handler 处理完后并没有设置 View 也没有设置 viewName，这时就需要从 request 获取 viewName 了，而如何从 request 获取 viewName 就是 RequestToViewNameTranslator 要做的事情。RequestToViewNameTranslator 接口定义如下：

```
public interface RequestToViewNameTranslator {
  @Nullable
  String getViewName(HttpServletRequest request) throws Exception;
}
```

RequestToViewNameTranslator 在 Spring MVC 容器里只可以配置一个，所以所有 request 到 ViewName 的转换规则都要在一个 Translator 里面全部实现。

LocalResolver

解析视图需要两个参数：一个是视图名，另一个是 Locale。视图名是处理器返回的（或者使用 RequestToViewNameTranslator 解析的默认视图名），Locale 是从哪里来的呢？这就是 LocaleResolver 要做的事情。

LocaleResolver 用于从 request 解析出 Locale。Locale 就是 zh-cn 之类，表示一个区域。有了这个就可以对不同区域的用户显示不同的结果，这就是 i18n（国际化）的基本原理，LocaleResolver 是 i18n 的基础。LocaleResolver 接口定义如下：

```
public interface LocaleResolver {
    Locale resolveLocale(HttpServletRequest request);
    void setLocale(HttpServletRequest request, @Nullable HttpServletResponse response,
        @Nullable Locale locale);
}
```

接口定义非常简单，只有 2 个方法，分别表示：从 request 解析出 Locale 和将特定的 Locale 设置给某个 request。容器会将 localeResolver 设置到 request 的 attribute 中，代码如下：

```
request.setAttribute(LOCALE_RESOLVER_ATTRIBUTE, this.localeResolver);
```

Spring MVC 中主要在两个地方用到了 Locale：

- ViewResolver 解析视图的时候。
- 使用到国际化资源或者主题的时候。

ThemeResolver

ThemeResolver 从名字就可以看出是解析主题用的。ThemeResolver 接口定义如下：

```
public interface ThemeResolver {
    String resolveThemeName(HttpServletRequest request);
    void setThemeName(HttpServletRequest request, @Nullable HttpServletResponse response,
```

```
        @Nullable String themeName);
}
```

不同的主题其实就是换了一套图片、显示效果以及样式等。Spring MVC 中一套主题对应一个 properties 文件，里面存放着跟当前主题相关的所有资源，如图片、css 样式表等。

Spring MVC 中跟主题有关的类主要有 ThemeResolver、ThemeSource 和 Theme。ThemeResolver 的作用是从 request 解析出主题名；ThemeSource 则是根据主题名找到具体的主题；Theme 是 ThemeSource 找出的一个具体的主题，可以通过它获取主题里具体的资源。

MultipartResolver

MultipartResolver 用于处理上传请求，处理方法是将普通的 request 包装成 MultipartHttpServletRequest，后者可以直接调用 getFile 方法获取到 File，如果上传多个文件，还可以调用 getFileMap 得到 FileName→File 结构的 Map，这样就使得上传请求的处理变得非常简单。

当然，这里做的其实是锦上添花的事情，如果上传的请求不用 MultipartResolver 封装成 MultipartHttpServletRequest，直接用原来的 request 也是可以的，所以在 Spring MVC 中此组件没有提供默认值。MultipartResolver 定义如下：

```
public interface MultipartResolver {
    boolean isMultipart(HttpServletRequest request);
    MultipartHttpServletRequest resolveMultipart(HttpServletRequest request) throws
        MultipartException;
    void cleanupMultipart(MultipartHttpServletRequest request);
}
```

这里一共有三个方法，作用分别是判断是不是上传请求、将 request 包装成 MultipartHttpServletRequest、处理完后清理上传过程中产生的临时资源。对上传请求可以简单地判断是不是 multipart / form-data 类型。

FlashMapManager

FlashMap 主要用在 redirect 中传递参数。而 FlashMapManager 是用来管理 FlashMap 的。

```
public interface FlashMapManager {
    @Nullable
    FlashMap retrieveAndUpdate(HttpServletRequest request, HttpServletResponse response);
    void saveOutputFlashMap(FlashMap flashMap, HttpServletRequest request,
        HttpServletResponse response);
}
```

retrieveAndUpdate 方法用于恢复参数，并将恢复过的和超时的参数从保存介质中删除；saveOutputFlashMap 用于将参数保存起来。

默认实现是 SessionFlashMapManager，它是将参数保存到 session 中。

8.2 HandlerMapping