项目 1　初 识 Spring

现代的 Java 应用开发大多是基于 Spring 的应用开发。使用 Spring 能够让开发人员可以专注于业务逻辑的开发，把代码中的基础设施交给 Spring 负责，以减少开发时间，提高编码效率。Spring 框架在 Java 应用开发中占有重要地位，是大多数 Java 开发人员必须要学习的一门技术。

任务 1.1　了 解 Spring

Spring 的中文意思是春天，意味着 Spring 的出现将为 Java 开发人员带来一个舒适、便捷的开发环境。本节将对 Spring 的相关知识做初步的介绍，包括 Spring 是什么、Spring 能做什么、Spring 的发展等内容。

1.1.1　Spring 简介

Spring 是一个基于 Java EE 应用程序开发的轻量级开源框架。该框架由罗德·约翰逊(Rod Johnson)于 2002 年创建，用于解决 Java EE 编程开发中的复杂性，实现敏捷开发。Spring 使开发人员能够使用普通的 JavaBean 实现业务流程，同时为所有的 JavaBean 提供了一个统一的应用配置框架。Spring 的两大核心技术就是控制反转(inversion of control，IOC)和面向切片编程(aspect oriented programming，AOP)。

1.1.2　Spring 的作用

Spring 将从图 1-1 所示的 7 个方面助力 Java 应用开发。

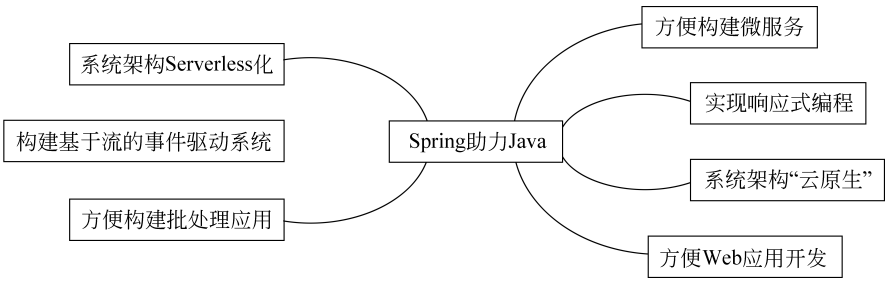

图 1-1　Spring 助力 Java

1. 方便构建微服务

微服务架构是当前 Java 开发的主流。微服务使得应用程序代码能够以可管理的、独立

的模块化形式交付，同时各模块业务功能相对隔离，能够使应用程序更容易维护。Spring 提供的 Spring Boot 框架使开发人员能够更方便、更快捷地构建微服务应用程序，另外，提供的 Spring Cloud 框架可以方便管理和监控各个微服务。

2. 实现响应式编程

响应式编程是一种基于数据流的异步非阻塞编程模型，其本质是对数据流或某种变化所作出的反应。在使用时，开发人员能利用响应式编程的背压机制更方便地进行流量控制。在高并发场景下，能够更好地利用多核心 CPU 的能力构建弹性的消息驱动系统。

3. 系统架构“云原生”

“云原生”的架构不仅能够使开发完毕的系统更方便地迁移上云，而且使系统中的非业务代码和业务代码最大化松耦合，对于系统中的非业务功能则交给 Spring 管理。Spring 提供的 Spring Cloud 框架负责应用程序在云中运行的许多非功能业务，如网关 API、服务配置、服务注册发现、负载均衡、状态跟踪，熔断机制等。

4. 方便 Web 应用开发

Spring 提供的许多子项目可以方便构建 Web 应用程序。例如，Spring 提供的 Spring Boot 框架可以简化 Web 应用程序的配置。提供的 Spring Security 框架集成了许多行业标准的身份验证协议，增强了 Web 应用程序的安全性。提供的 Spring Data 框架便于 Web 应用程序访问关系数据库和非关系数据库。

5. 系统架构 Serverless 化

Serverless 即无服务，无服务不是没有服务器，只是将服务器的运维、管理和分配都托管给了云提供商，开发人员只需关注业务逻辑代码的开发，而不必关注底层计算、存储资源的使用和运维。Spring 产品组合为构建 Serverless 系统提供了强大的功能集合，便于开发人员构建 Serverless 化的系统架构。

6. 构建基于流的事件驱动系统

事件驱动系统采用基于事件的异步数据收发模式，这样能够减少数据收发两端不必要的同步等待时间，提升系统效率。Spring 提供的 Spring Cloud Stream 框架整合了诸如 Kafka、RabbitMQ 等主流的流式消息组件，简化了构建事件驱动系统的复杂性。

7. 方便构建批处理应用

批处理模式就是系统无须与用户交互即可自动、高效地处理大量信息，如定时汇总计算、定时通知、定时任务调度等。Spring 提供的 Spring Batch 框架能够创建高性能、可扩展、弹性的批处理应用程序，同时利用 Spring Boot 框架简化构建批处理应用程序的复杂度。

1.1.3 Spring 的发展

在 2000 年左右，Java EE 程序开发开始兴起，当时 Java EE 程序大多基于 EJB 开发，而 EJB 是一个重量级的业务框架，有框架臃肿、开发效率低、运行开销大等缺点。为了解决上述问题，Rod Johnson 编写了一个轻便、灵巧、易于开发、测试和部署的轻量级开发框架——interface 21 框架。interface 21 框架就是 Spring 框架的前身。随后 Rod Johnson 以 interface 21 框架为基础，经过重新设计于 2004 年 3 月 24 日正式发布了 Spring 1.0 版，Spring 正式进入公众视野。Spring 的整个发展流程如图 1-2 所示。

目前，Spring 已经发展到了 Spring 6 版本。Spring 5 到 Spring 6 是一次大版本号升级，

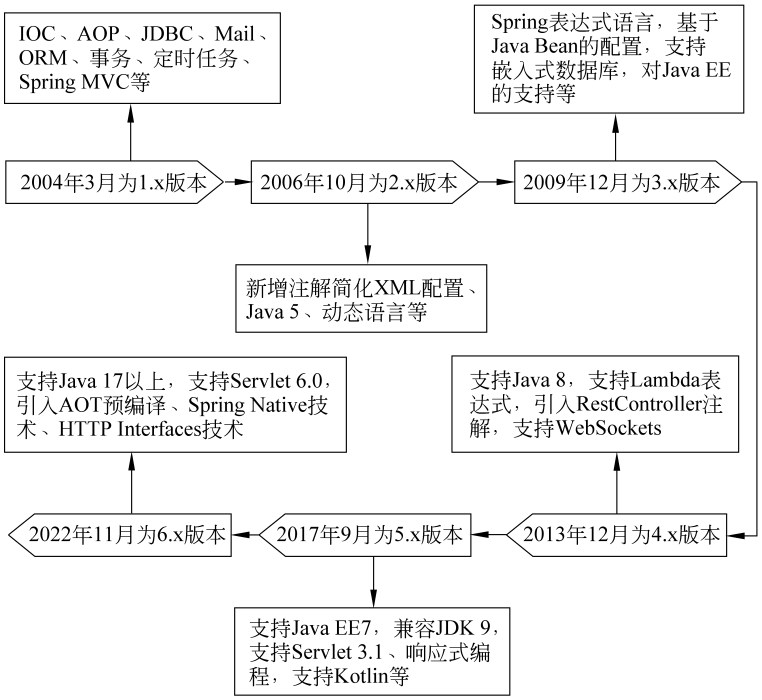

图 1-2　Spring 发展历程

这次升级是阻断式的，加入了很多新特性，同时不向下兼容。Spring 官方将 Spring 6 称为 Spring 下一个十年的新开端。

任务 1.2　认识 Spring 项目模板

Spring 构建了一系列项目模板，便于我们搭建自己的应用程序。下面介绍一些常用的项目模板。

1. Spring Boot

Spring 是一个大而全的框架。随着 Spring 整合了越来越多的框架和组件，Spring 本身变得越来越复杂，配置过多，容易让初学者陷入“配置灾难”。因此，Spring 推出了 Spring Boot 框架，Spring Boot 采用约定优于配置的原则，简化了 Spring 配置和应用程序部署，降低了 Spring 框架使用门槛。Spring Boot 可以基于最少配置创建独立的 Spring 应用程序，实现应用一键启动和部署。一个 Spring Boot 应用就可以看作一个单体的微服务项目。

2. Spring Cloud

当前主流软件架构大多是基于云原生的分布式架构，一个系统会根据业务功能做模块拆分。每个业务功能模块作为一个单独的服务，可以使用 Spring Boot 框架来开发，不同的服务之间进行相互调用。部署时，不同的服务将分布式部署在不同的服务器上。但是随着服务数量的越来越多，服务之间的调用和管理也越来越复杂，需要定制一套行之有效的框架来开发和管理分布式服务。

Spring Cloud 的出现就是为解决微服务架构中的服务治理问题。Spring Cloud 完全基于 Spring Boot 开发，它利用 Spring Boot 的开发便利性，简化了分布式服务的开发、配置和部署。Spring Cloud 整合了一系列框架，提供了服务发现注册、配置中心、消息总线、负载均衡、熔断、数据监控等功能。利用 Spring Cloud 可以把多个 Spring Boot 单体微服务项目统一管理起来，组成一个整体的微服务集群，实现有效管理。

3. Spring Cloud Data Flow

Spring Cloud Data Flow 是一个基于微服务的分布式数据处理框架，支持构建基于 Spring Cloud Stream 流处理和 Spring Cloud Task/Spring Batch 批处理的 Spring Boot 微服务项目。Spring Cloud Data Flow 的优势在于如果企业没有专业的大数据技术团队支持，只要熟悉 Java 和 Spring Boot，也能够利用 Spring Cloud Data Flow 编程实现大数据处理任务。

4. Spring Data

Spring Boot 框架底层默认采用 Spring Data 框架统一访问各种数据库。Spring Data 的作用就是为了简化数据库的访问，包括常见关系型数据库和非关系型数据库。

5. Spring Security

Spring Security 是一个安全访问控制框架。它利用了 Spring IOC 和 AOP 思想，为应用系统提供声明式的安全访问控制功能，使应用系统能够更方便、更快捷地实现安全控制。

任务 1.3　了解 Spring 容器

Spring 容器是 Spring 框架的核心。Spring 利用容器来帮助我们管理普通的 JavaBean 对象（在后续内容中 JavaBean 对象统一简称为 Bean）。Spring 提供了两种类型的容器：一种是 BeanFactory；另一种是 ApplicationContext。下面分别对两者做介绍。

1.3.1　BeanFactory

BeanFactory 是访问 Spring 容器的根接口，该接口为 Spring 依赖注入功能提供支持。BeanFactory 接口在 org.springframework.beans.factory.BeanFactory 包中定义。BeanFactory 提供了基本的容器功能，包括实例化 Bean 和获取 Bean。BeanFactory 在应用启动时不会去实例化 Bean，只有程序访问 Bean 的时候才会去实例化。这样能够使应用启动时占用较少资源，但是获取 Bean 的速度就会相对慢一些，因为临时创建 Bean 也需要时间。Spring 中有很多实现类和子接口继承了 BeanFactory 接口，因此开发中很少直接使用，而多使用其子类。

1.3.2　ApplicationContext

ApplicationContext 是 BeanFactory 的子接口，是开发中常用的 Spring 容器。它在 org.springframework.context.ApplicationContext 包中定义。与 BeanFactory 不同的是，ApplicationContext 在启动时就会实例化所有的 Bean，同时还可以选择性地为某些 Bean 配置懒加载来延迟 Bean 的实例化。此外，ApplicationContext 在 BeanFactory 的所有功能基础上还添加了对国际化、资源访问、事件传播、Web 应用等方面的支持。

实例化 ApplicationContext 容器方法如下：

```
ApplicationContext ctx= new AnnotationConfigApplicationContext(SpringDemoConfig.
class);
```

其中，AnnotationConfigApplicationContext 类会从类路径 ClassPath 中寻找标识为@Configuration的 Spring 配置类，用于实例化 ApplicationContext 容器。

任务 1.4　体验 Spring 编程

目前，Java 开发的主流工具是 Idea 且利用 Maven 工具进行项目管理。下面介绍如何利用 Idea 构建一个基本的 Spring Maven 项目。

1.4.1　环境准备

（1）已安装 Java 17。本书所有章节代码都基于 Spring 6 构建。Spring 6 支持的最低 Java 版本为 Java 17，因此需要确保在计算机上已安装 Java 17，并配置环境变量 JAVA_HOME 为 Java 17 的安装路径。本书所使用的 Java 版本为 Java 17.0.4。

（2）安装 Idea 并配置 Maven。Idea 版本为 2023。本书后续项目都使用 Maven 管理，Idea 自带 Maven 插件，不需额外安装。每个 Maven 项目都有一个 pom.xml 文件，Maven 中的依赖包是基于 pom.xml 文件里面的三个标签＜groupId＞、＜artifactId＞、＜version＞来定位的。其中，groupId 为项目组 id，一般为公司域名的倒写。artifactId 为构件 id，一般为项目名。version 为项目的版本。例如要引入 Spring 的 webmvc 相关 Jar 包，版本为 6.0.3。可以在 pom.xml 文件中添加＜dependencies＞标签，在＜dependencies＞标签内部添加如下信息。

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>6.0.3</version>
</dependency>
```

除了＜dependencies＞标签外，pom.xml 文件里也提供了很多其他标签，在后续编程中如有出现，再进行介绍。

1.4.2　创建 Spring Maven 项目

下面展示如何利用 Idea 创建一个简单的 Spring Maven 项目。

创建 Spring Maven 项目

建立 Spring 开发环境步骤说明

1.4.3 认识注解

在 Spring 编程中注解被大量使用。注解并不是 Spring 独有的，早在 Java 5 中就引入了注解的概念。Spring 在 Java 的基础上又扩充了很多新的注解，以方便编程。使用注解首先要学会看懂注解，这里就必须要了解一个概念——元注解。元注解用于描述注解，元注解本身也是一种注解，但是它能够应用到其他注解上面，对之进行解释说明。

元注解分为@Retention、@Documented、@Target、@Inherited、@Repeatable 五种，下面做简单介绍。

1. @Retention

@Retention 用于描述注解的生命周期。@Retention 注解有三种取值，具体如表 1-1 所示。

表 1-1 @Retention 注解

取值类型	描述	使用方法
RetentionPolicy.SOURCE	表示被修饰的注解只在源代码中保留，编译器不保留	@Retention(RetentionPolicy.SOURCE)
RetentionPolicy.CLASS	表示被修饰的注解只在.class 文件中，JVM 不保留	@Retention(RetentionPolicy.CLASS)
RetentionPolicy.RUNTIME	表示被修饰的注解会被 JVM 保留，在运行时有效	@Retention(RetentionPolicy.RUNTIME)

2. @Documented

@Documented 用于描述注解可以被加入 Javadoc 工具文档中。

3. @Target

@Target 用于描述注解的使用范围，包括类、方法、变量等。@Target 取值如表 1-2 所示。

表 1-2 @Target 注解

取值类型	描述	使用方法
ElementType.ANNOTATION_TYPE	表示被修饰的注解只能应用在其他注解上	@Target(ElementType.ANNOTATION_TYPE)
ElementType.CONSTRUCTOR	表示被修饰的注解只能应用在构造函数上	@Target(ElementType.CONSTRUCTOR)
ElementType.FIELD	表示被修饰的注解只能应用在类属性上	@Target(ElementType.FIELD)
ElementType.LOCAL_VARIABLE	表示被修饰的注解只能应用在局部变量上	@Target(ElementType.LOCAL_VARIABLE)
ElementType.METHOD	表示被修饰的注解只能应用在方法上	@Target(ElementType.METHOD)
ElementType.PACKAGE	表示被修饰的注解只能应用在包上	@Target(ElementType.PACKAGE)
ElementType.PARAMETER	表示被修饰的注解只能应用在方法参数上	@Target(ElementType.PARAMETER)
ElementType.TYPE	表示被修饰的注解只能应用在类、接口、枚举等元素上	@Target(ElementType.TYPE)

4. @Inherited

被@Inherited 修饰的注解作用在某父类上，如果该父类的子类没有被任何其他注解修饰，那么这个子类就继承了父类的注解。具体使用方法如下。

```
/* 定义 Father 注解,被@Inherited 修饰 */
@Inherited
public @interface Father {
}
/* 定义 Person 父类,被 Father 注解修饰 */
@Father
class Person{}
/* 定义 Person 的子类 man,也被 Father 注解修饰 */
class man extends Person{}
```

5. @Repeatable

@Repeatable 用于描述该注解可重复使用在同一类、方法和属性上。例如使用注解@PropertySources 加载多个配置文件，可以采用以下写法：

```
@PropertySources({@PropertySource(value="config1.xml"),
                  @PropertySource(value="config2.xml")})
public class Config{
}
```

这是因为@PropertySources 注解中有以下定义：

```
public @interface PropertySources {
    PropertySource[] value();
}
```

其中，PropertySource 也是一个注解，并被@Repeatable 修饰，修饰代码如下：

```
@Repeatable(PropertySources.class)
public @interface PropertySource {
}
```

因此，才能在@PropertySources 中多次重复使用@PropertySource 加载配置文件。

Spring 中的注解非常多，但它们共同被这五大元注解组合修饰。

1.4.4　基于注解方式的 Spring 编程

在 1.3.2 节中提到了我们可以通过 ApplicationContext 容器来管理 Bean。要获取具体的 Bean 有两种方式，分别是基于配置文件的方式和基于注解方式，其中基于注解方式因使用便捷成为项目开发中的主流。因此，这里重点介绍基于注解方式获取 Bean。

基于注解方式获取 Bean 要定义一个 Java 类作为 Spring 配置类并以@Configuration 注解修饰，在配置类中使用注解生成不同的 Bean。获取 Bean 时，首先获取 ApplicationContext 对象，然后调用 getBean 方法通过 id 或类型获取具体的 Bean。本书后续大部分 Spring 相关配置

均基于注解方式实现。

下面介绍如何以注解方式实例化 ApplicationContext 容器。

在 src/main/java 目录下分别创建两个文件夹 bean 和 config，如图 1-3 所示。bean 文件夹用于存放 JavaBean，config 文件夹用于存放 Spring 配置类。

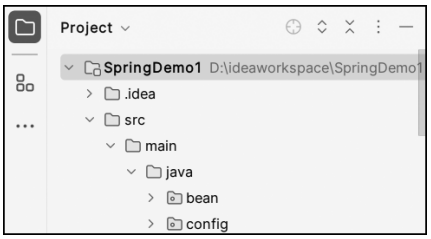

图 1-3　创建文件夹 bean 和 config

基于注解的 Spring 基本编程

在 config 文件夹下创建 SpringDemoConfig.java 文件作为 Spring 配置类，在 SpringDemoConfig.java 中输入以下代码。其中，@Configuration 用于将该类标识为 Spring 配置类，@ComponentScan("bean")用于设置生成 Bean 时自动扫描路径为 ClassPath 项目下的 bean 文件夹。

【SpringDemoConfig.java】

```
package config;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
@Configuration
@ComponentScan("bean")
public class SpringDemoConfig {
}
```

在 bean 文件夹下创建类 Demo1.java。该类中提供一个 say 方法，该方法用于控制台输出 hello demo1。Demo1.java 中代码如下，其中，@Component 注解用于将 Demo1 这个类标识为一个 Spring 组件类的 Bean。

【Demo1.java】

```
package bean;
import org.springframework.stereotype.Component;
@Component
public class Demo1 {
    public void say(){
        System.out.println("hello demo1");
    }
}
```

在项目根目录 src/main/java 下创建一个测试类 BeanTest.java，在测试类中调用 Demo1 类中的 say 方法打印输出 hello demo1。传统方式需要编写以下代码，手动创建一个 Demo1 实例对象，然后调用 Demo1 对象的 say 方法。

```
Demo1 demo1=new Demo1();
demo1.say();
```

使用 Spring 后，Demo1 实例对象的创建交给 Spring 容器来做，创建完毕可从 Spring 容器中利用 getBean 方法获取 Demo1 实例对象并调用 say 方法输出内容。在 BeanTest.java 中输入以下代码，代码中通过 id 和类型两种不同方式获取实例化 Demo1 对象。

```
import bean.Demo1;
import config.SpringDemoConfig;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
public class BeanTest {
    public static void main(String[] args){
        //实例化 ApplicationContext 容器
        ApplicationContext ctx=
        new AnnotationConfigApplicationContext(SpringDemoConfig.class);
        Demo1 deCtxByName=(Demo1)ctx.getBean("demo1");
        deCtxByName.say();
        Demo1 deCtxByType=ctx.getBean(Demo1.class);
        deCtxByType.say();
    }
}
```

运行 BeanTest.java 即可在控制台看到分别打印出 2 次 hello demo1，如图 1-4 所示。证明通过 Bean 的 id 和类型两种方法均可获取实例化 Bean 并调用方法输出结果。

图 1-4　输出 hello demo1

修改 Demo1.java 文件，在其中添加一个无参数构造方法，在构造方法中添加打印语句。下面所示粗体代码为新增内容。

【Demo1.java】

```
package bean;
import org.springframework.stereotype.Component;
@Component
public class Demo1 {
    public Demo1(){
        System.out.println("Constructor Demo1");
    }
```

```
    public void say(){
        System.out.println("hello demo1");
    }
}
```

修改 BeanTest.java 文件，main 方法中只保留第一条语句，用于获取 ApplicationContext 对象。注释如下面黑体代码。

【BeanTest.java】

```
import bean.Demo1;
import config.SpringDemoConfig;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
public class BeanTest {
    public static void main(String[] args){
        //实例化 ApplicationContext 容器
        ApplicationContext ctx=
        new AnnotationConfigApplicationContext(SpringDemoConfig.class);
        /* Demo1 deCtxByName=(Demo1)ctx.getBean("demo1");
        deCtxByName.say();
        Demo1 deCtxByType=ctx.getBean(Demo1.class);
        deCtxByType.say(); */
    }
}
```

运行 BeanTest.java，看到控制台打印出 Constructor Demo1，如图 1-5 所示。代码中即使不获取 Demo1 实例对象，在实例化 ApplicationContext 容器对象时，Demo1 的无参构造方法已经被调用，Demo1 实例对象已经创建完毕。

图 1-5　输出 Constructor Demo1

任务 1.5　了解 Spring 控制反转（IOC）

在传统编程中，大多数应用程序都是由多个类通过彼此的合作来实现复杂的业务逻辑，当需要使用类的对象时，一般会直接在程序中直接创建，对象和程序本身形成了硬编码关系，耦合度太高。如果能够将创建对象的权限转移给第三方容器实现，需要时再通过第三方容器获取对象的引用，这样能够降低代码的耦合度，有利于代码维护。这正是 IOC 产生的原因。