# Spring 框架全面指南

此博客文章是在 ChatGPT-4o 的協助下撰寫的。

---

**目錄**

- 結論

---

## 簡介

Spring 是構建企業級 Java 應用程序最受歡迎的框架之一。它為開發 Java 應用程序提供了全面的基礎設施支持。在本博客中，我們將涵蓋 Spring 生態系統的各個方面，包括 Spring Boot、數據管理、構建 RESTful 服務、調度、測試以及高級功能，如 Spring Advice API。

---

## Spring Boot 框架

**開始使用 Spring Boot** Spring Boot 使得創建獨立、生產級的基於 Spring 的應用程序變得非常容易。它對 Spring 平台和第三方庫採用了約定優於配置的方式，讓你可以以最少的配置開始。

- 初始設置：使用 Spring Initializr 創建一個新的 Spring Boot 項目。你可以選擇所需的依賴項，例如 Spring Web、Spring Data JPA 和 Spring Boot Actuator。
- 註解：了解關鍵的註解，如 `@SpringBootApplication`，它是 `@Configuration`、`@EnableAutoConfiguration` 和 `@ComponentScan` 的組合。
- 嵌入式服務器：Spring Boot 使用嵌入式服務器（如 Tomcat、Jetty 或 Undertow）來運行你的應用程序，因此你不需要將 WAR 文件部署到外部服務器。

**依賴注入** 依賴注入（DI）是 Spring 的核心原則之一。它允許創建鬆散耦合的組件，使你的代碼更具模塊化且更易於測試。

- @Autowired：此註解用於自動注入依賴項。它可以應用於構造函數、字段和方法。Spring 的依賴注入功能將自動解析並將協作 Bean 注入到你的 Bean 中。

  字段注入的示例：

```
@Component
public class UserService {

    @Autowired
    private UserRepository userRepository;
```

```
    // 業務方法
}
```

構造函數注入的示例：

```
@Component
public class UserService {

    private final UserRepository userRepository;

    @Autowired
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    // 業務方法
}
```

方法注入的示例：

```
@Component
public class UserService {

    private UserRepository userRepository;

    @Autowired
    public void setUserRepository(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    // 業務方法
}
```

- @Component、@Service、@Repository：這些是 `@Component` 註解的特殊化，用於指示類是 Spring Bean。它們還作為提示，表明註解類所扮演的角色。
  - @Component：這是任何 Spring 管理組件的通用註解。它可以用於將任何類標記為 Spring Bean。

    示例：

```
@Component
public class EmailValidator {

    public boolean isValid(String email) {
        // 驗證邏輯
        return true;
    }
}
```

- @Service：此註解是 `@Component` 的特殊化，用於將類標記為服務。它通常用於服務層，你在這裡實現業務邏輯。

  示例：

```
@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    public User findUserById(Long id) {
        return userRepository.findById(id).orElse(null);
    }
}
```

- @Repository：此註解也是 `@Component` 的特殊化。它用於指示類提供存儲、檢索、搜索、更新和刪除對象的機制。它還將持久性異常轉換為 Spring 的 DataAccessException 層次結構。

  示例：

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    // 自定義查詢方法
}
```

這些註解使你的 Spring 配置更具可讀性和簡潔性，並幫助 Spring 框架管理和連接不同 Bean 之間的依賴關係。

**Spring 中的事件**　Spring 的事件機制允許你創建和監聽應用程序事件。

- 自定義事件：通過擴展 `ApplicationEvent` 創建自定義事件。例如：

```
public class MyCustomEvent extends ApplicationEvent {
    private String message;

    public MyCustomEvent(Object source, String message) {
        super(source);
        this.message = message;
    }

    public String getMessage() {
        return message;
    }
}
```

- 事件監聽器：使用 `@EventListener` 或實現 `ApplicationListener` 來處理事件。例如：

```
@Component
public class MyEventListener {

    @EventListener
    public void handleMyCustomEvent(MyCustomEvent event) {
        System.out.println(" 收到 Spring 自定義事件 - " + event.getMessage());
    }
}
```

- 發布事件：使用 `ApplicationEventPublisher` 發布事件。例如：

```
@Component
public class MyEventPublisher {

    @Autowired
    private ApplicationEventPublisher applicationEventPublisher;

    public void publishCustomEvent(final String message) {
        System.out.println(" 發布自定義事件。");
        MyCustomEvent customEvent = new MyCustomEvent(this, message);
        applicationEventPublisher.publishEvent(customEvent);
    }
}
```

---

### 使用 Spring 進行數據管理

**Spring Data JDBC** Spring Data JDBC 提供了簡單而有效的 JDBC 訪問。

- 存儲庫：定義存儲庫以執行 CRUD 操作。例如：

```
public interface UserRepository extends CrudRepository<User, Long> {
}
```

- 查詢：使用註解如 `@Query` 來定義自定義查詢。例如：

```
@Query("SELECT * FROM users WHERE username = :username")
User findByUsername(String username);
```

**Spring Data JPA** Spring Data JPA 使得實現基於 JPA 的存儲庫變得非常容易。

- 實體映射：使用 `@Entity` 定義實體並將其映射到數據庫表。例如：

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String username;
    private String password;
    // getter 和 setter
}
```

- 存儲庫：通過擴展 `JpaRepository` 創建存儲庫接口。例如：

```
public interface UserRepository extends JpaRepository<User, Long> {
}
```

- 查詢方法：使用查詢方法來執行數據庫操作。例如：

```
List<User> findByUsername(String username);
```

**Spring Data Redis** Spring Data Redis 提供了基於 Redis 的數據訪問基礎設施。

- RedisTemplate：使用 `RedisTemplate` 與 Redis 進行交互。例如：

```
@Autowired
private RedisTemplate<String, Object> redisTemplate;

public void save(String key, Object value) {
    redisTemplate.opsForValue().set(key, value);
}

public Object find(String key) {
    return redisTemplate.opsForValue().get(key);
}
```

- 存儲庫：使用 `@Repository` 創建 Redis 存儲庫。例如：

```
@Repository
public interface RedisRepository extends CrudRepository<RedisEntity, String> {
}
```

**事務和 DAO 支持** Spring 簡化了事務管理和 DAO（數據訪問對象）支持。

- 事務管理：使用 `@Transactional` 來管理事務。例如：

```
@Transactional
public void saveUser(User user) {
    userRepository.save(user);
}
```

- DAO 模式：實現 DAO 模式以分離持久化邏輯。例如：

```
public class UserDao {
    @Autowired
    private JdbcTemplate jdbcTemplate;

    public User findById(Long id) {
        return jdbcTemplate.queryForObject("SELECT * FROM users WHERE id = ?", new Object[]{id}, ne
    }
}
```

**JDBC 和 ORM**　Spring 提供了對 JDBC 和 ORM（對象關係映射）的全面支持。

- JdbcTemplate：使用 `JdbcTemplate` 簡化 JDBC 操作。例如：

```
@Autowired
private JdbcTemplate jdbcTemplate;

public List<User> findAll() {
    return jdbcTemplate.query("SELECT * FROM users", new UserRowMapper());
}
```

- Hibernate：將 Hibernate 與 Spring 集成以獲得 ORM 支持。例如：

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String username;
    private String password;
    // getter 和 setter
}
```

---

### 構建 RESTful 服務

**Spring REST 客戶端**　Spring 使得構建 RESTful 客戶端變得非常容易。

- RestTemplate：使用 `RestTemplate` 進行 HTTP 請求。例如：

```
@Autowired
private RestTemplate restTemplate;

public String getUserInfo(String userId) {
    return restTemplate.getForObject("https://api.example.com/users/" + userId, String.class);
}
```

- WebClient：使用反應式 `WebClient` 進行非阻塞請求。例如：

```
@Autowired
private WebClient.Builder webClientBuilder;

public Mono<String> getUserInfo(String userId) {
    return webClientBuilder.build()
            .get()
            .uri("https://api.example.com/users/" + userId)
            .retrieve()
            .bodyToMono(String.class);
}
```

**FeignClient** Feign 是一個聲明式的 Web 服務客戶端。

- 設置：將 Feign 添加到你的項目中，並創建帶有 `@FeignClient` 註解的接口。例如：

```
@FeignClient(name = "user-service", url = "https://api.example.com")
public interface UserServiceClient {
    @GetMapping("/users/{id}")
    String getUserInfo(@PathVariable("id") String userId);
}
```

- 配置：使用攔截器和錯誤解碼器自定義 Feign 客戶端。例如：

```
@Bean
public RequestInterceptor requestInterceptor() {
    return requestTemplate -> requestTemplate.header("Authorization", "Bearer token");
}
```

---

**郵件、任務和調度**

**郵件支持** Spring 提供了發送郵件的支持。

- JavaMailSender：使用 `JavaMailSender` 發送郵件。例如：

```
@Autowired
private JavaMailSender mailSender;
```

```
public void sendEmail(String to, String subject, String body) {
    SimpleMailMessage message = new SimpleMailMessage();
    message.setTo(to);
    message.setSubject(subject);
    message.setText(body);
    mailSender.send(message);
}
```

- MimeMessage：創建帶有附件和 HTML 內容的豐富郵件。例如：

```
@Autowired
private JavaMailSender mailSender;

public void sendRichEmail(String to, String subject, String body, File attachment) throws Messaging
    MimeMessage message = mailSender.createMimeMessage();
    MimeMessageHelper helper = new MimeMessageHelper(message, true);
    helper.setTo(to);
    helper.setSubject(subject);
    helper.setText(body, true);
    helper.addAttachment(attachment.getName(), attachment);
    mailSender.send(message);
}
```

**任務執行和調度** Spring 的任務執行和調度支持使得運行任務變得非常容易。

- @Scheduled：使用 `@Scheduled` 調度任務。例如：

```
@Scheduled(fixedRate = 5000)
public void performTask() {
    System.out.println(" 每 5 秒運行一次的調度任務");
}
```

- 異步任務：使用 `@Async` 異步運行任務。例如：

```
@Async
public void performAsyncTask() {
    System.out.println(" 在後台運行的異步任務");
}
```

---

### Spring 中的測試

**使用 Mockito 進行測試**　Mockito 是一個強大的測試模擬庫。

- 模擬依賴項：使用 `@Mock` 和 `@InjectMocks` 創建模擬對象。例如：

```
@RunWith(MockitoJUnitRunner.class)
public class UserServiceTest {
    @Mock
    private UserRepository userRepository;

    @InjectMocks
    private UserService userService;

    @Test
    public void testFindUserById() {
        User user = new User();
        user.setId(1L);
        Mockito.when(userRepository.findById(1L)).thenReturn(Optional.of(user));

        User result = userService.findUserById(1L);
        assertNotNull(result);
        assertEquals(1L, result.getId().longValue());
    }
}
```

- 行為驗證：驗證與模擬對象的交互。例如：

```
Mockito.verify(userRepository, times(1)).findById(1L);
```

**使用 MockMvc 進行測試**　MockMvc 允許你測試 Spring MVC 控制器。

- 設置：在測試類中配置 MockMvc。例如：

```
@RunWith(SpringRunner.class)
@WebMvcTest(UserController.class)
```

```
public class UserControllerTest {
    @Autowired
    private MockMvc mockMvc;

    @Test
    public void testGetUser() throws Exception {
        mockMvc.perform(get("/users/1"))
            .andExpect(status().isOk())
            .andExpect(content().contentType(MediaType.APPLICATION_JSON))
            .andExpect(jsonPath("$.id").value(1));
    }
}
```

- 請求構建器：使用請求構建器模擬 HTTP 請求。例如：

```
mockMvc.perform(post("/users")
    .contentType(MediaType.APPLICATION_JSON)
    .content("{\"username\":\"john\",\"password\":\"secret\"}"))
    .andExpect(status().isCreated());
```

---

**監控和管理**

**Spring Boot Actuator** Spring Boot Actuator 提供了生產就緒的功能，用於監控和管理你的應用程序。

- 端點：使用端點如 `/actuator/health` 和 `/actuator/metrics` 來監控應用程序的健康狀況和指標。例如：

```
curl http://localhost:8080/actuator/health
```

- 自定義端點：創建自定義的 Actuator 端點。例如：

```
@RestController
@RequestMapping("/actuator")
public class CustomEndpoint {
    @GetMapping("/custom")
    public Map<String, String> customEndpoint() {
```

```
        Map<String, String> response = new HashMap<>();
        response.put("status", " 自定義 Actuator 端點");
        return response;
    }
}
```

---

### 高級主題

**Spring Advice API** Spring 的 Advice API 提供了高級的 AOP（面向切面編程）功能。

- @Aspect：使用 `@Aspect` 定義切面。例如：

```
@Aspect
@Component
public class LoggingAspect {
    @Before("execution(* com.example.service.*.*(..))")
    public void logBefore(JoinPoint joinPoint) {
        System.out.println(" 方法執行前: " + joinPoint.getSignature().getName());
    }

    @After("execution(* com.example.service.*.*(..))")
    public void logAfter(JoinPoint joinPoint) {
        System.out.println(" 方法執行後: " + joinPoint.getSignature().getName());
    }
}
```

- 連接點：使用連接點定義切面應用的位置。例如：

```
@Pointcut("execution(* com.example.service.*.*(..))")
public void serviceMethods() {}

@Around("serviceMethods()")
public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {
    System.out.println(" 方法執行前: " + joinPoint.getSignature().getName());
    Object result = joinPoint.proceed();
    System.out.println(" 方法執行後: " + joinPoint.getSignature().getName());
```

```
        return result;
    }
```

---

**結論**

Spring 是一個強大且多功能的框架，可以簡化企業級應用程序的開發。通過利用 Spring Boot、Spring Data、Spring REST 和其他 Spring 項目的功能，開發人員可以高效地構建健壯、可擴展且易於維護的應用程序。借助 Spring Boot Actuator 和測試框架等工具，你可以確保你的應用程序是生產就緒且經過良好測試的。