스프링핵심원리-의존관계 주입방법

2021년 8월 12일 목요일 오후 4:10

다양한 의존관계 주입 방법

- 생성자 주입
- 수정자 주입 (setter 주입)
- 필드 주입
- 일반 메소드 주입

생성자 주입
생성자 통해서 의존관계 주입받는 방법
생성자 호출 시점에 딱 1번 호출되는 것을 보장됨
=> 그때 값을 세팅하고 값을 지킬 수 있다는 것
=> **불변, 필수** 의존관계에 사용됨

+생성자가 딱 1개 있으면 @Autowired 생략해도됨

수정자 주입
setter라 불리는 필드의 값을 변경하는 수정자 메소드 통해서 의존관계 주입하는 방법
선택, 변경 가능성이 있는 의존관계에 사용
자바빈 프로퍼티 규약의 수정자 메소드 방식 사용하는 방법이다.
+ @Autowired 의 기본 동작은 주입할 대상 없으면 오류 발생,
주입할 대상이 없어도 동작하게 하려면 @Autowired(required =false)

자바빈 프로퍼티 규약: 자바에서는 과거부터 필드의 값 직접 변경하지 말고 set, get이라는 메소드를 통해서 값을 읽거나 수정하는 규칙

필드 주입
이름 그대로 필드에 바로 주입하는 방법
코드가 간결함 But, 외부에서 변경 불가해서 테스트하기 힘들다라는 치명적인 단점
DI 프레임워크 없으면 아무것도 안됨
=> 왠만하면 사용X

어플리케이션의 실제 코드와 상관없는 테스트코드
스프링 설정을 목적으로 하는 @Configuration 같은 곳에서만 특별한 용도로 사용

일반 메소드 주입

일반 메소드 통해서 주입받을 수 있음
한번에 여러 필드 주입 받을 수 잇음
일반적으로 사용 잘 안함

옵션 처리
주입할 스프링 빈이 없어도 동작해야할 경우가 있음

@Autowired(required = false) : 자동 주입할 대상이 없으면 수정자 메소드 자체가 호출 안됨
org.springframework.lang.@Nullable : 자동 주입할 대상 없으면 null이 입력됨

Optional<> : 자동 주입할 대상 없으면 Optional.empty가 입력됨

```
static class TestBean {

    @Autowired(required = false)  //아예 메소드 자체가 호출 안됨
    public void setNoBean1(Member noBean1) {
        System.out.println("noBean1 = " + noBean1);
    }

    @Autowired  // 호출되는데 null 로 나옴
    public void setNoBean2(@Nullable Member noBean2) {
        System.out.println("noBean2 = " + noBean2);
    }

    @Autowired //스프링 빈이 없으면 Optional.empty 출력됨
    public void setNoBean3(Optional<Member> noBean3) {
        System.out.println("noBean3 = " + noBean3);
    }

}
```

```
noBean2 = null
noBean3 = Optional.empty
```

생성자 주입을 선택해라
why?

불변
대부분의 의존관계 주입은 한번 일어나면 어플리케이션 종료 시점까지 의존관계 변경할 일X
(대부분의 의존관계는 변하면 안됨(불변))
수정자 주입 사용하면, set 메소드 public 으로 열어두어야함
=> 실수로 변경할 수 있는 가능성이 생김
생성자 주입은 객체를 생성할 때 1번 호출 => 불변하게 설계 가능

누락
프레임워크 없이 순수한 자바 코드를 단위 테스트하는 경우
실행은 되는데 의존관계 주입이 누락되어 Null Point Exception 발생하는 경우 생김
But 생성자 주입 사용하면 주입 누락시 컴파일 오류 발생하여 알 수 있음 (final 키워드 사용)

(생성자 주입 이후에 호출되는 나머지 방식들은 final 사용 못함)

결론 => 항상 생성자 주입 사용, 필요하면 수정자 주입, 필드주입은 사용하지 말자

롬복

대부분이 불변 => 생성자도 만들고 주입받는값 대입도 해야되고 코드량이 많아짐
=> 최적화

```
@Autowired
public OrderServiceImpl(MemberRepository memberRepository, DiscountPolicy discountPolicy) {
    this.memberRepository = memberRepository;
    this.discountPolicy = discountPolicy;
}
```

==

```
@RequiredArgsConstructor
```

final 붙은 필수값을 가지고 생성자 만들어줌

조회 빈이 2개 이상인 문제

@Autowired는 타입으로 조회함
타입으로 조회하기 때문에, 의존관계 자동 주입시 문제 발생 가능
하위 타입으로 지정하여 해결할 수 있지만 => DIP 위배 + 유연성 떨어짐

다른 해결법

1. @Autowired 필드명 매칭
   @Autowired 는 타입 매칭 시도하고, 여러 개 있으면 필드명, 매개변수 명으로 빈이름을 추가 매칭함
2. @Quilfier 라는 추가 구분자를 사용
   @Quilfier -> @Quilifier 끼리 매칭 -> 빈 이름 매칭
   (@Quilifier(~~) 로 못찾으면 ~~라는 이름의 스프링 빈을 추가로 찾음,
   But 이렇게 쓰지말자 헷갈림).
   결국 못찾으면 NoSuchBeanDefinitionException 예외 발생
   단점 : 주입받을 때 모든 코드에 @Quilfier 를 붙여주어야 함
3. @Primary 사용
   우선 순위를 정하는 방법 @Autowired 시에 여러 빈이 매칭되면 @Primary가 우선권을 가짐

메인 DB의 커넥션을 획득하는 스프링 빈은 @Primary 적용
서브 DB의 커넥션 빈을 획득하는 경우에는 @Qualifier 를 지정하여 명시적으로 획득하는 방식이 깔끔함
즉 @Primary 는 기본값처럼 동작 @Qualifier는 매우 상세하게 동작
스프링은 자동보다는 수동이, 넓은 범위의 선택보다는 좁은 범위의 선택의 우선 순위가 더 높음 => @Qualifier의 우선 순위이 더 높다

애노테이션 직접 만들기

애노테이션에는 상속이라는 개념 없음. 여러 애노테이션을 모아서 사용하는 기능을 스프링이 지원하는 것
@Qualifier 뿐 아니라 다른 애노테이션도 함께 조합해서 사용할 수 있음
ex) @Autowired 도 재정의가능 (건들 일이 없다만...)

조회한 빈이 모두 필요할 때

```
@Test
void findAllBean() {
    ApplicationContext ac = new AnnotationConfigApplicationContext(AutoAppConfig.class, DiscountService.class);
```

```java
@Test
void findAllBean() {
    ApplicationContext ac = new AnnotationConfigApplicationContext(AutoAppConfig.class, DiscountService.class);

    DiscountService discountService = ac.getBean(DiscountService.class);
    Member member = new Member( id: 1L,  name: "userA", Grade.VIP);
    int discountPrice = discountService.discount(member,  price: 10000,  discountCode: "fixDiscountPolicy");

    assertThat(discountService).isInstanceOf(DiscountService.class);
    assertThat(discountPrice).isEqualTo(1000);

    int rateDiscountPrice = discountService.discount(member,  price: 20000,  discountCode: "rateDiscountPolicy");
    assertThat(rateDiscountPrice).isEqualTo(2000);
}
```

```java
static class DiscountService {
    private final Map<String, DiscountPolicy> policyMap;
    private final List<DiscountPolicy> policies;

    @Autowired
    public DiscountService(Map<String, DiscountPolicy> policyMap, List<DiscountPolicy> policies) {
        this.policyMap = policyMap;
        this.policies = policies;
        System.out.println("policyMap = " + policyMap);
        System.out.println("policies = " + policies);
    }

    public int discount(Member member, int price, String discountCode) {
        DiscountPolicy discountPolicy = policyMap.get(discountCode);
        return discountPolicy.discount(member, price);
    }

}
```

DiscountService 는 Map 으로 모든 DiscountPolicy 를 주입받은다 이때 fixDiscountPolicy, rateDiscountPolicy 가 주입됨
discount() 메소드는 discountCode로 fixDiscountPolicy가 넘어오면 map에서 fixDiscountPolicy 스프링 빈을 찾아서 실행, rateDiscountPolicy 가 넘어오면 rateDiscountPolicy 찾아서 실행

자동, 수동의 올바른 기준

스프링이 갈수록 자동을 선호하는 추세, 계층에 맞추어 일반적인 어플리케이션 로직을 자동으로 스캔할 수 있도록 지원
=> 설정 정보를 기반으로 어플리케이션을 구성하는 부분과 실제 동작하는 부분을 명확하게 나누는 것이 이상적이지만, 스프링 빈 하나 등록할 때 자동으로 하는 것에 비해 수동으로 하기엔 너무 많은 노동력 소모, 규모 커지면 설정 정보 관리 자체도 부담
+ 자동으로 등록해도 OCP, DIP 지킬 수 있음

수동 빈 등록은 언제 사용하면 좋은가?
=>

어플리케이션을 크게 업무로직, 기술 지원 로직 으로 나눌 수 있음
업무 로직 빈 : 웹을 지원하는 컨트롤라, 핵심 비즈니스 로직 있는 서비스, 데이터 계층의 로직

을 처리하는 리포지토리 등
기술 지원 빈 : 기술적인 문제나 공통관심사(AOP)를 처리할 때 주로 사용, 데이터베이스 연결이나, 공통 로그 처리 등

업무 로직은 숫자 많고, 유사한 패턴 존재 => 자동 사용
기술 지원 로직은 수가 적고 어플리케이션 전반에 광범위하게 영향 미침 => 문제 찾기 힘듬
=> 수동 빈 사용으로 명확하게 드러내자