DI 프레임워크 구현 실습

실습 환경

- https://github.com/slipp/jwp-basic 저장소를 자신의 계정으로 fork한다.
- Fork한 저장소를 https://youtu.be/xid_GG8kL_w 동영상 참고해 로컬 개발 환경을 구축한다.
- jwp-basic 저장소의 **step10-di-getting-started** 브랜치로 변경한다.
    - 브랜치를 변경하는 방법은 https://youtu.be/VeTjDYI7UVs 동영상을 참고한다.
- src/test/java 디렉토리의 next.WebServerLauncher를 실행한 후 브라우저에서 http://localhost:8080으로 접속해 질문/답변 게시판 서비스 화면이 나타나는지 확인한다.

요구사항

새로 만든 MVC 프레임워크는 자바 리플렉션을 활용해 @Controller 애노테이션이 설정되어 있는 클래스를 찾아 인스턴스를 생성하고, URL 매핑 작업을 자동화했다. 같은 방법으로 각 클래스에 대한 인스턴스 생성 및 의존관계 설정을 애노테이션으로 자동화한다.

먼저 애노테이션은 각 클래스 역할에 맞도록 컨트롤러는 이미 추가되어 있는 @Controller, 서비스는 @Service, DAO는 @Repository 애노테이션을 설정한다. 이 3개의 설정으로 생성된 각 인스턴스 간의 의존관계는 @Inject 애노테이션을 사용한다.

DI 프레임워크를 활용한 DI 예제

```
@Controller
public class QnaController extends AbstractNewController {
    private MyQnaService qnaService;

    @Inject
    public QnaController(MyQnaService qnaService) {
        this.qnaService = qnaService;
    }

    @RequestMapping("/questions")
    public ModelAndView list(HttpServletRequest request, HttpServletResponse response) throws
Exception {
        return jspView("/qna/list.jsp");
    }
}
```

DI 프레임워크를 활용한 DI 예제

```
@Service
public class MyQnaService {
    private UserRepository userRepository;
    private QuestionRepository questionRepository;

    @Inject
    public MyQnaService(UserRepository userRepository, QuestionRepository questionRepository) {
        this.userRepository = userRepository;
        this.questionRepository = questionRepository;
    }

    [...]
}
```

DI 프레임워크를 활용한 DI 예제

```
@Repository
public class JdbcQuestionRepository implements QuestionRepository {
}

@Repository
public class JdbcUserRepository implements UserRepository {
}
```

DI 프레임워크 테스트 및 팁

효과적인 실습을 위해 애노테이션(core.annotation 패키지), DI가 설정되어 있는 예제 코드(core.di.factory.example), 요구사항을 만족해야 하는 테스트 코드(core.di.factory.BeanFactoryTest)를 제공하고 있다.

BeanFactoryTest의 di() 테스트가 성공하면 생성자를 활용하는 DI 프레임워크 구현을 완료한 것이다. 또한 구현 중 필요한 기능을 도와주기 위해 core.di.factory.BeanFactoryUtils 클래스를 제공하고 있다.

자바 클래스에 대한 인스턴스 생성은 자바 리플렉션 API를 직접 이용할 수도 있지만 이를 추상화한 Spring 프레임워크에서 제공하는org.springframework.beans.BeanUtils의 instantiateClass() 메소드를 사용해도 된다.

1단계 힌트

—

이 문제를 해결하려면 재귀함수를 사용해 구현할 수 있다. @Inject 애노테이션이 설정되어 있는 생성자를 통해 빈을 생성해야 한다. 그런데 이 생성자의 인자로 전달할 빈도 다른 빈과 의존관계에 있다. 이와 같이 꼬리에 꼬리를 물고 빈 간의 의존관계가 발생할 수 있다. 다른 빈과 의존관계를 가지지 않는 빈을 찾아 인스턴스를 생성할 때까지 재귀를 실행하는 방식으로 구현할 수 있다.

재귀를 통해 새로 생성한 빈은 BeanFactory의Map<Class<?>, Object>에 추가해 관리한다. 인스턴스를 생성하기 전에 먼저 Class<?>에 해당하는 빈이Map<Class<?>, Object>에 존재하는지 여부를 판단한 후에 존재하지 않을 경우 생성하는 방식으로 구현하면 된다. 이 같은 재사용 방법이 일반적인 캐시의 동작 원리이다.

2단계 힌트

—

실습

빈(Bean) 인스턴스를 생성하기 위한 재귀 함수를 지원하려면 Class에 대한 빈 인스턴스를 생성하는 메소드와 Constructor에 대한 빈 인스턴스를 생성하는 메소드가 필요하다.

힌트

```
private Object instantiateClass(Class<?> clazz) {
    [...]

    return null;
}

private Object instantiateConstructor(Constructor<?> constructor) {
    [...]

    return null;
}
```

실습

재귀함수의 시작은instantiateClass()에서 시작한다. @Inject 애노테이션이 설정되어 있는 생성자가 존재하면instantiateConstructor() 메소드를 통해 인스턴스를 생성하고, 존재하지 않을 경우 기본 생성자로 인스턴스를 생성한다.

힌트

instantiateConstructor() 메소드는 생성자의 인자로 전달할 빈이 생성되어Map<Class<?>, Object>에 이미 존재하면 해당 빈을 활용하고, 존재하지 않을 경우instantiateClass() 메소드를 통해 빈을 생성한다.

```
private Object instantiateConstructor(Constructor<?> constructor) {
    Class<?>[] parameterTypes = constructor.getParameterTypes();
    List<Object> args = Lists.newArrayList();
    for (Class<?> clazz : parameterTypes) {
        [...]
        args.add(bean);
    }
    return BeanUtils.instantiateClass(constructor, args.toArray());
}
```

추가 요구사항

—

요구사항

지금까지의 과정을 통해 DI 프레임워크를 완료했다면 다음 단계는 앞에서 구현한 MVC 프레임워크와의 통합이 필요하다. 여기서 구현한 DI 프레임워크를 활용할 경우 앞에서 @Controller이 설정되어 있는 클래스를 찾는 ControllerScanner를 DI 프레임워크가 있는 패키지로 이동해 @Controller, @Service, @Repository에 대한 지원이 가능하도록 개선한다.

클래스 이름도 @Controller 애노테이션만 찾던 역할에서 @Service, @Repository 애노테이션까지 확대 되었으니 BeanScanner로 이름을 리팩토링한다.

MVC 프레임워크의AnnotationHandlerMapping이 BeanFactory와 BeanScanner를 활용해 동작하도록 리팩토링한다.

힌트

이 리팩토링 과정은 생각보다 간단하다. BeanScanner는@Controller, @Service, @Repository이 설정되어 있는 모든 클래스를 찾아 Set에 저장하도록 하면 된다. 기존의 ControllerScanner와 같이 이 단계에서 빈 생성을 담당하지 않아도 된다.

AnnotationHandlerMapping은 @Controller가 설정되어 있는 빈만 관심이 있다. 따라서 BeanFactory에 getControllers() 메소드를 추가해 @Controller가 설정되어 있는 빈 목록을 Map<Class<?>, Object>으로 제공하면 된다.

```
Class<?> clazz = QnaController.class;
Annotation annotation = clazz.getAnnotation(Controller.class);
```