통합 구현(Spring, Django)

IoC와 DI 개념 파악하기

한국기술교육대학교
온라인평생교육원

학습내용

- 관심사와 제어권
- 제어의 역전
- 의존관계 주입

학습목표

- Spring에서 오브젝트의 관심사와 제어권에 대해 파악할 수 있다.
- Spring에서 오브젝트 간의 의존관계를 설명할 수 있다.
- Spring의 원칙과 패턴에 맞는 프로그래밍을 할 수 있다.

관심사와 제어권

1 관심사 파악

이메일을 발송하는 함수 예제

```
public void sendEmail(@RequestParam("to") List<String> to,
@RequestParam("cc") List<String> cc) {
  String subject = generateSubject(); // 제목 생성
  String body = generateBody(); //내용 생성
  Content subjectContent = new Content().withData(subject);
  Body bodyContent = new Body().withHtml(new Content().withData(body));
  Message message = new Message().withSubject(subjectContent)
  .withBody(bodyContent);
  Destination destination = new Destination(); //받는 사람 구성
  if (CollectionUtils.isNotEmpty(to)) {
    destination.withToAddresses(to);
  }
  if (CollectionUtils.isNotEmpty(cc)) {
    destination.withCcAddresses(cc);
  }
  SendEmailRequest sendEmailRequest = new
SendEmailRequest().withSource(source).withDestination(destination)
  .withMessage(message);
  EmailServiceClient emailServiceClient = new EmailServiceClient();
  emailServiceClient.sendEmail(sendEmailRequest); //실제 이메일 발송
}
```

관심사와 제어권

1 관심사 파악

이메일을 발송하는 함수 예제

```
public void sendEmail(@RequestParam("to") List<String> to,
@RequestParam("cc") List<String> cc) {
  String subject = generateSubject(); // 제목 생성
  String body = generateBody(); //내용 생성
  Content subjectContent = new Content().withData(subject);
  Body bodyContent = new Body().withHtml(new Content().withData(body));
  Message message = new Message().withSubject(subjectContent)
  .withBody(bodyContent);
  Destination destination = new Destination(); //받는 사람 구성
  if (CollectionUtils.isNotEmpty(to)) {
    destination.withToAddresses(to);
  }
  if (CollectionUtils.isNotEmpty(cc)) {
    destination.withCcAddresses(cc);
  }
  SendEmailRequest sendEmailRequest = new
SendEmailRequest().withSource(source).withDestination(destination)
  .withMessage(message);
  EmailServiceClient emailServiceClient = new EmailServiceClient();
  emailServiceClient.sendEmail(sendEmailRequest); //실제 이메일 발송
}
```

이메일 메시지 구성

이메일 발송

관심사와 제어권

1 관심사 파악

잘 동작하는 함수이지만 좀 더 효율적으로 수정해 본다면?

이메일 내용 및 수신자를 구성하는 함수를 분리하여 sendEmail함수에서는 순수하게 이메일 발송에만 전념하도록 수정

이 때, sendEmail 함수는 두 개의 관심사인 '이메일 내용 구성', '이메일 전송' 중 '이메일 전송'이라는 본연의 관심사에만 신경을 쓰도록 구성

```
public void sendEmail(SendEmailRequest sendEmailRequest)
  //구성된 이메일을 외부로부터 전달받음
  EmailServiceClient emailServiceClient = new EmailServiceClient();
  emailServiceClient.sendEmail(sendEmailRequest); //실제 이메일 발송
}
```

관심사와 제어권

1 관심사 파악

⟨1⟩ 리펙토링(Refactoring)

1. 외부의 동작방식에는 변화를 주지 않으면서 내부 구조를 효율적으로 변경해서 코드를 재구성
2. 코드 내부 설계가 개선되어 코드를 이해하기가 편해지고 이후 변화에 효율적으로 대응 가능
3. 생산성 및 코드 품질 상승

2 객체의 제어권

객체의 제어권이란?

어떠한 객체가 다른 객체를 생성, 관리하는 권한

- 아래 예제에서는 EmailServiceClient 객체의 생성 및 관리 권한이 sendEmail함수(혹은 sendEmail함수를 가지는 클래스)에 있음

```
public void sendEmail(SendEmailRequest sendEmailRequest)
  EmailServiceClient emailServiceClient = new EmailServiceClient();
  emailServiceClient.sendEmail(sendEmailRequest);
}
```

관심사와 제어권

2 객체의 제어권

- 이렇게 특정 객체에 대한 제어권을 가질 때의 문제점은?

```
@Controller
@RequestMapping(value = "/")
public class EmailController {

    @RequestMapping(value = "/sendEmail")
    @ResponseBody
    public String sendEmail() {
        EmailServiceClient emailServiceClient = new GmailServiceClient();
        SendEmailRequest sendEmailRequest = generateEmailRequest();
        emailServiceClient.sendEmail(sendEmailRequest);
        return "Success";
    }
}
```

EmailController의 sendEmail 함수의 관심사는 이메일을 구성해서 전달하는 것에 있음

EmailServiceClient의 특정 구현 클래스를 선택하고 생성하는 일은 또 다른 관심사에 해당됨

앞서 관심사에 따라 리팩토링한 결과와 상충됨

제어의 역전

1 제어권 이전

⟨1⟩ EmailController에서 '관계설정 기능'의 분리 필요

EmailController의 sendEmail은 이메일 전송이라는 본연의 기능에만 집중

EmailServiceClient의 구체 클래스의 선택 / 생성 불필요

```
@Controller
@RequestMapping(value = "/")
public class EmailController {

    @RequestMapping(value = "/sendEmail")
    @ResponseBody
    public String sendEmail() {
        EmailServiceClient emailServiceClient = new GmailServiceClient();
        SendEmailRequest sendEmailRequest = generateEmailRequest();
        emailServiceClient.sendEmail(sendEmailRequest);
        return "Success";
    }
}
```

⟨2⟩ 제어의 역전(IoC, Inversion of Control)

1. 임의의 객체는 자신이 사용할 다른 객체를 선택/생성하지 않음
2. 객체 스스로가 향후 어떻게 만들어지고 어디서 사용될지 알 수 없음
3. 모든 제어 권한은 다른 객체(상위)에게 위임

제어의 역전

1 제어권 이전

3 IoC의 장점

프로그램 소스 코드의 유연성 및 확장성 증가

IoC는 스프링에서 제공하는 모든 기능의 기초가 되는 기반기술

2 Spring의 IoC

- EmailController가 이미 최상위 클래스인데 EmailServiceClient의 제어권을 상위객체로 넘기려면 어떻게 해야 할까?

```
@Controller
@RequestMapping(value = "/")
public class EmailController {

    @RequestMapping(value = "/sendEmail")
    @ResponseBody
    public String sendEmail() {
        EmailServiceClient emailServiceClient = new GmailServiceClient();
        SendEmailRequest sendEmailRequest = generateEmailRequest();
        emailServiceClient.sendEmail(sendEmailRequest);
        return "Success";
    }
}
```

제어의 역전

2 Spring의 IoC

《1》 Object Factory

Spring의 역할	자체적으로 객체를 생성해서 관리하는 일종의 Factory 역할
Bean	이 때, Spring에서 제어권을 가지고 직접 생성, 관리하는 객체

《2》 Factory Class

1. 구체적으로, Spring은 org.springframework.beans.factory.BeanFactory를 이용해 객체를 생성, 관리할 수 있음
2. 일반적으로는 기능이 추가된, BeanFactory를 상속한 org.springframework.context.ApplicationContext 인터페이스를 통해 어플리케이션 전반에 걸쳐 제어를 담당함

《3》 Application Context의 역할

- IoC를 적용해서 관리할 모든 객체에 대한 제어권 담당
- 코드는 설정정보(Java or XML)를 통해 획득
- Java나 XML 설정파일에 각 Bean에 대한 정보를 기입 후 사용
- Annotation기반으로 동작하기 때문에 각 Bean들의 등록빈도 감소

```
<bean id="emailService" class="com.service.EmailService">
</bean>
```

XML에 Bean 등록 예

제어의 역전

2 Spring의 IoC

4 Application Context 활용 장점

5 Spring에서 관리하는 Bean의 수명

1. Bean은 Spring Container의 컨테이너 내에 한 개의 객체로 생성되며, 이를 Singleton Registry로 관리
2. 강제로 제거하지 않는 한 스프링 컨테이너 내에 계속해서 유지

6 일반적인 클라이언트/서버 구조에서 Singleton Registry를 사용하지 않는다면

클라이언트의 요청 1개당 1개의 객체 생성

즉, 클라이언트 수(요청)가 많아질수록 객체가 많이 생성되며,
이는 서버의 메모리를 차지

서버의 자원 사용량이 매우 높아져서 과부하에 걸릴 수 있음

의존관계 주입

1 의존관계 설정

의존관계(Dependency)

"A가 B에 의존한다"는 것으로 방향성이 있음

의존관계 주입
(DI, Dependency Injection)

- 스프링에서 객체 간의 관계설정 의도를 명확히 표현하는 용어
- 스프링을 다른 프레임워크와 차별화해 제공하는 기능은 DI라는 용어를 사용할 때 분명하게 드러남

스프링에서의 의존관계 주입 조건

클래스 모델이나 코드에서는 보통 인터페이스를 사용하기 때문에 런타임 시점에 어떤 구체 클래스가 적용될지 알 수 없음

런타임 시점의 의존관계는 Application Context같은 제3의 객체가 결정함

의존관계는 사용할 (의존할) 객체에 대한 레퍼런스를 제3의 객체가 제공 (주입, DI)해 줌으로써 만들어짐

의존관계 주입

2 Annotation을 이용한 DI

@Autowired

Spring Framework에서 지원하는 Dependency 정의 용도의 Annotation으로, Spring Framework에 종속적

@Inject

- JSR-330 표준 Annotation으로 Spring3부터 지원
- 특정 Framework에 종속되지 않은 어플리케이션을 구성하기 위해서는 @Autowired 보다 @Inject를 사용할 것을 권장
- pom.xml에 다음과 같이 추가

```
<!-- @Inject -->
<dependency>
  <groupId>javax.inject</groupId>
  <artifactId>javax.inject</artifactId>
  <version>1</version>
</dependency>
```

의존관계 주입

2 Annotation을 이용한 DI

EmailServiceClient를 주입받아 사용하는 활용 예

```
@Controller
@RequestMapping(value = "/")
public class EmailController {
    @Inject
    private EmailServiceClient emailServiceClient;

    @RequestMapping(value = "/sendEmail")
    @ResponseBody
    public String sendEmail() {
        SendEmailRequest sendEmailRequest = generateEmailRequest();
        emailServiceClient.sendEmail(sendEmailRequest);
        return "Success";
    }
}
```

학습정리

1. 관심사와 제어권

- 관심사 파악
  - 관심이 같은 사항들을 하나의 객체 또는 주변 객체로 모음
  - 관심이 다른 사항들은 가능한 따로 떨어져서 서로 영향을 주지 않도록 분리
- 리팩토링(Refactoring)
  - 외부의 동작방식에는 변화를 주지 않으면서 내부 구조를 효율적으로 변경해서 코드를 재구성
  - 코드 내부 설계가 개선되어 코드를 이해하기가 편해지고 이후 변화에 효율적으로 대응 가능
- 객체의 제어권
  - 객체를 선택/생성/관리하는 권한

2. 제어의 역전

- 제어권의 이전
  - 임의의 객체는 자신이 사용할 다른 객체를 선택/생성하지 않음
  - 객체 스스로가 향후 어떻게 만들어지고 어디서 사용될지 알 수 없음
  - 모든 제어 권한은 다른 객체(상위)에게 위임
- IoC의 장점
  - 프로그램 소스 코드의 유연성 및 확장성 증가
  - IoC는 스프링에서 제공하는 모든 기능의 기초가 되는 기반기술

학습정리

3. 의존관계 주입

- 의존관계 주입(DI, Dependency Injection)
  - 스프링에서 객체간의 관계설정 의도를 명확히 표현하는 용어
  - 스프링을 다른 프레임워크와 차별화하여 제공하는 기능

- 스프링에서의 의존관계 주입 조건
  - 클래스 모델이나 코드에서는 보통 인터페이스를 사용하기 때문에 런타임 시점에 어떤 구체 클래스가 적용될지 모름
  - 런타임 시점의 의존관계는 Application Context와 같은 제3의 객체가 결정함
  - 의존관계는 사용할(의존할) 객체에 대한 레퍼런스를 제3의 객체가 제공(주입, DI)해줌으로써 만들어짐