스프링 핵심원리

섹션 8. 빈 생명주기 콜백

섹션 9. 프로토타입 스코프

빈 생명주기 콜백

애플리케이션 시작시점에 필요한 연결을 미리 수행

애플리케이션 종료 시점에 연결해 놓은 것들을 모두 종료

-> 사용할 객체의 초기화와 종료 작업이 필요

```
public NetworkClient() {
    System.out.println("생성자 호출, url = " + url);
    connect();
    call("초기화 연결 메세지");
}
```

```
public NetworkClient networkClient() {
    NetworkClient networkClient = new NetworkClient();
    networkClient.setUrl("http://hello-spring.dev");
    return networkClient;
}
```

```
생성자 호출, url = null
connect: null
call: null message = 초기화 연결 메세지
```

객체를 생성하는 단계에는 연결정보가 없고,

생성한 후에 setter를 통해서 url이 존재

빈 생명주기 콜백

스프링 빈의 라이프사이클

1. 객체 생성
2. 의존관계 주입

* 생성자를 통한 DI는 예외

-〉 초기화 작업은 의존관계 주입이 모두 완료된 후에 호출되어야 함

빈 생명주기 콜백

스프링 빈의 이벤트 라이프사이클

1. 스프링 컨테이너 생성
2. 스프링 빈 생성
3. 의존관계 주입
4. 초기화 콜백
5. 애플리케이션 로직
6. 소멸 전 콜백
7. 스프링 종료

* 초기화 콜백: 빈이 생성되고, 빈의 의존관계 주입이 완료된 후 호출

* 소멸 전 콜백: 빈이 소멸되기 직전에 호출

빈 생명주기 콜백

+ 객체의 생성과 초기화 분리

생성자는 객체 생성에 필요한 필수 정보를 전달받고, 메모리를 할당해서 생성하는 역할이 메인

초기화는 생성된 객체를 활용해서 외부 동작을 수행

-〉 단일책임원칙에 맞추어 두 과정을 분리하는 걸 권장

빈 생명주기 콜백: 인터페이스(InitializingBean, DisposableBean)

스프링 프레임워크가 제공하는 두 가지 인터페이스를 사용

- InitialzingBean: 빈 객체 생성, 의존관계 주입 후 초기화를 위해
- DisposableBean: 빈 소멸 직전 종료를 위해

```java
public class NetworkClient implements InitializingBean, DisposableBean {

    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("NetworkClient.afterPropertiesSet");
        connect();
        call("초기화 연결 메세지");
    }

    @Override
    public void destroy() throws Exception {
        System.out.println("NetworkClient.destroy");
        disconnect();
    }
}
```

빈 생명주기 콜백: 인터페이스(InitializingBean, DisposableBean)

```
생성자 호출, url = null
connect: null
call: null message = 초기화 연결 메세지
```

```
생성자 호출, url = null
NetworkClient.afterPropertiesSet
connect: http://hello-spring.dev
call: http://hello-spring.dev message = 초기화 연결 메세지
NetworkClient.destroy
close http://hello-spring.dev
```

인터페이스 활용 시 단점

- 스프링에서 제공하는 인터페이스로, 코드 레벨 구현이 스프링에 의존적
- afterPropertiesSet(), destory() 메서드 이름 변경 불가
- 외부 라이브러리 사용할 때 적용 불가

-〉 현재는 거의 쓰이지 않는 방법

빈 생명주기 콜백: 빈 등록 초기화, 소멸 메서드 지정

```java
@Bean(initMethod = "init", destroyMethod = "close")
public NetworkClient networkClient() {
    NetworkClient networkClient = new NetworkClient();
    networkClient.setUrl("http://hello-spring.dev");
    return networkClient;
}
```

```java
public void init() throws Exception {
    System.out.println("NetworkClient.init");
    connect();
    call("초기화 연결 메세지");
}

public void close() throws Exception {
    System.out.println("NetworkClient.close");
    disconnect();
}
```

빈 생명주기 콜백: 빈 등록 초기화, 소멸 메서드 지정

설정 정보 활용 시 장점

- 메서드 이름을 자유롭게 지정 가능
- 스프링 코드에 의존 X
- 코드 레벨의 수정이 아닌 빈 설정 정보를 사용하기 때문에 외부 라이브러리에서도 적용 가능

종료 메서드 추론 기능

- 일반적으로 “close”, “shutdown” 이름으로 종료 메서드를 구현
- @Bean의 destroyMethod는 메서드 이름을 추론해서 알아서 동작

빈 생명주기 콜백: 애노테이션 @PostConstruct, @PreDestroy

```java
@Bean
public NetworkClient networkClient() {
    NetworkClient networkClient = new NetworkClient();
    networkClient.setUrl("http://hello-spring.dev");
    return networkClient;
}
```

```java
@PostConstruct
public void init() {
    System.out.println("NetworkClient.init");
    connect();
    call("초기화 연결 메세지");
}

@PreDestroy
public void close() {
    System.out.println("NetworkClient.close");
    disconnect();
}
```

빈 생명주기 콜백: 애노테이션 @PostConstruct, @PreDestroy

애노테이션 활용 시 특징

- 스프링에서 권장하는 방법
- 스프링 종속 기술이 아님 (javax.xxx 패키지를 import)
- 코드 레벨 수정이 필요하기 때문에 외부 라이브러리에 적용 불가

정리

- 일반적으로 스프링을 이용할 때는 애노테이션 방법 이용
- 코드 수정이 불가하다면 빈 설정 정보 활용 방법 이용

빈 스코프

빈 스코프: 빈이 존재할 수 있는 범위

빈 스코프 종류

- 싱글톤 스코프
  - 스프링 컨테이너 시작~종료까지 유지
  - 앞서 구현한 예시의 대부분에 해당
- 프로토타입 스코프
  - 해당 스코프의 빈 요청이 들어오면 생성과 의존관계 주입까지만 관여
- 웹 관련 스코프
  - request: 요청이 들어오고 나갈 때까지 유지
  - session: 세션이 생성되고 종료될 때까지 유지
  - application: 서블릿 컨텍스와 같은 범위로 유지

빈 스코프: 프로토타입 스코프

싱글톤 빈

- 해당 스코프의 빈을 조회하면 스프링 컨테이너는 항상 같은 객체를 반환
- 애플리케이션 최초 실행 시(스프링 컨테이너 시작) 빈으로 등록됨

프로토타입 빈

- 매 요청마다 다른 객체를 반환
- 해당 스코프의 빈을 요청하는 시점에 빈을 생성 및 의존관계 주입
- 클라이언트에 반환 후 스프링 컨테이너가 더 이상 관리 X
- @PreDestroy 등의 종료 메서드 자동 호출 X (클라이언트 책임)

빈 스코프: 프로토타입 스코프

```
AnnotationConfigApplicationContext ac = new AnnotationConfigApplicationContext(SingletonBean.class);

SingletonBean singletonBean1 = ac.getBean(SingletonBean.class);
SingletonBean singletonBean2 = ac.getBean(SingletonBean.class);
System.out.println("singletonBean1 = " + singletonBean1);
System.out.println("singletonBean2 = " + singletonBean2);
assertThat(singletonBean1).isSameAs(singletonBean2);

ac.close();
```

```
07:14:28.991 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean
 'singletonTest.SingletonBean'
SingletonBean.init
singletonBean1 = hello.core.scope.SingletonTest$SingletonBean@14f9390f
singletonBean2 = hello.core.scope.SingletonTest$SingletonBean@14f9390f
07:14:29.087 [main] DEBUG org.springframework.context.annotation.AnnotationConfigApplicationContext - Closing org.springframework.context.annotation
 .AnnotationConfigApplicationContext@6bedbc4d, started on Wed Nov 04 07:14:28 KST 2020
SingletonBean.destroy
```

빈 스코프: 프로토타입 스코프

```
AnnotationConfigApplicationContext ac = new AnnotationConfigApplicationContext(PrototypeBean.class);

System.out.println("find prototypeBean1");
PrototypeBean prototypeBean1 = ac.getBean(PrototypeBean.class);

System.out.println("find prototypeBean2");
PrototypeBean prototypeBean2 = ac.getBean(PrototypeBean.class);

System.out.println("prototypeBean1 = " + prototypeBean1);
System.out.println("prototypeBean2 = " + prototypeBean2);
assertThat(prototypeBean1).isNotSameAs(prototypeBean2);

ac.close();
```

```
find prototypeBean1
PrototypeBean.init
find prototypeBean2
PrototypeBean.init
prototypeBean1 = hello.core.scope.PrototypeTest$PrototypeBean@14f9390f
prototypeBean2 = hello.core.scope.PrototypeTest$PrototypeBean@6c0d7c83
07:21:36.846 [main] DEBUG org.springframework.context.annotation.AnnotationConfigApplicationContext - Closing org.springframework.context.annotation
 .AnnotationConfigApplicationContext@6bedbc4d, started on Wed Nov 04 07:21:36 KST 2020
```

빈 스코프: 프로토타입 스코프 문제점

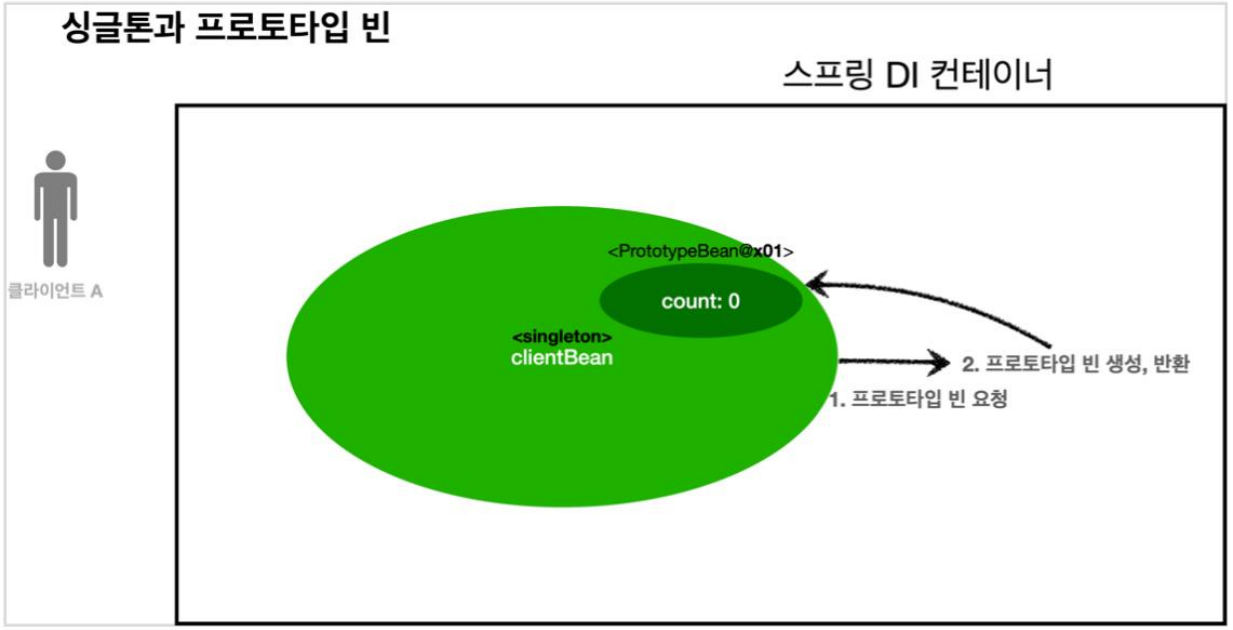
싱글톤 빈과 프로토타입 빈을 함께 사용하는 경우

싱글톤 빈 생성시점에 의존관계 주입으로 프로토타입 빈이 생성됨

```
static class ClientBean {
    private final PrototypeBean prototypeBean;

    @Autowired
    public ClientBean(PrototypeBean prototypeBean) {
        this.prototypeBean = prototypeBean;
    }

    public int logic() {
        prototypeBean.addCount();
        return prototypeBean.getCount();
    }
}
```



빈 스코프: 프로토타입 스코프 문제 해결

단순 해결책으로, 프로토타입 빈 객체가 필요할 때 스프링 컨테이너에서 직접 가져오도록 구현

```
static class ClientBean {
    public int logic() {
        System.out.println("logic start");
        AnnotationConfigApplicationContext ac = new AnnotationConfigApplicationContext(PrototypeBean.class);
        PrototypeBean bean = ac.getBean(PrototypeBean.class);
        bean.addCount();
        return bean.getCount();
    }
}
```

스프링 컨테이너에 지나치게 종속, 단위 테스트 어려움

필요한 기능은 스프링 컨테이너에서 의존관계를 찾는 Dependency Lookup

빈 스코프: 프로토타입 스코프 문제 해결

지정한 빈을 컨테이너에서 대신 찾아주는 (DL 서비스 제공) ObjectProvider 이용

과거의 ObjectFactory에 편의 기능을 추가

```
static class ClientBean {
    @Autowired
    private ObjectProvider<PrototypeBean> prototypeBeanObjectProvider;

    public int logic() {
        System.out.println("logic start");
        PrototypeBean object = prototypeBeanObjectProvider.getObject();
        object.addCount();
        return object.getCount();
    }
}
```

```
logic start
PrototypeBean.inithello.core.scope.SingletonWithPrototypeTest1$PrototypeBean@b2c5e07
logic start
PrototypeBean.inithello.core.scope.SingletonWithPrototypeTest1$PrototypeBean@4426bff1
```

빈 스코프: 프로토타입 스코프 문제 해결

자바 표준의 Provider로도 DL 기능 동작 가능

스프링이 아닌 다른 컨테이너에서도 사용 가능

딱 DL 정도의 기능만 제공

```
static class ClientBean {
    @Autowired
    private Provider<PrototypeBean> prototypeBeanObjectProvider;

    public int logic() {
        System.out.println("logic start");
        PrototypeBean object = prototypeBeanObjectProvider.get();
        object.addCount();
        return object.getCount();
    }
}
```

빈 스코프: 프로토타입 스코프 문제 해결

특별히 다른 컨테이너를 사용할 일이 없다면 스프링이 제공하는 provider 기능을 사용하는 걸 권장

- 대부분 스프링이 제공하는 기능이 더 편리하고 다양

이외의 컨테이너를 사용한다면 자바 표준의 provider 기능을 사용

빈 스코프: 웹 스코프

프로토타입과는 다르게 웹 스코프의 종료시점까지 컨테이너에서 관리

웹 스코프 종류

- request: 각 HTTP 요청마다 들어오고 나갈 때까지 유지
- session: HTTP Session가 시작되고 종료될 때까지 유지
- application: ServletContext가 시작되고 종료될 때까지 유지

*Servlet: 자바를 이용하여 웹 상의 클라이언트가 요청을 하면 그에 대한 결과를 반환하는 기술 (ex. Tomcat)

빈 스코프: request 스코프

스프링 부트 라이브러리를 추가해서 내장 톰켓 서버로 웹 서버와 스프링을 함께 실행

```java
@Controller
@RequiredArgsConstructor
public class LogDemoController {

    private final LogDemoService logDemoService;
    private final MyLogger myLogger;

    @RequestMapping("log-demo")
    @ResponseBody
    public String logDemo(HttpServletRequest request) {
        String requestURL = request.getRequestURL().toString();
        myLogger.setRequestURL(requestURL);

        myLogger.log("controller test");
        logDemoService.logic( id: "test id");
        return "OK";
    }

}
```

```java
@Component
@Scope(value = "request")
public class MyLogger {

    private String uuid;
    private String requestURL;

    public void setRequestURL(String requestURL) { this.requestURL = requestURL; }

    public void log(String message) {...}

    @PostConstruct
    public void init() {...}

    @PreDestroy
    public void destroy() { System.out.println("[" + this.uuid + "] request scope bean close: " + this); }
}
```

실행하면 LogDemoController 빈 스코프가 싱글톤이므로

초기에 등록이 되고, MyLogger를 주입받아야 하는데

MyLogger 빈 스코프가 request이어서 오류 발생

빈 스코프: request 스코프 문제 해결 - provider

ObjectProvider를 사용해서 request 스코프의 빈 생성을 지연

```java
@Controller
@RequiredArgsConstructor
public class LogDemoController {

    private final LogDemoService logDemoService;
    private final ObjectProvider<MyLogger> myLoggerObjectProvider;

    @RequestMapping("log-demo")
    @ResponseBody
    public String logDemo(HttpServletRequest request) {
        MyLogger myLogger = myLoggerObjectProvider.getObject();
        String requestURL = request.getRequestURL().toString();
        myLogger.setRequestURL(requestURL);

        myLogger.log("controller test");
        logDemoService.logic( id: "test id");
        return "OK";
    }

}
```

```java
@Service
@RequiredArgsConstructor
public class LogDemoService {

    private final ObjectProvider<MyLogger> myLoggerObjectProvider;

    public void logic(String id) {
        MyLogger myLogger = myLoggerObjectProvider.getObject();
        myLogger.log("service id = " + id);
    }
}
```

빈 스코프: request 스코프 문제 해결 - provider

spring

각 request마다 매번 객체를 생성하고 의존관계 주입 후 반환

```
[bc0088eb-ceb3-4090-9fe0-3b9b3aacd1aa] request scope bean create: hello.core.common.MyLogger@15bc0efb
[bc0088eb-ceb3-4090-9fe0-3b9b3aacd1aa][http://localhost:8080/log-demo] controller test
[bc0088eb-ceb3-4090-9fe0-3b9b3aacd1aa][http://localhost:8080/log-demo] service id = test id
[bc0088eb-ceb3-4090-9fe0-3b9b3aacd1aa] request scope bean close: hello.core.common.MyLogger@15bc0efb
[0a3ecab6-05bd-490e-aba8-8836f71b793b] request scope bean create: hello.core.common.MyLogger@49355a3e
[0a3ecab6-05bd-490e-aba8-8836f71b793b][http://localhost:8080/log-demo] controller test
[0a3ecab6-05bd-490e-aba8-8836f71b793b][http://localhost:8080/log-demo] service id = test id
[0a3ecab6-05bd-490e-aba8-8836f71b793b] request scope bean close: hello.core.common.MyLogger@49355a3e
```

빈 스코프: request 스코프 문제 해결 - proxy

proxy를 활용하여 가짜 객체를 주입하도록 구현

적용 대상에 맞추어 TARGET_CLASS, INTERFACES를 선택

```
@Component
@Scope(value = "request", proxyMode = ScopedProxyMode.TARGET_CLASS)
public class MyLogger {

    private String uuid;
    private String requestURL;

    public void setRequestURL(String requestURL) { this.requestURL = requestURL; }

    public void log(String message) {...}

    @PostConstruct
    public void init() {...}

    @PreDestroy
    public void destroy() { System.out.println("[" + this.uuid + "] request scope bean close: " + this); }
}
```

빈 스코프: request 스코프 문제 해결 - proxy

적용 대상의 가짜 프록스 클래스를 만든 후 HTTP request와 상관없이 다른 빈에 주입 가능

-〉 싱글톤 빈에서 request 빈을 주입해도 가짜 객체가 들어가므로 에러 없이 동작

동작 원리

- 바이트코드를 조작하는 CGLIB 라이브러리로 가짜 프록시 객체를 생성
- 스프링 컨테이너에도 프록시 객체가 들어감
- 진짜 빈 요청이 오면 프록시 객체 내부에서 진짜 빈을 찾는 위임 로직 포함

-〉 클라이언트 입장에서 원본인 지, 가짜인 지 모르고 동일하게 사용 가능 (다형성)