8. 빈 생명주기 콜백

#2.인강/2.핵심원리 - 기본편/기본편#

목차

빈 생명주기 콜백 시작

데이터베이스 커넥션 풀이나, 네트워크 소켓처럼 애플리케이션 시작 시점에 필요한 연결을 미리 해두고, 애플리케이션 종료 시점에 연결을 모두 종료하는 작업을 진행하려면, 객체의 초기화와 종료 작업이 필요하다.
이번시간에는 스프링을 통해 이러한 초기화 작업과 종료 작업을 어떻게 진행하는지 예제로 알아보자.

간단하게 외부 네트워크에 미리 연결하는 객체를 하나 생성한다고 가정해보자. 실제로 네트워크에 연결하는 것은 아니고, 단순히 문자만 출력하도록 했다. 이 `NetworkClient`는 애플리케이션 시작 시점에 `connect()`를 호출해서 연결을 맺어두어야 하고, 애플리케이션이 종료되면 `disConnect()`를 호출해서 연결을 끊어야 한다.

예제 코드, 테스트 하위에 생성

```
package hello.core.lifecycle;

public class NetworkClient {

    private String url;

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

```
    public void setUrl(String url) {
        this.url = url;
    }

    //서비스 시작시 호출
    public void connect() {
        System.out.println("connect: " + url);
    }

    public void call(String message) {
        System.out.println("call: " + url + " message = " + message);
    }

    //서비스 종료시 호출
    public void disconnect() {
        System.out.println("close: " + url);
    }
}
```

스프링 환경설정과 실행

```
package hello.core.lifecycle;

import org.junit.jupiter.api.Test;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

public class BeanLifeCycleTest {

    @Test
    public void lifeCycleTest() {
        ConfigurableApplicationContext ac = new
AnnotationConfigApplicationContext(LifeCycleConfig.class);
        NetworkClient client = ac.getBean(NetworkClient.class);
```

```
        ac.close(); //스프링 컨테이너를 종료, ConfigurableApplicationContext 필요

    }

    @Configuration
    static class LifeCycleConfig {

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

실행해보면 다음과 같은 이상한 결과가 나온다.

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

생성자 부분을 보면 url 정보 없이 connect가 호출되는 것을 확인할 수 있다.
너무 당연한 이야기이지만 객체를 생성하는 단계에는 url이 없고, 객체를 생성한 다음에 외부에서 수정자 주입을 통해서 `setUrl()`이 호출되어야 url이 존재하게 된다.

스프링 빈은 간단하게 다음과 같은 라이프사이클을 가진다.
객체 생성 → 의존관계 주입

스프링 빈은 객체를 생성하고, 의존관계 주입이 다 끝난 다음에야 필요한 데이터를 사용할 수 있는 준비가 완료된다. 따라서 초기화 작업은 의존관계 주입이 모두 완료되고 난 다음에 호출해야 한다. 그런데 개발자가 의존관계 주입이 모두 완료된 시점을 어떻게 알 수 있을까?
스프링은 의존관계 주입이 완료되면 스프링 빈에게 콜백 메서드를 통해서 초기화 시점을 알려주는 다양한 기능을 제공한다. 또한 **스프링은 스프링 컨테이너가 종료되기 직전에 소멸 콜백**을 준다. 따라서 안전하게 종료 작업을 진행할 수 있다.

스프링 빈의 이벤트 라이프사이클
스프링 컨테이너 생성 → 스프링 빈 생성 → 의존관계 주입 → 초기화 콜백 → 사용 → 소멸전 콜백 → 스프링 종료

- **초기화 콜백**: 빈이 생성되고, 빈의 의존관계 주입이 완료된 후 호출
- **소멸전 콜백**: 빈이 소멸되기 직전에 호출

스프링은 다양한 방식으로 생명주기 콜백을 지원한다.

> **참고: 객체의 생성과 초기화를 분리하자.**
> 생성자는 필수 정보(파라미터)를 받고, 메모리를 할당해서 객체를 생성하는 책임을 가진다. 반면에 초기화는 이렇게 생성된 값들을 활용해서 외부 커넥션을 연결하는등 무거운 동작을 수행한다.
> 따라서 생성자 안에서 무거운 초기화 작업을 함께 하는 것 보다는 객체를 생성하는 부분과 초기화 하는 부분을 명확하게 나누는 것이 유지보수 관점에서 좋다. 물론 초기화 작업이 내부 값들만 약간 변경하는 정도로 단순한 경우에는 생성자에서 한번에 다 처리하는게 더 나을 수 있다.

> **참고:** 싱글톤 빈들은 스프링 컨테이너가 종료될 때 싱글톤 빈들도 함께 종료되기 때문에 스프링 컨테이너가 종료되기 직전에 소멸전 콜백이 일어난다. 뒤에서 설명하겠지만 싱글톤 처럼 컨테이너의 시작과 종료까지 생존하는 빈도 있지만, 생명주기가 짧은 빈들도 있는데 이 빈들은 컨테이너와 무관하게 해당 빈이 종료되기 직전에 소멸전 콜백이 일어난다. 자세한 내용은 스코프에서 알아보겠다.

스프링은 크게 3가지 방법으로 빈 생명주기 콜백을 지원한다.
- 인터페이스(InitializingBean, DisposableBean)
- 설정 정보에 초기화 메서드, 종료 메서드 지정
- @PostConstruct, @PreDestroy 애노테이션 지원

하나씩 알아보자.

인터페이스 InitializingBean, DisposableBean

코드를 바로 보자

```
package hello.core.lifecycle;


import org.springframework.beans.factory.DisposableBean;
```

```java
import org.springframework.beans.factory.InitializingBean;

public class NetworkClient implements InitializingBean, DisposableBean {

    private String url;

    public NetworkClient() {
        System.out.println("생성자 호출, url = " + url);
    }

    public void setUrl(String url) {
        this.url = url;
    }

    //서비스 시작시 호출
    public void connect() {
        System.out.println("connect: " + url);
    }

    public void call(String message) {
        System.out.println("call: " + url + " message = " + message);
    }

    //서비스 종료시 호출
    public void disConnect() {
        System.out.println("close + " + url);
    }

    @Override
    public void afterPropertiesSet() throws Exception {
        connect();
        call("초기화 연결 메시지");
    }

    @Override
    public void destroy() throws Exception {
        disConnect();
    }
}
```

```

```

- `InitializingBean`은 `afterPropertiesSet()` 메서드로 초기화를 지원한다.
- `DisposableBean`은 `destroy()` 메서드로 소멸을 지원한다.

출력 결과

```
생성자 호출, url = null
NetworkClient.afterPropertiesSet
connect: http://hello-spring.dev
call: http://hello-spring.dev message = 초기화 연결 메시지
13:24:49.043 [main] DEBUG
org.springframework.context.annotation.AnnotationConfigApplicationContext -
Closing NetworkClient.destroy
close + http://hello-spring.dev
```

- 출력 결과를 보면 초기화 메서드가 주입 완료 후에 적절하게 호출 된 것을 확인할 수 있다.
- 그리고 스프링 컨테이너의 종료가 호출되자 소멸 메서드가 호출 된 것도 확인할 수 있다.

초기화, 소멸 인터페이스 단점

- 이 인터페이스는 스프링 전용 인터페이스다. 해당 코드가 스프링 전용 인터페이스에 의존한다.
- 초기화, 소멸 메서드의 이름을 변경할 수 없다.
- 내가 코드를 고칠 수 없는 외부 라이브러리에 적용할 수 없다.

> 참고: 인터페이스를 사용하는 초기화, 종료 방법은 스프링 초창기에 나온 방법들이고, 지금은 다음의 더 나은 방법들이 있어서 거의 사용하지 않는다.

빈 등록 초기화, 소멸 메서드 지정

설정 정보에 `@Bean(initMethod = "init", destroyMethod = "close")`처럼 초기화, 소멸 메서드를 지정할 수 있다.

설정 정보를 사용하도록 변경

```java
package hello.core.lifecycle;

public class NetworkClient {

    private String url;

    public NetworkClient() {
        System.out.println("생성자 호출, url = " + url);
    }

    public void setUrl(String url) {
        this.url = url;
    }

    //서비스 시작시 호출
    public void connect() {
        System.out.println("connect: " + url);
    }

    public void call(String message) {
        System.out.println("call: " + url + " message = " + message);
    }

    //서비스 종료시 호출
    public void disConnect() {
        System.out.println("close + " + url);
    }

    public void init() {
        System.out.println("NetworkClient.init");
        connect();
        call("초기화 연결 메시지");
    }

    public void close() {
        System.out.println("NetworkClient.close");
        disConnect();
    }
```

```
}
```

설정 정보에 초기화 소멸 메서드 지정

```
@Configuration
static class LifeCycleConfig {

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

결과

```
생성자 호출, url = null
NetworkClient.init
connect: http://hello-spring.dev
call: http://hello-spring.dev message = 초기화 연결 메시지
13:33:10.029 [main] DEBUG
org.springframework.context.annotation.AnnotationConfigApplicationContext -
Closing NetworkClient.close
close + http://hello-spring.dev
```

설정 정보 사용 특징

- 메서드 이름을 자유롭게 줄 수 있다.
- 스프링 빈이 스프링 코드에 의존하지 않는다.
- 코드가 아니라 설정 정보를 사용하기 때문에 코드를 고칠 수 없는 외부 라이브러리에도 초기화, 종료 메서드를 적용할 수 있다.

종료 메서드 추론

- `@Bean의 destroyMethod` 속성에는 아주 특별한 기능이 있다.
- 라이브러리는 대부분 `close`, `shutdown` 이라는 이름의 종료 메서드를 사용한다.
- @Bean의 `destroyMethod` 는 기본값이 `(inferred)` (추론)으로 등록되어 있다.
- 이 추론 기능은 `close`, `shutdown` 라는 이름의 메서드를 자동으로 호출해준다. 이름 그대로 종료 메서드를 추론해서 호출해준다.
- 따라서 직접 스프링 빈으로 등록하면 종료 메서드는 따로 적어주지 않아도 잘 동작한다.
- 추론 기능을 사용하기 싫으면 `destroyMethod=""` 처럼 빈 공백을 지정하면 된다.

애노테이션 @PostConstruct, @PreDestroy

우선 코드 먼저 보고 설명하겠다.

```
package hello.core.lifecycle;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

public class NetworkClient {

    private String url;

    public NetworkClient() {
        System.out.println("생성자 호출, url = " + url);
    }

    public void setUrl(String url) {
        this.url = url;
    }

    //서비스 시작시 호출
    public void connect() {
        System.out.println("connect: " + url);
    }

    public void call(String message) {
        System.out.println("call: " + url + " message = " + message);
```

```
    }

    //서비스 종료시 호출
    public void disConnect() {
        System.out.println("close + " + url);
    }

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

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

}
```

```
@Configuration
static class LifeCycleConfig {

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

실행 결과

```
생성자 호출, url = null
NetworkClient.init
connect: http://hello-spring.dev
call: http://hello-spring.dev message = 초기화 연결 메시지
19:40:50.269 [main] DEBUG
org.springframework.context.annotation.AnnotationConfigApplicationContext -
Closing NetworkClient.close
close + http://hello-spring.dev
```

`@PostConstruct` `@PreDestroy` 이 두 애노테이션을 사용하면 가장 편리하게 초기화와 종료를 실행할 수 있다.

@PostConstruct, @PreDestroy 애노테이션 특징

- 최신 스프링에서 가장 권장하는 방법이다.
- 애노테이션 하나만 붙이면 되므로 매우 편리하다.
- 패키지를 잘 보면 `javax.annotation.PostConstruct` 이다. 스프링에 종속적인 기술이 아니라 JSR-250 라는 자바 표준이다. 따라서 스프링이 아닌 다른 컨테이너에서도 동작한다.
- 컴포넌트 스캔과 잘 어울린다.
- 유일한 단점은 외부 라이브러리에는 적용하지 못한다는 것이다. 외부 라이브러리를 초기화, 종료 해야 하면 @Bean의 기능을 사용하자.

정리

- **@PostConstruct, @PreDestroy 애노테이션을 사용하자**
- 코드를 고칠 수 없는 외부 라이브러리를 초기화, 종료해야 하면 `@Bean` 의 `initMethod`, `destroyMethod` 를 사용하자.