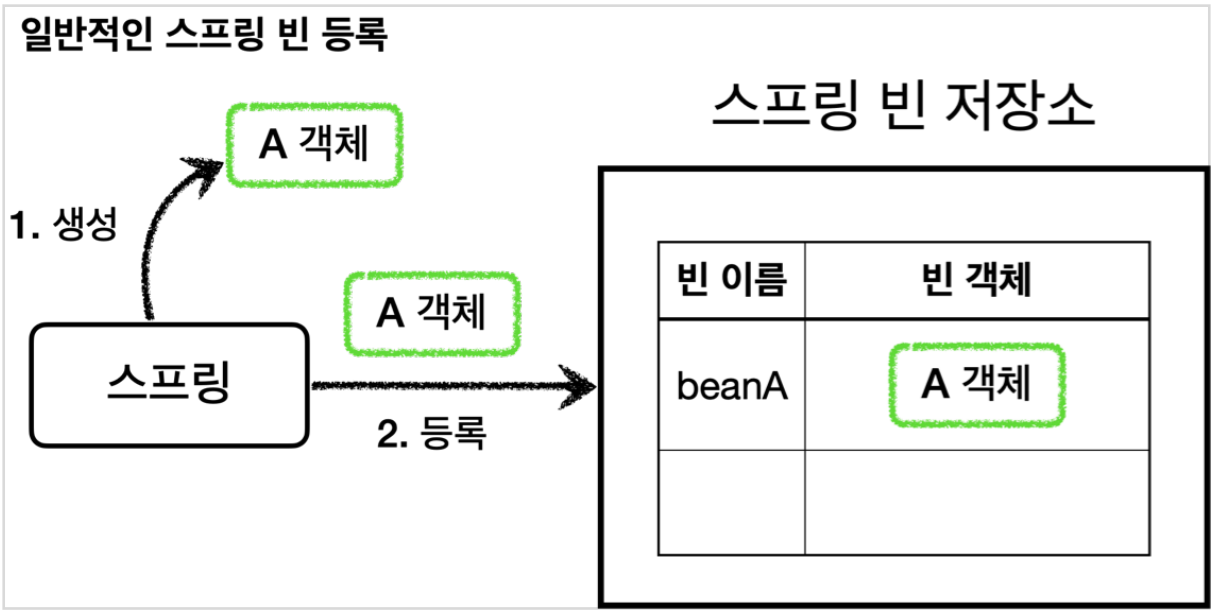
7. 빈 후처리기

#인강/6.핵심 원리 - 고급편/강의#

빈 후처리기 - 소개



`@Bean`이나 컴포넌트 스캔으로 스프링 빈을 등록하면, 스프링은 대상 객체를 생성하고 스프링 컨테이너 내부의 빈 저장소에 등록한다. 그리고 이후에는 스프링 컨테이너를 통해 등록한 스프링 빈을 조회해서 사용하면 된다.

빈 후처리기 - BeanPostProcessor

스프링이 빈 저장소에 등록할 목적으로 생성한 객체를 빈 저장소에 등록하기 직전에 조작하고 싶다면 빈 후처리기를 사용하면 된다.

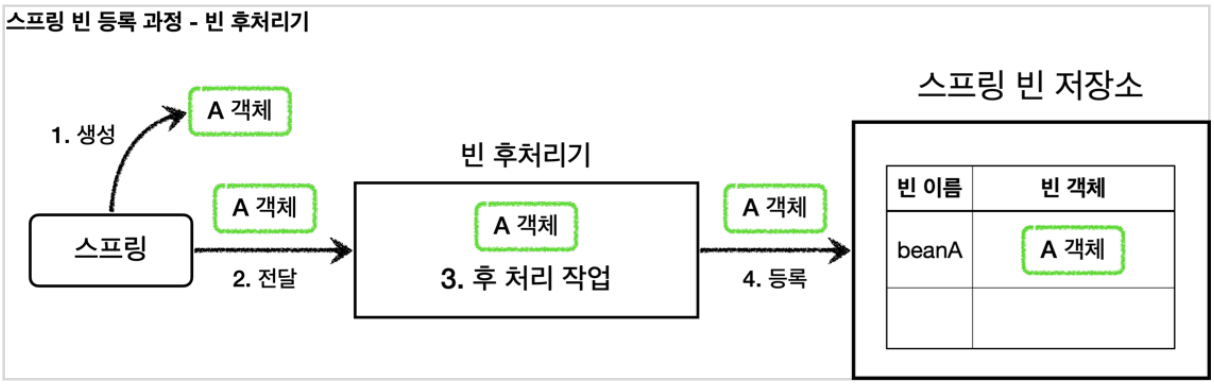
빈 포스트 프로세서(`BeanPostProcessor`)는 번역하면 빈 후처리기인데, 이름 그대로 빈을 생성한 후에 무언가를 처리하는 용도로 사용한다.

빈 후처리기 기능

빈 후처리기의 기능은 막강하다.

객체를 조작할 수도 있고, 완전히 다른 객체로 바꿔치기 하는 것도 가능하다.

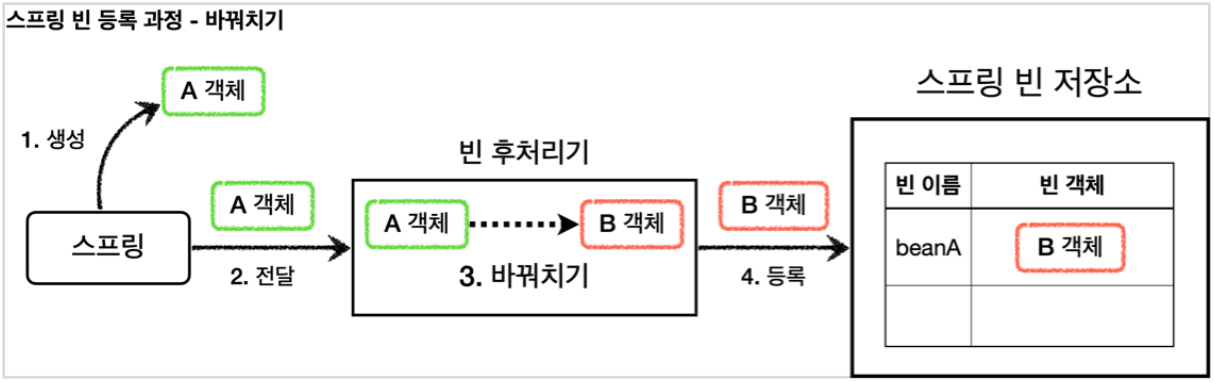
빈 후처리기 과정



빈 등록 과정을 빈 후처리기와 함께 살펴보자

- **1. 생성:** 스프링 빈 대상이 되는 객체를 생성한다. (`@Bean` , 컴포넌트 스캔 모두 포함)
- **2. 전달:** 생성된 객체를 빈 저장소에 등록하기 직전에 빈 후처리기에 전달한다.
- **3. 후 처리 작업:** 빈 후처리기는 전달된 스프링 빈 객체를 조작하거나 다른 객체로 바뀌치기 할 수 있다.
- **4. 등록:** 빈 후처리기는 빈을 반환한다. 전달 된 빈을 그대로 반환하면 해당 빈이 등록되고, 바꿔치기 하면 다른 객체가 빈 저장소에 등록된다.

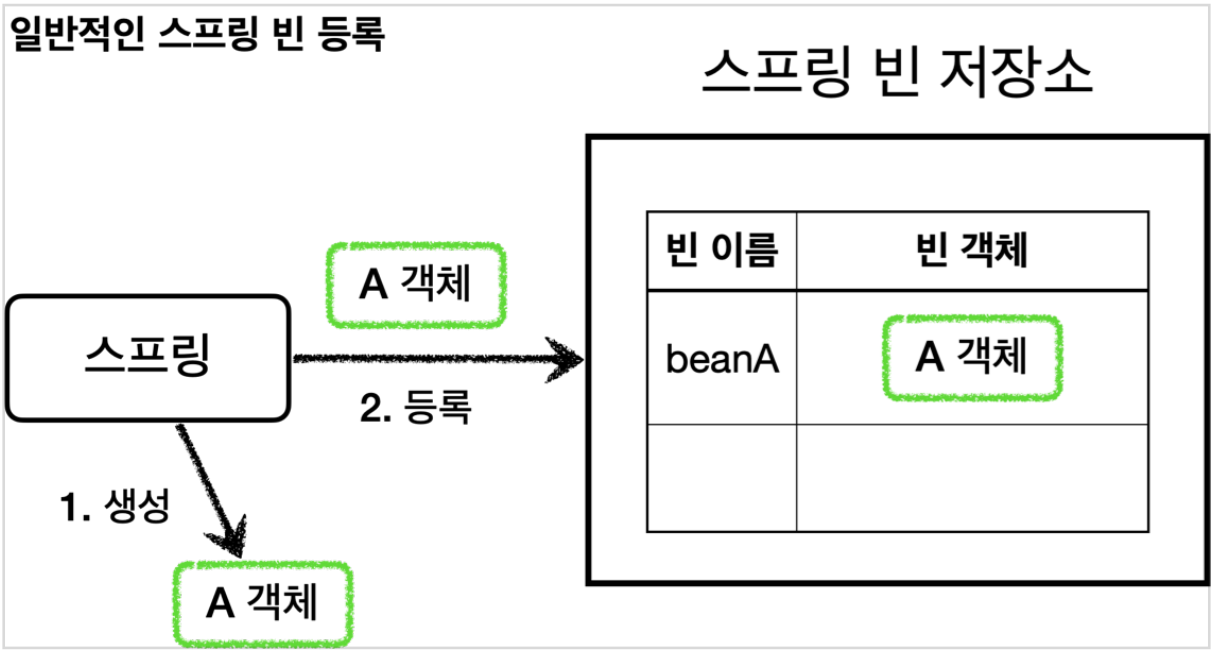
다른 객체로 바꿔치는 빈 후처리기



빈 후처리기 - 예제 코드1

일반적인 스프링 빈 등록 과정

빈 후처리기를 학습하기 전에 먼저 일반적인 스프링 빈 등록 과정을 코드로 작성해보자.



BasicTest

```
package hello.proxy.postprocessor;

import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.NoSuchBeanDefinitionException;
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 BasicTest {

    @Test
    void basicConfig() {
        ApplicationContext applicationContext = new
AnnotationConfigApplicationContext(BasicConfig.class);

        //A는 빈으로 등록된다.
```

```
        A a = applicationContext.getBean("beanA", A.class);
        a.helloA();

        //B는 빈으로 등록되지 않는다.
        Assertions.assertThrows(NoSuchBeanDefinitionException.class,
                () -> applicationContext.getBean(B.class));
    }

    @Slf4j
    @Configuration
    static class BasicConfig {
        @Bean(name = "beanA")
        public A a() {
            return new A();
        }
    }

    @Slf4j
    static class A {
        public void helloA() {
            log.info("hello A");
        }
    }

    @Slf4j
    static class B {
        public void helloB() {
            log.info("hello B");
        }
    }

}
```

`new AnnotationConfigApplicationContext(BasicConfig.class)`
스프링 컨테이너를 생성하면서 `BasicConfig.class`를 넘겨주었다. `BasicConfig.class` 설정 파일은 스프링 빈으로 등록된다.

등록

`BasicConfig.class`

```
@Bean(name = "beanA")
public A a() {
    return new A();
}
```

`beanA` 라는 이름으로 `A` 객체를 스프링 빈으로 등록했다.

조회

`A a = applicationContext.getBean("beanA", A.class)`

- `beanA` 라는 이름으로 `A` 타입의 스프링 빈을 찾을 수 있다.

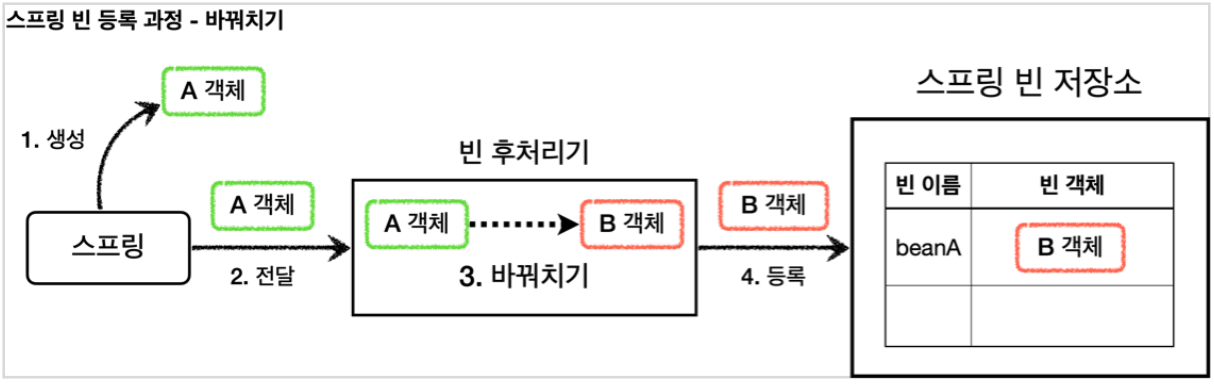
`applicationContext.getBean(B.class)`

- `B` 타입의 객체는 스프링 빈으로 등록한 적이 없기 때문에 스프링 컨테이너에서 찾을 수 없다.

빈 후처리기 - 예제 코드2

빈 후처리기 적용

이번에는 빈 후처리기를 통해서 A 객체를 B 객체로 바꿔치기 해보자.



BeanPostProcessor 인터페이스 - 스프링 제공

```
public interface BeanPostProcessor {
    Object postProcessBeforeInitialization(Object bean, String beanName) throws
```

```
BeansException
    Object postProcessAfterInitialization(Object bean, String beanName) throws
BeansException
}
```

- 빈 후처리기를 사용하려면 `BeanPostProcessor` 인터페이스를 구현하고, 스프링 빈으로 등록하면 된다.
- `postProcessBeforeInitialization` : 객체 생성 이후에 `@PostConstruct` 같은 초기화가 발생하기 전에 호출되는 포스트 프로세서이다.
- `postProcessAfterInitialization` : 객체 생성 이후에 `@PostConstruct` 같은 초기화가 발생한 다음에 호출되는 포스트 프로세서이다.

BeanPostProcessorTest

```
package hello.proxy.postprocessor;

import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.NoSuchBeanDefinitionException;
import org.springframework.beans.factory.config.BeanPostProcessor;
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 BeanPostProcessorTest {

    @Test
    void postProcessor() {
        ApplicationContext applicationContext = new
AnnotationConfigApplicationContext(BeanPostProcessorConfig.class);

        //beanA 이름으로 B 객체가 빈으로 등록된다.
        B b = applicationContext.getBean("beanA", B.class);
        b.helloB();

        //A는 빈으로 등록되지 않는다.
```

```
        Assertions.assertThrows(NoSuchBeanDefinitionException.class,
                () -> applicationContext.getBean(A.class));
    }

    @Slf4j
    @Configuration
    static class BeanPostProcessorConfig {

        @Bean(name = "beanA")
        public A a() {
            return new A();
        }

        @Bean
        public AToBPostProcessor helloPostProcessor() {
            return new AToBPostProcessor();
        }
    }

    @Slf4j
    static class A {
        public void helloA() {
            log.info("hello A");
        }
    }

    @Slf4j
    static class B {
        public void helloB() {
            log.info("hello B");
        }
    }

    @Slf4j
    static class AToBPostProcessor implements BeanPostProcessor {
        @Override
        public Object postProcessAfterInitialization(Object bean, String
beanName) throws BeansException {
            log.info("beanName={} bean={}", beanName, bean);
```

```
            if (bean instanceof A) {
                return new B();
            }
            return bean;
        }
    }
}
```

AToBPostProcessor

- 빈 후처리기이다. 인터페이스인 `BeanPostProcessor`를 구현하고, 스프링 빈으로 등록하면 스프링 컨테이너가 빈 후처리기로 인식하고 동작한다.
- 이 빈 후처리기는 A 객체를 새로운 B 객체로 바꿔치기 한다. 파라미터로 넘어오는 빈(`bean`) 객체가 `A`의 인스턴스이면 새로운 `B` 객체를 생성해서 반환한다. 여기서 `A` 대신에 반환된 값인 `B`가 스프링 컨테이너에 등록된다. 다음 실행결과를 보면 `beanName=beanA` , `bean=A` 객체의 인스턴스가 빈 후처리기에 넘어온 것을 확인할 수 있다.

실행 결과

```
..AToBPostProcessor - beanName=beanA
bean=hello.proxy.postprocessor...A@21362712
..B - hello B
```

`B b = applicationContext.getBean("beanA", B.class)`

- 실행 결과를 보면 최종적으로 `"beanA"`라는 스프링 빈 이름에 `A` 객체 대신에 `B` 객체가 등록된 것을 확인할 수 있다. `A`는 스프링 빈으로 등록조차 되지 않는다.

정리

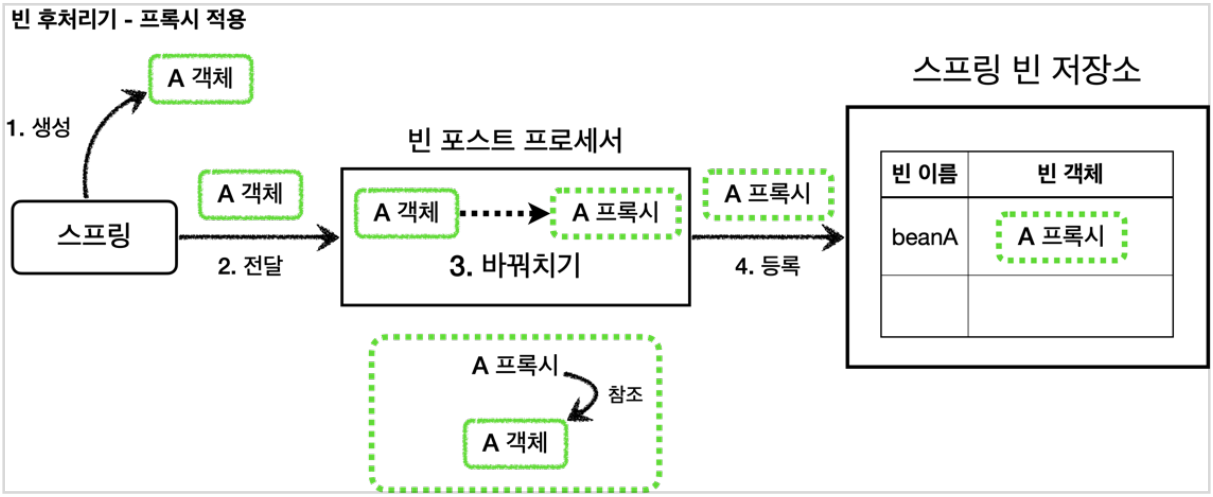
빈 후처리기는 빈을 조작하고 변경할 수 있는 후킹 포인트이다.
이것은 빈 객체를 조작하거나 심지어 다른 객체로 바꾸어 버릴 수 있을 정도로 막강하다.
여기서 조작이라는 것은 해당 객체의 특정 메서드를 호출하는 것을 뜻한다.
일반적으로 스프링 컨테이너가 등록하는, 특히 컴포넌트 스캔의 대상이 되는 빈들은 중간에 조작할 방법이 없는데, 빈 후처리기를 사용하면 개발자가 등록하는 모든 빈을 중간에 조작할 수 있다. 이 말은 **빈 객체를 프록시로 교체**하는 것도 가능하다는 뜻이다.

> **참고 - @PostConstruct의 비밀**
> `@PostConstruct`는 스프링 빈 생성 이후에 빈을 초기화 하는 역할을 한다. 그런데 생각해보면 빈의 초기화

> 라는 것이 단순히 `@PostConstruct` 애노테이션이 붙은 초기화 메서드를 한번 호출만 하면 된다. 쉽게 이야기해서 생성된 빈을 한번 조작하는 것이다.
> 따라서 빈을 조작하는 행위를 하는 적절한 빈 후처리기가 있으면 될 것 같다.
> 스프링은 `CommonAnnotationBeanPostProcessor`라는 빈 후처리기를 자동으로 등록하는데, 여기에서 `@PostConstruct` 애노테이션이 붙은 메서드를 호출한다. 따라서 스프링 스스로도 스프링 내부의 기능을 확장하기 위해 빈 후처리기를 사용한다.

빈 후처리기 - 적용

빈 후처리기를 사용해서 실제 객체 대신 프록시를 스프링 빈으로 등록해보자.
이렇게 하면 수동으로 등록하는 빈은 물론이고, 컴포넌트 스캔을 사용하는 빈까지 모두 프록시를 적용할 수 있다.
더 나아가서 설정 파일에 있는 수 많은 프록시 생성 코드도 한번에 제거할 수 있다.



PackageLogTraceProxyPostProcessor

```
package hello.proxy.config.v4_postprocessor.postprocessor;

import lombok.extern.slf4j.Slf4j;
import org.springframework.aop.Advisor;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;
```

```
@Slf4j
public class PackageLogTraceProxyPostProcessor implements BeanPostProcessor {

    private final String basePackage;
    private final Advisor advisor;

    public PackageLogTraceProxyPostProcessor(String basePackage, Advisor
advisor) {
        this.basePackage = basePackage;
        this.advisor = advisor;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName)
                                                                        throws
BeansException {

        log.info("param beanName={} bean={}", beanName, bean.getClass());

        //프록시 적용 대상 여부 체크
        //프록시 적용 대상이 아니면 원본을 그대로 반환
        String packageName = bean.getClass().getPackageName();
        if (!packageName.startsWith(basePackage)) {
            return bean;
        }

        //프록시 대상이면 프록시를 만들어서 반환
        ProxyFactory proxyFactory = new ProxyFactory(bean);
        proxyFactory.addAdvisor(advisor);

        Object proxy = proxyFactory.getProxy();
        log.info("create proxy: target={} proxy={}", bean.getClass(),
proxy.getClass());
        return proxy;
    }

}
```

- `PackageLogTraceProxyPostProcessor`는 원본 객체를 프록시 객체로 변환하는 역할을 한다. 이때 프록시 팩토리를 사용하는데, 프록시 팩토리는 `advisor`가 필요하기 때문에 이 부분은 외부에서 주입 받도록 했다.
- 모든 스프링 빈들에 프록시를 적용할 필요는 없다. 여기서는 특정 패키지와 그 하위에 위치한 스프링 빈들만 프록시를 적용한다. 여기서는 `hello.proxy.app`과 관련된 부분에만 적용하면 된다. 다른 패키지의 객체들은 원본 객체를 그대로 반환한다.
- 프록시 적용 대상의 반환 값을 보면 원본 객체 대신에 프록시 객체를 반환한다. 따라서 스프링 컨테이너에 원본 객체 대신에 프록시 객체가 스프링 빈으로 등록된다. 원본 객체는 스프링 빈으로 등록되지 않는다.

BeanPostProcessorConfig

```
package hello.proxy.config.v4_postprocessor;

import hello.proxy.config.AppV1Config;
import hello.proxy.config.AppV2Config;
import hello.proxy.config.v3_proxyfactory.advice.LogTraceAdvice;
import
hello.proxy.config.v4_postprocessor.postprocessor.PackageLogTraceProxyPostProce
ssor;
import hello.proxy.trace.logtrace.LogTrace;
import lombok.extern.slf4j.Slf4j;
import org.springframework.aop.Advisor;
import org.springframework.aop.support.DefaultPointcutAdvisor;
import org.springframework.aop.support.NameMatchMethodPointcut;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;

@Slf4j
@Configuration
@Import({AppV1Config.class, AppV2Config.class})
public class BeanPostProcessorConfig {

    @Bean
    public PackageLogTraceProxyPostProcessor
logTraceProxyPostProcessor(LogTrace logTrace) {
        return new PackageLogTraceProxyPostProcessor("hello.proxy.app",
```

```
getAdvisor(logTrace));
    }

    private Advisor getAdvisor(LogTrace logTrace) {
        //pointcut
        NameMatchMethodPointcut pointcut = new NameMatchMethodPointcut();
        pointcut.setMappedNames("request*", "order*", "save*");
        //advice
        LogTraceAdvice advice = new LogTraceAdvice(logTrace);
        //advisor = pointcut + advice
        return new DefaultPointcutAdvisor(pointcut, advice);
    }
}
```

- `@Import({AppV1Config.class, AppV2Config.class})` : V3는 컴포넌트 스캔으로 자동으로 스프링 빈으로 등록되지만, V1, V2 애플리케이션은 수동으로 스프링 빈으로 등록해야 동작한다. `ProxyApplication` 에서 등록해도 되지만 편의상 여기에 등록하자.
- `@Bean logTraceProxyPostProcessor()` : 특정 패키지를 기준으로 프록시를 생성하는 빈 후처리기를 스프링 빈으로 등록한다. 빈 후처리기는 스프링 빈으로만 등록하면 자동으로 동작한다. 여기에 프록시를 적용할 패키지 정보(`hello.proxy.app`)와 어드바이저(`getAdvisor(logTrace)`)를 넘겨준다.
- 이제 **프록시를 생성하는 코드가 설정 파일에는 필요 없다.** 순수한 빈 등록만 고민하면 된다. 프록시를 생성하고 프록시를 스프링 빈으로 등록하는 것은 빈 후처리기가 모두 처리해준다.

ProxyApplication

```
//@Import({AppV1Config.class, AppV2Config.class})
//@Import(InterfaceProxyConfig.class)
//@Import(ConcreteProxyConfig.class)
//@Import(DynamicProxyBasicConfig.class)
//@Import(DynamicProxyFilterConfig.class)
//@Import(ProxyFactoryConfigV1.class)
//@Import(ProxyFactoryConfigV2.class)
@Import(BeanPostProcessorConfig.class)
@SpringBootApplication(scanBasePackages = "hello.proxy.app")
public class ProxyApplication {

    public static void main(String[] args) {
        SpringApplication.run(ProxyApplication.class, args);
    }
```

```
    @Bean
    public LogTrace logTrace() {
        return new ThreadLocalLogTrace();
    }

}
```

`BeanPostProcessorConfig.class`를 등록하자.

애플리케이션 로딩 로그

중요 부분만 남기고 순서를 조정하고 축약했다.

```
#v1 애플리케이션 프록시 생성 - JDK 동적 프록시
create proxy: target=v1.OrderRepositoryV1Impl proxy=class com.sun.proxy.
$Proxy50
create proxy: target=v1.OrderServiceV1Impl proxy=class com.sun.proxy.$Proxy51
create proxy: target=v1.OrderControllerV1Impl proxy=class com.sun.proxy.
$Proxy52
#v2 애플리케이션 프록시 생성 - CGLIB
create proxy: target=v2.OrderRepositoryV2 proxy=v2.OrderRepositoryV2$
$EnhancerBySpringCGLIB$$x4
create proxy: target=v2.OrderServiceV2 proxy=v2.OrderServiceV2$
$EnhancerBySpringCGLIB$$x5
create proxy: target=v2.OrderControllerV2 proxy=v2.OrderControllerV2$
$EnhancerBySpringCGLIB$$x6
#v3 애플리케이션 프록시 생성 - CGLIB
create proxy: target=v3.OrderRepositoryV3 proxy=3.OrderRepositoryV3$
$EnhancerBySpringCGLIB$$x1
create proxy: target=v3.orderServiceV3 proxy=3.OrderServiceV3$
$EnhancerBySpringCGLIB$$x2
create proxy: target=v3.orderControllerV3 proxy=3.orderControllerV3$
$EnhancerBySpringCGLIB$$x3
```

- 여기서는 생략했지만, 실행해보면 스프링 부트가 기본으로 등록하는 수 많은 빈들이 빈 후처리기를 통과하는 것을 확인할 수 있다. 여기에 모두 프록시를 적용하는 것은 올바르지 않다. 꼭 필요한 곳에만 프록시를 적용해야 한다. 여기서는 `basePackage`를 사용해서 v1~v3 애플리케이션 관련 빈들만 프록시 적용 대상이 되도록 했다.
- **v1:** 인터페이스가 있으므로 JDK 동적 프록시가 적용된다.

- **v2:** 구체 클래스만 있으므로 CGLIB 프록시가 적용된다.
- **v3:** 구체 클래스만 있으므로 CGLIB 프록시가 적용된다.

컴포넌트 스캔에도 적용

여기서 중요한 포인트는 v1, v2와 같이 수동으로 등록한 빈 뿐만 아니라 컴포넌트 스캔을 통해 등록한 v3 빈들도 프록시를 적용할 수 있다는 점이다. 이것은 모두 빈 후처리기 덕분이다.

실행

- http://localhost:8080/v1/request?itemId=hello
- http://localhost:8080/v2/request?itemId=hello
- http://localhost:8080/v3/request?itemId=hello

실행해보면 모두 동일한 결과가 나오는 것을 확인할 수 있다.

프록시 적용 대상 여부 체크

- 애플리케이션을 실행해서 로그를 확인해보면 알겠지만, 우리가 직접 등록한 스프링 빈들 뿐만 아니라 스프링 부트가 기본으로 등록하는 수 많은 빈들이 빈 후처리기에 넘어온다. 그래서 어떤 빈을 프록시로 만들 것인지 기준이 필요하다. 여기서는 간단히 `basePackage`를 사용해서 특정 패키지를 기준으로 해당 패키지와 그 하위 패키지의 빈들을 프록시로 만든다.
- 스프링 부트가 기본으로 제공하는 빈 중에는 프록시 객체를 만들 수 없는 빈들도 있다. 따라서 모든 객체를 프록시로 만들 경우 오류가 발생한다.

빈 후처리기 - 정리

이전에 보았던 문제들이 빈 후처리기를 통해서 어떻게 해결되었는지 정리해보자.

문제1 - 너무 많은 설정

프록시를 직접 스프링 빈으로 등록하는 `ProxyFactoryConfigV1`, `ProxyFactoryConfigV2`와 같은 설정 파일은 프록시 관련 설정이 지나치게 많다는 문제가 있다.
예를 들어서 애플리케이션에 스프링 빈이 100개가 있다면 여기에 프록시를 통해 부가 기능을 적용하려면 100개의 프록시 설정 코드가 들어가야 한다. 무수히 많은 설정 파일 때문에 설정 지옥을 경험하게 될 것이다.
스프링 빈을 편리하게 등록하려고 컴포넌트 스캔까지 사용하는데, 이렇게 직접 등록하는 것도 모자라서, 프록시를 적용하는 코드까지 빈 생성 코드에 넣어야 했다.

문제2 - 컴포넌트 스캔

애플리케이션 V3처럼 컴포넌트 스캔을 사용하는 경우 지금까지 학습한 방법으로는 프록시 적용이 불가능했다.
왜냐하면 컴포넌트 스캔으로 이미 스프링 컨테이너에 실제 객체를 스프링 빈으로 등록을 다 해버린 상태이기 때문이다.
좀 더 풀어서 설명하자면, 지금까지 학습한 방식으로 프록시를 적용하려면, 원본 객체를 스프링 컨테이너에 빈으로 등록하는 것이 아니라 `ProxyFactoryConfigV1`에서 한 것 처럼, 프록시를 원본 객체 대신 스프링 컨테이너에 빈으로 등록해야 한다. 그런데 컴포넌트 스캔은 원본 객체를 스프링 빈으로 자동으로 등록하기 때문에 프록시 적용이 불가능하다.

문제 해결
빈 후처리기 덕분에 프록시를 생성하는 부분을 하나로 집중할 수 있다. 그리고 컴포넌트 스캔처럼 스프링이 직접 대상을 빈으로 등록하는 경우에도 중간에 빈 등록 과정을 가로채서 원본 대신에 프록시를 스프링 빈으로 등록할 수 있다.
덕분에 애플리케이션에 수 많은 스프링 빈이 추가되어도 프록시와 관련된 코드는 전혀 변경하지 않아도 된다. 그리고 컴포넌트 스캔을 사용해도 프록시가 모두 적용된다.

하지만 개발자의 욕심은 끝이 없다.
스프링은 프록시를 생성하기 위한 빈 후처리기를 이미 만들어서 제공한다.

> **중요**
> 프록시의 적용 대상 여부를 여기서는 간단히 패키지를 기준으로 설정했다. 그런데 잘 생각해보면 포인트컷을 사용하면 더 깔끔할 것 같다.
> 포인트컷은 이미 클래스, 메서드 단위의 필터 기능을 가지고 있기 때문에, 프록시 적용 대상 여부를 정밀하게 설정할 수 있다.
> 참고로 어드바이저는 포인트컷을 가지고 있다. 따라서 어드바이저를 통해 포인트컷을 확인할 수 있다.
> 뒤에서 학습하겠지만 스프링 AOP는 포인트컷을 사용해서 프록시 적용 대상 여부를 체크한다.
>
> 결과적으로 포인트컷은 다음 두 곳에 사용된다.
> 1. 프록시 적용 대상 여부를 체크해서 꼭 필요한 곳에만 프록시를 적용한다. (빈 후처리기 - 자동 프록시 생성)
> 2. 프록시의 어떤 메서드가 호출 되었을 때 어드바이스를 적용할 지 판단한다. (프록시 내부)

스프링이 제공하는 빈 후처리기1

주의 - 다음을 꼭 추가해주어야 한다.

build.gradle - 추가

```
implementation 'org.springframework.boot:spring-boot-starter-aop'
```

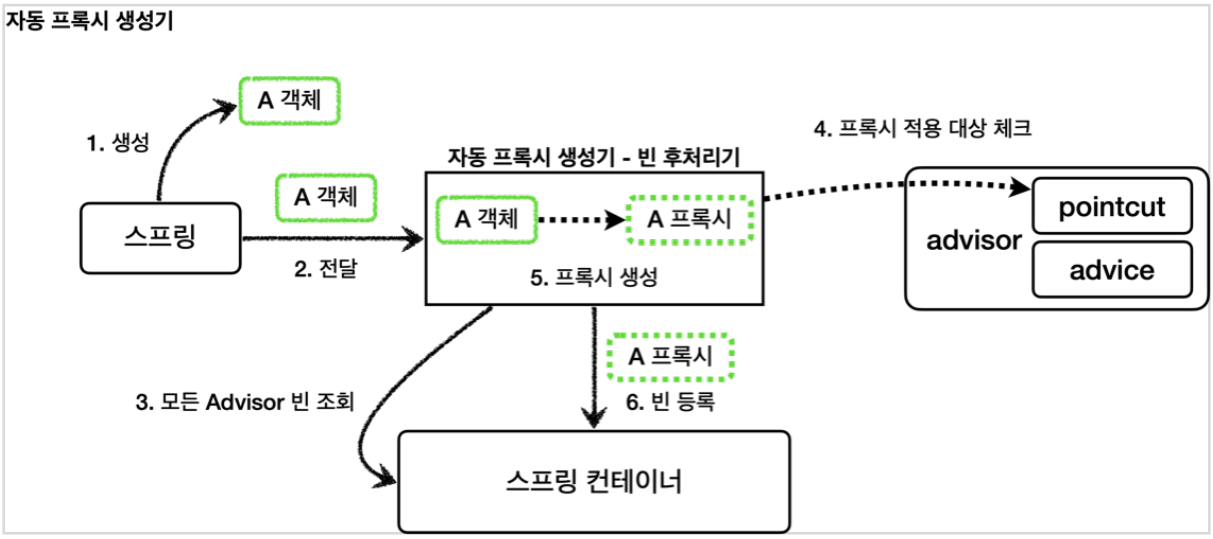
이 라이브러리를 추가하면 `aspectjweaver` 라는 `aspectJ` 관련 라이브러리를 등록하고, 스프링 부트가 AOP 관련 클래스를 자동으로 스프링 빈에 등록한다. 스프링 부트가 없던 시절에는 `@EnableAspectJAutoProxy` 를 직접 사용해야 했는데, 이 부분을 스프링 부트가 자동으로 처리해준다. `aspectJ` 는 뒤에서 설명한다. 스프링 부트가 활성화하는 빈은 `AopAutoConfiguration` 를 참고하자.

자동 프록시 생성기 - AutoProxyCreator

- 앞서 이야기한 스프링 부트 자동 설정으로 `AnnotationAwareAspectJAutoProxyCreator` 라는 빈 후처리기가 스프링 빈에 자동으로 등록된다.
- 이름 그대로 자동으로 프록시를 생성해주는 빈 후처리기이다.
- 이 빈 후처리기는 스프링 빈으로 등록된 `Advisor` 들을 자동으로 찾아서 프록시가 필요한 곳에 자동으로 프록시를 적용해준다.
- `Advisor` 안에는 `Pointcut` 과 `Advice` 가 이미 모두 포함되어 있다. 따라서 `Advisor` 만 알고 있으면 그 안에 있는 `Pointcut` 으로 어떤 스프링 빈에 프록시를 적용해야 할지 알 수 있다. 그리고 `Advice` 로 부가 기능을 적용하면 된다.

> **참고**
> `AnnotationAwareAspectJAutoProxyCreator` 는 @AspectJ와 관련된 AOP 기능도 자동으로 찾아서 처리해준다.
> `Advisor` 는 물론이고, `@Aspect` 도 자동으로 인식해서 프록시를 만들고 AOP를 적용해준다. `@Aspect` 에 대한 자세한 내용은 뒤에 설명한다.

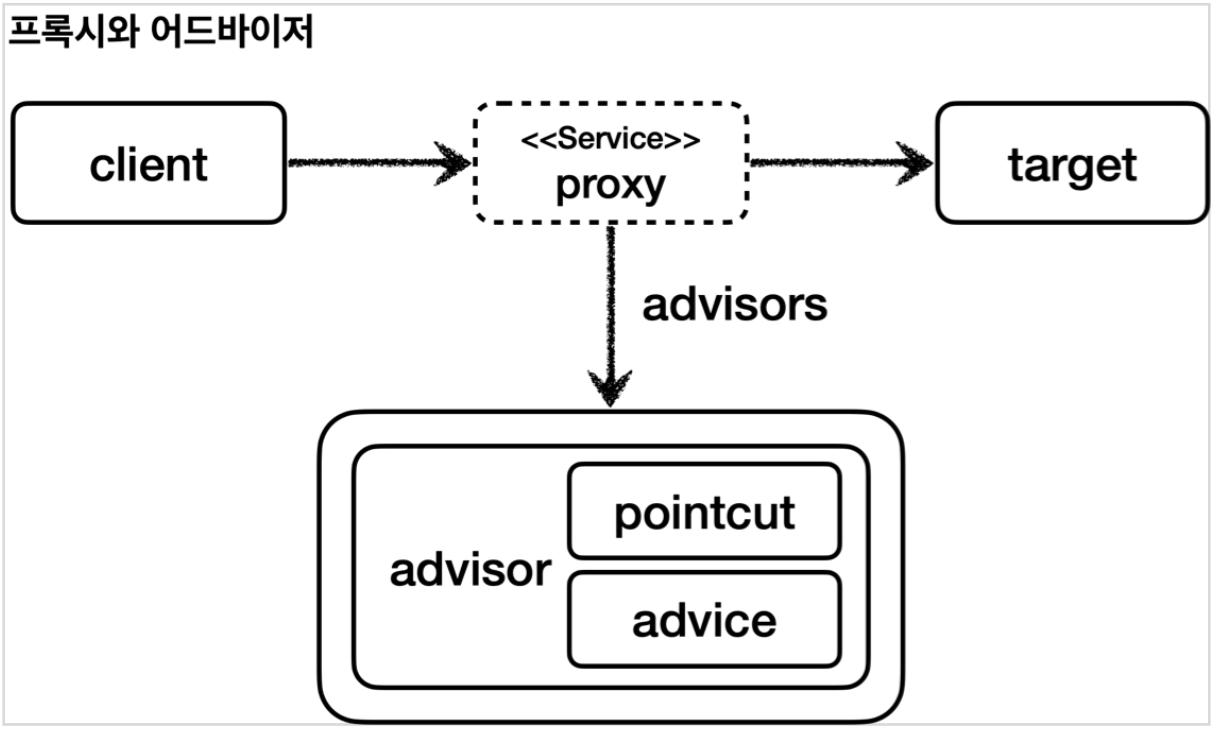
자동 프록시 생성기의 작동 과정 - 그림



자동 프록시 생성기의 작동 과정을 알아보자

- **1. 생성:** 스프링이 스프링 빈 대상이 되는 객체를 생성한다. (`@Bean` , 컴포넌트 스캔 모두 포함)
- **2. 전달:** 생성된 객체를 빈 저장소에 등록하기 직전에 빈 후처리기에 전달한다.
- **3. 모든 Advisor 빈 조회:** 자동 프록시 생성기 - 빈 후처리기는 스프링 컨테이너에서 모든 `Advisor` 를 조회한다.
- **4. 프록시 적용 대상 체크:** 앞서 조회한 `Advisor` 에 포함되어 있는 포인트컷을 사용해서 해당 객체가 프록시를 적용할 대상인지 아닌지 판단한다. 이때 객체의 클래스 정보는 물론이고, 해당 객체의 모든 메서드를 포인트컷에 하나하나 모두 매칭해본다. 그래서 조건이 하나라도 만족하면 프록시 적용 대상이 된다. 예를 들어서 10개의 메서드 중에 하나만 포인트컷 조건에 만족해도 프록시 적용 대상이 된다.
- **5. 프록시 생성:** 프록시 적용 대상이면 프록시를 생성하고 반환해서 프록시를 스프링 빈으로 등록한다. 만약 프록시 적용 대상이 아니라면 원본 객체를 반환해서 원본 객체를 스프링 빈으로 등록한다.
- **6. 빈 등록:** 반환된 객체는 스프링 빈으로 등록된다.

생성된 프록시



프록시는 내부에 어드바이저와 실제 호출해야할 대상 객체(`target`)을 알고 있다.

코드를 통해 바로 적용해보자.

AutoProxyConfig

```
package hello.proxy.config.v5_autoproxy;

import hello.proxy.config.AppV1Config;
import hello.proxy.config.AppV2Config;
```

```
import hello.proxy.config.v3_proxyfactory.advice.LogTraceAdvice;
import hello.proxy.trace.logtrace.LogTrace;
import org.springframework.aop.Advisor;
import org.springframework.aop.aspectj.AspectJExpressionPointcut;
import org.springframework.aop.support.DefaultPointcutAdvisor;
import org.springframework.aop.support.NameMatchMethodPointcut;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;

@Configuration
@Import({AppV1Config.class, AppV2Config.class})
public class AutoProxyConfig {

    @Bean
    public Advisor advisor1(LogTrace logTrace) {
        NameMatchMethodPointcut pointcut = new NameMatchMethodPointcut();
        pointcut.setMappedNames("request*", "order*", "save*");
        LogTraceAdvice advice = new LogTraceAdvice(logTrace);
        //advisor = pointcut + advice
        return new DefaultPointcutAdvisor(pointcut, advice);
    }

}
```

- `AutoProxyConfig` 코드를 보면 `advisor1` 이라는 어드바이저 하나만 등록했다.
- 빈 후처리기는 이제 등록하지 않아도 된다. 스프링은 자동 프록시 생성기라는 (`AnnotationAwareAspectJAutoProxyCreator`) 빈 후처리기를 자동으로 등록해준다.

```
//@Import({AppV1Config.class, AppV2Config.class})
//@Import(InterfaceProxyConfig.class)
//@Import(ConcreteProxyConfig.class)
//@Import(DynamicProxyBasicConfig.class)
//@Import(DynamicProxyFilterConfig.class)
//@Import(ProxyFactoryConfigV1.class)
//@Import(ProxyFactoryConfigV2.class)
//@Import(BeanPostProcessorConfig.class)
```

```java
@Import(AutoProxyConfig.class)
@SpringBootApplication(scanBasePackages = "hello.proxy.app")
public class ProxyApplication {

    public static void main(String[] args) {
        SpringApplication.run(ProxyApplication.class, args);
    }

    @Bean
    public LogTrace logTrace() {
        return new ThreadLocalLogTrace();
    }

}
```

실행

- http://localhost:8080/v1/request?itemId=hello
- http://localhost:8080/v2/request?itemId=hello
- http://localhost:8080/v3/request?itemId=hello

실행하면 모두 프록시 적용된 결과가 나오는 것을 확인할 수 있다.

실행하면 로그가 나오면 안됨

- http://localhost:8080/v1/no-log

로그가 출력되지 않는 것을 확인할 수 있다.

중요: 포인트컷은 2가지에 사용된다.

- **1. 프록시 적용 여부 판단 - 생성 단계**
  - 자동 프록시 생성기는 포인트컷을 사용해서 해당 빈이 프록시를 생성할 필요가 있는지 없는지 체크한다.
  - 클래스 + 메서드 조건을 모두 비교한다. 이때 모든 메서드를 체크하는데, 포인트컷 조건에 하나하나 매칭해본다. 만약 조건에 맞는 것이 하나라도 있으면 프록시를 생성한다.
    - 예) `orderControllerV1`은 `request()`, `noLog()`가 있다. 여기에서 `request()`가 조건에 만족하므로 프록시를 생성한다.

    - 만약 조건에 맞는 것이 하나도 없으면 프록시를 생성할 필요가 없으므로 프록시를 생성하지 않는다.
- **2. 어드바이스 적용 여부 판단 - 사용 단계**
    - 프록시가 호출되었을 때 부가 기능인 어드바이스를 적용할지 말지 포인트컷을 보고 판단한다.
    - 앞서 설명한 예에서 `orderControllerV1` 은 이미 프록시가 걸려있다.
    - `orderControllerV1` 의 `request()` 는 현재 포인트컷 조건에 만족하므로 프록시는 어드바이스를 먼저 호출하고, `target` 을 호출한다.
    - `orderControllerV1` 의 `noLog()` 는 현재 포인트컷 조건에 만족하지 않으므로 어드바이스를 호출하지 않고 바로 `target` 만 호출한다.

참고: 프록시를 모든 곳에 생성하는 것은 비용 낭비이다. 꼭 필요한 곳에 최소한의 프록시를 적용해야 한다. 그래서 자동 프록시 생성기는 모든 스프링 빈에 프록시를 적용하는 것이 아니라 포인트컷으로 한번 필터링해서 어드바이스가 사용될 가능성이 있는 곳에만 프록시를 생성한다.

스프링이 제공하는 빈 후처리기2

애플리케이션 로딩 로그

```
EnableWebMvcConfiguration.requestMappingHandlerAdapter()
EnableWebMvcConfiguration.requestMappingHandlerAdapter() time=63ms
```

애플리케이션 서버를 실행해보면, 스프링이 초기화 되면서 기대하지 않은 이러한 로그들이 올라온다. 그 이유는 지금 사용한 포인트컷이 단순히 메서드 이름에 `"request*", "order*", "save*"` 만 포함되어 있으면 매칭 된다고 판단하기 때문이다.
결국 스프링이 내부에서 사용하는 빈에도 메서드 이름에 `request` 라는 단어만 들어가 있으면 프록시가 만들어지고 되고, 어드바이스도 적용되는 것이다.

결론적으로 패키지에 메서드 이름까지 함께 지정할 수 있는 매우 정밀한 포인트컷이 필요하다.

AspectJExpressionPointcut
AspectJ라는 AOP에 특화된 포인트컷 표현식을 적용할 수 있다. AspectJ 포인트컷 표현식과 AOP는 조금 뒤에 자세히 설명하겠다. 지금은 특별한 표현식으로 복잡한 포인트컷을 만들 수 있구나 라고 대략 이해하면 된다.

AutoProxyConfig - advisor2 추가

```
@Bean
```

```
public Advisor advisor2(LogTrace logTrace) {
    AspectJExpressionPointcut pointcut = new AspectJExpressionPointcut();
    pointcut.setExpression("execution(* hello.proxy.app..*(..))");
    LogTraceAdvice advice = new LogTraceAdvice(logTrace);
    //advisor = pointcut + advice
    return new DefaultPointcutAdvisor(pointcut, advice);
}
```

주의

`advisor1`에 있는 `@Bean`은 꼭 주석처리해주어야 한다. 그렇지 않으면 어드바이저가 중복 등록된다.

- `AspectJExpressionPointcut` : AspectJ 포인트컷 표현식을 적용할 수 있다.
- `execution(* hello.proxy.app..*(..))` : AspectJ가 제공하는 포인트컷 표현식이다. 이후 자세히 설명하겠다. 지금은 간단히 알아보자.
  - `*` : 모든 반환 타입
  - `hello.proxy.app..` : 해당 패키지와 그 하위 패키지
  - `*(..)` : `*` 모든 메서드 이름, `(..)` 파라미터는 상관 없음

쉽게 이야기해서 `hello.proxy.app` 패키지와 그 하위 패키지의 모든 메서드는 포인트컷의 매칭 대상이 된다.

실행

- http://localhost:8080/v1/request?itemId=hello
- http://localhost:8080/v2/request?itemId=hello
- http://localhost:8080/v3/request?itemId=hello

실행하면 모두 동일한 결과가 나오는 것을 확인할 수 있다.

실행하면 로그가 나오면 안됨

- http://localhost:8080/v1/no-log

그런데 문제는 이 부분에 로그가 출력된다. `advisor2`에서는 단순히 `package`를 기준으로 포인트컷 매칭을 했기 때문이다.

AutoProxyConfig advisor3 추가

```
@Bean
public Advisor advisor3(LogTrace logTrace) {
    AspectJExpressionPointcut pointcut = new AspectJExpressionPointcut();
```

```
    pointcut.setExpression("execution(* hello.proxy.app..*(..)) && !execution(*
hello.proxy.app..noLog(..))");
    LogTraceAdvice advice = new LogTraceAdvice(logTrace);
    //advisor = pointcut + advice
    return new DefaultPointcutAdvisor(pointcut, advice);
}
```

주의

- `advisor1`, `advisor2` 에 있는 `@Bean`은 꼭 주석처리해주어야 한다. 그렇지 않으면 어드바이저가 중복 등록된다.

표현식을 다음과 같이 수정했다.

```
execution(* hello.proxy.app..*(..)) && !execution(* hello.proxy.app..noLog(..))
```

- `&&`: 두 조건을 모두 만족해야 함
- `!`: 반대

`hello.proxy.app` 패키지와 하위 패키지의 모든 메서드는 포인트컷의 매칭하되, `noLog()` 메서드는 제외하라는 뜻이다.

실행하면 로그가 나오면 안됨

- http://localhost:8080/v1/no-log

이제 로그가 남지 않는 것을 확인할 수 있다.

> **참고**
> AspectJ, AOP는 이후에 자세히 설명한다. 지금은 이런 포인트컷도 있구나 정도 알고 넘어가면 된다.

하나의 프록시, 여러 Advisor 적용

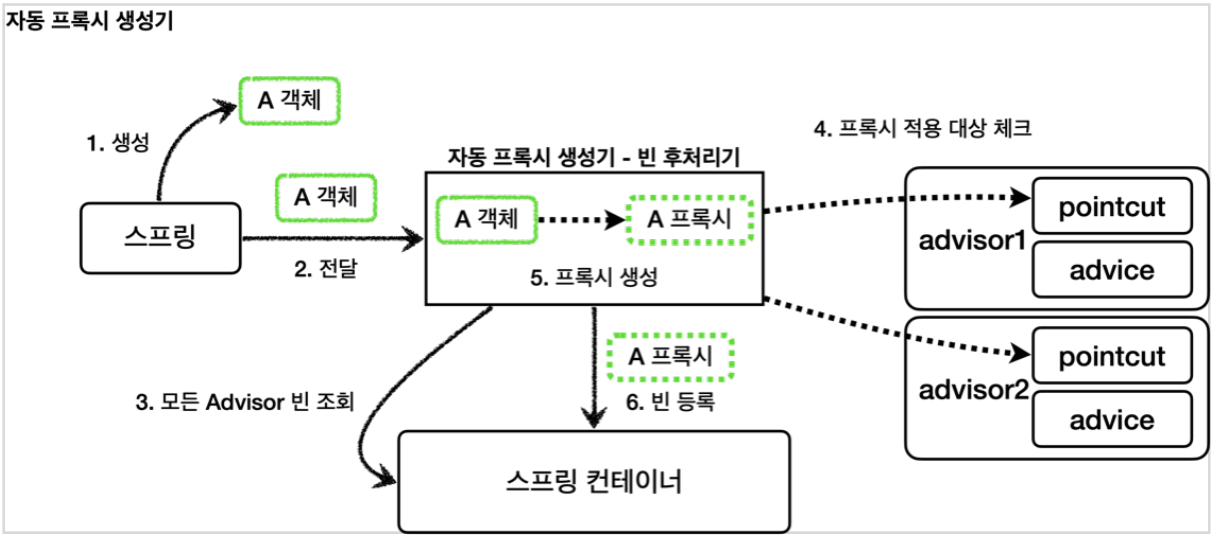
예를 들어서 어떤 스프링 빈이 `advisor1`, `advisor2`가 제공하는 포인트컷의 조건을 모두 만족하면 프록시 자동 생성기는 프록시를 몇 개 생성할까?
프록시 자동 생성기는 프록시를 하나만 생성한다. 왜냐하면 프록시 팩토리가 생성하는 프록시는 내부에 여러 `advisor`들을 포함할 수 있기 때문이다. 따라서 프록시를 여러 개 생성해서 비용을 낭비할 이유가

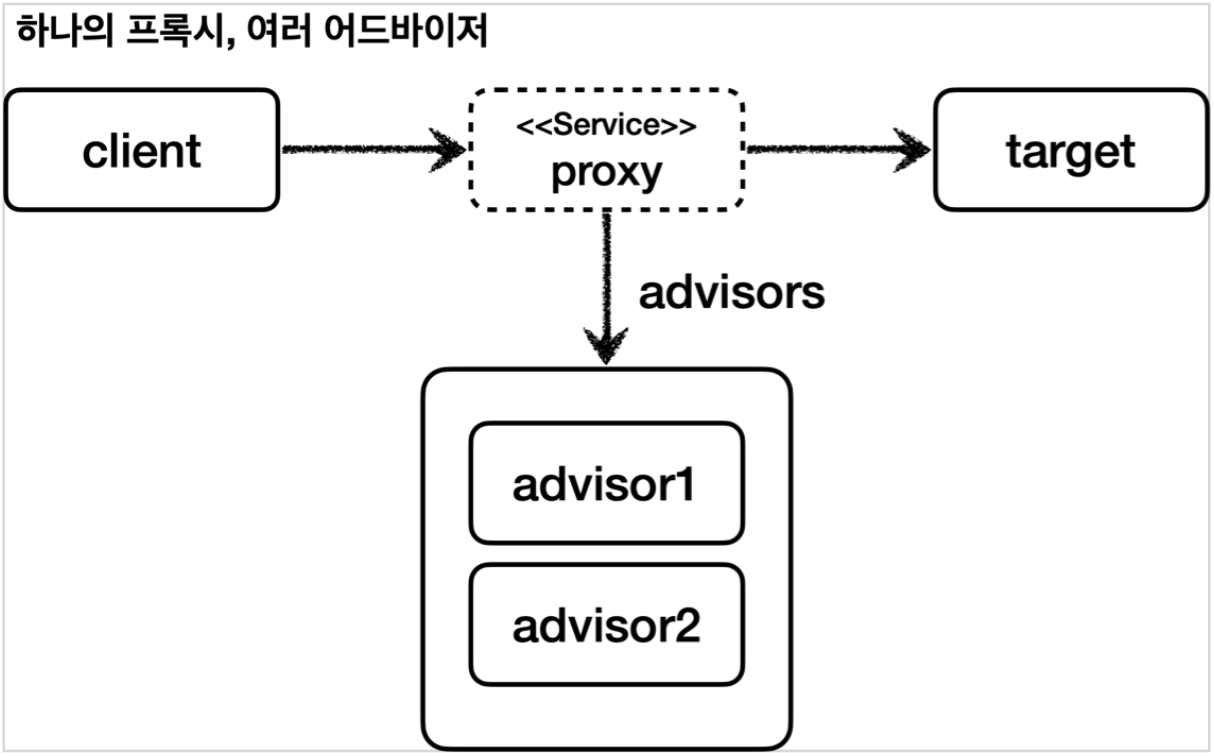
없다.

프록시 자동 생성기 상황별 정리

- `advisor1`의 포인트컷만 만족 ➡ 프록시1개 생성, 프록시에 `advisor1`만 포함
- `advisor1`, `advisor2`의 포인트컷을 모두 만족 ➡ 프록시1개 생성, 프록시에 `advisor1`, `advisor2` 모두 포함
- `advisor1`, `advisor2`의 포인트컷을 모두 만족하지 않음 ➡ 프록시가 생성되지 않음

이후에 설명할 스프링 AOP도 동일한 방식으로 동작한다.





정리

자동 프록시 생성기인 `AnnotationAwareAspectJAutoProxyCreator` 덕분에 개발자는 매우 편리하게 프록시를 적용할 수 있다. 이제 `Advisor` 만 스프링 빈으로 등록하면 된다.

`Advisor` = `Pointcut` + `Advice`

다음 시간에는 `@Aspect` 애노테이션을 사용해서 더 편리하게 포인트컷과 어드바이스를 만들고 프록시를 적용해보자.