Spring AOP에서 `Introduction`은 기존 객체에 새로운 인터페이스와 메소드를 동적으로 추가하는 기능을 말합니다. 이는 AOP의 핵심 기능 중 하나로, 기존 클래스나 객체에 대해 런타임에서 추가적인 기능을 구현할 수 있도록 합니다.

`Introduction`의 정의

`Introduction`은 Spring AOP의 `IntroductionAdvisor`를 통해 구현되며, 특정 인터페이스를 프록시 객체에 추가하는 방식으로 동작합니다. 이 방식은 기존 객체의 코드나 디자인을 변경하지 않고도 새로운 기능을 추가할 수 있게 해줍니다.

Introduction Advice

AOP(Aspect-Oriented Programming)의 "Introduction Advice"는 특정 클래스나 객체에 새로운 메서드나 필드(속성)를 추가하기 위한 기능입니다. 이는 AOP의 주요 기능 중 하나로, 기존의 코드에 수정 없이 새로운 기능을 주입할 수 있게 해줍니다. 이를 통해 코드의 재사용성을 높이고, 중복을 줄이며, 특정 관심사를 모듈화하는 데 큰 도움이 됩니다.

Introduction Advice의 주요 개념

1. **관심사의 분리(Separation of Concerns)**:
    - AOP의 가장 중요한 목표는 서로 다른 관심사를 코드에서 분리하는 것입니다. 이를 통해 비즈니스 로직과는 별개의 관심사(예: 로깅, 보안, 트랜잭션 관리)를 독립적으로 관리할 수 있습니다. Introduction Advice는 기존의 클래스에 새로운 기능을 동적으로 추가함으로써 이러한 관심사를 모듈화하고 코드의 복잡성을 줄이는 역할을 합니다.
2. **타겟 클래스(Target Class)**:
    - Introduction Advice가 적용되는 대상 클래스입니다. 이 클래스는 AOP 프레임워크에 의해 동적으로 확장되며, 원본 클래스의 소스 코드를 수정하지 않고도 새로운 메서드와 속성을 제공받게 됩니다. 타겟 클래스는 보통 특정 인터페이스를 구현하지 않는 기존 클래스이지만, Introduction Advice를 통해 런타임에 그 인터페이스를 구현하도록 만들어질 수 있습니다.
3. **인터페이스 구현(Implementing an Interface)**:
    - Introduction Advice의 핵심은 기존 클래스에 새로운 인터페이스를 구현시키는 것입니다. 예를 들어, 기존 클래스가 `Auditable`이라는 인터페이스를 구현하지 않더라도, Introduction Advice를 사용하면 해당 클래스가 해당 인터페이스를 구현하도록 할 수 있습니다. 이는 클래스에 새로운 메서드와 필드를 추가하는 방식으로 이루어지며, 이러한 메서드와 필드는 런타임에 동적으로 제공됩니다.
4. **AOP 프레임워크의 역할**:
    - AOP 프레임워크는 Introduction Advice를 통해 대상 클래스에 새로운 메서드와 인터페이스를 추가하는 역할을 합니다. 이는 주로 프록시 패턴을 통해 구현되며, 프록시는 타겟 클래스를 감싸고 그 클래스의 메서드 호출을 가로채어 필요한 새로운 기능을 제공합니다. 클라이언트 코드에서는 이러한 변화가 투명하게 처리되며, 실제로는 수정된 기능을 사용하는 것입니다.

예시

다음은 Introduction Advice의 일반적인 사용 예시입니다.

```
public interface Auditable {
    void setAuditInfo(String info);
    String getAuditInfo();
}
```

위와 같은 `Auditable` 인터페이스가 있다고 가정할 때, 특정 클래스가 이를 구현하지 않았다고 해도, Introduction Advice를 통해 해당 클래스가 런타임에 이 인터페이스를 구현하게 할 수 있습니다.

```
@Aspect
public class AuditableIntroduction {

    @DeclareParents(value="com.example.SomeClass+",
defaultImpl=DefaultAuditable.class)
    public static Auditable mixin;

    public static class DefaultAuditable implements Auditable {
        private String auditInfo;

        @Override
        public void setAuditInfo(String info) {
            this.auditInfo = info;
        }

        @Override
        public String getAuditInfo() {
            return this.auditInfo;
        }
    }
}
```

위 코드에서는 `SomeClass`가 `Auditable` 인터페이스를 구현하지 않았더라도, Introduction Advice를 사용하여 `SomeClass`가 `Auditable` 인터페이스를 구현하도록 만들 수 있습니다. 이렇게 되면 `SomeClass`의 객체는 런타임에 `Auditable` 인터페이스의 메서드도 사용할 수 있게 됩니다.

장점

1. **유연성(Flexibility)**:
    - Introduction Advice는 코드의 구조를 변경하지 않고도 새로운 기능을 추가할 수 있게 해줍니다. 이는 특히 다양한 클래스에 공통된 기능을 추가해야 할 때 매우 유용합니다.
2. **모듈화(Modularity)**:
    - 새로운 기능을 별도의 모듈로 분리하여 관리할 수 있습니다. 이를 통해 코드의 유지보수성을 크게 높일 수 있습니다.
3. **재사용성(Reusability)**:

- 동일한 기능을 여러 클래스에 쉽게 적용할 수 있습니다. 예를 들어, 여러 클래스에 공통적으로 필요한 기능을 Introduction Advice를 통해 추가하면, 코드 중복을 줄이고 관리가 수월해집니다.

4. **코드 수정 없이 확장 가능(Extensibility)**:
    - 기존의 코드를 전혀 수정하지 않고도, 런타임에 동적으로 기능을 추가할 수 있습니다. 이는 특히 코드 변경이 어렵거나 불가능한 상황에서 유용합니다.

IntroductionAdvisor: 도입의 정의와 관리

IntroductionAdvisor는 `Introduction` 기능을 관리하며, 특정 객체에 어떤 인터페이스를 추가할지, 이를 어떤 클래스에 적용할지를 결정합니다. 또한, 도입할 인터페이스의 유효성을 검사하여, 설정 오류를 방지하고 도입이 예상대로 작동하도록 보장합니다.

주요 역할

1. **도입할 인터페이스 정의**:
    - **getInterfaces() 메서드**: 이 메서드는 타겟 객체에 추가할 인터페이스를 지정합니다. 예를 들어, 특정 객체에 `Auditable` 인터페이스를 추가하려면, 이 메서드를 통해 `Auditable` 인터페이스가 반환됩니다.
2. **타겟 클래스 필터링**:
    - **getClassFilter() 메서드**: 이 메서드는 Introduction이 적용될 대상 클래스를 필터링하는 역할을 합니다. 예를 들어, 특정 패키지에 속한 클래스에만 도입을 적용하고자 할 때, 이 메서드를 사용해 해당 패키지에 속한 클래스만 필터링할 수 있습니다.
3. **인터페이스 유효성 검사**:
    - **validateInterfaces() 메서드**: 이 메서드는 도입할 인터페이스가 실제로 해당 클래스에서 구현될 수 있는지를 검증합니다. 이는 설정 오류를 방지하고 시스템의 안정성을 높이는 중요한 단계입니다.

IntroductionInterceptor: 도입된 인터페이스의 실행 관리

IntroductionInterceptor는 `IntroductionAdvisor`와 함께 작동하며, 도입된 인터페이스의 메서드 호출을 가로채고 처리하는 역할을 합니다. 이를 통해, 프록시 객체는 도입된 인터페이스의 메서드를 구현한 것처럼 동작할 수 있습니다.

주요 역할

1. **메서드 호출 가로채기**:
    - **invoke() 메서드**: 도입된 인터페이스의 메서드가 호출되면, `IntroductionInterceptor`는 이 호출을 가로채고 적절히 처리합니다. 예를 들어, `Auditable` 인터페이스가 도입된 객체에서 `setAuditInfo()` 메서드를 호출하면, 이 호출은 `invoke()` 메서드에 의해 처리됩니다.
2. **도입된 인터페이스 구현**:
    - `IntroductionInterceptor`는 도입된 인터페이스의 메서드를 실제로 구현하거나, 필요한 경우 delegate 객체를 통해 이 작업을 수행합니다. 이를 통해 도입된 인터페이스가 예상대로 작동하도

록 보장합니다.

3. **원하지 않는 인터페이스 억제**:
    - **suppressInterface(Class intf) 메서드**: delegate 객체가 구현했지만, AOP 프록시에 노출되지 말아야 할 인터페이스를 억제할 수 있습니다. 이를 통해 노출할 인터페이스를 제어할 수 있습니다.

> **프록시 객체:**
>
> - 프록시 객체는 원래의 비즈니스 객체를 감싸는 객체로, `IntroductionAdvisor`와 `IntroductionInterceptor`를 통해 새로운 인터페이스와 메소드를 제공합니다.

동작 방식

1. **인터페이스 정의**:
    - 새로운 인터페이스를 정의합니다. 예를 들어, `Lockable`이라는 인터페이스가 있다고 가정합니다.

```
public interface Lockable {
    void lock();
    void unlock();
}
```

2. `IntroductionAdvisor` **구현**:
    - `IntroductionAdvisor`를 구현하여 프록시가 새로운 인터페이스를 추가할 수 있도록 합니다. `IntroductionAdvisor`는 두 가지 주요 구성 요소를 포함합니다:
        - **Advice**: 메소드 호출의 처리를 정의합니다. (예: `LockMixin`이 이 역할을 담당)
        - **Interfaces**: 프록시가 구현할 인터페이스를 지정합니다.

```
public class LockMixinAdvisor extends IntroductionAdvisorSupport {
    public LockMixinAdvisor() {
        super(new LockMixin(), Lockable.class);
    }
}
```

    - `LockMixin`은 `Lockable` 인터페이스의 메소드를 실제로 구현하는 클래스입니다.

```
public class LockMixin implements Lockable {
    @Override
    public void lock() {
        // lock implementation
    }

    @Override
    public void unlock() {
        // unlock implementation
```

```
        }
    }
```

> [1^]:
>
> ### `IntroductionInterceptor`의 역할
>
> `IntroductionInterceptor`는 프록시가 호출된 메소드를 어떻게 처리할지 정의하는 인터셉터입니다.
>
> - **`LockMixin`**: `LockMixin`은 `Lockable` 인터페이스를 구현하는 클래스입니다. 이 클래스는 `Lockable` 인터페이스의 메소드를 실제로 구현합니다.

3. **프록시 생성**:
    - `ProxyFactory`를 사용하여 프록시를 생성하고, `IntroductionAdvisor`를 추가합니다.

```
ProxyFactory factory = new ProxyFactory(new MyTargetClass());
factory.addAdvisor(new LockMixinAdvisor());
factory.setProxyTargetClass(true);
MyTargetClass proxy = (MyTargetClass) factory.getProxy();
```

4. **프록시 사용**:
    - 프록시 객체는 `Lockable` 인터페이스를 구현하는 것처럼 동작합니다. 이를 통해 원래의 `MyTargetClass` 객체는 `Lockable` 인터페이스를 직접 구현하지 않았더라도, 프록시를 통해 `Lockable`의 메소드에 접근할 수 있습니다.

예시

```
public class MyTargetClass {
    // 기존 비즈니스 로직
}

// 프록시 객체
MyTargetClass proxy = (MyTargetClass) factory.getProxy();
Lockable lockable = (Lockable) proxy; // 프록시가 Lockable 인터페이스를 구현하므로 캐스팅 가능
lockable.lock(); // 실제 LockMixin의 lock() 메소드가 호출됨
```

결론

`Introduction`은 Spring AOP의 강력한 기능으로, 기존 비즈니스 객체에 새로운 인터페이스와 메소드를 동적으로 추가할 수 있게 해줍니다. 이를 통해 코드 변경 없이도 기능을 확장하고, 비즈니스 로직과 부가 기능을 분리하여 유지보수성과 확장성을 높일 수 있습니다.