IoC & AOP Detailed Explanation (Quickly Understand)

Guide frame Spring About 3610 words About 12 minutes

This article will explain IoC & AOP from the following questions:

- What is IoC?
- What problem does IoC solve?
- What is the difference between IoC and DI?
- What is AOP?
- What problem does AOP solve?
- What are the application scenarios of AOP?
- Why is AOP called aspect programming?
- What are the ways to implement AOP?

First of all, let me state that IoC and AOP were not proposed by Spring. They actually existed before Spring, but they were more theoretical at the time. Spring has implemented these two ideas very well at the technical level.

IoC (Inversion of control)

What is IoC?

IoC (Inversion of Control) stands for Inversion of Control. It's a concept, not a technical implementation. It describes the creation and management of domain objects in Java development.

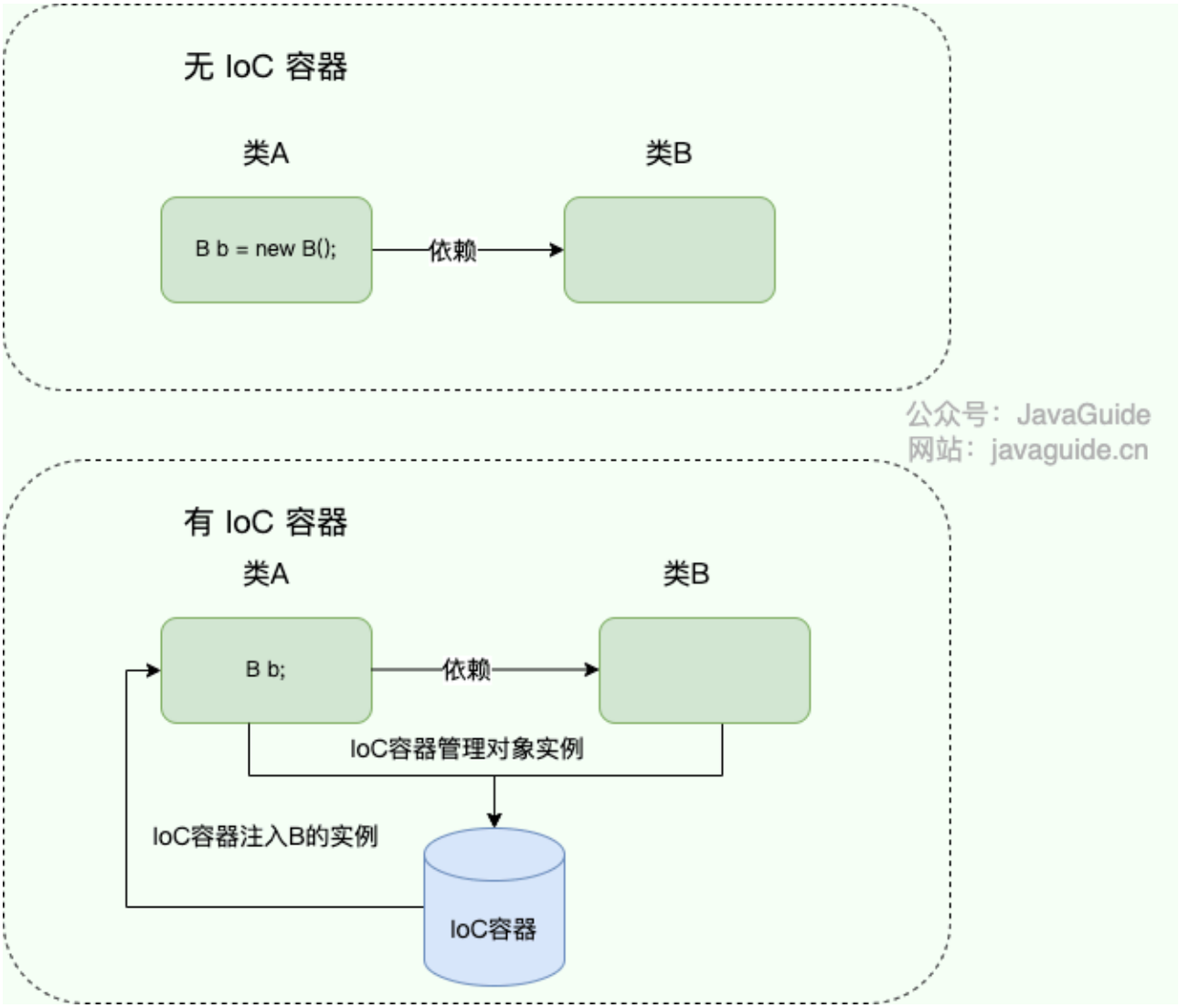
For example: existing class A depends on class B

- **Traditional development method** : often manually create a new object of B in class A through the new keyword
- **The development method using IoC thinking** : instead of creating objects with the new keyword, we use the IoC container (Spring framework) to help us instantiate objects. We can directly get the objects we need from the IoC container.

From the comparison of the above two development methods: we "lose a power" (the power to create and manage objects), but we also gain a benefit (no longer having to consider a series of things such as object creation and management).

Why is it called Inversion of Control?

- **Control** : refers to the power to create (instantiate, manage) objects
- **Inversion** : Control is given to the external environment (IoC container)



What problem does IoC solve?

The idea behind IoC is that two parties do not rely on each other, and a third-party container manages related resources. What are the benefits of this?

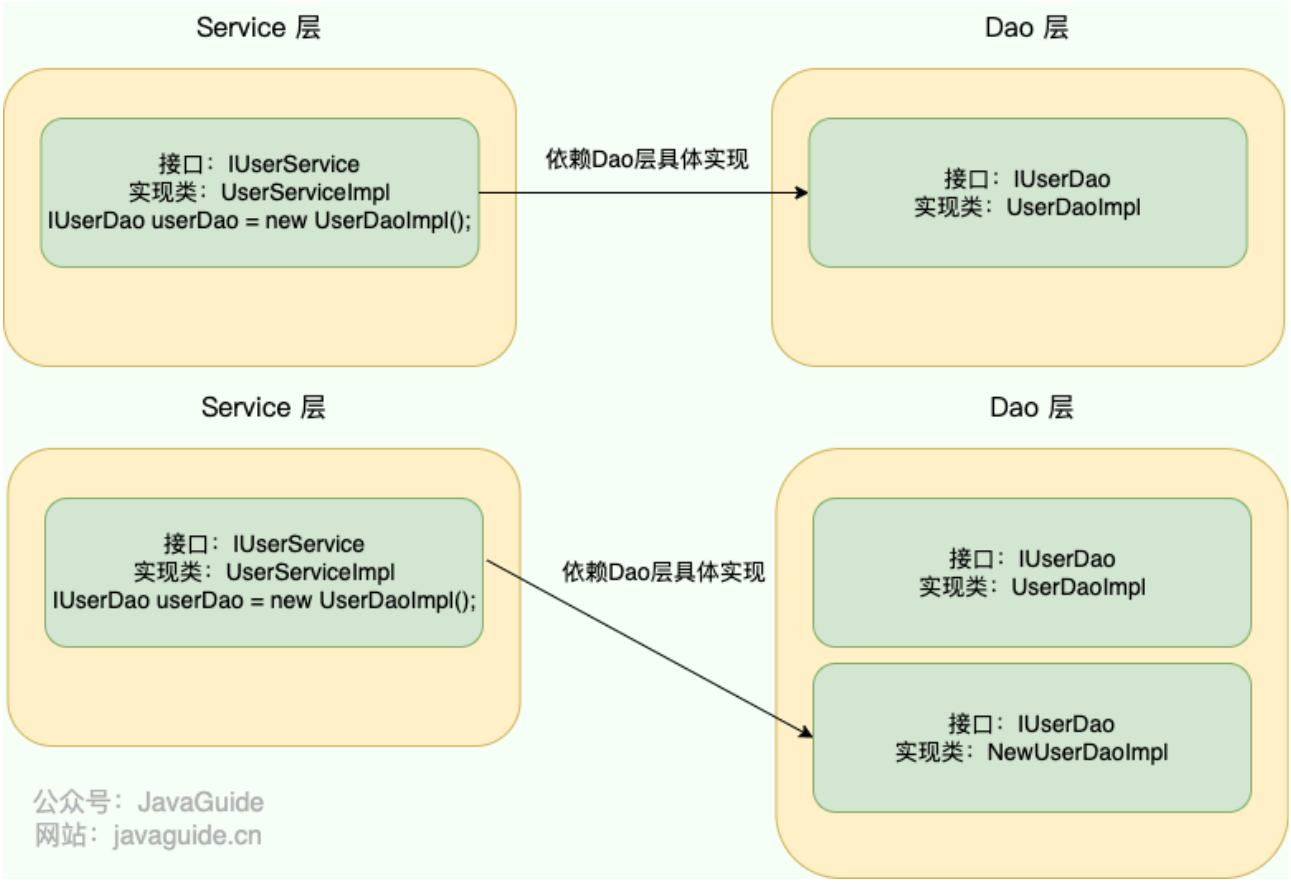
1. The coupling or dependency between objects is reduced;
2. Resources become easier to manage; for example, if you use the Spring container, you can easily implement a singleton.

For example, there is an operation for User, which is developed using the two-layer structure of Service and Dao.

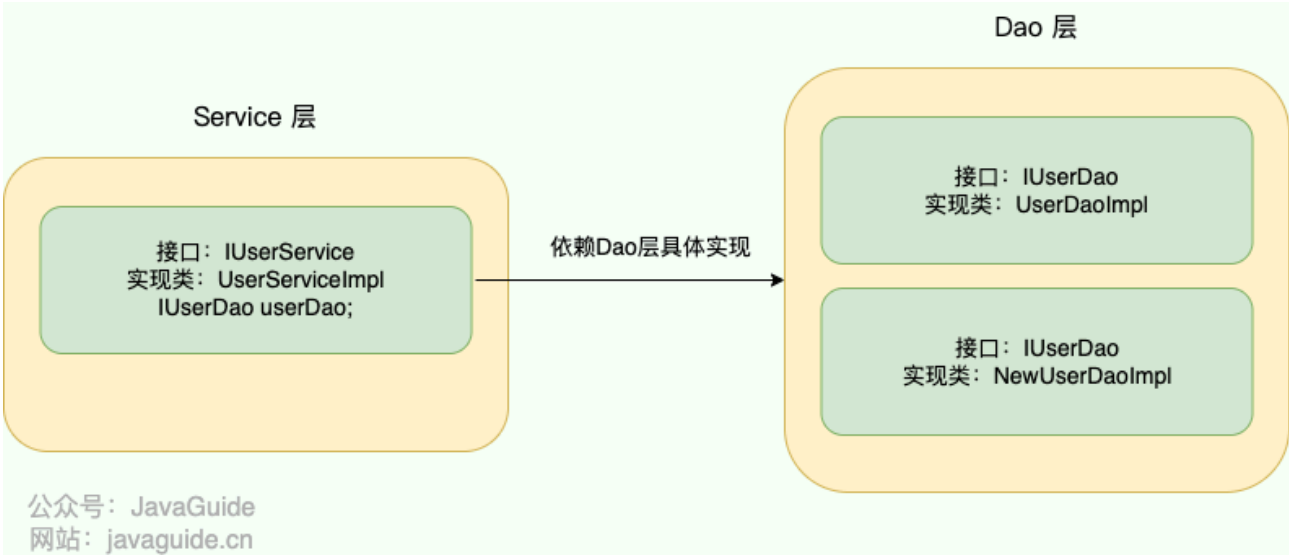
Without using the IoC concept, if the Service layer wants to use the specific implementation of the Dao layer, it needs to manually create a new specific implementation class in `UserServiceImpl` the new layer using the new keyword (it cannot directly create a new interface class). `IUserDao` `UserDaoImpl`

Perfect, this approach is also achievable, but let's imagine the following scenario:

During development, a new requirement suddenly arises: `IUserDao` developing a new implementation class for an interface. Because the server layer relies on `IUserDao` the implementation, we need to modify `UserServiceImpl` the new object. If only one class references `IUserDao` the implementation, the modification might be easy. However, if `IUserDao` the implementation is referenced in numerous places, changing the `IUserDao` implementation can be a real headache.



Using the idea of IoC, we hand over the control (creation and management) of the object to the IoC container to manage, and when we use it, we can just "ask" from the IoC container.

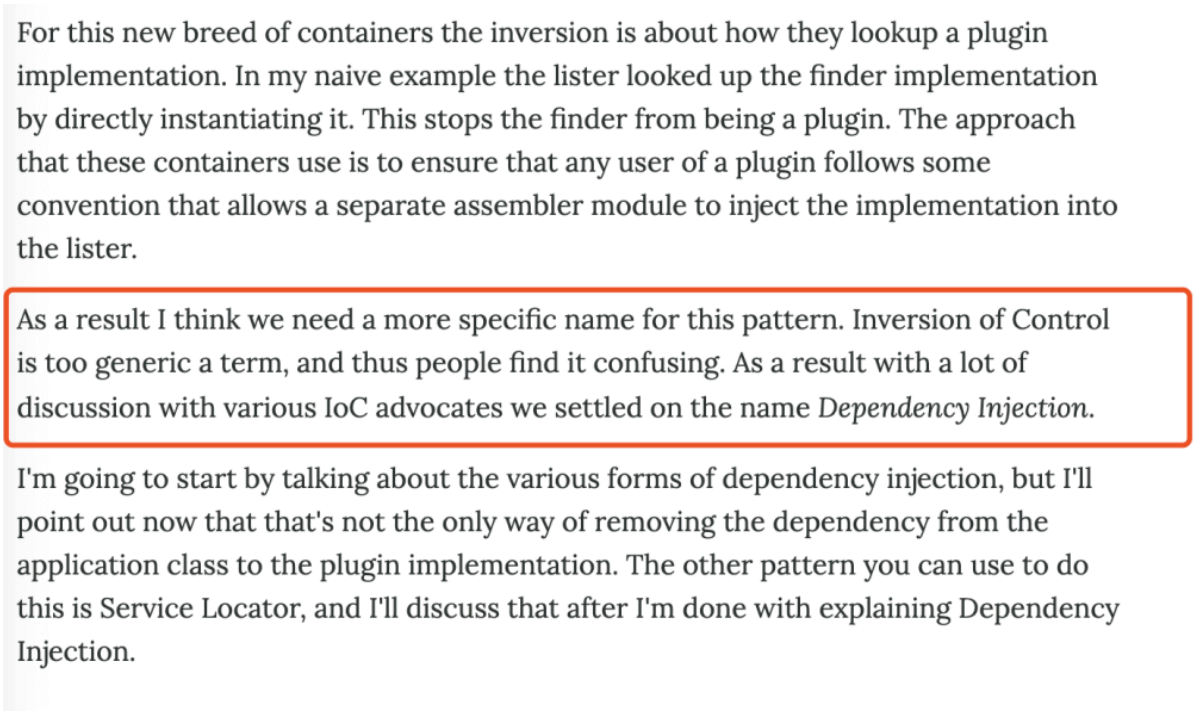


Is there a difference between IoC and DI?

IoC (Inverse of Control) is a design concept or pattern that **transfers control over manually created objects within a program to a third party, such as an IoC container.** In the commonly used Spring framework, the IoC container is essentially a Map (key, value) that stores various objects. However, IoC is also used in other languages and is not unique to Spring.

The most common and reasonable implementation of IoC is called Dependency Injection (DI).

Martin Fowler mentioned in an article that IoC should be renamed as DI. The original text is as follows, the original address is: https://martinfowler.com/articles/injection.html .



For this new breed of containers the inversion is about how they lookup a plugin implementation. In my naive example the lister looked up the finder implementation by directly instantiating it. This stops the finder from being a plugin. The approach that these containers use is to ensure that any user of a plugin follows some convention that allows a separate assembler module to inject the implementation into the lister.

As a result I think we need a more specific name for this pattern. Inversion of Control is too generic a term, and thus people find it confusing. As a result with a lot of discussion with various IoC advocates we settled on the name *Dependency Injection*.

I'm going to start by talking about the various forms of dependency injection, but I'll point out now that that's not the only way of removing the dependency from the application class to the plugin implementation. The other pattern you can use to do this is Service Locator, and I'll discuss that after I'm done with explaining Dependency Injection.

What Lao Ma probably meant is that IoC is too common and vague, and many people will be confused by it, so it is better to use DI to accurately name this pattern.

AOP (Aspect oriented programming)

We will not involve too many professional terms here. The core purpose is to make the idea of AOP clear.

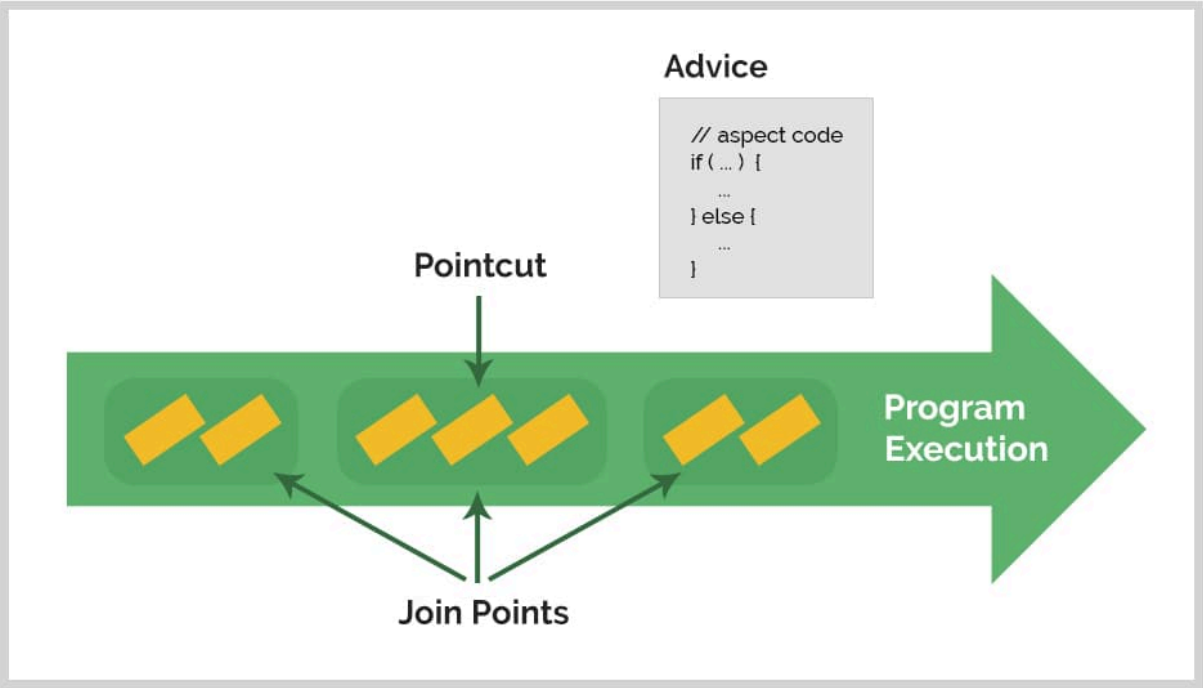
What is AOP?

AOP (Aspect Oriented Programming) is aspect-oriented programming. AOP is a continuation of OOP (object-oriented programming). The two complement each other and are not contradictory.

The purpose of AOP is to separate cross-cutting concerns (such as logging, transaction management, permission control, interface current limiting, interface idempotence, etc.) from core business logic. Through technologies such as dynamic proxies and bytecode manipulation, it enables code reuse and decoupling, improving code maintainability and scalability. The purpose of OOP is to encapsulate business logic according to the attributes and behaviors of objects. Through concepts such as classes, objects, inheritance, and polymorphism, it achieves code modularity and hierarchy (which also enables code r improving code readability and maintainability.

Why is AOP called Aspect-Oriented Programming?

AOP is called aspect-oriented programming because its core idea is to separate cross-cutting concerns from the core business logic and form **aspects** .



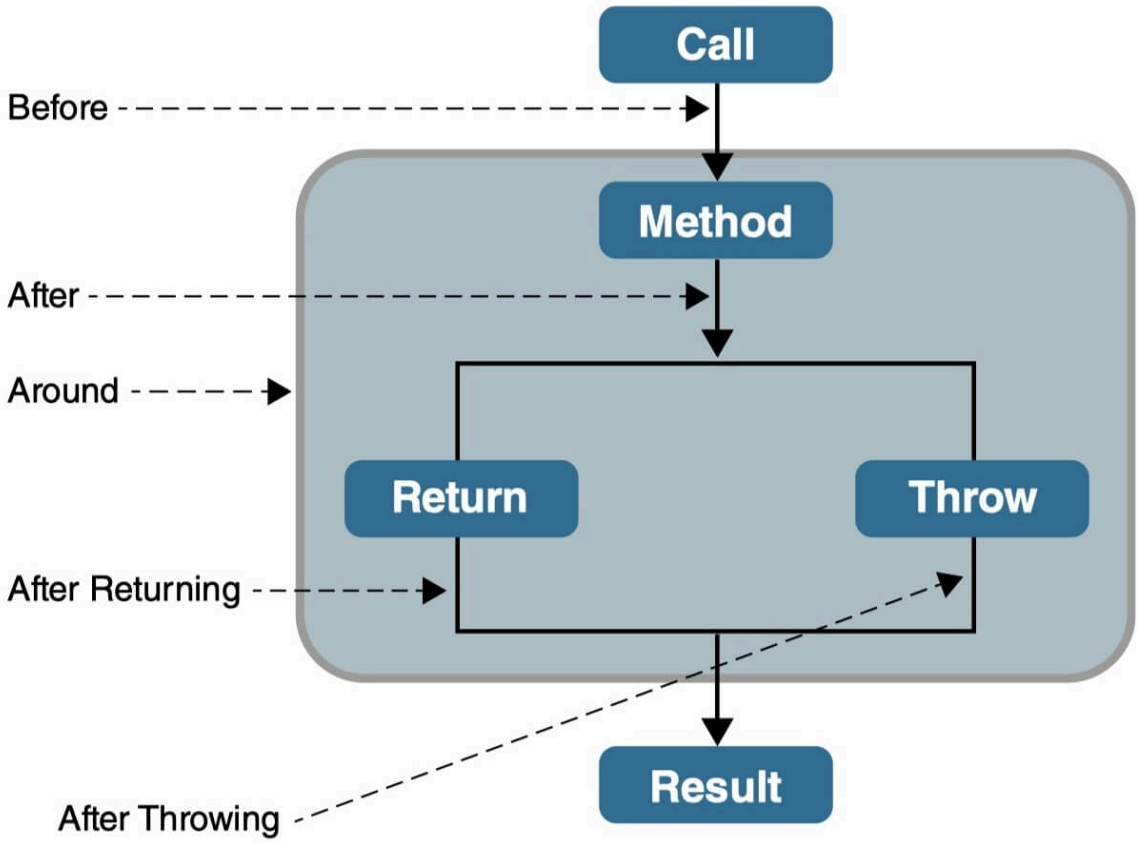
Here is a summary of the key AOP terms (it doesn't matter if you don't understand, you can continue reading):

- **Cross-cutting concerns** : Common behaviors in multiple classes or objects (such as logging, transaction management, permission control, interface current limiting, interface idempotence, etc.).
- **Aspect** : A class that encapsulates cross-cutting concerns. An aspect is a class. An aspect can define multiple notifications to implement specific functionality.
- **JoinPoint** : A join point is a specific moment when a method is called or executed (such as a method call, an exception is thrown, etc.).
- **Advice** : Advice is the action that an aspect performs at a join point. There are five types of advice: before, after, after-returning, after-throwing, and around. The first four types of advice execute before or after the target method, while around advice can control the execution of the target method.
- **Pointcut** : A pointcut is an expression that matches join points that should be enhanced by an aspect. Pointcuts can be defined using annotations, regular expressions,

logical operators, and more. For example, they `execution(* com.xyz.service..*(..))` match `com.xyz.service` classes or interfaces within a package and its subpackages.

- **Weaving** : Weaving is the process of connecting aspects to target objects, applying advice to join points that match pointcuts. There are two common weaving scenarios: compile-time weaving (e.g., AspectJ) and runtime weaving (e.g., AspectJ and Spring AOP).

What are the common types of AOP advice?



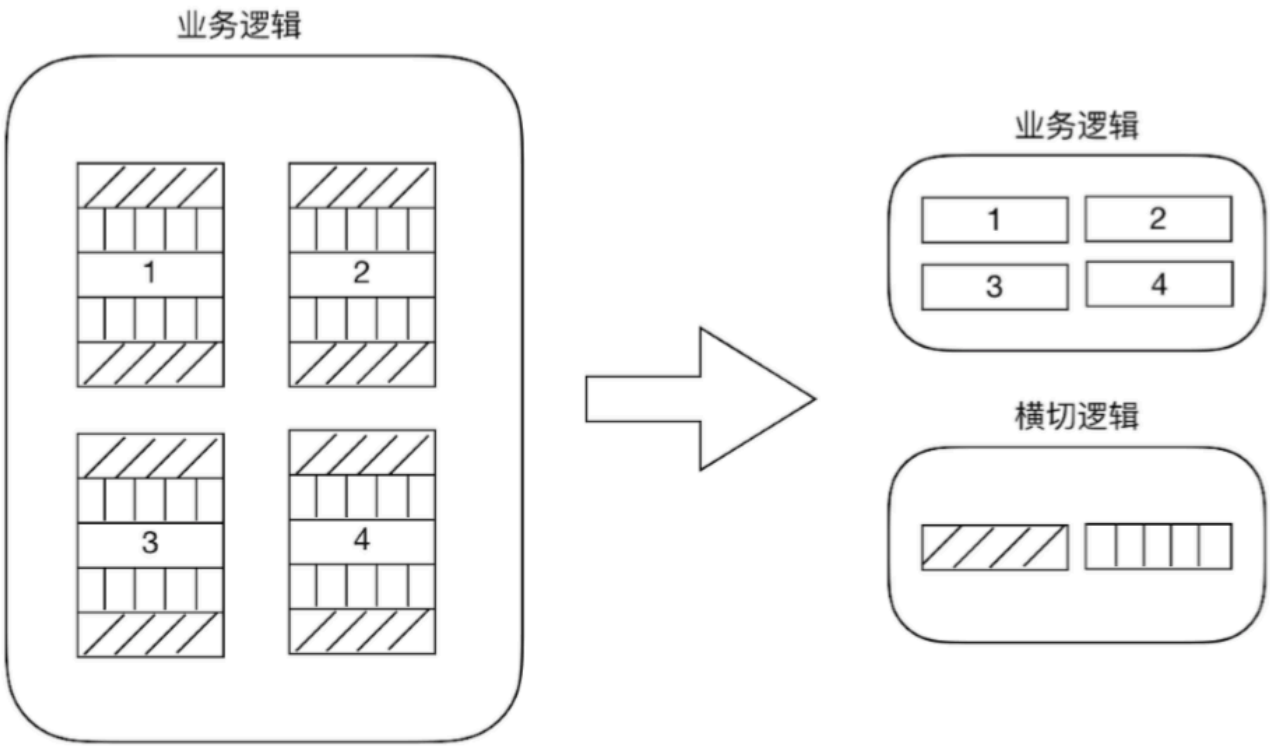
- **Before** (pre-notification): triggered before the target object's method is called
- **After** (post notification): triggered after the target object's method is called
- **AfterReturning** (return notification): triggered after the method call of the target object is completed and the result value is returned
- **AfterThrowing** (Exception Notification): Triggered after a method on the target object throws or triggers an exception. AfterReturning and AfterThrowing are mutually exclusive. If the method call succeeds without exception, a value is returned; if the method throws an exception, no value is returned.
- **Around** advice: Programmatically control the method call of the target object. Around advice has the widest range of operation among all advice types. Because it can directly

obtain the target object and the method to be executed, around advice can arbitrarily do things before or after the target object's method call, or even not call the target object's method.

What problem does AOP solve?

OOP doesn't handle common behaviors that are spread across multiple classes or objects well (such as logging, transaction management, permission control, interface throttling, interface idempotence, and so on). These behaviors are often called **cross-cutting concerns** . If we duplicate these behaviors in every class or object, the code becomes redundant, complex, and difficult to maintain.

AOP can separate cross-cutting concerns (such as logging, transaction management, permission control, interface current limiting, interface idempotence, etc.) from **core business logic (core concerns)** , achieving separation of concerns.



Let’s take logging as an example. If we need to log certain methods in a unified format, before using AOP technology, we need to write the logging logic code one by one, which is all repetitive logic.

```java
public CommonResponse<Object> method1() {
      //
      xxService.method1();
      //
      //
      ServletRequestAttributes attributes =
(ServletRequestAttributes)
RequestContextHolder.getRequestAttributes();
      HttpServletRequest request = attributes.getRequest();
      //                                                    ...
      return CommonResponse.success();
}

public CommonResponse<Object> method2() {
      //
      xxService.method2();
      //
      //
      ServletRequestAttributes attributes =
(ServletRequestAttributes)
RequestContextHolder.getRequestAttributes();
      HttpServletRequest request = attributes.getRequest();
      //                                                    ...
      return CommonResponse.success();
}

// ...
```

After using AOP technology, we can encapsulate the logging logic into an aspect, and then use pointcuts and notifications to specify which methods need to perform logging operations.

```java

//
@Target({ElementType.PARAMETER,ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Log {

    /**
     *
     */
```

```
    String description() default "";

    /**
     *         INSERT DELETE UPDATE OTHER
     */
    MethodType methodType() default MethodType.OTHER;
}

//
@Component
@Aspect
public class LogAspect {
  //              Log
  @Pointcut("@annotation(cn.javaguide.annotation.Log)")
  public void webLog() {
  }

   /**
   *
   */
  @Around("webLog()")
  public Object doAround(ProceedingJoinPoint joinPoint) throws
Throwable {
    //
  }

  //
}
```

In this case, we can implement logging with one line of annotation:

```java
@Log(description = "method1",methodType = MethodType.INSERT)
public CommonResponse<Object> method1() {
      //
      xxService.method1();
      //
      return CommonResponse.success();
}
```

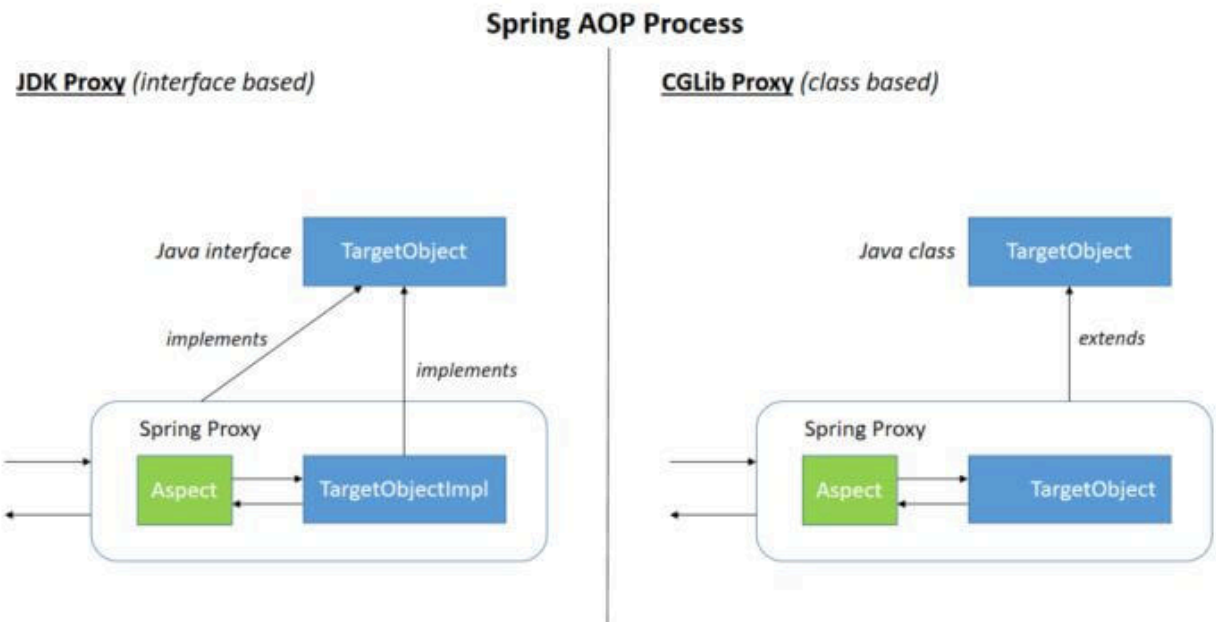
What are the application scenarios of AOP?

- Logging: Customize logging annotations and use AOP to implement logging with just one line of code.
- Performance statistics: Use AOP to count the execution time of the target method before and after execution, which facilitates optimization and analysis.
- Transaction management: `@Transactional` Annotations allow Spring to manage transactions for us, such as rolling back exceptions, eliminating the need for duplicate transaction management logic. `@Transactional` Annotations are implemented based on AOP.
- Permission control: Use AOP to determine whether the user has the required permissions before the target method is executed. If so, the target method is executed; otherwise, it is not executed. For example, Spring Security uses `@PreAuthorize` annotations to customize permission checks in just one line of code.
- Interface current limiting: Use AOP to limit the request through specific current limiting algorithms and implementations before the target method is executed.
- Cache management: Use AOP to read and update the cache before and after the target method is executed.
- ...

What are the ways to implement AOP?

Common implementation methods of AOP include dynamic proxy and bytecode manipulation.

Spring AOP is based on dynamic proxy. If the object to be proxied implements an interface, Spring AOP will use **JDK Proxy** to create a proxy object. However, for objects that do not implement the interface, JDK Proxy cannot be used for proxying. In this case, Spring AOP will use CGLIB to generate a subclass of the proxied object as a proxy, as shown in the following figure:



**Are the dynamic proxy strategies of Spring Boot and Spring the same? **Actually, they are different. Many people have misunderstood them.

Prior to Spring Boot 2.0, **JDK dynamic proxies** were used by default . If the target class did not implement an interface, an exception would be thrown, and the developer had to explicitly configure (`spring.aop.proxy-target-class=true`) **CGLIB dynamic proxies** or inject interfaces to resolve the issue. The code for automatically configuring AOP in Spring Boot 1.5.x is as follows:

```java
@Configuration
@ConditionalOnClass({ EnableAspectJAutoProxy.class, Aspect.class,
Advice.class })
@ConditionalOnProperty(prefix = "spring.aop", name = "auto",
havingValue = "true", matchIfMissing = true)
public class AopAutoConfiguration {

    @Configuration
    @EnableAspectJAutoProxy(proxyTargetClass = false)
 //              spring.aop.proxy-target-class=false

 //                                    Spring          JDK
    @ConditionalOnProperty(prefix = "spring.aop", name = "proxy-
target-class", havingValue = "false", matchIfMissing = true)
    public static class JdkDynamicAutoProxyConfiguration {

    }

```

```
    @Configuration
    @EnableAspectJAutoProxy(proxyTargetClass = true)
 //             spring.aop.proxy-target-class=true
 //                              CGLIB           Spring

    @ConditionalOnProperty(prefix = "spring.aop", name = "proxy-
target-class", havingValue = "true", matchIfMissing = false)
    public static class CglibAutoProxyConfiguration {

    }

}
```

Starting with Spring Boot 2.0, if the user does not configure anything, **CGLIB dynamic proxy** is used by default . If you need to force the use of JDK dynamic proxy, you can add: to the configuration file `spring.aop.proxy-target-class=false` . The Spring Boot 2.0 automatic configuration AOP code is as follows:

java

```
@Configuration
@ConditionalOnClass({ EnableAspectJAutoProxy.class, Aspect.class,
Advice.class,
        AnnotatedElement.class })
@ConditionalOnProperty(prefix = "spring.aop", name = "auto",
havingValue = "true", matchIfMissing = true)
public class AopAutoConfiguration {

    @Configuration
    @EnableAspectJAutoProxy(proxyTargetClass = false)
 //             spring.aop.proxy-target-class=false
 //                              JDK           Spring

    @ConditionalOnProperty(prefix = "spring.aop", name = "proxy-
target-class", havingValue = "false", matchIfMissing = false)
    public static class JdkDynamicAutoProxyConfiguration {

    }

    @Configuration
    @EnableAspectJAutoProxy(proxyTargetClass = true)
 //             spring.aop.proxy-target-class=true

 //                                  Spring          CGLIB
```

```
25      @ConditionalOnProperty(prefix = "spring.aop", name = "proxy-
target-class", havingValue = "true", matchIfMissing = true)
    public static class CglibAutoProxyConfiguration {

    }

}
```

Of course you can also use **AspectJ** ! Spring AOP has integrated AspectJ, which should be considered the most complete AOP framework in the Java ecosystem.

Spring AOP is a runtime enhancement, while AspectJ is a compile-time enhancement. Spring AOP is based on proxying, while AspectJ is based on bytecode manipulation.

Spring AOP has integrated AspectJ, which is arguably the most complete AOP framework in the Java ecosystem. AspectJ is more powerful than Spring AOP, but Spring AOP is relatively simpler.

If we have relatively few aspects, then the performance difference between the two is not significant. However, when there are too many aspects, it is better to choose AspectJ, which is much faster than Spring AOP.

refer to

- AOP in Spring Boot, is it a JDK dynamic proxy or a Cglib dynamic proxy? https://www.springcloud.io/post/2022-01/springboot-aop/
- Spring Proxying Mechanisms: https://docs.spring.io/spring-framework/reference/core/aop/proxying.html

Recently Updated2025/4/2 14:46
Contributors: Guide , Mr.Hope , ShimenTian , gaohang