Please visit website: http://cxyroad.com

什么？你用service调controller，那参数校验怎么办？

你好，我是柳岸花开。

你有没有遇到过用service调controller的场景？我司公有云用的微服务架构，私有云则用的合并完的微服务架构——一个单体，这就涉及到**用service调controller**，这个明显的问题是参数校验会失效，如果解决呢？

在任何软件系统中，确保数据的可靠性和完整性至关重要。特别是在微服务架构等复杂生态系统中，多个服务相互交互时，验证输入参数变得至关重要。在本文中，我们将探讨如何通过使用Spring AOP和Bean Validation实现统一的参数验证方法，从而提高API的可靠性。

引言：参数验证的重要性

想象一下，一个API接收到了格式不正确或无效的数据。这种情况可能导致意外行为、安全漏洞，甚至系统崩溃。为了降低这些风险，有必要在进一步处理之前验证输入参数。传统上，每个API端点可能独立处理验证，导致代码重复和不一致。然而，通过集中验证逻辑，我们可以简化流程，并确保应用程序的统一性。

使用Spring AOP实现统一验证

Spring面向切面编程（AOP）提供了一种强大的机制，用于模块化跨越点，如日志记录、安全性和验证。利用AOP，我们可以拦截方法调用，并在多个端点上统一应用验证逻辑。

在我们的实现中，我们创建了一个`ValidatorAspect`类，并用`@Aspect`注解进行注释，以将其标记为切面。该切面使用切入点表达式拦截特定包内控制器类的方法调用。

```
@Data
```

```
@Aspect
@Component
public class ValidatorAspect {
}
```

使用Bean Validation进行参数验证

为了执行参数验证，我们利用Bean Validation，这是一个标准的Java EE规范，提供了一种声明性的验证Java对象的方式。通过使用验证约束，如`@NotNull`或`@Size`，对方法参数进行注解，我们定义了可接受数据的规则。

在我们的切面中，我们注入了一个`SpringValidatorAdapter`，以利用Spring的验证能力。在拦截方法调用时，我们遍历方法参数，针对每个参数进行验证，并根据定义的约束检查是否违反。如果出现任何违规情况，我们抛出一个`ConstraintViolationException`，表示输入无效。

```
@Data
@Aspect
@Component
public class ValidatorAspect {

    private final SpringValidatorAdapter validator;

    /**
     * 拦截所有Controller
     *
     * @param joinPoint joinPoint
     */
    @Before("execution(* bob..*Controller.*(..)) || execution(*
log..*Controller.*(..)) || execution(* workflow..*Controller.*(..))")
    public void before(JoinPoint joinPoint) {
        for (Object arg : joinPoint.getArgs()) {
            Set<ConstraintViolation<Object>> constraintViolationSet =
validator.validate(arg);
            if (constraintViolationSet.isEmpty()) {
                continue;
            }
            ConstraintViolation<Object> constraintViolation =
constraintViolationSet.iterator().next();
            throw new
```

```
ConstraintViolationException(constraintViolation.getPropertyPath() +
constraintViolation.getMessage(), null);
        }

    }

}
```

通过部署模式条件保证一致性

为了保持灵活性，我们的验证机制根据部署模式有条件地启用。使用`@ConditionalOnDeployMode`，我们确保验证仅在特定的部署环境下生效，例如我们的情况下的“合并”模式。

```
@Data
@Aspect
@Component
@ConditionalOnDeployMode(mode = DeployModeEnum.MERGE)
public class ValidatorAspect {

    private final SpringValidatorAdapter validator;

    /**
     * 拦截所有Controller
     *
     * @param joinPoint joinPoint
     */
    @Before("execution(* bob..*Controller.*(..)) || execution(*
log..*Controller.*(..)) || execution(* workflow..*Controller.*(..))")
    public void before(JoinPoint joinPoint) {
        for (Object arg : joinPoint.getArgs()) {
            Set<ConstraintViolation<Object>> constraintViolationSet =
validator.validate(arg);
            if (constraintViolationSet.isEmpty()) {
                continue;
            }
            ConstraintViolation<Object> constraintViolation =
constraintViolationSet.iterator().next();
            throw new
ConstraintViolationException(constraintViolation.getPropertyPath() +
constraintViolation.getMessage(), null);
```

```
        }

    }

}

```

结论
--

总之，通过使用Spring AOP和Bean Validation实现统一的参数验证方法，我们提高了API的可靠性和健壮性。集中验证逻辑不仅减少了代码重复，还确保了应用程序的一致性。通过基于部署模式的条件激活，我们在不牺牲验证完整性的情况下保持了灵活性。通过采用这样的实践，我们可以在当今的动态环境中构建更具韧性和安全性的软件系统。

本文由[mdnice](http://cxyroad.com/
”https://mdnice.com/?platform=2”)多平台发布

原文链接: https://juejin.cn/post/7364448176574775347