

利用Spring AOP自定义注解解决日志和签名校验

一、需解决的问题

1. 部分API有签名参数（signature），Passport首先对签名进行校验，校验通过才会执行实现方法。

第一种实现方式(Origin)：在需要签名校验的接口里写校验的代码，例如：

```
boolean isValid = accountService.validSignature(appid, signature, client_signature);
if (!isValid) return ErrorUtil.buildError(ErrorUtil.ERR_CODE_COM_SING);
```

第二种实现方式(Spring Interception)：利用spring的拦截器功能，对指定的接口进行拦截，拦截器实现签名校验算法，例如：

```
<mvc:interceptors>
  <mvc:interceptor>
    <mvc:mapping path="/connect/share/**" />
    <mvc:mapping path="/friend/**" />
    <mvc:mapping path="/account/get_bind" />
    <mvc:mapping path="/account/get_associate" />
    <bean class="com.sogou.upd.passport.web.inteceptor.IdentityAndSecureInteceptor" />
  </mvc:interceptor>
</mvc:interceptors>
```

第三种实现方式(spring AOP)：自定义注解，对需要进行签名验证的方法添加注解，例如：

```
@SecureValid
@ResponseBody
@RequestMapping(value = "/share/add", method = RequestMethod.POST)
public Object addShare(HttpServletRequest req, HttpServletResponse res, InfoAPIRequestParams requestParams) {
    ...
}
```

2. 日志记录功能，例如：某些接口需要记录请求和响应，执行时间，类名，方法名等日志信息。也可采用以上三种方式实现。

3. 代码性能监控问题，例如方法调用时间、次数、线程和堆栈信息等。这类问题在后一个专题提出解决方案，采用以上三种方式实现缺点太多。

<2018年4月>

日	一	二	三	四	五	六
25	26	27	28	29	30	31
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	1	2	3	4	5

搜索

找找看

随笔档案

- 2012年10月 (1)
- 2012年9月 (2)
- 2012年8月 (1)
- 2012年6月 (1)
- 2012年2月 (1)
- 2011年10月 (1)
- 2011年7月 (5)
- 2011年6月 (6)
- 2011年5月 (15)
- 2011年4月 (2)

文章分类


- Hadoop(4)
- JAVA(20)
- JavaScript(7)
- Java开源(4)
- js/jQuery(1)
- Linux(4)
- Mahout(11)
- Spring(5)
- Web前端(4)
- 设计模式(1)
- 数据库(1)
- 统计分析笔记(3)

以下是三种实现方式比较：


实现方式	优点	缺点
Origin	不采用反射机制，性能最佳	逻辑复杂时，代码复用不好 需要在每个接口里写入相同代码（我太懒，就想写几个字母）
Spring Inter	非常适合对所有方法进行拦截，例如调试时打印所有方法执行时间 类似过滤器的功能，如日志处理、编码转换、权限检查是AOP的子功能	不采用反射机制，性能有所影响 需要在xml文件里配置对哪些接口进行拦截，比较麻烦
Spring AOP	使用方便，增加一个注解 非常灵活，可@Before，@After，@Around等	不采用反射机制，性能有所影响（性能对比后面详细展示）

二、Spring AOP 自定义注解的实现


在Maven中加入以下以依赖：



```
<!-- Spring AOP + AspectJ by shipengzhi -->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aop</artifactId>
  <version>3.0.6.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aspects</artifactId>
  <version>3.0.6.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjrt</artifactId>
  <version>1.6.11</version>
</dependency>
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjweaver</artifactId>
  <version>1.6.11</version>
</dependency>
<dependency>
  <groupId>cglib</groupId>
  <artifactId>cglib</artifactId>
  <version>2.1_3</version>
</dependency>
<!-- end -->
```



在spring-***.xml中加入spring支持，打开aop功能



```
头文件声明：
xmlns:aop="http://www.springframework.org/schema/aop"
```

Ajax框架学习

为什么“惜时”这么难？

Hadoop

- 基于 Apache Mahout 构建社会化推荐引擎
- 浅析Hadoop文件格式
- 探索推荐引擎内部的秘密，第 1 部分：推荐引擎初探
- 探索推荐引擎内部的秘密，第 2 部分：深入推荐引擎相关算法 - 协同过滤
- 探索推荐引擎内部的秘密，第 3 部分：深入推荐引擎相关算法 - 聚类
- 用 Hadoop MapReduce 进行大数据分析

JAVA

- Future & Promise
- Java SE1.6中的Synchronized
- Java 序列化的高级认识
- java.lang.ref--1. 用弱引用堵住内存泄漏
- Java并发编程Executor框架+Future
- java并发编程-构建块
- JAVA线程dump的分析 --- jstack pid
- JDK 5.0 中的并发
- 案例分析：基于消息的分布式架构
- 关于 java.util.concurrent 您不知道的 5 件事
- 关于 java.util.concurrent 您不知道的 5 件事，第 1 部分
- 关于多线程编程您不知道的 5 件事
- 深入探讨 java.lang.ref 包
- 探索并发编程

JavaScript

面向对象的 JavaScript 编程及其 Scope 处理

Java开源

- 分布式key/value存储系统 Tair
- 使用Jakarta Commons Pool处理对象池化

NoSQL

```
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
<!-- 自定义AOP -->
<aop:aspectj-autoproxy proxy-target-class="true">
    <aop:include name="controllerAspect" />
</aop:aspectj-autoproxy>
<bean id="controllerAspect" class="com.sogou.upd.passport.common.aspect.ControllerAspect">
</bean>

或:
<aop:aspectj-autoproxy>
```

MongoDB、Java及ORM
NoSQL 数据建模技术

Python/Shell

使用 Python RQ 的 Python 执行后台任务

Spring

使用 Spring 2.5 注释驱动的 IoC 功能

编写自定义注解。实现与方法所实现的功能进行描述，以便在通知中获取描述信息

```
/*
 * 校验签名合法性 自定义事务
 */
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
public @interface SecureValid {
    String desc() default "身份和安全验证开始...";
}
```

知识库

B树、B-树、B+树、B*树都是什么
B树、B-树、B+树、B*树都是什么
Corba
ExtJS--一种Ajax框架
Ext中文站
maven常见问题问答
Turbine百科
Webx简介

Copyright ©2018 跳刀的兔子

@Target 用于描述注解的使用范围（即：被描述的注解可以用在什么地方），其取值有：

取值	描述
CONSTRUCTOR	用于描述构造器（领盒饭）。
FIELD	用于描述域（领盒饭）。
LOCAL_VARIABLE	用于描述局部变量（领盒饭）。
METHOD	用于描述方法。
PACKAGE	用于描述包（领盒饭）。
PARAMETER	用于描述参数。
TYPE	用于描述类或接口（甚至 enum）。

@Retention 用于描述注解的生命周期（即：被描述的注解在什么范围内有效），其取值有：

取值	描述
SOURCE	在源文件中有效（即源文件保留，领盒饭）。

CLASS	在 class 文件中有效（即 class 保留，领盒饭）。
RUNTIME	在运行时有效（即运行时保留）。

@Documented 在默认的情况下javadoc命令不会将我们的annotation生成再doc中去的，所以使用该标记就是告诉jdk让它也将annotation生成到doc中去

@Inherited 比如有一个类A，在他上面有一个标记annotation，那么A的子类B是否不用再次标记annotation就可以继承得到呢，答案是肯定的

Annotation属性值 有以下三种：基本类型、数组类型、枚举类型

1：基本串类型

```
public @interface UserdefinedAnnotation {
    int value();
    String name();
    String address();
}

使用：
@UserdefinedAnnotation(value=123,name="wangwenjun",address="火星")
public static void main(String[] args) {
    System.out.println("hello");
}
```

如果一个annotation中只有一个属性名字叫value，我没在使用的时候可以给出属性名也可以省略。

```
public @interface UserdefinedAnnotation {
    int value();
}

也可以写成如下的形式

Java代码
@UserdefinedAnnotation(123) 等同于@UserdefinedAnnotation(value=123)
public static void main(String[] args) {
    System.out.println("hello");
}
```

2：数组类型 我们在自定义annotation中定义一个数组类型的属性，代码如下：

```
public @interface UserdefinedAnnotation {
    int[] value();
}

使用：
public class UseAnnotation {

    @UserdefinedAnnotation({123})
    public static void main(String[] args) {
        System.out.println("hello");
    }
}
```



注意1：其中123外面的大括号是可以被省略的，因为只有一个元素，如果里面有一个以上的元素的话，花括号是不能被省略的哦。比如{123, 234}。

注意2：其中属性名value我们在使用的时候进行了省略，那是因为他叫value，如果是其他名字我们就不可以进行省略了必须是@UserdefinedAnnotation（属性名={123,234}）这样的格式。

3：枚举类型



```
public enum DateEnum {  
    Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday  
}
```

然后在定义一个annotation

```
package com.wangwenjun.annatation.userdefined;
```

```
public @interface UserdefinedAnnotation {  
    DateEnum week();  
}
```

使用：

```
public class UseAnnotation {  
    @UserdefinedAnnotation(week=DateEnum.Sunday)  
    public static void main(String[] args) {  
        System.out.println("hello");  
    }  
}
```



4：默认值



```
public @interface UserdefinedAnnotation {  
    String name() default "zhangsan";  
}
```

使用：

```
public class UseAnnotation {  
    @UserdefinedAnnotation()  
    public static void main(String[] args) {  
        System.out.println("hello");  
    }  
}
```



5：注意

Annotation是不可以继承其他接口的，这一点是需要进行注意，这也是annotation的一个规定吧。

Annotation也是存在包结构的，在使用的时候直接进行导入即可。

Annotation类型的类型只支持原声数据类型，枚举类型和Class类型的一维数组，其他的类型或者用户自定义的类都是不可以作为annotation的类型，我查看过文档并且进行过测试。

编写操作日志切面通知实现类

在编写切面通知实现类之前，我们需要搞清楚我们需要哪些通知类型，是前置通知、后置通知、环绕通知或异常通知？根据我的需求，我们知道我们记录的操作日志有两种情况，一种是操作成功，一种是操作失败。操作成功时则方法肯定已经执行完成，顾我们需要实现一个后置通知；操作失败时则说明方法出现异常无法正常执行完成，顾还需要一个异常通知。代码如下：



```
@Aspect //该注解标示该类为切面类
```

```
@Component //注入依赖
public class LogAspect {

    //标注该方法体为后置通知, 当目标方法执行成功后执行该方法体
    @AfterReturning("within(com.abchina.irms..*) && @annotation(rl)")
    public void addLogSuccess(JoinPoint jp, rmpfLog rl){
        Object[] params = jp.getArgs();//获取目标方法体参数
        String params = parseParams(params); //解析目标方法体的参数
        String className = jp.getTarget().getClass().toString();//获取目标类名
        className = className.substring(className.indexOf("com"));
        String signature = jp.getSignature().toString();//获取目标方法签名
        String methodName = signature.substring(signature.lastIndexOf(".")+1, signature.indexOf("
"));

        String modelName = getModelName(className); //根据类名获取所属的模块

        ...
    }

    //标注该方法体为异常通知, 当目标方法出现异常时, 执行该方法体
    @AfterThrowing(pointcut="within(com.abchina.irms..*) && @annotation(rl)", throwing="ex")
    public void addLog(JoinPoint jp, rmpfLog rl, BusinessException ex){

        ...
    }
}
```



有两个相同的参数jp和rl, jp是切点对象, 通过该对象可以获取切点所切入方法所在的类, 方法名、参数等信息, 具体方法可以看方法体的实现; rl则是我们的自定义注解的对象, 通过该对象我们可以获取注解中参数值, 从而获取方法的描述信息。在异常通知中多出了一个ex参数, 该参数是方法执行时所抛出的异常, 从而可以获取相应的异常信息。此处为我写的自定义异常。注意: 如果指定异常参数, 则异常对象必须与通知所切入的方法体抛出的异常保持一致, 否则该通知不会执行。

@AfterReturning("within(com.abchina.irms..*) && @annotation(rl)")注解, 是指定该方法体为后置通知, 其有一个表达式参数, 用来检索符合条件的切点。该表达式指定com/abchina/irms目录下及其所有子目录下的所有带有@rmpfLog注解的方法体为切点。

@AfterThrowing(pointcut="within(com.abchina.irms..*) && @annotation(rl)", throwing="ex")注解, 是指定方法体为异常通知, 其有一个表达式参数和一个抛出异常参数。表达式参数与后置通知的表达式参数含义相同, 而抛出异常参数, 则表示如果com/abchina/irms目录下及其所有子目录下的所有带有@rmpfLog注解的方法体在执行时抛出BusinessException异常时该通知便会执行。

AOP的基本概念:

- 切面 (Aspect) : 通知和切入点共同组成了切面, 时间、地点和要发生的“故事”。
- 连接点 (Joinpoint) : 程序能够应用通知的一个“时机”, 这些“时机”就是连接点, 例如方法被调用时、异常被抛出时等等。
- 通知 (Advice) : 通知定义了切面是什么以及何时使用。描述了切面要完成的工作和何时需要执行这个工作。
- 切入点 (Pointcut) : 通知定义了切面要发生的“故事”和时间, 那么切入点就定义了“故事”发生的地点, 例如某个类或方法的名称。
- 目标对象 (Target Object) : 即被通知的对象。
- AOP代理 (AOP Proxy) 在Spring AOP中有两种代理方式, JDK动态代理和CGLIB代理。默认情况下, TargetObject实现了接口时, 则采用JDK动态代理; 反之, 采用CGLIB代理。
- 织入 (Weaving) 把切面应用到目标对象来创建新的代理对象的过程, 织入一般发生在如下几个时机:
 - (1) 编译时: 当一个类文件被编译时进行织入, 这需要特殊的编译器才能做到, 例如AspectJ的织入编译器;
 - (2) 类加载时: 使用特殊的ClassLoader在目标类被加载到程序之前增强类的字节代码;

(3) 运行时：切面在运行的某个时刻被织入，SpringAOP就是以这种方式织入切面的，原理是使用了JDK的动态代理。

切入点表达式，**Pointcut**的定义包括两个部分：Pointcut表示式(expression)和Pointcut签名(signature)。让我们先看看execution表示式的格式：

```
execution(modifier-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern)
throws-pattern?)
pattern分别表示修饰符匹配 (modifier-pattern?)、返回值匹配 (ret-type-pattern)、类路径匹配 (declaring-type-pattern?)、方法名匹配 (name-pattern)、参数匹配 ((param-pattern))、异常类型匹配 (throws-pattern?)，其中后面跟着“?”的是可选项。
```

在各个pattern中可以使用“*”来表示匹配所有。在(param-pattern)中，可以指定具体的参数类型，多个参数间用“,”隔开，各个也可以用“*”来表示匹配任意类型的参数，如(String)表示匹配一个String参数的方法；(*,String)表示匹配有两个参数的方法，第一个参数可以是任意类型，而第二个参数是String类型；可以用(..)表示零个或多个任意参数。

现在来看看几个例子：

1) execution(* *(..)) 表示匹配所有方法

2) execution(public * com.savage.service.UserService.*(..)) 表示匹配com.savage.server.UserService中所有的公有方法

3) execution(* com.savage.server..*(..)) 表示匹配com.savage.server包及其子包下的所有方法

除了execution表示式外，还有within、this、target、args等Pointcut表示式。一个Pointcut定义由Pointcut表示式和Pointcut签名组成，例如：

```
//Pointcut表示式
@Pointcut("execution(* com.savage.aop.MessageSender.*(..))")
//Point签名
private void log() {}
然后要使用所定义的Pointcut时，可以指定Pointcut签名，如：
@Before("og() ")
上面的定义等同与：
@Before("execution(* com.savage.aop.MessageSender.*(..))")
Pointcut定义时，还可以使用&&、||、!运算，如：
@Pointcut("logSender() || logReceiver()")
private void logMessage() {}
```

通知 (Advice) 类型

@Before 前置通知 (Before advice)：在某连接点 (JoinPoint) 之前执行的通知，但这个通知不能阻止连接点前的执行。

@After 后通知 (After advice)：当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。

@AfterReturning 返回后通知 (After return advice)：在某连接点正常完成后执行的通知，不包括抛出异常的情况。

@Around 环绕通知 (Around advice)：包围一个连接点的通知，类似Web中Servlet规范中的Filter的doFilter方法。可以在方法的调用前后完成自定义的行为，也可以选择 not 执行。

@AfterThrowing 抛出异常后通知 (After throwing advice)：在方法抛出异常退出时执行的通知。

自定义注解实现在Controller层面

```
/**
 * 对Controller进行安全和身份校验
 */
@Around("within(@org.springframework.stereotype.Controller *) && @annotation(is)")
public Object validIdentityAndSecure(ProceedingJoinPoint pjp, SecureValid is)
    throws Exception {
```

```
Object[] args = pjp.getArgs();
//Controller中所有方法的参数, 前两个分别为: Request, Response
HttpServletRequest request = (HttpServletRequest) args[0];
// HttpServletResponse response = (HttpServletResponse) args[1];

String appid = request.getParameter("appid");
int app_id = Integer.valueOf(appid);
String signature = request.getParameter("signature");
String clientSignature = request.getParameter("client_signature");
String uri = request.getRequestURI();

String provider = request.getParameter("provider");
if (StringUtils.isEmpty(provider)) {
    provider = "passport";
}

// 对appid和signature进行校验
try {
    appService.validateAppid(app_id);
    boolean isValid = accountService.validSignature(app_id, signature, clientSignature);
    if (!isValid) throw new ProblemException(ErrorUtil.ERR_CODE_COM_SING);
} catch (Exception e) {
    return handleException(e, provider, uri);
}

// 继续执行接下来的代码
Object retVal = null;
try {
    retVal = pjp.proceed();
} catch (Throwable e) {
    if (e instanceof Exception) { return handleException((Exception) e, provider, uri); }
}

// 目前的接口走不到这里
return retVal;
}
```



三、Spring拦截器的实现

在spring-*.xml中加入拦截器的配置

编写拦截器实现类



```
public class CostTimeInteceptor extends HandlerInterceptorAdapter {

    private static final Logger log = LoggerFactory.getLogger(CostTimeInteceptor.class);

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        long startTime = System.currentTimeMillis();
        request.setAttribute("startTime", startTime);
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {
        long startTime = (Long) request.getAttribute("startTime");
    }
}
```



```
        long endTime = System.currentTimeMillis();
        long executeTime = endTime - startTime;
        if (log.isInfoEnabled()) {
            log.info "[" + request.getRequestURI() + "] executeTime : " + executeTime + "ms");
        }
    }
}
```

四、性能对比

实验环境：对/account/get_associate接口，并发500，压测10分钟

指标	Origin	Spring Inter	Spring AOP
CPU	user%:26.57	user%:26.246	user%:24.123
	sys%:10.97	sys%:10.805	sys%:9.938
	cpu%:37.541	cpu%:37.051	cpu%:34.062
Load	13.85	13.92	12.21
QPS	6169	6093.2	5813.27
RT	0.242ms	0.242ms	0.235ms

采用AOP对响应时间无明显影响

采用AOP对Load无明显影响

采用AOP对CPU无明显影响

结论：使用AOP性能方面影响可忽略

分类: [Spring](#)

好文要顶

关注我

收藏该文

跳刀的兔子

关注 - 0

粉丝 - 41

6

0

+加关注

« 上一篇: [Java keytool 安全证书学习笔记](#)
» 下一篇: [反射和动态代理性能对比](#)

posted @ 2012-10-09 00:39 [跳刀的兔子](#) 阅读(74505) 评论(3) [编辑](#) [收藏](#)

发表评论

- #1楼 2014-11-26 09:49 | [thero](#)

能不能把示例写的清楚点啊，或者有源码，感觉运行起来会好很多啊

支持(0) 反对(0)
- #2楼 2015-02-10 11:49 | [jast90](#)

性能测试是怎么测试的？

支持(0) 反对(0)
- #3楼 2016-09-21 15:29 | [手边星辰](#)

3. 代码性能监控问题，例如方法调用时间、次数、线程和堆栈信息等。这类问题在后一个专题提出解决方案，采用以上三种方式实现缺点太多。

支持(0) 反对(0)

[刷新评论](#) [刷新页面](#) [返回顶部](#)

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问网站首页](#)。

- 【推荐】超50万VC++源码: 大型组态工控、电力仿真CAD与GIS源码库！
- 【报名】2050 大会 - 博客园程序员团聚（5.25 杭州·云栖小镇）
- 【推荐】华为云服务器低至3.3折,免费带宽升级,返千元好礼
- 【招聘】花大价钱找技术大牛我们是认真的！
- 【活动】腾讯云cps推广奖励，高转化+20%佣金等你来拿



- 最新IT新闻:
- 法国开发专用安全聊天应用 禁止政府官员使用Telegram
 - Facebook高管发文解释如何收集数据 不登录也能收集
 - 《华盛顿邮报》获奖 亚马逊CEO趁机揭短特朗普
 - 俄罗斯电信监管机构开始封杀Telegram
 - 智能助手Siri太易被唤醒？苹果考虑使用AI改进
- » 更多新闻...



- 最新知识库文章:
- 如何识别人的技术能力和水平？
 - 写给自学者入门指南
 - 和程序员谈恋爱
 - 学会学习
 - 优秀技术人的管理陷阱
- » 更多知识库文章...