Java 基础 - 注解机制详解

> 注解是JDK1.5版本开始引入的一个特性，用于对代码进行说明，可以对包、类、接口、字段、方法参数、局部变量等进行注解。

注解基础

注解是JDK1.5版本开始引入的一个特性，用于对代码进行说明，可以对包、类、接口、字段、方法参数、局部变量等进行注解。它主要的作用有以下四方面：

- 生成文档，通过代码里标识的元数据生成javadoc文档。
- 编译检查，通过代码里标识的元数据让编译器在编译期间进行检查验证。
- 编译时动态处理，编译时通过代码里标识的元数据动态处理，例如动态生成代码。
- 运行时动态处理，运行时通过代码里标识的元数据动态处理，例如使用反射注入实例。

这么来说是比较抽象的，我们具体看下注解的常见分类：

- **Java自带的标准注解**，包括`@Override`、`@Deprecated`和`@SuppressWarnings`，分别用于标明重写某个方法、标明某个类或方法过时、标明要忽略的警告，用这些注解标明后编译器就会进行检查。
- **元注解**，元注解是用于定义注解的注解，包括`@Retention`、`@Target`、`@Inherited`、`@Documented`，`@Retention`用于标明注解被保留的阶段，`@Target`用于标明注解使用的范围，`@Inherited`用于标明注解可继承，`@Documented`用于标明是否生成javadoc文档。
- **自定义注解**，可以根据自己的需求定义注解，并可用元注解对自定义注解进行注解。

接下来我们通过这个分类角度来理解注解。

*Java*内置注解

我们从最为常见的Java内置的注解开始说起，先看下下面的代码：

```
class A{
    public void test() {

    }
}

class B extends A{

    /**
     * 重载父类的test方法
     */
    @Override
    public void test() {
    }

    /**
```

```
         * 被弃用的方法
         */
    @Deprecated
    public void oldMethod() {
    }

    /**
         * 忽略告警
         *
         * @return
         */
    @SuppressWarnings("rawtypes")
    public List processList() {
        List list = new ArrayList();
        return list;
    }
}
```

Java 1.5开始自带的标准注解，包括@Override、@Deprecated和@SuppressWarnings：

- @Override：表示当前的方法定义将覆盖父类中的方法
- @Deprecated：表示代码被弃用，如果使用了被@Deprecated注解的代码则编译器将发出警告
- @SuppressWarnings：表示关闭编译器警告信息

我们再具体看下这几个内置注解，同时通过这几个内置注解中的元注解的定义来引出元注解。

内置注解 - @Override

我们先来看一下这个注解类型的定义：

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}
```

从它的定义我们可以看到，这个注解可以被用来修饰方法，并且它只在编译时有效，在编译后的class文件中便不再存在。这个注解的作用我们大家都不陌生，那就是告诉编译器被修饰的方法是重写的父类的中的相同签名的方法，编译器会对此做出检查，若发现父类中不存在这个方法或是存在的方法签名不同，则会报错。

内置注解 - @Deprecated

这个注解的定义如下：

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(value={CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE})
public @interface Deprecated {
}
```

从它的定义我们可以知道，它会被文档化，能够保留到运行时，能够修饰构造方法、属性、局部变量、方法、包、参数、类型。这个注解的作用是告诉编译器被修饰的程序元素已被“废弃”，不再建议用户使用。

内置注解 - @SuppressWarnings

这个注解我们也比较常用到，先来看下它的定义：

```
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings {
    String[] value();
}
```

它能够修饰的程序元素包括类型、属性、方法、参数、构造器、局部变量，只能存活在源码时，取值为String[]。它的作用是告诉编译器忽略指定的警告信息，它可以取的值如下所示：

参数	作用	原描述
all	抑制所有警告	to suppress all warnings
boxing	抑制装箱、拆箱操作时候的警告	to suppress warnings relative to boxing/unboxing operations
cast	抑制映射相关的警告	to suppress warnings relative to cast operations
dep-ann	抑制启用注释的警告	to suppress warnings relative to deprecated annotation
deprecation	抑制过期方法警告	to suppress warnings relative to deprecation
fallthrough	抑制确在switch中缺失breaks的警告	to suppress warnings relative to missing breaks in switch statements
finally	抑制finally模块没有返回的警告	to suppress warnings relative to finally block that don’t return
hiding	抑制与隐藏变数的区域变数相关的警告	to suppress warnings relative to locals that hide variable（）
incomplete-switch	忽略没有完整的switch语句	to suppress warnings relative to missing entries in a switch statement (enum case)
nls	忽略非nls格式的字符	to suppress warnings relative to non-nls string literals
null	忽略对null的操作	to suppress warnings relative to null analysis
rawtype	使用generics时忽略没有指定相应的类型	to suppress warnings relative to un-specific types when using
restriction	抑制与使用不建议或禁止参照相关的警告	to suppress warnings relative to usage of discouraged or
serial	忽略在serializable类中没有声明serialVersionUID变量	to suppress warnings relative to missing serialVersionUID field for a serializable class
static-access	抑制不正确的静态访问方式警告	to suppress warnings relative to incorrect static access
synthetic-access	抑制子类没有按最优方法访问内部类的警告	to suppress warnings relative to unoptimized access from inner classes
unchecked	抑制没有进行类型检查操作的警告	to suppress warnings relative to unchecked operations
unqualified-field-access	抑制没有权限访问的域的警告	to suppress warnings relative to field access unqualified

参数	作用	原描述
unused	抑制没被使用过的代码的警告	to suppress warnings relative to unused code

元注解

上述内置注解的定义中使用了一些元注解（注解类型进行注解的注解类），在JDK 1.5中提供了4个标准的元注解：`@Target, @Retention, @Documented, @Inherited`, 在JDK 1.8中提供了两个元注解 `@Repeatable`和`@Native`。

元注解 - @Target

> Target注解的作用是：描述注解的使用范围（即：被修饰的注解可以用在什么地方）。

Target注解用来说明那些被它所注解的注解类可修饰的对象范围：注解可以用于修饰 packages、types（类、接口、枚举、注解类）、类成员（方法、构造方法、成员变量、枚举值）、方法参数和本地变量（如循环变量、catch参数），在定义注解类时使用了@Target 能够更加清晰的知道它能够被用来修饰哪些对象，它的取值范围定义在ElementType 枚举中。

```
public enum ElementType {

    TYPE, // 类、接口、枚举类

    FIELD, // 成员变量（包括：枚举常量）

    METHOD, // 成员方法

    PARAMETER, // 方法参数

    CONSTRUCTOR, // 构造方法

    LOCAL_VARIABLE, // 局部变量

    ANNOTATION_TYPE, // 注解类

    PACKAGE, // 可用于修饰：包

    TYPE_PARAMETER, // 类型参数，JDK 1.8 新增

    TYPE_USE // 使用类型的任何地方，JDK 1.8 新增

}
```

元注解 - @Retention & @RetentionTarget

> Reteniton注解的作用是：描述注解保留的时间范围（即：被描述的注解在它所修饰的类中可以被保留到何时）。

Reteniton注解用来限定那些被它所注解的注解类在注解到其他类上以后，可被保留到何时，一共有三种策略，定义在RetentionPolicy枚举中。

```
public enum RetentionPolicy {

    SOURCE,    // 源文件保留
    CLASS,       // 编译期保留，默认值
    RUNTIME   // 运行期保留，可通过反射去获取注解信息
}
```

为了验证应用了这三种策略的注解类有何区别，分别使用三种策略各定义一个注解类做测试。

```
@Retention(RetentionPolicy.SOURCE)
public @interface SourcePolicy {

}
@Retention(RetentionPolicy.CLASS)
public @interface ClassPolicy {

}
@Retention(RetentionPolicy.RUNTIME)
public @interface RuntimePolicy {

}
```

用定义好的三个注解类分别去注解一个方法。

```
public class RetentionTest {

    @SourcePolicy
    public void sourcePolicy() {
    }

    @ClassPolicy
    public void classPolicy() {
    }

    @RuntimePolicy
    public void runtimePolicy() {
    }
}
```

通过执行 `javap -verbose RetentionTest`命令获取到的RetentionTest 的 class 字节码内容如下。

```
{
  public retention.RetentionTest();
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
        line 3: 0

  public void sourcePolicy();
    flags: ACC_PUBLIC
    Code:
      stack=0, locals=1, args_size=1
         0: return
      LineNumberTable:
```

```
        line 7: 0

  public void classPolicy();
    flags: ACC_PUBLIC
    Code:
      stack=0, locals=1, args_size=1
         0: return
      LineNumberTable:
        line 11: 0
    RuntimeInvisibleAnnotations:
      0: #11()

  public void runtimePolicy();
    flags: ACC_PUBLIC
    Code:
      stack=0, locals=1, args_size=1
         0: return
      LineNumberTable:
        line 15: 0
    RuntimeVisibleAnnotations:
      0: #14()
}
```

从 RetentionTest 的字节码内容我们可以得出以下两点结论：

- 编译器并没有记录下 sourcePolicy() 方法的注解信息；
- 编译器分别使用了 `RuntimeInvisibleAnnotations` 和 `RuntimeVisibleAnnotations` 属性去记录了`classPolicy()` 方法 和 `runtimePolicy()`方法 的注解信息；

元注解 - @Documented

> Documented注解的作用是：描述在使用 javadoc 工具为类生成帮助文档时是否要保留其注解信息。

以下代码在使用Javadoc工具可以生成`@TestDocAnnotation`注解信息。

```
import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

@Documented
@Target({ElementType.TYPE,ElementType.METHOD})
public @interface TestDocAnnotation {

    public String value() default "default";
}
```

```
@TestDocAnnotation("myMethodDoc")
public void testDoc() {

}
```

元注解 - @Inherited

> Inherited注解的作用：被它修饰的Annotation将具有继承性。如果某个类使用了被@Inherited修饰的Annotation，则其子类将自动具有该注解。

我们来测试下这个注解：

- 定义@Inherited注解：

```
@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE,ElementType.METHOD})
public @interface TestInheritedAnnotation {
    String [] values();
    int number();
}
```

- 使用这个注解

```
@TestInheritedAnnotation(values = {"value"}, number = 10)
public class Person {
}

class Student extends Person{
    @Test
    public void test(){
        Class clazz = Student.class;
        Annotation[] annotations = clazz.getAnnotations();
        for (Annotation annotation : annotations) {
            System.out.println(annotation.toString());
        }
    }
}
```

- 输出

```
xxxxxxx.TestInheritedAnnotation(values=[value], number=10)
```

即使Student类没有显示地被注解@TestInheritedAnnotation，但是它的父类Person被注解，而且@TestInheritedAnnotation被@Inherited注解，因此Student类自动有了该注解。

元注解 - @Repeatable (Java8)

@Repeatable请参考Java 8 - 重复注解

元注解 - @Native (Java8)

使用 @Native 注解修饰成员变量，则表示这个变量可以被本地代码引用，常常被代码生成工具使用。对于 @Native 注解不常使用，了解即可

注解与反射接口

> 定义注解后，如何获取注解中的内容呢？反射包java.lang.reflect下的AnnotatedElement接口提供这些方法。这里注意：只有注解被定义为RUNTIME后，该注解才能是运行时可见，当class文件被装载时被保存在class文件中的Annotation才会被虚拟机读取。

AnnotatedElement 接口是所有程序元素（Class、Method和Constructor）的父接口，所以程序通过反射获取了某个类的AnnotatedElement对象之后，程序就可以调用该对象的方法来访问Annotation信息。我们看下具体的先关接口

- `boolean isAnnotationPresent(Class<?extends Annotation> annotationClass)`

判断该程序元素上是否包含指定类型的注解，存在则返回true，否则返回false。注意：此方法会忽略注解对应的注解容器。

- `<T extends Annotation> T getAnnotation(Class<T> annotationClass)`

返回该程序元素上存在的、指定类型的注解，如果该类型注解不存在，则返回null。

- `Annotation[] getAnnotations()`

返回该程序元素上存在的所有注解，若没有注解，返回长度为0的数组。

- `<T extends Annotation> T[] getAnnotationsByType(Class<T> annotationClass)`

返回该程序元素上存在的、指定类型的注解数组。没有注解对应类型的注解时，返回长度为0的数组。该方法的调用者可以随意修改返回的数组，而不会对其他调用者返回的数组产生任何影响。`getAnnotationsByType`方法与`getAnnotation`的区别在于，`getAnnotationsByType`会检测注解对应的重复注解容器。若程序元素为类，当前类上找不到注解，且该注解为可继承的，则会去父类上检测对应的注解。

- `<T extends Annotation> T getDeclaredAnnotation(Class<T> annotationClass)`

返回直接存在于此元素上的所有注解。与此接口中的其他方法不同，该方法将忽略继承的注释。如果没有注释直接存在于此元素上，则返回null

- `<T extends Annotation> T[] getDeclaredAnnotationsByType(Class<T> annotationClass)`

返回直接存在于此元素上的所有注解。与此接口中的其他方法不同，该方法将忽略继承的注释

- `Annotation[] getDeclaredAnnotations()`

返回直接存在于此元素上的所有注解及注解对应的重复注解容器。与此接口中的其他方法不同，该方法将忽略继承的注解。如果没有注释直接存在于此元素上，则返回长度为零的一个数组。该方法的调用者可以随意修改返回的数组，而不会对其他调用者返回的数组产生任何影响。

自定义注解

> 当我们理解了内置注解, 元注解和获取注解的反射接口后，我们便可以开始自定义注解了。这个例子我把上述的知识点全部融入进来, 代码很简单：

- 定义自己的注解

```
package com.pdai.java.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyMethodAnnotation {

    public String title() default "";

    public String description() default "";

}
```

- 使用注解

```
package com.pdai.java.annotation;

import java.io.FileNotFoundException;
import java.lang.annotation.Annotation;
import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.List;

public class TestMethodAnnotation {

    @Override
    @MyMethodAnnotation(title = "toStringMethod", description = "override toString method")
    public String toString() {
        return "Override toString method";
    }

    @Deprecated
    @MyMethodAnnotation(title = "old static method", description = "deprecated old static
method")
    public static void oldMethod() {
        System.out.println("old method, don't use it.");
    }

    @SuppressWarnings({"unchecked", "deprecation"})
    @MyMethodAnnotation(title = "test method", description = "suppress warning static method")
    public static void genericsTest() throws FileNotFoundException {
        List l = new ArrayList();
        l.add("abc");
        oldMethod();
    }
}
```

- 用反射接口获取注解信息

在TestMethodAnnotation中添加Main方法进行测试：

```
public static void main(String[] args) {
    try {
        // 获取所有methods
        Method[] methods = TestMethodAnnotation.class.getClassLoader()
                .loadClass(("com.pdai.java.annotation.TestMethodAnnotation"))
                .getMethods();

        // 遍历
        for (Method method : methods) {
            // 方法上是否有MyMethodAnnotation注解
            if (method.isAnnotationPresent(MyMethodAnnotation.class)) {
                try {
                    // 获取并遍历方法上的所有注解
                    for (Annotation anno : method.getDeclaredAnnotations()) {
                        System.out.println("Annotation in Method '"
                                + method + "' : " + anno);
                    }

                    // 获取MyMethodAnnotation对象信息
                    MyMethodAnnotation methodAnno = method
                            .getAnnotation(MyMethodAnnotation.class);
```

```
                    System.out.println(methodAnno.title());

                } catch (Throwable ex) {
                    ex.printStackTrace();
                }
            }
        }
    } catch (SecurityException | ClassNotFoundException e) {
        e.printStackTrace();
    }
}
```

- 测试的输出

```
Annotation in Method 'public static void
com.pdai.java.annotation.TestMethodAnnotation.oldMethod()' : @java.lang.Deprecated()
Annotation in Method 'public static void
com.pdai.java.annotation.TestMethodAnnotation.oldMethod()' :
@com.pdai.java.annotation.MyMethodAnnotation(title=old static method, description=deprecated old
static method)
old static method
Annotation in Method 'public static void
com.pdai.java.annotation.TestMethodAnnotation.genericsTest() throws
java.io.FileNotFoundException' : @com.pdai.java.annotation.MyMethodAnnotation(title=test method,
description=suppress warning static method)
test method
Annotation in Method 'public java.lang.String
com.pdai.java.annotation.TestMethodAnnotation.toString()' :
@com.pdai.java.annotation.MyMethodAnnotation(title=toStringMethod, description=override toString
method)
toStringMethod
```

深入理解注解

TIP

接下来，我们从其它角度深入理解注解

*Java8*提供了哪些新的注解?

- `@Repeatable`

请参考Java 8 - 重复注解

- `ElementType.TYPE_USE`

请参考Java 8 - 类型注解

- `ElementType.TYPE_PARAMETER`

`ElementType.TYPE_USE`(此类型包括类型声明和类型参数声明，是为了方便设计者进行类型检查)包含了`ElementType.TYPE`(类、接口（包括注解类型）和枚举的声明)和`ElementType.TYPE_PARAMETER`(类型参数声明), 不妨再看个例子

```java
// 自定义ElementType.TYPE_PARAMETER注解
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE_PARAMETER)
public @interface MyNotEmpty {
}

// 自定义ElementType.TYPE_USE注解
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE_USE)
public @interface MyNotNull {
}

// 测试类
public class TypeParameterAndTypeUseAnnotation<@MyNotEmpty T>{

  //使用TYPE_PARAMETER类型，会编译不通过
//      public @MyNotEmpty T test(@MyNotEmpty T a){
//          new ArrayList<@MyNotEmpty String>();
//              return a;
//      }

  //使用TYPE_USE类型，编译通过
  public @MyNotNull T test2(@MyNotNull T a){
    new ArrayList<@MyNotNull String>();
    return a;
  }
}
```

注解支持继承吗?

> 注解是不支持继承的

不能使用关键字extends来继承某个@interface，但注解在编译后，编译器会自动继承java.lang.annotation.Annotation接口.

虽然反编译后发现注解继承了Annotation接口，请记住，即使Java的接口可以实现多继承，但定义注解时依然无法使用extends关键字继承@interface。

区别于注解的继承，被注解的子类继承父类注解可以用@Inherited： 如果某个类使用了被@Inherited修饰的Annotation，则其子类将自动具有该注解。

注解实现的原理?

推荐两篇文章:

- https://blog.csdn.net/qq_20009015/article/details/106038023
- https://www.race604.com/annotation-processing/

注解的应用场景

TIP

最后我们再看看实际开发中注解的一些应用场景。@pdai

配置化到注解化 - 框架的演进

Spring 框架 配置化到注解化的转变。

继承实现到注解实现 - *Junit3*到*Junit4*

> 一个模块的封装大多数人都是通过继承和组合等模式来实现的，但是如果结合注解将可以极大程度提高实现的优雅度（降低耦合度）。而Junit3 到Junit4的演化就是最好的一个例子。

- 被测试类

```
public class HelloWorld {

    public void sayHello(){
        System.out.println("hello....");
        throw new NumberFormatException();
    }

    public void sayWorld(){
        System.out.println("world....");
    }

    public String say(){
        return "hello world!";
    }

}
```

- Junit 3 实现UT

通过继承 TestCase来实现，初始化是通过Override父类方法来进行，测试方式通过test的前缀方法获取。

```
public class HelloWorldTest extends TestCase{
    private HelloWorld hw;

    @Override
    protected void setUp() throws Exception {
        super.setUp();
        hw=new HelloWorld();
    }

    //1.测试没有返回值
    public void testHello(){
        try {
            hw.sayHello();
        } catch (Exception e) {
            System.out.println("发生异常.....");
        }

    }
    public void testWorld(){
```

```
        hw.sayWorld();
    }
    //2.测试有返回值的方法
    // 返回字符串
    public void testSay(){
        assertEquals("测试失败", hw.say(), "hello world!");
    }
    //返回对象
    public void testObj(){
        assertNull("测试对象不为空", null);
        assertNotNull("测试对象为空",new String());
    }
    @Override
    protected void tearDown() throws Exception {
        super.tearDown();
        hw=null;
    }
}
```

- Junit 4 实现UT

通过定义@Before，@Test，@After等等注解来实现。

```
public class HelloWorldTest {
    private HelloWorld hw;

    @Before
    public void setUp() {
        hw = new HelloWorld();
    }

    @Test(expected=NumberFormatException.class)
    // 1.测试没有返回值,有别于junit3的使用，更加方便
    public void testHello() {
        hw.sayHello();
    }
    @Test
    public void testWorld() {
        hw.sayWorld();
    }

    @Test
    // 2.测试有返回值的方法
    // 返回字符串
    public void testSay() {
        assertEquals("测试失败", hw.say(), "hello world!");
    }

    @Test
    // 返回对象
    public void testObj() {
        assertNull("测试对象不为空", null);
        assertNotNull("测试对象为空", new String());
    }

    @After
    public void tearDown() throws Exception {
        hw = null;
    }
```

```
}
```

这里我们发现通过注解的方式，我们实现单元测试时将更为优雅。了解Junit4是如何实现运行的呢？可以看这篇文章：JUnit4源码分析运行原理 (opens new window)。

自定义注解和*AOP* - 通过切面实现解耦

> 最为常见的就是使用Spring AOP切面实现**统一的操作日志管理**，我这里找了一个开源项目中的例子（只展示主要代码），给你展示下如何通过注解实现解耦的。

- 自定义Log注解

```
@Target({ ElementType.PARAMETER, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Log {
    /**
     * 模块
     */
    public String title() default "";

    /**
     * 功能
     */
    public BusinessType businessType() default BusinessType.OTHER;

    /**
     * 操作人类别
     */
    public OperatorType operatorType() default OperatorType.MANAGE;

    /**
     * 是否保存请求的参数
     */
    public boolean isSaveRequestData() default true;
}
```

- 实现日志的切面, 对自定义注解Log作切点进行拦截

即对注解了@Log的方法进行切点拦截，

```
@Aspect
@Component
public class LogAspect {
    private static final Logger log = LoggerFactory.getLogger(LogAspect.class);

    /**
     * 配置织入点 - 自定义注解的包路径
     *
     */
    @Pointcut("@annotation(com.xxx.aspectj.lang.annotation.Log)")
    public void logPointCut() {
    }

    /**
     * 处理完请求后执行
```

```
     *
     * @param joinPoint 切点
     */
    @AfterReturning(pointcut = "logPointCut()", returning = "jsonResult")
    public void doAfterReturning(JoinPoint joinPoint, Object jsonResult) {
        handleLog(joinPoint, null, jsonResult);
    }

    /**
     * 拦截异常操作
     *
     * @param joinPoint 切点
     * @param e 异常
     */
    @AfterThrowing(value = "logPointCut()", throwing = "e")
    public void doAfterThrowing(JoinPoint joinPoint, Exception e) {
        handleLog(joinPoint, e, null);
    }

    protected void handleLog(final JoinPoint joinPoint, final Exception e, Object jsonResult) {
        try {
            // 获得注解
            Log controllerLog = getAnnotationLog(joinPoint);
            if (controllerLog == null) {
                return;
            }

            // 获取当前的用户
            User currentUser = ShiroUtils.getSysUser();

            // *========数据库日志=========*//
            OperLog operLog = new OperLog();
            operLog.setStatus(BusinessStatus.SUCCESS.ordinal());
            // 请求的地址
            String ip = ShiroUtils.getIp();
            operLog.setOperIp(ip);
            // 返回参数
            operLog.setJsonResult(JSONObject.toJSONString(jsonResult));

            operLog.setOperUrl(ServletUtils.getRequest().getRequestURI());
            if (currentUser != null) {
                operLog.setOperName(currentUser.getLoginName());
                if (StringUtils.isNotNull(currentUser.getDept())
                        && StringUtils.isNotEmpty(currentUser.getDept().getDeptName())) {
                    operLog.setDeptName(currentUser.getDept().getDeptName());
                }
            }

            if (e != null) {
                operLog.setStatus(BusinessStatus.FAIL.ordinal());
                operLog.setErrorMsg(StringUtils.substring(e.getMessage(), 0, 2000));
            }
            // 设置方法名称
            String className = joinPoint.getTarget().getClass().getName();
            String methodName = joinPoint.getSignature().getName();
            operLog.setMethod(className + "." + methodName + "()");
            // 设置请求方式
            operLog.setRequestMethod(ServletUtils.getRequest().getMethod());
            // 处理设置注解上的参数
            getControllerMethodDescription(controllerLog, operLog);
            // 保存数据库
```

```
            AsyncManager.me().execute(AsyncFactory.recordOper(operLog));
        } catch (Exception exp) {
            // 记录本地异常日志
            log.error("==前置通知异常==");
            log.error("异常信息:{}", exp.getMessage());
            exp.printStackTrace();
        }
    }

    /**
     * 获取注解中对方法的描述信息 用于Controller层注解
     *
     * @param log 日志
     * @param operLog 操作日志
     * @throws Exception
     */
    public void getControllerMethodDescription(Log log, OperLog operLog) throws Exception {
        // 设置action动作
        operLog.setBusinessType(log.businessType().ordinal());
        // 设置标题
        operLog.setTitle(log.title());
        // 设置操作人类别
        operLog.setOperatorType(log.operatorType().ordinal());
        // 是否需要保存request，参数和值
        if (log.isSaveRequestData()) {
            // 获取参数的信息，传入到数据库中。
            setRequestValue(operLog);
        }
    }

    /**
     * 获取请求的参数，放到log中
     *
     * @param operLog
     * @param request
     */
    private void setRequestValue(OperLog operLog) {
        Map<String, String[]> map = ServletUtils.getRequest().getParameterMap();
        String params = JSONObject.toJSONString(map);
        operLog.setOperParam(StringUtils.substring(params, 0, 2000));
    }

    /**
     * 是否存在注解，如果存在就获取
     */
    private Log getAnnotationLog(JoinPoint joinPoint) throws Exception {
        Signature signature = joinPoint.getSignature();
        MethodSignature methodSignature = (MethodSignature) signature;
        Method method = methodSignature.getMethod();

        if (method != null)
        {
            return method.getAnnotation(Log.class);
        }
        return null;
    }
}
```

- 使用@Log注解

以一个简单的CRUD操作为例, 这里展示部分代码：每对“部门”进行操作就会产生一条操作日志存入数据库。

```
@Controller
@RequestMapping("/system/dept")
public class DeptController extends BaseController {
    private String prefix = "system/dept";

    @Autowired
    private IDeptService deptService;

    /**
     * 新增保存部门
     */
    @Log(title = "部门管理", businessType = BusinessType.INSERT)
    @RequiresPermissions("system:dept:add")
    @PostMapping("/add")
    @ResponseBody
    public AjaxResult addSave(@Validated Dept dept) {
        if (UserConstants.DEPT_NAME_NOT_UNIQUE.equals(deptService.checkDeptNameUnique(dept))) {
            return error("新增部门'" + dept.getDeptName() + "'失败，部门名称已存在");
        }
        return toAjax(deptService.insertDept(dept));
    }

    /**
     * 保存
     */
    @Log(title = "部门管理", businessType = BusinessType.UPDATE)
    @RequiresPermissions("system:dept:edit")
    @PostMapping("/edit")
    @ResponseBody
    public AjaxResult editSave(@Validated Dept dept) {
        if (UserConstants.DEPT_NAME_NOT_UNIQUE.equals(deptService.checkDeptNameUnique(dept))) {
            return error("修改部门'" + dept.getDeptName() + "'失败，部门名称已存在");
        } else if(dept.getParentId().equals(dept.getDeptId())) {
            return error("修改部门'" + dept.getDeptName() + "'失败，上级部门不能是自己");
        }
        return toAjax(deptService.updateDept(dept));
    }

    /**
     * 删除
     */
    @Log(title = "部门管理", businessType = BusinessType.DELETE)
    @RequiresPermissions("system:dept:remove")
    @GetMapping("/remove/{deptId}")
    @ResponseBody
    public AjaxResult remove(@PathVariable("deptId") Long deptId) {
        if (deptService.selectDeptCount(deptId) > 0) {
            return AjaxResult.warn("存在下级部门,不允许删除");
        }
        if (deptService.checkDeptExistUser(deptId)) {
            return AjaxResult.warn("部门存在用户,不允许删除");
        }
        return toAjax(deptService.deleteDeptById(deptId));
    }

  // ...
}
```

> 权限管理也是通过类似的注解（`@RequiresPermissions`）机制来实现的。所以我们可以看到，通过注解+AOP最终的目标是为了实现模块的解耦。