Java之注解的定义及使用

> Java的注解在实际项目中使用得非常的多，特别是在使用了 Spring之后。
> 本文会介绍 Java注解的语法，以及在 Spring中使用注解的例子。

注解的语法

注解的例子

以Junit中的@Test注解为例

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {
    long timeout() default 0L;
}
```

可以看到@Test注解上有@Target()和@Retention()两个注解。
这种注解了注解的注解，称之为**元注解**。
跟声明了数据的数据，称为元数据是一种意思。

之后的注解的格式是

```
修饰符 @interface 注解名 {
    注解元素的声明1
    注解元素的声明2
}
```

注解的元素声明有两种形式

```
type elementName();
type elementName() default value;  // 带默认值
```

常见的元注解

@Target**注解**

@Target注解用于限制注解能在哪些项上应用，没有加@Target的注解可以应用于任何项上。

在java.lang.annotation.ElementType类中可以看到所有@Target接受的项

- TYPE 在【类、接口、注解】上使用
- FIELD 在【字段、枚举常量】上使用
- METHOD 在【方法】上使用
- PARAMETER 在【参数】上使用
- CONSTRUCTOR 在【构造器】上使用
- LOCAL_VARIABLE 在【局部变量】上使用
- ANNOTATION_TYPE 在【注解】上使用
- PACKAGE 在【包】上使用
- TYPE_PARAMETER 在【类型参数】上使用 Java 1.8 引入
- TYPE_USE 在【任何声明类型的地方】上使用 Java 1.8 引入

@Test注解只允许在方法上使用。

```
@Target(ElementType.METHOD)
public @interface Test { ... }
```

如果要支持多项，则传入多个值。

```
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface MyAnnotation { ... }
```

此外元注解也是注解，也符合注解的语法，如@Target注解。
@Target(ElementType.ANNOTATION_TYPE)表明@Target注解只能使用在注解上。

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Target {
    ElementType[] value();
}
```

@Retention注解

@Retention指定注解应该保留多长时间，默认是RetentionPolicy.CLASS。
在java.lang.annotation.RetentionPolicy可看到所有的项

- SOURCE 不包含在类文件中
- CLASS 包含在类文件中，不载入虚拟机
- RUNTIME 包含在类文件中，由虚拟机载入，可以用反射API获取

@Test注解会载入到虚拟机，可以通过代码获取

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Test { ... }
```

@Documented注解

主要用于归档工具识别。被注解的元素能被Javadoc或类似的工具文档化。

@Inherited注解

添加了@Inherited注解的注解，所注解的类的子类也将拥有这个注解

注解

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Inherited
public @interface MyAnnotation { ... }
```

父类

```
@MyAnnotation
class Parent { ... }
```

子类Child会把加在Parent上的@MyAnnotation继承下来

```
class Child extends Parent { ... }
```

@Repeatable注解

Java 1.8 引入的注解，标识注解是可重复使用的。

注解1

```
public @interface MyAnnotations {
    MyAnnotation[] value();
}
```

注解2

```
@Repeatable(MyAnnotations.class)
```

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

有使用@Repeatable()时的使用

```
@MyAnnotation(1)
@MyAnnotation(2)
@MyAnnotation(3)
public class MyTest { ... }
```

没使用@Repeatable()时的使用，@MyAnnotation去掉@Repeatable元注解

```
@MyAnnotations({
    @MyAnnotation(1),
    @MyAnnotation(2),
    @MyAnnotation(3)})
public class MyTest { ... }
```

这个注解还是非常有用的，让我们的代码变得简洁不少，
Spring的@ComponentScan注解也用到这个元注解。

元素的类型

支持的元素类型

- 8种基本数据类型（byte, short, char, int, long, float, double, boolean)
- String
- Class
- enum
- 注解类型
- 数组（所有上边类型的数组）

例子

枚举类

```
public enum Status {
    GOOD,
    BAD
}
```

注解1

```
@Target(ElementType.ANNOTATION_TYPE)
public @interface MyAnnotation1 {
    int val();
}
```

注解2

```
@Target(ElementType.TYPE)
public @interface MyAnnotation2 {

    boolean boo() default false;

    Class<?> cla() default Void.class;

    Status enu() default Status.GOOD;

    MyAnnotation1 anno() default @MyAnnotation1(val = 1);

    String[] arr();

}
```

使用时，无默认值的元素必须传值

```
@MyAnnotation2(
        cla = String.class,
        enu = Status.BAD,
        anno = @MyAnnotation1(val = 2),
        arr = {"a", "b"})
public class MyTest { ... }
```

Java内置的注解

@Override注解

告诉编译器这个是个覆盖父类的方法。如果父类删除了该方法，则子类会报错。

@Deprecated注解

表示被注解的元素已被弃用。

@SuppressWarnings注解

告诉编译器忽略警告。

@FunctionalInterface注解

Java 1.8 引入的注解。该注释会强制编译器javac检查一个接口是否符合函数接口的标准。

特别的注解

有两种比较特别的注解

- 标记注解： 注解中没有任何元素，使用时直接是 @XxxAnnotation, 不需要加括号
- 单值注解： 注解只有一个元素，且名字为value，使用时直接传值，不需要指定元素名@XxxAnnotation(100)

利用反射获取注解

Java的AnnotatedElement接口中有getAnnotation()等获取注解的方法。
而Method, Field, Class, Package等类均实现了这个接口，因此均有获取注解的能力。

例子

注解

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.METHOD})
public @interface MyAnno {
    String value();
}
```

被注解的元素

```
@MyAnno("class")
public class MyClass {

    @MyAnno("feild")
    private String str;

    @MyAnno("method")
    public void method() { }

}
```

获取注解

```
public class Test {

    public static void main(String[] args) throws Exception {

        MyClass obj = new MyClass();
        Class<?> clazz = obj.getClass();
```

```
        // 获取对象上的注解
        MyAnno anno = clazz.getAnnotation(MyAnno.class);
        System.out.println(anno.value());

        // 获取属性上的注解
        Field field = clazz.getDeclaredField("str");
        anno = field.getAnnotation(MyAnno.class);
        System.out.println(anno.value());

        // 获取方法上的注解
        Method method = clazz.getMethod("method");
        anno = method.getAnnotation(MyAnno.class);
        System.out.println(anno.value());
    }

}
```

在Spring中使用自定义注解

> 注解本身不会有任何的作用，需要有其他代码或工具的支持才有用。

需求

设想现有这样的需求，程序需要接收不同的命令CMD，
然后根据命令调用不同的处理类Handler。
很容易就会想到用Map来存储命令和处理类的映射关系。

由于项目可能是多个成员共同开发，不同成员实现各自负责的命令的处理逻辑。
因此希望开发成员只关注Handler的实现，不需要主动去Map中注册CMD和Handler的映射。

最终效果

最终希望看到效果是这样的

```
@CmdMapping(Cmd.LOGIN)
public class LoginHandler implements ICmdHandler {
    @Override
    public void handle() {
        System.out.println("handle login request");
    }
}

@CmdMapping(Cmd.LOGOUT)
public class LogoutHandler implements ICmdHandler {
    @Override
    public void handle() {
        System.out.println("handle logout request");
    }
}
```

开发人员增加自己的Handler，只需要创建新的类并注上@CmdMapping(Cmd.Xxx)即可。

具体做法

具体的实现是使用Spring和一个自定义的注解
定义@CmdMapping注解

```
@Documented
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Component
public @interface CmdMapping {
    int value();
}
```

@CmdMapping中有一个int类型的元素value，用于指定CMD。这里做成一个单值注解。
这里还加了Spring的@Component注解，因此注解了@CmdMapping的类也会被Spring创建实例。

然后是CMD接口，存储命令。

```
public interface Cmd {
    int REGISTER = 1;
    int LOGIN    = 2;
    int LOGOUT   = 3;
}
```

之后是处理类接口，现实情况接口会复杂得多，这里简化了。

```
public interface ICmdHandler {
    void handle();
}
```

上边说过，注解本身是不起作用的，需要其他的支持。下边就是让注解生效的部分了。
使用时调用handle()方法即可。

```
@Component
public class HandlerDispatcherServlet implements
    InitializingBean, ApplicationContextAware {

    private ApplicationContext context;

    private Map<Integer, ICmdHandler> handlers = new HashMap<>();

    public void handle(int cmd) {
        handlers.get(cmd).handle();
    }

    public void afterPropertiesSet() {

        String[] beanNames = this.context.getBeanNamesForType(Object.class);

        for (String beanName : beanNames) {

            if (ScopedProxyUtils.isScopedTarget(beanName)) {
                continue;
            }

            Class<?> beanType = this.context.getType(beanName);

            if (beanType != null) {

                CmdMapping annotation = AnnotatedElementUtils.findMergedAnnotation(
                        beanType, CmdMapping.class);

                if(annotation != null) {
                    handlers.put(annotation.value(), (ICmdHandler) context.getBean(beanType));
                }
            }
        }

    }

    public void setApplicationContext(ApplicationContext applicationContext)
            throws BeansException {
        this.context = applicationContext;
    }

}
```

主要工作都是Spring做，这里只是将实例化后的对象put到Map中。

测试代码

```
@ComponentScan("pers.custom.annotation")
public class Main {

    public static void main(String[] args) {

        AnnotationConfigApplicationContext context
            = new AnnotationConfigApplicationContext(Main.class);

```

```
        HandlerDispatcherServlet servlet = context.getBean(HandlerDispatcherServlet.class);10
        servlet.handle(Cmd.REGISTER);
        servlet.handle(Cmd.LOGIN);
        servlet.handle(Cmd.LOGOUT);

        context.close();
    }
}
```

> 完整项目

总结

可以看到使用注解能够写出很灵活的代码，注解也特别适合做为使用框架的一种方式。
所以学会使用注解还是很有用的，毕竟这对于上手框架或实现自己的框架都是非常重要的知识。