导入依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

类打上注解@Aspect表示类是一个切面类，并交给spring管理

```
@Aspect
@Component
@Slf4j
public class LangAspect {}
```

AOP五种通知类型			
序号	注解类型	名称	应用
1	@Before(value = “”)	前置通知：在方法执行前通知	前置通知，可以用来限制请求次数，控制并发请求数量
2	@Around(value = “”)	环绕通知：在目标方法执行之前和之后都可以执行额外代码的通知，最强大的通知，可决定目标方法是否继续执行。	可以用做日志，根据切点，不需要在每个方法上打注解,也可以用来修改请求参数
3	@AfterReturning(value = “”	后置通知：在方法正常执行完成进行通知，可以访问到方法的返回值的。	可以用来获取返回值并修改
4	@AfterThrowing(value = “”	异常通知：在方法出现异常时进行通知,可以访问到异常对象，且可以指定在出现特定异常时在执行通知	可以获取指定的异常类型并处理
5	@After(value = “”)	后置通知： 在目标方法执行后无论是否发生异常，执行通知,不能访问目标方法的执行的结果。	方法执行完要做的事情，还没到后置通知

以上5种都可以额外接收一个JoinPoint参数，来获取目标对象和目标方法相关信息，但一定要保证必须是第一个参数。
比如：

```
@Pointcut("execution(public * com.tecloman.web.modules.*.controller.*.*(..))")
    public void langPointCut() {}

@AfterReturning(value = "langPointCut()", returning = "r")
public R afterReturning(JoinPoint point, R r) throws Throwable { }
```

langPointCut() 切入点，方法中point就是切入对象，r是返回实例，可以获取返回值数据

Before **前置通知** 和 After**后置通知**：效果类似于try--catch--finally里的finally块，无论如何都会执行

环绕通知： proceed()方法-->ProceedingJoinPoin 参数调用此方法后，目标方法才会执行，否则目标方法永远不执行(控制目标方法的核心方法)、（返回值为目标方法的返回值）

```
@Around("logPointCut()")
public Object around(ProceedingJoinPoint point) throws Throwable {
   Object result = point.proceed();
   // 不返回result 方法就不会继续执行
   return result;
}
```

异常通知：throwing = "ex" ， 告诉Spring这个ex就是用来接受异常的，

唯一的要求就是参数列表一定不能乱写

- 通知方法是Spring利用反射调的，每次方法调用得确定这个方法的参数表的值；
- 参数表上的每一个参数，Spring必须都得知道是谁

```
@AfterThrowing(value = "langPointCut()", throwing = "ex")
public void afterThrowing(JoinPoint point, Exception ex) {
    MethodSignature signature = (MethodSignature) point.getSignature();
    Method method = signature.getMethod();
    // 实验证明，aop不能解决异常
    String className = point.getSignature().getDeclaringTypeName() + ".";
    log.error("报错的方法名：" + className + method.getName());
    log.error("报错信息：" + ex.getMessage());
}
```

通知方法执行顺序：

try{
@Before
method.invoke(obj,args);
@AfterReturning
}catch(e){
@AfterThrowing
}finally{
@After
}
通知方法的执行顺序:
正常执行：@Before（前置通知）=》@After（后置通知）=》@AfterReturning（正常返回）
异常执行：@Before（前置通知）=》@After（后置通知）=》@AfterThrowing（异常返回）

AOP的原理：Spring会创建目标对象的代理，根据切入点规则匹配对应的连接点，把连接点变为切入点，不会直接执行目标方法

，会被切面类中的通知进行增强。
AOP如何生成代理对象：如果目标对象实现了接口，那么使用java api Proxy类，如果目标对象没有实现接口，底层使用CGLIB
如果想强制使用CGLIB需要添加
@EnableAspectJAutoProxy(proxyTargetClass = true)。
动态代理简单来说就是在程序执行过程中，创建代理对象，通过代理对象执行方法，给目标类的方法增加额外的功能，也叫做功能增强

@Pointcut切入点表达式

@Pointcut切入点表达式	
1	execution：一般用于指定方法的执行，用的最多。
2	within：指定某些类型的全部方法执行，也可用来指定一个包
3	Spring Aop是基于动态代理的，生成的bean也是一个代理对象，this就是这个代理对象，当这个对象可以转换为指定的类型时，对应的切入点就是它了，Spring Aop将生效。
4	target：当被代理的对象可以转换为指定的类型时，对应的切入点就是它了，Spring Aop将生效
5	args：当执行的方法的参数是指定类型时生效
6	@target：当代理的目标对象上拥有指定的注解时生效
7	@args：当执行的方法参数类型上拥有指定的注解时生效。
8	@within：与@target类似，看官方文档和网上的说法都是@within只需要目标对象的类或者父类上有指定注解，则@within会生效，而@target则是必须是目标对象的类上有指定的注解。而根据笔者的测试这两者是只要目标类或父类上有指定的注解即可。
9	@annotation：当执行的方法上拥有指定的注解时生效(通常用于接口日志)
10	reference pointcut：(经常使用)表示引用其他命名切入点，只有@ApectJ风格支持，Schema风格不

11 **bean：当调用的方法是指定的bean的方法时生效。(Spring AOP自己扩展支持的)**

Pointcut定义时，还可以使用&&、||、！这三个运算。进行逻辑运算。可以把各种条件组合起来使用

```
// list方法切入点
@Pointcut("execution(public * com.tecloman.web.modules.*.controller.*.list(..))")
public void listPointCut() {}
// info方法切入点
@Pointcut("execution(public * com.tecloman.web.modules.*.controller.*.info(..))")
public void infoPointCut() {}
// all方法切入点
@Pointcut("execution(public * com.tecloman.web.modules.*.controller.*.all(..))")
public void allPointCut() {}
```

以上三个切点，满足一个则进入AOP通知

```
@Pointcut("listPointCut() || infoPointCut() || allPointCut()")
private void langPointCut(){}
```

execution(public *）表示所有public修饰的方法

within是用来指定类型的，指定类型中的所有方法将被拦截

```
    // 此处只能写实现类，接口拦截不了
    @Pointcut("within(com.tecloman.web.modules.service.impl.HssTypeServiceImpl)")
    public void pointCut() {
    }
```

Spring Aop是基于代理的，this就表示代理对象。this类型的Pointcut表达式的语法是this(type)，当生成的代理对象可以转换为type指定的类型时则表示匹配。基于JDK接口的代理和基于CGLIB的代理生成的代理对象是不一样的。（注意和上面within的区别）

```
   // 这样子，就可以拦截到AService所有的子类的所有外部调用方法
    @Pointcut("this(com.tecloman.web.modules.service.HssTypeService*)")
    public void pointCut() {
    }
```

Spring Aop是基于代理的，target则表示被代理的目标对象。当被代理的目标对象可以被转换为指定的类型时则表示匹配。 注意：和上面不一样，这里是target，因此如果要切入，只能写实现类了

```
    @Pointcut("target(com.tecloman.web.modules.impl.HssTypeServiceImpl)")
    public void pointCut() {
    }
```

args用来匹配方法参数的。

- 1、“args()”匹配任何不带参数的方法。
- 2、“args(java.lang.String)”匹配任何只带一个参数，而且这个参数的类型是String的方法。
- 3、“args(…)”带任意参数的方法。
- 4、“args(java.lang.String,…)”匹配带任意个参数，但是第一个参数的类型是String的方法。
- 5、“args(…,java.lang.String)”匹配带任意个参数，但是最后一个参数的类型是String的方法。

```
    @Pointcut("args()")
    public void pointCut() {
    }
```

@target匹配当被代理的目标对象对应的类型及其父类型上拥有指定的注解时。

```
  // 能够切入类上（非方法上）标准了Lang注解的所有外部调用方法
    @Pointcut("@target(com.tecloman.web.common.annotation.Lang)")
    public void pointCut() {
    }
```

@args匹配被调用的方法上含有参数，且对应的参数类型上拥有指定的注解的情况

```
// 匹配**方法参数类型上**拥有MyAnno注解的方法调用。
//如我们有一个方法add(MyParam param)接收一个MyParam类型的参数，而MyParam这个类是拥有注解Lang的，则它可以被Pointcut表达式匹配上
    @Pointcut("@args(com.tecloman.web.common.annotation.Lang)")
    public void pointCut() {
    }
```

@within用于匹配被代理的目标对象对应的类型或其父类型拥有指定的注解的情况，但只有在调用拥有指定注解的类上的方法时才匹配。
@within(com.tecloman.web.common.annotation.Lang)”
匹配被调用的方法声明的类上拥有MyAnno注解的情况。比如有一个ClassA上使用了注解MyAnno标注，并且定义了一个方法a()，那么在调用ClassA.a()方法时将匹配该Pointcut；如果有一个ClassB上没有MyAnno注解，但是它继承自ClassA，同时它上面定义了一个方法b()，那么在调用ClassB().b()方法时不会匹配该Pointcut，但是在调用ClassB().a()时将匹配该方法调用，因为a()是定义在父类型ClassA上的，且ClassA上使用了MyAnno注解。但是如果子类ClassB覆写了父类ClassA的a()方法，则调用ClassB.a()方法时也不匹配该Pointcut。

@annotation用于匹配**方法上**拥有指定注解的情况。

```
  // 可以匹配所有方法上标有此注解的方法
    @Pointcut("@annotation(com.tecloman.web.common.annotation.Lang)")
    public void pointCut() {
    }
```

bean用于匹配当调用的是指定的Spring的某个bean的方法时。

1、“bean(hssTypeService)”匹配Spring Bean容器中id或name为abc的bean的方法调用。
2、“bean(user*)”匹配所有id或name为以user开头的bean的方法调用。

```
// 这个就能切入到AServiceImpl类的素有的外部调用的方法里
    @Pointcut("bean(hssTypeService)")
    public void pointCut() {
    }
```