6.5.1 Spring AOP 支持的 AspectJ 切入点指示符

切入点指示符用来指示切入点表达式目的，，在 Spring AOP 中目前只有执行方法这一个连接点，Spring AOP 支持的 AspectJ 切入点指示符如下：

execution	用于匹配方法执行的连接点；
within	用于匹配指定类型内的方法执行；
this	用于匹配当前 AOP 代理对象类型的执行方法；注意是 AOP 代理对象的类型匹配，这样就可能包括引入接口也类型匹配；
target	用于匹配当前目标对象类型的执行方法；注意是目标对象的类型匹配，这样就不包括引入接口也类型匹配；
args	用于匹配当前执行的方法传入的参数为指定类型的执行方法；
@within	用于匹配所以持有指定注解类型内的方法；
@target	用于匹配当前目标对象类型的执行方法，其中目标对象持有指定的注解；
@args	用于匹配当前执行的方法传入的参数持有指定注解的执行；
@annotation	用于匹配当前执行方法持有指定注解的方法；
bean	Spring AOP 扩展的，AspectJ 没有对于指示符，用于匹配特定名称的 Bean 对象的执行方法；
reference pointcut	表示引用其他命名切入点，只有 @ApectJ 风格支持，Schema 风格不支持。

AspectJ 切入点支持的切入点指示符还有：call、get、set、preinitialization、staticinitialization、initialization、handler、adviceexecution、withincode、cflow、cflowbelow、if、@this、@withincode；但 Spring AOP 目前不支持这些指示符，使用这些指示符将抛出 IllegalArgumentException 异常。这些指示符 Spring AOP 可能会在以后进行扩展。

6.5.1 命名及匿名切入点

命名切入点可以被其他切入点引用，而匿名切入点是不可以的。

只有 @AspectJ 支持命名切入点，而 Schema 风格不支持命名切入点。如下所示， @AspectJ 使用如下方式引用命名切入点：

```
@Pointcut(
    value="execution(* cn.javass..*.sayBefore(java.lang.String)) && args(param)",
    argNames = "param")
public void beforePointcut(String param) {}
                    引用命名切入点
@Before(value = "beforePointcut(param)", argNames = "param")
public void beforeAdvice(String param) {
        System.out.println("==========before advice param:" + param);
}
```

6.5.2 ；类型匹配语法

首先让我们来了解下 AspectJ 类型匹配的通配符：

* ：匹配任何数量字符；

.. ：匹配任何数量字符的重复，如在类型模式中匹配任何数量子包；而在方法参数模式中匹配任何数量参数。

+ ：匹配指定类型的子类型；仅能作为后缀放在类型模式后边。

1. java.lang.String 匹配 String 类型；
2. java.*.String 匹配 java 包下的任何 “一级子包 ”下的 String 类型；
3. 如匹配 java.lang.String ，但不匹配 java.lang.ss.String
4. java..* 匹配 java 包及任何子包下的任何类型 ；
5. 如匹配 java.lang.String 、java.lang.annotation.Annotation
6. java.lang.*ing 匹配任何 java.lang 包下的以 ing 结尾的类型；
7. java.lang.Number+ 匹配 java.lang 包下的任何 Number 的自类型；
8. 如匹配 java.lang.Integer ，也匹配 java.math.BigInteger

接下来再看一下具体的匹配表达式类型吧：

匹配类型： 使用如下方式匹配

1. 注解？ 类的全限定名字

- 注解： 可选，类型上持有的注解，如 @Deprecated ；
- 类的全限定名： 必填，可以是任何类全限定名。

匹配方法执行： 使用如下方式匹配：

1. 注解？ 修饰符 ？ 返回值类型 类型声明 ?方法名 (参数列表) 异常列表？

- 注解： 可选，方法上持有的注解，如 @Deprecated ；
- 修饰符： 可选，如 public 、 protected ；
- 返回值类型： 必填，可以是任何类型模式； “ * 表示所有类型；
- 类型声明： 可选，可以是任何类型模式；
- 方法名： 必填，可以使用 “ * 进行模式匹配；
- 参数列表： “ () 表示方法没有任何参数； “ (..) 表示匹配接受任意个参数的方法， “ (..,java.lang.String) ” 表示匹配接受 java.lang.String 类型的参数结束， 且其前边可以接受有任意个参数的方法； “ (java.lang.String,..) 表示匹配接受 java.lang.String 类型的参数开始， 且其后边可以接受任意个参数的方法； “ (*, java.lang.String) 表示匹配接受 java.lang.String 类型的参数结束，且其前边接受有一个任意类型参数的方法；
- 异常列表： 可选，以 “ throws 异常全限定名列表 ”声明，异常全限定名列表如有多个以 “， ”分割，如 throws java.lang.IllegalArgumentException, java.lang.ArrayIndexOutOfBoundsException 。

匹配 Bean 名称： 可以使用 Bean 的 id 或 name 进行匹配， 并且可使用通配符 “ * ”;

6.5.3 组合切入点表达式

AspectJ 使用 且（&& ）、或（ || ）、非（ ！）来组合切入点表达式。

在 Schema 风格下，由于在 XML 中使用 “ &&需要使用转义字符 “ & ” 来代替之，所以很不方便，因此 Spring ASP 提供了 and 、 or 、 not 来代替 && 、 || 、 ！。

6.5.3 切入点使用示例

一、 execution ：使用 “ execution(方法表达式) ”匹配方法执行；

模式	描述
public * *(..)	任何公共方法的执行
* cn.javass..IPointcutService.*()	cn.javass 包及所有子包下 IPointcutService 接口中的任何无参方法
* cn.javass..*.*(..)	cn.javass 包及所有子包下任何类的任何方法
* cn.javass..IPointcutService.*(*)	cn.javass 包及所有子包下 IPointcutService 接口的任何只有一个参数方法
* (!cn.javass..IPointcutService+).*(..)	非“ cn.javass 包及所有子包下 IPointcutService 接口及子类型 ”的任何方法
* cn.javass..IPointcutService+.*()	cn.javass 包及所有子包下 IPointcutService 接口及子类型的的任何无参方法
* cn.javass..IPointcut*.test*(java.util.Date)	cn.javass 包及所有子包下 IPointcut 前缀类型的的以 test 开头的只有一个参数类型为 java.util.Date 的方法，注意该匹配是根据方法签名的参数类型进行匹配的，而不是根据执行时传入的参数类型决定的 如定义方法： public void test(Object obj); 即使执行时传入 java.util.Date ，也不会匹配的；
* cn.javass..IPointcut*.test*(..) throws IllegalArgumentException, ArrayIndexOutOfBoundsException	cn.javass 包及所有子包下 IPointcut 前缀类型的的任何方法，且抛出 IllegalArgumentException 和 ArrayIndexOutOfBoundsException 异常
* (cn.javass..IPointcutService+ && java.io.Serializable+).*(..)	任何实现了 cn.javass 包及所有子包下 IPointcutService 接口和 java.io.Serializable 接口的类型的任何方法
@java.lang.Deprecated * *(..)	任何持有 @java.lang.Deprecated 注解的方法
@java.lang.Deprecated @cn.javass..Secure * *(..)	任何持有 @java.lang.Deprecated 和 @cn.javass..Secure 注解的方法
@(java.lang.Deprecated || cn.javass..Secure) * *(..)	任何持有 @java.lang.Deprecated 或 @cn.javass..Secure 注解的方法
(@cn.javass..Secure *) *(..)	任何返回值类型持有 @cn.javass..Secure 的方法
* (@cn.javass..Secure *).*(..)	任何定义方法的类型持有 @cn.javass..Secure 的方法
* *(@cn.javass..Secure (*) , @cn.javass..Secure (*))	任何签名带有两个参数的方法，且这个两个参数都被 @Secure 标记了， 如 public void test(@Secure String str1, @Secure String str1);
* *((@ cn.javass..Secure *)) 或 * *(@ cn.javass..Secure *)	任何带有一个参数的方法，且该参数类型持有 @cn.javass..Secure ； 如 public void test(Model model); 且 Model 类上持有 @Secure 注解

* *(@cn.javass..Secure (@cn.javass..Secure *) , @ cn.javass..Secure (@cn.javass..Secure *))	任何带有两个参数的方法，且这两个参数都被 @ cn.javass..Secure 标记了；且这两个参数的类型上都持有 @ cn.javass..Secure ；
* *(java.util.Map<cn.javass..Model, cn.javass..Model> , ..)	任何带有一个 java.util.Map 参数的方法，且该参数类型是以 < cn.javass..Model, cn.javass..Model > 为泛型参数；注意只匹配第一个参数为 java.util.Map，不包括子类型； 如 public void test(HashMap<Model, Model> map, String str); 将不匹配，必须使用 “ * *(java.util.HashMap<cn.javass..Model,cn.javass.. Model> , ..) 进行匹配； 而 public void test(Map map, int i); 也将不匹配，因为泛型参数不匹配
* *(java.util.Collection<@cn.javass..S ecure *>)	任何带有一个参数 （类型为 java.util.Collection ）的方法，且该参数类型是有一个泛型参数，该泛型参数类型上持有 @cn.javass..Secure 注解； 如 public void test(Collection<Model> collection);Model 类型上持有 @cn.javass..Secure
* *(java.util.Set<? extends HashMap>)	任何带有一个参数的方法，且传入的参数类型是有一个泛型参数，该泛型参数类型继承与 HashMap ； Spring AOP 目前测试不能正常工作
* *(java.util.List<? super HashMap>)	任何带有一个参数的方法，且传入的参数类型是有一个泛型参数，该泛型参数类型是 HashMap 的基类型；如 public voi test(Map map) ； Spring AOP 目前测试不能正常工作
* *(*<@cn.javass..Secure *>)	任何带有一个参数的方法，且该参数类型是有一个泛型参数，该泛型参数类型上持有 @cn.javass..Secure 注解； Spring AOP 目前测试不能正常工作

二、within ：使用“ within(类型表达式) ”匹配指定类型内的方法执行；

模式	描述
within(cn.javass..*)	cn.javass 包及子包下的任何方法执行
within(cn.javass..IPointcutService+)	cn.javass 包或所有子包下 IPointcutService 类型及子类型的任何方法
within(@cn.javass..Secure *)	持有 cn.javass..Secure 注解的任何类型的任何方法 必须是在目标对象上声明这个注解，在接口上声明的对它不起作用

三、 this ：使用“ this(类型全限定名) ”匹配当前 AOP 代理对象类型的执行方法；注意是 AOP 代理对象的类型匹配，这样就可能包括引入接口方法也可以匹配；注意 this 中使用的表达式必须是类型全限定名，不支持通配符；

模式	描述
this(cn.javass.spring.chapter6.service.IPointcutService)	当前 AOP 对象实现了 IPointcutService 接口的任何方法
this(cn.javass.spring.chapter6.service.IIntroductionService)	当前 AOP 对象实现了 IIntroductionService 接口的任何方法 也可能是引入接口

四、 target ：使用“ target(类型全限定名) ”匹配当前目标对象类型的执行方法；注意是目标对象的类型匹配，这样就不包括引入接口也类型匹配；注意 target 中使用的表达式必须是类型全限定名，不支持通配符；

模式	描述
target(cn.javass.spring.chapter6.service.IPointcutService)	当前目标对象 （非 AOP 对象）实现了 IPointcutService 接口的任何方法
target(cn.javass.spring.chapter6.service.IIntroductionService)	当前目标对象 （非 AOP 对象） 实现了 IIntroductionService 接口的任何方法 不可能是引入接口

五、 args ：使用“ args(参数类型列表) ”匹配当前执行的方法传入的参数为指定类型的执行方法；注意是匹配传入的参数类型，不是匹配方法签名的参数类型；参数类型列表中的参数必须是类型全限定名，通配符不支持； args 属于动态切入点，这种切入点开销非常大，非特殊情况最好不要使用；

模式	描述
args (java.io.Serializable,..)	任何一个以接受 “传入参数类型为 java.io.Serializable 开”头，且其后可跟任意个任意类型的参数的方法执行， args 指定的参数类型是在运行时动态匹配的

六、 @within ：使用“ @within(注解类型) ”匹配所以持有指定注解类型内的方法；注解类型也必须是全限定类型名；

模式	描述
@within cn.javass.spring.chapter6.Secure)	任何目标对象对应的类型持有 Secure 注解的类方法； 必须是在目标对象上声明这个注解， 在接口上声明的对它不起作用

七、 @target ：使用“ @target(注解类型) ”匹配当前目标对象类型的执行方法，其中目标对象持有指定的注解；注解类型也必须是全限定类型名；

模式	描述
@target (cn.javass.spring.chapter6.Secure)	任何目标对象持有 Secure 注解的类方法； 必须是在目标对象上声明这个注解，在接口上声明的对它不起作用

八、 @args ：使用“ @args(注解列表) ”匹配当前执行的方法传入的参数持有指定注解的执行；注解类型也必须是全限定类型名；

模式	描述
@args (cn.javass.spring.chapter6.Secure)	任何一个只接受一个参数的方法，且方法运行时传入的参数持有注解 cn.javass.spring.chapter6.Secure ；动态切入点，类似于 arg 指示符；

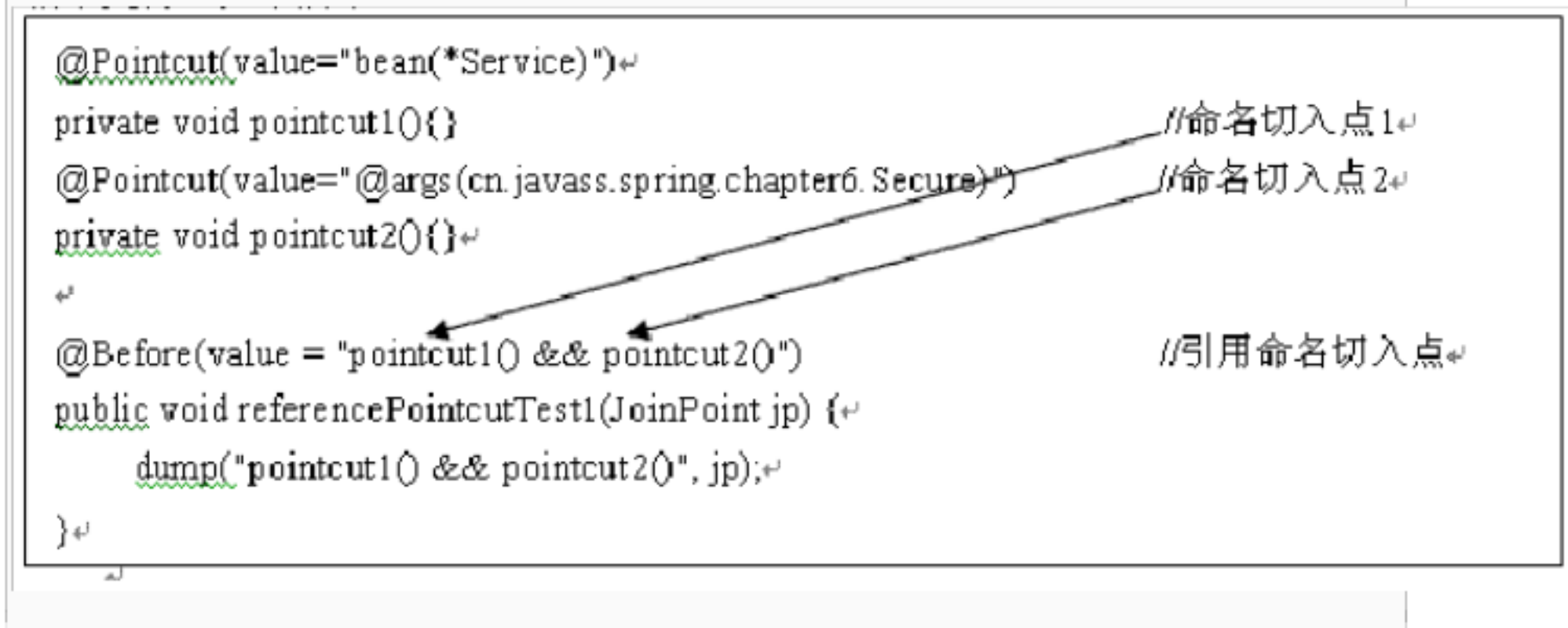
九、 @annotation ：使用“ @annotation(注解类型) ”匹配当前执行方法持有指定注解的方法；注解类型也必须是全限定类型名；

模式	描述
@annotation(cn.javass.spring.chapter6.Secure)	当前执行方法上持有注解 cn.javass.spring.chapter6.Secure 将被匹配

十、 bean ：使用“ bean(Bean id 或名字通配符) ”匹配特定名称的 Bean 对象的执行方法； Spring ASP 扩展的，在 AspectJ 中无相应概念；

模式	描述
bean(*Service)	匹配所有以 Service 命名（ id 或 name ）结尾的 Bean

十一、 reference　　pointcut　：表示引用其他命名切入点，只有 @ApectJ　风格支持，Schema　风格不支持，如下所示：



比如我们定义如下切面：

```
1.  package  cn.javass.spring.chapter6.aop;
2.  import  org.aspectj.lang.annotation.Aspect;
3.  import  org.aspectj.lang.annotation.Pointcut;
4.  @Aspect
5.  public  class  ReferencePointcutAspect {
6.      @Pointcut  (value=  "execution(* *())"  )
7.      public  void  pointcut() {}
8.  }
```

可以通过如下方式引用：

```
1.  @Before (value =  "cn.javass.spring.chapter6.aop.ReferencePointcutAspect.point
    cut()"  )
2.  public  void  referencePointcutTest2(JoinPoint jp) {}
```

除了可以在 @AspectJ 风格的切面内引用外，也可以在 Schema 风格的切面定义内引用，引用方式与 @AspectJ 完全一样。

到此我们切入点表达式语法示例就介绍完了，我们这些示例几乎包含了日常开发中的所有情况， 但当然还有更复杂的语法等等， 如果以上介绍的不能满足您的需要，请参考 AspectJ 文档。

由于测试代码相当长，所以为了节约篇幅本示例代码在 cn.javass.spring.chapter6. PointcutTest 文件中，需要时请参考该文件。

6.6 通知参数

前边章节已经介绍了声明通知，但如果想获取被被通知方法参数并传递给通知方法，该如何实现呢？接下来我们将介绍两种获取通知参数的方式。

- 使用 JoinPoint 获取 ：Spring AOP 提供使用 org.aspectj.lang.JoinPoint 类型获取连接点数据，任何通知方法的第一个参数都可以是 JoinPoint(环绕通知是 ProceedingJoinPoint ，JoinPoint 子类)，当然第一个参数位置也可以是 JoinPoint.StaticPart 类型，这个只返回连接点的静态部分。

1) JoinPoint :提供访问当前被通知方法的目标对象、 代理对象、方法参数等数据：

```
1.  package org.aspectj.lang;
2.  import  org.aspectj.lang.reflect.SourceLocation;
3.  public  interface  JoinPoint {
4.      String toString();                  // 连接点所在位置的相关信息
5.      String toShortString();             // 连接点所在位置的简短相关信息
6.      String toLongString();              // 连接点所在位置的全部相关信息
7.      Object getThis();                   // 返回 AOP代理对象
8.      Object getTarget();                 // 返回目标对象
9.      Object[] getArgs();                 // 返回被通知方法参数列表
10.     Signature getSignature();           // 返回当前连接点签名
11.     SourceLocation getSourceLocation();     // 返回连接点方法所在类文件中的位置
12.     String getKind();                   // 连接点类型
13.     StaticPart getStaticPart();         // 返回连接点静态部分
14. }
```

2）ProceedingJoinPoint ：用于环绕通知，使用 proceed() 方法来执行目标方法：

```
1.  public  interface  ProceedingJoinPoint  extends  JoinPoint {
2.      public  Object proceed()  throws  Throwable;
```

```
3.          public     Object proceed(Object[] args)              throws    Throwable;
4.    }
```

3) JoinPoint.StaticPart ：提供访问连接点的静态部分， 如被通知方法签名、连接点类型等：

```
1.    public     interface      StaticPart {
2.    Signature getSignature();                  //  返回当前连接点签名
3.    String getKind();                          //  连接点类型
4.          int    getId();                          //  唯一标识
5.    String toString();                         //  连接点所在位置的相关信息
6.        String toShortString();                      //  连接点所在位置的简短相关信息
7.        String toLongString();                       //  连接点所在位置的全部相关信息
8.    }
```

使用如下方式在通知方法上声明，必须是在第一个参数，然后使用 jp.getArgs() 就能获取到被通知方法参数：

```
1.    @Before (value=    "execution(* sayBefore(*))"            )
2.    public     void    before(JoinPoint jp) {}
3.
4.    @Before (value=    "execution(* sayBefore(*))"            )
5.    public     void    before(JoinPoint.StaticPart jp) {}
```

- 自动获取： 通过切入点表达式可以将相应的参数自动传递给通知方法， 例如前边章节讲过的返回值和异常是如何传递给通知方法的。

  在 Spring AOP 中, 除了 execution 和 bean 指示符不能传递参数给通知方法，其他指示符都可以将匹配的相应参数或对象自动传递给通知方法。

```
1.    @Before (value=    "execution(* test(*)) && args(param)"                , argNames=    "param"    )
2.    public     void    before1(String param) {
3.        System.out.println(              "===param:"     + param);
4.    }
```

切入点表达式 execution(* test(*)) && args(param) ：

1）首先 execution(* test(*)) 匹配任何方法名为 test，且有一个任何类型的参数；

2）args(param) 将首先查找通知方法上同名的参数，并在方法执行时（运行时）匹配传入的参数是使用该同名参数类型，即 java.lang.String ；如果匹配将把该被通知参数传递给通知方法上同名参数。

其他指示符（除了 execution 和 bean 指示符）都可以使用这种方式进行参数绑定。

在此有一个问题，即前边提到的类似于【 3.1.2 构造器注入】中的参数名注入限制：在 class 文件中没生成变量调试信息是获取不到方法参数名字的。

所以我们可以使用策略来确定参数名：

1. 如果我们通过 “ argNames 属性指定了参数名，那么就是要我们指定的；

查看复制到剪贴板打印

```
1. @Before (value= " args(param)" , argNames= "param" ) // 明确指定了
2. public void before1(String param) {
3. System.out.println( "===param:" + param);
4. }
```

1. 如果第一个参数类型是 JoinPoint 、ProceedingJoinPoint 或 JoinPoint.StaticPart 类型，应该从 “ argNames ”属性省略掉该参数名（可选，写上也对），这些类型对象会自动传入的，但必须作为第一个参数；

查看复制到剪贴板打印

```
1. @Before (value= " args(param)" , argNames= "param" ) // 明确指定了
2. public void before1(JoinPoint jp, String param) {
3. System.out.println( "===param:" + param);
4. }
```

1. 如果 “class 文件中含有变量调试信息 ”将使用这些方法签名中的参数名来确定参数名；

查看复制到剪贴板打印

```
1.  @Before (value=  " args(param)"  )  //  不需要  argNames  了
2.  public  void  before1(JoinPoint jp, String param) {
3.  System.out.println(  "===param:"  + param);
4.  }
```

1. 如果没有“class 文件中含有变量调试信息 ”，将尝试自己的参数匹配算法，如果发现参数绑定有二义性将抛出 AmbiguousBindingException 异常；对于只有一个绑定变量的切入点表达式，而通知方法只接受一个参数，说明绑定参数是明确的，从而能配对成功。

查看复制到剪贴板打印

```
1.  @Before (value=  " args(param)"  )
2.  public  void  before1(JoinPoint jp, String param) {
3.  System.out.println(  "===param:"  + param);
4.  }
```

1. 以上策略失败将抛出 IllegalArgumentException 。

接下来让我们示例一下组合情况吧：

查看复制到剪贴板打印

```
1.  @Before (args(param) && target(bean) &&  @annotation  (secure)",
2.  argNames=  "jp,param,bean,secure"  )
3.  public  void  before5(JoinPoint jp, String param,
4.  IPointcutService pointcutService, Secure secure) {
5.  ……
6.  }
```

该示例的执行步骤如图 6-5 所示。

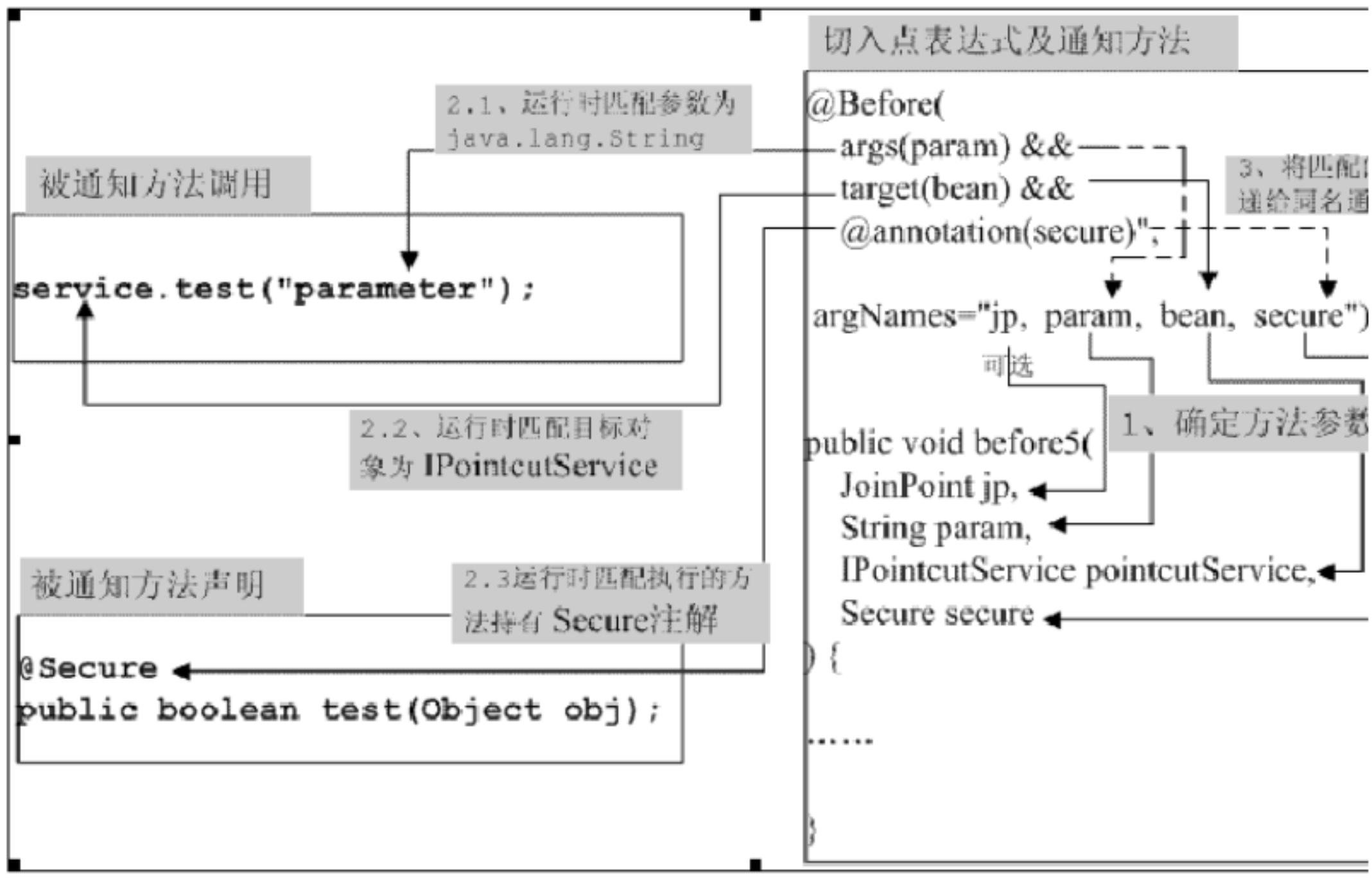


图 6-5 参数自动获取流程

图 6-5 参数自动获取流程

除了上边介绍的普通方式，也可以对使用命名切入点自动获取参数：

查看复制到剪贴板打印

```
1.  @Pointcut (value= "args(param)" , argNames= "param" )
2.  private void pointcut1(String param){}
3.  @Pointcut (value= "@annotation(secure)" , argNames= "secure" )
4.  private void pointcut2(Secure secure){}
5.
6.  @Before (value = "pointcut1(param) && pointcut2(secure)" ,
7.  argNames= "param, secure" )
8.  public void before6(JoinPoint jp, String param, Secure secure) {
9.  ……
10. }
```

自此给通知传递参数已经介绍完了，　示例代码在 cn.javass.spring.chapter6.ParameterTest　文件中。

在 Spring　配置文件中，所以 AOP 相关定义必须放在 <aop:config>　标签下，该标签下可以有 <aop:pointcut>　、<aop:advisor>　、<aop:aspect>　标签，配置顺序不可变。

- <aop:pointcut>　：用来定义切入点，该切入点可以重用；
- <aop:advisor>　：用来定义只有一个通知和一个切入点的切面；
- <aop:aspect>　：用来定义切面，该切面可以包含多个切入点和通知，而且标签内部的通知和切入点定义是无序的；和　advisor　的区别就在此，　advisor　只包含一个通知和一个切入点。

<aop:config>	AOP定义开始（有序）
<aop:pointcut/>	切入点定义（零个或多个）
<aop:advisor/>	Advisor定义（零个或多个）
<aop:aspect>	切面定义开始（零个或多个，无序）
<aop:pointcut/>	切入点定义（零个或多个）
<aop:before"/>	前置通知（零个或多个）
<aop:after-returning/>	后置返回通知（零个或多个）
<aop:after-throwing/>	后置异常通知（零个或多个）
<aop:after/>	后置最终通知（零个或多个）
<aop:around/>	环绕通知（零个或多个）
<aop:declare-parents/>	引入定义（零个或多个）
</aop:aspect>	切面定义开始（零个或多个）
</aop:config>	AOP定义结束