## [简介](http://blog.csdn.net/zl3450341/article/details/7673938)

**Aspectj是什么**

　　官方网站的的介绍是这样的：

* a seamless aspect-oriented extension to the Javatm programming language（一种基于Java平台的面向切面编程的语言）
* Java platform compatible（兼容Java平台，可以无缝扩展）
* easy to learn and use（易学易用）

**Aspectj能做什么**

        clean modularization of crosscutting concerns, such as error checking and handling, synchronization, context-sensitive behavior, performance optimizations, monitoring and logging, debugging support, and multi-object protocols。

         大意是说：干净的模块化横切关注点（也就是说单纯，基本上无侵入），如错误检查和处理，同步，上下文敏感的行为，性能优化，监控和记录，调试支持，多目标的协议。

**还有那些常用的Aop**，**以及他们的区别**

* Jboss Aop：我基本上没有用过，所以没有发言权
* Spring Aop：Spring自己原生的Aop，只能用一个词来形容：难用。　你需要实现大量的接口，继承大量的类，所以spring aop一度被千夫所指。这于他的无侵入，低耦合完全冲突。不过Spring对开源的优秀框架，组建向来是采用兼容，并入的态度。所以，后来的Spring 就提供了Aspectj支持，也就是我们后来所说的基于纯POJO的Aop。

　　　区别：Spring Aop采用的动态织入，而Aspectj是静态织入。静态织入：指在编译时期就织入，即：编译出来的class文件，字节码就已经被织入了。动态织入又分静动两种，静则指织入过程只在第一次调用时执行；动则指根据代码动态运行的中间状态来决定如何操作，每次调用Target的时候都执行。有不清楚的同学，可以自己补下基础的代理知识。

**Aop术语解释**

　　　基本上每篇Aop的文章都必讲的内容，不过笔者所将可能与他们不同，笔者不会照本宣科。

　　　pointcut：　是一个（组）基于正则表达式的表达式，有点绕，就是说他本身是一个表达式，但是他是基于正则语法的。通常一个pointcut，会选取程序中的某些我们感兴趣的执行点，或者说是程序执行点的集合。

　　　joinPoint：　通过pointcut选取出来的集合中的具体的一个执行点，我们就叫JoinPoint.

　　　Advice：　在选取出来的JoinPoint上要执行的操作、逻辑。关于５种类型，我不多说，不懂的同学自己补基础。

　　　aspect：　就是我们关注点的模块化。这个关注点可能会横切多个对象和模块，事务管理是横切关注点的很好的例子。它是一个抽象的概念，从软件的角度来说是指在应用程序不同模块中的某一个领域或方面。又pointcut 和　 advice组成。

　　　Target：被aspectj横切的对象。我们所说的joinPoint就是Target的某一行，如方法开始执行的地方、方法类调用某个其他方法的代码。

## [搭建开发环境](http://blog.csdn.net/zl3450341/article/details/7673942)

**下载Aspectj以及AJDT**

　　　　上一章已经列出了他的官方网站，自己上去download吧。AJDT是一个eclipse插件，开发aspectj必装，他可以提供语法检查，以及编译。这里要说一点重要的知识：

aspectj不能使用传统的JDK编译，他的编译器扩展自JDK。AJDT提供的编译功能，就为我们省了很多事，当然你也可以用命令行自己去编译（不过我从来没有这么做过）。

　　　　无论是apsectj的安装，还是AJDT网上还是有很多文章讲的。不会的同学可以自己Google。

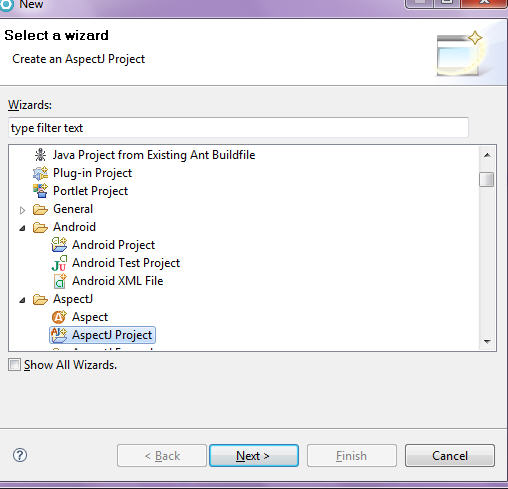
**创建项目**

　创建项目的时候，我们选择New--->Aspectj Project**。**当然，你也可以选择先创建一个普通的Java项目（包括JavaEE项目），然后选中project，右键---->convert to aspectj Project。　其实无论那种方式，无非都是为了：使用apsectj的编译器去编译代码，而不用JDk。

　有写同学可能用了Maven来管理项目，不用担心，maven已经提供了aspectj的编译插件。可以参见：<http://maven.apache.org/maven-1.x/plugins/aspectj/>　和　<http://mojo.codehaus.org/aspectj-maven-plugin/>

上面都有比较完整的使用介绍。到这：我们的环境就搭建好了，马上我门就要正式进入aspectj的开发了。怎么样？激动吧。下节我们就开始讲语法，并构建Hello World

**创建项目**



我们将project命名为：aspectjDemo。然后我们新建２个package：com.aspectj.demo.aspect　　和　com.aspectj.demo.test

前者用来方apsect。后者用来放测试类。如果你仔细的话，你会发现Aspectj的项目上面有个AJ的标志。

**二、创建Aspect**

　　　首先我们创建HelloWorld.java。他包含main()方法，但是没有方法体，代码如下：

package com.aspectj.demo.test;

public class HelloWorld {

/\*\*

\* @param args

\*/

public static void main(String[] args) {

}

}

接下来创建一个Aspectj。我们命名为：HelloAspect，其后缀名我：.aj　。这里千万别习惯性就New --->Class了哦～切记。接下来我们编写这个aspect的内容。

package com.aspectj.demo.aspect;

public aspect HelloAspect {

pointcut HelloWorldPointCut() : execution(\* com.aspectj.demo.test.HelloWorld.main(..));

before() : HelloWorldPointCut(){

System.out.println("Hello world");

}

}

**三、运行结果于分析**

　　运行HelloWorld，你会发现打印了Hello world.

　　另外不知道你发现没有在Line Number那里现实的深色的箭头？　移上去，看看他说什么？　advices HelloWorld.main(String[])。　意思是说：横切了HelloWorld的main(String[])方法。

　　同样在HelloWorld这边也有箭头，这是箭头的方向不同。鼠标移上去，读读上面的提示吧。～

　　从这个demo我们可以看出，Aspectj真的是很简单，就如第一章他的自我介绍一样：易学易用，而且不侵入。不需要继承任何类和接口。　只要编写一个pointcut和　advice就ok了。

　　　怎么样?有点感觉没有?　他感觉一下这个demo。下一章我们将开始讲他的基础语法。

## [pointcut基础语法](http://blog.csdn.net/zl3450341/article/details/7673950)

**一、aspect的定义**

　　　运行完HelloWorld以后，我们来看下aspect的基础语法：

１、定义一个切面：　关键字aspect。　这定义Java类的语法类似。

２、定义pointcut：　　［修饰符(public,protected.....)］　pointcut  poincut名字()　:  表达式;

3、定义advice：　　　通知类型()　：　pointcut名字(){ .......逻辑}

　　　一个最基本的aspect，就是这样组成的。值得一提的是：aspectj支持很多类型的pointcut，最基本的就是method call pointcut（方法级别），而Spring的aop 仅支持method call　pointcut。所以，在后面陆续的使用中，你将会发现aspectj的强大，　简直强大到有点过分。而至于advice，aspectj也一样，就是５种类型。

二、**pointcut的主要类型**

|  |  |
| --- | --- |
| **Methods and Constructors** | |
| call(*Signature*) | every call to any method or constructor matching *Signature* at the call site（方法和构造函数的调用点） |
| execution(*Signature*) | every execution of any method or constructor matching *Signature　（方法和构造函数的执行点）* |
| **Fields** | |
| get(*Signature*) | every reference to any field matching *Signature　　　（属性的读操作）* |
| set(*Signature*) | every assignment to any field matching *Signature*. The assigned value can be exposed with anargspointcut　（属性的写操作） |
| **Exception Handlers** | |
| handler(*TypePattern*) | every exception handler for any Throwable type in *TypePattern*. The exception value can be exposed with anargs pointcut（异常处理执行） |
| **Advice** | |
| adviceexecution() | every execution of any piece of advice（Advice执行） |
| **Initialization** | |
| staticinitialization(*TypePattern*) | every execution of a static initializer for any type in *TypePattern  (类初始化)* |
| initialization(*Signature*) | every initialization of an object when the first constructor called in the type matches*Signature*, encompassing the return from the super constructor call to the return of the first-called constructor (对象初始化) |
| preinitialization(*Signature*) | every pre-initialization of an object when the first constructor called in the type matches*Signature*, encompassing the entry of the first-called constructor to the call to the super constructor (对象预先初始化) |
| **Lexical** | |
| within(*TypePattern*) | every join point from code defined in a type in *TypePattern  (捕获在指定类或者方面中的程序体中的所有连接点，包括内部类)* |
| withincode(*Signature*) | every join point from code defined in a method or constructor matching *Signature (用于捕获在构造器或者方法中的所有连接点，包括在其中的本地类)* |

|  |  |
| --- | --- |
| **Instanceof checks and context exposure** | |
| this(*Type* or *Id*) | every join point when the currently executing object is an instance of *Type* or*Id*'s type(所有Type or id 的实例的执行点，匹配所有的连接点，如方法调用，属性设置，当前的执行对象为Account，或者其子类。) |
| target(*Type* or *Id*) | every join point when the target executing object is an instance of *Type* or*Id*'s type (配所有的连接点，目标对象为Type或Id) |
| args(*Type* or *Id*, ...) | every join point when the arguments are instances of *Type*s or the types of the*Id*s (参数类型为Type) |
| **Control Flow** | |
| cflow(*Pointcut*) | every join point in the control flow of each join point *P* picked out by*Pointcut*, including *P* itself  (捕获所有的连接点在指定的方法执行中，包括执行方法本身) |
| cflowbelow(*Pointcut*) | every join point below the control flow of each join point *P* picked out by*Pointcut*; does not include *P* itself　（捕获所有的连接点在指定的方法执行中，除了执行方法本身） |
| **Conditional** | |
| if(*Expression*) | every join point when the boolean *Expression* is true |

|  |  |
| --- | --- |
| **Combination　（逻辑／结合操作）** | |
| ! *Pointcut* | every join point not picked out by *Pointcut* |
| *Pointcut0* && *Pointcut1* | each join point picked out by both *Pointcut0* and *Pointcut1* |
| *Pointcut0* || *Pointcut1* | each join point picked out by either *Pointcut0* or *Pointcut1* |
| ( *Pointcut* ) | each join point picked out by *Pointcut* |

     前面说过pointcut基于正则的语法，那么肯定也支持通配符，含义如下：

\* 表示任何数量的字符，除了(.)   
.. 表示任何数量的字符包括任何数量的(.)   
+ 描述指定类型的任何子类或者子接口  
同java一样，提供了一元和二元的条件表达操作符。  
一元操作符：!  
二元操作符：||和&&  
优先权同java

## [args带参数的pointcut](http://blog.csdn.net/zl3450341/article/details/7673954)

上一节，我们overview了一下pointcut的类型，怎么够多吧？　不夸张的说基本涵盖了Java程序的所有生命周期。这就以为着：我们可以控制到一个已经存在的Java程序的任何地方和环节。可能你还不太懂上一节的东西，或是懵懂。不要急，接下来，我会把常用的几个一一帮你介绍。但是本节，我们仍然要看基础语法。

　　为了演示，我们要修改我们的HelloWorld.java。修改后的代码如下：

package com.aspectj.demo.test;

public class HelloWorld {

public static void main(int i){

System.out.println("int the main method i = " + i);

}

/\*\*

\* @param args

\*/

public static void main(String[] args) {

main(5);

}

}

我们增加了一个main(int i)的方法。再运行一下，发现拦截２次，也就是说：２个main()方法都被拦截，现在，leader说：我只要你拦截接受int参数的main()。怎么办？

那么我们修改一下HelloAspect。

pointcut HelloWorldPointCut() : execution(\* com.aspectj.demo.test.HelloWorld.main(int));

再运行一下，发现只拦截了一次吧。

　　可leader这人比较烦，他总是没完没了。他又说：我现在要获取main()方法里面的参数值。又加需求，广大同胞最痛恨的事情莫过于此。抱怨归抱怨，代码终归还得改吧，谁让我们吃这碗饭的勒。修改过后的代码如下：

package com.aspectj.demo.aspect;

public aspect HelloAspect {

pointcut HelloWorldPointCut(int i) : execution(\* com.aspectj.demo.test.HelloWorld.main(int)) && args(i);

before(int x) : HelloWorldPointCut(x){

x+=5;

System.out.println("in the aspect i = " +x);

}

}

赶紧运行一下。。看看结果～　　发现是不是获取到了？　leader又说了：我们要改变参数的值。可你也看到了，目前我们做的改变没有影响到main()方法啊。怎么办勒？这里我们先卖个关子，稍候我们介绍advice的时候来解决。

## [call，execution，within,withincode](http://blog.csdn.net/zl3450341/article/details/7673960)

本节开始，我们将详细介绍pointcut的语法

**一、call和execution**

　　　语法结构：execution([修饰符]　返回值类型　方法名(参数)　［异常模式］)　　蓝色表示可选部分。

　　　例子：

* execution(public \*.\*(..))　　所有的public方法。
* execution(\* hello(..))            所有的hello()方法
* execution(String hello(..))   所有返回值为String的hello方法。
* execution(\* hello(String))  　　所有参数为String类型的hello()
* execution(\* hello(String..))      至少有一个参数，且第一个参数类型为String的hello方法
* execution(\* com.aspect..\*(..))  　所有com.aspect包，以及子孙包下的所有方法
* execution(\* com..\*.\*Dao.find\*(..))　　com包下的所有一Dao结尾的类的一find开头的方法

　　　　上面的几个例子，包括了修饰符，返回值，方法名，以及参数。大家可以根据需要灵活运用。接下来讲讲call　和　execution的区别，从字面理解：call为调用，而execution为执行。　　事实上他们的区别也确实如此。

　　　　call捕获的joinpoint是签名方法的调用点，而execution捕获的则是执行点。为了有说服力，我写了一个demo.代码如下：

package com.aspectj.demo.test;

public class HelloWorld {

public static void main(int i){

System.out.println("in the main method i = " + i);

}

/\*\*

\* @param args

\*/

public static void main(String[] args) {

main(5);

}

}

package com.aspectj.demo.aspect;

public aspect HelloAspect {

pointcut HelloWorldPointCut() : call(\* main(int));

before() : HelloWorldPointCut(){

System.out.println("Entering : " + thisJoinPoint.getSourceLocation());

}

}

  我们拦截了参数为:int的main方法。　这里用到了一内置的对象：thisJoinPoint,他表示当前jionPoint.　跟我们在java中的this　其实是差不多的，如果你不明白，那么你多运行一下，好好体会一下。getSourceLocation（）表示元代吗的位置：　　　我们运行一下HelloWorld.java。　结果如下：

Entering : HelloWorld.java:14

in the main method i = 5

14是行号，即：main(5);

接下来，我们把HelloAspect中的call改为execution。发现打印结果如下：

Entering : HelloWorld.java:6

in the main method i = 5

在比照一下代码，看下输出。　　现在明白了吧？？　　　一个是调用的地方，一个是执行的地方。

　thisJoinPoint.getSourceLocation()　这段代码将会在我们以后的Demo中经常用到。这是一个跟踪调试的好办法。

**二、within 和　withincode**

　　　现在假设你还有另外一个class，他也包含main()方法。

package com.aspectj.demo.test;

public class HelloAspectDemo{

public static void main(String[] args) {

System.out.println("Hello aspectj");

}

}

我们的HelloAspect如下：

package com.aspectj.demo.aspect;

public aspect HelloAspect {

pointcut HelloWorldPointCut() : execution(\* main(..));

before() : HelloWorldPointCut(){

System.out.println("Entering : " + thisJoinPoint.getSourceLocation());

}

}

这样，便会把HelloAspectDemo的main方法捕获到，　　现在leader说：　HelloAspectDemo不需要捕获。那我们怎么办勒？你首先想到的肯定是修改pointcut，指定到我们的HelloWorld类。　这当然是可以的，假设：现在还有5个类，也有main方法，也需要拦截。那你这解决办法肯定就不行了。（当然你也可以用 ||　来组合他们）。这个时候就用到了我们的within了。　修改一下HelloAspect

package com.aspectj.demo.aspect;

import com.aspectj.demo.test.HelloAspectDemo;

public aspect HelloAspect {

pointcut HelloWorldPointCut() : execution(\* main(..)) && !within(HelloAspectDemo);

before() : HelloWorldPointCut(){

System.out.println("Entering : " + thisJoinPoint.getSourceLocation());

}

}

再运行一下HelloAspectDemo，你会发现他被排除了，也就是说他没有被拦截到。

　　withincode与within相似，不过withcode()接受的signature是方法，而不是类。用法，意思都差不多，只不过是使用场合不同。

在接下来的一章，我将给大家介绍aspectj中最难理解的一个概念：控制流（cfow()　cfowbelow()）。

## [控制流cfow,cfowbelow](http://blog.csdn.net/zl3450341/article/details/7673967)

**一、序言**

     cflow我认为是aspectj中最难理解的一个概念，至少我是这么认为的。当初刚接触aspectj的时候，可谓是为之颠倒，不只大家是否有相同的感觉。但有一点不可否认的就是：他觉得是aspectj强大功能之一。 他可以做到Spring AOP无法做到的场景。

**二、控制流**

     什么叫控制流？ 相信很多不理解cflow的同学跟我一样，刚开始也是卡在这里，那么本节，我们就先理清这个概念。先看一段代码：

package com.aspectj.demo.test;

import org.junit.Test;

public class TestCfow {

public void foo(){

System.out.println("foo......");

}

public void bar(){

foo();

System.out.println("bar.........");

}

@Test

public void testMethod(){

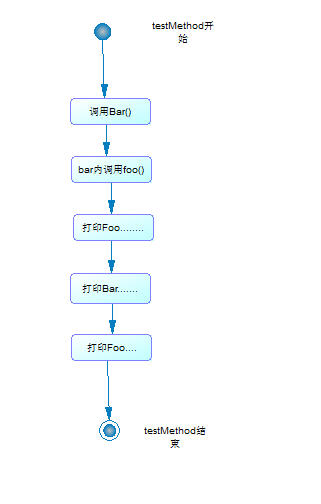
bar();

foo();

}

}

请用流程图画出testmethod()的流程，相信这个大家都画的出来吧？ 



那么现在你看着你画的图，我告诉你：这就是testMehtod（）的控制流。那么cfow(execution(\* testMethod())) 就是获取testMethod()的控制流。他将拦截到testMethod中的每行代码（包括：他流程里面调用的其它其他方法的流程，不管调用层次有多深，都属于他的控制流）。

       ps: 其实这里说是每行代码是不准的，其实是每行代码编译后的字节码。比如System.out.println() 其实编译后是3句话。

**三、demo说明**

       首先，我们来看下bar的控制流，用我们的老办法。看招：

package com.aspectj.demo.aspect;

public aspect CfowAspect {

pointcut barPoint() : execution(\* bar(..));

pointcut fooPoint() : execution(\* foo(..));

pointcut barCfow() : cflow(barPoint());//cflow的参数是一个pointcut

pointcut fooInBar() : barCfow() && fooPoint(); //获取bar流程内的foo方法调用

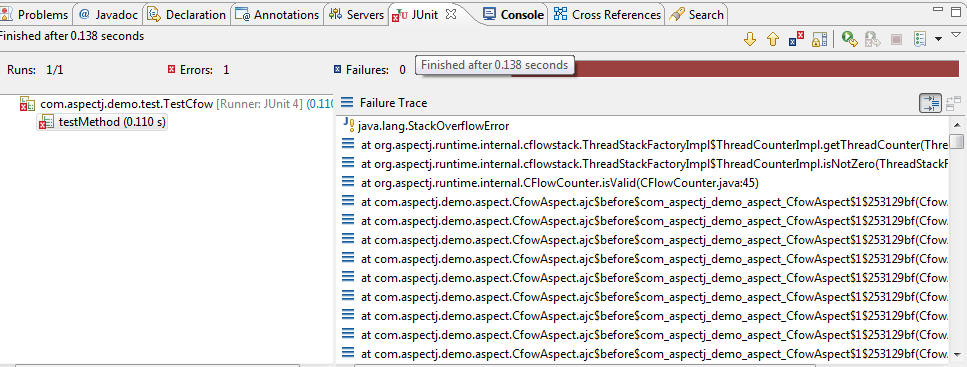
before() : barCfow(){

System.out.println("Enter:" + thisJoinPoint);

}

}

运行一下testCfow().看看是什么结果？



StackOverFow了, 那么为什么会溢出勒？其实是这样的：cflowAspect织入了 Bar(). 所以他也算是bar的控制流的一部分， 这样一来，他就自己拦截自己，形成一个死循环，所以就溢出了。恍然一个大悟吧！

那我们如何解决这个问题勒？记得上一节讲过的within么？ 让我们修改一下代码.

pointcut barCfow() : cflow(barPoint()) && !within(CfowAspect);

再次运行testMehtod()，打印结果如下：

Enter:execution(void com.aspectj.demo.test.TestCfow.bar())

Enter:call(void com.aspectj.demo.test.TestCfow.foo())

Enter:execution(void com.aspectj.demo.test.TestCfow.foo())

Enter:get(PrintStream java.lang.System.out)

Enter:call(void java.io.PrintStream.println(String))

foo......

Enter:get(PrintStream java.lang.System.out)

Enter:call(void java.io.PrintStream.println(String))

bar.........

foo......

每条打印，都可以看出是拦截的那句话，同时这个结果也验证了我给大家PS的那句话。始终要记得：aspectj是静态织入，所以他拦截的是字节码~。

    现在我们改变一下需求：  只拦截bar方法调用里面的foo()方法，也就是说我们testMethod()里面的foo() 调用不要拦截。

package com.aspectj.demo.aspect;

public aspect CfowAspect {

pointcut barPoint() : execution(\* bar(..));

pointcut fooPoint() : execution(\* foo(..));

pointcut barCfow() : cflow(barPoint()) && !within(CfowAspect);//cflow的参数是一个pointcut

pointcut fooInBar() : barCfow() && fooPoint(); //获取bar流程内的foo方法调用

before() : fooInBar(){

System.out.println("Enter:" + thisJoinPoint);

}

}

运行一下TestCflow。

Enter:execution(void com.aspectj.demo.test.TestCfow.foo())

foo......

bar.........

foo......

发现是不是只有bar方法里面的foo()被拦截了？  用Spring AOP你就无法实现这个需求吧？？  是不是突然觉得aspectj真的很强大？？但是我还是要说：这只是他强大的表现之一，冰山一角。 但是他绝对是使用频率最高的功能之一。

**四、总结**

       cfow()获取的是一个控制流程。他很少几乎不单独使用，一般与其他的pointcut 进行 &&运算。若要单独使用，一定要记得用!with()剔除asepct 本身。他是我最喜欢，也是我用的最多的功能，在实际的应用中也用的最广，请好好掌握他。

## [advice](http://blog.csdn.net/zl3450341/article/details/7673968)

asepctj有5种类型的advice

* before( *Formals* )
* after( *Formals* ) returning [ ( *Formal* ) ]
* after( *Formals* ) throwing [ ( *Formal* ) ]
* after( *Formals* )
* *Type* around( *Formals* )

  关于 前四种不想做过多的解释。before已经在我们之前的的Demo中用了无数次了，剩下的3个，我给一个基本的语法就可以了，用起来和before一样。

**[java]** [view plaincopy](http://blog.csdn.net/zl3450341/article/details/7673968)

1. aspect A {
2. pointcut publicCall(): call(**public** Object \*(..));
3. after() returning (Object o): publicCall() {
4. System.out.println("Returned normally with " + o);
5. }
6. after() throwing (Exception e): publicCall() {
7. System.out.println("Threw an exception: " + e);
8. }
9. after(): publicCall(){
10. System.out.println("Returned or threw an Exception");
11. }
12. }

   如果不太清楚的同学，可以自己把我们之前的Demo改进，看看结果便清楚。接下来，我们重点讲讲around通知：

**[java]** [view plaincopy](http://blog.csdn.net/zl3450341/article/details/7673968)

1. **package** com.aspectj.demo.aspect;
3. **import** com.aspectj.demo.test.HelloAspectDemo;


7. **public** aspect HelloAspect {
9. pointcut HelloWorldPointCut(**int** x) : execution(\* main(**int**)) && !within(HelloAspectDemo) && args(x);


13. **int** around(**int** x) : HelloWorldPointCut(x){
14. System.out.println("Entering : " + thisJoinPoint.getSourceLocation());
15. **int** newValue = proceed(x\*3);
16. **return** newValue;
17. }
18. }

**[java]** [view plaincopy](http://blog.csdn.net/zl3450341/article/details/7673968)

1. **package** com.aspectj.demo.test;
3. **public** **class** HelloWorld {

6. **public** **static** **int** main(**int** i){
7. System.out.println("in the main method  i = " + i);
8. **return** i;
9. }
11. /\*\*
12. \* @param args
13. \*/
14. **public** **static** **void** main(String[] args) {
15. main(5);
16. }
18. }

  最主要的就是 proceed（）这个方法~ 重要的还是自己感觉一下吧。

## [Aspectj5支持Annotaion](http://blog.csdn.net/zl3450341/article/details/7673973)

自从JDK5.0加入了annotation以后，asepctj也提供对annotaion的支持，而且命名也模仿JDK，从1.4的版本改为5.0  也就是Aspectj5，或者称@Aspectj。其中最重要的一项就是pointcut  支持对Annotaion的选取了。

        不管你做没做过设计，你肯定用过自定义的Annotation（如果有不会的同学请Google）。为什么要自定义Annotaion勒？其实就是要给我们的某些类或者方法，加上一个标示，用于与其他的普通类或者方法进行区分，并且通过Annotaion参数值携带一定的附加信息。

        那么@Aspectj 对 Annotation的支持是什么用的勒？  让我们看以下官方给出的Demo。

**一、类级别的Annotaion（假定我们定义了一个@Immutable的 类级别Annotation）。**

(@Immutable \*)

Matches any type with an @Immutable annotation.

(!@Immutable \*)

Matches any type which does not have an @Immutable annotation.

(@Immutable (org.xyz.\* || org.abc.\*))

Matches any type in the org.xyz or org.abc packages with the@Immutable annotation.

((@Immutable Foo+) || Goo)

Matches a type Foo or any of its subtypes, which have the@Immutable annotation, or a type Goo.

((@(Immutable || NonPersistent) org.xyz..\*)

Matches any type in a package beginning with the prefix org.xyz, which has either the @Immutable annotation or the @NonPersistentannotation.

(@Immutable @NonPersistent org.xyz..\*)

Matches any type in a package beginning with the prefix org.xyz, which has both an @Immutable annotation and an @NonPersistentannotation.

(@(@Inherited \*) org.xyz..\*)

Matches any type in a package beginning with the prefix org.xyz, which has an inheritable annotation. The annotation pattern @(@Inherited \*) matches any annotation of a type matching the type pattern@Inherited \*, which in turn matches any type with the @Inherited annotation.

**二、作用于Field的Annotaion。**

@SensitiveData \* \*

Matches a field of any type and any name, that has an annotation of type@SensitiveData

@SensitiveData List org.xyz..\*.\*

Matches a member field of a type in a package with prefixorg.xzy, where the field is of type List, and has an annotation of type@SensitiveData

(@SensitiveData \*) org.xyz..\*.\*

Matches a member field of a type in a package with prefixorg.xzy, where the field is of a type which has a @SensitiveData annotation.

@Foo (@Goo \*) (@Hoo \*).\*

Matches a field with an annotation @Foo, of a type with an annotation@Goo, declared in a type with annotation @Hoo.

@Persisted @Classified \* \*

Matches a field with an annotation @Persisted and an annotation@Classified.

**三、作用于Method的Annotaion**

@Oneway \* \*(..)

Matches a method with any return type and any name, that has an annotation of type@Oneway.

@Transaction \* (@Persistent org.xyz..\*).\*(..)

Matches a method with the @Transaction annotation, declared in a type with the@Persistent annotation, and in a package beginning with theorg.xyz prefix.

\* \*.\*(@Immutable \*,..)

Matches any method taking at least one parameter, where the parameter type has an annotation@Immutable.

上面的英问说明很简单，我没有翻译，如果觉得有困难的同学可以用Google翻译以下大致的意思，基本也就明白了。上面的列表基本上列出了所有我们在实际应用中会用到的语法，而且有了前面的基础，这块的Demo，我也就不一一写了，感兴趣的同学可以自己写点Demo感觉一下。

             本章唯一想说的就是3个东西：

@this(Foo)

匹配绑定了@Foo这个注解的任何对象

@args

**[java]** [view plaincopy](http://blog.csdn.net/zl3450341/article/details/7673973)

1. /\*\*
2. \* matches any join point with at least one argument, and where the
3. \* type of the first argument has the @Classified annotation
4. \*/
5. pointcut classifiedArgument() : @args(Classified,..);
7. /\*\*
8. \* matches any join point with three arguments, where the third
9. \* argument has an annotation of type @Untrusted.
10. \*/
11. pointcut untrustedData(Untrusted untrustedDataSource) :
12. @args(\*,\*,untrustedDataSource);

@target

     call(\* \*(..)) && @target(Classified)   匹配所有的方法，但是被匹配的方法所在的Class须有@Classfied注解

@withincode

@within(Foo)

Matches any join point where the executing code is defined within a type which has an annotation of typeFoo.

@annotaion（注解类型）

        匹配任何包含该注解的 Jointpoint

      使用以上语法，都必须要求你的注解的Policy是RUNTIME，否则会编译不通过。

参考资料以及Demo : <http://www.eclipse.org/aspectj/doc/released/adk15notebook/annotations-pointcuts-and-advice.html>

## [target() this() within()的区别](http://blog.csdn.net/zl3450341/article/details/7673979)

经过上面的一个阶段的学习，你可能已经知道了这3个关键字，而且多是用来做过滤的。你是否很迷茫：到底他们有什么区别？？首先来搭建测试环境。有如下代码：

**[java]** [view plaincopy](http://blog.csdn.net/zl3450341/article/details/7673979)

1. **package** com.aspectj.demo.different;
3. **public** **interface** Animal {

6. **public** **void** move();
7. }

**[java]** [view plaincopy](http://blog.csdn.net/zl3450341/article/details/7673979)

1. **package** com.aspectj.demo.different;
3. **public** **class** Snake **implements** Animal{
5. **public** **void** move() {
6. System.out.println("Snake is moving ....");
7. }
9. }

**[java]** [view plaincopy](http://blog.csdn.net/zl3450341/article/details/7673979)

1. **package** com.aspectj.demo.different;
3. **public** **class** Bird **implements** Animal {
5. **public** **void** move() {
6. System.out.println("Snake is flying ....");
7. }
9. }

**[java]** [view plaincopy](http://blog.csdn.net/zl3450341/article/details/7673979)

1. **package** com.aspectj.demo.different;
3. **import** java.util.ArrayList;
4. **import** java.util.List;
6. **import** org.junit.Test;
8. **public** **class** TestMove {

11. **public** **void** move(List<Animal> list){
12. **for**(Animal a : list){
13. a.move();
14. }
15. }
17. @Test
18. **public** **void** testAA(){
19. List<Animal> list = **new** ArrayList<Animal>();
20. list.add(**new** Bird());
21. list.add(**new** Snake());
22. move(list);
23. }
24. }

   这就是我们的测试环境：  一个Animal接口，2个实现类Snake和Bird以及TestMove测试类。   他们都有move()方法，那么下面我将给出不同的pointcut.让你先看输出结果，请一定要用心看每个测试的pointcut的细节变化，并将代码copy（或者手动输入）到eclipse中，执行，看结果、用心去体会，很多东西我们只可意会，无法言传。然后我们再来总结：

**1、测试一**

**[java]** [view plaincopy](http://blog.csdn.net/zl3450341/article/details/7673979)

1. **package** com.aspectj.demo.aspect;
3. **import** com.aspectj.demo.different.Animal;
5. **public** aspect MoveAspect {
7. pointcut MoveAspect() :   call(\* move(..));

10. before() : MoveAspect(){
11. System.out.println("Entering " + thisJoinPoint.getSourceLocation());
12. }
14. }

  打印结果如下：

**[html]** [view plaincopy](http://blog.csdn.net/zl3450341/article/details/7673979)

1. Entering TestMove.java:22
2. Entering TestMove.java:13
3. Snake is flying ....
4. Entering TestMove.java:13
5. Snake is moving ....

    结果：   拦截到了所有的move(..)方法调用点。

**2、测试二**

**[java]** [view plaincopy](http://blog.csdn.net/zl3450341/article/details/7673979)

1. **package** com.aspectj.demo.aspect;
3. **import** com.aspectj.demo.different.Animal;
5. **public** aspect MoveAspect {
7. pointcut MoveAspect() :   call(\* move(..)) && target(Animal);

10. before() : MoveAspect(){
11. System.out.println("Entering " + thisJoinPoint.getSourceLocation());
12. }
14. }

 打印：

**[html]** [view plaincopy](http://blog.csdn.net/zl3450341/article/details/7673979)

1. Entering TestMove.java:13
2. Snake is flying ....
3. Entering TestMove.java:13
4. Snake is moving ....

结果：所有Animal 的实现类的move()方法都被拦截到了，说明target()的获取继承关系。

**3、测试三**

**[java]** [view plaincopy](http://blog.csdn.net/zl3450341/article/details/7673979)

1. **package** com.aspectj.demo.aspect;
3. **import** com.aspectj.demo.different.Animal;
5. **public** aspect MoveAspect {
7. pointcut MoveAspect() :   call(\* move(..)) && **this**(Animal);

10. before() : MoveAspect(){
11. System.out.println("Entering " + thisJoinPoint.getSourceLocation());
12. }
14. }

 打印：

**[html]** [view plaincopy](http://blog.csdn.net/zl3450341/article/details/7673979)

1. Snake is flying ....
2. Snake is moving ....

结果： 没有拦截到任何方法。

**4、测试四**

**[java]** [view plaincopy](http://blog.csdn.net/zl3450341/article/details/7673979)

1. **package** com.aspectj.demo.aspect;
3. **import** com.aspectj.demo.different.TestMove;
5. **public** aspect MoveAspect {
7. pointcut MoveAspect() :   call(\* move(..)) && **this**(TestMove);

10. before() : MoveAspect(){
11. System.out.println("Entering " + thisJoinPoint.getSourceLocation());
12. }
14. }

 打印：

**[html]** [view plaincopy](http://blog.csdn.net/zl3450341/article/details/7673979)

1. Entering TestMove.java:22
2. Entering TestMove.java:13
3. Snake is flying ....
4. Entering TestMove.java:13
5. Snake is moving ....

结果： 在 TestMove类里面所有掉用move(..)方法的地方都被拦截到了。

**5、测试五**

**[java]** [view plaincopy](http://blog.csdn.net/zl3450341/article/details/7673979)

1. **package** com.aspectj.demo.aspect;
3. **import** com.aspectj.demo.different.Animal;
5. **public** aspect MoveAspect {
7. pointcut MoveAspect() :   execution(\* move(..)) && **this**(Animal);

10. before() : MoveAspect(){
11. System.out.println("Entering " + thisJoinPoint.getSourceLocation());
12. }
14. }

打印：

**[html]** [view plaincopy](http://blog.csdn.net/zl3450341/article/details/7673979)

1. Entering Bird.java:5
2. Snake is flying ....
3. Entering Snake.java:5
4. Snake is moving ....

结果：所有在Animal及字子类中执行move()的点都被拦截了

到现在为止，你应该感觉出了target和this那微妙的不同同了吧？如果还没有感觉，说明你没有认真体会。

**6、测试六**

**[java]** [view plaincopy](http://blog.csdn.net/zl3450341/article/details/7673979)

1. **package** com.aspectj.demo.aspect;
3. **import** com.aspectj.demo.different.Animal;
5. **public** aspect MoveAspect {
7. pointcut MoveAspect() :   execution(\* move(..)) && within(Animal);

10. before() : MoveAspect(){
11. System.out.println("Entering " + thisJoinPoint.getSourceLocation());
12. }
14. }

打印：

**[html]** [view plaincopy](http://blog.csdn.net/zl3450341/article/details/7673979)

1. Snake is flying ....
2. Snake is moving ....

结果：无拦截到任何方法，因为Animal是接口，无法直接调用move().方法。

**7、测试七**

**[java]** [view plaincopy](http://blog.csdn.net/zl3450341/article/details/7673979)

1. **package** com.aspectj.demo.aspect;
3. **import** com.aspectj.demo.different.Snake;

6. **public** aspect MoveAspect {
8. pointcut MoveAspect() :   execution(\* move(..)) && within(Snake);

11. before() : MoveAspect(){
12. System.out.println("Entering " + thisJoinPoint.getSourceLocation());
13. }
15. }

打印：

**[html]** [view plaincopy](http://blog.csdn.net/zl3450341/article/details/7673979)

1. Snake is flying ....
2. Entering Snake.java:5
3. Snake is moving ....

结果：只拦截到了Snake的move()方法

**8、测试八**

**[java]** [view plaincopy](http://blog.csdn.net/zl3450341/article/details/7673979)

1. **package** com.aspectj.demo.aspect;
3. **import** com.aspectj.demo.different.Animal;
4. **import** com.aspectj.demo.different.TestMove;


8. **public** aspect MoveAspect {
10. pointcut MoveAspect() :   call(\* move(..)) && target(Animal)&& **this**(TestMove);

13. before() : MoveAspect(){
14. System.out.println("Entering " + thisJoinPoint.getSourceLocation());
15. }
17. }

打印：

**[html]** [view plaincopy](http://blog.csdn.net/zl3450341/article/details/7673979)

1. Entering TestMove.java:13
2. Snake is flying ....
3. Entering TestMove.java:13
4. Snake is moving ....

结果：  拦截到了TestMove类中Animal以及他子类的move()方法调用。

**9、测试九**

**[java]** [view plaincopy](http://blog.csdn.net/zl3450341/article/details/7673979)

1. **package** com.aspectj.demo.aspect;
3. **import** com.aspectj.demo.different.Animal;
4. **import** com.aspectj.demo.different.TestMove;


8. **public** aspect MoveAspect {
10. pointcut MoveAspect(Animal a,TestMove t) :   call(\* move(..)) && target(a)&& **this**(t);

13. before(Animal a,TestMove t) : MoveAspect(a,t){
14. System.out.println("Entering " + thisJoinPoint.getSourceLocation() + "  target:" + a + "  this:" + t);
15. }
17. }

打印：

**[html]** [view plaincopy](http://blog.csdn.net/zl3450341/article/details/7673979)

1. Entering TestMove.java:13  target:com.aspectj.demo.different.Bird@7bde7bde  this:com.aspectj.demo.different.TestMove@52b252b2
2. Snake is flying ....
3. Entering TestMove.java:13  target:com.aspectj.demo.different.Snake@7f587f58  this:com.aspectj.demo.different.TestMove@52b252b2
4. Snake is moving ....

结果：   其实这组不算测试，这组Demo是告诉大家target()和this 怎么获取他们自己当前的实例，以及他们如何将实例传给Advice使用

  总结： 第一张我们介绍过，说apsectj是动态、静态植入结合的。 那么Target()  this()就是属于他动态植入的方式，within是静态植入的。故target(),this()需要在运行时才能确定那些被拦截。 比如刚才的例子，我们在给Animal加多一个实现类，用target() 他仍然可以被拦截。  所以target()和this()会用继承关系作用,也就是说：如果你的signature是一个基类，那么这个pointcut同时也会对他的子类也起作用。

             另外target 和 this 可以获取他们对应的实例。  但是within没法作到。

            target()是指：我们pointcut 所选取的Join point 的所有者，直白点说就是： 指明拦截的方法属于那个类。

            this()是指： 我们pointcut 所选取的Join point 的调用的所有者，就是说：方法是在那个类中被调用的。

ps: 关于 this的继承关系没有做测试，有兴趣的自己做一个。还是那句话：用心体会

## [基于注解的Aspect](http://blog.csdn.net/zl3450341/article/details/7673982)

我们还可以用注解的方式来写asepctj. 但是需要加入aspectjweave.jar。用法还是一样，就不想多将了，给个demo吧。

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

import org.aspectj.lang.annotation.Pointcut;

@Aspect

public class AnnotaionAspect {

@Pointcut("execution(\* com.ubs.ifop.application.ep..\*.\*(..))")

public void epAspectjPkg(){

}

@Pointcut("call(\* com.ubs.ifop.application.ep.aspectj.test.Config$IValue.get\*(String,String))")

public void ivalueAspectj(){}

@Pointcut("cflow(ivalueAspectj()) && !within(AnnotaionAspect)")

public void actuallyPoint(){}

@Before("actuallyPoint() && ivalueAspectj()")

public void testExecute(){

System.out.println("aaaaaaaaaaaaa");

}

}