spring-aop底层

> 学习spring-aop的目的是为了更好的了解java内存马，前一段时间学习了agert。然后现在学一下aop，而这些的aop不是讲spring中aop的使用而是aop的底层，因为里面涉及到java的代码和java的反射，然后就学着复习java安全知识。。。

AOP介绍

AOP：Aspect Oriented Programming 面向切面编程。不同于OOP（面向对象编程）

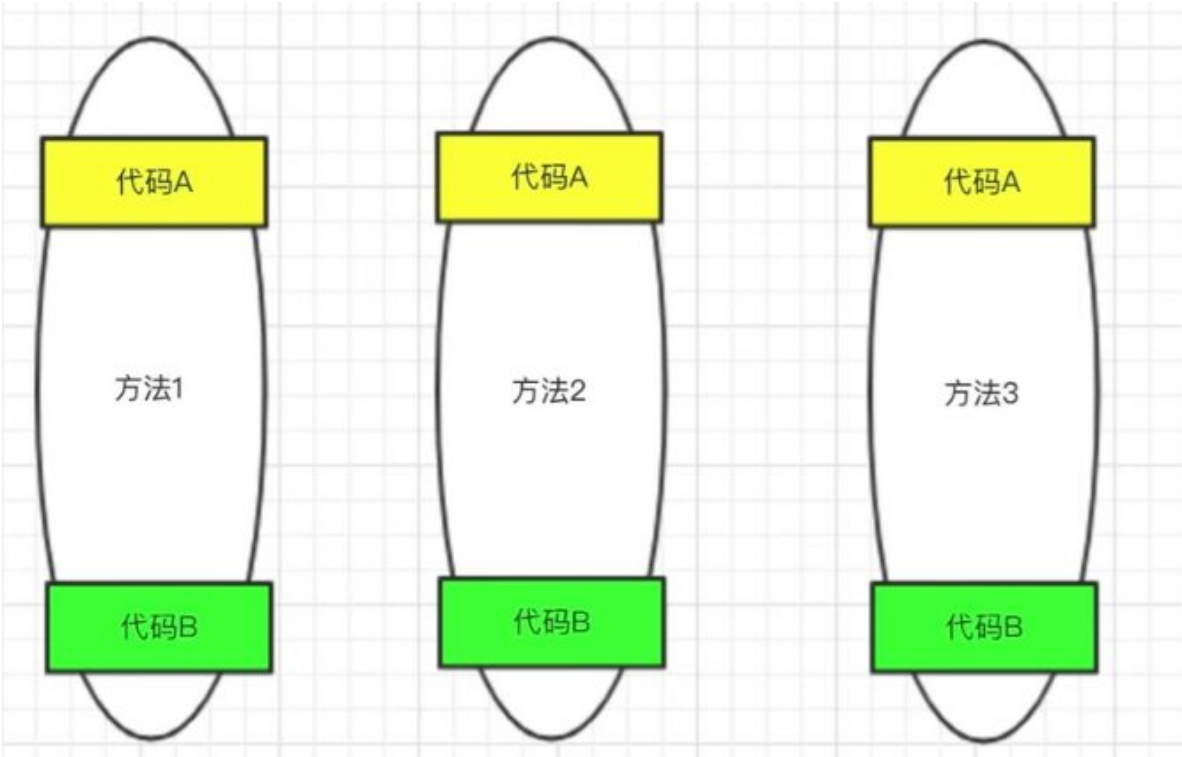
AOP 的优点：

- 降低模块之间的耦合度。
- 使系统更容易扩展。
- 更好的代码复用。
- 非业务代码更加集中，不分散，便于统一管理。
- 业务代码更加简洁存粹，不参杂其他代码的影响。

AOP 是对面向对象编程的一个补充，在运行时，动态地将代码切入到类的指定方法、指定位置上的编程思想就是面向切面编程。将不同方法的同一个位置抽象成一个切面对象，对该切面对象进行编程就是AOP。

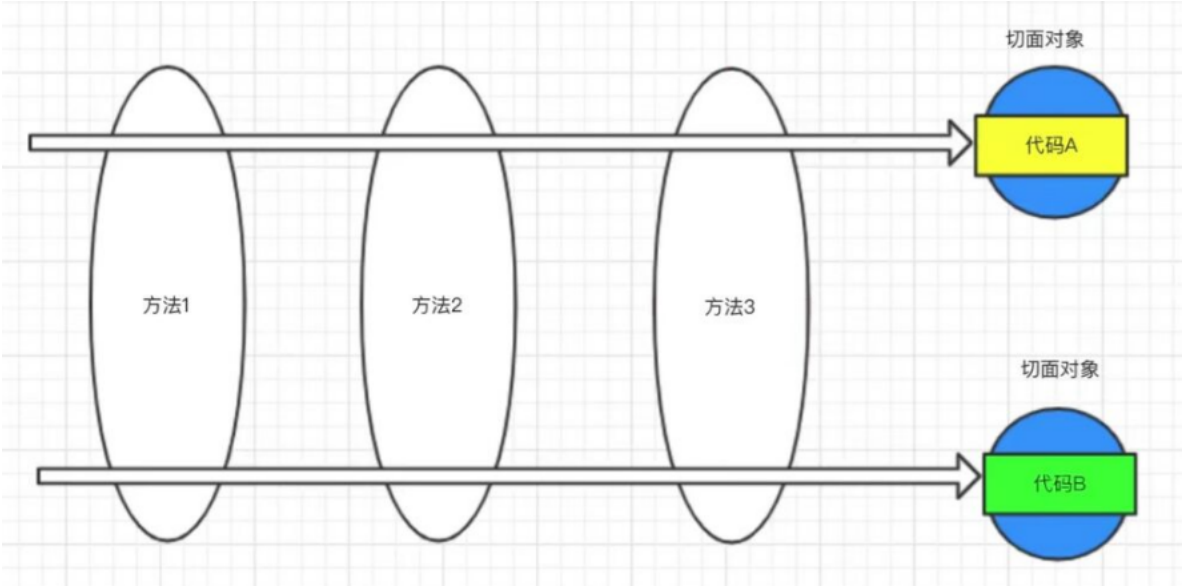
上面介绍太抽象了，用下面的图来说明aop的好处。

比如说我们有好多方法在方法开始前记录日志方法结束后记录日志，而普通的写法就是这样。



但是这样让程序耦合性太高，不方便。。。

而我们如果使用aop技术就可以很好的解决该问题。如下图。



我们将同一个位置给切出来，然后创建切面对象然后写入我们的代码是不是就非常好的解决了该问题，没错这就是aop的好处。

AOP底层实现

aop底层是通过java的代码和java的反射来实现的，所以我们通过写代码来介绍这些技术。

创建一个计算器接口 Cal，定义4个方法。

```
package Proxy3;

public interface Cal {
    public int add(int num1,int num2);
    public int sub(int num1,int num2);
    public int mul(int num1,int num2);
    public int div(int num1,int num2);
}
```

创建接口的实现类 CalImpl

```
package Proxy3.impl;

import Proxy3.Cal;

public class CalImpl implements Cal {
    public int add(int num1, int num2) {
        System.out.println("add方法的参数是["+num1+","+num2+"]");
        int result = num1+num2;
        System.out.println("add方法的结果是"+result);
        return result;
    }

    public int sub(int num1, int num2) {
        System.out.println("sub方法的参数是["+num1+","+num2+"]");
        int result = num1-num2;
        System.out.println("sub方法的结果是"+result);
        return result;
    }
```

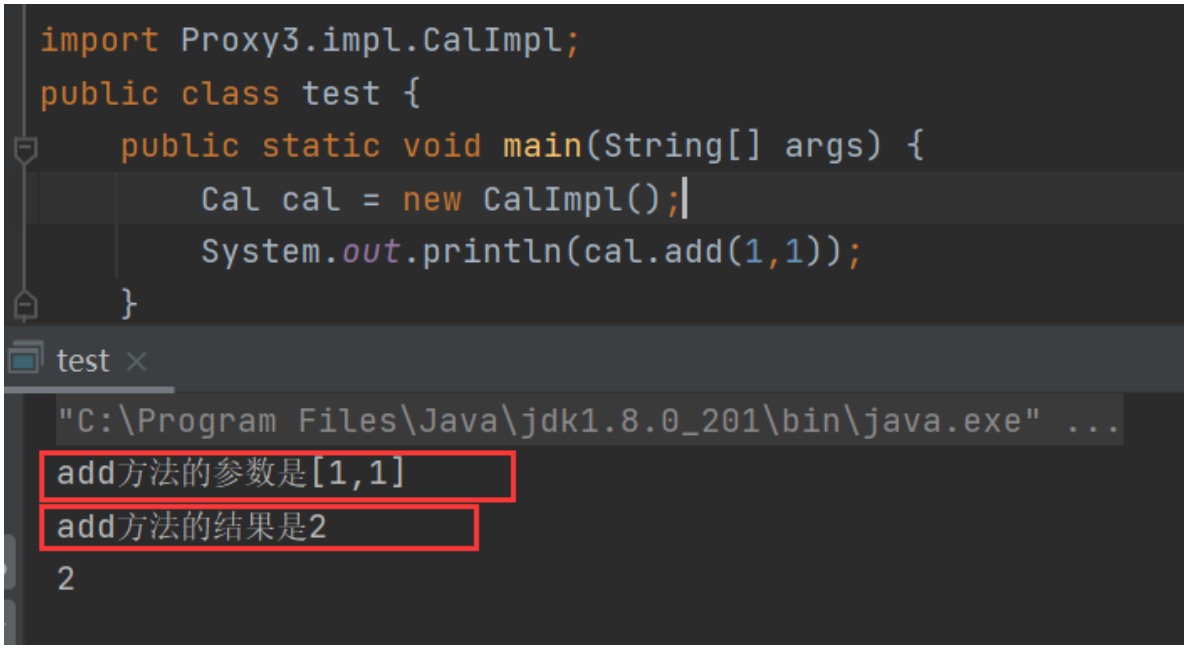
```
    public int mul(int num1, int num2) {
        System.out.println("mul方法的参数是["+num1+","+num2+"]");
        int result = num1*num2;
        System.out.println("mul方法的结果是"+result);
        return result;
    }

    public int div(int num1, int num2) {
        System.out.println("div方法的参数是["+num1+","+num2+"]");
        int result = num1/num2;
        System.out.println("div方法的结果是"+result);
        return result;
    }
}
```

然后创建test类去调用不同的方法。

```
package Proxy3;

import Proxy3.impl.CalImpl;
public class test {
    public static void main(String[] args) {
        Cal cal = new CalImpl();
        System.out.println(cal.add(1,1));
    }
}
```



成功输出了我们的日志信息，不过在上述代码中，日志信息和业务逻辑的耦合性很高，不利于系统的维护，使用 AOP 可以进行优化，如何来实现 AOP?

使用动态代理的方式来实现，给业务代码找一个代理，打印日志信息的工作交个代理来做，这样的话业务代码就只需要关注自身的业务即可。

这里代理就相当于买房子找中介一样，而中介要帮忙代理就需要具备你的功能，这里的代理也一样。需要代码对象具备被代理对象的功能。

要使用代理就需要实现InvocationHandler接口，并且需要重写invoke方法

```
public class MyInvocationHandler implements InvocationHandler {
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        return null;
    }
}
```

并且创建需要代理的对象，也就是给谁代理。

```
public class MyInvocationHandler implements InvocationHandler {
    //接收委托对象
    private Object object = null;
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        return null;
    }
}
```

然后还需要一个代理对象,也就是中介，并且**代理类需要动态生成**

```
public class MyInvocationHandler implements InvocationHandler {
    //接收委托对象
    private Object object = null;

    //返回代理对象
    public Object bind(Object object){//通过Proxy.newProxyInstance创建 而Proxy就是
反射的知识。
        this.object = object;
        return
Proxy.newProxyInstance(object.getClass().getClassLoader(),object.getClass().getI
nterfaces(),this);
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        return null;
    }
}
```

这里我们可以看一看Proxy.newProxyInstance的方法

```
public static Object newProxyInstance(ClassLoader loader,
                                       Class<?>[] interfaces,
                                       InvocationHandler h)
```

第一个参数`ClassLoader loader`加载器是为了将生成的代理类加载到jvm中。

第二个参数`Class<?>[] interfaces`委托类的接口信息，也就是刚刚说的代理类需要具备和被代理类一样的功能。

第上个参数`InvocationHandler h`是指谁来操作这个过程，也就是MyInvocationHandler类。

所以说整体流程就是通过拿到被代理类的接口信息动态的生成代理类，如何通过ClassLoader 加载器加载到jvm内存里面执行。

如何获得加载器?

因为类的加载执行都是通过ClassLoader加载到jvm内存中执行的，也就是运行时类，所以随便通过运行时类都可以拿到加载器。

`object.getClass().getClassLoader()`只不过是不同的加载器而已，这里又可以扩展地ClassLoader的知识（双亲委派）

然后写我们的Cal对象的调用代理类的信息

```
package Proxy3;

import Proxy3.impl.CalImpl;
public class test {
    public static void main(String[] args) {
        Cal cal = new CalImpl();
//        System.out.println(cal.add(1,1));

        //获得代理对象，也就具备被代理对象的全部功能
        MyInvocationHandler myInvocationHandler = new MyInvocationHandler();
        //如何接受，因为代理对象就是安装Cal接口实现的，所以可以直接以Cal对象来接受
        Cal proxy = (Cal) myInvocationHandler.bind(cal);
        //之后就直接代理对象来帮Cal类做事情了
        proxy.add(10,1);//会执行代理类的invoke方法，这里也就是cc链代理的核心了，调用代理对象会执行代理对象的invoke方法
    }
}
```

之后就需要在代理类MyInvocationHandler类里面实现**invoke**方法，

```
package Proxy3;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.Arrays;

public class MyInvocationHandler implements InvocationHandler {
    //接收委托对象
    private Object object = null;

    //返回代理对象
    public Object bind(Object object){
        this.object = object;
        return Proxy.newProxyInstance(object.getClass().getClassLoader(),object.getClass().getInterfaces(),this);
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
```

```
        //实现解耦合  proxy就是代理对象 method就是add方法 args就是参数10和1
        //这里有了对象，有了方法，有了参数,该什么？ 反射
        System.out.println(method.getName()+"方法的参数是："+
Arrays.toString(args));
        Object result = method.invoke(this.object,args);
        //反射：方法.invoke(对象，参数)  让被代理的对象调用add方法传递10和1
        System.out.println(method.getName()+"的结果是"+result);
        return result;
    }
}
```

然后我们删除之前CalImpl类的输入日志信息，运行test类就成功通过代理来实现输出日志功能，并且实现解耦合。

```
import Proxy3.impl.CalImpl;
public class test {
    public static void main(String[] args) {
        Cal cal = new CalImpl();
//        System.out.println(cal.add(1,1));

        //获得代理对象，也就具备被代理对象的全部功能
        MyInvocationHandler myInvocationHandler = new MyInvocationHandler();
        //如何接受，因为代理对象就是安装Cal接口实现的，所以可以直接以Cal对象来接受
        Cal proxy = (Cal) myInvocationHandler.bind(cal);
```

```
test
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
add方法的参数是：[10, 1]
add的结果是11
```

调试一下就知道了MyInvocationHandler类的object是为被代理类的信息也就是CalImpl类，invoke里面的proxy也是。

```
public class MyInvocationHandler implements InvocationHandler {
    //接收委托对象
    private Object object = null;  object: CalImpl@526
    //返回代理对象
    public Object bind(Object object){
        this.object = object;
        return Proxy.newProxyInstance(object.getClass().getClassLoader(),object.getClass().getInterfaces(), h: this);
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {   proxy: "Proxy3.impl.CalImpl@
        //实现解耦合  proxy就是代理对象 method就是add方法 args就是参数10和1
        //这里有了对象，有了方法，有了参数,该什么？ 反射
        + {$Proxy0@528} "Proxy3.impl.CalImpl@2a098129"
        System.out.println(method.getName()+"方法的参数是："+ Arrays.toString(args));   args: Object[2]@533    method: "pu
        Object result = method.invoke(this.object,args);
        //反射：方法.invoke(对象，参数)  让被代理的对象调用add方法传递10和1
```

总结

以上就是aop的底层原理，如果如果看了的应该懂，而且会联系到java的反序列化的知识。