# 1 引言

框架从某种意义上讲是某种应用的半成品，它是由一组组件所构成。对于程序的重用性与所设计的系统的扩展性以达到开发周期的缩减的目的与开发质量的提高等目的，往往是框架一直追求并良好的实现了的。

在软件设计中，最终遵循的还是一个设计理念，就是“高内聚，低耦合”。框架一般是将问题分割成若干子问题进行一一攻破，从而起到易于控制、扩展，易于分配资源的效果。设计过程中，常常引入“层”的概念，及将各个义务分层实现。其间难免会出现耦合，而耦合度过高会降低系统的扩展性和维护性。而框架主要工作在层与层之间，很好的解决了这一问题。在软件设计中有一个概念叫做 IoC，及控制反转，也叫 DI（依赖注入），它主要就是实现层与层之间的松耦合。

面向对象编程在软件设计中无处不在，非常完美的解决了代码重用。但有时候具体的业务贯穿整个系统，而往往这个业务是重复出现的，利用面向对象已不能很好解决。在这里便出现了 AOP（面向切面编程），将其中相同的业务抽取出来进行统一解决。在这里不得不说一下 Spring 框架的强大魅力，Spring 对 IoC 和 AOP 的操作可谓前无古人。

本文主要利用 IoC 和 AOP 的概念，解决层与层之间的依赖关系以及重复业务的处理。

## 1.1 研究背景

上世纪末与本世纪初，J2EE 开始盛行，主要归功于它对中间层概念提出了系统性标准。但事实上，它并没有取得实质性的成功，原因主要是因为不管从其效率、难度还是性能上来讲都不孚众望。

在 J2EE 早期阶段，都是利用 EJB 技术来开发 J2EE 应用的。但是，对于 EJB，其学习成本非常高也难于理解，而且要想应用 EJB 技术也是相当困难的。因为 EJB 强制程序员必须依照它的规范去继续各种不同的接口，这样便会导致代码冗余及相似。此外对于其配置既是纷繁复杂又是味同嚼蜡。对于使用 JNDI 查找对象也是如此。虽然 xdoclet 的应运而生和缓了其中部分的开发工作，但是 EJB 存在的各大问题都造成了对其使用的不方便性。随着 Java 语言的发展，AOP 和 IoC 等技术的逐渐成熟，一种新的 J2EE 解决方案应运而生，即轻量级框架。[1]

## 1.2　研究平台

本文主要是基于 Eclipse　平台，使用 Java 语言编写 IoC 和 AOP的实现程序。

### 1.2.1　Java 语言

Java 是一种面向对象的，由 Sun 公司开发的程序设计语言，具体研发是 James Gosling 及其同事，在上世纪 90 年代末正式推出。Java 的强大之处在于其跨平台性，可在不同操作系统上编写应用软件。Java 语言不同于其他编程语言，其优势主要体现在它具有通用、高效、安全等优点。而且该语言的应用领域也极其广泛。在微型电脑、数据中心、超级计算机以及各种网页应用等都能见到 Java 的身影。虽然 Java 的编程风格与之 C、C++非常接近，但与 C语言不同的是，Java 是完全的面相对象，对于 C++核心的面向对象技术它也是完美的继承了。同时，Java 一改 C中指针的概念，取而代之的是引用的概念。同时也摒弃了 C中运算符重载和多继承等特征。在此基础上，Java 也增加了自己的新特性，就是垃圾回收机制，对于不再引用而又一直在内存中的引用进行回收处理。程序员也从中得益而不用手动进行内存管理。

### 1.2.2　Eclipse

Eclipse　是一个开源的软件开发工具，同时也是功能完备，能进行商用的工业开发平台。主要组成为 Eclipse　项目、工具项目、技术项目，具体是指 Eclipse　Platform ，JDT，CDT，PDE。其中，Eclipse　Platform 是可扩展的集成开发环境；JDT是 Java 开发工具，主要用于 Java 开发；CDT是 C语言开发工具，主要用于 C开发；PDE则是对插件的开发。Eclipse　为构建 IDE和建造块建立坚实的基础。对于 Eclipse　Platform ，它允许第三方工具的无缝对接，从而起到无须辨别具体工具的功能体现在哪里的作用。

# 2　IoC 和 AOP

## 2.1　IoC（控制反转）

IoC，英文全称为 Inversion of Control　，及控制反转，主要用于降低程序间的耦合度。控制反转一般分为两种类型，依赖注入（Dependency Injection　，简称 DI）和依赖查找（Dependency Lookup）。依赖注入应用比较广泛。[2]

### 2.1.1　依赖注入

依赖注入就是容器全权负责组件，给予其回调接口和上下文条件。EJB 和 Apache Avalon 都使用这种方式。如此看来，对于依赖对象的查找以及资源的查找就必须使用容器提供的接口，控制反转也就体现在了回调接口上。容器提供应用代码资源也是通过回调接口的。[3]

### 2.1.2 实现方式

对于依赖注入，主要的实现方式分别为接口注入（Interface Injection ）、Set 方法注入（Setter Injection ）和构造注入（Constructor Injection ）。[4]

#### 2.1.2.1 接口注入

在接口中定义要注入的信息再通过接口来完成此功能就叫接口注入。具体示例如下：

编写一个 IBiz 接口，Dao 层的注入将通过这个接口进行。假设该接口中有一 getDao() 方法，用于获取数据访问层的对象。

```
public interface IBiz{
    /**
    * @param dao 数据访问层对象
    */
    public void getDao(Dao dao);
    }
```

对于想要进行数据库操作的类就必须得实现IBiz 接口，业务逻辑类Biz 实现这个接口 IBiz 。

```
    /*
    * 具体实现IBiz 接口的类，重写了getDao 方法
    */
    public class Biz implements IBusiness{
        private Dao dao;
```

```
        @Override
        public void getDao(Dao dao){
            =dao;
        }
    }
```

只有实现 IBiz 接口才能完成依赖注入。

#### 2.1.2.2 Set 方法注入

Set 方法注入就是在需要属性注入的类中定义一个 Set 方法，并设置注入元素为其参数。

假设业务层 Biz 依赖数据访问层，且持有其属性，需定义一个 Set 方法来接受数据访问层的注入。

```
public class Biz{
    //数据访问层对象的引用
    private Dao dao;
    public void setDao(Dao dao){
        =dao;
    }
    //其他调用数据访问层的方法及其他操作
}
```

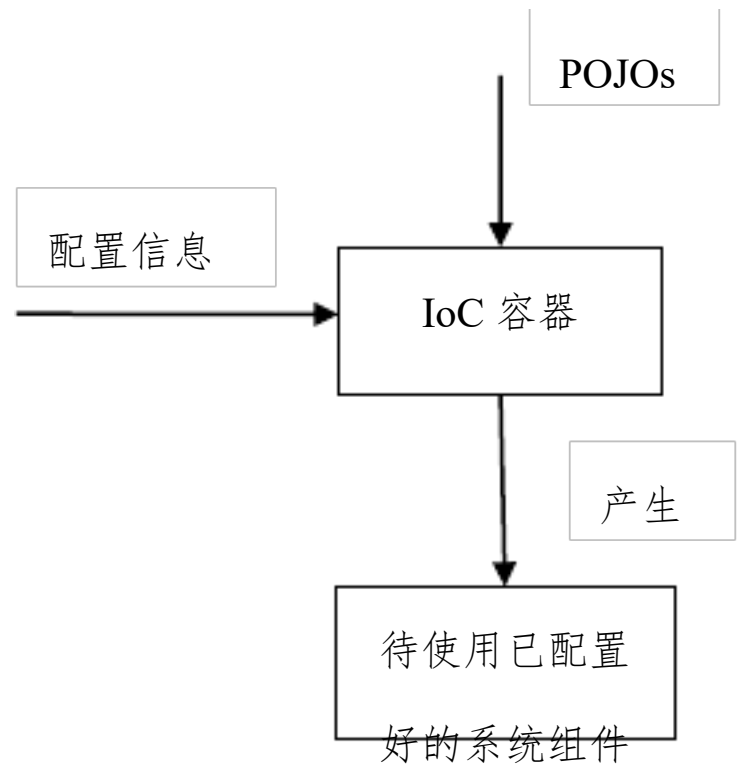
#### 2.1.2.3 构造注入

构造注入就是在需要属性注入的类中提供一个有参构造，其参数就是注入的元素。假设业务层 Biz 依赖数据访问层，且持有其属性，可通过一个构造器来接受数据访问层的注入。

```
public class Biz{
```

```
        private Dao dao;

        public Biz(Dao dao){

            =dao;

        }

            //其他调用数据访问层的方法及其他操作

    }
```

#### 2.1.2.4 IoC 图解




IoC 图解

## 2.2 AOP(面向切面编程)

AOP就是 Aspect Oriented Programming 的缩写，意为：面向切面编程。具体是指它是一种通过预编译和动态代理来实现程序功能的统一维护的技术。[5]AOP是 OOP的延续，是软件开发中的一个热点，是函数式编程的一种衍生范型。AOP技术可以使业务逻辑的各个功能模块耦合度降低，从而达到提高程序的重用性和开发效率。[6]

### 2.2.1 实现技术

对于 AOP技术的实现，主要可通过以下两种方式实现：一种是动态代理，对原有对象的行为通过截取消息，修饰消息，最终执行修饰后的行为；另一种是静态织入，

以特定语法创建切面，以达到编译器在编译期织入切面代码的目的。[7]然而殊途同归，实现 AOP的技术特性却是相同的，分别为：

（1） join point （连接点）：连接点是程序运行过程中具体的执行点，比如它可以是一个方法。对于连接点并不是一个具体的概念，所以在是实现 AOP时并不用去定义它。

（2） point cut （切入点）：对于切入点这个概念，它实质上是一个捕获连接点的一种结构。所以在实现 AOP中，可通过定义一个切入点来拦截相关方法的调用。

（3） advice （通知）：本质上是切入点是执行程序，具体执行切面的逻辑。

（4） aspect （方面）：切点与通知合称切面，虽与面向对象中的类相似，但它更多的表达的是对象间的横向关系。

（5） introduce （引入）：以附加属性方法的形式达到修改对象结构的目的。有的 AOP工具又将其称为 mixin 。

### 2.2.2 主要目的

对于 AOP的主要目的大致可分为事务处理、性能监测、日志记录、安全控制等。[8]

### 2.2.3 主要意图

对逻辑代码中事务处理、性能监测、日志记录、安全控制等的处理从中分离，重新独立到非向导业务方法中。此等做法的好处是在修改这些代码的时候不影响业务逻辑，进一步体现了软件开发的“开闭原则”。[9]

## 2.3 注解（Annotation ）

Annotation( 注解)。对于注解，主要作用大致是监测代码依赖性，创建文档，通过注解甚至可以执行基础的编译检查。[10]注解的编写方式是以“@”开头加上自定义的注解名，通过其参数个数的不同，大致可分为三类：单值注解、标记注解和完整注解。注解只是作为标识存在，一般不会直接影响到程序的语义，通过反射机制我们可以访问到这些元数据，元数据就是用来描述数据的数据。注解的存在级别一共有三种：RUNTIME,CLASS,SOURCE。RUNTIME表示运行时存在，CLASS表示能作用于 class 文件，SOURCE表示只存在源代码中。程序员可在编译时选择代码的存在级别。[11]

### 2.3.1 基本作用

虽然对于注解的作用还没有明确的定义，但是大致可以分为三种：

（1） 编写文档：检查代码中存在的标识注解来生成文档。

（2） 代码分析：以代码中标识的注解对代码进行分析。

（3） 编译检查：检查标识的注解使编译器能执行基础的编译检查。[12]

### 2.3.2 基本的内置注解

（1） @Override：只能用在方法上，用来申明该方法是改写父类的。

（2） @Deprecated：对于弃用的方法添加的注解。当程序员调用这些方法时，在编译时将会显示提示信息。该注解可添加在程序的所有元素上。

（3） @SuperressWarnings：暂时关闭警告信息。[13]

## 3 基于注解的 IoC 和 AOP的实现

就 IoC 而言，主要实现是靠设计模式中的工厂模式，工厂模式负责将大量具有Component注解的类进行实例化，而不必事先知道每次是要实例化哪一个类。换句话说，工厂模式对于具体的new的细节都进行了隐藏和封装。

对于工厂模式的优点主要分为下列四点：

（1） 代码结构清晰，具有良好的封装性。对于对象的创建不是无条件的。假设一个调用者需要一个具体的产品，调用者并不需要知道产品具体是如何被生产的。对于他而言只需要知道产品类名或者产品约束字符串就可以了，从而降低模块耦合度。

（2） 优秀的可扩展性。如果具体需求需要增加产品，不必具体对产品类进行修改，只需适当修改工厂类或者再增加一个工厂就可以了。

（3） 屏蔽产品类。对于具体产品是如何生产的，调用者并不用关心，他的关注点主要在于产品的接口。如果产品接口不发生变化，那么系统上层的模块也不会发生改变。具体的产品的实例化主要由工厂类管理，对于不同的产品对象的生产应取决于不同的工厂类。

（4） 关于工厂方法的经典应用就是解耦合了。处于较高层的模块只需知道实现类的接口，具体实现类无需关心。对于工厂方法，它也遵循迪米特和依赖倒置法则，仅仅依赖实现类的接口；除此之外，工厂方法也遵循里氏替换原则，子类可以随时随地替换父类。[14]

而对于 AOP来说，在扫描系统组件时，如果该组件存在 Interception 注解且已声

明需要拦截的方法。那么在调用存在 Interception 注解的类的方法时，如果该方法已被拦截，则该方法执行前和执行后会进行相应的操作。而实现 AOP技术主要用到了代理模式。

对于使用代理模式，只要有点体现在以下几个方面：

（1） 职责清晰：具体的产品类实现具体的业务，不必去在意非自身所要实现的业务。如需其他业务可通过后期代理完成附加，此种做法的好处是代码简洁清晰。

（2） 高扩展性：对于产品类可能需求不同，但是只要是实现了其接口的，可通过代理手段代理各种产品类，而代理类却不用做任何修改。

（3） 智能化：代理对象可在运行时才去调用具体的代理类，换言之就是代理类可以在运行时才确定代理对象。[15]

## 3.1 注解的编写

在本文中，主要用到的注解有@Component,@Property，@Aspect,@Interception 。@Component注解主要说明该类是一个组件，用于在初始化容器时将其实例创建放入一个 Map中；@Property 注解主要说明该类是组件类的属性，在运行时将注入属性，以便调用该类的方法；@Aspect 注解主要用于说明该类是一个切面类，用来执行在拦截方法执行过程中要处理的操作；@Interception 注解用于说明该类中的方法需要被拦截，可指定要拦截的方法，拦截下来的方法将先进行其他操作，如事务处理。具体注解的代码如下：

@Component注解：

```
public @interface Component {
    /组件在HashTable中的名字，当为空时默认为组件类名的小写

    /组件是否存在单例，默认为存在
    public boolean isSingleton() default true;
}
```

@Property 注解：

```
public @interface Property {
        /需要注入的属性注入的对象的名称，默认为空时则直接获取组件中属性的名称

}
```

@Aspect注解：

```
public @interface Aspect {
```

```
        /切面类的名称

    }
```

@Interception 注解：

```
    public @interface Interception {
        /栏截的方法
        public String[]methods();
    }
```

## 3.2 IoC 实现（对象工厂）

首先，在工厂初始化的时候会创建组件的对象，而对于组件类的存放放在对应的包中，对于读取的包及包的扫描则配置在 XML中。读取 XML获取包名及子包名代码如下：

```
File file=new File(()+xmlName);
DocumentBuilder builder=null;
try {
    builder=().newDocumentBuilder();
        Document doc=(file);
        Element ele=();
        NodeList nls=();
        Element ce=null;
        for (int i = 0; i < (); i++) {
            Node n=(i);
            if(()==){

                ce=(Element)n;
                break;
                }
            }
        }
        packageInfo=();
    } catch (Exception e) {
        () ;
}
```

其次通过包名，获取包中的所有类，而类的存放是以文件形式存在，则需进行文件的扫描。文件扫描代码如下：

```
public void findAndAddClassesInPackageByFile(String packageName,
String physicsPath, final boolean recursive, Set<Class<?>> classes) {
            /获取此包的目录建立一个file
            File dir=new File(physicsPath);
```

```
//
if(!()||!()){
        return;
}
/如果存在就获取包下的所有文件及目录
File[] dirFiles=(new FileFilter() {
        @Override
        public boolean accept(File file) {
        /如果可以循环（即包含子目录）（即编译好的java文件）

        }
});
/循环所有文件
for (File file : dirFiles) {
                /如果是目录则继续扫描
                if(()){

                }else{
                        //
                        String className=().substring(0,().length()-6);
                        //(className);
                        try {
                                /添加到集合中
                                //(packageName+'.'+className);

((().getContextClassLoader().loadClass(packageName+'.'+className));
                        } catch (Exception e) {
                                () ;
                        }
                }
        }
    }
```

获取包中所有类的代码如下：

```
    public Set<Class<?>>getClasses(String packages){
        /存放包中的class
        Set<Class<?>>classes=new LinkedHashSet<Class<?>>();
        /是否迭代循环
        boolean recursive=true;
        /将包名从以  隔开换成以  隔开
        String packageName=packages;

        //(packageToDir);
        /定义一个枚举的集合循环处理该目录下的所有东西
        Enumeration<URL> dirs;
```

```
try {
        dirs=().getContextClassLoader().getResources(packageToDir);
        //
        while(()){
            /获取下一个元素
            URL url=();
            /获取协议名
            String protocal=();
            /如果是文件形式保存在服务器上

                    类型的扫描
                /获取包的物理路劲

                //(physicsPath);
                    /以文件方式扫描整个包下的文件并添加到集合中

    findAndAddClassesInPackageByFile(packageName,physicsPath,recursive,classes);

                    /如果是jar文件，则定义一个jarFile
                    JarFile jar;
                    try {
                        /获取jar
                        jar=((JarURLConnection)()).getJarFile();
                        /从此jar包中获取一个枚举类
                        Enumeration<JarEntry>entries=();
                        /同样进行循环迭代
                        while(()){
                            /获取jar里的一个实体，可以是目录和一些jar包里的其他文
件，如META-INF等文件
                            JarEntry entry=();
                            String name=();
                            /如果以  开头
                            if((0)=='/'){
                                /获取  后的字符串
                                name=(1);
                            }
                            /如果前半部分与定义的包名相同
                            if((packageToDir)){
                                /若以  结尾是一个包

                                if(index!=-1){
                                    /获取包名且把  替换成

                                }
                                /若可迭代下去且是一个包
                                if((index!=-1)||recursive){
```

```
                                        //              获取正真的类名
                                        String className=()+1,()-6);
                                        try {
                                            /添加到classes中

                                        } catch (Exception e) {
                                            () ;
                                    }
                                }
                            }
                        }
                    }
                }catch (Exception e) {
                                    () ;
                                }
                            }
                        }


                    }
                } catch (Exception e) {
                    () ;
                }
                return classes;
            }
```

接着就是对工厂的初始化。具体代码如下：

```
    public void init() {
        /获取配置文件中的包信息
        String packageInfo=().getPackage();
        //(packageInfo);
        /获取包下的所有class
        Set<Class<?>>classes=getClasses(packageInfo);
        //(());
        for (Class<?> cls : classes) {
            /判断是否存在Component注解
            if(()){
                /获取注解对象
                Component component=();
                /获取注解name的属性值
                String comName=();
                try {
                    /判断注解comName是否为空

                        /如果为空将comName设置为类名的小写
                        comName=();
                        comName=(0,1).toLowerCase()+(1);
```

```
                    //(comName);
                }
                //                                                cms中
                (comName, ());
            } catch (Exception e) {
                () ;
            }
        }

    }
}
```

接下来则是在程序运行过程中，对存在@Property 注解的属性进行属性注入，获取其对应的实例对象。具体代码如下：

```
public <T> T getObject(String name,Class<T> cz){
    T target=(T)(name);
    /获取组件时首先要判断该组件是否存在
    if(target==null){
                                    对象不存在
    }

    /如果对象存在，则要判断对象是否需要注入属性，如果需要注入属性，则先注入属性在返回对象
    Field []fields=().getDeclaredFields();
    for (Field field : fields) {
        /判断该属性是否存在Property注解
        if(()){
            /获取属性名
            String fieldName=();
            Property pro=(Property)();
            String ref=();
            /判断ref是否为空
            ref=fieldName;
        }
        /获取set方法的名称

        try {
            Method m=(methodName, ());
            (target, (ref, ()));

        } catch (Exception e) {
            () ;
        }
        //target=(fieldName, cz);
}
```

```
}
if(().isAnnotationPresent()){
        Interception inter=().getAnnotation();
        //
        String methods[]=();
        SpringInterceptor mi=null;
        try {
                mi=(SpringInterceptor)(interceptionName).newInstance();
                (methods);
                (().newInstance());
        } catch(ClassNotFoundException e){
                () ;
        }catch (InstantiationException e) {
                // TODO Auto-generated catch block
                () ;
        } catch (IllegalAccessException e) {
                // TODO Auto-generated catch block
                () ;
        }
        return (T)();
    }
    return target;
}
```

在属性注入的同时会判断该属性是否存在拦截器以及要拦截的方法。接下来则是利用AOP实现拦截器功能。

## AOP实现（拦截器）

对于拦截器，其实就是代理模式的具体实现，首先需实现 InvocationHandler 接口，该接口是代理实例的调用处理程序实现的接口。对于各个代理实例都被联接了具体的处理程序代码。在代理实例调用处理程序时，将对该程序方法编码设置且将其指派到它的 invoke 方法。

以下是实现 InvocationHandler 的抽象类，该类需要一个目标对象及该目标对象需要拦截的方法。具体代码如下：

```
public abstract class SpringInterceptor implements InvocationHandler {
/目标对象
private Object target;
/所要拦截的方法
private String[] methodsName;
public Object getInstance(){
        Object obj=(().getClassLoader(), ().getInterfaces(), this);
        return obj;
```