AxiZ
Academy eXtend IT skill Zero

実践 Spring Framework + MyBatis

Spring Framework – DIコンテナ/AOP 編

ver. 1.2.1

TC. Techno-core Corporation 株式会社テクノコア
http://www.techno-core.jp

はじめに

Spring Frameworkとは?

現在のサーバサイドJava(Webアプリケーション)において、フレームワークによる開発がデファクトスタンダードとなっています。

また、Struts等に代表されるオープンソースフレームワークが採用されるケースがほとんどです。

Spring Frameworkもまた、SpringSourceにより配布されているオープンソースフレームワークです。

現在でもJava Webアプリケーション開発の(国内においては)代表選手である Struts1.xは、プレゼンテーション層をサポートするフレームワークですが、Spring FrameworkではWebアプリ全体(全ての層)をサポートすることが可能となっています。

また、SSH(Struts/Spring Framework/Hibernate)開発のように、他のフレームワークを統合して使用するための仕組みも備えています。

Spring Frameworkの最も核となる機能は、DIコンテナです。

オブジェクト指向プログラミングでは、オブジェクトが他のオブジェクトに依存し、規模が大きなものになるにつれ、この依存関係は膨大なものになっていきます。

DIコンテナが、このオブジェクト間の依存関係を疎にすることにより、モジュールの単純化が実現できます。

また、(大部分を)POJO<Plane Old Java Object>によるアプリケーション構築が可能であり、EJBのような大規模(複雑)かつ重量級のフレームワークと比較して、単体テスト等もし易いSpring Frameworkは「軽量コンテナ」として位置づけられています。

はじめに

DI(Dependency Injection)とは?

DIとは「Dependency Injection」の頭文字で、一般的に「依存性の注入」と訳されています。

では、「依存性の注入」とは何でしょうか。
依存性とは、「オブジェクトが正しく振る舞うために必要とする、外部の要素(情報や処理)」と言い換えることができます。

つまり、依存性の注入とは「必要な依存性はオブジェクト内で生成するのではなく、外部から注入する」という考え方のことです。

もう少し噛み砕くとすると、DIコンテナを利用することで「必要なオブジェクトを必要な時に生成するのではなく、必要なオブジェクトは必要な時には既にそこにある」状態になります。

言葉で説明すると抽象的でイメージしづらいので、実際に実装しながら理解を深めていきましょう。

☆今回の開発環境は、以下の構成で構築します。
Spring Framework3.2.2

開発環境構築

Spring Framework ライブラリ ダウンロード

http://goo.gl/X9ZjM から spring-framework-3.2.2.RELEASE-dist.zip をダウンロードします。

それぞれのzipファイルを解凍すると、以下の構成で展開されます。

【 spring-framework-3.2.2.RELEASE-dist.zip 】

```
spring-framework-3.2.2.RELEASE
  |-- libs
        |-- spring-aop-3.2.2.RELEASE.jar
        |-- spring-aspects-3.2.2.RELEASE.jar
        |-- spring-beans-3.2.2.RELEASE.jar
        |-- spring-context-3.2.2.RELEASE.jar
        |-- spring-core-3.2.2.RELEASE.jar
        |-- spring-expression-3.2.2.RELEASE.jar
        |-- 略...
  |-- docs
  |-- schema
  |-- license.txt
  |-- notice.txt
  |-- readme.txt
```

開発環境構築

開発用プロジェクト作成

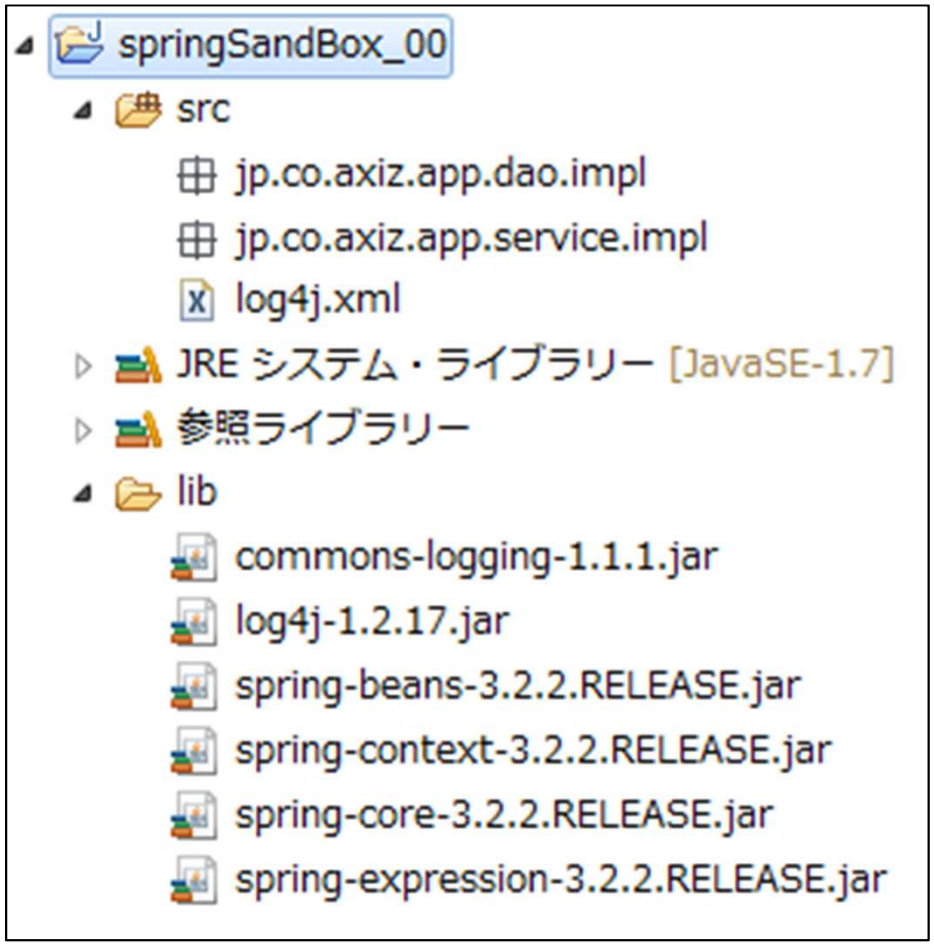
右図のパッケージ構成でプロジェクトを作成します。

・libフォルダには、4ページで解凍したライブラリの中から朱書きの.jarファイルと、「MyBatis – Object/Relational Mapping編」で解凍した **log4j-1.2.17.jarとcommons-logging-1.1.1.jar** を格納してビルド･パスに追加します

・同様に log4j.xmlをコピーして配置し、下記を追記します

【log4j.xml】

```
<!-- Spring関連のログ出力 -->
<logger name="org.springframework">
  <level value="INFO" />
  <appender-ref ref="console" />
</logger>
```



Spring FrameworkによるDI

DIその前に 1/2

まずは、DIを利用しない一般的なJavaのプログラムを実装してみます。
次の2つのクラスを実装し、プロジェクトに配置してください。

【DefaultLoginService.java】

```
package jp.co.axiz.app.service.impl;

import jp.co.axiz.app.dao.impl.DefaultUsrMstDao;

public class DefaultLoginService {

    /** {@link DefaultUsrMstDao}オブジェクトを保持します。 */
    private DefaultUsrMstDao usrMstDao;

    /**
     * 指定したユーザアカウントでのログインを許可するかどうかを判定します。
     * <p>ログインを許可する場合は{@code true}を、許可しない場合は{@code false}を返却します。</p>
     *
     * @param userId ユーザ{@code ID}
     * @param pwd ログインパスワード
     * @return 判定結果
     */
    public boolean isAllowLogin(String userId, String pwd) {

        usrMstDao = new DefaultUsrMstDao();

        if (0 < usrMstDao.getCountByAccount(userId, pwd)) {
            return true;
        }
        return false;
    }
}
```

Spring FrameworkによるDI

DIその前に 1/2

【DefaultUsrMstDao.java】

```
package jp.co.axiz.app.dao.impl;

public class DefaultUsrMstDao {

    /**
     * 指定した条件に合致するレコード数を取得します。
     *
     * @param userId ユーザ{@code ID}
     * @param pwd ログインパスワード
     * @return レコード数
     */
    public int getCountByAccount(String userId, String pwd) {
        // FIXME: 現状は固定で1を返却(仮)
        return 1;
    }
}
```

2つのプログラムは、ログイン処理を行うビジネスサービスです。

ユーザIDとパスワードの組み合わせによりログイン可否を判定します(現状は無条件でログイン成功しますが...)。

Spring FrameworkによるDI

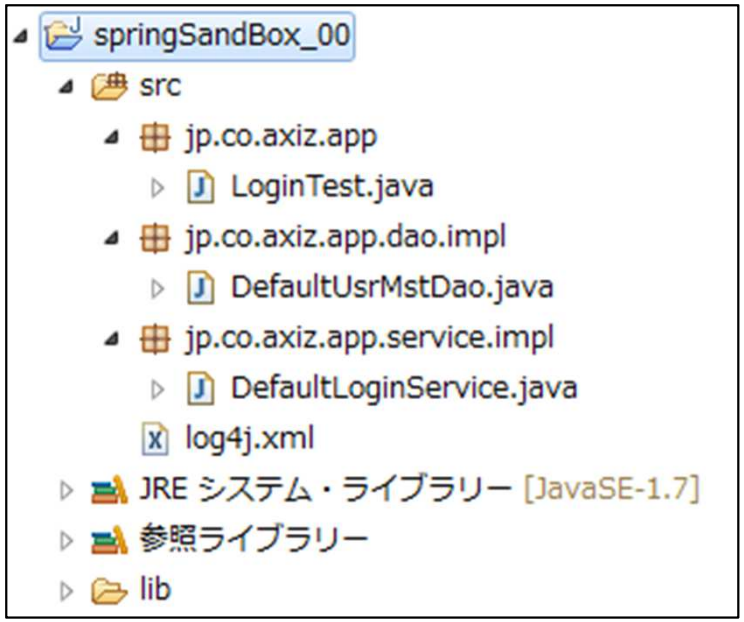
DIその前に 2/2

では、下記のテスト用クラスからログインサービスを呼び出します。

【LoginTest.java】

```
package jp.co.axiz.app;

import jp.co.axiz.app.service.impl.DefaultLoginService;

/**
 * ログイン処理の動作確認のためのテストクラス。
 *
 * @author {@code AxiZ} t.matsumoto
 */
public class LoginTest {
    /**
     * エントリ･ポイントとしての処理を実行します。
     * @param args デフォルト･パラメータ
     */
    public static void main(String[] args) {
        final DefaultLoginService loginService = new DefaultLoginService();
        final boolean isAllowLogin = loginService.isAllowLogin("user", "u_axiz");
        if (isAllowLogin) {
            System.out.println("***** ログインしました.");
        } else {
            System.out.println("***** ログインできません.");
        }
    }
}
```

☆配置後は以下となります。



・・・インスタンスメソッドを呼ぶ前に、それぞれ必要なインスタンスを生成する、ごく一般的なプログラムです。

Spring Frameworkの DIコンテナを利用することで、このプログラムからインスタンス生成(new Xxx();)のコードを排除することができます。

Spring FrameworkによるDI

DIコンテナ設定ファイル

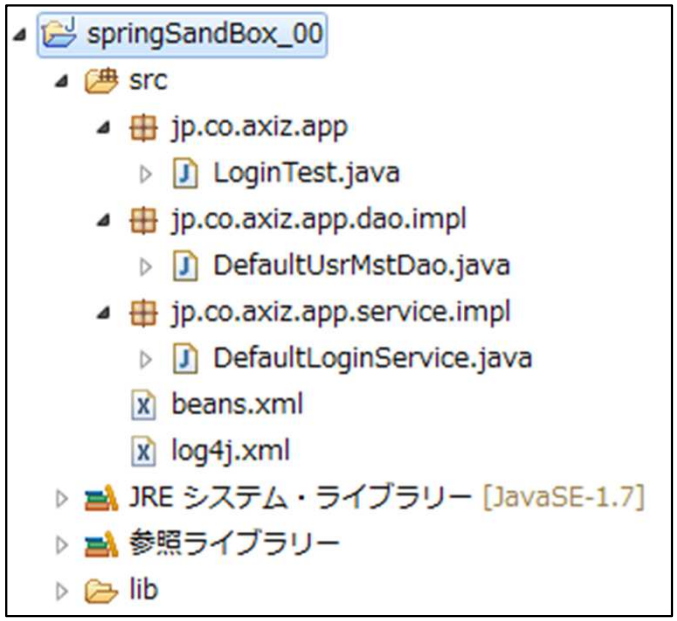
では、Spring FrameworkのDIコンテナを利用するための設定ファイルを作成し、プロジェクトに配置します。

作成するファイルは beans.xml(ファイル名は任意)、配置場所はソースフォルダ配下であれば任意ですが、今回は javaソースフォルダ直下とします。

【beans.xml】

```
<?xml version="1.0" encoding="UTF-8" ?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.2.xsd">

  <!-- ================================================== BusinessService -->

  <!-- ログインビジネスサービス -->
  <bean id="loginService" class="jp.co.axiz.app.service.impl.DefaultLoginService">
    <property name="usrMstDao" ref="usrMstDao" />
  </bean>


  <!-- ================================================== DataAccessObject -->

  <!-- ユーザマスタ(USR_MST)DAO -->
  <bean id="usrMstDao" class="jp.co.axiz.app.dao.impl.DefaultUsrMstDao" />

</beans>
```

☆配置後は以下となります。



今回作成した beans.xml(Spring DIコンテナ設定ファイル)は、「bean定義ファイル」と呼ぶこともあります。

Spring FrameworkによるDI

DIコンテナ 1/5

次に、DefaultLoginService と LoginTest を以下のように修正し、実行してみてください。
Eclipseのコンソール･ビューのログで、beans.xml が読み込まれていることと、loginService, usrMstDao が DI コンテナの管理下に置かれていることが確認できます。

【DefaultLoginService.java】

```
package jp.co.axiz.app.service.impl;

import jp.co.axiz.app.dao.impl.DefaultUsrMstDao;

public class DefaultLoginService {

    /** {@link UsrMstDao}オブジェクトを保持します。 */
    private DefaultUsrMstDao usrMstDao;

    /**
     * 指定した値を{@link #usrMstDao}に設定します。
     * @param usrMstDao {@link #usrMstDao}に設定する値
     */
    public void setUsrMstDao(DefaultUsrMstDao usrMstDao) {
        this.usrMstDao = usrMstDao;
    }

    /**
     * 指定したユーザアカウントでのログインを許可するかどうかを判定します。
     * <p>ログインを許可する場合は{@code true}を、許可しない場合は{@code false}を返却します。</p>
     *
     * @param userId ユーザ{@code ID}
     * @param pwd ログインパスワード
     * @return 判定結果
     */
    public boolean isAllowLogin(String userId, String pwd) {
        // usrMstDao = new DefaultUsrMstDao();

        if (0 < usrMstDao.getCountByAccount(userId, pwd)) {
            return true;
        }
        return false;
    }
}
```

Spring FrameworkによるDI

DIコンテナ 1/5

【LoginTest.java】

```
package jp.co.axiz.app;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import jp.co.axiz.app.service.impl.DefaultLoginService;

/**
 * ログイン処理の動作確認のためのテストクラス。
 *
 * @author {@code AxiZ} t.matsumoto
 */
public class LoginTest {

    /** {@code Bean}定義ファイル名を表す定数。 */
    private static final String BEANS_XML = "beans.xml";

    /**
     * エントリ･ポイントとしての処理を実行します。
     *
     * @param args デフォルト･パラメータ
     */
    public static void main(String[] args) {
        final ApplicationContext appContext = new ClassPathXmlApplicationContext(BEANS_XML);
        final DefaultLoginService loginService = appContext.getBean("loginService", DefaultLoginService.class);
        // final DefaultLoginService loginService = new DefaultLoginService();
        final boolean isAllwLogin = loginService.isAllowLogin("user", "u_axiz");
        if (isAllwLogin) {
            System.out.println("***** ログインしました.");
        } else {
            System.out.println("***** ログインできません.");
        }
    }
}
```

Spring FrameworkによるDI

DIコンテナ 2/5

オブジェクトをDIコンテナに管理させることによって、プログラムからインスタンス生成のコードが排除されました。

これによって「必要なオブジェクトを必要な時に生成するのではなく、必要なオブジェクトは必要な時には既にそこにある」状態は実現できましたが、依然としてオブジェクト同士は密接に依存しています。

そこで、さらにインタフェースを組み合わせて、依存関係をより「疎」にします。

下記の2つのインタフェースを定義し、プロジェクトに配置してください。

【LoginService.java】

```
package jp.co.axiz.app.service;

/**
 * ログイン処理ビジネスサービス･インターフェース。
 *
 * @author {@code AxiZ} t.matsumoto
 */
public interface LoginService {

    /**
     * 指定したユーザアカウントでのログインを許可するかどうかを判定します。<p />
     *
     * ログインを許可する場合は{@code true}を、
     * 許可しない場合は{@code false}を返却します。
     *
     * @param userId ユーザ{@code ID}
     * @param pwd ログインパスワード
     * @return 判定結果
     */
    boolean isAllowLogin(String userId, String pwd);
}
```

【UsrMstDao.java】

```
package jp.co.axiz.app.dao;

/**
 * ユーザマスタ({@code USR_MST})テーブルアクセスのための{@code DAO}インタフェース。
 *
 * @author {@code AxiZ} t.matsumoto
 */
public interface UsrMstDao {

    /**
     * 指定した条件に合致するレコード数を取得します。
     *
     * @param userId ユーザ{@code ID}
     * @param pwd ログインパスワード
     * @return レコード数
     */
    int getCountByAccount(String userId, String pwd);
}
```

Spring FrameworkによるDI

DIコンテナ 3/5

DefaultLoginService を LoginService の実装クラスに、DefaultUsrMstDao を UsrMstDao の実装クラスに、それぞれ修正します。

【DefaultLoginService.java】

```
public class DefaultLoginService implements LoginService {

    . .

    private UsrMstDao usrMstDao;

    . .

    public void setUsrMstDao(UsrMstDao usrMstDao) {
        this.usrMstDao = usrMstDao;
    }
```

【DefaultUsrMstDao.java】

```
public class DefaultUsrMstDao implements UsrMstDao {
```

Spring FrameworkによるDI

DIコンテナ 4/5

では、テスト用クラスも下記のように一部修正して、ログインサービスを呼び出します。

【LoginTest.java】

```
final LoginService loginService = appContext.getBean("loginService", LoginService.class);
```

☆配置後は以下となります。

Spring FrameworkによるDI

DIコンテナ 5/5

このように I/Fを介すことにより、DefaultLoginService と DefaultUsrMstDao のオブジェクト同士の結合は、より「疎」になりました。

これにより beans.xml の class属性の値を変更するだけで LoginService や UsrMstDao の実体を切り替えることが可能になります。

そこにはソースコードの修正は発生しません。

具体的な例を挙げると、Aさんが DefaultLoginService の実装を担当し、Bさんが DefaultUsrMstDao の実装を担当しているとします。

Aさんは実装できたので、ログイン処理のテストをしたいのですが、Bさんの作業はまだまだ時間が掛かりそうです。

よくあるパターンですが、この場合Aさんは usrMstDao の class 属性に UsrMstDao のモック実装クラスを設定すれば、自分の実装担当部分のテストができてしまうということです。

Bさんの実装が完了したら beans.xml を修正するだけでよいため、DefaultLoginService のソースコードには一切手を入れる必要はありません。

☆モック配置後は以下となります。

Spring FrameworkによるDI

DIコンテナ 5/5

では、実際に MockUsrMstDaoクラスを作成してみましょう。
#getCountByAccount(String, String) は固定で 0 を返却してください。

実行後、コンソール･ビューに ***** ログインできません. と出力されれば成功です。

Spring FrameworkによるDI

DIコンテナ Memo 1/2

今回は設定ファイル(beans.xml)によって、DIコンテナによるインジェクションを制御しましたが、アノテーションを使用してインジェクションを設定することも可能です。

今回使用したモジュールで手順を説明すると、

① DI に管理させるクラスにアノテーションを付与します。

```
@Service("loginService") // ビジネス・サービスであることを表す. ()内はID. @Component(" ID ") でも可.
public class DefaultLoginService implements LoginService { // 略. }

@Repository("usrMstDao") // データ・アクセス関連オブジェクトであることを表す. ()内はID. @Component(" ID ") でも可.
public class DefaultUsrMstDao implements UsrMstDao { // 略. }
```

② インジェクションさせたいプロパティにアノテーション@Autowired を付与します。

```
public class DefaultLoginService implements LoginService {
    // プロパティ usrMstDao に UsrMstDaoインタフェースの実装オブジェクトをインジェクションさせる.
    // @Qualifier(" ID ") を併用すれば、同インタフェース実装オブジェクトが複数存在しても ID による切り替えが可能.
    @Autowired
    private UsrMstDao usrMstDao;
    // セッターメソッドは不要.
}
```

③ beans.xml の <beans ...></beans> 内に **<context:component-scan base-package="jp.co.axiz.app" />** を定義します。

これにより、jp.co.axiz.appパッケージとそのサブパッケージ内のクラスがスキャンされ、付与されているアノテーションが有効になります。

Spring FrameworkによるDI

DIコンテナ Memo 2/2

前ページの ①～③ により、beans.xml に定義していた下記の記述が不要になります。

```
<bean id="loginService" class="jp.co.axiz.app.service.impl.DefaultLoginService">
  <property name="usrMstDao" ref="usrMstDao" />
</bean>
<bean id="usrMstDao" class="jp.co.axiz.app.dao.impl.DefaultUsrMstDao" />
```

また、**<context:component-scan />**タグを使用するため、beans.xml 先頭のスキーマ定義に追加が必要となります。

```
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation=" 略...
                    http://www.springframework.org/schema/context
                    http://www.springframework.org/schema/context/spring-context-3.2.xsd"
```

Spring Framework の DIコンテナは、コンポーネントを(デフォルトでは)Singletonオブジェクト(コンテナ内で唯一)で管理します。また、以下の設定により明示的に指定することもできます。

【コンフィグレーション･ベース】<bean id=" ... " class=" ... " **scope="prototype"** />
(scope要素なし、または scope="singleton"の場合は、Singletonとして管理する)

【アノテーション･ベース】**@Scope("prototype")**
(@Scopeアノテーションなし、または @Scope("singleton")の場合は、Singletonとして管理する)

上記のように **prototype** を指定した場合、DIコンテナがどのような振る舞いをするか検証してみましょう。

Spring FrameworkによるAOP

AOP(アスペクト指向プログラミング)とは?

AOPとは「Aspect Oriented Programming」の頭文字で、「アスペクト指向プログラミング」と呼ばれます。

これは、オブジェクト指向プログラミングにおいて「オブジェクトと機能の分離」を重視した概念のことです。

オブジェクト指向では、あるオブジェクトに必要な機能は、そのオブジェクト内に含めます。
しかしその結果、多数のクラスに同じ機能を実装しないといけないケースが発生してしまいます。
例えば「デバッグ用のログ出力」や「ログイン状態のチェック」等が挙げられます。

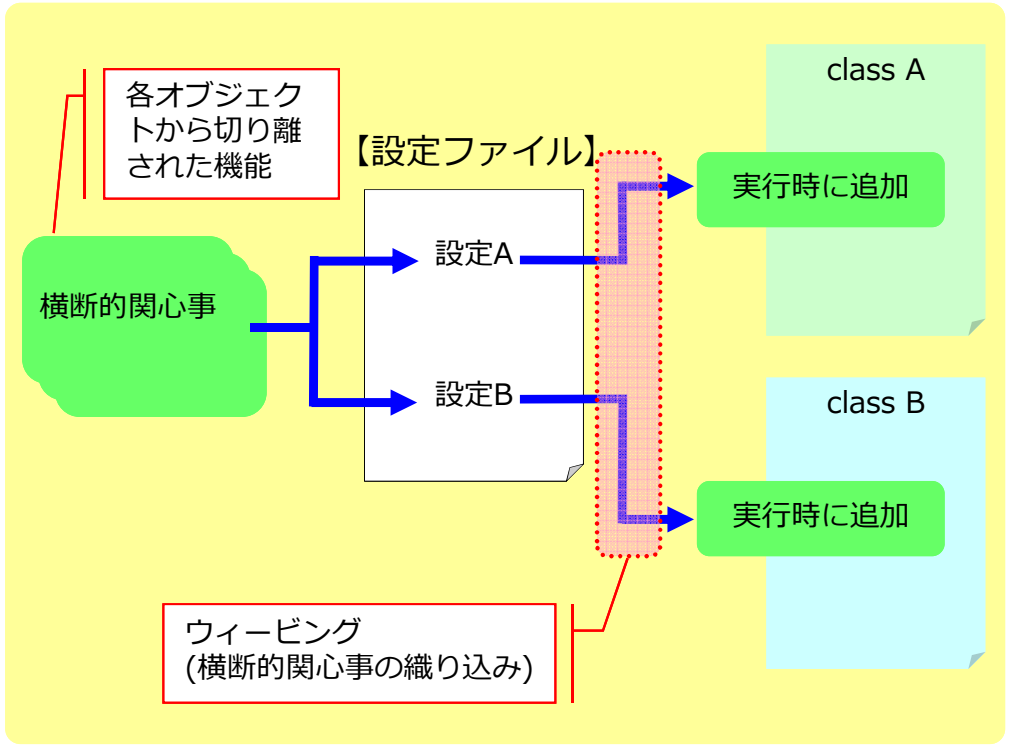
デバッグログは実装中には必要ですが、リリース時には不要なコードです。

ログ出力が不要になったら、そのコードを除去して再ビルドするといった手間が発生します。

「あちこちのオブジェクトで同じ機能が必要」なのであれば、その機能は別オブジェクトとして切り離し、外部から組み込んでしまえばいいといった考え方が「AOP(アスペクト指向プログラミング)」です。

この、「外部から組み込む機能」は「横断的関心事」と呼ばれます。

こうして横断的関心事を切り離したクラスは、メンテナンス性も向上します。



Spring FrameworkによるAOP

AOPライブラリ ダウンロード

では、実際にAOPでログ出力処理をウィービングするプログラムを実装します。
今回は AspectJ を利用します。

http://goo.gl/385mB から aopalliance.zip をダウンロードします。
http://goo.gl/3Wu30 から aspectjweaver-1.6.12.jar をダウンロードします。

aopalliance.zipファイルを解凍すると、以下の構成で展開されます。
また、Spring FrameworkのAOPライブラリも使用します。

【aopalliance.zip】

```
aopalliance.jar
  |-- aopalliance.jar
  |-- javadoc
        |-- 略...
```

【spring-framework-3.2.2.RELEASE】

```
spring-framework-3.2.2.RELEASE
  |-- libs
        |-- spring-aop-3.2.2.RELEASE.jar
  |-- 略...
```

上記朱書きの.jarファイルと、ダウンロードした **aspectjweaver-1.6.12.jar** を libフォルダに格納して、ビルド･パスに追加します。

Spring FrameworkによるAOP

AOP(アスペクト指向プログラミング) 1/2

jp.co.axiz.aopパッケージを作成し、右の TraceAdvice.javaを配置します。
また、 beans.xmlには AOP用の設定を追加します。

【beans.xml】

```
<?xml version="1.0" encoding="UTF-8" ?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-3.2.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop-3.2.xsd">

  <context:component-scan base-package="jp.co.axiz.app"/>

 <!-- ========================================================== AOP -->

  <!-- トレースログ用Advice -->
  <bean id="traceAdvice" class="jp.co.axiz.aop.TraceAdvice" />

  <!-- AOP Setting -->
  <aop:config>
    <aop:aspect id="traceAspect" ref="traceAdvice">
      <aop:pointcut id="tracePointcut" expression="execution(* isAllowLogin(..))" />
      <aop:around method="traceAround" pointcut-ref="tracePointcut" />
    </aop:aspect>
  </aop:config>
```

Spring FrameworkによるAOP

AOP(アスペクト指向プログラミング) 1/2

【TraceAdvice.java】

```
package jp.co.axiz.aop;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.aspectj.lang.ProceedingJoinPoint;

/**
 * トレースログ用{@code Advice}クラス。
 *
 * @author {@code AxiZ} t.matsumoto
 */
public class TraceAdvice {

    /** ログ出力に使用する{@link Log}を表す定数。 */
    private static final Log LOG = LogFactory.getLog(TraceAdvice.class);

    /**
     * ターゲットの処理時間(ミリ秒)トレースログを出力します。
     *
     * @param joinPoint 実行されるターゲット情報
     * @return 実行されたターゲットの戻り値
     * @throws Throwable 実行時例外が発生した場合
     */
    public Object traceAround(ProceedingJoinPoint joinPoint) throws Throwable {

        final String targetName = getTargetName(joinPoint);
        LOG.debug("***** [ " + targetName + " ]  start.");
        final long start = System.currentTimeMillis();
```

Spring FrameworkによるAOP

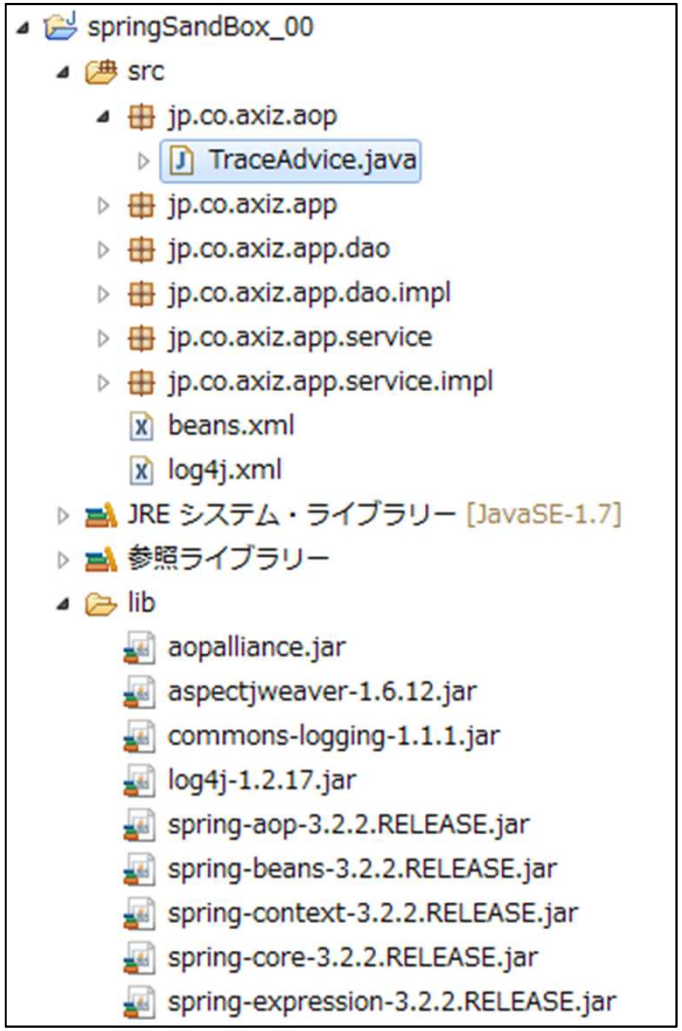
AOP(アスペクト指向プログラミング) 1/2

```
        // ターゲットの処理を実行.
        final Object obj = joinPoint.proceed();

        final Long processMsec = new Long(System.currentTimeMillis() - start);
        LOG.debug("***** " + String.format("[ %s ] %6d msec.", new Object[] { targetName, processMsec }));
        LOG.debug("***** [ " + targetName + " ] finish.");

        return obj;
    }

    /**
     * ターゲット名を取得する.
     *
     * @param joinPoint 実行されるターゲット情報
     * @return ターゲット名({@code ClassName#MethodName})
     */
    private String getTargetName(ProceedingJoinPoint joinPoint) {
        final String className  = joinPoint.getTarget().getClass().getName();
        final String methodName = joinPoint.getSignature().getName();
        return className + "#" + methodName;
    }
}
```

Spring FrameworkによるAOP

AOP(アスペクト指向プログラミング) 2/2

☆TraceAdvice.java配置後は
以下となります。



まずは、LoginTestクラスを実行してみてください。

Eclipseのコンソール･ビューに
DefaultLoginService#isAllowLoginメソッドのトレースログ(開始/実行時間(msec.)/終了)が出力されているはずです。

今回手を入れたリソースは、TraceAdviceクラスの新規追加とbeans.xmlへの設定追加のみです。
このように、既存の DefaultLoginServiceクラスを修正することなく、#isAllowLoginメソッドのトレースログを出力することができました。

つまり、beans.xmlに定義した設定に基づいて、ログ出力という横断的関心事を #isAllowLoginメソッドにウィービング(織り込み)しました。

これが「AOP(アスペクト指向プログラミング)」という考え方です。

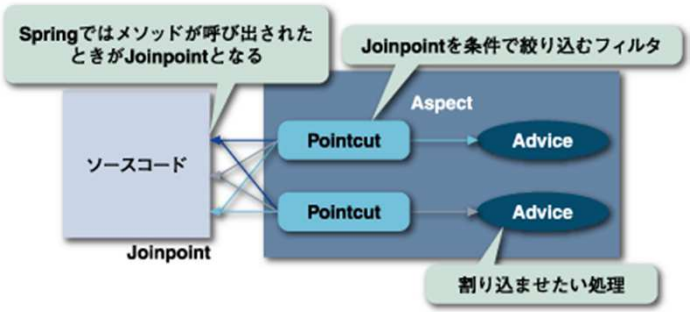
Spring FrameworkによるAOP

AOP用語解説 1/2

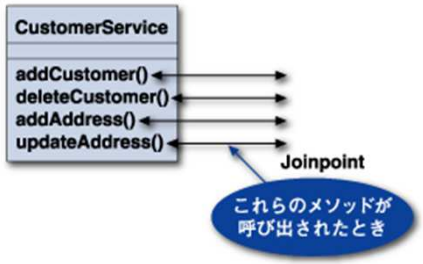
ここでは、AOPの代表的な用語を説明します。

Aspect アスペクト	横断的関心事が持つ処理と、いつ処理を適用するかをまとめたもの。 つまり、以下のAdviceとPointcutをまとめたもの。
Joinpoint ジョインポイント	Advice(処理)を割り込ませることが可能なポイントのこと。
Advice アドバイス	Joinpointで実行される処理。 例えば、今回のログ出力などの処理はAdviceと呼ばれ、Joinpointで実行される。
Pointcut ポイントカット	Adviceを適用したいJoinpointを正規表現などを用いた条件で絞り込むための フィルタ。

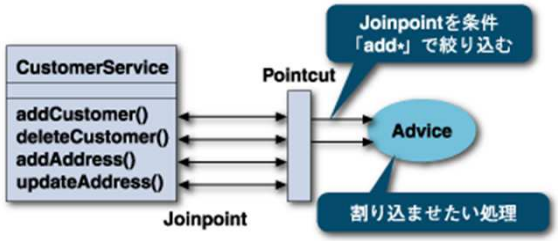
【図1】 関係図



【図2】
Joinpoint



【図3】
Pointcut



Spring FrameworkによるAOP

AOP用語解説 2/2

ここでは、AOPのAdviceタイプを説明します。

Around Advice	Joinpointの前後に実行するタイプのAdvice(今回使用しました) <aop:around ... />
Before Advice	Joinpointの前に実行するタイプのAdvice <aop:before ... />
After Advice	Joinpointの後に実行するタイプのAdvice <aop:after ... />
After-Returning Advice	Joinpointの対象メソッドが戻り値を返却した際に実行するタイプのAdvice <aop:after-returning ... />
After-Throwing Advice	Joinpointの対象メソッドが例外をスローした際に実行するタイプのAdvice <aop:after-throwing ... />

では、次に今回実装したソースコードを解説します。

Spring FrameworkによるAOP

Adviceクラス

【TraceAdvice.java】

```
 8 /**
 9  * トレースログ用{@code Advice}クラス。
10  *
11  * @author {@code AxiZ} t.matsumoto
12  */
13 public class TraceAdvice {
14
15     /** ログ出力に使用する{@link Log}を表す定数。 */
16     private static final Log LOG = LogFactory.getLog(TraceAdvice.class);
17
18     /**
19      * ターゲットの処理時間(ミリ秒)のトレースログを出力します。
20      *
21      * @param joinPoint 実行されるターゲット情報
22      * @return 実行されたターゲットの戻り値
23      * @throws Throwable 実行時例外が発生した場合
24      */
25     public Object traceAround(ProceedingJoinPoint joinPoint) throws Throwable {
26
27         final String targetName = getTargetName(joinPoint);
28         LOG.debug("***** [ " + targetName + " ]  start.");
29         final long start = System.currentTimeMillis();
30
31         // ターゲットの処理を実行.
32         final Object obj = joinPoint.proceed();
33
34         final Long processMsec = new Long(System.currentTimeMillis() - start);
35         LOG.debug("***** " +
               String.format("[ %s ] %6d msec.", new Object[] { targetName, processMsec }));
36         LOG.debug("***** [ " + targetName + " ] finish.");
37
38         return obj;
39     }
40
       // 以下省略...
```

クラス名は任意です。

今回の#traceAroundメソッドは、Around Adviceタイプとして実行されることを想定しています。

25行目: Around Adviceタイプで動作させるためのメソッドシグネチャです(メソッド名は任意)。

32行目: ジョインポイントのターゲットメソッドを実行し、戻り値を受け取っています。

38行目: ジョインポイントのターゲットメソッドの戻り値を返却します。

Spring FrameworkによるAOP

AOP設定

【beans.xml】

```
3:<beans xmlns="http://www.springframework.org/schema/beans"
4:       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5:       xmlns:context="http://www.springframework.org/schema/context"
6:       xmlns:aop="http://www.springframework.org/schema/aop"
7:       xsi:schemaLocation="http://www.springframework.org/schema/beans
8:                           http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
9:                           http://www.springframework.org/schema/context
10:                           http://www.springframework.org/schema/context/spring-context-3.2.xsd
11:                           http://www.springframework.org/schema/aop
12:                           http://www.springframework.org/schema/aop/spring-aop-3.2.xsd">
13:
14:  <context:component-scan base-package="jp.co.axiz.app"/>
15:
16:  <!-- ========================================================== AOP -->
17:
18:  <!-- トレースログ用Advice -->
19:  <bean id="traceAdvice" class="jp.co.axiz.aop.TraceAdvice" />
20:
21:  <!-- AOP Setting -->
22:  <aop:config>
23:    <aop:aspect id="traceAspect" ref="traceAdvice">
24:      <aop:pointcut id="tracePointcut" expression="execution(* isAllowLogin(..))" />
25:      <aop:around method="traceAround" pointcut-ref="tracePointcut" />
26:    </aop:aspect>
27:  </aop:config>
```

24行目、<aop:pointcut>のexpression属性でポイントカットを設定していますが、この execution() は AspectJ で定義されているポイントカットを使用しています。

execution() に指定した値により、ジョインポイントのターゲットメソッドをフィルタリングします。

6行目: AOP用スキーマ定義の追加

11行目: 同上

12行目: 同上

19行目: 作成したTraceAdviceクラスをDIコンテナ管理下に登録します。

23行目: ref属性でTraceAdviceクラスを参照し、アドバイスを定義します。

24行目: ポイントカットを定義します。

25行目: Adviceタイプを定義します(P.26参照)。
実行処理は #traceAround メソッドとし、実行するポイントカットは tracePointcutとして設定します。

Spring FrameworkによるAOP

expression属性に指定するexecution()の書式

execution() は、以下の書式で記述します。

execution(アクセス修飾子 戻り値の型 パッケージ名.クラス名.メソッド名(仮引数の型...) *throws* 例外の型)

ただし、「戻り値の型」「メソッド名」「仮引数の型」以外はオプション扱いのため、省略可能です。
また、以下のワイルドカードを使用してフィルタリングすることもできます。

- `*` ＝ `.` を含まない任意の文字列
- `..` ＝ `.` を含む任意の文字列

【書式例】

・全てのメソッド
```
execution(* *(..))
```

・getで始まる全てのメソッド
```
execution(* get*(..))
```

・jp.co.axiz.appパッケージとサブパッケージにあるすべてのクラスのすべてのpublicメソッド
```
execution(public * jp.co.axiz.app..*.*(..))
```

また、 expression属性では論理演算子(&&, ||, !)を使用した設定も可能です。

【書式例】

・getで始まる全てのメソッド or setで始まる全てのメソッドを、ジョインポイントのターゲットに指定
```
expression="execution(* get*(..)) || execution(* set*(..)) "
```