Spring フレームワーク

8. トランザクション管理 (Ver 1.2.7)

8.1 Spring のトランザクション抽象化

Spring では一貫したトランザクション管理の抽象化が提供されている。この抽象化は Spring で提供している抽象化の中でも最も重要なものの 1 つであり、下記のような利点が得られる。

- JTA,JDBC,Hibernate,iBATIS データベースレイヤや JDO のような異なるトランザクション API 全てに対し、一貫したプログラミングモデルを提供する。
- これらの多くのトランザクション API よりも単純で使いやすいプログラミング的トランザクション管理の API を提供する
- Spring のデータアクセス抽象化との統合
- Spring の宣言的トランザクション管理をサポートする

伝統的に、J2EE 開発者にはトランザクション管理を行うのに 2 つの選択肢、グローバルトランザクションを使うか、ローカルトランザクションを使うか、がある。グローバルトランザクションは JTA を使ってアプリケーションサーバによって管理される。ローカルトランザクションはリソースに依存する。例えば、あるトランザクションが JDBC コネクションに関連づけられているとする。この選択には深い意味合いが込められている。グローバルトランザクションには複数のトランザクションリソースに対して実行できるようにする機能がある。(注目されるのは、ほとんどのアプリケーションは単一のトランザクションリソースを用いているということだ。)ローカルトランザクションでは、アプリケーションサーバではトランザクション管理を行わないので複数のリソース間を跨いで正確さを保証することができないのだ。

グローバルトランザクションは重要度が低下している。コードは JTA という(一部にはその例外モデルのために)厄介な API を使う必要がある。さらに、JTA UserTransaction は通常 JNDI から取得する必要がある。これは、JTA を使うのに、JNDI と JTA の両方を一緒に使う必要がある、ということだ。JTA が通常はアプリケーションサーバ環境でしか使えないので、グローバルトランザクションはアプリケーションコードの再利用性を制限するのは明らかだ。

グローバルトランザクションを用いる好ましい方法は、EJB CMT(コンテナ管理トランザクション)つまり、宣言的トランザクション管理の形式(プログラマティックトランザクション管理と区別して)から使う方法だ。EJB CMT はトランザクション関連の JNDI ルックアップを使わなくていいようにする。-- しかしながら、もちろん EJB そのものは JNDI を使わないといけないが。つまり全部ではないが、トランザクションを制御するための Java コードをほとんど書かなくていいようになる。この重要度の低下は、CMT が(明らかに) JTA やアプリケーションサーバ環境と結びついているからだ。しかも、もしビジネスロジックを EJB、あるいは少なくともトランザクションに関係する EJB ファサードでしか実装しないと決めた場合にしか使えない。EJB 周りのネガティブファクタが一般にはとても大きいので、宣言的トランザクション管理という代案がある場合は、あまり魅力的な案にはなりえないのだ。

ローカルトランザクションを使うのはこれよりは簡単かもしれないが、これとは別の面で重大な欠点がある。ローカルトランザクションは、複数のトランザクションリソースを跨いだ処理ができないので、プログラミングモデルに介入しがちだ。例えば、JDBC コネクションを使っているトランザクションを管理するコードはグローバル JTA トランザクションと一緒に使うことができないのだ。

Spring では、これらの問題が解決される。アプリケーション開発者が任意の環境で一貫したプログラミングモデルを使うことができるようにする。コードを 1 回書けば、異なる環境で異なるトランザクション管理ストラテジの利便性が得られるのだ。Spring では、宣言的トランザクション管理とプログラマティックトランザクション管理の両方が容易されている。ほとんどのユーザにとっては、宣言的トランザクション管理は好ましいものであり、ほとんどのケースでお勧めできるものなのだ。

プログラマティックトランザクション管理と共に、開発者は Spring のトランザクション抽象化を使うことができる。これは、下のレイヤにある任意のトランザクションインフラ上で走らせることができるというものだ。好ましい宣言的モデルがあれば、開発者は、トランザクション管理に関するコードをほとんど、あるいは全く書く必要が通常なくなり、したがって Spring あるいはその他のトランザクション API に依存しなくなるのだ。

8.2 トランザクションストラテジ

Spring のトランザクション抽象化の鍵は、トランザクションストラテジに関する考え方だ。

これは、以下に示した org.springframework.transaction.PlatformTransactionManager インタフェースに取り入れられている。

```
public interface PlatformTransactionManager {

    TransactionStatus getTransaction(TransactionDefinition definition)
        throws TransactionException;

    void commit(TransactionStatus status) throws TransactionException;

    void rollback(TransactionStatus status) throws TransactionException;
}
```

これは、プログラマティカルに利用することもできるが、本来は SPI インタフェース。Spring の哲学に合致してインタフェースであるという点に注意してほしい。したがって、必要であれば、簡単にモックにするか、スタブを用意することができる。JNDI のようなルックアップストラテジに結びつけられてもいない。PlatformTransactionManager の実装は他の Spring IoC コンテナのオブジェクトと同じように実装されている。これは、たとえ JTA と一緒に動かす場合でさえも、それだけで有益な抽象化ができるという利点がある。つまり、トランザクションコードを JTA だと難しいものでももっと簡単にテストすることができるのだ。

Spring の哲学どおりに、TransactionException は未チェックだ。トランザクションインフラでの失敗は、ほとんどが致命的なものだ。アプリケーションコードがこの失敗から回復できるような稀なケースでは、アプリケーション開発者は TransactionException をキャッチして処理を行うかを決めることもできる。

getTransaction()メソッドは、TransactionDefinition パラメータにしたがって TransactionStatus オブジェクトを返す。この返される TransactionStatus は新しいトランザクションかあるいは既存のトランザクションを表している(もし、カレント呼び出しスタックにマッチするトランザクションが存在すれば)。

J2EE トランザクションコンテキストでは、TransactionStatus は処理スレッドと関連づけられている。

TransactionDefinition インタフェースは、下記のように指定されている。

- トランザクションの隔離:そのトランザクションが他のトランザクションから隔離されている程度。例えば、そのトランザクションは他のトランザクションからまだコミットしていない書き込みを参照することができるかどうか。
- トランザクションの伝播:通常トランザクションのスコープで実行される全てのコードは、そのトランザクション内で走る。しかしながら、トランザクションコンテキストが既に存在しているときに、トランザクションメソッドが実行されると、指定された振る舞いが実行されることがある。例えば、既存のトランザクション中で単純に走らせる(最も一般的なケース)、あるいは既存のトランザクションを一時的に停止し、新しいトランザクションを生成する。Spring では、EJB CMT でよく知られているトランザクションの伝播オプションを用意している。
- トランザクションのタイムアウト:そのトランザクションがタイムアウト(して下位レイヤのトランザクションインフラによって自動的にロールバック)するまでどれだけ長く実行されるか。
- リードオンリー状態:リードオンリートランザクションではデータは変更されない。リードオンリートランザクションは(Hibernate を使っているときのように)場合によっては最適化が有効になる。

これらのセッティングは標準の概念を反映する。必要であれば、トランザクションの隔離レベルや、その他の中核となるトランザクションの概念について議論しているリソースを参照してほしい。これらの中核となる概念を理解することは、Spring やその他のトランザクション管理ソリューションを使う上で必要なものなのだ。

TransactionStatus インタフェースはトランザクションコードで、単純なトランザクションの実行やトランザクションの状態問い合わせを制御するための簡単な方法だ。この概念は、全てのトランザクション API に共通しているように、よく知っているべきだろう。

```
public interface TransactionStatus {

    boolean isNewTransaction();

    void setRollbackOnly();

    boolean isRollbackOnly();
}
```

Spring のトランザクション管理がどれだけ使われようとも、PlatformTransactionManager の実装を定義することは必要不可欠だ。Spring のうまいやり方では、この重要な定義は Inversion of Control を使って作られている。

PlatformTransactionManager の実装は、JDBC,JTA,Hibernate など、それが動作する環境に関して知っている必要がある。

Spring の jPetStore サンプルアプリケーションの dataAccessContext-local.xml から抜き出した下記の例では、ローカルの PlatformTransactionManager の実装がどのように定義されるかを示したものだ。これは、JDBC と一緒に動作する。

我々は、JDBC の DataSource を定義して、その DataSource へのリファレンスを与えて Spring の DataSourceTransactionManager を使わなければならない。

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" ¥
    destroy-method="close">
    <property name="driverClassName" value="${jdbc.driverClassName}" />
    <property name="url" value="${jdbc.url}" />
    <property name="username" value="${jdbc.username}" />
    <property name="password" value="${jdbc.password}" />
</bean>
```

PlatformTransactionManager の定義はこのようになる。

```
<bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
</bean>
```

もし JTA を使うのであれば、同じサンプルアプリケーションの dataAccessContext-jta.xml ファイルのように、JNDI から取得した DataSource コンテナと、JtaTransactionManager の実装を使わなければならない。この JtaTransactionManager は DataSource や、他の特定のリソースについて知っている必要はなく、コンテナのグローバルトランザクション管理のように使う。

```
<bean id="dataSource" ¥
    class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/jpetstore" />
</bean>

<bean id="txManager" ¥
    class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

Hibernate のローカルトランザクションは、Spring の PetClinic サンプルアプリケーションから抜き出した以下の例のように、簡単に使うことができる。

この場合、Hibernate の LocalSessionFactory を定義する必要がある。このアプリケーションコードは、Hibernate セッションを取得するのに利用する。

DataSource ビーン定義は上述した例とよく似たものなので、ここでは割愛する(もし、それが DataSource コンテナであれば、コンテナではなく、トランザクションを管理する Spring のように非トランザクションになるはずだ)。

このケースにおいてこの「txManager」ビーンは HibernateTransactionManager クラスだ。DataSourceTransactionManager が DataSource への参照を必要とするのと同様、この HibernateTransactionManager は SessionFactory への参照を必要とする。

```
<bean id="sessionFactory" ¥
    class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="mappingResources">
      <list>
<value>org/springframework/samples/petclinic/hibernate/petclinic.hbm.xml</val
ue>
      </list>
    </property>
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">${hibernate.dialect}</prop>
        </props>
    </property>
</bean>

<bean id="txManager"
    class="org.springframework.orm.hibernate.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

Hibernate や JTA を使えば、JDBC や他のリソースストラテジと同じように簡単に **JtaTransactionManager を使うことができる。**

```
<bean id="txManager" ¥
    class="org.springframework.transaction.jta.JtaTransactionManager" />
```

これはグローバルトランザクションであり、任意のトランザクションリソースを集めることができるので、これが任意のリソース用 JTA を設定するのと全く同じことである、という点に注意してほしい。

この全ての場合において、アプリケーションコードは全く変更する必要はない。単に設定を変更するだけで、たとえ、その変更がローカルトランザクションからリモートトランザクションへの変更であったり、あるいはその逆であってもトランザクションの管理方法を変更することができる。

8.3 トランザクションでのリソース同期

異なるトランザクションマネージャがどういう風に生成されるか、また、トランザクションと同期させないといけない関連するリソースとどうやってリンクさせるか(つまり、DataSourceTransactionManager を JDBC の DataSource に、HibernateTransactionManager を Hibernate の SessionFactory に、など)について、今明らかにすべきだ。しかしながら、アプリケーションコードが直接あるいは間接的に永続性 API(JDBC、Hibernate、JDO など)を使って、どうやってこれらのリソースを取得し、適切に扱うのか、適切な PlatformTransactionManager を使って適切な生成/再利用/クリーンアップ、およびトランザクション同期のトリガをかける(オプション)ことを保障するのかという疑問が残る。

8.3.1 高レベルアプローチ

望ましい方法は、Spring の最も高レベルな永続性インテグレーション API を使うことだ。これらは、ネイティブの API を置き換えるものではなく、内部的にリソースの生成/再利用、クリーンアップ、オプションのリソースとのトランザクション同期、例外のマッピングを行う。よって、ユーザのデータアクセスコードはこれらに関することは何も意識する必要はなく、純粋に決まり文句ではない永続ロジックに集中することができる。通常、同じ*テンプレート*アプローチは、JdbcTemplate、HibernateTemplate、JdoTemplate などのようなクラスとして、すべての永続性 API がサポートされている。これらの統合クラスの詳細については、は本マニュアルで後述してある。

低レベルアプローチ

さらに低レベルには、(JDBC 用の)DataSourceUtils、(Hibernate 用の)SessoinFactoryUtils、(JDO 用の)PersistenceManagerFactoryUtils などがある。アプリケーションコードで直接ネイティブの永続性 API 用リソース型を扱いたい場合、これらのクラスが Spring で管理された適切なインスタンスが取得し、(オプションで)トランザクションがそのリソースと同期し、処理中に発生した例外を一貫した API に適切にマッピングされることを保証する。

例えば、JDBC では、DataSource の getConnection()メソッドを叩く伝統的な JDBC のやり方の代わりに、下記のように Spring の org.springframework.jdbc.datasource.DataSourceUtils を叩く。

```
Connection conn = DataSourceUtils.getConnection(dataSource);
```

既存のトランザクションが存在し、すでに同期した(リンクされた)コネクションがある場合、そのインスタンスが返される。そうでなければ、そのメソッド呼び出しは、新しいコネクションを生成する契機となる。これは、(オプションで)任意の既存のトランザクション同期し、同じトランザクション中でに後で再利用可能になる。これには任意の SQLException が Spring の CannotGetJdbcConnectionException --これは Spring のチェックなし DataAccessException 階層のひとつである--にラップされるという追加された利点がある。これは、SQLException から簡単に得られる以上の情報が得られ、たとえ、異なる永続テクノロジであっても、データベースを跨いだポータビリティが保証される。

これは、Spring のトランザクション管理(トランザクション同期はオプションだ)がなくてもちゃんと動作するので、トランザクション管理に Spring を使っていようが使っていまいが、この機能を利用することができることは、念を押しておくべきだろう。

もちろん、Spring の JDBC サポートや Hibernate サポートを一度でも使ったことがあれば、直接適切な API を使うより Spring の抽象化を使う方が気持ちよく仕事ができるので通常は DataSourceUtils や他のヘルパークラスを好んでは使わないだろう。例えば、もし、JDBC を使うのを単純にするために Spring の JdbcTemplate や jdbc.object パッケージを使うのであれば、正しいコネクションの検索がバックグランドで走り余計なコードを書く必要がなくなるだろう。

これらの低レベルなリソースアクセスクラスの詳細については、本マニュアルで後述している。

TransactionAwareDataSourceProxy

一番低いレベルには、TransactionAwareDataSourceProxy クラスがある。これは、ターゲットの DataSource に対するプロキシであり、Spring で管理されたトランザクションのアウェアネスを追加するためにターゲットの DataSource をラップする。この点で、J2EE サーバで提供されるトランザクションの JNDI DataSource に似ている。

標準の JDBC DataSource インタフェースの実装が叩かれ、渡されないといけない既存のコードが存在するときを除いて、このクラスが使われる必要はなく、また望ましくない。そういう場合は、このコードを利用可能であり、なおかつ Spring で管理するトランザクションにジョインすることもできる。前述したこれよりも上のレベルの抽象化を使って自分の新しいコードを書く方が望ましい。

詳細については、TransactionAwareDataSourceProxy の Javadoc を参照して欲しい。

8.4 プログラマティックなトランザクション管理

Spring では 2 つの方法によるプログラマティックなトランザクション管理が提供されている。

- TransactionTemplate を使う
- PlatformTransactionManager の実装を直接使う

我々は通常は、1 つめの方法をお勧めしている。

2 つめのやり方は JTA の UserTransaction の API を使うのと同じだ(でも、例外のハンドリングの煩わしさは解消されているが)。

8.4.1 TransactionTemplate を使う

TransactionTemplate では、JdncTemplate や HibernateTemplate のような他の Spring テンプレートと同じやり方が採用されている。これはアプリケーションコードの中でリソースの獲得／開放の処理をしないでいいようにコールバック方式を使っている。(try/catch/finally ももういらない。)他のテンプレートと同じように、TransactionTemplate もスレッドセーフである。

トランザクションコンテキストの中で実行しないといけないアプリケーションコードはこのようになる。値を返すのに TransactionCallback を使ってもかまわないという点に注目して欲しい。

```
Object result = tt.execute(new TransactionCallback() {
    public Object doInTransaction(TransactionStatus status) {
        updateOperation1();
        return resultOfUpdateOperation2();
    }
});
```

返り値がない場合は、このように TransactionCallbackWithoutResult を使うこと。

```
tt.execute(new TransactionCallbackWithoutResult() {
    protected void doInTransactionWithoutResult(TransactionStatus status) {
        updateOperation1();
        updateOperation2();
    }
});
```

コールバック中のコードは TransactionStatus オブジェクトにある setRollbackOnly()メソッドを叩けばトランザクションをロールバックすることができる。

TransactionTemplateを使いたいアプリケーションクラスは、PlatformTransactionManagerにアクセスしなければならない。これは通常は、JavaBeanプロパティかあるいはコンストラクタ引数として公開されている。

このようなクラスをモックやPlatformTransactionManagerスタブを使ってユニットテストを実施するのは簡単だ。そこではJNDIルックアップやスタティックマジックは用いない。これは単純なインタフェースであり、通常のように、Springを用いてユニットテストを簡単にすることができる。

8.4.2 PlatformTransactionManager を使う

さらに、トランザクションを管理するのにorg.springframework.transaction.PlatformTransactionManagerを直接利用してもよい。使おうとするPlatformTransactionManagerの実装をビーンリファレンスからビーンへ単に渡すだけだ。そして、TransactionDefinitionとTransactionStatusオブジェクトを使って、トランザクションの初期化、ロールバック、コミットを行う。

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition()
def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);

TransactionStatus status = txManager.getTransaction(def);

try {
    // execute your business logic here
}
catch (MyException ex) {
    txManager.rollback(status);

    throw ex;
}
txManager.commit(status);
```

8.5 宣言的トランザクション管理

Springでは宣言的トランザクション管理も提供している。これはSpring AOPによって可能となっているものではある。ところがトランザクション的アスペクトコードはSpringによるものであり、決まりきったやり方で利用される。AOPの概念はこういったコードを有効に使うためものではない、と一般に理解されている。

Springユーザのほとんどは、宣言的トランザクション管理を選んでいる。これは、アプリケーションコードへのインパクトを最小限に抑えるための1つの手段であり、不可侵な軽量コンテナにおける理想と最も合致しているのである。

ここで、EJB CMTを念頭においてSpringの宣言的トランザクション管理との類似点や相違点についての説明を始めることは有効かもしれない。これらの基本的なアプローチは似たようなものだ。つまり、トランザクションの振る舞い(あるいは欠けているもの)を個々のメソッドに落とし込むことができる。また

必要であれば、setRollbackOnly()をトランザクションコンテキストで呼ぶことも可能だ。異なっている点は以下のとおり。

- JTA と結び付けられる EJB の CMT と異なり、Spring の宣言的トランザクション管理は任意の環境で動作する。JDBC や JDO、Hibernate あるいは他のトランザクションでも設定を変更するだけで動作させることができる。
- Spring では EJB のような特殊なクラスだけではなく、どんな POJO にも宣言的トランザクション管理を適用させることができる。
- Spring では EJB とは異なる特徴である宣言的ロールバックルールが提供されている。これについては後述してある。ロールバックは単なるプログラマティカルにではなく宣言的にコントロールすることができる。
- Spring では、AOP を使ってトランザクションの振る舞いをカスタマイズする機会が得られる。例えば、トランザクションがロールバックする場合に自前の振る舞いを挿入したければそれができるのだ。他にも任意のアドバイスをトランザクションアドバイスに合わせて追加することもできる。EJB の CMT だと、setRollbackOnly()の他にコンテナのトランザクション管理に影響を及ぼす方法は用意されていない。
- Spring ではハイエンドなアプリケーションサーバではサポートされている、トランザクションコンテキストをリモート呼び出しにプロパゲートする機能はサポートしていない。こうした機能が必要なのであれば、EJB を使うことをお勧めする。しかしながら、この機能を安易に使ってはいけない。普通、トランザクションをリモート呼び出しにまで広げるようなことは必要とはしない。

ロールバックルールの概念は重要だ。このルールにより、どの例外(もしくは throwable)により自動ロールバックを発生させなければならないかを指定することができる。我々はこれを Java のコードでではなく、設定の中で宣言的に指定する。よって、カレントのトランザクションをロールバックするのに TransactionStatus オブジェクトの setRollbackOnly()を叩くことができる場合でも、MyApplicationException が発生したら必ずロールバックするというルールを指定することができる。これはビジネスオブジェクトをトランザクションインフラに依存させなくてもかまわないという十分な利点になるのだ。例えば、トランザクションあるいはその他の任意の Spring API をインポートする必要がないのだ。

EJB のデフォルトの振る舞いでは EB コンテナはシステム例外(通常はランタイム例外)が発生した場合は自動的にトランザクションをロールバックさせるようになっているが、EJB の CMT はアプリケーション例外(java.rmi.RemoteException 以外のチェック済み例外)が発生した場合に、トランザクションをロールバックを自動的には行わない。Spring では宣言的トランザクション管理のデフォルトの振る舞いは EJB の規約(ロールバックは未検査の例外のみ自動で行う)に従っているが、これのカスタマイズが有用であることが多い。

我々のベンチマークでは、Spring の宣言的トランザクション管理のパフォーマンスは EJB の CMT よりも勝っている。

Spring でトランザクションのプロキシを設定する通常の方法は、トランザクションのプロキシを生成するために TrnasactionProxyFactoryBean を使う方法だ。このファクトリビーンは、単に、Spring の汎用 ProxyFactoryBean の、ターゲットオブジェクトをラップするためのプロキシの生成を追加した特別版で、

通常 TransactionInterceptor を自動で生成し、このプロキシにアタッチし、決まり文句のコードを減らしてくれる。(ProxyFactoryBean のように、プロキシから適用させるために他のインタセプタや、AOP アドバイスを指定してもよい点には注意してほしい)。

TransactionProxyFactoryBean を使う場合、何よりもまず最初に、target 属性でトランザクション的プロキシにラップするターゲットオブジェクトを指定する必要がある。このターゲットオブジェクトは、通常 POJO ビーン定義だ。また他にも適切な PlatformTransactionManager への参照を指定する必要がある。最後に、transaction attributes を指定しなければならない。トランザクション属性には(上述したように)、どこに適用するのかと同様にどのトランザクションセマンティクスを使いたいのかを含める。ここで、下記の例について考えてみよう。

```
<!-- this example is in verbose form, see note later about concise for ¥
    multiple proxies! -->
<!-- the target bean to wrap transactionally -->
<bean id="petStoreTarget">
  ...
</bean>

<bean id="petStore"
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager" ref="txManager" />
    <property name="target" ref="petStoreTarget" />
    <property name="transactionAttributes">
        <props>
<prop key="insert*">PROPAGATION_REQUIRED,-MyCheckedException</prop>
            <prop key="update*">PROPAGATION_REQUIRED</prop>
            <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
        </props>
    </property>
</bean>
```

トランザクションプロキシはそのターゲットのインタフェースを実装する。この場合であれば petStoreTarget という ID をもつビーンだ。 (CGLIB を使えば、ターゲットクラスの非インタフェースメソッドを同様にトランザクションを考慮してプロキシすることができるという点に注意してほしい)。ターゲットがどのインタフェースも実装していない場合はこれは自動的に起こるが、強制的にいつも起こるように "proxyTargetClass"プロパティを true にセットする。もちろん通常は、クラスではなく、インタフェースでプログラムしたいのだが。)特定のターゲットインタフェースのみをプロキシするために、proxyInterfaces プロパティを使ってトランザクションを考慮したプロキシを制限することは可能(であり、通常はいい考え)だ。また、org.springframework.aop.framework.ProxyConfig から継承したいくつかのプロパティを使って TransactionProxyFactoryBean の振る舞いをカスタマイズし、すべての AOP プロキシファクトリで共有することも可能だ。

トランザクションのインタセプタは究極的には、特定のクラスの特定のメソッドに適用されるセマンティクスを定義しているトランザクションの属性を知るために(TransactionAttribute オブジェクトの形式で)Spring の TransactionAttributeSource インタフェースを実装したオブジェクトを利用する。最も基本的な方法は、TransactionAttributeSource インタフェース(Spring にはいくつか実装が用意されている)を実装しているビーンを生成するためのプロキシを生成するときにこの TransactionAttributeSource インスタンスを指定し、それを参照する(あるいは内部クラスとしてそれをラップする)プロキシファクトリビーンの transactionAttributeSource プロパティを直接設定することだ。反対に、このプロパティにテキスト文字列を設定して、Spring にあらかじめ登録されている **TransactionAttributeSourceEditor** が自動的にそのテキスト文字列を MethodMapTransactionAttributeSource インタフェースに変換するのを利用するのもよい。

しかしながら、この例で示したように、ほとんどのユーザはそうではなく transactionAttributes プロパティを設定することによりトランザクションの属性を定義する。このプロパティは Java.util.Properties 型をもっていて、内部的に NameMatchTransactionAttributeSource オブジェクトに変換される。

上述した定義でもわかるように、NameMatchTransactionAttributeSource オブジェクトには、名前/値のペアのリストが保持されている。各ペアのキーはメソッド、あるいはトランザクションのセマンティクスを適応させるメソッド(オプションで"*"のワイルドカードが使用可能)である。ここでのメソッド名は、パッケージ名と一致しないが、ラップされるターゲットオブジェクトのクラスに関連したものとみなされる。この例のように Properties の値として指定する場合、TransactionAttributeEditor により定義されているように、文字列フォーマットで指定する。このフォーマットは以下のようなものである。

PROPAGATION_NAME,ISOLATION_NAME,readOnly,timeout_NNNN,+Exception1,-Exception2

- 例外のハンドリング: RuntimeException ロールバック、通常の(チェックされた)例外は処理しない
- トランザクションの種類は、read と write
- アイソレーションレベル: TransactionDefinition.ISOLATION_DEFAULT
- タイムアウト: TransactionDefinition.TIMEOUT_DEFAULT

プロパゲーション設定とアイソレーションレベルの設定のフォーマットについては、org.springframework.transaction.transactionDefinition クラスの JavaDoc を参照してほしい。この文字列フォーマットは同じ値の Integer の定数名と同じものだ。

この例で、insert*のマッピングにロールバックルールが含まれていることに注意してほしい。-MyCheckedException がここで追記されているのは、メソッドが MyCheckedException あるいはそのサブクラスをスローしたら、トランザクションが自動的にロールバックされることを示している。ここではコンマで区切れば多重にロールバックルールを指定できる。-プレフィクスは強制ロールバック、+プレフィクスはコミットを表す。(これは本当に何をやっているかをわかっていれば、未検査例外の場合でさえもコミットしてもよいのだ)。

TransactionProxyFactoryBean によって、「preInterceptors」と「postInterceptors」プロパティを使って、interception の振る舞いを追加するために、オプションでプレアドバイス、ポストアドバイスを設定することができる。任意の数のプレアドバイス、ポストアドバイスを設定することが可能で、アドバイザ(ポ

イントカットを含めることができるような場合）や、メソッドインタセプタ、あるいは今の Spring のコンフィグレーションでサポートされている任意のアドバイス型(ThrowsAdvice,AfterReturuningAdvice もしくは BeforeAdvice のような、デフォルトでサポートされているもの)でもかまわない。おれらのアドバイスは、共有インスタンスモデルをサポートできなければならない。もしステートフルミックスインのような高度な AOP 機能がトランザクションプロキシに必要であれば、TransactionProxtFactoryBean といった便宜的なプロキシクリエータではなく、通常は汎用の org.springframework.aop.framework.ProxyFactoryBean を使うのがベストだ。

注意：ほとんど同じようなトランザクションプロキシを複数生成する必要がある場合、上記のような形の TransactionProxyFactoryBean 定義を使うことのは過度に冗長に見えるかもしれない。セクション 6.7 ″簡潔なプロキシ定義″にも述べたように、トランザクションプロキシ定義の冗長性を大幅に減らすために内部ビーン定義と一緒に親子ビーン定義の利点を使いたいと思うだろう。

8.5.1 トランザクション境界設定のためのソースアノテーション

XML ベースの transaction attribute source の定義は便利で、どんな環境でも動作するが、Java5 以降に乗り換える意思があるのであれば、アトリビュートソースの代わりに Spring がサポートするトランザクションアノテーションを JDK 標準フォーマットの中で使うことを検討したくなるのは確かだと思う。

Java のソースコード中に、トランザクションセマンティクスを直接的明示することで、影響を受けるコードに密接な宣言をすることになるが、通常は望ましくない関連ほどには危険性はないので、トランザクションとしてデプロイされるコードは通常この方法でデプロイするのが普通だ。

8.5.1.1 トランザクションアノテーション

org.springframework.transaction.annotation.Transactional アノテーションは、インタフェース、インタフェースメソッド、クラス、あるいはクラスメソッドがトランザクションのセマンティクスを持っていることを示すのに利用される。

```
@Transactional
public interface OrderService {

void createOrder(Order order);
List queryByCriteria(Order criteria);
```

このアノテーションは、インタフェース、クラス、あるいはメソッドがトランザクションであることを指定する。デフォルトのトランザクションセマンティクスは、read/write、PROPAGATION_REQUIRED、ISOLATION_DEFAULT、TIMEOUT_DEFAULT で、Exception ではなく RuntimeException 発生時にロールバックするというものである。

アノテーションのオプションのプロパティを使えばトランザクションの設定を変更することができる。

Transactional アノテーションのプロパティ		
プロパティ	型	説明
propagation	enum: Propagation	オプションのプロパゲーション設定(デフォルトは、**PROPAGATION_REQUIRED**)
isolation	enum: Isolation	オプションのアイソレーションレベル(デフォルトは ISOLATION_DEFAULT)
readOnly	boolean	read/write あるいはリードオンリートランザクション(デフォルトは false もしくは read/write)
rollbackFor	**Class** オブジェクトの配列。**Trowable** からの派生クラスであること。	オプションで指定する、発生したときにロールバックする例外クラスの配列。デフォルトでは、チェック済みの例外ではロールバックせず、未チェックの(**RuntimeException** から派生した)例外の場合にロールバック
rollbackForClassname	クラス名文字列の配列。クラスは、**Trowable** の派生であること。	オプションで指定する、発生したときにロールバックさせる例外クラスの名前の配列
noRollbackFor	**Class** オブジェクトの配列。**Trowable** からの派生であること。	オプションで指定するロールバックさせない例外クラスの配列。
noRollbackForClassname	クラス名文字列の配列。**Trowable** からの派生であること。	オプションで指定する、発生したときにロールバックさせない例外のクラス名の配列。

アノテーションはインタフェース定義やインタフェースにあるメソッド定義、クラス定義、あるいはクラスに定義されているメソッド定義の前に書く。インタフェースと、そのインタフェースを実装したクラスの両方に書かれるかもしれない。メソッドのトランザクションセマンティクスを評価する際には、継承階層の最も下位で評価される。

8.5.1.1.1. Transactional アノテーションの例

あるクラスをアノテートする定義:

```
public class OrderServiceImpl implements OrderService {

  @Transactional
  void createOrder(Order order);
  public List queryByCriteria(Order criteria);

}
```

下記の例では、インタフェースはリードオンリートランザクションとしてアノテートされる。これは、デフォルトでメソッドに対して設定される。この createOrder メソッドのアノテーションは、このメソッドをオーバライドし、read/write にトランザクションを設定し、(RuntimeException に関するデフォルトのロールバックルールに加えて)DuplicateOrderIdException(おそらくチェックされていない例外)がスローされた場合にトランザクションをロールバックするように指定している。

```
@Transactional(readOnly=true)
interface TestService {

  @Transactional(readOnly=false,
```

```
                    rollbackFor=DuplicateOrderIdException.class)
  void createOrder(Order order) throws DuplicateOrderIdException ;

  List queryByCriteria(Order criteria);
}
```

このインタフェースを実装するクラス定義もこの設定が、クラス、もしくはメソッドに対しオーバライドされる点に注意してほしい。

8.5.1.1.2. Transactional アノテーションを適用するように Spring に伝える

このアノテーションそのものをインスタンスをインタフェースやクラスの要素に追加しても実装クラスがトランザクションにラッピングされない。Spring に、トランザクションのプロキシをそのアノテーションを持つクラスに生成するように伝えてやらないといけないのだ。

キーは、org.springframework.transaction.annotation.AnnotationTransactionAttribteSource クラスの利点を生かし、クラスファイルから Annotations フォーマットのトランザクション属性を読むことだ。TransactionProxyFactoryBean を使った前述の例で、テキスト形式でトランザクション属性を指定する TransactionAttributes プロパティ AnnotationTransactionAtributeSource を指定する TransactionAttributeSource プロパティを直接使って書き換える。

```
<bean id="petStore" ¥

class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
¥
  <property name="transactionManager" ref="txManager"/>
  <property name="target" ref="petStoreTarget"/>
  <property name="transactionAttributeSource">
<bean ¥

class="org.springframework.transaction.annotation.AnnotationTransactionAttribute
Source"/> ¥
  </property>
</bean>
```

TransactionAttributeSource プロパティは各プロキシインスタンスごとに変更する必要はないので、コードのコピーを避けるために親のあるいは子のビーン定義を使う場合、このプロパティは親定義に基づいて設定されるだけで忘れられ、アトリビュートソースはは各クラスファイルから正しい設定を読み込むので、子ビーンの中でオーバライドする必要がない。

Transactional アノテーションが適用されることを保証するために APO を使う

前述した例は理想のものよりもまだ余計なものがある。クラスファイル中のアノテーション自体が、プロキシがアノテートされたクラスのために作成されるのに必要であるという表示として使われる場合、(ターゲットビーンを指定するための)プロキシごとの XML は原則的に必要ない。

もっと AOP にフォーカスしたアプローチだと、 (ターゲットビーンごとではなく、一回しか使われない)決まり文句の ML を減り、自動的にプロキシがすべてのクラス用に Transactional アノテーションと一緒

に生成されることを保証することができるようになる。Spring AOPは前の章で詳細に解説してあるので、一般的なAOPのドキュメントとして参照すべきであるが、キーポイントはDefaultAdvisorAutoproxyCreatorとBeanPostProcessorを使うことだ。これは、ビーンのポストプロセッサなので、生成されると同時に、生成されたすべてのビーンを参照する機会を持っている。もし、ビーンにTransactionalアノテーションが含まれていると、これをラップするのにトランザクショナルプロキシが自動で生成される。

```
<bean ¥

class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/> ¥

<bean ¥

class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor"> ¥
  <property name="transactionInterceptor" ref="txInterceptor"/>
</bean>

<bean id="txInterceptor" ¥
    class="org.springframework.transaction.interceptor.TransactionInterceptor"> ¥
  <property name="transactionManager" ref="txManager"/>
  <property name="transactionAttributeSource">
<bean ¥

class="org.springframework.transaction.annotation.AnnotationTransactionAttributeSource"/> ¥
  </property>
</bean>
```

これには数多くのクラスが関わっている。

- TramsactionInterceptor: AOPのアドバイス、実際には、メソッド呼び出しをインターセプトし、トランザクションでラップする。
- TransactionAttributeSourceAdvisor: AOPアドバイザ(アドバイスである、TransactionInterceptor、および(そのアドバイスを適用させる)ポイントカットを、TransactionAttributeSourceの形式で保持する。)
- AnnotationTransactionAttributeSource:クラスファイルから読み出されるトランザクション属性を提供するTransactionAttributeSourceの実装。
- DefaultAdvisorAutoProxyCreator:コンテキスト中からアドバイザをを探し出し、自動的に、トランザクションのラッパであるプロキシオブジェクトを生成する。

8.5.2 BeanNameAutoProxyCreator，別の宣言的アプローチ

TransactionProxyFactoryBean はとても有用であり、オブジェクトをトランザクションプロキシでラップする場合のフルコントロールが可能になる。親／子ビーン定義とターゲットを保持する内部ビーンを一緒に使い、Java5 のアノテーションがオプションとして利用できない場合、一般的にトランザクションをラッピングするベストな方法である。大量のビーンを完全に一意な方法でラップする必要がある場合（例えば、BeanNameAutoProxyCreator と呼ばれる BeanFactoryPostProcessor を使って「全部のメソッドをトランザクションにせよ」という決まり文句）は、この単純化された用例にはそれほど冗長にはなりえない別のアプローチが取れるだろう。

要約すると、ApplicationContext が一旦初期化情報を読み込むと、BeanPostProcessor インタフェースが実装されたビーンがインスタンス化され、ApplicationContext 内のほかのビーン全てに対し、後処理(post-process)を実施する機会ができる。よって、このメカニズムを使うと、適切に設定された BeanNameAutoProxyCreator が ApplicationContext 内のほかのビーン(名前で識別される)に後処理を実施するのに用いられ、トランザクションプロキシでラップされる。生成される実際のトランザクションプロキシは本質的には、TransactionProxyFactoryBean を使って生成されるのと同じなので、これ以上は触れない。

サンプルの設定をみてみよう。

```
<!-- Transaction Interceptor set up to do PROPAGATION_REQUIRED on all ¥
     methods -->
  <bean id="matchAllWithPropReq"
class="org.springframework.transaction.interceptor.MatchAlwaysTransactionAttribu
teSource">
     <property name="transactionAttribute"  value="PROPAGATION_REQUIRED"/>
  </bean>
  <bean id="matchAllTxInterceptor"
class="org.springframework.transaction.interceptor.TransactionInterceptor">
     <property name="transactionManager" ref="txManager" />
     <property name="transactionAttributeSource" ref="matchAllWithPropReq"/>
  </bean>

<!-- One BeanNameAutoProxyCreator handles all beans where we want all ¥
     methods to use
        PROPAGATION_REQUIRED -->
  <bean id="autoProxyCreator"
class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator"
>
     <property name="interceptorNames">
       <list>
         <idref local="matchAllTxInterceptor"/>
         <idref bean="hibInterceptor"/>
```

```
      </list>
    </property>
    <property name="beanNames">
      <list>
        <idref local="core-services-applicationControllerSevice"/>
        <idref local="core-services-deviceService"/>
        <idref local="core-services-authenticationService"/>
        <idref local="core-services-packagingMessageHandler"/>
        <idref local="core-services-sendEmail"/>
        <idref local="core-services-userService"/>
      </list>
    </property>
  </bean>
</beans>
```

ApplicationContext 内にすでに TransactionManager のインスタンスがあると仮定して、まず最初にやらなければいけないことは、使用する TransactionInterceptor インスタンスを生成することだ。TransactionInterceptor は、プロパティとして渡される TransactionAttributeSource を実装しているオブジェクトに基づいてインターセプトすべきメソッドを判断する。この場合、全てのメソッドとマッチするとても単純なケースを扱いたい。これは必ずしも最も効率的なやり方ではないが、セットアップが非常に早い。というのは、あらかじめ定義された特別な MatchAlwaysTransactionAttributeSource(これは単純に全てのメソッドにマッチする)を使うことができるからだ。もっと限定したいのであれば、MethodMapTransactionAttributeSource,NameMatchTransactionAttributeSource,AttributesTransactionAttributeSource のような他の派生を使うことも可能だ。

これで、トランザクションインタセプタができて、ApplicationContext に、同じやり方でラップしたい 6 つのビーンの名前と一緒に、定義した BeanNameAutoProxyCreator インスタンスに渡す。見てわかるように、最終的には 6 つのビーンを同じように TransactionProxyFactoryBean でラップするほど冗長にはならない。7 つ目のビーンをラップするには設定ファイルに 1 行追加するだけだ。

ここで複数のインタセプタが適用できるということに気づいたかもしれない。この場合では、すでに定義した(bean id =hibInterceptor)HibernateInterceptor も適用してある。これは、Hibernate のセッションを管理するものだ。

TransactionProxyFactoryBean と BeanNameAutoProxyCreator を切り替える場合にビーンの命名に関する考慮と一緒に気をつけてほしいことが 1 つある。前者には、ターゲットビーンが内部ビーンとして定義されていなければ、通常ラップしたいターゲットビーンに myServiceTarget と同じように id をつけ、プロキシオブジェクトには myService という id をつける。そして、ラップされたオブジェクトのユーザはみな、単純にプロキシ(つまり、myService)を参照する。(これは、命名規則のサンプルで、ポイントは、ターゲットオブジェクトはプロキシとは違う名前をもち、両方が ApplicationContext から利用可能である、という点である)。しかしながら、BeanNameAutoProxyCreator を使う場合、ターゲットオブジェクトに myService のような名前をつける。そして BeanNameAutoProxyCreator がターゲットオブジェクトを後処理し、プロキシを生成する場合、プロキシがアプリケーションコンテキストに元のビーンの名前で挿入

されるようにする。その点から、プロキシ(ラップされたオブジェクト)だけが ApplicationContext から利用可能になる。内部ビーンとしてターゲットを指定して TransactionProxyFactoryBean を使う場合は、内部ビーンには通常名前をつけないので、この命名の問題は関係ない。

8.5.3 AOP と Transaction

これまで本章を読んできてお分かりのように、Spring の宣言的トランザクション管理を効果的に用いるのに、本当に AOP のエキスパートになる必要はない、つーか、AOP に関して詳しく知らなくてもよい。しかしながら、Spring AOP の「パワーユーザ」になりたいのであれば、宣言的トランザクション管理と強力な AOP の能力を組み合わせるのが簡単だというのがわかるだろう。

8.6 プログラミング的トランザクション管理か宣言的トランザクション管理かを選択する

プログラミング的トランザクション管理は、トランザクション操作が少ない場合に限っては、いい方法だと思う。例えば、いくつかの更新操作にのみトランザクションが必要になるようなウェブアプリケーションがあるとすると、Spring や他のテクノロジを使ってトランザクションプロキシをセットアップしたいとは思わないだろう。この場合は、TransactionTemplate を使うのがいいやり方だと思う。

一方、大量のトランザクション操作があるようなアプリケーションであれば、宣言的トランザクション管理が有益だ。宣言的トランザクション管理では、トランザクション管理をビジネスロジックから切り離すことができ、また Spring ではその設定は難しくはない。Spring を使えば、EJB CMT よりも宣言的トランザクション管理の設定にかかるコストが大幅に削減される。

8.7 トランザクション管理のためにアプリケーションサーバが必要？

Spring のトランザクション管理機能、特に宣言的トランザクション管理は、J2EE アプリケーションがいつアプリケーションサーバを必要とするかについての従来の考え方を大きく変える。

特に、EJB を使って宣言的なトランザクションを実行するだけであれば、アプリケーションサーバは必要ではない。実際、強力な JTA 機能をもつアプリケーションサーバがあっても、Spring の宣言的トランザクションの方が強力で、EJB CMT よりも生産的なプログラミングモデルが用意されていると判断するのは正しい。

もし、多数のトランザクションリソースを集める必要がある場合に限り、アプリケーションサーバの JTA 機能が必要となる。多くのアプリケーションではこういった必要性には直面しない。例えば、多くのハイエンドアプリケーションでは単一でスケーラビリティが高い、Oracle 9i RAC のようなデータベースを用いる。

もちろん、JMS や JCA のようなアプリケーションサーバのほかの機能を必要とするかもしれない。しかしながら、JTA だけを必要とするのであれば、JOTM のようなオープンソースの JTA アドオンも検討することができる。(Spring は JOTM と[out of the box]統合する。)しかしながら、2004 年の早い時期に、ハイエンドアプリケーションサーバはより堅牢に XA トランザクションのサポートを提供する。

最も重要な点は、Spring では、アプリケーションサーバで一杯一杯の規模になるまで、あなたのアプリケーションをいつスケールさせるかを選択するすることができる、という点だ。JDBC コネクションのようなローカルトランザクションを使ってコードを書くには EJB CMT や JTA を使う他に手がなく、グローバルでコンテナで管理されたトランザクションで走るようなコードが必要な場合、多くの手戻りに直面するような時代はもう過ぎ去った。Spring であれば、設定に変更をいれないといけないだけで、コードに手を入れる必要はないのだ。

8.8 アプリケーションサーバに応じたインテグレーション

Spring のトランザクション抽象化は、通常アプリケーションサーバを問わない。さらに、JTA の UserTransaction オブジェクトや TransactionManager オブジェクト用にオプションで JNDI ルックアップを実行できる Spring の JtaTransactionManager クラスは、後者オブジェクトの位置を自動認識して設定することができるがこれは、アプリケーションサーバの種類による。TransactionManager インスタンスへのアクセスにより、トランザクションセマンティクスを拡張することができる。詳細については JtaTransactionManager の Javadoc を参照されたい。

8.8.1 BEA WebLogic

WebLogic 7.0, 8.1 あるいはそれ以降の環境では通常、JtaTransactionManager の代わりに WebLogicJtaTransactionManager を使いたいと思うだろう。これは特別に WebLogic に特化した,通常の JtaTransactionManager のサブクラスだ。これを使えば、トランザクション名、トランザクション単位のアイソレーションレベル、およびすべての場合においてトランザクションを適切に再会できるといった機能を含んだ、標準の JTA セマンティクス以上に、WebLogic で管理されたトランザクション環境で Spring のトランザクション定義をフルに使うことができる。

詳細については、Javadoc を参照してほしい。

8.8.2 IBM WebSphere

WebSphere 5.1, 5.0 あるいはバージョン 4 の環境では、Spring の WebSphereTransactionManagerFactoryBean クラスを使いたいかもしれない。これは WebSphere 環境で JTA の TransactionManager を検索するファクトリビーンであり、WebSphere のスタティックアクセスメソッドを使って検索を行う。このメソッドは WebSphere のバージョンごとに異なる。

一旦このファクトリビーンを使って JTA の TransactionManager インスタンスを取得すると、Spring の JtaTransactionManager には、JTA の UserTransaction オブジェクトしか使わない場合よりもトランザクションセマンティクスを拡張するために、このインスタンスへの参照が保持される。

詳細は、Javadoc を参照してほしい。

8.9 共通の問題

8.9.1 特定のデータソースに対する間違ったトランザクションマネージャの利用

開発者は、要求を満たすためには正しい PlatformTransactionManager の実装を使うように注意する必要がある。

どのようにSpringのトランザクション抽象化が、JTAグローバルトランザクションを使って行われているかを理解することは重要だ。適切に用いれば、矛盾は発生しない。Spring では単純化されたポータブルな抽象化が提供されるだけだ。

もしグローバルトランザクションを使う場合、全てのトランザクション操作に Spring の org.springframework.transaction.jta.JtaTransactionManager を使わないといけない。そうでないと、コンテナ DataSource のようなリソースに対するローカルトランザクションとみなされてしまう。このようなローカルトランザクションは意味をなさないし、優れたアプリケーションサーバであれば、エラーとして扱うだろう。

8.9.2 すでにアクティブでないトランザクションもしくは DataSource に関するウソの警告

非常に厳密な XADataSource の実装をもつ JTA 環境において、-- 現時点では、WebLogic と WebSphere のいくつかのバージョンだけ -- JTA の TransactionManager オブジェクトを意識せずに設定された Hibernate をそのような環境で使う場合、ウソの警告や例外がアプリケーションサーバのログに残されることがある。この警告、もしくは例外は、トランザクションがすでにアクティブでないという理由により、アクセスされたコネクションがすでに有効でない、あるいは JDBC アクセスがすでに有効でないということを言っている。例として、WebLogic から上げられた実際の例外についてみてみよう。

```
java.sql.SQLException: The transaction is no longer active - ¥
    status: 'Committed'.
  No further JDBC access is allowed within this transaction.
```

この警告は*unresolved*に記述されているように、解決するのは簡単だ。

9. ソースレベルメタデータのサポート(Ver 1.2.7)

9.1 ソースレベルメタデータ

ソースレベルメタデータはプログラム要素への属性やアノテーションを追加したものであり、通常はクラスやメソッドである。

例えば、下記のように、あるクラスにメタデータを追加することができる。

```
/**
 * Normal comments
 * @@org.springframework.transaction.interceptor.DefaultTransactionAttribute()
 */
public class PetStoreImpl implements PetStoreFacade, OrderService {
```

下記のように、メソッドにもメタデータを追加することができる。

```
/**
 * Normal comments
 * @@org.springframework.transaction.interceptor.RuleBasedTransactionAttribute()
 *
@@org.springframework.transaction.interceptor.RollbackRuleAttribute(Exception.class)
 *
@@org.springframework.transaction.interceptor.NoRollbackRuleAttribute("ServletException")
 */
public void echoException(Exception ex) throws Exception {
    ....
}
```

この例では、jakarta Commons の Attributes の文法を用いている。

ソースレベルメタデータは(Java 業界における)XDoclet、およびマイクロソフトの.Net プラットフォームのリリースで主流に躍り出た。ここでは、ソースレベル属性をトランザクションやプーリング、その他の振る舞いを制御するのに利用している。

このアプローチのもつ価値は、J2EE コミュニティで評価された。例えば、EJB で排他的に用いられた従来の XML デプロイメントデスクリプタよりも冗長性が大幅に軽減されるのだ。プログラムソースコードから何かしらを外部に切り出すことは望ましいことではあるが、重要な全体的な設定 -- とりわけトランザクション特性 -- はプログラムソースの一部だ。EJB の仕様で仮定されていることには反するが、(トランザクションタイムアウトのようなパラメータは変更されることはあっても)メソッドのトランザクション特性を変更するのはほとんど意味がない。

メタデータ属性は典型的にはアプリケーションクラスが求めるサービスについて記述するためにフレームワークのインフラで主に使われるが、メタデータ属性が実行時に問い合わせを受けられるようするべきものでもある。これは XDoclet のようなソリューションとの主たる違いであり、その違いは EJB 自動生成物のようなコードを生成する方法としてまず最初にメタデータを見るというものである。

この分野には数多くのソリューションがあり、その中には下記のようなものがある。

- 標準の Java アノテーション>:標準の Java のメタデータの実装(JSR-175 として開発され、Java5 で利用できるようになった)。Spring では、トランザクションの境界設定や JMX を用いるための Java5 のアノテーションをすでにサポート済みである。しかし Java 1.4 や 1.3 用のソリューションを必要としている
- XDoclet:うまく確立されたソリューションであり、まず第一にコード生成が意図されている。
- Java1.3 や 1.4 のための、中でも Commons Attributes が最も有望に思える、様々なオープンソースでのアトリビュートの実装。これらはすべて特別なプリコンパイル、ポストコンパイルの手順を必要とする。

9.2 Spring におけるメタデータのサポート

重要なテーマに関わる抽象的な概念の準備として、Spring ではメタデータ実装へのファサードが org.springframework.metadata.Attributes インタフェースの形で提供されている。

このようなファサードに付加価値があるのはいくつかの理由がある。

- Java5 では、言語レベルでメタデータのサポートを提供しているが、以下のような抽象化を提供することは今でも価値のあることだ。
    - Java 5 のメタデータはスタティックである。コンパイル時にクラスに関連づけられ、デプロイされた環境下では変更できない。そこで、デプロイ中にある属性を例えば XML ファイルでオーバライド可能な階層的なメタデータが求められている。
    - Java 5 のメタデータは Java のリフレクション API から返されるので、テストの際にモックテストを実施するのが不可能なのだ。Spring では、モックテストができるように簡単なインタフェースを提供している。
    - 1.3 や 1.4 のアプリケーションに向けたメタデータのサポートが少なくともあと 2 年は必要とされるだろう。Spring では現在、それに対するソリューションを提供することを目標としており、このような重要な分野では Java 5 の使用を強制されるようなことはない。
- (Spring 1.0-1.2 で用いられている)Commons Attributes のような現状のメタデータ API はテストするのがとてもやっかいだ。Spring では、モックテストがもっと簡単な単純なメタデータインタフェースを提供している。

Spring の Attributes インタフェースは下記のようなものである。

```
public interface Attributes {

    Collection getAttributes(Class targetClass);

    Collection getAttributes(Class targetClass, Class filter);

    Collection getAttributes(Method targetMethod);

    Collection getAttributes(Method targetMethod, Class filter);

    Collection getAttributes(Field targetField);

    Collection getAttributes(Field targetField, Class filter);
}
```

これは、必要最小限のインタフェースだ。JSR-175 では、メソッド引数の属性のように、これより多くの機能が提示されている。Spring 1.0 の時点では、Java 1.3 以降の EJB や.NET 風の効果的な宣言的エンタープライズサービスを提供するのに必要なメタデータのサブセットを提供することを目標とする。Spring 1.2 の時点では、よく似た JSR-175 のアノテーションは Commons Attributes と直接代わりになるので JDK1.5 でサポートされる。

このインタフェースは、.NET のような Object の属性を提示している点に注意してほしい。これは、(文字列属性しか提示しない)Nanning Aspects と(DR2 の時点での)JBoss 4 のような属性システムとは異なるものだ。Object 属性をサポートすることは十分な利点がある。属性によってクラス階層に組み入れることができ、また設定パラメータに賢く反応することができるようになる。

ほとんどの attribute の実装では、attribute クラスはコンストラクタ引数か JavaBean プロパティを使って設定する Commons Attributes では両方がサポートされている。

Spring の抽象化 API が全部そうであるように、Attributes はインタフェースだ。これにより attribute の実装のモックテストやユニットテストが簡単にできるようになっている。

9.3 Jakarta Commons 属性との統合

現状、Spring では Jakarta Commons の Attributes しか独自にはサポートしていないが、他のメタデータをサポートするために org.springframework.metadata.Attributes インタフェースの実装を用意するのは容易である。

Commons の Attributes 2.1(http://jakarta.apache.org/commons/attributes/)は使いでのある attributes のソリューションだ。コンストラクタ引数や javaBean プロパティからの attribute の設定をサポートしていて、attribute 定義におけるよりよい自己ドキュメンテーションを提供する。(JavaBean プロパティのサポートは Spring チームの要望により追加された)。

我々は、Commons Attributes の attribute 定義についてこれまで 2 つの例を見てきたので、概説する必要があると思う。

- attribute クラスのクラス名。これは上述したように FQN にすることができる。当該 attribute クラスがすでにインポート済みであれば、FQN は必要ない。さらに、attribute コンパイラの設定で "attribute packages"に指定することもできる。
- コンストラクタ引数かあるいは JavaBean プロパティによる任意の必要なパラメータ化

ビーンプロパティは下記のようなものである。

```
/**
 * @@MyAttribute(myBooleanJavaBeanProperty=true)
 */
```

コンストラクタ引数と JavaBean プロパティを(Spring の IoC の時のように)組み合わせることができる。

Java 1.5 の Attributes とは異なり、Commons Attributes は Java 言語に統合されていないので、ビルドプロセスの中で特別な属性コンパイルの手順を実行する必要がある。

To run Commons Attributes as part of the build process, you will need to do the following.

ビルドプロセスの中で Commons Attributes を実行するためには、下記の手順を踏む必要がある。

1. 必要なライブラリの Jar ファイルを$ANT_HOME/lib にコピーする。4 つのライブラリが必須であり、これらは全て Spring で配布されているものだ。
    - Commons Attributes コンパイラの Jar と API の Jar
    - XDoclet の xjavadoc.jar
    - Jakarta Commons の commons-collections.jar
2. 下記のように、Commons Attributes アントタスクをプロジェクトのビルドスクリプトにインポートする

```
<taskdef resource="org/apache/commons/attributes/anttasks.properties"/>
```

・ 次に、属性コンパイルタスクを定義する。これは、ソースコード中の属性を"コンパイル"するために、Commons Attributes の属性コンパイルタスクを用いる。この処理の結果、新たにソースコードが destdir 属性で指定された位置に生成される。この例では、テンポラリディレクトリを用いている。

```
<target name="compileAttributes">

  <attribute-compiler destdir="${commons.attributes.tempdir}">
    <fileset dir="${src.dir}" includes="**/*.java"/>
  </attribute-compiler>

</target>
```

このソース上で javac を実行するコンパイル対象はこの属性コンパイルタスクに依存させるべきであり、出力先ディレクトリに生成されたソースもコンパイルしなければならない。もし、属性定義にシンタックスエラーがあると、通常は属性コンパイラで検出される。しかしながら、もし属性定義が文法上妥当ではあるが、無効な型やクラス名を指定していると、生成された attribute クラスのコンパイルは失敗する。こ

のような場合、生成されたクラスを見れば問題の原因をつかむことができる。
Commons の Attributes は Maven もサポートしている。詳しい情報については、Commons Attributes のドキュメントを参照してほしい。
この属性コンパイルの手順は一見複雑に見えるが、実際は 1 回限りの手間でしかない。一旦設定してしまえば、属性コンパイル処理は追加されるので、通常はビルド処理が遅くなったと思うようなことはない。また一度コンパイル処理の設定をしてしまえば、本章で述べたような attributes を使うことで他の部分で多くの手間を省くことができるということがわかると思う。
もし属性インデックスのサポートが必要であれば(現状では、後述する、Spring の attribute-targeted ウェブコントローラでのみ必要となる)、コンパイルしたクラスの Jar ファイルに対してさらに別の手順が必要になる。この追加の手順では、Commons Attributes はソースコードに定義された全属性のインデックスを生成し、実行時のルックアップの効率化を行う。この手順は下記のようなものだ。

```
<attribute-indexer jarFile="myCompiledSources.jar">

    <classpath refid="master-classpath"/>

</attribute-indexer>
```

このビルド手順の例として、Spring の jPetStore サンプルアプリケーションの/attributes ディレクトリを見てほしい。ここにビルドスクリプトがあるのでこれをプロジェクトに合わせて修正して欲しい。
もし、あなたのユニットテストが attributes に依存しているのであれば、Commons の Attributes を使うのではなく、Spring の Attributes 抽象化を用いて依存関係を表現してみることをお勧めす。これによりよりポータブルになる、-- 例えば、もし将来 Java1.5 の attributes に差し替えても今作ったテストが動く -- だけでなく、テストを単純化できるのだ。Spring では、モックテストが簡単になるようなメタデータインタフェースを用意しているが、Commons の Attributes はスタティックな API なのだ。

9.4 メタデータと Spring AOP のオートプロキシ

メタデータ属性の最も重要な用途は Spring AOP との結合だ。これにより、.NET ライクなプログラミングモデル、つまりメタデータ属性を宣言したアプリケーションオブジェクトに宣言的サービスが自動的に提供されるような仕組みが提供される。このようなメタデータ属性は、宣言的トランザクション管理の場合のように本フレームワークによる独創的なサポートがされているかあるいは自前のものだ。
AOP とメタデータ属性の間には幅広いシナジー効果があるのだ。

9.4.1 基本事項

これは、Spring AOP のオートプロキシ機能により実現されている。設定はおそらくこんな感じになると思う。

```
<bean ¥

class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/> ¥
```

```
<bean ¥

class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor"> ¥
  <property name="transactionInterceptor" ref="txInterceptor"/>
</bean>

<bean id="txInterceptor" ¥
    class="org.springframework.transaction.interceptor.TransactionInterceptor"> ¥
  <property name="transactionManager" ref="transactionManager"/>
  <property name="transactionAttributeSource">
<bean ¥

class="org.springframework.transaction.interceptor.AttributesTransactionAttributeSource"> ¥
      <property name="attributes" ref="attributes"/>
    </bean>
  </property>
</bean>

<bean id="attributes" ¥
    class="org.springframework.metadata.commons.CommonsAttributes"/>
```

この基本的な考え方は、AOP の章にあるオートプロキシに関する議論でよくご理解いただけていると思う。

最も重要なビーン定義は autoproxy と transactionAdvisor という名前のビーンのものだ。ここで注意して欲しいのは、実際のビーン名は重要ではない。そのクラスがどういったものかが重要なのだ。

org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator クラスのビーン定義は、属するファクトリにある、アドバイザ実装にマッチした全てのインスタンスをアドバイス("オートプロキシ")する。このクラスは属性については何も知らないが、Advisor のポイントカットマッチングに従う。ポイントカットは属性のことをわかっているのだ。

従って、属性に基づいた宣言的トランザクション管理を提供する AOP アドバイザが必要なだけである。任意にカスタマイズした Advisor の実装を追加することも同様に可能であり、自動的に評価、適用が行われる。(もし、必要であれば、同じオートプロキシ設定の別の属性の条件にポイントカットがマッチする Advisor を使っても構わない。)

最後に、この attributes ビーンは Commons Attributes の Attributes の実装である。他のソースから属性を取得するために、org.springframework.metadata.Attributes のほかの実装に置き換えよう。

9.4.2 宣言的トランザクション管理

ソースレベル属性の最も一般的な使い方は、.NET 風の宣言的トランザクション管理の提供だ。一旦、上述したビーン定義を配置すれば、宣言的トランザクションを必要とするアプリケーションオブジェクト

をいくつでも定義することができる。トランザクション属性をもつクラスやメソッドだけにトランザクションアドバイスが渡されるようになる。必要トランザクション属性を定義する以外には、他に何もする必要はないのである。
.NET とは違い、トランザクション属性をクラスやメソッドごとに指定することもできる。クラスレベル属性は、指定されると、全てのメソッドで"継承される"。メソッド属性はクラスレベル属性でオーバライドされる。

9.4.3 プーリング

繰り返しになるが、.NET でのように、クラスレベル属性によってプーリングの振る舞いを有効にすることができる。Spring ではこの振る舞いを任意の POJO に適用することができる。pooling 属性を以下のようにプール対象とするビジネスオブジェクトに設定するだけでよい。

```
/**
 * @@org.springframework.aop.framework.autoproxy.target.PoolingAttribute(10)
 * @author Rod Johnson
 */
public class MyClass {
```

通常はオートプロキシのインフラ設定を必要とする。そして、プールする TargetSourceCreator を以下のように指定する必要がある。プーリングはターゲットの生成に影響を及ぼすので、通常のアドバイスは利用することができない。pooling 属性があれば、そのクラスに適用可能なアドバイザが存在していなくてもプーリングは適用される、という点に注意して欲しい。

```
<bean id="poolingTargetSourceCreator"
class="org.springframework.aop.framework.autoproxy.metadata.AttributesPoolingTargetSourceCreator">
  <property name="attributes" ref="attributes"/>
</bean>
```

適切なオートプロキシビーン定義には Pooling target source creator を含んだ"custom target source creators"のリストを指定する必要がある。上述した例を修正しこのプロパティを含むようにしたものを以下に示す。

```
<bean ¥

class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"> ¥
  <property name="customTargetSourceCreators">
    <list>
      <ref bean="poolingTargetSourceCreator"/>
    </list>
  </property>
</bean>
```

一般に、Spring でメタデータを使う場合のように、これは 1 回しか修正の手間がかからない。一度この設定をやってしまえば、追加したビジネスオブジェクトをプーリングするのはとても簡単だ。
プーリングが必要になることはまれである、という点には議論の余地がある。つまり、多くのビジネスオブ

ジェクトにプーリングを適用する必要性はめったにない。従って、この機能は通常はあまり日の目をみないと思う。
より詳細な情報については、org.springframework.aop.framework.autoproxy パッケージの Javadoc を参照してほしい。Commons の Pool に独自のコードを最小限使うのではなく、他のプーリングの実装を使うことも可能だ。

9.4.4 カスタムメタデータ

オートプロキシの基盤には柔軟性が備わっているので、.NET メタデータ属性よりも可能性がある。ある種の宣言的振る舞いを実装するのに、独自の属性を定義することができる。これには以下の手順を必要とする。

- 独自属性のクラスを定義する
- この独自属性が存在することで発火するポイントカットを持つ Spring AOP の Advisor を定義する。
- この Advisor をビーン定義として汎用のオートプロキシ基盤と同じアプリケーションコンテキストに追加する。
- 属性を POJO に追加する。

独自の宣言的セキュリティやキャッシュ機能のように潜在的に実施したいものがいくつかあると思う。これは、プロジェクトにおいて設定の手間を大きく削減することができる強力なメカニズムだ。しかしながら、この背景に AOP に依存している、という点については覚えておいてほしい。動作する Advisor が増えれば増えるほど、実行時の設定はより複雑になってしまう。(もし、どのアドバイスがどのオブジェクトに適用されるのかを知りたければ、リファレンスを org.springframework.aop.framework.Advised にキャストしてみればいい。これにより Advisor を調べることができると思う)。

9.5 MVC ウェブ層の設定を最小限にするために属性を使う

1.0 の時点での Spring のメタデータのその他の主な用途は、Spring MVC ウェブ設定を簡単にするオプションを提供することだ。
Spring MVC では、柔軟なハンドラのマッピング、つまり入力リクエストからコントローラ(あるいは他のハンドラ)インスタンスへのマッピングを提供する。通常、ハンドラマッピングは適切な Spring DispatcherServlet 用に xxx-servlet.xml ファイルで設定されている。
このマッピングを DispatcherServlet 設定ファイル内に保持することは通常はいい考えだ。最大限の柔軟性を確保できる。特に下記の点が挙げられる。

- コントローラインスタンスは明示的に、XML ビーン定義によって、Spring IoC に管理される。
- このハンドラマッピングはコントローラの外部であり、同じコントローラインスタンスが同じ DispatcherServlet コンテキストで複数のマッピングを与えられるかもしくは、異なる設定下では再利用される。

- Spring MVC は他のほとんどのフレームワークでも可能な、単なるリクエスト URL とコントローラのマッピングだけでなく、任意の条件に基づくマッピングのサポートが可能だ。

しかしながら、これは各コントローラごとに、典型的に必要になる両方のハンドラマッピング(通常はハンドラマッピング XML ビーン定義にある)とコントローラ自体のための XML マッピングの両方が必要になる、ということだ。
Spring ではソースレベル属性に基づく単純なアプローチが用意されており、これは単純なシナリオにおいては魅力のある選択肢だと思う。
本セクションで述べているこのアプローチは比較的単純な MVC シナリオに最も適している。これは、同じコントローラを別のマッピングで使うことできるとか、リクエスト URL ではない何かに基づくマッピングさせることができるといった Spring の MVC の能力をある程度制限する。
この方法では、コントローラは、マッピングする URL を 1 つだけ指定したクラスレベルのメタデータ属性を 1 つもしくは複数マークされる。
下記の例はこの方法を示したものだ。各々のケースで、Cruncher 型のビジネスオブジェクトに依存するコントローラを用意した。通常は、この依存性は依存性注入により解決される。この Cruncher は適切な DispatcherServlet XML ファイルもしくは親のコンテキストにあるビーン定義から利用可能でなければならない。
ここで、マッピングすべき URL を指定したコントローラクラスに属性を追加する。依存性は JavaBean プロパティかもしくはコンストラクタ引数を用いて表現することができる。この依存性は、オートワイヤで解決されなければならない。すなわち、確実に型 Cruncher のビジネスオブジェクトが 1 つコンテキスト中に存在していなければならない。

```
/**
 * Normal comments here
 * @author Rod Johnson
 * @@org.springframework.web.servlet.handler.metadata.PathMap("/bar.cgi")
 */
public class BarController extends AbstractController {

    private Cruncher cruncher;

    public void setCruncher(Cruncher cruncher) {
        this.cruncher = cruncher;
    }

    protected ModelAndView handleRequestInternal(
            HttpServletRequest request, HttpServletResponse response)
            throws Exception {
        System.out.println("Bar Crunching c and d =" +
            cruncher.concatenate("c", "d"));
        return new ModelAndView("test");
```

```
    }

}
```

この自動マッピングを有効にするには、下記を属性ハンドラマッピングを指定した適切な xxx-servlet.xml ファイルに追加する必要がある。この特別なハンドラマッピングは任意の数のコントローラと属性を上記のように扱うことができる。このビーン ID("commonsAttributesHandlerMapping")は重要ではない。型が問題なのだ。

```
<bean id="commonsAttributesHandlerMapping"
class="org.springframework.web.servlet.handler.metadata.CommonsPathMapHandlerMapping"/>
```

現状では、上述した例のように Attributes ビーン定義は必要とはしていない。というのは、このクラスは Spring のメタデータ抽象化経由ではなく、直接 Commons Attributes API で動作するからだ。

今は、各コントローラごとの XML 設定は必要ない。コントローラは自動的に指定された URL にマッピングされる。コントローラは Spring のオートワイヤ機能を使い、IoC の恩恵を受ける。例えば、上述した単純なコントローラの"cruncher"というビーンプロパティで示された依存性は現状のウェブアプリケーションコンテキストで自動的に解決される。これは、セッターインジェクションとコンストラクタインジェクションのどちらも解決可能であり、特別な設定はひとつも必要ない。

複数の URL パスのコンストラクタインジェクションの例を下記に示す。

```
/**
* Normal comments here
* @author Rod Johnson
*
* @@org.springframework.web.servlet.handler.metadata.PathMap("/foo.cgi")
* @@org.springframework.web.servlet.handler.metadata.PathMap("/baz.cgi")
*/
public class FooController extends AbstractController {

    private Cruncher cruncher;

    public FooController(Cruncher cruncher) {
        this.cruncher = cruncher;
    }

    protected ModelAndView handleRequestInternal(
            HttpServletRequest request, HttpServletResponse response)
            throws Exception {
        return new ModelAndView("test");
    }
```

```
}
```

この方法は、下記のような利点がある。

- 設定しないといけない項目が大幅に削減される。コントローラを追加する度に、XML 設定を追加する必要はない。属性ドリブンなトランザクション管理でのように、一旦基本的な基盤を用意すれば、アプリケーションクラスをさらに追加するのはとても簡単なのだ。
- Spring の IoC の強力な利点がコントローラを設定するのに利用できる。

この方法では以下のような制限がある。

- 1 回限りの手間であるがビルド手順が複雑になる。属性コンパイルの手順と属性にインデックスをつける手順が必要となる。しかしながら、一旦やってしまえば、これは問題にはならない。
- 現状では、Commons の Attributes にしか対応していない。だが、今後は他の attribute のサポートも追加されるとは思うが。
- このようなコントローラでは"型によるオートワイヤリングする"依存性注入のみがサポートされている。しかしながら、IoC が関係している、Struts のアクション(フレームワークによる IoC のサポートなしで)やおそらく、WebWork のアクション(基本的な IoC のみのサポートあり)以上のことはできていない。
- 自動的な IoC の解決の信頼性が揺らぐかもしれない。

型によるオートワイヤリングでは、指定された型の依存性は 1 つのみが存在していないといけないので、AOP を使う場合は注意を払う必要がある。例えば、transactionProxyFactoryBean を使う場合、Cruncher のようにビジネスインタフェースの実装を 2 つ、オリジナルの POJO の定義と、トランザクション AOP プロキシになってしまう。これでは、持っているアプリケーションコンテキストが型の依存性を明確に解決できないので、動作しない。これを解決するには、定義された Cruncher の実装が 1 つしかないので、オートプロキシ基盤を設定して、AOP オートプロキシを使うことだ。すると、この実装を自動的にアドバイスされる。なので、この方法であれば上述したような属性をターゲットとする宣言的サービスでも動作する。属性コンパイル手順は対象となるウェブコントローラを処理のに決まった手順でなければいけないので、これをセットアップするのは簡単だ。
他のメタデータ機能とは異なり、現状は Commons Attributes の実装,org.springframework.web.servlet.handler.metadata.CommonsPathMapHandlermapping だけしか利用することができない。この制限は、PathMap 属性をもつクラス全部で attributes の API を叩けるようにするために属性コンパイラだけでなく属性インデックシングも必要である、という事実によるものだ。インデクシングは現状では、org.springframework.metadata.Attributes 抽象インタフェースでは用意されていないが、ゆくゆくは用意されると思う。(他の属性の実装のサポートを追加したい、--これは、インデクシングをサポートしないといけない--という場合は、CommonsPathMapHanderMapping のスーパクラス、AbstractPathMapHandlerMapping を拡張し、2 つの protected な抽象メソッドを実装して、求める属性用の API を実装すれば簡単にできる)。
従って、新たにビルド手順に 2 つの処理を追加する必要がある。属性コンパイルと属性インデクシングだ。属性インデクサタスクを使うのは上述した。Commons Attributes では現状、インデクシングの入力

に対し、jar ファイルを必要とすることに注意してほしい。
メタデータマッピング方式のハンドラを使うのであば、いつでも Spring のこれまでの XML マッピング方式に切り替えることができるので、この選択肢に限定されることはない。このため、私はしばしばウェブアプリケーションをメタデータマッピングを使って作り始めることもある。

9.6 メタデータ属性の他の使い方

メタデータ属性のほかの用途については、ポピュラーになりつつあるように見える。Spring 1.2 の時点で、(JDK 1.3 以降で)Common Attributes、(JDK 1.5 で)JSR-175 のアノテーションにより、JMX アクセス用メタデータ属性がサポートされている。

9.7 追加したメタデータ API のサポートを追加する

他のメタデータ API をサポートしたいと思うのであれば、簡単にできるのでやるべきだ。
単に、org.springframework.metadata.Attributes インタフェースをあなたのメタデータ API に対するファサードとして実装するだけだ。そうすると、上述したようにこのオブジェクトを自分のビーン定義に含めることができるようになる。
メタデータドリブンなオートプロキシのような、メタデータを用いているフレームワークサービスは全て自動的にあなたの新しいメタデータプロバイダに使うことができるようになる。

10. DAO のサポート(Ver 1.2.7)

10.1 イントロ

この Spring における DAO(Data Access Object)のサポートは、JDBC、Hibernate、あるいは標準化された方法の JDO のようなデータアクセス技術を簡単に使えるようにすることを第 1 の目標とする。これによりこれらの技術の間での差し替えが簡単になり、また各技術に特有の例外をキャッチする心配をしなくていいようにすることができる。

10.2 一貫性例外ヒエラルキー

Spring では、**SQLException** のような各テクノロジ独自の例外から、ルート例外として **DataAccessException** をもつ独自の例外ヒエラルキーへの便利な変換が用意されている。この例外は元の例外をラッピングするので、何が間違ったのかといった情報を失うリスクは生じない。

JDBC の例外に加えて、Spring では Hibernate の例外もラッピングし、プロプライエタリで検査済み例外から抽象化された実行時例外の集合へ変換することができる。また JDO 例外についても同様だ。これにより、回復不可能で適切なレイヤでしか扱えないほとんどの永続化に関する例外を catches/throws や例外の宣言のいやな決まり文句に煩わされることなくハンドリングすることができるようになる。今までどおり、必要となるいかなるところででも例外をトラップし、ハンドリングすることができる。上述したように、JDBC 例外(特定の DB 用方言も含む)も、同じヒエラルキーに変換され、つまり一貫したプログラミングモデルにのっとりで JDBC で操作を実行できるということである。

上述したものは、ORM アクセスフレームワークのテンプレート版にも当てはまる。もし **Interceptor** ベースのクラスを使えば、**SessionFactoryUtils** の **convertHibernateAccessException** メソッドや **convertJdoAccessException** メソッドそれぞれへ委譲することによりアプリケーション自体が **HibernateException** や **JDOException** の処理に注意を払わなければならない。これらのメソッドはここで挙げた例外を **org.springframework.dao** 例外ヒエラルキーと互換性のある例外への変換を行う。JDOException は未検査なので、例外の観点からは汎用 DAO 抽象化が犠牲になるが、単に投げることもできる。

この Spring が用いる例外ヒエラルキーの概要は下記のダイアグラムのようになる。

10.3 DAO サポート用一貫性抽象クラス

JDBC や JDO、Hibernate のような様々なデータアクセステクノロジを一貫した方法で簡単に使えるようにするために、Spring では拡張可能な抽象 DAO クラス群を提供している。このアブストラクトクラスにはデータソースや使っているテクノロジに特化した設定を行うメソッドがある。

Dao をサポートするクラスには下記のものがある。

- **JdbcDaoSupport** – JDBC データアクセスオブジェクト用スーパクラスである。**DataSource** が設定されている必要があり、これをベースとする **JdbcTemplate** がサブクラスとして提供されている
- **HibernateDaoSupport** – Hibernate データアクセスオブジェクト用スーパクラス。**SessionFactory** が設定されている必要があり、これをベースとする **HibernateTemplate** がサブクラスとして提供されている。**SessionFactory** やフラッシュモード、例外の変換などに似た後者の設定を再利用するために **HibernateTemplate** により直接初期化するかどうかを選択することができる。
- **JdoDaoSupport** – JDO データアクセスオブジェクト用スーパクラス。**PersistenceManagerFactory** が設定されている必要があり、これをベースとする **JdoTemplate** がサブクラスとして提供されている。

11. JDBC を用いたデータアクセス(Ver 1.2.7)

11.1 イントロ

Spring で提供されているこの JDBC 抽象化フレームワークは 4 つの異なるパッケージ、core、datasource、object、support によって構成されている。

org.springframework.jdbc.core パッケージには、JdbcTemplate クラスと様々なコールバック用インタフェース、それに様々な関係するクラスが含まれている。

org.springframework.jdbc.datasource パッケージには、DataSource に簡単にアクセスするためのユーティリティクラスや、単純な DataSource の実装がいろいろ含まれており、この実装は変更していない JDBC コードを J2EE コンテナ外部でテスト、実行するために用いられる。ユーティリティクラスには、JNDI からのコネクションの取得と、必要であればコネクション切断のための静的なメソッドが用意されている。このメソッドでは、例えば DataSourceTransactionManager で使うためのスレッドバウンドコネクションがサポートされている。

次に、org.springframework.jdbc.object パッケージには、RDBMS の問い合わせ、更新、ストアドプロシージャをスレッドセーフで再利用可能なオブジェクトとして表現したクラスが含まれている。もちろん問い合わせから返されたオブジェクトはデータベースから"切り離される"が、この方法は JDO によりモデル化される。このより上位レベルの JDBC 抽象化は org.springframework.jdbc.core パッケージによる低レベル抽象化に依存している。

最後に、org.springframework.jdbc.support パッケージには SQLException の変換機能やいくつかのユーティリティクラスがある。

JDBC 処理中にスローされた例外は org.springframework.dao パッケージに定義されている例外に変換される。これはつまり、Spring の JDBC 抽象化レイヤを使っているコードでは、JDBC や RDBMS に依存したエラーハンドリングを実装する必要がない、ということだ。変換された例外は全て未検査であり、他の例外が呼び出し側へ伝播できる場合でも回復が可能な例外をキャッチするかどうかの選択は委ねられている。

11.2 JDBC Core のクラスを使って基本的な JDBC 処理やエラー処理を制御する

11.2.1 JdbcTemplate

これは、JDBC のコアパッケージの中でも中心的なクラスだ。これは、リソースの生成、解放をハンドリングしてくれるので JDBC が簡単に使えるようになる。これによりコネクションを常に閉じるのを忘れるようなよくあるエラーが回避できる。このクラスでは、ステートメントの生成、実行のようなコア JDBC のワーク

フローを実行し、アプリケーションコードから SQL の受け渡しと結果の抽出を分離する。このクラスでは、SQL の問い合わせ、ステートメントやストアドプロシージャコールの更新、ResultSet に設定されたイテレーションを模擬し、返されたパラメータ値の抽出を行う。さらに、JDBC 例外をキャッチし、org.springframework.dao パッケージで定義された、汎用的で、より有益な例外ヒエラルキーに変換する。

このクラスを使うコードでは、明確に定義された契約を持たせるコールバックインタフェースの実装のみが必要となる。PreparedStatementCreator コールバックインタフェースは、このクラスで渡されるコネクションを渡され、あらかじめ用意されたステートメントを生成し、SQL および必要となるパラメータを提供する。同じことはコールバックステートメントを生成する CallableStatementCreator インタフェースにも当てはまる。RowCallbackHandler インタフェースは ResultSet の各行から値の抽出を行う。

このクラスは DataSource のリファレンスで直接インスタンス化することでサービスを実装したり、アプリケーションコンテキスト内であらかじめ用意され、ビーンリファレンスとしてサービスに渡される、といった使い方ができる。(注：DataSource はサービスに直接渡される 1 つ目のケースでも、テンプレートがあらかじめ用意される 2 つ目のケースでも、常にアプリケーションコンテキスト内のビーンとして構築されるべきだ)。このクラスはコールバックインタフェースや、SQLExceptionTranslator インタフェースによりパラメータ化されているので、サブクラスを用意する必要はない。このクラスで発生した SQL 関連の問題については全てログに記録される。

11.2.2 DataSource

データベースから取得したデータで動作させるためには、データベースへのコネクションを取得する必要がある。Spring で行っている方法は、DataSource を経由するというものだ。DataSource は JDBC 仕様の一部で、汎用化されたコネクションファクトリとしてみなすことができる。このように見なすことにより、コンテナやフレームワークでコネクションプーリングやトランザクション管理に関する問題をアプリケーションコードから隠蔽することが可能になる。開発者としては、データベースにどのように接続するかといった詳細については知る必要がなく、データソースをセットアップする管理者の責任だ。コードを開発し、テストしている間は、その両方の役割を担う必要があるのがほとんどだと思うが、どのように生成したデータのソースが設定されているかは必ずしも知る必要はないのだ。

Spring の JDBC 層を使う場合、データソースを JNDI から取得するか、Spring の配布パッケージで提供されている実装を使って自分で設定するかを選択することができる。後者はウェブコンテナの外部でユニットテストを実施するのに便利だ。本章では、DriverManagerDataSource の実装を用いるが、後者の方法をカバーする実装は他にもいくつか用意されている。この DriverManagerDataSource は JDBC コネクションを取得する際に用いている方法とおそらく同じ方法で動作する。DriverManager がドライバクラスをロードできるようにするために、JDBC ドライバの完全なクラス名指定する。そして、ドライバ間で変わる URL を用意する必要がある。ここで使用する正しい値については、使うドライバのドキュメントを参照する必要がある。最後に、データベースに接続するためのユーザ名とパスワードを用意する。DriverManagerDataSource の設定するための例は下記のようになる。

```
DriverManagerDataSource dataSource = new DriverManagerDataSource();
dataSource.setDriverClassName( "org.hsqldb.jdbcDriver");
dataSource.setUrl( "jdbc:hsqldb:hsql://localhost:");
dataSource.setUsername( "sa");
```

```
dataSource.setPassword( "");
```

11.2.3 SQLExceptionTranslator

SQLExceptionTranslator は、SQLException とデータアクセスストラテジを問わない org.springframework.dao.DataAccessException との間で変換が可能なクラスにより実装されるインタフェースだ。

実装をよりいっそう正確に、(例えば、JDBC 用 SQLState コードを用いて)汎用的にも、(例えば Oracle のエラーコードを用いて)プロプライエタリにもすることができる。

SQLErrorCodeSQLExceptionTranslator はデフォルトで使われる。SQLExceptionTranslator を実装したものだ。この実装は特定のベンダコードを使う。SQLState の実装よりもより正確だが、ベンダに特化している。このエラーコードの変換は SQLErrorCodes という名前の JavaBean 型クラスで保持されたコードに基づいたものだ。このクラスは、"sql-error-codes.xml"という名前の設定ファイルのコンテンツに基づいて SQLErrorCodes を生成するファクトリであると名前が示唆する SQLErrorCodesFactory によって生成され、配置される。このファイルはベンダコードで占められ、DatabaseMetaData から得られた DatabaseProductName に基づいてカレントのデータベース用コードが使われる。

この SQLErrorCodeSQLExceptionTranslator は下記のマッチングルールを適用する。

- 任意のサブクラスで実装されたカスタムの変換を試す。このクラスが具象であり、これ自体が利用される場合は、このルールは適用されない点に注意してほしい。
- エラーコードマッチングを適用する。エラーコードはデフォルトで SQLErrorCodesFactory から取得する。これはクラスパスからエラーコードをルックアップし、データベースメタデータのデータベース名からエラーコードを入力する。
- フォールバックトランスレータを使う。SQLStateSQLExceptionTranslator はデフォルトのフォールバックトランスレータだ。

SQLErrorCodeSQLExceptionTranslator は下記のような方法を使って拡張することが可能だ。

```
public class MySQLErrorCodesTranslator extends
SQLErrorCodeSQLExceptionTranslator {
    protected DataAccessException customTranslate(String task, String sql,
SQLException sqlex) {
        if (sqlex.getErrorCode() == -12345)
            return new DeadlockLoserDataAccessException(task, sqlex);
        return null;
    }
}
```

この例で指定されているエラーコード"-12345"は変換され、他のエラーもデフォルトのトランスレータの実装で変換するように残される。このカスタムトランスレータを使うためには、setExceptionTranslator メソッドを使って JdbcTemplate に渡し、このトランスレータを必要とするデータアクセス処理全てにこの JdbcTemplate を使う必要がある。下記は、このカスタムトランスレータをどのように使うかを示した例である。

```
// create a JdbcTemplate and set data source
```

```
JdbcTemplate jt = new JdbcTemplate();
jt.setDataSource(dataSource);
// create a custom translator and set the datasource for the default translation lookup
MySQLErrorCodesTransalator tr = new MySQLErrorCodesTransalator();
tr.setDataSource(dataSource);
jt.setExceptionTranslator(tr);
// use the JdbcTemplate for this SqlUpdate
SqlUpdate su = new SqlUpdate();
su.setJdbcTemplate(jt);
su.setSql("update orders set shipping_charge = shipping_charge * 1.05");
su.compile();
su.update();
```

このカスタムトランスレータはデータソースに渡され sql-error-codes.xml でエラーコードをルックアップするのにこのデフォルト変換を使いたいので、カスタムトランスレータはデータソースに渡される。

11.2.4 実行手順

SQL ステートメントを実行するために、必要とされるとても小さなコードがある。ここで必要になるのは、DataSource と JdbcTemplate だけだ。一度使えば、JdbcTemplate で用意されている多くの便利なメソッドを使えるようになる。ここで新しいテーブルを生成するための必要最小限であるが、完全に機能するクラスのために取り込む必要のあるものを示した短いサンプルをお見せしよう。

```
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

public class ExecuteAStatement {
    private JdbcTemplate jt;
    private DataSource dataSource;

    public void doExecute() {
        jt = new JdbcTemplate(dataSource);
        jt.execute("create table mytable (id integer, name varchar(100))");
    }

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
}
```

11.2.5 クエリを実行する

実行方法に加えて、問い合わせの方法もたくさんある。そのうちのいくつかは、値をひとつだけ返すような問い合わせに使うように意図されたものだ。カウントを取得したり、1 つの行から特定の値を検索したいことがあると思う。その場合は、queryForInt や queryForLong、queryForObject を使うといい。最

後のものは、返された JDBC 型から引数として渡された Java クラスに変換するものである。型変換が無効の場合は、InvalidDataAccessApiUsageException が投げられる。ここに、1 つの問い合わせメソッド、1 つは int 型用、1 つは String 用の例をお見せする。

```
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

public class RunAQuery {
    private JdbcTemplate jt;
    private DataSource dataSource;

    public int getCount() {
        jt = new JdbcTemplate(dataSource);
        int count = jt.queryForInt("select count(*) from mytable");
        return count;
    }

    public String getName() {
        jt = new JdbcTemplate(dataSource);
         String name = (String) jt.queryForObject("select name from mytable",
String.class);

        return name;
    }

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
}
```

値を 1 つしか結果を返さないクエリメソッドに加えて、クエリが返した各行をエントリにもつリストを返すメソッドがいくつか用意されている。最も汎用的なものは、各エントリが Map になっていてそのマップの各エントリが各行のカラム値を表すようなリストを返す queryForList だ。上述した例に、全ての行のリストを検索するようなメソッドを追加するとすると、下記のようになる。

```
    public List getList() {
        jt = new JdbcTemplate(dataSource);
        List rows = jt.queryForList("select * from mytable");
        return rows;
    }
```

これで返されるリストは [{name=Bob, id=1}, {name=Mary, id=2}] のようになっている。

11.2.6 データベースを更新する

他にも利用可能な更新メソッドが多数用意されている。あるプライマリキー用カラムを更新するような例をお見せしよう。この例では、行パラメータ用プレースホルダをもつ SQL ステートメントを使っている。ほとんどのクエリや更新メソッドにはこの機能が具備されている。このパラメータ値はオブジェクトの配列として渡される。

```
import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

public class ExecuteAnUpdate {
    private JdbcTemplate jt;
    private DataSource dataSource;

    public void setName(int id, String name) {
        jt = new JdbcTemplate(dataSource);
        jt.update("update mytable set name = ? where id = ?", new Object[] {name,
new Integer(id)});
    }

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
}
```

11.3 データベースへの接続方法を制御する

11.3.1 DataSourceUtils

JNDI からコネクションを取得したり、必要に応じてコネクションをクローズするためのスタティックメソッドを提供する Helper クラスでは、DataSourceTransactionManager で使うためのスレッドバウンドコネクションをサポートしている。

注意: getDataSourceFromJndi メソッドは ApplicationContext 各々でビーンファクトリやアプリケーションコンテキストを使わないアプリケーションでターゲットにされる。自前のビーンやファクトリにある JdbcTemplate インスタンスもあらかじめ設定しておくのが望ましい。JndiObjectFactoryBean は DataSource を JNDI からフェッチし、DataSource ビーンリファレンスを他のビーンに渡すのに用いることができる。別の DataSource への切り替えについては、どのように設定しているかだけの問題である。FactoryBean の定義を他の非 JNDI な DataSource に切り替えることもできる。

11.3.2 SmartDataSource

リレーショナルデータベースへの接続を提供することができるクラスで実装されるべきインタフェースである。javax.sql.DataSource インタフェースを拡張し、操作した後でコネクションが閉じているかどうか

を問い合わせることができる。これはコネクションを使いまわしたい場合には、とても役に立つことがある。

11.3.3 AbstractDataSource

興味対象ではない糊の役割をする、Spring の DataSource の実装用の抽象ベースクラスである。自前の DataSource の実装を書いている場合はこのクラスを拡張するとよい。

11.3.4 SingleConnectionDataSource

使用後に接続を閉じないシングルコネクションをラップする SmartDataSource の実装である。当然、マルチスレッドでは動かない。

もし、クライアントコードが、永続化ツールを使う場合のようにコネクションがプールされているという想定でクローズを呼び出す場合、suppressClose に true をセットする。これは物理的なコネクションの代わりに、クローズ抑制プロキシを返す。この返り値はネイティブの Oracle コネクション等にキャストすることはできない。

これは、主にテスト用クラスだ。例えば、簡単な JNDI 環境と接続し、アプリケーションサーバの外でコードを簡単にテストができるようになる。DriverManagerDataSource とは反対に、いつも同じコネクションを使いまわし物理的なコネクションを必要以上に生成するのを避けている。

11.3.5 DriverManagerDataSource

ビーンプロパティからプレーンで古い JDBC ドライバを設定する SmartDataSource の実装である。毎回新しいコネクションを返す。

これは、望ましい ApplicationContext 内の DataSource ビーンや、あるいは単純な JNDI 環境とあわせて J2EE コンテナの外部でのテストやスタンドアロン環境でのテストに役に立つ。プーリングを想定した Connection.close()のコールは単にコネクションを閉じるので、任意の DataSource を意識した永続化のコードはちゃんと動くはずだ。しかしながら、commons-dbcp のような JavaBean スタイルのコネクションプールを使うのは、たとえテスト環境といえどもとても簡単なので、DriverManagerDataSource 上でそのようなコネクションプールを使うのが、多くの場合は望ましい。

TransactionAwareDataSourceProxy

これは、Spring 管理下のアェアネスを追加するためにターゲットの DataSource をラップする、プロキシだ。この点で、J2EE サーバで提供されているトランザクション処理された JNDI DataSource に似ている。

標準 JDBC の DataSource インタフェースの実装に必ず呼ばれ、渡される既存のコードが存在する場合を除いて、このクラスを必要とし、このクラスを使うのが望ましいことはほとんどないはずだ。この場合、このコードを使えるようにするのは可能ではあるが、Spring 管理のトランザクションに入る。通常は、リソース管理に JdbcTemplate や DataSourceUtils のようなもっとレイヤの高い抽象化を使った自前の新しいコードを書くのが望ましい。

さらなる詳細は、TransactionAwareDataSourceProxy の Javadoc を参照してほしい。

11.3.7 DataSourceTransactionManager

単一の JDBC データソース用 PlatformTransactionManager の実装。特定のデータソースから JDBC コネクションをスレッドにバインドし、潜在的にデータソース毎に 1 本のスレッドコネクションを割り当てる効果がある。

アプリケーションコードでは、J2EE 標準の DataSource.getConnection の代わりに DataSourceUtils.getConnection(DataSource)から JDBC コネクションを検索する必要がある。これは、検査済み SQLException の代わりに org.springframework.dao の未検査例外を投げるのでとにかく推奨される。JdbcTemplate のような全てのフレームワーククラスはこの戦略を暗黙的に利用する。このトランザクションマネージャが一緒に用いられない場合は、ルックアップストラテジは共通のもののように正確に動作し、どんな場合でも使用することができる。

独自の分離レベルや適切な JDBC ステートメント問い合わせタイムアウトとして適用されるタイムアウトをサポートする。後者をサポートするために、アプリケーションコードは生成されたステートメント各々に JdbcTemplate を使うか DataSourceUtils.applyTransactionTimeout メソッドを呼ばないといけない。

この実装は、単一のリソースの場合には JtaTransactionManager の代わりに用いられる。というのも、JTA をサポートするコンテナを必要としないからだ。もし、要求されたコネクションルックアップパターンに固執するのであれば、両者での切り替えは単に設定次第である。JTA では独自の分離レベルをサポートしていない点に注意してほしい。

11.4 JDBC 操作を Java オブジェクトとしてモデリングする

この org.springframework.jdbc.object パッケージには、よりオブジェクト指向的な方法でデータベースにアクセスできるようにするためのクラス群が含まれている。クエリを実行したり、ビジネスオブジェクトが格納され、そのビジネスオブジェクトのプロパティにマッピングされたリレーショナルカラムデータが付与されたリストとして結果を受け取ることができる。さらにストアドプロシージャを実行したり、アップデートの実行、ステートメントの挿入、削除も行える。

11.4.1 SqlQuery

SQL クエリを表現するための、再利用可能でスレッドセーフなオブジェクト。サブクラスでは、ResultSet をイテレートする間、結果を格納しておけるようなオブジェクトを提供するために newResultReader()メソッドを実装しなければならない。このクラスを拡張した MappingSqlQuery で、行を Java オブジェクトにマッピングするためのもっと便利な実装が提供されているので、このクラスが直接利用されることはほとんどない。SqlQuery を拡張した他の実装としては、MappingSqlQueryWithParameters や UpdatableSqlQuery がある。

11.4.2 MappingSqlQuery

MappingSqlQuery は再利用可能な問い合わせではあるが、具象サブクラスでは、JDBC の ResultSet の各行をオブジェクトに変換するための abstract の mapRow(ResultSet, int)メソッドを実装しなければならない。

SqlQuery の全ての実装の中で、これが最もよく利用され、また一番簡単なやり方でもある。

下記は、customer テーブルから取り出したデータを Customer という Java オブジェクトにマッピングする独自のクエリの例を示したコード片である。

```
private class CustomerMappingQuery extends MappingSqlQuery {

    public CustomerMappingQuery(DataSource ds) {
        super(ds, "SELECT id, name FROM customer WHERE id = ?");
```

```
        super.declareParameter(new SqlParameter("id", Types.INTEGER));
        compile();
    }

    public Object mapRow(ResultSet rs, int rowNumber) throws SQLException {
        Customer cust = new Customer();
        cust.setId((Integer) rs.getObject("id"));
        cust.setName(rs.getString("name"));
        return cust;
    }
}
```

我々はこの唯一のパラメータとして DataSource をとるこの customer のクエリにコンストラクタを提供する。このコンストラクタでは、DataSource やこのクエリの行を検索するために実行されるべき SQL を渡してスーパクラスのコンストラクタを呼び出す。この SQL には、実行時に渡されるパラメータのためのプレースホルダが含まれているので PreparedStatement を生成するために利用される。各パラメータは SqlParameter で渡される declareParameter メソッドを使って宣言されなければならない。この SqlParameter は名前 java.sql.Types で定義されたと JDBC 型をもつ。全てのパラメータが定義された後、ステートメントがその後実行される準備が整うのでコンパイルメソッドを呼び出す。

では、この独自クエリがインスタンス化され実行されるところのコードを見てみよう。

```
    public Customer getCustomer(Integer id) {
        CustomerMappingQuery custQry = new
CustomerMappingQuery(dataSource);
        Object[] parms = new Object[1];
        parms[0] = id;
        List customers = custQry.execute(parms);
        if (customers.size() > 0)
            return (Customer) customers.get(0);
        else
            return null;
    }
```

この例のメソッドは、唯一のパラメータとして渡される id をもつ customer を検索する。CustomerMappingQuery クラスのインスタンスを生成した後、渡された全てのパラメータを含んだオブジェクトの配列を生成する。この場合、パラメータは 1 つしかなく、Integer として渡される。これで、このパラメータ配列を使ってクエリを実行する準備が整い、クエリにより返される各行に対応する Customer オブジェクトを含んだリストを取得する。この場合は、もしマッチするとすれば、1 つのエントリのみだろう。

11.4.3 SqlUpdate

RdbmsOperation というサブクラスは SQL Update を表す。クエリのように update オブジェクトは再利用可能だ。全ての RdbmsOperation オブジェクトのように、update はパラメータを持つことができ、これは、SQL で定義される。

このクラスでは query オブジェクトの execute()と類似した多くの update()メソッドが用意されている。

このクラスは具象である。サブクラスを作ることはできる(例えば、独自の update メソッドを追加するため)が、SQL の設定やパラメータを宣言することで、パラメタライズ化するのは簡単だ。

```
import java.sql.Types;

import javax.sql.DataSource;

import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.SqlUpdate;

public class UpdateCreditRating   extends SqlUpdate {
    public UpdateCreditRating(DataSource ds) {
        setDataSource(ds);
        setSql("update customer set credit_rating = ? where id = ?");
        declareParameter(new SqlParameter(Types.NUMERIC));
        declareParameter(new SqlParameter(Types.NUMERIC));
        compile();
    }

    /**
     * @param id for the Customer to be updated
     * @param rating the new value for credit rating
     * @return number of rows updated
     */
    public int run(int id, int rating) {
        Object[] params =
            new Object[] {
                new Integer(rating),
                new Integer(id)};
        return update(params);
    }
}
```

11.4.4 StoredProcedure

RDBMS のストアドプロシージャをオブジェクト抽象化したスーパクラスである。このクラスはアブストラクトで execute メソッドは protected であり、より限定された型用のサブクラスから以外から使われるのを防いでいる。

継承された SQL のプロパティは RDBMS のストアドプロシージャの名前だ。JDBC3.0 では名前付パラメータが取り入れられたが、このクラスで提供されている他の機能も JDBC3.0 が必要となる。

オラクルの sysdate()関数を呼び出すプログラムの例をお見せしよう。このストアドプロシージャ機能を使うには、StoredProcedure を拡張したクラスを作らなければならない。これには入力パラメータはなく、SqlOutParameter クラスを使った date として宣言された出力パラメータが 1 つある。execute()メソッドは各エントリに key というパラメータ名を使った出力パラメータが宣言されたマップを返す。

```
import java.sql.Types;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

import javax.sql.DataSource;

import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.datasource.*;
import org.springframework.jdbc.object.StoredProcedure;

public class TestStoredProcedure {

    public static void main(String[] args)  {
        TestStoredProcedure t = new TestStoredProcedure();
        t.test();
        System.out.println("Done!");
    }

    void test() {
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName("oracle.jdbc.OracleDriver");
        ds.setUrl("jdbc:oracle:thin:@localhost:1521:mydb");
        ds.setUsername("scott");
        ds.setPassword("tiger");

        MyStoredProcedure sproc = new MyStoredProcedure(ds);
        Map res = sproc.execute();
        printMap(res);
```

```
    }

    private class MyStoredProcedure extends StoredProcedure {
        public static final String SQL = "sysdate";

        public MyStoredProcedure(DataSource ds) {
            setDataSource(ds);
            setFunction(true);
            setSql(SQL);
            declareParameter(new SqlOutParameter("date", Types.DATE));
            compile();
        }

        public Map execute() {
            Map out = execute(new HashMap());
            return out;
        }

    }

    private static void printMap(Map r) {
        Iterator i = r.entrySet().iterator();
        while (i.hasNext()) {
            System.out.println((String) i.next().toString());
        }
    }
}
```

11.4.5 SqlFunction

結果を 1 行だけ返すクエリ用 SQL の"関数"ラッパである。デフォルトでは **int** を返す振る舞いになっているが、特殊な型のパラメータを返すメソッドを使ってオーバライドすることができる。これは、**JdbcTemplate** で **QueryForXxx** メソッドを使うのと似たものである。**SqlFunction** の利点は、**JdbcTemplate** の生成はバックグランドで実行されるので、自分で生成する必要がない、ということだ。

このクラスは、**"select user()"**や、**"select sysdate from dual"**のようなクエリを使って結果を 1 つだけ返す SQL 関数を呼び出すのに使うためのものだ。より複雑なストアド関数の呼び出しやストアドプロシージャやストアド関数を起動するための **CallableStatement** を使うためののものではない。このような処理を行うには、**StoredProcedure** や **SqlCall** を使うこと。

これは具象クラスで、通常はサブクラスを作る必要はない。このパッケージを使ったコードは、この型のオブジェクトを生成したり、SQL やパラメータを宣言でき、従って関数を実行するために適切な **run** メソッドを繰り返し起動することができる。テーブルの行数を問い合わせる例をお見せしよう。

```
    public int countRows() {
        SqlFunction sf = new SqlFunction(dataSource, "select count(*) from
mytable");
        sf.compile();
        return sf.run();
    }
```

12. O/R マッピングを用いたデータアクセス(Ver 1.2.7)

12.1 イントロ

Spring では、リソース管理、DAO 実装のサポート、トランザクションストラテジといった観点で Hibernate や JDO、Oracle TopLink、Apache OJB や iBATIS SQL Maps とのインテグレーションを提供している。例えば Hibernate では、多くの典型的な Hibernate 統合上の問題に注力した IoC の便利な機能をサポートしている。これらのサポートはすべて Spring の汎用トランザクションや DAO 例外階層に従う O/R マッピングをまとめたものである。統合の方法としては通常、Spring DAO の「テンプレート」を使うか、あるいは直接 Hibernate/JDO/TopLink/等の API を使わずに DAO を実装するかの 2 パターンがある。どちらの場合でも、DAO は依存性注入により設定され、Spring のリソースやトランザクション管理に組み入れられる。

Spring を追加することでデータアクセスのアプリケーションを作成するのに、O/R マッピングレイヤを使うと決めた場合には十分なサポートが用意されている。まず第一に、一旦 Spring で O/R マッピングサポートを使い始めると、すべてを行う必要はない、ということを知っておくべきだ。どの程度であろうとも、組織内で似たようなインフラの構築に注力し、リスクを冒すと決定する前に、Spring のやり方を検討し、ヘッジしてもよいのだ。全てが 1 つの再利用可能な JavaBeans のセットとして設計されているので、いかなるテクノロジを使っていようが、多くの O/R マッピングのサポートはライブラリスタイルで使ってもよいのである。設定とデプロイが容易になる、という点で、ApplicationContext 内部を使うことはさらなる利点がある。そのため、本セクション中の例のほとんどは、ApplicationContext 内部の設定に関するものをご覧いただいている。

自前の O/R マッピングの DAO を作成するのに Spring を使うと、下記のような利点がある:

- テストが簡単。Spring の IoC 方式は実装を差し替えたり、Hibernate の SessionFactory インスタンスや JDBC DataSource、トランザクションマネージャ、(必要であれば)マッパーオブジェクトの実装の場所を設定するのが簡単になる。これにより永続性に関するコードの各部を切り離して、単独で試験するのがとても簡単になる。
- 共通のデータアクセス例外。Spring では各自が選択した O/R マッピングツールから例外をラップして、それをプロプライエタリな(つまり潜在的にチェックされた)例外を共通のランタイム DataAccessException 階層に変換することができる。これによりリカバーできない永続性例外のほとんどを適切なレイヤだけで catch/Throw の決まり文句や例外の宣言をいちいち書かなくてもハンドリングすることができる。また必要であればどこででも例外をトラップし、ハンドリングすることも可能だ。JDBC 例外(各 DB に特有のものを含む)も同じ階層に変換可能であり、これはつまり、一貫したプログラミングモデルで JDBC を使った操作が可能である、というだということを忘れないで欲しい。

- 汎用のリソース管理。Spring のアプリケーションコンテキストは Hibernate の sessionFactory インスタンスや JDBC DataSource、iBATIS SQL MAPs の設定オブジェクト、あるいはその他の関係のあるリソースの位置や設定内容をハンドリングすることができる。このため、その値を管理したり変更するのが容易である。Spring では効果的に簡単に安全に、永続リソースをハンドリングすることができる。例えば:Hibernate を用いた関係のあるコードは効果的に適切なトランザクション処理を行うには通常同じ Hibernate の Session を必要とする。Spring では、明示的な 'template'ラッパクラスを Java のコードレベルで使おうと、(生の Hibernate3 の API を基盤とした DAO 用に)カレントの Session を Hibernate の SessioNFactory から取り出して使おうと、簡単にかつ透過的に Session を生成し、既存のスレッドにバインドすることができる。したがって、Spring では任意のトランザクション環境(ローカルであろうと、JTA であろうと)典型的な Hibernate の使い方をする中で頻繁に発生する多くの問題を解決する。
- 統合されたトランザクション管理。Spring では、Java のコードレベルで、宣言的に、あるいは AOP スタイルのメソッドインタセプタ、明示的な'template'ラッパクラスのいずれかを使って自分で書いた O/R マッピングコードをラップすることができる。どの方法ででも、トランザクションのセマンティクスはハンドリング可能で、例外が発生した場合には適切なトランザクション処理(ロールバック等)が行われる。上述したように、Hibernate/JDO 関連のコードを変更しなくても、いろんなトランザクションマネージャを使ったり、差し替えたりできるという利点が得られる。例えば、ローカルトランザクションと JTA とで、どちらのシナリオででも(宣言的トランザクションのような)同じフルサービスを使うことができる。これはバッチ処理や BLOB のストリーミングのような、O/R マッピングの操作で共通のトランザクションをシェアする必要があるが O/R マッピングに適さないようなデータアクセスに便利だ。
- ベンダロックインを回避し、組み合わせによる実装戦略を可能にする。Hibernate は強力で、柔軟で、オープンソースであり、フリーではあるが、プロプライエタリな API を使うことも可能だ。さらに、複雑な O/R マッピング戦略を必要としないアプリケーションで使う分には iBATIS は素晴らしい、しかも軽量であると主張することができた。機能的な理由や、パフォーマンス、あるいは何か他の理由により、他の実装に切り替える必要が発生した場合、もし選択肢があれば、通常標準の抽象化された API を使ってメジャーなアプリケーションの機能を実装するのが望ましい。例えば、データアクセス機能を実装するマッパ/DAO オブジェクトが簡単に取り替えられる IoC アプローチと同様、Spring による Hibernate トランザクションと例外の抽象化により、Hibernate に特化したコードを、Hibernate の能力を制限することなく、アプリケーションの中で一箇所に集めておくことが簡単にできる。DAO を扱うそれよりも上位レイヤサービスのコードではその実装についてなんら意識する必要もない。このアプローチでは、さらにレガシーコードを使い続けたり、各テクノロジの能力をレバレッジするという点で潜在的に大きな利点のある非侵略的な方法で、mix-and-match アプローチ(つまり、あるデータアクセスでは Hibernate を使って、あるものはや JDBC、また別のところでは iBATIS を使ったもの)を使って簡単にデータアクセスを実装することができるという利点もある。
- 汎用のリソースマネジメント。Spring のアプリケーションコンテキストは Hibarnate のセッションファクトリや、JDBC のデータソース、iBatis の SQL マップコンフィグレーションオブジェクトや関連する他のリソースのロケーションとコンフィグレーションを扱うことができる。これにより簡単に管理、

変更を行うことができるという価値が得られる。Spirng では、Hibernate セッションを容易に、しかも安全にハンドリングが可能になる。Hibernate を用いた関連するコードは通常、適切にトランザクション処理を行うためには、Hibernate の同じ Session オブジェクトを使う必要がある。Spring では、宣言的 AOP メソッドインタセプタアプローチか、あるいは Java のコードレベルで明示的なテンプレートラッパクラスのどちらかを使って、簡単にセッションを生成し、カレントのスレッドにバインドすることができる。よって、Spring は、Hibernate のフォーラムで繰り返し持ち上がる用法上の問題の多くを解決できるのだ。

- 例外のラッピング。Spring では、あなたが選択した O/R マッピングツールから例外をラッピングし、プロプライエタリなチェック済み例外から一連の抽象的なランタイム例外に変換することができる。これにより、適切なレイヤでしかキャッチできないほとんどの永続例外を、ありきたりの catch/throw 句や例外の宣言を使うことなくハンドリングすることができる。また、必要なところであれば、どこででも例外を捕まえて処理することもできる。覚えておいて欲しいのは、(DB に特化した方言も含めた)JDBC 例外も同じ階層に変換される、つまり、一貫したプログラミングモデルで、一貫したやり方で JDBC を操作できるということだ。
- 統合されたトランザクション管理。Spring では、あなたの O/R マッピングコードを、宣言的 AOP スタイルのメソッドインタセプタか、あるいは Java コードレベルの明示的な「テンプレート」ラッパクラスのどちらかでラップすることができる。どちらの場合でも、トランザクションのセマンティクスをハンドリングでき、例外を処理する場合には適切なトランザクション処理(ロールバックとか)を行うことができる。後述するように、Hibernate に関係するコードに影響を与えずに、様々なトランザクションマネージャを使ったり、差し替えたりできるようになる、という利点もある。また、JDBC に関するコードも完全に O/R マッピングを行うために使うコードに透過的に統合することができる。これは、例えば Hibernate あるいは iBatis で実装されていない機能を扱うのに役に立つ。

12.2 Hibernate

12.2.1 リソース管理

典型的なビジネスアプリケーションでは、繰り返し現れるリソース管理コードで分断されることがある。多くのプロジェクトがこの問題のための独自の解決策を発明しようとし、時々プログラミング上の便宜のために、失敗の適切な処理を犠牲にしている。Spring ではリソースをハンドリングするためのとてもシンプルなソリューションを主張している。テンプレートを用いた IoC、つまりコールバックインタフェースを備えたインフラクラス、あるいは、AOP インタセプタの適用だ。このインフラは適切なリソースハンドリングや、特定の API の例外を未チェックのインフラ例外階層へ適切に変換を行うためのものだ。Spring では(任意のデータアクセスストラテジに適用可能な)DAO 例外階層を導入する。直接 JDBC を叩くために、前のセクションで言及されている JdbcTemplate クラスがコネクションをハンドリングし、SQLException を DataAccessException 階層に適切にマッピングする。これには、データベース特有の SQL エラーコードを意味のある例外クラスへの翻訳が含まれている。これは、それぞれの Spring トランザクションマネージャにより JTA トランザクションと JDBC トランザクションのどちらもサポートされる。Spring では、他にも JdbcTemplate に類似した HibernateTemplate / JdoTemplate や、HibernateInterceptor / JdoInterceptor、Hibernate/JDO トランザクションマネージャから構成され、Hibernate や JDO のサポー

トも提供されている。主たるゴールは、任意のデータアクセスとトランザクションテクノロジを使いつつ、リソースルックアップをハードコーディングせず、シングルトンを強要せず、独自のサービスレジストリを使わずにアプリケーションをきれいに階層化できるようにすることである。アプリケーションオブジェクトをシンプルに、一貫性をもって書き上げる方法の１つは、それを再利用可能で、コンテナに依存させないように可能な限り保つことである。個々のデータアクセス機能はすべて個々に利用可能であるが、Spring のアプリケーションコンテキストのコンセプトにうまく統合されており、XML ベースのコンフィグレーションとプレーンな JavaBean インスタンス間の Spring に依存しない相互参照が提供されている。典型的な Spring のアプリでは、データアクセステンプレート、データアクセスオブジェクト(これは、テンプレートを使う)、トランザクションマネージャ、ビジネスオブジェクト(これはデータアクセスオブジェクトとトランザクションマネージャを利用する)、ウェブビューリゾルバ、ウェブコントローラ(これは、ビジネスオブジェクトを利用する)など、重要なオブジェクトの多くは JavaBeans である。

12.2.2 アプリケーションコンテキストでのリソース定義

アプリケーションオブジェクトをハードコーディングされたリソースのルックアップにくくりつけるのを避けるために、Spring では、JDBC の DataSource や、あるいは Hibernate の SessionFactory のようにアプリケーションコンテキスト中のビーンとしてリソースを定義できるようになっている。リソースにアクセスする必要があるアプリケーションオブジェクトはあらかじめ定義されたインスタンスへの参照をビーンリファレンスから受けるだけだ(次のセクションの DAO 定義でこのことを示している)。下記の XML アプリケーションコンテキスト定義から抜粋したものは、冒頭で JDBC DataSource と Hibternate の SessionFactory セットアップの方法を示したものだ。

```
<beans>

<bean id="myDataSource" ¥
    class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName">
            <value>java:comp/env/jdbc/myds</value>
        </property>
    </bean>

<bean id="mySessionFactory" ¥
    class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
        <property name="mappingResources">
            <list>
                <value>product.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <props>
<prop key="hibernate.dialect">net.sf.hibernate.dialect.MySQLDialect</prop>
            </props>
```

```
        </property>
        <property name="dataSource">
            <ref bean="myDataSource"/>
        </property>
    </bean>

    ...

</beans>
```

Jakarta Commons の DBCP BasicDataSource のように JNDI に配置された DataSource をローカルに定義されたものに切り替えているのは、コンフィグレーションのためだけのものである。

```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" ¥
    destroy-method="close">
    <property name="driverClassName">
        <value>org.hsqldb.jdbcDriver</value>
    </property>
    <property name="url">
        <value>jdbc:hsqldb:hsql://localhost:9001</value>
    </property>
    <property name="username">
        <value>sa</value>
    </property>
    <property name="password">
        <value></value>
    </property>
</bean>
```

他にも JNDI に配置された SessionFactory も利用することができるが、これは実際には EJB コンテキストの外側を必要としない(「コンテナリソース VS ローカルリソース」のセクションでの議論を参照してほしい)。

12.2.3 Inversion of Control; テンプレートとコールバック

テンプレートを用いた基本的なプログラミングモデルでは任意のカスタムデータアクセスオブジェクトやビジネスオブジェクトの一部を成せるようにする方法は下記のようになる。周囲を取り巻くオブジェクトには実装上の制限は一切なく、Hibernate の SessionFactory を提供する必要があるだけだ。これはどこからでも取得できるが、単純な setSessionFactory ビーンプロパティセッターを使って Spring のアプリケーションコンテキストからビーンリファレンスとして取得するのが望ましい。下記の断片では、Spring のアプリケーションコンテキストにおける、先に定義した SessionFactory を参照する DAO の定義と、DAO のメソッドの実装例を示したものである。

```
<beans>
```

```
    <bean id="myProductDao" class="product.ProductDaoImpl">
        <property name="sessionFactory">
            <ref bean="mySessionFactory"/>
        </property>
    </bean>

    ...

</beans>
public class ProductDaoImpl implements ProductDao {

    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    public List loadProductsByCategory(final String category) {
        HibernateTemplate hibernateTemplate =
            new HibernateTemplate(this.sessionFactory);

        return (List) hibernateTemplate.execute(
            new HibernateCallback() {
                public Object doInHibernate(Session session) throws
HibernateException {
                    List result = session.find(
                        "from test.Product product where product.category=?",
                        category, Hibernate.STRING);
                    // do some further stuff with the result list
                    return result;
                }
            }
        );
    }
}
```

任意の Hibernate のデータアクセス用ではコールバックの実装が有効だ。HibernateTemplate は Session が適切に、オープン/クローズされていることを保証し、自動的トランザクションに適応させる。このテンプレートインスタンスはスレッドセーフ、再利用可能であり、周囲のクラスのインスタンス変数として保持する。単一の find、load、savaOrUpdate あるいは delete の呼び出しのような 1 ステップのアク

ション用には、HibernateTemplate では、1 行のコールバック実装のような置き換えが可能な別の便利なメソッドが用意されている。さらに、Spring では、SessionFactory を受け取るための getSessionFactory、またサブクラスで利用される getHibernateTemplatesetSessionFactory メソッドが用意されている便利な HibernateDaoSupport ベースクラスが用意されている。これらを組み合わせれば、典型的な要件に対しては、とても単純な DAO 実装ができるようになる。

```
public class ProductDaoImpl extends HibernateDaoSupport implements ProductDao {

    public List loadProductsByCategory(String category) {
        return getHibernateTemplate().find(
            "from test.Product product where product.category=?", category,
            Hibernate.STRING);
    }
}
```

12.2.4 テンプレートの代わりに AOP インタセプタを適用する

HibernateTemplate を用いる別の方法は、コールバック実装を try/catch ブロックを委譲するストレートな Hibernate コードや、アプリケーションコンテキスト中の個々のインタセプタコンフィグレーションにおきかえる Spring の AOP HibernateInterceptor だ。下記の断片では、個々の DAO,インタセプタ、Spring アプリケーションコンテキストのプロキシ定義、DAO メソッド実装の例を示したものである。

```
<beans>

    ...

    <bean id="myHibernateInterceptor"
        class="org.springframework.orm.hibernate.HibernateInterceptor">
        <property name="sessionFactory">
            <ref bean="mySessionFactory"/>
        </property>
    </bean>

    <bean id="myProductDaoTarget" class="product.ProductDaoImpl">
        <property name="sessionFactory">
            <ref bean="mySessionFactory"/>
        </property>
    </bean>

<bean id="myProductDao" ¥
    class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyInterfaces">
            <value>product.ProductDao</value>
```

```
        </property>
        <property name="interceptorNames">
            <list>
                <value>myHibernateInterceptor</value>
                <value>myProductDaoTarget</value>
            </list>
        </property>
    </bean>

    ...

</beans>
public class ProductDaoImpl extends HibernateDaoSupport implements ProductDao {

    public List loadProductsByCategory(final String category) throws MyException {
        Session session = SessionFactoryUtils.getSession(getSessionFactory(), false);
        try {
            List result = session.find(
                "from test.Product product where product.category=?",
                category, Hibernate.STRING);
            if (result == null) {
                throw new MyException("invalid search result");
            }
            return result;
        }
        catch (HibernateException ex) {
            throw SessionFactoryUtils.convertHibernateAccessException(ex);
        }
    }
}
```

このメソッドは、スレッドにバインドされた Session を開いて、メソッドを呼んだあとに、閉じるといったことをするのに HibernateInterceptor を使うだけでうまくいくだろう。getSession の「false」フラグは Session がすでに存在するということを保証する。さもなければ、見つからない場合 SessionFactoryUtils が新しいセッションを生成する。SessionHolder がスレッドにすでにバインドされていれば、例えば、HibernateTransactionManager のトランザクションによって SessionFactoryUtils がどのような場合でも自動でトランザクションに参加する。HibernateTemplate は同じインフラの SessionFactoryUtils を内部的に利用する。HibernateInterceptor の主たる利点は、任意のチェック済みアプリケーション例外が、HibernateTemplate がコールバックつき未チェック例外に限定されていてもデータアクセスのコードで投げられるようになる、ということだ。でも、注意していただきたいことは、ア

プリケーソン例外をそれぞれチェックして投げるのをコールバックの後に引き伸ばすことができる、ということだ。インターセプタの一番の欠点は、コンテキストで特別な設定を必要とすることだ。HibernateTemplate の便利なメソッドがあれば、多くのシナリオに向けたより単純な手段が得られる。

12.2.5 プログラムによるトランザクション宣言

このような低レベルのデータアクセスサービス上では、トランザクションは、どんなオペレーションの数も測ってアプリケーションのより上位のレベルで区別することができる。その周囲のビジネスオブジェクト実装上にも制限はなく、Spring の PlatformTransactionManager を必要とするだけだ。再度、後者は、どこからでも取得することができるが、setTransactionManager メソッドを使ってビーンリファレンスとして取得するのが望ましい － productDAO が setProductDao メソッドから設定しないといけないのとちょうど同じだ。下記のソース片は、Spring のアプリケーションコンテキスト中のトランザクションマネージャとビジネスオブジェクトの定義とビジネスメソッドの実装例を示したものである。

```
<beans>

    ...

    <bean id="myTransactionManager"
class="org.springframework.orm.hibernate.HibernateTransactionManager">
        <property name="sessionFactory">
            <ref bean="mySessionFactory"/>
        </property>
    </bean>

    <bean id="myProductService" class="product.ProductServiceImpl">
        <property name="transactionManager">
            <ref bean="myTransactionManager"/>
        </property>
        <property name="productDao">
            <ref bean="myProductDao"/>
        </property>
    </bean>

</beans>
public class ProductServiceImpl implements ProductService {

    private PlatformTransactionManager transactionManager;
    private ProductDao productDao;

    public void setTransactionManager(PlatformTransactionManager
transactionManager) {
```

```
        this.transactionManager = transactionManager;
    }

    public void setProductDao(ProductDao productDao) {
        this.productDao = productDao;
    }

    public void increasePriceOfAllProductsInCategory(final String category) {
        TransactionTemplate transactionTemplate = new
TransactionTemplate(this.transactionManager);

transactionTemplate.setPropagationBehavior(TransactionDefinition.PROPAGATION_
REQUIRED);
        transactionTemplate.execute(
            new TransactionCallbackWithoutResult() {
                public void doInTransactionWithoutResult(TransactionStatus status)
{
                    List productsToChange =
productDAO.loadProductsByCategory(category);
                    ...
                }
            }
        );
    }
}
```

12.2.6 宣言的トランザクション区分

反対に、トランザクションを区別するコードをアプリケーションコンテキストのインターセプタ設定に置き換えて、Spring の AOP TransactionInterceptor を使うこともできる。これにより、ビジネスオブジェクトに、トランザクションを区別するコードをビジネスメソッドごとに繰り返し書かなくてもよくなる。さらに、プロパゲーションの振る舞いや、独立レベルのようなトランザクションのセマンティクスはコンフィグレーションファイルの中で変更でき、ビジネスオブジェクトの実装に影響をあたえない。

```
<beans>

    ...

    <bean id="myTransactionManager"
class="org.springframework.orm.hibernate.HibernateTransactionManager">
        <property name="sessionFactory">
            <ref bean="mySessionFactory"/>
```

```
        </property>
    </bean>

    <bean id="myTransactionInterceptor"
class="org.springframework.transaction.interceptor.TransactionInterceptor">
        <property name="transactionManager">
            <ref bean="myTransactionManager"/>
        </property>
        <property name="transactionAttributeSource">
            <value>
                product.ProductService.increasePrice*=PROPAGATION_REQUIRED
product.ProductService.someOtherBusinessMethod=PROPAGATION_MANDATORY
            </value>
        </property>
    </bean>

    <bean id="myProductServiceTarget" class="product.ProductServiceImpl">
        <property name="productDao">
            <ref bean="myProductDao"/>
        </property>
    </bean>

<bean id="myProductService" ¥
    class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyInterfaces">
            <value>product.ProductService</value>
        </property>
        <property name="interceptorNames">
            <list>
                <value>myTransactionInterceptor</value>
                <value>myProductServiceTarget</value>
            </list>
        </property>
    </bean>

</beans>
public class ProductServiceImpl implements ProductService {

    private ProductDao productDao;
```

```
    public void setProductDao(ProductDao productDao) {
        this.productDao = productDao;
    }

    public void increasePriceOfAllProductsInCategory(final String category) {
        List productsToChange =
this.productDAO.loadProductsByCategory(category);
        ...
    }

    ...

}
```

HibernateInterceptor、TransactionInterceptor がコールバックコードでチェック済みアプリケーション例外を投げられるので、TransactionTemplate は未チェック例外はコールバックでは制限されている。TransactionTemplate は、未チェックアプリケーション例外の場合や、トランザクションがアプリケーションで、(TransactionStatus により)rollback-only にマーキングされている場合には、ロールバックのトリガーとなる。TransactionInterceptor はデフォルトで同じ振る舞いをするが、メソッドごとのロールバックポリシーは設定が可能だ。特に、他に AOP インタセプタが存在しない場合、宣言的トランザクションを設定するための別の方法は、TransactionProxyFactoryBean だ。TransactionProxyFactoryBean は特定のターゲットビーン用に、プロキシ定義自身をトランザクション設定と組み合わせる。これにより、必要な設定が、ターゲットビーン 1 つとプロキシビーン 1 つに減らすことができる。さらに、トランザクションをもつメソッドが定義されているこれらのインタフェースやクラスを指定しなくてもいいのである。

```
<beans>

    ...

    <bean id="myTransactionManager"
class="org.springframework.orm.hibernate.HibernateTransactionManager">
        <property name="sessionFactory">
            <ref bean="mySessionFactory"/>
        </property>
    </bean>

    <bean id="myProductServiceTarget" class="product.ProductServiceImpl">
        <property name="productDao">
            <ref bean="myProductDao"/>
        </property>
```

```
    </bean>

    <bean id="myProductService"
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
        <property name="transactionManager">
            <ref bean="myTransactionManager"/>
        </property>
        <property name="target">
            <ref bean="myProductServiceTarget"/>
        </property>
        <property name="transactionAttributes">
            <props>
                <prop key="increasePrice*">PROPAGATION_REQUIRED</prop>
<prop key="someOtherBusinessMethod">PROPAGATION_MANDATORY</prop>
            </props>
        </property>
    </bean>

</beans>
```

12.2.7 トランザクション管理戦略

TransactionTemplate と TransactionInterceptor は共に、実際のトランザクション処理を、(単一の Hibernate SessionFactory 用に、ThreadLocal Session を使って)や、Hibernate アプリケーション用の JtaTransactionManager(コンテナの JTA サブシステムに委譲する)HibernateTransactionManager になる PlatformTransactionManager インスタンスに委譲する。独自の PlatformTransactionManager の実装を使うこともできる。なので、例えば、アプリケーションデプロイしたら分散トランザクションが必要になった場合に、ネイティブの Hibernate のトランザクション管理を JTA に切り替えるのは、設定上の問題だけとなる。単純に、Hibernate のトランザクションマネージャを Spring の JTA トランザクション実装に置き換えるだけなのだ。汎用のトランザクション管理 API を使うのとちょうど同じように、トランザクションの分離とデータアクセスコードはどちらも変更せずに動作する。複数の Hibernate セッションファクトリをまたぐ分散トランザクションには、複数の LocalSessionFactoryBean 定義をもつトランザクションストラテジとして JtaTransactionManager を組み合わせる。その場合、各 DAO は　各ビーンプロパティに渡される特定の SessionFactory のリファレンスを取得する。もしJDBC データソースに依存しているものが全てトランザクションコンテナにあれば、ビジネスオブジェクトはストラテジとして JtaTransactionManager を使っているのと同じように、特別意識しなくても任意の数の DAO、任意の数のセッションファクトリをまたいだトランザクションを区切ることができる。

```
<beans>

<bean id="myDataSource1" ¥
    class="org.springframework.jndi.JndiObjectFactoryBean">
```

```
        <property name="jndiName">
            <value>java:comp/env/jdbc/myds1</value>
        </property>
    </bean>

<bean id="myDataSource2" ¥
    class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName">
            <value>java:comp/env/jdbc/myds2</value>
        </property>
    </bean>

<bean id="mySessionFactory1" ¥
    class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
        <property name="mappingResources">
            <list>
                <value>product.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <props>
<prop key="hibernate.dialect">net.sf.hibernate.dialect.MySQLDialect</prop>
            </props>
        </property>
        <property name="dataSource">
            <ref bean="myDataSource1"/>
        </property>
    </bean>

<bean id="mySessionFactory2" ¥
    class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
        <property name="mappingResources">
            <list>
                <value>inventory.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <props>
<prop key="hibernate.dialect">net.sf.hibernate.dialect.OracleDialect</prop>
```

```
            </props>
        </property>
        <property name="dataSource">
            <ref bean="myDataSource2"/>
        </property>
    </bean>

    <bean id="myTransactionManager"
        class="org.springframework.transaction.jta.JtaTransactionManager"/>

    <bean id="myProductDao" class="product.ProductDaoImpl">
        <property name="sessionFactory">
            <ref bean="mySessionFactory1"/>
        </property>
    </bean>

    <bean id="myInventoryDao" class="product.InventoryDaoImpl">
        <property name="sessionFactory">
            <ref bean="mySessionFactory2"/>
        </property>
    </bean>

    <bean id="myProductServiceTarget" class="product.ProductServiceImpl">
        <property name="productDao">
            <ref bean="myProductDao"/>
        </property>
        <property name="inventoryDao">
            <ref bean="myInventoryDao"/>
        </property>
    </bean>

    <bean id="myProductService"
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
        <property name="transactionManager">
            <ref bean="myTransactionManager"/>
        </property>
        <property name="target">
            <ref bean="myProductServiceTarget"/>
        </property>
```

```
        <property name="transactionAttributes">
            <props>
                <prop key="increasePrice*">PROPAGATION_REQUIRED</prop>
<prop key="someOtherBusinessMethod">PROPAGATION_MANDATORY</prop>
            </props>
        </property>
    </bean>

</beans>
```

HibernateTransactionManager と JtaTransactionManager はどちらも Hibernate で適切な JVM レベルのキャッシュをハンドリングすることができる。- (トランザクションを初期化するのに、EJB を使わないのと同じように)コンテナ依存のトランザクションマネージャによるルックアップや、JCA コネクションを使わずに。加えて、HibernateTransactionManager は、Hibernate で使われる JDBC コネクションを生の JDBC アクセスコードに見せることもできる。これにより、JTA を使わなくても、1 つのデータベースにアクセスするのと同じように、Hibernate/JDBC 混成のデータアクセスを完全に高いレベルでトランザクションを分離することができるようになる。

なお、宣言的にトランザクションを区切るために TransactionProxyFactoryBean を使う別の方法については、節[8.5.2 BeanNameAutoProxyCreator, 別の宣言的アプローチ]を参照してほしい。

12.2.8 コンテナリソース vs ローカルリソース

Spring のリソース管理により、JNDI の SessionFactory と、ローカルの SessionFactory を JNDI の DataSource と同じくアプリケーションコードを 1 行も変更せずに、簡単に交換ができるようになる。リソース定義をコンテナに入れるか、あるいはローカルでアプリケーションと一緒にするかは、利用するトランザクションストラテジ次第だ。Spring で定義されたローカルの SessionFactory と比較すると、マニュアルで登録した JNDI の SessionFactory には何も利点がない。もし、Hibernate の JCA コネクタを使って登録すると、特に EJB 内で、透過的に JTA トランザクションに参加させることができるという利点がある。Spring におけるトランザクションサポートに関する重要な利点は、全くコンテナに依存しない、という点だ。JTA 以外の任意のストラテジが設定されていると、スタンドアロンで、あるいはテスト環境でもうまく動作する。特に、典型的な単一のデータベーストランザクションの場合、とても軽量で、強力な JTA の代用となる。トランザクションを動作させるのに、ローカルで EJB のステートレスセッションビーンを使うと、EJB コンテナと JTA の両方に依存してしまう。- たとえ、たった 1 つのデータベースにしかアクセスしていない、CMT を使う宣言的トランザクションのためだけにしか SLSB を使わないとしても。JTA をプログラム的に使う代わりに、J2EE 環境も必要になる。JTA、もしくは JNDI DataSource という観点では、JTA はコンテナへの依存性はない。非 Spring の JTA 駆動の Hibernate トランザクションでは、適切な JVM レベルでキャッシングを効かすためには、Hibernate の JCA コネクタ、あるいは特別な Hiberante トランザクションコードを設定された JTATransaction と一緒に使う必要がある。Spring 駆動のトランザクションは、ローカルの JDBC DataSource を使うのと同じように(もちろん、単一のデータベースにアクセスする場合)ローカルで定義された Hibernate の SessionFactory でうまく動かすことができる。したがって、実際に分散トランザクションの必要性に直面したら、Spring の JTA トランザクションストラテジに帰着せざるを得ない。JCA コネクタはコンテナに特有のデプロイ手順、JCA がサポートされていることがまず最初に

明らかに必要である、という点に注意いただきたい。これは、ローカルのリソース定義と Spring 駆動トランザクションを使った単純なウェブアプリをデプロイするよりもはるかに大変だ。また、例えば WebLogic Express が JCA を提供していないように、エンタープライズ版のコンテナを用意する必要があることもある。ローカルリソースと 1 つの単一データベースにまたがるトランザクションを使う Spring アプリは(JTA や JCA、あるいは EJB なしで)、Tomcat や Resin あるいは生の Jetty でもどんな J2EE ウェブコンテナで動かすことができる。さらに、このようなミドル層はデスクトップアプリケーションや、テストスイートで簡単に再利用することができる。 これで、全て検討した:もし EJB を使わないのであれば、ローカルの SessionFactory セットアップと Spring の HibernateTransactionManager あるいは JtaTransactionManager を使うことになる。これで、コンテナをデプロイする手間をかけなくても適切なトランザクションの JVM レベルのキャッシングと分散トランザクションを含んだ全ての利点が得られる。JCA コネクタを使った Hibernate の SessionFactory の JNDI に登録するのは、EJB を使うの見合った価値しかない。

12.2.9 サンプル

Spring の配布パッケージに同梱されている PetClinic のサンプルには、別の DAO 実装と Hibernate、JDBC、Apache OJB 用アプリケーションコンテキスト設定が用意されている。よって、Petclinic は、Spring ウェブアプリで Hibernate の使い方を示したサンプルアプリとして使える。また、宣言的トランザクションを別のトランザクションストラテジにレバレッジする。

12.3 JDO

12.4 iBATIS

Spring では、org.springframework.orm.ibatis パッケージにより、iBATIS SqlMaps 1.3 と 2.0 がサポートされている。iBATIS では、Hibernate がサポートしているのとよく似ている同じテンプレートスタイルのプログラミングが Hibernate で使うのと同じようにサポートされており、iBatis でも Spring の例外階層がうまく動くようにサポートされているので、Spring が持っている IoC 機能を全て用いることができる。

概要、そして 12.4.1 1.3.x と 2.0 との違い

Spring では、iBATIS SqlMaps 1.3 と 2.0 の両方がサポートされている。まずは、両者の違いについてみてみよう。

iBATIS SqlMaps 1.3,2.0 用サポートクラス		
機能	1.3.x	2.0
SqlMap の生成	SqlMapFactoryBean	SqlMapClientFactoryBean
テンプレートスタイルヘルパークラス	SqlMapTemplate	SqlMapClientTemplate
MappedStatement を使うためのコールバック	SqlMapCallback	SqlMapClientCallback
DAO 用スーパクラス	SqlMapDaoSupport	SqlMapClientDaoSupport

SqlMap のセットアップ

iBATIS SqlMaps を使うと、ステートメントとリザルトマップを含んだ SqlMap のコンフィグレーションファイルが生成される。Spring では SqlMapFactoryBean、あるいは SqlMapClientFactoryBean を使ってこれらを読み込むことができる。後者は、SqlMaps 2.0 と一緒に利用される。

```
public class Account {
	private String name;
	private String email;

	public String getName() {
		return this.name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public String getEmail() {
		return this.email;
	}

	public void setEmail(String email) {
		this.email = email;
	}
}
```

このクラスをマッピングしたいとしよう。おそらく、下記のような SqlMap を作成する必要があるだろう。クエリを使えば、電子メールアドレスを使ってユーザを検索できるようになる。Account.xml はこうなる:

```
<sql-map name="Account">
	<result-map name="result" class="examples.Account">
		<property name="name" column="NAME" columnIndex="1"/>
		<property name="email" column="EMAIL" columnIndex="2"/>
	</result-map>

	<mapped-statement name="getAccountByEmail" result-map="result">
		select
			ACCOUNT.NAME,
			ACCOUNT.EMAIL
		from ACCOUNT
		where ACCOUNT.EMAIL = #value#
	</mapped-statement>
```

```
        <mapped-statement name="insertAccount">
                insert into ACCOUNT (NAME, EMAIL) values (#name#, #email#)
        </mapped-statement>
</sql-map>
```

SqlMap を定義したら、次は iBATIS のコンフィグレーションファイル(sqlmap-config.xml)を作成しなければいけない。

```
<sql-map-config>

        <sql-map resource="example/Account.xml"/>

</sql-map-config>
```

iBATIS はリソースをクラスパスからロードするので、必ずクラスパスのどこかに Account.xml ファイルを追加するように。

Spring を使えば、SqlMapFactoryBean を使って、SqlMap を簡単にセットアップができる。

```
<bean id="sqlMap" class="org.springframework.orm.ibatis.SqlMapFactoryBean">
        <property ¥

name="configLocation"><value>WEB-INF/sqlmap-config.xml</value></property
>
</bean>
```

SqlMapDaoSupport を使う

SqlMapDaoSupport クラスでは HibernateDaoSupport と JdbcDaoSupport 型に似たサポートクラスを提供している。これで DAO を実装してみよう。

```
public class SqlMapAccountDao extends SqlMapDaoSupport implements AccountDao
{

        public Account getAccount(String email) throws DataAccessException {
                return (Account)
getSqlMapTemplate().executeQueryForObject("getAccountByEmail", email);
        }

        public void insertAccount(Account account) throws DataAccessException {
                getSqlMapTemplate().executeUpdate("insertAccount", account);
        }
}
```

As you can see, we're using the SqlMapTemplate to execute the query. Spring has initialized the SqlMap for us using the SqlMapFactoryBean and when setting up the SqlMapAccountDao as follows, you're all set to go:

これを見てわかるように、クエリを実行するために SqlMapTemplate を使っている。Spring は SqlMapFactoryBean を使って SqlMap を初期化した。また、下記のように、SqlMapAccountDao をセットアップするときは、全部設定する。

```
<!-- for more information about using datasource, have a look at the JDBC ¥
    chapter -->
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" ¥
    destroy-method="close">
        <property ¥

name="driverClassName"><value>${jdbc.driverClassName}</value></property>
        <property name="url"><value>${jdbc.url}</value></property>
        <property
name="username"><value>${jdbc.username}</value></property>
        <property
name="password"><value>${jdbc.password}</value></property>
</bean>

<bean id="accountDao" class="example.SqlMapAccountDao">
        <property name="dataSource"><ref local="dataSource"/></property>
        <property name="sqlMap"><ref local="sqlMap"/></property>
</bean>
```

トランザクション管理

アプリケーションに iBATIS を使って宣言的トランザクション管理を追加するのはかなり簡単だ。基本的に、やらないといけないことは、トランザクションマネージャをアプリケーションコンテキストに追加して、例えば TransactionProxyFactoryBean を使って宣言的にトランザクション境界を設定するするだけだ。さらなる詳細は、部[8. トランザクション管理 (Ver 1.2.7)]に書かれている。

(もっと詳細に書くべし！)

13 ウェブ MVC フレームワーク(Ver 1.2.7)

13.1 ウェブ MVC フレームワークのはじめに

Spring のウェブ MVC フレームワークは、アップロードファイルのサポートと同様に、リクエストをハンドラにディスパッチし、ハンドラマッピング、ビュー解決、ロケールやテーマの解決が設定可能な DispatcherServlet を中心に設計されている。このデフォルトハンドラは、ちょうど ModelAndView handleRequest(request,response)メソッドが示しているように、とても単純な Controller インタフェースである。これはすでにアプリケーションコントローラに利用されているが、例えば AbstractController、AbstractCommandController、SimpleFormController から構成される含まれた実装階層を好むだろう。アプリケーションコントローラは通常、これらのサブクラスになる。適切なベースクラスを選択することができ、もしフォームがなければ FormController を必要とはしない、ということに注意して欲しい。これは Struts と大きく異なる点だ。

コマンド、あるいはフォームオブジェクトとしてどんなオブジェクトを用いてもよく、あるインタフェースを実装したり、あるベースクラスから派生させる必要はない。Spring のデータバインディングは柔軟性が高く、例えば、アプリケーションによって評価されるバリデーションエラーのような型のミスマッチをシステムエラーとしてでなく扱う。よって、不正なサブミッションを扱えるようにしたり、String を適切に変換するためだけにビジネスオブジェクトのプロパティをフォームオブジェクトに String として複製する必要はない。その代わり、ビジネスオブジェクトに直接バインドするのが望ましい。これは全てのアクション型に対する Action や ActionForm のような必要とするベースクラスを中心に構築されている Struts とのもうひとつの大きな違いだ。

WebWork と比べて、Spring にはより多くの区別されたオブジェクトの役割がある。Spring では Controller、オプショナルコマンド、フォームオブジェクト、ビューに渡されるモデルの概念をサポートする。このモデルには通常コマンドやフォームオブジェクトだけでなく、任意の参照データも含まれている。それに対し、WebWork の Action では全ての役割が 1 つの単一オブジェクトに組み込まれている。WebWork では、既存のビジネスオブジェクトを自分のフォームとして使うことができるが、それは、そのビジネスオブジェクトが期待する Action クラスのビーンプロパティにしている場合にのみ可能だ。最後に、リクエストを処理するのと同じ Action インスタンスが評価とビューの設置に用いられる。よって、参照データも Action のビーンプロパティとしてモデル化される必要がある。このようにひとつのオブジェクト与えられる役割があまりにも多すぎる。

Spring のビュー解決は非常に柔軟だ。ある Controller の実装ですらビューを直接 ModelAndView として null を返すレスポンスに書き込むことができる。通常、ModelAndView インスタンスはビーン名と(コマンドやフォームのような、参照データを含んでいる)適切なオブジェクトを含んだ、ビュー名とモデルのマップから構成される。ビュー名の解決は、ビーン名から、プロパティファイルから、あるいは自前の ViewResolver の実装から高度に設定が可能だ。この抽象モデルのマップは頑張らなくてもビューテクノロジを完全に抽象化することができる。JSP、Velocity であろうと他のどんなレンダリングテクノロジであ

ろうと、直接統合することが可能だ。このモデルマップはJSPリクエストの属性や、Velocityのテンプレートモデルのような適切なフォーマットに単純に変換される。

13.1.1 他の MVC 実装をプラグインする

他の MVC 実装を好んで使うプロジェクトがいくつかあるのには理由がある。スキルやツールへのそれまでのの投資が生かせることを期待するチームはとても多い。また、Struts フレームワークに関する知識や経験がとても豊富だ。なので、もし Struts のアーキテクチャ上の欠点を受け入れることができるのであれば、Struts は今までどおりWeb 層における有効な選択肢となり得るのだ。同じことがWebWorkやその他のウェブ MVC フレームワークにも当てはまる。

もし、Spring のウェブ MVC は使いたくない、でも Spring で利用可能な他の解決策を使いたいというのであれば、その自分で選んだウェブ MVC フレームワークを Spring と簡単に統合することができる。Spring のルートアプリケーションコンテキストをContextLoaderListenerから起動し、ServletContext 属性(あるいは、Springの個々のヘルパーメソッド)を使ってStrutsやWebWorkのアクションからアクセスするだけだ。ここで注意して欲しいのは、"プラグイン"ではないので、それ専用に統合する必要は全くないということだ。ウェブ層の観点から、ルートアプリケーションコンテキストインスタンスをエントリポイントとする Spring をライブラリとして使うだけだ。

あなたが登録したビーンや Spring のサービスは全て Spring のウェブ MVC がなくてもあなたの思うがままだ。Spring はこのシナリオの中ではStrutsやWebWorkとは競合しない。ビーン設定からデータアクセスやトランザクションのハンドリングまで、純粋なウェブ MVC フレームワークが対象としない多くの分野にも適用できます。なので、例えば、JDBC や Hibernate のトランザクションを抽象化するのに使いたいだけであったとしても Spring のミドル層やデータアクセス層を使えば、あなたのアプリケーションをリッチにすることができるのである。

13.1.2 Spring MVC の機能

Spring のウェブモジュールではユニークなウェブサポート機能を豊富に提供している。以下に挙げる:

- ロールが明確に分かれている-コントローラ、バリデータ、コマンドオブジェクト、フォームオブジェクト、モデルオブジェクト、DispatcherServlet、ハンドラマッピング、ビューリゾルバなど。各ロールは専用のオブジェクトによって遂行される。
- フレームワークとアプリケーションクラスのどちらもが JavaBeans として強力で素直な設定。これには、ウェブコントローラからビジネスオブジェクトやバリデータへのようなコンテキストを跨ぐ参照が簡単にできるといったものが含まれている。
- 順応性や、不可侵性。なんでもかんでも単一のコントローラから派生させるのではなくて、シナリオに適したコントローラサブクラス(プレーン、コマンド、フォーム、ウィザード、マルチアクション、あるいは独自のもの)を使うこと。
- ビジネスコードの再利用性-複製は必要ない。既存のビジネスオブジェクトを特定のフレームワークのベースクラスを拡張するためにミラーリングしないでコマンドやフォームオブジェクトとして使うことができる。
- カスタマイズ可能なバインディングとバリデーション-入力された値を保持しておくアプリケーションレベルのバリデーションエラーのような型の不一致や、マニュアル操作のパージングやビジネスオブジェクトへの変換を行う文字列のみの形式のオブジェクトの代わりに、ローカライズされた日付と数字の組み合わせなど

- カスタマイズ可能なハンドラマッピングやビュー解決−ハンドラマッピングやビュー解決のストラテジは、単純なURL ベースの設定から精巧な専用の解決ストラテジまで広範囲に及ぶ。これは特別な技術を押し付けがちなウェブ MVC フレームワークに比べてとても柔軟なものだ。
- 柔軟なモデルの移行−名前/値のマップを用いたモデル移行により任意のビューテクノロジとの統合が容易である。
- カスタマイズ可能なロケールやテーマ解決は Spring のタグライブラリの有無に関わらず JSP、JSTL、特別なブリッジなしで Velocity がサポートされている。
- 単純だが強力なタグライブラリにより、HTML 生成にコストをかけずにマークアップコードの柔軟性を最大限にする。

13.2 DispatcherServlet

Spring のウェブ MVC フレームワークは、多くの他のウェブ MVC フレームワークのように、リクエストドリブンなウェブ MVC フレームワークで、リクエストをコントローラにディスパッチするサーブレットやウェブアプリケーション開発に沿った機能を提供するように設計されている。しかしながら、Spring の DispatcherServlet はそれだけではない。これは、Spring の ApplicationContext と完全に統合されており、Spring の持つ他の全ての機能を使うことが可能だ。

通常のサーブレットと同様、DispatcherServlet はそれぞれのウェブアプリケーションにある web.xml で宣言されている。DispatcherServlet に処理させたいリクエストは、同じ web.xml ファイルの中で URL マッピングを用いて対応づけされていなければならない。

```
<web-app>
    ...
    <servlet>
        <servlet-name>example</servlet-name>
<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>example</servlet-name>
        <url-pattern>*.form</url-pattern>
    </servlet-mapping>
</web-app>
```

上記例では、.form で終わる全てのリクエストは DispatcherServlet にて処理される。ここでこの DispatcherServlet の設定が必要になる。

3.11 章の ApplicationContext の導入で解説したように、Spring における ApplicationContext はスコープになる。このウェブ MVC フレームワークでは、各 DispatcherServlet はそれぞれが WebApplicationContext を持ち、ルートの WebApplicationContext ですでに定義されているビーンに継承される。この継承されたビーン定義はサーブレットごとのスコープの中でオーバライドされ、新しいスコープに特化したビーンが指定されたサーブレットインスタンスのローカルに定義される。

このフレームワークは、DispatcherServlet の初期化時に、そのウェブアプリケーションの WEB-INF ディレクトリから*[servlet 名]-servlet.xml* という名前のファイルを探し、そこに(グローバルスコープにある同じ名前のビーン定義を上書きして)このビーン定義を生成する。

DispatcherServlet によって利用されるコンフィグの場所は、サーブレットの初期化パラメータで変更することができる。(詳細は下記を参照されたい)

この WebApplicationContext は、ウェブアプリケーションに必要な特別な機能がいくつかあることを除いては、通常の ApplicationContext と全く同じだ。通常の ApplicationContext と違う点は、テーマの解決が可能であること(13.7 章、「テーマを使う」を参照して欲しい)と、(ServletContext へのリンクをもつことで)どのサーブレットと関連をもつかを知っていることだ。WebApplicationContext は ServletContext 内部にあり、必要な場合は、RequestCOntextUtils を使って常に WebApplicationContext をルックアップすることができる。

Spring の DispatcherServlet はリクエストを処理し、適切なビューを表示できるように自身が利用する特別なビーンと密接になっている。これらのビーンは Spring フレームワークに含まれており、他のビーンと同じように WebApplicationContext で設定することが可能である。ビーンについては、それぞれ下記に詳細が述べられている。ここでは、どのようなビーンが用意されているか、DistapcherServlet に関しての話題を続けられるようにしよう。ほとんどのビーンでは、デフォルトの振る舞いが用意されているので、どのように設定すればいいか心配する必要はない。

13.1 WebApplicationContext の特別なビーン

表記	説明
ハンドラマッピング	(節[ハンドラマッピング])前処理、あるいは後処理や、(例えば、コントローラに指定された URL に合致するなど)特定の条件にマッチした場合に実行されるコントローラのリスト
コントローラ	(節[13.3 コントローラ]) MVC の一部として、実際の機能(あるいは少なくとも機能にアクセス)を提供するビーン
ビューリゾルバ	(節[ビューとビューの解決])ビュー名からビューを解決する。**DispatcherServlet** で利用される
ロケールリゾルバ	(節[ロケールを使う])国際化されたビューを利用可能にするためにクライアントが利用しているロケールを解決する。
テーマリゾルバ	(節[テーマを使う])ウェブアプリケーションが用いるテーマを解決し、パーソナライズされたレイアウトを提供する
マルチパートリゾルバ	(節[Spring のマルチパート(ファイルアップロード)サポート]) HTML フォームからファイルのアップロード処理機能を提供する
ハンドラ例外リゾルバ	(節[例外をハンドリングする])ビューへの例外のマッピングや他のもっと複雑な例外ハンドリングコードの実装機能を提供する

DispatcherServlet がセットアップされ、リクエストがその特定の DispatcherServlet に来ると、処理が開始される。下記のリストは、リクエストが来て DispatcherServlet によって処理される場合の完全な処理手順である。

1. WebApplicationContext が検索され、コントローラや処理に利用される他の要素で利用するために、リクエストにバインドされる。デフォルトでは、DispatcherServlet.WEB_APPLICATION_CONTEXT_ATTRIBUTE というキーにバインドされる。
2. リクエスト処理中(ビューのレンダリング、データの適用等)に、処理中の要素がロケールを解決するのに使うロケールリゾルバがリクエストにバインドされる。もしリゾルバを使わない場合は何もしないので、もしロケールを解決する必要がない場合はこれを使う必要はない。
3. ビューのような要素がどのテーマを使うかを決定するためにテーマリゾルバがリクエストにバインドされる。もし使わなければ、テーマリゾルバは何もしないので、テーマが必要なければ無視してもよい。
4. マルチパートリゾルバが指定されている場合、リクエストはマルチパートを探し出し、見つかれば、他の要素により後で行われる処理用に MultipartHttpServletRequest にラップする。(マルチパート処理に関する更なる情報については、節[MultipartResolver を使う]を参照のこと)
5. 適切なハンドラが検索される。ハンドラが見つかると、ハンドラ(プリプロセッサ、ポストプロセッサ、コントローラ)に関連づけられた例外のチェーンがモデルを準備するために実行される。
6. モデルが返されると、WebApplicationContext で設定されたビューリゾルバを使ってビューがレンダリングされる。もし、返されるモデルがない場合(これは、プリプロセッサ、ポストプロセッサがリクエストをインターセプトするために生じる。例えば、セキュリティ的な理由による)、ビューはレンダリングされない。

リクエスト処理中に投げられるかもしれない例外は WebApplicationContext 中で宣言されている handlerexception リゾルバにピップアップされる。この例外リゾルバを使えば、その例外が投げられた場合の独自の振る舞いを定義することができる。

Spring の DispatcherServlet には、サーブレット API で指定されている最終更新日を返す機能もサポートされている。特定のリクエストに対し最終更新日を決定する処理手順は単純だ。DispatcherServlet は最初に適切なハンドラマッピングをルックアップし、その見つかったハンドラが ***LastModified*** インタフェースが実装されているかをテストする。実装されていれば、long 型の getLastModified(request)の値がクライアントに返される。

Spring の DispatcherServlet は web.xml ファイルのコンテキストパラメータや、サーブレットの初期化パラメータを追加することで、カスタマイズできる。この方法でカスタマイズできるものを下記に挙げる。

13.2 DispatcherServlet の初期化パラメータ

パラメータ	説明
contextClass	**WebApplicationContext** を実装するクラス。これはコンテキストをインスタンス化するのにこのサーブレットにより用いられる。このパラメータが指定されていない場合は、**xmlWebApplicationContext** が使われる。
contextConfigLocation	コンテキストがどこで見つかったかを示すために(**contextClass** で指定される)コンテキストインスタンスに渡される文字列。この文字列は、(複数のコンテキストロケーション、二重に定義されるビーンの場合には最後のものが取得される)多重コンテキストをサポートするために複数の文字列に(デリミタにカンマを使って)分割される。
namespace	**WebApplicationContext** の名前空間。デフォルトは**[サーバ名]-servlet**。

13.3 コントローラ

コントローラという考え方は、MVC デザインパターンの一部である。コントローラはアプリケーションの振る舞いを定義する、あるいは少なくともアプリケーションの振る舞いへのアクセスを提供する。コントローラはユーザの入力を解釈し、その入力を ビューによってユーザに提示された知覚可能なモデルに変換する。Spring ではコントローラという概念が、様々な種類のコントローラを実装できるようにとても抽象的な方法で実装されている。Spring には、formcontroller、commandcontroller、あるいはウィザード形式のロジックを実行するコントローラなどが含まれている。

Spring のコントローラアーキテクチャの基本は org.springframework.web.servlet.mvc.Controller インタフェースで、下記にソースリストを挙げる

```
public interface Controller {

    /**
     * Process the request and return a ModelAndView object which the
DispatcherServlet
     * will render.
     */
    ModelAndView handleRequest(
        HttpServletRequest request,
        HttpServletResponse response)
    throws Exception;
}
```

ここからわかるように、この Controller インタフェースではリクエストを処理し、適切なモデルやビューを返すことができる単一のメソッドを必要とする。この 3 つの概念、モデルとビュー、そしてコントローラは、Spring の MVC 実装の基礎をなすものである。Controller インタフェースは完全に抽象的なものではあるが、Spring では必要になるであろう、多くの機能を持ったコントローラが多数用意されている。Controller インタフェースリクエストを処理し、モデルやビューを返すという、は全てのコントローラで必要とされる共通的な機能を定義しただけのものだ。

AbstractController と WebContentGenerator

もちろん、コントローラインタフェースだけでは全然十分ではない。基本的なインフラを提供するために、Spring の Controller は全て AbstractController から継承し、キャッシュのサポートや例えば MIME タイプの設定を提供する。

AbstractController で提供される機能	
機能	**説明**
supportedMethods	このコントローラが受け入れるのはどのメソッドかを示す。通常これは、**GET** と **POST** の両方が設定されているが、サポートしたいメソッドを反映するのに修正することができる。もし、サポートされていないリクエストがコントローラに受信されるとクライアントは、(**ServletException** を使って)これが通知される。
requiredSession	このコントローラが動作するのにセッションを必要とするかどうかを示す。この機能は全てのコントローラに提供されている。このようなコントローラがリクエストを受信したときにセッションが存在

AbstractController で提供される機能	
機能	説明
	しない場合、ユーザには **ServletException** を使って通知される。
synchronizeSession	ユーザのセッションで、このコントローラでの処理を同期させたい場合にこれを使う。特に、**handleRequestInternal** メソッドを上書きするコントローラにこの変数を指定すれば、そのコントローラが同期される。
cacheSeconds	HTTP レスポンスでコントローラにキャッシングディレクティブを生成させたい場合にここに正の整数値を指定する。デフォルトでは、キャッシングディレクティブが含まれないように-1 が設定されている。
useExpiresHeader	HTTP 1.0 互換の"Expires"ヘッダを指定するようにコントローラを微調整する。デフォルトでは、変更しなくてもいいように **true** が設定されている。
useCacheHeader	HTTP 1.1 互換の"Cache-Control"ヘッダを指定するようにコントローラを微調整する。デフォルトでは変更しなくてもいいように **true** が設定されている。

最後の 2 つのプロパティは実際には AbstractController のスーパクラスである WebContentGenerator の一部であるが、完全を期するためにここに入れた。

AbstractController を自前のコントローラのベースクラスとして使う場合(あなたがさせたいことを行うかもしれないいろんなコントローラがすでに用意されているのでこれは、おススメしない)、handleRequestInternal(httpServletRequest、httpServletResponse)をオーバライドして、ロジックを実装し、ModelAndView オブジェクトを返すだけだ。ここでは、ウェブアプリケーションコンテキストにあるクラスと宣言からなる短い例を挙げる。

```
package samples;

public class SampleController extends AbstractController {

    public ModelAndView handleRequestInternal(
        HttpServletRequest request,
        HttpServletResponse response)
    throws Exception {
        ModelAndView mav = new ModelAndView("foo");
        mav.addObject("message", "Hello World!");
        return mav;
    }
}
<bean id="sampleController" class="samples.SampleController">
    <property name="cacheSeconds" value="120"/>
</bean>
```

この非常に単純なコントローラを動作させるには、ウェブアプリケーションコンテキスト中の上述のクラスと宣言が、ハンドラマッピング(節[ハンドラマッピング]参照)の設定と合わせて必要なことすべてだ。こ

のコントローラは、再度チェックするまえの 2 分間、キャッシュするようにクライアントに指令するキャッシングディレクティブを生成する。このコントローラは、ハードコーディングされたビュー(あんまりよくない)、名前づけられたインデックス(ビューに関する詳細は節[ビューとビューの解決]を参照のこと)を返す。

他の単純なコントローラ

AbstractController を拡張することは可能だが、Spring では単純な MVC アプリケーションで共通的に利用される機能を持った数多くのコントローラの実装が用意されている。ParameterizableViewController は、返すビューの名前をウェブアプリケーションコンテキストの中で指定できる(つまり、ビュー名をハードコーディングする必要はないよ)ことを除いては基本的には上述の例と同じだ。

UrlFilenameViewController は URL を精査し、リクエストされたファイルのファイル名(http://www.springframework.org/index.html のファイル名は index)を検索し、ビュー名として利用する。これ以上は何もない。

MultiActionController

Spring では複数のアクションを 1 つのコントローラにまとめて、機能をひとまとめにするための多重アクションコントローラが用意されている。この多重アクションコントローラは別個のパッケージ、org.springframework.web.servlet.mvc.multiaction にあり、リクエストをメソッド名にマッピングし、正しいメソッド名を起動することができる。1 つのコントローラ中に、多数の共通的な機能を持たせ、例えば振る舞いを微調整するために複数のエントリポイントをコントローラに設定したい場合は多重アクションコントローラを使うのが手っ取り早い。

MultiActionController で用意されている機能

機能	説明
delegate	**MultiActionController** には利用シナリオが二つある。1 つは **MultiActionController** のサブクラスを作って、**MethodNameResolver** で解決するメソッドをそのサブクラスに指定する(この場合、delegate を指定する必要はない)、あるいは委譲オブジェクトを定義し、**Resolver** で解決されるそのオブジェクトのメソッドが起動される。このシナリオを選ぶ場合は、この設定パラメータをコントローラとして使って委譲を定義する必要がある
methodNameResolver	なんらかの方法で来たリクエストに基づいて **MultiActionController** が起動すべきメソッドを解決する必要がある。この設定パラメータを使ってこの動作を行うリゾルバを定義することができる。

多重アクションコントローラに定義されているメソッドは、下記のシグネチャに従う必要がある:

```
// actionName can be replaced by any methodname
ModelAndView actionName(HttpServletRequest, HttpServletResponse);
```

メソッドのオーバロードは許されていない。MultiActionController を混乱させるからだ。また、指定したメソッドが投げる例外をハンドリングできる例外ハンドラを定義することができる。例外ハンドラメソッド

はちょうど他のアクションメソッドのように ModelAndView オブジェクトを返し、下記のシグネチャに従う必要がある。

```
// anyMeaningfulName can be replaced by any methodname
ModelAndView anyMeaningfulName(HttpServletRequest, HttpServletResponse,
ExceptionClass);
```

この ExceptionClass は java.lang.Exception や java.lang.RuntimeException のサブクラスと同様、任意の例外になりえる。

MethodNameResolver は受信したリクエストに基づくメソッド名を解決するために用意されている。自由に使えるリゾルバが 3 つあるが、もちろん、自分の欲しいものを自分で実装することも可能だ。

- ParameterMethodNameResolver -リクエストパラメータを解決し、メソッド名としてそれを使うことができる。(http://www.sf.net/index.view?testParam=testIt では、testIt(HttpServletRequest,HttpServletResponse)メソッドが呼ばれる)。paramName という設定パラメータには、期待されるパラメータを指定する。
- InternalPathMethodNameResolver -パスからファイル名を検索し、それをメソッド名として使う(http://www.sf.net/testing.view では testing(HttpServletRequest, HttpServletResponse)メソッドが呼ばれる)。
- PropertiesMethodNameResolver -ユーザ定義されたプロパティオブジェクトをメソッド名にマッピングされるリクエスト URL で利用する。このプロパティに/index/welcome.html=doIt が含まれていて、/index/welcome.html へのリクエストを受信した場合、doIt(HttpServletRequest, HttpServletResponse)メソッドが呼ばれる。このメソッド名リゾルバは PathMatcher と一緒に動作する。なので、プロパティに/**/welcome?.html が含まれていれば、これも動作する。

ここにいくつかの例がある。1 つめは、ParameterMethodNameResolver と delegate プロパティの例で、これはリクエストを含むパラメータつきの URL へのリクエストを受け、retrieveIndex に設定される:

```
<bean id="paramResolver" ¥
    class="org....mvc.multiaction.ParameterMethodNameResolver">
  <property name="paramName"><value>method</value></property>
</bean>

<bean id="paramMultiController" ¥
    class="org....mvc.multiaction.MultiActionController">
<property name="methodNameResolver"><ref
bean="paramResolver"/></property>
  <property name="delegate"><ref bean="sampleDelegate"/></property>
</bean>

<bean id="sampleDelegate" class="samples.SampleDelegate"/>
```

下記も一緒に

```
public class SampleDelegate {
```

```
    public ModelAndView retrieveIndex(
        HttpServletRequest req,
        HttpServletResponse resp) {

        return new ModelAndView("index", "date", new
Long(System.currentTimeMillis()));
    }
}
```

上述したような委譲を用いる場合、URL を定義したメソッドにマッチングさせるのに PropertiesMethodNameResolver を使うこともできる:

```
<bean id="propsResolver" ¥
    class="org....mvc.multiaction.PropertiesMethodNameResolver">
  <property name="mappings">
    <props>
      <prop key="/index/welcome.html">retrieveIndex</prop>
      <prop key="/**/notwelcome.html">retrieveIndex</prop>
      <prop key="/*/user?.html">retrieveIndex</prop>
    </props>
  </property>
</bean>

<bean id="paramMultiController" ¥
    class="org....mvc.multiaction.MultiActionController">
<property name="methodNameResolver"><ref
bean="propsResolver"/></property>
    <property name="delegate"><ref bean="sampleDelegate"/></property>
</bean>
```

CommandControllers

Spring の ***CommandController*** *群*は Spring MVC パッケージの基礎的な部分だ。コマンドコントローラはデータオブジェクトとインタラクションし、動的に HttpServletRequest からのパラメータを指定されたデータオブジェクトへバインドする手段を提供する。これらは、Struts の ActionForm と似たような役割をなすが、Spring では、データオブジェクトはフレームワーク向けのインタフェースを実装する必要はない。まず、どんなコマンドコントローラが利用できるか、それを使って何ができるかを概観しよう。

- AbstractCommandController -自前のコマンドコントローラを生成するのに使うことができるコマンドコントローラで、リクエストパラメータを指定したデータオブジェクトにバインドすることができる。このクラスにはフォーム機能は用意されていないが、バリデーション機能や リクエストからのパラメータがセットされたコマンドオブジェクトで行うことをコントローラ自身に指定することができる。

- AbstractFormController -フォームのサブミットをサポートする抽象コントローラである。このコントローラを使って、フォームをモデル化し、コントローラの中で検索したコマンドオブジェクトを使ってフォームを設置することができる。ユーザがフォームに入力した後、AbstractFormController が各フィールドにバインドし、バリデートし、適切なアクションを処理するためにコントローラにオブジェクトを渡す。サポートされている機能は、無効のフォームのサブミット(resumbmission)、バリデーション、通常のフォームのワークフロー。どのビューがフォームの表示に使われるかを決定するメソッドを実装する。もしフォームが必要な場合はこのコントローラを使うこと。但し、どのビューをユーザに見せるかをアプリケーションコンテキスト内で指定することはできない。
- SimpleFormController -対応するコマンドオブジェクトを備えたフォームを生成する場合よりも多くをサポートする具体的な FormController。SimpleFormController にはコマンドオブジェクト、フォームのビュー名、フォームのサブミットがされたときに、ユーザに見せたいページのビュー名、他を指定しよう。
- AbstractWizardFormController -クラス名が示唆するように、これは、あなたの WizardController が継承すべき抽象クラスだ。これはつまり、validatePage()、processFinish、それに processCancel のメソッドを実装しなければならないということだ。

おそらく、少なくとも setPages()や setCommandName()を呼ばないといけない契約を書きたいであろう。前者は、String 型の配列を 1 つ引数にとる。この配列は、ウィザードを含むビューのリストだ。後者は、String 型を 1 つ引数にとる。これは、ビューからコマンドオブジェクトを参照するのに利用される。

AbstractFormController の例のように、コマンドオブジェクト - フォームからデータが挿入される JavaBean - を用いることが必要とされる。これは 2 つあるうちの一方のやり方でできる。一方は、コマンドオブジェクトクラスのコンストラクタから setCommandClass()を呼ぶ、あるいは formBackingObject()メソッドを実装する方法だ。

AbstractWizardFormController には、オーバライドするかもしれない多数の具象メソッドが用意されている。この中で最も有用だと思われるものは、モデルデータを Map の形式でビューに渡すのに使うことができる referenceData、もしウィザードがページの順序を変更する必要がある、あるいは動的にページを省くための getTargetPage、それに、組み込みバインディングとバリデーションのワークフローをオーバライドしたいときのための onBindAndValidate だろう。

最後に、ユーザが今のページでバリデーション違反をしている場合でもウィザードの中で前に戻ったり先に進んだりできるようにする getTargetPage から呼ぶことができる setAllowDirtyBack と setAllowDirtyForward は指摘しておく価値があるだろう。

メソッドの全リストについては、AbstractWizardFormController の JavaDoc を参照してほしい。jPetStore のこのウィザードの実装例は、Spring の配布物:org.springframework.samples.jpetstore.web.spring.OrderFormController に含まれている。

ハンドラマッピング

ハンドラマッピングを使えば、入ってくるウェブのリクエストを適切なハンドラへマッピングできる。例えば、SimpleUrlHandlerMapping や BeanNameUrlHandlerMapping のようにそのまま利用できるハンドラマッピングがいくつか用意されているが、まずは、handlerMapping の一般的な概念を学ぶことにしよう。

基本的な HandlerMapping が提供する機能は、HandlerExceptionChain を配布することだ。これには、入ってくるリクエストに合致するハンドラが含まれていなければならない。またリクエストに応答するハンドラインタセプタのリストも含まれている。リクエストが来ると、DispatcherServlet がハンドラマッピングに渡し、リクエストを識別し、適切な HandlerExecutionChain に対応づける。そして、DispatcherServlet が(もしあれば)このチェインの中でハンドラとインタセプタを実行する。

オプションで、(実際にハンドラが実行された前、あるいは後、もしくはその両方で実行される)インタセプタを含めることができるコンフィグ可能なハンドラマッピングの概念は、とても強力だ。多くの支援機能を独自の HandlerMappings に組み込むことができる。リクエストの URL だけでなく、そのリクエストと関連するセッションの特定の状態に基づいてハンドラを選択する独自のハンドラマッピングを考えてみてほしい。

本セクションでは 2 つの Spring で最も共通的に利用されるハンドラマッピングについて述べる。これは両方とも、AbstractHandlerMapping を拡張したもので、下記のプロパティをともに持つ。

- interceptors:利用するインタセプタのリスト。HandlerInterceptor については、節[HandlerInterceptor を追加する]で議論する。
- defaultHandler:このハンドラマッピングが合致したハンドラにない場合にこのハンドラマッピングがデフォルトで用いるハンドラ。
- order: order プロパティ(org.springframework.core.Ordered インタフェースを参照のこと)の値に基づいて、Spring がコンテキスト中で利用できる全てのハンドラマッピングをソートし、最初にマッチしたハンドラを適用する。
- alwaysUseFullPath:このプロパティが true に設定されていると、Spring はカレントのサーブレットコンテキストで適切なハンドラを見つけるのにフルパスを用いる。このプロパティが false に設定されていると(これはデフォルト)、カレントのサーブレットのマッピングが使われる。例えば、もしサーブレットが/testing/*を使ってマッピングされていて、alwaysUseFullPath プロパティが true に設定されていると、/testing/viewPage.html が用いられる。このプロパティが false に設定されていれば、/viewPage.html が用いられる。
- urlPathHelper:このプロパティを使うと、URL を期待している場合に UrlPathHelper を微調整することができる。通常、このデフォルト値は変更する必要はないはずだ。
- urlDecode:このプロパティのデフォルト値は false である。HttpServletRequest はリクエストの URL やデコード*されていない* URI を返す。HandlerMapping が適切なハンドラを見つけるのに使う前にデコードしておきたい場合、これを true に設定する(これには、JDK 1.4 が必要である点には注意)。このデコーディングメソッドはリクエストで指定されたエンコーディングかもしくは、デフォルトの ISO-8859-1 エンコーディングスキームを用いる。
- lazyInitHandlers: *シングルトン*ハンドラの遅延初期化を行う(プロトタイプハンドラは常に遅延初期化が行われる)。デフォルト値は false。

(注:最後の 4 つのプロパティは *org.springframework.web.servlet.handler.AbstractUrlHandlerMapping* のサブクラスでしか利用できない。)

BeanNameUrlHandlerMapping

とても単純で、しかしとても強力はハンドラマッピングは BeanNameUrlHandlerMapping で、これは、受信する HTTP リクエストをウェブアプリケーションコンテキストで定義されているビーン名にマップする。ユーザがアカウントを入力するのを可能にしたくて、フォームをレンダリングするための適切な FormController(Command と FormController に関する詳細については mvc-controller-command_ja.html を参照してほしい)や JSP のビュー(あるいは Velocity のテンプレート)がすでに提供されている。BeanNameUrlHandlerMapping を使う場合、http://samples.com/editaccount.form の URL の HTTP リクエストを以下のように、適切な FormController にマップする。

```
<beans>
<bean id="handlerMapping" ¥

class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/> ¥

<bean name="/editaccount.form" ¥
    class="org.springframework.web.servlet.mvc.SimpleFormController">
    <property name="formView"><value>account</value></property>
    <property
name="successView"><value>account-created</value></property>
    <property name="commandName"><value>Account</value></property>
    <property
name="commandClass"><value>samples.Account</value></property>
  </bean>
<beans>
```

/editaccount.form の URL へのリクエストは全て、ここでは上述したソースリストにある FormController で処理される。もちろん、.form で終わるリクエスト全てに通用させるために、同じように、web.xml にサーブレットマッピングを定義しないといけない。

```
<web-app>
    ...
    <servlet>
        <servlet-name>sample</servlet-name>
<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

   <!-- Maps the sample dispatcher to /*.form -->
    <servlet-mapping>
        <servlet-name>sample</servlet-name>
        <url-pattern>*.form</url-pattern>
```

```
    </servlet-mapping>
    ...
</web-app>
```

注:もし BeanNameUrlHandlerMapping を使いたい場合は、(上述したように)ウェブアプリケーションコンテキストに定義する必要はない。デフォルトでは、ハンドラマッピングがコンテキスト中に見つからなかった場合、DispatcherServlet があなたのために BeanNameUrlHandlerMapping を生成する。

SimpleUrlHandlerMapping

さらに(あるいははるかに)強力なハンドラマッピングは、SimpleUrlHandlerMapping だ。このマッピングは、アプリケーションコンテキストの中で設定ができて、Ant スタイルのパスマッチングの機能を持っている(org.springframework.util.PathMatcher を参照のこと)。例を挙げよう:

```
<web-app>
    ...
    <servlet>
        <servlet-name>sample</servlet-name>
<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <!-- Maps the sample dispatcher to /*.form -->
    <servlet-mapping>
        <servlet-name>sample</servlet-name>
        <url-pattern>*.form</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>sample</servlet-name>
        <url-pattern>*.html</url-pattern>
    </servlet-mapping>
    ...
</web-app>
```

.html や.form で終わるリクエストは全て sample ディスパッチャサーブレットで処理される。

```
<beans>
    <bean id="handlerMapping"
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <props>
<prop key="/*/account.form">editAccountFormController</prop>
<prop key="/*/editaccount.form">editAccountFormController</prop>
                <prop key="/ex/view*.html">someViewController</prop>
                <prop key="/**/help.html">helpController</prop>
```

```
            </props>
        </property>
    </bean>

    <bean id="someViewController"
class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>

    <bean id="editAccountFormController"
          class="org.springframework.web.servlet.mvc.SimpleFormController">
        <property name="formView"><value>account</value></property>
<property name="successView"><value>account-created</value></property>
        <property name="commandName"><value>Account</value></property>
<property name="commandClass"><value>samples.Account</value></property>
    </bean>
<beans>
```

このハンドラマッピングは任意のディレクトリの help.html へのリクエストを UrlFilenameViewController である helpController にルーティングする(コントローラに関しての詳細は、節[13.3 コントローラ]にある)。view ではじまるリソースへのリクエストや、ディレクトリ ex での.html で終わるリクエストは someViewController にルーティングされる。2 つのさらなるマッピングが editAccountFormController 用に定義されている。

HandlerInterceptor を追加する

Spring のハンドラマッピングメカニズムはハンドラインタセプタの概念がある。これは例えば、リーダのチェックなど、特定の機能をあるリクエストに適用したい場合、にとても役に立つ。

ハンドラマッピングに置かれるインタセプタは org.springframework.web.servlet パッケージから HandlerInterceptor を実装しなければならない。このインタフェースは、3 つのメソッドが定義されている。ひとつは、実際のハンドラが実行される前に呼ばれる。1 つは、ハンドラが実行された後で呼ばれる。もうひとつは、リクエストが完全に完了した後で呼ばれる。この 3 つのメソッドで、全ての前処理、後処理を行うのに十分な柔軟性が提供されるはずだ。

この preHandle メソッドはブール値を返す。このメソッドは、実行チェーンの処理を途切れさせたりつなげたりするのに使うことができる。このメソッドが true を返したら、ハンドラの実行チェーンは、継続される。false を返したら、DispatcherServlet はインタセプタ自身がリクエストを処理するとみなし(例えば、適切なビューをレンダリングするとか)、他のインタセプタや、実行チェーンにある実際のハンドラの実行を継続しない。

下記の例は、時刻が朝の 9 時から夕方 6 時の間でなければ、全てのリクエストをインターセプトし、ユーザを特定のページに転送する。

```
<beans>
    <bean id="handlerMapping"
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="interceptors">
```

```
            <list>
                <ref bean="officeHoursInterceptor"/>
            </list>
        </property>
        <property name="mappings">
            <props>
                <prop key="/*.form">editAccountFormController</prop>
                <prop key="/*.view">editAccountFormController</prop>
            </props>
        </property>
    </bean>

    <bean id="officeHoursInterceptor"
          class="samples.TimeBasedAccessInterceptor">
        <property name="openingTime"><value>9</value></property>
        <property name="closingTime"><value>18</value></property>
    </bean>
<beans>
```

パッケージのサンプル：

```
public class TimeBasedAccessInterceptor extends HandlerInterceptorAdapter {

    private int openingTime;
    private int closingTime;
    public void setOpeningTime(int openingTime) {
        this.openingTime = openingTime;
    }
    public void setClosingTime(int closingTime) {
        this.closingTime = closingTime;
    }
    public boolean preHandle(
            HttpServletRequest request,
            HttpServletResponse response,
            Object handler)
    throws Exception {
        Calendar cal = Calendar.getInstance();
        int hour = cal.get(HOUR_OF_DAY);
        if (openingTime <= hour < closingTime) {
            return true;
        } else {
```

```
            response.sendRedirect("http://host.com/outsideOfficeHours.html");
            return false;
        }
    }
}
```

現在時刻が営業時間外の場合、リクエストが TimeBasedAccessInterceptor によってインターセプトされ、ユーザはスタティックな html ファイルにリダイレクトされ、例えば、このウェブサイトは、オフィスアワー中しかアクセスできない、と伝える。

見てきたように、Spring には、HandlerInterceptor を簡単に拡張できるようなアダプタが用意されている。

ビューとビューの解決

ウェブアプリケーション用の MVC フレームワークでは、全てビューをアドレスする方法が提供されている。Spring ではビューリゾルバが提供されていて、特定のビューテクノロジを新たに習得することなくモデルをブラウザにレンダリングできる。そのまんまで Spring では、例えば Java Server Pages や、Velocity テンプレートや、XSLT ビューを利用することができる。では様々なビューテクノロジとの統合に関して詳しく述べる。

Spring がビューを扱う上で重要な 2 つのインタフェースが、ViewResolver と View だ。ViewResolver はビュー名と実際のビューとのマッピングを提供する。View インタフェースはリクエストに対する準備をアドレスし、ビューテクノロジの中の 1 つにリクエストを渡す。

ViewResolvers

節[13.3 コントローラ]で議論したように、Spring ウェブ MVC フレームワークのコントローラはすべて、MOdelAndView インスタンスを返す。Spring における View はビュー名でアドレスされ、ビューリゾルバによって解決される。Spring にはかなり多くのビューリゾルバが用意されている。そのほぼ全てを挙げ、例を 2,3 挙げよう。

View リゾルバ	
ビューリゾルバ	**説明**
AbstractCachingViewResolver	キャッシングされたビューを扱う抽象的なビューリゾルバである。ビューは使われる前に準備が必要になることがある。このビューリゾルバを拡張すればビューをキャッシングすることができる。
XmlViewResolver	Spring のビーンファクトリと同じ DTD の XML で書かれたコンフィグレーションファイルを受け付けることができる **ViewResolver** の実装である。デフォルトのコンフィグレーションファイルは、**/WEB-INF/views.xml** である。
ResourceBundleViewResolver	**bundle basename** で指定された **ResourceBundle** でビーン定義を利用する **ViewResolver** の実装である。この bundle は通常クラスパスにあるプロパティファイルの中で定義されている。デフォルトのファイル名は **views.properties** である。
UrlBasedViewResolver	シンボリックなビュー名を、明示的なマッピング定義なしで URL へ直接解決できる **ViewResolver** の簡単な実装である。これは、シンボリック名が簡単な方法で

View リゾルバ	
ビューリゾルバ	**説明**
	任意のマッピングを必要とせずにビューリソースの名前にマッチする場合であればこれが適切である。
InternalResourceViewResolver	**InternalResourceView** をサポートする **UrlBasedViewResolver** の便利なサブクラス(つまり、サーブレットと JSP)であり、**JstlView** や **TilesView** のようなサブクラスである。リゾルバにより生成されるビュークラスはすべて **setViewClass** で指定することができる。詳細については、**UrlBasedViewResolver** の Javadoc を参照して欲しい。
VelocityViewResolver / FreeMarkerViewResolver	**VelocityView**(つまり、Velocity テンプレート)、あるいは **FreeMarkerView** のそれぞれをサポートする **UrlBasedViewResolver** の便利なサブクラスであり、それらの独自サブクラスである。

例として、JSP をビューテクノロジとして使う場合には UrlBasedViewResolver を使うことができる。このビューリゾルバはビュー名を URL に変換し、ビューを表示するためにリクエストを RequestDispatcher に渡す。

```
<bean id="viewResolver"
        class="org.springframework.web.servlet.view.UrlBasedViewResolver">
    <property name="prefix"><value>/WEB-INF/jsp/</value></property>
    <property name="suffix"><value>.jsp</value></property>
</bean>
```

ビュー名として test を返すと、このビューリゾルバはリクエストを RequestDispatcher に渡し、これが /WEB-INF/jsp/test.jsp に渡す。

異なるビューテクノロジを 1 つのウェブアプリケーションで組み合わせる場合は、ResourceBundleViewResolver を使うことができる:

```
<bean id="viewResolver"
class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
    <property name="basename"><value>views</value></property>
    <property
name="defaultParentView"><value>parentView</value></property>
</bean>
```

ResourceBundleViewResolver は basename で識別された ResourceBundle を精査し、解決される各ビューについて、ビュークラスとして[ビュー名].class というプロパティの値が、ビューの URL として[ビュー名].url というプロパティの値が用いられる。お分かりのように、プロパティファイルにある全てのビューから親のビューを識別することができる。このように、例えば、デフォルトのビュークラスを識別することができる。

キャッシュに関する注 - AbstractCachingViewResolver キャッシュビューインスタンスのサブクラスは解決できる。特定のビューテクノロジを用いた場合、パフォーマンスは劇的に改善される。キャッシュのプロパティを false に設定すれば、キャッシュをオフにできる。さらに、あるビューを実行時にリフレッシュ

できるようにしたい場合(例えば Velocity テンプレートが修正された場合)は、
removeFromCache(String viewName, Locale loc)メソッドを使えばよい。

ViewResolver をつなぐ

Spring では複数のビューリゾルバがサポートされている。これにより、リゾルバをつないだり、例えばある状況で特定のビューをオーバライドしたりすることができる。ビューリゾルバをつなぐというのはとても単純だ。 - アプリケーションコンテキストに複数のリゾルバを追加して、必要であれば順序を指定するために order プロパティを設定する。注意して欲しいのは、order のプロパティが高ければ高いほど、チェーンの中でのビューリゾルバの位置は後ろになる。

下記の例では、ビューリゾルバのチェインは、InternalResourceViewResolver(これは常に自動的にチェインの中で最後に設定される)と、Excel のビュー(これは、InternalResourceViewResolver ではサポートされていない)を指定する XmlViewResolver2 つのリゾルバからなる:

```
<bean id="jspViewResolver" ¥
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
<property name="viewClass" ¥
    value="org.springframework.web.servlet.view.JstlView"/>
  <property name="prefix" value="/WEB-INF/jsp/"/>
  <property name="suffix" value=".jsp"/>
</bean>

<bean id="excelViewResolver" ¥
    class="org.springframework.web.servlet.view.XmlViewResolver">
  <property name="order" value="1"/>
  <property name="location" value="/WEB-INF/views.xml"/>
</bean>

### views.xml

<beans>
  <bean name="report" class="org.springframework.example.ReportExcelView"/>
</beans>
```

特定のビューリゾルバ解決すべきビューがなかった場合、Spring は他のビューリゾルバが設定されていないかコンテキストを精査する。他にビューリゾルバがあれば、これを引き続き精査する。もしなければ例外を投げる。

他にも気をつけておくべきことがある。- ビューリゾルバの仕様では、ビューリゾルバはビューが見つからないことを示すために null を返すことができる、とされている。でも全てのビューリゾルバがそうだとは限らない。これは、場合によっては、リゾルバが単にビューが存在するかどうかを判断できないからだ。例えば、InternalResourceViewResolver は内部で RequestDispatcher を使ってディスパッチするのが JSP が存在する場合の 1 つのやり方であり、これが唯一の方法だ。同じことは VelocityViewResolver 他にについても当てはまる。もし、ビューが存在しない場合にそれを知らせないビューリゾルバを扱っ

ているのであれば、ビューリゾルバの Javadoc をチェックしてほしい。つまり、チェインの最後以外に InternalResourceViewResolver を置くのは、結果としてチェーンを全て精査されることはない、ということだ。というのは、InternalResourceViewResolver は常にビューを返すからだ。

ビューをリダイレクトする

すでに述べられているように、コントローラは通常、ビューリゾルバが特定のビューテクノロジ用に解決する論理的なビュー名を返す。JSP のような実際にはサーブレット/JSP エンジンで処理されるビューテクノロジでは、サーブレット API の RequestDispatcher.forward()あるいは RequestDispatcher.include()をつかって、最終的には InternalResourceViewResolver/InternalResourceView で forward、あるいは include される。Velocity、XSLT 等、他のビューテクノロジでは、ビュー自身がコンテンツをレスポンスストリームに生成する。

ビューがレンダリングされる前に、クライアントに HTTP をリダイレクトするのが望ましいことがある。これは、例えば、あるコントローラが POST されたデータで呼ばれ、レスポンスが実際には他のコントローラに委譲される(例えば、フォームのサブミットの成功とか)場合には望ましい。この場合、通常の内部的なフォワードは、他のコントローラが同じ POST データを参照するということであり、他の予期されるデータと混同する可能性がある場合は潜在的な問題となる。結果を表示する前にリダイレクトを行うほかの理由は、これによりユーザがフォームデータのサブミットを二度行うのを防ぐことにある。ブラウザが最初に POST を送ると、リダイレクトされたされたものが表示され、GET を実行する。これは関係ないので、カレントのページには POST の結果が反映されるのではなく、GET の結果が反映される。よって、画面をリフレッシュすることで、ユーザが意図せず同じデータを再度 POST してしまうことはない。このリフレッシュは、最初の POST データを再送信するのではなく、結果のページを GET するするだけになる。

RedirectView

コントローラのレスポンスの結果として強制的にリダイレクトする方法の 1 つは、コントローラが Spring の RedirectView のインスタンスを生成して返すようにすることだ。この場合、DispatcherServlet は通常のビュー解決メカニズムを利用するのではなく、(リダイレクトする)ビューが与えられ、動作するように依頼するだけだ。

RedirectView は単に HttpServletResponse.sendRedirect()を呼ぶ。これはクライアントブラウザに HTTP リダイレクトとして返される。モデルの属性はすべて単に HTTP のクエリパラメータとして示される。これはつまり、モデルは、文字列形式の HTTP クエリパラメータに変換できる(一般的には文字列あるいは文字列に変換できる)オブジェクトだけを含むべきであるというだ。

RedirectView を使い、ビューはコントローラ自身から生成される場合、少なくともリダイレクト URL はコントローラに注入されるのが通常は望ましい。というのは、コントローラに埋め込まれるのではなく、ビュー名などと同様にコンテキストに設定されるほうが好ましいからだ。

redirect:プレフィクス

RedirectView の利用がうまくいくのであれば、コントローラ自身が RedirectView を生成している場合、コントローラにはリダイレクトが発生していることに気づくことはない。これは、本当に次善の策であり、コントローラはどのようにレスポンスが処理されるのかを意識するべきでない。通常注入されるビュー名という観点からのみ意識するべきなのだ。

特別な redirect:プレフィクスであれば、これが実現できる。もし redirect:プレフィクスをもつビュー名が返されると、 UrlBasedViewResolver(とそのサブクラスすべて)はリダイレクトが必要であるという特別な表示として認識する。残りのビュー名は、リダイレクトの URL として扱われる。

正味の効果は、 コントローラが RedirectView を返す場合と同じだが、コントローラ自身は 論理的なビュー名の観点で扱うことができる。redirect:http://myhost.com/some/arbitrary/path.html は絶対 URL へリダイレクトするのに対し、redirect:/my/response/controller.html のような論理的なビュー名は相対的なカレントのサーブレットコンテキストへリダイレクトする。重要な点は、このリダイレクトビュー名は他の論理的ビュー名と同様にコントローラに注入されるがコントローラにはリダイレクトが発生していることはわからない、という点だ。

forward:プレフィクス

ビュー名が UrlBasedViewResolver やそのサブクラスによって解決されるように、特別な forward:プレフィクスを使うことができる。これはすべて URL とみなすビュー名に対し InternalResourceView(最終的には RequestDispatcher.forward()が実行される)を生成する。したがって、とにかく(例えば JSP 用に)InternalResourceViewResolver/InternalResourceView を使う場合、このプレフィクスを使うことはないが、他のビューテクノロジをメインに使っていて、しかしサーブレット/JSP エンジンで処理されるリソースにフォワードが強制的に起こるようにしたい場合に用いることができる。もしこれを多用する必要があるのであれば、複数のビューリゾルバをチェインにしてもよいだろう。

redirect:プレフィクスと同様、このプレフィクスがついたビュー名がコントローラに注入されても、そのコントローラはレスポンスの処理において何か特別なことが起こることは意識しない。

ロケールを使う

Spring のアーキテクチャがサポートする国際化の大部分は、Spring ウェブ MVC フレームワークでサポートされているものだ。DispatcherServlet では、自動的にクライアントのロケールを使ってメッセージを解決することができる。これは LocaleResolver オブジェクトで行われる。

リクエストが来たら、DispatcherServlet はロケールリゾルバを探し、もしそれが見つかると、ロケールを設定するためにそれを使おうとする。RequestContext.getLocale()メソッドを使って、常にロケールリゾルバで解決されるロケールを検索することができる。

自動のロケール解決と合わせて、ハンドラマッピングにインタセプタをアタッチすることもでき(ハンドラマッピングインタセプタの詳細については節[HandlerInterceptor を追加する]を参照してほしい)、特定の状況で、例えばリクエストのパラメータに基づいてロケールを変更することができる。

ロケールリゾルバやインタセプタは全て org.springframework.web.servlet.i18n パッケージで定義されており、通常の方法で、アプリケーションコンテキストで設定することができる。ここでは、Spring に含まれているロケールリゾルバを列挙する。

AcceptHeaderLocaleResolver

このロケールリゾルバは、クライアントのブラウザによって送信されたリクエストにある accept-language ヘッダを識別する。通常、このヘッダフィールドにはクライアントのオペレーティングシステムのロケールが設定されている。

CookieLocaleResolver

このロケールリゾルバはロケールを特定するために、クライアントに埋め込まれたクッキーを識別する。クッキーがあれば、ロケールを識別するために利用する。このロケールリゾルバのプロパティを使ってクッキー名を、最大の有効期限とともに識別することができる。

```
<bean id="localeResolver">
    <property name="cookieName"><value>clientlanguage</value></property>
<!-- in seconds. If set to -1, the cookie is not persisted (deleted when ¥
    browser shuts down) -->
    <property name="cookieMaxAge"><value>100000</value></property>
</bean>
```

これは、CookieLocaleResolver を定義する例である。

WebApplicationContext の特殊なビーン		
プロパティ	デフォルト	説明
cookieName	classname + LOCALE	クッキー名
cookieMaxAge	Integer.MAX_INT	クッキーがクライアントに永続化される有効期限。-1 の場合は、クッキーは永続化されない。これは、クライアントがブラウザをシャットダウンするまでのみ有効である。
cookiePath	/	このパラメータを使って、あなたのサイトの特定部分にのみクッキーを見せることができる。cookiePath が指定されていると、そのクッキーはそのパスとその配下に対してのみ有効になる。

SessionLocaleResolver

SessionLocaleResolver を使えば、そのユーザのリクエストと関連付けられているセッションからロケールを検索することができる。

LocaleChangeInterceptor

LocaleChangeInterceptor を使ってロケールの変更処理を組み込むことができる。このインタセプタはハンドラマッピングの 1 つに追加する必要がある(節[ハンドラマッピング]を参照のこと)。これはリクエストにあるパラメータを検出し、ロケールを変更する(コンテキストにある LocaleResolver の setLocale() を呼ぶ。)

```
<bean id="localeChangeInterceptor"
        class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
    <property name="paramName"><value>siteLanguage</value></property>
</bean>

<bean id="localeResolver"
        class="org.springframework.web.servlet.i18n.CookieLocaleResolver"/>

<bean id="urlMapping"
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
```

```
    <property name="interceptors">
        <list>
            <ref bean="localeChangeInterceptor"/>
        </list>
    </property>
    <property name="mappings">
        <props>
            <prop key="/**/*.view">someController</prop>
        </props>
    </property>
</bean>
```

siteLanguage という名前のパラメータを含んだ*.view リソースへの呼び出しはすべてロケールを変更する。よって、http://www.sf.net/home.view?siteLanguage=nl という呼び出しは、サイトの言語をオランダ語に変更する。

テーマを使う

イントロ

Spring ウェブ MVC フレームワークで提供されるテーマのサポートにより、アプリケーションのルックアンドフィールをテーマにあわせることで、ユーザエクスペリエンスを大幅に拡張することができる。テーマは基本的にアプリケーションのビジュアル的なスタイルに効果のある静的なリソースを集めたもので、よくあるものは、スタイルシートと画像だ。

テーマを定義する

ウェブアプリケーションにテーマを使いたい場合、org.springframework.ui.context.ThemeSource を設定する必要がある。WebApplicationContext インタフェースは ThemeSource を拡張するが、実装に関する責任は委譲する。デフォルトでは、この委譲はクラスパスのルートからプロパティファイルを読む org.springframework.ui.context.support.ResourceBundleThemeSource になる。自前の ThemeSource の実装を使いたい、あるいは ResourceBundleThemeSource の basename プレフィクスを設定する必要がある場合は、アプリケーションコンテキストに予約名"themeSource"でビーンを登録する。ウェブアプリケーションコンテキストは自動的にこのビーンを検出し、利用する。

ResourceBundleThemeSource を使う場合、テーマは、単純なプロパティファイルに定義される。このプロパティファイルには、テーマを構成するリソースのリストが列挙されている。これは、その例だ。

```
styleSheet=/themes/cool/style.css
background=/themes/cool/img/coolBg.jpg
```

プロパティのキーは、ビューのコードからテーマを構成する要素を参照するに使われる名前だ。JSP ではこれは通常、spring:theme というカスタムタグを用いて行われるが、これは、spring:message タグととても似たものだ。下記の JSP の断片では、ルックアンドフィールをカスタマイズするために上述したテーマを使っている。

```
<taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<html>
  <head>
```

```
<link rel="stylesheet" href="<spring:theme code="styleSheet"/>" ¥
    type="text/css"/>
  </head>
  <body background="<spring:theme code="background"/>">
    ...
  </body>
</html>
```

デフォルトでは、ResourceBundleThemeSource は空の basename プレフィクスを使う。結果、プロパティファイルはクラスパスのルートから読み込まれ、テーマ定義 cool.properties をクラスパスのルート、つまり/WEB-INF/classes に配置する必要がある。ここで注意して欲しいことは、ResourceBundleThemeSource は標準の Java リソースバンドル読み込みメカニズムを使っているので、テーマの完全な国際化が可能である、ということだ。例えば、オランダ語のテキストを表示するための特別な背景画像を参照する/WEB-INF/classes/cool_nl.properties を使うことができる。

テーマリゾルバ

テーマを定義したら、あとは、そのテーマを使うかどうかを決めるだけだ。DispatcherServlet はどの ThemeResolver の実装を使うかを見つけるために"themeResolver"という名前のビーンを探す。テーマリゾルバは LocaleResolver と同じ方法で動く。特定のリクエスト用に使うべきテーマを検出し、リクエストのテーマを変更することもできる。下記のテーマリゾルバが Spring で提供されている。

FixedThemeResolver	固定のテーマを選択し、"defaultThemeName"プロパティを使って設定する。
SessionThemeResolver	テーマはユーザの HTTP セッションで維持される。各セッションごとに一度設定する必要があるだけだが、セッション間で永続化はされない。
CookieThemeResolver	選択されたテーマはクライアントのマシン上のクッキーに保存される。

Spring では、ThemeChangeInterceptor も用意されていて、簡単なリクエストパラメータを含めることにより各リクエストごとにテーマを変更することができる。

Spring のマルチパート(ファイルアップロード)サポート

イントロ

Spring ではウェブアプリケーションで、ファイルアップロードを扱うマルチパートのサポートが組み込まれている。マルチパートサポートの設計は、org.springframework.web.multipart パッケージで定義されている、取り外しが可能な MultipartResolver オブジェクトで行われている。*Commons FileUpload*(http://jakarta.apache.org/commons/fileupload)や *COS FileUpload*(http://servlets.com/cos)でそのまま使える MultipartResolver が Spring では提供されている。以降では、ファイルがどのようにアップロードされるかを述べる。

デベロッパの中には、自分でマルチパートの処理を行いたい人がいるように、デフォルトでは、Spring ではマルチパートの処理は行われない。マルチパートリゾルバをウェブアプリケーションコンテキストに追加することでこれが可能になる。追加すれば、各リクエストがマルチパートを含んでいるかどうかを見るために精査されるようになる。マルチパートが見つからなければ、リクエスト期待通りに継続される。しかしながら、マルチパートがリクエスト中に見つかれば、コンテキストですでに宣言されている MultipartResolver が利用される。そして、リクエスト中の multipart 属性が他の属性と同じように扱われる。

MultipartResolver を使う

下記の例は、CommonsMultipartResolver の使い方を示したものである:

```
<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">

<!-- one of the properties available; the maximum file size in bytes -->
    <property name="maxUploadSize">
        <value>100000</value>
    </property>
</bean>
```

この例では、CosMultipartResolver を使った例だ:

```
<bean id="multipartResolver"
    class="org.springframework.web.multipart.cos.CosMultipartResolver">

<!-- one of the properties available; the maximum file size in bytes -->
    <property name="maxUploadSize">
        <value>100000</value>
    </property>
</bean>
```

マルチパートリゾルバが動作するにはもちろん、適切な jar をクラスパスで指定する必要がある。CommonsMultipartResolver の場合、commons-fileupload.jar を使う必要があるし、CosMultipartResolver の場合は、cos.jar を使う必要がある。

Spring でマルチパートなリクエストを扱うためにどのように設定するかを見てきた。ここでは、実際にどうやって使うかの話をしよう。Spring の DispatcherServlet がマルチパートなリクエストを検出すると、コンテキスト中に宣言されているリゾルバがアクティベートされ、リクエストに渡される。基本的なことは、カレントの HttpServletRequest がマルチパートをサポートする MultipartHttpServletRequest にラップされるということだ。MultipartHttpServletRequest を使ってこのリクエストに含まれているマルチパートに関する情報を取得し、実際のマルチパートそのものをコントローラに取得される。

フォームでファイルアップロードをハンドリングする

MultipartResolver が動作を終えると、リクエストは他と同じように処理される。これを使うために、アップロードフィールドのあるフォームを生成し、Spring にフォームのあるファイルをバインドする。自動的に String やプリミティブな型への変換は行われない他のプロパティと同様、バイナリデータをビーンに取り込むために、ServletRequestDatabinder でカスタムエディタを登録しないといけない。

したがって、ウェブサイトのフォームを使ってファイルのアップロードをできるようにするには、リゾルバ、ビーンを処理するコントローラにマッピングされる URL,そしてそのコントローラそのものを宣言する。

```
<beans>

    ...
```

```
    <bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver"/
>

<bean id="urlMapping" ¥
    class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <props>
                <prop key="/upload.form">fileUploadController</prop>
            </props>
        </property>
    </bean>

    <bean id="fileUploadController" class="examples.FileUploadController">
<property ¥

name="commandClass"><value>examples.FileUploadBean</value></property>
        <property name="formView"><value>fileuploadform</value></property>
        <property
name="successView"><value>confirmation</value></property>
    </bean>

</beans>
```

そして、コントローラとファイルプロパティを保持する実際のビーンを生成する。

```
// snippet from FileUploadController
public class FileUploadController extends SimpleFormController {

    protected ModelAndView onSubmit(
        HttpServletRequest request,
        HttpServletResponse response,
        Object command,
        BindException errors)
        throws ServletException, IOException {

        // cast the bean
        FileUploadBean bean = (FileUploadBean)command;

        // let's see if there's content there
        byte[] file = bean.getFile();
```

```
        if (file == null) {
            // hmm, that's strange, the user did not upload anything
        }

        // well, let's do nothing with the bean for now and return:
        return super.onSubmit(request, response, command, errors);
    }

    protected void initBinder(
        HttpServletRequest request,
        ServletRequestDataBinder binder)
        throws ServletException {
        // to actually be able to convert Multipart instance to byte[]
        // we have to register a custom editor (in this case the
        // ByteArrayMultipartEditor
        binder.registerCustomEditor(byte[].class, new
ByteArrayMultipartFileEditor());
        // now Spring knows how to handle multipart object and convert them
    }

}

// snippet from FileUploadBean
public class FileUploadBean {
    private byte[] file;

    public void setFile(byte[] file) {
        this.file = file;
    }

    public byte[] getFile() {
        return file;
    }
}
```

これでわかるように、FileUploadBean には、ファイルを保持するbyte[]型のプロパティがある。Spring にリゾルバが見つけたマルチパートオブジェクトをビーンで指定されたプロパティに変換する方法を知らせるために、コントローラはカスタムエディタを登録する。この例では、ビーン自身に byte[]プロパティは何もしないが、何でもできる(データベースに保存したり、誰かにメールしたり)。

でも、まだ終わりじゃない。実際にユーザが何かをアップロードするには、フォームを作らないといけない:

```
<html>
    <head>
        <title>Upload a file please</title>
    </head>
    <body>
        <h1>Please upload a file</h1>
<form method="post" action="upload.form" enctype="multipart/form-data">
            <input type="file" name="file"/>
            <input type="submit"/>
        </form>
    </body>
</html>
```

これでわかるように、byte[]を保持するビーンのプロパティにちなんだ名前のフィールドを作った。さらに、encoding 属性を追加し、ブラウザがマルチパートフィールドをどうやってエンコードすればいいかを知らせる必要がある(これを忘れてはいけない)。これですべてだ。

例外をハンドリングする

Spring では、リクエストに合致したコントローラでリクエストが処理される間に期待していない例外が発生するのを防ぐために HandlerExceptionResolver が用意されている。HandlerExceptionResolver はウェブアプリケーションデスクリプタ web.xml で定義できる例外マッピングにいくらか似ている。しかしながら、これは例外をハンドリングするのに柔軟な方法を提供する。これは、例外が投げられたときに、どのハンドラが実行されるかについての情報を提供する。さらに、例外処理のプログラム的な方法によりリクエストが他の URL に転送される前に適切にレスポンスするためのオプションが提供される(例外マッピングに特化したサーブレットを使う場合と同じだ)。

resolveException(Exception, Handler)メソッドの実装と ModelAndView を返すというだけの HandlerExceptionResolver の実装と合わせて、SimpleMappingExceptionResolver も使うことができる。このリゾルバは投げられるかもしれない任意の例外のクラス名を取り、ビュー名にマッピングすることができる。これは、機能的には、サーブレット API の例外マッピング機能と同じではあるが、他のハンドラよりも、より精度の高い例外マッピングを実装することができる。