Springフレームワークの紹介

ロッドジョンソン著 金田忠士訳

http://www.theserverside.com/articles/article.tss?l=SpringFramework

2003/12

概 要

今年の夏、Springフレームワークに関しての噂話を聞いたことがあるかもしれない。本稿では、Springが何をやり遂げようとしているか、そして私がどうのようにJ2EEアプリケーションを開発するのに役に立つと信じているのかを説明しようと思う。

目 次

1　また他のフレームワーク?

ひょっとすると「もう他のフレームワークなんていらない」と考えているかもしれない。既に多くのオープンソースの (あるいはプロプライエタリな)J2EE フレームワークがあるのに、なぜわざわざこの記事を読まないといけないのか、あるいは Spring フレームワークをダウンロードしないといけないのか。

私は下記に挙げた 2，3 の理由で spring はユニークだと考えている。

- 他の著名なフレームワークではサポートしない機能を持っている。Spring では、ビジネスオブジェクトを管理する方法を提供することにフォーカスしている。
- Spring は包括的且つモジュール性がある。Spring はレイヤ化されたアーキテクチャであり、これは任意の部分だけというような使い方が可能である、ということ意味しているが、アーキテクチャとして内部的には一貫性が取れている。従って学習曲線の最大値の効果が得られるのである。JDBC を使うためだけに Spring を使うことにしてもビジネスオブジェクト全部を管理するのに Spring を使っても構わないのである。
- テストが容易に実施できるコードを書くのを支援するように最初から最後まで設計されている。Spring はテストドリブンプロジェクトにとって理想のフレームワークなのだ。

Spring さえあれば、プロジェクトに他のフレームワークを用意する必要性はない。Spring は潜在的にワンストップであり、典型的なアプリケーションに関連するほとんどのインフラに用いることができる。もはや他のフレームワークは必要ない。

2003 年 2 月から始まったばかりのオープンソースプロジェクトではあるが、Spring には古い遺産がある。このオープンソースプロジェクトは 2002 年終わりの拙著「Expert One-on-One J2EE　Designe and Development」に掲載の基盤コードから始まった。「Expert One-on-One J2EE」では Spring の背景で考えられているアーキテクチャの基礎的な部分についても触れている。しかしながらアーキテクチャに関するコンセプトは 2000 年初頭までさかのぼり、うまくいった一連の商用プロジェクト用に私がインフラを開発した経験が反映されている。

2003 年 1 月から Spring は SourceForge でホスティングされている。今は、とてもアクティブな 6 人を含めた 10 人の開発者がいる。

2　Spring のアーキテクチャ的な利点

仕様の話に入る前に、Spring がプロジェクトにもたらしてくれる利点について見てみよう。

- EJB を使うと決めようが決めまいが、Spring はミドル層のオブジェクトを効果的に構成することができる。Struts や他の特定の J2EE API に対応したフレームワークしか使わないのであれば、Spring にどこまでを委ねるかは思い通りだ。

- Spring では多くのプロジェクトで見られるシングルトンの激増を抑えることが可能だ。私の経験ではこれは大きな問題で、シングルトンが増えるとテストがしづらくなりオブジェクト指向ではなくなるのだ。

- Spring ではアプリケーションとプロジェクトにおいて一貫した方法でコンフィグレーションを扱うことによって様々なカスタムプロパティファイルフォーマットを用いる必要がなくなる。　あるクラスが探すプロパティキーやシステムプロパティがいったい何なのか分からなくて Javadoc やソースコードまで読まないといけなかったりでもしたら？Spring では単純にそのクラスの JavaBean プロパティを見るだけだ。Inversion of Control(後述する) を使ってるのでこんなに単純にできるのだ。

- Spring はクラスではなくインタフェースにするプログラミングコストをほとんどゼロに減らすことによっていいプログラミングプラクティスを促す。

- Spring を用いて構築されたアプリケーションができるだけ SpringAPI に依存しないように Spring は設計されている。Spring アプリケーションにおけるほとんどのビジネスオブジェクトには Spring への依存性はない。

- Spring を使って構築したアプリケーションはユニットテストが簡単に実施できる。

- Spring は EJB の利用をアプリケーションアーキテクチャの決定要素ではなく実装上の選択とすることができる。ビジネスインタフェースを呼び出しコードに影響することなく、POJO やローカル EJB として実装するという選択が可能なのだ。

- Spring では EJB を使わずとも多くの問題が解決できるようになる。Spring は EJB の代わりとなる、多くの Web アプリケーションに適切なものを提供可能なのだ。例えば、Spring では EJB コンテナを使わないで宣言的なトランザクション管理を実現するために AOP を使うことができるし、もしデータベースを 1 つしか使わないのであれば、JTA 実装すら必要ない。

- Spring ではデータアクセスに関して、JDBC を使っていようが、Hibernate のような O/R マッピングプロダクトを使っていようが一貫性のあるフレームワークを提供する。

Spring によって、本当にあなたの抱えている問題に対する最もシンプルで実現可能なソリューションを実装することができるようになるのである。そしてこれはとても価値あることだ。

3 Spring は何をしてくれるの?

Spring では多くの機能が提供されているので、個々の主な機能についてざっと概観しよう。

3.1 ミッションステートメント

まずは、Spring のスコープを明らかにしよう。Spring は多くの分野をカバーするが、何に注力し、何に注力してはいけないかという明確なビジョンが我々にはあるのだ。

Spring のメインとなる目標は J2EE をより使いやすく、また優れたプログラミングの実施を促進することである。

Spring は車輪の再発明はやらない。よって、ロギングパッケージやコネクションプール、分散トランザクション調停が Spring にないことに気づくだろう。これらは全てオープンソースプロジェクト (Commons Logging のように。これは、我々が全てのログ出力に使っているものだ。あるいは Commons DBCP) やあるいはあなたのアプリケーションサーバで提供されているものだ。同じ理由で、我々は O/R マッピングレイヤについても提供しない。この問題に対しては、Hibernate や JDO のような優れたソリューションがあるからだ。

Spring では既存のテクノロジをより使いやすくすることを目標としている。例えば、低レベルトランザクション調停は我々のターゲットではないが JTA 上の抽象レイヤやその他のトランザクションストラテジを提供している。

もし我々が何か新しいものを提供できると思わなければ、Spring は他のオープンソースプロジェクトとは直接には競合しない。例えば、多くの開発者のように、我々にも Struts に決して満足できず、MVC Web フレームワークには改善の余地があると考えている。いくつかの点において、軽量 IoC コンテナや AOP フレームワークのように、Spring には直接競合する部分があるが、これらは解決ソリューションがポピュラーになっていない分野だ。(Spring はこの分野のパイオニアだったのだ)。

Spring ではさらに内部一貫性による利点がある。開発者はみな、同じ賛美歌シート、つまり「Expert One-on-One J2EE Design and Development」に忠実なままの基本的な考え方で歌を奏でている。そして我々もいろんな分野に跨る Inversion of Control のような、その中心的なコンセプトを使うことができた。

Spring はどんなアプリケーションサーバでも同じように使うことができる。もちろん、このポータビリティを保証することは常にチャレンジであるが、我々は何かのプラットフォームに特化したり、スタンダードから外れることは避け、WebLogic や Tomcat、Resin、JBoss、WebShere やその他のアプリケーションサーバのユーザをサポートする。

3.2 Inversion of control コンテナ

Spring の設計上核となっているものは `org.springframework.beans` パッケージであり、これは JavaBeans 動作するように設計されたものである。このパッケージは通常、ユーザが直接利用することはないが、他の多くの機能の土台として役に立っている。

その次に抽象度の高いレイヤは「ビーンファクトリ」だ。Spring ビーンファクトリは汎用のファクトリで、オブジェクトを名前で検索できるようにし、オブジェクト間の関連を管理できるというものだ。

ビーンファクトリでは 2 つのオブジェクトのモードをサポートしている。

- 「シングルトン」:この場合、オブジェクトにはルックアップで検索対象となる特定の名前がついた 1 個の共有インスタンスが存在する。これはデフォルトで、ほとんどの場合に用いられるモードだ。これはステートレスサービスオブジェクトに向いている。
- 「プロトタイプ」:この場合は、検索の度に独立したオブジェクトが生成される。例えば、各ユーザが独自のオブジェクトを所有できるようにするのに用いることができると思う。

`org.springframework.beans.factory` はシンプルなインタフェースなので、一連の基礎となる記憶方法としてインプリメントすることが可能だ。これが必要なユーザはあまりいないがあなたのものも容易にインプリメントできる。最も一般的に利用されるビーンファクトリの定義は次のとおり。

- `XmlBeanFactory`:これはクラスや名前がつけられたオブジェクトのプロパティを定義している単純で直感的な XML 構造を解析する。我々は簡単に書けるように DTD を用意している。
- `ListableBeanFactoryImpl`:これはプロパティファイルで定義されているビーン定義を解析したり、ビーンファクトリをプログラムで生成する機能を提供している。

ビーン定義は各々、POJO(クラス名と JavaBean 初期化プロパティが定義されている) やファクトリビーンにすることができる。ファクトリビーンインタフェースによって 1 枚皮をかぶせる。通常これは、AOP やその他のアプ

ローチで用いられるプロキシオブジェクト、例えば宣言的なトランザクション管理を追加するプロキシを生成するのに用いられる。(これは、概念的にはEJB インターセプションに似ているが、実際はもっと単純に動くものだ。)

BeanFaftory は先祖から定義を「継承している」階層に自由に参加させることができる。このことで独自のオブジェクト群を抱えるコントローラサーブレットのような個々のリソース間でアプリケーション全体で共通のコンフィグレーションを共有することが可能となる。

JavaBeans を使おうとする動機は、「Expert One-on-One J2EE Design and Development」の 4 章に書いた。これは、ServerSide でフリーの PDF が閲覧可能だ。(http://www.theserverside.com/articles/article.jsp?l=RodJohnsonInterview)

`BeanFactory` の概念によって Spring は Inversion of Control コンテナである。(私はこのコンテナという用語はどうもしっくりこない。EJB コンテナのようなヘビーウェイトなコンテナのイメージが沸いてしまうからだ。)Spring の `BeanFactory` はコード一行で生成することができるコンテナで、特別なデプロイ手順を必要とはしないのだ。)

Inversion of Control の背景にある考え方は、「電話してこないで!必要だったらこちらから電話するから」というハリウッドの原理で表現されることが多い。IoC では何かをするときに発生する責任をアプリケーションコードからフレームワークへ移す。これは、コンフィグレーションがこれに関係するということは、EJB のような伝統的なコンテナアーキテクチャでは、コンポネントはコンテナを「オブジェクト X はどこで、これは私の作業に必要なものだ」と言って呼ぶところ、IoC ではコンテナがそのコンポネントはオブジェクト X が必要だと指摘し、実行時にそれを提供する。このコンテナではこの指摘は (JavaBean プロパティのような) メソッドのシグネチャを、そして XML のようなコンフィグレーションデータ頼りに、行っている。

IoC には様々な重要な利点がある。例えばこのようなものだ。

- コンポネントには実行時に協調して動くものを調べる必要がないので、書いて維持するのがとても簡単になる。Spring での IoC のバージョンでは、コンポネントは　他のコンポネントへの依存性を JavaBean の setter メソッドで表現する。EJB で同じことをやろうとすると、JNDI を調べて必要なコードを書く必要があるだろう。

- これと同じ理由で、アプリケーションコードはテストが容易になる。JavaBean プロパティは単純で、Java の核であり、テストが簡単だ。オブジェクトの生成や適切なプロパティの設定を行う自前の JUnit テストメソッドを書くだけだ。

- 優れた IoC の実装では強い型付けを維持する。もしコラボレータを調べるために汎用的なファクトリが必要であれば、得られた結果を望んでいる型へキャストしないといけない。これは大きな問題ではないが、エ

レガントじゃない。IoC だとあなたのコードの中の強く型付けされた依存性はフレームワークが責任をもってタイプキャストを行う。これは、型が不一致してるとフレームワークがアプリケーションを設定するときにエラーが発生するということだ。あなたは自分のコードの中のクラスのキャスト例外について心配しなくていいのだ。

- ビジネスオブジェクトのほとんどは IoC コンテナ API に依存しない。このため、レガシーコードが使いやすくなり、IoC コンテナ内外のオブジェクトが使いやすくなる。例えば、Spring ユーザは Jakarta Commons DBCP DataSource を Spring ビーンとして設定することがある。実際にこれをするのにそれ用のコードを書く必要はないのだ。我々は IoC コンテナがでしゃばり過ぎだと言っているのではない。これを使えばあなたのコードが API への依存性に侵略されなくなるということなのだ。どんな JavaBean でも Spring ビーンファクトリでコンポネントにすることができるのだ。

この最後のポイントは強調しておくべきだろう。IoC はアプリケーションコードのコンテナへの依存性を最小限にするという点で EJB のようなこれまでのコンテナアーキテクチャとは異なるのだ。これはつまり、あなたのビジネスオブジェクトが潜在的に別の IoC フレームワーク、あるいは任意のフレームワークの外側でコードを変更することなく実行することができるということなのだ。

私や Spring ユーザの経験では、IoC がアプリケーションコードにもたらす利点は強調しづらいものだ。

J2EE コミュニティではようやくにぎわってきたが、IoC という概念はそれほど新しいものではない。IoC コンテナは他にもある。著名なものとしては、Apache の Avalon や PicoContainer、HiveMind がある。Avalon は強力だし長い歴史があるにも係わらず有名にはなっていない。Avalon はかなりヘビーウェイトかつ複雑で他の新しい IoC ソリューションに比べてさしでがましいようだ。PicoContainer はライトウェイトで JavaBean プロパティではなく、コンストラクタで依存性を表現する点を強調している。Spring とは違い、各型のたった 1 つのオブジェクトが定義できるように設計されている (おそらく Java コードではない任意のメタデータを排除した結果による制限だろう)。Spring と PicoContainer や他の IoC フレームワークを比較については、Spring のウェブサイト (http://www.springframework.org/docs/lightweight_container.html) にある「The Spring Framework - A Lightweight Container」という記事を参照して欲しい。このページには PicoContainer のウェブサイトへのリンクも用意してある。

Spring のビーンファクトリは非常に軽量だ。ユーザはアプレット内部でもスタンドアロンの Swing アプリケーションででもそれを使うことができる。(EJB コンテナといっしょでもちゃんと動く)。特別なデプロイ手順はなく、立

ち上げに時間を要することもない。アプリケーションの任意のレイヤで即座にコンテナをインスタンス化することができるのでとても便利なのだ。

Spring のビーンファクトリのコンセプトは Spring 全体で用いられており、これは、Spring が内部的に一貫性を持っているということの一番の根拠になっている。Spring は他に IoC コンテナに関してもユニークで、フレームワークの全ての機能を通して基本コンセプトとして IoC を用いている。

アプリケーション開発者にとって最も重要なことは、一つあるいはそれ以上のビーンファクトリがビジネスオブジェクトのために定義されたレイヤを提供する、ということだ。これはローカルセッションビーンに似てはいるが、これと比較してもいたってシンプルなものだ。EJB とは違い、このレイヤにあるオブジェクトには相互関係を持たせることができ、その関係は所持しているファクトリで管理される。ビジネスオブジェクトのレイヤが既に定義されていることは、有益なアーキテクチャにとってとても重要なことだ。

Spring の `ApplicationContext` はビーンファクトリのサブインタフェースであり、下記をサポートする。

- 国際化をサポートしたメッセージのルックアップ
- アプリケーションオブジェクトをパブリッシュしたり、イベントの通知を受けるように登録したりできるイベント機構
- ポータブルなファイルやリソースへのアクセス

3.3 XmlBeanFactory の例

Spring ユーザは通常アプリケーションの設定を”bean definition”ファイルに XML で記述する。Spring XML ビーン定義文書のルートは <beans> 要素である。この <beans> 要素には 1 つ以上 <bean> 定義がある。通常は、各々のビーン定義でクラスやプロパティを指定するが、この他にもコードからこのビーンにアクセスするのに名前として使う ID を指定する必要がある。

ここで、簡単な例を見てみよう。この例は、J2EE アプリケーションでよく見られるような関連をもつ 3 つのアプリケーションオブジェクトを設定するものだ。

- J2EE データソース
- このデータソースを使う DAO
- 処理の中でこの DAO を利用するビジネスオブジェクト

下記に挙げる例では、Jakarta Commons DBCP プロジェクトの `BasicDataSource` を使う。このクラスは、(他の多く既存のクラスと同様に)Java-Bean スタイルの設定を使うので、Spring ビーンファクトリで簡単に使うことができる。

シャットダウンの際に呼ばないといけないcloseメソッドは、Springの”destroy-method”属性から登録することができ、BasicDataSourceにSpringインタフェースを実装しないといけないのを避けることができる。

```
<beans>
  <bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close"
    <property name="driverClassName">
      <value>com.mysql.jdbc.Driver</value>
    </property>
    <property name="url">
      <value>jdbc:mysql://localhost:3306/mydb</value>
    </property>
    <property name="username">
      <value>root</value>
    </property>
  </bean>
</beans>
```

我々が関心のあるBasicDataSourceのプロパティは全部文字列なので、値は、<value>要素で指定する。Springでは標準のJavaBeanプロパティエディタのメカニズムを使っており、必要であれば文字列表現を他の型に変換することができる。

それでは、このDataSourceへのビーン参照を持つDAOを定義しよう。ビーン間の関連は、<ref>要素を使って指定する：

```
<bean id="exampleDataAccessObject"
              class="example.ExampleDataAccessObject">
  <property name="dataSource"><ref bean="myDataSource"/></property>
</bean>
```

このビジネスオブジェクトはDAOへのリファレンスと、int型のプロパティ(exampleParam)を持っている。

```
<bean id="exampleBusinessObject"
              class="example.ExampleBusinessObject">
    <property name="dataAccessObject"><ref bean="exampleDataAccessObject"/></property>
    <property name="exampleParam"><value>10</value></property>
  </bean>

</beans>
```

オブジェクト間の関連は、この例のように通常コンフィグレーションで明示的に設定する。我々はこれはいいことだと思う。でも、Springでは、PicoContainerにならって我々が”autowire”と呼んでいる、ビーン間の依存性がわかるものも提供している。これを用意したことによる制限は、PicoContainer

と同様に、もし、特定の型のビーンが複数あると、どのインスタンスがその型の依存性を解決すべきなのかを算出することが不可能だということだ。肯定的には、ファクトリが初期化されたときには得られる依存性が不十分なのだ。(Spring では、さらに明示的なコンフィグレーションのための依存性チェックが可能であり、このゴールを達成することができる)。

もし、この関連性を明示的にコーディングしたいのであれば、上述の例では、下記のようにして autowire 機能を使うことができた。

```
<bean id="exampleBusinessObject"
        class="example.ExampleBusinessObject"
        autowire="byType">

    <property name="exampleParam"><value>10</value></property>
 </bean>
```

この使い方で、Spring は `exampleBusinessObject` の `dataSource` プロパティに現在のビーンファクトリで見つかる `dataSource` の実装が設定されているはずだ。もし存在しないか、ビーンファクトリが要求している型のビーンが複数あればエラーになる。これは、リファレンスではないので、まだ `exampleParam` プロパティを設定する必要がある。

Autowire のサポートや依存性チェックは既に CVS に登録されていて、Spring 1.0M2(2003/10/20 予定) で利用可能になる。本稿で議論しているその他の全ての機能については今のの 1.0M1 リリースで利用可能だ。

Java コードを変更することなく XML ファイルが変更できるように、Java コードから関連を外在化することはそれをコード化すること以上に大きなメリットがある。例えば、我々は `myDataSource` ビーンの定義を他のコネクションプールやテストデータソースを使うために別のビーンクラスを参照するように変更するのは簡単にできた。データソースを他の XML アプリケーションサーバ Spring の JNDI ロケーションファクトリビーンを使ってデータソースを単一の他の XML ベースのアプリケーションサーバから取得することができた。

ここで、ビジネスオブジェクトに関する Java コードの例をみてみよう。下記のコードでは、Spring へ依存していないということに注目して欲しい。EJB コンテナとは違って、Spring ビーンファクトリは侵略的ではないのだ。通常は、アプリケーションオブジェクトでこういったことを意識してコーディングする必要はないのだ。

```
public class ExampleBusinessObject implements MyBusinessObject {

        private ExampleDataAccessObject dao;
        private int exampleParam;
```

```
        public void setDataAccessObject(ExampleDataAccessObject dao) {
                this.dao = dao;
        }

        public void setExampleParam(int exampleParam) {
                this.exampleParam = exampleParam;
        }

        public void myBusinessMethod() {
                // do stuff using dao
        }
}
```

参照プロパティ設定子に注目して欲しい。これは、ビーン定義文書では、XML 参照に相当するものだ。これは、オブジェクトを使うまえに、Spring によって起動されるものである。

このようなアプリケーションビーンでは Spring に依存する必要はない。ビーンでは、Spring インタフェースや拡張 Spring クラスを実装する必要もない。単に、JavaBeans 命名規約に注意する必要があるだけだ。Spring アプリケーションコンテキスト外、例えばテスト環境でこれを再利用するのは簡単だ。デフォルトコンストラクタでインスタンス化して、プロパティを一つ一つ `setDataSource()` や `setExampleParam()` メソッドで設定する。引数のないコンストラクタさえあれば、1 行でプログラマティックコンストラクションをサポートしたいのであれば、自由に複数のプロパティを要する他のコンストラクタを定義することができる。

ビジネスオブジェクトインタフェースの呼び出し側に宣言されていない JavaBean プロパティが動くということに注目して欲しい。これは、実装についての詳細なのだ。我々は、異なるビーンプロパティをもつ異なる実装のクラスを接続されるオブジェクトや呼び出しコードに手を加えずに”プラグイン”することができたのだ。

もちろん、Spring XML ビーンファクトリにはここで述べられていることよりももっと可能性があるが、本稿ではあなた方へ基本アプローチを提示しなければならない。簡単なプロパティや、JavaBeans プロパティエディタが用意されているプロパティと同様、Spring では自動的にリストやマップ、`Java.util.Properties` をハンドリングすることができる。

ビーンファクトリやアプリケーションコンテキストは通常下記のような、J2EE サーバで定義されているスコープとの関連がある。

- サーブレットコンテキスト：Spring MVC フレームワークでは、アプリケーションコンテキストは各 Web アプリケーションが抱える共通オブジェクトごとに定義されている。Spring では、Struts や WebWork, あるいは他の Web フレームワークと同じようにそのようなコンテキスト

をリスナやサーブレットから Spring MVC フレームワークに依存せずにインスタンス化する機能を提供している。

- サーブレット：Spring MVC フレームワークの各コントローラサーブレットは自身のアプリケーションコンテキストを持っており、root(アプリケーション内の) アプリケーションコンテキストと区別する。これも Struts や他の MVC フレームワークでも簡単にできる。

- EJB: Spring では EJB オーサリングを簡単にし、EJB Jar ファイル中の XML ドキュメントからビーンファクトリをロードできるようになる EJB 用に便利なスーパクラスを提供している。

J2EE 仕様で提供されているこれらのフックは一般に、ビーンファクトリのブートストラップでシングルトンを使う必要性を避けている。

しかしながら、ビーンファクトリをプログラムでインスタンス化することは望むのであればたいしたことではない。例えば、我々はビーンファクトリを生成し、上記で定義したビジネスオブジェクトへのリファレンスを下記の 3 行のコードで取得することができた。

```
InputStream is = getClass().getResourceAsStream("myFile.xml");
XmlBeanFactory bf = new XmlBeanFactory(is);
MyBusinessObject mbo = (MyBusinessObject) bf.getBean("exampleBusinessObject");
```

このコードはアプリケーションサーバの外で動くものだ。Spring IoC コンテナはピュア Java なので J2EE にすら依存していない。

3.4 JDBC の抽象化とデータアクセス例外の階層

データアクセスは Spring が光り輝く別の分野だ。

JDBC は基盤になっているデータベースを本当にうまく抽象化しているが、API がとても使い辛い。難点としては以下のようなものがある。

- ResultSet や、ステートメント、(最も重要な) コネクションを使ったら確実クローズするには煩雑なエラーハンドリングが必要だ。これはつまり、JDBC を正しく使うことは、結局コードをたくさん書くということだ。さらに、いろんなエラーに共通の源泉でもある。コネクションがリークすると負荷ですぐにアプリケーションをダウンさせてしまう。

- 比較的有益でない `SQLException`. JDBC は例外階層を扱わず、全てのエラーに対し `SQLException` を投げる。何が実際に悪かったのか-例えば、その問題はデッドロックだったのかあるいは SQL が無効だったのか-がわかるためには `SQLState` やエラーコードを調べる必要がある。この値の意味はデータベースごとに異なるのだ。

Spring ではこの問題に対して 2 つの方法で対処する。

- 冗長でエラーになりがちな例外処理をアプリケーションコードからフレームワークへ移すための API を提供する。このフレームワークでは全ての例外を処理する。アプリケーションコードでは適切な SQL を発行しその結果を抽出することに専念できるのである。
- あなたのアプリケーションコードが `SQLException` の代わりに動作するように意味のある例外階層を提供する。Spring が `DataSource` からコネクションを最初に取得すると、データベースを確認するためのメタデータを検査する。ここで得られた情報は `SQLException` と `org.springframework.dao.DataAccessExceptio` からの子孫の階層にある正しい例外とのマッピングを取るために利用される。従ってあなたのコードでは意味のある例外を使うことができ、プロプライエタリな `SQLState` やエラーコードに悩む必要はなくなるのだ。Spring のデータアクセス例外は JDBC に特有のものではないので、あなたの DAO は投げられるかも知れない例外のために JDBC と結合する必要はないのである。

Springでは、2つのレベルでJDBC APIを提供している。1つは、`org.springframework.jdbc.core` パッケージで、コールバックを使って制御 - つまり、エラーハンドリングやコネクションの取得、解放- をアプリケーションコードからフレームワーク内部へ移すためのものだ。これは異なる Inversion of Control のタイプだが、コンフィグレーション管理に使っているものと同じぐらい有益なものだ。

Springでは似たようなコールバックアプローチを、JDO(`PersistenceManager` の取得や放棄)、(JTA を用いた) トランザクション管理や JNDI のようなリソースを取得したり解放するための特別な手順を含む様々な他の API を用意するために使う。このようなコールバックを実行する Spring クラスはテンプレートと呼ばれる。

例えば、Spring の `JdbcTemplate` オブジェクトは下記のリストのようにSQL 問い合わせや結果を退避するのに使われる。

```
JdbcTemplate template = new JdbcTemplate(dataSource);
final List names = new LinkedList();
template.query("SELECT USER.NAME FROM USER",new RowCallbackHandler() {
  public void processRow(ResultSet rs) throws SQLException {
    names.add(rs.getString(1));
  }
});
```

コールバックされるアプリケーションコードでは自由に `SQLException` を投げられるという点に注目して欲しい。Spring ではどんな例外でもキャッチできるし、またそれ自身の階層で再度スローすることができる。アプリケー

ション開発者はどの例外をキャッチしたりハンドリングするかを選択することができる。

JdbcTemplate ではあらかじめ用意されたステートメントやバッチによる更新などの異なるシナリオをサポートするために多くのメソッドが提供される。Spring における JDBC の抽象化は、たとえ巨大な結果セットを扱うアプリケーションのケースでさえも標準の JDBC よりも性能のオーバヘッドが低く抑えられている。

これよりも高いレベルの JDBC 抽象化は org.springframework.jdbc.object パッケージにある。これは、コア JDBC コールバック機能で構築されているが、RDBMS 操作 - 問い合わせでも更新、ストアドプロシージャでも- が Java オブジェクトとしてモデリングされている API を提供する。この API は部分的に直感的でとても使いやすいと私が感じた JDO 問い合わせ API にインスパイアされたものだ。

ユーザオブジェクトを返すための問い合わせオブジェクトはこのような感じだ。

```
class UserQuery extends MappingSqlQuery {
  public UserQuery(DataSource datasource) {
    super(datasource, "SELECT * FROM PUB_USER_ADDRESS WHERE USER_ID = ?");
    declareParameter(new SqlParameter(Types.NUMERIC));
    compile();
  }

  // Map a result set row to a Java object
  protected Object mapRow(ResultSet rs, int rownum) throws SQLException {
    User user = new User();
    user.setId(rs.getLong("USER_ID"));
    user.setForename(rs.getString("FORENAME"));
    return user;
  }

  public User findUser(long id) {
    // Use superclass convenience method to provide strong typing
    return (User) findObject(id);
  }
}
```

このクラスは以下のように使うことが可能。

```
  User user = userQuery.findUser(25);
```

このオブジェクトは DAO では多くの場合インナークラスだ。これはサブクラスで変なことをしなければスレッドセーフだ。

他に、org.springframework.jdbc.object パッケージで重要なクラスとして StoredProcedure クラスがある。Spring では、ビジネスメソッドを 1

つもったJavaクラスで代用してストアドプロシージャを可能にしている。好みによって、そのストアドプロシージャを実装したインタフェースを定義することができる。これは、アプリケーションコードをストアドプロシージャを使うということに依存しないようにすることができる、ということなのだ。

Springにおけるデータアクセス例外階層は未チェックの(ランタイム)例外が基になっている。いくつかのプロジェクトでSpringを使ってみて、これが正しい判断だったとますます確信しているのだ。

データアクセス例外は通常復元できない。例えば、データベースに接続できないと、特定のビジネスオブジェクトは恐らくその問題に対処できないだろう。潜在的な例外の1つは楽観的ロックを妨害することだが、全てのアプリケーションが楽観的ロックを使うとは限らない。通常正しく処理ができない致命的な例外をキャッチするように無理やりコードを書くことはいけないことだ。サーブレットやEJBコンテナのようにトップレベルのハンドラに伝播させるのが通常は適切だろう。Springのデータアクセス例外は全て`DataAccessException`のサブクラスなので、全てのSpringのデータアクセス例外をキャッチすると決めてしまえば実際にそうすることは簡単だ。

もし未チェックのデータアクセス例外から復旧したいのであれば、そうすることも可能だ、ということに注目していただきたい。復旧可能な条件だけを扱うようにコードを書くことが可能なのだ。例えば、楽観的ロック妨害しか復旧できないと考えているのであれば、下記のようにSpring DAOを使ってコードを書くことができる。

```
try {
  // do work
}
catch (OptimisticLockingFailureException ex) {
  // I'm interested in this
}
```

もし、Springのデータアクセス例外がチェックされたものであれば、下記のようなコードを書く必要があるだろう。とにかく、これを書くと決めることができたと言う点に注目して欲しい。

```
try {
  // do work
}
catch (OptimisticLockingFailureException ex) {
  // I'm interested in this
}
catch (DataAccessException ex) {
  // Fatal; just rethrow it
}
```

最初の例に対する潜在的な反論 - コンパイラは無理やり潜在的に復旧可能な例外をハンドリングすることはできない - は 2 つ目にも当てはまる。従って無理やりベース例外 (`DataAccessException`) をキャッチしようとするとコンパイラがサブクラス (`OptimisticLockingFailureException`) をチェックすることができなくなってしまう。よって、コンパイラは復旧不可能な問題を処理するためのコードを書かせるが、復旧可能な問題を扱わせるのに何の手助けもしないだろう。

Spring で未チェックのデータアクセス例外を使うことは、多くの (おそらく大部分の) 成功した永続フレームワークと整合が取れている。(もちろん、これは、部分的に JDO にインスパイアされている)。JDBC はチェック済み例外を扱うわずかなデータアクセス API のうちの 1 つだ。例えば TopLink や JDO では未チェック例外を排他的に使う。Gavin King は今では Hibernate が未チェック例外を選ぶべきだったと信じているのだ。

Spring の JDBC ではいくつかのやり方が利用可能だ。

- JDBC を使うのに final ブロックを再度書く必要はない
- 全体的に書く必要のあるコードはとても少ないだろう
- カラム名が間違っていたときに返す不明瞭なエラーコードを処理するために RDBMS のドキュメントを調べる必要はないと思う。あなたのアプリケーションは特定の RDBMS に特化したエラーハンドリングコードに依存しないのだ。
- どんな永続化技術を用いていようとも、特定のデータアクセス API に依存したビジネスロジックなしで DAO パターン実装するのが簡単に感じるだろう。

実際我々は、この全てが結局は本質的な生産性の向上とバグを減らすことになることがわかった。私はかつて JDBC コードを書くのが嫌だった。今では不意の JDBC リソース管理よりも実行したい SQL に集中できることがわかったのだ。

Spring の JDBC 抽象化は、望めば Spring の他のパーツを使っていることに注力することなくスタンドアロンとして使うことができる。

3.5 O/R マッピングの統合

リレーショナルデータアクセスを使うのではなく、O/R マッピングを使いたいこともちろんあるだろう。全面的なアプリケーションフレームワークではこれもサポートしなければならない。従って Spring では Hibernate 2.x

と JDO を外部で統合している。このデータアクセスアーキテクチャによって任意の基幹データアクセステクノロジとの統合が可能となっている。Spring や Hibernate は特に上手く統合されている。

なぜ、Hibernate を直接使うのではなく、Hibernate プラス Spring を使うのだろうか。それは、下記に挙げた観点で Spring が十分に価値があるからだ。

- セッション管理。Spring では効率よく、簡単に、しかも安全に Hibernate のセッションを扱うことができる。Hibernate を使うコードは一般に、効率と自前のトランザクションを処理するために、同じ Hibernate" セッション" オブジェクトを使う必要がある。Spring では、宣言的な AOP のメソッドインタセプタアプローチか、あるいは Java コードレベルで明示的な、" テンプレート" ラッパクラスを使って、透過的にセッションを生成し、カレントスレッドにバインドするのが簡単にできる。よって、Hibernate フォーラムで繰り返し起こっている使う上での多くの問題を Spring が解決するのだ。

- リソース管理。Spring のアプリケーションコンテキストでは、Hibernate セッションファクトリや JDBC データソース、その他の関連するリソースの位置や設定を扱うことができる。このため、これらの値を管理したり変更するのが簡単になる。

- 統合トランザクション管理。Spring では Hibernate コードを宣言的で AOP スタイルのメソッドインタセプタか、Java 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保たれ、例外の場合における自前のトランザクション処理 (ロールバックとか) も処理される。以下で議論しているように、Hibernate に関係するコードを使わ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入れ替えたりといった利便が得られる。加えて、JDBC に関するコードは Hibernate コードに完全に統合することができる。これは Hibernate に実装されていない機能を処理するのにとても役に立つ。

- 上述した例外のラッピング。Spring ではプロプライエタリなチェックした例外から抽象的なランタイム例外を組み合わせたものへ変換して Hibernate 例外をラップすることができる。これによって復旧できない、特定のレイヤでしか発生しないようなほとんどの永続化に関する例外を、決まりきった `catch/throw` や例外の宣言に悩まされずに処理することができるようになる。必要とするいかなる場合でも例外をトラップしたり処理することができる。JDBC 例外 (DB に特定の方言も含む) は同じ階層に変換される、ということを覚えておいて欲しい。これは、一貫したプログラミングモデルで JDBC の操作が実行可能である、ということを意味しているのだ。

- ベンダの制約を避けること。Hibernate は強力、柔軟、オープンソース、フリーだが、プロプライエタリな API も使う。　選択肢が与えられた場合、機能性やパフォーマンスや何か他の理由により他の実装に差し替える必要がある場合、標準的な、あるいは抽象化された API を使ってメジャーなアプリケーション機能を実装するのが、通常は望ましい。Spring における Hibernate トランザクションや Hibernate 例外の抽象化は、データアクセス機能を実装しているマッパー/DAO オブジェクトを差し替えるのをより簡単にしている IoC アプローチと共に、Hibernate の有用性を損なうことなく、全ての Hibernate 用コードをあなたのアプリケーションの一部に集約やすくしている。

- テストが簡単。Spring の inversion of control アプローチにより、Hibernate セッションファクトリ、データソース、トランザクションマネージャ、マッパオブジェクト実装 (必要であれば）の実装や位置を入れ替えることが簡単になっている。これにより、永続化関連のコードを切り離して単独でテストするのがとても簡単になっているのだ。

3.6　トランザクション管理

データアクセス API の抽象化は十分ではない。トランザクション管理についてもっと考慮する必要があるのだ。JTA は明確な解決策ではあるが、直接使用するにはやっかいな API だし、そのため、多くの J2EE 開発者は、EJB CMT がトランザクション管理に関する唯一の合理的な代替案だと考えている。

Spring ではトランザクション管理のために独自の抽象化を提供している。Spring ではこれを配布のために利用している。

- `JdbcTemplate` に似たコールバックテンプレートによる、方針に沿ったトランザクション管理。これは直接 JTA を使うよりもずっと簡単だ。

- EJB CMT に似ているが、EJB コンテナを必要としない宣言的なトランザクション管理

Spring におけるトランザクションの抽象化は、JTA やその他のトランザクション管理テクノロジと結合していない、と言う点で独特なものだ。Spring ではアプリケーションコードから (JDBC のような) 土台となるトランザクションインフラを切り離すというトランザクション戦略のコンセプトを用いているのだ。

なぜ、こういったことに注意を払わなければいけないのだろうか。JTA はトランザクション管理の最適解ではないのだろうか。もし、データベースを１つしか使わないアプリケーションを書いていたら、JTA の複雑さは必要ないだろうし、XA トランザクションや 2 フェーズコミットには関心を払わない

だろう。そのような機能を提供するハイエンドなアプリケーションサーバすら必要じゃないかもしれない。しかし、一方で、コードを複数のデータソースが扱えるように書き直さないといけないようになるのは望んでいないだろう。

JDBC や Hibernate トランザクションを直接使って JTA のオーバヘッドを避けることを決めたばあいを想像して欲しい。複数のデータソースを使う必要があるのであれば、トランザクション管理のコードを全て剥ぎ取って、JTA トランザクション処理に置換えないといけない。これはあまり楽しいものではないが、これによって私自身も含めてほとんどの J2EE のライターがグローバル JTA トランザクションを排他的に使うことを推奨するようになった。しかしながら、Spring のトランザクション抽象化機能を使えば、トランザクション戦略が実際に使っていたのが JDBC であろうと Hibernate であろうと Spring に JTA を使うように再設定するだけだ。これはコードの修正ではなく、設定の修正だ。よって、Spring では、スケールアップと同様スケールダウンも可能なアプリケーションを書くことができるようになるのだ。

3.7 AOP

最近、企業関連の AOP ソリューションの適用について興味深いことがたくさんあった。

Spring における AOP サポートの最初の目標は、J2EE サービスを POJO に提供することだ。これは、JBoss4 の目的と似ている。しかしながら、Spring の AOP はアプリケーションサーバ間のポータビリティについてアドバンテージがあり、そのためベンダに縛られるリスクがない。これは、Web コンテナでも EJB コンテナでも動作し、WebLogic,Tomcat,JBoss,Resin,Orion やその他のアプリケーションサーバや Web コンテナで実績がある。

Spring の AOP はメソッドインターセプトをサポートする。キーの AOP コンセプトがサポートするものは以下のものが含まれる。

- インターセプト：独自の振る舞いはどんなインタフェースやクラスに対してもメソッドを起動する前後に挿入される。これは AspectJ 用語でいう「アラウンドアドバイス」に似ている。
- 導入：あるアドバイスによってあるオブジェクトに追加のインタフェースを実装するように指定すること。これにより、mixin 継承が可能になる。
- 静的ポイントカットと動的ポイントカット：プログラムの実行においてインターセプトを実施する場所を指定すること。静的なポイントカットはメソッドのシグネチャに関係する。動的ポイントカットもまた引数が評価されるメソッドの引数も関係するかもしれない。ポイントカットは

インターセプタとは別々に定義され、標準のインターセプタは異なるアプリケーションやコードコンテキストにおいて適用される。

Spring ではステートフルインタセプタ (アドバイスされたオブジェクトごとに 1 つのインスタンス) とステートレスインタセプタ (全てのアドバイスに対して 1 つのインスタンス) の両方をサポートする。

Spring ではフィールドインターセプトはサポートしない。これは、設計上熟慮した上で決定したことだ。私は常々フィールドインターセプトはカプセル化を壊すものだと考えている。私は AOP を、OOP に反するものではなく、OOP を引き立たせるものだと思いたい。もし、5 年や 10 年もの間、AOP の学習曲線上をなぞり、AOP がアプリケーション設計のトップの座席を与える快適さを感じたとしても私は驚かないだろう。(しかしながら、この点で、AspectJ のように言語に基づいた解決策は今日よりももっと魅力的かもしれない。)

Spring では AOP を (インタフェースが存在する) 動的プロキシか、(クラスのプロキシを可能にする)CGLIB の実行時バイトコード生成を使って実装してある。このアプローチは双方ともどんなアプリケーションサーバででも動作する。

Spring は AOP アライアンスインタフェース (http://www.sourceforge.net/projects/aopalliance/) を実装した最初の AOP フレームワークだ。これは、
AOP フレームワーク間でインターセプタを透過にするインタフェースを定義しようという試みを表している。

これは、現在進行中であり、TheServerSide やその他のところで　このようなインターセプタが「本当の AOP」であるかどうかについてまだ議論に結論は出てはいないし、私はこれがなんと呼ばれるかについては特に注意を払っていない。単に、実際に役に立てるのかどうかだ。私は、等しく「宣言的ミドルウェア」と喜んで呼ぼうと思う。Spring の AOP は、単純で軽い、ステートレスセッションビーンの代替案で、モノシリックな EJB コンテナの必要性をなくし、必要とするサービスだけを持った独自のコンテナを作れるようにしたものだと考えて欲しい。ローカルなステートレスセッションビーンによって推奨された粒度が明らかにされるに似ているので我々はどんなアドバイスでもどんな POJO でもお勧めはしない。(しかしながら、EJB とは違って、適切な場面でまれなシナリオできめ細かいオブジェクトに Spring の AOP を自由に適用することが可能だ。)

Spring におけるアドバイスオブジェクトは、クラスローダレベルではなく、インスタンスレベルなので、同じクラスに複数のインスタンスに異なるアドバイスをつけたり、アドバイスの無いインスタンスをアドバイスつきインスタンスと同じように使うことが可能なのだ。

恐らく Spring AOP の最も一般的な利用は、宣言的なトランザクション管理のためのものだ。これは、上述した TransactionTemplate 抽象をベースに

構築されていて、宣言的トランザクション管理をいろんな POJO に伝えることができる。トランザクション戦略によって、ベースとなるメカニズムは JTA や JDBC,Hibernate やその他のトランザクション管理に関する API が利用できる。

Spring の宣言的トランザクション管理は EJB CMT と似ているが、下記の点で異なる。

- トランザクション管理はどんな POJO にでも適用が可能だ。我々は、ビジネスオブジェクトがインタフェースを実装するようにお勧めする。だが、これは、よいプログラミングを行う上での問題であり、フレームワークによって強制されるものではない。

- プログラムに基づいたロールバックは Spring のトランザクション API を使うことでトランザクション POJO で実現することができる。我々は `ThreadLocal` 変数を使ってこれのための静的なメソッドを提供する。従って、`EJBContext` のようにコンテキストオブジェクトをロールバック全体に波及させる必要はない。

- 「ロールバックルール」を宣言的に定義することができる。EJB ではアプリケーション例外をキャッチした際に自動的にトランザクションをロールバックすることはしない (未チェックの例外とその他 Throwable の場合のみ) が、アプリケーション開発者は任意の例外でトランザクションをロールバックしたいことがしばしばある。Spring トランザクション管理では、宣言的にどの例外やそのサブクラスで自動的にロールバックさせるかを指定することができる。デフォルト動作では EJB と同じだが、自動ロールバックを未チェック例外と同様にチェック済み例外にも指定することができる。これは、Spring トランザクション API(EJB のプログラム上のロールバックが `EJBContext` 上で行われるのと同じ) へ依存させるプログラム的にロールバックさせる必要性を最小限にする重要な利点だ。

- トランザクション管理は JTA とくくりつけられてはいない。前述したように、Spring トランザクション管理は異なるトランザクション戦略で動作する。

もちろん、SpringAOP をアプリケーションに特化したアスペクトを実装するのに利用するのも可能だ。これを利用するのもしないのも　AOP の概念を使うことが Spring の能力よりもどれだけ快適になるかの程度によるが、とても役に立つだろう。我々がこれまで見たことのある成功事例には以下のようなものがある。

- 標準の J2EE セキュリティインフラストラクチャの能力をはるかに超え

ている複雑なセキュリティチェックが求められるようなカスタムセキュリティインターセプション

- 開発中に利用するデバッグアスペクトとプロファイリングアスペクト
- システム管理者や普段あまりないシナリオのユーザにアラートメールを送信するインタセプタ

アプリケーションに特有のアスペクトは、多くのメソッドで何度も出てくる決り文句のようなコードをなくすのに威力を発揮する手だ。

Spring　AOP は Spring のビーンファクトリの概念を透過的に統合されている。Spring ビーンファクトリからオブジェクトを取得するコードはアドバイスされているかどうかを知っている必要がない。どのようなオブジェクトであっても、その契約はそのオブジェクトが実装するインタフェースによって定義される

下記の XML の一節は、AOP プロキシの定義の仕方を示したものだ。

```
<bean id="myTest" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="proxyInterfaces">
    <value>org.springframework.beans.ITestBean</value>
  </property>
  <property name="interceptorNames">
    <list>
      <value>txInterceptor</value>
      <value>target</value>
    </list>
  </property>
</bean>
```

このクラスのビーン定義は通常 AOP フレームワークの `ProxyFactoryBean` である。リファレンスに使われたり、`BeanFactory` の `getBean()` メソッドから返されるビーンの型はプロキシインタフェースに依存するのだが。(多重プロキシメソッドはサポートされている)。`ProxyFactoryBean` の「`interceptorNames`」プロパティは String のリストを取る。(ビーン名はビーンリファレンスよりも頻繁に使われるはずだ、プロキシが「`prototype`」の場合シングルトンビーン定義よりもステートフルインタセプタの新しいインスタンスが生成される必要があるかもしれないので)。このリストにある名前はインタセプタかあるいはポイントカット（インタセプタといつそれらが適用されるのかに関する情報) だろう。この前述のリストにある「`target`」の値は自動的にターゲットオブジェクトをラッピングする「invoker interceptor」を生成する。これはプロキシインタフェースを実装するファクトリ中のビーンの名前だ。この例での myTest ビーンはビーンファクトリ中の他のビーンのように使われる。例えば、他のオブジェクトが <ref> 要素から参照することができ、この参照は Spring IoC によって設定される。

また、これはめったに使われることはないが、BeanFactory を使わずに AOP プロキシをプログラマティックに生成することも可能だ。

```
TestBean target = new TestBean();
DebugInterceptor di = new DebugInterceptor();
MyInterceptor mi = new MyInterceptor();
ProxyFactory factory = new ProxyFactory(target);
factory.addInterceptor(0, di);
factory.addInterceptor(1, mi);
// An "invoker interceptor" is automatically added to wrap the target
ITestBean tb = (ITestBean) factory.getProxy();
```

我々は、Java コードとアプリケーションの結合性を弱くすることは一般的にベストである信じているし、これは、AOP についても例外ではない。

Spring のこの AOP の実現という点における一番の競合は Jon Tirsen の Nanning Aspects(http://nanning.codehaus.org/) だ。

私は、AOP をエンタープライズサービスを配備するために EJB の代替案として使うことの重要性が増していると思う。これは、前進している Spring でさらに重要視していくつもりだ。

3.8 MVC ウェブフレームワーク

Spring にはパワフルで柔軟性の高い MVC Web フレームワークが用意されている。

Spring の MVC モデルは、Struts の MVC モデルに一番似ている。Spring Controller は全てのクライアントの振る舞いが単一のインスタンス上でが実行されるマルチスレッドサービスオブジェクトであるという点で Struts Action に似ている。しかしながら、我々は Spring の MVC が Struts に比べてとても大きな利点を持っていると考えている。例えば以下の点についてだ。

- Spring ではコントローラと、JavaBean モデル、それにビューをとても明確に区別する
- Spring の MVC はとても柔軟だ。アクションとフォームオブジェクトに決まった継承を強制する (つまり、Java のコンクリート継承であなたのたった一つ好みを除外する)Struts と違って、Spring MVC は全てインタフェースがベースになっている。さらに、Spring MVC フレームワークのありとあらゆる部分の周りに、自分のインタフェースを差し込むことでコンフィグレーションすることができる。もちろん、我々はオプション実装として便利なクラスも提供している。
- Spring の MVC は本当にビューに対して懐疑的だ。望まなければ JSP の使用を押し付けられることはない。Velocity や XSLT、あるいは他の

ビューテクノロジを使っても構わない。もし独自のビューメカニズム、例えば自前のテンプレート言語を使いたければ、それを統合するための Spring View インタフェースを簡単に実装することができる。

- Spring Controller は他のオブジェクトのように IoC を使って設定を行う。このことによりテストがしやすくなり、Spring で管理されている他のオブジェクトと綺麗に統合できるようになるのだ。
- Web 層はビジネスオブジェクト層の上の薄い層になる。これにより優れた実践が促進される。Struts やその他の特定の目的用フレームワークだとビジネスオブジェクトの実装を自力でやらないといけない。Spring ではアプリケーションの全ての層に向けた統合フレームワークを提供するのだ。

Struts1.1 でのように、Spring MVC アプリケーションで必要になるのと同じだけのディスパッチャサーブレットを用意することができる。

下記の例は、　簡単な Spring Controller が同じアプリケーションコンテキストで定義されたビジネスオブジェクトにどのようにアクセスできるかを示したものだ。このコントローラは `handleRequest()` メソッドで Google 検索を実行する。

```
public class GoogleSearchController implements Controller {
  private IGoogleSearchPort google;
  private String googleKey;
  public void setGoogle(IGoogleSearchPort google) {
    this.google = google;
  }

  public void setGoogleKey(String googleKey) {
    this.googleKey = googleKey;
  }

  public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    String query = request.getParameter("query");
    GoogleSearchResult result =
    // Google のちゃんとした定義は省略する

    // google ビジネスオブジェクトを使う
    google.doGoogleSearch(this.googleKey, query,
                          start, maxResults, filter, restrict,
                          safeSearch, lr, ie, oe);

    return new ModelAndView("googleResults", "result", result);
  }
}
```

このコードが得られるプロトタイプでは、IGoogleSearchPort は GLUE ウェブサービスプロキシで、それは Spring FactoryBean から返されたものだ。しかしながら、このコントローラはこのコントローラが依存しているウェブサービスライブラリからは IoC によって切り離されている。このインタフェースは下記の議論のように純粋な Java オブジェクト、テストスタブ、擬似オブジェクトあるいは EJB プロキシで実装されたのと同じようにすることができる。このコントローラにはリソースの参照は含まれておらず、ウェブインタラクションをサポートするのに必要なコード以外は含まれていない。

Spring ではデータバインディング、フォーム、ウィザードやもっと複雑なワークフローもサポートされている。

Thomas Risberg の Spring MVC チュートリアル (http://www.springframework.org/docs/MVC-step-by-step/Spring-MVC-step-by-step.html) が Spring MVC フレームワークの入門にとてもいい。他にも「Web MVC with the Spring Framework(http://www.springframework.org/docs/web_mvc.html)」を参照して欲しい.

もし、あなたが自分の好みの MVC フレームワークに満足しているのであれば、Spring がレイヤ構造をもっているので我々の MVC レイヤを除いた残りの Spring を使ってみるのがいいと思う。Spring ユーザの中には、Spring をミドル層の管理やデータアクセスに使ってウェブ層には Struts や WebWork あるいは Tapestry を使っている人もいる。

3.9 EJB を実装する

もしあなたが EJB を使うのであれば、Spring では EJB 実装と EJB アクセスクライアントに関しての重要な利点を提供できる。

ビジネスロジックを EJB ファサードの背後にある POJO へリファクタリングすることは今ではベストプラクティスとして広く受け入れられている。(とりわけ、EJB がコンテナに大きく依存し、切り離してテストするのが非常に難しいので、これによってビジネスロジックのユニットテストを実施するのが簡単になるからだ。)Spring では、EJB Jar ファイルに含まれる XML ドキュメントに従って自動的に BeanFactory をロードすることでユニットテストをとても簡単にできるようになる、便利なセッションビーンやメッセージドリブンビーンのスーパクラスを提供している。

これはつまり、下記のようにステートレスセッション EJB がこのようにコラボレータを取得して使う、ということだ。

```
import org.springframework.ejb.support.AbstractStatelessSessionBean;

public class MyEJB extends AbstractStatelessSessionBean implements MyBusinessInterface {
  private MyPOJO myPOJO;
```

```
  protected void onEjbCreate() {
    this.myPOJO = getBeanFactory().getBean("myPOJO");
  }

  public void myBusinessMethod() {
    this.myPOJO.invokeMethod();
  }
}
```

MyPOJO がインタフェースであると仮定して、この実装クラス (および、プリミティブプロパティや他のコラボレータのようなこれが必要とする全てのコンフィグレーション) は XML のビーンファクトリ定義に隠蔽されている。

Spring には下記のように、標準の `ejb-jar.xml` というデプロイメントデスクリプタにある `ejb/BeanFactoryPath` という名前の環境変数定義から読み込む XML ドキュメントの場所を知らせる。

```
<session>
  <ejb-name>myComponent</ejb-name>
  <local-home>com.test.ejb.myEjbBeanLocalHome</local-home>
  <local>com.mycom.MyComponentLocal</local>
  <ejb-class>com.mycom.MyComponentEJB</ejb-class>
  <session-type>Stateless</session-type>
  <transaction-type>Container</transaction-type>

  <env-entry>
    <env-entry-name>ejb/BeanFactoryPath</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>/myComponent-ejb-beans.xml</env-entry-value></env-entry>
  </env-entry>
</session>
```

この `myComponent-ejb-beans.xml` ファイルはクラスパス、この場合は EJB Jar ファイルのルートからロードされる。各 EJB ごとに独自の XML ドキュメントを指定することができるので、このメカニズムは EJB Jar ファイル毎に何度も使うことができる。

Spring のスーパクラスでは `setySessionContext()` や `ejbCreate()` のような EJB ライフサイクルメソッドを実装が実装されていて、アプリケーション開発者には Spring の `onEjbCreate()` メソッドの実装が自由にできるようにしてある。

3.10 EJB を使う

Spring では EJB を実装するのと同じように、とても簡単に EJB を使えるようになっている。多くの EJB アプリケーションではサービスロケータやビジネスデリケートパターンを使う。これはクライアントコード全体にわたっ

て JNDI ルックアップをするよりはましだけど、通常の実装としてはとても不利だ。例えば下記のような点について。

- EJB を使う典型的なコードはサービスロケータやビジネスデリゲートシングルトンに依存してしまうので、テストするのがとても難しい。
- ビジネスデリゲートなしでサービスロケータパターンを使う場合には、アプリケーションコードが、結局は EJB ホームの create() メソッドから起動され、例外が発生するとそれに対処しないといけない。したがって EJB API や複雑な EJB プログラミングモデルとの密接なつながりが残ってしまう。
- ビジネスデリゲートパターンの実装は結局は、たくさんのコードを複製するということなので、つまり EJB の同じメソッドを呼び出す簡単なメソッドをいっぱい書かないといけない。

こういう様々な理由のため、Sun アドベンチャービルダや OTN J2EE バーチャルショッピングモールのようなアプリケーションにあるような従来の EJB アクセスは生産性を下げることになり、とても複雑なものになってしまうのだ。

Spring は、これをコードレスビジネスデリゲートで克服する。Spring では、他のサービスロケータや他の JDNI ルックアップ、あるいは　ハードコーディングされた複製メソッドは実際の値を追加しない限り必要ない。

例えば、ローカル EJB を使うウェブコントローラがあると考えて欲しい。我々はベストプラクティスに従って EJB ビジネスメソッドインタフェースパターンを使うだろう。その結果、EJB のローカルインタフェースは EJB に準拠しないビジネスメソッドインタフェースを拡張する。(これをやってしまう一番の理由は、ローカルインタフェースのメソッドシグネチャと、ビーンの実装クラスの同期が自動的に行われることを保証するためだ)。このビジネスメソッドインタフェースを MyComponent と呼ぼう。もちろん、この他にもローカルホームインタフェースを実装し、セッションビーンを実装したビーン実装クラスと MyComponent ビジネスメソッドインタフェースを提供する必要がある。

Spring の EJB アクセスでは、Java コードで Web 層コントローラを EJB 実装を結びつけるためにやらないといけないことは、　MyComponent 型のセッターメソッドをコントローラに見せることだけだ。これで下記のようにインスタンス変数としてリファレンスを保存する。

```
private MyComponent myComponent;

public void setMyComponent(MyComponent myComponent) {
  this.myComponent = myComponent;
}
```

我々は、引き続きこのインスタンス変数をあらゆるビジネスメソッド中で使用することができる。

Spring では、このような XML ビーン定義エントリから残りの作業を自動的に行う。LocalStatelessSessionProxyFactoryBean は汎用的なファクトリビーンで、これはどんな EJB にも使うことができる。このファクトリビーンが生成するオブジェクトは Spring によって MyComponent 型へ自動的にキャストすることができる。

```
<bean id="myComponent" class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBe
  <property name="jndiName"><value>myComponent</value></property>
  <property name="businessInterface"><value>com.mycom.MyComponent</value></property>
</bean>

<bean id="myController" class = "com.mycom.myController">
  <property name="myComponent"><ref bean="myComponent"/></property>
</bean>
```

AOP コンセプトを使ったその結果を役に立たせることを強いられることはないが、Spring AOP フレームワークのおかげでちょっとしたハプニングがその背後ではたくさんある。「myComponent」ビーン定義はビジネスメソッドインタフェースを実装した EJB のプロキシを生成する。EJB ローカルホームは起動時にキャッシュされ、そのため JDNI ルックアップも 1 回だけだ。EJB が起動されるたびに、このプロキシはローカル EJB の create() メソッドを叩いて EJB 上の対応するビジネスメソッドを起動する。

myController ビーン定義はこのプロキシにコントローラクラスの myController プロパティを設定する。

この EJB アクセスメカニズムによってアプリケーションコードは大幅に単純化される。

- Web 層のコードは EJB の使用に依存していない。この EJB リファレンスを POJO や擬似オブジェクト、あるいはテストスタブに置き換えたいのであれば、Java コードをいじることなく簡単に myComponent ビーン定義を変更することができる。
- アプリケーションの一部として JNDI ルックアップや他の EJB 準拠なコードを 1 行たりとも書く必要はないのだ。

ベンチマークや実際のアプリケーションでの経験から、このアプローチ (ターゲット EJB のリフレクション的な起動を含めて) のパフォーマンス上オーバヘッドは極小さいものであり、通常の使い方では、検出できないレベルであることが示されている。アプリケーションサーバでは、EJB インフラに関連

したコストがあるので我々は EJB へのきめ細かい呼び出しを行いたくない、ということを覚えておいて欲しい。

同様の `org.springframework.ejb.access.SimpleRemoteStatelessSessionProxyFactoryBean` ファクトリビーンを使って同じアプローチをさらにリモート EJB に適用することも可能だ。しかしながら、リモート EJB のビジネスメソッドインタフェース上の `RemoteException` を隠すことはできない。

3.11 テストする

あなたが引き寄せられたように、私や他の Spring 開発者は包括的なユニットテストの重要性についての熱心な信仰者だ。我々はフレームワークが徹底的にユニットテストされ、フレームワーク設計の一番のゴールがフレームワーク上に構築されたアプリケーションを簡単にユニットテストを実施できるようにすることが必要だと信じている。

Spring そのものは優れたユニットテストスイートを持っている。我々の 1.0M1 時点でのユニットテストカバレッジは 75 %以上で、1.0RC1 では 80%以上のユニットテストカバレッジを達成したいと考えている。我々はテストファースト開発の利点がこのプロジェクトでとてもリアルなものであることに気づいた。例えば、世界中に散らばったチームとして非常に効率よく動くようになったし、ユーザは CVS スナップショットが安定してきたとか十分に使えるというコメントを寄せている。

我々は Spring 上で構築されたアプリケーションのテストがとても簡単なのは以下の理由からだと考えている。

- IoC によりユニットテストが促進される
- アプリケーションには JNDI のような J2EE サービスを直接使ったコードは含まれていない。そんなコードはとてもテストが難しいのだ
- Spring ビーンファクトリやコンテキストはコンテナの外からセットアップが可能

Spring ビーンファクトリをコンテナの外からセットアップできることで、開発プロセスにとって興味深い副産物が得られる。いくつかの Spring を使ったウェブアプリケーションプロジェクトでは、作業はビジネスインタフェースを定義しウェブコンテナの外部の実装のテストを統合することから始まった。ビジネス機能が本質的に完成した後はウェブインタフェースを提供するための薄い層を追加するだけだ。

3.12 誰が Spring を使っているの?

Spring は比較的新しいプロジェクトではあるが、尊敬に値し、なおかつ成長し続けるユーザコミュニティを持っている。また Spring を用いた多くのアプリケーション製品が既に存在する。ユーザには著名な多国籍投資銀行 (大規模プロジェクト)、有名なドットコム企業や様々なウェブ開発コンサルタント、ヘルスケア企業や学術機関も含まれている。

ほとんどのユーザは Spring の全パーツを使っているが、一部コンポネントを単独で使用するユーザもいる。例えば多くのユーザは我々の JDBC や別のデータアクセス機能を使うところから始めている。

3.13 ロードマップ

我々の一番の優先事項は Spring の 1.0 を遅くても今年中にリリースすることだ。でも長期的なゴールもいくつかある。

バージョン 1.0 最終版に向けて予定されている主な拡張は、ソースレベルのメタデータサポートだ。これは、我々の AOP フレームワークを使うためのものだ (でも、制限するものではない)。これによって C#スタイルの属性駆動トランザクション管理が可能になり、宣言的なエンタープライズサービスを典型的な使い方にするのがとても簡単になるだろう。属性のサポートは Spring1.0 の最終リリースには取り込まれる予定であり、リリースされる際には JSR-175 と統合されるように設計される予定だ。

1.0 以降では、下記のような拡張が予定されている。

- 我々の JDBC を比較可能な抽象化することによる JMS サポートとトランザクションのサポート
- ビーンファクトリの動的再設定
- ウェブサービスをエクスポート可能にする
- IDE やその他ツールのサポート

アジャイルプロジェクトとして、我々はユーザの要求に第一に従って進めていく。従って、誰も使っていない機能を開発するようなことはしないし、ユーザコミュニティの声に注意深く耳を傾けるつもりだ。

4 サマリ

Spring は J2EE の多くの共通の問題を解決する強力なフレームワークだ。

Spring ではビジネスオブジェクトを管理する一貫した手段を提供し、クラス指向ではなくインタフェース指向のプログラミングのようなベストプラク

ティスを促進する。Spring のアーキテクチャ上の基本は JavaBean プロパティの利用に基づく Inversion of Control コンテナである。しかしながら、これは全体像のほんの一部に過ぎない。Spring では、アーキテクチャ上の全てのレイヤにおける包括的なソリューションにおいて IoC コンテナを基本的な構造として使っている点で独特なものなのだ。

Spring では独特のデータアクセスの抽象化を提供する。これには、単純で生産性の高い JDBC フレームワークも含まれている。このフレームワークは生産性を大きく改善しエラーの可能性を減少させる。Spring のデータアクセスアーキテクチャは Hibernate やその他の O/R マッピングソリューションとも統合されている。

Spring ではさらに独特のトランザクション管理の抽象化を提供する。これは、JTA や JDBC のような様々なベースとなるトランザクションテクノロジに対して一貫したプログラミングモデルを可能とする。

Spring は標準の Java で書かれた AOP フレームワークを提供する。これは、宣言的なトランザクション管理やその他のエンタープライズサービスを POJO に適用したり、あるいは、-もし必要であれば- 自分で独自のアスペクトを実装できるようにするものである。多くのアプリケーションが伝統的に EJB に関連づけられたキーサービスのいくつかを利用していても、EJB の複雑さを回避できるぐらい強力なのだ。

Spring では、強力で柔軟な MVC ウェブフレームワークを提供する。これは、IoC コンテナに全面的に統合することが可能だ。

4.1 さらなる情報

Spring についてもっと知りたければ以下のリソースを参照して欲しい。

- Expert One-on-One J2EE Design and Development(Rod Johnson,Wrox, 2002,)[邦訳:実践 J2EE システムデザインソフトバンクパブリッシング]. Spring はこの本の出版以降とても進化し、改善されたが、Spring 開発の動機を理解するには今でもここから入るのが一番だ。
- Spring のホームページ: http://www.springframework.org. ここには Javadoc やいくつかのチュートリアルが置いてある。
- Sourceforge のフォーラムやダウンロード
- Spring ユーザや Spring 開発者のメーリングリスト

我々は Spring のドキュメントやサンプルをよりよいものにすべく最善を尽くしている。またフォームやメーリングリストでの問い合わせに対してのレスポンスの速さについても誇りを持っている。我々はあなたがすぐにコミュニティに参加してくれるの歓迎したい。

4.2 著者について

ロッドジョンソンは Java 開発者およびアーキテクトとして 7 年以上の経験を積み、J2EE プラットフォームが出現して以来、これに従事している。彼は Expert One-on-One J2EE Dedign and Development(Wrox, 2002) の著者であり、J2EE に関して他の分権にもいくつか寄稿している。現在は、Wiley の J2EE アーキテクチャに関する次の本を執筆中である。Rod は 2 つの Java 仕様策定委員会のメンバであり、常任のカンファレンススピーカである。現在、英国にてコンサルタントとして働いている。