JBoss Enterprise Application Platform 5

Microcontainer ユーザーガイド

JBoss Enterprise Application Platform 5 での使用向け
エディッション 5.1.2

Last Updated: 2017-10-26

JBoss Enterprise Application Platform 5 Microcontainer ユーザーガイド

JBoss Enterprise Application Platform 5 での使用向け
エディッション 5.1.2

Mark Newton
Red Hat
mark.newton@jboss.org

Aleš Justin
Red Hat
ajustin@redhat.com

編集者

Misty Stanley-Jones
Red Hat
misty@redhat.com

法律上の通知

概要

このガイドは、JBoss Microcontainer を使用してアプリケーションのためにカスタマイズされたモジュール形式の Java 環境を展開したい Java 開発者を対象としています。

目次

パート I. MICROCONTAINER の概要 - 説明付きチュートリアル

第1章 このガイドの使用の前提条件

このガイドの例を使用するには、いくつかのサポートされるソフトウェアをインストールおよび設定し、例のコードをダウンロードする必要があります。

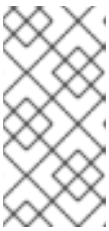
1.1. MAVEN のインストール

このプロジェクトで使用される例では **Maven** v2.2.0 以降が必要です。Apache Maven ホームページから **Maven** を直接ダウンロードし、手順1.1「Maven のインストール」 に示されたようにシステムをインストールおよび設定します。

手順1.1 Maven のインストール

1. **Java Developer Kit 1.6 以上がインストールされていることを確認します。これは Enterprise Platform の要件でもあります。**
   システムに **Java** がインストールされ、**`JAVA_HOME`** 環境変数が **`~/.bash_profile`** (Linux の場合) またはシステムプロパティ (Windows の場合) で設定されていることを確認します。環境変数の設定の詳細については、この手順の手順 ステップ 4 を参照してください。

2. **ダウンロードされた Maven**

   注記

   この手順と今後の手順では、使用しているオペレーティングシステムに応じて Maven が推奨された場所に保存されていることを前提とします。他の Java アプリケーションと同様に、Maven はシステムの適切な任意の場所にインストールできます。

   http://maven.apache.org/download.html にアクセスしてください。

   `apache-maven-2.2.1-bin.zip` などのコンパイルされた zip アーカイブリンクをクリックします。

   リストからダウンロードミラーを選択します。

   Linux ユーザーの場合

   zip アーカイブを **home** ディレクトリに保存します。

   Windows ユーザーの場合

   zip アーカイブを **`C:\Documents and Settings\`*`user_name`*** ディレクトリに保存します。

3. **Maven のインストール**

   Linux ユーザーの場合

   zip ファイルを **home** ディレクトリに解凍します。手順 2 で zip アーカイブを選択し、ディレクトリの名前を変更しない場合は、解凍されたディレクトリの名前が **`apache-maven-`*`version`*** になります。

   Windows ユーザーの場合

   zip アーカイブを C:\Program Files\Apache Software Foundation に解凍します。手順 2 で zip アーカイブを選択し、ディレクトリの名前を変更しない場合は、解凍されたディレクトリの名前が **`apache-maven-`*`version`*** になります。

4. **環境変数の設定**

   Linux ユーザーの場合

   ~/.bash_profile に次の行を追加します。*[username]* を実際のユーザー名に変更し、Maven ディレクトリが実際のディレクトリ名になるようにしてください。バージョン番号は以下に示されているものと異なる場合があります。

   ```
   export M2_HOME=/home/[username]/apache-maven-2.2.1 export
   M2=$M2_HOME/bin export
      PATH=$M2:$PATH
   ```

   パスの先頭に **M2** を含めることにより、インストールした **Maven** バージョンが、デフォルトで使用されます。また、**JAVA_HOME** 環境変数のパスをシステム上の JDK の場所に設定することもできます。

   Windows ユーザーの場合

   M2_HOME、**M2**、および **JAVA_HOME** 環境変数を追加します。

   1. **Start+Pause|Break** を押します。[システムプロパティ] ダイアログボックスが表示されます。
   2. [**詳細設定 (Advanced)**] タブをクリックし、次に [**環境変数 (Environment Variables)**] ボタンをクリックします。
   3. [**システム変数 (System Variables)**] で [**パス (Path)**] を選択します。
   4. [**編集 (Edit)**] をクリックし、各エントリを区切るためにセミコロンを使用して 2 つの **Maven** パスを追加します。パスを引用符で囲む必要はありません。
      - 可変 **M2_HOME** を追加し、パスを **C:\Program Files\Apache Software Foundation\apache-maven-2.2.1** に設定します。
      - 可変 **M2** を追加し、値を **%M2_HOME%\bin** に設定します。
   5. 同じダイアログで、**JAVA_HOME** 環境変数を作成します。
      - 可変 **%JAVA_HOME%** を追加し、値を JDK の場所に設定します (**C:\Program Files\Java\jdk1.6.0_02** など)。
   6. 同じダイアログで、パスの環境変数を更新または作成します。
      - 可変 **%M2%** を追加してコマンドラインから Maven を実行できるようにします。
      - 可変 **%JAVA_HOME%\bin** を追加し、適切な Java インストールへのパスを設定します。
   7. [**システムプロパティ (System Properties)**] ダイアログボックスが閉じるまで [**OK**] をクリックします。

5. **.bash_profile の変更の実装**

   Linux ユーザーのみ

   現在のターミナルセッションで .bash_profile に加えられた変更を更新するには、.bash_profile に対して source を実行します。

```
[localhost]$ source ~/.bash_profile
```

6. **gnome-terminal プロファイルの更新**

   Linux ユーザーのみ

   gnome-terminal (または Konsole ターミナル) の後続する反復で新しい環境変数を読み取るようターミナルプロファイルを更新します。

   1. **[編集 (Edit)]** → **[プロファイル (Profiles)]**をクリックします。
   2. [**デフォルト (Default)**] を選択し、[**編集 (Edit)**] ボタンをクリックします。
   3. [**プロファイルの編集 (Editing Profile)**] ダイアログで、[**タイトルおよびコマンド (Title and Command)**] タブをクリックします。
   4. [**ログインシェルとしてコマンドを実行 (Run command as login shell)**] チェックボックスを選択します。
   5. 開かれたすべてのターミナルダイアログボックスを閉じます。

7. **環境変数の変更と Maven のインストールの確認**

   Linux ユーザーの場合

   変更が正しく実装されたことを確認するために、ターミナルを開き、次のコマンドを実行します。

   - **echo $M2_HOME** を実行します。次の結果が返されます。

     ```
     [localhost]$ echo $M2_HOME /home/username/apache-maven-2.2.1
     ```

   - **echo $M2** を実行します。次の結果が返されます。

     ```
     [localhost]$ echo $M2 /home/username/apache-maven-2.2.1/bin
     ```

   - **echo $PATH** を実行し、**Maven /bin** ディレクトリが含まれることを確認します。

     ```
     [localhost]$ echo $PATH /home/username/apache-maven-2.2.1/bin
     ```

   - **which mvn** を実行します。**Maven** 実行可能ファイルへのパスが表示されます。

     ```
     [localhost]$ which mvn ~/apache-maven-2.2.1/bin/mvn
     ```

   - **mvn -version** を実行します。**Maven** バージョン、関連する Java バージョン、およびオペレーティングシステム情報が表示されます。

     ```
     [localhost]$ $ mvn -version Apache Maven 2.2.1 (r801777; 2009-08-
     07 05:16:01+1000) Java version:
            1.6.0_0 Java home: /usr/lib/jvm/java-1.6.0-openjdk-
     1.6.0.0/jre Default locale: en_US, platform encoding: UTF-8 OS
            name: "Linux" version: "2.6.30.9-96.fc11.i586" arch:
     "i386" Family:
            "unix"
     ```

Windows ユーザーの場合

変更が正しく実装されたことを確認するには、ターミナルを開き、次のコマンドを実行します。

- コマンドプロンプトで、**mvn -version** を実行します。

```
C:\> mvn -version Apache
       Maven 2.2.1 (r801777; 2009-08-06 12:16:01-0700) Java
version: 1.6.0_17 Java home: C:\Sun\SDK\jdk\jre Default
       locale: en_US, platform encoding: Cp1252 OS name: "windows
xp" version: "5.1" arch:
       "x86" Family: "windows"
```

この時点で、このガイドの例で使用する **Maven** が正しく設定されました。

1.2. MICROCONTAINER の例の特殊な MAVEN 設定

Maven は必要に応じて依存関係を取得するモジュールビルドシステムです。このガイドの例では、XML のブロックが **~/.m2/settings.xml** (Linux) または **C:\Documents and Settings*username*\.m2\settings.xml** (Windows) の 例1.1「**settings.xml** サンプルファイル」 に含まれていることを前提とします。このファイルが存在しない場合は、最初にファイルを作成できます。

例1.1 settings.xml サンプルファイル

```
 <settings>
  <profiles>
    <profile>
      <id>jboss-public-repository</id>
      <repositories>
        <repository>
          <id>jboss-public-repository-group</id>
          <name>JBoss Public Maven Repository Group</name>

<url>https://repository.jboss.org/nexus/content/groups/public/</url>
          <layout>default</layout>
          <releases>
            <enabled>true</enabled>
            <updatePolicy>never</updatePolicy>
          </releases>
          <snapshots>
            <enabled>true</enabled>
            <updatePolicy>never</updatePolicy>
          </snapshots>
        </repository>
      </repositories>
      <pluginRepositories>
        <pluginRepository>
          <id>jboss-public-repository-group</id>
          <name>JBoss Public Maven Repository Group</name>

<url>https://repository.jboss.org/nexus/content/groups/public/</url>
          <layout>default</layout>
          <releases>
```

```
            <enabled>true</enabled>
            <updatePolicy>never</updatePolicy>
          </releases>
          <snapshots>
            <enabled>true</enabled>
            <updatePolicy>never</updatePolicy>
          </snapshots>
        </pluginRepository>
      </pluginRepositories>
    </profile>


    <profile>
      <id>jboss-deprecated-repository</id>
      <repositories>
        <repository>
          <id>jboss-deprecated-repository</id>
          <name>JBoss Deprecated Maven Repository</name>

<url>https://repository.jboss.org/nexus/content/repositories/deprecated/
</url>
          <layout>default</layout>
          <releases>
            <enabled>true</enabled>
            <updatePolicy>never</updatePolicy>
          </releases>
          <snapshots>
            <enabled>false</enabled>
            <updatePolicy>never</updatePolicy>
          </snapshots>
        </repository>
      </repositories>
    </profile>
  </profiles>

  <activeProfiles>
    <activeProfile>jboss-public-repository</activeProfile>
    <activeProfile>jboss-deprecated-repository</activeProfile>
  </activeProfiles>

</settings>
```

1.3. 例のダウンロード

このガイドの例では、JBoss Microcontainer に依存する maven プロジェクトを Maven を使用して作成する方法を示しています。これらは images/examples.zip からダウンロードできます。この場所は変更する可能性がありますが、利便性のために含まれています。

例を含む ZIP ファイルをダウンロードしたら、都合が良い場所に解凍し、例を確認してその構造について学んでください。

第2章 MICROCONTAINER の概要

JBoss Microcontainer は JBoss JMX Microkernel のリファクタリングであり、JBoss アプリケーションサーバー外部で直接 POJO 展開とスタンドアロン使用をサポートします。

Microcontainer はオブジェクト指向プログラミングテクニックを使用してソフトウェアを迅速に展開したい Java 開発者の特定のニーズを満たすために設計されています。また、モバイルコンピューティングプラットフォームから大規模なグリッドコンピューティング環境までさまざまなデバイスでソフトウェアを展開できます。

2.1. 機能

- JMX マイクロカーネル のすべての機能
- 直接 POJO 展開 (標準/XMBean または MBeanProxy の必要無し)
- 直接 IOC スタイル依存関係の挿入
- 改善されたライフサイクル管理
- 依存関係に対する追加の制御
- 透過的な AOP 統合
- 仮想ファイルシステム
- 仮想展開フレームワーク
- OSGi クラスローディング

2.2. 定義

このガイドでは、一般的でない用語を使用ことがあります。これらの用語の一部は Microcontainer 定義リスト で定義されています。

Microcontainer 定義リスト

JMX マイクロカーネル

JBoss マイクロカーネルはモジュール形式の Java 環境であり、開発者が環境の一部であるコンポーネントを選択し、残りの作業を任せることができるという点で J2EE などの標準的な環境と異なります。

POJO

Plain Old Java Object (POJO) がモジュール形式の再利用可能な Java オブジェクトです。この名前は該当するオブジェクトが特殊なオブジェクトではなく (特に Enterprise JavaBean ではなく) 通常の Java オブジェクトであることを強調するために使用されます。この用語は Martin Fowler 氏、Rebecca Parsons 氏、および Josh MacKenzie 氏が 2000 年 9 月に行った会議から生まれました (彼らは Entity Bean を使用するのではなくビジネスロジックを通常の java オブジェクトにエンコーディングすることの多くの利点について指摘しました)。

Java Bean

Java Bean はビルダープールで視覚的に操作できる再利用可能なソフトウェアコンポーネントです。

Java Bean は独立したコードの集まりであり、特定のベースクラスまたはインターフェースから継承する必要がありません。Java Bean は主にグラフィカル IDE で作成され、単純なテキストエディタで開発することもできます。

AOP

Aspect-Oriented Programming (AOP) は、二次的な機能またはサポートされる機能が主なビジネスロジックから分離されるプログラミングパラダイムであり、オブジェクト指向プログラミングのサブセットです。

2.3. インストール

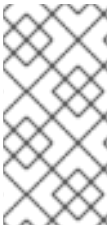
Microcontainer は Enterprise Platform の統合部分です。Enterprise Platform のインストールと設定の詳細については、管理および設定ガイドを参照してください。

第3章 サービスのビルド

サービスは複数のクライアントで必要な作業を実行するコードの集まりです。ここでも目的のために、サービスの定義にいくつかの追加制約を割り当てます。サービスは、クライアントが参照または呼び出すことができる一意の名前を持っている必要があります。サービスの内部はクライアントに対して不可視であり、重要でない必要があります。これは、オブジェクト指向プログラミング (OOP) の「ブラックボックス」という概念です。OOP では、各オブジェクトが独立しており、他のオブジェクトはジョブをどのように行うかについて知る必要がありません。

Microcontainer のコンテキストでは、サービスは POJO からビルドされます。POJO はそれ自体サービスといっていいものですが、一意の名前でアクセスできず、POJO を必要なクライアントにより作成する必要があります。

POJO はクライアントにより実行時に作成する必要がありますが、正しく定義されたインターフェースを提供するために異なるクラスで実装する必要はありません。フィールドとメソッドが削除されず、これらへのアクセスが制限されない限り、新しく作成された POJO を使用するようクライアントを再コンパイルする必要はありません。



注記

インターフェースの実装は、クライアントが**代替の実装**を選択できるようにする場合のみ必要です。インターフェースに対してクライアントがコンパイルされる場合は、クライアントを再コンパイルせずにインターフェースのさまざまな実装を提供できます。インターフェースにより、メソッドシグネチャは変更されなくなります。

このガイドの残りの部分は、ヒューマンリソースの作成と、Microcontainer を使用したアプリケーションのビジネスロジックのキャプチャおよびモジューラ化から構成されます。Microcontainer がインストールされた場合、サンプルコードは **`examples/User_Guide/gettingStarted/humanResourcesService`** に存在します。

3.1. ヒューマンリソースサンプルの概要

例のファイルのディレクトリ構造については、Maven 標準ディレクトリレイアウトが使用されることに注意してください。

Java ソースファイルは **`examples/User_Guide/gettingStarted/humanResourcesService/src/main/java/org/jboss/example/service`** ディレクトリ下のパッケージに存在します (ZIP ファイルを解凍後)。これらの各クラスは特殊なインターフェースを実装しない単純な POJO を表します。最も重要なクラスは、クライアントが呼び出すすべてのパブリックメソッドを提供するサービスエントリポイントを表す **`HRManager`** です。

`HRManager` クラスにより提供されるメソッド

- **`addEmployee`**(Employee ***`employee`***)
- **`removeEmployee`**(Employee ***`employee`***)
- **`getEmployee`**(String ***`firstName`***, String ***`lastName`***)
- **`getEmployees`**()
- **`getSalary`**(Employee ***`employee`***)

- **setSalary**(Employee ***employee***, Integer ***newSalary***)
- **isHiringFreeze**()
- **setHiringFreeze**(boolean ***hiringFreeze***)
- **getSalaryStrategy**()
- **setSalaryStrategy**(SalaryStrategy ***strategy***)

ヒューマンリソースサービスは従業員とその詳細 (この場合は住所や給料) のリストを保持する複数のクラスから構成されます。**SalaryStrategy** インターフェースを使用すると、HRManager を設定でき、さまざまな従業員ロールの給料に下限と上限を設定するためにさまざまな給料方針実装が利用できるようになります。

3.2. HRMANAGER サンプルプロジェクトのコンパイル

ソースコードをコンパイルするには、**humanResourcesService/** ディレクトリから **mvn compile** を発行します。これにより、コンパイルされたクラスを含む **target/classes** という名前の新しいディレクトリが作成されます。プロジェクトをクリーンアップし、ターゲットディレクトリを削除するには、**mvn clean** コマンドを発行します。

3.3. POJO の作成

POJO は使用する前に作成する必要があります。名前で POJO インスタンスへの参照を登録できるようにするネーミングメカニズムが必要です。クライアントは POJO を使用するためにこの名前を必要とします。

Microcontainer は、**コントローラ**のようなメカニズムを提供します。コントローラを使用すると、POJO ベースのサービスをランタイム環境に展開できます。

3.3.1. XML 展開記述子

クラスのコンパイル後に、XML 展開記述子を使用してクラスのインスタンスを作成します。記述子には、個々のインスタンスを表す Bean のリストが含まれます。各 Bean は一意の名前を持つため、実行時にクライアントによって呼び出すことができます。以下の記述子は HRManager のインスタンスを展開します。

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

<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:jboss:bean-deployer:2.0 bean-deployer_2_0.xsd"
xmlns="urn:jboss:bean-deployer:2.0">
  <bean name="HRService" class="org.jboss.example.service.HRManager"/>
</deployment>
```

この XML は **HRManager** クラスのインスタンスを作成し、HRService という名前で登録します。このファイルは実行時に Microcontainer に関連付けられた XML デプロイヤに渡され、実際の展開を実行し、Bean をインスタンス化します。

3.4. POJO をお互いに接続

個々の POJO インスタンスは比較的単純な動作のみを提供できます。POJO の実際の価値は POJO をお互いに接続し、複雑なタスクを実行できることにあります。さまざまな給料方針実装を選択するために、POJO をどのようにお互いに接続できますか?

この XML 展開記述子は以下のようになります。

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

<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:jboss:bean-deployer:2.0 bean-deployer_2_0.xsd"
xmlns="urn:jboss:bean-deployer:2.0">
  <bean name="HRService" class="org.jboss.example.service.HRManager">
    <property name="salaryStrategy"><inject
bean="AgeBasedSalary"/></property>
  </bean>
  <bean name="AgeBasedSalary"
class="org.jboss.example.service.util.AgeBasedSalaryStrategy"/>
</deployment>
```

この XML は追加の <bean> 要素を含めることにより選択された給料方針実装を作成します。今回は AgeBasedSalaryStrategy が選択されます。次に、コードは、HRService Bean を使用して作成された **HRManager** のインスタンスにこの Bean への参照を挿入します。挿入は **HRManager** クラスに **setSalaryStrategy(SalaryStrategy strategy)** メソッドが含まれるため可能です。背後で JBoss Microcontainer は新しく作成された HRManager インスタンスでこのメソッドを呼び出し、**AgeBasedSalaryStrategy** インスタンスへの参照を渡します。

以下のコードを記述した場合、XML 展開記述子により、同じイベントシーケンスが発生します。

```
HRManager hrService = new HRManager();
AgeBasedSalaryStrategy ageBasedSalary = new AgeBasedSalaryStrategy();
hrService.setSalaryStrategy(ageBasedSalary);
```

適切な setter メソッドを使用して挿入を実行する以外に、JBoss Microcontainer は必要に応じてコンストラクタパラメータを使用して挿入を実行することもできます。詳細については、Part II「POJO 開発 (POJO Development)」の章「挿入 (Injection)」を参照してください。

3.4.1. 特別な考慮事項

展開記述子で <bean> 要素を使用するクラスのインスタンスを作成することは可能ですが、これは必ずしも最良の方法ではありません。たとえば、**Employee** クラスと **Address** クラスのインスタンスを作成することは必要ありません。クライアントはユーザーからの入力に応じてこれらのインスタンスを作成します。これらのインスタンスはサービスの一部になりますが、展開記述子で参照されません。

コードのコメント

各 Bean が一意の名前 (上記のように挿入を実行するために使用されます) を持つ限り、展開記述子内で複数の Bean を定義できます。ただし、すべての Bean は必ずしもサービスを表しません。サービスは単一の Bean を使用して実装でき、通常複数の Bean は一緒に使用されます。1 つの Bean は通常サービスエントリポイントを表し、クライアントにより呼び出されるパブリックメソッドを含みます。この例では、エントリポイントは HRService Bean です。XML 展開記述子は Bean がサービスを表すかどうか、または Bean がサービスエントリポイントかどうかを示しません。非サービス Bean からサービス Bean を記述するためにコメントと明確なネーミングスキームを使用することが推奨されます。

3.5. サービスの使用

POJO を作成し、POJO をお互いに接続してサービスを形成した後で、サービスを設定、テスト、およびパッケージ化する必要があります。

3.5.1. サービスの設定

サービスは少なくとも以下の 2 つの方法で設定できます。

- POJO インスタンス間での参照の挿入
- POJO プロパティへの値の挿入

この例では、2 つ目のメソッドが使用されます。以下の展開記述子は以下の方法で HRManager インスタンスを設定します。

- 採用停止が実装されました。
- **AgeBasedSalaryStrategy** は新しい最小および最大給料値を実装します。

POJO インスタンス間で参照を挿入することはサービスの設定の 1 つの方法ですが、POJO プロパティに値を挿入することもできます。以下の展開記述子は採用停止を行うために HRManager インスタンスを設定する方法と新しい最小および最大給料値を指定する AgeBasedSalaryStrategy を設定する方法を示しています。

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

<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:jboss:bean-deployer:2.0 bean-deployer_2_0.xsd"
xmlns="urn:jboss:bean-deployer:2.0">

  <bean name="HRService" class="org.jboss.example.service.HRManager">
    <property name="hiringFreeze">false</property>
    <property name="salaryStrategy"><inject
bean="AgeBasedSalary"/></property>
  </bean>

  <bean name="AgeBasedSalary"
class="org.jboss.example.service.util.AgeBasedSalaryStrategy">
    <property name="minSalary">1000</property> <property
name="maxSalary">80000</property>
  </bean>

</deployment>
```

クラスは値を挿入できるよう関連するプロパティに対してパブリック setter メソッドを持つ必要があります。たとえば、**HRManager** クラスは **setHiringFreeze(boolean hiringFreeze)** メソッドを持ち、**AgeBasedSalaryStrategy** クラスは **setMinSalary(int minSalary)** メソッドと **setMaxSalary(int maxSalary)** メソッドを持ちます。

展開記述子の値は、JavaBean PropertyEditors によって文字列から関連するタイプ (ブール型や整数など) に変換されます。多くの PropertyEditors はデフォルトで標準的なタイプに対して提供されますが、必要な場合は独自のものを作成できます。詳細については、Part II「POJO 開発 (POJO Development)」の章「プロパティ (Properties)」を参照してください。

3.5.2. サービスのテスト

POJO を作成し、POJO をお互いに接続してサービスを形成した後に、POJO をテストする必要があります。JBoss Microcontainer では、**MicrocontainerTest** クラスを使用して個々の POJO とサービスをユニットテストできます。

org.jboss.test.kernel.junit.MicrocontainerTest クラスは **junit.framework.TestCase** から継承され、JBoss Microcontainer をブートストラップし、BasicXMLDeployer を追加することにより各テストを設定します。次に、このクラスは XML 展開記述子のクラスパスをテストクラスと同じ名前で検索します (**.xml** で終了し、クラスのパッケージ名を表すディレクトリ構造に存在します)。このファイルにあるすべての Bean は展開され、**getBean(String name)** という名前の便利なメソッドを使用してアクセスできます。

これらの展開記述子のサンプルは 例3.1「**src/test/resources** ディレクトリのリスト」 にあります。

例3.1 src/test/resources ディレクトリのリスト

```
├── log4j.properties
└── org
    └── jboss
        └── example
            └── service
                ├── HRManagerAgeBasedTestCase.xml
                ├── HRManagerLocationBasedTestCase.xml
                ├── HRManagerTestCase.xml
                └── util
                    ├── AgeBasedSalaryTestCase.xml
                    └── LocationBasedSalaryTestCase.xml
```

テストコードは src/test/java ディレクトリに存在します。

例3.2 src/test/java ディレクトリのリスト

```
└── org
    └── jboss
        └── example
            └── service
                ├── HRManagerAgeBasedTestCase.java
                ├── HRManagerLocationBasedTestCase.java
                ├── HRManagerTestCase.java
                ├── HRManagerTest.java
                ├── HRManagerTestSuite.java
                └── util
                    ├── AgeBasedSalaryTestCase.java
                    ├── LocationBasedSalaryTestCase.java
                    └── SalaryStrategyTestSuite.java
```

テストの基礎として使用する複数の従業員を設定するために、**HRManagerTest** クラスは、**MicrocontainerTest** を拡張します。個々のテストケースはサブクラス **HRManagerTest** を使用して実際の作業を実行します。また、便利さのために個々のテストケースを一緒にグループ化するために使用されるいくつかの **TestSuite** クラスが含まれます。

テストを実行するために、**humanResourcesService/** ディレクトリから **mvn test** を入力します。

各テストを実行する前に関連する XML ファイルから Bean を起動および展開する JBoss Microcontainer を示す **DEBUG** ログ出力を確認する必要があります。テストの終了後に、Bean は展開解除され、Microcontainer はシャットダウンされます。

注記

HRManagerTestCase、**AgeBasedSalaryTestCase**、**LocationBasedSalaryTestCase** などの一部のテストは個々の POJO をユニットテストします。**HRManagerAgeBasedTestCase** や **HRManagerLocationBasedTestCase** などの他のテストはサービス全体をユニットテストします。どちらの方法でも、テストは同じように実行されます。MicrocontainerTest クラスを使用すると、コードの任意の部分に対して包括的なテストを簡単に設定および実行できます。

ここでは、**Address** クラスと **Employee** クラスはテストされません。これらのクラスのテストを記述するのは個々のユーザー次第です。

3.5.3. サービスのパッケージ化

サービスのテスト後は、他のユーザーが使用できるようにサービスをパッケージ化します。これを実行する最も単純な方法は、すべてのクラスを含む JAR を作成することです。サービスを設定する適切なデフォルトの方法が存在する場合は、展開記述子を含めることを選択できます。ただし、これはオプションです。

手順3.1 サービスのパッケージ化

1. **META-INF ディレクトリに展開記述子を格納します (オプション)。**
   展開記述子を含めることを選択する場合は、通常、展開記述子に **jboss-beans.xml** という名前を付け、展開記述子を **META-INF** ディレクトリに格納する必要があります。これは Enterprise Platform のデフォルトレイアウトであり、JAR デプロイヤはこのレイアウトを認識し、自動的に展開を実行します。

   展開記述子は別のファイルとしてヒューマンリソースサンプルに含まれません。これは、このサービスは記述子を直接編集することにより設定されるからです。

2. **JAR を生成します。**
   コンパイルされたすべてのクラスを含む JAR を生成するために、**humanResourcesService/** ディレクトリから **mvn package** を入力します。

3. **他の Maven プロジェクトで JAR を利用できるようにします。**
   他の Maven プロジェクトで JAR を利用できるようにするには、**mvn install** を入力して JAR をローカル Maven レポジトリにコピーします。JAR の最終的なレイアウトは 例 3.3「**org/jboss/example/service** ディレクトリと **META-INF** ディレクトリのリスト」 に示されています。

例3.3 org/jboss/example/service ディレクトリと META-INF ディレクトリのリスト

```
`-- org
    `-- jboss
        `-- example
            `-- service
                |-- Address.java
                |-- Employee.java
                |-- HRManager.java
                `-- util
```

```
                    |-- AgeBasedSalaryStrategy.java
                    |-- LocationBasedSalaryStrategy.java
                    `-- SalaryStrategy.java
`--META-INF
    `-- MANIFEST.MF
    `-- maven
        `-- org.jboss.micrcontainer.examples
         `-- humanResourceService
```

注記

META-INF/maven ディレクトリは Maven により自動的に作成され、異なるビルドシステムを使用する場合は存在しません。

第4章 サービスの使用

前の章では、サービスの作成、設定、テスト、およびパッケージ化について説明しました。次の手順では、サービスを使用して実際の作業を実行するクライアントを作成します。

この例のクライアントは *Text User Interface (TUI)* を使用してユーザーからの入力と出力結果を受け取ります。これにより、サンプルコードのサイズと複雑さが軽減されます。

必要なすべてのファイルは **examples/User_Guide/gettingstarted/commandLineClient** ディレクトリに存在し、例4.1「**examples/User_Guide/gettingstarted/commandLineClient** ディレクトリのリスト」 で示されたように Maven 標準ディレクトリレイアウトに従います。

例4.1 examples/User_Guide/gettingstarted/commandLineClient ディレクトリのリスト

```
├── pom.xml
├── src
│   ├── main
│   │   ├── assembly
│   │   │   ├── aop.xml
│   │   │   ├── classloader.xml
│   │   │   ├── common.xml
│   │   │   └── pojo.xml
│   │   ├── config
│   │   │   ├── aop-beans.xml
│   │   │   ├── classloader-beans.xml
│   │   │   ├── pojo-beans.xml
│   │   │   └── run.sh
│   │   ├── java
│   │   │   └── org
│   │   │       └── jboss
│   │   │           └── example
│   │   │               └── client
│   │   │                   ├── Client.java
│   │   │                   ├── ConsoleInput.java
│   │   │                   ├── EmbeddedBootstrap.java
│   │   │                   └── UserInterface.java
│   │   └── resources
│   │       └── log4j.properties
│   └── test
│       ├── java
│       │   └── org
│       │       └── jboss
│       │           └── example
│       │               └── client
│       │                   ├── ClientTestCase.java
│       │                   ├── ClientTestSuite.java
│       │                   └── MockUserInterface.java
│       └── resources
│           └── jboss-beans.xml
└── target
    └── classes
        └── log4j.properties
```

クライアントは、**org/jboss/example/client** ディレクトリに存在する 3 つのクラスと 1 つのインターフェースから構成されます。

UserInterface は、ユーザー入力を要求するためにクライアントが実行時に呼び出すメソッドを定義します。**ConsoleInput** は、ユーザーがクライアントと対話するために使用する TUI を作成する **UserInterface** の実装です。この設計の利点は後で **UserInterface** の Swing 実装を簡単に作成し、TUI を GUI と置き換えることができることです。また、スクリプトで data-entry プロセスをシミュレーションすることもできます。次に、例3.2「**src/test/java** ディレクトリのリスト」 にある従来の JUnit テストケースを使用してクライアントの動作を自動的にチェックできます。

ビルドが動作するには、最初に **mvn install command** を使用して **examples/User_Guide/gettingStarted/auditAspect** ディレクトリから **auditAspect.jar** をビルドおよびインストールする必要があります。ローカルの Maven リポジトリで利用可能な **auditAspect.jar** に依存する AOP に基づいたものを含む複数のさまざまなクライアントディストリビューションが作成されます。

以前に **examples/User_Guide/gettingStarted** ディレクトリから **mvn install** を入力した場合は、**humanResourcesService.jar** と **auditAspect.jar** が、クライアントとともにすでにビルドおよびパッケージ化されているため、この手順は必要ありません。

ソースコードをコンパイルするために、**commandLineClient** ディレクトリから **mvn package** コマンドを発行したときに 手順4.1「ソースコードのコンパイル」 のすべての手順が実行されます。

手順4.1 ソースコードのコンパイル

1. ユニットテストを実行します。
2. クライアント JAR をビルドします。
3. 必要なすべてのファイルを含むディストリビューションを構築します。

クライアントをコンパイルおよびパッケージ化した後に、**commandLineClient/target** ディレクトリのディレクトリ構造には、例4.2「**commandLineClient/target** ディレクトリのサブディレクトリ」 で説明されたサブディレクトリが含まれます。

> **例4.2 commandLineClient/target ディレクトリのサブディレクトリ**
>
> **client-pojo**
> AOP なしでサービスを呼び出すために使用されます。
>
> **client-cl**
> クラスローディング機能のデモを行うために使用されます。
>
> **client-aop**
> AOP サポートの追加。詳細については、*5章AOP を使用した動作の追加* を参照してください。

各サブディレクトリは、さまざまな設定でクライアントを実行するために必要なすべてのシェルスクリプト、JAR、および XML 展開記述子がある異なるディストリビューションを表します。この章の残りの部分では、例4.3「**client-pojo** ディレクトリのリスト」 に一覧表示される **client-pojo** サブディレクトリにある **client-pojo** ディストリビューションを使用します。

例4.3 client-pojo ディレクトリのリスト

```
        |-- client-1.0.0.jar
        |-- jboss-beans.xml
        |-- lib
        |   |-- concurrent-1.3.4.jar
        |   |-- humanResourcesService-1.0.0.jar
        |   |-- jboss-common-core-2.0.4.GA.jar
        |   |-- jboss-common-core-2.2.1.GA.jar
        |   |-- jboss-common-logging-log4j-2.0.4.GA.jar
        |   |-- jboss-common-logging-spi-2.0.4.GA.jar
        |   |-- jboss-container-2.0.0.Beta6.jar
        |   |-- jboss-dependency-2.0.0.Beta6.jar
        |   |-- jboss-kernel-2.0.0.Beta6.jar
        |   |-- jbossxb-2.0.0.CR4.jar
        |   |-- log4j-1.2.14.jar
        |   `-- xercesImpl-2.7.1.jar
        `-- run.sh
```

クライアントを実行するには、**client-pojo** ディレクトリに移動し、**./run.sh** を入力します。例4.4「HRManager メニュー画面」 が表示されます。

例4.4 HRManager メニュー画面

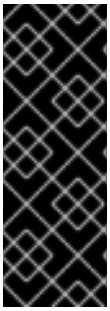
```
        Menu:

        d) Deploy Human Resources service
        u) Undeploy Human Resources service

        a) Add employee
        l) List employees
        r) Remove employee
        g) Get a salary
        s) Set a salary
        t) Toggle hiring freeze

        m) Display menu
        p) Print service status
        q) Quit
        >
```

オプションを選択するには、左側に表示された文字を入力し、**RETURN** を押します。たとえば、メニューオプションを表示するには、**m** を入力し、次に **RETURN** を押します。複数の文字を入力するか、無効なオプションを入力すると、エラーメッセージが表示されます。



重要

run.sh スクリプトは java.ext.dirs システムプロパティを使用して **lib/** ディレクトリにあるすべての JAR をクラスパスに追加することにより実行時環境を設定します。また、-cp フラグを使用して現在のディレクトリと **client-1.0.0.jar** を追加して **jboss-beans.xml** 展開記述子を、アプリケーションを起動するために呼び出される **org.jboss.example.client.Client** クラスとともに実行時に見つけることができます。

4.1. MICROCONTAINER のブートストラップ

クライアントを使用してサービスを展開および呼び出す前に、構築中に起こったことを確認してください。

```
public Client(final boolean useBus) throws Exception {
    this.useBus = useBus;

    ClassLoader cl = Thread.currentThread().getContextClassLoader();
    url = cl.getResource("jboss-beans.xml");

    // Start JBoss Microcontainer
    bootstrap = new EmbeddedBootstrap();
    bootstrap.run();

    kernel = bootstrap.getKernel();
    controller = kernel.getController();
    bus = kernel.getBus();
}
```

最初に、**jboss-beans.xml** 展開記述子を表す URL が作成されます。これは、XML デプロイヤがファイルで宣言された Bean をデプロイおよびデプロイ解除できるようにするために後で必要になります。**jboss-beans.xml** ファイルがクラスパスに含まれるため、アプリケーションクラスローダーの **getResource()** メソッドが使用されます。これはオプションです。URL が有効かつ到達可能である限り、展開記述子の名前と場所は重要ではありません。

次に、XML デプロイヤとともに JBoss Microcontainer のインスタンスが作成されます。このプロセスは *ブートストラッピング*と呼ばれ、**BasicBootstrap** という名前の便利なクラスが Microcontainer の一部として提供され、プログラムを使用して設定できるようになります。XML デプロイヤを追加するには、以下のように **BasicBootstrap** を拡張して **EmbeddedBootstrap** クラスを作成し、保護された **bootstrap()** メソッドをオーバーライドするようにします。

```
public class EmbeddedBootstrap extends BasicBootstrap {
    protected BasicXMLDeployer deployer;
    public EmbeddedBootstrap() throws Exception {
 super();
    }

    public void bootstrap() throws Throwable {
```

```
   super.bootstrap();
   deployer = new BasicXMLDeployer(getKernel());
   Runtime.getRuntime().addShutdownHook(new Shutdown());
      }

      public void deploy(URL url) {
   ...
   deployer.deploy(url);
   ...
       }

      public void undeploy(URL url) {
   ...
   deployer.undeploy(url);
   ...
       }

      protected class Shutdown extends Thread {
   public void run() {
       log.info("Shutting down");
       deployer.shutdown();
   }
      }
}
```

shutdown フックを使用すると、JVM が終了したときに、すべての Bean が正しい順序で展開解除されます。パブリック **deploy**/**undeploy** メソッドは **BasicXMLDeployer** に委譲され、**jboss-beans.xml** で宣言された Bean を展開および展開解除できるようになります。

最後に、Microcontainer コントローラとバスへの参照が復元され、名前で Bean 参照をルックアップし、必要に応じて Bean 参照を直接または間接にアクセスできるようになります。

4.2. サービスの展開

クライアントの作成後に、ヒューマンリソースサービスを展開できます。これは、TUI から **d** オプションを入力することによって行われます。出力は **BasicXMLDeployer** が URL を使用して **jboss-beans.xml** ファイルを解析し、内部に含まれる Bean をインスタンス化したことを示しています。

注記

Bean のクラスが **lib/humanResourcesService.jar** ファイル内部の拡張クラスパスで利用可能であるため、Microcontainer は Bean をインスタンス化できます。また、これらのクラスを展開されたディレクトリ構造に格納し、アプリケーションクラスパスに追加することもできますが、通常はクラスを JAR でパッケージ化する方が便利です。

展開記述子は、**humanResourcesService.jar** ファイルとは完全に独立しています。これにより、テストのために展開記述子を簡単に編集できるようになります。例の **jboss-beans.xml** ファイルには、可能ないくつかの設定を示す XML のコメントアウトされた断片が複数含まれます。

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="urn:jboss:bean-deployer:2.0 bean-
deployer_2_0.xsd"
     xmlns="urn:jboss:bean-deployer:2.0">

  <bean name="HRService" class="org.jboss.example.service.HRManager">
    <!-- <property name="hiringFreeze">true</property>
  <property name="salaryStrategy"><inject bean="AgeBasedSalary"/>
</property> -->
  </bean>

  <!-- <bean name="AgeBasedSalary"
class="org.jboss.example.service.util.AgeBasedSalaryStrategy">
       <property name="minSalary">1000</property>
       <property name="maxSalary">80000</property>
     </bean>

     <bean name="LocationBasedSalary"
    class="org.jboss.example.service.util.LocationBasedSalaryStrategy">
       <property name="minSalary">2000</property>
       <property name="maxSalary">90000</property>
       </bean> -->

</deployment>
```

重要

実行時にサービスをアクセスする方法に応じて、アプリケーションをシャットダウンし、再び起動してサービスを再展開し、変更を確認する必要があることがあります。これにより、アプリケーションの複雑さが減少し、実行時のパフォーマンスが向上します。また、アプリケーションの実行中にサービスを単に再展開することもできます。これにより、複雑さが増加し、実行時のパフォーマンスが低下します。アプリケーションを設計する場合は、このトレードオフを考慮してください。

4.3. 直接アクセス

クライアントの起動時に **run.sh** スクリプトにパラメータが提供されない場合は、サービスの展開後に Microcontainer コントローラを使用して**HRService** Bean への参照がルックアップされます。

```
private HRManager manager;
...
private final static String HRSERVICE = "HRService";

...
void deploy() {
    bootstrap.deploy(url);
    if (!useBus && manager == null) {
 ControllerContext context = controller.getInstalledContext(HRSERVICE);
```

```
  if (context != null) { manager = (HRManager) context.getTarget(); }
     }
}
```

例では、Bean インスタンスへの参照を直接ルックアップする代わりに、最初に **`ControllerContext`** への参照をルックアップし、**`getTarget()`** メソッドを使用してコンテキストから Bean インスタンスへの参照を取得します。Bean は Microcontainer で Microcontainer 内の Bean の状態 に示されたいずれかの状態で存在できます。

Microcontainer 内の Bean の状態

- NOT_INSTALLED
- DESCRIBED
- INSTANTIATED
- CONFIGURED
- INSTALLED

Bean の状態を追跡するには、現在の状態を定義する *コンテキスト*という名前の別のオブジェクトで Bean をラップします。コンテキストの名前は Bean 名と同じです。コンテキストが INSTALLED 状態に到達すると、それを表す Bean を展開するようよう見なされます。

サービスエントリポイントを表す Bean インスタンスへの参照を作成した後に、Bean のメソッドを呼び出して作業を実行できます。

```
@SuppressWarnings("unchecked")

    Set<Employee> listEmployees() {
    if (useBus)
  ...
    else
  return manager.getEmployees();
}
```

クライアントは実際の Bean インスタンスへの参照を使用するため、サービスに直接アクセスします。各メソッド呼び出しは Bean に対して直接行われるため、パフォーマンスは良好です。ただし、アプリケーション実行中にサービスを再設定および再展開する場合はどうなるのでしょうか?

再設定は XML 展開記述子の変更を行い、ファイルを保存することによって実現されます。サービスを再展開するには、現在のインスタンスを展開解除する必要があります。展開解除中に Microcontainer コントローラは、依存する Bean とともに Bean インスタンスへの参照を解放します。これらの Bean は、アプリケーションによって必要とされなくなるため、結果的にガーベッジコレクションの対象とな

ります。サービスを再展開すると、新しい設定を表す新しい Bean インスタンスが作成されます。クライアントからのこれ以降のルックアップは新しいこれらのインスタンスへの参照を取得し、再設定されたサービスにアクセスできます。

問題はサービスエントリポイントを表す Bean インスタンスへの参照が、初めてそのサービスを展開するときにキャッシュされることです。Bean インスタンスはキャッシュされた参照を使用してアクセスでき、クライアントがその参照を開放するまでガーベッジコレクションにより収集されないため、サービスの展開解除は無効です。同様に、クライアントはすでにキャッシュされた参照を持つため、サービスを再び展開した場合に別のルックアップが行われません。したがって、最初のサービス設定を表す Bean インスタンスを使用し続けます。

この動作は、**u** を入力し、次に **RETURN** を押して現在のサービスを展開解除することによってテストできます。サービスが展開解除されていてもクライアントからサービスに引き続きアクセスできます。次に、**jboss-beans.xml** ファイルを使用して設定を一部変更し、ファイルを保存し、**d** オプションを使用して再び展開します。**p** オプションを使用してサービスの状態を出力すると、クライアントが、展開されたサービスの初期インスタンスにアクセスしていることが示されます。

警告

サービスの再展開時に毎回新しい参照をルックアップするクライアントを変更する場合であっても、新しい開発者が間違ってこの参照のコピーを他のオブジェクトに渡すことがあります。再展開時にこれらすべての参照がクリーンアップされない場合、同じキャッシュの問題が発生することがあります。

再設定されたサービスを安定的に再展開するには、'q' オプションを使用して完全にアプリケーションをシャットダウンし、**run.sh** スクリプトを使用して再起動します。トランザクション、メッセージング、永続化などのエンタープライズサービスの場合、一般的にこれらのサービスは常に使用中であるため、これは完全に許容される動作です。これらのサービスは実行時に再展開できず、直接アクセスを使用して得られる高いパフォーマンスの恩恵を受けません。サービスがこのカテゴリに属する場合は、Microcontainer コントローラで直接アクセスを使用することを考慮してください。

4.4. 間接アクセス

run.sh スクリプトはオプションのパラメータ bus を使用して呼び出すことができます。これにより、ヒューマンリソースサービスへの呼び出しは Microcontainer バスを使用するようになります。

Microcontainer コントローラから取得された Bean インスタンスへの直接参照を使用する代わりに、新しい動作はバスで **invoke()** メソッドを呼び出し、Bean 名、メソッド名、メソッド引数、およびメソッドタイプを渡します。バスはこの情報を使用してクライアントの代わりに Bean を呼び出します。

```
private final static String HRSERVICE = "HRService";

...

    @SuppressWarnings("unchecked")
 Set<Employee> listEmployees() {
 if (useBus)
     return (Set<Employee>) invoke(HRSERVICE, "getEmployees", new Object[]
{}, new String[] {});
 else
```

```
     return manager.getEmployees();
    }

private Object invoke(String serviceName, String methodName, Object[]
args, String[] types) {
    Object result = null;
    try {
 result = bus.invoke(serviceName, methodName, args, types);
    } catch (Throwable t) {
 t.printStackTrace();
    }
    return result;
}
```

バスは名前付き Bean インスタンスへの参照をルックアップし、リフレクションを使用して選択されたメソッドを呼び出します。クライアントは Bean インスタンスへの直接参照を持たないため、サービスに間接的にアクセスするといえます。バスは参照をキャッシュしないため、サービス設定に安全に変更を加えることができ、実行時に再展開できます。クライアントによる以降の呼び出しは予期されたように新しい参照を使用します。クライアントとサービスは接続解除されます。

注記

この動作はサービスを展開し、**p** オプションを使用して状態を出力することによってテストできます。**u** オプションを使用してサービスを展開解除し、サービスにアクセスできないことを確認します。次に、**jboss-beans.xml** ファイルを一部変更し、変更内容を保存し、**d** オプションを使用して再び展開します。**p** オプションを使用して再び状態を出力します。クライアントは新しいサービス設定にアクセスします。

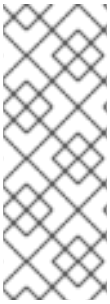
バスはリフレクションを使用して Bean インスタンスを呼び出すため、直接アクセスよりも低速です。この方法の利点は、バスのみが Bean インスタンスへの参照を持つことです。サービスが再展開された場合、既存のすべての参照はクリーンアップし、新しいものと置き換えることができます。このようにして、サービスは実行時に安定的に再展開できます。そんなに頻繁に使用されないサービスや特定のアプリケーションに固有なサービスは、Microcontainer バスを使用した間接アクセスの優良な候補となります。多くの場合、パフォーマンスの低下よりも提供される柔軟性の方が重要です。

4.5. 動的クラスローディング

これまでアプリケーションですべてのクラスをロードするために拡張機能とアプリケーションクラスローダーを使用してきました。ここで示されているように、アプリケーションクラスパスは、現在のディレクトリと **client-1.0.0.jar** を含めるために **run.sh** スクリプトで -cp フラグを使用して設定されます。

```
java -Djava.ext.dirs=`pwd`/lib -cp .:client-1.0.0.jar
org.jboss.example.client.Client $1
```

便利さを追求するために、**lib** ディレクトリ内の JAR は、-cp フラグの後に各 JAR への絶対パスを指定する代わりに java.ext.dirs システムプロパティを使用して拡張クラスローダーのクラスパスに追加されました。**classloader** 拡張は **classloader** アプリケーションの親であるため、クライアントクラスは Microcontainer のすべてのクラスとヒューマンリソースサービスクラスを実行時に見つけることができます。



注記

Java バージョン 6 以上を使用すると、ワイルドカードを使用してすべての JAR を -cp フラグで含めることができます (**`java -cp `pwd`/lib/*:.:client-1.0.0.jar org.jboss.example.client.Client $1`**)。

ここでは、アプリケーションのすべてのクラスがアプリケーションクラスローダーのクラスパスに追加され、拡張クラスローダーのクラスパスはデフォルト値を保持します。

実行時に追加サービスを展開する必要がある場合はどうなりますか? 新しいサービスが JAR ファイルにパッケージ化された場合、クラスをロードする前にサービスがクラスローダーから不可視である必要があります。起動時にアプリケーションクラスローダー (および拡張クラスローダー) のクラスパスはすでに設定されているため、JAR の URL を追加することは簡単ではありません。サービスクラスがディレクトリ構造に含まれる場合に、同じ状況が当てはまります。最上位のディレクトリが現在のディレクトリ (アプリケーションクラスパス上) にない限り、クラスはアプリケーションクラスローダーにより検出されません。

一部のクラスを変更して既存のサービスを再展開する場合は、既存のクラスローダーがクラスを再ロードすることを阻止するセキュリティ上の制約を回避する必要があります。

この目的は、サービスの Bean を展開するために新しいサービスのクラスの場所を認識する、または既存のサービスのクラスの新しいバージョンをロードできる新しいクラスローダーを作成することです。JBoss Microcontainer はこれを実現するために展開記述子で <classloader> 要素を使用します。

`client-cl` ディストリビューションには、例4.5「**`commandLineClient/target/client-cl`** ディレクトリのリスト」 に示されたファイルが含まれます。

例4.5 `commandLineClient/target/client-cl` ディレクトリのリスト

```
|-- client-1.0.0.jar
|-- jboss-beans.xml
|-- lib
|   |-- concurrent-1.3.4.jar
|   |-- jboss-common-core-2.0.4.GA.jar
|   |-- jboss-common-core-2.2.1.GA.jar
|   |-- jboss-common-logging-log4j-2.0.4.GA.jar
|   |-- jboss-common-logging-spi-2.0.4.GA.jar
|   |-- jboss-container-2.0.0.Beta6.jar
|   |-- jboss-dependency-2.0.0.Beta6.jar
|   |-- jboss-kernel-2.0.0.Beta6.jar
|   |-- jbossxb-2.0.0.CR4.jar
|   |-- log4j-1.2.14.jar
|   `-- xercesImpl-2.7.1.jar
|-- otherLib
|   `-- humanResourcesService-1.0.0.jar
|`-- run.sh
```

`humanResourcesService.jar` ファイルは **`otherLib`** という名前の新しいサブディレクトリに移動されました。クラスパスが run.sh スクリプトで設定された拡張またはアプリケーションクラスローダーでは利用できなくなりました。

```
java -Djava.ext.dirs=`pwd`/lib -cp .:client-1.0.0.jar
org.jboss.example.client.Client $1
```

この問題を回避するには、サービスの展開中に新しいクラスローダーを作成し、サービスクラスにロードし、Bean のインスタンスを作成します。これがどのように実行されるかを確認するには、**jboss-beans.xml** ファイルの内容を参照してください。

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

<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="urn:jboss:bean-deployer:2.0 bean-
deployer_2_0.xsd"
     xmlns="urn:jboss:bean-deployer:2.0">

  <bean name="URL" class="java.net.URL">
    <constructor>

<parameter>file:/Users/newtonm/jbossmc/microcontainer/trunk/docs/examples/
User_Guide/gettingStarted/commandLineClient/target/client-
cl.dir/otherLib/humanResourcesService-1.0.0.jar</parameter>
    </constructor>
  </bean>

  <bean name="customCL" class="java.net.URLClassLoader">
    <constructor>
      <parameter>
 <array>
   <inject bean="URL"/>
 </array>
      </parameter>
    </constructor>
  </bean>

  <bean name="HRService" class="org.jboss.example.service.HRManager">
    <classloader><inject bean="customCL"/></classloader>
    <!-- <property name="hiringFreeze">true</property>
  <property name="salaryStrategy"><inject bean="AgeBasedSalary"/>
</property> -->
  </bean>

  <!-- <bean name="AgeBasedSalary"
class="org.jboss.example.service.util.AgeBasedSalaryStrategy">
       <property name="minSalary">1000</property>
       <property name="maxSalary">80000</property>
       </bean>

<bean name="LocationBasedSalary"
class="org.jboss.example.service.util.LocationBasedSalaryStrategy">
<property name="minSalary">2000</property>
<property name="maxSalary">90000</property>
</bean> -->

</deployment>
```

1. 最初に、ローカルファイルシステムで **humanResourcesService.jar** ファイルの場所を指定するために、コンストラクタでパラメータ挿入を使用して **URL** という名前の **java.net.URL** のインスタンスを作成します。

2. 次に、URL Bean をアレイの唯一の要素としてコンストラクタに挿入して **URLClassLoader** のインスタンスを作成します。

3. <classloader> 要素を **HRService** Bean 定義に含め、**customCL** Bean を挿入します。これにより、**HRManager** クラスを customCL クラスローダーによりロードする必要があるようになります。

展開の他の Bean で使用するクラスローダーを決定する方法が必要です。展開のすべての Bean は現在のスレッドのコンテキストクラスローダーを使用します。この場合は、展開を処理するスレッドが起動時にアプリケーションクラスローダーに設定されるコンテキストクラスローダーを持つアプリケーションの主要なスレッドになります。必要な場合は、例4.6「異なるクラスローダーの指定」 で示されているように <classloader> 要素を使用して展開全体に対して異なるクラスローダーを指定できます。

例4.6 異なるクラスローダーの指定

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

<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="urn:jboss:bean-deployer:2.0 bean-
deployer_2_0.xsd"
     xmlns="urn:jboss:bean-deployer:2.0">

  <classloader><inject bean="customCL"/></classloader>

  <bean name="URL" class="java.net.URL">
    <constructor>

<parameter>file:/Users/newtonm/jbossmc/microcontainer/trunk/docs/example
s/User_Guide/gettingStarted/commandLineClient/target/client-
cl.dir/otherLib/humanResourcesService-1.0.0.jar</parameter>
    </constructor>
  </bean>

  <bean name="customCL" class="java.net.URLClassLoader">
    <constructor>
      <parameter>
 <array>
   <inject bean="URL"/>
 </array>
      </parameter>
    </constructor>
  </bean>

  ...

</deployment>
```

これは、**AgeBasedSalary** Bean または **LocationBasedSalary** Bean をコメント解除することによりサービスの再設定を可能にするために必要です。Bean レベルで指定されたクラスローダーは展開レベルのクラスローダーよりも優先されます。展開レベルのクラスローダーを無効にし、Bean に異なるクラスローダーを使用するには、以下のように <null/> 値を使用します。

```
<bean name="HRService" class="org.jboss.example.service.HRManager">
  <classloader><null/></classloader>
</bean>
```

4.5.1. 展開記述子で作成されたクラスローダーの問題

展開記述子を使用してサービスに新しいクラスローダーを作成する場合は、アプリケーションクラスローダーからロードされたクラスにアクセスできないことがあります。HRManager の例では、Microcontainer コントローラを使用するときに、クライアントは Bean インスタンスへの直接参照をキャッシュできなくなります。

この動作を確認するには、**run.sh** コマンドを使用してクライアントを起動し、サービスを展開します。**java.lang.NoClassDefFoundError** 例外がスローされ、アプリケーションが終了します。

このシナリオでは、バスを使用してサービスに間接的にアクセスし、アプリケーションクラスパスでクライアントにより共有されたクラスへのアクセス提供する必要があります。この例では、影響があるクラスは **Address**、**Employee**、および **SalaryStrategy** です。

第5章 AOP を使用した動作の追加

Object Oriented Programming (OOP) では、カプセル化、継承、および多相性を含むソフトウェア開発に役に立つ多くのテクニックを使用できます。ただし、多くの異なるクラスで頻繁に繰り返されるアドレス指定ロジックの問題が解決されません。この例には、ロギング、セキュリティ、従来は各クラスにハードコーディングされるトランザクションロジックがあります。このタイプのロジックは、*クロスカッティングコンサーン*と呼ばれます。

Aspect Oriented Programming (AOP) を使用すると、クラスのコンパイル後にクロスカッティングコンサーンを適用できます。これにより、クラスの主な目的ではないロジックがソースコードから分離され、保守が簡略化されます。このメソッドは AOP 実装に依存します。通常は、クラスがインターフェースを実装する場合は、クラスのインスタンスへの各メソッドコールがプロキシを経由します。このプロキシは同じインターフェースを実装し、必要な動作を追加します。別の方法として、インターフェースが使用されない場合は、コンパイルされたクラスの Java バイトコードが変更されます。元のメソッドは、クロスカッティングロジックを実装するメソッドにより名前が変更され、置換されます。これらの新しいメソッドは、クロスカッティングロジックの実行後に元のメソッドを呼び出します。同じ結果をもたらす別の方法は、バイトコードを変更して、メソッドをオーバーライドする元のクラスのサブクラスを作成することです。オーバーライドされたメソッドは、スーパークラスの対応するメソッドを呼び出す前にクロスカッティングロジックを実行します。

JBoss AOP は AOP 向けのフレームワークです。**JBoss AOP** を使用すると、従来の Java クラスおよびメソッドを使用してクロスカッティングコンサーンを作成できます。AOP の用語では、各コンサーンは単純な POJO を使用して実装する*アスペクト*によって表されます。動作は*アドバイス*と呼ばれるアスペクト内のメソッドにより提供されます。これらのアドバイスはパラメータ、戻り値タイプ、スローされる例外について一定のルールに従います。このフレームワーク内では、クロスカッティングコンサーンを保持しやくする継承、カプセル化、変換などのオブジェクト指向のテクニックを使用できます。アスペクトは対象とするコンストラクタ、メソッド、およびフィールドを指定できる言語を使用してコードに適用されます。設定ファイルを編集することにより、複数のクラスの動作を素早く変更できます。

この章には、Microcontainer とともに JBoss AOP を使用して監査アスペクトを作成し、人事サービスに適用する方法を示す例が含まれます。監査コードは **`HRManager`** クラス内に置くことができますが、主な目的に関係ないコードでクラスがわかりにくくなり、巨大化し、保守しにくくなります。アスペクトの設計はモジュール化を提供し、将来プロジェクトのスコープが変わったときに他のクラスを監査しやすくなります。

また、AOP を使用してデプロイメントフェーズで追加の動作を適用することもできます。この例では、Bean インスタンスへのプロキシを作成し、基本的な JNDI サービスにバインドして、Microcontainer コントローラの代わりに JNDI ルックアップを使用して Bean インスタンスにアクセスできるようにします。

5.1. アスペクトの作成

`examples/User_Guide/gettingStarted/auditAspect` ディレクトリには、アスペクトを作成するのに必要なすべてのファイルが含まれます。

- **`pom.xml`**
- **`src/main/java/org/jboss/example/aspect/AuditAspect.java`**

例5.1 サンプル POJO

```
public class AuditAspect {

    private String logDir;
```

```
    private BufferedWriter out;

    public AuditAspect() {
        logDir = System.getProperty("user.dir") + "/log";

        File directory = new File(logDir);
        if (!directory.exists()) {
            directory.mkdir();
        }
    }

    public Object audit(ConstructorInvocation inv) throws Throwable {
        SimpleDateFormat formatter = new SimpleDateFormat("ddMMyyyy-
kkmmss");
        Calendar now = Calendar.getInstance();
        String filename = "auditLog-" +
formatter.format(now.getTime());

        File auditLog = new File(logDir + "/" + filename);
        auditLog.createNewFile();
        out = new BufferedWriter(new FileWriter(auditLog));
        return inv.invokeNext();
    }

    public Object audit(MethodInvocation inv) throws Throwable {
        String name = inv.getMethod().getName();
        Object[] args = inv.getArguments();
        Object retVal = inv.invokeNext();

        StringBuffer buffer = new StringBuffer();
        for (int i=0; i < args.length; i++) {
            if (i > 0) {
                buffer.append(", ");
            }
            buffer.append(args[i].toString());
        }

        if (out != null) {
            out.write("Method: " + name);
            if (buffer.length() > 0) {
                out.write(" Args: " + buffer.toString());
            }
            if (retVal != null) {
  out.write(" Return: " + retVal.toString());
            }
            out.write("\n");
            out.flush();
        }
        return retVal;
    }
}
```

手順5.1 POJO の作成

1. コンストラクタは現在のワーキングディレクトリに **log** ディレクトリが存在するかどうかを確認し、存在しない場合はそのディレクトリを作成します。

2. 次にアドバイスが定義されます。このアドバイスは、対象クラスのコンストラクタが呼び出されると必ず呼び出されます。これにより、別々のファイルの対象クラスのさまざまなインスタンスに対して実行されたメソッドコールを記録する新しいログファイルが **log** ディレクトリに作成されます。

3. 最後に、別のアドバイスが定義されます。このアドバイスは対象クラスに対して実行された各メソッドコールに適用されます。メソッド名と引数は、戻り値とともに格納されます。この情報は、監査レコードを構築するために使用され、監査レコードは現在のログファイルに書き込まれます。複数のクロスカッティングコンサーンが適用された場合や、対象コンストラクタまたはメソッドを呼び出す場合、各アドバイスは、アドバイスを連結する **inv.invokeNext()** を呼び出します。

注記

各アドバイスは、パラメータとして呼び出しオブジェクトを取得するメソッドを使用して実装され、Throwable をスローし、Object を返します。設計時に、これらのアドバイスが適用されるコンストラクタまたはメソッドがわからない場合は、できるだけタイプを汎用的にしてください。

クラスをコンパイルして、他の例で使用できる **auditAspect.jar** ファイルを作成するには、**auditAspect** ディレクトリから **mvn install** を入力します。

5.2. AOP 用 MICROCONTAINER の設定

HR サービスに監査アスペクトを適用する前に、拡張クラスパスに複数の JAR を追加する必要があります。これらは、**examples/User_Guide/gettingStarted/commandLineClient/target/client-aop.dir** ディレクトリにある **client-aop** ディストリビューションの **lib** サブディレクトリにあります。

例5.2 examples/User_Guide/gettingStarted/commandLineClient/target/client-aop.dir ディレクトリのリスト

```
|-- client-1.0.0.jar
|-- jboss-beans.xml
|-- lib
|-- auditAspect-1.0.0.jar
|-- concurrent-1.3.4.jar
|-- humanResourcesService-1.0.0.jar
|-- javassist-3.6.0.GA.jar
|-- jboss-aop-2.0.0.beta1.jar
|-- jboss-aop-mc-int-2.0.0.Beta6.jar
|-- jboss-common-core-2.0.4.GA.jar
|-- jboss-common-core-2.2.1.GA.jar
|-- jboss-common-logging-log4j-2.0.4.GA.jar
|-- jboss-common-logging-spi-2.0.4.GA.jar
|-- jboss-container-2.0.0.Beta6.jar
```

```
        |-- jboss-dependency-2.0.0.Beta6.jar
        |-- jboss-kernel-2.0.0.Beta6.jar
        |-- jbossxb-2.0.0.CR4.jar
        |-- log4j-1.2.14.jar
        |-- trove-2.1.1.jar
        `-- xercesImpl-2.7.1.jar
        |-- log
        `-- auditLog-18062010-122537
        `-- run.sh
```

最初に、ロジックを実行するために実行時にアスペクトのインスタンスを作成する **lib/auditAspect-1.0.0.jar** が必要です。次に、JBoss AOP (jboss-aop.jar) 用の jar ファイルは依存関係の javassist と trove とともに AOP 機能を追加します。最後に、XML 展開記述子内部でアスペクトを定義できるようにする XML スキーマ定義を含む jboss-aop-mc-int jar が必要です。また、jboss-aop-mc-int jar には、通常の Bean と Microcontainer 内のアスペクト Bean 間の依存関係を作成する統合コードが含まれ、展開時と展開解除フェーズで動作を追加できるようになります。

client-aop ディストリビューションを作成するために Maven2 を使用しているため、**pom.xml** ファイルで適切な依存関係を宣言し、有効なアセンブリ記述子を作成してこれらの JAR ファイルを追加する必要があります。例5.3「AOP 用のサンプル **pom.xml** の一部」 には、サンプルの **pom.xml** の断片が示されています。Ant を使用してビルドを実行する場合は、手順が異なります。

例5.3 AOP 用のサンプル pom.xml の一部

```
<dependency>
  <groupId>org.jboss.microcontainer.examples</groupId>
  <artifactId>jboss-oap</artifactId>
  <version>2.0.0</version>
</dependency>
<dependency>
  <groupId>org.jboss.microcontainer.examples</groupId>
  <artifactId>javassist</artifactId>
  <version>3.6.0.GA</version>
</dependency>
<dependency>
  <groupId>org.jboss.microcontainer.examples</groupId>
  <artifactId>trove</artifactId>
  <version>2.1.1</version>
</dependency>
<dependency>
  <groupId>org.jboss.microcontainer.examples</groupId>
  <artifactId>jboss-aop-mc-int</artifactId>
  <version>2.0.0.Beta6</version>
</dependency>
```

5.3. アスペクトの適用

この時点で有効なディストリビューションには必要なものすべてが含まれるため、監査アスペクトを適用するよう **jboss-beans.xml** を設定できます。このファイルは **examples/User_Guide/gettingStarted/commandLineClient/target/client-aop.dir** に含まれます。

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

<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="urn:jboss:bean-deployer:2.0 bean-
deployer_2_0.xsd"
            xmlns="urn:jboss:bean-deployer:2.0">

  <bean name="AspectManager" class="org.jboss.aop.AspectManager">
    <constructor factoryClass="org.jboss.aop.AspectManager"
   factoryMethod="instance"/>
  </bean>

  <aop:aspect xmlns:aop="urn:jboss:aop-beans:1.0"
       name="AuditAspect" class="org.jboss.example.aspect.AuditAspect"
       method="audit" pointcut="execution(public
org.jboss.example.service.HRManager->new(..)) OR
execution(public * org.jboss.example.service.HRManager->*(..))">
  </aop:aspect>

  ...

</deployment>
```

手順5.2 アスペクトを適用するコードの説明

1. クラスにアスペクトを適用する前に、<bean> 要素を使用して **`org.jboss.aop.AspectManager`** のインスタンスを作成する必要があります。JVM では実行時に AspectManager のインスタンスが 1 つしか必要ないため、ここでは、従来のコンストラクタを呼び出す代わりにファクトリメソッドが使用されます。

2. 次に、<aop:aspect> 要素を使用して AuditAspect と呼ばれるアスペクトのインスタンスが作成されます。これは <bean> 要素に似ており、name 属性と class 属性が同じように使用されます。ただし、これには他のクラス内のコンストラクタとメソッドに対してアスペクト内のアドバイスを適用またはバインドするために使用できる method 属性と pointcut 属性も含まれます。これらの属性は、**`HRManager`** クラス内のすべてのパブリックコンストラクタおよびメソッドに対して監査アドバイスをバインドします。**`audit`** メソッドはさまざまなパラメータで **`AuditAspect`** クラス内にオーバーロードされたため、**`audit`** メソッドのみを指定する必要があります。JBoss AOP はコンストラクタが呼び出されるか、メソッドが呼び出されるかに応じてどれを選択するかを実行時に認識します。

実行時に監査アスペクトを適用するのに必要なのはこの追加設定だけであり、監査動作が **Human Resources** サービスに追加されます。これは、**`run.sh`** スクリプトを使用してクライアントを実行することによりテストできます。**`AuditAspect`** Bean が Microcontainer によって作成される場合、起動時に **`log`** ディレクトリが **`lib`** ディレクトリとともに作成されます。Human Resources サービスの各展開では、新しいログファイルが **`log`** ディレクトリに作成されます。このログファイルには、クライアントからサービスへのすべてのコールのレコードが含まれます。ファイルには **`auditLog-28112007-163902`** のような名前が付けられ、例5.4「AOP ログ出力例」 のような出力が含まれます。

例5.4 AOP ログ出力例

```
        Method: getEmployees Return: []
        Method: addEmployee Args: (Santa Claus, 1 Reindeer Avenue,
Lapland City - 25/12/1860) Return: true
        Method: getSalary Args: (Santa Claus, null - Birth date
unknown) Return: 10000
        Method: getEmployees Return: [(Santa Claus, 1 Reindeer Avenue,
Lapland City - 25/12/1860)]
        Method: isHiringFreeze Return: false
        Method: getEmployees Return: [(Santa Claus, 1 Reindeer Avenue,
Lapland City - 25/12/1860)]
        Method: getSalaryStrategy
```

監査動作を削除するには、展開記述子で関連する XML の部分をコメントアウトし、アプリケーションを再起動します。

警告

展開の順序は重要です。特に、各アスペクトは、適用される Bean の前に宣言して Microcontainer がその順序で展開するようにする必要があります。これは、インスタンスが作成され、コントローラにインスタンスへの参照が保存される前に、クロスカッティングロジックを追加するために Microcontainer が通常の Bean クラスのバイトコードを変更する必要がある場合があるためです。通常の Bean インスタンスがすでに作成されている場合、これは不可能です。

5.4. ライフサイクルコールバック

Microcontainer を使用してインスタンス化する Bean にアスペクトを適用する以外に、展開および展開解除プロセス中に動作を追加することもできます。「直接アクセス」 で説明されたように、Bean は展開されるときに複数の異なる状態を経ます。これらの状態は以下のとおりです。

NOT_INSTALLED

Bean に含まれる展開記述子は、Bean 自体のすべてのアノテーションとともに解析されました。

DESCRIBED

AOP により作成されたすべての依存関係は Bean に追加され、カスタムアノテーションが処理されました。

INSTANTIATED

Bean のインスタンスが作成されました。

CONFIGURED

プロパティが、他の Bean へのすべての参照とともに Bean に挿入されました。

CREATE

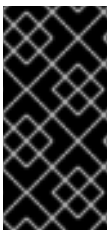
`create` メソッド (Bean で定義された場合) が呼び出されました。

START

`start` メソッド (Bean で定義された場合) が呼び出されました。

INSTALLED

展開記述子で定義されたすべてのカスタムインストールアクションが実行され、Bean にアクセスできる状態になりました。



重要

CREATE と **START** の状態はレガシーのために含まれています。これにより、Enterprise Platform の以前のバージョンで MBean として実装されたサービスが Enterprise Platform 5.1 で Bean として実装されたときに正常に動作するようになります。Bean で対応する create/start メソッドを定義しない場合、Bean はこれらの状態を経ます。

これらの状態は Bean のライフサイクルを表します。<aop> 要素の追加セットを使用して任意の場所に適用する複数のコールバックを定義できます。

<aop:lifecycle-describe>

DESCRIBED 状態になるとき、または DESCRIBED 状態から脱するときに適用される

<aop:lifecycle-instantiate>

INSTANTIATED 状態になるとき、または INSTANTIATED 状態から脱するときに適用される

<aop:lifecycle-configure>

CONFIGURED 状態になるとき、または INSTANTIATED 状態から脱するときに適用される

<aop:lifecycle-create>

CREATE 状態になるとき、または CREATE 状態から脱するときに適用される

<aop:lifecycle-start>

START 状態になるとき、または START 状態から脱するときに適用される

<aop:lifecycle-install>

INSTALLED 状態になるとき、または INSTALLED 状態から脱するときに適用される

<bean> 要素や <aop:aspect> 要素のように、<aop:lifecycle-> 要素には name 属性と class 属性が含まれます。Microcontainer はこれらの属性を使用して **`callback`** クラスのインスタンスを作成し、展開および展開解除中に Bean が関連する状態になる、または関連する状態から脱するときに使用できるよう名前を付けます。例5.5「classes 属性の使用」 で示されているように、これらのクラス属性を使用してコールバックにより影響を受ける Bean を指定できます。

例5.5 classes 属性の使用

```
<aop:lifecycle-install xmlns:aop="urn:jboss:aop-beans:1.0"

name="InstallAdvice"

class="org.jboss.test.microcontainer.support.LifecycleCallback"
```

```
classes="@org.jboss.test.microcontainer.support.Install">

</aop:lifecycle-install>
```

このコードは、Bean クラスが **INSTALLED** の状態になる前と **INSTALLED** の状態から脱した後に、**lifecycleCallback** クラスの追加ロジックが、***@org.jboss.test.microcontainer.support.Install*** でアノテートされたすべての Bean クラスに適用されることを指定します。

例5.6「インストールメソッドとアンインストールメソッド」 で示されているように、コールバッククラスが動作するには、コールバッククラスに ***ControllerContext*** をパラメータとして取得する **install** メソッドと **uninstall** メソッドが含まれる必要があります。

例5.6 インストールメソッドとアンインストールメソッド

```
import org.jboss.dependency.spi.ControllerContext;

public class LifecycleCallback {

    public void install(ControllerContext ctx) {
        System.out.println("Bean " + ctx.getName() + " is being
installed";
    }
    public void uninstall(ControllerContext ctx) {
     System.out.println("Bean " + ctx.getName() + " is being
uninstalled";
    }
}
```

install メソッドは Bean の展開中に呼び出され、**uninstall** メソッドは展開解除中に呼び出されます。

注記

動作はコールバックを使用して展開および展開解除プロセスに追加されますが、ここでは実際に AOP が使用されません。動作が適用される Bean クラスを決定するには、JBoss AOP の *pointcut expression* 機能が使用されます。

5.5. JNDI 経由のサービスルックアップの追加

これまで、サービスを表す Bean インスタンスへの参照をルックアップするために Microcontainer を使用してきました。これは、コントローラにアクセスする前に Microcontainer カーネルへの参照が必要になるため、理想的ではありません。このことについては、例5.7「Bean への参照のルックアップ」 で説明されています。

例5.7 Bean への参照のルックアップ

```
private HRManager manager;
private EmbeddedBootstrap bootstrap;
private Kernel kernel;
private KernelController controller;
private final static String HRSERVICE = "HRService";

...

// Start JBoss Microcontainer
bootstrap = new EmbeddedBootstrap();
bootstrap.run();

kernel = bootstrap.getKernel();
controller = kernel.getController();

...

 ControllerContext context = controller.getInstalledContext(HRSERVICE);
if (context != null) { manager = (HRManager) context.getTarget(); }
```

サービスをルックアップする各クライアントへのカーネル参照を処理することは、Microcontainer 設定へのアクセスを幅広く提供することになるため、セキュリティ上のリクスになります。セキュリティを強化するために、ServiceLocator パターンを適用し、クライアントととしてルックアップを実行するクラスを使用します。ライフサイクルコールバックを使用して、展開時に Bean 参照をその名前とともに ServiceLocator に渡します。このシナリオでは、ServiceLocator は Microcontainer についてまったく知らなくてもルックアップできます。展開解除により、結果的に ServiceLocator から Bean 参照が削除され、ルックアップが行われなくなります。

独自の ServiceLocator 実装を作成することは困難ではありません。JBoss Naming Service (JBoss NS) などの既存のものを統合すると、さらに簡単に作成でき、Java Naming and Directory Interface (JNDI) の仕様に準拠するという利点もあります。JNDI により、クライアントは、共通の API を使用して異なる、場合によっては複数のネーミングサービスにアクセスできるようになります。

手順5.3 独自の ServiceLocator 実装の作成

1. 最初に Microcontainer を使用して JBoss NS のインスタンスを作成します。
2. 次に、展開および展開解除中に Bean 参照のバインディングおよびアンバインディングを実行するライフサイクルコールバックを追加します。
3. アノテーションを使用して参照をバインドする Bean クラスをマークします。
4. この時点で、これまでに説明したショートハンドポイントカット表現を使用して実行時に Bean を見つけることができます。

パート II. MICROCONTAINER の高度なコンセプト

このセクションでは、高度なコンセプトと Microcontainer の興味深い一部の機能について説明します。このガイドの他の部分にあるサンプルコードは例として不完全であると見なされ、必要に応じてこれらを解釈および拡張することはプログラマの責任となります。

第6章 コンポーネントモデル

JBoss Microcontainer は複数の人気のある POJO コンポーネントモデル内で動作します。コンポーネントは、高度なアプリケーションを簡単に開発および構築できる再利用可能なソフトウェアプログラムです。これらのコンポーネントモデルとの効果的な統合は Microcontainer の重要な目的です。Microcontainer で使用できる人気のあるコンポーネントモデルは JMX、Spring、Guice などです。

6.1. コンポーネントモデルとの可能な対話

いくつかの人気があるコンポーネントモデルとの対話について説明する前に、可能な対話タイプについて理解することが重要です。JMX MBean はコンポーネントモデルの 1 つの例です。これらの対話には、MBean 操作の実行、属性の参照、属性の設定、名前付き MBean 間の明示的な依存関係の宣言が含まれます。

Microcontainer のデフォルトの動作と対話は、通常他の *Inversion of Control (IoC)* コンテナから取得でき、MBean により提供される機能に似ています (操作のプレーンなメソッド呼び出し、属性の setter/getter、明示的な依存関係を含む)。

6.2. 依存関係を持たない BEAN

例6.1「プレーンな POJO の展開記述子」 は、依存関係がないプレーンな POJO の展開記述子を示しています。これは、Microcontainer を Spring または Guice と統合する土台となります。

例6.1 プレーンな POJO の展開記述子

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <bean name="PlainPojo" class="org.jboss.demos.models.plain.Pojo"/>

  <beanfactory name="PojoFactory"
class="org.jboss.demos.models.plain.Pojo">
    <property
name="factoryClass">org.jboss.demos.models.plain.PojoFactory</property>
  </beanfactory>

</deployment>
```

6.3. MICROCONTAINER を SPRING と使用

例6.2 Spring がサポートされた記述子

```
<beans xmlns="urn:jboss:spring-beans:2.0">

  <!-- Adding @Spring annotation handler -->
  <bean id="SpringAnnotationPlugin"
class="org.jboss.spring.annotations.SpringBeanAnnotationPlugin" />

  <bean id="SpringPojo" class="org.jboss.demos.models.spring.Pojo"/>

</beans>
```

このファイルの名前領域はプレーンな Microcontainer Bean のファイルと異なります。**urn:jboss:spring-beans:2.0** 名前領域は Spring スキーマポートのバージョンを参照し、Bean の Spring スタイルを定義します。Microcontainer (Spring の Bean ファクトリ表記ではなく) は Bean を展開します。

例6.3 Spring を Microcontainer と使用

```
public class Pojo extends AbstractPojo implements BeanNameAware {
    private String beanName;

    public void setBeanName(String name)
    {
 beanName = name;
    }

    public String getBeanName()
    {
 return beanName;
    }

    public void start()
    {
 if ("SpringPojo".equals(getBeanName()) == false)
     throw new IllegalArgumentException("Name doesn't match: " +
getBeanName());
    }
}
```

BeanNameAware インターフェースを実装することにより、SpringPojo Bean は Spring のライブラリに対する依存関係を持ちますが、この目的はいくつかの Spring のコールバック動作を公開および模倣することです。

6.4. GUICE を MICROCONTAINER と使用

Guice の目的はタイプの照合です。Guice Bean はモジュールを使用して生成および設定されます。

例6.4 Microcontainer での Guice 統合の展開記述子

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <bean name="GuicePlugin"
class="org.jboss.guice.spi.GuiceKernelRegistryEntryPlugin">
```

```
    <constructor>
      <parameter>
 <array elementClass="com.google.inject.Module">
   <bean class="org.jboss.demos.models.guice.PojoModule"/>
 </array>
      </parameter>
    </constructor>
  </bean>

</deployment>
```

このファイルで確認する 2 つの重要な部分は **PojoModule** と **GuiceKernelRegistryEntryPlugin** です。例6.5「Guice に対する Bean の設定」 と同様に、 **PojoModule** は Bean を設定します。例6.6「Guice と Microcontainer の統合」 で示されたように、**GuiceKernelRegistryEntryPlugin** は、Microcontainer との対話を提供します。.

例6.5 Guice に対する Bean の設定

```
public class PojoModule extends AbstractModule {
    private Controller controller;

    @Constructor
 public PojoModule(@Inject(
      bean = KernelConstants.KERNEL_CONTROLLER_NAME)
     Controller controller)
    {
 this.controller = controller;
    }

    protected void configure()
    {
 bind(Controller.class).toInstance(controller);
 bind(IPojo.class).to(Pojo.class).in(Scopes.SINGLETON);
 bind(IPojo.class).annotatedWith(FromMC.class).
     toProvider(GuiceIntegration.fromMicrocontainer(IPojo.class,
"PlainPojo"));
    }
}
```

例6.6 Guice と Microcontainer の統合

```
public class GuiceKernelRegistryEntryPlugin implements
```

```
KernelRegistryPlugin {
    private Injector injector;

    public GuiceKernelRegistryEntryPlugin(Module... modules)
    {
 injector = Guice.createInjector(modules);
    }

    public void destroy()
    {
 injector = null;
    }

    public KernelRegistryEntry getEntry(Object name)
    {
 KernelRegistryEntry entry = null;
 try
     {
  if (name instanceof Class<?>)
      {
   Class<?> clazz = (Class<?>)name;
   entry = new AbstractKernelRegistryEntry(name,
injector.getInstance(clazz));
      }
  else if (name instanceof Key)
      {
   Key<?> key = (Key<?>)name;
   entry = new AbstractKernelRegistryEntry(name,
injector.getInstance(key));
      }
     }
 catch (Exception ignored)
     {
     }
 return entry;
    }
}
```

注記

Injector は **Modules** クラスから作成され、Bean を照合するためにルックアップを行います。レガシー MBean の宣言と使用については、「レガシー MBean とさまざまなコンポーネントモデルの組み合わせ」 を参照してください。

6.5. レガシー MBEAN とさまざまなコンポーネントモデルの組み合わせ

さまざまなコンテンツモデルを組み合わせる最も単純な例は、例6.7「MBean への POJO の挿入」 に示されています。

例6.7 MBean への POJO の挿入

```
<server>

  <mbean code="org.jboss.demos.models.mbeans.Pojo"
name="jboss.demos:service=pojo">
    <attribute name="OtherPojo"><inject bean="PlainPojo"/></attribute>
  </mbean>

</server>
```

Microcontainer で MBean 展開を行うには、コンポーネントモデルのまったく新しいハンドラを記述する必要があります。詳細については、**`system-jmx-beans.xml`** を参照してください。このファイルのコードは JBoss Application Server ソースコードの system-jmx サブプロジェクトに存在します。

6.6. MBEAN としての POJO の公開

例6.8 MBean としての既存の POJO の公開

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <bean name="AnnotatedJMXPojo"
class="org.jboss.demos.models.jmx.AtJmxPojo"/>

  <bean name="XmlJMXPojo" class="org.jboss.demos.models.mbeans.Pojo">

<annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX(exposedInterfa
ce=org.jboss.demos.models.mbeans.PojoMBean.class,
registerDirectly=true)</annotation>
  </bean>

  <bean name="ExposedPojo" class="org.jboss.demos.models.jmx.Pojo"/>

  <bean name="AnnotatedExposePojo"
class="org.jboss.demos.models.jmx.ExposePojo">
    <constructor>
      <parameter><inject bean="ExposedPojo"/></parameter>
    </constructor>
  </bean>

</deployment>
```

この記述子は既存の POJO を MBean として公開し、MBean サーバーに登録します。

POJO を MBean として公開するには、POJO を @JMX アノテーションで終了します (**`org.jboss.aop.microcontainer.aspects.jmx.JMX`** をインポートしたことを前提とします)。Bean は直接公開したり、プロパティで公開したりできます。

例6.9 プロパティを使用して POJO を MBean として公開

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <bean name="XMLLoginConfig"
class="org.jboss.demos.models.old.XMLLoginConfig"/>

  <bean name="SecurityConfig"
class="org.jboss.demos.models.old.SecurityConfig">
    <property name="defaultLoginConfig"><inject
bean="XMLLoginConfig"/></property>
  </bean>

  <bean name="SecurityChecker"
class="org.jboss.demos.models.old.Checker">
    <property name="loginConfig"><inject
bean="jboss.security:service=XMLLoginConfig"/></property>
    <property name="securityConfig"><inject
bean="jboss.security:service=SecurityConfig"/></property>
  </bean>

</deployment>
```

プレーンな POJO をルックアップするか、MBean サーバーから MBean へのハンドルを取得することにより、挿入ルックアップタイプのいずれかを使用できます。挿入オプションの 1 つは、*autowiring* と呼ばれることもあるタイプ挿入を使用します (例6.10「オートワイヤー」 を参照)。

例6.10 オートワイヤー

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <bean name="FromGuice"
class="org.jboss.demos.models.plain.FromGuice">
    <constructor><parameter><inject
bean="PlainPojo"/></parameter></constructor>
    <property name="guicePojo"><inject/></property>
  </bean>

  <bean name="AllPojos" class="org.jboss.demos.models.plain.AllPojos">
    <property name="directMBean"><inject
bean="jboss.demos:service=pojo"/></property>
    <property name="exposedMBean"><inject
bean="jboss.demos:service=ExposedPojo"/></property>
    <property name="exposedMBean"><inject
bean="jboss.demos:service=ExposedPojo"/></property>
  </bean>

</deployment>
```

FromGuice Bean はタイプ照合を使用して Guice Bean を挿入し、**PlainPojo** は一般的な名前挿入で挿入されます。この時点で、Guice バインディングが予期されたように動作するかどうかをテストできます (例6.11「Guice 機能のテスト」 を参照)。

例6.11 Guice 機能のテスト

```
public class FromGuice {
    private IPojo plainPojo;
    private org.jboss.demos.models.guice.Pojo guicePojo;

    public FromGuice(IPojo plainPojo)
    {
 this.plainPojo = plainPojo;
    }

    public void setGuicePojo(org.jboss.demos.models.guice.Pojo
guicePojo)
    {
 this.guicePojo = guicePojo;
    }

    public void start()
    {
 f (plainPojo != guicePojo.getMcPojo())
     throw new IllegalArgumentException("Pojos are not the same: " +
plainPojo + "!=" + guicePojo.getMcPojo());
    }
}
```

例6.11「Guice 機能のテスト」 はエイリアスコンポーネントモデルのみを提供します。エイリアスは小さなことですが必要な機能です。実際の依存関係として実装するために、Microcontainer 内部の新しいコンポーネントモデルとして導入する必要があります。実装の詳細は 例6.12「AbstractController ソースコード」 に示されています。

例6.12 AbstractController ソースコード

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">
  <alias name="SpringPojo">springPojo</alias>
</deployment>
```

この記述子は SpringPojo 名を springPojo エイリアスに対してマッピングします。実際のコンポーネントモデルとしてのエイリアスの利点は、Bean 展開のタイミングが重要でなくなることです。エイリアスは実際の Bean がトリガするまでインストールされていない状態で待機します。

第7章 高度な既存関係の挿入と IOC

今日、*Dependency injection (DI)* (*Inversion of Control (IoC)* とも呼ばれます) は、コンテナまたはコンポーネントモデルの表記を含む多くのフレームワークで中心的な働きをしています。コンポーネントモデルについては以前の章で説明しました。Microcontainer よりも以前にあった JBoss JMX カーネルは、主に MBean サーバーから MBean へのアクセスの制限により、小規模な DI/IoC サポートのみを提供していました。新しい POJO ベースのコンポーネントモデルでは、複数の新しく興味深い機能を使用できます。

この章では、JBoss Microcontainer を使用してさまざまな DI コンセプトを適用する方法について説明します。これらのコンセプトは XML コードを使用して表されますが、これらのほとんどの機能はアノテーションを使用して適用することもできます。

7.1. 値ファクトリ

値ファクトリはユーザーに値を生成することに特化した 1 つまたは複数のメソッドを持つ Bean です。例7.1「値ファクトリ」 を参照してください。

例7.1 値ファクトリ

```
<bean name="Binding" class="org.jboss.demos.ioc.vf.PortBindingManager">
  <constructor>
    <parameter>
      <map keyClass="java.lang.String" valueClass="java.lang.Integer">
 <entry>
   <key>http</key>
   <value>80</value>
 </entry>
 <entry>
   <key>ssh</key>
   <value>22</value>
 </entry>
      </map>
    </parameter>
  </constructor>
</bean>
<bean name="PortsConfig" class="org.jboss.demos.ioc.vf.PortsConfig">
  <property name="http"><value-factory bean="Binding" method="getPort"
parameter="http"/></property>
  <property name="ssh"><value-factory bean="Binding" method="getPort"
parameter="ssh"/></property>
  <property name="ftp">
    <value-factory bean="Binding" method="getPort">
      <parameter>ftp</parameter>
      <parameter>21</parameter>
    </value-factory>
  </property>
  <property name="mail">
    <value-factory bean="Binding" method="getPort">
      <parameter>mail</parameter>
      <parameter>25</parameter>
    </value-factory>
  </property>
</bean>
```

例7.2「PortsConfig」 は、PortsConfig Bean が Binding Bean を使用して **getPort** メソッドを呼び出して値を取得する方法を示しています。

例7.2 PortsConfig

```
public class PortBindingManager {
    private Map<String, Integer> bindings;
    public PortBindingManager(Map<String, Integer> bindings)
    {
 this.bindings = bindings;
    }
    public Integer getPort(String key)
    {
 return getPort(key, null);
    }
    public Integer getPort(String key, Integer defaultValue)
    {
 if (bindings == null)
     return defaultValue;
 Integer value = bindings.get(key);
 if (value != null)
     return value;
 if (defaultValue != null)
     bindings.put(key, defaultValue);
 return defaultValue;
    }
}
```

7.2. コールバック

例7.3「Bean を収集およびフィルタリングするコールバック」 に示された記述子により、ユーザーは特定のタイプのすべての Bean を収集し、一致する Bean の数を制限することができます。

例7.3「Bean を収集およびフィルタリングするコールバック」 の記述子と併せて、例7.4「すべての Editor を収集するパーサー」 に示された Java コードはすべての **Editor** を収集する **Parser** を示しています。

例7.3 Bean を収集およびフィルタリングするコールバック

```
<bean name="checker" class="org.jboss.demos.ioc.callback.Checker">
  <constructor>
    <parameter>
```

```
      <value-factory bean="parser" method="parse">
 <parameter>
   <array elementClass="java.lang.Object">
     <value>http://www.jboss.org</value>
     <value>SI</value>
     <value>3.14</value>
     <value>42</value>
   </array>
 </parameter>
      </value-factory>
    </parameter>
  </constructor>
</bean>
<bean name="editorA" class="org.jboss.demos.ioc.callback.DoubleEditor"/>
<bean name="editorB" class="org.jboss.demos.ioc.callback.LocaleEditor"/>
<bean name="parser" class="org.jboss.demos.ioc.callback.Parser">
  <incallback method="addEditor" cardinality="4..n"/>
  <uncallback method="removeEditor"/>
</bean>
<bean name="editorC" class="org.jboss.demos.ioc.callback.LongEditor"/>
<bean name="editorD" class="org.jboss.demos.ioc.callback.URLEditor"/>
```

例7.4 すべての Editor を収集するパーサー

```
public class Parser {
    private Set<Editor> editors = new HashSet<Editor>();
    ...
 public void addEditor(Editor editor)
    {
 editors.add(editor);
    }
    public void removeEditor(Editor editor)
    {
 editors.remove(editor);
    }
}
```

incallback と **uncallback** は照合のためにメソッド名を使用します。

```
<incallback method="addEditor" cardinality="4..n"/>
<uncallback method="removeEditor"/>
```

下限は Bean が実際に Configured の状態から変化することを可能にするエディタの数を制御します: **`cardinality=4..n/>`**

結果的に、**`Checker`** が作成され、パーサーをチェックします。これについては、例7.5「パーサーのチェッカー」 で示されています。

例7.5 パーサーのチェッカー

```
public void create() throws Throwable {
    Set<String> strings = new TreeSet<String>
(String.CASE_INSENSITIVE_ORDER);
    for (Object element : elements)
 strings.add(element.toString());
    if (expected.equals(strings) == false)
 throw new IllegalArgumentException("Illegal expected set: " + expected
+ "!=" + strings);
}
```

7.3. BEAN アクセスモード

デフォルトの BeanAccessMode では、Bean のフィールドが検査されません。ただし、異なる BeanAccessMode を指定する場合、フィールドは Bean のプロパティの一部としてアクセス可能になります。実装については、例7.6「可能な BeanAccessMode 定義」、例7.7「BeanAccessMode の設定」、および 例7.8「**`FieldsBean`** クラス」 を参照してください。

例7.6 可能な BeanAccessMode 定義

```
public enum BeanAccessMode {
    STANDARD(BeanInfoCreator.STANDARD), // Getters and Setters
 FIELDS(BeanInfoCreator.FIELDS), // Getters/Setters and fields without
getters and setters
 ALL(BeanInfoCreator.ALL); // As above but with non public fields
included
}
```

ここでは、String 値がプライベートな String フィールドに設定されます。

例7.7 BeanAccessMode の設定

```
<bean name="FieldsBean" class="org.jboss.demos.ioc.access.FieldsBean"
access-mode="ALL">
  <property name="string">InternalString</property>
</bean>
```

例7.8 FieldsBean クラス

```
public class FieldsBean {
    private String string;
    public void start()
    {
 if (string == null)
     throw new IllegalArgumentException("Strings should be set!");
    }
}
```

7.4. BEAN エイリアス

各 Bean には任意の数のエイリアスを指定できます。Microcontainer コンポーネント名はオブジェクトとして処理されるため、エイリアスタイプは制限されません。デフォルトでは、システムプロパティの置換は行われません。例7.9「単純な Bean エイリアス」 で示されているように、置換フラグを明示的に設定する必要があります。

例7.9 単純な Bean エイリアス

```
<bean name="SimpleName" class="java.lang.Object">
  <alias>SimpleAlias</alias>
  <alias replace="true">${some.system.property}</alias>
  <alias class="java.lang.Integer">12345</alias>
  <alias><javabean xmlns="urn:jboss:javabean:2.0"
class="org.jboss.demos.bootstrap.Main"/></alias>
</bean>
```

7.5. XML (または METADATA) アノテーションサポート

AOP サポートは JBoss Microcontainer の主要な機能です。AOP アスペクトとプレーンな Bean は任意の組み合わせで使用できます。例7.10「アノテーションに基づいたメソッドの傍受」 はアノテーションに基づいてメソッド呼び出しを傍受しようとします。アノテーションの種類は問いません。true クラスアノテーションでも、xml 設定により追加されたアノテーションでも使用できます。

例7.10 アノテーションに基づいたメソッドの傍受

```
<interceptor xmlns="urn:jboss:aop-beans:1.0" name="StopWatchInterceptor"
class="org.jboss.demos.ioc.annotations.StopWatchInterceptor"/>

<bind xmlns="urn:jboss:aop-beans:1.0" pointcut="execution(*
@org.jboss.demos.ioc.annotations.StopWatchLog->*(..)) OR execution(* *-
>@org.jboss.demos.ioc.annotations.StopWatchLog(..))">
  <interceptor-ref name="StopWatchInterceptor"/>
</bind>
</interceptor>
```

```
public class StopWatchInterceptor implements Interceptor {
    ...
 public Object invoke(Invocation invocation) throws Throwable
    {
 Object target = invocation.getTargetObject();
 long time = System.currentTimeMillis();
 log.info("Invocation [" + target + "] start: " + time);
 try
     {
  return invocation.invokeNext();
     }
 finally
     {
  log.info("Invocation [" + target + "] time: " +
(System.currentTimeMillis() - time));
     }
    }
}
```

例7.11「true クラスがアノテートされたエクスキュータ」 と 例7.12「XML アノテーションを持つ単純なエクスキュータ」 は、エクスキュータを実装するさまざま方法の一部を示しています。

例7.11 true クラスがアノテートされたエクスキュータ

```
<bean name="AnnotatedExecutor"
class="org.jboss.demos.ioc.annotations.AnnotatedExecutor">
```

```
public class AnnotatedExecutor implements Executor {
    ...
 @StopWatchLog // <-- Pointcut match!
     public void execute() throws Exception {
 delegate.execute();
    }
}
```

例7.12 XML アノテーションを持つ単純なエクスキュータ

```
<bean name="SimpleExecutor"
class="org.jboss.demos.ioc.annotations.SimpleExecutor">
  <annotation>@org.jboss.demos.ioc.annotations.StopWatchLog</annotation>
// <-- Pointcut match!
</bean>
```

```
public class SimpleExecutor implements Executor {
    private static Random random = new Random();
    public void execute() throws Exception
    {
 Thread.sleep(Math.abs(random.nextLong() % 101));
    }
}
```

エクスキュータ呼び出し Bean の追加後に、例7.13「エクスキュータログ出力」 などのログ出力を探して展開時に動作しているエクスキュータを確認できます。

例7.13 エクスキュータログ出力

```
    JBoss-MC-Demo  INFO [15-12-2008 13:57:39] StopWatch - Invocation
[org.jboss.demos.ioc.annotations.AnnotatedExecutor@4d28c7] start:
1229345859234
    JBoss-MC-Demo  INFO [15-12-2008 13:57:39] StopWatch - Invocation
[org.jboss.demos.ioc.annotations.AnnotatedExecutor@4d28c7] time: 31
    JBoss-MC-Demo  INFO [15-12-2008 13:57:39] StopWatch - Invocation
[org.jboss.demos.ioc.annotations.SimpleExecutor@1b044df] start:
1229345859265
    JBoss-MC-Demo  INFO [15-12-2008 13:57:39] StopWatch - Invocation
[org.jboss.demos.ioc.annotations.SimpleExecutor@1b044df] time: 47
```

7.6. オートワイヤー

オートワイヤー (つまり、コンテキスト挿入) は、IoC フレームワークで一般的な機能です。例 7.14「オートワイヤーでの包含と除外」 はオートワイヤーで Bean を使用または除外する方法について示しています。

例7.14 オートワイヤーでの包含と除外

```
<bean name="Square" class="org.jboss.demos.ioc.autowire.Square"
autowire-candidate="false"/>
<bean name="Circle" class="org.jboss.demos.ioc.autowire.Circle"/>
<bean name="ShapeUser" class="org.jboss.demos.ioc.autowire.ShapeUser">
  <constructor>
    <parameter><inject/></parameter>
  </constructor>
</bean>
<bean name="ShapeHolder"
class="org.jboss.demos.ioc.autowire.ShapeHolder">
  <incallback method="addShape"/>
  <uncallback method="removeShape"/>
</bean>
<bean name="ShapeChecker"
class="org.jboss.demos.ioc.autowire.ShapesChecker"/>
```

両方のケース (ShapeUser と ShapeChecker) では、Square がコンテキストバインディングで除外されるため、Circle のみを使用する必要があります。

7.7. BEAN ファクトリ

特定の Bean の複数のインスタンスが必要な場合は、Bean ファクトリパターンを使用する必要があります。Microcontainer のジョブは、Bean ファクトリをプレーンな Bean のように設定およびインストールすることです。Bean ファクトリの **createBean** メソッドを呼び出す必要があります。

デフォルトでは、Microcontainer により **GenericBeanFactory** インスタンスが作成されますが、独自のファクトリを設定できます。唯一の制限は、シグネチャおよび設定フックが **AbstractBeanFactory** のいずれかに似ていることです。

例7.15 汎用的な Bean ファクトリ

```
<bean name="Object" class="java.lang.Object"/>
```

```
<beanfactory name="DefaultPrototype"
class="org.jboss.demos.ioc.factory.Prototype">
  <property name="value"><inject bean="Object"/></property>
</beanfactory>
<beanfactory name="EnhancedPrototype"
class="org.jboss.demos.ioc.factory.Prototype"
factoryClass="org.jboss.demos.ioc.factory.EnhancedBeanFactory">
  <property name="value"><inject bean="Object"/></property>
</beanfactory>
<beanfactory name="ProxiedPrototype"
class="org.jboss.demos.ioc.factory.UnmodifiablePrototype"
factoryClass="org.jboss.demos.ioc.factory.EnhancedBeanFactory">
  <property name="value"><inject bean="Object"/></property>
</beanfactory>
<bean name="PrototypeCreator"
class="org.jboss.demos.ioc.factory.PrototypeCreator">
  <property name="default"><inject bean="DefaultPrototype"/></property>
  <property name="enhanced"><inject
bean="EnhancedPrototype"/></property>
  <property name="proxied"><inject bean="ProxiedPrototype"/></property>
</bean>
```

拡張された BeanFactory の使用については、例7.16「拡張された BeanFactory」 を参照してください。

例7.16 拡張された BeanFactory

```
public class EnhancedBeanFactory extends GenericBeanFactory {
    public EnhancedBeanFactory(KernelConfigurator configurator)
    {
 super(configurator);
    }
    public Object createBean() throws Throwable
    {
 Object bean = super.createBean();
 Class clazz = bean.getClass();
 if (clazz.isAnnotationPresent(SetterProxy.class))
     {
  Set<Class> interfaces = new HashSet<Class>();
  addInterfaces(clazz, interfaces);
  return Proxy.newProxyInstance(
           clazz.getClassLoader(),
           interfaces.toArray(new Class[interfaces.size()]),
           new SetterInterceptor(bean)
           );
    }
 else
    {
  return bean;
    }
```

```
    }
    protected static void addInterfaces(Class clazz, Set<Class>
interfaces)
    {
 if (clazz == null)
     return;
 interfaces.addAll(Arrays.asList(clazz.getInterfaces()));
 addInterfaces(clazz.getSuperclass(), interfaces);
    }
    private class SetterInterceptor implements InvocationHandler
    {
 private Object target;
 private SetterInterceptor(Object target)
 {
     this.target = target;
 }
 public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable
 {
     String methodName = method.getName();
     if (methodName.startsWith("set"))
  throw new IllegalArgumentException("Cannot invoke setters.");
     return method.invoke(target, args);
 }
    }
}
```

```
public class PrototypeCreator {
    ...
 public void create() throws Throwable
    {
 ValueInvoker vi1 = (ValueInvoker)bfDefault.createBean();
 vi1.setValue("default");
 ValueInvoker vi2 = (ValueInvoker)enhanced.createBean();
 vi2.setValue("enhanced");
 ValueInvoker vi3 = (ValueInvoker)proxied.createBean();
 try
     {
  vi3.setValue("default");
  throw new Error("Should not be here.");
     }
 catch (Exception ignored)
     {
     }
    }
```

7.8. BEAN メタデータビルダー

コードで Microcontainer を使用する場合は、Bean メタデータを作成および設定するために **`BeanMetaDataBuilder`** を使用します。

例7.17 `BeanMetaDataBuilder`

```
<bean name="BuilderUtil"
class="org.jboss.demos.ioc.builder.BuilderUtil"/>
<bean name="BuilderExampleHolder"
class="org.jboss.demos.ioc.builder.BuilderExampleHolder">
  <constructor>
    <parameter><inject bean="BUExample"/></parameter>
  </constructor>
</bean>
```

このコンセプトを使用すると、Microcontainer 実装の詳細にコードが公開されません。

```
public class BuilderUtil {
    private KernelController controller;
    @Constructor
 public BuilderUtil(@Inject(bean =
KernelConstants.KERNEL_CONTROLLER_NAME) KernelController controller) {
 this.controller = controller;
    }
    public void create() throws Throwable {
 BeanMetaDataBuilder builder =
BeanMetaDataBuilder.createBuilder("BUExample",
BuilderExample.class.getName());
 builder.addStartParameter(Kernel.class.getName(),
builder.createInject(KernelConstants.KERNEL_NAME));
 controller.install(builder.getBeanMetaData());
    }
    public void destroy() {
 controller.uninstall("BUExample");
    }
}
```

7.9. カスタム CLASSLOADER

Microcontainer では、1 つの Bean あたり 1 つのカスタム ClassLoader を定義できます。展開全体に対してクラスローダーを定義する場合は、循環的な依存関係 (たとえば、クラスロード自体に依存する新しく定義されたクラスローダー) を作成しないようにしてください。

例7.18 1 つの Bean あたり 1 つの ClassLoader の定義

```
<classloader><inject bean="custom-classloader:0.0.0"/></classloader>
<!-- this will be explained in future article -->
<classloader name="custom-classloader" xmlns="urn:jboss:classloader:1.0"
export-all="NON_EMPTY" import-all="true"/>
<bean name="CustomCL"
class="org.jboss.demos.ioc.classloader.CustomClassLoader">
  <constructor>
    <parameter><inject bean="custom-classloader:0.0.0"/></parameter>
  </constructor>
```

```
  <property name="pattern">org\.jboss\.demos\.ioc\..+</property>
</bean>
<bean name="CB1" class="org.jboss.demos.ioc.classloader.CustomBean"/>
<bean name="CB2" class="org.jboss.demos.ioc.classloader.CustomBean">
  <classloader><inject bean="CustomCL"/></classloader>
</bean>
```

例7.19「カスタム ClassLoader テスト」 は、CB2 Bean がロード可能なパッケージ範囲を制限するカスタム ClassLoader を使用することを検証するテストを示しています。

例7.19 カスタム ClassLoader テスト

```
public class CustomClassLoader extends ClassLoader {
    private Pattern pattern;
    public CustomClassLoader(ClassLoader parent) {
 super(parent);
    }
    public Class<?> loadClass(String name) throws
ClassNotFoundException {
 if (pattern == null || pattern.matcher(name).matches())
     return super.loadClass(name);
 else
     throw new ClassNotFoundException("Name '" + name + "' doesn't
match pattern: " + pattern);
    }
    public void setPattern(String regexp) {
 pattern = Pattern.compile(regexp);
    }
}
```

7.10. コントローラモード

デフォルトでは、Microcontainer は AUTO コントローラモードを使用します。これにより、Bean は依存関係についてできるだけ遠くにプッシュされます。MANUAL と ON_DEMAND の 2 つの他のモードがあります。

Bean が ON_DEMAND とマークされている場合、Bean は 他の Bean が明示的にその Bean に依存するまで使用またはインストールされません。MANUAL モードでは、Microcontainer ユーザーは状態ラダーでBean を前方または後方にプッシュする必要があります。

例7.20 Bean コントローラモード

```
<bean name="OptionalService"
class="org.jboss.demos.ioc.mode.OptionalService" mode="On Demand"/>
<bean name="OptionalServiceUser"
class="org.jboss.demos.ioc.mode.OptionalServiceUser"/>
<bean name="ManualService"
class="org.jboss.demos.ioc.mode.ManualService" mode="Manual"/>
<bean name="ManualServiceUser"
class="org.jboss.demos.ioc.mode.ManualServiceUser">
  <start>
    <parameter><inject bean="ManualService" fromContext="context"
state="Not Installed"/></parameter>
  </start>
</bean>
```

注記

inject クラスの fromContext 属性を使用して、Bean と変更不可能な Microcontainer コンポーネント表現を挿入できます。

ON_DEMAND および MANUAL Bean 処理を行うために Microcontainer API を使用する方法については、**OptionalServiceUser** のコードを確認してください。

7.11. 循環

Bean はお互いに循環的に依存できます。たとえば、A は構築時に B に依存し、B は設定時に A に依存します。Microcontainer の粒度が細かい状態ライフサイクルの分離により、この問題は簡単に解決できます。

例7.21 Bean ライフサイクル分離

```
<bean name="cycleA" class="org.jboss.demos.ioc.cycle.CyclePojo">
  <property name="dependency"><inject bean="cycleB"/></property>
</bean>
<bean name="cycleB" class="org.jboss.demos.ioc.cycle.CyclePojo">
  <constructor><parameter><inject bean="cycleA"
state="Instantiated"/></parameter></constructor>
</bean>
<bean name="cycleC" class="org.jboss.demos.ioc.cycle.CyclePojo">
  <property name="dependency"><inject bean="cycleD"/></property>
</bean>
<bean name="cycleD" class="org.jboss.demos.ioc.cycle.CyclePojo">
  <property name="dependency"><inject bean="cycleC"
state="Instantiated"/></property>
</bean>
```

7.12. 需要と供給

場合によっては、挿入などで、2 つの Bean 間の依存関係がすぐにわからない場合があります。例 7.22「静的コードの使用」 で示されているように、このような依存関係は明確に記述する必要があります。

例7.22 静的コードの使用

```
<bean name="TMDemand"
class="org.jboss.demos.ioc.demandsupply.TMDemander">
  <demand>TM</demand>
</bean>
<bean name="SimpleTMSupply"
class="org.jboss.demos.ioc.demandsupply.SimpleTMSupplyer">
  <supply>TM</supply>
</bean>
```

7.13. インストール

Bean はさまざまな状態を経るため、他の Bean または同じ Bean でいくつかのメソッドを呼び出したいことがあります。例7.23「さまざまな状態でのメソッドの呼び出し」 は、**Entry** が **RepositoryManager** の **add** メソッドと **removeEntry** メソッドを呼び出して自分自身を登録および登録解除する方法を示しています。

例7.23 さまざまな状態でのメソッドの呼び出し

```
<bean name="RepositoryManager"
class="org.jboss.demos.ioc.install.RepositoryManager">
  <install method="addEntry">
    <parameter><inject fromContext="name"/></parameter>
    <parameter><this/></parameter>
  </install>
  <uninstall method="removeEntry">
    <parameter><inject fromContext="name"/></parameter>
  </uninstall>
</bean>
<bean name="Entry" class="org.jboss.demos.ioc.install.SimpleEntry">
  <install bean="RepositoryManager" method="addEntry"
state="Instantiated">
    <parameter><inject fromContext="name"/></parameter>
    <parameter><this/></parameter>
  </install>
```

```
  <uninstall bean="RepositoryManager" method="removeEntry"
state="Configured">
    <parameter><inject fromContext="name"/></parameter>
  </uninstall>
</bean>
```

7.14. レイジーモック

ほとんど使用されない Bean に対する依存関係が存在し、設定するのに長い時間がかかることがあります。この依存関係を解決するには、例7.24「レイジーモック」 で示されているように、Bean のレイジーモックを使用できます。実際に Bean が必要な場合は、ターゲット Bean を呼び出し使用します (このときまでに Bean がインストールされていることが望まれます)。

例7.24 レイジーモック

```
<bean name="lazyA" class="org.jboss.demos.ioc.lazy.LazyImpl">
  <constructor>
    <parameter>
      <lazy bean="lazyB">
 <interface>org.jboss.demos.ioc.lazy.ILazyPojo</interface>
      </lazy>
    </parameter>
  </constructor>
</bean>
<bean name="lazyB" class="org.jboss.demos.ioc.lazy.LazyImpl">
  <constructor>
    <parameter>
      <lazy bean="lazyA">
 <interface>org.jboss.demos.ioc.lazy.ILazyPojo</interface>
      </lazy>
    </parameter>
  </constructor>
</bean>
<lazy name="anotherLazy" bean="Pojo" exposeClass="true"/>
<bean name="Pojo" class="org.jboss.demos.ioc.lazy.Pojo"/>
```

7.15. ライフサイクル

デフォルトでは、Microcontainer はさまざまな状態を経るときに **create** メソッド、**start** メソッド、および **destroy** のメソッドを使用します。ただし、Microcontainer でこれらのメソッドを呼び出したくないこともあります。このために、ignore フラグが用意されています。

例7.25 Bean ライフサイクル

```
<bean name="FullLifecycleBean-3"
class="org.jboss.demos.ioc.lifecycle.FullLifecycleBean"/>
<bean name="FullLifecycleBean-2"
class="org.jboss.demos.ioc.lifecycle.FullLifecycleBean">
  <create ignored="true"/>
</bean>
<bean name="FullLifecycleBean-1"
class="org.jboss.demos.ioc.lifecycle.FullLifecycleBean">
  <start ignored="true"/>
</bean>
```

第8章 仮想ファイルシステム

リソース処理コードの重複は開発者の共通の問題です。ほとんどの場合、このコードは、ファイル、ディレクトリ、または JAR の場合はリモート URL などの特定のリソースに関する情報の決定を処理します。他の重複の問題はネストされたアーカイブの処理に関するコードです。例8.1「リソースの重複の問題」 はこの問題を示しています。

例8.1 リソースの重複の問題

```
public static URL[] search(ClassLoader cl, String prefix, String suffix)
throws IOException {
    Enumeration[] e = new Enumeration[]{
 cl.getResources(prefix),
 cl.getResources(prefix + "MANIFEST.MF")
    };
    Set all = new LinkedHashSet();
    URL url;
    URLConnection conn;
    JarFile jarFile;
    for (int i = 0, s = e.length; i < s; ++i)
 {
     while (e[i].hasMoreElements())
  {
      url = (URL)e[i].nextElement();
      conn = url.openConnection();
      conn.setUseCaches(false);
      conn.setDefaultUseCaches(false);
      if (conn instanceof JarURLConnection)
   {
       jarFile = ((JarURLConnection)conn).getJarFile();
   }
      else
   {
       jarFile = getAlternativeJarFile(url);
   }
      if (jarFile != null)
   {
       searchJar(cl, all, jarFile, prefix, suffix);
   }
      else
   {
       boolean searchDone = searchDir(all, new
File(URLDecoder.decode(url.getFile(), "UTF-8")), suffix);
       if (searchDone == false)
    {
        searchFromURL(all, prefix, suffix, url);
    }
   }
  }
 }
    return (URL[])all.toArray(new URL[all.size()]);
}

private static boolean searchDir(Set result, File file, String suffix)
throws IOException
{
```

```
    if (file.exists() && file.isDirectory())
 {
     File[] fc = file.listFiles();
     String path;
     for (int i = 0; i < fc.length; i++)
  {
      path = fc[i].getAbsolutePath();
      if (fc[i].isDirectory())
   {
       searchDir(result, fc[i], suffix);
   }
      else if (path.endsWith(suffix))
   {
       result.add(fc[i].toURL());
   }
  }
     return true;
 }
    return false;
}
```

また、Windows システムでのファイルロックには多くの問題があり、開発者はデプロイフォルダのファイルのロックを防ぐ別の場所にホットデプロイ可能なすべてのアーカイブをコピーする必要があります (削除とファイルシステムベースのデプロイ解除が回避されます)。ファイルロックは大きな問題であり、唯一の解決策はすべてのリソースロードコードを一箇所に集めることでした。

VFS プロジェクトはこれらすべての問題を解決するために作成されました。VFS は *仮想ファイルシステム (Virtual File System)* を表します。

8.1. VFS パブリック API

VFS の使用 で示されているように、VFS は主な 2 つの目的のために使用されます。

VFS の使用

- 単純なリソースナビゲーション
- ビジターパターン *API* (Application Programmer Interface)

説明されたように、プレーン JDK では、リソースの処理とナビゲーションは複雑です。常にリソースタイプをチェックする必要があり、これらのチェックは困難な場合があります。VFS はリソースを単一のリソースタイプ VirtualFile に抽象化します。

例8.2 VirtualFile リソースタイプ

```
public class VirtualFile implements Serializable {
    /**
     * Get certificates.
     *
     * @return the certificates associated with this virtual file
     */
    Certificate[] getCertificates()

    /**
```

```
     * Get the simple VF name (X.java)
     *
     * @return the simple file name
     * @throws IllegalStateException if the file is closed
     */
 String getName()

    /**
     * Get the VFS relative path name (org/jboss/X.java)
     *
     * @return the VFS relative path name
     * @throws IllegalStateException if the file is closed
     */
 String getPathName()

    /**
     * Get the VF URL (file://root/org/jboss/X.java)
     *
     * @return the full URL to the VF in the VFS.
     * @throws MalformedURLException if a url cannot be parsed
     * @throws URISyntaxException if a uri cannot be parsed
     * @throws IllegalStateException if the file is closed
     */
 URL toURL() throws MalformedURLException, URISyntaxException

    /**
     * Get the VF URI (file://root/org/jboss/X.java)
     *
     * @return the full URI to the VF in the VFS.
     * @throws URISyntaxException if a uri cannot be parsed
     * @throws IllegalStateException if the file is closed
     * @throws MalformedURLException for a bad url
     */
      URI toURI() throws MalformedURLException, URISyntaxException

    /**
     * When the file was last modified
     *
     * @return the last modified time
     * @throws IOException for any problem accessing the virtual file
system
     * @throws IllegalStateException if the file is closed
     */
      long getLastModified() throws IOException

    /**
     * Returns true if the file has been modified since this method was
last called
     * Last modified time is initialized at handler instantiation.
     *
     * @return true if modifed, false otherwise
     * @throws IOException for any error
     */
      boolean hasBeenModified() throws IOException

    /**
```

```
     * Get the size
     *
     * @return the size
     * @throws IOException for any problem accessing the virtual file
system
     * @throws IllegalStateException if the file is closed
     */
      long getSize() throws IOException

    /**
     * Tests whether the underlying implementation file still exists.
     * @return true if the file exists, false otherwise.
     * @throws IOException - thrown on failure to detect existence.
     */
      boolean exists() throws IOException

    /**
     * Whether it is a simple leaf of the VFS,
     * i.e. whether it can contain other files
     *
     * @return true if a simple file.
     * @throws IOException for any problem accessing the virtual file
system
     * @throws IllegalStateException if the file is closed
     */
      boolean isLeaf() throws IOException

    /**
     * Is the file archive.
     *
     * @return true if archive, false otherwise
     * @throws IOException for any error
     */
      boolean isArchive() throws IOException

    /**
     * Whether it is hidden
     *
     * @return true when hidden
     * @throws IOException for any problem accessing the virtual file
system
     * @throws IllegalStateException if the file is closed
     */
      boolean isHidden() throws IOException

    /**
     * Access the file contents.
     *
     * @return an InputStream for the file contents.
     * @throws IOException for any error accessing the file system
     * @throws IllegalStateException if the file is closed
     */
      InputStream openStream() throws IOException

    /**
     * Do file cleanup.
```

```
     *
     * e.g. delete temp files
     */
      void cleanup()

    /**
     * Close the file resources (stream, etc.)
     */
      void close()

    /**
     * Delete this virtual file
     *
     * @return true if file was deleted
     * @throws IOException if an error occurs
     */
      boolean delete() throws IOException

    /**
     * Delete this virtual file
     *
     * @param gracePeriod max time to wait for any locks (in
milliseconds)
     * @return true if file was deleted
     * @throws IOException if an error occurs
     */
      boolean delete(int gracePeriod) throws IOException

    /**
     * Get the VFS instance for this virtual file
     *
     * @return the VFS
     * @throws IllegalStateException if the file is closed
     */
      VFS getVFS()

    /**
     * Get the parent
     *
     * @return the parent or null if there is no parent
     * @throws IOException for any problem accessing the virtual file
system
     * @throws IllegalStateException if the file is closed
     */
      VirtualFile getParent() throws IOException

    /**
     * Get a child
     *
     * @param path the path
     * @return the child or <code>null</code> if not found
     * @throws IOException for any problem accessing the VFS
     * @throws IllegalArgumentException if the path is null
     * @throws IllegalStateException if the file is closed or it is a
leaf node
     */
```

```
     VirtualFile getChild(String path) throws IOException

    /**
     * Get the children
     *
     * @return the children
     * @throws IOException for any problem accessing the virtual file
system
     * @throws IllegalStateException if the file is closed
     */
     List<VirtualFile> getChildren() throws IOException

    /**
     * Get the children
     *
     * @param filter to filter the children
     * @return the children
     * @throws IOException for any problem accessing the virtual file
system
     * @throws IllegalStateException if the file is closed or it is a
leaf node
     */
     List<VirtualFile> getChildren(VirtualFileFilter filter) throws
IOException

    /**
     * Get all the children recursively<p>
     *
     * This always uses {@link VisitorAttributes#RECURSE}
     *
     * @return the children
     * @throws IOException for any problem accessing the virtual file
system
     * @throws IllegalStateException if the file is closed
     */
     List<VirtualFile> getChildrenRecursively() throws IOException

    /**
     * Get all the children recursively<p>
     *
     * This always uses {@link VisitorAttributes#RECURSE}
     *
     * @param filter to filter the children
     * @return the children
     * @throws IOException for any problem accessing the virtual file
system
     * @throws IllegalStateException if the file is closed or it is a
leaf node
     */
     List<VirtualFile> getChildrenRecursively(VirtualFileFilter
filter) throws IOException

    /**
     * Visit the virtual file system
     *
     * @param visitor the visitor
```

```
     * @throws IOException for any problem accessing the virtual file
system
     * @throws IllegalArgumentException if the visitor is null
     * @throws IllegalStateException if the file is closed
     */
      void visit(VirtualFileVisitor visitor) throws IOException
      }
```

通常の読み取り専用ファイルシステム操作のすべてと、リソースをクリーンアップまたは削除する複数のオプションが利用可能です。クリーンアップまたは削除処理は、ネストされた jar を処理するために作成されたファイルなどのいくつかの内部一時ファイルを扱うときに必要です。

JDK の File または URL リソース処理から新しい VirtualFile に切り替えるには、URL または URI パラメターを使用して **VFS** クラスによって提供されるルート VirtualFile が必要です。

例8.3 VFS クラスの使用

```
public class VFS {
    /**
     * Get the virtual file system for a root uri
     *
     * @param rootURI the root URI
     * @return the virtual file system
     * @throws IOException if there is a problem accessing the VFS
     * @throws IllegalArgumentException if the rootURL is null
     */
    static VFS getVFS(URI rootURI) throws IOException

    /**
     * Create new root
     *
     * @param rootURI the root url
     * @return the virtual file
     * @throws IOException if there is a problem accessing the VFS
     * @throws IllegalArgumentException if the rootURL
     */
    static VirtualFile createNewRoot(URI rootURI) throws IOException

    /**
     * Get the root virtual file
     *
     * @param rootURI the root uri
     * @return the virtual file
     * @throws IOException if there is a problem accessing the VFS
     * @throws IllegalArgumentException if the rootURL is null
     */
    static VirtualFile getRoot(URI rootURI) throws IOException

    /**
     * Get the virtual file system for a root url
     *
     * @param rootURL the root url
     * @return the virtual file system
     * @throws IOException if there is a problem accessing the VFS
```

```
     * @throws IllegalArgumentException if the rootURL is null
     */
    static VFS getVFS(URL rootURL) throws IOException

    /**
     * Create new root
     *
     * @param rootURL the root url
     * @return the virtual file
     * @throws IOException if there is a problem accessing the VFS
     * @throws IllegalArgumentException if the rootURL
     */
    static VirtualFile createNewRoot(URL rootURL) throws IOException

    /**
     * Get the root virtual file
     *
     * @param rootURL the root url
     * @return the virtual file
     * @throws IOException if there is a problem accessing the VFS
     * @throws IllegalArgumentException if the rootURL
     */
    static VirtualFile getRoot(URL rootURL) throws IOException

    /**
     * Get the root file of this VFS
     *
     * @return the root
     * @throws IOException for any problem accessing the VFS
     */
    VirtualFile getRoot() throws IOException
}
```

3 つの異なるメソッドは似ています。

- **getVFS**
- **createNewRoot**
- **getRoot**

getVFS は VFS インスタンスを返しますが、VirtualFile インスタンスを作成しません。これは、VirtualFile ルートを作成するよう指示する前に VFS インスタンス (VFS クラス API の javadoc を参照) を設定するのを支援するメソッドが存在するため、重要です。

その一方で、他の 2 つのメソッドがルート作成にデフォルト設定を使用します。**createNewRoot** と **getRoot** の差異は、キャッシュの詳細にあります。このことについては、後で説明します。

例8.4 getVFS の使用

```
URL rootURL = ...; // get root url
VFS vfs = VFS.getVFS(rootURL);
// configure vfs instance
VirtualFile root1 = vfs.getRoot();
```

```
// or you can get root directly
VirtualFile root2 = VFS.crateNewRoot(rootURL);
VirtualFile root3 = VFS.getRoot(rootURL);
```

他の VFS API について役に立つことは、適切なビジターパターンの実装です。さまざまなリソースを再帰的に収集することは非常に重要です (プレーンな JDK リソースロードでは実行するのが難しいタスク)。

例8.5 リソースの再帰的な収集

```
public interface VirtualFileVisitor {
    /**
     * Get the search attribues for this visitor
     *
     * @return the attributes
     */
    VisitorAttributes getAttributes();

    /**
     * Visit a virtual file
     *
     * @param virtualFile the virtual file being visited
     */
    void visit(VirtualFile virtualFile);
}

VirtualFile root = ...; // get root
VirtualFileVisitor visitor = new SuffixVisitor(".class"); // get all
classes
root.visit(visitor);
```

8.2. VFS アーキテクチャ

パブリック API が非常に直感的である一方で、実際の実装詳細は複雑さを追加します。いくつかのコンセプトについて詳しく説明する必要があります。

VFS インスタンスを作成するたびに、一致する **VFSContext** インスタンスが作成されます。この作成は、**VFSContextFactory** によって行われます。さまざまなプロトコルがさまざまな **VFSContextFactory** インスタンスに対してマッピングされます。たとえば、**file/vfsfile** は **FileSystemContextFactory** に対してマッピングされ、**zip/vfszip** は **ZipEntryContextFactory** に対してマッピングされます。

VirtualFile インスタンスが作成されるたびに、一致する VirtualFileHandler が作成されます。この VirtualFileHandler インスタンスはさまざまなリソースタイプを適切に処理する方法を知っています。**VirtualFile** API は呼出を VirtualFileHandler 参照に委譲します。

VFSContext インスタンスはリソースタイプに応じて VirtualFileHandler インスタンスを作成する方法を知っています。たとえば、ZipEntryContextFactory は **ZipEntryContext** を作成し、次に **ZipEntryHandler** が作成されます。

8.3. 既存の実装

ファイル、ディレクトリ (FileHandler)、および zip アーカイブ (ZipEntryHandler) 以外に、Microcontainer は他の高度な使用ケースもサポートします。最初の例は、Eclipse で *Linked Resources* と呼ばれるものに似ている *Assembled* です。この目的はさまざまなツリーから既存のリソースを取得し、1 つのリソースツリーに「模擬的」に集めることです。

例8.6 アセンブルされれた VirtualFileHandlers の実装

```
AssembledDirectory sar =
AssembledContextFactory.getInstance().create("assembled.sar");

URL url = getResource("/vfs/test/jar1.jar");
VirtualFile jar1 = VFS.getRoot(url);
sar.addChild(jar1);

url = getResource("/tmp/app/ext.jar");
VirtualFile ext1 = VFS.getRoot(url);
sar.addChild(ext);

AssembledDirectory metainf = sar.mkdir("META-INF");

url = getResource("/config/jboss-service.xml");
VirtualFile serviceVF = VFS.getRoot(url);
metainf.addChild(serviceVF);

AssembledDirectory app = sar.mkdir("app.jar");
url = getResource("/app/someapp/classes");
VirtualFile appVF = VFS.getRoot(url);
app.addPath(appVF, new SuffixFilter(".class"));
```

別の実装は*メモリ内*ファイルです。この実装は、AOP で生成されたバイトを簡単に処理する必要性から生まれました。一時ファイルを使用する代わりに、バイトをメモリ内 VirtualFileHandlers にドロップできます。

例8.7 メモリ内 VirtualFileHandlers の実装

```
URL url = new URL("vfsmemory://aopdomain/org/acme/test/Test.class");
byte[] bytes = ...; // some AOP generated class bytes
MemoryFileFactory.putFile(url,  bytes);

VirtualFile classFile = VFS.getVirtualFile(new
URL("vfsmemory://aopdomain"), "org/acme/test/Test.class");
InputStream bis = classFile.openStream(); // e.g. load class from input
stream
```

8.4. 拡張フック

Assembled と **Memory** で行ったように、新しいプロトコルで VFS を拡張することは簡単です。必要なのは **VFSContexFactory**、**VFSContext**、**VirtualFileHandler**、**FileHandlerPlugin**、および **URLStreamHandler** 実装だけです。**VFSContextFactory** は重要でない一方で、他のものはタスクの複雑さに依存します。**rar**、**tar**、**gzip**、または **remote** アクセスを実装できます。

新しいプロトコルの実装後に、新しい VFSContextFactory を VFSContextFactoryLocator で登録します。

8.5. 機能

Microcontainer 開発者が直面する最初の大きな問題の 1 つは、ネストされたリソース (具体的にはネストされた jar ファイル) を適切に使用することです。たとえば、通常の ear 展開は **`gema.ear/ui.war/WEB-INF/lib/struts.jar`** です。

`struts.jar` の内容を読み取るには、2 つのオプションがあります。

- メモリ内のリソースの処理
- ネストされた jar の最上位一時コピーの再帰的な作成

最初のオプションの実装は簡単ですが、メモリを非常に消費し、メモリ内に大きなアプリケーションが存在する可能性があります。他の方法大量の一時ファイルを残します。このファイルはエンドユーザーに対して不可視であり、展開解除後になくなります。

ユーザーがネストされたリソースを参照する VFS URL インスタンスにアクセスするというシナリオを考えてください。

プレーンな VFS がこれを処理する方法は一からパス全体を再作成することです。ネストされたリソースは何回も展開され、大量の一時ファイルが発生します。

Microcontainer は **`VFSRegistry`**、**`VFSCache`**、および **`TempInfo`** を使用してこれを回避します。

VFS (**`createNewRoot`** ではなく **`getRoot`**) を介して VirtualFile を要求する場合、VFS は **`VFSRegistry`** 実装がファイルを提供することを求めます。既存の **`DefaultVFSRegistry`** は最初に一致するルート **`VFSContext`** が提供された URI に対して存在するかどうかを確認します。存在する場合は、**`DefaultVFSRegistry`** は最初に既存の **`TempInfo`** (一時ファイルへのリンク) にナビゲートしようとし、このような一時ファイルが存在しないときに通常のナビゲーションに戻ります。このように、すでに展開された一時ファイルを再利用して、時間とディスク領域を節約できます。キャッシュに一致する **`VFSContext`** がない場合は、コードにより新しい **`VFSCache`** エントリが作成され、デフォルトのナビゲーションのままになります。

`VFSCache` が、キャッシュされた VFSContext エントリを処理する方法は、使用された実装によって異なります。**`VFSCache`** は **`VFSCacheFactory`** を介して設定可能です。デフォルトでは、何もキャッシュされませんが、**`Least Recently Used (LRU)`** や **`timed cache`** などのアルゴリズムを使用した役に立つ複数の既存の **`VFSCache`** 実装が存在します。

第9章 CLASSLOADING レイヤ

JBoss は常にクラスローディングを特異な方法で扱ってきました。Microcontainer に含まれる新しいクラスローディングレイヤも例外ではありません。クラスローディングは非デフォルトのクラスローディングが必要な場合に使用できるオプションのアドオンです。OSGi 形式のクラスローディングの要求が高まり、新しい複数の Java クラスローディング仕様が見込まれる中で、EAP 5.1 のクラスローディングレイヤの変更は役に立ち、タイムリーなことです。

Microcontainer クラスローディングレイヤは抽象レイヤです。詳細のほとんどは private メソッドと package-private メソッドの背後に隠され、API を構成するパブリッククラスとメソッドから利用できる拡張と機能は影響を受けません。つまり、ポリシーに背いてコーディングし、クラスローダーの詳細に背いてコーディングするわけではありません。

ClassLoader プロジェクトは 3 つのサブプロジェクトに分割されます。

- classloader
- classloading
- classloading-vfs

`classloader` には、特定のクラスローディングポリシーなしでカスタム **`java.lang.ClassLoader`** 拡張が 含まれます。クラスローディングポリシーには、ロード元とロード方法の情報が含まれます。

`Classloading` は Microcontainer の依存関係メカニズムの拡張です。VFS 支援の実装は **`classloading-vfs`** です。VFS の実装については、8章*仮想ファイルシステム* を参照してください。

9.1. CLASSLOADER

`ClassLoader` 実装は接続可能なポリシーをサポートし、最終的なクラスであり、変更することを目的としていません。独自の ClassLoader 実装を記述するには、クラスとリソースを探し、クラスローダーと関連付けられた他のルールを指定する単純な API を提供する **`ClassLoaderPolicy`** を記述します。

クラスローディングをカスタマイズするには、**`ClassLoaderPolicy`** をインスタンス化し、**`ClassLoaderSystem`** で登録してカスタム **`ClassLoader`** を作成します。また、**`ClassLoaderDomain`** を作成して **`ClassLoaderSystem`** をパーティション化することもできます。

`ClassLoader` レイヤには、DelegateLoader モデル、クラスローディング、リソースフィルタ、親子委譲ポリシーなどの実装も含まれます。

ランタイムは JMX 対応であり、各クラスローダーに使用されるポリシーを公開します。また、ロード元を調べるためにクラスローディング統計とデバッグメソッドも提供します。

> **例9.1 `ClassLoaderPolicy` クラス**
>
> **`ClassLoaderPolicy`** はクラスローディングがどのように動作するかを制御します。
>
> ```
> public abstract class ClassLoaderPolicy extends BaseClassLoaderPolicy {
>     public DelegateLoader getExported()
>
>         public String[] getPackageNames()
>
>         protected List<? extends DelegateLoader> getDelegates()
> ```

```
        protected boolean isImportAll()
        protected boolean isCacheable()
        protected boolean isBlackListable()

        public abstract URL getResource(String path);

    public InputStream getResourceAsStream(String path)

        public abstract void getResources(String name, Set<URL> urls)
throws IOException;

    protected ProtectionDomain getProtectionDomain(String className,
String path)
        public PackageInformation getPackageInformation(String
packageName)
        public PackageInformation getClassPackageInformation(String
className, String packageName)

        protected ClassLoader isJDKRequest(String name)
        }
}
```

以下に **ClassLoaderPolicy** の 2 つの例を示します。最初の例では正規表現に基づいてリソースを取得し、2 つ目の例では暗号化されたリソースを処理します。

例9.2 正規表現がサポートされた ClassLoaderPolicy

```
public class RegexpClassLoaderPolicy extends ClassLoaderPolicy {
    private VirtualFile[] roots;
    private String[] packageNames;

    public RegexpClassLoaderPolicy(VirtualFile[] roots)
    {
 this.roots = roots;
    }

    @Override
 public String[] getPackageNames()
    {
 if (packageNames == null)
     {
  Set<String> exportedPackages =
PackageVisitor.determineAllPackages(roots, null, ExportAll.NON_EMPTY,
null, null, null);
  packageNames = exportedPackages.toArray(new
String[exportedPackages.size()]);
     }
 return packageNames;
    }
```

```
    protected Pattern createPattern(String regexp)
    {
 boolean outside = true;
 StringBuilder builder = new StringBuilder();
 for (int i = 0; i < regexp.length(); i++)
     {
  char ch = regexp.charAt(i);
  if ((ch == '[' || ch == ']' || ch == '.') && escaped(regexp, i) ==
false)
      {
   switch (ch)
       {
       case '[' : outside = false; break;
       case ']' : outside = true; break;
       case '.' : if (outside) builder.append("\\"); break;
       }
      }

  builder.append(ch);
     }
 return Pattern.compile(builder.toString());
    }

    protected boolean escaped(String regexp, int i)
    {
 return i > 0 && regexp.charAt(i - 1) == '\\';
    }

    public URL getResource(String path)
    {
 Pattern pattern = createPattern(path);
 for (VirtualFile root : roots)
     {
  URL url = findURL(root, root, pattern);
  if (url != null)
      return url;
     }
 return null;
    }

    private URL findURL(VirtualFile root, VirtualFile file, Pattern
pattern)
    {
 try
     {
  String path = AbstractStructureDeployer.getRelativePath(root, file);
  Matcher matcher = pattern.matcher(path);
  if (matcher.matches())
      return file.toURL();

  List<VirtualFile> children = file.getChildren();
  for (VirtualFile child : children)
      {
   URL url = findURL(root, child, pattern);
   if (url != null)
```

```
        return url;
       }

  return null;
     }
 catch (Exception e)
     {
  throw new RuntimeException(e);
     }
    }

    public void getResources(String name, Set<URL> urls) throws
IOException
    {
 Pattern pattern = createPattern(name);
 for (VirtualFile root : roots)
     {
  RegexpVisitor visitor = new RegexpVisitor(root, pattern);
  root.visit(visitor);
  urls.addAll(visitor.getUrls());
     }
    }

    private static class RegexpVisitor implements VirtualFileVisitor
    {
 private VirtualFile root;
 private Pattern pattern;
 private Set<URL> urls = new HashSet<URL>();

 private RegexpVisitor(VirtualFile root, Pattern pattern)
 {
     this.root = root;
     this.pattern = pattern;
 }

 public VisitorAttributes getAttributes()
 {
     return VisitorAttributes.RECURSE_LEAVES_ONLY;
 }

 public void visit(VirtualFile file)
 {
     try
  {
      String path = AbstractStructureDeployer.getRelativePath(root,
file);
      Matcher matcher = pattern.matcher(path);
      if (matcher.matches())
   urls.add(file.toURL());
  }
     catch (Exception e)
  {
      throw new RuntimeException(e);
  }
 }
```

```
 public Set<URL> getUrls()
 {
     return urls;
 }
    }
}
```

RegexpClassLoaderPolicy は単純なメカニズムを使用して一致するリソースを見つけます。実際の実装はさらに包括的であり、洗練されています。

```
public class RegexpService extends PrintService {
    public void start() throws Exception
    {
 System.out.println();

 ClassLoader cl = getClass().getClassLoader();
 Enumeration<URL> urls = cl.getResources("config/[^.]+\\.[^.]{1,4}");
 while (urls.hasMoreElements())
     {
  URL url = urls.nextElement();
  print(url.openStream(), url.toExternalForm());
     }
    }
}
```

正規表現サービスは正規表現パターン **config/[^.]+\\.[^.]{1,4}** を使用して **config/**/ ディレクトリ下にあるリソースを一覧表示します。接尾辞の長さは制限されているため、excluded.properties などのファイル名は無視されます。

例9.3 暗号化がサポートされた ClassLoaderPolicy

```
public class CrypterClassLoaderPolicy extends VFSClassLoaderPolicy {
    private Crypter crypter;

    public CrypterClassLoaderPolicy(String name, VirtualFile[] roots,
VirtualFile[] excludedRoots, Crypter crypter) {
        super(name, roots, excludedRoots);
        this.crypter = crypter;
    }

    @Override
    public URL getResource(String path) {
        try
            {
```

```
                URL resource = super.getResource(path);
                return wrap(resource);
            }
        catch (IOException e)
            {
                throw new RuntimeException(e);
            }
    }

    @Override
    public InputStream getResourceAsStream(String path) {
        InputStream stream = super.getResourceAsStream(path);
        return crypter.crypt(stream);
    }

    @Override
    public void getResources(String name, Set<URL> urls) throws
IOException {
        super.getResources(name, urls);
        Set<URL> temp = new HashSet<URL>(urls.size());
        for (URL url : urls)
            {
                temp.add(wrap(url));
            }
        urls.clear();
        urls.addAll(temp);
    }

    protected URL wrap(URL url) throws IOException {
        return new URL(url.getProtocol(), url.getHost(), url.getPort(),
url.getFile(), new CrypterURLStreamHandler(crypter));
    }
}
```

例9.3「暗号化がサポートされた ClassLoaderPolicy」 は JAR の暗号化方法を示しています。適切なフィルタを指定することにより、暗号化するリソースを設定できます。ここでは、**META-INF/** ディレクトリの内容を除いてすべてのものが暗号化されます。

```
public class EncryptedService extends PrintService {
    public void start() throws Exception
    {
 ClassLoader cl = getClass().getClassLoader();

 URL url = cl.getResource("config/settings.txt");
 if (url == null)
     throw new IllegalArgumentException("No such settings.txt.");

 InputStream is = url.openStream();
 print(is, "Printing settings:\n");

 is = cl.getResourceAsStream("config/properties.xml");
```

```
 if (is == null)
     throw new IllegalArgumentException("No such properties.xml.");

 print(is, "\nPrinting properties:\n");
    }
}
```

このサービスは 2 つの設定ファイルの内容を出力します。これは暗号化されたリソースの復号化がクラスローディングレイヤの背後に隠されることを示します。

これを適切にテストするには、ポリシーモジュールを自分で暗号化するか、既存の暗号化されたものを使用します。これを実行するには、EncryptedService を **ClassLoaderSystem** とデプロイヤに適切に関連付ける必要があります。

ClassLoaderSystem のパーティション化については、この章の後半で説明します。

9.2. CLASSLOADING

ClassLoader 抽象化を直接使用する代わりに、**ClassLoader** 依存関係の宣言を含む **ClassLoading** モジュールを作成できます。依存関係が指定されたら、 **ClassLoaderPolicy**が構築され、適切に一緒に接続されます。

ClassLoaders が実際に存在する前に **ClassLoaders** を定義することを支援するために、抽象化には **ClassLoadingMetaData** モデルが含まれます。

ClassLoadingMetaData は新しい JBoss EAP プロファイルサービス内で Managed Object として公開できます。これにより、システム管理者は実装の詳細よりも抽象的なポリシーの詳細を扱うことができるようになります。

例9.4 Managed Object としての ClassLoadingMetaData Exposed

```
public class ClassLoadingMetaData extends NameAndVersionSupport {
    /** The serialVersionUID */
    private static final long serialVersionUID = -2782951093046585620L;

    /** The classloading domain */
    private String domain;

    /** The parent domain */
    private String parentDomain;

    /** Whether to make a subdeployment classloader a top-level
classloader */
    private boolean topLevelClassLoader = false;

    /** Whether to enforce j2se classloading compliance */
    private boolean j2seClassLoadingCompliance = true;

    /** Whether we are cacheable */
```

```
    private boolean cacheable = true;

    /** Whether we are blacklistable */
    private boolean blackListable = true;

    /** Whether to export all */
    private ExportAll exportAll;

    /** Whether to import all */
    private boolean importAll;

    /** The included packages */
    private String includedPackages;

    /** The excluded packages */
    private String excludedPackages;

    /** The excluded for export */
    private String excludedExportPackages;

    /** The included packages */
    private ClassFilter included;

    /** The excluded packages */
    private ClassFilter excluded;

    /** The excluded for export */
    private ClassFilter excludedExport;

    /** The requirements */
    private RequirementsMetaData requirements = new
RequirementsMetaData();

    /** The capabilities */
    private CapabilitiesMetaData capabilities = new
CapabilitiesMetaData();

    ... setters & getters
```

例9.5「XML で定義されたクラスローディング API」 と 例9.6「Java で定義されたクラスローディング API」 は、それぞれ XML と Java で定義されたクラスローディング API を示しています。

例9.5 XML で定義されたクラスローディング API

```
<classloading xmlns="urn:jboss:classloading:1.0"
       name="ptd-jsf-1.0.war"
       domain="ptd-jsf-1.0.war"
       parent-domain="ptd-ear-1.0.ear"
       export-all="NON_EMPTY"
```

```
        import-all="true"
        parent-first="true"/>
```

例9.6 Java で定義されたクラスローディング API

```
ClassLoadingMetaData clmd = new ClassLoadingMetaData();
if (name != null)
    clmd.setDomain(name + "_Domain");
clmd.setParentDomain(parentDomain);
clmd.setImportAll(true);
clmd.setExportAll(ExportAll.NON_EMPTY);
clmd.setVersion(Version.DEFAULT_VERSION);
```

展開への ClassLoadingMetaData の追加は、プログラムを使用するか、**`jboss-classloading.xml`** で宣言することにより行えます。

例9.7 `jboss-classloading.xml` を使用した ClassLoadingMetaData の追加

```
<classloading xmlns="urn:jboss:classloading:1.0"
       domain="DefaultDomain"
       top-level-classloader="true"
       export-all="NON_EMPTY"
       import-all="true">
</classloading>
```

DefaultDomain は、独自のドメインを定義しないすべてのアプリケーション間で共有されます。

例9.8 一般的なドメインレベルの分離

```
<classloading xmlns="urn:jboss:classloading:1.0"
       domain="IsolatedDomain"
       export-all="NON_EMPTY"
       import-all="true">
</classloading>
```

例9.9 特定の親による分離

```
<classloading xmlns="urn:jboss:classloading:1.0"
```

```
        domain="IsolatedWithParentDomain"
        parent-domain="DefaultDomain"
        export-all="NON_EMPTY"
        import-all="true">
</classloading>
```

例9.10 j2seClassLoadingCompliance に準拠しない

```
<classloading xmlns="urn:jboss:classloading:1.0"
        parent-first="false">
</classloading>
```

`.war` 展開はデフォルトでこのメソッドを使用します。デフォルトの parent-first ルックアップを行う代わりに最初に独自のリソースをチェックします。

例9.11 一般的な OSGi 実装

```
<classloading xmlns="urn:jboss:classloading:1.0">
  <requirements>
    <package name="org.jboss.dependency.spi"/>
  </requirements>
  <capabilities>
    <package name="org.jboss.cache.api"/>
    <package name="org.jboss.kernel.spi"/>
  </capabilities>
</classloading>
```

例9.12 粒度が細かいパッケージではなくモジュールとライブラリ全体のインポートとエクスポート

```
<classloading xmlns="urn:jboss:classloading:1.0">
  <requirements>
    <module name="jboss-reflect.jar"/>
  </requirements>
  <capabilities>
    <module name="jboss-cache.jar"/>
  </capabilities>
</classloading>
```

```
<classloading xmlns="urn:jboss:classloading:1.0">
  <requirements>
    <package name="si.acme.foobar"/>
    <module name="jboss-reflect.jar"/>
  </requirements>
  <capabilities>
    <package name="org.alesj.cl"/>
    <module name="jboss-cache.jar"/>
  </capabilities>
</classloading>
```

また、パッケージとモジュールを使用して要件と機能のタイプを組み合わせることもできます。

クラスローディングサブプロジェクトは非常に小さい resource-visitor-pattern 実装を使用します。

ClassLoader プロジェクトでは、展開とクラスローディング間の接続が、フィルタリングなどのビジターパターンに制限を適切に適用するために必要なすべての情報を保持する **Module** クラスを使用して行われます。

例9.13 ResourceVisitor インターフェースと ResourceContext インターフェース

```
public interface ResourceVisitor {
    ResourceFilter getFilter();

    void visit(ResourceContext resource);
}

public interface ResourceContext {
    URL getUrl();

    ClassLoader getClassLoader();

    String getResourceName();

    String getClassName();

    boolean isClass();

    Class<?> loadClass();

    InputStream getInputStream() throws IOException;

    byte[] getBytes() throws IOException;
}
```

モジュールを使用するには、ResourceVisitor インスタンスをインスタンス化し、**Module::visit** メソッドに渡します。この機能は、展開のアノテーション使用をインデックス化するために展開フレームワークで使用されます。

9.3. クラスローディング VFS

これらの例では、JBoss 仮想ファイルシステムプロジェクトを使用してクラスとリソースをロードする **ClassLoaderPolicy** の実装を提供します。このアイデアを直接使用するか、クラスローディングフレームワークと組み合わせて使用できます。

オプションで、Microcontainer 設定内部でモジュールを設定できます。

例9.14 クラスローディングモジュールデプロイヤ

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <classloader name="anys-classloader"
xmlns="urn:jboss:classloader:1.0" import-all="true" domain="Anys"
parent-domain="DefaultDomain">
    <capabilities>
      <package
name="org.jboss.test.deployers.vfs.reflect.support.web"/>
    </capabilities>
    <root>${jboss.tests.url}</root>
  </classloader>

  <bean name="AnyServlet"
class="org.jboss.test.deployers.vfs.reflect.support.web.AnyServlet">
    <classloader><inject bean="anys-classloader:0.0.0"/></classloader>
  </bean>

</deployment>
```

VFSClassLoaderFactory クラスは XML デプロイヤを **VFSClassLoaderPolicyModule** に変換し、実際の **ClassLoader** インスタンスを作成します。次に、この新しい **ClassLoader** インスタンスを Bean で直接使用できます。

注記

VFSClassLoaderFactory は **ClassLoadingMetaData** を拡張し、この場合は **ClassLoadingMetaData** に関するすべての例が適用されます。

第10章 仮想展開フレームワーク

新しい*仮想展開フレームワーク (VDF)* を使用すると、Microcontainer でデプロイヤをより簡単に管理できます。この章では、役に立ついくつかの機能について詳述します。

10.1. 展開タイプのアグノスティック処理

仮想展開の従来のタイプは、共有されたクラス領域またはドメインにすでに存在するクラスに基づきます。この場合、エンドプロダクトは主なクライアントからサーバーにインストールされた新しいサービスです。これを行う従来の方法は、記述子ファイルをアップロードすることです。新しい VDF を使用すると、バイトを渡し、**`Deployment`** インスタンスにシリアル化することによりこのプロセスが単純化されます。

最初のものを拡張する展開の他のタイプはプレーンなファイルシステムベースの展開であり、Microcontainer VFS によって支援されます。この方法の詳細については、8章*仮想ファイルシステム* を参照してください。

10.2. 展開ライフサイクルロジックからの構造認識の分離

展開上で実際の作業を実行するには、最初に、クラスパスとメタデータの場所を含む構造を理解する必要があります。

メタデータの場所には、**`my-jboss-beans.xml`**、**`web.xml`**、**`ejb-jar.xml`** などの設定ファイルが含まれます。クラスパスは **`WEB-INF/classes`** や **`myapp.ear/lib`** などのクラスローダーのルートです。

構造を考慮して、実際の展開処理に進むことができます。

一般的な展開ライフサイクル

1. **`MainDeployer`** は認識のために展開を **`StructuralDeployer`** のセットに渡し、Deployment コンテキストを受け取ります。

2. 次に、**`MainDeployer`** は、適切な **`Deployer`** により処理するために結果となる Deployment コンテキストを **`Deployers`** に渡します。

MainDeployer は使用するデプロイヤを決定する責任を持つブローカーです。

仮想展開またはプログラムを使用した展開の場合、事前に決定された既存の StructureMetaData 情報は構造情報を読み取り、StructuredMetaData 情報の処理 で説明されたいずれかの方法で処理します。

StructuredMetaData 情報の処理

VFS ベースの展開

構造認識は StructureDeployers のセットに転送されます。

JEE-specification-defined 構造

一致する StructureDeployer 実装は以下のとおりです。

- EarStructure
- WarStructure

- JarStructure

DeclarativeStructures

展開内部の **META-INF/jboss-structure.xml** ファイルを検索し、解析して適切な **StructureMetaData** を構築します。

FileStructures

-jboss-beans.xml や **-service.xml** などの既知の設定ファイルのみを認識します。

例10.1 jboss-structure.xml の例

```
<structure>
  <context
comparator="org.jboss.test.deployment.test.SomeDeploymentComparatorTo
p">
    <path name=""/>
    <metaDataPath>
      <path name="META-INF"/>
    </metaDataPath>
    <classpath>
      <path name="lib" suffixes=".jar"/>
    </classpath>
  </context>
</structure>
```

EarStructure の場合、最初に最上位の展開を認識し、次にサブ展開を再帰的に処理します。

汎用的な **StructureDeployer** インターフェースにより提供された汎用的な **GroupingStructure** クラスを使用してカスタム **StructureDeployer** を実装できます。

認識された展開構造を用意したら、それを実際のデプロイヤに渡すことができます。デプロイヤオブジェクトは、**DeploymentStage** ごとにデプロイヤチェーンを使用して実際のデプロイヤを処理する方法を認識します。

例10.2 展開ステージ

```
public interface DeploymentStages {
   /** The not installed stage - nothing is done here */
   DeploymentStage NOT_INSTALLED = new DeploymentStage("Not Installed");

   /** The pre parse stage - where pre parsing stuff can be prepared;
altDD, ignore, ... */
   DeploymentStage PRE_PARSE = new DeploymentStage("PreParse",
NOT_INSTALLED);

   /** The parse stage - where metadata is read */
   DeploymentStage PARSE = new DeploymentStage("Parse", PRE_PARSE);

   /** The post parse stage - where metadata can be fixed up */
   DeploymentStage POST_PARSE = new DeploymentStage("PostParse",
PARSE);
```

```
   /** The pre describe stage - where default dependencies metadata can
be created */
   DeploymentStage PRE_DESCRIBE = new DeploymentStage("PreDescribe",
POST_PARSE);

   /** The describe stage - where dependencies are established */
   DeploymentStage DESCRIBE = new DeploymentStage("Describe",
PRE_DESCRIBE);

   /** The classloader stage - where classloaders are created */
   DeploymentStage CLASSLOADER = new DeploymentStage("ClassLoader",
DESCRIBE);

   /** The post classloader stage - e.g. aop */
   DeploymentStage POST_CLASSLOADER = new
DeploymentStage("PostClassLoader", CLASSLOADER);

   /** The pre real stage - where before real deployments are done */
   DeploymentStage PRE_REAL = new DeploymentStage("PreReal",
POST_CLASSLOADER);

   /** The real stage - where real deployment processing is done */
   DeploymentStage REAL = new DeploymentStage("Real", PRE_REAL);

   /** The installed stage - could be used to provide valve in future?
*/
   DeploymentStage INSTALLED = new DeploymentStage("Installed", REAL);
}
```

既存の展開ステージは Microcontainer の組込みコントローラ状態にマッピングされ、汎用コントローラ状態の展開ライフサイクルを中心としたビューを提供します。

デプロイヤ内部では、展開は Microcontainer のコンポーネント **`DeploymentControllerContext`** に変換されます。Microcontainer の状態マシンは依存関係を処理します。

展開は展開ステージで順番に処理されます。各デプロイヤに対して、デプロイヤの **`parent-first`** プロパティを使用して、展開された全体の階層順序が処理されます。このプロパティはデフォルトで **`true`** に設定されます。

また、デプロイヤが処理する階層レベルを指定することもできます。**`all`**、**`top level`**、**`components only`**、または **`no components`** を選択できます。

ここでは、Microcontainer によるコンポーネントモデルの処理と依存関係処理を行う方法が適用されます。未解決の依存関係がある場合、展開は現在の状態で待機し、現在の状態が必要な状態でない場合にエラーが報告されることがあります。

新しいデプロイヤの追加は、既存の多くのヘルパーデプロイヤのいずれかを拡張することにより実現されます。

一部のデプロイヤは実際に VFS 支援展開が必要であり、他のデプロイヤは一般的な展開を使用できます。ほとんどの場合、解析デプロイヤは VFS の支援が必要です。

警告

また、デプロイヤは各展開、サブ展開、およびコンポーネントで再帰的に実行されることにも注意してください。プロセスのできるだけ早い段階で、デプロイヤが現在の展開を処理する必要があるかどうかをコードで決定する必要があります。

例10.3 展開に関する情報を出力する単純なデプロイヤ

```
public class StdioDeployer extends AbstractDeployer {
   public void deploy(DeploymentUnit unit) throws DeploymentException
   {
      System.out.println("Deploying unit: " + unit);
   }

   @Override
   public void undeploy(DeploymentUnit unit)
   {
      System.out.println("Undeploying unit: " + unit);
   }
}
```

この記述を JBoss Application Server の **deployers/** ディレクトリの **-jboss-beans.xml** ファイルのいずれかに追加します。この結果、**MainDeployerImpl** Bean が Microcontainer の IoC コールバック処理によりこのデプロイヤを取得します。

例10.4 単純な展開記述子

```
<bean name="StdioDeployer" class="org.jboss.acme.StdioDeployer"/>
```

10.3. 添付という形式での自然なフロー制御

VDF には*添付*と呼ばれるメカニズムがあり、あるデプロイヤから次のデプロイヤに情報を渡すことを支援します。添付は少し拡張された **java.util.Map** として実装され、その各エントリは添付を表します。

いくつかのデプロイヤはプロデューサーであり、他のデプロイヤはコンシューマです。同じデプロイヤが両方の機能を実行することもできます。いくつかのデプロイヤはメタデータまたはユーティリティインスタンスを作成し、これらを *添付*マップに格納します。他のデプロイヤは、このデータに追加の処理を行う前に、これらの添付の必要性を宣言し、添付マップからデータを取得します。

*自然順序*とはデプロイヤの順序が決定される方法を意味します。一般的な自然順序は *before* と *after* の相対的な用語を使用します。ただし、添付メカニズムがすでに使用されている場合は、添付が生成または消費される方法でデプロイヤの順序を決定できます。

各添付はキーを持ち、デプロイヤは生成される添付にキーを渡します。各添付はキーを持ち、デプロイヤは生成された添付にキーを渡します。デプロイヤが添付を生成する場合、キーは*出力*と呼ばれます。デプロイヤが添付を消費する場合、キーは*入力*と呼ばれます。

デプロイヤには*通常*の入力と*必須*の入力があります。通常の入力は自然な順序を決定するためにのみ使用されます。必須の入力は順序決定するのも支援しますが、別の機能も持ちます。必須の入力を使用すると、その必須の入力に対応する添付が添付マップに存在するかどうかを確認することにより、デプロイヤが該当する展開に対して実際に重要であるかどうかを決定できます。

警告

相対的な順序付けは引き続きサポートされますが、推奨されない方法と見なされ、将来のリリースでサポートされなくなる可能性があります。

10.4. クライアント、ユーザー、およびサーバーの使用と実装の詳細

これらの機能は実装の詳細を隠し、実装の使用でエラーを少なくし、同時に開発プロセスを効率化します。

この目的は開発者が DeploymentUnit を見る一方でクライアントが開発 API のみを見るようにすることです。サーバー実装の詳細は DeploymentContext に含まれます。必要な情報のみが特定のレベルの展開のライフサイクルに公開されます。

コンポーネントはすでにデプロイヤの階層処理の一部として説明されました。最上位の展開とサブ展開は展開の構造階層の自然な表現である一方で、コンポーネントは新しい VDF コンセプトです。コンポーネントの考えは、Microcontainer 内部の **`ControllerContexts`** との 1:1 マッピングを持つことです。この理由については、コンポーネントが **`ControllerContexts`** と 1:1 でマップする理由 を参照してください。

コンポーネントが `ControllerContexts` と 1:1 でマップする理由

ネーミング

コンポーネントユニットの名前は、**`ControllerContext`** の名前と同じです。

get*Scope() and get*MetaData()

このインスタンスに対して Microcontainer により使用される同じ MDR コンテキストを返します。

IncompleteDeploymentException (IDE)

IDE が展開で不明な依存関係を出力するためには、ControllerContext 名を知る必要があります。この名前は、**`BeanMetaDataDeployer`** や **`AbstractRealDeployer`** の **`setUseUnitName()`** メソッドなどの、指定するコンポーネントデプロイヤで Component DeploymentUnit の名前を収集することにより検出されます。

10.5. 単一の状態マシン

すべての Microcontainer コンポーネントは、単一のエントリポイントまたは単一の状態マシンによって処理されます。展開も例外ではありません。

この機能は、展開で **jboss-dependency.xml** 設定ファイルを使用することにより利用できます。

例10.5 jboss-dependency.xml

```
<dependency xmlns="urn:jboss:dependency:1.0">
  <item whenRequired="Real"
dependentState="Create">TransactionManager</item> (1)
  <item>my-human-readable-deployment-alias</item> (2)
</dependency>
```

XML の人為的なコールアウト (1) と (2) に注意してください。

(1) は他のサービスの依存関係の定義方法を示しています。この例では、展開が 'Real' ステージになる前に **TransactionManager** を作成する必要があります。

(2) は、追加情報が不明なためもう少し複雑です。デフォルトでは、Microcontainer 内部の展開名は URI 名であり、手入力によりエラーが発生しやすくなります。したがって、他の展開で依存関係を簡単に宣言するために、URI 名を回避するエイリアスメカニズムが必要です。**aliases.txt** という名前のプレーンテキストファイルを展開に追加できます。このファイルの各行にはエイリアスが含まれ、展開アーカイブに参照に使用される 1 つまたは複数の単純な名前を提供します。

10.6. クラスでのアノテーションのスキャン

現在の JEE 仕様では、設定ファイルの数が減少しますが、コンテナが @annotations を使用してほとんどの作業を行うことが必要になりました。@annotation 情報を取得するには、コンテナがクラスをスキャンする必要があります。このスキャンにより、パフォーマンスペナルティが作成されます。

ただし、スキャンの量を減らすため、Microcontainer は **jboss-scanning.xml** により別の記述子フックを提供します。

例10.6 jboss-scanning.xml

```
<scanning xmlns="urn:jboss:scanning:1.0">
  <path name="myejbs.jar">
    <include name="com.acme.foo"/>
    <exclude name="com.acme.foo.bar"/>
  </path>
  <path name="my.war/WEB-INF/classes">
    <include name="com.acme.foo"/>
  </path>
</scanning>
```

この例は、Java Enterprise Edition バージョン 5 以上のアノテートされたメタデータ情報をスキャンするときに含める、または除外する相対パスの単純な記述を示しています。

付録A 改訂履歴

改訂 5.1.2-2.400 **2013-10-31** **Rüdiger Landmann**

Rebuild with publican 4.0.0

改訂 5.1.2-2 **2012-07-18** **Anthony Towns**

Rebuild for Publican 3.0

改訂 5.1.2-100 **Thu 8 December 2011** **Russell Dickenson**

JBoss Enterprise Application Platform 5.1.2 GA 向けの変更が含まれます。このガイドの内容の変更については『 リリースノート 5.1.2』を参照してください。