

序文

Jalapeño チュートリアルへようこそ。

Jalapeño(Java Language Persistence with No Mapping)は、Caché 用の Java 永続性ライブラリです。このライブラリを使用することで、オブジェクト・リレーショナル・マッピングを使用せずに POJO (Plain Ordinary Java Object、プレーンな従来型 Java オブジェクト)を Caché に格納できる単純で効果的なメカニズムを Java アプリケーションに実装できます。Caché は、継承、リレーションシップ、コレクション、複雑なデータ型などのオブジェクトの特性を維持しながら、POJO を実際のオブジェクトとして格納します。さらに、Caché は自動的にオブジェクトを SQL 標準リレーショナル・テーブルに投影します。Jalapeño を利用することで、Java アプリケーションでは、従来のリレーショナル JDBC の手法だけでなくオブジェクト指向の手法で格納された POJO にアクセスできます。標準的な SQL ベースのレポート・ツールで、格納された POJO データにアクセスすることもできます。

このチュートリアルは、2 つに分かれています。次の順序で始めてください。

"第 I 章: Jalapeñoクイック・スタート"では、POJOクラス定義からのオブジェクト・データベース・スキーマの生成、Cachéへの POJOインスタンスの保存およびCachéからのPOJOインスタンスの取得に関する基本手順を説明します。

"第 II 章: その他の例"では、最初にJalapeñoとそのコンポーネントの詳細について説明します。次にJalapeñoアノテーションを使用して、JalapeñoランタイムAPI経由でPOJOデータにアクセスする高度なオブジェクト・データベース・スキーマやメソッドの作成例を詳しく説明します。

Note:

このチュートリアルを読むには、Javaプログラミングに関する基本的な知識が必要です。Javaプログラミングの詳細は、Sun MicrosystemsのWebサイトに掲載されている"[The Java Tutorials](#)"を参照してください。このチュートリアルでは、Caché プログラミングに関する知識は必要ありません。

はじめに

チュートリアル第 I 章のクイック・スタートでは、Jalapeño の概要を説明します。この章では、以下の内容について学習します。

- Jalapeño を使用するための要件
- Jalapeño SchemaBuilder ウィザードを使用して Java クラスからデータベース・スキーマを生成する方法
- Jalapeño **ObjectManager** クラスを使用して Java オブジェクトを Caché に保存する方法
- Jalapeño **ObjectManager** クラスを使用して Java オブジェクトを Caché から取得する方法

要件とインストール

Jalapeño を使用するには、以下のプラットフォームが必要です。

1. Caché 5.2 以降
2. J2SE 1.5 以降

Jalapeño は、Caché データベースの標準インストールの一部としてインストールされます。

基本的な手順

Jalapeño を使用して、POJO を Caché に保存し、次に POJO を Caché から取得する基本的な手順は以下のようになります。

1. POJO クラスを作成し、コンパイルします。クラス定義にアノテーションを記述すれば、より高度なデータベース・スキーマを作成できます。
2. SchemaBuilder ウィザードを使用して、POJO クラスを基にデータベース・スキーマを生成します。
3. **ObjectManager** クラスのメソッドを使用して、POJO の永続性を管理します。**ObjectManager** クラスのメソッドによって、POJO をデータベースに保存したり、データベースから取得したり、更新したりできます。

POJO を作成します

最初の手順は POJO クラス定義を作成することです。簡単な例を以下に示します。

```
package basic;
public class Contact {
    public String name;
    public String contactType;
}
```

このクラスはインタフェースを実装せず、他のクラスを拡張しません。これは純粋な POJO です。["第 II 章:その他の例"](#) では、生成するデータベース・スキーマへ、制約、インデックス、リレーションシップなどのオブジェクトの特性を加えるために、クラス定義にアノテーションを追加する方法を示します。

以下の操作を行なう前に、この Contact.java をコンパイルし、Contact.class を生成しておく必要があります。

スキーマの作成

2 番目の手順では、SchemaBuilder ウィザードを使用して、POJO 定義を基にデータベース・スキーマを生成します。手順は以下のようになります。

1. *CacheDB.jar* が含まれているディレクトリでコマンド・ウィンドウを開きます。このファイルは `< cachesys > ¥Dev¥java¥lib¥JDK15¥CacheDB.jar` にあります。
CacheDB.jar がクラス・パスにある場合は、以下のコマンドを使用して SchemaBuilder Wizard を起動します。

```
>java com.jalapeno.tools.SchemaBuilderWizard
```

CacheDB.jar がクラス・パスにない場合は、`-cp` オプションを使用して含めます。

```
>java -cp < cachesys > ¥Dev¥java¥lib¥JDK15¥CacheDB.jar  
com.jalapeno.tools.SchemaBuilderWizard
```

2. Cache 接続情報およびスキーマのネームスペースを入力します。

SchemaBuilder

File

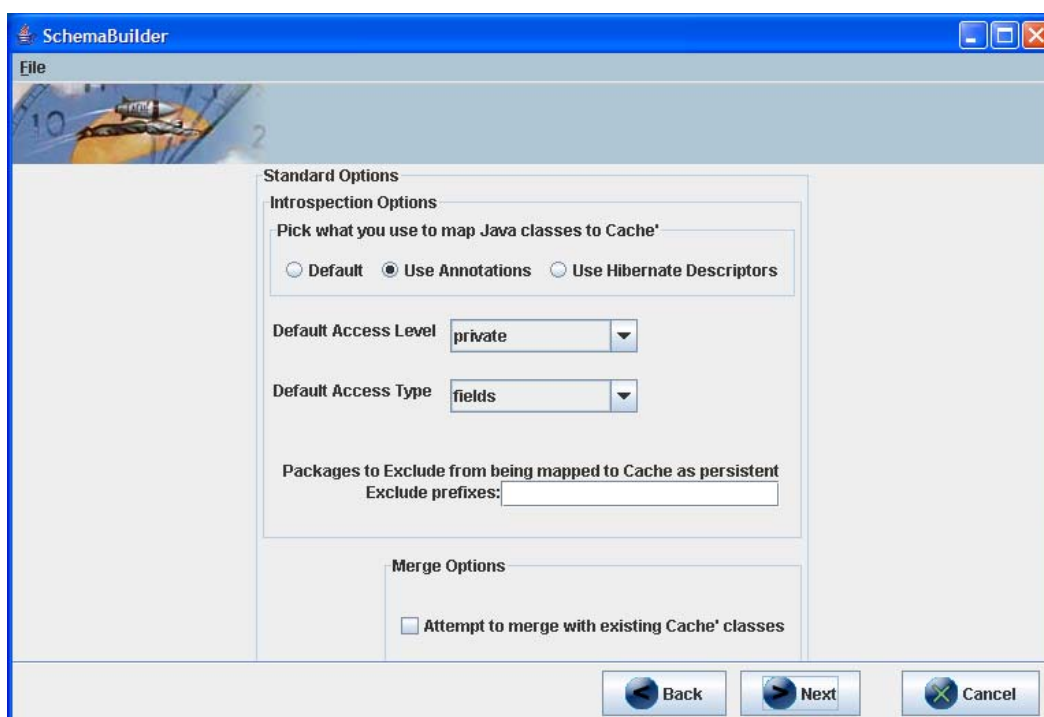
Please enter Connection details for the Cache Database and click "Show Namespaces" button.
Be sure to select the correct Namespace before clicking Next.

Server: localhost
Port: 1972
User: _SYSTEM
Password: ****

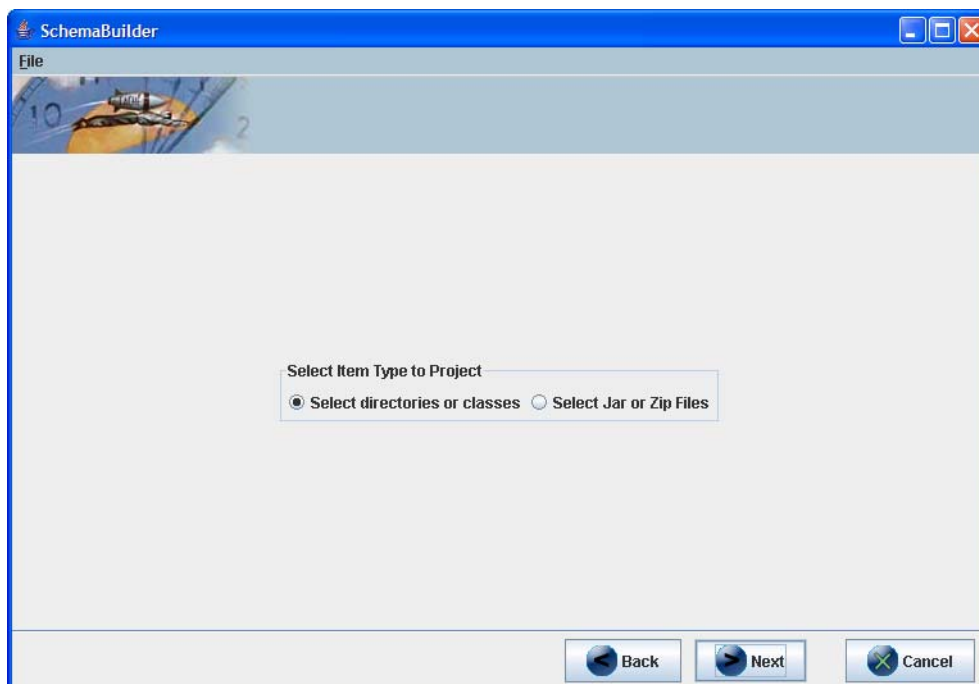
Show Namespaces: USER

Back Next Cancel

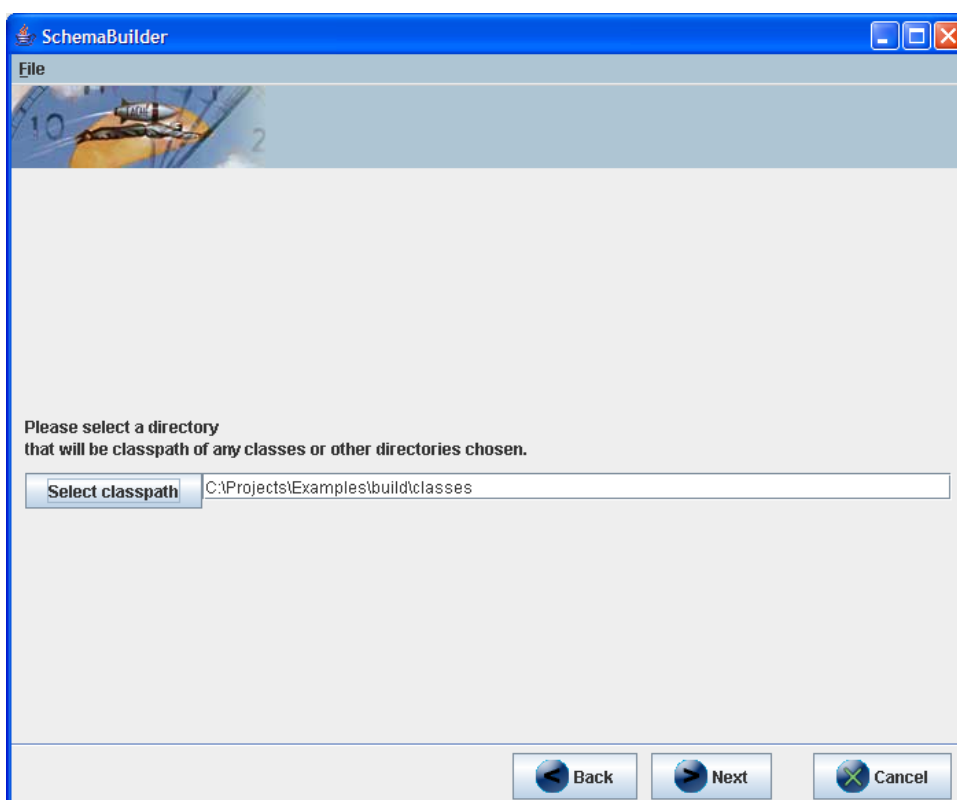
3. **[Next]**をクリックします。次の画面で、データベース・スキーマの生成方法を指定する情報を入力します。ここでは、以下の情報を入力します。
 1. **[Introspection Option]**で**[Use Annotations]** をクリックします。
 2. **[Default Access Level]**で **private** を選択します。
 3. **[Default Access Type]**で **fields** を選択します。
 4. **[Exclude Prefixes]**フィールドは空欄のままにします。
 5. **[Attempt to merge with existing Cache' Classes]**にチェックが付いている場合は、チェックを外します。



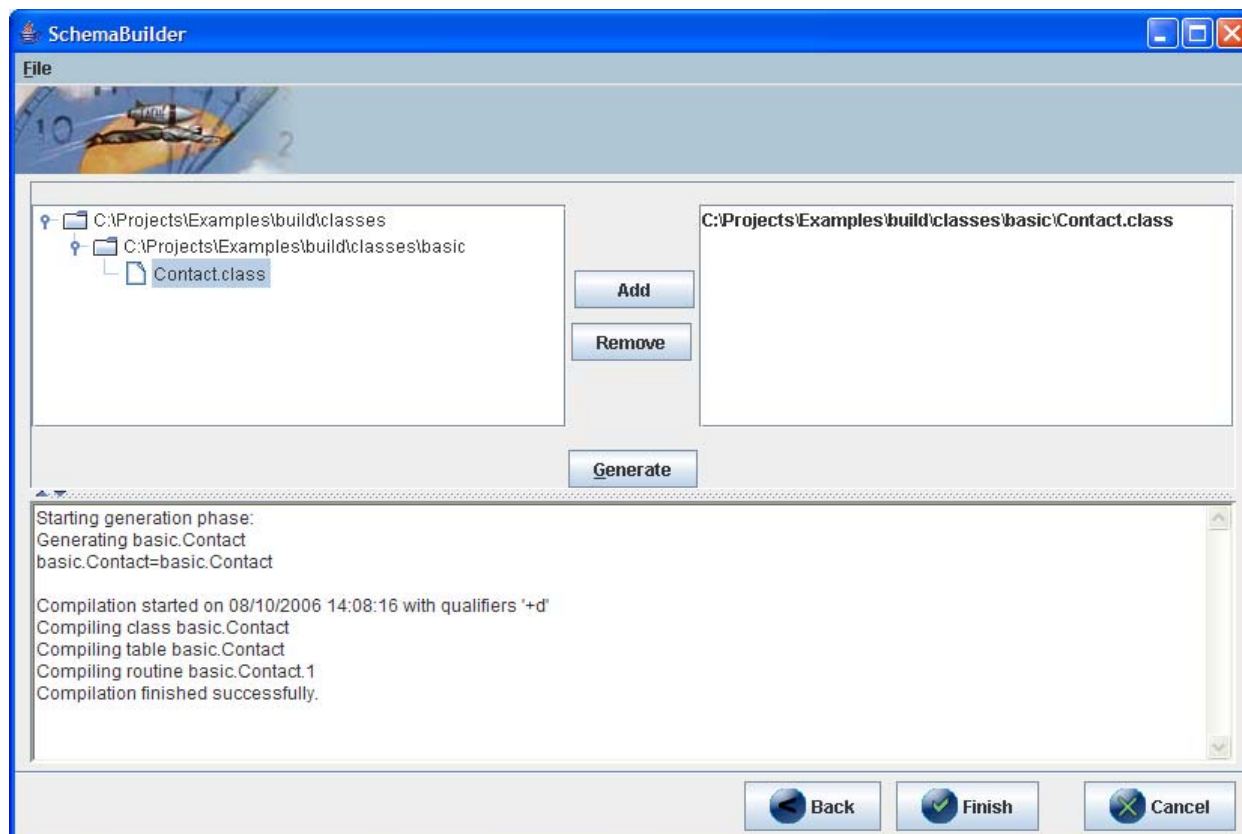
4. **[Next]**をクリックします。次の画面の**[Select Item Type to Project]**で**[Select directories or classes]**をクリックします。



5. 次の画面で、POJO クラス・ファイルのクラス・パスを入力します。以下の画像には、このクラス・パス `<project-home>#build#classes` が表示されています。クラスは **basic** パッケージ内にあるので、フィールド自体は `<project-home>#build#classes#basic` にあります。



6. **[Next]**をクリックします。以下の操作を行います。
 1. 左上のパネルで、*basic*ノードをクリックして、*Contact.class*ファイルを表示します。 *Contact.class*ファイルをクリックします。
 2. **[Add]**ボタンをクリックして、このファイルを右側のパネルに追加します。
 3. **[Generate]**をクリックします。下側のパネルに表示される内容を確認します。



Caché に、**Contact** のデータベース・スキーマが追加されます。

スキーマの表示

Caché スタジオを使用して、Caché クラス定義としてオブジェクトを表示することができます。手順は以下のようになります。

1. Caché キューブをクリックし、メニューの[スタジオ]をクリックします。
2. Caché スタジオのメイン・メニュー・バーで、[ファイル]→[ネームスペース変更]をクリックします。[接続マネージャ]で、スキーマを含むネームスペースをクリックします。このネームスペースは、SchemaBuilder ウィザードの最初の画面で選択したものです。
3. ワークスペース・ナビゲータで、[basic]フォルダをクリックし、[Contact]をクリックします。

クラス定義は、以下のように表示されます。

```

Class basic.Contact Extends %Library.Persistent [ SqlTableName = Contact ]
{
    Property contactType As %Library.String(JAVATYPE = "java.lang.String", MAXLEN = 4096);
    Property name As %Library.String(JAVATYPE = "java.lang.String", MAXLEN = 4096);
    %XData JavaBlock
    {
        %<JavaBlock>
        <Package implementation="basic.cache" pojo="basic"></Package>
        <UseSameNames>false</UseSameNames>
        <Name implementation="Contact" pojo="Contact"></Name>
        <ResolveNameCollisions>false</ResolveNameCollisions>
        <EagerFetchRequired>true</EagerFetchRequired>
        </JavaBlock>
    }
}

```

Caché システム管理ポータルを使用して、Contact のリレーショナル・データベース・スキーマを表示することもできます。手順は以下のようになります。

1. Caché キューブをクリックし、メニューの[システム管理ポータル] をクリックします。
2. [データ管理]列で、[SQL]をクリックします。
3. [ネームスペース]列(左端)で、スキーマを含むネームスペースをクリックします。このネームスペースは、SchemaBuilder ウィザードの最初の画面で選択したものです。
4. [SQL 操作]列で、[SQL スキーマを参照]をクリックします。
5. basic スキーマのリンクをクリックします。
6. [テーブルを開く]リンクをクリックします。

このスキーマは以下のように表示されます。

更新 ウィンドウを閉じる			
データベース USER 中の basic.Contact			最終更新: 2007-05-17 15:58:58.153
#	ID	contactType	name
			完了

永続性マネージャの作成

com.jalapeno.ObjectManager クラスには、POJO の永続性を管理するためのメソッドがあります。**basic.Contact** の永続性を管理するには、新しい Java 永続性マネージャ・クラスとして **basic.DBService** を作成します。これには、以下の操作を行います。

1. ObjectManager のインスタンスを作成し、java.sql.Connection オブジェクトで初期化します。
2. Contact インスタンスを Caché に保存するための、ObjectManager save メソッドを使用する saveContact メソッドを含めます。
3. Caché 内のすべての Contact インスタンスを開くための、ObjectManager openByQuery メソッドを使用する allContacts メソッドを含めます。

以下の **import** 文を **DBService** クラスに記述します。

```
import com.jalapeno.ApplicationContext;
import com.jalapeno.ObjectManager;
import java.sql.Connection;
import java.sql.DriverManager;
import java.util.Iterator;
```

DBService コンストラクタで **ObjectManager** インスタンスを作成します。このコンストラクタでは、**com.jalapeno.ApplicationContext** クラスで提供されているファクトリ・メソッドの **createObjectManager** を使用します。**host**、**username**、**password** および **url** には、正しい Caché システムとネームスペースの値を指定する必要があります。また、**objectManager** 変数には、クラスのスコープがあることに注意してください。つまり、すべてのメソッドの外部で宣言されています。

```
public DBService ()
    throws Exception
{
    String      host = "localhost";
    String      username="_SYSTEM"; // null for default
    String      password="SYS"; // null for default

    String      url="jdbc:Cache://" + host + ":1972/USER";
```

```

Class.forName ("com.intersys.jdbc.CacheDriver");
Connection connection =
    DriverManager.getConnection (url, username, password);
objectManager = ApplicationContext.createObjectManager (connection);
}

```

DBService saveContact メソッドは、引数に **Contact** インスタンスを1つ取ります。このメソッドは、**ObjectManager save** メソッドを使用して、Cache に保存します。

```

protected void saveContact(Contact contact )
throws Exception
{
    objectManager.save(contact, true);
}

```

DBService allContacts メソッドは、データベースからすべての **Contact** インスタンスを取得します。このメソッドは、開いている一連の POJO に **Iterator** を返します。このメソッドは、**ObjectManager openByQuery** メソッドを使用してオブジェクトを取得します。

```

protected Iterator allContacts ()
throws Exception
{
    return objectManager.openByQuery (Contact.class, null, null);
}

```

Note:

DBService クラスをコンパイルして実行するには、クラス・パスに CacheDB.jar が含まれている必要があります。

"[第 11 章 : その他の例](#)" では、**save** および **openByQuery** メソッドとその引数について詳しく説明します。

※DBService クラスの完全なコードは、以下のとおりです。

```
package basic;
import com.jalapeno.ApplicationContext;
import com.jalapeno.ObjectManager;
import java.sql.Connection;
import java.sql.DriverManager;
import java.util.Iterator;
public class DBService
{
    ObjectManager objectManager;
public DBService ()
    throws Exception
    {
        String      host = "localhost";
        String      username="_SYSTEM"; // null for default
        String      password="SYS"; // null for default
        String      url="jdbc:Cache://" + host + ":1972/USER";
        Class.forName ("com.intersys.jdbc.CacheDriver");
        Connection connection =
            DriverManager.getConnection (url, username, password);
        objectManager = ApplicationContext.createObjectManager (connection);
    }
protected void saveContact(Contact contact )
    throws Exception
    {
        objectManager.save(contact, true);
    }

protected Iterator allContacts ()
throws Exception
    {
        return objectManager.openByQuery (Contact.class, null, null);
    }
}
```

テスト・クラスの実成

Jalapeño の動作を確認するために、テスト・クラスを作成します。以下の **Test** というクラスは、次の操作を行う **main** メソッドを含みます。

1. **DBService** のインスタンスを作成します。
2. 2 つの **Contact** インスタンスを作成します。
3. **DBService** を使用して、**Contact** インスタンスを Cache に保存します。
4. **DBService** を使用して、**Contact** インスタンスを Cache から取得します。

Test の完全なコードは以下のとおりです。

```
package basic;
import java.util.Iterator;
public class Test {
    private static DBService service;
    public static void main (String[] args) throws Exception
    {
        service = new DBService();

        //Create and save two contacts
        Contact contact = new Contact();
        contact.name = "Smith,John";
        contact.contactType = "Business";
        service.saveContact(contact);
        contact = new Contact();
        contact.name="Smith,Jane";
        contact.contactType="Business";
        service.saveContact(contact);

        //retrieve and display contacts
        for (Iterator iter = service.allContacts(); iter.hasNext();)
        {
            Contact con = (Contact)iter.next();
            System.out.printf("Name: %s Type: %s¥n", con.name, con.contactType);
        }
    }
}
```

データの表示

Testの実行後、Cachéには2つの**Contact**インスタンスが追加されます。システム管理ポータルを使用してこのデータを表示することができます。これを行うには、[\[スキーマの表示\]](#)ページに記載されている手順に従います。

このデータは以下のように表示されます。

#	ID	contactType	name
1	1	Business	Smith,John
2	2	Business	Smith,Jane

完了

更新 | ウィンドウを閉じる

データベース USER 中の basic.Contact

最終更新: 2007-05-17 16:38:21.044

要約

このチュートリアルの第 I 章では、以下について学習しました。

- Jalapeño を使用するためのシステム要件
- SchemaBuilder ウィザードを使用した、POJO からの Caché データベーススキーマの生成
- **ObjectManager** クラスのインスタンスの作成
- **ObjectManager** メソッドを使用した、POJO の Caché への保存
- **ObjectManager** メソッドを使用した、Caché からの POJO の取得

次の作業

ここでは、Jalapeño の動作の最も簡単な例を示します。Jalapeño には、さらに多くの機能があります。Jalapeño は、特に以下の機能を提供しています。

- インデックス、リレーションシップ、埋め込みオブジェクトおよび制約を含む、より高度なデータベース・スキーマを生成することができる一連のアノテーション
- スキーマ生成を詳細に制御できる SchemaBuilder ウィザードのオプションとアノテーション
- POJO の保存と取得を柔軟に操作できる **ObjectManager** メソッド

これらのトピックは、すべて"[第 II 章 : その他の例](#)"で説明しています。

はじめに

このチュートリアルの第 II 章では、Jalapeño の機能について詳しく説明し、第 I 章で紹介した Jalapeño の動作についてより高度な例を示します。この章の学習を終えると、以下のことを実行できるようになります。

1. SchemaBuilder ウィザードのアクセス・タイプおよびアクセス・レベル・オプションを使用したデータベース・スキーマ生成の制御
2. アノテーションを使用した、データベース・スキーマへのインデックス、制約、リレーションシップおよび埋め込みオブジェクトの追加
3. ObjectManager save、insert および update メソッドを使用した、データベースへのデータの保存
4. ObjectManager の openByQuery および openByPrimaryKey メソッドを使用した、SQL 条件、クエリ、主キーなどによる Caché からのオブジェクトの取得

Jalapeño の概要

Jalapeño は、Plain Ordinary Java Object (プレーンな従来型 Java オブジェクト、POJO)を Caché に保存するためのメカニズムを提供します。

Jalapeño と Caché を使用する主な利点は以下のとおりです。

- *マッピングを使用せずに簡単に POJO の永続性を実現*

Jalapeño は、Java イントロスペクションを使用して POJO クラス定義を分析し、対応するオブジェクト・データベース・スキーマを生成します。このスキーマには、継承、コレクション、複雑なデータ型、リレーションシップなどのオブジェクトの特性を組み込むことができます。スキーマには、インデックスや制約などのデータベースの特性を組み込むこともできます。これらの特性はすべて、標準的な Java アノテーションを使用してデータベース・スキーマに追加できます。XML マッピング・ドキュメントは必要ありません。

- *オブジェクト・アクセス*

Jalapeño は、Caché に POJO を実際のオブジェクトとして格納し、プロパティ、リレーションシップ、継承などのオブジェクトの特性を維持します。Jalapeño ランタイム API Java アプリケーションを使用すると、POJO をデータベースに保存したり、データベースから POJO を取得したり、データベースに対してクエリを実行したりすることができます。

- *標準 SQL アクセス*

Caché は自動的にすべてのオブジェクトを SQL 標準リレーショナル・テーブルに投影します。Java アプリケーションでは、Jalapeño を使用する他に、従来の JDBC を通じて POJO データにアクセスできます。標準的な SQL レポート・ツールでデータにアクセスすることもできます。

- **データベースの独立性**

Jalapeño を通して、SQL の標準のデータ定義言語 (Data Definition Language、DDL) を使用してオブジェクト・データベース・スキーマを対応するリレーショナル・データベースにエクスポートできます。Jalapeño が、標準リレーショナル・データベースと情報を交換するときは自動的に標準 JDBC を使用します。

Jalapeño が、Caché オブジェクト・データベースと情報を交換するときは、高性能なオブジェクト・ベースのプロトコルを使用します。

- **プラットフォームの独立性**

Jalapeño ライブラリは、Java 1.5 以降の JVM または J2EE アプリケーション・サーバ環境で動作します。Caché オブジェクト・データベース・サーバは、多くのプラットフォームで使用可能です。これらのプラットフォームのリストは、Caché ドキュメント: [\[はじめに\]-\[サポート対象プラットフォーム\]](#) を参照してください。

Jalapeño コンポーネントとリソース

Jalapeño には、3 つの主要コンポーネントがあります。これらのコンポーネントはすべて CacheDB.jar に含まれています。Cache の標準インストールを行った場合、CacheDB.jar ファイルは < cachesys > ¥ Dev ¥ java ¥ lib ¥ JDK15 に保存されます。

コンポーネント	説明
SchemaBuilder ウィザード	POJO クラス定義を基に Cache データベース・スキーマを生成するツール
ObjectManager クラス	POJO 永続性を管理するためのメソッドを提供するクラス
com.jalapeño.annotations	アノテーションを定義するクラスのパッケージ。これらのアノテーションを POJO 内で使用することで、Cache データベース・スキーマを作成することができます。

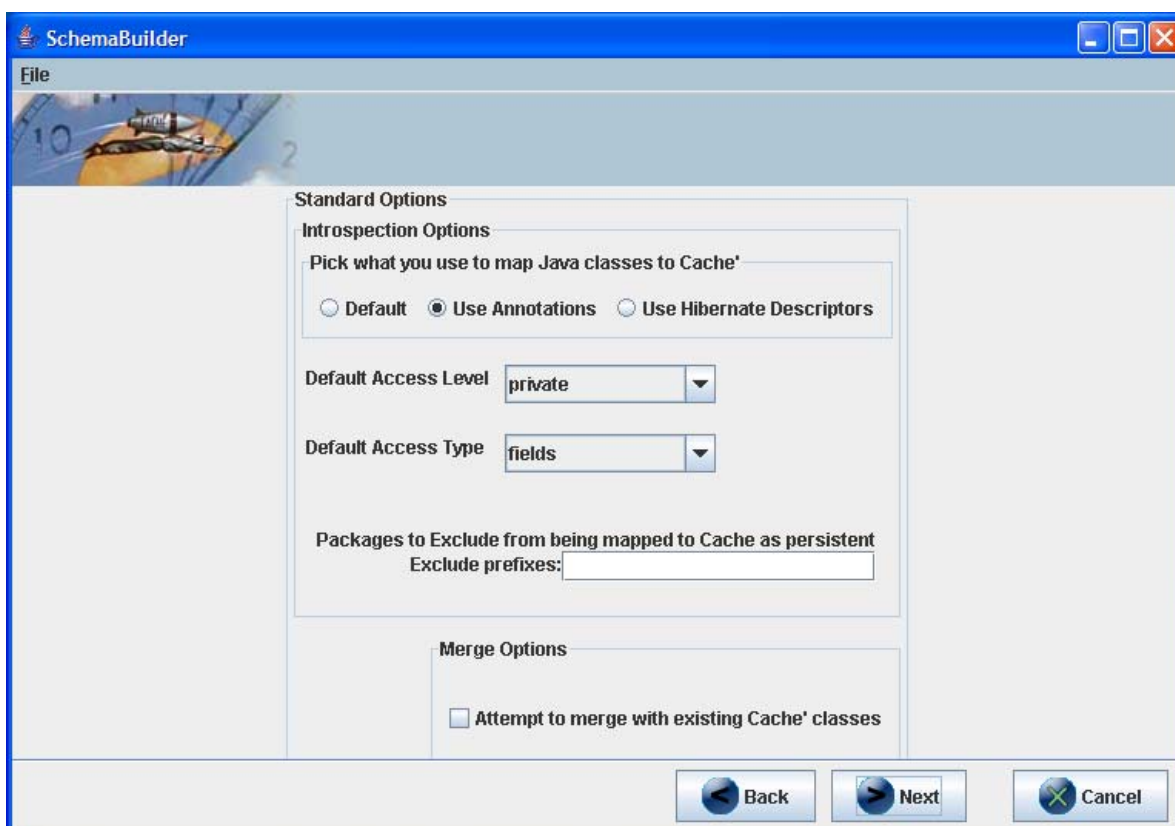
さらに、Cache には Jalapeño を使用するためのリソースも用意されています。

1. ドキュメント — Cacheドキュメント: [\[Cache言語バインディング\]](#)-[\[Cache JalapeñoでのJavaの使用法\]](#)を参照してください。
2. サンプル・プログラム — "< cachesys > ¥ Dev ¥ java ¥ samples" を参照してください。

SchemaBuilder ウィザード

SchemaBuilderウィザードは、POJOクラス定義を基にCacheオブジェクト・データベース・スキーマを生成するためのグラフィカル・ツールです。このウィザードはJava イントロスペクションを使用して、クラス定義を検査し、適切なスキーマを生成します。"[スキーマの作成](#)"では、ウィザードを起動して、POJOのスキーマを生成するための基本的な手順について説明しています。

このウィザードの 2 番目の画面では、[Default Access Level]および[Default Access Type]を選択できます。



以下のテーブルは、これらの 2 つのオプションで指定できる値の説明です。

SchemaBuilder Wizard のオプション

オプション	説明
Access Level	スキーマに含める POJO の要素の最も厳しいアクセス・レベルを指定します。アクセス修飾子が同じレベルかそれ以下の POJO 要素がスキーマに含まれます。例えば、アクセス・レベルが <code>private</code> である場合は、指定されたアクセス・タイプのすべての要素がデータベース・スキーマに組み込まれます。アクセス・レベルが <code>public</code> である場合は、 <code>public</code> の POJO 要素のみが組み込まれ、 <code>protected</code> メンバまたは <code>private</code> メンバの要素は組み込まれません。
Access Type	スキーマを生成するために使用する POJO 要素の種類を指定します。主なオプションには、“getters and setters”および“fields”などがあります。“getters and setters”を選択した場合、ウィザードは POJO の“get”および“set”メソッドに基づいてデータベース・スキーマにフィールドを生成します。スキーマ・フィールドの名前には、“get”または“set”の後のすべての文字が使用されます。“field”を選択した場合、ウィザードは POJO フィールドに基づいてスキーマを生成します。スキーマ・フィールドの名前は、POJO フィールドの名前と同じになります。

上記のオプションは既定値です。これらのオプションは、POJO 定義に含まれているアノテーションによってオーバーライドすることができます。

ObjectManager

Jalapeño は、Caché での POJO の永続性を管理するための **ObjectManager** インタフェースを提供しています。 **com.jalapeno.ApplicationContext** クラスの **createObjectManager** メソッドを使用してこのインタフェースのインスタンスを作成することができます。このメソッドに、正しい接続情報で初期化された **java.sql.Connection** オブジェクトを渡します。

ObjectManager が提供するメソッドは次のとおりです。

一部のオブジェクト・マネージャのメソッド

メソッド名	説明
insert	<p>POJO に対応するデータベースにオブジェクトを挿入します。以下の引数を取ります。</p> <ul style="list-style-type: none"> Object pojo — 保存する POJO です。 boolean deep — 真の場合、ディープ・インサートを実行します。つまり、POJO のオブジェクト参照も挿入します。 <p>オブジェクト ID を返します。</p>
save	<p>指定された POJO に対応するデータベース内のオブジェクトを更新するか、データベース・オブジェクトが存在しない場合は、新規データベース・オブジェクトを作成します。以下の引数を取ります。</p> <ul style="list-style-type: none"> Object pojo — 保存する POJO です。 boolean deep — 真の場合、ディープ・インサートを実行します。つまり、POJO のオブジェクト参照も挿入します。 <p>オブジェクト ID を返します。</p>
update	<p>指定された POJO に対応するデータベース内のオブジェクトを更新します。以下の引数を取ります。</p> <ul style="list-style-type: none"> Object pojo — 保存する POJO です。 boolean deep — 真の場合、ディープ・アップデートを実行します。つまり、POJO のオブジェクト参照も更新します。 <p>オブジェクト ID を返します。</p>

一部のオブジェクト・マネージャのメソッド (追加)

メソッド名	説明
removeFromDatabase	<p>指定された POJO に対応するデータベース内のオブジェクトを削除します。以下の引数を取ります。</p> <ul style="list-style-type: none"> Object pojo — 削除するデータベース・オブジェクトに対応する POJO です。
openByPrimaryKey	<p>指定された主キーを使用してデータベース・オブジェクトを検索し、メモリ内でインスタンス化します。以下の引数を取ります。</p> <ul style="list-style-type: none"> Class clazz — POJO のクラスです。 Object primaryKey — getPrimaryKey によって返された主キーまたはオブジェクト値の文字列値です。 <p>インスタンス化されたオブジェクトへの参照を返します。</p>
openByQuery	<p>指定された SQL 条件を満たすクラスのデータベース・オブジェクトをメモリ内でインスタンス化します。以下の引数を取ります。</p> <ul style="list-style-type: none"> Class clazz — 開くオブジェクトが属するクラス(またはそのサブクラス) String sql — 開くオブジェクトを特定する SQL 条件。使用可能な条件の規則に関するドキュメントを参照してください。以下にリンクを示します。 Object [] params — SQL 条件のパラメータ値 <p>すべての開いているオブジェクトに対して反復される java.util.Iterator インスタンスを返します。</p>

Note:

上記すべてのメソッドは、[java.lang.Exception](#) 型の例外を返します。

ObjectManagerの全メソッドのリストと各メソッドの詳細は、Cacheドキュメント: [\[Cache言語バインディング\]](#)-[\[Cache Jalapeño でのJavaの使用法\]](#)-[\[Jalapeño ランタイム・ライブラリ・リファレンス\]](#)を参照してください。< cachesys > ¥Dev¥java¥docにあるJavaDocスタイル・クラスのドキュメントも参照してください。

save: 更新または挿入

ObjectManager `save` メソッドは、状況に応じて既存のデータベース・オブジェクトを更新するか、新規データベース・オブジェクトを挿入します。どちらの操作を実行するかを決定する規則は次のとおりです。

`save` メソッドは、以下のいずれかの条件が成立している場合は、既存のオブジェクトを更新します。

このメソッドは以下の順序で条件を評価します。

1. POJO が、既存のデータベース・インスタンスにアタッチされている。
2. POJO のクラスが内部データベース ID のプレースホルダを含み、この ID に対する POJO の値が `null` ではない。
* 内部データベース ID については以下のページで説明します。
3. POJO のクラスが内部データベース ID のプレースホルダを含んでおらず、POJO が既存のデータベース・インスタンスにアタッチされていないが、同じ主キー値を持つインスタンスがデータベース内に存在する。

Annotations — クラス・レベル

Jalapeño は多数のアノテーションを提供しており、それらを POJO クラス定義に追加することで、POJO に対して生成されたオブジェクト・データベース・スキーマ (Cache クラス定義) に追加の構造を加えることができます。これらのアノテーションは、Cache クラスを生成するときに SchemaBuilder ウィザードによって読み取られます。以下のテーブルは、いくつかのクラス・レベル・アノテーションとその要素の説明です。

クラス・レベル・アノテーション

アノテーション	説明
@Access	<p>アクセス・レベルとタイプによって、生成された Cache クラスのマッピングを制限します。</p> <ul style="list-style-type: none"> level — スキーマを生成するために使用される POJO メンバのアクセス・レベルを指定します。可能な値は、<code>AccessLevel.UNDEFINED</code> (既定値、"SchemaBuilder ウィザード"で指定された値を使用)、<code>AccessLevel.PUBLIC</code>、<code>AccessLevel.PROTECTED</code>、<code>AccessLevel.PRIVATE</code> です。 type — スキーマを生成するために使用される POJO メンバの種類、フィールドまたはプロパティを指定します。可能な値は、<code>AccessType.PROPERTY</code> (既定値、アクセサ・メソッドがマップされる)と <code>AccessType.FIELD</code> (フィールドがマップされる)です。
@CacheClass	<p>スキーマのクラス・レベル情報を提供します。以下の要素があります。</p> <ul style="list-style-type: none"> primaryKey — 対応するスキーマの主キーを指定します。 sqlTableName — POJO に対する Cache の SQL プロジェクトのテーブル名を指定します。 populatable — 生成された Cache クラスに Cache データ入力メカニズムを使用できるようにするかどうかを指定します。

	<ul style="list-style-type: none"> • <code>xmlSerializable</code> — 生成された Caché クラスを XML にシリアル化できるようにするかどうかを指定します。
@Index	<p>生成されたデータベース・スキーマ内にインデックスを指定します。以下の要素があります。</p> <ul style="list-style-type: none"> • <code>name</code> — インデックスの名前 • <code>propertyNames</code> — インデックスの定義に使用するスキーマ定義内のプロパティの名前 • <code>isPrimaryKey</code> — 真の値はインデックスが主キーも兼ねることを示します。 <code>false</code> の値(既定値) は、インデックスが主キーでないことを示します。
@Embeddable	<p>このマーク・アノテーションは、生成されたスキーマがシリアルか、埋め込み可能かを示します。</p>
@ClassParameter	<p>生成された Caché クラス(オブジェクト・スキーマ)のクラス・レベル・パラメータを指定します。以下の 2 つの要素があります。</p> <ul style="list-style-type: none"> • <code>name</code> — Caché クラス・パラメータの名前 • <code>value</code> — Caché クラス・パラメータの値

Note:

クラス・レベル・アノテーションの完全なリストについては、Cachéドキュメント: [\[Caché言語バインディング\]-\[Caché Jalapeñoでの Javaの使用法\]-\[アノテーションの使用法\]](#)の章を参照してください。 < cachesys > ¥Dev¥java¥docにあるJavaDocスタイル・クラスのドキュメントも参照してください。

Annotations — フィールド・レベル

Jalapeño にはフィールド・レベル・アノテーションも含まれています。以下のテーブルに、一部のアノテーションを紹介します。

フィールド・レベル・アノテーション

アノテーション	説明
@ID	<p>POJO プロパティがデータベース ID を表すことを指定します。要素には以下が含まれます。</p> <ul style="list-style-type: none"> type — 指定可能な値は SYSTEM_ASSIGNED および USER_ASSIGNED
@CacheProperty	<p>対応する Cache プロパティ名、タイプおよび SQL 名を指定します。以下の要素があります。</p> <ul style="list-style-type: none"> name — Cache プロパティ名 type — Cache プロパティ・タイプ sqlColumnName — Cache SQL 列名
@Collection	<p>フェッチ方法と、対応する Cache コレクションのタイプおよび要素タイプを指定します。以下の要素があります。</p> <ul style="list-style-type: none"> type — Cache コレクションのタイプ。指定可能な値は、ARRAY および LIST です。 elementType — 対応する Cache コレクションに含まれる要素タイプ fetch — コレクション要素をメモリにロードする方法。指定可能な値は、LAZY および EAGER です。
@Relationship	<p>対応する Cache クラスにリレーションシップを定義します。以下の要素があります。</p> <ul style="list-style-type: none"> type — 対応する Cache クラスから見たリレーションシップのタイプ。指定可能な値は、ONE_TO_MANY および MANY_TO_ONE です。 inverseClass — リレーションシップの逆クラスを表す Cache クラスの名前 inverseProperty — 逆リレーションシップを表す Cache プロパティの名前

	<ul style="list-style-type: none">• parentChild — Caché リレーションシップが親子の関係であるかどうかを指定します。指定可能な値は、TRUE および FALSE です。• fetch — リレーションシップの多数側の要素をメモリにロードする方法。指定可能な値は、LAZY および EAGER です。
--	---

例

第 II 章の後半では、Jalapeño の動作の例題について説明します。この例題では、Jalapeño アノテーションを使用して以下の操作を行う方法を示します。

- SchemaBuilder ウィザードの AccessType および AccessLevel の制御
- Caché オブジェクト・スキーマでのインデックスの定義
- Caché オブジェクト・スキーマでの一対多リレーションシップの作成
- Caché オブジェクト・スキーマのプロパティ名の制御
- Caché オブジェクト・スキーマでのプロパティ・パラメータの設定
- Caché オブジェクト・スキーマでのシリアル(埋め込み可能)クラスの実装
- POJO の Caché データベース ID へのプレースホルダの定義

さらに、この例題では以下の **ObjectManager** メソッドの使用方法を示します。

- **save** — データベース・オブジェクトの挿入と更新の両方を行います。
- **openByQuery** — SQL 条件と、完全な SQL のパラメータ化されたクエリの両方を使用します。

Note:

完成した例題のコードは、InterSystems Developer's Cornerの Web サイトからダウンロードすることができます。<http://vista.intersystems.com/samples/PojoTutorial.zip> にアクセスしてダウンロードしてください。

例題のコードは、Caché の標準インストールにも含まれています。そのコードは、<cache>¥Dev¥tutorials¥pojo にあります。

例題の POJO の概要

この例題は 3 つの POJO クラスを使用します。また、**Contact** は以下のように定義します。3 つのクラスでは、すべて getter および setter メソッドでフィールドがカプセル化されています。基本的なクラス定義は以下のようになります(getter と setter は省略されています)。

Contact

```
package contacts;

import java.util.ArrayList;
import java.util.List;

public class Contact {
    private Address primaryAddress;
    private String name;
    private String type;
    private List <PhoneNumber> phoneNumbers;

    public Contact() {
        setPhoneNumbers(new ArrayList <PhoneNumber> ());
    }

    //getters and setters
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    ...
}
```

汎用コレクションを使用して phoneNumbers プロパティを定義し、そのコレクションの要素タイプとして **PhoneNumber** を指定しています。コレクションの要素タイプを指定しない限り、コレクションに対してスキーマを生成することはできません。

PhoneNumber

```
package contacts;

public class PhoneNumber {
    private String type;
    private Contact owner;
    private String number;

    public PhoneNumber() {
    }

    //getters and setters
    public String getType() {
        return type;
    }
    public void setType(String type) {
        this.type = type;
    }

    ...

}
```

Address

```
package contacts;

public class Address {
    private String mCity;
    private String mState;
    private String mStreet;
    private String mZip;

    public Address() {
    }

    //getters and setters
    public String getCity() {
        return mCity;
    }
    public void setCity(String value) {
        mCity = value;
    }

    ...
}
```

※これら 3 つのクラスに対して、SchemaBuilder ウィザードを使用して、Cache オブジェクト・スキーマを生成します。

生成は、各段階毎に実行しても、テスト実施前にまとめて実行しても、どちらでも構いません。

@Access の使用法

3つの POJO クラスはそれぞれ **@Access** アノテーションを使用して、オブジェクト・スキーマの生成を制御します。

Contact と **PhoneNumber** では、以下の同じ **@Access** アノテーションを使用します。

- type=AccessType.FIELD
- level=AccessLevel.PRIVATE

この指定により、SchemaBuilder ウィザードは、Caché オブジェクト・スキーマに **Contact** および **PhoneNumber** の各フィールドに対応するプロパティを生成します。クラス定義の関連する部分は以下のようになります。

Contact

```
package contacts;
import com.jalapeno.annotations.*;

@Access(type = AccessType.FIELD, level = AccessLevel.PRIVATE)
public class Contact {
...
}
```

PhoneNumber

```
package contacts;
import com.jalapeno.annotations.*;
import java.util.ArrayList;
import java.util.List;

@Access(type = AccessType.FIELD, level = AccessLevel.PRIVATE)
public class PhoneNumber {
...
}
```

Address の場合は異なります。 **Address** では、**@Access** アノテーションに以下の要素値を使用します。

- `type=AccessType.PROPERTY`

`AccessType.PROPERTY` 値の指定により、SchemaBuilder ウィザードは、オブジェクト・スキーマのプロパティを生成するために、フィールドではなくPOJOの `get` メソッドおよび `set` メソッドを使用します。例えば、**Address** は `setCity` メソッドを含みます。このメソッドから対応する Cache スキーマに City プロパティが生成されます。アノテーションは、`type` プロパティを定義しません。したがって、SchemaBuilder ウィザードによって指定される値が使用されます。ただし、すべてのアクセサ・メソッドが `public` なので、SchemaBuilder ウィザードは、ウィザードで指定される `level` 値に関係なく、各メソッドのオブジェクト・スキーマでフィールドを生成します。関連するコードは以下のとおりです。

Address

```
package contacts;
import com.jalapeno.annotations.*;

@Access (type = AccessType.PROPERTY)
public class Address{
...
}
```

@Index の使用

Contact および **PhoneNumber** の両方は、**@Index** アノテーションを使用して、オブジェクト・スキーマにインデックスを定義します。関連するコードは以下のとおりです。

Contact

```
package contacts;

import com.jalapeno.annotations.*;
import java.util.ArrayList;
import java.util.List;

@Access(type = AccessType.FIELD, level = AccessLevel.PRIVATE)
@Index(name = "NameIndex", propertyNames = { "name" },
      isUnique = false, isPrimaryKey = false)
public class Contact {
...
}
```

Caché オブジェクト・スキーマの対応するインデックスの仕様は以下のとおりです。

- インデックス名は NameIndex です。
- インデックスは、Caché スキーマの *name* プロパティに定義します。
- インデックスは一意でなく、主キーでもありません。

Contact の Caché クラス定義に生成されるインデックス定義は以下のとおりです。

```
Index NameIndex On name;
```

PhoneNumber

```
package contacts;

import com.jalapeno.annotations.*;
@Access(type = AccessType.FIELD, level = AccessLevel.PRIVATE)
@Index(name = "NumberIndex", propertyNames = { "number" }, isPrimaryKey =
true)
public class PhoneNumber {
...
}
```

Caché オブジェクト・スキーマの対応するインデックスの仕様は以下のとおりです。

- インデックス名は NumberIndex です。
- インデックスは、Caché スキーマの number プロパティに定義します。
- このインデックスは主キーです。

PhoneNumber の Caché クラス定義に生成されるインデックス定義は以下のとおりです。

```
Index NumberIndex On number [ PrimaryKey, Unique ];
```

@Embeddable の使用

Address クラスは **@Embeddable** アノテーションを使用して、生成する Caché クラスが Serial (または Embeddable) として定義されるように指定します。関連するコードは以下のとおりです。

```
package contacts;

import com.jalapeno.annotations.*;

@Access (type = AccessType.PROPERTY)
@Embeddable
public class Address{
```

対応する Caché クラス定義の関連する部分は以下のとおりです。

```
Class contacts.Address Extends %Library.SerialObject [ ClassType = serial ]
```

Note:

Cachéシリアル(または埋め込み可能)オブジェクトの詳細は、Cachéドキュメント: [\[Caché開発ガイド\]-\[Cachéオブジェクトの使用法\]-\[Cachéオブジェクト・モデル\]-\[クラスのタイプ\]](#)を参照してください。

@Relationship の使用

Contact および **PhoneNumber** POJO クラス定義は、**@Relationship** アノテーションを使用して、対応する Caché スキーマとの間に一対多のリレーションシップを作成します。 **Contact** スキーマがリレーションシップの単一側を形成します、**PhoneNumber** スキーマがリレーションシップの多数側を形成します。 リレーションシップは、**PhoneNumber** の phoneNumbers プロパティと **PhoneNumber** の owner プロパティを使用して定義します。

Contact

```
...
@Relationship(type=RelationshipType.ONE_TO_MANY,inverseClass=
                "contacts.PhoneNumber",inverseProperty="owner")
public List<PhoneNumber> phoneNumbers;
...
```

このアノテーションでは以下の項目を指定します。

- **Contact** から見たリレーションシップ・タイプは一対多です。
- リレーションシップをサポートする **Contact** のプロパティは phoneNumbers です。
- リレーションシップの相手側を形成する逆クラスは **PhoneNumber** です。
- **PhoneNumber** のリレーションシップをサポートするプロパティは owner です。

Caché **Contact** クラスのリレーションシップ定義は、以下のとおりです。

```
Relationship phoneNumbers As contacts.PhoneNumber(JAVATYPE = "java.util.List")
                [ Cardinality = many, Inverse = owner ];
```

PhoneNumber

```
...
@Relationship(type=RelationshipType.MANY_TO_ONE,
                inverseClass="contacts.Contact", inverseProperty="phoneNumbers")
private Contact owner;
...
```

このアノテーションでは以下の項目を指定します。

- **PhoneNumber** から見たリレーションシップ・タイプは一対多です。
- リレーションシップをサポートする **PhoneNumber** のプロパティは owner です。
- リレーションシップの相手側を形成する逆クラスは **Contact** です。
- **Contact** のリレーションシップをサポートするプロパティは phoneNumbers です。

Caché **PhoneNumber** クラスのリレーションシップ定義は、以下のとおりです。

```
Relationship owner As contacts.Contact(JAVATYPE = "contacts.Contact")
    [ Cardinality = one, Inverse = phoneNumbers ];
```

システム管理ポータルに表示される **PhoneNumbers** テーブルは、以下のとおりです。このテーブルには[Owner]列があり、**Contact** ID 値が含まれています。

更新 ウィンドウを閉じる	
ネームスペース USER 中の contacts.PhoneNumber	最終更新: 2007-05-17 18:22:06.820
#	ID PhoneNumberType number owner
完了	

システム管理ポータルに表示される **Contact** テーブルは、以下のとおりです。このテーブルには **PhoneNumber** の情報は含まれていません。リレーションシップに関する情報は、1 か所にのみ格納する必要があります。

更新 ウィンドウを閉じる	
ネームスペース USER 中の contacts.Contact	最終更新: 2007-05-17 18:25:23.911
#	ID ContactType name primaryAddress_City primaryAddress_State primaryAddress_Street primaryAddress_Zip
完了	

Note:

システム管理ポータルを使用して Cachéテーブルを表示する方法は、このチュートリアルの第1章の"スキーマの表示"ページを参照してください。

@CacheProperty の使用

Contact および **PhoneNumber** の POJO クラス定義は、**@CacheProperty** アノテーションを使用して、Cache スキーマの対応するフィールドの名前とタイプを設定します。

Contact

```
...
@CacheProperty (name="ContactType", type="%Library.String")
private String type;
...
```

このアノテーションでは以下の項目を指定します。

- 対応する Cache プロパティの名前は ContactType です。
- 対応する Cache プロパティのタイプは [%Library.String](#) です。

PhoneNumber

```
...
@CacheProperty(name="PhoneNumberType", type="%Library.String")
private String type;
...
```

@PropertyParameter および @PropertyParameters の使用

Contact および **PhoneNumber** の POJO クラス定義では、**@PropertyParameter** アノテーションを使用して、VALUELIST および DISPLAYLIST を定義します。各 POJO 定義には、**@PropertyParameter** アノテーションの 2 つのインスタンスが必要です。各 POJO 定義は、**@PropertyParameters** を使用して複数の **@PropertyParameter** インスタンスを宣言します。

Contact

```
...
    @PropertyParameters({
        @PropertyParameter(name="VALUELIST", value="Business,Personal"),
        @PropertyParameter(name="DISPLAYLIST", value="Bus,Pers")
    })

    @CacheProperty (name="ContactType", type="%Library.String")
    private String type;
...

```

@PropertyParameters および **@PropertyParameter** 定義は以下を指定します。

- Caché ContactType プロパティに指定可能な値は、“Business” および “Personal” です。これ以外の type の値を使用して **Contact** POJO を Caché に保存しようとする、データベース例外が発生します。
- このプロパティに対する Caché LogicalToDisplay メソッドによって返される値は、“Bus” および “Pers” です。

対応する Caché プロパティ定義は以下のとおりです。

```
Property ContactType As %Library.String(DISPLAYLIST = "Bus,Pers",
    JAVATYPE = "java.lang.String", MAXLEN = 4096,
    VALUELIST = "Business,Personal") [ ClientName = type ];

```

PhoneNumber

```
@PropertyParameters({
    @PropertyParameter(name="VALUELIST", value=",Home,Office,Mobile"),
    @PropertyParameter(name="DISPLAYLIST", value=",H,O,M")
})
@CacheProperty(name="PhoneNumberType", type="%Library.String")
private String type;
```

@PropertyParameters および **@PropertyParameter** 定義は以下を指定します。

- Cache PhoneNumberType プロパティに指定可能な値は、“Home”、“Office”および“Mobile”です。これ以外の type の値を使用して **PhoneNumber** POJO を Cache に保存しようとする、データベース例外が発生します。
- このプロパティに対して Cache LogicalToDisplay メソッドによって返される値は、“H”、“O”および“M” です。

対応する Cache プロパティ定義は以下のとおりです。

```
Property PhoneNumberType As %Library.String(DISPLAYLIST = ",H,O,M",
    JAVATYPE = "java.lang.String", MAXLEN = 4096,
    VALUELIST = ",Home,Office,Mobile") [ ClientName = type ];
```

Note:

Cache LogicalToDisplayメソッドの詳細は、Cacheドキュメント: [\[Cache開発ガイド\]](#)-[\[Cacheオブジェクトの使用法\]](#)-[\[データ型\]](#)-[\[データ形式と変換メソッド\]](#)の説明を参照してください。

@ID の使用

Contact クラスでは **@ID** アノテーションを使用して、idPlaceholder フィールドに、システムによって割り当てられた ID を格納するように指定します。

Contact

```
@ID(type = IDType.SYSTEM_ASSIGNED)
protected String idPlaceholder;
```

このアノテーションにより、idPlaceholder は Cache オブジェクト ID のプレースホルダになります。

ObjectManager **save** メソッドは、このプロパティを使用して既存のデータベース・インスタンスを更新するか、新規データベース・インスタンスを挿入するかを決定します。POJO にプレースホルダがあり、その値が **NULL** でない場合、**save** は既存のデータベース・インスタンスを更新します。POJO にプレースホルダがあり、その値が **NULL** の場合、**save** は新規データベース・インスタンスを挿入します。

ObjectManager の使用法

例題では、**DBService** クラスに永続性コードを実装します。以下の **ObjectManager** メソッドを使用して永続性コードを実装します。

- **save** — 新規データベース・インスタンスを挿入し、既存のデータベース・インスタンスを更新するために使用します。
- **openByQuery** — 既存のデータベース・インスタンスに対応する一連の POJO をメモリ内でインスタンス化するために使用します。

以下のコードは **DBService** コンストラクタを示しています。コンストラクタは、このクラス全体で使用する **ObjectManager** インスタンスを作成します。

```
package contacts;
import com.jalapeno.ApplicationContext;
import com.jalapeno.ObjectManager;
import java.sql.Connection;
import java.sql.DriverManager;
import java.util.Iterator;

public class DBService {
    ObjectManager objectManager;

    public DBService ()
        throws Exception
    {
        String          host = "localhost";
        String          username="_SYSTEM";
        String          password="SYS";
        String          url="jdbc:Cache://" + host + ":1972/USER";

        Class.forName ("com.intersys.jdbc.CacheDriver");
        Connection connection =
            DriverManager.getConnection (url, username, password);
        objectManager = ApplicationContext.createObjectManager (connection);
    }
    ...
}
```

このコンストラクタは **ApplicationContext** **createObjectManager** メソッドを使用して **ObjectManager** インスタンスを作成します。そしてメソッドに、適切に初期化された **java.sql.Connection** オブジェクトを渡します。

save の使用法

DBService クラスは、**Contact** インスタンスを挿入して更新する **saveContact** メソッドを実装します。

完成したメソッドは以下のとおりです。

```
protected void saveContact(Contact contact )
    throws Exception
{
    objectManager.save(contact, true);
}
```

このメソッドについて、以下のことに注意してください。

- このメソッドはディープ・セーブを実行します。**Contact** インスタンスによって参照されるすべてのオブジェクトが保存されます。
- **Contact** POJO 定義は Cache オブジェクト ID のプレースホルダを定義するので、**ObjectManager save** メソッドは状況に応じて、挿入または更新を実行します。

openByQuery の使用法

DBService クラスは 3 つのメソッドで **openByQuery** を使用します。

- **allContacts** — データベース内の各 **Contact** インスタンスに対して、メモリ内で POJO インスタンスを開きます。
- **contactsByName** — 指定された name 値を持つデータベース内の各 **Contact** インスタンスに対して、メモリ内で POJO インスタンスを開きます。
- **phonesByName** — 指定された name の **Contact** インスタンスと関連するデータベース内の各 **PhoneNumber** インスタンスに対して、メモリ内で POJO インスタンスを開きます。

allContacts のコードは以下のとおりです。

```
protected Iterator allContacts ()
    throws Exception
{
    return objectManager.openByQuery (Contact.class, null, null);
}
```

contactsByName のコードは以下のとおりです。

```
public Contact contactsByName (String name)
    throws Exception
{
    String query = "name = ? Order By name";
    Iterator it = objectManager.openByQuery (Contact.class, query,
                                             new Object[]{ name});

    if (it.hasNext ())
        return (Contact) it.next ();
    throw new Exception ("Contact not found.");
}
```

このように記述した **openByQuery** は、引数に完全なクエリではなく SQL 条件を取ります。

Cachéドキュメントには、SQL 条件を規定する規則のリストがあります。このリストへのリンクは、下記の Note: を参照してください。

phonesByName のコードは以下のとおりです。

```
public Iterator phonesByName(String name)
throws Exception
{
    String query = "Select Distinct contacts.PhoneNumber.%ID," +
        " contacts.PhoneNumber.number, contacts.PhoneNumber.PhoneNumberType" +
        " From contacts.phoneNumber Where contacts.phoneNumber.owner ->" +
        " name %Startswith ?";

    return objectManager.openByQuery(query, new Object[]{name});
}
```

このように記述した **openByQuery** は、引数に完全な SQL クエリを取ります。ドキュメントには、SQL クエリの内容を規定する規則のリストがあります。このリストへのリンクは、下記の Note: を参照してください。

Note:

openByQueryに使用するSQL条件とSQLクエリの規則については、Cachéドキュメント:

[\[Caché言語バインディング\]](#)-[\[Caché JalapeñoでのJavaの使用法\]](#)-[\[Jalapeñoランタイム・ライブラリ・リファレンス\]](#)-[\[ObjectManager Interface\]](#)-[\[openByQuery\]](#) エントリを参照してください。

テスト

Test クラスは、Cache にオブジェクトを入力し、それらのオブジェクトを取得し、さらにそのプロパティを表示するために、いくつかの簡単なメソッドを実行します。これらのメソッドは、すべて **main** メソッドから実行します。

Test メソッド

Method (メソッド)	説明
Populate	複数の Contact 、 PhoneNumber および Address POJO を作成し、データベースに挿入します。
listContacts	データベース内のすべての Contact インスタンスを表示します。 DBService allContacts メソッドを使用してオブジェクトを取得します。
displayContact	1 つの Contact インスタンスを表示します。 DBService contactByName メソッドを使用してオブジェクトを取得します。
updateContact	Contact の type プロパティの値を切り替えます。 DBService saveContact メソッドを使用してデータベース・オブジェクトを取得します。
displayPhonesByName	指定された <i>name</i> 値を持つ Contact インスタンスに関連付けられたすべての PhoneNumber インスタンスを表示します。

※コードの詳細は、ダウンロードした(もしくは< cachesys>¥Dev¥tutorials¥pojo の) サンプルクラスのコードを参照して下さい。

例題の実行

例題のコードをビルドし、実行するには、次の手順を実行する必要があります。

1. Caché システムに対する正しい接続情報を DBService コンストラクタに追加します。
2. すべての Java クラスをコンパイルします。
3. SchemaBuilder ウィザードを使用してデータベース・スキーマを生成します。第 I 章の"スキーマの作成"ページに記載されている指示に従います。ここでは、Contact、PhoneNumber および Address POJO クラスに対するスキーマを生成します。
4. Java Test クラスをコンパイルします。

データベースのクリーンアップ

Jalapeño にはデータベースのデータを削除するためのユーティリティが含まれています。特に、この削除によってデータベース・スキーマが変更される場合は、**Test** を実行していないときに行う必要があります。このユーティリティは、以下のコマンド行で実行します。

```
>java -cp <Path To contacts classes>;<cachedsys>%Dev%java¥¥lib¥JDK15¥  
CacheDB.jar com.jalapeno.tools.util.KillExtent -host <hostname> -port <port>  
-namespace <namespace> -class contacts.Contact  
-class contacts.PhoneNumber -class contacts.Address
```

第 II 章-Chapter23

生成されたスキーマ

Test を 1 回実行した後、生成されたデータベース・テーブルは以下のようになります。

Contact

[更新](#) | [ウインドウを閉じる](#)

ネームスペース USER 中の contacts.Contact

最終更新: 2007-05-17 18:30:31.315

#	ID	ContactType	name	primaryAddress_City	primaryAddress_State	primaryAddress_Street	primaryAddress_Zip
1	1	Bus	Smith, John J.	Cambridge	MA	One Memorial Dr.	02142
2	2	Bus	Smith, Jane J.	Newton	MA	25 Glendale Avenue	02459
3	3	Bus	Jones, John P.	Worcester	MA	100 Main Street	03129
完了							

PhoneNumber

[更新](#) | [ウインドウを閉じる](#)

ネームスペース USER 中の contacts.PhoneNumber

最終更新: 2007-05-17 18:31:33.053

#	ID	PhoneNumberType	number	owner
1	1	H	881-447-4430	1
2	2	O	794-177-5087	1
3	3	M	073-813-0833	1
4	4	H	935-498-0637	2
5	5	H	670-407-9194	2
6	6	M	884-702-2649	2
7	7	M	375-490-6513	2
8	8	H	936-334-4523	2
9	9	M	954-081-5536	3
10	10	O	179-762-6737	3
11	11	H	114-754-8993	3
12	12	O	726-115-8901	3
13	13	M	399-316-7978	3
14	14	H	367-053-3000	3
完了				

※データはランダムに作成されますので、内容は上記と異なる場合があります。

要約

このチュートリアルの第 II 章では、以下について学習しました。

- POJO の永続性のために Jalapeño および Caché を使用する利点
- SchemaBuilder ウィザードのアクセス・タイプおよびアクセス・レベル・オプションを使用した、データベース・スキーマ生成の制御
- アノテーションを使用した、データベース・スキーマへのインデックス、制約、リレーションシップおよび埋め込みオブジェクトの追加
- **ObjectManager** `save`、`insert` および `update` メソッドによる、データベースへのデータの保存
- **ObjectManager** の `openByQuery` および `openByPrimaryKey` メソッドを使用した、主キーだけでなく SQL 条件やクエリによる Caché からのオブジェクトの取得

演習

以下の演習では、*Jalapeño* を使用して単純な銀行業務アプリケーションを作成する手順について説明します。

手順 1

3 つのクラス **Account**、**Address**、および **Customer** を定義します。*Jalapeño* アノテーションを使用して、POJO クラスと Caché オブジェクト・スキーマの間の対応を定義します。以下のテーブルは POJO メンバと Caché メンバの関係を示しています。

Account

POJO	Caché スキーマ
<i>long number</i>	<i>Property number As %Integer</i>
<i>String type</i>	<i>Property type As %String</i> 指定可能な値は、“checking” および “savings” です。
<i>float balance</i>	<i>property balance As %Float</i>
<i>Customer owner</i>	<i>Relationship owner As bank.customer</i> Cardiality=one, inverse=accounts

この Caché **Account** スキーマは *number* にインデックスを持ちます。このインデックスは主キーです。

Address

POJO	Caché スキーマ
<i>String mCity</i>	<i>City As %String</i>
<i>String mState</i>	<i>State As %String</i>
<i>String mStreet</i>	<i>Street As %String</i>
<i>String mZip</i>	<i>Zip As %String</i>

Caché **Address** クラスはシリアル（埋め込み可能）クラスです。

Customer

POJO	Caché スキーマ
<i>String idPlaceholder</i> システムによって割り当てられたデータベース ID を表します。	
<i>String name</i>	<i>Property name As %String</i>
<i>Address primaryAddress</i>	<i>Property primaryAddress As bank.address</i>
<i>String ssn</i>	<i>Property ssn As %String</i> 指定可能な値は、3N1"- "2N1"- "4N1 のパターンに制限されています。
<i>List <Account>accounts.</i>	<i>Relationship accounts As bank.account</i> Cardinality=many, inverse=owner

この Caché **Customer** スキーマは *ssn* にインデックスを持ちます。このインデックスは主キーです。

手順 II

DBService という Java クラスを定義し、POJO の永続性を管理します。**DBService** は以下のメソッドを実装します。

DBService

Method (メソッド)	説明
saveCustomer	新規 Customer を挿入するか、既存の Customer を更新します。
allCustomers	データベース内のすべての Customer オブジェクトを表す一連の Customer POJO のイテレーターを返します。
customerByName	指定された <i>name</i> 値を持つデータベース・オブジェクトに対応する Customer POJO を返します。
customerBySSN	指定された <i>ssn</i> 値を持つデータベース・オブジェクトに対応する Customer POJO を返します。

手順 III

以下を実行する Java **Test** クラスを作成します。

1. **DBService** **saveCustomer** メソッドを使用して複数の **Customer**、**Account**、および **Address** オブジェクトをデータベースに追加します。
2. **DBService** **allCustomers**、**customerByName**、および **customerBySSN** メソッドをテストします。

Note:

完成した例題のコードは、InterSystems Developer's CornerのWebサイトからダウンロードすることができます。<http://vista.intersystems.com/samples/PojoTutorial.zip>にアクセスしてダウンロードしてください。

例題のコードは、Caché の標準インストールにも含まれています。そのコードは、`<cachsys>%Dev%tutorials%pojo` にあります。