第一章

初探

Java是一种使用用户定义类（user-defined class）运行时环境的程序语言。这些用户类的实例可代表存储于数据库、文件系统或主机事务处理系统的真实数据。此外，小型环境也需要有办法来管理在本机存储空间内的持久性（persistent）数据。

由于各种数据来源的访问数据的技术不尽相同，因此对程序开发人员而言，访问数据会是一种挑战，因为各种数据来源需要用不同的应用程序接口（API）。这表示至少需要熟悉两种程序语言才能开发这些数据来源的事务逻辑（business logic）：Java程序语言及数据来源所要求的特定数据访问语言。此外，各种数据来源的数据访问语言也可能会不同，也更提高了学习及使用各种数据来源的成本。

在发行“Java数据对象”（Java Data Object，JDO）之前，有三种储存Java数据的标准：串行化（serialization）、JDBC（Java DataBase Connectivity）以及EJB（Enterprise Java Bean）的CMP（Container Managed Persistence）。串行化是用来将对象状态与其所引用（reference）的对象图写入输出流（output stream）。它保留了Java对象的关系，因此日后可以重建完整的对象图。不过串行化并不支持事件、查询以及多位用户之间的数据共享，它只允许在原始串行化层面上访问数据，因此当应用程序需要管理多个串行化时会十分麻烦。串行化只能用于简单程序中的数据保存，或用于无法有效支持数据库的嵌入式环境中。

JDBC则需要明确地管理字段值，并将其对应至关系数据库的数据表。开发者不得不处理两种截然不同的数据模型、程序语言及数据访问的规范 —— Java与SQL的关系数据模型。由于实现关系数据模型与Java对象模型之间的对应关系，所花的开发工作过于庞大，因此多数开发者从未替其数据定义对象模型。他们只是编写程序上的Java程序代码来处理底层关系数据库的数据表，因此最终仍无法有效利用面向对象程序开发的优点。

EJB组件结构是设计用来支持分布式对象运算的，它也提供经由CMP（Container Managed Persistence）的数据保存功能，这主要归因于其分布式功能。EJB 应用程序比JDO要更复杂而且有更多系统额外开销（overhead），不过，借着与EJB容器（container）的集成，JDO的实现已经可以在EJB容器环境中提供数据保存的功能。如果应用程序需要对象能持久保存，但不需具备分布式对象功能，你可以使用JDO来取代EJB组件。在EJB环境中使用JDO最普遍的方式是让EJB的Session Bean直接管理JDO对象，而避免使用Entity Bean。EJB组件必须在受管理的应用程序服务器环境中运行；但是JDO应用程序却可在受管理或未管理的环境中运行，因此具备灵活性，可选择最合适的环境来运行应用程序。

如果能将重点放在设计Java对象模型，并使用JDO直接存储类的实例，就可以更有效率地开发应用程序。你只需处理单一信息模型，JDBC则需要先了解关系模型与SQL语言，而在使用EJB CMP时，你还得被迫学习及处理此结构的其他功能。此外，EJB还有建构模型上的限制，而JDO却无此限制。

JDO定义了持久保存类与JDO运行时环境之间的关系。JDO的设计目的是用来广泛支持不同的数据来源，甚至包括一般不被认为是数据库的来源。因此我们用"数据存储空间"（datastore）一词泛指以JDO访问的底层数据来源。

本章用一间名为Media Mania的虚构公司所开发的小型应用程序，来探索一些JDO的基本功能。这间公司在遍布美国的分店中租售各种形式的娱乐媒体。这些分店有信息站（kiosk）可以提供关于电影及片中演员的信息，这些信息可以帮助客户及店员选择吸引客户的商品。

定义持久保存的对象模型

图1-1是表示Media Mania对象模型中的类及其关系的"统一建模语言"（Unified Modeling Language，UML）示意图。一个`Movie`实例代表一部电影，至少在某一部电影中扮演其中角色的演员则由`Actor`实例来代表。`Role`类代表演员在一部电影中所扮演的特定角色，因此也代表`Movie`与含有属性（该角色的名称）的`Actor`之间的关系。每一部电影都有一个以上的角色。演员可以在多部电影中担任某种角色，或在单部电影中饰演好几个人。

我们将这些持久保存类以及用来管理其实例的应用程序放入Java的`com.mediamania.prototype`包中。

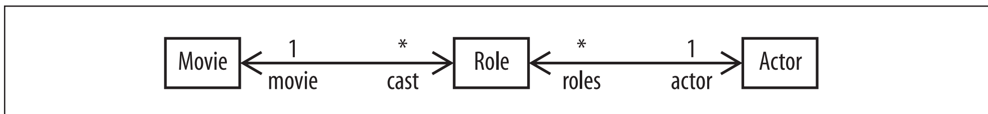

图 1-1：Media Mania 对象模型的 UML 示意图

持久保存的类

这一节我们要让Movie、Actor与Role类能持久保存，以便将其实例存入数据存储空间。首先我们会检查各个类的完整源代码。每个类都有import语名，因此可以清楚知道哪个包含有范例中所用的类。

例1-1列出了Movie类的源代码。JDO是定义在javax.jdo包中，请注意，此类并不用导入任何特定的JDO类。在java.util包中所定义的Java引用与集合可用来代表类之间的关系，而且这也是多数Java应用程序的标准做法。

Movie类的字段使用标准的Java类型，如String、Date以及int。你可以将字段声明成私有的（private），不一定要替各个字段定义公共的（public）get与set方法。Movie类含有一些方法来取得及设定类中的私有字段，不过这些方法也用在应用程序的其他部分，而且也不是JDO所必要的。你也可以利用封装技术，只提供支持被模型化的抽象数据的方法。此类也有静态（static）字段，它们不会被存入数据存储空间中。

genres字段是保存电影类型（动作片、文艺爱情片、悬疑片等）的String，Set接口则用来引用一组代表电影演员表的Role实例。addRole()方法会将元素新增至cast集合中，而getCast()则返回含有cast集合且不能修改的Set。这些方法都不是JDO的需求，而是为了应用程序的方便而实现的。parseReleaseDate()与formatReleaseDate()方法是用来标准化电影出片日期的格式。为了维持源代码的简洁，如果parseReleaseDate()的参数格式有误，即返回null值。

例 1-1：Movie.java

```
package com.mediamania.prototype;

import java.util.Set;
import java.util.HashSet;
import java.util.Collections;
import java.util.Date;
import java.util.Calendar;
import java.text.SimpleDateFormat;
import java.text.ParsePosition;

public class Movie {
```

```
private static SimpleDateFormat yearFmt = new SimpleDateFormat("yyyy");
public  static final String[] MPAAratings =
                            { "G", "PG", "PG-13", "R", "NC-17", "NR" };
private String      title;
private Date        releaseDate;
private int         runningTime;
private String      rating;
private String      webSite;
private String      genres;
private Set         cast;    // 数据类型：Role

private Movie()
{}

public Movie(String title, Date release, int duration, String rating,
             String genres) {
    this.title = title;
    releaseDate = release;
    runningTime = duration;
    this.rating = rating;
    this.genres = genres;
    cast = new HashSet();
}
public String getTitle() {
    return title;
}
public Date getReleaseDate() {
    return releaseDate;
}
public String getRating() {
    return rating;
}
public int getRunningTime() {
    return runningTime;
}
public void setWebSite(String site) {
    webSite = site;
}
public String getWebSite() {
    return webSite;
}
public String getGenres() {
    return genres;
}
public void addRole(Role role) {
    cast.add(role);
}
public Set getCast() {
    return Collections.unmodifiableSet(cast);
}
public static Date parseReleaseDate(String val) {
    Date date = null;
    try {
```

```
            date = yearFmt.parse(val);
        } catch (java.text.ParseException exc) {}
        return date;
    }
    public String formatReleaseDate() {
        return yearFmt.format(releaseDate);
    }
}
```

为了让类能持久保存，JDO有一项要求：无参数的构造函数。如果在类中未定义任何构造函数，则编译器会自动产生无参数的构造函数；不过，如果定义了任何含有参数的构造函数，就不会产生此默认的构造函数。在此情况下，你需要自行提供无参数的构造函数；如果不想让应用程序的源代码使用它，可以将其定义为`private`。某些JDO实现可以替你产生一个默认的构造函数，不过此功能会视产品的类型而异，而且是无法移植的。

例1-2提供`Actor`类的源代码。在此展示用的范例中，所有的演员都有可辨识他们的唯一的名字（这可以是与本名不同的艺名）。因此，我们用一个`String`来表示演员的名字。每位演员可以扮演一个以上的角色，而`roles`成员则描述于`Actor`端`Actor`与`Role`之间的关系。在行❶的批注只是提供说明，在JDO中没有任何功能上的用途。行❷及行❸的`addRole()`与`removeRole()`方法则用来为应用程序维护`Actor`实例与其相关的`Role`实例间的关系。

例1-2：Actor.java

```
package com.mediamania.prototype;

import java.util.Set;
import java.util.HashSet;
import java.util.Collections;

public class Actor {
    private String  name;
❶   private Set     roles; // 数据类型：Role

    private Actor()
    {}
    public Actor(String name) {
        this.name = name;
        roles = new HashSet();
    }
    public String getName() {
        return name;
    }
❷  public void addRole(Role role) {
        roles.add(role);
    }
❸  public void removeRole(Role role) {
        roles.remove(role);
```

```
    }
    public Set getRoles() {
        return Collections.unmodifiableSet(roles);
    }
}
```

最后，例1-3列出了Role类的源代码。此类为Movie与Actor间的关系的对象模型，并包括在影片中由该演员所扮演的特定角色的名称。Role构造函数会初始化指向Movie与Actor的引用，并且借着调用在Movie及Actor类中都有定义的addRole()更新此关系中的另一端。

例1-3：Role.java

```
package com.mediamania.prototype;

public class Role {
    private String  name;
    private Actor   actor;
    private Movie   movie;

    private Role()
    {}
    public Role(String name, Actor actor, Movie movie) {
        this.name = name;
        this.actor = actor;
        this.movie = movie;
        actor.addRole(this);
        movie.addRole(this);
    }
    public String getName() {
        return name;
    }
    public Actor getActor() {
        return actor;
    }
    public Movie getMovie() {
        return movie;
    }
}
```

我们已经看过以数据存储空间来保存实例的类的完整源代码。这些类不需导入并使用任何JDO类型。此外，除了提供无参数的构造函数外，也不需定义数据或方法便能将这些类设定成可持久保存。用来访问及修改字段，以及定义与管理实例间关系的软件只需依循大多数Java应用程序所用的标准惯例。

将类声明为持久保存

你必须判断出哪些类应该能持久保存，并定义在 Java 中无法表示的持久保存的相关信息。JDO 会使用 XML 格式的元数据文件来设定此类信息。

你可以在一个以上的 XML 文件中，依据类或包来定义元数据。类的元数据文件的文件名是以类的名称加上扩展名 *.jdo*。因此，`Movie` 类的元数据文件名应为 *Movie.jdo*，并与 *Movie.class* 文件放在同一目录中。Java 包的元数据文件则命名为 *package.jdo*，它可以包含多个类与子包的元数据。例 1-4 提供了 Media Mania 对象模型的元数据。此元数据是针对包而设定的，并放入 *com/mediamania/protopye/package.jdo* 文件中。

例 1-4：prototype/package.jdo 文件中的 JDO 元数据

```
   <?xml version="1.0" encoding="UTF-8" ?>
❶  <!DOCTYPE jdo PUBLIC
       "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 1.0//EN"
       "http://java.sun.com/dtd/jdo_1_0.dtd">
   <jdo>
❷      <package name="com.mediamania.prototype" >
❸               <class name="Movie" >
❹                   <field name="cast" >
❺                       <collection element-type="Role"/>
                   </field>
               </class>
❻          <class name="Role" />
           <class name="Actor" >
               <field name="roles" >
                   <collection element-type="Role"/>
               </field>
           </class>
       </package>
   </jdo>
```

行❶上所指定的 *jdo_1_0.dtd* 描述可在 JDO 元数据文件中使用的 XML 元素。在 JDO 中，此“文件类型定义”（Document Type Definition，DTD）已经标准化，因此应该随附在JDO产品中；你也可以从 *http://java.sun.com/dtd* 下载。此外，你也可以更改`DOCTYPE`指向本机文件系统中的文件。

元数据文件可以包含数个拥有持久保存类的包的永久信息，每一个包都以带有该Java包名称的`package`元素来定义。行❷显示的是`com.mediamania.prototype`包的`package`元素，在`package`元素中含有嵌套的`class`元素来辨识包的持久保存类（例如，在行❸有`Movie`类的`class`元素）。此文件可以包含连续多个`package`元素，但不能是嵌套的。

如果必须对类的特定字段设定信息，则在`class`元素中可以使用嵌套的`field`元素，如行❹所示。例如，你可以定义模型中各个集合的数据类型。虽然这不是必要的，但却可产生更有效率的对应关系。`Movie`类含有称为`cast`的集合，而`Actor`类则含有称为`roles`的集合，两者都含有指向`Role`的引用。行❺设定`cast`的数据类型。多数情况下，元数据中会给予属性的默认值，以提供最常用的引用值。

在默认情况下，所有能持久保存的字段都会被设为持久保存，但static与final字段则不能持久保存。在默认情况下，在Java中声明为transient的字段不能持久保存，但是在元数据文件中，却可以将此字段定义为持久保存。第四章会描述这个功能。

第四、十、十二及十三章会说明能替类与字段指定的其他特征。对于类似`Role`这种不含任何集合的简单类，如果不需要使用其他元数据的属性，你可以如行❻那样列出元数据中的类。

项目的编译环境

本节中会说明编译及运行JDO应用程序的开发环境。这包括项目的目录结构、编译应用程序所需的jar文件以及增强持久保存类的语法。本节稍后会描述类的增强。环境设定有一部分会视所用的JDO实现而异，因此读者的特定项目的开发环境及目录结构可能会与书中不同。

你可以使用Sun公司的JDO参考实现，或其他自行挑选的实现，本书的范例则使用JDO参考实现。你可以访问*http://www.jcp.org*并选择JSR-12来下载JDO的参考实现。当安装完JDO实现后，则需要建立项目的目录结构，并定义包括编译与运行应用程序所需的所有目录及jar文件的classpath。

JDO会在编译过程中加入称为“类的增强”（class enhancement）的新步骤。每个持久保存类都必须被增强，才能在JDO运行时环境中使用。先以Java编译器来编译持久保存类，产生其类文件；然后类增强程序会读入这些类文件与JDO元数据，并产生增强过的新类文件，以便能在JDO环境中运行。你的JDO应用程序必须加载这些增强过的类文件。JDO参考实现内含“参考增强工具”（reference enhancer）的增强程序。

使用JDO参考实现所需的Jar

当使用JDO参考实现时，在开发过程中，应该在classpath中加入下列的jar文件。而在运行时，所有这些jar文件也应该在classpath中。

jdo.jar

定义在 JDO 规范中的标准接口与类。

jdori.jar

Sun 公司针对 JDO 规范的参考实现。

btree.jar

JDO参考实现用以管理文件中的数据存储的软件。参考实现会使用文件来储存持久保存的实例。

jta.jar

Java Transaction API。定义在 `javax.transaction` 包中的 `Synchronization` 接口会于 JDO 接口中用到，它包含在此 jar 文件内；此文件中定义的其他工具对 JDO 实现也可能会有用处。你可以从 *http://java.sun.com/products/jta/index.html* 下载此 jar 文件。

antlr.jar

在 JDO 查询语言的实现中所用的解析技术。参考实现使用 Antlr 2.7.0 版。你可以从 *http://www.antlr.org* 下载此文件。

xerces.jar

参考实现使用 Xerces-J 1.4.3 版来解析 XML。你可以从 *http://xml.apache.org/xerces-j/* 下载此文件。

前三个 jar 文件已包含在 JDO 参考实现中，而后三个则可以从相关网站上下载。

参考实现还包括额外的jar文件 —— *jdori-enhancer.jar*，其中含有参考增强工具的实现。*jdori-enhancer.jar* 中的类也包含在 *jdori.jar* 之中。多数情况下，在开发及运行时环境中都会使用 *jdori.jar*，而不需要 *jdori-enhancer.jar*。*jdori-enhancer.jar* 是单独包装的，所以你能独立地使用此参考增强工具来增强类，而不用管 JDO 实现的版本。除了参考实现外，某些实现也会发布此 jar 文件来与其实现搭配。

如果使用不同的 JDO 实现，其说明文件应该会提供所有必要的 jar 文件的列表。实现通常会将所有必要的 jar 文件放入其安装目录下的特定目录中。含有定义于 JDO 中的接口的 *jdo.jar* 文件则应该用于所有的实现。通常在厂商的实现中也会含有此 jar 文件。JDOcentral.com（*http://www.jdocentral.com*）提供许多 JDO 的资源，包括许多商用 JDO 实现的免费试用版。

项目的目录结构

对于Media Mania应用程序的开发环境，你应该使用下列的目录结构。在文件系统中必须先建立项目的 *root* 目录，下列的目录则位于项目的 *root* 目录之下：

src
: 此目录含有应用程序所有的源代码，在 *src* 之下则有 *com/mediamania/prototype* 的子目录层次结构（对应于Java的 `com.mediamania.prototype` 包）。这也是放置 *Movie.java*、*Actor.java* 以及 *Role.java* 源代码文件之处。

classes
: 当编译完Java源代码之后，其类文件会放在此目录中。

enhanced
: 此目录含有增强过的类文件（由增强工具产生）。

database
: 此目录含有参考实现用来储存持久保存数据的文件。

虽然此特殊的目录结构并非JDO或参考实现的要求，但你却需要了解它后才能了解本章对 Media Mania 应用程序的说明。

当运行JDO应用程序时，Java运行时环境必须加载增强过的类文件，这些文件位于enhanced目录中。因此，在classpath中，*enhanced* 目录必须放在 *classes* 目录之前。另外一种替代做法是，你可以就地直接增强并以增强过的文件来取代未增强过的类文件。

增强持久保存类

在JDO环境中管理类实例前，必须先增强类。JDO增强工具会将数据及方法加入类中，让JDO实现能管理其实例。增强工具会利用JDO元数据读入由Java编译器产生的类文件，并产生增强过而且包含必要功能的新类文件。JDO已将增强工具所产生的修改过程标准化，因此增强过的类文件具有“二进制兼容性”（binary compatible），可用于任何JDO实现。这些增强过的类文件也与任何特定的数据存储空间无关。

如前所述，Sun公司的JDO参考实现所提供的增强工具被称为“参考增强工具”。JDO供货商可以提供自己的增强工具，而执行增强工具所需的命令行语法可能会不同于此处所示的。每种实现都应该提供文件，说明如何增强类，以便用于其实现中。

例1-5提供了参考增强工具的命令，来增强Media Mania应用程序中的持久保存类。`-d` 参数设定 *root* 目录来放置增强过的文件，此处我们指定为 *enhanced* 目录。增强工具会

接受一组JDO元数据文件及一组待增强的类文件。依据操作系统与编译环境的不同，目录分隔符与命令行连续符号也会有差异。

例 1-5：增强持久保存类

```
java com.sun.jdori.enhancer.Main -d enhanced  \
      classes/com/mediamania/prototype/package.jdo  \
      classes/com/mediamania/prototype/Movie.class  \
      classes/com/mediamania/prototype/Actor.class  \
      classes/com/mediamania/prototype/Role.class
```

虽然将元数据文件与源代码放入同一目录会比较方便，JDO规范却建议经由加载类文件的类加载器所加载的资源来访问元数据文件。在编译及运行时都需要用到元数据，因此，我们将 *package.jdo* 元数据文件放在 `prototype` 包目录中的 *classes* 目录下。

虽然例 1-5 一次性列出在对象模型中所有持久保存类的类文件，但你也可以单独增强各个类。当执行此命令时，它会将增强过的新类文件放入 *enhanced* 目录中。

建立数据存储空间连接与事务

在增强类后，就可以将其实例存入数据存储空间。接着，我们会说明应用程序如何与数据存储空间建立连接并进行事务内的操作。我们会开始编写直接利用JDO接口的软件，应用程序所使用的 JDO 接口都定义在 `javax.jdo` 包内。

JDO 含有称为 `PersistenceManager` 的接口，它具备与数据存储空间连接的功能。`PersistenceManager`拥有JDO `Transaction`接口的实例，可用来控制事务的启动与完成。凭借在 `PersistenceManager` 实例上调用 `currentTransaction()` 方法，你可以取得 `Transaction` 实例。

取得 PersistenceManager

`PersistenceManagerFactory`用来设定并取得 `PersistenceManager`。`Persistence-ManagerFactory`中的方法则可设定属性（property），控制从工厂（factory）所取得的 `PersistenceManager` 实例的行为。因此，JDO 应用程序执行的第一步就是取得 `PersistenceManagerFactory`实例。请调用下列`JDOHelper`类的静态方法来取得此实例：

```
static PersistenceManagerFactory getPersistenceManagerFactory(Properties props);
```

`Properties`实例的属性参数值可以用编写程序的方式填入，或者从属性文件（property file）载入。例 1-6 列出了在Media Mania应用程序中所使用的属性文件的内容。位于行

❶的PersistenceManagerFactoryClass属性以提供实现PersistenceManagerFactory接口的类名称的方式来指定所使用的JDO实现。在此例中，我们设定的是在Sun公司的JDO参考实现中定义的类。其余的属性包括用来连接特定数据存储空间的连接URL，以及建立数据存储空间连接所需的用户名称与密码。

例 1-6：jdo.properties 的内容

```
❶ javax.jdo.PersistenceManagerFactoryClass=com.sun.jdori.fostore.FOStorePMF
javax.jdo.option.ConnectionURL=fostore:database/fostoredb
javax.jdo.option.ConnectionUserName=dave
javax.jdo.option.ConnectionPassword=jdo4me
javax.jdo.option.Optimistic=false
```

连接URL的格式依据欲访问的特定数据存储空间类型而异，JDO参考实现本身拥有一种存储设备 —— 文件对象数据库（File Object Store, FOStore）。在例1-6中的ConnectionURL属性指明数据存储空间是位于项目的*root*目录下的*database*目录中。在此例中，我们提供的是相对路径；不过，你也可以提供指向数据存储空间的绝对路径。例1-6 中的URL指明 FOStore 数据存储文件的主文件名是*fostoredb*。

如果使用不同的实现，则需要替这些属性设定不同的参数值。你可能还需要提供其他属性的参数值，请查阅实现的说明文件来确定必要的属性。

建立 FOStore 数据存储空间

使用FOStore之前，必须先建立数据存储空间。例 1-7 中的程序会利用*jdo.properties*文件来建立数据存储空间，所有的应用程序都会使用此属性文件。行❶会将*jdo.properties*中的属性加载到 Properties 实例中；程序会在行❷加入 com.sun.jdori.option.ConnectionCreate属性来指出应该建立数据存储空间；将此属性设为true会指示JDO实现建立数据存储空间；然后，我们会调用位于行❸的getPersistenceManagerFactory()以取得 PersistenceManagerFactory；行❹则会建立 PersistenceManager。

为了完成数据存储空间的建立，我们也必须启动（begin）及确认（commit）事务。在行❺会调用 PersistenceManager 的currentTransaction()方法，以访问与PersistenceManager相关联的Transaction实例；在行❻及❼则调用Transaction的begin()与commit()方法，来启动及确认事务；当运行此程序时，在*database*目录中会产生 FOStore 的数据存储空间。有两个文件会产生：*fostore.btd*与*fostore.btx*。

例 1-7：建立 FOStore 数据存储空间

```
package com.mediamania;

import java.io.FileInputStream;
import java.io.InputStream;
import java.util.Properties;
import javax.jdo.JDOHelper;
import javax.jdo.PersistenceManagerFactory;
import javax.jdo.PersistenceManager;
import javax.jdo.Transaction;

public class CreateDatabase {
    public static void main(String[] args) {
        create();
    }
    public static void create() {
        try {
            InputStream propertyStream = new FileInputStream("jdo.properties");
            Properties jdoproperties = new Properties();
❶           jdoproperties.load(propertyStream);
❷           jdoproperties.put("com.sun.jdori.option.ConnectionCreate", "true");
            PersistenceManagerFactory pmf =
❸                       JDOHelper.getPersistenceManagerFactory(jdoproperties);
❹           PersistenceManager pm = pmf.getPersistenceManager();
❺           Transaction tx = pm.currentTransaction();
❻           tx.begin();
❼           tx.commit();
        } catch (Exception e) {
            System.err.println("Exception creating the database");
            e.printStackTrace();
            System.exit(-1);
        }
    }
}
```

JDO参考实现提供这种编写程序的方法来建立数据存储空间，大多数的数据库软件则会提供与JDO不同的工具来建立数据库。JDO并未定义标准的、与供货商无关的接口来建立数据库，建立数据存储空间永远取决于特定的数据存储空间。本程序只是示范如何用FOStore的数据存储空间来建立数据存储空间。

此外，当使用JDO来处理关系数据库时，通常还有额外的步骤来建立或对应至现存的关系模式（schema）。建立对应于JDO对象模型的模式的程序则与所采用的实现有关，你应该查阅所使用的实现的说明文件来确定必要的步骤。

操作实例

现在我们已经有可以储存类实例的数据存储空间了，每个应用程序都需要取得PersistenceManager来访问及更新数据存储空间。例1-8提供了MediaManiaApp类的源代码，此类为本书中所有应用程序的基础类（base class），每个应用程序都是MediaManiaApp的具体子类，并在execute()方法中实现其应用程序逻辑。

MediaManiaApp拥有一个构造函数，可从*jdo.properties*载入属性（行❶）。从文件载入属性之后，它会调用getPropertyOverrides()并将返回的属性合并至jdoproperties。应用程序子类可以重新定义getPropertyOverrides()以提供任何额外的属性，或更改*jdo.properties*文件所设定的属性。此构造函数会取得PersistenceManagerFactory（行❷）及PersistenceManager（行❸）。我们也提供getPersistenceManager()方法，以便从MediaManiaApp类之外访问PersistenceManager。行❹则会取得与PersistenceManager相关联的Transaction。

应用程序子类会调用定义在MediaManiaApp类中的executeTransaction()。此方法会在行❺处启动事务，然后在行❻调用execute()来执行子类的特定功能。

我们选择此特别的应用程序类的设计，来简化并减少范例中建立运行环境时多余的程序代码。这不是JDO的要求，你可以选择最适合于你的应用程序环境的做法。

execute()方法（由子类实现）返回后，程序会试着确认事务（行❼）。如果抛出了任何异常（exception），就会将事务回滚（roll back），并将异常信息输出至错误流。

例1-8：MediaManiaApp的基础类

```
package com.mediamania;

import java.io.FileInputStream;
import java.io.InputStream;
import java.util.Properties;
import java.util.Map;
import java.util.HashMap;
import javax.jdo.JDOHelper;
import javax.jdo.PersistenceManagerFactory;
import javax.jdo.PersistenceManager;
import javax.jdo.Transaction;
public abstract class MediaManiaApp {
    protected PersistenceManagerFactory pmf;
    protected PersistenceManager        pm;
    protected Transaction               tx;

    public abstract void execute();  // 定义在具体的应用程序子类中
```

```
    protected static Map getPropertyOverrides() {
        return new HashMap();
    }
    public MediaManiaApp() {
        try {
            InputStream propertyStream = new FileInputStream("jdo.properties");
            Properties jdoproperties = new Properties();
❶           jdoproperties.load(propertyStream);
            jdoproperties.putAll(getPropertyOverrides());
❷           pmf = JDOHelper.getPersistenceManagerFactory(jdoproperties);
❸           pm = pmf.getPersistenceManager();
❹           tx = pm.currentTransaction();
        } catch (Exception e) {
            e.printStackTrace(System.err);
            System.exit(-1);
        }
    }
    public PersistenceManager getPersistenceManager() {
        return pm;
    }
    public void executeTransaction() {
        try {
❺           tx.begin();
❻           execute();
❼           tx.commit();
        } catch (Throwable exception) {
            exception.printStackTrace(System.err);
            if (tx.isActive()) tx.rollback();
        }
    }
}
```

让实例能持久保存

让我们看一个简单的应用程序 —— CreateMovie。如例 1-9 所示，它会让一个 Movie 实例能持久保存。此应用程序的功能性放在 execute() 方法中。在构造 CreateMovie 的实例后，我们会调用定义在 MediaManiaApp 基础类中的 executeTransaction()，此方法会调用定义在此类中的 execute()。在行 ❺ 处，execute() 方法会建立一个 Movie 实例。调用行 ❻ 中 PersistenceManager 的 makePersistent() 方法则会让 Movie 实例能持久保存。如果在 executeTransaction() 中成功地确认事务，就会将 Movie 实例存入数据存储空间。

例 1-9：建立 Movie 实例并让它能持久保存

```
package com.mediamania.prototype;

import java.util.Calendar;
import java.util.Date;
```

```
import com.mediamania.MediaManiaApp;

public class CreateMovie extends MediaManiaApp {
    public static void main(String[] args) {
        CreateMovie createMovie = new CreateMovie();
        createMovie.executeTransaction();
    }
    public void execute() {
        Calendar cal = Calendar.getInstance();
        cal.clear();
        cal.set(Calendar.YEAR, 1997);
        Date date = cal.getTime();
❺      Movie movie = new Movie("Titanic", date, 194, "PG-13", "historical, drama");
❻      pm.makePersistent(movie);
    }
}
```

现在，让我们来看稍微复杂些的应用程序。如例1-10所示，`LoadMovies`会读取含有电影资料的文件，并创建多个`Movie`实例。文件名称是以参数的形式传给应用程序，而`LoadMovies`构造函数会初始化`BufferedReader`来读取数据。`execute()`方法会从文件一次读取一行数据，并调用`parseMovieData()`来解析输入数据行，在行上创建`Movie`实例，并让它能持久保存在行上。当在`executeTransaction()`中确认完事务后，所有新创建的`Movie`实例都会被存入数据存储空间。

例1-10：LoadMovies

```
package com.mediamania.prototype;

import java.io.FileReader;
import java.io.BufferedReader;
import java.util.Calendar;
import java.util.Date;
import java.util.StringTokenizer;
import javax.jdo.PersistenceManager;
import com.mediamania.MediaManiaApp;

public class LoadMovies extends MediaManiaApp {
    private BufferedReader  reader;

    public static void main(String[] args) {
        LoadMovies loadMovies = new LoadMovies(args[0]);
        loadMovies.executeTransaction();
    }
    public LoadMovies(String filename) {
        try {
            FileReader fr = new FileReader(filename);
            reader = new BufferedReader(fr);
        } catch (Exception e) {
            System.err.print("Unable to open input file ");
            System.err.println(filename);
```

```
            e.printStackTrace();
            System.exit(-1);
        }
    }
    public void execute() {
        try {
            while ( reader.ready() ) {
                String line = reader.readLine();
                parseMovieData(line);
            }
        } catch (java.io.IOException e) {
            System.err.println("Exception reading input file");
            e.printStackTrace(System.err);
        }
    }
    public void parseMovieData(String line) {
        StringTokenizer tokenizer = new StringTokenizer(line, ";");
        String title = tokenizer.nextToken();
        String dateStr = tokenizer.nextToken();
        Date releaseDate = Movie.parseReleaseDate(dateStr);
        int runningTime = 0;
        try {
            runningTime = Integer.parseInt(tokenizer.nextToken());
        } catch (java.lang.NumberFormatException e) {
            System.err.print("Exception parsing running time for ");
            System.err.println(title);
        }
        String rating = tokenizer.nextToken();
        String genres = tokenizer.nextToken();
❶      Movie movie = new Movie(title, releaseDate, runningTime, rating, genres);
❷      pm.makePersistent(movie);
    }
}
```

电影数据文件的格式为：

```
片名 ; 出片日期 ; 片长 ; 影片分级;影片类型1,影片类型2,影片类型3
```

出片日期的格式是在`Movie`类中处理的，因此要调用`parseReleaseDate()`从输入数据来创建`Date`实例。一部电影由一个以上的类型来描述，这些类型列在数据行的尾端。

访问实例

现在让我们访问数据存储空间中的`Movie`实例，以确认这些实例已经被成功地存入存储库。访问JDO中的实例有下列几种方式：

- 迭代类的extent（范围）
- 寻访对象模型

- 执行查询

“ extent ”是一种用来访问特定类的所有实例或是类及其所有子类的机制。如果应用程序只想访问一部分的实例，可以由过滤器来执行查询，限制返回的实例必须符合Boolean判断式。在应用程序访问数据存储空间中的实例后，它可以通过检查引用并迭代对象模型中的集合元素，来寻访数据存储空间中的相关实例，而尚未存入内存的实例可视需要从数据储存空间中读取。这些访问实例的机制经常混合使用，而JDO可确保每个PersistenceManager中，所有持久保存的实例在应用程序的内存中只会出现一次。每个PersistenceManager会管理一个事务环境。

迭代类的 extent

JDO提供Extent接口来访问类的extent。extent可访问类的所有实例，但使用extent并不表示所有的实例都会放在内存中。在例1-11中的PrintMovies应用程序会使用Movie的extent。

例1-11：迭代Movie的extent

```
package com.mediamania.prototype;

import java.util.Iterator;
import java.util.Set;
import javax.jdo.PersistenceManager;
import javax.jdo.Extent;
import com.mediamania.MediaManiaApp;

public class PrintMovies extends MediaManiaApp {

    public static void main(String[] args) {
        PrintMovies movies = new PrintMovies();
        movies.executeTransaction();
    }

    public void execute() {
❶       Extent extent = pm.getExtent(Movie.class, true);
❷       Iterator iter = extent.iterator();
        while (iter.hasNext()) {
❸           Movie movie = (Movie) iter.next();
            System.out.print(movie.getTitle());       System.out.print(";");
            System.out.print(movie.getRating());      System.out.print(";");
            System.out.print(movie.formatReleaseDate()); System.out.print(";");
            System.out.print(movie.getRunningTime()); System.out.print(";");
❹           System.out.println(movie.getGenres());

❺           Set cast = movie.getCast();
            Iterator castIterator = cast.iterator();
            while (castIterator.hasNext()) {
```

```
❻                   Role role = (Role) castIterator.next();
                    System.out.print("\t");
                    System.out.print(role.getName());
                    System.out.print(", ");
❼                   System.out.println(role.getActor().getName());
                }
            }
❽           extent.close(iter);
        }
    }
```

在行❶，我们会从PersistenceManager取得Movie类的Extent。第二个参数表示是否要引入Movie子类的实例。即使有子类的实例存在，参数值false也只会让程序返回Movie实例。虽然目前还没有任何继承Movie类的子类，但提供true值会返回未来可能会定义的任何子类的实例。行❷会调用Extent接口中的iterator()方法来取得Iterator，以便访问extent中的各个元素。行❸则使用Iterator访问Movie实例。接着，应用程序就可以操作Movie实例，并取得关于要显示的影片的数据。而行❹调用getGenres()来取得影片的类型；行❺可取得Role的集合；在行❻，我们取得指向Role的引用，并显示出角色名称；行❼调用getActor()以寻访该角色的Actor，此方法是定义在Role类中。然后，我们会显示出演员的名字。

当应用程序完成迭代extent的操作后，行❽会关闭Iterator以释放迭代extent所需的任何资源。多个Iterator实例可以并发地作用在同一Extent上。此方法会关闭特定的Iterator，closeAll()则会关闭与同一Extent有关的所有Iterator实例。

寻访对象模型

例1-11示范如何逐一访问Movie的extent。不过，借着行❻来迭代对象模型的集合，我们也寻访到一组相关的Role实例；行❼使用Role实例来浏览指向相关的Actor实例的引用；行❺及❼分别示范一对多及一对一的关系。类与类间的关系有一个重要的特性，它指明是否有一或多个相关的实例。对于只有一个实例的情况，会使用“引用”（reference）；而当可能有一个以上的实例时，则使用“集合”（collection）。

访问这些相关的实例所需的语法与寻访内存中的实例的标准做法相同。应用程序不需要直接调用行❸至❼之间的JDO接口，它只需逐一寻访内存中的对象。除非应用程序直接访问相关的实例，否则这些实例不会从数据存储空间中被读取并在内存中被实例化。访问数据存储空间是具有透明性的，在需要时才会将实例写入内存中。某些实现会提供不同于Java接口的机制，让你能改变实现的访问及缓冲算法。你的Java应用程序并没有这些最佳化机制，不过却可以利用它们来影响整体的效能。

相关的持久保存实例在 JDO 环境中的访问方式，与在非 JDO 环境中访问瞬时的实例是一样的，因此你可以设计软件让它与在JDO环境中的使用方式无关。在不具任何JDO知识或其他持久保存考虑下写出的软件，还是可以经由JDO寻访数据库中的对象。此功能可以大幅提高开发的生产力，并可迅速且轻易地将现存的软件并入 JDO 的环境中。

执行查询

你也可以对 `Extent` 进行查询的操作。JDO `Query` 接口是用来选择符合某些条件的实例。本章其余的范例需要依据唯一的名称来访问特定的`Actor`或`Movie`，如例 1-12 所示，这些方法几乎都是相同的：`getActor()`会依据名称来查询，以取得`Actor`；而`getMovie()`则依据名称来查询，以取得`Movie`。

例 1-12：PrototypeQueries 类中的查询方法

```
package com.mediamania.prototype;

import java.util.Collection;
import java.util.Iterator;
import javax.jdo.PersistenceManager;
import javax.jdo.Extent;
import javax.jdo.Query;

public class PrototypeQueries {
    public static Actor getActor(PersistenceManager pm, String actorName)
    {
❶       Extent actorExtent = pm.getExtent(Actor.class, true);
❷       Query query = pm.newQuery(actorExtent, "name == actorName");
❸       query.declareParameters("String actorName");
❹       Collection result = (Collection) query.execute(actorName);
        Iterator iter = result.iterator();
        Actor actor = null;
❺       if (iter.hasNext()) actor = (Actor)iter.next();
❻       query.close(result);
        return actor;
    }
    public static Movie getMovie(PersistenceManager pm, String movieTitle)
    {
        Extent movieExtent = pm.getExtent(Movie.class, true);
        Query query = pm.newQuery(movieExtent, "title == movieTitle");
        query.declareParameters("String movieTitle");
        Collection result = (Collection) query.execute(movieTitle);
        Iterator iter = result.iterator();
        Movie movie = null;
        if (iter.hasNext()) movie = (Movie)iter.next();
        query.close(result);
        return movie;
    }
}
```

首先请检查 getActor()。在行 ❶，我们取得指向 Actor 的 extent 的引用；行 ❷ 则使用 PersistenceManager接口中定义的 newQuery() 方法来创建 Query 的实例。程序会同时初始化此查询实例与 extent，并将查询过滤器应用至 extent。

过滤器中的name 标识符是Actor类中的name 字段，用来决定如何解释此标识符的命名空间则是根据用来初始化Query实例的 Extent 类。过滤器的表达式要求 Actor 的name 字段必须等于actorName。在过滤器中，我们可以直接使用 == 运算符来比较两个 String，而非使用 Java 的语法（name.equals(actorName)）。

在行 ❸ 声明的 actorName 标识符是一种“查询参数”（query paramter），它可以让你提供在执行查询时所用的参数值。我们选择使用相同的名称 —— actorName，来作为方法参数及查询参数。这种做法是非必要的，而且Java的方法参数与查询参数之间也没有直接的关联。行 ❹ 会将 getActor()的actorName参数当成 actorName 查询参数来执行查询。

执行 Query.execute()所产生的结果的类型被声明为 Object。在 JDO 1.0.1 中，返回的实例永远是 Collection，所以必须将查询结果类型转换成 Collection；在 JDO 1.0.1 中则声明为返回 Object，以便让未来的扩充版能返回 Collection 以外的值。接着程序会取得Iterator，并在行❺尝试访问元素，此处我们假设对应于所给予的名称只能有一个 Actor 实例。在返回结果之前，行 ❻ 会关闭查询结果，以释放任何相关的资源。如果此方法找到对应于输入名称的Actor实例，就会返回该实例；否则，如果查询结果没有任何元素，则返回 null 值。

修改实例

现在让我们检查两种修改数据存储空间中的实例的应用程序。在事务中，当应用程序从数据存储空间中取得实例后，它就可以修改实例的字段。当事务确认后，所有对实例做过的修改都会自动存入数据存储空间。

在例1-13中的UpdateWebSite应用程序是用来设定影片的网站。它会接受两个参数：第一个是电影的片名，第二个则是影片网站的URL。在初始化应用程序的实例后，会调用 executeTransaction()，而它又会调用定义在此类中的 execute() 方法。

行 ❶ 会调用 getMovie()（在例 1-12 中定义）来取得对应于指定片名的 Movie。如果getMovie()返回 null 值，应用程序会显示找不到符合该片名的Movie，并返回主程序；否则，在行 ❷ 调用 setWebSite()（为例 1-1 的 Movie 类所定义的），将 Movie 的webSite 字段设为参数值。当 executeTransaction()确认事务后，对 Movie 实例的修改会自动写入数据存储空间。

例 1-13：修改属性

```
package com.mediamania.prototype;

import com.mediamania.MediaManiaApp;

public class UpdateWebSite extends MediaManiaApp {
    private String  movieTitle;
    private String  newWebSite;

    public static void main (String[] args) {
        String title = args[0];
        String website = args[1];
        UpdateWebSite update = new UpdateWebSite(title, website);
        update.executeTransaction();
    }
    public UpdateWebSite(String title, String site) {
        movieTitle = title;
        newWebSite = site;
    }
    public void execute() {
❶       Movie movie = PrototypeQueries.getMovie(pm, movieTitle);
        if (movie == null) {
            System.err.print("Could not access movie with title of ");
            System.err.println(movieTitle);
            return;
        }
❷       movie.setWebSite(newWebSite);
    }
}
```

如例 1-13 所示，应用程序并不需要直接调用 JDO 接口来修改 Movie 的字段，它会访问实例并调用方法来修改网站的字段。此方法是使用标准的Java语法来修改字段。为了将数据写回数据存储空间，在确认事务前不需额外的程序，JDO 环境会自动将修改结果写入数据存储空间。此应用程序会对持久保存实例执行操作，但是并不直接导入或使用任何 JDO 接口。

现在来看看比较大型的应用程序 —— LoadRoles，此程序会示范一些JDO的功能。如例 1-14 所示，LoadRoles 会负责加载关于电影角色及扮演这些角色的演员的信息。LoadRoles 会取得一个参数，此参数指定要读取数据的文件名，而构造函数则会初始化BufferedReader来读取文件。它会读取文本文件，其中每一行数据表示一个角色，如下列的格式：

```
片名； 演员的名字 ； 角色名称
```

通常，某部影片的所有角色都会集中放于此文件中；LoadRoles 会执行少许的最佳化来判断目前所处理的角色信息是否与文件中前一笔角色信息属于同一部影片。

例 1-14：修改实例及依可达性来设定持续性

```
package com.mediamania.prototype;

import java.io.FileReader;
import java.io.BufferedReader;
import java.util.StringTokenizer;
import com.mediamania.MediaManiaApp;

public class LoadRoles extends MediaManiaApp {
    private BufferedReader  reader;

    public static void main(String[] args) {
        LoadRoles loadRoles = new LoadRoles(args[0]);
        loadRoles.executeTransaction();
    }
    public LoadRoles(String filename) {
        try {
            FileReader fr = new FileReader(filename);
            reader = new BufferedReader(fr);
        } catch(java.io.IOException e){
            System.err.print("Unable to open input file ");
            System.err.println(filename);
            System.exit(-1);
        }
    }
    public void execute() {
        String lastTitle = "";
        Movie movie = null;
        try {
            while (reader.ready()) {
                String line = reader.readLine();
                StringTokenizer tokenizer = new StringTokenizer(line, ";");
                String title     = tokenizer.nextToken();
                String actorName = tokenizer.nextToken();
                String roleName  = tokenizer.nextToken();
                if (!title.equals(lastTitle)) {
❶                   movie = PrototypeQueries.getMovie(pm, title);
                    if (movie == null) {
                        System.err.print("Movie title not found: ");
                        System.err.println(title);
                        continue;
                    }
                    lastTitle = title;
                }
❷               Actor actor = PrototypeQueries.getActor(pm, actorName);
                if (actor == null) {
❸                   actor = new Actor(actorName);
❹                   pm.makePersistent(actor);
                }
❺               Role role = new Role(roleName, actor, movie);
            }
```

```
        } catch (java.io.IOException e) {
            System.err.println("Exception reading input file");
            System.err.println(e);
            return;
        }
    }
}
```

execute()方法会读取文件中的所有内容。首先，它会检查新条目的片名是否与先前的条目的片名一样。如果不同，则行❶会调用getMovie()来访问新片名的Movie；如果在数据存储空间中并没有该片名的Movie，则应用程序会显示错误信息，并跳过该条目。在行❷，我们尝试以指定的名称来访问Actor实例。如果在数据存储空间中没有任何Actor拥有这个名称，就会建立新的Actor，并在行❸设定此名称，然后行❹将此实例设为能持久保存。

这时，我们已经读取输入文件，并在数据存储空间中查询文件中与名称相关联的实例。在行❺，我们会执行应用程序的实际任务，这里会创建新的Role实例。Role构造函数已经定义在例 1-3 中，此处再次列出以便更仔细地检查一遍：

```
    public Role(String name, Actor actor, Movie movie) {
❶       this.name = name;
❷       this.actor = actor;
❸       this.movie = movie;
❹       actor.addRole(this);
❺       movie.addRole(this);
    }
```

行❶会初始化Role的name。行❷会建立指向相关Actor的引用，而行❸则会建立指向相关Movie实例的引用。Actor与Role之间的关系，以及Movie与Role之间的关系是双向的，因此也需要更新所有关系的另一端。在行❹，我们会对actor调用addRole()，将Role加到Actor类中的roles集合中；同样地，行❺对movie调用addRole()，将Role加到Movie类中的cast集合中。将Role当成元素加到Actor.roles及Movie.cast会修改actor及movie所引用的实例。

在Role构造函数中，只需初始化指向实例的引用，就可建立与一个实例的关系，而将引用加入集合中，即可建立与一个以上的实例的关系。此过程是在Java中表达关系的方式，而且JDO也直接支持它。在确认事务后，在内存中建立的关系会保存在数据存储空间中。

从Role构造函数返回后，execute()会处理文件中的下一个条目。当处理完文件内容时，while循环就会结束。

你可能已经注意到我们从未对已创建好的Role实例调用makePersistent()。不过，由

于JDO支持“依可达性的持久保存”(persistence-by-reachability)，因此在确认事务时，仍会将Role实例存入数据存储空间。如果持久保存实例可以（直接或间接）访问持久保存类的瞬时（非持续的）实例，则在确认事务时，依可达性的持久保存规则也会让此瞬时实例能持久保存。你可以经由引用或引用的集合来获得实例。可从某一实例获得的所有实例组合成的对象图，被称为该实例的“相关实例完全闭包”(complete closure)。这种可达性算法适用于所有持久保存实例，并适用于内存中其所有的引用，因而让完全闭包也能持久保存。

删除持久保存实例的所有引用并不会自动删除实例，你需要明确地删除实例，这部分会在下节谈到。如果在事务过程中，从持久保存实例建立了指向瞬时实例的引用，但你改变了此引用，并且在确认事务时并没有持久保存实例会引用到瞬时实例，则此引用仍然是瞬时的。

依可达性的持久保存规则可让你不需直接调用JDO接口来存储实例。你可以将大部分的程序专注于建立内存中实例间的关系，而JDO实现则负责存储任何新实例以及在内存中的实例间所建立的新关系。你的应用程序可以在内存中构造出十分复杂的对象图，而且只需从持久保存实例建立指向对象图的引用就可以让它们能持久保存了。

删除实例

现在让我们检查会从数据存储空间中删除一些实例的应用程序。在例1-15中，DeleteMovie应用程序用来删除Movie实例，欲删除的电影片名会被当作此程序的参数。行❶会存取Movie实例，如果没有符合该片名的电影，应用程序就会显示出错误信息并返回主程序。行❻则调用deletePersistent()来删除该Movie实例。

例1-15：从数据存储空间中删除Movie实例

```
package com.mediamania.prototype;

import java.util.Collection;
import java.util.Set;
import java.util.Iterator;
import javax.jdo.PersistenceManager;
import com.mediamania.MediaManiaApp;

public class DeleteMovie extends MediaManiaApp {
    private String  movieTitle;

    public static void main(String[] args) {
        String title = args[0];
        DeleteMovie deleteMovie = new DeleteMovie(title);
        deleteMovie.executeTransaction();
    }
    public DeleteMovie(String title) {
```

```
        movieTitle = title;
    }
    public void execute() {
❶      Movie movie = PrototypeQueries.getMovie(pm, movieTitle);
        if (movie == null) {
            System.err.print("Could not access movie with title of ");
            System.err.println(movieTitle);
            return;
        }
❷      Set cast = movie.getCast();
        Iterator iter = cast.iterator();
        while (iter.hasNext()) {
            Role role = (Role) iter.next();
❸          Actor actor = role.getActor();
❹          actor.removeRole(role);
        }
❺              pm.deletePersistentAll(cast);
❻              pm.deletePersistent(movie);
    }
}
```

不过，我们也需要删除与Movie相关的Role实例；此外，因为Actor包含指向Role实例的引用，因此也需要删除此引用。在行❷，我们会访问与Movie相关的Role实例的集合。然后，在行❸迭代Role并访问与其相关联的Actor。因为正要删除Role实例，所以在行❹会删除actor指向Role的引用。在行❺，我们会调用deletePersistentAll()来删除电影演员表中的所有Role实例。当确认事务时，程序会从数据存储空间中删除Movie实例与相关的Role实例，并更新与Movie相关的Actor实例，因此它们不会再引用到被删除的Role实例。

你必须明确地调用这些deletePersistent()方法，才能从数据存储空间中删除实例。这些方法并不是使用依可达性的持久保存算法的makePersistent()的相反面。此外，并没有与Java资源回收对等的JDO数据存储空间能够在实例不再被数据存储空间中的任何实例引用时自动删除这些实例。实现与持久保存的资源回收相同的机制是非常复杂的工作，而且通常这种系统的效能都很差。

小结

如前所述，利用传统的Java模型、语法及程序技巧，大部分的应用程序可以用完全与JDO无关的方式来编写，只需采用Java对象模型就可以定义应用程序的持久保存信息模型。当经由extent或查询来访问数据存储空间中的实例时，你所使用的程序看起来与其他Java程序访问内存中的实例的方法是没有差别的。你不需要学习其他的数据模型或像SQL那样的数据访问语言，你也不需要推敲在数据库与内存中的对象之间如何提供数据的对应关系，相对地，你可以充分利用Java面向对象的功能，且没有任何限制。这包括

使用继承与多态，而利用类似JDBC及EJB结构的技术是无法达成的。此外，利用对象模型来开发应用程序所需的软件会远少于利用其他结构来开发的程序。单纯的、一般的Java对象可以存入数据存储空间，并以透明的方式来被访问。JDO提供非常容易学习及有生产力的环境，来编译管理持久保存数据的Java应用程序。