

第十章

扩展 JPetStore

前一章介绍了 Simple Spider 服务以及它的控制台用户接口与 Web 服务端点。在这一章中，我们会看到将 Spider 加入 JPetStore 这已经写好的应用程序中有多简单。有些人可能会质疑 JPetStore 本来就已经有搜索工具了；但该工具只搜索数据库中的动物，而不搜索网站上的所有网页。我们的客户需要搜索整个网站；JPetStore 目前的版本至少有一页是完全无法搜索到的（Help 页），并且描述不同动物的文字无法显示在查询中。

我们会将 Spider 加到 JPetStore 中，专注在代码需要修改以能够集成的部分。此外，我们会将现有的持久化框架换成 Hibernate。通过谨慎地追随核心原则，我们的代码能够重用，并且因为 JPetStore 是建立在轻量化的框架上（Spring），它不会无理地对代码作要求以引入搜索功能或新的持久层。经过几个来回，最后将会是简单的且几乎完全透明的。

现有搜索功能的概观

JPetStore 现有的搜索功能采用一个或多个以空格隔开的关键字，并返回一组动物，这组动物的名称或种类带有该关键字。对“dog”进行搜索会找到六条结果，而搜索“snake”会找到一条。然而，搜索“venomless”找不到结果，即使 EST-11 这一条的名称是“Venomless Rattlesnake”。更糟的是，其他的页（像是 Help）都不会出现在搜索结果中；后续可能会加入的网页也一样不会，除非它在数据库中有动物的数据。

搜索功能有着下列的结构（见图 10-1）：

1. JPetStore 应用程序的网页都带有输入搜索条件的文字方块与 Search（搜索）按钮。

- 2. 按下按钮会发出请求（对 `/shop/searchProducts.do`）并将搜索关键字传入请求中。
- 3. `petstore-servlet.xml` 这个 MVC 部分的配置文件有以下的定义：

```
<bean name="/shop/searchProducts.do"
class="org.springframework.samples.JPetStore.web.spring.SearchProductsController">
  <property name="petStore"><ref bean="petStore"/></property>
</bean>
```

这会对 “`/shop/searchProducts.do`” 产生请求（request）处理过程并将它映射到 `SearchProductController`的实例上，同时传入称为`petStore`的`petStoreImpl`实例。

- 4. `SearchProductsController`将实例化一个实现`ProductsDao`接口的类的实例，要求它搜索数据库中所指定的关键字。
- 5. `ProductsDao` 查询数据库并对每个返回的数据行创建一个 `Product` 的实例。
- 6. `ProductDao` 传回一个包含了所有 `Product` 实例的 `HashMap` 到 `SearchProductsController`。
- 7. `SearchProductsController`创建新的 `ModelAndView`，传入显示结果的JSP名称（`SearchProducts`）以及带值的 `HashMap`。此 JSP 网页接着会使用 `PagedListHolder` 这个 control（有内置换页功能的 `list/table`）来展示出结果。

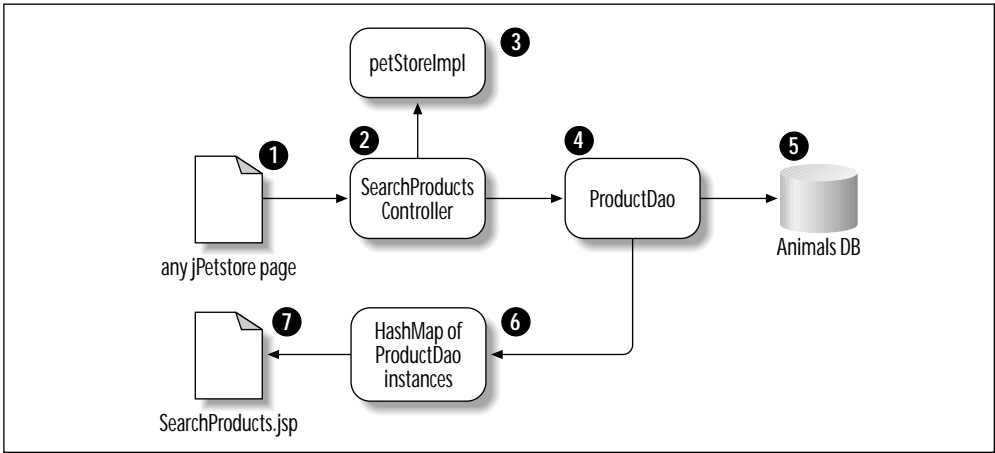


图 10-1：最初的 JPetStore 搜索架构

只有 `ProductsDao` 知道如何与底层数据互动。`Product` 是个带有每项产品信息的类，而视图（`SearchProducts.jsp`）将返回结果，并逐个取出以创建输出页。

对 Spider 的决定

我们已经鉴别过现有的搜索功能如何运作并发现它的限制：搜索功能只会搜索数据库中的产品，而不是整个网站，并且它也不会搜索所有的产品数据。它所返回的结果是极度受限的——虽然是良格式化的。

Simple Spider是个爬站形式的搜索功能，而不是关注于数据库来搜索的：它搜索整个网站，不只是产品的数据库表，并且它视任何用户可见到的文本数据为搜索范围的一部分。Spider 有个主要的限制——因为它基于爬站过程的结果，它只能委托链接来找到其他网页。如果某个网页仅能通过某种服务器端的逻辑来访问（例如，从下拉菜单选取产品然后送出要求给服务器产生一个客户端或服务端的网页重新导向），那么爬站程序决不会碰到此网页，并且它也不会被搜索到。

像这种应用程序功能受限的问题，我们得决定要改善现有的服务或整个换掉。JPetStore的搜索限制部分是因为这个服务的基础本来就是这样（它搜索数据库而不是网站）。将它重新改造成有全站搜索功能是很没效率的。Spider很明显的是可行的方案，但我们必须考虑到现有的部分（要记得吃什么就像什么）。如果JPetStore使用一大堆的服务器端逻辑来处理导航，则Spider就是无法提供完整的搜索。在这个例子中，所有的网站导航都是由客户端所处理的，因此Spider非常适合对付这个问题，并且能够与我们现有的应用程序并存。

扩展 JPetStore

我们已经判定此应用程序现有的服务层不适合目前的需求。此外，我们已经决定要将此服务换成合适的解决方案。这项替换刚好可以来测试扩展：换掉服务有多容易？这会涉及到编写新程序吗？或只要更换配置服务就好？

为了要用 Simple Spider 来换掉现有的功能，我们需要将输出格式作一点点修改（返回结果会用完整的 URL 来取代产品的实例）。编写新的控件来启动 Simple Spider 以淘汰 ProductsDao对象，并修改我们的映射层以指向新的控件。最后，我们会使用Spider的配置服务，以让 Spider 能够在新网站上运行得更好。

观察这些需求，我们已经可以知道我们仅需要编写少于 100 行的程序，并做些细微的配置变动来让它运行。这样付出的代价对所取得的成果来说是合理的。因为 JPetStore 与 Simple Spider 是一开始就被设计成容许扩展的，所以它们两者在最小的花费之下就可以合作得很好。

相对而言，如果选择通过Web服务接口来连接Spider而不是将它直接集成进JPetStore的话，我们也可以编写更少的代码，并且实际上几乎不用做任何事情。因为这个Web服

务已经存在，它就可以违反“每次做好一件事”的法则来构建，以便加入似乎是没什么用处的接口。虽然如此，在本例中在慢速管道机制上（HTTP）传递膨胀消息（XML/SOAP）的成本还是太大了，特别是在只要做一点点的工作就可以取得更快更有效率的集成时更是如此。

替换控制器

首先，让我们换掉 SearchProductsController。下面是该类的主要方法：

```
public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse
                                response) throws Exception {
    if (request.getParameter("search") != null) {
        String keyword = request.getParameter("keyword");
        if (keyword == null || keyword.length() == 0) {
            return new ModelAndView("Error", "message",
                "Please enter a keyword to search for,
                then press the search button.");
        }
        else {
            PagedListHolder productList = new PagedListHolder(
                this.petStore.searchProductList(keyword.toLowerCase()));
            productList.setPageSize(4);
            request.getSession().setAttribute(
                "SearchProductsController_productList", productList);
            return new ModelAndView("SearchProducts", "productList",
                productList);
        }
    }
    else {
        String page = request.getParameter("page");
        PagedListHolder productList = (PagedListHolder)
            request.getSession().getAttribute("SearchProductsController_productList");

        if ("next".equals(page)) {
            productList.nextPage();
        }
        else if ("previous".equals(page)) {
            productList.previousPage();
        }
        return new ModelAndView("SearchProducts", "productList", productList);
    }
}
```

这个方法 会返回 ModelAndView 的新实例，并且 Spring 会用它判别要装载哪个 JSP，以及如何与数据连接上。它会取用 HttpServletRequest 与 HttpServletResponse，以便直接与 HTTP 消息互动。

这个方法所做的第一件事是确定用户输入了搜索条件。如果没有,它会向用户显示出错;如果有,它会创建称为 `productList`、页大小上限(每页行数)设为4的 `PagedListHolder`。最后,它会调用 `petStore` 实例的 `searchProductList` 方法来调用 `ProductsDao`,并于最后返回 `Product` 实例的 `HashMap`。第二个子句是用于用户点击清单的下一页或上一页按钮之时。

改写或换掉?

细心的程序员会问这样一个问题:改写这个类来运用 `Spider` 或写出新的控件,哪一样是比较合理的?为了回答这个问题,我们必须先考虑到三个更明确的问题:

1. 我们能取得源代码吗?现在我们有 `JPetStore` 应用程序,但我们有源代码或者只是可执行文件吗?如果我们没有源代码,则可以直接得出结论,我们只能换掉类;无法将它改写。
2. 我们是否会再次需要用到原来的服务?假设我们有源代码并且可以改写这个类,我们是否能够预见有必要回头或再利用到数据库搜索的功能?为了灵活性,我们通常会需要保持旧功能不变,这代表我们必须替换,而不是改写。但是.....
3. 现有的类实现是很容易重用的接口吗?如果我们要替换这个类,要花多大的力气才能让应用程序认识并接受新的类?把这件事情想像成器官移植,要做多少事与用多少药才能让患者不会排斥新器官?新类的修改是局部的还是全局的?

以下是这些问题的答案:是的,我们有源代码;是的,我们想要维持旧的服务以便能够再次使用;是的,控制器实现的是一个非常简单的接口。此控件只需要实现 `handleRequest` 这个方法,它接受 `HttpServletRequest` 与 `HttpServletResponse` 参数并返回 `ModelAndView`。这意味着 `JPetStore` 应用程序不需要做任何全系统的修改才能使用新的控制器,只要我们支持此接口即可。

实现接口

要取代这个类,我们必须写出称为 `SearchPagesController` 的控制器类。它必须实现定义 `handleRequest` 这个方法的 `Controller` 接口。

```
public class SearchPagesController implements 控件{  
    ...  
}
```

下面是我们的控制器的 `handleRequest` 方法:

```

public ModelAndView handleRequest(HttpServletRequest request,
                                HttpServletResponse response) throws Exception {
    if (request.getParameter("search") != null) {
        String keyword = request.getParameter("keyword");
        if (keyword == null || keyword.length() == 0) {
            return new ModelAndView("Error", "message", "Please enter a
                keyword to search for, then press the search button.");
        }
        else {
            ConfigBean cfg = new ConfigBean();
            String indxpath = "";
            try
            {
                indxpath = cfg.getCurIndexPath();
            }
            catch(Exception ex)
            {
                return new ModelAndView("Error", "message",
                    "Could not find current index path.");
            }

            QueryBean qb = new QueryBean(indxpath, keyword, "contents");

            qb.execute();

            HashMap hits = new HashMap(qb.getResults().length);
            for(int i =0;i<qb.getResults().length;i++)
            {
                hits.put("hits", qb.getResults()[i]);
            }
            return new ModelAndView("SearchProducts", hits);
        }
    }
}

```

对我们的搜索功能来说,我们不会使用分页后结果。我们只需以一页来列出所有的结果;因此,我们无需处理上/下页的代码。我们的控件还要检查 null 关键字并返回错误。此服务其余的部分几乎是与第九章的控制台应用程序一模一样。首先创建 ConfigBean 的实例以找出此网站的最新索引,然后根据索引路径创建 QueryBean。最后,执行此查询并将所有的 HitBean 放入一个 HashMap 中返回给视图。

如此的使用方式是与上一章的一样;唯一的不同之处在于返回数据的格式。相对于传回 HitBean 的原始数组,ModelAndView 对象需要 HashMap。创建它是很容易的,并且我们现在有一个几乎不花力气、全新的 Spider 应用程序访问点。

最后还有一项细节需要关注。原始的 SearchProductsController 有个称为 petStore 的 PetStoreFacade 类型字段,它是 Spring 框架所产出的。为了要完全取代原来的,我们的新控制器需要显露出相同的属性与 accessor 方法,虽然它们并没有在应用程序的任何独立接口中出现过。当你要扩充或修改应用程序的时候经常会出现这样的情况。

```
private PetStoreFacade petStore;

public void setPetStore(PetStoreFacade petStore) {
    this.petStore = petStore;
}
```

用于 JPetStore 注册新的类

最后,我们通知JPetStore有新的控制器存在。如果JPetStore没有写成可扩充的,我们必须修改代码以让它能用。举例来说,如果有JPetStore的方法会直接创建SearchProductsController的实例,我们必须逐行修改代码以创建SearchPagesController。

然而,JPetStore已经准备好要扩充了——部分是因为它是基于Spring框架的。为了要告诉JPetStore有新的控制器,我们只需修改一个配置文件(*petstore-servlets.xml*)。这个文件会告诉Spring要创建哪些对象以及如何将它们连接以组成应用程序。现在只需找出用来启用SearchProductsController的配置并将它改指向新的SearchPagesController。

```
<bean name="/shop/searchProducts.do"
      class="org.springframework.samples.Jpetstore.web.spring.SearchPagesController">
  <property name="petStore"><ref bean="petStore"/></property>
</bean>
```

我们告诉此应用程序将“/shop/searchProducts.do”的要求映射到SearchPagesController的实例上。我们同时告诉它以SearchPagesController供应给petStore的当前实例(在称为petStore的属性中)。

原则的应用

- 保持简单:controller的逻辑是Spider的简单调用;控制器接口非常简单(只有一个方法)
- 选择正确的工具:Spring与Spider
- 一次做好一件事:因为Spider被包装得很好,它可以很容易地加入到现存的服务中;controller处理Spider的调用,并且JSP只负责显示结果——对MVC样式的绝佳示范
- 力求透明:网站不在乎是如何编制索引的;它可以很容易的转换数据导向或网页导向的搜索技术
- 允许扩展:我们以最少的代码加入搜索功能来扩展;JPetStore的配置能力可以在不改代码的情形下认知新的服务

用户界面 (JSP)

用户界面是相当直接的。相对于直接把结果列到控制台上或创建存放结果的 XML 文档 (例如, 第九章的 Web Service 实现), 这一次我们要写出用表将结果超链接迭代列出的 JSP。

原始的 JPetStore 使用 PagedListHolder 来展示搜索功能, 因为它可以用表来显示有图的产品。由于图文件的大小变化很大, JPetStore 不打算在同一页显示太多的产品以至于网页太长。我们的结果是由返回的 URL 超链接以及相关等级所组成的; 所以我们使用简单的表来展示搜索结果。

再一次, 我们面对了改写或替换的问题。跟上一次一样, 我们有三问题要考虑:

1. 我们可以获得源代码吗? 一定可以, 因为 JSP 在开发与部署模式下都只是个文本文件而已。
2. 我们会重用现有的服务吗? 会, 但在此例中 JSP 因为太容易重建所以没什么差别。
3. 现有的版本实现某些标准接口吗? 没有, 因为 JSP 只是静态 HTML 与动态内容的混合而已。

因为修改相当小且因为 JSP 很容易直接改 (无需编译), 我们只需要修改现存的 *SearchProducts.jsp* 文件就好。这个策略可以让我们不必修改配置设置:

```
<%@ include file="IncludeTop.jsp" %>

<table align="left" bgcolor="#008800" border="0" cellspacing="2" cellpadding="2">
<tr>
  <td bgcolor="#FFFF88">
    <a href="<c:url value="/shop/index.do"/>">
      <b><font color="BLACK" size="2"> &lt;&lt; Main Menu</font></b>
    </a>
  </td>
</tr>
</table>
<table align="center" bgcolor="#008800" border="0" cellspacing="2"
  cellpadding="3">

  <tr bgcolor="#CCCCCC"> <td><b>URL</b></td> <td><b>Rank</b></td> </tr>
  <c:forEach var="page" items="{hits}">
    <tr bgcolor="#FFFF88">
      <td><a href="<c:out value="{page.url}"/>">
        <c:out value="{page.url}"/></a>
      </td>
      <td>
        <c:out value="{page.score}"/>
      </td>
    </tr>
  </c:forEach>
</table>
```



```
        </tr>
    </c:forEach>
</table>

<%@ include file="IncludeBottom.jsp" %>
```

此应用程序的 JSP 文件有在 *IncludeTop.jsp* 与 *IncludeBottom.jsp* 中定义的标准文件头与文件尾。我们只要作出包含指令间的结果就好。以创建 JSP 样式的 `forEach` 循环开始，并用称为 `page` 的枚举成员（enumerator）来指向称为 `hits` 的 `HashMap` 中的每个成员。对每个结果项目，我们将它作成带有 URL（文本标题与指向网页的 `HREF`）和相关等级的表的行。JSP 使用反射处理变量与属性的映射。然而在实现此页时，因为第一个原因（唯一的一个）而要对原始的 `Spider` 作修改。

修改源代码以符合 JSP

JSP 将字段映射到 `<out>` 这个标记上，而不是使用 `getter` 与 `setter`。不幸的是，我们原始的 `HitBean` 实现将所有的数据标识为私有的，并且只显露出 `getter` 与 `setter`（通常这是很合适的策略）。因为我们现在必须要有直接显露的字段，我们需要对 `Spider` 做些修改。原始的类以下面这些声明开始：

```
final String url;
final String title;
final float score;
```

它变成：

```
public final String url;
public final String title;
public final float score;
```

如果没有 `Spider` 的源文件会如何？

想想当我们不是要扩充的应用程序（`JPetStore`）或要集成的服务（`Simple Spider`）的作者时会发生什么事，这是很有启发性的。如果我们无法取得任一项目的源代码，我们还是可以进行想要的扩展。对 `JPetStore` 来说，要做的只有修改配置文件与 JSP（永远都会有源代码），并加入新的类。

如果我们无法取得 `HitBean` 这个类的源代码，要怎样才能让它与 JSP 协作呢？答案很简单：写出一个包装类来显露出正确的属性（或使用已经出现过的 Web 服务接口）：

```
public class HitBeanWrapper {
    private HitBean _hitbean;
    public String url;
```

```
public String title;
public float score;

public HitBeanWrapper(HitBean hitbean)
{
    _hitbean = hitbean;
    url = hitbean.getUrl();
    title = hitbean.getTitle();
    score = hitbean.getScore();
}

public String getScoreAsString() {
    return _hitbean.getScoreAsString();
}
}
```

这也需要改变 SearchPagesController 的 handleRequest 方法：

```
HashMap hits = new HashMap(qb.getResults().length);
for(int i =0;i<qb.getResults().length;i++)
{
    hits.put("hits", new HitBeanWrapper(qb.getResults()[i]));
}
return new ModelAndView("SearchProducts", hits);
```

就这样。我们修改过 Spider 所需的部分，以让它能合并到 JPetStore 应用程序中。

法则的应用

- 保持简单：以显示 URL 页来取代展示产品信息；使用简单的表输出替换 PagedListHolder（需要它的理由已经消失）
- 选择正确的工具：使用表而不是 PagedListHolder；JSP
- 一次做好一件事：JSP 专注于输出的显示而不牵涉搜索
- 力求透明：HitBean 显示简单的数据属性；如果无法取得源代码时使用 HitBean 的包装器
- 允许扩展：无

设置索引程序

现在搜索服务已经集成到应用程序中，我们会设定索引程序自动以固定的周期更新网站。如果你还记得前一章，控制台与 Web 服务两者都有机制可让你启动编制索引的服务，而不是执行搜索。问题是，索引程序要怎样集成到 JPetStore 中？

嵌入 JPetStore 中或由外部启动？

第一种方法是让索引程序成为 JPetStore 应用程序自身的一部分：换言之，将代码加入到 JPetStore 中以启动索引程序。JPetStore 可以于受到用户请求或根据调度来启动。这两种方法都有问题：如果我们显示用户接口来启动索引程序，我们就必须要用某种安全性的分隔来只让管理员用户操作。目前 JPetStore 并没有内置这样的机制。建立安全机制来包装索引程序似乎会是个大问题——太复杂又没有什么好处。这代表手动的访问点出局。

另外一个选项是在 JPetStore 应用程序中创建调度程序。不管是哪一种架构，调度程序需要 JPetStore 保持执行状态才能启动索引。因为 JPetStore 是个需要网页服务器与容器的应用程序，它的生命周期完全要看外在的宿主而定。如果为了某些原因需要而关闭网页服务器，JPetStore 也会跟着关闭。如果索引程序刚好在这个时段中，它就不会运行。此外，编写调度代码已经完全脱离了 JPetStore 的主题，这问题就跟在 Simple Spider 中是一样的。JPetStore 应用程序应该只做一件事：在 Web 目录中显示动物。

除了在别处唤起索引程序外别无它法。使用现有的调度系统是个好方法：在 Windows 上用 *schtasks*，而在 Linux 上用 *cron*。让我们在 Windows 上实现索引程序的调度。

使用系统调度程序

为了方便使用，我们创建出批处理文件来执行服务的启动。我们要调用 Java 运行时来运行 ConsoleSearch 这个类的 main 方法，传入 JPetStore 的起点。此命令（也就是批处理文件的内容）看起来如下所示：

```
java c:\the\path\to\ConsoleSearch /i:http://localhost/JPetStore
```

我们将它存入命名为 *JPetStoreIndexer.bat* 的批处理文件。为了简化，我们把它存储在 *c:\commands*。

要让索引程序的调度于每天凌晨 2:00 执行，下达这样的命令（以本地管理员身份登录）：

```
c>schtasks /create /tn "JPetStore Indexer" /tr:c:\commands\JPetStoreIndexer.bat  
/sc daily /st 02:00:00
```

/tn 标记为文本创建独一无二的名称；/tr 指向实际执行的命令；/sc 是时间间隔；而 /st 指定以那个间隔启动索引程序的时间。

相类似的，在 Linux 上修改 *crontab* 文件并启动 *cron daemon* 来完成同样的工作。

欣赏成果

这个解决方案的美感在于此应用程序被重新设定，以同时在容器环境（Spring）与直接调用环境中（通过调度程序直接调用Java的运行时）执行而不用任何额外的代码。因为它的简单架构与松散耦合的服务，Spider本身可以同时两个环境中执行得很好。我们不必编写新的访问点或新的UI，甚至是改变配置。更棒的是我们能够将前面章节中的应用程序在不会花费太多力气的前提下重新塑造它的内部服务。偶尔回头欣赏一下成果是很不错的，这样才能了解基本法则带来了什么样的好处。

法则的应用

- 保持简单：使用系统提供的调度程序与现有的控制台程序
- 选择正确的工具：*schtasks*、*cron*、*ConsoleSearch*
- 一次做好一件事：Spider与JPetStore都不用担心索引程序的调度；调度只关心索引而不管其他的功能
- 力求透明：调度程序不知道索引程序的实现细节与索引编制的结果：那都在配置文件中处理
- 允许扩展：无

运用配置服务

如果直接开始以当前的配置使用搜索服务，我们会注意到一个问题。我们的搜索返回一大堆结果——比数据库中的产品数量还多。事实上，搜索“dog”会返回20条以上的记录，然而数据库中只有六只狗。

这是由于爬站服务的特点。在没有额外辅助下，爬站程序会找出网页上的每个链接并跟随下去，把结果加入索引中。问题来自于除了可以让用户浏览目录中的动物之外，还有链接可以让用户把动物加到他的购物车中、从购物车中移除项目的链接、签入网站的链接（默认情形下，JPetStore会装载储存于文本框中的身份证明）以及登录的链接，这些都会让爬站程序跟进——因此产生有session ID的全新链接。

我们必须设法确保爬站程序不会跟进所有额外的链接，并产生对用户没有用途的结果。在第九章的前面几节我们曾提到在天真的爬站方式中会遇到的三项问题：

无限循环

一旦碰到已经跟进过的链接，爬站程序应该将它忽略。

外部跳转

因为我们寻找的是 `http://localhost/JPetStore` ,我们不打算要编制外部资源链接的索引 :这会让我们将整个网际网络作索引 (或几个小时后用光内存 ,而导致应用程序崩溃)。

不应被编制索引的页面

在这种情况下 ,那些像登录、带有 session ID 等的网页应该要排除。

我们的爬虫 / 索引服务会自动处理前两项议题。让我们回头看看代码。IndexLinks 这个类在考虑新的链接时会参考三个集合 :

```
Set linksAlreadyFollowed = new HashSet();
HashSet linkPrefixesToFollow = new HashSet();
HashSet linkPrefixesToAvoid = new HashSet();
```

每回跟进一个链接时 ,它就会被加入到 linksAlreadyFollowed。爬虫程序决不会再次访问已经储存于此的链接。另外两个集合放的是允许与拒绝的链接前缀字符串。当我们调用 IndexLinks.setInitialLink 时 ,我们会把根路径加入到 linkPrefixesToFollow 这个集 (set) 中 :

```
linkPrefixesToFollow.add(new URL(initialLink));
```

IndexLinks 也显露出了 initAvoidPrefixesFromSystemProperties ,它会告诉 IndexLinks 这个 bean 去读取配置系统属性 ,以初始化此集合的内容 :

```
public void initAvoidPrefixesFromSystemProperties()
    throws MalformedURLException {
    String avoidPrefixes = System.getProperty("com.relevance.ss.AvoidLinks");
    if (avoidPrefixes == null || avoidPrefixes.length() == 0) return;
    String[] prefixes = avoidPrefixes.split(" ");
    if (prefixes != null && prefixes.length != 0) {
        setAvoidPrefixes(prefixes);
    }
}
```

首先 ,这个逻辑检查链接确定符合 linkPrefixesToFollow 中的前缀字符串。对我们而言 ,唯一储存于此的值为 `http://localhost/JPetStore`。如果它是这个下面的网页 ,我们还要确定这个链接不符合 linkPrefixesToAvoid 中的前缀字符串。

特别说明 :好的文件说明是可维护与灵活性的重要一环。注意 ,Simple Spider的代码相当缺乏注释。另外一方面 ,它的方法与类型名称相当的长 (例如 initAvoidPrefixesFromSystemProperties) ,这会让注解变得多余 ,因为名称就能很清楚地描述。严谨的注释原则之外 ,好的名称通常是程序可读性的关键。

我们要做的是产出linkPrefixesToAvoid这个集合。ConsoleSearch已经调用了initAvoidPrefixesFromSystemProperties，所以我们只要加入必要的值到com.relevance.ss.properties文件中就行：

```
AvoidLinks=http://localhost:8080/ JpetStore /shop/signonForm.do
http://localhost:8080/ JPetStore /shop/viewCart.do
http://localhost:8080/ JPetStore /shop/searchProducts.do
http://localhost:8080/ JPetStore /shop/viewCategory.do;jsessionId=
http://localhost:8080/ JPetStore /shop/addItemToCart.do
http://localhost:8080/ JPetStore /shop/removeItemFromCart.do
```

这些前缀字符串代表应用程序的登入表单、显示购物车的链接、搜索结果、显示登录成功的页、加入购物车的页与移除购物车项目的页。

法则的应用

- 保持简单：使用现有的 Properties 工具而不是 XML
- 选择正确的工具：java.util.Properties
- 一次做好一件事：服务负责跟进所提供的链接；配置文件负责链接是否要跟进的决策
- 力求透明：服务并未预先知道要接受的链接类型；配置文件让此决策对服务透明
- 允许扩展：可扩展的允许链接类型列表

加入 Hibernate

JPetStore使用相对直接的架构来提供数据库访问。有一个接口层来提供DAO的映射而不用担心实际实现的细节。DAO则是依据后端数据库而定；我们将会看到定向HSQLDB（Hypersonic SQL）的示例。

现有架构

让我们看一下如何管理Product这个类。Product是个代表目录中一个项的领域对象。

```
package org.springframework.samples. JPetStore.domain;
import java.io.Serializable;
public class Product implements Serializable {

    private String productId;
    private String categoryId;
    private String name;
    private String description;
```

```

    public String getProductId() { return productId; }
    public void setProductId(String productId) { this.productId = productId.trim(); }

    public String getCategoryId() { return categoryId; }
    public void setCategoryId(String categoryId) { this.categoryId = categoryId; }

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

    public String getDescription() { return description; }
    public void setDescription(String description) { this.description = description; }

    public String toString() {
        return getName();
    }
}

```

它的持久化是通过实现 `ProductDao` 这个接口的对象来管理的。`ProductDao` 必须能够依据 ID 装载特定产品，或装载一系列来自某类别或关键字的产品。

```

public interface ProductDao {

    List getProductListByCategory(String categoryId) throws DataAccessException;
    List searchProductList(String keywords) throws DataAccessException;
    Product getProduct(String productId) throws DataAccessException;

}

```

目前有个称为 `SqlMapProductDao` 的类，它将通过 SQL 映射文件寻找在 Hypersonic SQL 中的产品信息。

现有域对象的 Hibernate 映射

要用 Hibernate 取代这个架构，我们首先要创建定义域对象与数据库之间关系的映射文件。再看一下 `Product`，我们会创建出下面这个被称为 *Product.hbm.xml* 的映射文件：

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping
    package="org.springframework.samples.JPetStore.domain">

    <class name="Product" table="product">
        <id name="productId"
            column="productId"
            type="string">
            <generator class="native"/>
        </id>
    </class>

```

```
<property name="categoryId" column="category" type="string"/>
<property name="name" column="name" type="string"/>
<property name="description" column="description" type="string"/>
</class>

</hibernate-mapping>
```

在这个映射文件中，我们先识别出要映射的包与特定的类（`org.springframework.samples.JPetStore.domain.Product`）。我们必须告诉它要映射哪个表（本例中为“product”），然后把域对象的个别属性映射到该表中的列。这个文件需要保存到类路径下的某处，我们会创建项目结构称为“hibernate”的新目录来保存映射文件与新的DAO。

Hibernate 的 DAO

下个步骤是创建以 Hibernate 作为持久层并代替原版中所用的 SQL 映射的 DAO。新的 DAO 需要实现 `ProductDao` 接口，就像原来的 DAO 一样。然而，接口的实现将完全不同。

下面是新 DAO 的代码：

```
public class HibernateProductDao implements ProductDao {
    SessionFactory factory;
    Configuration cfg;

    public HibernateProductDao() {
        try {
            cfg = new Configuration().addClass(
                org.springframework.samples.JPetStore.domain.Product.class);
            factory = cfg.buildSessionFactory();
        } catch (Exception ex) {
            System.out.println("Hibernate configuration failed: " + ex);
        }
    }

    public List getProductListByCategory(String categoryId)
        throws DataAccessException {
        List results = null;
        try {
            Session session = factory.openSession();
            results = session.find("from product where product.category = ?",
                categoryId, Hibernate.STRING);

            session.close();
        } catch (Exception ex) {
            System.out.println("Failed to connect to database:" + ex);
        }
        return results;
    }
}
```



```
public List searchProductList(String keywords) throws DataAccessException {
    return null;
}

public Product getProduct(String productId) throws DataAccessException {
    Product p = null;
    try {
        Session session = factory.openSession();
        p = (Product)session.load(Product.class, productId);
        session.close();
    } catch (Exception ex) {
        System.out.println("failed to connect to database: " + ex);
        p = null;
    }

    return p;
}
```

首先，我们需要一种方法来与 Hibernate 进行交互。如同第七章所示，我们需要创建 Hibernate 的 `SessionFactory` 并使用它来获得与数据库交互的 `Session`。此 DAO 的构造函数会实例化新的 Hibernate 配置、从类的路径基于加到配置中的类名称来装载映射文件。之后它会从 `Configuration` 取得 `SessionFactory`。

每个方法都使用 `SessionFactory` 来对数据库开启新的 `Session`。`getProduct` 方法是最直接的；首先取得 `Session`。然后要求它根据指定的 `productId` 装载 `Product` 的实例。注意，调用 `session.load()` 的结果是 `Object` 类型的，必须要转换成 `Product`。最后，我们关闭 `Session`。Hibernate 处理所有的 SQL 命令、查询映射文件、将 `productId` 映射到 `table` 中的正确列上、产出所有的属性以及其他事情。

`getProductListByCategory()` 方法相对没有那么直接；它取用 `categoryId` 并返回所有符合该类别的产品。在此例中，我们不能依靠内置生成的 SQL；必须自行建立查询。我们先从 `SessionFactory` 获得 `Session`，然后使用 `session.find()` 来返回 `Object` 的 `List`。此方法在本例中需要三个参数：HSQL 查询（带有以“？”表示的查询参数）填入查询参数的值以及参数的类型。

如第七章所示，HSQL (Hibernate SQL) 查询很像标准的 SQL 语句，除了不需要“SELECT [value]”部分之外，因为 Hibernate 会根据映射帮我们填入此部分。这个方法会查询 `Product` 这个表中所有 `categoryId` 等于所传入值的数据行，并对返回结果集中每个数据行创建 `Product` 的实例。所有的实例会放在 `List` 中返回。

DAO 的最后一个方法是 `searchProductList`，它更为复杂，但幸运的是，我们无需实

现它。因为我们已经用 Simple Spider 替换了原来的搜索功能，这个方法现在就不会被调用，所以只要返回 `null` 即可（因为 `ProductDao` 的接口还是会要求使用这个方法）。

要完成这个新架构，我们只需对其余的五个域对象重复这些步骤。每个对象都要有映射文件以及适当 DAO 接口的实现。

改变应用程序配置

为了要让新的 DAO 能够在 JPetStore 上运行，我们需要修改一些配置文件。首先，我们必须建立全局 `hibernate.properties` 文件，它告诉 Hibernate 使用哪个数据库以及如何使用。JPetStore 目前配置成使用本机的 Hypersonic SQL，用户名称为“sa”并且没有密码（千万不要在实际情形下这么做）。`hibernate.properties` 文件就像下面这样：

```
hibernate.connection.driver_class = org.hsqldb.jdbcDriver
hibernate.connection.url = jdbc:hsqldb:hsqldb://localhost:9002
hibernate.connection.username = sa
hibernate.connection.password =
hibernate.dialect=net.sf.hibernate.dialect.HSQLDialect
hibernate.show_sql=true
```

这个文件应该与其他全局配置文件一样存储在项目的根路径。Hibernate 会根据名称来寻找。

接着，开启 JPetStore 的 `dataAccessContext-*.xml` 文件（一个是 `dataAccessContext-jta.xml`，而另一个是 `dataAccessContext-local.xml`）。其中有一段是 DAO 的映射。将映射点改成新的 DAO，并去掉不需要的属性。举例来说，原来的 `ProductDao` 映射是：

```
<bean id="productDao" class="org.springframework.samples.JPetStore.
    dao.ibatis.SqlMapProductDao">
    <property name="dataSource"><ref local="dataSource"/></property>
    <property name="sqlMap"><ref local="sqlMap"/></property>
</bean>
```

改成：

```
<bean id="productDao" class="org.springframework.samples.JPetStore.dao.
    hibernate.HibernateProductDao"/>
```

我们可以去掉属性是因为 Hibernate 版的 DAO 不需要通过控件来传入任何配置信息；Hibernate 会帮我们管理这些问题。

一旦你成功地修改所有的 DAO 引用，最后要做的是引入必要的 `jar` 文件到类路径中。Hibernate 需要下面的 `jar`：`hibernate2.jar`、`cglib2.jar`、`encache.jar`、`commons-collections.jar`、`dom4j.jar` 与 `jta.jar`（全部都包括在 Hibernate 的下载中）。

Spring 的内置 Hibernate 支持

现在你已经看过明确的方法,让我们简单看一下Spring对Hibernate的支持性基础设施。Spring通过它的“控制逆转”架构能够完全地帮你管理SessionFactory的创建。它提供一个新的类(称为HibernateDaoSupport)来让指定应用程序的DAO使用标准的、模板驱动的调用,以便与数据源进行交互。

要对它进行设置,你必须修改DAO以扩展HibernateDaoSupport。因此下面这行代码:

```
public class HibernateProductDao implements ProductDao
```

将变成:

```
public class HibernateProductDao extends HibernateDaoSupport implements ProductDao
```

然后加入下面这段代码来让Spring传入SessionFactory:

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

加入上面这项之后,你的DAO可以使用由HibernateDaoSupport所提供的HibernateTemplate对象。这个新的类是通过继承来自HibernateDaoSupport的getHibernateTemplate()来访问,它会呈现出一系列的辅助方法来与数据库进行交互,其中有load、save、update、saveOrUpdate、get与find等。我们的ProductDao会变得简单一些:

```
public class HibernateProductDao extends HibernateDaoSupport implements ProductDao {
    private SessionFactory sessionFactory;

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

    public HibernateProductDao() {
    }

    public List getProductListByCategory(String categoryId) {
        return getHibernateTemplate().find("from product where
            product.category = ?", categoryId, Hibernate.STRING);
    }

    public List searchProductList(String keywords) throws DataAccessException {
        return null;
    }
}
```

```
        public Product getProduct(String ProductID) throws DataAccessException {
            return (Product) getHibernateTemplate().load(Product.class, productId);
        }
    }
}
```

要设定这些，你就必须对配置文件做些修改。现在必须为 `SessionFactory` 加入定义在 `ProductDao` 这个 bean 中的属性：

```
<bean id="productDao"
      class="org.springframework.samples.JPetStore.dao.hibernate.HibernateProductDao">
    <property name="sessionFactory"/>
    <ref bean="mySessionFactory"/>
</bean>
```

然后加入 `mySessionFactory` 这个 bean 的属性：

```
<bean id="mySessionFactory"
      class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
    <property name="mappingResources">
        <list>
            <value>product.hbm.xml</value>
        </list>
        <!-- etc. -->
    </property>
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">net.sf.hibernate.dialect.HSQLDialect</prop>
        </props>
    </property>
    <property name="dataSource">
        <ref bean="dataSource"/>
    </property>
</bean>
```

加入所有的映射文件到 `mappingResources` 属性中，并确定 `dataSource` 属性涉及到已经设定好的 `dataSource` bean。通过这些最少的修改，你的 DAO 就会更为标准化且更为紧凑，而且 Spring 会帮你处理所有 `SessionFactory` 和 `Session` 的实现细节。你就可以专注于手上的问题，而不是如何支持框架。

就这样！我们完全换掉现有的代码而无需改动源代码。我们只是加入新的类到项目中并对配置做些修改，然后就 Hibernate 了！

法则的应用

- 保持简单：域对象还是不需知道持久化的逻辑，Hibernate 会负责所有的配置设定
- 选择正确的工具：Hibernate

- 一次做好一件事：域模型专注于业务问题，DAO 专注于数据的操纵，并且是对数据库一无所知
- 力求透明：域模型完全不知道持久层
- 允许扩展：Spring 的配置和 IoC（控制逆转）让我们可以更换持久层

小结

在前两章中，我们接受客户提出的通用、灵活的网站搜索引擎需求，并加以重新定义以符合我们的基本法则。然后设计出简单、直接的应用程序来符合这些需求，并利用现有的工具（虽然是非正统的）来完成复杂的工作。结果产出了低于 Google 开价的 18,000 美元的搜索应用程序，就算我们把寻找开源工具和编写这两章的费用都加到设计与开发成本上，也还是比较便宜！并且坦白说我们其实并不廉价。简单化确实有它的代价。

我们已经知道用核心基本原则的思想去集成两个应用程序有多简单。因为JPetStore示例内置于轻量化框架（Spring）中，并且运用到最常见的设计模式（MVC），介绍服务的替换工作只是个骗小孩子的把戏而已。

因为Spider写得很好并通过配置服务提供了灵活性，在新的上下文中、容器环境下，以全新的用户接口使用它是很容易的。而且，因为Hibernate也是依据同样的法则来创建，在项目中用它来取代现有的持久化机制也是难以置信的简单。

这些示例展示了Java不一定是困难的。事实上，如果你专注于本书所描述的基本法则，Java还可以很有趣。