Spring 2.0 宝典

内容简介:

Spring 是目前最流行的 J2EE 框架，在 J2EE 应用的各层都有其不俗的表现。Spring 提倡的“实用主义”法则大大简化了 J2EE 的开发。

本书由浅入深、全面地介绍了 Spring 的结构体系，内容覆盖到 Spring 近 80%的 API。全书分 21 章，内容涵盖了 Spring 的核心机制、依赖注入、资源访问、AOP 框架、事务框架、整合 Hibernate、DAO 支持、JDBC 支持、MVC 框架、整合第三方表现层技术、整合第三方 MVC 框架、远程访问支持、EJB 访问和实现、Spring 对测试的简化、Spring 对 JMS 和 JavaMail 的支持等。本书的示例都是笔者精心挑选，具有很强的针对性，力求让读者可以明白 Spring 每个方面的知识点。最后的两个综合案例，采用最科学的轻量级 J2EE 结构，涉及的框架有 Spring，Struts，WebWork2，Hibernate，FreeMarker，Velocity 等，知识面相当全面，具有很好的示范作用，务必让读者感受到高质量 J2EE 应用的魅力。

本书适用于有较好的 Java 编程基础，有一定的 J2EE 编程基础的用户。本书既可以作为 Spring 的学习指南，也可作为实际开发人员的参考手册。

1.4 Spring 的基本设计思想

Spring 实现了两种基本设计模式：

- 工厂模式
- 单态模式

Spring 容器是实例化和管理全部 bean 的工厂，工厂模式可将 Java 对象的调用者从被调用者的实现逻辑中分离出来，调用者只关心被调用者必须满足的某种规则（接口），而不必关心实例的具体实现过程，具体的实现由 bean 工厂完成。

Spring 默认将所有的 bean 设置成单态模式，即：对所有相同 id 的 bean 的请求，都将返回同一个共享实例。单态模式可大大降低 Java 对象创建和销毁时的系统开销。使用 Spring 将 bean 设成单态行为，则无须自己完成单态模式。

1.4.1 单态模式的回顾

单态模式限制了类实例的创建，采用这种模式设计的类，可以保证仅有一个实例，并提供访问该实例的全局访问点。J2EE 应用的大量组件，都需要保证一个类只有一个实例。比如数据库引擎访问点只能有一个。更多的时候，为了提高性能，程序应尽量减少 Java 对象的创建和销毁时开销。使用单态模式可避免 Java 类的频繁实例化，让相同类的全部实例共享同一内存区。

为了防止单态模式的类被多次实例化，应将类的构造器设成私有的，这样，保证只能通过静态方法获得类实例。而该静态方法则保证每次返回的实例都是同一个，这就需将该类的实例设置成类属性，该属性需要被静态方法访问，因此该属性应设成静态属性。下面给出单态模式的示例代码：

//单态模式测试类

```
public class SingletonTest
{
   //该类的一个普通属性。
     int value ;
     //使用静态属性类保存该类的一个实例。
   private static SingletonTest instance;
   //构造器私有化，避免该类被多次实例。
     private SingletonTest()
   {
            System.out.println("正在执行构造器…");
   }
   //提供静态方法来返回该类的实例。
     public static SingletonTest getInstance()
   {
            //实例化类实例前，先检查该类的实例是否存在
        if (instance == null)
            {
                //如果不存在，则新建一个实例。
                instance = new SingletonTest();
            }
            //返回该类的成员变量:该类的实例。
            return instance;
   }
      //以下提供对普通属性 value 的 setter 和 getter 方法
   public int getValue()
   {
        return value;
   }
   public void setValue(int values)
   {
        this.value = value;
   }
   public static void main(String[] args)
   {
        SingletonTest   t1 = SingletonTest .getInstance();
        SingletonTest   t2 = SingletonTest .getInstance();
        t2.setValue(9);
            System.out.println(t1 == t2);
   }
}
```

根据程序最后的打印结果，可以看出类的两个实例完全相同，这证明：单态模式的类的全部实例是同一共享实例。程序里虽然获得了类的两个实例，但实际上只执行一次构造器，因为

对于单态模式的类，不管有多少次的创建实例请求，都只执行一次构造器。

1.4.2 工厂模式的回顾

工厂模式根据调用数据返回某个类的一个实例，此类可能是多个类的某一个类。通常，这些类满足共同的规则（接口）或父类。调用者只关心工厂生产的实例是否满足某种规范，即实现了某个接口；是否可供自己正常调用（调用者仅仅使用）。该模式提供各对象之间清晰的角色划分，降低程序的耦合。

接口产生的全部实例通常实现相同接口，接口里定义全部实例共同拥有的方法，这些方法在不同的实现类中实现方式不同。程序调用者无须关心方法的具体实现，从而降低系统异构的代价。下面是工厂模式的示例代码：

```
//Person 接口定义
public interface Person
{
        /**
         * @param name 对 name 打招呼
         * @return 打招呼的字符串
         */
    public String sayHello(String name);
        /**
         * @param name 对 name 告别
         * @return 告别的字符串
         */
    public String sayGoodBye(String name);
}
```

该接口定义 Person 的规范，该接口必须拥有两个方法：能打招呼、能告别。规范要求实现该接口的类必须具有这两个方法：

```
//American 类实现 Person 接口
public class American implements Person
{
        /**
         * @param name 对 name 打招呼
         * @return 打招呼的字符串
         */
    public String sayHello(String name)
    {
          return name + ",Hello";
    }
        /**
         * @param name 对 name 告别
         * @return 告别的字符串
         */
    public String sayGoodBye(String name)
```

```
    {
        return name + ",Good Bye";
    }
}
```

下面是实现 Person 接口的另一个实现类：Chinese

```
public class Chinese implements Person
{
    /**
     * @param name 对 name 打招呼
     * @return 打招呼的字符串
     */
    public String sayHello(String name)
    {
        return name + "，您好";
    }
    /**
     * @param name 对 name 告别
     * @return 告别的字符串
     */
    public String sayGoodBye(String name)
    {
        return name + "，下次再见";
    }
}
```

然后看 Person 工厂的代码：

```
public class PersonFactory
{
    /**
     * 获得 Person 实例的实例工厂方法
     * @ param ethnic 调用该实例工厂方法传入的参数
     * @ return 返回 Person 实例
     */
    public Person getPerson(String ethnic)
    {
            //根据参数返回 Person 接口的实例。
        if (ethnic.equalsIgnoreCase("chin"))
        {
            return new Chinese();
        }
        else
        {
            return new American();
```

```
        }
    }
}
```

最简单的工厂模式的框架基本如上所示。

主程序部分仅仅需要与工厂耦合，而无须与具体的实现类耦合在一起。下面是主程序部分：

```
public class FactroyTest
{
    public static void main(String[] args)
      {
            //创建 PersonFactory 的实例，获得工厂实例
            PersonFactory pf = new PersonFactory();
            //定义接口 Person 的实例，面向接口编程
            Person p = null;
            //使用工厂获得 Person 的实例
            p = pf.getPerson("chin");
            //下面调用 Person 接口的方法
            System.out.println(p.sayHello("wawa"));
            System.out.println(p.sayGoodBye("wawa"));
            //使用工厂获得 Person 的另一个实例
        p = pf.getPerson("ame");
            //再次调用 Person 接口的方法
        System.out.println(p.sayHello("wawa"));
        System.out.println(p.sayGoodBye("wawa"));
    }
}
```

主程序从 Person 接口的具体类中解耦出来，而且，程序调用者无须关心 Person 的实例化过程，角色划分清晰。主程序仅仅与工厂服务定位结合在一起：获得工厂的引用，程序将可获得所有工厂能产生的实例。具体类的变化，重要接口不发生任何改变，调用者程序代码部分几乎无须发生任何改动。

下面看 Spring 对这两种模式的实现。

1.4.3 Spring 对单态与工厂模式的实现

无须修改程序的接口和实现类。Spring 提供工厂模式的实现，因此，对于 PersonFactory 工厂类，此处不再需要。Spring 使用配置文件管理所有的 bean，该 bean 就是 Spring 工厂能产生的全部实例。下面是关于两个实例的配置文件：

```
<!-- 下面是 XML 文件的文件头-->
<?xml version="1.0" encoding="gb2312"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
      "http://www.springframework.org/dtd/spring-beans.dtd">
<!--  beans 是 Spring 配置文件的根元素-->
<beans>
```

```
    <!-- 定义第一个 bean,该 bean 的 id 为 chinese->
    <bean id="chinese" class="lee.Chinese"/>
    <!-- 定义第二个 bean,该 bean 的 id 为 american-->
  <bean id="american" class="lee.American"/>
</beans>
```

主程序部分如下：

```
public class SpringTest
{
  public static void main(String[] args)
  {
        //实例化 Spring 容器
    ApplicationContext ctx = new FileSystemXmlApplicationContext("bean.xml");
        //定义 Person 接口的实例
    Person p = null;
        //通过 Spring 上下文获得 chinese 实例
    p = (Person)ctx.getBean("chinese");
        //执行 chinese 实例的方法
    System.out.println(p.sayHello("wawa"));
    System.out.println(p.sayGoodBye("wawa"));
        //通过 Spring 上下文获得 american 实例
    p = (Person)ctx.getBean("american");
        //执行 american 实例的方法
    System.out.println(p.sayHello("wawa"));
    System.out.println(p.sayGoodBye("wawa"));
  }
}
```

使用 Spring 至少有一个好处：即使没有工厂类 PersonFactory，程序一样可以使用工厂模式。所有工厂模式的功能，Spring 完全可以提供。下面对主程序部分做出简单的修改：

```
public class SpringTest
{
  public static void main(String[] args)
  {
        //实例化 Spring 容器
    ApplicationContext ctx = new FileSystemXmlApplicationContext("bean.xml");
        //定义 Person 接口的实例 p1
    Person p1 = null;
        //通过 Spring 上下文获得 chinese 实例
    p1 = (Person)ctx.getBean("chinese");
        //定义 Person 接口的实例 p1
        Person p2 = null;
    p2 = (Person)ctx.getBean("chinese");
        System.out.println(p1 == p2);
```

```
    }
}
```

程序执行结果是：

```
true
```

表明：Spring 对接受容器管理的全部 bean，默认采用单态模式管理。除非必要，笔者建议不要随便更改 bean 的行为方式：性能上，单态的 bean 比非单态的 bean 更优秀。

仔细检查上面的代码，发现如下特点：

- 除测试用主程序部分，代码并未出现 Spring 特定的类和接口。
- 调用者代码，也就是测试用主程序部分，仅仅面向 Person 接口编程。而无须知道实现类的具体名称。同时，可以通过修改配置文件来切换底层的具体实现类。
- 工厂通常无须多个实例，因此，工厂应该采用单态模式设计。Spring 的上下文，也就是产生 bean 实例的工厂，已被设计成单态的。

Spring 实现的工厂模式，不仅提供了创建 bean 的功能，还提供对 bean 生命周期的管理。最重要的是：还可管理 bean 与 bean 之间的依赖关系，以及 bean 的属性值。

1.5 Spring 的核心机制

Spring 能有效地组织 J2EE 应用各层的对象。不管是控制层的 Action 对象，还是业务层的 Service 对象，还是持久层的 DAO 对象，都可在 Spring 的管理下有机地协调、运行。Spring 将各层的对象以松耦合的方式组织在一起，Action 对象无须关心 Service 对象的具体实现，Service 对象无须关心持久层对象的具体实现，各层对象的调用完全面向接口。当系统需要重构时，代码的改写量将大大减少。

上面所说的一切都得宜于 Spring 的核心机制，依赖注入。依赖注入让 bean 与 bean 之间以配置文件组织在一起，而不是以硬编码的方式耦合在一起。

1.5.1 理解依赖注入

依赖注入（Dependency Injection）和控制反转（Inversion of Control）是同一个概念。具体含义是：当某个角色（可能是一个 Java 实例，调用者）需要另一个角色（另一个 Java 实例，被调用者）的协助时，在传统的程序设计过程中，通常由调用者来创建被调用者的实例。但在 Spring 里，创建被调用者的工作不再由调用者来完成，因此称为控制反转；创建被调用者实例的工作通常由 Spring 容器来完成，然后注入调用者，因此也称为依赖注入。

不管是依赖注入，还是控制反转，都说明 Spring 采用动态、灵活的方式来管理各种对象。对象与对象之间的具体实现互相透明。在理解依赖注入之前，看如下这个问题在各种社会形态里如何解决：一个人（Java 实例，调用者）需要一把斧子（Java 实例，被调用者）。

（1）原始社会里，几乎没有社会分工。需要斧子的人（调用者）只能自己去磨一把斧子（被调用者）。对应的情形为：Java 程序里的调用者自己创建被调用者。

（2）进入工业社会，工厂出现。斧子不再由普通人完成，而在工厂里被生产出来，此时需要斧子的人（调用者）找到工厂，购买斧子，无须关心斧子的制造过程。对应 Java 程序的简单工厂的设计模式。

（3）进入“按需分配”社会，需要斧子的人不需要找到工厂，坐在家里发出一个简单指令：需要斧子。斧子就自然出现在他面前。对应 Spring 的依赖注入。

第一种情况下，Java 实例的调用者创建被调用的 Java 实例，必然要求被调用的 Java 类出现在调用者的代码里。无法实现二者之间的松耦合。

第二种情况下，调用者无须关心被调用者具体实现过程，只需要找到符合某种标准（接

口）的实例，即可使用。此时调用的代码面向接口编程，可以让调用者和被调用者解耦，这也是工厂模式大量使用的原因。但调用者需要自己定位工厂，调用者与特定工厂耦合在一起。

第三种情况下，调用者无须自己定位工厂，程序运行到需要被调用者时，系统自动提供被调用者实例。事实上，调用者和被调用者都处于 Spring 的管理下，二者之间的依赖关系由 Spring 提供。

所谓依赖注入，是指程序运行过程中，如果需要调用另一个对象协助时，无须在代码中创建被调用者，而是依赖于外部的注入。Spring 的依赖注入对调用者和被调用者几乎没有任何要求，完全支持对 POJO 之间依赖关系的管理。依赖注入通常有两种：

- 设值注入。
- 构造注入。

1.5.2 设值注入

设值注入是指通过 setter 方法传入被调用者的实例。这种注入方式简单、直观，因而在 Spring 的依赖注入里大量使用。看下面代码，是 Person 的接口

```
//定义 Person 接口
public interface Person
{
    //Person 接口里定义一个使用斧子的方法
    public void useAxe();
}
```

然后是 Axe 的接口

```
//定义 Axe 接口
public interface Axe
{
    //Axe 接口里有个砍的方法
    public void chop();
}
```

Person 的实现类

```
//Chinese 实现 Person 接口
public class Chinese implements Person
{
    //面向 Axe 接口编程，而不是具体的实现类
    private Axe axe;
    //默认的构造器
    public Chinese()
    {
    }
    //设值注入所需的 setter 方法
    public void setAxe(Axe axe)
    {
        this.axe = axe;
    }
```

```
        //实现 Person 接口的 useAxe 方法
    public void useAxe()
        {
            System.out.println(axe.chop());
        }
}
```

Axe 的第一个实现类

```
//Axe 的第一个实现类 StoneAxe
public class StoneAxe implements Axe
{
        //默认构造器
    public StoneAxe()
        {
        }
        //实现 Axe 接口的 chop 方法
        public String chop()
        {
            return "石斧砍柴好慢";
        }
}
```

下面采用 Spring 的配置文件将 Person 实例和 Axe 实例组织在一起。配置文件如下所示：

```
<!-- 下面是标准的 XML 文件头 -->
<?xml version="1.0" encoding="gb2312"?>
<!-- 下面一行定义 Spring 的 XML 配置文件的 dtd -->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- 以上三行对所有的 Spring 配置文件都是相同的 -->
<!-- Spring 配置文件的根元素 -->
<beans>
    <!—定义第一 bean，该 bean 的 id 是 chinese, class 指定该 bean 实例的实现类 -->
    <bean id="chinese" class="lee.Chinese">
        <!-- property 元素用来指定需要容器注入的属性，axe 属性需要容器注入
          此处是设值注入，因此 Chinese 类必须拥有 setAxe 方法 -->
        <property name="axe">
            <!-- 此处将另一个 bean 的引用注入给 chinese bean -->
            <ref local=”stoneAxe”/>
        </property>
    </bean>
    <!-- 定义 stoneAxe bean -->
      <bean id="stoneAxe" class="lee.StoneAxe"/>
</beans>
```

从配置文件中，可以看到 Spring 管理 bean 的灵巧性。bean 与 bean 之间的依赖关系放在配置文件里组织，而不是写在代码里。通过配置文件的指定，Spring 能精确地为每个 bean 注入属性。因此，配置文件里的 bean 的 class 元素，不能仅仅是接口，而必须是真正的实现类。

Spring 会自动接管每个 bean 定义里的 property 元素定义。Spring 会在执行无参数的构造器后、创建默认的 bean 实例后，调用对应的 setter 方法为程序注入属性值。property 定义的属性值将不再由该 bean 来主动创建、管理，而改为被动接收 Spring 的注入。

每个 bean 的 id 属性是该 bean 的惟一标识，程序通过 id 属性访问 bean，bean 与 bean 的依赖关系也通过 id 属性完成。

下面看主程序部分：

```
public class BeanTest
{
   //主方法，程序的入口
     public static void main(String[] args)throws Exception
     {
            //因为是独立的应用程序，显式地实例化 Spring 的上下文。
        ApplicationContext ctx = new FileSystemXmlApplicationContext("bean.xml");
            //通过 Person bean 的 id 来获取 bean 实例，面向接口编程，因此
            //此处强制类型转换为接口类型
            Person p = (Person)ctx.getBean("chinese");
            //直接执行 Person 的 userAxe()方法。
            p.useAxe();
   }
}
```

程序的执行结果如下：

```
石斧砍柴好慢
```

主程序调用 Person 的 useAxe()方法时，该方法的方法体内需要使用 Axe 的实例，但程序里没有任何地方将特定的 Person 实例和 Axe 实例耦合在一起。或者说，程序里没有为 Person 实例传入 Axe 的实例，Axe 实例由 Spring 在运行期间动态注入。

Person 实例不仅不需要了解 Axe 实例的具体实现，甚至无须了解 Axe 的创建过程。程序在运行到需要 Axe 实例的时候，Spring 创建了 Axe 实例，然后注入给需要 Axe 实例的调用者。Person 实例运行到需要 Axe 实例的地方，自然就产生了 Axe 实例，用来供 Person 实例使用。

调用者不仅无须关心被调用者的实现过程，连工厂定位都可以省略（真是按需分配啊！）。下面也给出使用 Ant 编译和运行该应用的简单脚本：

```
<?xml version="1.0"?>
<!-- 定义编译该项目的基本信息-->
<project name="spring" basedir="." default=".">
   <!-- 定义编译和运行该项目时所需的库文件 -->
     <path id="classpath">
           <!-- 该路径下存放 spring.jar 和其他第三方类库 -->
        <fileset dir="..\..\lib">
           <include name="*.jar"/>
```

```
        </fileset>
            <!-- 同时还需要引用已经编译过的 class 文件-->
        <pathelement path="."/>
    </path>
        <!-- 编译全部的 java 文件-->
        <target name="compile" description="Compile all source code">
            <!-- 指定编译后的 class 文件的存放位置 -->
        <javac destdir="." debug="true"
            deprecation="false" optimize="false" failonerror="true">
                <!-- 指定需要编译的源文件的存放位置 -->
            <src path="."/>
                <!-- 指定编译这些 java 文件需要的类库位置-->
            <classpath refid="classpath"/>
        </javac>
    </target>
        <!-- 运行特定的主程序 -->
        <target name="run" description="run the main class" depends="compile">
            <!-- 指定运行的主程序:lee.BeanTest。-->
        <java classname="lee.BeanTest" fork="yes" failonerror="true">
                <!-- 指定运行这些 java 文件需要的类库位置-->
            <classpath refid="classpath"/>
        </java>
    </target>
</project>
```

如果需要改写 Axe 的实现类。或者说，提供另一个实现类给 Person 实例使用。Person 接口、Chinese 类都无须改变。只需提供另一个 Axe 的实现，然后对配置文件进行简单的修改即可。

Axe 的另一个实现如下：

```
//Axe 的另一个实现类 SteelAxe
public class SteelAxe implements Axe
{
    //默认构造器
    public SteelAxe()
    {
    }
    //实现 Axe 接口的 chop 方法
    public String chop()
    {
        return "钢斧砍柴真快";
    }
}
```

然后，修改原来的 Spring 配置文件，在其中增加如下一行：

```
<!-- 定义一个 steelAxe bean-->
<bean id="steelAxe" class="lee.SteelAxe"/>
```

该行重新定义了一个 Axe 的实现：SteelAxe。然后修改 chinese bean 的配置，将原来传入 stoneAxe 的地方改为传入 steelAxe。也就是将

```
<ref local=”stoneAxe”/>
```

改成

```
<ref local=”steelAxe”/>
```

此时再次执行程序，将得到如下结果：

```
钢斧砍柴真快
```

Person 与 Axe 之间没有任何代码耦合关系，bean 与 bean 之间的依赖关系由 Spring 管理。采用 setter 方法为目标 bean 注入属性的方式，称为设值注入。

业务对象的更换变得相当简单，对象与对象之间的依赖关系从代码里分离出来，通过配置文件动态管理。

1.5.3 构造注入

所谓构造注入，指通过构造函数来完成依赖关系的设定，而不是通过 setter 方法。对前面代码 Chinese 类做简单的修改，修改后的代码如下：

```
//Chinese 实现 Person 接口
public class Chinese implements Person
{
    //面向 Axe 接口编程，而不是具体的实现类
    private Axe axe;
    //默认的构造器
    public Chinese()
    {
    }
    //构造注入所需的带参数的构造器
    public Chinse(Axe axe)
    {
        this.axe = axe;
    }
    //实现 Person 接口的 useAxe 方法
    public void useAxe()
    {
        System.out.println(axe.chop());
    }
}
```

此时无须 Chinese 类里的 setAxe 方法，构造 Person 实例时，Spring 为 Person 实例注入所依赖的 Axe 实例。构造注入的配置文件也需做简单的修改，修改后的配置文件如下：

```
<!-- 下面是标准的 XML 文件头 -->
<?xml version="1.0" encoding="gb2312"?>
<!-- 下面一行定义 Spring 的 XML 配置文件的 dtd -->
```

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- 以上三行对所有的 Spring 配置文件都是相同的 -->
<!--  Spring 配置文件的根元素 -->
<beans>
    <!—定义第一个 bean，该 bean 的 id 是 chinese, class 指定该 bean 实例的实现类 -->
    <bean id="chinese" class="lee.Chinese">
        <constructor-arg><ref bean="steelAxe"/></constructor-arg>
    </bean>
    <!-- 定义 stoneAxe bean -->
    <bean id="steelAxe" class="lee.SteelAxe"/>
</beans>
```

执行效果与使用 steelAxe 设值注入时的执行效果完全一样。区别在于：创建 Person 实例中 Axe 属性的时机不同——设值注入是现创建一个默认的 bean 实例，然后调用对应的构造方法注入依赖关系。而构造注入则在创建 bean 实例时，已经完成了依赖关系的注入。

1.5.4 两种注入方式的对比

设值注入和构造注入，都是 Spring 支持的依赖注入模式。也是目前主流的依赖注入模式。两种注入模式各有优点：

1.5.4.1 设值注入的优点

- 与传统的 JavaBean 的写法更相似，程序开发人员更容易了解、接受。通过 setter 方法设定依赖关系显得更加直观、自然。
- 对于复杂的依赖关系，如果采用构造注入，会导致构造器过于臃肿，难以阅读。Spring 在创建 bean 实例时，需要同时实例化其依赖的全部实例，因而导致性能下降。而使用设值注入，则能避免这些问题。
- 尤其是某些属性可选的情况下，多参数的构造器更加笨重。

1.5.4.2 构造注入的优点

- 可以在构造器中决定依赖关系的注入顺序。优先依赖的优先注入。例如，组件中其他依赖关系的注入，常常需要依赖于 Datasource 的注入。采用构造注入，可以在代码中清晰地决定注入顺序。
- 对于依赖关系无须变化的 bean，构造注入更有用处。因为没有 setter 方法，所有的依赖关系全部在构造器内设定。因此，无须担心后续的代码对依赖关系产生破坏。
- 依赖关系只能在构造器中设定，则只有组件的创建者才能改变组件的依赖关系。对组件的调用者而言，组件内部的依赖关系完全透明，更符合高内聚的原则。

建议采用以设值注入为主，构造注入为辅的注入策略。对于依赖关系无须变化的注入，尽量采用构造注入；而其他的依赖关系的注入，则考虑采用设值注入。

1.6 小结

本章简短的介绍了 Spring 的起源和背景。分别讲述了利用 Eclipse 和 Ant 开发 Spring 应用的步骤。同时也对 Spring 体系作了一个简单预览，以便读者对 Spring 架构形成总体把握。本章还详细介绍了两种基本设计模式、两种设计模式在 J2EE 应用中的用法，以及 Spring 对两种模式的实现。

最后形象地介绍了依赖注入的概念，详细地讲解了 Spring 对两种依赖注入的支持和用

法。同时对两种依赖注入进行简单的对比。请读者务必理解 Spring 的核心机制：依赖注入。此外，设值注入、构造注入应能理解各自的区别，并可以正确应用。

本章要点

Hibernate 入门知识

Hibernate 基本映射、关系映射

Hibernate 数据查询等知识。

Hibernate 与 Spring 整合的各种方式

Hibernate 在 Spring 中的事务管理

Hibernate 是目前最流行的开源对象关系映射(ORM)框架。Hibernate 采用低侵入式的设计，也即完全采用普通的 Java 对象（POJO），而不必继承 Hibernate 的某个基类，或实现 Hibernate 的某个接口。Hibernate 是面向对象的程序设计语言和关系数据库之间的桥梁，Hibernate 允许程序开发者采用面向对象的方式来操作关系数据库。

10.1 ORM 介绍

ORM 的全称是 Object/Relation Mapping，即对象/关系映射。ORM 也可理解是一种规范，具体的 ORM 框架可作为应用程序和数据库的桥梁。目前 ORM 的产品非常多，比如 Apache 组织下的 OJB，Oracle 的 TopLink，JDO 等等。

10.1.1 什么是 ORM

ORM 并不是一种具体的产品，而是一类框架的总称，它概述了这类框架的基本特征：完成面向对象的程序设计语言到关系数据库的映射。基于 ORM 框架完成映射后，既可利用面向对象程序设计语言的简单易用性，又可利用关系数据库的技术优势。

面向对象程序设计语言与关系数据库发展不同步时，需要一种中间解决方案，ORM 框架就是这样的解决方案。笔者认为，随着面向对象数据库的发展，其理论逐步完善，最终会取代关系数据库。只是这个过程不可一蹴而就，ORM 框架在此期间内会蓬勃发展。但随着面向对象数据库的出现，ORM 工具会自动消亡。

10.1.2 为什么需要 ORM

在上一节已经基本回答了这个问题，面向对象的程序设计语言，代表了目前程序设计语言的主流和趋势，其具备非常多的优势，比如：

- 面向对象的建模、操作。
- 多态、继承。
- 摒弃难以理解的过程。
- 简单易用，易理解性。

但数据库的发展并未与程序设计语言同步，而且，关系数据库系统的某些优势，也是面向对象的语言目前无法解决的。比如：

- 大量数据操作查找、排序。
- 集合数据连接操作、映射。
- 数据库访问的并发、事务。
- 数据库的约束、隔离。

面对这种面向对象语言与关系数据库系统并存的局面，采用 ORM 就变成一种必然。

10.1.3 流行的 ORM 框架简介

目前 ORM 框架的产品非常多，除了各大著名公司、组织的产品外，甚至，其他一些小团队也都有推出自己的 ORM 框架。目前流行的 ORM 框架有如下这些产品。

- 大名鼎鼎的 Hibernate：出自 Gavin King 的手笔，目前最流行的开源 ORM 框架，其灵巧的设计，优秀的性能，还有丰富的文档，都是其迅速风靡全球的重要因素。
- 传统的 Entity EJB：Entity EJB 实质上也是一种 ORM 技术，这是一种备受争议的组件技术，很多人说它非常优秀，也有人说它一钱不值。事实上，EJB 为 J2EE 的蓬勃发展赢得了极高的声誉，就笔者的实际开发经验而言，EJB 作为一种重量级、高花费的 ORM 技术上，具有不可比拟的优势。但由于其必须运行在 EJB 容器内，而且学习曲线陡峭，开发周期、成本相对较高，因而限制 EJB 的广泛使用。
- IBATIS：Apache 软件基金组织的子项目。与其称它是一种 ORM 框架，不如称它是一种 “Sql Mapping”框架。相对 Hibernate 的完全对象化封装，iBATIS 更加灵活，但开发过程中开发人员需要完成的代码量更大，而且需要直接编写 SQL 语句。
- Oracle 的 TopLink：作为一个遵循 OTN 协议的商业产品，TopLink 在开发过程中可以自由下载和使用，但一旦作为商业产品使用，则需要收取费用。可能正是这一点，导致了 TopLink 的市场占有率。
- OJB：Apache 软件基金组织的子项目。开源的 ORM 框架，但由于开发文档不是太多，而且 OJB 的规范一直并不稳定，因此并未在开发者中赢得广泛的支持。

Hibernate 是目前最流行的 ORM 框架，其采用非常优雅的方式，将 SQL 操作完全包装成对象化的操作。其作者 Gavin King 在持久层设计上极富经验，采用非常少的代码实现了整个框架，同时完全开放源代码，即使偶尔遇到无法理解的情况，可参照源代码来理解其在持久层上灵巧而智能的设计。

目前，Hibernate 在国内的开发人员相当多，Hibernate 的文档非常丰富，这些都为学习 Hibernate 铺平了道路,因而 Hibernate 的学习相对简单一些。下面对比 Hibernate 和传统 JDBC 操作数据库持久层差异。

10.2.1 采用传统 JDBC 操作数据库

先看这样一个需求：向数据库里增加一条新闻，新闻有新闻 Id、新闻标题、新闻内容三个属性。在传统的 JDBC 数据库访问里，完成此功能并不难。本程序采用 MySql 数据库，我们可采用如下方法来来完成：

```
import java.sql.*;
public class NewsDao
{
    /**
     * @param News 需要保存的新闻实例
     */
    public void saveNews(News news)
   {
      Connection conn = null;
      PreparedStatement pstmt = null;
      int newsId = news.getId();
```

```
String title = news.getTitle();
String content = news.getContent();
try
{
   //注册驱动
        Class.forName("com.mysql.jdbc.Driver");
        /* hibernate：想连接的数据库
          user：接数据库的用户名
          pass：连接数据库的密码
        */
   String url="jdbc:mysql://localhost/hibernate?user=root&password=pass";
   //获取连接
        conn= DriverManager.getConnection(url);
        //创建预编译的 Statement
        pstmt=conn.prepareStatement("insert into news_table values(?,?,?)");
        //下面语句为预编译 Statement 传入参数
   pstmt.setInt(1,newsId);
   pstmt.setString(2,title);
        pstmt.setString(3,content);
        //执行更新
        pstmt.executeUpdate();
}
catch (ClassNotFoundException cnf)
{
   cnf.printStackTrace();
}
catch (SQLException se)
{
   se.printStackTrace();
}
finally
{
   try
   {
      //关闭预编译的 Statement
            if (pstmt != null)pstmt.close();
            //关闭连接
      if (conn != null) conn.close();
   }
   catch (SQLException se2)
   {
      se2.printStackTrace();
```

```
        }
      }
    }
}
```

这种操作方式丝毫没有面向对象的优雅和易用，而是一种纯粹的过程式操作，在这种简单的数据库访问里，我们没有过多地感觉到这种方式的复杂与缺陷。但我们还是体会到 Hibernate 的灵巧。

10.2.2 Hibernate 下载和安装

Hibernate 目前的最新版本是 3.1.2，本章所用的代码也是基于该版本测试通过。安装和使用 Hibernate 请按如下步骤进行：

（1）登录 http://www.hibernate.org 网站，下载 Hibernate 的二进制包。windows 平台下载 zip 包，linux 平台下载 tar 包。

（2）解压缩刚下载的压缩包，在 hibernate-3.1 路径下有个 hibernate3.jar 的压缩文件，该文件是 Hibernate 的核心类库文件。该路径下还有 lib 路径，该路径包含 Hibernate 编译和运行的第三方类库。关于这些类库的使用，请参看该路径下的 readme.txt 文件。

（3）将必需的 Hibernate 类库添加到 JDK 的 CLASSPATH 里，或者使用 Ant 工具。总之，编译和运行时可以找到这些类即可。对于 Web 应用，则应将这些类库增加到 WEB-INF/lib 下。

10.2.3 Hibernate 初探

在使用 Hibernate 之前，首先了解一个概念：PO（Persistent Object）持久化对象。持久化对象的作用是完成持久化操作，简单地说，通过该对象可对数据执行增、删、改的操作——以面向对象的方式操作数据库。

Hibernate 里的 PO 是非常简单的，前面已经说过 Hibernate 是低侵入式的设计，完全采用普通 Java 对象来作为持久化对象使用，看下面的 POJO（普通 Java 对象）类

```
public class News
{
    int id;
    String title;
    String content;
        public void setId(int id)
    {
                this.id = id;
        }
        public int getId()
     {
                return (this.id);
        }
       ……
}
```

为了节省篇幅，笔者并未列出该类的 title 属性，和 content 属性的 setter 与 getter 方法，读者可自行增加。这个类与常规的 JavaBean 没有任何区别，是个非常标准的简单 JavaBean。

目前，这个普通的 JavaBean 还不具备持久化操作的能力，为了使其具备持久化操作的能力，Hibernate 采用 XML 映射文件，该映射文件也是非常简单。下面提供该 XML 文件的全部代码：

```
<?xml version="1.0" encoding=”gb2312”?>
<!DOCTYPE hibernate-mapping
   PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!--上面四行对所有的 hibernate 映射文件都相同 -->
<!-- hibernate-mapping 是映射文件的根元素 -->
<hibernate-mapping>
    <!-- 每个 class 元素对应一个持久化对象 -->
    <class name="News" table=”news_table”>
       <-- id 元素定义持久化类的标识属性 -->
     <id name = "id" unsaved-value = "null">
        <generator class="increment"/>
     </id>
     <!--   property 元素定义常规属性 -->
     <property name="title"/>
     <property name="content"/>
   </class>
</hibernate-mapping>
```

对这个文件简单地解释一下：从 1 到 4 行，是该 XML 文件的文件头部分，定义该文件的 XML 版本，还有 DTD，这四行对于所有 Hibernate3.x 的映射文件全部相同。hibernate-mapping 元素是所有 Hibernate 映射文件的根元素，这个根元素对所有的映射文件都是相同的。

hibernate-mapping 元素下有子元素 class 元素，每个 class 元素映射一个 PO，更准确地说，应该是持久化类。可以看到：PO = POJO + 映射文件

现在，即可通过这个持久化类完成数据库的访问：插入一条新闻。在插入一条新闻之前，还必须完成接受 Hibernate 管理的数据库的配置——连接数据库所需的用户名、密码，以及数据库名等等基本信息。配置这些基本信息，可通过.properties 的属性文件，或 hibernate.cfg.xml 配置文件配置。本文采用 hibernate.cfg.xml 的配置方式。下面是这个配置文件的详细代码：

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
      "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
      "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<!--上面四行对所有的 hibernate 连接配置文件都相同 -->
<!-- hibernate- configuration 是连接配置文件的根元素 -->
<hibernate-configuration>
    <session-factory>
       <!-- 指定连接数据库所用的驱动 -->
     <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
```

```
    <!-- 指定连接数据库的 url，Hibernate 连接的数据库名 -->
    <property name="connection.url">jdbc:mysql://localhost/hibernate</property>
    <!-- 指定连接数据库的用户名 -->
    <property name="connection.username">root</property>
    <!-- 指定连接数据库的密码 -->
    <property name="connection.password">pass</property>
    <!-- 指定连接池的大小-->
    <property name="connection.pool_size">5</property>
    <!-- 指定数据库方言-->
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <!-- 根据需要自动创建数据库-->
    <property name="hbm2ddl.auto">create</property>
    <!-- 罗列所有的映射文静-->
    <mapping resource="News.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

该配置文件非常简单，前面四行都是 XML 的基本定义和 DTD 声明，所有 Hibernate 配置文件前面四行都完全相同。Hibernate 配置文件的根元素是 hibernate-configuration，根元素里有子元素 session-factory 元素，该元素依次有很多 property 元素，property 元素依次定义连接数据库的驱动、url、用户名、密码、数据库连接池的大小（关于连接池的概念和用法请看 9.2 节）。还定义一个名为 dialect 的属性，该属性定义 Hibernate 连接的数据库类型是 MySQL，针对该数据库的特性，Hibernate 会在访问时进行优化。最后一行 mapping 定义持久化类的映射文件。如果有多个持久化映射文件，罗列多个 mapping 元素在此处即可。下面是完成新闻插入的代码

```
public class NewsDaoHibernate
{
      Configuration configuration;
      SessionFactory sessionFactory;
      Session session;
  public void saveNews(News news)
  {
        //实例化 Configuration
        configuration=new Configuration().configure();
        //实例化 SessionFactory
        sessionFactory = configuration.buildSessionFactory();
        //实例化 Session
        session = sessionFactory.openSession();
        //开始事务
        Transaction tx = session.beginTransaction();
        //增加新闻
        session.save(news);
        //提交事务
```

```
            tx.commit();
            //关闭 Session
            session.close();
        }
}
```

此时的代码结构非常清晰，保存新闻仅仅只需要个语句：session.save(news)；而且是完全对象化的操作方式，可以说是非常简单、明了。代码显示：执行 session.save(News)之前，先要获取 Session 对象。PO 只有在 Session 的管理下，才可完成数据库访问。随 PO 与 Session 的关系，PO 可有如下三个状态：瞬态、持久化、脱管。

对 PO 的操作，必须在 Session 的管理下才能同步到数据库。Session 由 SessionFactory 工厂产生，SessionFactory 是数据库编译后的内存镜像，通常，一个应用对应一个 SessionFactory 对象。SessionFactory 对象由 Configuration 对象生成。Configuration 对象用来加载 Hibernate 配置文件。最后，使用如下方法来完成对新闻的增加：

```
public static void main(String[] args)
{
    News n = new News();
     n.setTitle("新闻标题");
     n.setContent("新闻内容");
     NewsDaoHibernate ndh = new NewsDaoHibernate();
     ndh.saveNews(n);
}
```

这里仅仅提供一个主方法，相信读者可以将其补充完整，然后完成新闻的添加。

10.2.4 Hibernate 的基本映射

在上面的例子里，可看到一个简单的 Hibernate 映射文件，每个 Hibernate 映射文件的基本结构都是相同的。映射文件的根元素为 hibernate-mapping 元素，这个元素下可以拥有多个 class 子元素，每个 class 子元素完成一个持久化类的映射。如下是一个映射文件的基本结构，在 hibernate-mapping 元素下可以有多个 class 子元素。

```
<hibernate-mapping>
            <class/>
            <class/>
            ……
</hibernate-mapping>
```

接下来看 class 元素，每个 class 元素对应一个持久化类。首先，必须采用 name 元素来指定该持久化类的类名，此处的类名应该是全限定的类名。如果不使用全限定的类名，则必须在 hibernate-mapping 元素里指定 package 元素，package 元素指定持久化类所在的包路径。

持久化类都需要有一个标识属性，该标识属性用来标识该久化类的实例，因此，标识属性通常被映射成数据表主键。标识属性通过 id 元素来指定。id 元素的 name 属性的值，就是持久化类标识属性名。

通常，标识属性应该指定主键生成策略，Hibernate 建议数据表采用逻辑主键，而不要采用有物理含义的实体主键。逻辑主键没有实际意义，仅仅用来标识一行记录，通常由 Hibernate 负责生成。主键生成器负责生成数据表记录的主键。通常采用如下常见的主键生成器：

- increment：对 long，short 或 int 的数据列生成自增长主键
- identity：对 long，short 或 int 的数据列，对如 SQL Server，MySQL 等支持自动增长列的数据库，生成自增长主键。
- sequence：对 long、short 或 int 的数据列，对如 Oracle，DB2 等支持 Sequence 的数据库，生成自增长主键。
- uuid：对字符串列的数据采用 128-位 UUID 算法，生成唯一的字符串主键。

property 元素定义持久化类的普通属性，该持久化类有多少个普通属性，就需要有多少个 property 元素，class 元素的结构如下所示：

```
<class name="News" table=”news_table”>
   <!-- 定义标识属性-->
    <id name = "id" unsaved-value = "null">
       <!-- 定义主键生成器-->
     <generator class="increment"/>
    </id>
    <!-- 用于定义普通属性 -->
   <property />
   <property />
</class>
```

<property/>元素的定义相对简单，基本 property 映射只需要 name 属性，name 属性映射持久类的属性名。如果想指定属性在数据表里存储的列名，还可以定义 column 属性来强制指定列名。列名默认与属性名相同。

10.2.5 Hibernate 的关系映射

关系是关系型数据库的最基本特征，也是客观世界最基本，最必须的抽象。关系可分为如下两个类：

- 单向关系
- 双向关系

单向关系里又分为：

- 1 – 1
- 1 – N
- N – 1
- N –N

双向关系里也可分为：

- 1 – 1
- 1 – N
- N – N

双向关系里没有 N – 1 因为双向关系 1 – N 和 N –1 是完全相同的。

单向的 N – 1

先看如下两个 POJO，

```
import java.util.Set;
import java.util.HashSet;
public class Person
```

```
{
    private int personid;
     private String name;
     private int age;
    private Address address;

     // 此处省略 personid,name,age 三个属性的 setter.getter 方法
   /**
    * @return 返回此人对应的地址
    * /
  public Address getAddress()
    {
     return address;
   }
   /**
    * @param address 修改人对应的地址
    * /
  public void setAddress(Address address)
    {
     this.address = address;
  }
}
```

这是 Person 对应的 POJO，每个 Person 单向地对应一个 Address。无法从 Address 端来访问 Person。下面是 Address 的 POJO

```
public class Address
{
     private int addressid;
     private String addressdetail;
     /**
      * 设置该地址的标识属性
      * @param addressid Address 标识属性
      */
     public void setAddressid(int addressid)
    {
            this.addressid = addressid;
     }
     /**
      * 设置该地址的详细地址位置
      * @param addressdetail Address 的详细地址位置
      */
     public void setAddressdetail(String addressdetail)
    {
```

```
            this.addressdetail = addressdetail;
        }
        /**
         * 返回该地址的标识属性
         * @return 地址的标识属性
         */
        public int getAddressid()
       {
            return (this.addressid);
        }
        /**
         * 返回该地址的详细地址位置
         * @return 地址的详细地址位置
         */
        public String getAddressdetail()
       {
            return (this.addressdetail);
        }
}
```

POJO 加上其 XML 配置文件，就成为 PO。Person 和 Address 两个 POJO 之间，存在单向 N – 1 的关系。因此，映射文件在基本映射的基础上，增加关系映射。两个 POJO 对应的映射文件如下所示：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping >
    <!-- 下面 class 元素映射的是持久化类 Person-->
    <class name="Person">
        <!-- 映射标识属性 personid -->
            <id name="personid" >
            <!-- 定义主键生成器 -->
                <generator class="identity"/>
        </id>
        <property name="name"/>
        <property name="age"/>
            <!-- 用来映射关联的 PO column，是 Address 在该表中的外键列名 –>
            <many-to-one name="address" column="addressId" />
     </class>
     <!--  下面持久化类映射 Address -->
     <class name="Address">
          <!--  映射标识属性 -->
```

```
        <id name="addressid">
            <generator class="identity"/>
        </id>
        <property name="addressdetail"/>
    </class>
</hibernate-mapping>
```

关系映射成功后，对于从前复杂的 JDBC 多表连接查询，可采用面向对象的方式来访问记录。例如：需要查询 id 为 6 的人所在地址。以前的 SQL 查询将是 2 个表的连接查询。Hibernate 允许我们通过如下简单的代码完成。

```
public class AddressDaoHibernate
{
        Configuration configuration;
        SessionFactory sessionFactory;
        Session session;
        /**
         * @param personid 需要查询的人的 id
         * @return 返回 id 为 personid 的人的住址
      */
    public String find(int personid)
    {
            //实例化 Configuration
            configuration=new Configuration().configure();
            //实例化 SessionFactory
            sessionFactory = configuration.buildSessionFactory();
            //实例化 Session
            session = sessionFactory.openSession();
            //开始事务
            Transaction tx = session.beginTransaction();
            //查询到 id 为 personid 的 Address
            Person p = (Person)session.load(Person.class,new Integer(personid));
            //通过 p 直接返回地址
            Address a = p.getAddress();
            //打印出地址信息
            return a. getAddressdetail();
            //提交事务
            tx.commit();
            //关闭 Session
            session.close();
      }
}
```

上面的操作，完全看不出底层数据表之间如何互相连接。这种连接无须程序开发者关心，程序开发者的操作方式完全是面向对象的。

双向 N - 1 关系映射

在双向映射中，Person 这个 POJO 类无须任何修改，不再赘叙。下面给出 Address 的源文件：

```
public class Address
{
    private int addressid;
    private String addressdetail;
    //一个地址可以对应多个人
   private List persons = new ArrayList();
   /**
    * @param addressid 地址 id
    */
    public void setAddressid(int addressid)
    {
        this.addressid = addressid;
    }
   /**
    * @param addressdetail 地址详细信息
    */
    public void setAddressdetail(String addressdetail)
    {
        this.addressdetail = addressdetail;
    }
   /**
    * @return 地址 id
    */
    public int getAddressid()
    {
        return addressid;
    }
   /**
    * @return 地址详细信息
    */
    public String getAddressdetail()
    {
        return addressdetail;
    }
   /**
    * @return 住在该地址全部的人
    */
  public List getPersons()
    {
```

```
        return persons;
    }
      /**
       * @param 住在该地址全部的人
        */
    public void setPersons(List persons)
        {
       this.persons = persons;
    }
}
```

Person 的映射文件无须修改，修改后的 Address 映射文件如下所示：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="lee">
     <class name="Address">
        <!-- 映射标识属性 -->
       <id name="addressid">
          <generator class="identity"/>
       </id>
          <!-- 映射普通属性 -->
       <property name="addressdetail"/>
          <!-- 映射关系持久化类 -->
              <list name="persons">
                    <!-- 该 key 元素的 column 值，应与 person 表中的外键列名相同-->
                    <key column="addressId"/>
                    <!--  list 元素必须的索引列-->
                    <list-index column="addressorder"/>
                    <!-- 映射到持久化类 Person -->
                    <one-to-many class="Person"/>
              </list>
       </class>
</hibernate-mapping>
```

此文件与之前的 Address 的映射非常相似，只是增加了 list 元素。list 元素用来映射持久化类的集合属性。如果集合属性对于其他持久类，则为 list 元素增加 one-to-many 或 many-to-many 元素，用来映射持久化类。list 是有序集合，所以 list 元素增加 list-index 元素，该元素映射集合里元素的索引值。

双向 N – N 关系映射

与双向的 N – 1 的关系映射相比，Person 端做出简单修改。Person 可对应多个 Address。修改后后的 Person POJO 如下：

```
public class Person
```

```
{
        private int personid;
        private String name;
        private int age;
       private List addresses = new ArrayList();

        //此处省略了 personid,name,age 三个属性的 setter,getter 方法，读者应该自行补齐

        /**
         * @return 此人的所有住址
         */
      public List getAddresses()
      {
        return addresses;
      }
        /**
         * @return 此人的所有住址的集合
         */
      public void setAddresses(List addresses)
      {
        this.addresses = addresses;
     }
}
```

Address 的 POJO 不需要任何修改。N – N 的关系映射，需要无采用过主、外键的约束限定，必须使用连接表。因此，在配置文件中需要指定连接表的表名，并且，两个持久化类的连接表的表名必须一致。下面是两个 POJO 的映射文件

```
<hibernate-mapping package="lee">
        <!-- 映射持久化类 Person-->
      <class name="Person">
              <!-- 映射标识属性-->
        <id name="personid" >
           <generator class="identity"/>
        </id>
            <!-- 映射普通属性-->
        <property name="name"/>
        <property name="age"/>

           <!-- 映射关系持久化类，table 属性的值必须与另一个持久化类的 table 属性值相同
-->
        <list name="addresses" table="jointable">
                  <!-- 该 key 元素的 column 值是连接表中的列名-->
                <key column="personId"/>
```

```
            <!--  list 元素必须的索引列-->
              <list-index column="addressOrder"/>
              <!-- 映射到持久化类 Address -->
        <many-to-many column="addressId" class="Address"/>
      </list>
    </class>
    <!-- 映射持久化类 Address-->
    <class name="Address">
        <!-- 该 key 元素的 column 值是连接中的列名-->
      <id name="addressid">
        <generator class="identity"/>
      </id>
        <!-- 映射普通属性-->
      <property name="addressdetail"/>
        <!-- 映射关系持久化类,table 属性的值必须与另一个持久化类的 table 属性值相同-->
        <!--  inverse 属性为 true 表示关系由对方控制-->
      <list name="persons" inverse="true" table="jointable">
            <!-- 该 key 元素的 column 值是连接表中的列名-->
            <key column="addressId"/>
              <!--  list 元素必须的索引列-->
                <list-index column="personOrder"/>
             <!-- 映射到持久化类 Person -->
        <many-to-many column="personId"   class="Person"/>
      </list>
   </class>
</hibernate-mapping>
```

本节提供三种关系映射的范例，但由于篇幅原因，其他的关系映射不可能一一列出，请读者参考以上三种关系映射自行理解。

10.2.6 Hibernate 的 HQL 查询

数据查询指根据给定条件来选择记录。数据查询是持久化层必需的操作，也是 DAO 对象必需的基本功能。Hibernate 配备功能强大的查询语言，这种查询语言被称为 Hibernate Query Language（HQL），也就是 Hibernate 查询语言。

HQL 的关键字不区分大小写，但类名、属性名和各种类型的值对象都是区分大小写的。HQL 的语法看起来与 SQL 非常接近，但不要忘记 Hibernate 是面向对象的，因此 HQL 被设计成面向对象。所以，使用 HQL 查询过程中，操作对象是类、实例或属性；而不是表、列、行等概念。同时，HQL 完全可以理解如继承、多态和关联等面向对象的概念。

10.2.6.1 from 语句

Hibernate 最简单的查询语句如下：

```
from lee.Person;
```

该语句将查询到 Person 的全部实例。请注意：lee.Person 不是表名，而是持久化类名，POJO 和 XML 映射文件的代码不再提供，请读者参考上面内容。from 或的类名，还可以指定别名，如下所示：

from lee.Person as p

下面代码演示简单的 from 查询：

```
public class HqlQuery
{
    public static void main(String[] args)throws Exception
        {
            //实例化 HqlQuery 对象
          HqlQuery mgr = new HqlQuery();
            //调用 HQL 查询方法
          mgr.findPersons();
          mgr.findPersonsByHappenDate();
    }
        private void findPersons()
        {
            //通过工具类 HibernateUtil 的 currentSession 方法开始 Session
            Session sess = HibernateUtil.currentSession();
            //开始事务
            Transaction tx = sess.beginTransaction();
            //使用 Session 的 createQuery 方法创建查询对象
          Query q = sess.createQuery("from Person p");
            //执行 Query 对象的 list 方法，返回查询的全部实例
            q.list();
            //遍历结果集，返回全部的查询记录
          for (Iterator pit = pl.iterator() ; pit.hasNext(); )
          {
             Person p = ( Person )pit.next();
                    System.out.println(p.getName());
          }
            //提交事务
            tx.commit();
            //调用工具类的关闭 Session 的方法。
            HibernateUtil.closeSession();
        }
}
```

这段代码将可查询出 Person 的全部实例。正如前面所讲，所有的持久化操作都应在 Session 的管理下进行，Query 是 Session 执行 createQuery 的返回值。createQuery 的参数为 HQL 语句字符串。

Hibernate 执行持久化操作总需要打开 Session，因此，采用工具类 HibernateUtil 将打开 Session 和关闭 Session 封装起来。同时，由于 Session 不是线程安全对象，程序中采用 ThreadLocal 变量来保存 Session，保证每次操作的 Session 不会相互影响。下面给出 HibernateUtil 的源代码：

```
public class HibernateUtil
```

```
{
    public static final SessionFactory sessionFactory;
    //静态初始化块，在执行该类静态方法、构造类实例之前执行
   static
      {
      try
            {
        //采用默认的 hibernate.cfg.xml 来启动一个 Configuration 的实例
                Configuration configuration=new Configuration().configure();
                //由 Configuration 的实例来创建一个 SessionFactory 实例
          sessionFactory = configuration.buildSessionFactory();
        }
            catch (Throwable ex)
            {
          //打印异常
          System.err.println("Initial SessionFactory creation failed." + ex);
                //抛出新的业务异常
          throw new ExceptionInInitializerError(ex);
        }
   }
      //ThreadLocal 是隔离多个线程的数据共享，不存在多个线程之间共享资源，因此不再需要对线程同步
      public static final ThreadLocal session = new ThreadLocal();
      /**
       * 该方法用来获 Session 对象，如果该线程已有 Session 对象，则直接返回。
      * 否则创建新的 Session 对象返回。
       * @ return 返回绑定到该线程 Session 对象
       */
   public static Session currentSession() throws HibernateException
      {
            //直接从 ThreadLocal 里取得该线程的 Session
        Session s = (Session) session.get();
        //如果该线程还没有 Session，则创建一个新的 Session
        if (s == null)
            {
          s = sessionFactory.openSession();
          //将获得的 Session 变量存储在 ThreadLocal 变量里
          session.set(s);
        }
        return s;
   }
      /**
```

```
     * 该方法用来关闭 Session 对象。
     */
    public static void closeSession() throws HibernateException
    {
        //获取绑定到该线程的 Session 对象。
        Session s = (Session) session.get();
        //如果该 Session 存在，关闭它。
        if (s != null) s.close();
        //同时将 ThreadLocal 值清空。
        session.set(null);
    }
}
```

10.2.6.2 where 子句确定条件

where 子句允许缩小返回的实例列表的范围。如果 from 后的持久化没有指定别名，可以使用属性名来直接引用属性。如果指定了别名，则必须使用完整的属性名，也就是采用别名限制属性名。看下面两行 HQL 语句：

```
from lee.Person where age > 6;
from lee.Person as p where p.age >6;
```

上面两行代码的效果完全相同。JDBC 里的 PreparedStatement 支持预编译，然后接受参数，通过这种方式，可以使用相同的 Statement 执行不同的 SQL 操作，提高执行的效率。Query 也支持预编译的方式，预编译的方式对 HQL 中无法确定的参数采用占位符，HQL 中的占位符可以指定名字，采用冒号（英文:）后加占位符名的方式来指定占位符名字。如下所示：

```
from lee.Person as p where p.age >:age;
```

Query 提供系列的 setXXX 方法，用来为不确定的参数传入实际的值。setXXX 方法提供两个版本，一种支持按参数索引传值，一种支持按参数名传值。如下所示：

- setXXX(int parameterIndex , Object parameterValue)
- setXXX(String parameterName , Object parameterValue)

给出带 where 字句，HQL 带占位符的查询示例：

```
public class HqlQuery
{
    public static void main(String[] args)throws Exception
    {
        //实例化 HqlQuery 对象
        HqlQuery mgr = new HqlQuery();
        //调用 HQL 查询方法
        mgr.findPersons();
        mgr.findPersonsByHappenDate();
    }
    private void findPersons()
    {
        //通过工具类 HibernateUtil 的 currentSession 方法开始 Session
```

```
            Session sess = HibernateUtil.currentSession();
            //开始事务
            Transaction tx = sess.beginTransaction();
            //使用 Session 的 createQuery 方法创建查询对象
        Query q = sess.createQuery("from Person as p where p.age > :age ")；
            //执行 Query 对象的 list 方法，返回查询的全部实例
            q.setInt(“age”,20);
            q.list();
            //遍历结果集，返回全部的查询记录
        for (Iterator pit = pl.iterator() ; pit.hasNext(); )
        {
           Person p = ( Person )pit.next();
                  System.out.println(p.getName());
        }
            //提交事务
            tx.commit();
            //调用工具类的关闭 Session 的方法。
            HibernateUtil.closeSession();
      }
}
```

10.2.6.3 select 子句

select 子句，用于选择将哪些对象和属性返回到结果集中，考虑如下情况：

```
//选所有人的儿子的另一种表达
select s
from Person as p
inner join p.son as s
```

该语句将选出所有人的儿子。事实上，可以用如下更简单的语句来表示

```
//选所有人的儿子的另一种表达
select p.son from Person p
```

select 语句可返回数据类型为任何类型的值，包括返回类型为组件的属性，甚至返回组件属性的属性值。如下所显示：

```
//选出人名字以 Ja 开头的人
select p.name from Person p
where p.name like 'Ja%'
//选择所有人的儿子的名字
//其中 son 属性为组件属性
select p.son.name from Person as p
```

select 语句可以返回多个对象和（或）属性，直接存放在 List 对象中

```
//将选择的人及其儿子放入一个集合
select new list(p,s)
from Person as p
    join p.son as s
```

也可能直接返回一个类型安全的 Java 对象,

```
//以选择的人及其儿子构造 Family 实例
//前提是 Family 类有构造函数 Family(Person p , Son s)
select new Family(p,s)
from Person as p
    join p.son as s
```

10.2.7 Hibernate 的 Criteria 查询

Hibernate 带一种更对象化的查询方式。这种查询方式更直观，而且可扩展。Criteria 查询也称为条件查询，条件查询使用 org.hibernate.Criteria 接口表示持久化类的一个查询，Session 是产生 Criteria 实例的工厂，创建 Criteria 时，需以持久化类的 class 作为参数。如下所示：

```
//以 Session 为工厂，创建持久化类 Person 的条件查询(Criteria)对象
Session.createCriteria(“Person.class”)
```

Criteria 使用组合器模式来增加查询条件，每条查询条件以 org.hibernate.criterion.Criterion 接口的一个实例表示，org.hibernate.criterion.Restrictions 类则作为 Criterion 实例的静态工厂类，该类提供如下系列静态方法来产生查询条件

- in(String propertyName, Collection values)：属性值在集合值里
- like(String propertyName, Object value)：属性值与目标值满足模式匹配
- lt(String propertyName, Object value)：属性值小于目标值
- le(String propertyName, Object value)：属性值小于等于目标值
- gt(String propertyName, Object value)：属性值大于目标值
- ge(String propertyName, Object value)：属性值大于等于目标值

看如下查询：

```
//以 Session 为工厂，创建 Person 类条件查询对象 cri
Criteria cri = Session.createCriteria(Person.class);
//为条件查询增加查询：name 以 Fritz 开始
cri.add(Restrictions.like("name", "Fritz%"));
//条件查询增加条件：age 大于 6
cri.add(Restrictions.gt("age", 6));
//查询出条件查询的所有实例，以 List 返回。
List personList = cri.list();
```

上文的查询将查询出 name 以 Fritz 开头，age 大于 6 的全部实例。上面的代码比 HQL 的查询更加直观，更加对象化。

10.3 整合 Hibernate

对 Hibernate，Spring 提供很多 IoC 的特性的支持，方便处理大部分典型的 Hibernate 整合的问题。所有的这些，都遵守 Spring 通用的事务和 DAO 异常体系。Spring 整合 Hibernatejiang，使持久层的访问更加容易，Spring 管理 Hibernate 持久层有如下优势：

- 通用的资源管理：Spring 的 ApplicationContext 能管理 SessionFactory，使得配置值很容易被管理和修改。无须使用 Hibernate 的配置文件。
- 有效的 Session 管理：Spring 提供了有效，简单和安全的 Hibernate Session 处理。

- 方便的事务管理：Hibernate 的事务管理处理不好，会限制 Hibernate 的表现，而 Spring 的声明式事务管理粒度是方法级。
- 异常包装。Spring 能够包装 Hibernate 异常，把它们从 checked exception 变为 runtime exception。开发者可选择在恰当的层处理数据库的不可恢复异常，从而避免繁琐的 catch/throw 以及异常声明。

10.4 管理 SessionFactory

SessionFactory：单个数据库映射关系编译后的内存镜像。大部分情况下，一个 J2EE 应用对应一个数据库。Spring 通过 ApplicationContext 管理 SessionFactory，无须采用单独 Hiberate 应用必需的 hibernate.cfg.xml 文件。

SessionFactory 与数据库的连接，都由 Spring 的配置管理。实际的 J2EE 应用，通常使用数据源，数据源会采用依赖注入的方式，传给 Hibernate 的 SessionFactory。具体配置如下所示：

```
<?xml version="1.0" encoding="gb2312"?>
<!-- Spring 配置文件的 DTD 定义-->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
   "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素是 beans-->
<beans>
      <!--定义数据源, 该 bean 的 ID 为 dataSource-->
      <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
            <!-- 指定数据库驱动-->
        <property name="driverClassName"><value>com.mysql.jdbc.Driver</value></property>
            <!-- 指定连接数据库的 URL-->
        <property name="url"><value>jdbc:mysql://wonder:3306/j2ee</value></property>
            <!-- root 为数据库的用户名-->
        <property name="username"><value>root</value></property>
            <!-- pass 为数据库密码-->
        <property name="password"><value>pass</value></property>
   </bean>
      <!--定义 Hibernate 的 SessionFactory-->
    <bean id="sessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
            <!-- 依赖注入数据源，注入正是上文定义的 dataSource>
        <property name="dataSource"><ref local="dataSource"/></property>
            <!-- mappingResouces 属性用来列出全部映射文件>
        <property name="mappingResources">
           <list>
              <!--以下用来列出所有的 PO 映射文件-->
                     <value>lee/MyTest.hbm.xml</value>
           </list>
```

```
        </property>
        <!--定义 Hibernate 的 SessionFactory 的属性 -->
        <property name="hibernateProperties">
            <props>
                    <!-- 指定 Hibernate 的连接方言-->
                    <prop
key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
                    <!-- 不同数据库连接，启动时选择 create,update,create-drop-->
                <prop key="hibernate.hbm2ddl.auto">update</prop>
            </props>
        </property>
    </bean>
</beans>
```

SessionFactory 由 ApplicationContext 管理，会随着应用启动时候自动加载。SessionFactory 可以被处于 ApplicaionContext 管理的任意一个 bean 引用，比如 DAO。Hibernate 的数据库访问需要在 Session 管理下，而 SessionFactory 是 Session 的工厂。Spring 采用依赖注入为 DAO 对象注入 SessionFactory 的引用。

Spring 更提供 Hibernate 的简化访问方式，Spring 采用模板设计模式，提供 Hibernate 访问与其他持久层访问的一致性。如果需要使用容器管理的数据源，则无须提供数据驱动等信息，只需要提供数据源的 JNDI 即可。对上文的 SessionFactory 无须任何修改，只需将 dataSource 的配置替换成 JNDI 数据源，将原有的 dataSource Bean 替换成如下所示：

```
<!-- 此处配置 JNDI 数据源-->
<bean id="myDataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName">
            <!-- 指定数据源的 JNDI-->
            <value>java:comp/env/jdbc/myds</value>
        </property>
</bean>
```

10.5 Spring 对 Hibernate 的简化

Hibernate 的持久层访问必须按如下步骤进行：

（1）创建 Configuration 实例

（2）创建 SessionFactory 实例

（3）创建 Session 实例

（4）打开事务

（5）开始持久化访问

（6）提交事务

（7）如果遇到异常，回滚事务

（8）关闭 Session

在 HQL 查询一节，已经采用 HibernateUtils 工具类封装部分过程。但依然不够简洁，需要通过代码显式地打开 Session，显式地开始事务，然后关闭事务，关闭 Session。而 Hibernate 提供更简单的方式操作持久层，无须显式地打开 Session，无须在代码中执行任何的事务操作语句。

对 Hibernate 的简化，还基于 Spring 对异常处理的简化。底层数据库异常几乎都不可恢复，强制处理底层数据库几乎没有任何意义，但传统 JDBC 数据库访问的异常都是 checked 异常，必须使用 try...catch 块处理。

Spring 包装了 Hibernate 异常，转换到 DataAccessException 继承树内，所有 DataAccessException 全部是 runtime 异常，并不强制捕捉。归纳起来，Spring 对 Hibernate 的简化主要有如下几个方面：

- 基于依赖注入的 SessionFactory 管理机制。SessionFactory 是执行持久化操作的核心组件。传统 Hibernate 应用中，SessionFactory 必须手动创建；通过依赖注入，代码无须关心 SessionFactory，SessionFactory 的创建，维护由 BeanFactory 负责管理。
- 更优秀的 Session 管理机制。Spring 提供“每事务一次 Session”的机制，该机制能大大提高系统性能，而且 Spring 对 Session 的管理是透明的，无须在代码中操作 Session。
- 统一的事务管理。无论是编程式事务，还是声明式事务，Spring 都提供一致的编程模型，无须繁琐的开始事务，显式提交、回滚。如果使用声明式事务管理，事务管理逻辑与代码分离，事务可在全局事务和局部事务之间切换。
- 统一的异常处理机制。不再强制开发者在持久层捕捉异常，持久层异常被包装成 DataAccessException 异常的子类，开发者可以自己决定在合适的层处理异常，将底层数据库异常包装成业务异常。
- HibernateTemplate 支持类。HibernateTempate 能完成大量 Hibernate 持久层操作，这些操作大多只需一行代码，非常简洁。

10.6 使用 HbernateTemplate

HibernateTemplate 提供持久层访问模板化，使用 HibernateTemplate 无须实现特定接口，它只需要提供一个 SessionFactory 的引用，就可执行持久化操作。SessionFactoyr 对象可通过构造参数传入，或通过设值方式传入。如下：

```
//获取 Spring 上下文
ApplicationContext ctx = new FileSystemXmlApplicationContext("bean.xml");
//通过上下文获得 SessionFactory
SessionFactory sessionFactory = (SessionFactory) ctx.getBean(“sessionFactory”);
```

然后创建 HibernateTemplate 实例。HibernateTemplate 提供如下三个构造函数

- HibernateTemplate()
- HibernateTemplate(org.hibernate.SessionFactory sessionFactory)
- HibernateTemplate(org.hibernate.SessionFactory sessionFactory, boolean allowCreate)

第一个构造函数，构造一个默认的 HibernateTemplate 实例，因此，使用 HibernateTemplate 实例之前，还必须使用方法 setSessionFactory(SessionFactory sessionFactory) 来为 HibernateTemplate 传入 SessionFactory 的引用。

第二个构造函数，在构造时已经传入 SessionFactory 引用。

第三个构造函数，其 boolean 型参数表明：如果当前线程已经存在一个非事务性的 Session，是否直接返回此非事务性的 Session。

对于在 Web 应用，通常启动时自动加载 ApplicationContext，SessionFactory 和 DAO 对象都处在 Spring 上下文管理下，因此无须在代码中显式设置，可采用依赖注入解耦 SessionFactory 和 DAO，依赖关系通过配置文件来设置，如下所示：

```
<?xml version="1.0" encoding="gb2312"?>
<!-- Spring 配置文件的 DTD 定义-->
```

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<!--  Spring 配置文件的根元素是 beans-->
<beans>
        <!--定义数据源,该 bean 的 ID 为 dataSource-->
        <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
              <!-- 指定数据库驱动-->
          <property name="driverClassName"><value>com.mysql.jdbc.Driver</value></property>
              <!-- 指定连接数据库的 URL-->
          <property name="url"><value>jdbc:mysql://wonder:3306/j2ee</value></property>
              <!--  root 为数据库的用户名-->
          <property name="username"><value>root</value></property>
              <!--  pass 为数据库密码-->
          <property name="password"><value>pass</value></property>
    </bean>
        <!--定义 Hibernate 的 SessionFactory-->
     <bean id="sessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
              <!-- 依赖注入数据源，注入正是上文定义的 dataSource>
          <property name="dataSource"><ref local="dataSource"/></property>
              <!--  mappingResouces 属性用来列出全部映射文件>
          <property name="mappingResources">
             <list>
               <!--以下用来列出所有的 PO 映射文件-->
                        <value>lee/Person.hbm.xml</value>
             </list>
          </property>
         <!--定义 Hibernate 的 SessionFactory 的属性 -->
          <property name="hibernateProperties">
                <props>
                        <!-- 指定 Hibernate 的连接方言-->
                         <prop
key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
                        <!-- 不同数据库连接，启动时选择 create,update,create-drop-->
                   <prop key="hibernate.hbm2ddl.auto">update</prop>
                </props>
       </property>
     </bean>
     <!-- 配置 Person 持久化类的 DAO bean-->
     <bean id="personDao" class="lee.PersonDaoImpl">
             <!-- 采用依赖注入来传入 SessionFactory 的引用>
```

```
            <property name="sessionFactory"><ref local="sessionFactory"/></property>
    </bean>
</beans>
```

DAO 实现类中，可采用更简单的方式来取得 HibernateTemplate 的实例。代码如下：

```
public class PersnDAOImpl implements PersonDAO
{
   //以私有的成员变量来保存 SessionFactory。
     private SessionFactory sessionFactory;
      //设值注入 SessionFactory 必需的 setter 方法
   public void setSessionFactory(SessionFactory sessionFactory)
     {
        this.sessionFactory = sessionFactory;
   }

   public List loadPersonByName(final String name)
     {
        HibernateTemplate hibernateTemplate =
              new HibernateTemplate(this.sessionFactory);
            //此处采用 HibernateTemplate 完成数据库访问
   }
}
```

10.6.1 HibernateTemplate 的常规用法

HibernateTemplate 提供非常多的常用方法来完成基本的操作，比如通常的增加、删除、修改、查询等操作，Spring 2.0 更增加对命名 SQL 查询的支持，也增加对分页的支持。大部分情况下，使用 Hibernate 的常规用法，就可完成大多数 DAO 对象的 CRUD 操作。下面是 HibernateTemplate 的常用方法简介：

- void delete(Object entity)：删除指定持久化实例
- deleteAll(Collection entities)：删除集合内全部持久化类实例
- find(String queryString)：根据 HQL 查询字符串来返回实例集合
- findByNamedQuery(String queryName)：根据命名查询返回实例集合
- get(Class entityClass, Serializable id)：根据主键加载特定持久化类的实例
- save(Object entity)：保存新的实例
- saveOrUpdate(Object entity)：根据实例状态，选择保存或者更新
- update(Object entity)：更新实例的状态，要求 entity 是持久状态
- setMaxResults(int maxResults)：设置分页的大小

下面是一个完整 DAO 类的源代码：

```
public class PersonDAOHibernate implements PersonDAO
{
   //采用 log4j 来完成调试时的日志功能
     private static Log log = LogFactory.getLog(NewsDAOHibernate.class);
   //以私有的成员变量来保存 SessionFactory。
```

```
  private SessionFactory sessionFactory;
  //以私有变量的方式保存 HibernateTemplate
  private HibernateTemplate hibernateTemplate = null;
   //设值注入 SessionFactory 必需的 setter 方法
public void setSessionFactory(SessionFactory sessionFactory)
  {
     this.sessionFactory = sessionFactory;
}
   //初始化本 DAO 所需的 HibernateTemplate
  public HIbernateTemplate getHibernateTemplate()
  {
        //首先，检查原来的 hibernateTemplate 实例是否还存在
        if ( hibernateTemplate == null)
        {
              //如果不存在，新建一个 HibernateTemplate 实例
              hibernateTemplate = new HibernateTemplate(sessionFactory);
        }
        return hibernateTemplate;
  }
   //返回全部的人的实例
public List getPersons()
   {
         //通过 HibernateTemplate 的 find 方法返回 Person 的全部实例
     return getHibernateTemplate().find("from Person");
}
   /**
    * 根据主键返回特定实例
    * @ return 特定主键对应的 Person 实例
    * @ param 主键值
public News getNews(int personid)
   {
     return (Person)getHibernateTemplate().get(Person.class, new Integer(personid));
}
   /**
    * @ person 需要保存的 Person 实例
    */
public void savePerson(Person person)
   {
         getHibernateTemplate().saveOrUpdate(person);
}
   /**
    * @ param personid 需要删除 Person 实例的主键
```

```
     * /
    public void removePerson(int personid)
      {
          //先加载特定实例
        Object p = getHibernateTemplate().load(Person.class, new Integer(personid));
          //删除特定实例
        getHibernateTemplate().delete(p);
    }
}
```

10.6.2 Hibernate 的复杂用法 HibernateCallback

HibernateTemplate 还提供一种更加灵活的方式来操作数据库，通过这种方式可以完全使用 Hibernate 的操作方式。HibernateTemplate 的灵活访问方式是通过如下两个方法完成：

- Object execute(HibernateCallback action)
- List execute(HibernateCallback action)

这两个方法都需要一个 HibernateCallback 的实例，HibernateCallback 实例可在任何有效的 Hibernate 数据访问中使用。程序开发者通过 HibernateCallback，可以完全使用 Hibernate 灵活的方式来访问数据库，解决 Spring 封装 Hibernate 后灵活性不足的缺陷。HibernateCallback 是一个接口，该接口只有一个方法 doInHibernate(org.hibernate.Session session)，该方法只有一个参数 Session。

通常，程序中采用实现 HibernateCallback 的匿名内部类来获取 HibernateCallback 的实例，方法 doInHibernate 的方法体就是 Spring 执行的持久化操作。具体代码如下：

```
public class PersonDaoImpl implements PersonDao
{
  //私有实例变量保存 SessionFactory
     private SessionFactory sessionFactory;
     //依赖注入必须的 setter 方法
  public void setSessionFactory(SessionFactory sessionFactory)
     {
       this.sessionFactory = sessionFactory;
  }
     /**
      * 通过人名查找所有匹配该名的 Person 实例
      * @param name 匹配的人名
      * @return 匹配该任命的全部 Person 集合
      */
  public List findPersonsByName(final String name)
     {
       //创建 HibernateTemplate 实例
          HibernateTemplate hibernateTemplate =
              new HibernateTemplate(this.sessionFactory);
       //返回 HibernateTemplate 的 execute 的结果
```

```
        return (List) hibernateTemplate.execute(
      //创建匿名内部类
        new HibernateCallback()
        {
    public Object doInHibernate(Session session) throws HibernateException
            {
        //使用条件查询的方法返回
                List result = session.createCriteria(Person.class)
                                .add(Restrictions.like("name", name+"%")
                                .list();
                return result;
          }
      });
  }
}
```

注意：方法 doInHibernate 方法内可以访问 Session，该 Session 对象是绑定到该线程的 Session 实例。该方法内的持久层操作，与不使用 Spring 时的持久层操作完全相同。这保证对于复杂的持久层访问，依然可以使用 Hibernate 的访问方式。

10.7 Hibernate 的 DAO 实现

DAO 对象是模块化的数据库访问组件，DAO 对象通常包括：对持久化类的基本 CRUD 操作（插入、查询、更新、删除）操作。Spring 对 Hibernate 的 DAO 实现提供了良好的支持。主要有如下两种方式的 DAO 实现：

- 继承 HibernateDaoSupport 的实现 DAO
- 基于 Hibernate3.0 实现 DAO

不管采用哪一种实现，这种 DAO 对象都极好地融合到 Spring 的 ApplicationContext 中，遵循依赖注入模式，提高解耦。

10.7.1 继承 HibernateDaoSupport 实现 DAO

Spring 为 Hibernate 的 DAO 提供工具类：HibernateDaoSupport。该类主要提供如下两个方法，方便 DAO 的实现：

- public final HibernateTemplate getHibernateTemplate()
- public final void setSessionFactory(SessionFactory sessionFactory)

其中，setSessionFactory 方法用来接收 Spring 的 ApplicationContext 的依赖注入，可接收配置在 Spring 的 SessionFactory 实例，getHibernateTemplate 方法则用来根据刚才的 SessionFactory 产生 Session，最后生成 HibernateTemplate 来完成数据库访问。

典型的继承 HibernateDaoSupport 的 DAO 实现的代码如下：

```
public class PersonDAOHibernate extends HibernateDaoSupport implements PersonDAO
{
```

```
//采用 log4j 来完成调试时的日志功能
  private static Log log = LogFactory.getLog(NewsDAOHibernate.class);
   //返回全部的人的实例
public List getPersons()
   {
         //通过 HibernateTemplate 的 find 方法返回 Person 的全部实例
     return getHibernateTemplate().find("from Person");
}
   /**
    * 根据主键返回特定实例
    * @ return 特定主键对应的 Person 实例
    * @ param 主键值
public News getPerson(int personid)
   {
     return (Person)getHibernateTemplate().get(Person.class, new Integer(personid));
}
   /**
    * @ person 需要保存的 Person 实例
    */
public void savePerson(Person person)
   {
         getHibernateTemplate().saveOrUpdate(person);
}
   /**
    * @ param personid 需要删除 Person 实例的主键
    * /
public void removePerson(int personid)
   {
```

```
        //先加载特定实例
        Object p = getHibernateTemplate().load(Person.class, new Integer(personid));
        //删除特定实例
        getHibernateTemplate().delete(p);
    }
}
```

可以与前面的 PersonDAOHibernate 对比，会发现代码量大大减少。事实上，DAO 的实现依然借助于 HibernateTemplate 的模板访问方式，只是，HibernateDaoSupport 将依赖注入 SessionFactory 的工作已经完成，获取 HibernateTemplate 的工作也已完成。该 DAO 的配置必须依赖于 SessionFactory，具体的配置如下：

```
<?xml version="1.0" encoding="gb2312"?>
<!-- Spring 配置文件的 DTD 定义-->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
   "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素是 beans-->
<beans>
    <!--定义数据源,该 bean 的 ID 为 dataSource-->
    <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <!-- 指定数据库驱动-->
        <property name="driverClassName"><value>com.mysql.jdbc.Driver</value></property>
        <!-- 指定连接数据库的 URL-->
        <property name="url"><value>jdbc:mysql://wonder:3306/j2ee</value></property>
        <!-- root 为数据库的用户名-->
        <property name="username"><value>root</value></property>
        <!-- pass 为数据库密码-->
        <property name="password"><value>pass</value></property>
    </bean>
```

```
    <!--定义 Hibernate 的 SessionFactory-->
  <bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactor
yBean">
        <!-- 依赖注入数据源，注入正是上文定义的 dataSource>
    <property name="dataSource"><ref local="dataSource"/></property>
        <!-- mappingResouces 属性用来列出全部映射文件>
    <property name="mappingResources">
      <list>
        <!--以下用来列出所有的 PO 映射文件-->
            <value>lee/Person.hbm.xml</value>
      </list>
    </property>
   <!--定义 Hibernate 的 SessionFactory 的属性 -->
    <property name="hibernateProperties">
        <props>
            <!-- 指定 Hibernate 的连接方言-->
             <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</pro
p>
            <!-- 不同数据库连接，启动时选择 create,update,create-drop-->
          <prop key="hibernate.hbm2ddl.auto">update</prop>
        </props>
   </property>
  </bean>
  <!-- 配置 Person 持久化类的 DAO bean-->
  <bean id="personDAO" class="lee. PersonDAOHibernate">
        <!-- 采用依赖注入来传入 SessionFactory 的引用>
        <property name="sessionFactory"><ref local="sessionFactory"/></property>
  </bean>
```

</beans>

程序中可以通过显式的编码来获得 personDAO bean，然后执行 CRUD 操作。也可通过依赖注入，将 personDAO 的实例注入其他 bean 属性，再执行 CRUD 操作。

在继承 HibrnateDaoSupport 的 DAO 实现里，Hibernate Session 的管理完全不需要 Hibernate 代码打开，而由 Spring 来管理。Spring 会根据实际的操作，采用“每次事务打开一次 session”的策略，自动提高数据库访问的性能。

10.7.1 基于 Hibernate3.0 实现 DAO

Hibernate3.0.1 提供了一种新的技术："contextual Sessions"。通过此机制，Hibernate 可以自己管理 Session，从而保证每次事务一个 Session。该机制类似于 Spring 的每次事务一次 Hibernate Session 的同步策略。

Hibernate 的"contextual Sessions"，是通过 SessionFactory 的 getCurrentSession()方法实现的，该方法会返回由当前 JTA 事务保持的 Session，如果当前 JTA 事务关联的 Session 不存在，系统打开一次新的 Session，并关联到当前的 JTA 事务，如果当前 JTA 事务关联的 Session 已经存在，则直接返回该 Session 即可。执行该操作的前提是 Hibernate 处于事务管理下。通常，Spring 为 Hibernate 提供事务管理。

基于 Hibernate 3.0 的 DAO 的实现，只需 Spring 注入 SessionFactory，然后由 Hibernate 自己管理 Session。即：通过 SessionFactory 的 getCurrentSession 方法，返回当前事务关联的 Session。持久化操作在 Session 管理如常进行。完整的基于 Hibernate3.0 的 DAO 实现的代码如下

```
public class PersonDaoImpl implements PersonDao
{
    //私有成员变量保存 SessionFactory
    private SessionFactory sessionFactory;
    /**
     * 依赖注入 SessionFactory 必须的 setter 方法
     * @ sessionFactory
     */
     public void setSessionFactory(SessionFactory sessionFactory)
     {
          this.sessionFactory = sessionFactory;
  }

    /**
     * 根据名字查找 Person 的实例。
```

```
     * @param name 需要查找的 Person 的名字
     * @return 匹配名字的 Person 实例的集合
     */
    public Collection findPersonsByName(String name)
    {
        return this.sessionFactory.getCurrentSession()
                                  .createQuery("from lee.Person p where p.name=?")
                                  .setParameter(0, name)
                           .list();
    }
    /**
     * 根据 Person id 加再 Person 实例。
     * @param id 需要 load 的 Person 实例
     * @return 特定 id 的 Person 实例。
     */
    public Person findPersonsById(int id)
    {
        return (Person)this.sessionFactory.getCurrentSession()
            .load(Person.class,new Integer(id))
    }
}
```

该 DAO 的数据库访问方式，类似于传统的 Hibernate 的访问，区别在于获取 Session 的方式不同。传统的 Hibernate 的 SessionFactory，采用工具类 HibernateUtils 来保存成静态成员变量，每次采用 HibernateUtils 打开 Session。

传统的 Session 访问方式，很容易造成“每次数据库操作打开一次 Session”的方式，该方式效率低下，也是 Hibernate 不推荐采用的策略。Hibernate 推荐采用“每次事务打开一次 Session”。基于该原因，Hibernate3.0.1 提供"contextual Sessions”的技术，最终达到与继承 HibernateDaoSupport 的 DAO 实现相同的目的。

同样，此 DAO bean 也需要配置在 Spring 的上下文中，需要依赖于 SessionFactory bean。SessionFactory bean 由 Spring 在运行时动态为 DAO bean 注入。具体的配置文件，读者可参考上文的配置文件写出。

10.8 事务管理

Hibernate 建议所有的数据库访问都应放在事务内进行，即使只进行只读操作。事务又应该尽可能地短，长事务会导致系统长时间无法释放，因而降低系统并发的负载。Spring 同时支持编程式事务和声明式事务。尽量考虑使用声明式事务，声明式事务管理可分离业务逻辑和事务管理逻辑，具备良好的适应性。

10.8.1 编程式的事务管理

编程式事务管理建议使用 TransactionTemplate 来完成事务操作，TransactionTemplate 采用 Callback 避免让开发者重复书写打开事务，提交事务，回滚事务等代码，同时，TransactionTemplate 无须书写大量的 try...catch 块。

关于使用 TransactionTemplate 的示例，请参阅 7.4.1 使用 TransactionTemplate。需要是使用 Spring 的事务管理，首先需配置一个 PlatformTransactionManager bean。配置该 bean 时，应根据事务环境选择合适的实现类。下面示例是 Hibernate 局部事务管理的配置示例：

```
<?xml version="1.0" encoding="gb2312"?>

<!-- Spring 配置文件的 DTD 定义-->

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"

   "http://www.springframework.org/dtd/spring-beans.dtd">

<!-- Spring 配置文件的根元素是 beans-->

<beans>

      <!--定义数据源,该 bean 的 ID 为 dataSource-->

      <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataS

ource">

            <!-- 指定数据库驱动-->

        <property name="driverClassName"><value>com.mysql.jdbc.Driver</value></propert

y>

            <!-- 指定连接数据库的 URL-->

        <property name="url"><value>jdbc:mysql://wonder:3306/j2ee</value></property>

            <!-- root 为数据库的用户名-->

        <property name="username"><value>root</value></property>

            <!-- pass 为数据库密码-->

        <property name="password"><value>pass</value></property>

   </bean>
```

```
<!--定义 Hibernate 的 SessionFactory-->
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactor
yBean">
<!-- 依赖注入数据源，注入正是上文定义的 dataSource>
<property name="dataSource"><ref local="dataSource"/></property>
<!-- mappingResouces 属性用来列出全部映射文件>
<property name="mappingResources">
<list>
<!--以下用来列出所有的 PO 映射文件-->
<value>lee/MyTest.hbm.xml</value>
</list>
</property>
<!--定义 Hibernate 的 SessionFactory 的属性 -->
<property name="hibernateProperties">
<props>
<!-- 指定 Hibernate 的连接方言-->
<prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</pro
p>
<!-- 不同数据库连接，启动时选择 create,update,create-drop-->
<prop key="hibernate.hbm2ddl.auto">update</prop>
</props>
</property>
</bean>
<!-- 配置 Hibernate 的事务管理器 -->
<!-- 使用 HibernateTransactionManager 类，该类是 PlatformTransactionManager 接口
针对采用 Hibernate 持久化连接的特定实现。-->
<bean id="transactionManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
```

```
        <!-- HibernateTransactionManager bean 需要依赖注入一个 SessionFactory bean
的引用-->
        <property name="sessionFactory"><ref local="sessionFactory"/></property>
    </bean>
</beans>
```

下面是采用 TransactionTemplate 和 HibernateTemplate 的事务操作代码：

```
public class TransactionTest
{
    public static void main(String[] args)
    {
        //因为并未在 Web 应用中测试，故需要手动创建 Spring 的上下文
        final ApplicationContext ctx = new FileSystemXmlApplicationContext("bean.xml");
        //获得 Spring 上下文的事务管理器
        PlatformTransactionManager transactionManager=
            (PlatformTransactionManager)ctx.getBean("transactionManager");
        final SessionFactory sessionFactory = (SessionFactory)ctx.getBean("sessionFactory
");
            (PlatformTransactionManager)ctx.getBean("transactionManager");
        //以事务管理器实例为参数，创建 TransactionTemplate 对象
        TransactionTemplate tt = new TransactionTemplate(transactionManager);
        //设置 TransactionTemplate 的事务传播属性
        transactionTemplate.setPropagationBehavior(TransactionDefinition.PROPAGATION
_REQUIRED);
        //执行 TransactionTemplate 的 execute 方法，该方法需要 TransactionCallback 实例。
        tt.execute(new TransactionCallbackWithoutResult()
            //采用 TransactionCallbackWithoutResult 匿名内部类的形式执行
            {
                protected void doInTransactionWithoutResult(TransactionStatus ts)
```

```
                {
                    try
                    {
                        以 SessionFactory 实例为参数创建 HibernateTemplate
                     HibernateTemplate hibernateTemplate =
                            new HibernateTemplate(this.sessionFactory);
                        Person p1 = new Person("Jack");
                        //保存第一个实例
                        hibernateTemplate.save(p1);
                        //让下面的数据库操抛出异常即可看出事务效果。前面的操
作也不会生效
                        Person p2 = new Person("Black");
                        //保存第二个实例,可将 Person 的 name 属性设为标识属性,
并引起主键重复的异
                    //常，可看出前一条记录也不会进数据库
                        hibernateTemplate.save(p2);

                    }
                    catch (Exception e)
                    {
                        ts.setRollbackOnly();
                    }
                }
            });
        }
}
```

10.8.2 声明式的事务管理

通常建议采用声明式事务管理。

交叉参考：

声明式事务的配置，请参考 7.5 节 声明式事务的介绍。

关于声明式事务管理的配置方式，通常有如下三种：

- 使用 TransactionProxyFactoryBean 为目标 bean 生成事务代理的配置。此方式是最传统，配置文件最臃肿、难以阅读的方式。
- 采用 bean 继承的事务代理配置方式，比较简洁，但依然是增量式配置。
- 使用 BeanNameAutoProxyCreator，根据 bean name 自动生成事务代理的方式。

建议采用第三种配置方式，详细的配置代码如下：

```
<?xml version="1.0" encoding="gb2312"?>
<!-- Spring 配置文件的 DTD 定义-->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素是 beans-->
<beans>
        <!--定义数据源,该 bean 的 ID 为 dataSource-->
        <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataS
ource">
                <!-- 指定数据库驱动-->
          <property name="driverClassName"><value>com.mysql.jdbc.Driver</value></propert
y>
                <!-- 指定连接数据库的 URL-->
          <property name="url"><value>jdbc:mysql://wonder:3306/j2ee</value></property>
                <!-- root 为数据库的用户名-->
          <property name="username"><value>root</value></property>
                <!-- pass 为数据库密码-->
          <property name="password"><value>pass</value></property>
    </bean>
        <!--定义 Hibernate 的 SessionFactory-->
```

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactor
yBean">
        <!-- 依赖注入数据源，注入上面定义的 dataSource>
    <property name="dataSource"><ref local="dataSource"/></property>
        <!-- mappingResouces 属性用来列出全部映射文件>
    <property name="mappingResources">
        <list>
            <!--以下用来列出所有的 PO 映射文件-->
                    <value>lee/Person.hbm.xml</value>
        </list>
    </property>
    <!--定义 Hibernate 的 SessionFactory 的属性 -->
    <property name="hibernateProperties">
            <props>
                    <!-- 指定 Hibernate 的连接方言-->
                     <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</pro
p>
                    <!-- 不同数据库连接，启动时选择 create,update,create-drop-->
                <prop key="hibernate.hbm2ddl.auto">update</prop>
            </props>
    </property>
</bean>
    <!-- 配置 Hibernate 的事务管理器 -->
    <!-- 使用 HibernateTransactionManager 类，该类是 PlatformTransactionManager 接口。
        针对采用 Hibernate 持久化连接的特定实现。-->
    <bean id="transactionManager"
      class="org.springframework.orm.hibernate3.HibernateTransactionManager">
```

```
    <!-- HibernateTransactionManager bean 需要依赖注入一个 SessionFactory bean 的引用-
->
        <property name="sessionFactory"><ref local="sessionFactory"/></property>
    </bean>
    <!-- 定义事务拦截器 bean>
     <bean id="transactionInterceptor"
    class="org.springframework.transaction.interceptor.TransactionInterceptor">
        <!-- 事务拦截器 bean 需要依赖注入一个事务管理器>
       <property name="transactionManager" ref="transactionManager"/>
        <property name="transactionAttributes">
            <!-- 下面定义事务传播属性-->
            <props>
                <prop key="insert*">PROPAGATION_REQUIRED </prop>
                <prop key="find*">PROPAGATION_REQUIRED,readOnly</prop>
                <prop key="*">PROPAGATION_REQUIRED</prop>
            </props>
        </property>
     </bean>
    <!—定义一个 BeanPostProcessor bean
    Spring 提供 BeanPostProcessor 的实现类 BeanNameAutoProxyCreator-->
     <bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreato
r">
        <!-- 指定对满足哪些 bean name 的 bean 自动生成业务代理 -->
        <property name="beanNames">
            <!-- 此处还可以列出更多的需要生成事务代理的目标 bean-->
             <ref local=" personDAO "/>
            <!-- 有一个需要生成事务代理的目标 bean, 此处就增加一行-->
        </property>
```

```
        <!-- 下面定义 BeanNameAutoProxyCreator 所需要的事务拦截器-->
        <property name="interceptorNames">
            <list>
                <value>transactionInterceptor</value>
            <!-- 此处还增加其他新的 Interceptor -->
            </list>
        </property>
    </bean>
    <bean id="personDAO" class="lee.PersonDAOImpl" />
    <!—此处还可增加更多 bean 定义-->
</beans>
```

TranscationInterceptor 是一个事务拦截器 bean，需要传入一个 TransactionManager 的引用。配置中使用 Spring 依赖注入该属性，事务拦截器的事务属性通过 transactionAttributes 来指定，该属性有 props 子元素，配置文件中定义了三个事务传播规则：

所有以 insert 开始的方法，采用 PROPAGATION_REQUIRED 的事务传播规则。程序抛出 MyException 异常及其子异常时，自动回滚事务。所有以 find 开头的方法，采用 PROPAGATION_REQUIRED 事务传播规则，并且只读。其他方法，则采用 PROPAGATION_REQUIRED 的事务传播规则。

BeanNameAutoProxyCreator 是个根据 bean 名生成自动代理的代理创建器，该 bean 通常需要接受两个参数。第一个是 beanNames 属性，该属性用来设置哪些 bean 需要自动生成代理。另一个属性是 interceptorNames，该属性则指定事务拦截器，自动创建事务代理时，系统会根据这些事务拦截器的属性来生成对应的事务代理。

10.8.3 事务策略的思考

考虑采用 Spring 的事务管理机制。Spring 的事务管理都是通过 PlatformTransactionManager 完成。在 Hibernate 应用中，PlatformTransactionManager 可能是 Hibernat SessionFactory，也可能是 JtaTransactionManager。前者是采用局部事务管理的实现，后者是采用基于 JTA 支持的全局事务管理的实现。因此，即使应用运行于支持 JTA 事务的应用服务器环境，考虑使用 Spring 的声明式事务管理也是个不错的选择。

10.9 小结

本章简单介绍了 ORM 相关知识。详细介绍了 Hibernate 的基本使用、简单的配置，以及关系映射配置。也详细介绍了 Hibernate 的两种常用的查询方法：HQL 查询和条件查询。重点介绍 Spring 对 Hibernate 的简化和支持，以及 Spring 整合 Hibernate 的各种方式。最后，给出了 Hibernate 在 Spring 中的两种事务管理示例程序及相关说明。

本章要点

远程访问的发展和意义

WebSerivce 的作用

RMI 的用法以及整合 Spring 的用法

Axis 的用法以及整合 Spring 的用法

Hessian 的用法以及整合 Spring 的用法

Spring 中 HTTP Invoker 的用法

分布计算是计算机界由来已久的话题。随着网络技术飞速发展，面向对象的技术成熟，现在，分布计算具有跨越网络，跨平台的能力，使其成为新一代计算技术的主流。

Java 语言的跨平台性、可移植性使其能开发网络应用。同时，Java 语言内建的远程方法调用（RMI）特性，允许客户机直接调用服务器上对象的方法，更是简化了分布式应用的开发。J2EE 架构是面向分布式应用设计的。它的许多技术：JNDI，RMI，EJB 等，无一不是面向分布式的应用。所有的分布式应用都涉及到一个共同的问题：远程访问。

14.1 远程访问简介

远程访问是分布式应用基础。远程访问允许客户机访问其他计算机的对象和服务，远程访问也是 J2EE 架构的基础。

14.1.1 远程访问的意义

远程访问指网络中的一台计算机，可以使用另一台计算机的服务、对象或方法，效果就像使用本地的服务、对象或方法一样。通过远程访问，程序可以在物理位置不同的机器上运行。从而实现分布式计算。

大型的企业级应用面临着更多的挑战：库存信息存储在一个程序中，而订单系统又存储在另一个应用中，而客户信息则存储在第三个应用中。如果公司需要将这三个部分集成一个系统，分布式开发则必不可少。因而必须使用远程服务。

- 远程访问可提供多个计算单元的协同服务。数据库 Server 专门提供数据库服务，而 Web 服务器则专门提供 Web 服务，为了性能的优化，不通的逻辑计算单元会分布在不同的物理节点上，通过远程访问，可以使多个计算单元协同服务。
- 远程访问能提供更好的可靠性和可用性。计算机的出现故障的可能性时刻存在：物理的故障、各种突发事件，或者系统的突然宕机，都会导致系统不可用。如果没有处于分布式应用中，某一个物理节点出现问题时，整个系统都不可用。但如果包含远程访问支持，出现问题的节点可以自动 failover 到另一个正常运行的节点上。从而保证整个系统的正常运行。
- 远程访问能提供更好的透明性。当一个节点访问另一个节点的方法或服务时，无须关心远程节点的具体实现，远程节点的实现对于调用者完全透明。
- 远程访问能提供更好的开放性和可伸缩性。远程访问通常面向接口编程，远程服务只需向外提供服务接口、客户机的调用面向接口。因此，提供了更好的可伸缩性。

14.1.2 常用的远程访问技术

远程访问作为现在的主流技术正在蓬勃发展，目前常见的远程访问技术有如下：

- Java 所提供的内建的远程访问技术：Remote Method Invocation（RMI），也就是远程方法调用。远程方法调用，允许 Java 程序调用网络中另一台机器的 Java 方法，仿佛那个方法就在本地机器上一样。
- Hsession：一个轻量级的 Java 远程访问的解决方案。Hessian 很像 WebService，只不过它不使用 SOAP 协议，而是使用它自己的 binary 协议。Hessian 的 server 端提供一个 Servlet 基类，client 端获得一个 Service 接口（也就是 stub）之后调用上面的方法，stub 将方法调用 marshal 之后，通过 HTTP 传到 server，server 借助 reflection 调用 Service 方法。
- EJB：经典的、重量级的远程访问技术。通过 Remote 接口提供自己的业务服务，使用 JNDI 定位远程服务。
- Burlap：是利用 XML RPC 协议的远程访问技术，也是一种轻量级的实现。利用 Burlap WebService 协议不需要大型的框架，也不用学习其它协议。
- JAX-RPC：表示 XML 的远程调用。从 J2EE1.4 开始引进，也是 J2EE1.4 WebService 的核心技术。

14.2 WebService 简介

WebService 使用基于 XML 的消息处理，作为基本的数据通讯方式，消除使用不同组件模型、操作系统和编程语言之间存在的差异，使异构系统能作为单个计算机网络协同运行。WebService 建立在一些通用协议的基础上，如 HTTP，SOAP，XML，WSDL，UDDI 等。这些协议在涉及到操作系统、对象模型和编程语言时，没有任何倾向，因此具备很强的生命力。

14.2.1 WebService 的特点

WebService 的主要目标是跨平台。为了达到这一目标，WebService 完全基于 XML（可扩展标记语言）、XSD（XMLSchema）等独立于平台、独立于软件供应商的标准，是创建可互操作的、分布式应用程序的新平台。WebService 主要用于如下三个方面：

14.2.1.1 远距离的通信

如果应用程序有成千上万的用户，而且分布在世界各地，最关键的是运行在完全不同的平台上。各个组件之间的通信是个非常棘手的问题。因为，客户端和服务器之间通常会有防火墙或者代理服务器，并且程序运行在不同的平台上。

传统的解决方案是通过 Web server 对外提供服务，比如 Servlet 技术，但 Servlet 花费太多的精力生成界面，组件之间的通信完全不需要界面部分，而是依赖于服务的互相引用。

将中间层组件换成 WebService，组件可以直接调用中间层组件，省掉建立 Servlet 的表现层。由 WebService 组成的中间层，完全可以在异构系统整合或其它场合下重用。最后，通过 WebService 把应用的逻辑和数据“暴露”出来，可以让其它平台上的客户重用这些应用。

14.2.1.2 应用程序集成

企业里经常需要整合不同语言的、在不同平台上运行的应用程序，这种整合将花费很大的开发力量。应用程序经常需要从 IBM 主机上获取数据，或者把数据发送到 UNIX 应用程序中去。即使在同一个平台上，不同软件厂商的各种软件也常常需要整合。通过 WebService，应用程序可以以标准方式“暴露”功能和数据，供其他应用程序使用。

例如，面对这样的订单登记应用，用于登记客户发来的新订单，包括客户信息、发货地址、数量、价格和付款方式等内容；还有一个订单执行程序，用于管理实际的货物发送。而

两个应用程序来自不同软件厂商。新订单进入系统之后，订单登记程序应通知订单执行程序发送货物。

这样，可以在订单执行程序包装一层 WebService，将“AddOrder”函数“暴露”出来。订单登记程序可以调用这个函数来发送货物，而无须理会该程序的运行平台、具体实现等细节。

14.2.1.3 B2B 的整合

WebService 可用来整合应用，从而使公司内部的商务处理更加智能化。整合跨公司业务交易成为 B2B 整合。

通过 WebService，公司可以把关键的商务应用“暴露”给指定的供应商和客户。例如，将电子下单系统和电子发票系统“暴露”出来，客户可以发送电子订单，供应商则可以发送电子采购发票。当然， WebService 只是 B2B 整合的关键部分，还需要许多其他部分才能实现整合。

用 WebService 来实现 B2B 整合好处在于：可以实现跨平台的互操作性。只要把商务逻辑“暴露”出来，成为 WebService，可以让任何指定的合作伙伴调用这些逻辑，不管他们的系统在什么平台上运行，使用什么开发语言。这样，可以大大减少 B2B 整合的时间和成本。

14.2.1.4 软件和数据重用

软件重用是一个很大的主题，重用的形式很多，重用的程度各有不同。基本的形式是源代码模块或者类一级的重用，另一种形式是二进制形式的组件重用。

当前的可重用的组件存在一个很大的限制：重用仅限于代码，数据不能重用。原因在于，发布组件甚至源代码都比较容易，但要发布数据就没那么容易，除非是不会经常变化的静态数据。

WebService 既允许重用代码，也允许重用的数据。通过 WebService，客户也不必像从前那样，要先从第三方购买、安装组件，再通过应用程序调用组件；应用程序可以直接调用远端的 WebService。WebService 提供商可以按时间或使用次数来对服务收费。这样的服务通过组件重用是不可能的，因为数据不可能与应用一起发布，即使一起发布，也不能实时更新。

另一种软件重用的情况是，整合多个应用程序的功能。例如，要建立门户站点应用，让用户既可以查看股市行情，又可以管理自己的日程安排，看可以浏览国际新闻，还可以在线购买电影票。现在 Web 上有很多供应商，在其应用中分别实现了这些功能。一旦他们把这些功能通过 WebService“暴露”出来，就可非常容易地将这些功能集成到你的门户站点中，为用户提供统一的、友好的界面。

总结起来，WebService 有如下特点：

- 封装性：WebService 是一种部署在 Web 应用上的对象，具备良好的封装性。对使用者而言，仅能看到服务描述，而该服务的具体实现、运行平台都是透明，调用者无须关心，也无法关心。WebService 作为整体提供服务。
- 松散耦合：当 WebService 的实现发生改变时，调用者是无法感受到这种改变的。对调用者而言，只要服务实现的接口没有变化，具体实现的改变是完全透明的。
- 使用标准协议：WebService 所有的公共协议都使用标准协议描述、传输和交换。这些标准协议在各种平台上完全相同。
- 高度整合的能力：由于 WebService 采用简单的、易理解的标准 Web 协议作为通信协议，完全屏蔽了不同平台的差异，无论是 CORBA、DCOM 还是 EJB，都可以通过这种标准的协议进行互操作，实现系统的最高可整合性。

- 高度的开放性：WebService 可以与其他的 WebService 进行交互，具有语言和平台无关性，支持 CORBA，EJB，DCOM 等多种组件标准，支持各种通讯协议如：HTTP，SMTP，FTP 和 RMI 等。

14.2.2 WebService 的主要技术

WebService 建立在一些技术标准上，设计的主要技术 SOAP（Simple Object Access Protocol 简单对象访问协议），WSDL（WebService Description Language WebService 描述语言），UDDI（Universal Description，Description and Integration 统一描述、发现和整合协议）。

14.2.2.1 SOAP（简单对象访问协议）

SOAP（Simple Object Access Protocol 简单对象访问协议）是 WebService 的根本。它是一种具有扩展性的 XML 消息协议。SOAP 允许一个应用程序向另一个应用程序发送 XML 消息，SOAP 消息是从 SOAP 发送者传至 SOAP 接收者的单路消息，任何应用程序均可作为发送者或接收者。SOAP 仅定义消息结构和消息处理的协议，与底层的传输协议独立。因此，SOAP 协议能通过 HTTP，JMS 或 SMTP 协议传输。目前，大多采用 HTTP 传输 SOAP 消息。SOAP 包括如下三个部分：

- SOAP 封装结构：该结构定义消息的整体框架，确定消息中的内容，内容的处理者，内容哪些部分是可选的，哪些部分是必需的。
- SOAP 编码规则：定义应用程序程序之间数据交换机制。
- SOAP RPC 表示：定义远程过程调用的应答的协议。

在 SAOP 封装、SOAP 编码规则和 SOAP RPC 协议之外，该规范还定义了两个协议绑定，分别确定在有 HTTP 扩展框架、无 HTTP 扩展框架的情况下，如何在 HTTP 消息中传输 SOAP 消息。

14.2.2.1 WSDL（WebService 描述语言）

WSDL（WebService Description Language WebService 描述语言）使用 XML 描述 WebService,包括访问和使用 WebService 所必需的信息，定义该 WebService 的位置、功能以及如何通信等描述信息。WSDL 文件包含如下三个部分：

- WHAT 部分：包括类型（types）、消息（messages）、portType 元素，它们定义了 客户端及服务器端交互的消息及数据类型。
- HOW 部分：包括 binding 元素，用于描述 WebService 的实现细节。binding 元素将 portType 绑定到一种特定的协议。
- WHERE 部分：包括 Service 元素，它将端口类型元素，binding 元素以及 WebService 实际的地址放在一起，可在 WSDL 文件的最后部分看到 Service 元素。

14.2.2.1 UDDI（统一描述、发现和整合协议）

UDDI（Universal Description，Description and Integration 统一描述、发现和整合协议）是一套信息注册规范，它具有如下特点：

- 基于 Web。
- 分布式。

UDDI 还包括一组使企业能将自身提供 WebService 注册，以使别的企业发现访问协议的实现标准。UDDI 的核心组件是 UDDI 商业注册，它使用 XML 文件来描述企业及其提供的 WebService。

通过使用 UDDI 的发现服务，企业可以注册 WebService，允许别的企业调用本身的 WebService。企业通过 UDDI 商业注册中心的 Web 界面，将这些 WebService 的信息加入 UDDI 商业注册中心。该 WebService 就可以被发现和调用。通过上面的介绍，WebService 的运行

模式如图 14.1 所示。

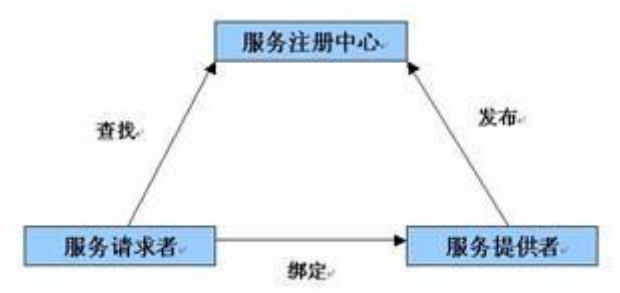

图 14.1 WebService 示意

14.3 Spring 对远程访问的支持

Spring 支持集成各种远程访问技术。使用 Spring 对远程访问技术的支持，可以降低开发远程访问服务的难度，同时支持将 POJO 暴露成远程服务。Spring 提供对下面四种远程访问技术的支持：

- 远程方法调用（RMI）：通过使用 RmiProxyFactoryBean 和 RmiServiceExporter，Spring 支持传统的 RMI 和通过 RMI 调用器的远程调用。
- Spring 的 HTTP 调用器：Spring 提供的一种特殊的远程访问策略，它支持任何 Java 接口，允许 Java 对象能通过 HTTP 传送。对应的支持类是 HttpInvokerProxyFactoryBean 和 HttpInvokerServiceExporter。
- Hessian：通过使用 HessianProxyFactoryBean 和 HessianServiceExporter，可使用 Caucho 提供的轻量级远程访问协议，提供远程服务。
- Burlap：Burlap 是基于 XML 的，它可以完全代替 Hessian。Spring 提供的支持类有 BurlapProxyFactoryBean 和 BurlapServiceExporter。
- JAX RPC：基于 WebService 的远程服务。

14.4.1 RMI 介绍

RMI 是 Remote Method Invocation（远程方法调用）的 所写。它允许一个 Java 程序调用网络中另一台计算机上的 Java 方法，就如调用本机的方法一样。实现 RMI 调用的程序和被调用的方法，都必须是 Java 代码，即客户端和服务器端都必须通过纯 Java 实现。

RMI 是基于 Java 的分布式编程模型，使用 RMI 进行远程方法调用时，无须考虑方法底层的网络传输细节。下面使用 RMI 的示例程序：

先编写 RMI 服务器端，RMI 需要通过远程接口“暴露”服务。也就是说，所有想被客户机调用的方法都必须在 Remote 接口里声明，否则无法完成调用。远程接口如下：

```
远程接口必须集成 java.rmi.Remote 接口
public interface Server extends Remote
{
    //所有在 Remote 接口里声明的方法都必须抛出 RemoteException 异常
   String helloWorld(String name) throws RemoteException;
   Person getPerson(String name,int age)throws RemoteException;
}
```

远程接口必须继承 java.rmi.Remote 接口。远程接口里声明的方法会通过网络传输，而网络是不可靠的，因此，所有的远程方法都必须抛出 RemoteException。RMI 服务是典型的

面向接口编程，只有在远程接口里定义的方法才会作为远程服务。

下面是远程服务提供类，远程服务提供类必须实现远程接口，并继承 java.rmi.server.UnicastRemoteObject 对象，继承该类能“暴露”远程服务。

```
//远程服务类，远程服务类必须继承 UnicastRemoteObject，并实现 Remote 接口
public class ServerImpl extends UnicastRemoteObject implements Server
{
    //远程服务类必须拥有构造器，且构造器必须抛出 RemoteException 异常
   public ServerImpl()throws RemoteException
   {
    }
    //实现 Remote 接口必须实现的方法
   public String helloWorld(String name)throws RemoteException
   {
       return name + ", 您好!";
   }
    //实现 Remote 接口必须实现的方法
   public Person getPerson(String name,int age)throws RemoteException
   {
       return new Person(name,age);
   }
     //下面是服务类的本地方法，不会“暴露”为远程服务。
     public void info()
    {
          System.out.println(“我是本地方法”);
    }
    //下面提供程序入口，将远程类实例绑定为本机的服务。
   public static void main(String[] args)throws Exception
    {
          //创建远程服务类实例
       Server imp = new ServerImpl();
          //注册远程服务的端口
          LocateRegistry.createRegistry(1099);
          //将远程服务实例绑定为远程服务。
          Naming.rebind("rmi://:1099/fdf", imp);
   }
}
```

远程服务类必须有构造器，即使找个构造器什么都不做。而且，构造器必须抛出 RemoteException 异常。将两个编辑好的源文件存盘，然后编译。对于使用 RMI，仅仅编译还不够，还必须使用 rmic 命令编译服务类，编译服务类是为了生成 stub 和 skeleton——查看存放 class 文件的地方，多了如下两个 class 文件：

- ServerImpl_Stub.class
- ServerImpl_Skel.class

这两个类就是服务类生成的 stub 和 skeleton。客户端程序面向接口编程，客户端部分需要 Server 接口的 class 文件，还需要 stub 文件。客户端的源代码如下：

```
public class RMIClient
{
    //主方法，程序入口
  public static void main(String[] args)throws Exception
  {
        //通过 JNDI 查找远程服务
      Server ser = (Server)Naming.lookup("rmi://:1099/fdf");
        //调用远程方法
      System.out.println(ser.helloWorld("yeeku"));
        //调用远程方法。
      System.out.println(ser.getPerson("yeeku",28));
  }
}
```

从客户端程序看，无法感受到 Server 的实现在远端。程序调用 Server 实例方法时，与平常调用方法只有非常细微的区别：调用远程方法必须抛出 RemoteException。

再看远程方法的返回值，helloWorld 的返回值是 String，而 getPerson 的返回值则是 Person 对象。远程方法的返回值必须有一个要求：实现 Serializable 接口。因为远程的方法的参数、返回值都必须在网络上传输，网络只能传输字节流，因此，要求参数、返回值都可以转换成字节流——即实现序列化。主程序的如下一行代码，用于注册远程服务端口：

```
LocateRegistry.createRegistry(1099);
```

1099 是 RMI 服务的默认端口。然后执行如下代码绑定远程服务

```
Naming.rebind("rmi://:1099/fdf", imp);
```

客户端使用 JNDI 查找，查找远程服务名。将远程服务对象类型转换成远程接口类型，客户端面向接口编程。RMI 的具体实现，依然是依赖于底层的 Socket 编程。RMI 依赖于 TCP/IP 传输协议，服务器端 skeleton 建立 ServerSocket 监听请求，而客户端建立 Socket 请求连接。RMI 实现了网络传输的多线程、IO 等底层细节。这些细节的实现就隐藏在 rmic 命令的执行中：使用 rmic 命令编译时生成的两个 class 文件：

- stub：该文件用于与客户端交流，建立 Socket 请求连接。
- skeleton：该文件用于与服务器端交流，建立 ServerSocket 监听请求。

RMI 的原理示意如图 14.2 所示。

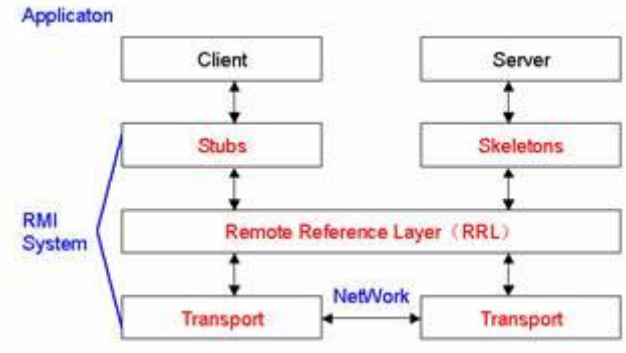

图 14.2 RMI 原理示意图

14.4.2 使用 RmiServiceExporter 提供服务

Spring 将 RMI 包装得更简单。使用 Spring 的 RMI 支持，应用可以透明地通过 RMI 提供服务。根本无须传统的 RMI 繁琐步骤：无须编写远程接口，无须编写远程服务主类。

在配置 Spring 的 RMI，类似于远程 EJB 的配置，只是没有安全上下文传递和远程事务传递的支持。当使用 RMI 调用器时，可以插入应用的安全框架或安全逻辑。看下面的“远程”接口，注意对比前面传统 RMI 的远程接口。

```
public interface Person
{
    //希望通过远程服务暴露的远程方法
    public void useAxe();
}
```

该接口无须继承 java.rmi.Remote 接口，完全是一个普通接口。同样，该接口的实现类也是普通的 JavaBean，该接口的实现类的源代码如下：

```
//实现 Person 接口
public class Chinese implements Person
{
    //成员变量
    private Axe axe;
    public Chinese()
    {
        System.out.println("Spring 实例化主调 bean：Chinese 实例...");
    }
    //依赖注入必须的 setter 方法
    public void setAxe(Axe axe)
    {
        System.out.println("Spring 执行依赖关系注入...");
      this.axe = axe;
    }
    //实现 Person 接口必须实现的方法
    public void useAxe()
    {
        System.out.println(axe.chop());
    }
}
```

程序中的 Chinese 类不再需要继承 java.rmi.server.UnicastRemoteObject 类，只是个 JavaBean。该类还依赖于 Axe 实例。这个实例不会作为远程服务暴露，仅仅作为 Chinese 的依赖使用。下面是 Axe 实例的接口和实现类：

```
Axe 接口
public interface Axe
{
    //该实例的业务方法
```

```
        public String chop();
}
//Axe 的实现类
public class SteelAxe implements Axe
{
        public SteelAxe()
        {
        }
        //实现 Axe 接口必须实现的方法
        public String chop()
        {
                return "钢斧砍柴真快";
        }
}
```

到目前为止，看到是两个简单的接口和对应的实现类。看不出任何提供远程服务的能力。Spring 可以将 Person 接口里的方法作为远程服务暴露出来。Spring 暴露远程方法的类是 RmiServiceExporter，该类可以将一个普通 bean 实例绑定成远程服务。将该 bean 实例绑定为远程服务的完整配置文件如下：

```
<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
        "http://www.springframework.org/dtd/spring-beans.dtd">
<!--  Spring 配置文件的根元素 -->
<beans>
        <!-- 配置 chinese 的依赖 bean-->
        <bean id="steelAxe" class="lee.SteelAxe"/>
        <!-- 配置 chinese bean-->
        <bean id="chinese" class="lee.Chinese" dependency-check="all">
                <!-- 配置依赖注入-->
                <property name="axe">
                        <ref local="steelAxe"/>
                </property>
    </bean>
    <!-- 使用 RmiServiceExporter 将目标 bean 暴露成远程服务-->
    <bean class="org.springframework.remoting.rmi.RmiServiceExporter">
          <!-- 指定暴露出来的远程服务名 -->
          <property name="serviceName">
                  <!-- 远程服务名为 chineseRemote-->
              <value>chineseRemote</value>
          </property>
                <!-- 配置需要暴露的目标 bean-->
          <property name="service">
```

```
            <ref bean="chinese"/>
        </property>
            <!-- 配置需要暴露的目标 bean 所实现的接口
                 该接口被远程接口对待-->
        <property name="serviceInterface">
            <value>lee.Person</value>
      </property>
        <!--1099 是 RMI 的默认端口，指定 RMI 远程服务的端口 -->
         <property name="registryPort"><value>1099</value></property>
   </bean>
</beans>
```

从上面的配置文件看出，chinese bean 没有丝毫独特之处。这正是 Spring RMI 支持的魅力：使用 RmiServiceExporter，可以将普通 bean 实例作为 RMI 远程服务导出。该远程服务可以使用 RmiProxyFactoryBean 访问，也可以使用传统 RMI 业务访问该服务。

通常推荐使用 RmiProxyFactoryBean 来访问该服务，RmiProxyFactoryBean 是个 FactoryBean，与前面的 FactoryBean 相似，客户端请求时不会获得它本身，而是它产生的实例。

该远程服务随 ApplicationContext 初始化时启动，Spring 会启动新线程来提供 RMI 服务，因此，提供 RMI 服务不会阻塞代码的执行。主程序如下：

```
public class BeanTest
{
   public static void main(String[] args)throws Exception
   {
            //创建 ApplicationContext 实例
       ApplicationContext ctx = new FileSystemXmlApplicationContext("bean.xml");
       System.out.println("============");
   }
}
```

主程序运行后，并不会立即结束——因为已经作服务器启动。第二行代码也会执行，并不会因为提供远程服务而阻塞。

14.4.3 在客户端连接服务

客户端访问 RMI 远程服务访问步骤都是一样的。先通过 JNDI 查找，然后调用远程方法。看采用传统的客户端访问 RMI 远程服务，客户端如下：

```
public class RMIClient
{
   public static void main(String[] args)throws Exception
     {
           //将查找的远程服务直接打印
       System.out.println(Naming.lookup("rmi://localhost:1099/chineseRemote"));
           //调用远程方法
       //p.useAxe();
```

```
    }
}
```

客户端可以通过 JNDI 查找到远程服务，但无法调用。将远程服务打印出来发现：

[java] RmiInvocationWrapper_Stub[UnicastRef [liveRef: [endpoint:[192.168.1.100:1713](remote),objID:[-1282787d:10b8fe03f0d:-8000, 0]]]]

因为，并不是通过传统的 RMI 开发步骤提供的远程服务，客户端并没有提供 stub 类。因此，客户端不能通过常规方法调用远程服务。

Spring 提供了 RMI 的客户端支持，Spring 通过 RmiProxyFactoryBean 连接 RMI 远程服务。RmiProxyFactoryBean 是个 FactoryBean。Spring 透明地创建调用器，使用 RmiProxyFactoryBean 使用该远程方法。客户端的配置文件如下：

```
<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
        "http://www.springframework.org/dtd/spring-beans.dtd">
<!--  Spring 配置文件的根元素 -->
<beans>
       <!-- 配置 Test  bean-->
    <bean id="test" class="lee.Test">
              <!--  test bean 依赖于远程服务-->
          <property name="person">
                    <!-- 定义依赖 bean，依赖 bean 通过远程服务获得-->
             <ref bean="person"/>
          </property>
      </bean>
      <!-- 通过 RmiProxyFactoryBean 访问远程服务-->
      <bean id="person" class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
              <!-- 指定远程服务的 url，后的服务名应与服务器端的服务名相同-->
           <property name="serviceUrl">
             <value>rmi://127.0.0.1:1099/chineseRemote</value>
          </property>
              <!-- 指定远程服务的接口名-->
           <property name="serviceInterface">
             <value>lee.Person</value>
          </property>
      </bean>
</beans>
```

客户端需要 Person 接口，这与传统的 RMI 方法并没有区别。客户端的主程序如下：

```
public class BeanTest
{
    public static void main(String[] args)throws Exception
    {
       ApplicationContext ctx = new FileSystemXmlApplicationContext("bean.xml");
```

```
        Test t = (Test)ctx.getBean("test");
        t.test();
    }
}
```

通过该主程序可以调用远程服务器端方法。而客户端本地没有 Person 实现类，只有一个 Person 接口，客户端程序并不关心 Person 的实现，即使不在本地都没有关系。

14.5 使用 Hessian 通过 Http 提供服务

Hessian 通常通过 Web 应用来提供服务，因此非常类似于 WebService。只是它不使用 SOAP 协议。

14.5.1 Hessian 介绍

相比 WebService，Hessian 更简单、快捷。采用的是二进制 RPC 协议，因为采用的是二进制协议，所以它很适合于发送二进制数据。下面演示一个简单的 Hessian 示例程序。

14.5.1.1 Hessian 的下载和安装

Hessian 的下载和安装请按如下步骤进行：

（1）登陆 http://www.caucho.com/hessian/下载 Hessian 的 Java 二进制包，笔者成书之时，Hessian 的最新版本是 Hessian 3.0.13。下载 hessian-3.0.13.jar 文件。

（2）将该文件复制到名为 hessian 的 Web 应用下，所有的 jar 文件都应该放在 WEB-INF/lib 下，该文件也不例外。

（3）为了编译 Hessian 客户端程序，建议将 hessian-3.0.13.jar 添加到环境变量里。

14.5.1.2 Hessian 服务器端

推荐采用面向接口编程，因此，Hessian 服务建议通过接口暴露。服务接口如下：

```
//服务接口
public interface Hello
{
        //方法声明
    public String hello(String name);
}
```

接口的实现类如下：

```
//服务实现类，实现 Hello 接口
public class HelloImpl implements Hello
{
    public String hello(String name)
    {
        return "hello " + name + "欢迎学习 Hessian";
    }
}
```

这个接口和实现类简单得难以置信。它们没有任何特别之处，这正是 Hessian 的魅力，

代码污染降低到最小。当然，只是示例程序，所以服务也相当简单。Hessian 要求远程服务通过 Servlet 暴露出来，必须在 web.xml 文件中配置该 Servlet。web.xml 的详细配置如下：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!--  Web 配置文件的文件头，包含 dtd 等信息-->
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
     "http://java.sun.com/dtd/web-app_2_3.dtd">
<!--  Web 配置文件的根元素-->
<web-app>
      <servlet>
            <!-- 配置 Servlet 名，后面根据该名完成远程服务映射-->
          <servlet-name>hessianService</servlet-name>
             <!--  Hessian 远程服务需要 HessianServlet 暴露-->
             <servlet-class>com.caucho.hessian.server.HessianServlet</servlet-class>
             <!-- 随应用启动而启动>
          <load-on-startup>1</load-on-startup>
             <!-- 使用 init-param 配置服务的实现类-->
          <init-param>
             <param-name>service-class</param-name>
             <param-value>lee.HelloImpl</param-value>
          </init-param>
      </servlet>
     <!-- 映射 Servlet 的 url，该 Servlet 的 url 就是 Hessian 服务名-->
    <servlet-mapping>
          <servlet-name>hessianService</servlet-name>
             <!-- 远程服务名为 hessianService-->
          <url-pattern>/hessianService</url-pattern>
    </servlet-mapping>
</web-app>
```

将刚才的接口和实现放 Web 应用的 WEB-INF/class 路径下，编译它们。然后将此 web.xml 文件放在 WEB-INF 下，启动 Web 服务器。笔者使用的 Web 服务器是 Tomcat5.0.28，Tomcat 的端口是 8888。则远程服务的 url 为：http://localhost:8888/hessian/hessianService。

14.5.1.3 Hessian 客户机端

Hessian 的服务可以用 HessianProxyFactory 工具类调用，还可以在小型智能设备上调用。HessianProxyFactory 的 create 方法，用于获取 Hessian 服务的远程引用。Hessian 的客户端如下：

```
public class HessianClient
{
    public static void main(String []args) throws Exception
    {
            //Hessian 服务的 url
        String url = "http://localhost:8888/hessian/hessianService";
```

```
            //创建 HessianProxyFactory 实例
        HessianProxyFactory factory = new HessianProxyFactory();
            //获得 Hessian 服务的远程引用
        Hello d = (Hello) factory.create(Hello.class, url);
            //调用远程服务。
        System.out.println("下面调用 Hessian 服务: " + d.hello("yeeku"));
    }
}
```

客户端仅仅需要 Hello 接口，而无须真实的实现类。如果使用小型智能设备作为客户端的运行环境，客户端代码片段如下：

```
//创建 Hessian 输入流，用于输入请求
MicroHessianInput in = new MicroHessianInput();
//Hessian 服务的 url
String url = "http://localhost:8888/hessian/hessianService";
//创建 HttpSConnection 实例
HttpConnection c = (HttpConnection) Connector.open(url);
//设置参数提交方式
c.setRequestMethod(HttpConnection.POST);
//打开输出流，准备调用服务器方法
OutputStream os = c.openOutputStream();
//以输出流创建 MicroHessianOutput 对象，该对象用于调用 hessian 的方法
MicroHessianOutput out = new MicroHessianOutput(os);
//调用远程方法:hello 是方法名,yeeku 是参数
out.call("hello", “yeeku”);
os.flush();
//打开输入流，准备接收返回值
is = c.openInputStream();
以输入流为参数，创建 MicroHessianInput 对象
MicroHessianInput in = new MicroHessianInput(is);
获得返回值
Object value = in.readReply(“yeeku”);
```

14.5.2 为 Hessian 装配 DispatcherServlet

通过上面的介绍，可以看出：Hessian 通过 Servlet 提供远程服务。需要将匹配某个模式的请求映射到 Hessian 服务。Spring 的 DispatcherServlet 可以完成该功能，DispatcherServlet 可将匹配模式的请求转发到 Hessian 服务。

在 web.xml 文件中配置 DispatcherServlet 拦截所有的远程服务请求，当然需要将 DispatcherServlet 配置成 load-on-startup Servlet。即在 web.xml 文件中增加如下两段

```
<!-- 将 DispatcherServlet 配置成 load-on-startup servlet-->
<servlet>
    <!-- Servlet name-->
    <servlet-name>remoting</servlet-name>
```

```
    <!--  Serlvet 实现类-->
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<!-- 定义 Servlet url 映射-->
<servlet-mapping>
    <servlet-name>remoting</servlet-name>
    <!-- 所有匹配/remoting/*的请求被 DispatcherServlet 截获-->
    <url-pattern>/remoting/*</url-pattern>
</servlet-mapping>
```

根据 DispatcherServlet 的源代码可以知道，DispatcherServlet 拦截了/remote/*请求，则需要加载 remoting-servlet.xml 的配置文件。同时，为了在应用启动时创建 ApplicationContext，则应使用 ContextLoaderListener 来加载 Spring。修改后的 web.xml 文件详细代码如下：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!--  Web 配置文件的文件头，包含 dtd 信息 -->
<!DOCTYPE web-app
   PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
   "http://java.sun.com/dtd/web-app_2_3.dtd">
<!--  Web 配置文件的根元素-->
<web-app>
      <!-- 强制指定 Spring 配置文件的路径-->
      <context-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/remoting-servlet.xml</param-value>
      </context-param>
      <!-- 使用 ContextLoaderListener 保证启动时加载 Spring -->
   <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
   </listener>
     <!-- 将 DispatcherServlet 配置成 load-on-startup Servlet-->
   <servlet>
        <servlet-name>remoting</servlet-name>
            <!--  Serlvet 实现类-->
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
   </servlet>
     <!-- 定义 Servlet url 映射-->
   <servlet-mapping>
        <servlet-name>remoting</servlet-name>
        <url-pattern>/remoting/*</url-pattern>
   </servlet-mapping>
</web-app>
```

至此，依然看不出 Hessian 远程服务的影子，只是定义了“请求转发器”，该转发器将匹配/remoting/*的请求截获，转发给 context 的 bean 处理。因此，下一步在 context 定义 bean。

14.5.3 使用 HessianServiceExporter 提供 bean 服务

根据上面的描述，必须提供 remoting-servlet.xml 文件。通常，在 remoting-servlet.xml 文件里定义 Hessian 服务即可。Spring 使用 HessianServiceExporter，将一个常规 bean 导出成 Hessian 服务。类似于 RmiServiceExporter，HessianServiceExporter 可将一个普通 bean 导出成远程服务。remoting-servlet.xml 的详细配置如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Spring 配置文件的文件头，包含 dtd 信息-->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
   "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
      <!-- 定义普通 bean 实例-->
   <bean id="helloService" class="lee.HelloImpl"/>
      <!-- 使用 HessianServiceExporter 将普通 bean 导出成 Hessian 服务-->
   <bean name="/helloService"
class="org.springframework.remoting.caucho.HessianServiceExporter">
            <!-- 需要导出的目标 bean-->
        <property name="service" ref="helloService"/>
            <!-- Hessian 服务的接口-->
        <property name="serviceInterface" value="lee.Hello"/>
   </bean>
</beans>
```

在该配置下，Hessian 服务的 url 是 http://localhost:8888/hessian-spring/remoting/helloService，其中 hessian-spring 是该应用的 url。Spring 使用 DispatcherServlet 拦截到匹配/remoting/*的请求，然后将该请求转发到对应的 bean，该 bean 在 remoting-servlet.xml 文件中以 HessianServiceExporter 定义。

14.5.4 在客户端连接服务

Spring 提供的 Hessian 服务是标准服务。因此，完全可以使用前面的客户端程序来访问，只需要修改 url 为 http://localhost:8888/hessian-spring/remoting/helloService 即可。

为了充分利用 IoC 特性，Spring 还提供 HessianProxyFactoryBean 连接 Hessian 服务，HessianProxyFactoryBean 用于连接 Hessian 服务，类似所有的 FactoryBean，对它的请求不会返回实例本身， 而是它生成的实例。配置 HessianProxyFactoryBean bean 时候，只需要指定 Hessian 服务的 url，以及 Hessian 服务实现的接口。详细的配置如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Spring 配置文件的文件头，包含 dtd 信息-->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
   "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素-->
<beans>
```

```
        <!-- 依赖于远程服务的测试 bean-->
      <bean id="test" class="lee.Test">
            <!-- 配置依赖注入-->
          <property name="hello">
             <ref local="helloService"/>
          </property>
    </bean>
       <!-- 使用 HessianProxyFactoryBean 连接远程 Hessian 服务-->
    <bean id="helloService"
class="org.springframework.remoting.caucho.HessianProxyFactoryBean">
            <!-- 确定 Hessian 服务的 url-->
       <property name="serviceUrl"
value="http://localhost:8888/hessian-spring/remoting/helloService"/>
            <!-- 确定 Hessian 服务实现的接口-->
       <property name="serviceInterface" value="lee.Hello"/>
    </bean>
</beans>
```

Test 是测试 bean，该 bean 依赖于 helloService bean，其源代码如下：

```
public class Test
{
       //依赖实例
    private Hello hello;
       //依赖注入的 setter 方法
    public void setHello(Hello h)
    {
          this.hello = h;
    }
      public void test(String name)
    {
          System.out.println(hello.hello(name));
    }
}
```

与 RmiServiceExporter 类似，Test 的代码中没有丝毫远程访问的痕迹。Test 完全不用理会 Hello 的底层实现。Test 可以在远程服务的不同协议之间切换，甚至可以在远程服务和本地服务之间切换。

14.6 通过 Web Service 提供服务

WebsService 让一个程序可以透明地调用互联网的程序，不用管具体的实现细节。只要 WebService 公开了服务接口，远程客户端就可以调用服务。WebService 是基于 http 协议的组件服务，WebService 是分散式应用程序的发展趋势。

14.6.3 从客户端访问 Spring 发布的 WebSer...

14.6.1 WebService 的开源实现

WebService 更多是一种标准，而不是一种具体的技术。不同的平台，不同的语言大都提供 WebService 的开发实现。在 Java 领域，WebService 的一个成熟实现是 Axis。

- Axis 是 Apache 的一个开源 WebService 引擎，它是目前最为成熟的开源 WebService 引擎之一。
- Apache Axis 是符合 W3C 的 SOAP 协议的实现。
- 该项目是属于 Apache SOAP 协议的子项目。

14.6.1.1 Axis 的下载

登陆 http://www.apache.org/dyn/closer.cgi/ws/axis/1_4 站点，Axis 的最新版本是 1.4，笔者的示例程序都是基于该版本完成。下载到 axis-bin-1_4.zip 文件，解压缩该文件，发现如下的文件结构：

- docs：存放 Axis 的说明文档。
- lib：存放 Axis 的二进制发布包。
- samples：存放利用 Axis 发布 WebService 的示例代码。
- webapps：存放安装 Axis 的基础应用。
- xmls：存放相关配置文档。
- 还有 LICENSE 等相关说明文档。

14.6.1.2 Axis 的安装

安装 Axis，首先必须保证系统已经正确安装了 Web 服务器，笔者使用 Tomcat5.0.28。也可使用其他应用服务器，例如 WebLogic。webapps 下的 axis 路径全部复制到 Tomcat 的 webapps 路径下。然后打开浏览器，输入：http://localhost:8888/axis/。

如果出现如图 14.3 所示界面表示 Axis 的基本类库已经安装成功，地址中的 8888 是笔者的 Tomcat 的端口。单击如图 14.3 所示页面的左上方“Validation”链接，出现如图 14.4 所示界面：

图 14.3 Axis 安装成功界面

图 14.4 Axis 校验界面

该页面显示 Axis 的必需组件还缺少一个没有安装，两个可选组件也没有安装，单击上面的提示链接，分别登陆：

- http://java.sun.com/products/javabeans/glasgow/jaf.html
- http://java.sun.com/products/javamail/
- http://xml.apache.org/security/

依次下载如下三个压缩文件：

- jaf-1_1-fr.zip
- javamail-1_4.zip
- xml-security-bin-1_2_1.zip

将这三个压缩文件解压缩。将 jaf-1_1-fr.zip 压缩包中的 activation.jar 文件，复制到 axis 应用的 WEB-INF\lib 下；将 javamail-1_4.zip 压缩包中的 mail.jar 文件，复制到 axis 应用的 WEB-INF/lib 下；再将 xml-security-bin-1_2_1.zip 的 lib 下 xmlsec-1.2.1.jar 和 xalan.jar 文件，复制到 WEB-INF/lib 下。再次校验 Axis，看到页面提示全部安装成功。

14.6.1.3 开发自己的 WebService

完成了前面的部署后，就可以开始开发自己的 WebService 了。首先编写需要提供服务的类，该类不需要实现任何接口，也不需要继承任何父类，只需要是 POJO 即可。Axis 支持将普通方法暴露成 WebService。看如下的服务类：

```
//服务类，应该声明为 public
public class Hello
{
    //服务方法
   public String hello()
   {
     return "hello ," + name + ",Welcome to Axis";
   }
}
```

该类简单得难以置信，但正是个简单的类，它可以提供 WebService。将该类的文件名存为 Hello.jws。

注意：不是 java 后缀，而是 jws 后缀。

将该文件放入 axis 应用的根路径下，也就是与 WEB-INF 同一级路径，然后重启动 Tomcat，或者在 Tomcat 控制台重新加载 axis 应用。然后，打开浏览器输入如下地址：

http://localhost:8888/axis/Hello.jws

看到如图 14.5 所示界面：

图 14.5 Hello WebService

这表示该 WebService 发布成功，单击“Click to see the WSDL”链接，可查看该 WebService

的 WSDL 描述。成功发布的 WebSerivice 通过网络访问，因为 SOAP 协议基于 HTTP 协议，因此 WebService 可以在互联网上访问。访问该 WebService 的客户端代码如下：

```
public class WebServiceClient
{
      //程序的入口
   public static void main(String args[])
   {
        System.out.println("开始调用 WebService");
        try
        {
                //WebService 所在的 url
           String endpoint = "http://localhost:8888/axis/Hello.jws";
                //创建 Serivce 对象，Service 对用用于创建 Call 对象
            Service service = new Service();
                //创建 Call 对象，Call 对象用于调用服务
           Call call = (Call)service.createCall();
                为 Call 对象设置 WebService 的 url
           call.setTargetEndpointAddress(new java.net.URL(endpoint));
           //为 Call 对象设置调用的方法名
           call.setOperationName("hello");
                //调用 WebService 的方法，并获得返回值
           String s = (String)call.invoke(new Object[] {"中国人"});
                //输出返回值
           System.out.println(s);
        }
        catch (Exception e)
        {
           System.out.println(e.toString());
        }
        System.out.println("调用 WebService 正常结束");
   }
}
```

借助于 Axis 可以将 POJO 发布成 WebService，远程客户端不需要任何接口或类，因为远程客户端直接调用方法，直接使用 Call 对象调用方法。

14.6.2 Spring 对 WebService 的支持

Spring 使用 ServletEndpointSupport 来暴露 WebService。假设有如下接口：

```
//业务接口
public interface Hello
{
   public String hello(String name);
}
```

以及如下的实现类：

```
//实现类，实现上面的业务接口
public class   HelloImpl implements Hello
{
      //实现业务接口必须实现的方法
    public String hello(String name)
    {
        return name + "你好，欢迎学习 Spring 和 Axis";
    }
}
```

上面是非常简单的面向接口编程的应用结构，一个接口以及对应的实现类。如果需要将该实现类暴露成 WebService，则编写一个 WebService 类，让该类实现 Hello 接口，并继承 ServletEndpointSupport 类。该类的源代码如下：

```
//实现 Hello 接口，继承 ServletEndpointSupport 工具类
public class HelloEndpoint extends ServletEndpointSupport implements Hello
{
      //将真实的业务 bean 包装成 WebService
    private Hello h;
      //该方法由 Spring 调用，将目标业务 bean 注入。
     protected void onInit()
    {
        this.h = (Hello) getWebApplicationContext().getBean("hello");
    }
      //将业务 bean 的业务方法暴露成 WebService
    public String hello(String name)
     {
        return h.hello(name);
    }
}
```

然后提供 Spring 的配置文件，配置文件中部署业务 bean，配置文件如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!--  Spring 配置文件的文件头，包含 DTD 等信息-->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
   "http://www.springframework.org/dtd/spring-beans.dtd">
<!--  Spring 配置文件的根元素-->
<beans>
      <!-- 配置 Spring 的业务 bean-->
   <bean id="hello" class="lee.HelloImpl"/>
</beans>
```

然后修改 web.xml 文件，让 AxisServlet 拦截某些请求，这是 Axis 必需的。web.xml 的详细配置如下：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<!--  Web 应用配置文件的文件头,包含 dtd 等信息-->
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD WebApplication 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<!--  Web 应用配置文件的根元素-->
<web-app>
      <!-- 用于初始化 ApplicationContext 的 listener-->
      <listener>
            <listener-class>org.springframework.web.context.ContextLoaderListener</listener-cl
ass>
      </listener>
     <!-- 定义 AxisServlet-->
   <servlet>
        <servlet-name>AxisServlet</servlet-name>
        <servlet-class>org.apache.axis.transport.http.AxisServlet</servlet-class>
   </servlet>
      <!--  映射 AxisServlet,使用通配符-->
      <servlet-mapping>
            <servlet-name>AxisServlet</servlet-name>
            <url-pattern>/axis/*</url-pattern>
      </servlet-mapping>
</web-app>
```

配置文件中，看到所有匹配/axis/*模式的请求都由 AxisServlet 处理。即：Spring 发布的 WebService 都在 axis 下。然后编写 wsdl 文件，或者使用工具生成。wsdl 文件如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!--  wsdl 的根元素，包含 schema 等信息-->
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
    <!--  wsdl 的全局配置-->
     <globalConfiguration>
          <parameter name="adminPassword" value="admin"/>
          <parameter name="sendXsiTypes" value="true"/>
          <parameter name="sendMultiRefs" value="true"/>
          <parameter name="sendXMLDeclaration" value="true"/>
          <parameter name="axis.sendMinimizedElements" value="true"/>
          <requestFlow>
               <handler type="java:org.apache.axis.handlers.JWSHandler">
                    <parameter name="scope" value="session"/>
               </handler>
               <handler type="java:org.apache.axis.handlers.JWSHandler">
                    <parameter name="scope" value="request"/>
                    <parameter name="extension" value=".jwr"/>
               </handler>
```

```
            </requestFlow>
        </globalConfiguration>
        <handler name="Authenticate"
type="java:org.apache.axis.handlers.SimpleAuthenticationHandler"/>
        <handler name="LocalResponder"
type="java:org.apache.axis.transport.local.LocalResponder"/>
        <handler name="URLMapper" type="java:org.apache.axis.handlers.http.URLMapper"/>
        <!-- 定义 WebService 的管理台-->
        <service name="AdminService" provider="java:MSG">
            <parameter name="allowedMethods" value="AdminService"/>
            <parameter name="enableRemoteAdmin" value="false"/>
            <parameter name="className" value="org.apache.axis.utils.Admin"/>
            <namespace>http://xml.apache.org/axis/wsdd/</namespace>
        </service>
        <!-- 定义自己的 WebService-->
        <service name="HelloService" provider="java:RPC">
            <parameter name="allowedMethods" value="*"/>
            <!-- 定义 WebService 的实现类-->
            <parameter name="className" value="lee.HelloEndpoint"/>
        </service>
        <!-- 定义 WebSerivice 的系统服务。-->
        <service name="Version" provider="java:RPC">
            <parameter name="allowedMethods" value="getVersion"/>
            <parameter name="className" value="org.apache.axis.Version"/>
        </service>
        <transport name="http">
            <requestFlow>
                <handler type="URLMapper"/>
                <handler type="java:org.apache.axis.handlers.http.HTTPAuthHandler"/>
            </requestFlow>
        </transport>
        <transport name="local">
            <responseFlow>
                <handler type="LocalResponder"/>
            </responseFlow>
        </transport>
</deployment>
```

经过这些步骤，可将部署在 Spring 的普通 bean 发布成 WebService。

14.6.3 从客户端访问 Spring 发布的 WebService

Spring 发布的 WebService 是标准的 WebSerivce。因此可以使用标准客户端访问，也可以利用 IoC 容器来管理 WebService，利用 IoC 容器时，必须借助于

JaxRpcPortProxyFactoryBean。

14.6.3.1 利用标准客户端访问 WebService

标准客户端访问 WebService 非常简单，只需要获得 WebService 的 url 以及方法名即可。根据上面的发布，知道 Webservice 的 url 如下：http://localhost:8888/axis-spring/axis/HelloService。

对上面的客户端代码稍作修改，修改后的客户端代码如下：

```
public class WebServiceClient
{
      //程序的入口
   public static void main(String args[])
   {
        System.out.println("开始调用 WebService");
        try
        {
                //WebService 所在的 url
           String endpoint = "http://localhost:8888/axis-spring/axis/HelloService";
                //创建 Serivce 对象，Service 对用用于创建 Call 对象
            Service service = new Service();
                //创建 Call 对象，Call 对象用于调用服务
           Call call = (Call)service.createCall();
                为 Call 对象设置 WebService 的 url
           call.setTargetEndpointAddress(new java.net.URL(endpoint));
           //为 Call 对象设置调用的方法名
           call.setOperationName("hello");
                //调用 WebService 的方法，并获得返回值
           String s = (String)call.invoke(new Object[] {"中国人"});
                //输出返回值
           System.out.println(s);
        }
        catch (Exception e)
        {
           System.out.println(e.toString());
        }
        System.out.println("调用 WebService 正常结束");
    }
}
```

14.6.3.2 利用 IoC 容器管理 WebService

利用 IoC 容器管理 WebService，必须借助于 JaxRpcPortProxyFactoryBean 类，该类是个工厂 bean，与所有的工厂 bean 一样，对该 bean 的请求将返回它的产品。

配置 JaxRpcPortProxyFactoryBean，只需提供 WebService 的 url、命名空间等必需信息，将可以返回 WebService 服务。JaxRpcPortProxyFactoryBean 的配置如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!--  Spring 配置文件的文件头，包含 dtd 等信息-->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
```

```
    "http://www.springframework.org/dtd/spring-beans.dtd">
<!--  Spring 配置文件的根元素-->
<beans>
      <!-- 配置测试 bean-->
   <bean id="test" class="lee.Test">
        <property name="hello">
          <ref local="helloService"/>
        </property>
     </bean>
    <!-- 配置 WebService bean-->
   <bean id="helloService"
class="org.springframework.remoting.jaxrpc.JaxRpcPortProxyFactoryBean">
            <!-- 配置 WebService 实现的接口-->
        <property name="serviceInterface">
          <value>lee.Hello</value>
        </property>
            <!-- 配置 WebService 的 wsdl 的 url-->
        <property name="wsdlDocumentUrl">
          <value>http://localhost:8888/axis-spring/axis/HelloService?wsdl</value>
        </property>
            <!-- 配置 WebService 的命名空间 uri-->
        <property name="namespaceUri">
          <value>http://localhost:8888/axis-spring/axis/HelloService</value>
        </property>
            <!-- 配置 WebService 的服务名-->
        <property name="serviceName">
          <value>HelloService</value>
        </property>
          <!-- 配置 WebService 的 portName-->
        <property name="portName">
          <value>HelloService"</value>
        </property>
   </bean>
</beans>
```

14.7 使用 httpInvoker 提供远程服务

使用 HttpInvoker，不需要额外的类库。和 Hessian 的轻量级传输协议不同的是，Spring HttpInvoker 使用 Java 序列化来序列化参数和返回值，然后基于 HTTP 协议传输经序列化后的对象。当参数或返回值是复杂类型，并且不能通过 Hessian 的序列化机制序列化时，HttpInvoker 就很有优势。

14.7.1 输出业务对象

通过 HttpInvoker 提供远程服务与 Hessian 非常相似：Spring 为 Hessian 提供 HessianServiceExporter，Spring 为 HttpInvoker 提供 HttpInvokerServiceExporter。该类也可将普通 baen 实例输出成远程服务。配置方法与 HessianServiceExporter 也非常相似。

下面是使用 HttpInvokerServiceExporter 提供远程服务的示例。在该示例里，远程服务 bean 的方法会返回一个 Person 对象，注意：Person 对象必须实现 Serializable 或 Externalizable 接口，因为采用了 Java 序列化机制来传输参数和返回值。下面是远程服务提供类：

```
public class HelloImpl implements Hello
{
    //远程服务方法，该方法返回一个可序列化的对象
   public Person hello(String name)
   {
       Person p = new Person();
           p.setName(name);
           p.setAge(13);
           return p;
   }
}
```

下面是 Person 的实现类，该类实现了 Serializable 接口：

```
public class Person implements Serializable
{
    //Person 类的两个属性
    private String name;
    private int age;
    //name 属性的 getter 方法
    public String getName()
    {
        return name;
    }
    //name 属性的 setter 方法
    public void setName(String name)
    {
        this.name = name;
    }
    //age 属性的 getter 方法
    public int getAge()
    {
        return age;
    }
    //age 属性的 setter 方法
    public void setAge(int age)
```

```
    {
        this.age = age;
    }
}
```

然后修改 web.xml 配置文件，该配置文件里，需要让 DispatcherServlet 拦截匹配模式的请求，该请求由 DispatcherServlet 转发给 context 中对应的 bean 处理。修改后的 web.xml 配置文件如下：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- Web 应用配置文件的文件头，包含 dtd 等信息-->
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
   "http://java.sun.com/dtd/web-app_2_3.dtd">
<!-- Web 应用配置文件的根元素-->
<web-app>
    <!-- 配置 DispatcherServlet-->
  <servlet>
      <servlet-name>remoting</servlet-name>
      <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <!-- 配置该 Servlet 随应用启动时候启动-->
      <load-on-startup>1</load-on-startup>
  </servlet>
    <!-- 配置 DispatcherServlet 映射的 url-->
  <servlet-mapping>
      <servlet-name>remoting</servlet-name>
      <url-pattern>/remoting/*</url-pattern>
  </servlet-mapping>
</web-app>
```

上面配置文件中，没有使用 ContextLoaderListener 来初始化 ApplicationContext，也没有指定 ApplicationContext 配置文件的位置。这些都是通过 DispatcherServlet 默认加载的，DispatcherServlet 在创建时，会加载 ApplicationContext，默认查找的配置文件为：[servlet-name]-servlet.xml 文件。对该配置默认加载 remoting-servlet.xml 文件。因此，在 WEB-INF 下提供 remoting-servlet.xml 文件即可。remoting-servlet.xml 文件如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Spring 配置文件的文件头，包含 dtd 等信息-->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
   "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素-->
<beans>
<!-- 配置业务服务 bean-->
  <bean id="helloService" class="lee.HelloImpl"/>
  <!-- 将业务服务 bean 暴露成远程服务-->
  <bean name="/helloService"
```

```
        class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
        <!-- 指定需要暴露的业务服务 bean-->
    <property name="service" ref="helloService"/>
        <!-- 指定暴露的远程服务实现的接口-->
    <property name="serviceInterface" value="lee.Hello"/>
  </bean>
</beans>
```

DispatcherServlet 转发请求的默认策略是：将请求转发到 context 中与请求同名的 bean。上面配置中远程服务的 url 为：http://localhost:8888/httpInvoker/remoting/helloService，其中 httpInvoker 为该应用的虚拟路径。

14.7.2 客户端连接服务

Spring 提供了 HttpInvokerProxyFactoryBean 工厂 bean 连接服务。类似于 Hessian 的 HessianProxyFactoryBean，配置 HttpInvokerProxyFactoryBean 时，只需指定服务的 url 以及服务实现的接口。通过使用代理，Spring 可将调用转换成 POST 请求发送到指定服务。详细的配置如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Spring 配置文件的文件头，包含 dtd 等信息-->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素-->
<beans>
    <!-- 配置测试 bean，该测试 bean 依赖于远程服务 bean-->
  <bean id="test" class="lee.Test">
        <!-- 配置依赖注入-->
    <property name="hello">
      <ref local="helloService"/>
    </property>
  </bean>
    <!-- 配置连接远程服务的 bean-->
  <bean id="helloService"
class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
        <!-- 远程服务的 url-->
        <property name="serviceUrl"
value="http://localhost:8888/httpInvoker/remoting/helloService"/>
        <!-- 远程服务所实现的接口-->
    <property name="serviceInterface" value="lee.Hello"/>
  </bean>
</beans>
```

通过上面的配置，客户端可以访问 httpInvoker 提供的服务，默认情况下，HttpInvokerPropxy 使用 J2SE 的 HTTP Client 功能。可通过设置 httpInvokerRequestExecutor 属性使用 Commons HttpClient。通过对配置文件简单修改，将

```
    <bean id="helloService"
class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
            <!-- 远程服务的 url-->
            <property name="serviceUrl"
value="http://localhost:8888/httpInvoker/remoting/helloService"/>
            <!-- 远程服务所实现的接口-->
        <property name="serviceInterface" value="lee.Hello"/>
    </bean>
```

修改成：

```
    <bean id="helloService"
class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
            <!-- 远程服务的 url-->
            <property name="serviceUrl"
value="http://localhost:8888/httpInvoker/remoting/helloService"/>
            <!-- 远程服务所实现的接口-->
        <property name="serviceInterface" value="lee.Hello"/>
            <!-- 指定使用 Commons HttpClient 功能-->
            <property name="httpInvokerRequestExecutor">
                <bean
                    class="org.springframework.remoting.httpinvoker.CommonsHttpInvokerR
equestExecutor"/>
            </property>
    </bean>
```

14.8 小结

本章简要介绍了远程访问的发展以及意义。详细介绍如下几种常用的远程访问技术：RMI，Hessian，WebService，httpInvoker，包括这几种远程访问技术的使用。重点介绍了 Spring 对上面几种远程访问技术的支持，结合示例，重点介绍了如何通过 Spring 提供远程服务，以及客户端如何连接这些远程服务。