

(2) 运行代码清单 16-7 中的 JDBCJobStoreRunner 恢复 trigger1_1 的触发器, 运行一段时间后退出, 这时 quartz_simple_triggers 表中的数据如下:

	TRIGGER_NAME	TRIGGER_GROUP	REPEAT_COUNT	REPEAT_INTERVAL	TIMES_TRIGGERED
▶ 1	trigger1_1	...	tgroup1	...	96
					2000
					22

首先 Quartz 会将原 REPEAT_COUNT 的值减去 TIMES_TRIGGERED 的值得到新的 REPEAT_COUNT 值, 并记录已经运行的次数 (重新从 0 开始计算)。

(3) 重新启动 JDBCJobStoreRunner 运行后, 数据又将发生相应的变化, 如下:

	TRIGGER_NAME	TRIGGER_GROUP	REPEAT_COUNT	REPEAT_INTERVAL	TIMES_TRIGGERED
▶ 1	trigger1_1	...	tgroup1	...	74
					2000
					7

TIMES_TRIGGERED 重新从 0 开始计数, 而 REPEAT_COUNT 在原有值的基础上进行调整。

(4) 继续运行直至完成所有剩余的次数, 再次查询 quartz_simple_triggers 表, 如下:

	TRIGGER_NAME	TRIGGER_GROUP	REPEAT_COUNT	REPEAT_INTERVAL	TIMES_TRIGGERED

这时, 该表中的记录已经变空。

值得注意的是, 如果用户使用 JDBC 保存任务调度数据, 那么运行代码清单 16-2 中的 SimpleTriggerRunner 然后退出, 当再次希望运行 SimpleTriggerRunner 时, 系统将抛出 JobDetail 重名的异常, 如下:

```
Unable to store Job with name: 'job1_1' and group: 'jgroup1', because one already exists with this identification.
```

因为每次调用 Scheduler#scheduleJob() 方法时, Quartz 都会将 JobDetail 和 Trigger 的信息保存到数据库中。如果数据库中已经有同名的 JobDetail 或 Trigger, 就会产生异常。

16.3 在 Spring 中使用 Quartz

Spring 为创建 Quartz 的 Scheduler、Trigger 和 JobDetail 提供了便利的 FactoryBean 类, 以便能够在 Spring 容器中享受注入的好处。此外, Spring 还提供了一些便利工具类, 用于直接将 Spring 中的 Bean 包装成合法的任务。Spring 进一步降低了使用 Quartz 的难度, 能以更具 Spring 风格的方式使用 Quartz。概括来说, Spring 提供了两方面的支持。

(1) 为 Quartz 的重要组件类提供更具 Bean 风格的扩展类。

(2) 提供创建 Scheduler 的 BeanFactory 类, 方便在 Spring 环境下创建对应的组件对象, 并结合 Spring 容器生命周期执行启动和停止的动作。

16.3.1 创建 JobDetail

用户可以直接使用 Quartz 的 JobDetail 在 Spring 中配置一个 JobDetail Bean, 但是 JobDetail 使用带参的构造函数, 对于习惯通过属性配置的 Spring 用户来说存在使用上的

不便。为此, Spring 通过扩展 `JobDetail` 提供了一个更具 Bean 风格的 `JobDetailFactoryBean`。此外, Spring 还提供了一个 `MethodInvokingJobDetailFactoryBean`, 通过这个 `FactoryBean` 可以将 Spring 容器中 Bean 的方法包装成 Quartz 任务, 这样开发者就不必为 Job 创建对应的类。

1. JobDetailFactoryBean

`JobDetailFactoryBean` 扩展于 Quartz 的 `JobDetail`。使用该 Bean 声明 `JobDetail` 时, Bean 的名字即任务的名字, 如果没有指定所属组, 就使用默认组。除了 `JobDetail` 中的属性外, 还定义了以下属性。

- `jobClass`: 类型为 `Class`, 实现 `Job` 接口的任务类。
- `beanName`: 默认为 Bean 的 id 名, 通过该属性显式指定 Bean 名称, 它对应任务的名称。
- `jobDataAsMap`: 类型为 `Map`, 为任务所对应的 `JobDataMap` 提供值。之所以需要提供这个属性, 是因为用户无法在 Spring 配置文件中为 `JobDataMap` 类型的属性提供信息, 所以 Spring 通过 `jobDataAsMap` 设置 `JobDataMap` 的值。
- `applicationContextJobDataKey`: 用户可以将 Spring `ApplicationContext` 的引用保存到 `JobDataMap` 中, 以便在 Job 的代码中访问 `ApplicationContext`。为了达到这个目的, 用户需要指定一个键, 用于在 `jobDataAsMap` 中保存 `ApplicationContext`。如果不设置此键, `JobDetailBean` 就不会将 `ApplicationContext` 放入 `JobDataMap` 中。
- `jobListenerNames`: 类型为 `String[]`, 指定注册在 `Scheduler` 中的 `JobListeners` 名称, 以便让这些监听器对本任务的事件进行监听。

下面的配置片段使用 `JobDetailBean` 在 Spring 中配置一个 `JobDetail`。

```
<bean name="jobDetail" class="org.springframework.scheduling.quartz.JobDetailFactoryBean"
  p:jobClass="com.smart.quartz.MyJob"
  p:applicationContextJobDataKey="applicationContext">
  <property name="jobDataAsMap">
    <map>
      <entry key="size" value="10" />
    </map>
  </property>
</bean>
```

`JobDetailFactoryBean` 封装了 `MyJob` 任务类, 并为 Job 对应的 `JobDataMap` 设置了一个键为 `size` 的数据。此外, 通过指定 `applicationContextJobDataKey`, 让 Job 的 `JobDataMap` 持有 Spring `ApplicationContext` 的引用。

这样, `MyJob` 在运行时就可以通过 `JobDataMap` 访问到 `size` 和 `ApplicationContext`。来看一下 `MyJob` 的代码, 如代码清单 16-8 所示。

代码清单 16-8 MyJob

```
package com.smart.quartz;
...
import org.quartz.Job;
import org.quartz.JobExecutionContext;
```



```

import org.quartz.JobExecutionException;
import org.springframework.context.ApplicationContext;
public class MyJob implements Job {
    public void execute(JobExecutionContext jctx) throws JobExecutionException {
        Map dataMap = jctx.getJobDetail().getJobDataMap(); ①
        String size = (String) dataMap.get("size"); ②
        ApplicationContext ctx = (ApplicationContext) dataMap.get("applicationContext"); ③
        System.out.println("size:"+size);
        dataMap.put("size", size+"0"); ④
        //do sth...
    }
}

```

① 获取 JobDetail 关联的 JobDataMap
② 对 JobDataMap 所做的更改是否会被持久化取决于任务的类型

在②处获取 size 值，在③处可以根据键 applicationContext 获取 ApplicationContext，有了 ApplicationContext 的引用，Job 就可以毫无障碍地访问 Spring 容器中的任何 Bean。MyJob 可以在 execute() 方法中对 JobDataMap 进行更改，如④处所示。如果 MyJob 实现了 Job 接口，则这种更改对于下一次执行是不可见的；如果 MyJob 实现了 StatefulJob 接口，则这种更改对于下一次执行是可见的。

2. MethodInvokingJobDetailFactoryBean

通常情况下，任务都定义在一个业务类方法中。这时，为了满足 Quartz Job 接口的规定，还需要定义一个引用业务类方法的实现类。为了避免创建这个只包含一行调用代码的 Job 实现类，Spring 提供了 MethodInvokingJobDetailFactoryBean，借由该 FactoryBean，可以将一个 Bean 的某个方法封装成满足 Quartz 要求的 Job。来看一个具体的例子：

```

<bean id="jobDetail_1"
    class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean"
    p:targetObject-ref="myService" ①
    p:targetMethod="doJob" ②
    p:concurrent="false"/>③

```

① 引用一个 Bean
② 指定目标 Bean 的方法
③ 指定最终封装出的任务是否有状态

```

<bean id="myService" class="com.smart.service.MyService"/>

```

jobDetail_1 将 MyService#doJob() 封装成一个任务，同时通过 concurrent 属性指定任务的类型。默认情况下封装为无状态的任务。如果希望封装为有状态的任务，仅需将 concurrent 属性设置为 true 就可以了。Spring 通过名为 concurrent 的属性指定任务的类型，能够更直接地描述任务执行的方式（有状态的任务不能并发执行，无状态的任务可并发执行），对于不熟悉 Quartz 内部机制的用户来说，比起 stateful，concurrent 显然更简明达意些。

MyService 服务类拥有一个 doJob() 方法，它的代码如下所示：

```

package com.smart.service;
public class MyService {
    public void doJob() { ①
        System.out.println("in MyService.dojob().");
    }
}

```

① 被封装成任务的目标方法

doJob()方法既可以是 static 的,也可以是非 static 的,但不能拥有方法入参。通过 MethodInvokingJobDetailFactoryBean 产生的 JobDetail 不能被序列化,所以不能被持久化到数据库中。如果希望使用持久化任务,则用户只能创建正规的 Quartz 的 Job 实现类。

16.3.2 创建 Trigger

Quartz 中另一个重要的组件就是 Trigger, Spring 按照相似的思路分别为 SimpleTrigger 和 CronTrigger 提供了更具 Bean 风格的 SimpleTriggerFactoryBean 和 CronTriggerFactoryBean 扩展类,通过这两个扩展类可以更容易地在 Spring 中以 Bean 的方式配置 Trigger。

1. SimpleTriggerFactoryBean

在默认情况下,通过 SimpleTriggerFactoryBean 配置的 Trigger 名称即为 Bean 的名称,属于默认组。SimpleTriggerFactoryBean 在 SimpleTrigger 的基础上新增了以下属性。

- jobDetail: 对应的 JobDetail。
- beanName: 默认为 Bean 的 id 名,通过该属性显式指定 Bean 名称,它对应 Trigger 的名称。
- jobDataAsMap: 以 Map 类型为 Trigger 关联的 JobDataMap 提供值。
- startDelay: 延迟多少时间开始触发,单位为毫秒,默认值为 0。
- triggerListenerNames: 类型为 String[], 指定注册在 Scheduler 中的 TriggerListener 名称,以便让这些监听器对本触发器的事件进行监听。

下面的实例使用 SimpleTriggerFactoryBean 定义了一个 Trigger, 该 Trigger 和 jobDetail 相关联,延迟 1 秒后启动,时间间隔为 2 秒,重复执行 100 次。此外,还为 Trigger 设置了 JobDataMap 数据。

```
<bean id="simpleTrigger" class="org.springframework.scheduling.quartz. Simple
TriggerFactoryBean "
    p:jobDetail-ref="jobDetail"
    p:startDelay="1000"
    p:repeatInterval="2000"
    p:repeatCount="100">
    <property name="jobDataAsMap">①
        <map>
            <entry key="count" value="10" />
        </map>
    </property>
</bean>
```

需要特别注意的是,在①处配置的 Map 数据将填充到 Trigger 的 JobDataMap 中,执行任务时必须通过以下方式获取配置的值:

```
package com.smart.quartz;
...
public class MyJob implements StatefulJob {
    public void execute(JobExecutionContext jctx) throws JobExecutionException {
```

```

Map dataMap = jctx.getTrigger().getJobDataMap(); ①
String count = dataMap.get("count");
dataMap.put("count", "30"); ②
...
}

```

← 获取 Trigger 的 JobDataMap

← 对 JobDataMap 的更改不会被持久化, 不影响下次的执行

2. CronTriggerFactoryBean

CronTriggerFactoryBean 扩展于 CronTrigger, 触发器的名称即为 Bean 的名称, 保存在默认组中。在 CronTrigger 的基础上, 新增的属性和 SimpleTriggerFactoryBean 大致相同, 配置的方法也和 SimpleTriggerFactoryBean 相似。下面给出一个简单的例子:

```

<bean id="checkImagesTrigger"
    class="org.springframework.scheduling.quartz.CronTriggerFactoryBean"
    p:jobDetail-ref="jobDetail"
    p:cronExpression="0/5 * * * * ?"/>

```

16.3.3 创建 Scheduler

Quartz 的 SchedulerFactory 是标准的工厂类, 不太适合在 Spring 环境下使用。此外, 为了保证 Scheduler 能够感知 Spring 容器的生命周期, 在 Spring 容器启动后, Scheduler 自动开始工作, 而在 Spring 容器关闭前, 自动关闭 Scheduler。为此, Spring 提供了 SchedulerFactoryBean, 这个 FactoryBean 大致拥有以下功能。

- ❑ 以更具 Bean 风格的方式为 Scheduler 提供配置信息。
- ❑ 让 Scheduler 和 Spring 容器的生命周期建立关联, 相生相息。
- ❑ 通过属性配置的方式代替 Quartz 自身的配置文件。

来看一个 SchedulerFactoryBean 配置的例子, 如代码清单 16-9 所示。

代码清单 16-9 SchedulerFactoryBean 配置

```

<bean id="scheduler"
    class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
    <property name="triggers"> ① ← 注册多个 Trigger
        <list>
            <ref bean="simpleTrigger" />
        </list>
    </property>
    <property name="schedulerContextAsMap"> ② ← 以 Map 类型设置 SchedulerContext 数据
        <map>
            <entry key="timeout" value="30" />
        </map>
    </property>

    <property name="configLocation"
        value="classpath:com/smart/quartz/quartz.properties" /> ③ ← 显式指定 Quartz 的配置文件地址
    </bean>

```

SchedulerFactoryBean 的 triggers 属性为 Trigger[] 类型, 可以通过该属性注册多个 Trigger, 在 ① 处注册了一个 Trigger。Scheduler 拥有一个类似于 ServletContext 的

SchedulerContext。SchedulerFactoryBean 允许用户以 Map 的形式设置 SchedulerContext 的参数值，如②处所示。在默认情况下，Quartz 在类路径下查询 quartz.properties 配置文件，用户也可以通过 configLocation 属性显式指定配置文件的位置，如③处所示。

除了实例中所用的属性外，SchedulerFactoryBean 还拥有一些常见的属性。

- ❑ calendars: 类型为 Map，通过该属性向 Scheduler 注册 Calendar。
- ❑ jobDetails: 类型为 JobDetail[]，通过该属性向 Scheduler 注册 JobDetail。
- ❑ autoStartup: SchedulerFactoryBean 在初始化后是否马上启动 Scheduler，默认为 true。如果设置为 false，则需要手工启动 Scheduler。
- ❑ startupDelay: 在 SchedulerFactoryBean 初始化完成后，延迟多少秒启动 Scheduler，默认值为 0，表示马上启动。除非拥有需要立即执行的任务，一般情况下，可以通过 startupDelay 属性让 Scheduler 延迟一小段时间后启动，以便让 Spring 能够更快初始化容器中剩余的 Bean。

SchedulerFactoryBean 的一个重要功能是允许用户将 Quartz 配置文件中的信息转移到 Spring 配置文件中，由此带来的好处是配置信息的集中化管理，同时我们不必熟悉多种框架有差异的配置文件结构。回忆一个 Spring 集成的 Hibernate 框架，就知道这是 Spring 在集成第三方框架时采用的惯用招数。SchedulerFactoryBean 通过以下属性代替框架的自身配置文件。

- ❑ dataSource: 当需要使用数据库来持久化任务调度数据时，用户可以在 Quartz 中配置数据源，也可以直接在 Spring 中通过 dataSource 指定一个 Spring 管理的数据源。如果指定了该属性，那么，即使在 quartz.properties 中已经定义了数据源，也会被 dataSource 覆盖。
- ❑ transactionManager: 可以通过该属性设置一个 Spring 事务管理器。在设置 dataSource 时，Spring 强烈推荐用户使用一个事务管理器，否则数据表锁定可能无法正常工作。
- ❑ nonTransactionalDataSource: 在全局事务的情况下，如果用户不希望 Scheduler 执行的相关数据操作参与到全局事务中，则可以通过该属性指定数据源。在 Spring 本地事务的情况下，使用 dataSource 属性就足够了。
- ❑ quartzProperties: 类型为 Properties，允许用户在 Spring 中定义 Quartz 的属性，其值将覆盖 quartz.properties 配置文件中的设置。这些属性必须是 Quartz 能够识别的合法属性，在配置时，需要查看 Quartz 的相关文档。下面是一个配置 quartzProperties 属性的例子：

```
<bean id="scheduler" class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
...
  <property name="quartzProperties">
    <props>
      <prop key="org.quartz.threadPool.class">① ← Quartz 属性项 1
        org.quartz.simpl.SimpleThreadPool
      </prop>
      <prop key="org.quartz.threadPool.threadCount">10</prop>② ← Quartz 属性项 2
    </props>
  </property>
</bean>
```

```
</props>
</property>
</bean>
```

在实际应用中，我们并不总是在程序部署的时候就确定需要哪些任务，往往需要在运行期根据业务数据动态产生触发器和任务。用户完全可以在运行期通过代码调用 `SchedulerFactoryBean` 获取 `Scheduler` 实例，然后动态注册触发器和任务。

16.4 在 Spring 中使用 JDK Timer

在 Java 1.3 以后的版本中，通过 `java.util.Timer` 和 `java.util.TimerTask` 这两个类提供了简单的任务调度功能，称之为 Java Timer。Java Timer 允许按照固定频率重复执行某项任务，这比直接通过编写底层线程程序进行任务调度要轻松许多，但是对于诸如“在每周周一 8:00 执行”这种和日历相关的任务调度需求来说，Java Timer 就无能为力了。

此外，JDK Timer 只适合对执行时间非常短的任务进行调度，因为在 Timer 中所有的 `TimerTask` 都在同一背景线程中执行，长时间的任务会严重影响到 Timer 的调度工作。所以 Java Timer 的使用范围很窄，在业务需求超过 Java Timer 的能力范围时，用户应该考虑前面介绍的 Quartz 框架。

16.4.1 Timer 和 TimerTask

`TimerTask` 代表一个需要多次执行的任务，它实现了 `Runnable` 接口，可以在 `run()` 方法中定义任务逻辑。而 `Timer` 负责制定调度规则并调度 `TimerTask`。

1. TimerTask

`TimerTask` 相当于 Quartz 中的 `Job`，代表一个被调度的任务。二者最主要的区别在于，每当执行任务时，Quartz 都会创建一个 `Job` 实例，而 JDK Timer 则使用相同的 `TimerTask` 实例。所以，如果 `TimerTask` 类中拥有状态，那么这些状态对于后面的执行是可见的。从这点上来说，`TimerTask` 更像 `StatefulJob` 而非 `Job`。`TimerTask` 实现了 `Runnable` 接口，是一个抽象类，它只有以下 3 个方法。

- ❑ `abstract void run()`：子类覆盖这个方法并定义任务执行逻辑，每次执行任务时，`run()` 方法就被调用一次。
- ❑ `boolean cancel()`：取消任务。假设任务被安排执行 N 次，那么在调用该方法后，后续的执行安排将取消。
- ❑ `long scheduledExecutionTime()`：返回此任务的计划执行时间点。如果在任务执行过程中调用此方法，则返回此次执行所对应的计划执行时间（一个任务的实际执行时间和计划执行时间可能不一致）。该方法一般在 `run()` 方法内调用，用户可以通过该方法判断本次执行的时间点是否过晚，并据此决定是否要取消本次执行。该方法一般在固定频率执行时使用才会有意义。

2. Timer

Timer 只能以这样的方式对任务进行调度：在延迟一段时间或在指定时间点后运行一次任务或周期性地运行任务。实际上，在 Timer 内部使用 `Object#wait(long time)` 进行任务的时间调度。这种机制不能保证任务的实时执行，只是一个粗略的近似值。

每个 Timer 对象都有一个对应的“背景线程”，它负责调度并执行 Timer 中所有的 TimerTask。由于所有的 TimerTask 都在这个线程中执行，所以 TimerTask 的执行时间应该比较短。如果一个 TimerTask 的执行占用了过多的时间，那么后面的任务就会受到影响。由于后续任务在调度时间轴上受到了“挤压”，所以可能会造成“扎堆”执行的情况。

当 Timer 中所有的 TimerTask 已经执行完成并且 Timer 对象没有外部引用时，Timer 的任务执行线程才会结束，但这可能需要很长的时间。因此，Timer 在默认情况下使用非守护线程 (daemon Thread)，这样用户就可以在应用程序中通过 `Timer#cancel()` 方法手工结束 Timer。如果希望尽快结束 Timer 中的任务，则可以调用 `TimerTask#cancel()` 方法来达到目的。



提示

Java 有两种线程：用户线程 (User Thread) 和守护线程 (Daemon Thread)。守护线程是指在程序后台运行，提供一种通用服务的线程，垃圾回收线程就是一种典型的守护线程。守护线程是为主程序服务的，因此，当所有的用户线程都结束时，守护线程就会自动终止。因此，也可以将守护线程形象地称为奴仆线程，“主在我存，主亡我死”。将线程转换为守护线程可以通过调用 Thread 对象的 `setDaemon(true)` 方法来实现。

Timer 的构造函数在创建 Timer 对象的同时将启动一个 Timer 背景线程。先来了解一下 Timer 的几个构造函数。

- `Timer()`：创建一个 Timer，背景线程是非守护线程。
- `Timer(boolean isDaemon)`：创建一个 Timer，当 `isDaemon` 为 `true` 时，背景线程为守护线程，守护线程将在应用程序主线程停止后自动退出。该方法是 Java 5.0 新增的。
- `Timer(String name)`：与 `Timer()` 类似，只是通过 `name` 为关联背景线程指定名称。
- `Timer(String name, boolean isDaemon)`：与 `Timer(boolean isDaemon)` 类似并为关联背景线程指定名称。该方法是 Java 5.0 新增的。

通过以下方法执行一次任务。

- `schedule(TimerTask task, Date time)`：在特定时间点执行一次任务。
- `schedule(TimerTask task, long delay)`：延迟指定时间后执行一次任务，`delay` 的单位为毫秒。

通过以下方法按固定间隔执行任务，间隔时长为上次任务执行完成的时间点到下次任务开始执行的时间点，任务的执行可能产生时间的漂移。

- ❑ `schedule(TimerTask task, Date firstTime, long period)`: 从指定时间点开始周期性地执行任务，`period` 的单位为毫秒，后一次执行将在前一次执行完成后才开始计时。如任务被安排每 2 秒执行一次，假设第一次任务在 0 秒时间点开始执行并花费了 1.5 秒，则第二次将在第 3.5 秒时执行。
- ❑ `schedule(TimerTask task, long delay, long period)`: 在延迟指定时间后，周期性地执行任务。

通过以下方法按固定频率执行任务。

- ❑ `scheduleAtFixedRate(TimerTask task, Date firstTime, long period)`: 在指定时间点后，以指定频率执行任务。假设一个任务被安排每 2 秒执行一次，如果第一次执行花费了 1.5 秒，则在 0.5 秒后，第二次任务又会开始执行，以保证固定的执行频率。
- ❑ `scheduleAtFixedRate(TimerTask task, long delay, long period)`: 在延迟一段时间后，以指定频率执行任务。

此外，Timer 还拥有几个控制方法。

- ❑ `cancel()`: 取消 Timer 的执行，并丢弃所有被调度的 TimerTask，不过正在执行的任务不受影响。Timer 被取消后，不能调度新的 TimerTask。
- ❑ `purge()`: 将所有已经取消的 TimerTask 从 Timer 列队中清除。如果 TimerTask 没有外部引用，就可以被垃圾回收。一般情况下无须调用该方法，只有在某些特殊情况下，当一次性取消多个 TimerTasks 后，调用该方法才有意义。

3. Java Timer 实例

首先创建一个任务，并在执行 10 次以后退出，代码如下：

```
package com.smart.basic.timer;
import java.util.Date;
import java.util.TimerTask;
public class SimpleTimerTask extends TimerTask{
    private int count = 0;
    public void run() {
        System.out.println("execute task.");
        Date exeTime = (new Date(scheduledExecutionTime())); ① ← 获取本次安排执行的时间点
        System.out.println("本次任务安排执行时间点为: "+exeTime);
        if(++count > 10){cancel();} ② ← 在任务执行 10 次后主动退出
    }
}
```

通过扩展 TimerTask 并实现 run() 抽象方法定义一个任务。在 JDK Timer 中没有指定执行特定次数任务的方法。用户可以在任务的 run() 方法中通过自定义代码实现。

下面通过 Timer 以固定延迟时间的方式每 5 秒执行一次任务。

```
package com.smart.basic.timer;
import java.util.Timer;
import java.util.TimerTask;
```



```
public class TimerRunner {
    public static void main(String[] args) {
        Timer timer = new Timer();
        TimerTask task = new SimpleTimerTask();
        timer.schedule(task, 1000L, 5000L); ①
    }
}
```

在延迟 1 秒后，每 5 秒执行一次任务

运行以上代码，将输出以下信息：

```
execute task.
本次任务安排执行时间点为: Thu Mar 15 18:39:37 CST 2007
execute task.
本次任务安排执行时间点为: Thu Mar 15 18:39:42 CST 2007
execute task.
本次任务安排执行时间点为: Thu Mar 15 18:39:47 CST 2007
```

16.4.2 Spring 对 Java Timer 的支持

Spring 在 `org.springframework.scheduling.timer` 中提供了几个 JDK Timer 的支持类，主要在以下 3 个方面对 JDK Timer 提供支持。

(1) `ScheduledTimerTask`，它对 `TimerTask` 提供封装并提供相关的配置。

(2) 通过 `MethodInvokingTimerTaskFactoryBean` 类可以将一个 Bean 的方法封装为 `TimerTask`。

(3) 通过 `TimerFactoryBean` 可以更方便地配置 Timer。此外，让 Timer 的生命周期和 Spring 容器的生命周期相关，在初始化 `TimerFactoryBean` 后启动 Timer，在 Spring 容器关闭前取消 Timer。

1. ScheduledTimerTask

JDK Timer 标准的 API 要求在使用 Timer 方法进行任务调度时才指定调度的规则，这种方式不太适合进行 Bean 的配置，因此，Spring 提供了 `ScheduledTimerTask`，通过属性指定任务和调度规则。请看下面的代码：

```
<bean id="timerTask" class="com.smart.basic.timer.SimpleTimerTask" />

<bean id="scheduledTask"
    class="org.springframework.scheduling.timer.ScheduledTimerTask"
    p:timerTask-ref="timerTask1" ① ← 指定调度任务
    p:delay="1000" ② ← 延迟时间，单位为毫秒
    p:period="1000" /> ③ ← 周期时间，单位为毫秒
```

如果希望只运行一次任务，则将 `period` 设置为 0 或负值。在默认情况下，采用固定时间间隔的调度方式，可以通过 `fixedRate` 属性设置以固定频率的方式执行任务。`SimpleTimerTask` 还可以将实现了 `Runnable` 接口的类封装成一个任务，用户可以通过 `runnable` 属性进行设置。

2. MethodInvokingTimerTaskFactoryBean

类似于 Quartz 的 `MethodInvokingJobDetailFactoryBean`，Spring 也为 JDK Timer 提供

了一个方便类，用于将 Bean 方法封装成一个 TimerTask。

```
<bean id="myService" class="com.smart.service.MyService" />

<bean id="timerTask1" ① ← 将返回一个 TimerTask 实例
    class="org.springframework.scheduling.timer.MethodInvokingTimerTask FactoryBean"
    p:targetObject-ref="myService" ② ←
    p:targetMethod="doJob" /> ③ ← 业务方法 业务 Bean
```

3. TimerFactoryBean

类似于 Quartz 的 SchedulerFactoryBean，Spring 为 Timer 提供了 TimerFactoryBean 类。用户可以将多个 ScheduledTimerTask 注册到 TimerFactoryBean 中，TimerFactoryBean 将返回一个 Timer 实例。在 TimerFactoryBean 初始化完成后，对应的 Timer 启动，在 Spring 容器关闭之前，TimerFactoryBean 将取消 Timer。请看下面的配置代码：

```
<bean id="timer" class="org.springframework.scheduling.timer.TimerFactoryBean">
    <property name="scheduledTimerTasks">
        <list>
            <ref bean="scheduledTask" />
        </list>
    </property>
</bean>
```

scheduledTimerTasks 属性的类型为 ScheduledTimerTask[]，可以注入多个 ScheduledTimerTask。此外，TimerFactoryBean 还拥有一个 daemon 属性，指定生成 Timer 的背景线程是否为守护线程。

16.5 Spring 对 Java 5.0 Executor 的支持

Java 5.0 新增了一个并发工具包 java.util.concurrent，该工具包由 Doug Lea 设计并作为 JSR-166 添加到 Java 5.0 中。这是一个非常流行的并发工具包。它提供了功能强大的、高层次的线程构造器，包含执行器、线程任务框架、线程安全队列、计时器、锁（包含原子级别的锁）和其他一些同步的基本类型。

执行器 Executor 是并发工具包中一个重要的类，它对 Runnable 实例的执行进行了抽象，实现者可以提供具体的实现，如简单地以一个线程来运行 Runnable，或者通过一个线程池为 Runnable 提供共享线程。

因为 Executor 是 Java 5.0 新增的类，所以 Java 5.0 提供的实现类大多拥有线程池的内在支持。Spring 为 Executor 处理引入了一个新的抽象层，以便将线程池引入 Java 1.3 和 Java 1.4 环境中，同时屏蔽掉 Java 1.3、1.4、5.0 及 Java EE 环境中线程池实现的差异。

16.5.1 了解 Java 5.0 的 Executor

java.util.concurrent.Executor 接口的主要目的是将“任务提交”和“任务执行”分离解耦。该接口定义了任务提交的方法，实现者可以提供不同的任务执行机制，指定不同的线程使用规则和调度方案。

Executor 只有一个方法：void execute(Runnable command)，它接收任何实现了 Runnable 的实例，这个实例代表一个待执行的任务。可以使用如下代码提交任务：

```
Executor executor = anExecutor;
executor.execute(new RunnableTask1()); ①
executor.execute(new RunnableTask2()); ②
```

提交一个任务
提交另一个任务

Executor 本身并没有要求实现者以何种方式执行这些任务，一个简单的实现类甚至可以在提交任务时，立即在主线程中执行它们。下面是一个 Executor 最简单的实现：

```
public class SimpleExecutor implements Executor {
    public void execute(Runnable r) {
        r.run(); ①
    }
}
```

在提交时直接
执行任务

但在更多的情况下，需要在一个异步线程中执行任务，而非在主线程中执行任务。下面是一个稍微有意义的实现，它为每个任务开启一个新的执行线程。

```
class ThreadPerTaskExecutor implements Executor {
    public void execute(Runnable r) {
        new Thread(r).start(); ①
    }
}
```

在新的执行线程
中执行任务

但以上这些粗陋的实现并没有多大的实际应用价值，这也不符合 Executor 的设计初衷。真正有意义的实现需要引入线程、列队、调度等机制，这样的执行器才更贴近现实的需求。

Executor 接口还有两个子接口：ExecutorService 和 ScheduledExecutorService。ExecutorService 添加了结束任务的管理方法，此外，在提交任务时还可以获取一个 Future 实例，以便通过这个实例跟踪异步任务的执行情况。而 ScheduledExecutorService 可以对任务进行调度，如指定执行的延迟时间及执行的周期。如 ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit)方法将安排一个任务在延迟一段时间后执行。

Java 5.0 本身提供的 ThreadPoolExecutor 类实现了 Executor 和 ExecutorService 这两个接口，它使用一个线程池对提交的任务进行调度。如果需要处理数量巨大的短小并发任务（如 Web 服务器、数据库服务器、邮件服务器之类的应用程序需要处理来自远程的大量短小的任务），则采用线程池可以带来明显的好处。为每个请求创建一个新线程的开销很大，对于短小的任务，线程创建和销毁的时间可能比处理实际任务的时间还要长。除此以外，活动的线程需要消耗资源，过多的线程将导致大量的内存占用。线程池能让多个任务重用同一个线程，线程创建的开销被分摊到多个任务上。此外，由于在任

务到达时线程已经存在，所以也消除了任务执行时因创建线程所带来的延迟。可以通过适当地调整线程池中的参数，当任务的数目超过某个阈值时，强制新任务排队等待，直到获得一个线程来处理，从而防止资源无限制消耗而引发的系统问题。

ThreadPoolExecutor 的子类 ScheduledThreadPoolExecutor 实现了 ScheduledExecutorService 接口，添加了对任务的调度功能，如指定延迟一小段时间后执行任务，让任务周期性执行。该类明显优于 Java 1.3 中的 Timer，因为它通过内建的线程池让每个任务在独立的执行线程中执行，而非让所有任务在单一的背景线程中执行。Timer 经常出现的时间漂移、任务挤压等问题基本上得到了规避。

在 java.util.concurrent 中为创建这些接口实例提供了一个综合性的工厂类 Executors，它拥有以下众多方便的静态工厂方法。

- `public static ExecutorService newFixedThreadPool(int nThreads)`: 创建一个线程池，重复使用一组固定的线程执行任务。
- `public static ExecutorService newCachedThreadPool()`: 线程池是动态的，不够用时将创建新的线程，长时间不用的线程将被收回。
- `public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize, Thread Factory threadFactory)`: 创建一个线程池，可在指定延迟后执行或者定期执行。

来看一个具体的例子，如代码清单 16-10 所示。

代码清单 16-10 ExecutorExample

```
package com.smart.basic.executor;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
public class ExecutorExample {
    private Executor executor; ① ← 声明一个执行器
    public void setExecutor(Executor executor) {
        this.executor = executor;
    }
    public void executeTasks() { ② ← 用执行器执行多个任务
        for (int i = 0; i < 6; i++) {
            executor.execute(new SimpleTask("task" + i));
        }
    }
    public static void main(String[] args) {
        ExecutorExample ee = new ExecutorExample();
        ee.setExecutor(Executors.newFixedThreadPool(3)); ③ ← 通过工厂类创建一个带 3 个线程的固定线程池的执行器
        ee.executeTasks();
    }
}
class SimpleTask implements Runnable { ④ ← 任务类
    private String taskName;
    public SimpleTask(String taskName) {
        this.taskName = taskName;
    }
    public void run() {
```

```

        System.out.println("do "+taskName+"... in Thread:" + Thread.currentThread().
getId());
    }
}

```

运行以上代码，输出以下信息：

```

do task0... in Thread:7
do task1... in Thread:8
do task2... in Thread:9
do task3... in Thread:7
do task5... in Thread:9
do task4... in Thread:8

```

可见，这 6 个任务共享了线程池中的 3 个线程。由于 `ExecutorService` 用线程池中的 3 个线程服务于提交的任务，从而避免了为每个任务创建独立线程的代价，具有更高的运行性能。

16.5.2 Spring 对 Executor 所提供的抽象

Spring 的 `org.springframework.core.task.TaskExecutor` 接口等同于 `java.util.concurrent.Executor` 接口。该接口和 Java 5.0 的 `Executor` 接口拥有相同的 `execute(Runnable task)` 方法。`TaskExecutor` 拥有一个 `SchedulingTaskExecutor` 子接口，新增了任务调度规则定制的功能。

在 Spring 发行包中预定义了一些 `TaskExecutor` 实现，它们可以满足大部分应用要求，一般情况下，用户不必自行编写实现类。下面是 `TaskExecutor` 的实现类。

SyncTaskExecutor: 位于 `org.springframework.core.task` 包中，实现了 `TaskExecutor` 接口。这个实现不会异步执行任务；相反，每次调用都在发起调用的主线程中执行。

下面是 `SchedulingTaskExecutor` 的实现类。

- ❑ **SimpleAsyncTaskExecutor:** 位于 `org.springframework.core.task` 包中。该类没有使用线程池，在每次执行任务时都创建一个新线程。但是，它依然支持对并发总数设限，当超过并发总数限制时，阻塞新的任务直到有可用的资源。
- ❑ **ConcurrentTaskExecutor:** 位于 `org.springframework.scheduling.concurrent` 包中。该类是 Java 5.0 的 `Executor` 的适配器，以便将 Java 5.0 的 `Executor` 当作 Spring 的 `TaskExecutor` 使用。
- ❑ **SimpleThreadPoolTaskExecutor:** 位于 `org.springframework.scheduling.quartz` 包中。该类实际上是继承于 Quartz 的 `SimpleThreadPool` 类的子类，它将监听 Spring 的生命周期回调。当用户有线程池，需要在 Quartz 和非 Quartz 组件中共用时，该类可以发挥它的用处。
- ❑ **ThreadPoolTaskExecutor:** 位于 `org.springframework.scheduling.concurrent` 包中。该类只能在 Java 5.0 中使用，它暴露了一些属性，方便在 Spring 中配置一个 `java.util.concurrent.ThreadPoolExecutor`，并把它包装成 `TaskExecutor`。

- ❑ **TimerTaskExecutor**: 位于 `org.springframework.scheduling.timer` 包中。该类使用一个 `Timer` 作为其后台实现。

代码清单 16-10 中的 `ExecutorExample` 只能在 Java 5.0 下运行, 如果用 Spring 的 `TaskExecutor` 替换①处的 `Executor`, 当程序需部署到低版本的 Java 环境中时, 仅需要选择一个合适的实现就可以了。

```
package com.smart.executor;
import org.springframework.core.task.SimpleAsyncTaskExecutor;
import org.springframework.core.task.TaskExecutor;
public class ExecutorExample {
    private TaskExecutor executor; ①
    public void setExecutor(TaskExecutor executor) {
        this.executor = executor;
    }
    public void executeTasks() {
        for (int i = 0; i < 6; i++) {
            executor.execute(new SimpleTask("task" + i));
        }
    }
    public static void main(String[] args) {
        ExecutorExample ee = new ExecutorExample();
        ee.setExecutor(new SimpleAsyncTaskExecutor()); ②
        ee.executeTasks();
    }
}
class SimpleTask implements Runnable {
    ...
}
```

使用 Spring 的 TaskExecutor 替换 Java 5.0 的 Executor

使用 Spring 的 TaskExecutor 实现类

如果将 `ExecutorExample` 配置成一个 Bean, 通过注入的方式提供 `executor` 属性, 就可以方便地选用不同的实现版本。如果是在 Java 5.0 版本中, 则用户可以选用 `ThreadPoolTaskExecutor`; 而在 JDK 低版本中则可以使用 `SimpleAsyncTaskExecutor`。这样, 程序就可以在不同的 JDK 版本中进行移植。

16.6 实际应用中的任务调度

对于那些运行规则固定的静态任务(如每隔 30 分钟更新缓存), 当然可以通过 Spring 配置文件定义调度规则并在 Spring 容器中启动运行调度。但事情并非总是这么简单, 在实际应用中, 往往需要根据业务数据动态产生任务。举个例子: 在论坛系统中, 当发现某些用户发表一些非法的内容时, 作为惩罚手段, 往往需要将该用户锁定一段时间。这个功能就需要通过操作业务功能动态地创建任务来完成: 在锁定用户后, 经过一段时间, 通过任务自动将其解锁。随着业务系统的运行, 任务源源不断地产生, 又源源不断地得到执行。

在实际的应用系统中, 任务的执行时间点往往是非常关键的, 不允许其发生时间漂

移(如电力控制系统)。此外,任务的执行往往是与日历相关的(如在本月 15 日上午 10:00 执行),所以 Quartz 往往是更适合的选择。

16.6.1 如何产生任务

在实际应用中,有两种不同的创建动态任务的方式。

(1) 业务流程产生型。

(2) 扫描线程产生型。

前者表示在运行一个业务时,在业务操作过程中产生任务;而后者由一个任务线程定时扫描业务数据库的任务表,并根据业务数据产生任务。

1. 在业务流程中产生

某个业务需要用到任务,如果任务的执行时间点离业务的操作时间点不是太长,则可以直接在业务流程中安排好任务。假设在电力传输管理系统中有一个功能,可以将一条传输线路在某段时间内停止供电。安排停电的时间点离业务的操作时间点一般不会太长,因此,可以在用户执行线路停电安排的业务时,立即向 Scheduler 中注册两个任务:一个是在某一时间点执行断电的任务;另一个是在某一时间点执行恢复供电的任务。

当然,考虑到系统重启的情况,可能需要使用持久化任务,以便在系统重启后能够恢复已经安排的任务。

2. 由扫描任务周期性产生

但在有些情况下不适合在业务流程中产生任务,而应当通过定期扫描功能,根据业务数据动态产生任务。

在论坛系统中,为了清除一些无效的注册账号,可以定义这样的规则:如果账号注册后在半年内都没有激活,则将这些账号删除。对于系统来说,引发这个潜在账号清除任务的业务是用户注册功能,但是不应该在注册账号后就安排一个清除账号的任务:首先,在注册用户时,并不知道这个账号在将来是否会满足清除的条件;其次,任务的执行时间点离当前点太远,现在就安排显然“为时过早”。

一般来说,像清除账号这样的任务对执行时间点的要求并不会太高,所以只要安排一个以天为周期定时执行的任务,将当天满足条件的账号删除即可。

现在来考虑另一种比较极端的需求。假设对清除账号的执行时间点有严格的要求:清除账号的时间点只能精确地发生在一年之后的那个时间点上($365 \times 24 \times 60 \times 60 \times 1000$ 毫秒之后)。这时,定期周期执行的清除任务就不再适用了。因为如果清除任务执行频率过小,就不能满足精确执行点的要求;如果频率过大,则会对业务数据库产生频繁访问,对数据库的性能产生不良影响。此外,在需要访问数据库的情况下,单纯从技术角度来看,任务执行频率本身是受限的,用户很难安排秒级的任务。

为了保证严格的执行时间点并尽量减小对数据库的影响,需要一个用于产生最近执

行任务的扫描任务定期查询数据库，并为那些在一小段时间后就要执行的潜在任务进行动态安排，如图 16-2 所示。

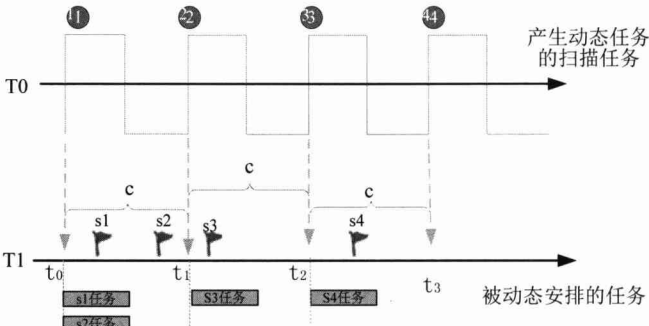


图 16-2 通过周期性扫描动态安排近期某个时间点执行的任务

T0 对应一个定时的任务，它负责周期性地扫描业务表，找出出在后续的扫描周期时间范围内将要执行的任务，创建并安排这些任务。通过这种方式可以带来 3 个好处。

(1) 降低对数据库的影响：扫描任务自身的执行频率不必太高，如可以让扫描任务 30 分钟执行一次，将 30 分钟后所有需要执行的任务都安排好。这样，扫描任务在 30 分钟之内只会进行一次数据库访问，对数据库的影响微乎其微。

(2) 缩短调度器中任务列队的长度：由于不是将所有的潜在任务提前很长时间就进行安排，而仅对一个扫描周期内的任务进行安排，所以调度器中任务列表的长度可以得到有效的控制。

(3) 保证任务在精确的时间点执行：由于任务提前得到安排，任务执行时间点的精确性可以得到很好的保证。当然，由于扫描任务本身需要一定的执行时间，如果任务的执行时间点离扫描任务的扫描点很近，则执行时间点的精确性将会受到一些影响。这时，用户可以将安排任务的范围调大一些，以取消这种任务执行点和扫描周期点过近的问题。不过，一般情况下，这种影响是可以接受的。

如果系统存在许多依赖一个主任务动态创建出来的子任务，那么最好通过一张任务表来维护这些任务，而不要将它们分散在不同的业务表中。假设用户账号清除任务的条件信息在用户表中，锁定用户任务的信息在锁定用户表中，商品过期任务的信息在商品表中。这时就需要分别为这些任务定义一个扫描任务，或者在扫描任务内遍历所有相关的业务表。如果和任务有关的业务表非常多（如 100 张），为了动态创建任务，扫描任务就会对数据库产生很大的影响。如果通过一张任务表记录所有潜在的任务，并在业务操作过程中动态维护这张任务表，则仅需一个扫描任务查询这张任务表就可以了，有效地降低了创建动态任务对数据库的影响。使用统一任务表的方式也是有代价的，任务有关的业务模块在业务功能之外需要额外地考虑维护任务表中的数据。比如，在用户注册模块，在用户注册完成后，就向任务表添加一条清除账号的任务；而在用户激活模块，则需要将清除该用户账号的任务记录删除。

16.6.2 任务调度对应用程序集群的影响

对于有集群要求的 Web 应用程序来说,如果应用系统本身有任务调度的功能,就必须在系统设计初期仔细分析任务调度功能是否适合集群。按任务执行结果影响的范围,可以将任务为两种类型。

- 全局任务:指那些执行结果会影响到应用系统全局的任务。例如,每天凌晨生成业务报表、定期调用短信接口发送短信、定期清除系统过期数据等任务,它们的执行都会给整个系统带来“全局可见”的结果。所以,在传统的集群系统中,全局任务最好在一个独立部署的服务节点上执行,否则可能会因重复多次执行而引发系统逻辑的错误。
- 本地任务:和全局任务相对,指执行结果的影响范围仅限于本地,不会造成全局影响的任务。如定期刷新本地缓存、定期清除本地节点临时文件等任务,它们的执行结果只对本地服务节点有影响,需要在每个本地服务节点部署任务。

Quartz 可支持集群部署,其原理很简单,即让多个调度节点互为热备,在同一时刻只有一个调度节点是激活的,任务只在这个激活的调度节点中执行,其他的调度节点处于“休眠”状态;当激活的调度节点崩溃时,则唤醒某一个“休眠”的调度节点,以接管任务调度的工作。

Quartz 可通过两种方式实现集群:其一是通过一个中间数据库,使集群节点相互感知,以实现故障切换;其二是通过 Terracotta 实现集群。Terracotta 是一个 JVM 级的开源集群框架,简单地说,就是将多个 JVM 透明化为一个 JVM 使用。如何配置 Quartz 集群已经超出了本书的讨论范围,读者可查阅相关资料了解。

16.6.3 任务调度云

虽然 Quartz 支持集群方案,但其集群方案存在一个比较大的问题,即无法区分全局任务和本地任务,只能以调度节点为单位开启或关闭任务。对于既拥有全局任务又拥有本地任务的应用系统来说,Quartz 集群很难满足要求。此外,Quartz 缺少集群调度节点和任务管理控制台,不便进行节点调度和任务监控。

鉴于此,笔者曾基于 Quartz 研发了一个任务调度集群方案,称其为“任务调度云”,它实现了以下几个目标:

- 任务调度服务无须双机热备硬件支持,具有“软故障迁移”功能,避免了系统的单点故障。
- 任务调度服务无须独立部署,且不会造成“单任务多次运行”的问题。
- 任务调度的服务器可分配、可管理、可切换、可监控。
- 任务本身可管理、可控制、可监控。
- 任务的程序开发简单易行,对调度运行环境完全透明。

任务调度云的基础设施由任务调度资源识别器、故障迁移监控器、控制命令执行器及运行日志记录器组成,如图 16-3 所示。其基础实现原理是通过协商机制将云中的某个服务节点标志为“主节点”,全局任务只在主节点中执行,本地任务在各个节点都执行。通过心跳机制监测各节点的活动状态,如果主节点不活动了,则由协商机制选取云中的下一个活动节点作为主节点。

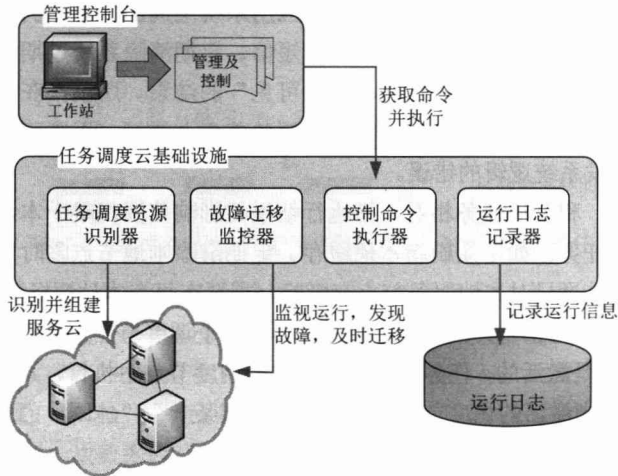


图 16-3 任务调度云

在任务调度云启动时,任务调度云的资源识别器将首先开始工作,根据管理控制台的资源定义将指定的服务器加入任务调度云中,以完成任务调度云的组建。

而故障迁移器将负责监控云中任务调度服务器的运行状态。当发生故障时,及时将任务迁移到备用任务调度服务器中,实现调度服务的热切换,达到故障迁移的目的。

任务调度管理员可在管理控制台向任务调度云发布控制命令,控制命令执行器在发现新命令后,下载命令并在云中执行命令。

运行日志记录器负责记录任务调度云中任务运行的详细日志,包括运行开始时间、结束时间、运行结果及运行在云中的具体位置等,以便管理员对任务运行进行审查和监控。

从任务的构成上看,一个任务由两个方面组成:其一是被调度的执行逻辑;其二是调度规则。执行逻辑可以描述为一个接口方法,而调度规则可通过周期表达式或 Cron 表达式指定。在任务调度云中,调度规则在管理控制台中定义和维护,而执行逻辑通过 Job 接口描述。Job 接口定义了如下方法。

- `execute()`: 该方法承载任务执行的逻辑,开发者只需实现该方法并在其中定义需要进行调度的执行逻辑即可。
- `isGlobal()`: 确定任务是全局任务还是本地任务,返回 `true` 表示是全局任务,否则为本地任务。本地任务在所有的任务调度节点都被执行,而全局任务只会在集群中某个激活的调度节点中执行。

- ❑ `isNeedLog()`: 是否要让任务调度云记录任务的运行日志, 可以在管理控制台查看到任务运行的日志。
- ❑ `getJobGroupName()`和 `getJobName()`: 任务全限定名, 由分组名+任务名组成。任务调度云通过任务全限定名将调度规则应用于特定的任务上, 同时还通过全限定名进行名称的定位和控制。

在任务调度云中开发一个任务非常简单, 只需扩展任务抽象类实现 `execute()`方法就可以了。以下就是一个具体任务的例子:

```
@Component("simpleJob")
public class SimpleJob extends AbstractBeanJob {
    public void execute(JobExecutionContext jobExecutionContext) {
        System.out.println("执行任务逻辑!");
    }
}
```

将任务类以一个 Bean 的方式进行定义, 而任务全限定名直接取自 Bean 的名称, 无须额外设定。Job 的接口方法在 `AbstractBeanJob` 抽象类中都有默认值, 默认值指定这个 `SimpleJob` 是一个全局任务, 需要任务运行日志且任务的全限定名为 `DEFAULT.simpleJob`。在管理控制台注册一个名为 `DEFAULT.simpleJob` 的任务, 它提供了任务对应的调度规则。

笔者已经将任务调度云的完整方案整理成一篇技术文章《面向云计算的 EPGIS 平台中的任务调度服务研究》, 发表在《电力信息化》2011 年第 5 期中, 感兴趣的读者可以参考阅读。

16.6.4 Web 应用程序中调度器的启动和关闭问题

我们知道, 静态变量是 `ClassLoader` 级别的, 如果 Web 应用程序停止, 那么这些静态变量也会从 JVM 中清除。但线程是 `JVM` 级别的, 如果用户在 Web 应用中启动了一个线程, 那么这个线程的生命周期并不会和 Web 应用程序保持同步。也就是说, 即使停止了 Web 应用, 这个线程依旧是活动的。正是因为这个很隐晦的问题, 所以很多有经验的开发者不太赞成在 Web 应用中私自启动线程。

如果手工使用 `JDK Timer` (`Quartz` 的 `Scheduler`), 在 Web 容器启动时启动非守护线程的 `Timer`, 当 Web 容器关闭时, 除非用户手工关闭这个 `Timer`, 否则 `Timer` 中的任务还会继续执行。

下面通过一个小例子来演示这个“灵异现象”。通过 `ServletContextListener` 在 Web 容器启动时创建一个 `Timer` 并周期性地执行一个任务, 如代码清单 16-11 所示。

代码清单 16-11 StartCycleRunTask: 容器监听器

```
package com.smart.web;
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;
```

```

import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
public class StartCycleRunTask implements ServletContextListener {
    private Timer timer;

    public void contextDestroyed(ServletContextEvent arg0) {①
        System.out.println("Web应用程序启动关闭...");
    }
    public void contextInitialized(ServletContextEvent arg0) {②
        System.out.println("Web应用程序启动...");
        timer = new Timer();②-1
        TimerTask task = new Simple TimerTask();
        timer.schedule(task, 1000L, 5000L);②-2
    }
}
class SimpleTimerTask extends TimerTask {③
    private int count;
    public void run() {
        System.out.println(++count+"execute task..." + (new Date()));
    }
}

```

该方法在 Web 容器关闭时执行

在 Web 容器启动时自动执行该方法

创建一个 Timer, 在 Timer 内部自动创建一个背景线程

注册一个每隔 5 秒执行一次的任务

任务

在 web.xml 中声明这个 Web 容器监听器。

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app>
...
<listener>
    <listener-class>com.smart.web.StartCycleRunTask</listener-class>
</listener>
</web-app>

```

在 Tomcat 中部署这个 Web 应用并启动后, 用户将看到任务每隔 5 秒执行一次, 在控制台上输出如图 16-4 所示的信息。

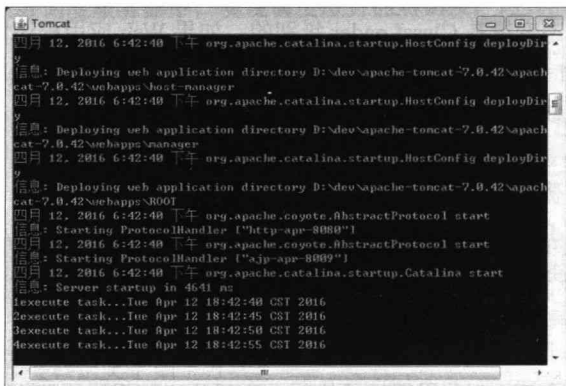


图 16-4 Web 应用程序启动, 任务周期执行

运行一段时间后, 登录 Tomcat 管理后台, 将对应的 Web 应用 (chapter16) 关闭, Tomcat 的后台管理界面如图 16-5 所示。

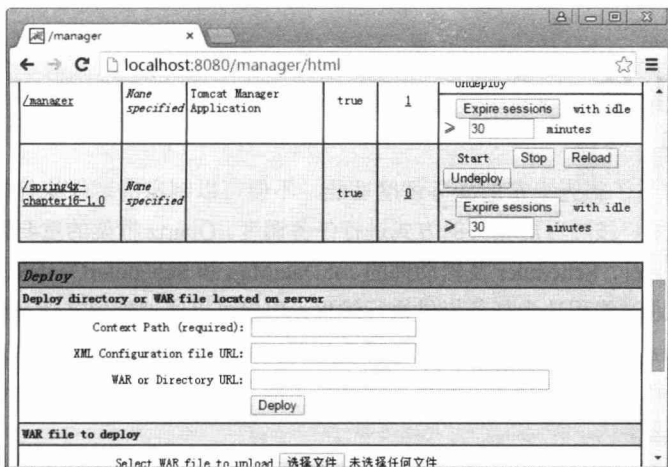


图 16-5 在 Tomcat 管理后台关闭相应的 Web 应用

转到 Tomcat 控制台，用户将看到，虽然 Web 应用已经关闭，但 Timer 任务还在“我行我素”地执行（见图 16-6）——舞台已拆，表演照旧。

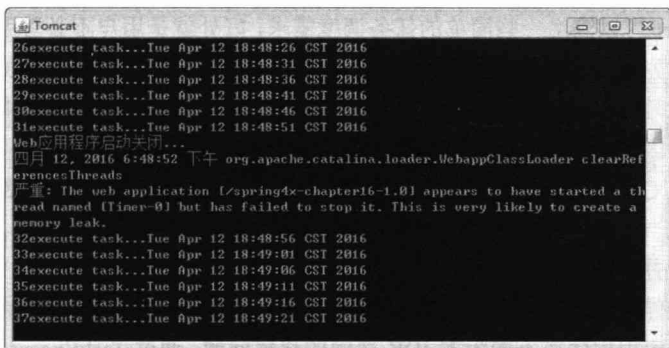


图 16-6 Web 应用关闭后，任务继续执行

可以通过改变代码清单 16-11 中的代码，在 `contextDestroyed(ServletContextEvent arg0)` 中添加 `timer.cancel()`，在 Web 容器关闭后手工停止 Timer 来结束任务。

Spring 为 JDK Timer 和 Quartz Scheduler 所提供的 `TimerFactoryBean` 和 `SchedulerFactoryBean` 能够与 Spring 容器的生命周期关联，在 Spring 容器启动时启动调度器，而在 Spring 容器关闭时停止调度器。所以在 Spring 中通过这两个 `FactoryBean` 配置调度器，再从 Spring IoC 中获取调度器的引用进行任务调度，这样就不会出现这种 Web 容器关闭而任务依然执行的问题。而如果用户在程序中直接使用 Timer 或 Scheduler，如不进行额外的处理，则会出现这一问题。

16.7 小结

Quartz 提供了极为丰富的任务调度功能，不但可以制定周期性执行的任务调度方案，还可以让用户按照日历相关的方式进行任务调度。Quartz 框架的重要组件包括 Job、JobDetail、Trigger、Scheduler 及辅助性的 JobDataMap 和 SchedulerContext。Quartz 拥有一个线程池，通过线程池为任务提供执行线程，用户可以通过配置文件对线程池进行参数定制。Quartz 的另一个重要功能是将任务调度信息持久化到数据库中，以便系统重启时能够恢复已经安排的任务。此外，Quartz 还拥有完善的事件体系，允许用户注册各种事件的监听器。

Spring 为 Quartz 的 JobDetail 和 Trigger 提供了更具 Bean 风格的支持类，这使得用户能够更方便地在 Spring 中通过配置定制这些组件实例。Spring 的 SchedulerFactoryBean 让用户可以脱离 Quartz 自身的配置体系，而以更具有 Spring 风格的方式定义 Scheduler。此外，还可以享受 Scheduler 生命周期和 Spring 容器生命周期绑定的好处。

JDK Timer 可以满足一些简单的任务调度需求，好处就是用户不必引用 JDK 之外的第三方类库，学习也很简单；但必须保证任务是短小型的任务，任务应该很快就能完成，否则将产生“时间漂移”的问题。此外，使用 JDK Timer 的任务对执行时间点应该没有严格的要求，因为 JDK Timer 只能做到近似的时间安排。

不管是在 Quartz 中还是在 JDK Timer 中，Spring 都可以将 Spring 容器中 Bean 的方法直接封装成一个任务，方便任务实例的制定。

对于大量并发的短小型任务，使用线程池进行任务调度可以带来明显的好处，如节省资源、增强伸缩性、获得更快的响应速度。Java 5.0 新增加了支持线程池的 Executor，Spring 考虑到应用程序在不同版本 JDK 之间的移植问题，通过 TaskExecutor 对 Executor 进行了抽象，这样，当应用部署到不同版本的 JDK 中时，仅需选择适合的实现类就可以了。

开发任务调度的应用程序并非想象中那样简单，在实际应用中，需要考虑到动态任务的产生机制，以照顾数据库性能、任务队列长度、任务执行时间点精度等方方面面的问题。此外，对于需要集群部署的 Web 应用程序，应该将业务应用和任务调度应用分别部署，以避免一个任务多次执行的问题。Spring 提供了用于创建 Quartz、JDK Timer 基础设施的 FactoryBean，可以保证 Quartz 和 JDK Timer 调度线程在 Spring 容器关闭时自动停止。

第 17 章

Spring MVC

大部分 Java 应用都是 Web 应用，展现层是 Web 应用不可忽略的重要环节。Spring 为展现层提供了一个优秀的 Web 框架——Spring MVC。和众多其他的 Web 框架一样，它基于 MVC 的设计理念。此外，它采用了松散耦合、可插拔的组件结构，比其他的 MVC 框架更具扩展性和灵活性。Spring MVC 通过一套 MVC 注解，让 POJO 成为处理请求的控制器，无须实现任何接口。同时，Spring MVC 还支持 REST 风格的 URL 请求：注解驱动及 REST 风格的 Spring MVC 是 Spring 的出色功能之一。此外，Spring MVC 在数据绑定、视图解析、本地化处理及静态资源处理上都有许多不俗的表现。它在框架设计、扩展性、灵活性等方面全面超越了 Struts、WebWork 等 MVC 框架，从原来的追赶者一跃成为 MVC 的领跑者。

本章主要内容：

- ◆ Spring MVC 体系概述
- ◆ 注解驱动的控制器
- ◆ 数据绑定、输入/输出格式化及数据校验
- ◆ 视图解析
- ◆ 本地化解析
- ◆ 文件上传
- ◆ WebSocket 支持
- ◆ 静态资源处理、请求拦截器、异常处理

本章亮点：

- ◆ 深入分析并图解 Spring MVC 体系结构
- ◆ 对处理方法入参绑定及视图解析进行详细分析

17.1 Spring MVC 体系概述

Spring MVC 框架围绕 DispatcherServlet 这个核心展开，DispatcherServlet 是 Spring MVC 的总导演、总策划，它负责截获请求并将其分派给相应的处理器处理。Spring MVC 框架包括注解驱动控制器、请求及响应的信息处理、视图解析、本地化解析、上传文件解析、异常处理及表单标签绑定等内容。

17.1.1 体系结构

Spring MVC 是基于 Model 2 实现的技术框架，Model 2 是经典的 MVC (Model、View、Control) 模型在 Web 应用中的变体，这个改变主要源于 HTTP 协议的无状态性。Model 2 的目的和 MVC 一样，也是利用处理器分离模型、视图和控制，达到不同技术层级间松散层耦合的效果，提高系统灵活性、复用性和可维护性。在大多数情况下，可以将 Model 2 与 MVC 等同起来。

在利用 Model 2 之前，把所有的展现逻辑和业务逻辑集中在一起，有时也称这种应用模式为 Model 1。Model 1 的主要缺点就是紧耦合，复用性差，维护成本高。

由于 Spring MVC 是基于 Model 2 实现的框架，所以它底层的机制也是 MVC，通过图 17-1 描述 Spring MVC 的整体架构。

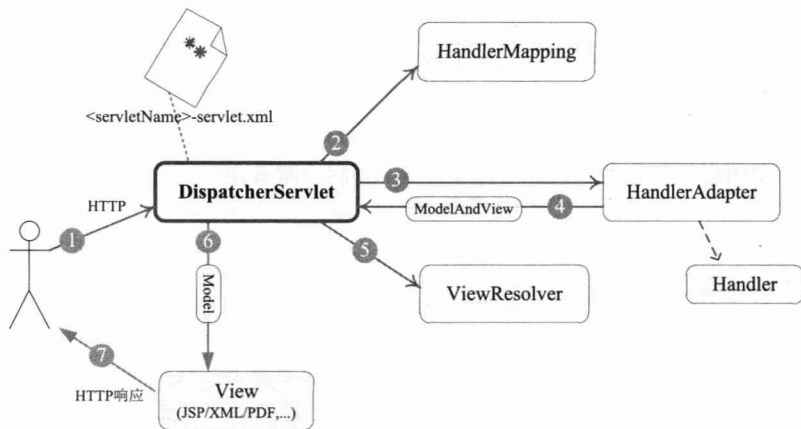


图 17-1 Spring MVC 框架模型

从接收请求到返回响应，Spring MVC 框架的众多组件通力配合、各司其职，有条不紊地完成分内的工作。在整个框架中，DispatcherServlet 处于核心的位置，它负责协调和组织不同组件以完成请求处理并返回响应的工作。和大多数 Web MVC 框架一样，Spring MVC 通过一个前端 Servlet 接收所有的请求，并将具体工作委托给其他组件进行

处理, DispatcherServlet 就是 Spring MVC 的前端 Servlet。下面对 Spring MVC 处理请求的整体过程进行讲解。

(1) 整个过程始于客户端发出一个 HTTP 请求, Web 应用服务器接收到这个请求。如果匹配 DispatcherServlet 的请求映射路径(在 web.xml 中指定), 则 Web 容器将该请求转交给 DispatcherServlet 处理。

(2) DispatcherServlet 接收到这个请求后, 将根据请求的信息(包括 URL、HTTP 方法、请求报文头、请求参数、Cookie 等)及 HandlerMapping 的配置找到处理请求的处理器(Handler)。可将 HandlerMapping 看作路由控制器, 将 Handler 看作目标主机。值得注意的是, 在 Spring MVC 中并没有定义一个 Handler 接口, 实际上, 任何一个 Object 都可以成为请求处理器。

(3) 当 DispatcherServlet 根据 HandlerMapping 得到对应当前请求的 Handler 后, 通过 HandlerAdapter 对 Handler 进行封装, 再以统一的适配器接口调用 Handler。HandlerAdapter 是 Spring MVC 的框架级接口, 顾名思义, HandlerAdapter 是一个适配器, 它用统一的接口对各种 Handler 方法进行调用。

(4) 处理器完成业务逻辑的处理后将返回一个 ModelAndView 给 DispatcherServlet, ModelAndView 包含了视图逻辑名和模型数据信息。

(5) ModelAndView 中包含的是“逻辑视图名”而非真正的视图对象, DispatcherServlet 借由 ViewResolver 完成逻辑视图名到真实视图对象的解析工作。

(6) 当得到真实的视图对象 View 后, DispatcherServlet 就使用这个 View 对象对 ModelAndView 中的模型数据进行视图渲染。

(7) 最终客户端得到的响应消息可能是一个普通的 HTML 页面, 也可能是一个 XML 或 JSON 串, 甚至是一张图片或一个 PDF 文档等不同的媒体形式。

以上每个步骤都包含丰富的知识点, 本章将逐步揭示每个组件的“庐山真面目”。不过现在请收好所有的好奇心, 我们第一步要做的是在 web.xml 中配置好 DispatcherServlet, 让 Spring MVC 的“心脏”跳动起来。

17.1.2 配置 DispatcherServlet

DispatcherServlet 是 Spring MVC 的“灵魂”和“心脏”, 它负责接收 HTTP 请求并协调 Spring MVC 的各个组件完成请求处理的工作。和任何 Servlet 一样, 用户必须在 web.xml 中配置好 DispatcherServlet。我们在第 2 章中已经配置了一个简单的 DispatcherServlet, 这里进一步分析其具体的配置。

要了解 Spring MVC 框架的工作机理, 必须回答以下 3 个问题。

(1) DispatcherServlet 框架如何截获特定的 HTTP 请求并交由 Spring MVC 框架处理?

(2) 位于 Web 层的 Spring 容器(WebApplicationContext)如何与位于业务层的 Spring

容器 (ApplicationContext) 建立关联, 以使 Web 层的 Bean 可以调用业务层的 Bean?

(3) 如何初始化 Spring MVC 的各个组件, 并将它们装配到 DispatcherServlet 中?

1. 配置 DispatcherServlet, 截获特定的 URL 请求

大家知道, 我们可以在 web.xml 中配置一个 Servlet, 并通过 <servlet-mapping> 指定其处理的 URL。这是传统的 DispatcherServlet 配置方式。而 Spring 4.0 已全面支持 Servlet 3.0, 因此也可以采用编程式的配置方式。这里先采用传统的 web.xml 的方式进行讲解, 然后介绍基于 Servlet 3.0 的新方式。假设我们希望 Spring MVC 的 DispatcherServlet 能截获并处理所有以 .html 结束的 URL 请求, 那么可以在 web.xml 中按如下方式进行配置, 如代码清单 17-1 所示。

代码清单 17-1 web.xml: 配置 DispatcherServlet

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>classpath:/applicationContext.xml</param-value>
</context-param>
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>

<servlet> ② ← 声明 DispatcherServlet
  <servlet-name>smart</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping> ③ ← 名为 DispatcherServlet 匹配的 URL 模式
  <servlet-name>smart</servlet-name>
  <url-pattern>*.html</url-pattern>
</servlet-mapping>
```

① 业务层和持久层的 Spring 配置文件, 这些配置文件被父 Spring 容器所使用

在①处, 通过 contextConfigLocation 参数指定业务层 Spring 容器的配置文件 (多个配置文件使用逗号分隔)。ContextLoaderListener 是一个 ServletContextListener, 它通过 contextConfigLocation 参数所指定的 Spring 配置文件启动“业务层”的 Spring 容器。

在②处配置了名为 smart 的 DispatcherServlet, 它默认自动加载 /WEB-INF/smart-servlet.xml (<servlet-Name>-servlet.xml) 的 Spring 配置文件, 启动 Web 层的 Spring 容器。

在③处, 通过 <servlet-mapping> 指定 DispatcherServlet 处理所有以 .html 为后缀的 HTTP 请求, 即所有带 .html 后缀的 HTTP 请求都会被 DispatcherServlet 截获并处理。

我们知道, 多个 Spring 容器之间可设置为父子级的关系, 以实现良好的解耦。在这里, “Web 层” Spring 容器将作为“业务层” Spring 容器的子容器, 即“Web 层”容器可以引用“业务层”容器的 Bean, 而“业务层”容器却访问不到“Web 层”容器的 Bean。

需要提醒的是，一个 web.xml 可以配置多个 DispatcherServlet，通过其 `<servlet-mapping>` 配置，让每个 DispatcherServlet 处理不同的请求。

DispatcherServlet 遵循“契约优于配置”的原则，在大多数情况下，用户无须进行额外的配置，只需按契约行事即可。

如果确实要对 DispatcherServlet 的默认规则进行调整，则 DispatcherServlet 是“敞开胸怀”的。下面是常用的一些配置参数，可通过 `<servlet>` 的 `<init-param>` 指定。

- namespace: DispatcherServlet 对应的命名空间，默认为 `<servlet-name>-servlet`，用于构造 Spring 配置文件的路径。在显式指定该属性后，配置文件对应的路径为 `WEB-INF/<namespace>.xml`，而非 `WEB-INF/<servlet-name>-servlet.xml`。如果将 namespace 设置为 sample，则对应的 Spring 配置文件为 `WEB-INF/sample.xml`。
- contextConfigLocation: 如果 DispatcherServlet 上下文对应的 Spring 配置文件有多个，则可以使用该属性按照 Spring 资源路径的方式指定。如“`classpath:sample1.xml, classpath:sample2.xml`”，DispatcherServlet 将使用类路径下的 `sample1.xml` 和 `sample2.xml` 这两个配置文件初始化 `WebApplicationContext`。
- publishContext: 布尔类型的属性，默认值为 `true`。DispatcherServlet 根据该属性决定是否将 `WebApplicationContext` 发布到 `ServletContext` 的属性列表中，以便调用者可借由 `ServletContext` 找到 `WebApplicationContext` 实例，对应的属性名为 `DispatcherServlet#getServletContextAttributeName()` 方法的返回值。
- publishEvents: 布尔类型的属性。当 DispatcherServlet 处理完一个请求后，是否需要向容器发布一个 `ServletRequestHandledEvent` 事件，默认值为 `true`。如果容器中没有任何事件监听器，则可以将该属性设置为 `false`，以便提高运行性能。

下面的代码显式指定 Web 层的 Spring 配置文件。

```
<servlet>
    <servlet-name>smart</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            /WEB-INF/spring/webApplicationContext.xml
        </param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

之前提到 Spring 4.0 已全面支持 Servlet 3.0，因此，在 Servlet 3.0 环境中，也可以使用编程的方式来配置 Servlet 容器。下面的代码可达到和代码清单 17-1 同样的效果。

```
public class SmartApplicationInitializer implements WebApplicationInitializer {
    @Override
    public void onStartup(ServletContext container) {
        ServletRegistration.Dynamic registration = container.addServlet("dispatcher", new
            DispatcherServlet());
    }
}
```

```

        registration.setLoadOnStartup(1);
        registration.addMapping("*.html*");
    }
}

```

接下来看看 Servlet 3.0 的实现原理。在 Servlet 3.0 环境中，容器会在类路径中查找实现 `javax.servlet.ServletContainerInitializer` 的类，如果发现已有实现类，就会调用它来配置 Servlet 容器。在 Spring 中，`org.springframework.web.SpringServletContainerInitializer` 类实现了该接口，同时这个类又会查找实现 `org.springframework.web.WebApplicationInitializer` 接口的类，并将配置任务交给这些实现类去完成。另外，Spring 提供了一个便利的抽象类 `AbstractAnnotationConfigDispatcherServletInitializer` 来实现这个接口，使得它在注册 `DispatcherServlet` 时只需简单地指定它的 Servlet 映射即可。在上述示例中，当应用部署到 Servlet 3.0 容器中时，容器启动时会自动发现它，并使用它来配置 Servlet 上下文。

2. 探究 DispatcherServlet 的内部逻辑

现在剩下的最后一个是：Spring 如何将上下文中的 Spring MVC 组件装配到 `DispatcherServlet` 中？通过查看 `DispatcherServlet` 的 `initStrategies()` 方法的代码，一切真相就大白于天下了。

```

protected void initStrategies(ApplicationContext context) {
    initMultipartResolver(); //①初始化上传文件解析器（直译为多部分请求解析器）
    initLocaleResolver(); //②初始化本地化解析器
    initThemeResolver(); //③初始化主题解析器
    initHandlerMappings(); //④初始化处理器映射器
    initHandlerAdapters(); //⑤初始化处理器适配器
    initHandlerExceptionResolvers(); //⑥初始化处理器异常解析器
    initRequestToViewNameTranslator(); //⑦初始化请求到视图名翻译器
    initViewResolvers(); //⑧初始化视图解析器
}

```

`initStrategies()` 方法将在 `WebApplicationContext` 初始化后自动执行，此时 Spring 上下文中的 Bean 已经初始化完毕。该方法的工作原理是：通过反射机制查找并装配 Spring 容器中用户显式自定义的组件 Bean，如果找不到，则装配默认的组件实例。

Spring MVC 定义了一套默认的组件实现类，也就是说，即使在 Spring 容器中没有显式定义组件 Bean，`DispatcherServlet` 也会装配好一套可用的默认组件。在 `spring-webmvc-4.x.jar` 包的 `org.springframework.web.servlet` 类路径下拥有一个 `DispatcherServlet.properties` 配置文件，该文件指定了 `DispatcherServlet` 所使用的默认组件。

```

##本地化解析器
org.springframework.web.servlet.LocaleResolver=
☞org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver

##主题解析器
org.springframework.web.servlet.ThemeResolver=
☞org.springframework.web.servlet.theme.FixedThemeResolver

##处理器映射（共2个）
org.springframework.web.servlet.HandlerMapping=

```



```

org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping,\
org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping

##处理器适配器 (共3个)
org.springframework.web.servlet.HandlerAdapter=
org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter,\
org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,\
org.springframework.web.servlet.mvc.method.annotation
    .RequestMappingHandlerAdapter

##异常处理器 (共3个)
org.springframework.web.servlet.HandlerExceptionResolver=
org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerException
Resolver,\
org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionHandler,\
org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver

##视图名称翻译器
org.springframework.web.servlet.RequestToViewNameTranslator=
org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator

##视图解析器
org.springframework.web.servlet.ViewResolver=org.springframework.web.servlet.view.
InternalResourceViewResolver

```

如果用户希望采用非默认类型的组件，则只需在 Spring 配置文件中配置自定义的组件 Bean 即可。Spring MVC 一旦发现上下文中有用户自定义的组件，就不会使用默认的组件。下面通过表 17-1 进一步了解 DispatcherServlet 装配每种组件的过程。

表 17-1 DispatcherServlet 装配各型组件的逻辑

组件类型	发现机制
文件上传 解析器 (☆)	(1) 查找名为 multipartResolver、类型为 MultipartResolver 的 Bean 作为该类型的组件。 (2) 没有默认的实现类。 如果用户没有在上文中显式定义这一类型的组件，则 DispatcherServlet 中将不会拥有该类型的组件
本地化 解析器 (☆)	(1) 查找名为 localeResolver、类型为 LocaleResolver 的 Bean 作为该类型的组件。 (2) 如果 (1) 找不到，则使用默认的实现类 (AcceptHeaderLocaleResolver) 创建该类型的组件
主题 解析器 (☆)	(1) 查找名为 themeResolver、类型为 LocaleResolver 的 Bean 作为该类型的组件。 (2) 如果 (1) 找不到，则使用默认的实现类 (FixedThemeResolver) 创建该类型的组件
处理器 映射器 (★)	(1) 如果 detectAllHandlerMappings 属性为 true (默认为 true)，则根据类型匹配机制查找上下文及父 Spring 容器中所有类型为 HandlerMapping 的 Bean，将它们作为该类型的组件。 (2) 如果 detectAllHandlerMappings 属性为 false，则查找名为 handlerMapping、类型为 HandlerMapping 的 Bean 作为该类型的组件。 (3) 如果通过以上两种方式都找不到，则使用 BeanNameUrlHandlerMapping 实现类创建该类型的组件
处理器 适配器 (★)	(1) 如果 detectAllHandlerAdapters 属性为 true (默认为 true)，则根据类型匹配机制查找上下文及父 Spring 容器中所有类型为 HandlerAdapter 的 Bean，将它们作为该类型的组件。 (2) 如果 detectAllHandlerAdapters 属性为 false，则查找名为 handlerAdapter、类型为 HandlerAdapter 的 Bean 作为该类型的组件。 (3) 如果通过以上两种方式都找不到，则使用 DispatcherServlet.properties 配置文件中指定的 3 个实现类分别创建一个适配器，并添加到适配器列表中

续表

组件类型	发现机制
处理器异常 解析器(★)	(1) 如果 detectAllHandlerExceptionResolvers 属性为 true (默认为 true), 则根据类型匹配 (HandlerExceptionResolver) 机制查找上下文及父 Spring 容器中所有匹配的 Bean 作为该类型的组件。 (2) 如果 detectAllHandlerExceptionResolvers 属性为 false, 则查找名为 handlerExceptionResolver、类型为 HandlerExceptionResolver 的 Bean 作为该类型的组件。 (3) 如果通过以上两种方式都找不到, 则查找 DispatcherServlet.properties 中定义的默认实现类, 不过该文件中没有对应处理器异常解析器的默认实现类 (用户可以更改属性文件)
视图名 翻译器(☆)	(1) 查找名为 viewNameTranslator、类型为 RequestToViewNameTranslator 的 Bean 作为该类型的组件。 (2) 如果 (1) 找不到, 则使用默认的实现类 (DefaultRequestToViewNameTranslator) 创建该类型的组件
视图 解析器(★)	(1) 如果 detectAllViewResolvers 属性为 true (默认为 true), 则根据类型匹配 (ViewResolver) 机制查找上下文及父 Spring 容器中所有匹配的 Bean 作为该类型的组件。 (2) 如果 detectAllViewResolvers 属性为 false, 则查找名为 viewResolver、类型为 ViewResolver 的 Bean 作为该类型的组件。 (3) 如果通过以上两种方式都找不到, 则通过 DispatcherServlet.properties 中定义的默认实现类 (InternalResourceViewResolver) 创建该类型的组件

有些组件最多允许存在一个实例, 如 MultipartResolver、LocaleResolver 等, 在表 17-1 中使用☆进行标注; 而另一些组件允许存在多个实例, 如 HandlerMapping、HandlerAdapter 等, 在表 17-1 中使用★进行标注。同一类型的组件如果存在多个, 那么它们之间的优先级顺序如何确定呢? 这些组件都实现了 org.springframework.core.Ordered 接口, 可通过 order 属性确定优先级顺序, 值越小优先级越高。

简言之, 当 DispatcherServlet 初始化后, 就会自动扫描上下文的 Bean, 根据名称或类型匹配的机制查找自定义的组件, 找不到时则使用 DispatcherServlet.properties 定义的默认组件。

17.1.3 一个简单的实例

在学习了 Spring MVC 框架的整体结构后, 下面通过一个简单的实例讲解 Spring MVC 开发的基本过程。Spring MVC 应用开发一般包括以下几个步骤。

- (1) 配置 web.xml, 指定业务层对应的 Spring 配置文件, 定义 DispatcherServlet。
- (2) 编写处理请求的控制器 (处理器)。
- (3) 编写视图对象, 这里使用 JSP 作为视图。
- (4) 配置 Spring MVC 的配置文件, 使控制器、视图解析器等生效。

17.1.2 节已经详细讲解了如何在 web.xml 中配置 Spring 业务层容器及定义 DispatcherServlet 的知识, 所以这里直接从第 (2) 步开始。

1. 编写处理请求的控制器

Spring MVC 通过@Controller 注解即可将一个 POJO 转化为处理请求的控制器, 通

过 `@RequestMapping` 为控制器指定处理哪些 URL 的请求。UserController 是一个负责用户处理的控制器，其代码如代码清单 17-2 所示。

代码清单 17-2 UserController.java

```
package com.smart.web;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller ① ← 使UserController成为处理请求的控制器
@RequestMapping("/user") ② ← 处理来自/user URI的请求
public class UserController {

    @RequestMapping("/register") ③
    public String register(){
        return "user/register";
    }
}
```

③ ← 返回一个String类型的逻辑视图名

首先使用 `@Controller` 对 UserController 类进行标注，使其成为一个可处理 HTTP 请求的控制器。然后使用 `@RequestMapping` 对 UserController 及其 register() 方法进行标注，确定 register() 对应的请求 URL。

在 UserController 类定义处标注的 `@RequestMapping` 限定了 UserController 类处理所有 URI 为 /user 的请求，它相对于 Web 容器部署根路径。UserController 类可以定义多个处理方法，处理来自 /user URI 的请求。假设 Web 容器的部署根路径为 /chapter17，则代码清单 17-2 中的 register() 方法将处理所有来自 /chapter17/user/ register.html 的请求。值得说明的是，② 处类级的 `@RequestMapping` 不是必需的，可以直接在方法中标注 `@RequestMapping`，此时，方法处 `@RequestMapping` 指定的 URI 则是相对于部署根路径的。

register() 方法返回一个字符串 user/register，它代表一个逻辑视图名，将由视图解析器解析为一个具体的视图对象。在本例中，它将被映射为 /WEB-INF/views/user/register.jsp。稍后，读者将了解如何装配完成这一任务的视图解析器。

2. 编写视图对象

我们使用一个 register.jsp 作为用户的注册页面，UserController#register() 方法处理完成后，将转向这个 register.jsp 页面，如代码清单 17-3 所示。

代码清单 17-3 register.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<title>新增用户</title>
</head>
<body>
    <form method="post" action="<c:url value="/user.html"/>">①
        <table>
            <tr>
                <td>用户名: </td>
```

将表单提交到/user.html控制器中

```

        <td><input type="text" name="userName"/></td>
    </tr>
    <tr>
        <td>密码: </td>
        <td><input type="password" name="password"/></td>
    </tr>
    <tr>
        <td>姓名: </td>
        <td><input type="text" name="realName"/></td>
    </tr>
    <tr>
        <td colspan="2"><input type="submit" name="提交"/></td>
    </tr>
</table>
</form>
</body>
</html>

```

register.jsp 很简单,它包括了一个表单,单击“提交”按钮后,表单提交到/user.html 进行处理。UserController 添加了一个 createUser()方法用于处理表单提交的请求,如代码清单 17-4 所示。

代码清单 17-4 UserController.java

```

package com.smart.web;
...
import org.springframework.web.servlet.ModelAndView;

@Controller
@RequestMapping("/user")
public class UserController {

    @Autowired
    private UserService userService;① ← 注入业务层的Bean

    @RequestMapping(method=RequestMethod.POST)② ← 处理/user的请求,不过请求的方法必须为POST
    public ModelAndView createUser(User user) {③ ← 将表单值映射到User对象中,调用UserService保存user,返回ModelAndView
        userService.createUser(user);
        ModelAndView mav = new ModelAndView();
        mav.setViewName("user/createSuccess");
        mav.addObject("user", user);
        return mav;
    }
    ...
}

```

createUser()方法处的@RequestMapping 注解让 createUser()处理 URI 为/user.html 且请求方法为 POST 的请求。Spring MVC 自动将表单中的数据按参数名和 User 属性名匹配的方式进行绑定,将参数值填充到 User 的相应属性中。调用业务层的 UserService 进行业务处理,进而返回 ModelAndView 对象,逻辑视图名为 user/createSuccess, user 作为模型数据暴露给视图对象。

User 对象的代码如代码清单 17-5 所示。

代码清单 17-5 User.java

```
package com.smart.domain;

public class User {
    private String userName;
    private String password;
    private String realName;
    ...
}
```

注意到 User 中的参数名和代码清单 17-3 中的表单组件是相同的, 这样 Spring MVC 即可将表单组件值填充到 User 的相应属性中。

视图解析器将 user/createSuccess 解析为 /WEB-INF/views/user/createSuccess.jsp 的视图对象, createSuccess.jsp 可以访问到模型中的数据。createSuccess.jsp 页面代码如代码清单 17-6 所示。

代码清单 17-6 createSuccess.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<html>
    <head>
        <title>用户创建成功</title>
    </head>
    <body>
        恭喜, 用户${user.userName}创建成功。①
    </body>
</html>
```

访问Model中的属性

由于 UserController#createUser()通过@ModelAttribute("user")将 User 对象放到模型中, 所以 createSuccess.jsp 可以通过\${user.userName}访问模型中的数据。

3. 配置 Spring MVC 的配置文件

要使以上实例正常工作, 需要在 Spring MVC 配置文件中进行简单的配置, 如代码清单 17-7 所示。

代码清单 17-7 smart-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-4.0.xsd">
    <context:component-scan base-package="com.smart.web"/> ①
    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver" ②
        p:prefix="/WEB-INF/views/"
        p:suffix=".jsp"/>
</beans>
```

在①处通过<context:component-scan>使 Spring 扫描 com.smart.web 包下所有的类,

让标注 Spring 注解的类生效。而在②处定义了一个视图名称解析器，将视图逻辑名解析为 /WEB-INF/views/<viewName>.jsp 的视图对象，如图 17-2 所示。



图 17-2 注册模块页面

4. 运行用户注册模块

将用户模块部署到 Web 服务器中，Web 服务器服务端口为 8080，部署根路径为 /chapter17，通过 <http://localhost:8080/chapter17/user/register.html> 即可访问用户注册页面。填写用户注册信息，单击“提交”按钮将表单提交到 <http://localhost:8080/chapter17/user.html> 地址中，UserController 控制器的 createUser() 方法响应这个请求，并导向 createSuccess.jsp 页面中。

对于用户注册表单提交直到返回响应这一过程，用实物对象对组件接口进行替换，得到如图 17-3 所示的交互图。

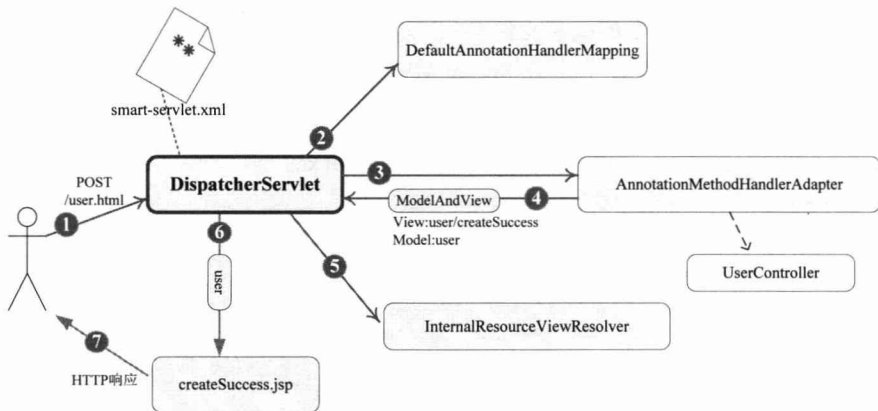


图 17-3 处理/user.html 表单提交的整体过程

再简要描述一下 Spring MVC 处理/user.html 的整个过程。

- ① DispatcherServlet 接收到客户端的/user.html 请求。
- ② DispatcherServlet 使用 DefaultAnnotationHandlerMapping 查找负责处理该请求的处理器。

- ③ DispatcherServlet 将请求分发给名为/user.html 的 UserController 处理器。
- ④ 处理器完成业务处理后, 返回 ModelAndView 对象, 其中 View 的逻辑名为/user/createSuccess, 而模型包含一个键为 user 的 User 对象。
- ⑤ DispatcherServlet 调用 InternalResourceViewResolver 组件对 ModelAndView 中的逻辑视图名进行解析, 得到真实的/WEB-INF/view/user/createSuccess.jsp 视图对象。
- ⑥ DispatcherServlet 使用/WEB-INF/view/user/createSuccess.jsp 对模型中的 user 模型对象进行渲染。
- ⑦ 返回响应页面给客户端。

通过这个例子, 我们了解了开发一个 Spring MVC 应用所需经历的大体过程。这个例子太过简单, 每个步骤都是最简的实现。在本章的后续章节中, 我们将对以上各个步骤进行深入分析。

17.2 注解驱动的控制器的

17.2.1 使用@RequestMapping 映射请求

在 POJO 类定义处标注@Controller, 再通过<context:component-scan/>扫描相应的类包, 即可使 POJO 成为一个能处理 HTTP 请求的控制器。

用户可以创建数量 unlimited 的控制器, 分别处理不同的业务请求, 如 LogonController、UserController、ForumController 等。每个控制器可拥有多个处理请求的方法, 每个方法负责不同的请求操作。如何将请求映射到对应的控制器方法中是 Spring MVC 框架的重要任务之一, 这项任务由@RequestMapping 承担。

在控制器的类定义及方法定义处都可以标注@RequestMapping, 类定义处的@RequestMapping 提供初步的请求映射信息, 方法定义处的@RequestMapping 提供进一步的细分映射信息。DispatcherServlet 截获请求后, 就通过控制器上@RequestMapping 提供的映射信息确定请求所对应的处理方法。

将请求映射到控制器处理方法的工作包含一系列映射规则, 这些规则是根据请求中的各种信息制定的, 具体包括请求 URL、请求参数、请求方法、请求头这 4 个方面的信息项。

1. 通过请求 URL 进行映射

@RequestMapping 使用 value 值指定请求的 URL, 如@RequestMapping("/user")、@RequestMapping("/register")等。需要注意的是, @RequestMapping 在类定义处指定的 URL 相对于 Web 应用的部署路径, 而在方法定义处指定的 URL 则相对于类定义处指定的 URL。如果在类定义处未标注@RequestMapping, 则仅在处理方法处标注@RequestMapping, 此时, 方法处指定的 URL 则相对于 Web 应用的部署路径, 如代码清单 17-8 所示。

代码清单 17-8 UserController.java: 在类定义处不使用@RequestMapping

```
package com.smart.web;
import org.springframework.web.bind.annotation.RequestMapping;
...
@Controller
public class UserController {

    @RequestMapping(path="/user/createUser")
    public String createUser(@ModelAttribute("user") User user){
        ...
        return "user/createSuccess";
    }

    @RequestMapping("/user/register")
    public String register(@ModelAttribute("user") User user){
        return "user/register";
    }
}
```

这样, /user/register.html 请求将由 register()方法处理, 而/user/createUser.html 请求将由 createUser()方法处理。注意, 它们都相对于 Web 应用的部署路径。

同一控制器的多个处理方法负责处理相同业务模块的不同操作, 但凡设计合理的 Web 应用都会将这些操作请求安排在某一相同的 URL 之下。所以除非特别的原因, 建议不要舍弃类定义处的@RequestMapping。

@RequestMapping 不但支持标准的 URL, 还支持 Ant 风格(?, *和**字符, 参见 4.3.2 节)的和带{xxx}占位符的 URL。以下 URL 都是合法的。

- ❑ /user/*/createUser: 匹配/user/aaa/createUser、/user/bbb/createUser 等 URL。
- ❑ /user/**/createUser: 匹配/user/createUser、/user/aaa/bbb/createUser 等 URL。
- ❑ /user/createUser?: 匹配/user/createUseraa、/user/createUserbb 等 URL。
- ❑ /user/{userId}: 匹配 user/123、user/456 等 URL。
- ❑ /user/**/{userId}: 匹配 user/aaa/bbb/123、user/aaa/456 等 URL。
- ❑ company/{companyId}/user/{userId}/detail: 匹配 company/123/user/456/detail 等 URL。

通过@PathVariable 可以将 URL 中的占位符参数绑定到控制器处理方法的入参中, 如代码清单 17-9 所示。

代码清单 17-9 UserController.java: 使用@PathVariable

```
package com.smart.web;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.servlet.ModelAndView;
...
@Controller
@RequestMapping("/user")
public class UserController {

    @RequestMapping("/{userId}")
    public ModelAndView showDetail(@PathVariable("userId") String userId){
        ModelAndView mav = new ModelAndView();
    }
}
```

```

        mav.setViewName("user/showDetail");
        mav.addObject("user", userService.getUserById(userId));
        return mav;
    }
}

```

URL 中的 {xxx} 占位符可以通过 `@PathVariable("xxx")` 绑定到操作方法的入参中。类定位处 `@RequestMapping` 的 URL 如果使用占位符的参数，则也可以绑定到处理方法的入参中，如代码清单 17-10 所示。

代码清单 17-10 UserController.java: 使用 `@PathVariable`

```

@Controller
@RequestMapping("/owners/{ownerId}")
public class RelativePathUriTemplateController {

    @RequestMapping("/pets/{petId}")
    public void findPet(@PathVariable String ownerId, @PathVariable String petId, Model
model) {
        ...
    }
}

```

在默认情况下，Java 类的反射对象并未包含方法入参的名称，在 Java 8.0 中可以通过 `javac -parameters` 生成方法入参的元数据信息，在低版本的 Java 中则可以通过 `javac -g` 打开生成所有调试信息的开发，这样也会包含方法入参名的元数据信息。所以要使代码清单 17-10 中的 `findPet()` 入参成功绑定 URL 中的占位符参数，必须保证在编译时输出方法名元信息。在 Maven 中可以显式配置 `maven-compiler-plugin` 编译插件，开启编译输出调试信息的开关。

```

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
    <source>${java.version}</source>
    <target>${java.version}</target>
    <encoding>UTF-8</encoding>
    <debug>true</debug>
  </configuration>
</plugin>

```

不过编译时打开 `debug` 开关会使目标类变大，对运行效率也有一定的负面影响。正式编译部署时往往将此开关取消，所以最好在 `@PathVariable` 中显式指定绑定的参数名，以避免因编译方式不同造成参数绑定失败的隐患。

2. 通过请求参数、请求方法或请求头进行映射

HTTP 请求报文除 URL 外，还拥有其他众多的信息。以下是一个标准的 HTTP 请求报文，如图 17-4 所示。



```

①请求方法 ②请求URL ③HTTP协议及版本
POST /chapter17/user.html HTTP/1.1
④报
文
头
Accept: image/jpeg, application/x-ms-application, ..., */*
Referer: http://localhost:8088/chapter17/user/register.html?
code=100&time=123123
Accept-Language: zh-CN
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1;
Content-Type: application/x-www-form-urlencoded
Host: localhost:8088
Content-Length: 112
⑤报
文
体
Connection: Keep-Alive
Cache-Control: no-cache
Cookie: JSESSIONID=24DF2688E37EE4F66D9669D2542AC17B
name=tom&password=1234&realName=tomson

```

图 17-4 HTTP 请求报文

①是请求方法，GET 和 POST 是最常见的 HTTP 方法，除此以外还包括 DELETE、HEAD、OPTIONS、PUT、TRACE。不过，当前的大多数浏览器只支持 GET 和 POST，Spring 提供了一个 HiddenHttpMethodFilter，允许通过_method 表单参数指定这些特殊的 HTTP 方法（实际上还是通过 POST 提交表单）。服务器端配置了 HiddenHttpMethodFilter 后，Spring 会根据_method 参数指定的值模拟出相应的 HTTP 方法，这样就可以使用这些 HTTP 方法对处理方法进行映射了。

②是请求对应的 URL 地址，它和报文头的 Host 属性组成完整的请求 URL。

③是协议名称及版本号。

④是 HTTP 的报文头，报文头包含若干个属性，格式为“属性名:属性值”，服务器端据此获取客户端的信息。

⑤是报文体，它将一个页面表单中的组件值通过 param1=value1¶m2=value2 的键值对形式编码成一个格式化串，它承载多个请求参数的数据。不但报文体可以传递请求参数，请求 URL 也可以通过类似于/chapter17/user.html?param1=value1¶m2=value2 的方式传递请求参数。



实战经验

HttpWatch 是强大的网页数据分析工具，安装后将集成到 Internet Explorer 工具栏中。它无须代理服务器或一些复杂的网络监控工具，就能抓取请求及响应的完整信息，包括 Cookies、消息头、查询参数、响应报文等，是 Web 应用开发人员的必备工具。笔者有篇网文对 HTTP 有较全面的介绍，地址为 <http://www.iteye.com/topic/1124408>，欢迎阅读。

@RequestMapping 除了可以使用请求 URL 映射请求外，还可以使用请求方法、请求头参数及请求参数（报文体和 URL 包含的请求参数）映射请求，如代码清单 17-11 所示。

代码清单 17-11 UserController.java: 使用其他信息映射请求

```

@Controller
@RequestMapping("/user")
public class UserController {

    @RequestMapping(path="/delete",method=RequestMethod.POST,params="userId") ①
    public String test1(@RequestParam("userId") String userId){
        //do sth
        return "user/test1";
    }

    @RequestMapping(path="/show",headers="content-type=text/*") ②
    public String test2(@RequestParam("userId") String userId){
        //do sth
        return "user/test2";
    }
}

```

使用请求方法及请求参数映射请求

使用报文头映射请求

@RequestMapping 的 value、method、params 及 headers 分别表示请求 URL、请求方法、请求参数及报文头的映射条件，它们之间是与的关系，联合使用多个条件项可让请求映射更加精确化。

params 和 headers 分别通过请求参数及报文头属性进行映射，它们支持简单的映射表达式。下面以 params 表达式为例进行说明，headers 可以参照 params 来理解。

- "param1": 表示请求须包含名为 param1 的请求参数。
- "!param1": 表示请求不能包含名为 param1 的请求参数。
- "param1!=value1": 表示请求包含名为 param1 的请求参数，但其值不能为 value1。
- {"param1=value1","param2"}: 表示请求必须包含名为 param1 和 param2 的两个请求参数，且 param1 参数的值必须为 value1。

17.2.2 请求处理方法签名

@RequestMapping 像一个迎宾，将 HTTP 请求正确地迎接到主人的面前（负责处理该请求的方法）；主人起身相迎，进而与 HTTP 请求交杯换盏、把酒言欢；最后在欢声笑语中送走 HTTP 请求（返回响应）。Spring MVC 通过分析处理方法的签名，将 HTTP 请求信息绑定到处理方法的相应入参中，然后再调用处理方法得到返回值，最后对返回值进行处理并返回响应。

Spring MVC 对控制器处理方法签名的限制是很宽松的，用户几乎可以按自己喜欢的方式进行方法签名。在必要时对方法及方法入参标注相应的注解（如 @PathVariable、@RequestParam、@RequestHeader 等）即可，Spring MVC 会优雅地完成剩下的工作：将 HTTP 请求的信息绑定到相应的方法入参中，并根据方法返回值类型做出相应的后续处理。

一般情况下，处理方法的返回值类型为 ModelAndView 或 String，前者包含模型和

逻辑视图名，而后者仅代表一个逻辑视图名。下面来看几个典型的方法签名，如代码清单 17-12 所示。

代码清单 17-12 几种典型的处理方法签名

```
//①请求参数按名称匹配的方式绑定到方法入参中，方法返回的字符串代表逻辑视图名
@RequestMapping(path="/handle1")
public String handle1(@RequestParam("userName") String userName,
    @RequestParam("password") String password,
    @RequestParam("realName") String realName){
    return "success";
}

//②将Cookie值及报文头属性绑定到入参中，方法返回ModelAndView
@RequestMapping(path="/handle2")
public ModelAndView handle2(@CookieValue("JSESSIONID") String sessionId,
    @RequestHeader("Accept-Language") String accpetLanguage){
    ModelAndView mav = new ModelAndView();
    mav.setViewName("success");
    mav.addObject("user", new User());
    return mav;
}

//③请求参数按名称匹配的方式绑定到user的属性中，方法返回的字符串代表逻辑视图名
@RequestMapping(path="/handle3")
public String handle3(User user){
    return "success";
}

//④直接将HTTP请求对象传递给处理方法，方法返回的字符串代表逻辑视图名
@RequestMapping(path="/handle4")
public String handle4(HttpServletRequest request){
    return "success";
}
```

从代码清单 17-12 中可以发现，Spring MVC 会结合方法入参类型、入参的注解、返回值的类型，按照“契约”进行相应的处理。

17.2.3 使用矩阵变量绑定参数

RFC3986 定义了包含 name-value 的规范。随之在 Spring MVC 3.2 中出现了 `@MatrixVariable` 注解，该注解的出现使得开发人员能够将请求中的矩阵变量（MatrixVariable）绑定到处理器的方法参数中。而 Spring 4.0 更全面地支持这个规范，这也是 Spring 4.0 众多吸引人的新特性之一。接下来我们就一起来了解这个新特性的使用方式。

在 Matrix Variable 中，多个变量可以使用“;”（分号）分隔，例如：

```
/books;author=Tom;year=2016
```

如果一个变量对应多个值，那么可以使用“,”（逗号）分隔，例如：

```
author =smart1, smart2, smart3
```

或者使用重复的变量名，例如：

```
author = smart1; author = smart2; author = smart3
```

下面举一个例子来说明，代码如下：

```
// GET /books/22;a=11;b=22
@RequestMapping(value = "/books/{bookId}", method = RequestMethod.GET)
public void findBookId(@PathVariableString bookId, @MatrixVariable int a) {
    ...
}
```

相应的 bookId 和 a 都会被映射到这个方法中，如果匹配不到，则会报“bad request”。

如果 URI 只是“/books /11”，则也可以映射到这个方法中，但需要指定空值不报错：

@MatrixVariable (required=false)。

再来看一个更复杂的例子，以深入理解，代码如下：

```
//GET /books/42;a=11/authors/21;q=22
@RequestMapping(value = "/ books/{bookId}/ authors/{ authorId}",
    method = RequestMethod.GET)
public void findBook(
    @MatrixVariable(value="a", pathVar=" bookId ") int q1,
    @MatrixVariable(value="a", pathVar=" authorId ") int q2) {
    // q1 == 11
    // q2 == 22
}
```

针对每个 Path Variable 绑定一个 Matrix Variable，然后使用 value 和 pathVar 属性就能找到该值。

另外，Matrix Variable 也自带了一些属性可供选择，例如，是否必需，默认值。举一个例子来说明，代码如下：

```
// GET /books/42
@RequestMapping(value = "/books/{bookId}", method = RequestMethod.GET)
public void findBook(@MatrixVariable(required=true, defaultValue="1") int q) {
    // q == 1
}
```

默认 Matrix Variable 功能是开启的，如果不希望开启该功能，则需要手工将 Request MappingHandlerMapping 中的 removeSemicolonContent 属性设置为 true，即<mvc:annotation-driven enable-matrix-variables="true"/>。

17.2.4 请求处理方法签名详细说明

本节将学习如何对处理方法进行签名，包括如何设置方法入参以绑定请求信息、如何定义返回值类型、Spring MVC 对不同签名的处理方法如何进行调用等内容。

1. 使用@RequestParam 绑定请求参数值

在 17.2.1 节中我们指出，Java 类反射对象默认不记录方法入参的名称，因此需要在

方法入参处使用 `@RequestParam` 注解指定其对应的请求参数。`@RequestParam` 注解有以下 3 个参数。

- `value`: 参数名。
- `required`: 是否必需, 默认为 `true`, 表示请求中必须包含对应的参数名, 如果不存在则抛出异常。
- `defaultValue`: 默认参数名, 在设置该参数时, 自动将 `required` 设为 `false`。极少情况需要使用该参数, 也不推荐使用该参数。

下面的实例将 `userName` 和 `age` 请求参数分别绑定到 `handle11()` 方法的 `userName` 和 `age` 中, 并自动完成类型转换。如果不存在 `age` 请求参数, 则将抛出异常。

```
@RequestMapping(path = "/handle11")
public String handle11(
    @RequestParam(value = "userName", required = false) String userName,
    @RequestParam("age") int age) {
    ...
}
```

2. 使用 `@CookieValue` 绑定请求中的 Cookie 值

使用 `@CookieValue` 可让处理方法入参绑定某个 Cookie 的值, 它和 `@RequestParam` 拥有 3 个一样的参数。来看一个使用 `@CookieValue` 的实例, 代码如下:

```
@RequestMapping(path = "/handle12")
public String handle12(
    @CookieValue(value="sessionId", required=false) String sessionId,
    @RequestParam("age") int age) {
    ...
}
```

`@CookieValue` 的 `value` 属性指定了 Cookie 的名称, `required` 为 `false`, 表示请求中没有相应的 Cookie 时也不会报错。

3. 使用 `@RequestHeader` 绑定请求报文头的属性值

如图 17-4 所示, 请求报文包含了若干个报文头属性, 服务器可据此获知客户端的信息, 通过 `@RequestHeader` 即可将报文头属性值绑定到处理方法的入参中。

```
@RequestMapping(path = "/handle13")
public String handle13(@RequestHeader("Accept-Encoding") String encoding,
    @RequestHeader("Keep-Alive") long keepAlive) {
    ...
}
```

`@RequestHeader` 和 `@RequestParam` 拥有 3 个一样的参数, 此处不再赘述。

4. 使用命令/表单对象绑定请求参数值

所谓命令/表单对象并不需要实现任何接口, 仅是一个拥有若干属性的 POJO。在代码清单 17-12 中, ③处的 `User` 就是一个命令/表单对象。Spring MVC 会按请求参数名和命令/表单对象属性名匹配的方式, 自动为该对象填充属性值。支持级联的属性名, 如 `dept.deptId`、`dept.address.tel` 等。


```
@RequestMapping(path = "/handle14")
public String handle14(User user) {
    ...
}
```

假设 User 类的结构如图 17-5 所示, 则如下的 URL 请求, 其请求参数将正确地填充到 User 对象中。

```
/handle14.html?userName=tom&dept.deptId=1&dept.address.tel=102
```

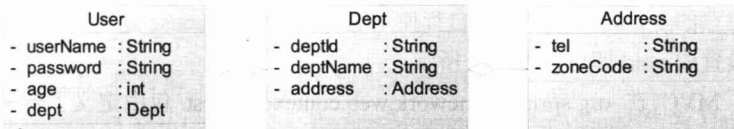


图 17-5 HTTP 请求报文

5. 使用 Servlet API 对象作为入参

在 Spring MVC 中, 控制器类可以不依赖任何 Servlet API 对象, 但是 Spring MVC 并不能阻止我们使用 Servlet API 的类作为处理方法的入参。以下处理方法都可以正确地工作, 如代码清单 17-13 所示。

代码清单 17-13 使用 Servlet API 作为入参

```
package com.smart.web;

import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.util.WebUtils;
...
@Controller
@RequestMapping("/user")
public class UserController {

    @RequestMapping(path = "/handle21")
    public void handle21(HttpServletRequest request, HttpServletResponse response) {①
        String userName = WebUtils.findParameterValue(request, "userName");
        response.addCookie(new Cookie("userName", userName));
    }

    @RequestMapping(path = "/handle22")
    public ModelAndView handle22(HttpServletRequest request) {②
        String userName = WebUtils.findParameterValue(request, "userName");
        ModelAndView mav = new ModelAndView();
        mav.setViewName("success");
        mav.addObject("userName", userName);
        return mav;
    }

    @RequestMapping(path = "/handle23")
    public String handle23(HttpSession session) {③
        session.setAttribute("sessionId", 1234);
        return "success";
    }

    @RequestMapping(path = "/handle24")
    public String handle24(HttpServletRequest request, @RequestParam("userName") String userName) {④
```

同时使用 `HttpServletRequest/HttpServletResponse` 作为入参

仅使用 `HttpServletRequest` 作为入参

使用 `HttpSession` 作为入参

既使用 `HttpServletRequest`, 又使用基本类型的入参

```

...
    return "success";
}
}

```

在使用 Servlet API 的类作为入参时，Spring MVC 会自动将 Web 层对应的 Servlet 对象传递给处理方法的入参。处理方法入参可以同时使用 Servlet API 类的入参和其他符合要求的入参，它们之间的位置顺序没有特殊要求。

值得注意的是，如果处理方法自行使用 `HttpServletResponse` 返回响应，则处理方法的返回值设置成 `void` 即可，如①处所示。

Spring MVC 在 `org.springframework.web.context.request` 包中定义了若干个可代理 Servlet 原生 API 类的接口，如 `WebRequest` 和 `NativeWebRequest`，它们也允许作为处理类的入参，通过这些代理类可访问请求对象的任何信息，如下：

```

@RequestMapping(path = "/handle25")
public String handle25(WebRequest request) {
    String userName = request.getParameter("userName");
    return "success";
}

```

6. 使用 I/O 对象作为入参

Servlet 的 `ServletRequest` 拥有 `getInputStream()` 和 `getReader()` 方法，可以通过它们读取请求的信息。相应的，Servlet 的 `ServletResponse` 拥有 `getOutputStream()` 和 `getWriter()` 方法，可以通过它们输出响应信息。

Spring MVC 允许控制器的处理方法使用 `java.io.InputStream/java.io.Reader` 及 `java.io.OutputStream/java.io.Writer` 作为方法的入参，Spring MVC 将获取 `ServletRequest` 的 `InputStream/Reader` 或 `ServletResponse` 的 `OutputStream/Writer`，然后传递给控制器的处理方法，如代码清单 17-14 所示。

代码清单 17-14 使用 `OutputStream` 输出一张图片

```

package com.smart.web;

import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;
import org.springframework.util.FileCopyUtils;

@Controller
@RequestMapping("/user")
public class UserController {

    @RequestMapping(path = "/handle31")
    public void handle31(OutputStream os) throws IOException{
        Resource res = new ClassPathResource("/image.jpg");//读取类路径下的图片文件
        FileCopyUtils.copy(res.getInputStream(), os);        //将图片写到输出流中
    }
}

```

7. 其他类型的参数

控制器处理方法的入参除支持以上类型的参数外，还支持 `java.util.Locale`、`java.`

security.Principal, 可以通过 Servlet 的 HttpServletRequest 的 getLocale() 和 getUserPrincipal() 方法得到相应的值。如果处理方法的入参类型为 Locale 或 Principal, 则 Spring MVC 自动从请求对象中获取相应的对象并传递给处理方法的入参。

17.2.5 使用 HttpMessageConverter<T>

HttpMessageConverter<T> 是 Spring 的一个重要接口, 它负责将请求信息转换为一个对象 (类型为 T), 将对象 (类型为 T) 输出为响应信息。

DispatcherServlet 默认已经安装了 RequestMappingHandlerAdapter 作为 HandlerAdapter 的组件实现类, HttpMessageConverter 即由 RequestMappingHandlerAdapter 使用, 将请求信息转换为对象, 或将对象转换为响应信息。

HttpMessageConverter<T> 接口定义了以下几个方法。

- ❑ Boolean canRead(Class<?> clazz, MediaType mediaType): 指定转换器可以读取的对象类型, 即转换器可将请求信息转换为 clazz 类型的对象; 同时指定支持的 MIME 媒体类型 (如 text/html、application/json 等), MIME 媒体类型在 RFC2616 中定义 (MIME 类型说明可参见 http://www.w3school.com.cn/media/media_mimeref.asp)。
- ❑ Boolean canWrite(Class<?> clazz, MediaType mediaType): 指定转换器可以将 clazz 类型的对象写到响应流中, 响应流支持的媒体类型在 mediaType 中定义。
- ❑ List<MediaType> getSupportedMediaTypes(): 该转换器支持的媒体类型。
- ❑ T read(Class<? extends T> clazz, HttpInputMessage inputMessage): 将请求信息流转换为 T 类型的对象。
- ❑ void write(T t, MediaType contentType, HttpOutputMessage outputMessage): 将 T 类型的对象写到响应流中, 同时指定响应的媒体类型为 contentType。

1. HttpMessageConverter<T>的实现类

Spring 为 HttpMessageConverter<T> 提供了众多的实现类, 它们组成了一个功能强大、用途广泛的 HttpMessageConverter<T> 家族, 具体说明如表 17-2 所示。

表 17-2 HttpMessageConverter 实现类

实 现 类	功能说明
StringHttpMessageConverter	用途: 将请求信息转换为字符串。 (1) T 为 String 类型。 (2) 可读取所有媒体类型 (*/) 的请求信息, 可通过设置 supportedMediaTypes 属性指定媒体类型。 (3) 响应信息的媒体类型为 text/plain (即 Content-Type 的值)
FormHttpMessageConverter	用途: 将表单数据读取到 MultiValueMap 中。 (1) T 为 org.springframework.util.MultiValueMap<String, ?> 类型。 (2) 支持读取 application/x-www-form-urlencoded 的媒体类型, 但不支持读取 multipart/form-data 的媒体类型。

续表

实 现 类	功能说明
FormHttpMessageConverter	(3) 可写 application/x-www-form-urlencoded 及 multipart/form-data 媒体类型的响应信息
AllEncompassingFormHttpMessageConverter	扩展于 FormHttpMessageConverter。如果部分表单属性是 XML 数据, 则可用该转换器进行读取
ResourceHttpMessageConverter	用途: 读/写 org.springframework.core.io.Resource 对象。 (1) T 为 org.springframework.core.io.Resource 类型。 (2) 可读取所有媒体类型 (*) 的请求信息。 (3) 如果类路径下提供了 JAF (Java Activation Framework), 则根据 Resource 的类型指定响应的媒体类型; 否则响应的媒体类型为 application/octet-stream
BufferedImageHttpMessageConverter	用途: 读/写 BufferedImage 对象。 (1) T 为 BufferedImage 类型。 (2) 可以读取所有媒体类型。 (3) 返回 BufferedImage 相应的媒体类型, 也可以通过 contentType 显式指定
ByteArrayHttpMessageConverter	用途: 读/写二进制数据。 (1) T 为 byte[] 类型。 (2) 可读取所有媒体类型 (*) 的请求信息, 可通过设置 supportedMediaTypes 属性指定媒体类型。 (3) 响应信息的媒体类型为 application/octet-stream
SourceHttpMessageConverter	用途: 读/写 javax.xml.transform.Source 类型的数据。 (1) T 为 javax.xml.transform.Source 类型及其扩展类, 包括 javax.xml.transform.dom.DOMSource、javax.xml.transform.sax.SAXSource 及 javax.xml.transform.stream.StreamSource。 (2) 可读取 text/xml 和 application/xml 媒体类型的请求。 (3) 响应信息的媒体类型为 text/xml 或 application/xml
MarshallingHttpMessageConverter	用途: 通过 Spring 的 org.springframework.oxm.Marshaller (将 Java 对象转换为 XML) 和 Unmarshaller (将 XML 解析为 Java 对象) 读/写 XML 消息。 (1) T 为 Object 类型。 (2) 可读取 text/xml 和 application/xml 媒体类型的请求。 (3) 响应信息的媒体类型为 text/xml 或 application/xml
JaxbRootElementHttpMessageConverter	用途: 通过 JAXB2 读/写 XML 消息, 并将请求消息转换到标注 XmlRootElement 和 XmlType 注解的类中。 (1) T 为 Object 类型。 (2) 可读取 text/xml 和 application/xml 媒体类型的请求。 (3) 响应信息的媒体类型为 text/xml 或 application/xml
MappingJackson2HttpMessageConverter	用途: 利用 Jackson 开源类包的 ObjectMapper 读/写 JSON 数据。 (1) T 为 Object 类型。 (2) 可读取 application/json 类型的数据。 (3) 响应信息的媒体类型为 application/json
RssChannelHttpMessageConverter	用途: 能够读/写 RSS 种子消息。 (1) T 为 com.sun.syndication.feed.rss.Channel 类型。 (2) 可读取 application/rss+xml 类型的数据。 (3) 响应信息的媒体类型为 application/rss+xml

续表

实 现 类	功能说明
AtomFeedHttpMessageConverter	用途：和 RssChannelHttpMessageConverter 一样，能够读/写 RSS 种子消息。 (1) T 为 com.sun.syndication.feed.atom.Feed 类型。 (2) 可读取 application/atom+xml 类型的数据。 (3) 响应信息的媒体类型为 application/atom+xml

RequestMappingHandlerAdapter 默认已经装配了以下 HttpMessageConverter:

- ☐ StringHttpMessageConverter。
- ☐ ByteArrayHttpMessageConverter。
- ☐ SourceHttpMessageConverter。
- ☐ AllEncompassingFormHttpMessageConverter。

如果需要装配其他类型的 HttpMessageConverter，则可在 Spring 的 Web 容器上下文中自行定义一个 RequestMappingHandlerAdapter，如代码清单 17-15 所示。

代码清单 17-15 smart-servlet.xml

```

...
<!--① 定义一个RequestMappingHandlerAdapter -->
<bean class="org.springframework.web.servlet.mvc
        .method.annotation.RequestMappingHandlerAdapter "
        p:messageConverters-ref="messageConverters"/>

<!--① HttpMessageConverter列表-->
<util:list id="messageConverters">
    <bean class="org.springframework.http.converter.BufferedImageHttpMessageConverter" />
    <bean class="org.springframework.http.converter.ByteArrayHttpMessageConverter" />
    <bean class="org.springframework.http.converter.StringHttpMessageConverter" />
    <bean class="org.springframework.http.converter
        .support.AllEncompassingFormHttpMessageConverter" />
</util:list>
...

```

如果在 Spring Web 容器中显式定义了一个 RequestMappingHandlerAdapter，则 Spring MVC 将使用它覆盖默认的 RequestMappingHandlerAdapter。

2. 使用 HttpMessageConverter<T>

如何使用 HttpMessageConverter<T>将请求信息转换并绑定到处理方法的入参中呢？Spring MVC 提供了两种途径。

- ☐ 使用@RequestBody/@ResponseBody 对处理方法进行标注。
- ☐ 使用 HttpEntity<T>/ResponseEntity<T>作为处理方法的入参或返回值。

下面分别通过实例进行说明。首先来看使用@RequestBody/@ResponseBody 的例子，如代码清单 17-16 所示。

代码清单 17-16 UserController.java：使用@RequestBody/@ResponseBody

```

package com.smart.web;
...
import org.springframework.web.bind.annotation.PathVariable;

```

```

import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.util.FileCopyUtils;

@Controller
@RequestMapping("/user")
public class UserController {

    ...
    @RequestMapping(path = "/handle41")
    public String handle41(@RequestBody String requestBody) {①
        System.out.println(requestBody);
        return "success";
    }

    @ResponseBody ②
    @RequestMapping(path = "/handle42/{imageId}")
    public byte[] handle42(@PathVariable("imageId") String imageId) throws IOException {
        System.out.println("load image of "+imageId);
        Resource res = new ClassPathResource("/image.jpg");
        byte[] fileData = FileCopyUtils.copyToByteArray(res.getInputStream());
        return fileData;
    }
}

```

将请求报文转换为字符串绑定到 requestBody 入参中

读取一张图片，并将图片数据输出到响应流中，客户端将显示这张图片

在代码清单 17-15 中，已经为 RequestMappingHandlerAdapter 注册了若干个 `HttpMessageConverter`。handle41()方法的 requestBody 入参标注了一个 `@RequestBody` 注解，如①处所示，Spring MVC 将根据 requestBody 的类型查找匹配的 `HttpMessageConverter`。由于 `StringHttpMessageConverter` 的泛型类型对应 `String`，所以 `StringHttpMessageConverter` 将被 Spring MVC 选中，用它将请求体信息进行转换并将结果绑定到 requestBody 入参上。

handle42()方法拥有一个 `@ResponseBody` 注解，如②处所示。由于方法返回值类型为 `byte[]`，所以 Spring MVC 根据类型匹配的查找规则将使用 `ByteArrayHttpMessageConverter` 对返回值进行处理，即将图片数据流输出到客户端。

下面编写一个测试用例，通过 `RestTemplate` 对 handle41()及 handle42()这两个方法进行测试，如代码清单 17-17 所示。

代码清单 17-17 UserControllerTest.java

```

package com.smart.web;
...import org.junit.Test;
import org.springframework.core.io.FileSystemResource;
import org.springframework.core.io.Resource;
import org.springframework.util.*;
import org.springframework.web.client.RestTemplate;

public class UserControllerTest {

    @Test
    public void testHandle41() {
        RestTemplate restTemplate = new RestTemplate();
    }
}

```



```

MultiValueMap<String, String> form = new LinkedMultiValueMap<String, String>();
form.add("userName", "tom");    第一个参数为 URL, 第二个参数通过 MultiValueMap
form.add("password", "123456"); 准备报文体的参数数据
form.add("age", "45");
restTemplate.postForLocation(①
    "http://localhost:8080/chapter17/user/handle41.html", form);
}

```

```

@Test
public void testHandle42() throws IOException{    第二个参数为报文体参数数据, 第三
    RestTemplate restTemplate = new RestTemplate(); 个参数指定方法的返回值类型, 第四
    byte[] response = restTemplate.postForObject( ② 个参数为 URL 占位符参数的值
        "http://localhost:8080/chapter17/user/handle42/{itemId}.html",
        null, byte[].class, "1233");
    Resource outFile = new FileSystemResource("d:/image_copy.jpg");
    FileCopyUtils.copy(response, outFile.getFile());
}
}

```

RestTemplate 是 Spring 的模板类, 在客户端程序中可使用该类调用 Web 服务器端的服务, 它支持 REST 风格的 URL。此外, 它像 RequestMappingHandlerAdapter 一样拥有一张 HttpResponseMessageConverter 的注册表, RestTemplate 默认已经注册了以下 HttpResponseMessageConverter:

- ☐ ByteArrayHttpMessageConverter。
- ☐ StringHttpMessageConverter。
- ☐ ResourceHttpMessageConverter。
- ☐ SourceHttpMessageConverter。
- ☐ AllEncompassingFormHttpMessageConverter。

所以, 在默认情况下, RestTemplate 就可以利用这些 HttpResponseMessageConverter 对响应数据进行相应的转换处理。可通过 RestTemplate 的 setMessageConverters(List<HttpMessageConverter<?>> messageConverters)方法手工注册 HttpResponseMessageConverter。

和 @RequestBody/@ResponseBody 类似, HttpEntity<?>不但可以访问请求和响应报文体的数据, 还可以访问请求和响应报文头的信息。Spring MVC 根据 HttpEntity 的泛型类型查找对应的 HttpResponseMessageConverter。

使用 HttpEntity<?>对代码清单 17-16 中的两个方法进行改造, 完成相似的功能, 如代码清单 17-18 所示。

代码清单 17-18 UserController.java: 使用 HttpEntity<?>

```

@RequestMapping(path = "/handle43")
public String handle43(HttpEntity<String> httpEntity){
    long contentLen = httpEntity.getHeaders().getContentLength();
    System.out.println(httpEntity.getBody());
    return "success";
}

```

①
使用 StringHttpMessageConverter 将请求报文体及报文头的信息绑定到 httpEntity 中, 在方法中可以对相应信息进行访问

```

@RequestMapping(path = "/handle44/{imageId}")
public ResponseEntity<byte[]> handle44(

```



```

    @PathVariable("imageId")String imageId) throws Throwable { ②
        Resource res = new ClassPathResource("/image.jpg");
        byte[] fileData =FileCopyUtils.copyToByteArray(res.getInputStream());
        ResponseEntity<byte[]> responseEntity =
            new ResponseEntity<byte[]>(fileData,HttpStatus.OK);
        return responseEntity;
    }

```

在方法中创建 `HttpEntity<byte[]>` 对象并返回, `ByteArrayHttpMessageConverter` 负责将其输出到响应流中

在①处使用 `HttpEntity<String>` 指定入参的类型, Spring MVC 分析出泛型类型为 `String`, 使用 `StringHttpMessageConverter` 将请求体内容绑定到 `httpEntity` 中, 返回的 `String` 类型的值为逻辑视图名。

②处的处理方法返回值类型为 `ResponseEntity<byte[]>`, Spring MVC 分析出泛型类型为 `byte[]`, 使用 `ByteArrayHttpMessageConverter` 输出图片数据流。

通过以上两个实例, 可以得出以下几条结论。

- ❑ 当控制器处理方法使用 `@RequestBody/@ResponseBody` 或 `HttpEntity<T>/ResponseEntity<T>` 时, Spring MVC 才使用注册的 `HttpMessageConverter` 对请求/响应消息进行处理。
- ❑ 当控制器处理方法使用 `@RequestBody/@ResponseBody` 或 `HttpEntity<T>/ResponseEntity<T>` 时, Spring 首先根据请求头或响应头的 `Accept` 属性选择匹配的 `HttpMessageConverter`, 然后根据参数类型或泛型类型的过滤得到匹配的 `HttpMessageConverter`, 如果找不到可用的 `HttpMessageConverter` 则报错。
- ❑ `@RequestBody` 和 `@ResponseBody` 不需要成对出现。如果方法入参使用了 `@RequestBody`, 则 Spring MVC 选择匹配的 `HttpMessageConverter` 将请求消息转换并绑定到该入参中。如果处理方法标注了 `@ResponseBody`, 则 Spring MVC 选择匹配的 `HttpMessageConverter` 将方法返回值转换并输出响应消息。
- ❑ `HttpEntity<T>/ResponseEntity<T>` 的功能和 `@RequestBody/@ResponseBody` 相似。

3. 处理 XML 和 JSON

Spring MVC 提供了几个处理 XML 和 JSON 格式的请求/响应消息的 `HttpMessageConverter`。

- ❑ `MarshallingHttpMessageConverter`: 处理 XML 格式的请求或响应消息。
- ❑ `Jaxb2RootElementHttpMessageConverter`: 同上, 底层使用 JAXB。
- ❑ `MappingJackson2HttpMessageConverter`: 处理 JSON 格式的请求或响应消息。

因此, 只要在 Spring Web 容器中为 `RequestMappingHandlerAdapter` 装配好相应的处理 XML 和 JSON 格式的请求/响应消息的 `HttpMessageConverter`, 并在交互中通过请求的 `Accept` 指定 MIME 类型, Spring MVC 就可使服务器端的处理方法和客户端透明地通过 XML 或 JSON 格式的消息进行通信, 开发者几乎无须关心通信层数据格式的问题, 可以将精力集中到业务层的处理上。单就这一点而言, 其他 MVC 框架和 Spring MVC 相比, 就如诸葛亮给关云长的评语一样: “犹未及美髯公之绝伦超群也。”

首先为 RequestMappingHandlerAdapter 装配可处理 XML 和 JSON 格式的请求/响应消息的 HttpMessage Converter, 如代码清单 17-19 所示。

代码清单 17-19 smart-servlet.xml

```
...
<bean class="org.springframework.web.servlet.mvc
    .method.annotation.RequestMappingHandlerAdapter"
    p:messageConverters-ref="messageConverters"/>
<util:list id="messageConverters">
    <bean class="org.springframework.http.converter.BufferedImageHttpMessageConverter" />
    <bean class="org.springframework.http.converter.ByteArrayHttpMessageConverter" />
    <bean class="org.springframework.http.converter.StringHttpMessageConverter" />
    <bean class="org.springframework.http.converter.
        support.AllEncompassingFormHttpMessageConverter"/>
    <bean class="org.springframework.http.converter.xml
        ↗ .MarshallingHttpMessageConverter"
        p:marshaller-ref="xmlMarshaller"
        p:unmarshaller-ref="xmlMarshaller">
    </bean>
    <bean class="org.springframework.http.converter.json
        ↗ .MappingJackson2HttpMessageConverter"/>
</util:list>

<!--①声明Marshaller, 使用XStream技术-->
<bean id="xmlMarshaller" class="org.springframework.oxm.xstream.XStreamMarshaller">
    <property name="streamDriver">
        <bean class="com.thoughtworks.xstream.io.xml.StaxDriver"/>②↖
    </property>
    使用 STAX 对 XML 消息进行处理,
    STAX 占用内存少, 响应速度也很快

    <property name="annotatedClasses">
        使用 XStream 的注解定义 XML 转换规则,
        使用 XStream 注解的类在此声明
        <list>
            <value>com.smart.domain.User</value> ③↖
        </list>
    </property>
</bean>
...
```

然后在控制器中编写相应的方法, 如代码清单 17-20 所示。

代码清单 17-20 UserController.java: 支持XML和JSON格式的消息处理方法

```
@RequestMapping(path = "/handle51")
public ResponseEntity<User> handle51(HttpEntity<User> requestEntity) {
    User user = requestEntity.getBody();
    user.setUserId("1000");
    return new ResponseEntity<User>(user, HttpStatus.OK);
}
```

对于服务器端的处理方法而言, 除使用@RequestBody/@ResponseBody 或 HttpEntity<T>/ResponseEntity<T>进行方法签名外, 不需要进行任何额外的处理, 借由 Spring MVC 中装配的 HttpMessageConverter, 它便拥有了处理 XML 及 JSON 格式的消息的能力。

在接收到一个 HTTP 请求时, handle51()如何知道请求消息的格式? 在处理完成后, 又根据什么确定响应消息的格式? 答案很简单: 通过请求消息头的 Content-Type 及

Accept 属性确定。下面使用 RestTemplate 编写调用 handle51() 方法的客户端程序，如代码清单 17-21 所示。

代码清单 17-21 UserControllerTest.java: 使用XML格式的请求/响应消息

```
package com.smart.web;
...
import com.smart.domain.User;
import com.thoughtworks.xstream.io.xml.StaxDriver;

public class UserControllerTest {

    //①使用RestTemplate测试UserController#handle51()方法
    @Test
    public void testHandle51() throws IOException{
        RestTemplate restTemplate = buildRestTemplate();

        User user = new User();
        user.setUserName("tom");
        user.setPassword("1234");
        user.setRealName("汤姆"); ①-1

        HttpHeaders entityHeaders = new HttpHeaders();
        entityHeaders.setContentType(MediaType.valueOf("application/xml;UTF-8"));
        entityHeaders.setAccept(Collections.singletonList(MediaType.APPLICATION_XML)); ①-2

        HttpEntity<User> requestEntity = new HttpEntity<User>(user, entityHeaders);
        ResponseEntity<User> responseEntity = restTemplate.exchange(
            "http://localhost:8080/chapter17/user/handle51.html",
            HttpMethod.POST, requestEntity, User.class); ①-3

        User responseUser = responseEntity.getBody();
        Assert.assertNotNull(responseUser);
        Assert.assertEquals("1000", responseUser.getUserId());
        Assert.assertEquals("tom", responseUser.getUserName());
        Assert.assertEquals("汤姆", responseUser.getRealName()); ①-4

    }

    //②创建RestTemplate实例
    private RestTemplate buildRestTemplate() {
        RestTemplate restTemplate = new RestTemplate();

        XStreamMarshaller xmlMarshaller = new XStreamMarshaller();
        xmlMarshaller.setStreamDriver(new StaxDriver());
        xmlMarshaller.setAnnotatedClasses(new Class[]{User.class}); ②-1

        MarshallingHttpMessageConverter xmlConverter =
            new MarshallingHttpMessageConverter();
        xmlConverter.setMarshaller(xmlMarshaller);
        xmlConverter.setUnmarshaller(xmlMarshaller);
        restTemplate.getMessageConverters().add(xmlConverter); ②-2
    }
}
```

创建 User 对象，它将通过 RestTemplate 流化为 XML 请求报文

指定请求的报文头信息

将 User 流化为 XML，放到报文体中，同时指定请求方法及报文头

将请求响应消息转换为 User 对象，并对响应值进行判断

使用 XStream 流化器，使用 STAX 技术处理 XML，同时加载使用了 XStream 注解的 User 类

创建处理 XML 报文的 HttpMessageConverter，将其组装到 RestTemplate 中

创建处理 JSON 报文的 `HttpMessageConverter`，
将其组装到 `RestTemplate` 中

```
MappingJackson2HttpMessageConverter jsonConverter =
    new MappingJackson2HttpMessageConverter();
restTemplate.getMessageConverters().add(jsonConverter);
return restTemplate;
```

②-3

服务器端启动 Web 服务，运行 `testhandle51WithXml()` 测试方法，使用网络监控工具（如 TcpTrace）拦截请求响应报文，如图 17-6 所示。

请求
报文

```
POST /chapter17/user/handle51.html HTTP/1.1
Accept: application/xml
Content-Type: application/xml; charset=UTF-8
User-Agent: Java/1.6.0_14
Host: localhost:8080
Connection: keep-alive
Content-Length: 101
```

```
<?xml version="1.0" ?><message userName="tom" password="1234"
realName="汤姆" salary="0"></message>
```

响应
报文

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/xml
Transfer-Encoding: chunked
Date: Mon, 12 Sep 2015 12:01:46 GMT
```

```
6f
<?xml version="1.0" ?><message id="1000" userName="tom" password="1234"
realName="汤姆" salary="0"></message>
0
```

图 17-6 HTTP 请求响应报文（XML 格式）

通过以上 HTTP 请求/响应报文，我们清楚地知道客户端的 `User` 对象被流化为一段对应的 XML 报文（阴影部分），同时通过报文头属性 `Accept` 和 `Content-Type` 指定接收的 MIME 类型和本请求的报文内容均为 `application/xml`。

请求报文被服务器端的 `UserController#handle51()` 方法正确处理，它根据请求的报文头属性 `Accept` 决定将服务器端的 `User` 对象流化为 XML 并返回 HTTP 响应报文（阴影部分），同时指定响应报文的 `Content-Type` 属性为 `application/xml`。

如果希望通过 JSON 方式进行通信，则仅需对客户端代码进行轻微的调整即可，服务器端代码无须作任何更改，如代码清单 17-22 所示。

代码清单 17-22 `UserControllerTest.java`：使用 JSON 格式的请求/响应消息

```
public void testhandle51(){
    @Test
    RestTemplate restTemplate = buildRestTemplate();

    ...

    HttpHeaders entityHeaders = new HttpHeaders();
    entityHeaders.setContentType(MediaType.valueOf("application/json;UTF-8"));
    entityHeaders.setAccept(Collections.singletonList(MediaType.APPLICATION_JSON));
    HttpEntity<User> requestEntity = new HttpEntity<User>(user, entityHeaders);
```

```

...
}

```

将请求报文头的 Content-Type 及 Accept 属性更改为 application/json 即可。再次执行 testhandle51()方法, 观察 HTTP 请求/响应报文, 如图 17-7 所示。

请求 报 文	<pre> POST /chapter17/user/handle51.html HTTP/1.1 Accept: application/json Content-Type: application/json User-Agent: Java/1.6.0_14 Host: localhost:8080 Connection: keep-alive Content-Length: 109 {"userId":null,"realName":"tomson","birthday":null,"salary":0,"dept":null,"password":"1234","userName":"tom"} </pre>
响 应 报 文	<pre> HTTP/1.1 200 OK Server: Apache-Coyote/1.1 Content-Type: application/json Transfer-Encoding: chunked Date: Mon, 12 Sep 2015 12:41:24 GMT 6f {"password":"1234","userName":"tom","birthday":null,"salary":0,"realName":"tomson","userId":"1000","dept":null} 0 </pre>

图 17-7 HTTP 请求响应报文 (JSON 格式)

可见, 请求报文头的 Content-Type 及 Accept 属性更改为 application/json, User 对象的数据以 JSON 格式进行传递。

17.2.6 使用 @RestController 和 AsyncRestTemplate

1. @RestController

从 Spring 4.0 开始, Spring 以 Servlet 3.0 为基础进行开发。如果使用 Spring MVC 测试框架, 则需要指定 Servlet 3.0 兼容的 JAR 包 (因为其 Mock 的对象都是基于 Servlet 3.0 的)。为方便 Rest 的开发, Spring 引入了一个新的 @RestController 注解, 该注解已经标注了 @ResponseBody 和 @Controller。

```

@Controller
@ResponseBody
public @interface RestController {
}

```

这样, 通过直接在控制器上标注新的 @RestController, 就不需要在每个 @RequestMapping 方法上添加 @ResponseBody 了。

```

@RestController
public class UserController {
}

```

当我们使用 REST 风格开发应用程序时, Spring MVC 仅需以下两行配置就可以了:

```

<context:component-scan base-package="com.smart.*" />
<mvc:annotation-driven/>

```

2. AsyncRestTemplate

Spring 4.0 添加了一个 `AsyncRestTemplate`，支持以异步无阻塞方式进行服务访问。以下是服务器端的 Rest 服务实现类，如代码清单 17-23 所示。

代码清单 17-23 UserController.java

```
@RestController
public class UserController {
    private UserService userService;
    @Autowired
    public UserController(UserService userService) {
        this.userService = userService;
    }
    @RequestMapping("/api")
    public Callable<User> api() {
        System.out.println("====hello");
        return new Callable<User>() {
            @Override
            public User call() throws Exception {
                Thread.sleep(10L * 1000); //①暂停10秒
                User user = new User();
                user.setId(1L);
                user.setName("haha");
                return user;
            }
        };
    }
}
```

在这里，我们模拟一个执行时间为 10 秒的服务器方法，如果客户端使用 `RestTemplate`，则将以同步方式进行调用，即客户端代码需要等待服务器端返回后才继续执行。下面使用 `AsyncRestTemplate` 以异步的方式进行服务调用，如代码清单 17-24 所示。

代码清单 17-24 main()方法

```
public static void main(String[] args) {
    AsyncRestTemplate template = new AsyncRestTemplate();

    //①调用完后立即返回（没有阻塞）
    ListenableFuture<ResponseEntity<User>> future =
        template.getForEntity("http://localhost:8080/chapter17/api", User.class);

    //②处理服务器端响应的异步回调方法
    future.addCallback(new ListenableFutureCallback<ResponseEntity<User>>() {
        @Override
        public void onSuccess(ResponseEntity<User> result) {
            System.out.println("====client get result : " + result.getBody());
        }

        @Override
        public void onFailure(Throwable t) {
            System.out.println("====client failure : " + t);
        }
    });
    System.out.println("==no wait");
}
```


①处的执行会立即返回，不会同步阻塞。待服务器端返回请求响应后，②处注册的回调函数会被自动异步调用。

AsyncRestTemplate 默认使用 SimpleClientHttpRequestFactory 进行 HTTP 操作，其底层通过 java.net.HttpURLConnection 实现。也可以使用其他的实现方式，如 template.setAsyncRequestFactory(new HttpComponentsAsyncClientHttpRequestFactory()) 语句将使用 Apache 的 http components 作为底层访问组件。

17.2.7 处理模型数据

对于 MVC 框架来说，模型数据是最重要的，因为控制（C）是为了产生模型数据（M），而视图（V）则是为了渲染模型数据。

通过前面的学习我们已经知道，Spring MVC 通过 @RequestMapping 将请求引导到处理方法上，使用合适的方法签名将请求消息绑定到入参中。方法入参绑定请求消息只是处理方法的第一步，还有更为重要的任务等待完成，即根据入参执行相应的逻辑，产生模型数据，导向到特定视图中。

将模型数据暴露给视图是 Spring MVC 框架的一项重要工作。Spring MVC 提供了多种途径输出模型数据，介绍如下。

- ❑ ModelAndView: 当处理方法返回值类型为 ModelAndView 时，方法体即可通过该对象添加模型数据。
- ❑ @ModelAttribute: 在方法入参标注该注解后，入参的对象就会放到数据模型中。
- ❑ Map 及 Model: 如果方法入参为 org.springframework.ui.Model、org.springframework.ui.ModelMap 或 java.util.Map，则当处理方法返回时，Map 中的数据会自动添加到模型中。
- ❑ @SessionAttributes: 将模型中的某个属性暂存到 HttpSession 中，以便多个请求之间可以共享这个属性。

1. ModelAndView

控制器处理方法的返回值如果为 ModelAndView，则其既包含视图信息，又包含模型数据信息，这样 Spring MVC 就可以使用视图对模型数据进行渲染了。可以简单地将模型数据看成一个 Map<String,Object> 对象。

在处理方法的方法体中，可以使用如下方法添加模型数据。

- ❑ ModelAndView addObject(String attributeName, Object attributeValue)。
- ❑ ModelAndView addAllObjects(Map<String,?> modelMap)。

可以通过如下方法设置视图。

- ❑ void setView(View view): 指定一个具体的视图对象。
- ❑ void setViewName(String viewName): 指定一个逻辑视图名。

ModelAndView 的使用非常简单，请参照 17.1.3 节的实例。

2. @ModelAttribute

如果希望将方法入参对象添加到模型中,则仅需在相应入参前使用@ModelAttribute注解即可。来看一个具体的实例,如代码清单 17-25 所示。

代码清单 17-25 UserController.java: 在方法入参前使用@ModelAttribute注解

```
@RequestMapping(path = "/handle61")
public String handle61(@ModelAttribute("user") User user){
    user.setUserId("1000");
    return "/user/createSuccess";
}
```

Spring MVC 将请求消息绑定到 User 对象中,然后再以 user 为键将 User 对象放到模型中。在准备对视图进行渲染前, Spring MVC 还会进一步将模型中的数据转储到视图的上下文中并暴露给视图对象。对于 JSP 视图来说, Spring MVC 会将模型数据转储到 ServletRequest 的属性列表中(通过 ServletRequest#setAttribute(String name, Object o)方法保存)。

handle61()方法返回的逻辑视图名为/user/createSuccess,对应 createSuccess.jsp 视图对象,这样 createSuccess.jsp 就可以使用\${user.userName}等方式顺利地访问到模型中的数据了。

除了可以在方法入参上使用@ModelAttribute 注解外,还可以在方法定义中使用@ModelAttribute 注解。Spring MVC 在调用目标处理方法前,会先逐个调用在方法级上标注了@ModelAttribute 注解的方法,并将这些方法的返回值添加到模型中。下面是在方法级上使用@ModelAttribute 注解的实例,如代码清单 17-26 所示。

代码清单 17-26 UserController.java: 在方法定义处使用@ModelAttribute注解

```
@ModelAttribute("user")
public User getUser(){
    User user = new User();
    user.setUserId("1001");
    return user;
}

@RequestMapping(path = "/handle62")
public String handle62(@ModelAttribute("user") User user){
    user.setUserName("tom");
    return "/user/showUser";
}
```

在访问 UserController 中的任何一个请求处理方法前, Spring MVC 先执行该方法, 并将返回值以 user 为键添加到模型中

① ←

在此, 模型数据会赋给 User 的入参, 然后再根据 HTTP 请求消息进一步填充覆盖 user 对象

② ←

在访问 UserController 中的任何一个请求处理方法前, 都会事先执行标注了@ModelAttribute 的 getUser()方法, 并将其返回值以 user 为键添加到模型中。

由于②处的 handle62()方法使用了入参级的@ModelAttribute 注解, 且属性名和①处方法级@ModelAttribute 的属性名相同。这时, Spring MVC 会将①处获取的模型属性先赋值给②处的入参 user, 然后再根据 HTTP 请求消息对 user 进行填充覆盖, 得到一个整合版本的 user 对象。

**提示**

处理方法入参最多只能使用一个 Spring MVC 的注解，如 `handle62(@ModelAttribute("user") User user)` 的 `user` 入参使用了 `@ModelAttribute`，就不能再使用 `@RequestParam` 或 `@CookieValue`。如果使用了两个注解，则 Spring MVC 将抛出异常。

3. Map 及 Model

Spring MVC 在内部使用一个 `org.springframework.ui.Model` 接口存储模型数据，它的功能类似于 `java.util.Map`，但它比 `Map` 易用。`org.springframework.ui.ModelMap` 实现了 `Map` 接口，而 `org.springframework.ui.ExtendedModelMap` 扩展于 `ModelMap` 的同时实现了 `Model` 接口。

Spring MVC 在调用方法前会创建一个隐含的模型对象，作为模型数据的存储容器，我们称之为“隐含模型”。如果处理方法的入参为 `Map` 或 `Model` 类型，则 Spring MVC 会将隐含模型的引用传递给这些入参。在方法体内，开发者可以通过这个入参对象访问到模型中的所有数据，也可以向模型中添加新的属性数据。来看一个简单的例子，如代码清单 17-27 所示。

代码清单 17-27 UserController.java: 使用 `ModelMap` 的入参

```
@ModelAttribute("user")
public User getUser(){
    User user = new User();
    user.setUserId("1001");
    return user;
}

@RequestMapping(path = "/handle63")
public String handle63(ModelMap modelMap){
    modelMap.addAttribute("testAttr", "value1");
    User user = (User)modelMap.get("user");
    user.setUserName("tom");
    return "/user/showUser";
}
```

Spring MVC 将请求对应的隐含模型对象传递给 `modelMap`，因此在方法中可以通过它访问模型中的数据

Spring MVC 一旦发现处理方法有 `Map` 或 `Model` 类型的入参，就会将请求内在的隐含模型对象传递给这些参数，因此在方法体中可以通过这个入参对模型中的数据进行读/写操作。

4. @SessionAttributes

如果希望在多个请求之间共用某个模型属性数据，则可以在控制器类中标注一个 `@SessionAttributes`，Spring MVC 会将模型中对应的属性暂存到 `HttpSession` 中。来看一个实例，如代码清单 17-28 所示。

代码清单 17-28 UserController.java: 使用 `@SessionAttributes` 注解

```
package com.smart.web;

import org.springframework.web.bind.annotation.ModelAttribute;
```

```
import org.springframework.web.bind.annotation.SessionAttributes;
import org.springframework.web.bind.support.SessionStatus;
```

```
@Controller
```

```
@RequestMapping("/user")
```

```
@SessionAttributes("user") ①
```

将②处的模型属性自动保存到
HttpSession 中

```
public class UserController {
```

```
    @RequestMapping(path = "/handle71")
```

```
    public String handle71(@ModelAttribute("user") User user){ ②
```

```
        user.setUserName("John");
```

```
        return "redirect:/user/handle72.html";
```

```
    }
```

```
    @RequestMapping(path = "/handle72")
```

```
    public String handle72(ModelMap modelMap, SessionStatus sessionStatus){
```

```
        User user = (User)modelMap.get("user"); ③
```

```
        if(user != null){
```

```
            user.setUserName("Jetty");
```

```
            sessionStatus.setComplete(); ④
```

读取模型中的数据

```
        }
```

```
        return "/user/showUser";
```

让 Spring MVC 清除本处理器对应的会话属性

```
    }
```

在①处标注的 `@SessionAttributes("user")` 会自动将本处理器中任何处理方法属性名为 `user` 的模型属性透明地存储到 `HttpSession` 中。在②处，`handle71()` 方法的 `User user` 入参会添加到隐含模型中，于是这个模型属性在 `handle71()` 方法执行时，会由 Spring MVC 将其透明地保存到 `HttpSession` 中。

`handle71()` 返回的逻辑视图名为 `redirect:handle72.html`，它将发起另一个请求，而这个请求由 `handle72()` 负责处理。`handle72()` 和 `handle71()` 位于不同的请求上下文中，之所以在③处可以获取名为 `user` 的模型属性，就是因为 `@SessionAttributes("user")` 透明地将 `handle71()` 的 `user` 模型属性存储到 `HttpSession` 中，而 `handle72()` 的隐含模型又自动从 `HttpSession` 中获取到这个模型属性。



提示

我们知道，在一般情况下，控制器方法返回字符串类型的值会被当成逻辑视图名处理。但如果字符串带“forward:”或“redirect:”前缀，则 Spring MVC 将对它们进行特殊处理：将“forward:”或“redirect:”当成指示符，其后的字符作为 URL 处理，如“redirect:handle72.html”或“forward:http://www.baidu.com”。`redirect` 会让浏览器发起一个新的请求，而 `forward` 请求与当前请求同属一个请求。

`handle72()` 方法还包含一个 `SessionStatus` 入参，当调用 `SessionStatus#setComplete()` 方法时，Spring MVC 会清除该控制器类的所有会话属性；否则这个会话属性会一直保存在 `HttpSession` 中。

很可惜，当启动 Web 应用并向 `handle71()` 发送请求时，Spring MVC 会抛出以下异常信息：

```
org.springframework.web.HttpSessionRequiredException: Session attribute 'user'
required - not found in session
...
```

这个异常曾令笔者大伤脑筋，因为 Spring 仅宣称，@SessionAttributes 的作用是将处理方法对应的模型属性透明地保存到 HttpSession 中，并没有要求 HttpSession 中必须事先拥有对应的模型属性。通过研究 Spring MVC 的源码，才找到了问题的答案。

原来 Spring MVC 对 @ModelAttribute 及 @SessionAttributes 的处理遵循一个特定的流程，当流程条件不满足时就会报错。这个处理流程简单说明如下。

- ① Spring MVC 在调用处理方法前，在请求线程中自动创建一个隐含的模型对象。
- ② 调用所有标注了 @ModelAttribute 的方法，并将方法返回值添加到隐含模型中。
- ③ 查看 Session 中是否存在 @SessionAttributes("xxx") 所指定的 xxx 属性，如果有，则将其添加到隐含模型中。如果隐含模型中已经有 xxx 属性，则该步操作会覆盖模型中已有的属性值。
- ④ 对标注了 @ModelAttribute("xxx") 处理方法的入参按如下流程处理。
 - ④.1 如果隐含模型拥有名为 xxx 的属性，则将其赋给该入参，再用请求消息填充该入参对象直接返回，否则转到 ④.2。
 - ④.2 如果 xxx 是会话属性，即在处理类定义处标注了 @SessionAttributes("xxx")，则尝试从会话中获取该属性，并将其赋给该入参，然后再用请求消息填充该入参对象。如果在会话中找不到对应的属性，则抛出 HttpSessionRequiredException 异常。否则转到 ④.3。
 - ④.3 如果隐含模型中不存在 xxx 属性，且 xxx 也不是会话属性，则创建入参的对象实例，然后再用请求消息填充该入参。

分析代码清单 17-28，由于在 ① 处标注了 @SessionAttributes("user")，因此 user 为会话属性，Spring MVC 在对 handle71(@ModelAttribute("user") User user) 进行处理时，会先在隐含模型中查询是否有对应的属性，如果不存在，则继续在会话中查询该属性。由于在会话中也不存在该属性，因此报 HttpSessionRequiredException 异常。

解决该异常的方法很简单，只需添加一个标注 @ModelAttribute("user") 的方法，以便让 Spring MVC 在处理 handle71(@ModelAttribute("user") User user) 方法前先向隐含模型中添加 user 属性，这样 ④.1 步就会执行，而 ④.2 步不会执行，这样就不会抛出 HttpSessionRequiredException 异常，如代码清单 17-29 所示。

代码清单 17-29 UserController.java: 解决 HttpSessionRequiredException 异常问题

```
package com.smart.web;

import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.SessionAttributes;
import org.springframework.web.bind.support.SessionStatus;

@Controller
@RequestMapping("/user")
@SessionAttributes("user")
```

```
public class UserController {
```

```
    @ModelAttribute("user")
    public User getUser(){
        User user = new User();
        user.setUserId("1001");
        return user;
    }
```

该方法会向隐含模型中添加一个名为
user 的模型属性

①

```
    @RequestMapping(path = "/handle71")
    public String handle71(@ModelAttribute("user") User user){
        user.setUserName("John");
        return "redirect:handle72.html";
    }
```

```
    @RequestMapping(path = "/handle72")
    public String handle72(ModelMap modelMap, SessionStatus sessionStatus){
        User user = (User)modelMap.get("user");
        if(user != null){
            user.setUserName("Jetty");
            sessionStatus.setComplete();
        }
        return "/user/showUser";
    }
}
```

`@SessionAttributes` 除了可以通过属性名指定需要放到会话中的属性外，还可以通过模型属性的对象类型指定哪些模型属性需要放到会话中。如 `@SessionAttributes(types=User.class)` 会将隐含模型中所有类型为 `User.class` 的属性添加到会话中。

此外，`@SessionAttributes` 还可以通过属性名及 `types` 一起指定，二者都允许多值，放到会话中的属性是二者的并集。如以下声明方式都是合法的。

- ☐ `@SessionAttributes(value={"user1","user2"})`: 将名为 `user1` 及 `user2` 的模型属性添加到会话中。
- ☐ `@SessionAttributes(types={User.class,Dept.class})`: 将模型中所有类型为 `User` 及 `Dept` 的属性添加到会话中。
- ☐ `@SessionAttributes(value={"user1","user2"},types={Dept.class})`: 将名为 `user1` 及 `user2` 的模型属性添加到会话中，同时将所有类型为 `Dept` 的模型属性添加到会话中。

17.3 处理方法的数据绑定

在 17.2 节中我们知道，Spring 会根据请求方法签名的不同，将请求消息中的信息以一定的方式转换并绑定到请求方法的入参中。当请求消息到达真正需要调用的方法时（如指定的业务方法），Spring MVC 还有很多工作要做，包括数据转换、数据格式化及数据校验等。

17.3.1 数据绑定流程剖析

Spring MVC 通过反射机制对目标处理方法的签名进行分析，将请求消息绑定到处理方法的入参中。数据绑定的核心部件是 `DataBinder`，其运行机制描述如图 17-8 所示。

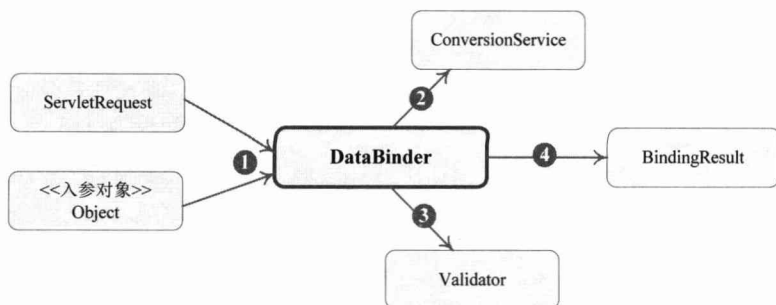


图 17-8 数据绑定

Spring MVC 主框架将 `ServletRequest` 对象及处理方法的入参对象实例传递给 `DataBinder`，`DataBinder` 首先调用装配在 Spring Web 上下文中的 `ConversionService` 组件进行数据类型转换、数据格式化等工作，将 `ServletRequest` 中的消息填充到入参对象中，然后调用 `Validator` 组件对已经绑定了请求消息数据的入参对象进行数据合法性校验，最终生成数据绑定结果 `BindingResult` 对象。`BindingResult` 包含了已完成数据绑定的入参对象，还包含相应的校验错误对象。Spring MVC 抽取 `BindingResult` 中的入参对象及校验错误对象，将它们赋给处理方法的相应入参。

后面我们将对数据绑定过程中发生的数据转换、数据格式化及数据校验等工作进行详细阐述。

17.3.2 数据转换

在第 6 章中，我们已经学习了如何编写并装配自定义属性编辑器的知识。Java 标准的 `PropertyEditor` 的核心功能是将一个字符串转换为一个 Java 对象，以便根据界面的输入或配置文件中的配置字符串构造出一个 JVM 内部的 Java 对象。

但是 Java 原生的 `PropertyEditor` 存在以下不足：

- ❑ 只能用于字符串到 Java 对象的转换，不适用于任意两个 Java 类型之间的转换。
- ❑ 对源对象及目标对象所在的上下文信息（如注解、所在宿主类的结构等）不敏感，在类型转换时不能利用这些上下文信息实施高级的转换逻辑。

鉴于此，Spring 在核心模型中添加了一个通用的类型转换模块，类型转换模块位于 `org.springframework.core.convert` 包中。Spring 希望用这个类型转换体系替换 Java 标准的 `PropertyEditor`。但由于历史原因，Spring 将同时支持两者，在 Bean 配置、Spring MVC 处理方法入参绑定中使用它们。

1. ConversionService

ConversionService 是 Spring 类型转换体系的核心接口，它位于 org.springframework.core.convert 包中，也是该包中的唯一一个接口。ConversionService 接口定义了以下 4 个方法。

- ❑ boolean canConvert(Class<?> sourceType, Class<?> targetType): 判断是否可以将一个 Java 类转换为另一个 Java 类。
- ❑ Boolean canConvert(TypeDescriptor sourceType, TypeDescriptor targetType): 需转换的类将以成员变量的方式出现在宿主类中。TypeDescriptor 不但描述了需转换类的信息，还描述了从宿主类的上下文信息，如成员变量上的注解，成员变量是否以数组、集合或 Map 的方式呈现等。类型转换逻辑可以利用这些信息做出各种灵活的控制。
- ❑ <T> T convert(Object source, Class<T> targetType): 将原类型对象转换为目标类型对象。
- ❑ Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType): 将对象从原类型对象转换为目标类型对象，此时往往会用到所在宿主类的上下文信息。

第一个和第三个接口方法类似于 PropertyEditor，它们不关注类型对象所在的上下文信息，只简单地完成两个类型对象的转换，唯一的区别在于这两个方法支持任意两个类型的转换。而第二个和第四个接口方法会参考类型对象所在宿主类的上下文信息，并利用这些信息进行类型转换。

可以利用 org.springframework.context.support.ConversionServiceFactoryBean 在 Spring 的上下文中定义一个 ConversionService。Spring 将自动识别出上下文中的 ConversionService，并在 Bean 属性配置及 Spring MVC 处理方法入参绑定等场合使用它进行数据转换。具体配置如下：

```
<bean id="conversionService"
      class="org.springframework.context.support.ConversionServiceFactoryBean"/>
```

该 FactoryBean 创建 ConversionService 内建了很多转换器，可完成大多数 Java 类型的转换工作。除了包括将 String 对象转换为各种基础类型的对象外，还包括 String、Number、Array、Collection、Map、Properties 及 Object 之间的转换器。

可通过 ConversionServiceFactoryBean 的 converters 属性注册自定义的类型转换器，代码如下：

```
<bean id="conversionService"
      class="org.springframework.context.support.ConversionServiceFactoryBean">
  <property name="converters">
    <list>
      <bean class="com.smart.MyCustomConverter1"/>
      <bean class="com.smart.MyCustomConverter2"/>
    </list>
  </property>
</bean>
```


在 6.2 节中我们知道,通过 CustomEditorConfigurer 注册的自定义属性编辑器必须实现 PropertyEditor 接口。现在注册到 ConversionServiceFactoryBean 中的自定义转换器必须实现哪些接口,并满足哪些约定呢?我们将在接下来的小节中进行说明。

2. Spring 支持哪些转换器

Spring 在 org.springframework.core.convert.converter 包中定义了 3 种类型的转换器接口,实现任意一个转换器接口都可以作为自定义转换器注册到 ConversionServiceFactoryBean 中。这 3 种类型的转换器接口分别为:

- ❑ Converter<S,T>。
- ❑ GenericConverter。
- ❑ ConverterFactory。

首先需要了解的转换器接口是 Converter 接口,它是 Spring 中最简单的一个转换器接口,仅包括一个接口方法。该接口定义如下:

```
package org.springframework.core.convert.converter;
public interface Converter<S, T> {
    T convert(S source);
}
```

T convert(S source)负责将 S 类型的对象转换为 T 类型的对象。如果希望将一种类型的对象转换为另一种类型及其子类的对象,举例来说,将 String 转换为 Number 及 Number 子类(Integer、Long、Double 等)对象,就需要一系列的 Converter,如 StringToInteger、StringToLong 及 StringToDouble 等。Spring 提供了一个将相同系列多个“同质”Converter 封装在一起的 ConverterFactory 接口,定义如下:

```
package org.springframework.core.convert.converter;
public interface ConverterFactory<S, R> {
    <T extends R> Converter<S, T> getConverter(Class<T> targetType);
}
```

S 为转换的源类型, R 为目标类型的基类,而 T 为扩展于 R 基类的类型。如 Spring 的 StringToNumberConverterFactory 就实现了 ConverterFactory 接口,封装了 String 转换到各种数据类型的 Converter。

Converter 只负责将一个类型的对象转换为另一个类型的对象,并没有考虑类型对象所在宿主类上下文的信息,因此并不能完成“复杂”类型转换工作。GenericConverter 接口会根据源类对象及目标类对象所在宿主类的上下文信息进行类型转换工作,其接口定义如下:

```
package org.springframework.core.convert.converter;
public interface GenericConverter {
    public Set<ConvertiblePair> getConvertibleTypes();
    Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);
}
```

ConvertiblePair 封装了源类型和目标类型,组成一个“对子”,而 TypeDescriptor 包含了需转换类型对象所在宿主类的信息,因此 GenericConverter 的 convert()接口方法可以利用这些上下文信息进行类型转换工作。

ConditionalGenericConverter 扩展于 GenericConverter 接口,并添加了一个接口方法,如下:

```
boolean matches(TypeDescriptor sourceType, TypeDescriptor targetType)
```

该接口方法根据源类型及目标类型所在宿主类的上下文信息决定是否要进行类型转换,只有该接口方法返回 true 时,才调用 convert()方法完成类型转换。正是因为转换之前有一个是否要进行类型转换的条件判断动作,因此该接口命名为 ConditionalGenericConverter,即带“条件”的通用转换器。

ConversionServiceFactoryBean 的 converters 属性可接受 Converter、ConverterFactory、GenericConverter 或 ConditionalGenericConverter 接口的实现类,并把这些转换器的转换逻辑统一封装到一个 ConversionService 实例对象中 (GenericConversionService)。Spring 在 Bean 属性配置及 Spring MVC 请求消息绑定时将利用这个 ConversionService 实例完成类型转换工作。以上过程可通过图 17-9 来理解。

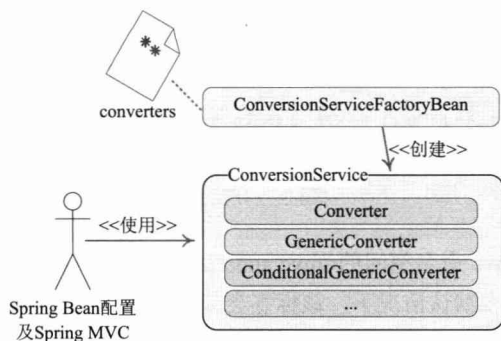


图 17-9 ConversionService 创建过程示意图

3. 在 Spring MVC 中使用 ConversionService

假设处理方法有一个 User 类型的入参,我们希望将一个格式化的请求参数字符串直接转换为 User 对象,该字符串的格式为:

```
<userName>:<password>:<realName>
```

这就要求我们定义一个负责将格式化的 String 转换为 User 对象的自定义转换器,如代码清单 17-30 所示。

代码清单 17-30 StringToUserConverter.java: User 对象转换器

```
package com.smart.domain;
import org.springframework.core.convert.converter.Converter;
public class StringToUserConverter implements Converter<String, User>{
```

```
    public User convert(String source) {
        User user = new User();
        if(source != null){
            String[] items = source.split(":");
            user.setUserName(items[0]);
            user.setPassword(items[1]);
            user.setRealName(items[2]);
```

将<userName>:<password>:<realName>
的格式化串转换为 User 对象

①

```

    }
    return user;
}
}

```

编写好 `StringToUserConverter` 后，接下来将其安装到 Spring 上下文中，如代码清单 17-31 所示。

代码清单 17-31 smart-servlet.xml: 将 `StringToUserConverter` 安装到 Spring 上下文中

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
...
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="...
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd">
...
    <mvc:annotation-driven conversion-service="conversionService"/> ①
    <bean id="conversionService"
        class="org.springframework.context.support.ConversionServiceFactoryBean">
        <property name="converters">
            <list>
                <bean class="com.smart.domain.StringToUserConverter"/> ②
            </list>
        </property>
    </bean>
</beans>

```

装配自定义的 `ConversionService`

装配 `StringToUserConverter`

在①处使用了 `mvc` 命名空间的 `<mvc:annotation-driven/>` 标签，该标签可简化 Spring MVC 的相关配置。在默认情况下，该标签会创建并注册一个默认的 `DefaultAnnotationHandlerMapping` 和一个 `RequestMappingHandlerAdapter` 实例。如果上下文中存在自定义的对应组件 Bean，则 Spring MVC 会自动利用自定义的 Bean 覆盖默认的 Bean。

除此之外，`<mvc:annotation-driven/>` 标签还会注册一个默认的 `ConversionService`，即 `FormattingConversionServiceFactoryBean`（参见 17.3.3 节），以满足大多数类型转换的需求。现在由于要注册一个自定义的 `StringToUserConverter`，因此，需要显式定义一个 `ConversionService` 覆盖 `<mvc:annotation-driven/>` 中的默认实现，这是通过设置 `<mvc:annotation-driven/>` 的 `conversion-service` 属性来完成的。

在装配好 `StringToUserConverter` 后，就可以在任何控制器的处理方法中使用这个转换器了。在 `UserController` 中添加一个使用 `StringToUserConverter` 的 `handle81()` 方法，如代码清单 17-32 所示。

代码清单 17-32 UserController.java: 使用 `StringToUserConverter`

```

@RequestMapping(path = "/handle81")
public String handle81(@RequestParam("user") User user, ModelMap modelMap) {
    modelMap.put("user", user);
    return "/user/showUser";
}

```

启动 Web 容器，发送一个如下 URL 请求：

```
http://localhost:8080/chapter17/user/handle81.html?user=tom:1234:tomson
```

user 请求参数的 tom:1234:tomson 值将会被 StringToUserConverter 正式地转换并绑定到 handle81()方法的 User 入参中。

4. 使用@InitBinder 和 WebBindingInitializer 装配自定义编辑器

Spring MVC 在支持新的转换器框架的同时, 也支持 JavaBeans 的 PropertyEditor。可以在控制器中使用@InitBinder 添加自定义的编辑器, 也可以通过 WebBindingInitializer 装配在全局范围内使用的编辑器, 如代码清单 17-33 所示。

代码清单 17-33 UserController.java: 使用@InitBinder

```
package com.smart.web;
import org.springframework.web.bind.annotation.InitBinder;
import com.smart.domain.UserEditor;

@Controller
@RequestMapping("/user")
public class UserController {

    @InitBinder ①
    public void initBinder(WebDataBinder binder) {
        binder.registerCustomEditor(User.class, new UserEditor()); ②
    }
}
```

在控制器初始化时调用

注册一个自定义的编辑器

另外, Spring 4.0 可以使用 addCustomFormatter 指定格式化程序实现, 这样就不需要实现一个 PropertyEditor 的实例, 如下:

```
@InitBinder
public void initBinder(WebDataBinder binder) {
    binder.addCustomFormatter(new DateFormatter("yyyy-MM-dd"));
}
```

Spring MVC 使用 WebDataBinder 处理请求消息和处理方法入参的绑定工作。在②处通过 registerCustomEditor()方法为 User 注册一个自定义的编辑器, UserEditor 是实现 PropertyEditor 接口的编辑器。

如果希望在全局范围内使用 UserEditor 编辑器, 则可实现 WebBindingInitializer 接口并在该实现类中注册 UserEditor, 如代码清单 17-34 所示。

代码清单 17-34 MyBindingInitializer.java: 注册全局的自定义编辑器

```
package com.smart.web;

import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.support.WebBindingInitializer;
import org.springframework.web.context.request.WebRequest;

import com.smart.domain.User;
import com.smart.domain.UserEditor;

public class MyBindingInitializer implements WebBindingInitializer{
    public void initBinder(WebDataBinder binder, WebRequest request) {
        binder.registerCustomEditor(User.class, new UserEditor()); ①
    }
}
```

上述代码在 `initBinder()` 接口方法中注册了 `UserEditor` 编辑器。接下来还需要在 Spring 上下文中通过 `RequestMappingHandlerAdapter` 装配 `MyBindingInitializer`，如代码清单 17-35 所示。

代码清单 17-35 smart-servlet.xml：将 `MyBindingInitializer` 安装到 Spring 上下文中

```
...
<bean class="org.springframework.web.servlet.mvc
                                .method.annotation.RequestMappingHandlerAdapter">
    <property name="webBindingInitializer">
        <bean class="com.smart.web.MyBindingInitializer" />
    </property>
</bean>
```

对于同一个类型对象来说，如果既在 `ConversionService` 中装配了自定义转换器，又通过 `WebBindingInitializer` 装配了自定义编辑器，同时还在控制器中通过 `@InitBinder` 装配了自定义编辑器，那么 Spring MVC 将按以下优先顺序查找对应类型的编辑器：

- ☐ 查询通过 `@InitBinder` 装配的自定义编辑器。
- ☐ 查询通过 `ConversionService` 装配的自定义转换器。
- ☐ 查询通过 `WebBindingInitializer` 装配的自定义编辑器。

17.3.3 数据格式化

Spring 使用转换器进行源类型对象到目标类型对象的转换，Spring 的转换器并不提供输入及输出信息格式化的工作。如果需要转换的源类型数据（一般是字符串）是从客户端界面中传递过来的，为了方便使用者观看，这些数据往往具有一定的格式。举例来说，像日期、时间、数字、货币等数据都是具有一定格式的，在不同的本地化环境中，同一类型的数据还会相应地呈现不同的显示格式。

如何从格式化的数据中获取真正的数据以完成数据绑定，并将处理完成的数据输出为格式化的数据，是 Spring 格式化框架要解决的问题。Spring 引入了一个新的格式化框架，这个框架位于 `org.springframework.format` 类包中。首先来了解一下格式化框架最重要的 `Formatter<T>` 接口。

1. `Formatter<T>`

`Formatter<T>` 接口扩展于 `Printer<T>` 和 `Parser<T>` 接口。

```
package org.springframework.format;
public interface Formatter<T> extends Printer<T>, Parser<T> {
}
```

`Printer<T>` 负责对象的格式化输出，而 `Parser<T>` 负责对象的格式化输入，在接口中各定义了一个方法。先来看一下 `Printer<T>` 的接口方法。

```
String print(T fieldValue, Locale locale);
```

`print()` 接口方法将类型为 `T` 的成员对象根据本地化的不同输出为不同的格式化的字符串。

Parser<T>的接口方法也很简单。

```
T parse(String clientValue, Locale locale) throws ParseException;
```

parse()接口方法参考本地化信息将一个格式化的字符串转换为 T 类型的对象，即完成格式化对象的输入工作。

Spring 的 org.springframework.format.datetime 包中提供了一个用于时间对象格式化的 DateFormatter 实现类；而 org.springframework.format.number 包中提供了 3 个用于数字对象格式化的实现类。

- ☐ NumberFormatter: 用于数字类型对象的格式化。
- ☐ CurrencyFormatter: 用于货币类型对象的格式化。
- ☐ PercentFormatter: 用于百分数数字类型对象的格式化。

可以手工调用这些 Formatter 接口实现类进行对象数据输入/输出的格式化工作，但是随着 Java 技术的发展，这种硬编码的格式化方式显然已经过时。Spring 提供了注解驱动的属性对象格式化功能，可在 Bean 属性设置、Spring MVC 处理方法入参数据绑定、模型数据输出时自动通过注解应用格式化功能。

2. 注解驱动格式化的重要接口

为了让注解和格式化的属性类型关联起来，Spring 在 Formatter<T>所在的包中还提供了一个 AnnotationFormatterFactory<A extends Annotation>接口。来看一下该接口的几个方法。

- ☐ Set<Class<?>> getFieldTypes(): 注解 A 的应用范围，即哪些属性类可以标注 A 注解。
- ☐ Parser<?> getParser (A annotation, Class<?> fieldType): 根据注解 A 获取特定属性类型的 Parser。
- ☐ Printer<?> getPrinter (A annotation, Class<?> fieldType): 根据注解 A 获取特定属性类型的 Printer。

Spring 提供了两个内建的实现类，分别支持数字及日期类型的注解驱动格式化。

- ☐ NumberFormatAnnotationFormatterFactory: 支持对数字类型的属性使用 @NumberFormat 注解（位于 org.springframework.format.annotation 包中）。
- ☐ JodaDateTimeFormatAnnotationFormatterFactory: 支持对日期类型的属性使用 @DateTimeFormat 注解，该注解和 @NumberFormat 注解位于相同的包中。

现在剩下的问题是：如何在 Spring 上下文中启用这个基于注解驱动的格式化模块，并在实际应用中使用时使用格式化功能？

3. 启用注解驱动格式化功能

聪明的读者已经发现，对属性对象的输入/输出进行格式化，从本质上讲依然属于“类型转换”的范畴。不错！Spring 就是基于对象转换框架植入“格式化”功能的。

Spring 在格式化模块中定义了一个实现 ConversionService 接口的 FormattingConversionService 实现类，该实现类扩展了 GenericConversionService，因此它既具有类型转换功能，

又具有格式化功能。

相对于 `ConversionService` 的 `ConversionServiceFactoryBean` 工厂类, `FormattingConversionService` 也拥有一个对应的 `FormattingConversionServiceFactoryBean` 工厂类, 后者用于在 Spring 上下文中构造一个 `FormattingConversionService`。通过这个工厂类, 既可以注册自定义的转换器, 还可以注册自定义的注解驱动逻辑。

由于 `FormattingConversionServiceFactoryBean` 在内部会自动注册 `NumberFormatAnnotationFormatterFactory` 和 `JodaDateTimeFormatAnnotationFormatterFactory`, 因此, 装配了 `FormattingConversionServiceFactoryBean` 后, 就可以在 Spring MVC 入参绑定及模型数据输出时使用注解驱动的格式化功能。

首先在 Spring 上下文中装配 `FormattingConversionServiceFactoryBean`, 如代码清单 17-36 所示。

代码清单 17-36 smart-servlet.xml: 装配 `FormattingConversionServiceFactoryBean`

```
...
<mvc:annotation-driven conversion-service="conversionService"/>
<bean id="conversionService"
    class="org.springframework.format.support.FormattingConversionServiceFactory
    Bean">① <
    <property name="converters">
        <list>
            <bean class="com.smart.domain.StringToUserConverter"/>
        </list>
    </property>
</bean>
```

替换原来的 `ConversionServiceFactoryBean`

值得注意的是, `<mvc:annotation-driven/>` 标签内部默认创建的 `ConversionService` 实例就是一个 `FormattingConversionServiceFactoryBean`。

装配好 `FormattingConversionServiceFactoryBean` 后, Spring MVC 对处理方法的入参绑定就支持注解驱动功能了。

4. 注解驱动格式化实例

在 `User` 中添加两个新属性, 并标注格式化注解, 如代码清单 17-37 所示。

代码清单 17-37 User.java: 使用注解驱动格式化功能

```
package com.smart.domain;

import java.util.Date;
import org.springframework.format.annotation.DateTimeFormat;
import org.springframework.format.annotation.NumberFormat;
...

public class User {
    ...
    @DateTimeFormat(pattern="yyyy-MM-dd") ①
    private Date birthday;

    @NumberFormat(pattern="#,###.##") ②
    private long salary;
```

可将形如 1980-09-01 的字符串转换到 `Date` 类型的 birthday 属性中

可将形如 4,500.00 的字符串转换到 `long` 类型的 salary 属性中


```
//省略get/setter方法
```

birthday 属性标注了 `@DateTimeFormat`，并指定其格式为“yyyy-MM-dd”；而 salary 属性标注了 `@NumberFormat`，并指定其格式为“#,###.##”。

`@DateTimeFormat` 注解可对 `java.util.Date`、`java.util.Calendar`、`java.lang.Long` 及 Joda 时间类型的属性进行标注。因 Spring 4.0 已全面支持 Java SE 8.0，所以也可以使用 JDK 8.0 的 `java.time` 包。它支持以下几个互斥的属性。

- iso: 类型为 `DateTimeFormat.ISO`。以下是几个常用的可选值。
 - `DateTimeFormat.ISO.DATE`: 格式为 yyyy-MM-dd。
 - `DateTimeFormat.ISO.DATE_TIME`: 格式为 yyyy-MM-dd hh:mm:ss.SSSZ。
 - `DateTimeFormat.ISO.TIME`: 格式为 hh:mm:ss.SSSZ。
 - `DateTimeFormat.ISO.NONE`: 表示不应该使用 ISO 格式的时间。
- pattern: 类型为 `String`，使用自定义的时间格式化串，如 yyyy/mm/dd h:mm:ss。
- style: 类型为 `String`，通过样式指定日期/时间的格式，由两位字符组成，第一位表示日期的样式，第二位表示时间的样式。以下是几个常用的可选值。
 - S: 短日期/时间的样式。
 - M: 中日期/时间的样式。
 - L: 长日期/时间的样式。
 - F: 完整日期/时间的样式。
 - -: 忽略日期或时间的样式。

而 `@NumberFormat` 可对类似于数字类型的属性进行标注，它拥有两个互斥的属性，具体说明如下。

- pattern: 类型为 `String`，使用自定义的数字格式化串，如 “##,###.##”。
- style: 类型为 `NumberFormat.Style`。以下是几个常用的可选值。
 - `NumberFormat.Style.CURRENCY`: 货币类型。
 - `NumberFormat.Style.NUMBER`: 正常数字类型。
 - `NumberFormat.Style.PERCENT`: 百分数类型。

我们在 `UserController` 中新增了一个 `handle82(User user)` 处理方法，在 `UserControllerTest` 中通过如下方法进行测试：

```
@Test
public void testHandle82() {
    RestTemplate restTemplate = new RestTemplate();
    MultiValueMap<String, String> form = new LinkedMultiValueMap<String, String>();
    form.add("userName", "tom");
    form.add("password", "123456");
    form.add("age", "45");
    form.add("birthday", "1980-01-01");
    form.add("salary", "4,500.00");
    String html = restTemplate.postForObject(

```

① 请求参数的值都是格式化的字符串

```
        "http://localhost:8080/chapter17/user/handle82.html", form, String.class);
        Assert.assertNotNull(html);
        Assert.assertTrue(html.indexOf("tom") > -1);
    }
}
```

如果希望在视图页面中将模型属性数据也以格式化的方式进行渲染，则可通过 Spring 的页面标签显示模型数据来达到目的。

17.3.4 数据校验

应用程序在执行业务逻辑前，必须通过数据校验保证接收到的输入数据是正确合法的，如代表生日的日期应该是一个过去的时间、工资的数值必须是一个正数等。一般情况下，应用程序的开发是分层的，不同层的代码由不同的开发人员负责。很多时候，同样的数据验证会出现在不同的层中，这样就会导致代码冗余，违反了 DRY 原则。为了避免这样的情况，最好将验证逻辑和相应的域模型进行绑定，将代码验证的逻辑集中起来管理。

1. JSR-303

JSR-303 是 Java 为 Bean 数据合法性校验所提供的标准框架，它已经包含在 Java EE 6.0 中。JSR-303 通过在 Bean 属性上标注类似于 @NotNull、@Max 等标准的注解指定校验规则，并通过标准的验证接口对 Bean 进行验证。可以通过 <http://jcp.org/en/jsr/detail?id=303> 了解 JSR-303 的详细内容。

JSR-303 定义了一套可标注在成员变量、属性方法上的校验注解，说明如表 17-3 所示。

表 17-3 JSR-303 注解

注 解	功能说明
@Null	被注释的元素必须为 null
@NotNull	被注释的元素必须不为 null
@AssertTrue	被注释的元素必须为 true
@AssertFalse	被注释的元素必须为 false
@Min(value)	被注释的元素必须是一个数字，其值必须大于等于指定的最小值
@Max(value)	被注释的元素必须是一个数字，其值必须小于等于指定的最大值
@DecimalMin(value)	被注释的元素必须是一个数字，其值必须大于等于指定的最小值
@DecimalMax(value)	被注释的元素必须是一个数字，其值必须小于等于指定的最大值
@Size(max, min)	被注释的元素的大小必须在指定的范围内
@Digits(integer, fraction)	被注释的元素必须是一个数字，其值必须在可接受的范围内
@Past	被注释的元素必须是一个过去的日期
@Future	被注释的元素必须是一个将来的日期

Hibernate Validator 是 JSR-303 的一个参考实现，它除了支持所有标准的校验注解外，还支持如表 17-4 所示的扩展注解。

表 17-4 Hibernate Validator扩展注解

注 解	功能说明
@Email	被注释的元素必须是电子邮箱地址
@Length	被注释的字符串的大小必须在指定的范围内
@NotEmpty	被注释的字符串必须非空
@Range	被注释的元素必须在合适的范围内

JSR-303 的核心接口是 javax.validation.Validator，该接口根据目标对象类中所标注的校验注解进行数据校验，并得到校验结果。

2. Spring 校验框架

Spring 拥有自己独立的数据校验框架，同时支持 JSR-303 标准的校验框架。Spring 的 DataBinder 在进行数据绑定时，可同时调用校验框架完成数据校验工作。在 Spring MVC 中，则可直接通过注解驱动的方式进行数据校验。

Spring 的 org.springframework.validation 是校验框架所在的包，下面来了解一下校验框架的几个重要接口和类。

首先要了解的是 Validator，该接口拥有以下两个方法。

- ☐ boolean supports (Class<?> clazz): 该校验器能够对 clazz 类型的对象进行校验。
- ☐ void validate (Object target, Errors errors): 对目标类 target 进行校验，并将校验错误记录在 errors 中。

LocalValidatorFactoryBean 既实现了 Spring 的 Validator 接口，又实现了 JSR-303 的 Validator 接口。只要在 Spring 容器中定义了一个 LocalValidatorFactoryBean，即可将其注入需要数据校验的 Bean 中。定义 LocalValidatorFactoryBean 的 Bean 非常简单，代码如下：

```
<bean id="validator"
    class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"/>
```

值得注意的是，Spring 本身没有提供 JSR-303 的实现，所以必须将 JSR-303 的实现者（如 Hibernate Validator）的 JAR 文件放到类路径下，Spring 将自动加载并装配好 JSR-303 的实现者。

3. Spring MVC 数据校验

<mvc:annotation-driven/>会默认装配一个 LocalValidatorFactoryBean，通过在处理方法的入参上标注 @Valid 注解，即可让 Spring MVC 在完成数据绑定后执行数据校验工作。

下面对代码清单 17-37 进行更改，添加 JSR-303 的校验注解，如代码清单 17-38 所示。

代码清单 17-38 User.java：添加校验注解

```
package com.smart.domain;
...

```

```

import javax.validation.constraints.*;

import org.springframework.format.annotation.DateTimeFormat;
import org.springframework.format.annotation.NumberFormat;

public class User {

    private String userId; // 通过正则表达式进行校验，匹配 4~30 个
                           // 包含数字、字母及下划线的字符

    @Pattern(regexp="w{4,30}") ①
    private String userName; // 通过正则表达式进行校验，匹配 6~30 个
                             // 非空白的字符

    @Pattern(regexp="S{6,30}") ②
    private String password;

    @Length(min=2,max=100) ③ // 这是 Hibernate Validator 的扩展度，
                             // 将属性值的长度限制在 2~100 之间
    private String realName;

    @Past ④ // 时间值必须是一个过去的时间
    @DateTimeFormat(pattern="yyyy-MM-dd")
    private Date birthday;

    @DecimalMin(value="1000.00")
    @DecimalMax(value="100000.00") ⑤ // 数据必须在 1000.00~100000.00 之间
    @NumberFormat(pattern="#,###.##")
    private long salary;
}

```

在 User 类的属性上标注校验注解后，接下来的问题是如何在 Spring MVC 中使用注解所声明的限制规则进行数据校验。我们在 UserController 处理器中添加一个处理方法进行演示，如代码清单 17-39 所示。

代码清单 17-39 UserController.java: 对处理方法的入参数据进行校验

```

package com.smart.web;
import javax.validation.Valid;
import org.springframework.validation.BindingResult;
...

@Controller
@RequestMapping("/user")
public class UserController {

    ... // 在入参对象前添加@Valid 注解，同时在其后
        // 声明一个BindingResult 入参

    @RequestMapping(path = "/handle91")
    public String handle91(@Valid @ModelAttribute("user") User user, ①
                           BindingResult bindingResult) {

        if(bindingResult.hasErrors()) { ②
            return "/user/register3";
        } else {
            return "/user/showUser";
        }
    }
}

```

根据 BindingResult 即可判断是否存在错误，如果有错误则转到注册页面

在已经标注了 JSR-303 注解的表单/命令对象前标注一个@Valid，Spring MVC 框架在将请求数据绑定到该入参对象后，就会调用校验框架根据注解声明的校验规则实施校

验。这里的关键问题是：校验所产生的校验结果保存在什么地方？又如何传递给请求处理方法？Spring MVC 是通过处理方法的签名来保存校验结果的：前一个表单/命令对象的校验结果保存在其后的入参中，这个保存校验结果的入参必须为 `BindingResult` 或 `Errors` 类型，这两个类都位于 `org.springframework.validation` 包中。其签名约定可通过图 17-10 来说明。

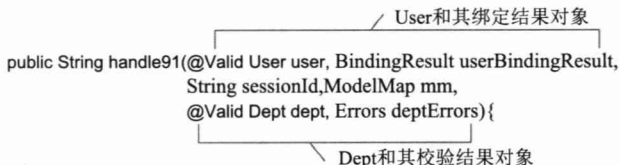


图 17-10 数据校验时处理方法入参的签名规约

概括来说，需校验的表单/命令对象和其绑定结果对象或错误对象是成对出现的，它们之间不允许声明其他入参。

`Errors` 接口提供了获取错误信息的方法，如 `getErrorCount()` 方法获取错误的数量，`getFieldErrors(String field)` 方法得到成员属性的校验错误列表。而 `BindingResult` 接口扩展了 `Errors` 接口，以便可以使用 Spring 的 `org.springframework.validation.Validator` 对数据进行校验，同时获取数据绑定结果对象的信息。`BindingResult` 接口通过 `DataBinder.getBindingResult()` 方法获取。

4. 如何获取校验结果

在表单/命令对象类的属性中标注校验注解，在处理方法对应的入参前添加 `@Valid`，Spring MVC 就会实施校验并将校验结果保存在被校验入参对象之后的 `BindingResult` 或 `Errors` 入参中。

在处理方法内部可以通过 `BindingResult` 或 `Errors` 入参对象获取错误信息，如代码清单 17-36 中的②处所示，这样就可以通过 `bindingResult.hasErrors()` 方法判断 `User` 入参对象是否存在校验错误。还可以通过接口方法获取更多的信息，以下是几个常用的方法。

- ❑ `FieldError getFieldError(String field)`：根据属性名获取对应的校验错误。
- ❑ `List<FieldError> getFieldErrors()`：获取所有的属性校验错误。
- ❑ `Object getFieldValue(String field)`：获取属性值。
- ❑ `int getErrorCount()`：获取错误数量。

5. 如何在页面中显示错误

由于表单/命令对象所对应的请求一般是从客户端的网页中传送过来的，因此，如果发生了错误，则必须通过网页显示错误，提示用户更正错误。

Spring MVC 除了将表单/命令对象的校验结果保存到对应的 `BindingResult` 或 `Errors` 对象中外，还将所有的校验结果保存到“隐含模型”中（参见 17.2.5 节）。也就是说，即使处理方法的签名中没有对应表单/命令对象的校验结果入参，校验结果也不会丢失，它们始终可以从“隐含模型”对象中获取。校验结果对象和被校验的表单/命令对象是一

对一的关系，但“隐含模型”除了存储模型数据外，还保存了所有被校验的表单/命令对象的校验结果。隐含模型中的所有数据最终将通过 `HttpServletRequest` 的属性列表暴露给 JSP 视图对象，因此我们可以很容易地在 JSP 中获取校验错误信息。

抛开 Spring MVC 的所有组件，单从输入/输出的层面分析 Spring MVC 框架的数据流，如图 17-11 所示。

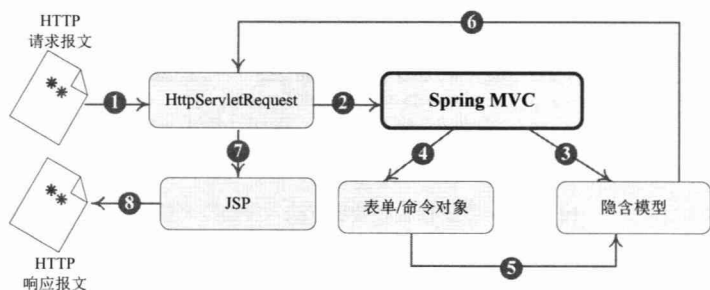


图 17-11 Spring MVC 的数据流图

- ① HTTP 请求报文到达 Web 服务器，Web 服务器将其封装成一个 `HttpServletRequest` 对象。
- ② Spring MVC 框架截获这个 `HttpServletRequest` 对象。
- ③ Spring MVC 创建一个隐含模型对象，作为处理本次请求的上下文数据存放处。
- ④ Spring MVC 将 `HttpServletRequest` 对象数据绑定到处理方法的入参对象中（表单/命令对象）。
- ⑤ 将绑定错误信息、检验错误信息都保存到隐含模型中。
- ⑥ 将本次请求的对应隐含模型数据存放到 `HttpServletRequest` 属性列表中，暴露给视图对象。
- ⑦ 视图对象对已经存放在 `HttpServletRequest` 属性列表中的模型数据进行渲染。
- ⑧ 将渲染后的 HTTP 响应报文发送给客户端。

理解了 Spring MVC 的整体处理过程后，可以肯定的是，校验错误信息一定包含在隐含模型中，且会通过 `HttpServletRequest` 暴露给视图对象。

可通过 Spring 的 `<form:errors path="propName"/>` 标签在 JSP 页面中显示错误信息，如代码清单 17-40 所示。

代码清单 17-40 register3.jsp: 显示错误信息

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>①
<html>
<head>
<title>注册用户</title>
<style>.errorClass{color:red}</style>
</head>
```

引入 Spring 表单标签


```

<body>
<form modelAttribute="user" action="/chapter17/user/handle91.html">
  <form:errors path="*" />②
  <table>
    <tr>
      <td>用户名: </td>
      <td>
        <form:errors path="userName" cssClass="errorClass" />③
        <form:input path="userName" />
      </td>
    </tr>
    <tr>
      <td>密码: </td>
      <td>
        <form:errors path="password" cssClass="errorClass" />④
        <form:input path="password" />
      </td>
    </tr>
    ...
    <tr>
      <td colspan="2"><input type="submit" name="提交" /></td>
    </tr>
  </table>
</form>
</body>
</html>

```

显示表单对象所有的错误信息

显示 userName 属性的错误信息

显示 password 属性的错误信息

在②处，当检测到存在错误信息时，转到 register3.jsp 视图页面。register3.jsp 在①处引入了 Spring 的标签库，可以通过如②处所示的方式显示所有的错误信息，也可以通过如③或④处所示的方式显示特定属性的错误信息。显示错误信息的页面如图 17-12 所示。

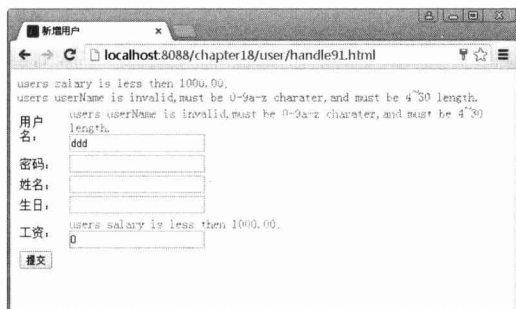


图 17-12 显示错误信息的页面

从图 17-12 中可以看出，Spring MVC 已经对表单的数据进行了合法性校验，并通过 `<form:errors path="*" />` 标签在页面顶部显示所有的校验错误信息，而通过 `<form:errors path="propertyName" />` 标签可显示某个属性的校验错误信息。

通过观察页面上的错误信息，可以发现这些错误信息都比较生硬和机械。实际上，这些校验错误信息都是校验框架根据校验规则自动产生的。在实际应用中，不但希望错误信息更加人性化、更具可读性，还希望可显示本地化的错误信息。Spring MVC 支持对错误信息进行本地化显示。

6. 通过国际化资源显示错误信息

每个属性在数据绑定和数据校验发生错误时，都会生成一个对应的 `FieldError` 对象。`FieldError` 对象实现了 `org.springframework.context.MessageSourceResolvable` 接口，`MessageSourceResolvable` 顾名思义是可用国际化资源进行解析的对象。`MessageSourceResolvable` 拥有 3 个接口方法。

- ❑ `Object[] getArguments()`: 返回一组参数对象。
- ❑ `String[] getCodes()`: 返回一组消息代码，每个代码对应一个资源属性。可以使用 `getArguments()` 方法返回的参数对资源属性值进行参数替换。
- ❑ `String getDefaultMessage()`: 默认的消息。由于我们没有装配相应的国际化资源，因此图 17-12 中显示的错误信息都是默认的。

当一个属性校验失败后，校验框架会为该属性生成 4 个消息代码，这些消息代码以校验注解类名为前缀，结合类名、属性名及属性类型名生成多个对应的消息代码。

如 `User` 类的 `password` 属性标注了一个 `@Pattern` 注解，当该属性值不满足 `@Pattern` 所定义的限制规则时，就会产生以下 4 个错误代码。

- ❑ `Pattern.user.password`: 根据类名、属性名产生的错误代码。
- ❑ `Pattern.password`: 根据属性名产生的错误代码。
- ❑ `Pattern.java.lang.String`: 根据属性类型产生的错误代码。
- ❑ `Pattern`: 根据验证注解名产生的错误代码。

当使用 `Spring MVC` 标签显示错误信息时，`Spring MVC` 会查看 `Web` 上下文是否装配了对应的国际化消息。如果没有，则显示默认的错误信息；否则使用国际化消息对错误代码进行翻译。

值得注意的是，如果在数据类型转换或数据格式转换时发生错误，或者该有的参数不存在，或调用处理方法时发生错误，则都会在隐含模型中创建错误信息。其错误代码前缀说明如下。

- ❑ `required`: 必要的参数不存在，如 `@RequestParam("param1")` 标注了一个入参，但是请求参数不存在 `param1` 参数。
- ❑ `typeMismatch`: 在数据绑定时，发生数据类型不匹配的问题。
- ❑ `methodInvocation`: `Spring MVC` 在调用处理方法时发生了错误。

举例来说，如果将“aaa”的非数值参数传递给 `User` 对象的 `salary` 属性，则将发生数据转换错误，`Spring` 将为该错误生成以下错误代码：

- ❑ `typeMismatch.user.salary`。
- ❑ `typeMismatch.salary`。
- ❑ `typeMismatch.long`。
- ❑ `typeMismatch`。

错误代码的产生逻辑可以通过 `DataBinder` 的 `setMessageCodesResolver()` 方法注入一个自定义的 `MessageCodesResolver` 对象来调整，不过一般情况下采用默认的产生器就可以了。

在知道错误对象的错误代码是对应国际化消息的键名后，接下来的工作就非常简单了：定义一个国际化资源，在国际化资源文件中为错误代码定义相应的本地化消息。

我们在 `i18n` 下添加基名为 `messages` 的国际化资源，一个是默认的 `messages.properties`，另一个是对应中国大陆的 `messages_zh_CN.properties`。来看一下 `messages_zh_CN.properties` 资源文件的内容，如图 17-13 所示。

name	value
Pattern.user.userName	用户名不正确，必须是4~30个英文及_的字符
Pattern.user.password	密码不正确，必须是6~30个字符，不允许空格
Length.user.realName	姓名不合法，长度必须是2~100个字符
Past.user.birthday	日期格式不正确，必须是一个过去的日期
DecimalMin.user.salary	工资必须在1000~100000之间
DecimalMax.user.salary	\${DecimalMin.user.salary}

图 17-13 `messages_zh_CN.properties` (使用 MyEclipse Properties Editor 查看)

接下来的工作是在 `smart-servlet.xml` 中装配好这个国际化资源。

```
<bean id="messageSource"
      class="org.springframework.context.support.ResourceBundleMessageSource"
      p:basename="i18n/messages"/>
```

可以通过 `ResourceBundleMessageSource` 的 `basename` 属性指定一个国际化资源，还可以通过 `basenames` 属性指定多个国际化资源。重新启动 Web 容器，当发生错误时，`register3.jsp` 所显示的错误页面如图 17-14 所示。

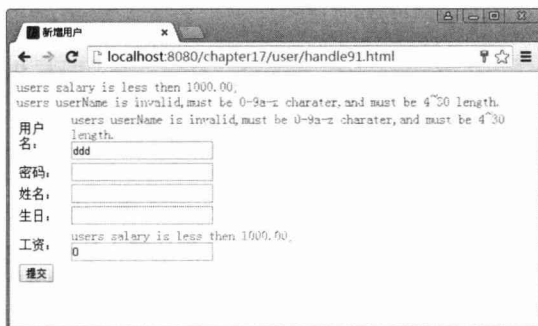


图 17-14 使用国际化资源显示错误信息

7. 自定义校验规则

JSR-303 允许定义自定义校验注解及对应的校验器类，Spring 的校验框架完全支持 JSR-303 的自定义扩展注解。关于 JSR-303 自定义校验注解的内容已经超出本书的范畴，感兴趣的读者可以阅读相关材料。

这里要介绍的是如何给请求处理类装配一个自定义的 `Validator`，或者直接在处理方法中使用自定义的 `Validator` 对入参进行校验。可以在 `UserController` 中标注 `@InitBinder` 注解的 `initBinder()` 方法中装配自定义的 `MyValidator`。

```
@InitBinder
public void initBinder(WebDataBinder binder) {
    ...
    binder.setValidator(new UserValidator()); // 在进行数据绑定时使用的校验器
}
```

当然，还可以借助请求处理方法的签名传递一个 `Errors` 或 `BindingResult` 对象进来，然后在处理方法中直接校验，如代码清单 17-41 所示。

代码清单 17-41 UserController.java：在处理方法中通过代码进行校验

```
package com.smart.web;

import org.springframework.validation.ValidationUtils;
import org.springframework.validation.BindingResult;

@Controller
@RequestMapping("/user")
public class UserController {

    @RequestMapping(path = "/handle92")
    public String handle92(@ModelAttribute("user")User user,
                           BindingResult bindingResult){ ①
        ValidationUtils.rejectIfEmptyOrWhitespace(bindingResult, "userName",
"required");②
        if("aaaa".equalsIgnoreCase(user.getUserName())){
            bindingResult.rejectValue("userName", "reserved");③
        }
        if(bindingResult.hasErrors()){
            return "/user/register4";
        }else{
            return "/user/showUser";
        }
    }
}
```

声明一个 `BindingResult` 入参

使用校验器工具类进行校验

手工生成一条错误信息

在①处，处理方法签名声明了一个 `BindingResult` 入参，Spring MVC 框架将把对应本处理方法的 `BindingResult` 对象传递给该入参，以便处理方法借由这个入参访问绑定结果对象。

`ValidationUtils` 提供了空值验证的便捷方法，如②处所示，如果对应属性值为空或空白字符，则将生成一条错误信息。而在③处，通过手工判断 `userName` 是否等于“aaaa”，如果等于“aaaa”，则将产生一条错误信息。

在②处产生的错误信息对应的错误代码包括：

- ☐ `required.user.userName`。
- ☐ `required.userName`。
- ☐ `required.java.lang.String`。
- ☐ `required`。

相应的，在③处产生的错误信息对应的错误代码包括：

- ☐ `reserved.user.userName`。
- ☐ `reserved.userName`。

☐ reserved.java.lang.String。

☐ reserved。

可以从错误代码组中任选一个错误代码，并在国际化资源文件中定义错误代码对应的信息内容。这样，当发生错误时，就会显示相应的本地化错误信息。



提示

通过 binder.setValidator() 方法设置自定义的 Validator 后，Spring MVC 将使用它对入参对象进行校验，而不再使用 Spring MVC 框架装配的 Validator 对入参对象进行校验。换句话说，即使在入参上标注了 @Valid 注解，也不会再根据入参对象类中的 JSR-303 注解进行校验。

17.4 视图和视图解析器

请求处理方法执行完成后，最终返回一个 ModelAndView 对象。对于那些返回 String、View 或 ModelMap 等类型的处理方法，Spring MVC 也会在内部将它们装配成一个 ModelAndView 对象，该对象包含了视图逻辑名和模型对象的信息。

Spring MVC 借助视图解析器（ViewResolver）得到最终的视图对象（View），这可能就是我们常见的 JSP 视图，也可能是一个基于 FreeMarker、Velocity 模板技术的视图，还可能是 PDF、Excel、XML、JSON 等各种形式的视图。

对于最终究竟采取何种视图对象对模型数据进行渲染，处理器并不关心，处理器的工作重点聚焦在生产模型数据的工作上，从而实现 MVC 的充分解耦。

17.4.1 认识视图

视图的作用是渲染模型数据，将模型里的数据以某种形式呈现给客户。视图对象可以是常见的 JSP，还可以是 Excel 或 PDF 等形式不一的媒体形式。为了实现视图模型和具体实现技术的解耦，Spring 在 org.springframework.web.servlet 包中定义了一个高度抽象的 View 接口，该接口中定义了两个方法。

- ☐ String getContentType(): 视图对应的 MIME 类型，如 text/html、image/jpeg 等。
- ☐ void render(Map model, HttpServletRequest request, HttpServletResponse response): 将模型数据以某种 MIME 类型渲染出来。

视图对象是一个 Bean，通常情况下，视图对象由视图解析器负责实例化。由于视图 Bean 是无状态的，所以它们不会有线程安全的问题。

不同类型的视图实现技术对应不同的 View 实现类，这些视图实现类都位于 org.springframework.web.servlet.view 包中，通过表 17-5 来说明。

表 17-5 不同的视图实现类

大 类	视图类型	说 明
URL 资源视图	InternalResourceView	将 JSP 或其他资源封装成一个视图，这是 InternalResourceViewResolver 默认使用的视图实现类
	JstlView	如果 JSP 文件中使用了 JSTL 国际化标签的功能，则需要使用该视图类，而非 InternalResourceView 视图类
XSTL 视图	XsltView	XSTL 驱动的视图
Tiles 视图	TilesView	基于 Tiles 页面布局的视图
	TilesJstlView	如果 Tiles 模板的 JSP 组成文件使用了 JSTL，则需要使用该视图替换 TilesView 视图
文档视图	AbstractExcelView	Excel 文档视图的抽象类，开发者可以通过该抽象类实现自己的 Excel 文档视图。该视图实现类基于 POI 构造 Excel 文档
	AbstractJExcelView	和 AbstractExcelView 类似，只不过它基于 JExcelAPI
	AbstractPdfStamperView	PDF 文档视图的抽象类，通过 AcroForm 技术对 PDF 文档进行操作
	AbstractPdfView	PDF 文档视图的抽象类，可以通过该抽象类实现自己的 PDF 文档视图。该视图实现类基于 iText 构造 PDF 文档
模板视图	FreeMarkerView	使用 FreeMarker 模板引擎的视图
	VelocityLayoutView	几个使用 Velocity 模板引擎的视图
	VelocityToolboxView	
	VelocityView	
报表视图	ConfigurableJasperReportsView JasperReportsCsvView JasperReportsHtmlView JasperReportsMultiFormatView JasperReportsPdfView JasperReportsXlsView	几个使用 JasperReports 报表技术的视图
XML 和 JSON 视图	MarshallingView	通过 oxm 的 Marshaller 技术将模型数据以 XML 方式输出
	MappingJackson2JsonView	将模型数据通过 Jackson 开源框架的 ObjectMapper 以 JSON 方式输出
其他视图	RedirectView	进行重定向的视图，可以重定向到上下文的绝对路径或相对路径下，也可以重定向到当前请求的相对路径下

17.4.2 认识视图解析器

Spring MVC 为逻辑视图名的解析提供了不同的策略，可以在 Spring Web 上下文中配置一种或多种解析策略，并指定它们之间的先后顺序。每种解析策略对应一个具体的视图解析器实现类。视图解析器的工作比较单一，即将逻辑视图名解析为一个具体的视图对象。所有视图解析器都实现了 ViewResolver 接口，该接口仅有一个方法。

```
View resolveViewName(String viewName, Locale locale)

resolveViewName()方法的签名清楚地向我们传达了视图解析器工作的内涵：根据逻
```

辑视图名和本地化对象得到一个视图对象。Spring 拥有众多的视图解析器实现类，通过表 17-6 进行概括性说明。

表 17-6 视图解析器实现类

大 类	视图解析器类型	说 明
解析为 Bean 名字	BeanNameViewResolver	将逻辑视图名解析为一个 Bean，Bean 的 id 等于逻辑视图名
	XmlViewResolver	和 BeanNameViewResolver 类似，只不过目标视图 Bean 对象定义在一个独立的 XML 文件中，而非定义在 DispatcherServlet 上下文的主配置文件中
国际化解析	ResourceBundleViewResolver	在国际化资源文件中定义视图实现类及相关的信息。使用该视图解析器可以为不同的本地化类型提供不同的解析结果
解析为 URL 文件	InternalResourceViewResolver	将视图名解析为一个 URL 文件，一般使用该解析器将视图名映射为保存在 WEB-INF 目录中的程序文件（如 JSP）
	XsltViewResolver	将视图名解析为一个指定 XSLT 样式表的 URL 文件
	JasperReportsViewResolver	JasperReports 是一个基于 Java 的开源报表工具，该解析器将视图名解析为报表文件所对应的 URL
模板文件视图	FreeMarkerViewResolver	解析为基于 FreeMarker 模板技术的模板文件
	VelocityViewResolver	解析为基于 Velocity 模板技术的模板文件
	VelocityLayoutViewResolver	
内容协商	ContentNegotiatingViewResolver	它不负责具体的视图解析，而是作为一个中间人的角色根据请求所要求的 MIME 类型，从上下文中选择一个适合的视图解析器，再将视图解析工作委托其负责

用户可以选择一种视图解析器或混用多种视图解析器，每个视图解析器都实现了 Ordered 接口并开放出一个 orderNo 属性，可以通过该属性指定解析器的优先顺序，值越小优先级越高。有些视图解析器默认为最高优先级（如 ContentNegotiatingViewResolver），而有些视图解析器默认为最低优先级（如 InternalResourceViewResolver、XsltViewResolver 等），具体请参考 API 文档。

Spring MVC 会按照视图解析器的优先级顺序对逻辑视图名进行解析，直到解析成功并返回视图对象，否则将抛出 ServletException 异常。

17.4.3 JSP 和 JSTL

1. 使用 InternalResourceViewResolver

JSP 是最常见的视图技术，在 17.1.3 节中就使用了 InternalResourceViewResolver 作为视图解析器，其配置如下：

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"
      p:prefix="/WEB-INF/views/" p:suffix=".jsp"/>
```

代码清单 17-4 中的 createUser()处理方法返回名为 user/createSuccess 的逻辑视图名，InternalResourceViewResolver 负责对此进行解析，将得到如图 17-15 所示的解析结果。



图 17-15 逻辑视图名解析为 URI 资源

InternalResourceViewResolver 默认使用 InternalResourceView 作为视图实现类。如果 JSP 文件使用了 JSTL 的国际化功能,确切地说,当 JSP 页面使用 JSTL 的<fmt:message/> 标签时,用户需要使用 JstlView 替换默认的视图实现类,如下:

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"
      p:viewClass="org.springframework.web.servlet.view.JstlView"
      p:prefix="/WEB-INF/views/" p:suffix=".jsp"/>
```

下面通过一个例子演示使用 JSTL 的<fmt:message/>标签让 JSP 页面实现本地化输出。首先在 UserController 中添加一个方法。

```
@RequestMapping(path = "/showUserList")
public String showUserList(ModelMap mm){
    Calendar calendar = new GregorianCalendar();
    List<User> userList = new ArrayList<User>();
    User user1 = new User();
    user1.setUserName("tom");
    user1.setRealName("汤姆");
    calendar.set(1980, 1, 1);
    user1.setBirthDay(calendar.getTime());
    User user2 = new User();
    user2.setUserName("john");
    user2.setRealName("约翰");
    user2.setBirthDay(calendar.getTime());
    userList.add(user1);
    userList.add(user2);
    mm.addAttribute("userList",userList);
    return "user/userList";
}
```

userList.jsp 的代码如代码清单 17-42 所示。

代码清单 17-42 userList.jsp: 通过JSTL的<fmt:message>标签使用国际化资源

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
<html>
    <head>
        <title><fmt:message key="website.title"/></title>
    </head>
    <body>
        <fmt:message key="user.userList.title"/>
        <table>
            <c:forEach items="${userList}" var="user">
```

① ← 显示本地化的信息

```

        <tr>
            <td>
                <a href="<c:url value="/user/showUser/${user.userName}.
html"/>">
                    ${user.userName}
                </a>
            </td>
            <td>
                ${user.realName}
            </td>
            <td>
                <fmt:formatDate value="${topic.createDate}" pattern=
"yyyy-MM-dd"/>
            </td>
        </tr>
    </c:forEach>
</table>
</body>
</html>

```

②

使用 JSTL 标签

userList.jsp 所使用的国际化资源在 content 资源文件中定义，更改 smart-servlet.xml 中 ResourceBundleMessageSource 的配置，添加此国际化资源。

```

<bean id="messageSource"
    class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basenames">
        <list>
            <value>i18n/messages</value>
            <value>i18n/content</value>
        </list>
    </property>
</bean>

```

这样，userList.jsp 就可以根据客户端的不同显示相应的本地化页面。除了可以使用 JSTL 的标签外，Spring 也提供了一个轻量级的标签库，可以在 JSP 文件中使用这些标签。

2. 使用 Spring 表单标签

通过 Spring 表单标签，可以很容易地将模型数据中的表单/命令对象绑定到 HTML 表单元素中。在本章前面的内容中，我们已经使用了一些 Spring 表单标签，本节将对 Spring 表单标签进行全面的介绍，首先从<form:form>标签开始。

和使用任何 JSP 扩展标签一样，在使用 Spring 表单标签之前，用户必须在 JSP 页面中添加一行引用 Spring 表单标签的声明。

```

<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%> ①
<html>
    ... ②
</html>

```

引用表单标签的声明

声明后，在页面中就可以使用 Spring 表单标签了

下面是一个使用<form:form>表单标签的实例，它将最终生成一个 HTML 的 form 表单。

```

<form:form modelAttribute="user">
    用户名: <form:input path="userName" /> <br>

```

```
密 码: <form:password path="password" /><br>
Email: <form:input path="email" /><br>
<input type="submit" value="注册" name="testSubmit"/>
<input type="reset" value="重置" />
</form:form>
```

一般情况下，通过 GET 请求获取表单页面，通过 POST 请求提交表单页面，因此，获取表单页面和提交表单页面的 URL 地址是相同的。只要满足这条最佳实践的契约，`<form:form>` 标签就无须通过 `action` 属性指定表单提交的目标 URL。

可以通过 `modelAttribute` 指定绑定的模型属性。如果 `modelAttribute` 属性不指定，那么默认从模型中尝试获取名为 `command` 的表单对象。如果不存在此表单对象，则将发生错误。通过 `action` 指定处理表单提交的 URL 地址。此外，`<form:form>` 标签还拥有多个可设置的属性，这些属性大都是 HTML 表单标签属性的镜像，如 `id`、`onclick`、`ondblclick`、`tabindex` 等。需要注意的是，这些属性都是小写的，而对应 HTML 标签的属性名则没有这个限制。

Spring 提供了十多个表单组件标签，如 `<form:input/>`、`<form:select/>` 等，用以绑定表单对象的属性值。这些标签大都拥有以下属性。

- ☐ `path`: 用属性路径表示的表单对象属性，如 `userName`、`dept.deptName`、`dept.address`、`telephone`、`userName` 等。
- ☐ `htmlEscape`: 绑定的表单属性值是否要对 HTML 特殊字符进行转换，默认值为 `true`。
- ☐ `cssClass`: 表单组件对应的 CSS 样式类名。
- ☐ `cssErrorClass`: 当表单组件的数据存在相应的错误时（提交表单后由服务器端产生）采用的 CSS 样式类。
- ☐ `cssStyle`: 表单组件对应的 CSS 样式串。

此外，表单组件标签也拥有 HTML 镜像标签的各种属性，如 `id`、`onclick`、`ondblclick`、`tabindex` 等。下面通过表 17-7 对 Spring 表单标签进行说明。

表 17-7 Spring 表单标签

Spring 表单标签	说 明
<code><form:input/></code>	输入框组件标签，如 <code><form:input path="userName"/></code>
<code><form:password/></code>	密码框组件标签，如 <code><form:password path="password" /></code>
<code><form:hidden/></code>	隐藏框组件标签，如 <code><form:hidden path="userId" /></code>
<code><form:textarea/></code>	多行输入框组件标签，如 <code><form:textarea path="notes" rows="3" cols="20" /></code>
<code><form:radiobutton/></code>	单选框组件标签，当表单对象对应的属性值和 <code>value</code> 值相等时，单选框选中。下面是一个代表性别的单选框： <code><form:radiobutton path="sex" value="0"/>男</code> <code><form:radiobutton path="sex" value="1"/>女</code> 当表单对象的 <code>sex</code> 属性为 0 时，第一个单选框选中
<code><form:radiobuttons/></code>	单选框组件组标签，用于构造多个单选框，例如： <code><form:radiobuttons path="sex" items="\${sexOptions}"</code> <code>cssClass="radioCls" delimiter="
" /></code>

续表

Spring 表单标签	说 明
<code><form:radiobuttons/></code>	<code>sexOptions</code> 可以是一个 <code>List</code>、<code>String[]</code> 或 <code>Map</code>，该标签将产生和此变量元素相同的单选框组件。如果 <code>sexOptions</code> 为 <code>List</code> 类型，则单选框的 <code>value</code> 和 <code>lable</code> 相同；如果 <code>sexOptions</code> 为 <code>Map</code> 类型，则默认情况下 <code>value</code> 为 <code>Map</code> 的 <code>key</code>，而 <code>lable</code> 为 <code>Map</code> 的 <code>value</code>。可以通过标签的 <code>itemValue</code> 及 <code>itemLabel</code> 属性更改这一默认行为： <pre><form:radiobuttons path="sex" items="{sexOptions}" itemValue="value" itemLabel="key"/></pre> <code>key</code> 和 <code>value</code> 分别代表 <code>Map</code> 的键和值。<code><form:checkboxes/></code> 及 <code><form:select/></code> 标签也拥有相似的功能。 当单选框的 <code>path</code> 属性值和 <code>sexOptions</code> 中的某个相同时，单选框选中。多个单选框通过 <code>delimiter</code> 指定分隔符
<code><form:checkbox/></code>	复选框组件标签，如 <code><form:checkbox path="favorites" value="1"/></code>
<code><form:checkboxes/></code>	复选框组件组标签，用于构造多个复选框，例如： <pre><form:checkboxes path="preferences.interests" items="{interests}" cssClass="note" delimiter="
"/></pre> 该标签的大部分特性和 <code><form:radiobuttons/></code> 类似
<code><form:select/></code>	下拉框组件标签，例如： <pre><form:select path="city" items="{cities}"/></pre> 该标签的大部分特性和 <code><form:radiobuttons /></code> 类似
<code><form:option/></code>	下拉框选项组件标签，例如： <pre><form:select path="city"> <form:option value="" label="--请选择--"/> <form:options items="{cityMap}" itemValue="key" itemLabel="value"/> </form:select></pre> 该标签的大部分特性和 <code><form:radiobuttons /></code> 类似
<code><form:errors/></code>	显示表单数据转换或数据校验所对应的错误，如 <code><form:errors path="*" /></code> 显示表单所有的错误，<code><form:errors path="user*" /></code> 显示所有以 <code>user</code> 为前缀的属性对应的错误，而 <code><form:errors path="userName" /></code> 显示特定表单对象属性的错误。可以通过 <code>cssClass</code> 属性指定错误的 CSS 样式类，如 <code><form:errors path="userName" cssClass="errorClass"/></code>

3. 关于复选框、单选框及下拉框和表单对象属性的映射问题

对于以下复选框标签：

```
<form:checkboxes path="preferences.interests" items="{interests}"
其生成的对应 HTML 如下：
<input id="favorites1" name="favorites" type="checkbox" value="1" checked="checked"/>
<input id="favorites2" name="favorites" type="checkbox" value="2" />
<input id="favorites3" name="favorites" type="checkbox" value="3" checked="checked"/>
<input id="favorites4" name="favorites" type="checkbox" value="4"/>
<input type="hidden" value="1" name="_favorites"/> literature
```

读者可能已经注意到复选框组件的后面附加了一个 `hidden` 组件，如粗体部分所示。这是因为当 HTML 页面中的所有复选框都没有勾选时，表单提交所对应的 HTTP 请求报文不会包含该复选框的参数名，这给 Spring 的表单数据绑定机制带来了麻烦，因为无法触发 `setFavorites()` 方法的调用（如果这个表单对象已经缓存在 Session 中，且该属性已经有值，那么这个属性值将不会被设置为空）。

解决方法就是在每个复选框后添加一个隐藏组件，并且在对应的复选框名字前添加

一个下画线作为隐藏组件的名字。这样一来，相当于告诉 Spring MVC：在这个表单中存在这样一个复选框，Spring MVC 可以据此保证服务器端的表单对象和页面中的表单组件数据的一致性。

复选框、单选框及下拉框的组件标签都拥有类似的功能。如果手工编写 HTML 代码，在碰到单选框、复选框及下拉框时，也应该采用类似的方式，以保证 Spring MVC 数据绑定机制正常工作。当然，如果表单对象没有缓存在 Session 中，而是每次提交都重新创建，则不存在这个问题，也就无须添加一个带下画线的隐藏组件。

17.4.4 模板视图

FreeMarker 和 Velocity 是除 JSP 外被使用最多的页面模板技术。页面模板编写好页面结构，模板页面中使用一些特殊的变量标识符绑定 Java 对象的动态数据。

Spring 对 FreeMarker 和 Velocity 都提供了支持，用户可以使用它们渲染模型中的数据。Velocity 视图的使用过程和 FreeMarker 大同小异，所以我们主要以 FreeMarker 为主进行学习。

FreeMarker 是一个模板引擎，是一个基于模板生成文本输出的通用工具，FreeMarker 可以基于模板产生 HTML、XML、Java 源代码等多种类型的输出内容。用户可以使用 FreeMarker 生成基于模板的一段字符内容、直接输出为一个文件，或作为结果在程序中使用。虽然 FreeMarker 具有一些编程能力，但通常由 Java 程序准备模型数据，FreeMarker 仅负责基于模板对模型数据进行渲染的工作。

1. 编写一个 FreeMarker 模板

使用 FreeMarker 技术对代码清单 17-42 中的页面进行重写，并将它存放在 WEB-INF/ftl 目录下，如代码清单 17-43 所示。

代码清单 17-43 userListFtl.ftl：显示用户列表数据的页面

```
<html>
  <head>
    <title>smart</title>
  </head>
  <body>
    用户列表
    <table>
      <#list userList as user>①
        <tr>
          <td>
            <a href="/user/showUser/${user.userName}.html">${user.userName}
          </a>
          <td>${user.realName}</td>
          <td> ${user.birthday?string("yyyy-MM-dd")} </td>③
        </tr>
      </#list>
    </table>
```

对模型数据中的 userList 属性对象进行迭代

②

显示模型属性对象中的值

格式化日期

```
</body>
</html>
```

在 `UserController` 中添加一个处理方法，其返回的逻辑视图名为 `userListFtl`，模型数据中包含了一个名为 `userList` 的属性对象，如代码清单 17-44 所示。

代码清单 17-44 `UserController.java`：显示用户列表数据的处理方法

```
@RequestMapping(path = "/showUserListByFtl")
public String showUserListInFtl(ModelMap mm) {
    List<User> userList = new ArrayList<User>();
    ...

    mm.addAttribute("userList", userList); ①
    return "userListFtl"; ②
}
```

以 `userList` 为属性名，将 `User` 列表添加到模型数据中。

逻辑视图名为 `userListFtl`

接下来的工作就是在 Spring Web 上下文中装配好 FreeMarker 的基础设施和 FreeMarker 视图解析器，使 Spring MVC 能够将 `userListFtl` 解析为 `userListFtl.ftl` 对应的视图对象。

2. 如何在 Spring Web 上下文中配置 FreeMarker

在 Spring Web 上下文中配置 FreeMarker 的过程如代码清单 17-45 所示。

代码清单 17-45 `smart-servlet.xml`：装配 FreeMarker 基础设施和解析器

```
...
<bean class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer"
    p:templateLoaderPath="/WEB-INF/ftl" ①-1
    p:defaultEncoding="UTF-8" ①-2
    <property name="freemarkerSettings"> ①-3
        <props>
            <prop key="classic_compatible">true</prop>
        </props>
    </property>
</bean>

<bean class="org.springframework.web.servlet.view.freemarker.FreeMarkerViewResolver"
    p:order="5" ②-1
    p:suffix=".ftl" ②-2
    p:contentType="text/html; charset=utf-8"/> ②-3
```

在①处通过 `FreeMarkerConfigurer` 配置了 FreeMarker 环境，首先在①-1 处指定了模板文件存放的路径。由于我们的模板文件采用 UTF-8 编码格式，所以必须显式配置 `defaultEncoding` 属性；否则将采用系统默认的编码，造成乱码现象。

FreeMarker 拥有丰富的自定义属性，用户可以通过 `freemarkerSettings` 进行统一的设置（需要阅读 FreeMarker 的参考文档，以获取可配置的属性信息），这些属性名称都采用小写并用“_”连接。在①-3 处配置的 `classic_compatible` 属性非常重要，否则当 FreeMarker 模板碰到值为 `null` 的对象属性时，将抛出异常。将 `classic_compatible` 属性设置为 `true` 后，FreeMarker 将采用类似于 JSTL 标签的行为处理模型属性数据，返回一个空白字符串，而非抛出系统异常。

在搭建好 FreeMarker 环境后，就可以通过 `FreeMarkerViewResolver` 视图解析器将

userListFtl 的逻辑视图名解析为一个对应的 FreeMarkerView 视图对象。

在②-1 处指定该视图解析器的优先级, 它将优先于 InternalResourceViewResolver 执行 (因为 InternalResourceViewResolver 默认的优先级最低)。而在②-2 处指定了后缀, 这样 userListFtl 的逻辑视图名将解析为 /WEB-INF/ftl/userListFtl.ftl 视图对象。由于 userListFtl.ftl 模板最终产生的是 HTML, 且我们希望使用 UTF-8 编码格式输出内容, 所以需要通过②-3 处的 contentType 属性进行相应配置。如果不指定该属性, 则输出的 HTML 将产生中文乱码问题。

3. 使用 Spring 为 FreeMarker 提供的宏

Spring 为 FreeMarker 提供了许多有用的宏, 这些宏类似于 Spring 的 JSP 表单标签, 此外还提供了一些其他有用的功能。考虑一下代码清单 17-43 中②处的以下代码:

```
<a href="/user/showUser/${user.userName}.html">${user.userName}</a>
```

这里使用的 URL 地址没有考虑部署路径, 如果部署路径不是 "/", 则将发生 URL 地址错误。由于应用部署目录需要在最终部署时才能确定, 因此不能直接通过硬编码的方式指定。在 JSP 中, 一般通过 JSTL 的 <c:url> 将 URL 转换为包含应用部署路径的绝对 URL。

```
<a href="<c:url value="/user/showUser/${user.userName}.html"/>">${topic.title} </a>
```

但在 FreeMarker 的模板中, 该如何完成这一功能呢? 答案是使用 Spring 的 FreeMarker 宏。此外, Spring 的 FreeMarker 宏还提供了本地化信息显示功能。使用该宏对代码清单 17-43 中的代码进行调整, 如代码清单 17-46 所示。

代码清单 17-46 userListFtl.ftl: 使用 Spring 的宏

```
<#import "spring.ftl" as spring /> ①
<html>                                引入 Spring 的 FreeMarker 宏定义文件
  <head>
    <title><@spring.message "website.title"/></title> ②
  </head>
  <body>                                引用国际化资源
    <@spring.message "user.userList.title"/>
    <table>
      <#list userList as user>
        <tr>                                使用 Spring 的宏对 URL 进行格式化
          <td>
            <a href="<@spring.url '/user/showUser/${user.userName}.html' />">③
              ${user.userName}
            </a>
          </td>
          <td>${user.realName}</td>
          <td> ${user.birthdate?string("yyyy-MM-dd")}</td>
        </tr>
      </#list>
    </table>
  </body>
</html>
```

首先在①处引入 Spring 的 FreeMarker 宏定义文件, 并指定一个命名空间名, 一般

设置为 `spring`。然后在②处通过`<@spring.message/>`引用国际化资源显示本地化信息。最后在③处通过`<@spring.url/>`宏将`'/user/showUser/${user.userName}.html'`转换为包含部署路径的绝对路径。

Spring 为 FreeMarker 提供的表单宏和其他宏统一在表 17-8 中介绍。

表 17-8 Spring为FreeMarker提供的宏

宏 类 型	说 明
国际化信息，code 代表国际化资源的代码	<code><@spring.message code/></code> 示例： <code><@spring.message "user.userName"/></code>
国际化信息，如果没有对应代码，则使用默认值（text）指定	<code><@spring.messageText code, text/></code> 示例： <code><@spring. messageText "user.userName" "用户名"/></code>
获取相对于应用服务器的绝对路径	<code><@spring.url relativeUrl/></code> 示例： <code><@spring.url "/register.html"/></code>
输入框组件	<code><@spring.formInput path, attributes, fieldType/></code> 示例： <code><@spring.formInput "user.userName" "class='inputSty'" /></code>
隐藏组件	<code><@spring.formHiddenInput path, attributes/></code> 示例： <code><@spring. formHiddenInput "user.userName"/></code>
密码组件	<code><@spring.formPasswordInput path, attributes/></code> 示例： <code><@spring.formPasswordInput "user.password"/></code>
多行文本框	<code><@spring.formTextarea path, attributes/></code> 示例： <code><@spring.formTextarea "user.desc" "cols=3 rows=2"/></code>
单选下拉框	<code><@spring.formSingleSelect path, options, attributes/></code> 示例： <code><@spring.formSingleSelect "user.sex" "sexList"/></code>
多选框	<code><@spring.formMultiSelect path, options, attributes/></code> 示例： <code><@spring. formMultiSelect "user.sex" "sexList"/></code>
单选框	<code><@spring.formRadioButtons path, options separator, attributes/></code> 示例： <code><@spring. formRadioButtons "user.sex" "sexList" " " /></code>
复选框	<code><@spring.formCheckboxes path, options, separator, attributes/></code> 示例： <code><@spring. formCheckboxes "user.sex" "sexList" " " /></code>
绑定组件的错误信息 (需要和组件配合使用)	<code><@spring.showErrors separator, classOrStyle/></code> 示例： <code><@spring.formInput "user.userName" "class='inputSty'" /></code> <code><@spring.showErrors "
" "errorSty"/></code>

此外，也可以使用 Spring 低版本中的`<@spring.bind "command.userName"/>`绑定表单对象的某个属性（绑定到名为 `status` 的请求参数中），其后可以通过`${spring.status.expression}`和`${spring.status.value}`引用这个属性的名字和值。不过，这种方式显然已经过时，直接使用 Spring 的 FreeMarker 宏是更简单的方式。

17.4.5 Excel

如果希望使用 Excel 展示用户列表，则仅需扩展 Spring 的 `AbstractExcelView` 或

AbstractJExcelView 即可。实现 buildExcelDocument() 方法，在方法中使用模型数据对象即可构造 Excel 文档。AbstractExcelView 基于 POI API，而 AbstractJExcelView 基于 JExcelAPI。下面通过扩展 AbstractExcelView 定义显示用户列表的 Excel 视图类，如代码清单 17-47 所示。

代码清单 17-47 UserListExcelView.java

```
package com.smart.web;
...
import org.apache.commons.lang.time.DateFormatUtils;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;
import org.springframework.web.servlet.view.document.AbstractExcelView;

public class UserListExcelView extends AbstractExcelView {①

    @Override
    protected void buildExcelDocument(Map<String, Object> model,
        HSSFWorkbook workbook, HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        response.setHeader("Content-Disposition", "inline; filename="+
            new String("用户列表".getBytes(), "iso8859-1"));②
        List<User> userList = (List<User>) model.get("userList");
        HSSFSheet sheet = workbook.createSheet("users");
        HSSFRow header = sheet.createRow(0);
        header.createCell(0).setCellValue("账号");
        header.createCell(1).setCellValue("姓名");
        header.createCell(2).setCellValue("生日");

        int rowNum = 1;
        for (User user : userList) {
            HSSFRow row = sheet.createRow(rowNum++);
            row.createCell(0).setCellValue(user.getUserName());
            row.createCell(1).setCellValue(user.getRealName());
            String createDate = DateFormatUtils.format(user.getBirthDay(),
                "yyyy-MM-dd");
            row.createCell(2).setCellValue(createDate);
        }
    }
}
```

继承于 AbstractExcelView,
使用 POI 技术创建 Excel 文档

Excel 文档名称必须编码为
iso8859-1, 否则会出现乱码

需要特别注意的是②处对响应报文头的设置，它让浏览器在页面中直接显示 Excel 文件，而非显示一个对话框提示用户下载或打开文件。如果使用的是 IE 浏览器，则在使用该报文头时，IE 会在页面中直接显示对应的 Excel 文件；不过 Chrome 等浏览器还是采用下载的方式，其主要出于安全方面的考虑。如果希望在 IE 浏览器中显示提示用户下载或打开文件的对话框，则可以对响应报文头进行调整：

```
response.setHeader("Content-Disposition", "attachment; filename="+
    new String("用户列表".getBytes(), "iso8859-1"));
```

编写好 Excel 视图类后，必须在 smart-servlet.xml 中进行相应的配置。

```
<bean class="org.springframework.web.servlet.view.BeanNameViewResolver"p:order="10"/>
<bean id="userListExcel" class="com.smart.web.UserListExcelView"/>
```

由于 `UserListExcelView` 代表的 Excel 视图对象是一个 Bean，因此，我们使用 `BeanNameViewResolver` 作为视图解析器，将其优先级序号设置为 10，这样它的优先级将排在 17.4.4 节所介绍的 `FreeMarkerViewResolver` 之后。

在 `UserController` 中添加一个相应的请求处理方法。

```
@RequestMapping(path = "/showUserListByXls")
public String showUserListInExcel (ModelMap mm) {
    ...
    userList.add(user1);
    userList.add(user2);
    mm.addAttribute("userList", userList);
    return "userListExcel";
}
```

这样，在浏览器地址栏中输入如下 URL 时，根据浏览器的不同，将显示或下载一个 Excel 文件。

```
http://localhost:8080/chapter17/user/showUserListByXls.html
```

17.4.6 PDF

PDF 视图和 Excel 类似，也使用一个 Bean 作为视图对象，如代码清单 17-48 所示。

代码清单 17-48 `UserListPdfView.java`

```
package com.smart.web;
...
import org.apache.commons.lang.time.DateFormatUtils;
import org.springframework.web.servlet.view.document.AbstractPdfView;
import com.lowagie.text.Cell;
import com.lowagie.text.Document;
import com.lowagie.text.Element;
import com.lowagie.text.Font;
import com.lowagie.text.Phrase;
import com.lowagie.text.Table;
import com.lowagie.text.pdf.BaseFont;
import com.lowagie.text.pdf.PdfWriter;

public class UserListPdfView extends AbstractPdfView {

    @Override
    protected void buildPdfDocument (Map<String, Object> model,
        Document document, PdfWriter writer, HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        response.setHeader("Content-Disposition", "inline; filename="+
            new String("用户列表".getBytes(), "iso8859-1"));
        List<User> userList = (List<User>) model.get("userList");
        Table table = new Table(3);
        table.setWidth(80);
        table.setBorder(1);
        table.getDefaultCell().setHorizontalAlignment(Element.ALIGN_CENTER);
```

```

        table.getDefaultCell().setVerticalAlignment(Element.ALIGN_MIDDLE);

        BaseFont cnBaseFont = BaseFont.createFont("STSongStd-Light",
                                                    "UniGB-UCS2-H", false);
        Font cnFont = new Font(cnBaseFont, 10, Font.NORMAL, Color.BLUE);

        table.addCell(buildFontCell("账号", cnFont));
        table.addCell(buildFontCell("姓名", cnFont));
        table.addCell(buildFontCell("生日", cnFont));
        for (User user : userList) {
            table.addCell(user.getUserName());
            table.addCell(buildFontCell(user.getRealName(), cnFont));
            String createDate = DateFormatUtils.format(user.getBirthday(),
                "yyyy-MM-dd");
            table.addCell(createDate);
        }
        document.add(table);
    }

    private Cell buildFontCell(String content, Font font) throws RuntimeException {
        try {
            Phrase phrase = new Phrase(content, font);
            return new Cell(phrase);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}

```

使用中文字体 ①

② 对于中文字符，要使用中文字段构造 Cell 对象，否则会产生乱码

③ 英文字符可直接添加到 Cell 中

④

默认的 lowagie 类包不支持亚洲文字（包括简体中文、繁体中文、日文、韩文等），当输出内容遇到亚洲文字时，直接输出空白字符。针对这个问题，lowagie 提供了一个附加的 iTextAsian.jar 包，它包含了亚洲文字字体，需要在 pom.xml 中添加这个构件（由于 Maven 公服上没有这个构件，所以本书配套网盘的 libs 目录下提供了该构件）。

```

<dependency>
    <groupId>com.lowagie</groupId>
    <artifactId>iTextAsian</artifactId>
    <version>2.0.7</version>
    <scope>system</scope>
    <systemPath>${basedir}/../libs/iTextAsian.jar</systemPath>
</dependency>

```

在①处创建了一个中文字体对象。在④处添加了一个 buildFontCell()方法，它将需要输出的内容通过 Phrase 对象使用特殊字体进行封装，只要传入的字体对象是中文字体，返回的 Cell 对象中的中文内容就可以正常显示。

UserListPdfView 和 UserListExcelView 视图一样，可以采用 BeanNameViewResolver 作为视图解析器，因此仅需在 smart-servlet.xml 中添加视图 Bean 的声明即可。

```
<bean id="userListPdf" class="com.smart.web.UserListPdfView"/>
```

在 UserController 中添加一个 showUserListInPdf()方法。

```

@RequestMapping(path = "/showUserListByPdf")
public String showUserListInPdf(ModelMap mm) {

```

```

List<User> userList = new ArrayList<User>();
...
mm.addAttribute("userList",userList);
return "userListPdf";
}

```

在浏览器地址栏中输入如下 URL 时, 根据浏览器的不同, 将显示或下载一个 PDF 文件。

<http://localhost:8080/chapter17/user/showUserListByPdf.html>

17.4.7 输出 XML

Spring MVC 还可以将模型中的数据以 XML 的形式输出, 其对应的视图对象为 `MarshallingView`。下面在 `smart-servlet.xml` 中添加 `MarshallingView` 的配置, 如代码清单 17-49 所示。

代码清单 17-49 smart-servlet.xml: 添加 `MarshallingView`

```

...
<bean id="userListXml"
class="org.springframework.web.servlet.view.xml.MarshallingView"
    p:modelKey="userList"
    p:marshaller-ref="xmlMarshaller"/>
<bean id="xmlMarshaller" class="org.springframework.oxm.xstream.XStreamMarshaller">
    <property name="streamDriver">
        <bean class="com.thoughtworks.xstream.io.xml.StaxDriver" />
    </property>
    <property name="annotatedClasses">
        <list>
            <value>com.smart.domain.User</value>
        </list>
    </property>
</bean>

```

`MarshallingView` 使用 `Marshaller` 将模型数据转换为 XML, 通过 `marshaller` 属性注入一个 `Marshaller` 实例。在默认情况下, `MarshallingView` 会将模型中的所有属性都转换为 XML。由于模型属性包含很多隐式数据, 直接将模型中的所有数据全部输出往往并不是我们所期望的。`MarshallingView` 允许通过 `modelKey` 指定模型中的哪个属性需要输出为 XML。

在 `UserController` 中添加一个处理方法, 返回 `userListXml` 逻辑视图名。

```

@RequestMapping(path = "/showUserListByXml")
public String showUserListInXml(ModelMap mm){
    List<User> userList = new ArrayList<User>();
    ...
    mm.addAttribute("userList",userList);
    return "userListXml";
}

```

在浏览器地址栏中输入如下 URL 时, 将返回 XML 内容的响应。

<http://localhost:8080/chapter17/user/showUserListByXml.html>

17.4.8 输出 JSON

Spring 4.0 MVC 的 MappingJackson2JsonView 借助 Jackson 框架的 ObjectMapper 将模型数据转换为 JSON 格式输出。由于 MappingJackson2JsonView 也是一个 Bean，可以通过 BeanNameViewResolver 进行解析，因此仅需在 smart-servlet.xml 中直接配置即可。

```
<bean id="userListJson"
      class="org.springframework.web.servlet.view.json.MappingJackson2JsonView"
      p:modelKeys="userList"/>
```

在默认情况下，MappingJackson2JsonView 会将模型中的所有数据全部输出为 JSON，这显然是不适合的，可以通过 modelKeys 指定模型中的哪些属性需要输出。

在 UserController 中添加一个返回 userListJson 逻辑视图名的方法。

```
@RequestMapping(path = "/showUserListByJson")
public String showUserListInJson(ModelMap mm){
    List<User> userList = new ArrayList<User>();
    ...
    mm.addAttribute("userList",userList);
    return "userListJson";
}
```

在浏览器地址栏中输入 <http://localhost:8080/chapter17/user/showUserListByJson.html>，将以 JSON 格式输出模型中名为 userList 的属性对象。

17.4.9 使用 XmlViewResolver

如果视图对象的 Bean 数目太多，那么直接在 smart-servlet.xml 文件中配置，势必影响主配置文件的简洁性。XmlViewResolver 和 BeanNameViewResolver 功能相似，唯一不同的是它可以将视图 Bean 定义在一个独立的 XML 文件中。要使用 XmlViewResolver，必须在 smart-servlet.xml 添加以下片段：

```
<bean class="org.springframework.web.servlet.view.XmlViewResolver"
      p:order="20"
      p:location="WEB-INF/views/smart-views.xml"/>
```

在默认情况下，XmlViewResolver 在 WEB-INF/views.xml 中查找视图 Bean 的定义文件。我们将视图 Bean 定义在 WEB-INF/views/smart-views.xml 文件中，所以需要通过 location 显式指定。

定义视图 Bean 的 smart-views.xml 文件，其格式和普通的 Spring 配置文件没有任何区别。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-4.0.xsd">
    <bean id="userListJson1"
```



```

class="org.springframework.web.servlet.view.json.MappingJackson2JsonView" ①
p:modelKeys="userList" />
<bean id="userListExcell" class="com.smart.web.UserListExcelView"/>
<bean id="userListPdf1" class="com.smart.web.UserListPdfView"/>
</beans>

```

定义视图 Bean

不过在 smart-views.xml 文件中定义的 Bean 不能被 Spring Web 上下文的其他 Bean 引用，它是被 XmlViewResolver 独享的。

17.4.10 使用 ResourceBundleViewResolver

对于同一个逻辑视图名，如果希望为不同地区的用户提供不同类型的视图，则应该考虑使用 ResourceBundleViewResolver。和 XmlViewResolver 将视图定义在 XML 文件中不同，ResourceBundleViewResolver 通过一个国际化资源文件定义视图对象。

例如，假设小春论坛需要考虑不同国家和地区客户的不同展现需求：中国人喜欢使用 HTML 页面的展现方式，而美国人喜欢使用 PDF 的展现方式——虽然这个需求很“怪诞”，但不影响我们把它作为一个例子。可以通过 ResourceBundleViewResolver 来实现这个需求，在 smart-servlet.xml 中添加以下代码：

```

<bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver"
p:order="30"
p:basename="i18n/views"/>

```

通过 basename 属性指定视图国际化资源文件的基名。对应于中国大陆的视图资源文件为 views_zh_CN.properties，其内容为：

```

userList18n.(class)=org.springframework.web.servlet.view.JstlView ①
userList18n.url=/WEB-INF/views/user/userList.jsp ②

```

其中，使用<逻辑视图名>.(class)属性设置视图实现类，如①处所示。如果视图实现需要一个 URL 资源的支持，则可以通过<逻辑视图名>.url 设置这个 URL 资源。对于 JstlView 视图实现类来说，其实是通过调用 JstlView#setUrl()设置<逻辑视图名>.url 指定 JSP 文件的。

对应于美国的视图资源文件为 views_en_US.properties，其内容为：

```

userList18n.(class)=com.smart.web.UserListPdfView

```

当美国用户查看论坛的用户列表时，ResourceBundleViewResolver 视图解析器将其解析为一个 PDF 文档。

在一个视图国际化资源文件中可以定义多个视图对象，如下：

```

welcomeView.(class)=org.springframework.web.servlet.view.JstlView ① 第一个视图
welcomeView.url=/WEB-INF/jsp/welcome.jsp

vetsView.(class)=org.springframework.web.servlet.view.JstlView ① 第二个视图
vetsView.url=/WEB-INF/jsp/vets.jsp
...

```


17.4.11 混合使用多种视图技术

1. ContentNegotiatingViewResolver

Spring 最重要的一个新功能就是对 REST 编程风格的支持。REST 风格的应用对资源的 URL 定义有严格的要求：一个资源对象对应唯一的 URL。在前面的例子中，我们使用不同的视图显示用户列表，如果把“用户列表”看作一个资源，那么我们希望通过一个相同的 URL 访问这个资源，而不是使用不同的 URL 访问该资源的不同视图。

17.2.5 节介绍了使用 `HttpMessageConverter` 对标注了 `@ResponseBody` 或返回值为 `ResponseEntity` 的处理器方法进行响应信息转换的内容，Spring MVC 可以根据请求报文头的 `Accept` 属性选择适合的 `HttpMessageConverter` 将处理方法的返回值以 XML、JSON 等不同形式输出响应。也就是说，调用者可以通过设置请求报文头的 `Accept` 值控制服务器端返回的数据格式，从而实现对同一资源采用相同 URL 的 REST 编程风格。

但是基于 `HttpMessageConverter` 的实现方式存在以下限制。

- ❑ 只能通过请求报文头的 `Accept` 值控制服务器端返回的数据格式。如果客户端是浏览器，那么，除非使用 AJAX，否则很难控制 `Accept` 报文头的值。一般情况下，该值是由浏览器自己决定的。
- ❑ 无法通过 URL 扩展名或请求参数控制服务器端的资源输出形式，因此无法将其对应一个 URL 发布出去。
- ❑ 如果希望使用 XML、JSON、一个网页等形式输出资源，则 `HttpMessageConverter` 很难达到要求。因为 `HttpMessageConverter` 很难调用一个视图对象渲染模型，它直接负责将资源输出为某一内容形式。

所以，如果希望将资源以 XML、JSON 等纯数据的格式输出，且不在意使用报文头控制资源输出，那么适合选择 `HttpMessageConverter` 实现方式。否则，建议采用 Spring MVC 的 `ContentNegotiatingViewResolver` 视图解析器，虽然它和 `HttpMessageConverter` 在功能上有一些重叠，但在使用上更加灵活。

`ContentNegotiatingViewResolver` 名为“视图解析器”，但它并不是一个传统意义上的视图解析器，它像一个仲裁机构或一个代理人，根据请求信息从上下文中选择一个适合的视图解析器负责解析。因此，一般将 `ContentNegotiatingViewResolver` 的优先级设为最高，以保证优先调用 `ContentNegotiatingViewResolver`。

`ContentNegotiatingViewResolver` 根据请求所要求的 MIME 类型决定由哪个视图解析器负责处理，它按照如下方式工作。

① 如果其 `favorPathExtension` 属性设置为 `true`（默认为 `true`），则根据 URL 中的文件扩展名确定 MIME 类型（如 `userList.xml`、`userList.json` 等）。

② 如果其 `favorParameter` 属性设置为 `true`（默认为 `false`），则根据请求参数的值确定 MIME 类型。默认的请求参数是 `format`。可以通过 `parameterName` 属性指定一个自定义的参数。

③ 如果还没有找到对应的 MIME 类型，且 Java Activation Framework (JAF) 位于类路径下，则通过 JAF 的 `FileTypeMap.getContentType(url)` 方法对 URL 进行判断，以获取对应的 MIME 类型。

④ 如果以上步骤都失败，且 `ignoreAcceptHeader` 属性设置为 `false` (默认为 `false`)，则采用 `Accept` 请求报文头的值确定 MIME 类型。由于不同浏览器产生的 `Accept` 头都是不一样的，所以一般不建议采用 `Accept` 确定 MIME 类型。

2. 使用同一 URL 获取不同形式的返回内容

假设希望使用以下 REST 风格的 URL 以不同的 MIME 格式获取相同的资源(用户列表)。

- ☐ `/user/showUserListMix.html`: 返回一个 HTML 页面显示的用户列表。
- ☐ `/user/showUserListMix.html?content=xml`: 返回 XML 格式的用户列表。
- ☐ `/user/showUserListMix.html?content=json`: 返回 JSON 格式的用户列表。

首先，在 `UserController` 中添加一个处理方法。

```
@RequestMapping(path = "/showUserListMix")
public String showUserListMix(ModelMap mm){
    List<User> userList = new ArrayList<User>();
    ...
    mm.addAttribute("userList",userList);
    return "userListMix";
}
```

接着，在 `smart-servlet.xml` 中添加以下配置片段，配置一个 `ContentNegotiatingViewResolver`，采用请求参数指定内容资源的返回类型，如代码清单 17-50 所示。

代码清单 17-50 smart-servlet.xml: 添加 `ContentNegotiatingViewResolver`

```
<bean id="contentNegotiationManager"
class="org.springframework.web.accept.ContentNegotiationManagerFactoryBean"
    p:ignoreAcceptHeader="true"
    p:favorPathExtension="false"
    p:favorParameter="true"
    p:parameterName="format"
    p:defaultContentType="text/html">
    <property name="mediaTypes">
        <value>
            html=text/html
            xml=application/xml
            json=application/json
        </value>
    </property>
</bean>

<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver"
    p:order="0" >
    <property name="contentNegotiationManager" ref="contentNegotiationManager"/>
    <property name="defaultViews">
        <list>
```

① 不支持扩展文件名，不支持 `Accept` 报文头指定 MIME 类型，通过请求参数指定 MIME 类型，参数名为 `content`，请求类型的参数值和 MIME 类型的映射列表

② 使该视图解析器优先级最高

```

        <bean class="org.springframework.web.servlet.view.json.MappingJackson2
JsonView"
            p:modelKeys="userList" />③
        <bean class="org.springframework.web.servlet.view.xml.MarshallingView"
            p:modelKey="userList" p:marshaller-ref="xmlMarshaller" />④
    </list>
</property>
</bean>
<!-- 以下是上下文中已经配置的视图解析器 -->
<bean class="org.springframework.web.servlet.view.BeanNameViewResolver"
    p:order="10" />
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"⑤
    p:order="100"
    p:viewClass="org.springframework.web.servlet.view.JstlView"
    p:prefix="/WEB-INF/views/"
    p:suffix=".jsp" />

```

ContentNegotiatingViewResolver 会在上下文中查找和 MIME 类型匹配的视图解析器，并委托它们进行视图解析。这样通过 content 请求参数就可以控制资源的输出格式。

③处配置的是 JSON 视图对象，而④处配置的是 XML 视图对象，它们都是默认的候选视图对象，会覆盖对应视图解析器返回的视图对象。换句话说，当资源 URL 带 content=json 参数时，直接返回③处对应的视图对象；当资源 URL 带 content=xml 参数时，直接返回④处对应的视图对象；当资源 URL 带 content=html 参数时，由⑤处的 InternalResourceViewResolver 解析；当资源 URL 不带 content 参数时，由于已经通过 defaultContentType="text/html" 指定了默认的 MIME 类型，因此也由⑤处的 InternalResourceViewResolver 解析。

17.5 本地化解析

17.5.1 本地化的概念

一般情况下，Web 应用根据客户端浏览器的设置判断客户端的本地化类型，用户可以通过 IE 菜单“工具→Internet 选项→语言”，在打开的“语言首选项”对话框中选择本地化类型。浏览器中设置的本地化类型会包含在 HTML 请求报文头中发送给 Web 服务器，确切地说是通过报文头的 Accept-Language 参数将“语言首选项”对话框中选择语言发送到服务器，成为服务器判别客户端本地化类型的依据。

如果 Web 应用基于这种方式提供本地化页面，则用户只得通过改变浏览器的设置进行本地化类型的切换。读者是否见过一些网站在页面的右上角提供了语言切换的图标，如英文、中文、繁体等字样。这种语言切换功能不要求用户更改浏览器的设置，它们通过 Cookie、Session 或请求参数即可切换本地化类型，为用户的使用带来了便利。

在默认情况下, Spring MVC 根据 Accept-Language 参数判断客户端的本地化类型。此外, 它还提供了多种指定客户端本地化类型的方式, 如通过 Cookie、Session 指定。事实上, 当收到请求时, Spring MVC 在上下文中寻找一个本地化解析器(LocaleResolver), 找到后使用它获取请求所对应的本地化类型信息。

除此之外, Spring MVC 还允许装配一个动态更改本地化类型的拦截器, 这样通过指定一个请求参数就可以控制单个请求的本地化类型。本地化解析器和拦截器都定义在 org.springframework.web.servlet.i18n 包中, 用户可以在 DispatcherServlet 上下文中配置它们。

Spring 提供了以下 4 个本地化解析器。

- **AcceptHeaderLocaleResolver**: 根据 HTTP 报文头的 Accept-Language 参数确定本地化类型。如果没有显式定义本地化解析器, 则 Spring MVC 默认采用 AcceptHeaderLocaleResolver。
- **CookieLocaleResolver**: 根据指定的 Cookie 值确定本地化类型。
- **SessionLocaleResolver**: 根据 Session 中特定的属性值确定本地化类型。
- **LocaleChangeInterceptor**: 从请求参数中获取本次请求对应的本地化类型。

在 17.3.4 节的实例中, 我们为表单校验错误信息提供了国际化的支持, 当表单数据校验存在错误时, Spring MVC 使用默认的 AcceptHeaderLocaleResolver 抽取请求报文头的 Accept-Language 属性以确定客户端本地化类型, 并返回对应的本地化错误信息。下面采用其他的本地化解析器解析客户端的本地化类型。

17.5.2 使用 CookieLocaleResolver

要使用 Cookie 保存本地化类型信息, 服务器端要做的事情很简单, 只需在 Spring MVC 上下文中配置一个 CookieLocaleResolver 就可以了, DispatcherServlet 会自动识别本地化解析器并装配它。

```
<bean id="localeResolver"
class="org.springframework.web.servlet.i18n.CookieLocaleResolver"
  p:cookieName="clientLanguage"① ← 客户端保存本地化类型的 Cookie 名
  p:cookieMaxAge="100000"② ← Cookie 最大存活秒数
  p:cookiePath="/" ③ ← Cookie 保存路径
  p:defaultLocale="zh_CN"/>④ ← 默认本地化类型
```

这样, 客户端只要通过 JavaScript 更改 clientLanguage 这个 Cookie 的值, 就可以控制服务器端返回相应的本地化页面。



实战经验

在实际的 Web 应用中, 一般将用户个性化配置信息(包括本地化类型信息)保存在数据库中。当用户登录系统时, 先加载这些个性化信息并保存在特定的媒介中(如

Cookie 或 Session), 这样用户的个性化设置就可以永远生效而不会随着 Cookie 或 Session 的清除或过期而消失。

17.5.3 使用 SessionLocaleResolver

SessionLocaleResolver 查找 Session 中属性名为 SessionLocaleResolver.LOCALE_SESSION_ATTRIBUTE_NAME 的属性, 并将其转换为 Locale 对象, 以此作为客户端的本地化类型。如果应用程序使用 Session 维护客户的信息, 则 SessionLocaleResolver 是最适合不过的。

用户仅需将 SessionLocaleResolver 的配置添加到 Spring MVC 上下文的配置文件中即可。

```
<bean id="localeResolver" class="org.springframework.web.servlet.i18n.SessionLocaleResolver"/>
```

SessionLocaleResolver 和 CookieLocaleResolver 的区别是, 前者一般要求用户登录后生成相应的用户会话才有效, 而后者只要浏览器有 Cookie 存在即可生效。

17.5.4 使用 LocaleChangeInterceptor

很多国际型的网站都允许通过一个请求参数控制网站的本地化, 如 `www.xxx.com?locale=zh_CN` 返回对应中国大陆的本地化网页, 而 `www.xxx.com?locale=en` 返回本地化为英语的网页。这样, 网站使用者可以通过对 URL 的控制返回不同本地化的页面, 非常灵活。在 Spring MVC 中, 这一需求可以通过 LocaleChangeInterceptor 过滤器来完成。

在 Spring MVC 上下文中通过 `<mvc:interceptors>` 配置过滤器。

```
<bean id="localeResolver"
      class="org.springframework.web.servlet.i18n.CookieLocaleResolver"
      p:cookieName="clientLanguage"
      p:cookieMaxAge="100000"
      p:cookiePath="/"
      p:defaultLocale="zh_CN"/>①
<mvc:interceptors>
  <bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor" />②
</mvc:interceptors>
```

LocaleChangeInterceptor 在默认情况下通过 locale 请求参数获取本次请求对应的本地化类型, 用户可以通过其 paramName 属性指定一个其他参数名。由于 LocaleChangeInterceptor 的主要任务是从请求中获取本地化类型并将其设置给真正的本地化解析器, 所以在配置 LocaleChangeInterceptor 之前, 必须在上下文中先配置一个本地化解析器, 如①处所示。值得注意的是, 由于 AcceptHeaderLocaleResolver 是从请求报文头获取本地化信息的, 因此不能被动态更改。如果配置了 LocaleChangeInterceptor, 则只能选择 CookieLocaleResolver 或 SessionLocaleResolver。

17.6 文件上传

Spring MVC 为文件上传提供了直接支持，这种支持是通过即插即用的 `MultipartResolver` 实现的。Spring 使用 Jakarta Commons FileUpload 技术实现了一个 `MultipartResolver` 实现类：`CommonsMultipartResolver`。

在 Spring MVC 上下文中默认没有装配 `MultipartResolver`，因此默认情况下不能处理文件的上传工作。如果想使用 Spring 的文件上传功能，则需要先在上下文中配置 `MultipartResolver`。

17.6.1 配置 MultipartResolver

下面使用 `CommonsMultipartResolver` 配置一个 `MultipartResolver` 解析器。

```
<bean id="multipartResolver"
      class="org.springframework.web.multipart.commons.CommonsMultipartResolver"
      p:defaultEncoding="UTF-8"① ← 请求的编码格式，默认为 ISO-8859-1
      p:maxUploadSize="5242880"② ← 上传文件大小上限，单位为字节(5MB)
      p:uploadTempDir="file:/d:/temp"/>③ ← 上传文件的临时路径
```

`defaultEncoding` 必须和用户 JSP 的 `pageEncoding` 属性一致，以便正确读取表单的内容。`uploadTempDir` 是文件上传过程中所使用的临时目录，文件上传完成后，临时目录中的临时文件会被自动清除。

为了让 `CommonsMultipartResolver` 正常工作，必须先将 Jakarta Commons FileUpload 及 Jakarta Commons io 的类包添加到类路径下。

17.6.2 编写控制器和文件上传表单页面

在 `UserController` 中添加一个用于处理用户头像上传的方法，如代码清单 17-51 所示。

代码清单 17-51 UserController.java: 上传文件

```
package com.smart.web;
...
import org.springframework.web.multipart.MultipartFile;
@Controller
@RequestMapping("/user")
public class UserController {

    @RequestMapping(path = "/uploadPage") ①
    public String updatePage() {
        return "uploadPage";
    }
    @RequestMapping(path = "/upload")
    public String updateThumb(@RequestParam("name") String name, ②
                             @RequestParam("file") MultipartFile file) throws Exception{
        // 上传的文件自动绑定到 MultipartFile 中
```

```

        if (!file.isEmpty()) {
            file.transferTo(new File("d:/temp/"+file.getOriginalFilename()));
            return "redirect:success.html";
        }else{
            return "redirect:fail.html";
        }
    }
}

```

Spring MVC 会将上传文件绑定到 `MultipartFile` 对象中。`MultipartFile` 提供了获取上传文件内容、文件名等方法, 通过其 `transferTo()` 方法还可将文件存储到硬件中, 具体说明如下。

- ❑ `byte[] getBytes()`: 获取文件数据。
- ❑ `String getContentType()`: 获取文件 MIME 类型, 如 `image/jpeg`、`text/plain` 等。
- ❑ `InputStream getInputStream()`: 获取文件流。
- ❑ `String getName()`: 获取表单中文件组件的名字。
- ❑ `String getOriginalFilename()`: 获取上传文件的原名。
- ❑ `long getSize()`: 获取文件的字节大小, 单位为 `Byte`。
- ❑ `boolean isEmpty()`: 是否有上传的文件。
- ❑ `void transferTo(File dest)`: 可以使用该文件将上传文件保存到一个目标文件中。

负责上传文件的表单和一般表单有一些区别, 表单的编码类型必须是 `multipart/form-data` 类型。

```

<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
    <head>
        <title>请上传用户头像</title>
    </head>
    <body>
        <h1>
            请选择上传的头像文件
        </h1>
        <form method="post" action="<c:url value="/user/upload.html"/>"
            enctype="multipart/form-data">①
            <input type="text" name="name" />
            <input type="file" name="file" />②
            <input type="submit" />
        </form>
    </body>
</html>

```

指定表单内容类型,
以便支持文件上传

上传文件的组件名

17.7 WebSocket 支持

17.7.1 使用 WebSocket

通常应用程序之间发送消息会使用诸如 JMS、AMQP 等技术, 但是如果要实现浏览

器与服务器的全双工通信，则以上技术是不适用的，但在互联网时代，这样的需求又是普遍存在的。随着 HTML 5 的诞生，一个新的协议也应运而生，这便是 WebSocket 协议，它很好地解决了浏览器与服务器全双工通信的问题，而且相对于传统的解决方案能更好地节省服务器资源和带宽并达到实时通信。Java EE 7.0 已经支持 WebSocket 协议，相应的，Spring 4.0 也为 WebSocket 通信提供了支持。

Spring 4.0 主要为 WebSocket 通信提供了以下几个方面的支持。

- ❑ 发送和接收消息的 API。
- ❑ 用来发送消息的模板。
- ❑ 支持 SockJS，用来解决浏览器、服务器及代理不支持 WebSocket 的问题。

事实上，WebSocket 通信可以用在任何类型的应用中，其最简单的形式是在两个应用之间建立通信的通道，如图 17-16 所示。

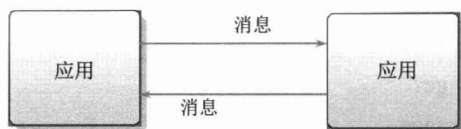


图 17-16 应用之间的 WebSocket 通信

如图 17-16 所示，位于 WebSocket 一端的应用发送消息，而另一端的应用处理消息。因为它是全双工通信的，所以每一端既可以发送又可以处理消息。但是 WebSocket 最为常见的应用场景是实现服务器和基于浏览器的应用之间的通信。在浏览器中使用 JavaScript 开启一个到达服务器的连接，服务器通过这个连接发送更新到浏览器中。相对于传统的在 Web 端轮询服务器及使用 Flash 中的 Socket 和 XMLSocket，这种方式更加高效、便捷。

下面通过一个在 Web 端使用 JavaScript 与服务器端相互传递“hello world!”文本信息的例子来展示 WebSocket 的用法。

Spring 提供了一个 WebSocketHandler 接口，该接口定义了 5 个 WebSocket 相关的接口方法。一如 Spring 惯有的优雅和便捷性，Spring 拥有一个 WebSocketHandler 接口的抽象实现类 AbstractWebSocketHandler，继承该类便可以选择性地实现感兴趣的方法，如代码清单 17-52 所示。

代码清单 17-52 MyWebSocketHandler.java

```

public class MyWebSocketHandler extends AbstractWebSocketHandler {
    @Override
    protected void handleTextMessage(
        WebSocketSession session, TextMessage message) throws Exception {
        //处理文本消息
        System.out.println("收到消息" + message.getPayload());
        //模拟延时
        Thread.sleep(2000);
        //发送文本消息
        System.out.println("发送消息: hello world!");
        session.sendMessage(new TextMessage("hello world!"));
    }
}
  
```

```

    }

    @Override
    public void afterConnectionEstablished(WebSocketSession session) throws Exception {
        System.out.println("建立连接");
    }

    @Override
    public void afterConnectionClosed(WebSocketSession session, CloseStatus status)
        throws Exception {
        System.out.println("关闭连接");
    }
}

```

可以看到，尽管 `AbstractWebSocketHandler` 是一个抽象类，但是我们并不需要重载所有的方法，在该抽象类中所有的方法都以空操作的方式实现，所以我们可选择性地重载相应的方法。除了重载 `WebSocketHandler` 中所定义的 5 个方法外，`AbstractWebSocketHandler` 还定义了以下 3 个方法：

- ☐ `handleTextMessage`。
- ☐ `handleBinaryMessage`。
- ☐ `handlePongMessage`。

这 3 个方法是 `handleMessage()` 方法针对具体不同类型消息处理的实现，顾名思义，`handleTextMessage()` 是处理文本消息类型的方法，`handleBinaryMessage()` 是处理二进制消息类型的方法，`handlePongMessage()` 是处理 Pong 消息类型的方法。因为 `MyWebSocketHandler` 将处理文本类型的消息 “hello world!”，所以我们选择重载 `handleTextMessage()` 方法，在接收文本消息后再简单地输出消息。这样，一个简单的消息处理类就完成了，需要在 `applicationContext.xml` 中增加相关的配置。

```

//将helloHandler映射到“/hello”
<websocket:handlers>
    <websocket:mapping handler="helloHandler" path="/hello"/>
</websocket:handlers>

//声明MyWebSocketHandler Bean
<bean id="helloHandler" class="com.smart.web.MyWebSocketHandler"/>

```

接下来看看客户端是怎么实现的。它会发送 “hello world!” 文本消息到服务器，并监听来自服务器的文本消息。代码清单 17-53 展示了如何使用 JavaScript 开启一个 WebSocket 并使用它发送消息给服务器。

代码清单 17-53 hello.jsp: 客户端代码

```

<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<html>
<head>
    <title>websocket测试</title>
    <script type="text/javascript">
        var url = 'ws://' + window.location.host + '<%=request.getContextPath()%>/hello';
        var sock = new WebSocket(url); //打开WebSocket
    </script>

```

```

sock.onopen = function() { //处理连接开启事件
    console.log('开启WebSocket连接!');
    sayHello();
}

sock.onmessage = function(e) { //处理消息
    console.log('接收消息: ', e.data);
    setTimeout(function() {sayHello()}, 2000);
}

sock.onclose = function() { //连接关闭事件
    console.log('关闭WebSocket连接!');
}

function sayHello() { //发送消息
    console.log('发送消息: hello world!');
    sock.send('hello world!');
}
</script>
</head>
<body>
    hello world!
</body>
</html>

```

在上述代码清单中，使用原生的 JavaScript 创建了 WebSocket 实例。通过创建 WebSocket 实例，实际上打开了一个给定 URL 的 WebSocket。注意，URL 使用了“ws://”前缀，表明这是一个 WebSocket 连接。

WebSocket 创建完毕后，接下来的代码实现了 WebSocket 的事件处理功能。WebSocket 的 onopen、onmessage 和 onclose 事件分别对应 MyWebSocketHandler 类的 afterConnectionEstablished()、handleTextMessage() 和 afterConnectionClosed() 方法。在 onopen 事件中会调用 sayHello() 方法，在该 WebSocket 上发送“hello world!”消息，通过发送“hello world!”，这个双向通信即建立起来。服务器端的 MyWebSocketHandler 作为响应将“hello world!”发送回来，当客户端收到来自服务器端的消息后，onmessage 又会发送一个“hello world!”给服务器端，如此循环反复。

因为推送信息打印在 Web 控制台上，所以读者可以打开浏览器的开发者调试工具来观察效果。如果是 Chrome 浏览器，则按 F12 键后单击 Console 即可观察控制台上的输出，如图 17-17 所示。

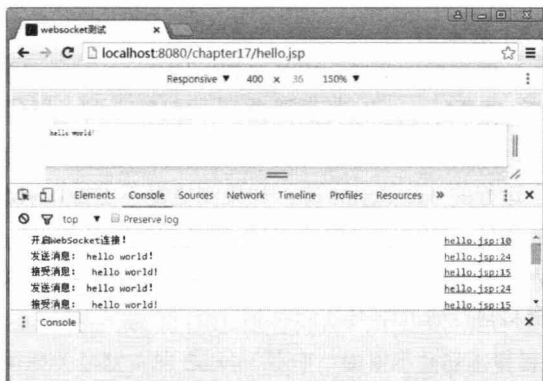


图 17-17 Web 客户端控制台输出信息

在服务器端，也可以看到相关的日志信息，如图 17-18 所示。

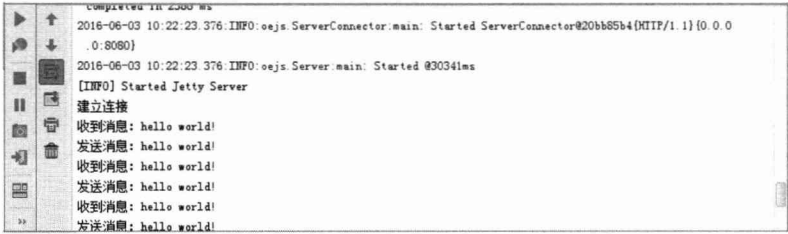


图 17-18 服务器端控制台输出信息

17.7.2 WebSocket 的限制

值得注意的是，WebSocket 是一个比较新的规范，在 Web 浏览器和应用服务器上依然没有得到全面的支持。FireFox 和 Chrome 早已完整支持 WebSocket，但是其他浏览器才刚刚开始支持 WebSocket。下面是主流浏览器对 WebSocket 支持的情况说明。

- ☐ Internet Explorer: 10.0（完整支持）。
- ☐ FireFox: 4.0（部分支持），6.0（完整支持）。
- ☐ Chrome: 4.0（部分支持），13.0（完整支持）。
- ☐ Safari: 5.0（部分支持），12.10（完整支持）。
- ☐ Opera: 11.0（部分支持），12.10（完整支持）。
- ☐ iOS Safari: 4.2（部分支持），6.0（完整支持）。。
- ☐ Android Browser: 4.4（完整支持）。

可以看出，使用 WebSocket 在浏览器端存在一定的限制，其实服务器端的情况也并不乐观，Tomcat 从 7.0.26 版本开始才部分支持 WebSocket，直到 8.0 版本后才真正地全面支持 WebSocket，Jetty 也是到 7.0 版本才开始支持 WebSocket 的。因本书其他章节使用的 Jetty 版本为 6.1.25，并不支持 WebSocket，所以本章所使用的 Jetty 插件特地使用 9.2.1 版本。

即便在浏览器和服务器都支持 WebSocket 的情况下也可能出现问题，比如防火墙通常会限制 HTTP 以外的流量，有可能还没有配置允许进行 WebSocket 通信。但是不要灰心，只要是好的东西总会有完美的替代方案，这就是 SockJS。SockJS 是一个在浏览器上运行的 JavaScript 库，如果浏览器不支持 WebSocket，则该库可以模拟对 WebSocket 的支持，实现浏览器和 Web 服务器之间低延迟、全双工、跨域的通信通道，其 API 和 WebSocket 几乎一样。关于 SockJS 本书不再详细介绍，有兴趣的读者可在网上查阅相关资料自行研究学习。

17.8 杂项

17.8.1 静态资源处理

优雅 REST 风格的资源 URL 不希望带.html 或.do 等后缀，以下是几个优雅的 URL：

- /blog/tom: 用户 tom 的 blog 资源。
- /forum/java: java 论坛版块资源。
- /order/4321: 订单号为 4321 的订单资源。

由于早期的 Spring MVC 不能很好地处理静态资源，所以在 web.xml 中配置 DispatcherServlet 的请求映射时，往往采用*.do、*.xhtml 等方式。这就决定了请求 URL 必须是一个带后缀的 URL，而无法采用真正 REST 风格的 URL。

如果将 DispatcherServlet 请求映射配置为“/”，则 Spring MVC 将捕获 Web 容器所有的请求，包括静态资源的请求，Spring MVC 会将它们当成一个普通请求处理，因找不到对应的处理器而导致错误。

如何让 Spring 框架能够捕获所有 URL 的请求，同时又将静态资源的请求转由 Web 容器处理，是可将 DispatcherServlet 的请求映射配置为“/”的前提。由于 REST 是 Spring 的重要功能之一，所以 Spring 团队很看重静态资源处理这项任务，给出了堪称经典的两种解决方案。

在学习这两个方案之前，先调整 web.xml 中 DispatcherServlet 的配置，使其可以捕获所有的请求。

```
<servlet>
  <servlet-name>smart</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>smart</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
```

通过<url-pattern>/</url-pattern>的配置，所有 URL 请求都将被 Spring MVC 的 DispatcherServlet 截获。

1. 采用<mvc:default-servlet-handler/>

在 smart-servlet.xml 中配置<mvc:default-servlet-handler/>后，会在 Spring MVC 上下文中定义一个 org.springframework.web.servlet.resource.DefaultServletHttpRequestHandler，它将充当一个检查员的角色，对进入 DispatcherServlet 的 URL 进行筛查。如果发现是静态资源的请求，就将该请求转由 Web 应用服务器默认的 Servlet 处理；如果不是静态资源的请求，则由 DispatcherServlet 继续处理。

一般 Web 应用服务器（包括 Tomcat、Jetty、Glassfish、JBoss、Resin、WebLogic

和 WebSphere) 默认的 Servlet 名称都是 default, 因此, DefaultServletHttpRequestHandler 可以找到它。如果用户所使用的 Web 应用服务器的默认 Servlet 名称不是 default, 则需要通过 default-servlet-name 属性显式指定。

```
<mvc:default-servlet-handler default-servlet-name="yourServerDefaultServlet Name"/>
```

2. 采用<mvc:resources/>

<mvc:default-servlet-handler/>将静态资源的处理经由 Spring MVC 框架交回 Web 应用服务器。而<mvc:resources/>更进一步, 由 Spring MVC 框架自己处理静态资源, 并添加一些有用的附加功能。

首先, <mvc:resources/>允许静态资源放置在任何地方, 如 WEB-INF 目录下、类路径下等, 甚至可以将 JavaScript 等静态文件打包到 JAR 包中。通过 location 属性指定静态资源的位置, 由于 location 属性是 Resource 类型, 因此可以使用诸如“classpath:”等的资源前缀指定资源位置。传统 Web 容器的静态资源只能放在 Web 容器的根路径下, <mvc:resources/>则完全打破了这个限制。

其次, <mvc:resources/>依据当前著名的 Page Speed、YSlow 等浏览器优化原则对静态资源提供优化。可以通过 cacheSeconds 属性指定静态资源在浏览器端的缓存时间, 一般可将该时间设置为一年, 以充分利用浏览器端的缓存。在输出静态资源时, 会根据配置设置好响应报文头的 Expires 和 Cache-Control 值。

在接收到静态资源的获取请求时, 会检查请求头的 Last-Modified 值。如果静态资源没有发生变化, 则直接返回 303 响应状态码, 指示客户端使用浏览器缓存的数据, 而非将静态资源的内容输出到客户端, 以充分节省带宽, 提高程序性能。

在 smart-servlet.xml 中添加以下配置:

```
<mvc:resources mapping="/resources/**" location="/", classpath:/META-INF/publicResources/" />
```

以上配置将 Web 根路径“/”及类路径/META-INF/publicResources/下的目录映射为 /resources 路径。假设 Web 根路径下拥有 images 和 js 这两个资源目录, 则可以通过如图 17-19 所示的方式引用静态资源。

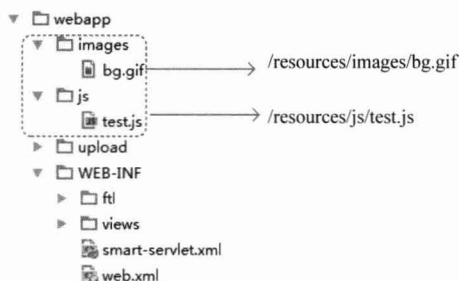


图 17-19 静态资源路径映射

假设类路径/META-INF/publicResources/下还拥有 images/bg1.gif 和 js/test1.js, 则也可以在网页中通过 /resources/images/bg1.gif 和 /resources/js/test1.js 进行引用, 如代码清单 17-54 所示。

代码清单 17-54 test.jsp: 引用静态资源文件

```

<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
  <head>
    <title>静态资源测试页面</title>
    <script src="<c:url value="/resources/js/test.js"/>" type="text/javascript">
  </script>
  </head>
  <body>
    <script>test();</script>
  </body>
</html>

```

通过逻辑映射路径引用资源文件

由于<mvc:resources/>可以将多个物理路径映射为一个逻辑路径,因此,一个用逻辑路径表示的资源在多个物理路径下都存在。对于这个问题,<mvc:resources/>的处理机制是,只要在一个物理路径下找到匹配的资源后就返回,查找的顺序和物理路径在 location 中的配置顺序一致。

聪明的读者可能会问:既然将 Web 根路径“/”映射为“/resources/**”,是否可以在网页中通过“/resources/WEB-INF/web.xml”访问这个敏感的文件呢?答案是否定的。Spring MVC 在处理映射的静态资源时,会查看引用路径是否包含 WEB-INF 或 META-INF。如果包括,则直接返回 null 值,以保护安全文件不泄露出去。当然,如果将 WEB-INF/设置在 location 属性中,则可以通过/resources/web.xml 的 URL 查看到 web.xml。

```
<mvc:resources mapping="/resources/**" location="/WEB-INF/" />
```

所以使用<mvc:resources/>时需要特别注意,不要一不小心将不希望暴露的资源泄露出去。

通过<mvc:resources/>的 cache-period 属性可以设置静态资源在客户端浏览器中的缓存有效时间。

```
<mvc:resources mapping="/resources/**"
location="/,classpath:/META-INF/publicResources/" cache-period="31536000"/>
```

一般情况下,将 cache-period 设置为一年,以便充分利用客户端的缓存数据。

在发布新版本的应用时,即使服务器端的 JavaScript、CSS 等静态资源文件已经发生了变化,但是由于客户端浏览器本身缓存管理机制的问题,客户端并不会从服务器端下载新的静态资源。一个好的解决办法是:网页中引用静态资源的路径添加应用的发布版本号,这样在发布新的部署版本时,由于版本号的变更造成网页中静态资源路径发生更改,从而使这些静态资源成为“新的资源”,客户端浏览器就会下载这个“新的资源”,而不会使用缓存中的数据。针对这个解决思路,可以通过<mvc:resources/>的静态资源逻辑路径给出一个通用的解决方案。

将发布版本号包含到<mvc:resources/>的静态资源逻辑路径中。首先创建一个 ServletContextAware 实现类,如代码清单 17-55 所示。

代码清单 17-55 ResourcePathExposer.java: 上传文件

```

package com.smart.web;
import javax.servlet.ServletContext;
import org.springframework.web.context.ServletContextAware;
public class ResourcePathExposer implements ServletContextAware {
    private ServletContext servletContext;
    private String resourceRoot; 在实际应用中, 可以在外部属性文件或数据库中保存应用的
    public void init() {          发布版本号, 在此处获取之。此处仅仅提供了一个模拟值
        String version = "1.2.1"; ① ← 资源逻辑路径带上应用的发布版本号

        resourceRoot = "/resources-" + version; ② ←

        servletContext().setAttribute("resourceRoot", ③ ←
            servletContext().getContextPath()+resourceRoot);
    }                                     将资源逻辑路径暴露到 ServletContext
                                       的属性列表中
    public void setServletContext(ServletContext servletContext) {
        this.servletContext = servletContext;
    }

    public String getResourceRoot() {
        return resourceRoot;
    }

    public ServletContext getServletContext() {
        return servletContext;
    }
}

```

在 ResourcePathExposer 中获取应用程序的发布版本号, 产生一个带版本号的静态资源路径 resourceRoot, 同时将其值发布到 ServletContext 中, 这样 JSP 文件就可以通过 `#{resourceRoot}` 引用其值了。

接下来要调整 smart-servlet.xml 中的配置, 以便使用带版本的静态资源逻辑路径。

```

<bean id="rpe" class="com.smart.web.ResourcePathExposer" ①
    init-method="init"/>
<mvc:resources mapping="#{rpe.resourceRoot}/**" ②
    location="/" cache-period="31536000"/>

```

在①处配置好 ResourcePathExposer, 并指定其初始化方法为 init(), 以便在容器启动时让其初始化 resourceRoot 的值。由于其实现了 ServletContextAware 接口, 因此, Spring 会在初始化该 Bean 时将 ServletContext 引用注入进来。

在②处通过 Spring EL 表达式引用 ResourcePathExposer 的 resourceRoot 属性值, 生成动态的静态资源逻辑路径。

最后调整网页中引用静态资源的方式, 如代码清单 17-56 所示。

代码清单 17-56 test.jsp: 引用静态资源文件

```

<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<html>
<head>
<title>静态资源测试页面</title>

```

```

<script src="${resourceRoot}/js/test.js" type="text/javascript"> </script>
</head>
<body>
  <script>test();</script>
</body>
</html>

```

① 引用 ResourcePathExposer 通过 ServletContext 暴露的 resourceRoot 属性值

由于①处引用的 resourceRoot 值和<mvc:resources/>通过#{rpe.resourceRoot}引用的值是一样的，所以可以正确访问到物理静态资源。这样，在每次发布新版本后，随着发布版本号的更改，客户端就会自动下载新的静态资源。



实战经验

要想成为一名 Web 开发高手，不能满足于知道如何做，而要抛开现象探究本质。笔者认为，要成为一名 Web 开发高手，必须熟悉以下内容：

(1) 每次请求和响应的背后究竟发生了哪些步骤，客户端服务器是如何通过 HTTP 请求报文进行交互的。

(2) 深刻掌握 MIME 类型的知识。

(3) 深刻掌握 HTTP 响应状态码的知识，如 404、303 究竟代表什么。

17.8.2 装配拦截器

当收到请求时，DispatcherServlet 将请求交给处理器映射（HandlerMapping），让它找出对应该请求的 HandlerExecutionChain 对象。在讲解 HandlerMapping 之前，有必要认识一下这个 HandlerExecutionChain 对象。

HandlerExecutionChain 顾名思义是一个执行链，它包含一个处理该请求的处理器（Handler），同时包括若干个对该请求实施拦截的拦截器（HandlerInterceptor）。当 HandlerMapping 返回 HandlerExecutionChain 后，DispatcherServlet 将请求交给定义在 HandlerExecutionChain 中的拦截器和处理器一并处理。

HandlerExecutionChain 是负责处理请求并返回 ModelAndView 的处理执行链，其结构如图 17-20 所示。请求在被 Handler 执行的前后，链中装配的 HandlerInterceptor 会实施拦截操作。

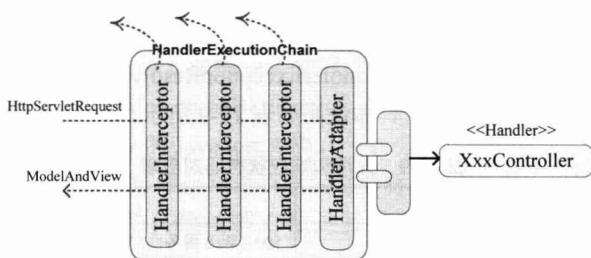


图 17-20 HandlerExecutionChain 的结构

拦截器到底做了什么事情？我们通过考查拦截器的几个接口方法进行了解。

- ❑ `boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)`: 在请求到达 Handler 之前，先执行这个前置处理方法。当该方法返回 `false` 时，请求直接返回，不会传递到链中的下一个拦截器，更不会传递到处理器链末端的 Handler 中。只有返回 `true` 时，请求才向链中的下一个处理节点传递。
- ❑ `void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView)`: 在请求被 HandlerAdapter 执行后，执行这个后置处理方法。
- ❑ `void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)`: 在响应已经被渲染后，执行该方法。

位于处理器链末端的是一个 Handler，DispatcherServlet 通过 HandlerAdapter 适配器对 Handler 进行封装，并按统一的适配器接口对 Handler 处理方法进行调用。

```
<mvc:interceptors>
  <mvc:interceptor>
    <mapping path="/secure/*"/>
    <bean class="com.smart.web.MyInterceptor" />
  </mvc:interceptor>
</mvc:interceptors>
```

可以在 `smart-servlet.xml` 中配置多个拦截器，每个拦截器都可以指定一个匹配的映射路径，以限制拦截器的作用范围。

17.8.3 异常处理

Spring MVC 通过 `HandlerExceptionResolver` 处理程序的异常，包括处理器映射、数据绑定及处理器执行时发生的异常。`HandlerExceptionResolver` 仅有一个接口方法。

```
ModelAndView resolveException(HttpServletRequest request, HttpServletResponse response,
    Object handler, Exception ex)
```

当发生异常时，Spring MVC 将调用 `resolveException()` 方法，并转到 `ModelAndView` 对应的视图中，作为一个异常报告页面反馈给用户。

`HandlerExceptionResolver` 拥有 4 个实现类，分别是 `DefaultHandlerExceptionResolver`、`SimpleMappingExceptionHandler`、`AnnotationMethodHandlerExceptionResolver` 及 `ResponseStatusExceptionHandler`。

1. DefaultHandlerExceptionResolver

Spring MVC 默认装配了 `DefaultHandlerExceptionResolver`，它会将 Spring MVC 框架的异常转换为相应的响应状态码，具体说明如表 17-9 所示。

表 17-9 异常和响应状态码对应表

异常类型	响应状态码
<code>ConversionNotSupportedException</code>	500 (Web 服务内部错误)
<code>HttpMediaTypeNotAcceptableException</code>	406 (无和请求 <code>accept</code> 匹配的 MIME 类型)

续表

异常类型	响应状态码
HttpMediaTypeNotSupportedException	415（不支持的 MIME 类型）
HttpMessageNotReadableException	400（坏的请求）
HttpMessageNotWritableException	500
HttpRequestMethodNotSupportedException	405（不支持的请求方法）
MissingServletRequestParameterException	400
NoSuchRequestHandlingMethodException	404（找不到匹配的资源）
TypeMismatchException	400

可以在 web.xml 中为响应状态码配置一个对应的页面，代码如下：

```
<error-page>
  <error-code>404</error-code>
  <location>/404.htm</location>
</error-page>
```

2. AnnotationMethodHandlerExceptionResolver

Spring MVC 默认注册了 AnnotationMethodHandlerExceptionResolver，它允许通过 @ExceptionHandler 注解指定处理特定异常的方法，如代码清单 17-57 所示。

代码清单 17-57 UserController.java

```
package com.smart.web;
import org.springframework.web.bind.annotation.ExceptionHandler;

@Controller
@RequestMapping("/user")
public class UserController {
    @RequestMapping(path = "/throwException")①
    public String throwException() {
        if(2>1){
            throw new RuntimeException("ddd");
        }
        return "success";
    }

    @ExceptionHandler(RuntimeException.class)②
    public String handleException(RuntimeException re, HttpServletRequest request) {
        return "forward:/error.jsp";
    }
}
```

①处的处理方法在调用时抛出一个 RuntimeException 异常，它会被处于同一处理器类中②处的 handleException() 方法捕获。@ExceptionHandler 可以指定多个异常，如 @ExceptionHandler({AException.class,BException.class})。异常处理方法的标签非常灵活，请参见 ExceptionHandler 的 Javadoc 文档。

不清楚是 Spring MVC 的 Bug 还是有意为之，标注 @ExceptionHandler 的异常处理方法只能对同一处理类中的其他处理方法进行异常响应处理，笔者现在还受困于 @ExceptionHandler 的这个限制中。

`ResponseStatusExceptionHandler` 和 `ResponseStatusExceptionHandler` 类似，允许通过 `@ResponseStatus` 注解标注一个方法，用于处理特定类型的响应状态码。

3. SimpleMappingExceptionHandler

如果希望对所有异常进行统一处理，则可以使用 `SimpleMappingExceptionHandler`，它将异常类名映射为视图名，即发生异常时使用对应的视图报告异常。

请看下面的异常映射配置片段：

```
<bean id=" handlerExceptionHandler "
    class="org.springframework.web.servlet.handler.SimpleMappingExceptionHandler">
    <property name="exceptionMappings">
        <props>
            <prop key="org.springframework.dao.DataAccessException">dataAccessFailure
        </prop>①
            <prop key="org.springframework.transaction.TransactionException">dataAccessFailure
        </prop>②
        </props>
    </property>
</bean>
```

在①处指定当控制器发生 `DataAccessException` 异常时，使用 `dataAccessFailure` 视图显示。在②处指定当控制器发生 `TransactionException` 异常时，使用 `dataAccessFailure` 视图显示。

当然，用户也可以自己实现 `ExceptionHandler` 覆盖 `resolveException()` 接口方法，编写自己的异常解析器，执行一些特定的工作，如将异常信息保存到数据库中等。编写好自定义的 `ExceptionHandler`，在 `smart-servlet.xml` 中注册成一个 Bean 即可工作。

17.8.4 RequestContextHolder 的使用

在 Spring API 中提供了一个非常便捷的工具类 `RequestContextHolder`，能够在 Controller 中获取 request 和 session 对象。其使用方法如下：

```
HttpServletRequest request = ((ServletRequestAttributes)
    RequestContextHolder.getRequestAttributes()).getRequest();
```

需要注意的是，如果直接使用该工具类，则会抛出一个空指针异常，原因是还需要在 `web.xml` 中配置一个监听器，代码如下：

```
<listener>
    <listener-class>
        org.springframework.web.context.request.RequestContextListener
    </listener-class>
</listener>
```

17.9 小结

Spring MVC 4.0 和早期版本相比有了一个质的飞跃，如更全面和方便地支持

REST 风格的 Web 编程、注解驱动、处理方法签名非常灵活、处理器方法不依赖于 Servlet API 等。

由于 Spring MVC 框架在后台做了很多隐性工作，所以想深入掌握 Spring MVC 4.0 并非易事。本章在学习 Spring MVC 的各项功能的同时，深入内部了解了其后台的运作机理。只有了解了这些运作机理，才能更好地使用这个先进的 MVC 框架。

第 18 章

实战案例开发

本章将带领大家开发一个完整的论坛应用案例，体会实际应用开发所需的各项技术及关注要点。学习完本案例后，读者即可胜任使用 Spring+Hibernate 经典框架开发实际应用的工作。

本章主要内容：

- ◆ 如何通过 UML 图描述应用的需求和设计
- ◆ 对于大型的 Web 应用，应该如何设计类和 Web 目录的结构
- ◆ 如何设计 Web 应用的持久层、服务层和 Web 层
- ◆ 如何测试 Web 应用的持久层、服务层和 Web 层

本章亮点：

- ◆ 如何描述 Web 应用需求和设计
- ◆ 如何对 Web 应用各分层实施单元测试

18.1 论坛案例概述

本章将通过一个具体的“小春论坛”案例开发来讲解使用 Spring+Hibernate 集成框架的开发过程。下面先来了解一下这个论坛的整体功能。

18.1.1 论坛整体功能结构

相信大多数读者都使用过网络论坛，论坛的内容主题可谓五花八门，但对于开发人员来讲，它们的功能却都是相近的。可以通过图 18-1 了解论坛的基本功能组成。

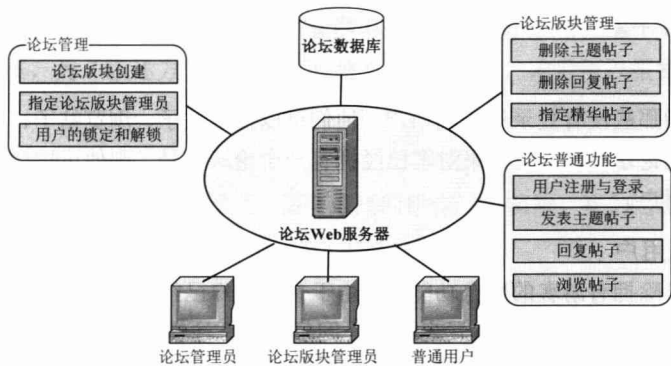


图 18-1 论坛整体功能结构

具体来讲，大部分论坛都包括以下功能模块。

- 论坛普通功能：包括用户注册与登录、发表主题帖子、回复帖子、浏览帖子等功能，这些功能是普通用户所拥有的。
- 论坛版块管理：包括删除主题帖子、删除回复帖子、指定精华帖子等功能，这些功能是论坛版块管理员及论坛管理员所拥有的。
- 论坛管理：包括论坛版块创建、指定论坛版块管理员、用户的锁定和解锁等功能，这些功能是论坛管理员所拥有的。

18.1.2 论坛用例描述

可以将论坛的用户角色划分为 4 种类型：游客、普通用户、论坛版块管理员及论坛管理员。这 4 种类型角色的操作权限是依次递增的，举例来说：所有普通用户拥有的操作功能，论坛版块管理员都拥有；而所有论坛版块管理员拥有的功能，论坛管理员也都拥有。通过如图 18-2 所示的系统用例图来描述这 4 个系统角色和用例的关系。

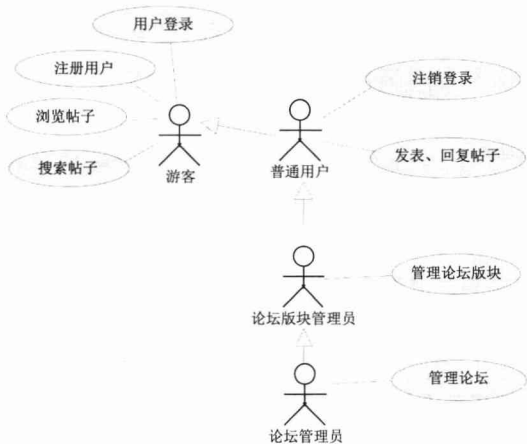


图 18-2 系统一级用例图

下面分别对这些角色及他们的操作功能进行说明。

1. 游客

游客是指那些没有登录论坛的用户，他们可以搜索帖子、浏览帖子、调用论坛注册功能注册成为论坛的用户。如果游客已经拥有一个论坛账号，则他可以登录论坛，成为特定类型的用户。

2. 普通用户

普通用户除拥有游客的所有功能外（由于普通用户已经登录，所以不能进行用户注册和用户登录操作），他还可以发表帖子、回复帖子、注销登录。当用户注销登录后，他就成为游客。普通用户的二级用例图如图 18-3 所示。

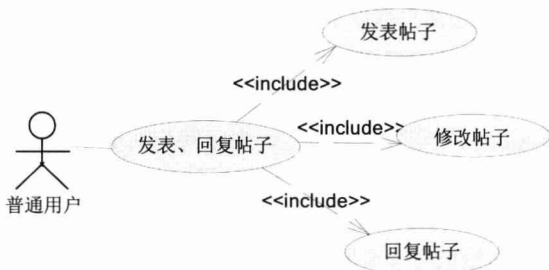


图 18-3 普通用户的二级用例图

3. 论坛版块管理员

一个论坛一般拥有多个论坛版块，每个论坛版块可以拥有一个或多个论坛版块管理员。论坛版块管理员负责管理论坛版块的帖子。论坛版块管理员的二级用例图如图 18-4 所示。

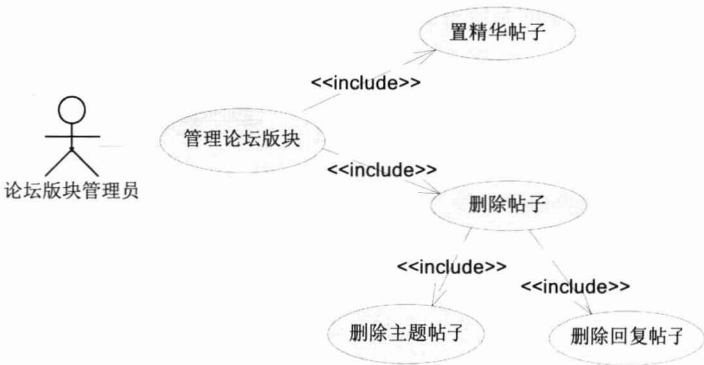


图 18-4 论坛版块管理员的二级用例图

在这里，我们要明确两个概念：主题帖子和回复帖子，前者指用户在论坛版块中发表的一个话题所对应的内容帖，一个话题可以拥有多个跟帖，后者就是回复帖子。所以

删除帖子包括删除主题帖子和删除回复帖子两个功能。置精华帖子的操作对象是主题帖子而非回复帖子，为了叙述方便，我们直接称之为“帖子”，读者可以根据上下文确定具体所指。

4. 论坛管理员

该角色用户是整个论坛的管理员，拥有最高的操作功能权限。我们通过以下二级用例图描述论坛管理员的操作功能，如图 18-5 所示。

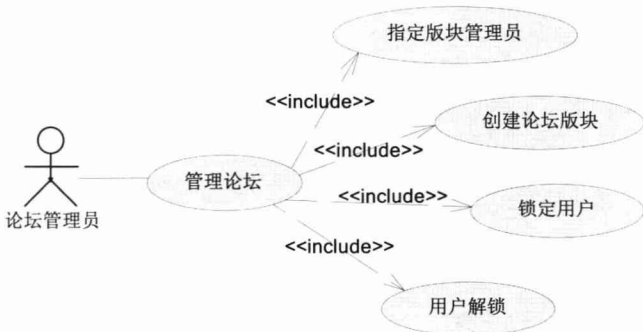


图 18-5 论坛管理员的二级用例图

论坛管理员可以创建一个论坛版块，为论坛版块指定若干个管理员，还可以锁定某些不遵守规则的用户，当然也可以对已经锁定的用户进行解锁。

18.1.3 主要功能流程描述

本小节将对论坛的主要功能进行描述，为页面设计和程序设计提供依据。

1. 用户登录

和第 2 章所讲解的用户登录不同，本案例的用户登录功能涉及的步骤比较复杂，它更接近于真实应用的用户登录功能。可以通过如图 18-6 所示的活动图对用户登录进行具体描述。

用户登录需要判断许多种可能的情况，如用户名不正确、用户密码不正确、用户已经被锁定不允许登录等。在通过以上所有检查后，用户登录才算成功，这时需要完成登录成功后的业务操作，如添加用户积分、记录用户登录日志、更新用户最后登录时间等。为了简化实例功能，这里仅进行添加用户积分的操作。操作完成后，需要将用户对象添加到 HTTP Session 中，以便后续操作可以直接从 Session 中获取用户信息。

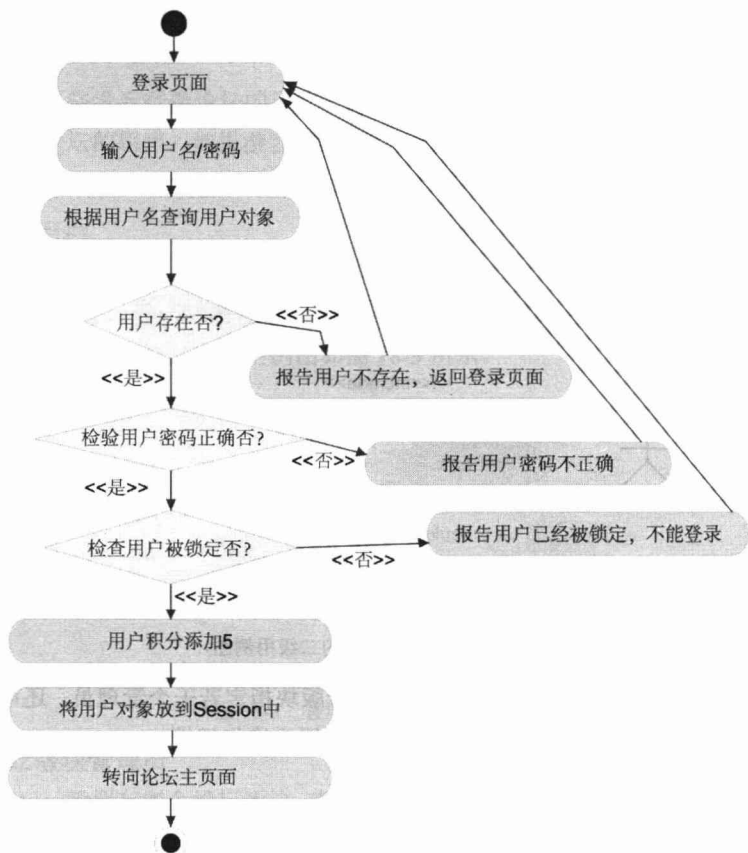


图 18-6 论坛用户登录

2. 发表主题帖子

发表主题帖子功能的整体流程如图 18-7 所示。

在提交并成功保存主题帖子后, 需要进行一些相关的后续操作, 如将论坛板块的帖子数加 1, 给主题帖子作者添加 10 个积分, 然后刷新论坛板块帖子列表。

3. 回复主题帖

回复一个主题帖子即新建一个回复帖子, 这个功能可以通过如图 18-8 所示的活动图进行描述。

和发表主题帖子类似, 回复帖子保存成功后, 还必须进行一些相关的操作, 如用户积分加 5、主题帖子的回复数加 1, 并更新主题帖子的最后回复时间, 以便这个刚被回复过的主题帖子能够排到主题帖子列表的最前面。

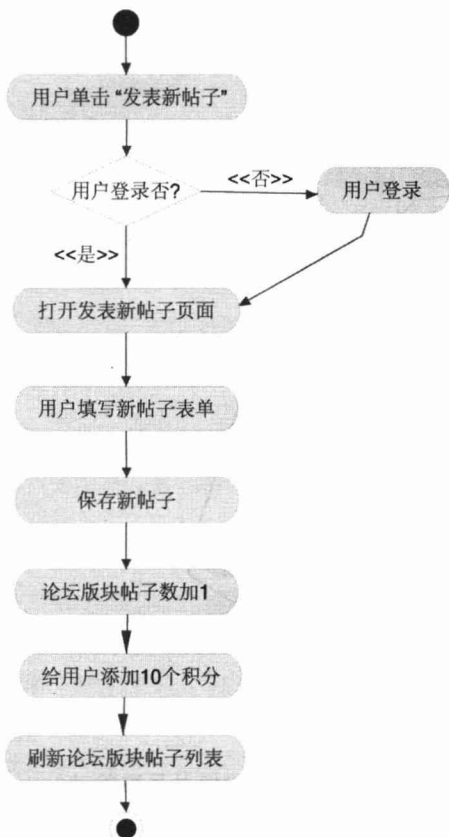


图 18-7 发表主题帖子

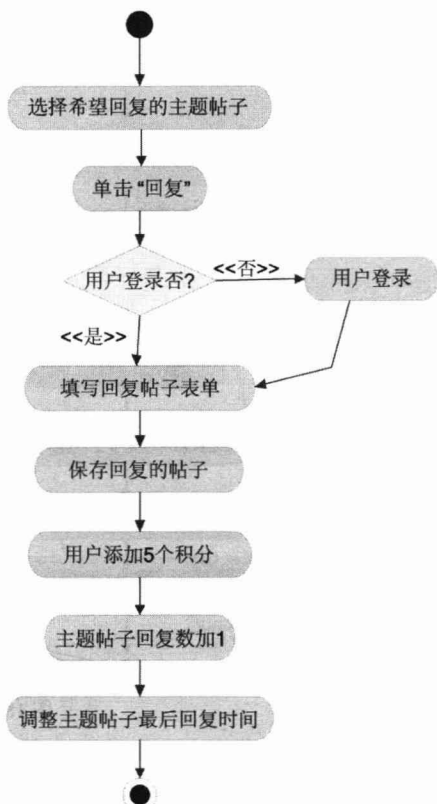


图 18-8 回复主题帖子

4. 删除帖子

论坛版块管理员、论坛管理员都可以删除帖子，包括删除主题帖子和删除回复帖子。可以通过如图 18-9 所示的活动图对删除帖子的整体流程进行描述。

删除帖子基本上是执行发表主题帖子和回复主题帖子的反过程，不过为了防止论坛用户恣意发表一些不合法或垃圾性质的帖子，提高论坛内容的整体质量，被删除帖子的作者需要惩罚性地扣除较多的积分，如回复帖子被删除时将被扣除 20 个积分，而主题帖子被删除时将被扣除 50 个积分。

5. 置精华帖子

论坛管理员、论坛版块管理员可以将一些质量较高的主题帖子置为精华帖子，方便其他用户快速查看到这些帖子，这一功能的操作流程如图 18-10 所示。

当主题帖子被设置为精华帖子时，发表该帖子的作者将被追加 100 个积分，以示奖赏。

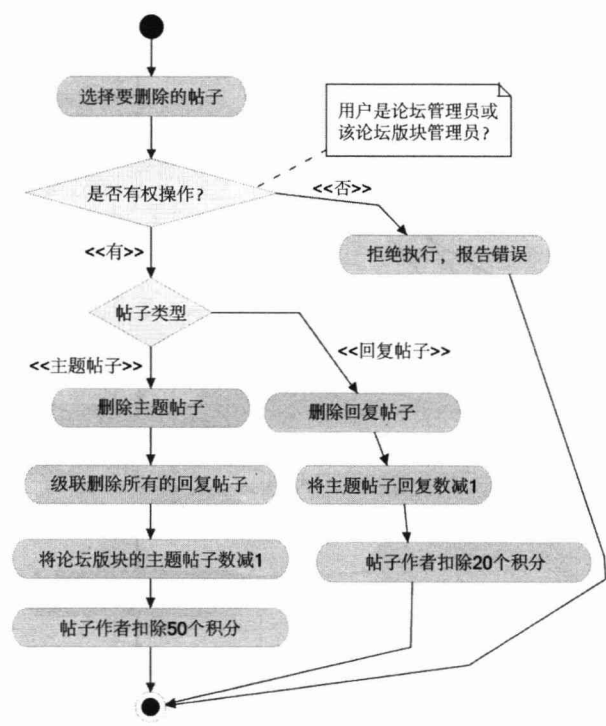


图 18-9 删除帖子

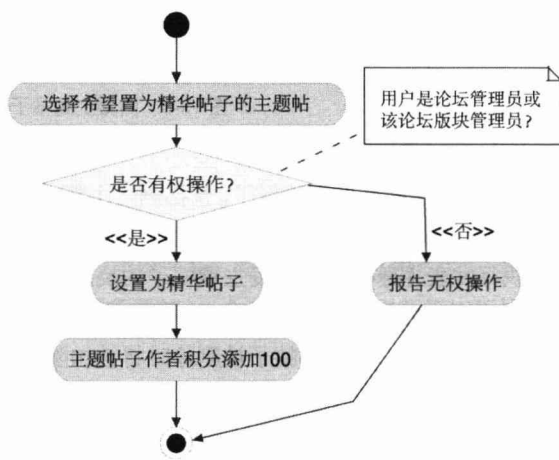


图 18-10 置为精华帖子

6. 指定论坛版块管理员

指定论坛版块管理员的权限由论坛管理员完成，用户可以被指定为若干个论坛版块的管理员，一个论坛版块也可以拥有多个管理员。指定论坛版块管理员功能的操作流程如图 18-11 所示。

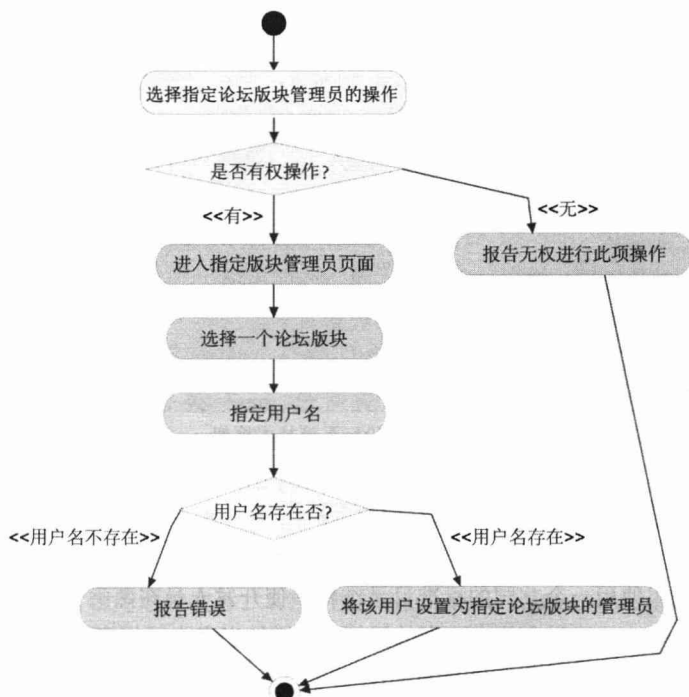


图 18-11 指定论坛版块管理员

论坛管理员从列表选择一个论坛版块，然后输入希望成为该论坛版块管理员的用户名，系统必须判断这个用户名是否存在，如果不存在则必须报告错误，以便论坛管理员调整用户名。

18.2 系统设计

18.2.1 技术框架选择

我们拟使用如图 18-12 所示的技术框架来完成论坛应用程序的开发。为了解决中文乱码问题，我们在 Web 层提供一个字符编码转换过滤器。Web 层使用 Spring MVC 进行请求的处理和响应，视图采用 JSP 2.0 和 JSTL 技术处理。

服务层采用 Spring 4.0，持久层采用 Hibernate 4.2.0，而持久层通过 Spring 提供的支持类集成到 Spring 中。系统严格采取 Web 层、服务层和持久层三层体系结构，上一层的程序可以调用下一层的程序，反之则不行，达到层与层之间松耦合的目的。

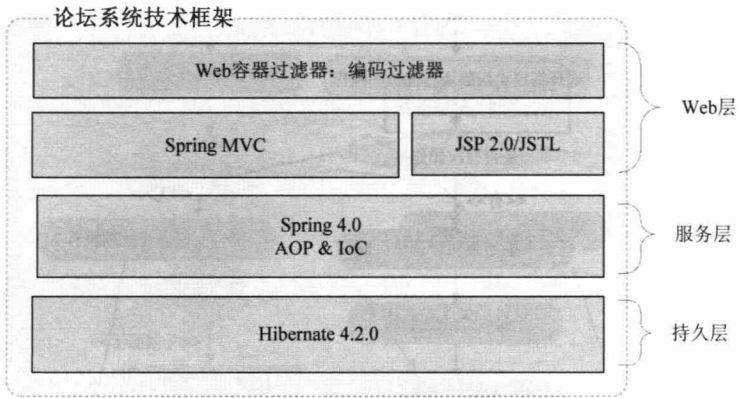


图 18-12 论坛系统技术框架

18.2.2 采用 Maven 构建项目

Maven 提倡使用一个共同的标准目录结构，使开发人员在熟悉了一个 Maven 工程后，对其他的 Maven 工程也能有清晰的了解，这样做也省去了很多设置上的麻烦。本工程使用 Maven 推荐的标准目录结构，如图 18-13 所示。

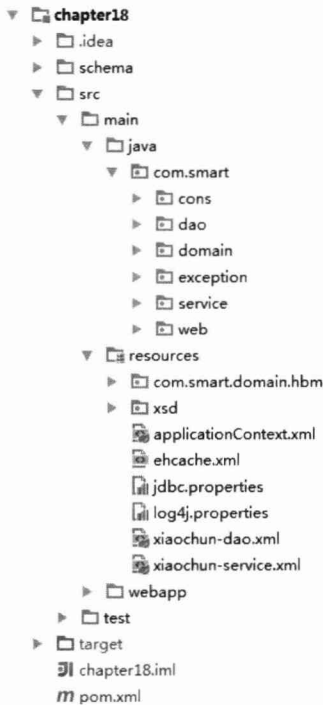


图 18-13 源码包目录结构

所有的源代码文件位于 `src/main` 文件夹中，在 `main` 文件夹中规划了 3 个子文件夹，其中 `resources` 文件夹用于放置系统配置文件，`java` 文件夹用于放置 Java 源代码文件，`webapp` 文件夹用于放置 web 应用的程序文件。所有类位于 `com.smart` 包中，该类包下为每个分层提供了一个相应的类包，如 `dao` 对应持久层的程序，而 `service` 和 `web` 分别对应服务层和 Web 层的程序。由于 PO 会在多个层中出现，因此我们为其提供了一个单独的 `domain` 包。为了避免在程序中直接使用字面值常量，需要通过常量定义的方式予以规避，我们在 `cons` 包中定义了应用级的常量。为了统一管理应用系统异常体系，我们在 `exception` 包中定义了业务异常类及系统异常等。可以在分层包下再按功能模块定义子包，由于我们的论坛应用比较简单，所以每个分层包下不再设子包。

我们为 DAO 和服务类 Bean 分别提供了一个 Spring 配置文件，前者为 `xiaochun-dao.xml`，后者为 `xiaochun-service.xml`。`jdbc.properties` 属性文件提供了数据库连接的信息，这个属性文件将被 `xiaochun-service.xml` 使用。`log4j.properties` 属性文件是 Log4J 的配置文件。我们将这些配置文件直接放置在类路径下。

`webapp` 目录结构很简单，我们将大部分的 JSP 放置在 `WEB-INF/jsp` 目录中，防止用户直接通过 URL 调用这些文件。`WEB-INF/xiaochun-servlet.xml` 为 Spring MVC 的配置文件。如果项目的 JSP 文件数目很多，则可以在 `WEB-INF/jsp` 目录下按功能模块划分为多个子文件夹。一般的 Web 应用都会在 Web 根目录下创建 `images`、`css`、`js` 等文件夹，分别放置图片、CSS 及 JS 的资源文件。我们的论坛应用比较简单，没有这些资源，所以这些文件夹没有出现于目录结构中。

18.2.3 单元测试类包结构规划

规划好程序的类包结构之后，需要根据应用程序分层结构规划相应的单元测试结构。为了使单元测试模块清晰、可读，一般情况下，可以根据应用程序分层建立相应的单元测试目录结构。在 Maven 目录规范中，`test` 和 `main` 为同级目录，如图 18-14 所示。

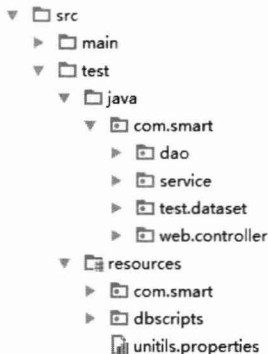


图 18-14 单元测试类包结构

与应用程序代码结构一样，测试所用的源代码文件使用专门的文件夹进行管理，所

有与单元测试相关的文件都放置在 `src/test` 文件夹中, 在 `test` 文件夹中规划了两个子文件夹, 其中 `resources` 文件夹用于放置测试资源文件, `java` 文件夹用于放置测试 Java 源代码文件。所有的测试类位于 `com.smart` 包中, 该类包下为每个分层提供了一个相应的类包, 如 `dao` 对应持久层的测试代码, 而 `service` 和 `web` 分别对应服务层和 Web 层的程序测试代码。

18.2.4 系统架构图

可以将论坛划分为 4 个功能模块, 分别是用户管理、论坛管理、版块管理及论坛基础功能, 如图 18-15 所示。

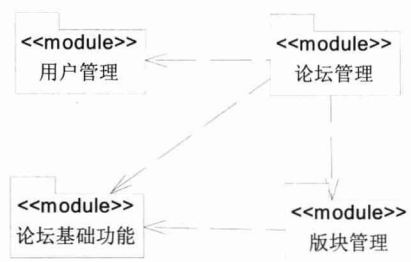


图 18-15 系统架构图

用户管理模块包括用户注册、登录、注销、用户个人信息维护、密码更改等功能（有些功能本案例未实现）；而论坛管理模块包括论坛版块创建、论坛版块管理员指定、用户锁定/解锁等功能；版块管理模块包括主题帖子删除、回复帖子删除、置精华帖子等功能；论坛基础功能模块则包括帖子搜索、论坛版块列表、论坛版块主题帖子列表、帖子浏览、发表主题帖子、发表回复帖子等基础性论坛功能。

18.2.5 PO 类设计

论坛应用共有 7 个 PO 类, 其关系通过类图说明, 如图 18-16 所示。

`BaseDomain` 是所有 PO 类的基类, 它实现了 `Serializable` 接口。所有的 PO 类分别介绍如下。

- ❑ `Board`: 论坛版块 PO 类。
- ❑ `Topic`: 论坛主题 PO 类, 包含主题帖子的作者、所属论坛版块、创建时间、浏览数、回复数等信息, 其中 `mainPost` 对应主题帖子。
- ❑ `Post`: 帖子 PO 类, 一个 `Topic` 拥有一个 `MainPost` (主题帖子), 但拥有若干个 `Post` (回复帖子)。
- ❑ `User`: 论坛用户 PO 类。
- ❑ `LoginLog`: 论坛用户登录日志 PO 类。

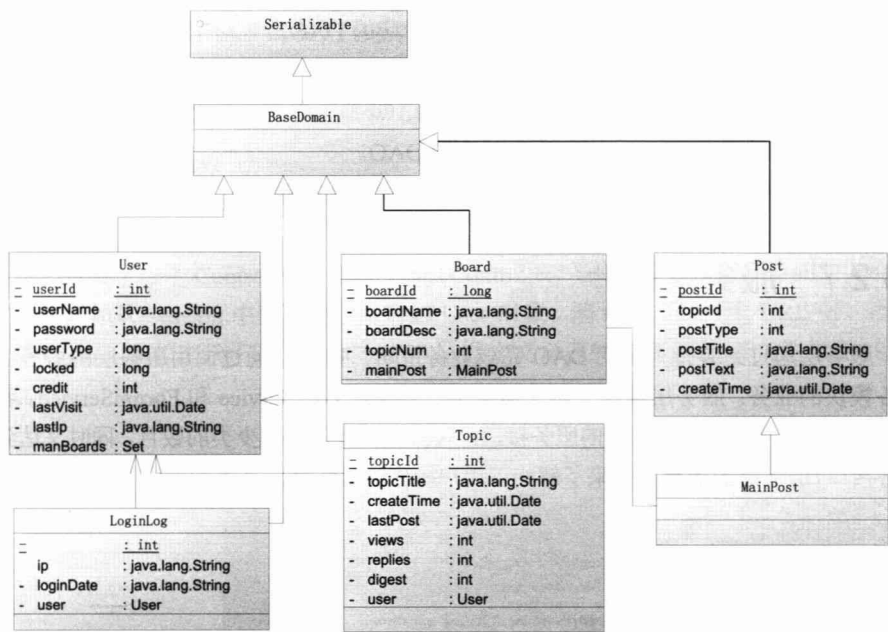


图 18-16 PO 类图

18.2.6 持久层设计

持久层采用 Hibernate 技术，创建所有 DAO 的基类 `BaseDao<T>`，并注入 Spring 为 Hibernate 提供的 `HibernateTemplate` 模板类。`BaseDao<T>` 提供了常见数据的操作方法，子类仅需定义那些个性化的数据操作方法即可。`BaseDao<T>` 使用了 JDK 5.0 泛型的技术，T 为 DAO 操作的 PO 类类型，子类在继承 `BaseDao<T>` 时仅需指定 T 的类型，`BaseDao<T>` 中的方法就可以确定操作的 PO 类型。持久层的类图如图 18-17 所示。

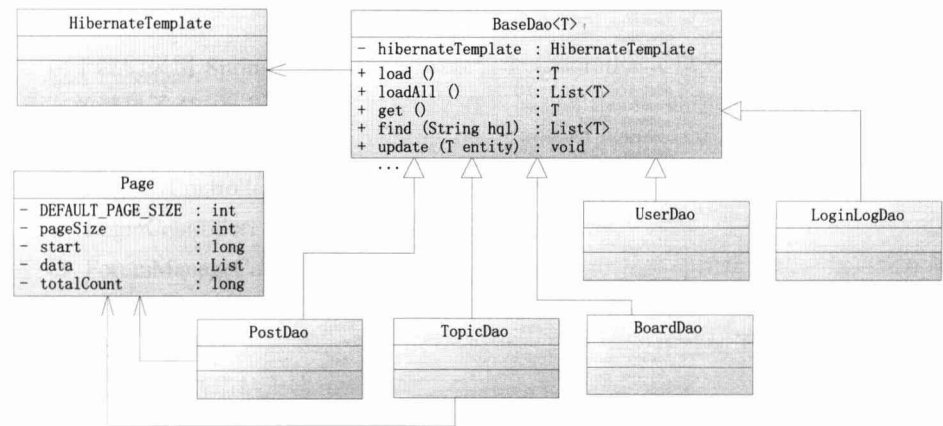


图 18-17 持久层的类图

- ❑ BoardDao: 论坛版块 Board 持久化类对应的 DAO。
- ❑ TopicDao: 主题 Topic 持久化类对应的 DAO。
- ❑ PostDao: 帖子 Post 持久化类对应的 DAO。
- ❑ UserDao: 用户 User 持久化类对应的 DAO。
- ❑ LoginLogDao: 用户登录日志 LoginLog 持久化类对应的 DAO。

18.2.7 服务层设计

服务层通过封装持久层的 DAO 完成业务逻辑，Web 层通过调用服务层的服务类完成各模块的业务。服务层提供了两个服务类，分别是 UserService 和 ForumService。我们直接使用服务类，不提供相应的服务接口，这样可以有效地减少类的数目，同时又达到了服务接口+服务类的效果。先来了解一下用户操作的服务类，其类图如图 18-18 所示。

UserService		
- userDao	: UserDao	
- loginLogDao	: LoginLog	
+ register (User user)		: void
+ update (User user)		: void
+ getUserByUserName (String userName)		: User
+ getUserById (int userId)		: User
+ lockUser (String userName)		: void
+ unlockUser (String userName)		: void
+ queryUserByUserName (String userName)		: List<User>
+ getAllUsers ()		: List<User>
+ loginSuccess (User user)		: void

图 18-18 用户操作的服务类

UserService 通过调用持久层的 UserDao 操作持久化对象，它提供了保存、更新、锁定、解锁等对 User 持久类的操作方法，同时提供了根据用户名或者用户 ID 查询单个用户及根据用户名模糊查询多个用户的方法。

操作论坛版块、主题、帖子等论坛功能使用的服务方法封装在 ForumService 中，图 18-19 即 ForumService 的类图。

ForumService		
- topicDao	: TopicDao	
- boardDao	: BoardDao	
- postDao	: PostDao	
- userDao	: UserDao	
+ addTopic (Topic topic)		: void
+ removeTopic (int topicId)		: void
+ addPost (Post post)		: void
+ removePost (int postId)		: void
+ addBoard (Board board)		: void
+ removeBoard (int boardId)		: void
+ makeDigestTopic (int topicId)		: void
+ getAllBoards ()		: List<Board>
+ getPagedTopics (int boardId, int pageNo, int pageSize)		: Page
+ getPagedPosts (int topicId, int pageNo, int pageSize)		: Page
+ queryTopicByTitle (String title, int pageNo, int pageSize)		: Page
+ getBoardById (int boardId)		: Board
+ getTopicByTopicId (int topicId)		: Topic
+ getPostByPostId (int postId)		: Post
+ addBoardManager (int boardId, String userName)		: void
+ updateTopic (Topic topic)		: void
+ updatePost (Post post)		: void

图 18-19 论坛操作功能服务类

ForumService 类中联合使用了 TopicDao、UserDao、BoardDao 及 PostDao 这 4 个 DAO 类, 利用这些 DAO 类共同完成论坛功能的各项业务操作。这些功能包括论坛管理功能、论坛版块管理功能及发表主题帖子、回复帖子等。

18.2.8 Web 层设计

我们定义了一个 Controller 的基类: BaseController, 它提供了其他 Controller 共有的一些方法, 如从 Session 中获取登录用户的 User 对象、将请求转向一个 URL 等。所有具体的 Controller 都继承于这个 BaseController, 并定义自己的请求方法, 其类图如图 18-20 所示。

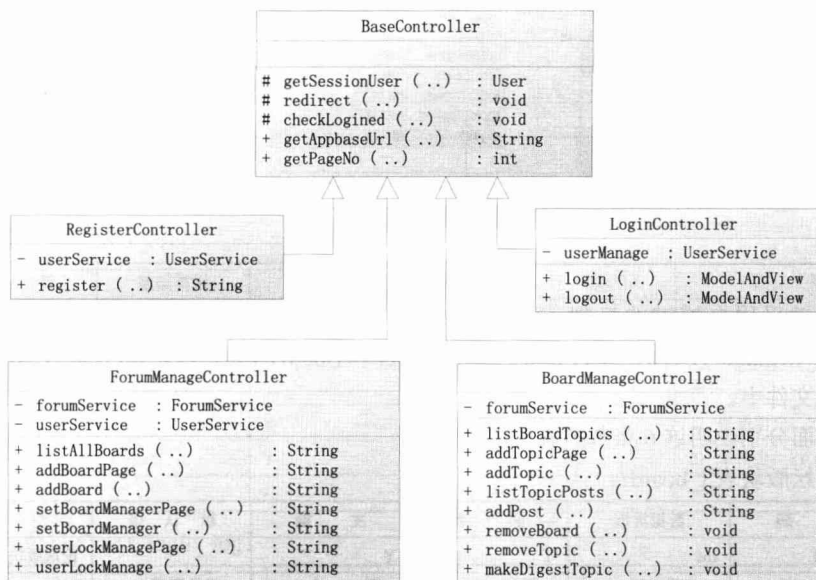


图 18-20 Web 层类图

由于我们采用 Spring 注解 MVC, 所以一个 Controller 可以处理多种不同的请求, 从而有效避免了 Controller 类数量的膨胀。在实际应用中, 可处理多种请求的 Controller 比仅处理一种请求的 Controller 更受青睐。下面对类图中的类分别进行说明。

- ❑ **RegisterController**: 用户注册的控制器。
- ❑ **LoginController**: 用户登录、登录注销的控制器。
- ❑ **ForumManageController**: 论坛管理的控制器, 包括添加论坛版块、指定论坛版块管理员、对用户进行锁定/解锁。
- ❑ **BoardManageController**: 论坛的基本功能, 包括发表主题帖子、回复帖子、删除帖子、置精华帖子等。

18.2.9 数据库设计

论坛应用共包括 6 张数据表，其中 t_board_manager 用于维护 t_board 和 t_user 的多对多关系。表结构如图 18-21 所示。

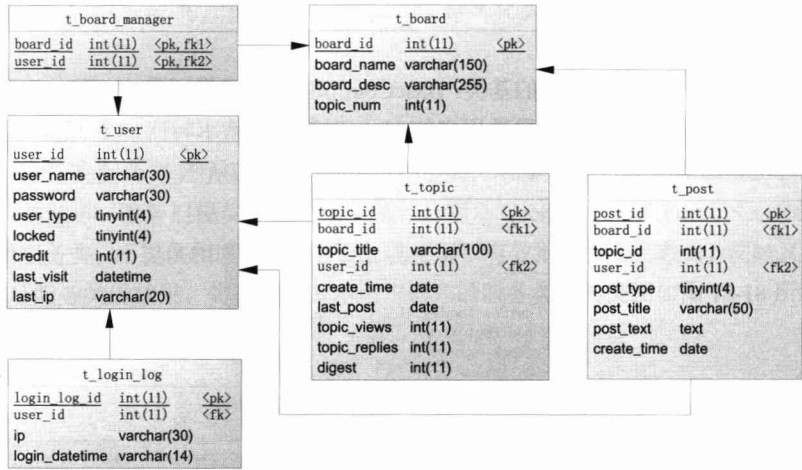


图 18-21 表结构图

主键采用自增键的机制，不采用处键。这些表都可以找到对应的 PO 类，但 t_board_manager 没有对应的 PO 类，它对应 User 和 Board 的多对多关系，反映在 Hibernate 的映射文件中。

下面分别说明这 6 张数据表的字段。

论坛版块表 t_board:

代 码	数据类型	必 填	主 键	默 认 值	注 释
board_id	int	Y	Y		论坛版块 ID
board_name	varchar(150)	Y	N	"	论坛版块名
board_desc	varchar(255)	N	N	NULL	论坛版块描述
topic_num	int	Y	N	0	帖子数目

用户管理版块关联表 t_board_manager:

代 码	数据类型	必 填	主 键	默 认 值	注 释
board_id	int	Y	Y		
user_id	int	Y	Y		

话题表 t_topic:

代 码	数据类型	必 填	主 键	默 认 值	注 释
topic_id	int	Y	Y		帖子 ID
board_id	int	Y	N		所属论坛
topic_title	varchar(100)	Y	N	"	帖子标题，该列建立索引
user_id	int	Y	N	0	发表用户，该列建立索引

续表

代 码	数据类型	必 填	主 键	默 认 值	注 释
create_time	datetime	Y	N		发表时间
last_post	datetime	Y	N		最后回复时间
topic_views	int	Y	N	1	浏览数
topic_replies	int	Y	N	0	回复数
digest	int	Y	N		0: 不是精华话题; 1: 是精华话题

帖子表 t_post:

代 码	数据类型	必 填	主 键	默 认 值	注 释
post_id	int	Y	Y		帖子 ID
board_id	int	Y	N	0	论坛 ID
topic_id	int	Y	N	0	话题 ID, 该列建立索引
user_id	int	Y	N	0	发表者 ID
post_type	tinyint	Y	N	2	帖子类型。1: 主题帖子; 2: 回复帖子
post_title	varchar(50)	Y	N		帖子标题
post_text	text	Y	N		帖子内容
create_time	datetime	Y	N		创建时间

论坛用户表 t_user:

代 码	数据类型	必 填	主 键	默认值	注 释
user_id	int	Y	Y		用户 ID
user_name	varchar(30)	Y	N		用户名, 该列建立索引
password	varchar(30)	Y	N	"	密码
user_type	tinyint	Y	N	1	1: 普通用户; 2: 管理员
locked	tinyint	Y	N	0	0: 未锁定; 1: 锁定
credit	int	N	N		积分

登录日志表 t_login_log:

代 码	数据类型	必 填	主 键	默认值	注 释
login_log_id	int	Y	Y		日志 ID
user_id	int	Y	N	0	发表者 ID
ip	varchar(30)	Y	N		登录 IP
login_datetime	datetime	Y	N		登录时间

18.3 开发前的准备

① 通过 `mysql -uroot -p123456` 登录 MySQL 数据库, 运行 `source <项目地址>/schema/sampled.sql` 脚本创建论坛数据库。该脚本还同时初始化了两个用户: 一个是 `john/1234` (普通用户), 另一个是 `tom/123456` (系统管理员)。

- ② 使用 IDEA 建立一个名为 chapter18 的 Maven 模块，使用 Java 1.8 版本。
- ③ 在 pom.xml 中增加依赖配置。pom.xml 文件可从本书配套网盘中的对应章节获取。

18.4 持久层开发

一般来说，我们将 PO 和 DAO 的类统一划归到持久层中，持久层既负责将 PO 持久化到数据中，也负责从数据库中加载数据到 PO 对象中。

18.4.1 PO 类

所有的 PO 类都直接或间接地继承 BaseDomain 类，这个 PO 基类的代码如代码清单 18-1 所示。

代码清单 18-1 BaseDomain.java

```
package com.smart.domain;
import java.io.Serializable;
import org.apache.commons.lang.builder.ToStringBuilder;

//① 实现了Serializable接口，以便JVM可以序列化PO实例
public class BaseDomain implements Serializable{

    //② 统一的toString()方法
    public String toString() {
        return ToStringBuilder.reflectionToString(this);
    }
}
```

一般情况下，PO 类最好都实现 Serializable 接口，这样 JVM 就能够方便地将 PO 实例序列化到硬盘中，或者通过流的方式进行发送，为缓存、集群等功能带来便利。我们往往需要将 PO 对象打印为一个字符串，这是由对象的 toString()方法来完成的，这里通过 Apache 的 ToStringBuilder 工具类提供统一的实现。

下面先看一下 Board PO 类及 Hibernate JPA 注解配置，如代码清单 18-2 所示。

代码清单 18-2 Board.java

```
package com.smart.domain;
...
@Entity
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
@Table(name = "t_board")
public class Board extends BaseDomain {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "board_id")
    private int boardId;
```

```

@Column(name = "board_name")
private String boardName;

@Column(name = "board_desc")
private String boardDesc;

@Column(name = "topic_num")
private int topicNum ;

//省略属性的get/setter方法
}

```

每个持久化 PO 类都是一个实体 Bean，通过在类的定义中使用@Entity 注解来进行声明；通过@Table 注解为 Board 指定对应数据库表、目录和 schema 的名字；通过@Cache 注解为 Board 设置缓存策略。Hibernate 提供了以下几种缓存策略。

- ❑ CacheConcurrencyStrategy.NONE：不使用缓存。
- ❑ CacheConcurrencyStrategy.READ_ONLY：只读模式，在此模式下，如果对数据进行更新操作，则会产生异常。
- ❑ CacheConcurrencyStrategy.READ_WRITE：读/写模式，该模式在更新缓存的时候会对缓存里面的数据加锁，其他事务如果去取相应的缓存数据，发现被锁了，就会直接去数据库查询。
- ❑ CacheConcurrencyStrategy.NONSTRICT_READ_WRITE：不严格的读/写模式，使用该模式不会对缓存数据加锁。
- ❑ CacheConcurrencyStrategy.TRANSACTIONAL：事务模式，指缓存支持事务，当事务回滚时，缓存也能回滚，目前只支持 JTA 环境。

通过@Id 注解可将 Board 中的 boardId 属性定义为主键，使用@GenerateValue 注解定义主键生成策略（分别是 AUTO、TABLE、IDENTITY、SEQUENCE）。通过@Column 注解将 Board 的各个属性映射到数据库表 t_board 中相应的列。

下面再来看一下 Post PO 类及 Hibernate JPA 注解配置，如代码清单 18-3 所示。

代码清单 18-3 Post.java

```

package com.smart.domain;
...
@Entity
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
@Table(name = "t_post")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "post_type", discriminatorType =
DiscriminatorType.STRING)
@DiscriminatorValue("1")
public class Post extends BaseDomain {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "post_id")
    private int postId;
}

```

```

@Column(name = "post_title")
private String postTitle;

@Column(name = "post_text")
private String postText;

@Column(name = "board_id")
private int boardId;

@Column(name = "create_time")
private Date createTime;

@ManyToOne
@JoinColumn(name = "user_id")
private User user;

@ManyToOne(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
@JoinColumn(name="topic_id")
private Topic topic;

//省略属性的get/setter方法
}

```

Post（回复的帖子）和其子类 MainPost（主题帖）都映射到 t_post 表中，t_post 表通过 post_type 字段值分别两者。当 post_type=1 时，对应 MainPost；当 post_type=2 时，对应 Post。

通过 @Inheritance 注解指定了 PO 映射继承关系。Hibernate 共提供了 3 种方式：每个类一张表（InheritanceType.TABLE_PER_CLASS）、连接的子类（InheritanceType.JOINED）和每个类层次结构一张表（InheritanceType.SINGLE_TABLE）。通过 @DiscriminatorColumn 注解定义了辨别符列。对于继承层次结构中的每个类，@DiscriminatorValue 注解指定了用来辨别该类的值。辨别符列名字默认为 DTYPE，其默认值为实体名，类型为 DiscriminatorType.STRING。

通过 @ManyToOne 注解定义了对多一关系，通过 @JoinColumn 注解定义了对多一的关联关系。如果没有 @JoinColumn 注解，则系统自动处理，在主表中将创建连接列，列名为“主题的关联属性名 + 下划线 + 被关联端的主键列名”。

其他的 PO 类和 Board、Post 类似，都是一堆属性的集合，并通过 JPA 注解配置 PO 类与数据表的映射关系，这里就不一一列出阐述。

18.4.2 DAO 基类

1. DAO 基类的基本方法

在编写完 PO 类及相应的 hbm 映射文件后，我们着手编写负责持久化 PO 和查找 PO 的 DAO 类。由于每个 PO 的 DAO 类都需要执行一些相同的操作，如保存、更新、删除 PO 及根据 ID 加载 PO 等，所以可以编写一个提供这些通用操作的基类，让所有 PO 的 DAO 类都继承这个 DAO 基类，其代码如代码清单 18-4 所示。

代码清单 18-4 BaseDao.java

```

package com.smart.dao;
import org.hibernate.Query;
...

// DAO基类, 其他DAO可以直接继承这个DAO, 不但可以复用共用的方法, 还可以获得泛型的好处
public class BaseDao<T>{
    private Class<T> entityClass;
    @Autowired
    private HibernateTemplate hibernateTemplate;

    //通过反射获取子类确定的泛型类
    public BaseDao() {
        Type genType = getClass().getGenericSuperclass();
        Type[] params = ((ParameterizedType) genType).getActualTypeArguments();
        entityClass = (Class) params[0];
    }

    //根据ID加载PO实例
    public T load(Serializable id) {
        return (T) getHibernateTemplate().load(entityClass, id);
    }

    //根据ID获取PO实例
    public T get(Serializable id) {
        return (T) getHibernateTemplate().get(entityClass, id);
    }

    //获取PO的所有对象
    public List<T> loadAll() {
        return getHibernateTemplate().loadAll(entityClass);
    }

    //保存PO
    public void save(T entity) {
        getHibernateTemplate().save(entity);
    }

    //删除PO
    public void remove(T entity) {
        getHibernateTemplate().delete(entity);
    }

    //更改PO
    public void update(T entity) {
        getHibernateTemplate().update(entity);
    }

    //执行HQL查询
    public List find(String hql) {
        return this.getHibernateTemplate().find(hql);
    }

    //执行带参的HQL查询
    public List find(String hql, Object... params) {

```

```

        return this.getHibernateTemplate().find(hql,params);
    }
    //对延迟加载的实体PO进行初始化
    public void initialize(Object entity) {
        this.getHibernateTemplate().initialize(entity);
    }
    ...
}

```

基类直接注入 Spring 为 Hibernate 提供的 `HibernateTemplate` 模板操作类，这样就可以借由这个 `HibernateTemplate` 模板操作类执行 Hibernate 的各项操作了。读者可能已经注意到基类的类名 (`BaseDao<T>`) 使用了 JDK 5.0 的泛型技术，这是为了让 DAO 子类可以使用泛型技术绑定特定类型的 PO 类，避免强制类型转换带来的麻烦。通过扩展这个 DAO 基类，DAO 子类仅需要声明泛型对应的 PO 类并实现那些非通用性的方法即可，大大减少了 DAO 子类的代码量。

2. 对数据分页的支持

除此以外，我们还在 `BaseDao` 中提供了对数据分页的支持。代码清单 18-5 列出了 `BaseDao` 中和数据分页相关的一些方法。

代码清单 18-5 BaseDao.java

```

package com.smart.dao;
import java.util.*;
import org.hibernate.Query;
...
public class BaseDao<T>{
    ...
    //分页查询函数，使用HQL
    public Page pagedQuery(String hql, int pageNo, int pageSize, Object... values) {
        Assert.hasText(hql);
        Assert.isTrue(pageNo >= 1, "pageNo should start from 1");
        // Count查询
        String countQueryString = "select count (*) " + removeSelect(removeOrders(hql));
        List countlist = getHibernateTemplate().find(countQueryString, values);
        long totalCount = (Long) countlist.get(0);

        if (totalCount < 1)
            return new Page();
        // 实际查询返回分页对象
        int startIndex = Page.getStartOfPage(pageNo, pageSize);
        Query query = createQuery(hql, values);
        List list = query.setFirstResult(startIndex).setMaxResults(pageSize).list();
        return new Page(startIndex, totalCount, pageSize, list);
    }

    //创建Query对象
    public Query createQuery(String hql, Object... values) {
        Assert.hasText(hql);
        Query query = getSession().createQuery(hql);
        for (int i = 0; i < values.length; i++) {
            query.setParameter(i, values[i]);
        }
    }
}

```



```

        return query;
    }

    //去除HQL的select子句
    private static String removeSelect(String hql) {
        Assert.hasText(hql);
        int beginPos = hql.toLowerCase().indexOf("from");
        Assert.isTrue(beginPos != -1, "hql : " + hql + " must has a keyword 'from'");
        return hql.substring(beginPos);
    }

    //去除HQL的orderby子句
    private static String removeOrders(String hql) {
        Assert.hasText(hql);
        Pattern p = Pattern.compile("order\\s*by[\\w|\\W|\\s|\\S]*", Pattern.CASE_
INSENSITIVE);
        Matcher m = p.matcher(hql);
        StringBuffer sb = new StringBuffer();
        while (m.find()) {
            m.appendReplacement(sb, "");
        }
        m.appendTail(sb);
        return sb.toString();
    }
    ...
}

```

这样，仅需提供 HQL 及分页的一些配置信息，就可以获取特定页面的数据。特定页面的信息通过 Page 类进行表达。下面来看一下 Page 类的代码，如代码清单 18-6 所示。

代码清单 18-6 Page.java

```

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
...
//分页对象，包含当前页数据及分页信息，如总记录数
public class Page implements Serializable {
    private static int DEFAULT_PAGE_SIZE = 20;
    private int pageSize = DEFAULT_PAGE_SIZE; // 每页的记录数
    private long start; // 当前页第一条数据在List中的位置，从0开始
    private List data; // 当前页中存放的记录，类型一般为List
    private long totalCount; // 总记录数

    //构造方法，只构造空页
    public Page() {
        this(0, 0, DEFAULT_PAGE_SIZE, new ArrayList());
    }

    //默认的构造方法
    public Page(long start, long totalSize, int pageSize, List data) {
        this.pageSize = pageSize;
        this.start = start;
        this.totalCount = totalSize;
        this.data = data;
    }
}

```



```

//取总页数
public long getTotalPageCount() {
    if (totalCount % pageSize == 0)
        return totalCount / pageSize;
    else
        return totalCount / pageSize + 1;
}

//取该页当前页码, 页码从1开始
public long getCurrentPageNo() {
    return start / pageSize + 1;
}

//该页是否有下一页
public boolean hasNextPage() {
    return this.getCurrentPageNo() < this.getTotalPageCount();
}

//该页是否有上一页
public boolean hasPreviousPage() {
    return this.getCurrentPageNo() > 1;
}

//获取任意一页第一条数据在数据集中的位置, 每页条数使用默认值
protected static int getStartOfPage(int pageNo) {
    return getStartOfPage(pageNo, DEFAULT_PAGE_SIZE);
}

//获取任意一页第一条数据在数据集中的位置
public static int getStartOfPage(int pageNo, int pageSize) {
    return (pageNo - 1) * pageSize;
}
...
}

```

这个 Page 分页类持有两部分信息：一部分信息是分页数据，另一部分信息是分页控制信息，如每页的数据行数、当前的页码、总页数等。也就是说，当调用分页查询方法时，将返回包含业务数据和分页信息的 Page 对象，而非仅包含业务数据的 List 对象，这一点值得读者注意。

18.4.3 通过扩展基类定义 DAO 类

来看一下 BoardDao 类的代码，如代码清单 18-7 所示。

代码清单 18-7 BoardDao.java

```

package com.smart.dao;
import java.util.Iterator;
import com.smart.domain.Board;
...
//① 扩展BaseDao, 并确定泛型的类为Board

```

```

@Repository
public class BoardDao extends BaseDao<Board>{
    private static final String GET_BOARD_NUM = "select count(f.boardId) from Board f";

    //② 获取论坛版块数目的方法
    public long getBoardNum() {
        Iterator iter = getHibernateTemplate().iterate(GET_BOARD_NUM);
        return ((Long)iter.next());
    }
}

```

BoardDao 是操作 Board 的 DAO 类，它扩展于 BoardDao<T>，同时指定泛型类型 T 为 Board，这样在基类中定义的 save(T obj)和 update(T obj)等通用方法的入参就确定为 Board。由于通用性的方法已经在基类中实现，所以 BoardDao 仅需实现一个非通用性的 getBoardNum()方法就可以了，这个方法返回所有论坛版块的数目。

再来看一下 TopicDao 类的代码，如代码清单 18-8 所示。

代码清单 18-8 TopicDao.java

```

package com.smart.dao;
import java.util.List;
import com.smart.domain.Topic;
@Repository
public class TopicDao extends BaseDao<Topic> {
    private static final String GET_BOARD_DIGEST_TOPICS = "from Topic t where t.boardId = ?
        and digest > 0 order by t.lastPost desc,digest desc";
    private static final String GET_PAGED_TOPICS = "from Topic where boardId = ?
        order by lastPost desc";
    private static final QUERY_TOPIC_BY_TITILE = "from Topic where topicTitle like ?
        order by lastPost desc";

    //①获取论坛版块某一页的精华主题帖子，按最后回复时间及精华级别降序排列
    public Page getBoardDigestTopics(int boardId,int pageNo,int pageSize){
        return pagedQuery(GET_BOARD_DIGEST_TOPICS,pageNo,pageSize, boardId);
    }

    //② 获取论坛版块某一页的主题帖子
    public Page getPagedTopics(int boardId,int pageNo,int pageSize) {
        return pagedQuery(GET_PAGED_TOPICS,pageNo,pageSize, boardId);
    }

    //③ 获取和主题帖子标题模糊匹配的主题帖子（某一页的数据）
    public Page queryTopicByTitle(String title, int pageNo, int pageSize) {
        return pagedQuery(QUERY_TOPIC_BY_TITILE,pageNo,pageSize);
    }
}

```

由于需要列出一个论坛版块的主题帖子，所以我们提供了 getPagedTopics(int boardId,int pageNo,int pageSize)方法，而 getBoardDigestTopics(int boardId)方法用于获取论坛版块精华主题帖子。由于用户需要以关键字为条件查询匹配的帖子，所以我们提供了一个对主题帖子的标题执行模糊查询的 queryTopicByTitle(String title, int pageNo, int pageSize)方法。由此可以知道，DAO 的方法需要根据具体的业务需求确定，它为服务

层的 Service 类提供数据获取的实现。DAO 层还有另外 3 个 DAO 类，分别是 UserDao、LoginLogDao 和 PostDao，读者可以通过本章案例查看它们的具体代码。

18.4.4 DAO Bean 的装配

在完成 DAO 的开发后，需要在 Spring 配置文件中将它们定义为 Bean。我们在 src/main/resources 目录下创建一个用于配置 DAO 的 Spring 配置文件 xiaochun-dao.xml。在定义这些 DAO 之前，需要先定义一些基础设施，如数据源、HibernateTemplate 等，如代码清单 18-9 所示。

代码清单 18-9 xiaochun-dao.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:p="http://www.springframework.org/schema/p"
  ...>
  <!-- 扫描com.smart.dao包下所有标注@Repository的DAO组件 -->
  <context:component-scan base-package="com.smart.dao"/>

  <!--① 引入定义JDBC连接的属性文件-->
  <context:property-placeholder location="classpath:jdbc.properties"/>

  <!--② 定义一个数据源-->
  <bean id="dataSource"
    class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close"
    p:driverClassName="${jdbc.driverClassName}"
    p:url="${jdbc.url}"
    p:username="${jdbc.username}"
    p:password="${jdbc.password}" />

  <!--③ 定义Hibernate的SessionFactory-->
  <bean id="sessionFactory"
    class="org.springframework.orm.hibernate4.annotation.AnnotationSession
FactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="packagesToScan"><!--③-1 扫描基于JPA注解的PO类目录-->
      <list>
        <value>com.smart.domain</value>
      </list>
    </property>
    <!--③-2 指定Hibernate的属性信息-->
    <property name="hibernateProperties">
      <props>
        <!--③-2-1 指定数据库的类型为MySQL-->
        <prop key="hibernate.dialect">
          org.hibernate.dialect.MySQLDialect
        </prop>
      </props>
    </property>
  </bean>
```

```

        <!--③-2-2 在提供数据库操作里显示SQL,
        方便开发期的调试, 在部署时应该将其设计为false-->
        <prop key="hibernate.show_sql">true</prop>
    </props>
</property>
</bean>

<!--④ 定义HibernateTemplate-->
<bean id="hibernateTemplate"
    class="org.springframework.orm.hibernate4.HibernateTemplate"
    p:sessionFactory-ref="sessionFactory" />
</beans>

```

在①处引入了一个外部的属性文件，这个属性文件定义了 JDBC 连接的相关信息，其内容如代码清单 18-10 所示。

代码清单 18-10 jdbc.properties

```

#Mysql
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost/sampledb?useUnicode=true&characterEncoding=UTF-8
jdbc.username=root
jdbc.password=123456

```

在②处定义了一个数据源，需要特别指出的是 jdbc.url 属性，它附带了两个参数：useUnicode=true&characterEncoding=UTF-8，这两个参数告诉 JDBC 在和 MySQL 数据库通信时需要使用特定的编码。这是因为 JDBC 在默认情况下会采用操作系统的默认编码和数据库通信（如中文操作系统一般是 GBK），如果数据库采用的编码不是系统默认的编码，则需要显式指定通信的编码格式，否则就会产生中文乱码问题。由于我们的数据库采用了 UTF-8 编码格式，所以需要添加这两个参数。从最优项目实践来讲，这种直接在 Spring 中定义数据源的方式并不是最佳方式，至于如何优化改进，请参见 18.11 节。

在③处定义了一个 Hibernate Session 工厂，这个 Session 工厂 Bean 需要使用数据源，此外还必须为其指定 Hibernate 注解包扫描路径。由于论坛的 PO 是基于 Hibernate JPA 注解的，所以仅需定义 packagesToScan 属性就可以了。Hibernate 的配置文件可以定义诸多 Hibernate 属性，在 Spring 中可以通过 hibernateProperties 定义这些属性，这里仅定义了两个 Hibernate 属性。

在④处定义了一个 HibernateTemplate 的实现，HibernateTemplate 是 Spring 提供的旨在简化 Hibernate API 调用的模板类。

18.4.5 使用 Hibernate 二级缓存

Hibernate 拥有一级和二级缓存。一级缓存是由 Session 实现的，天生拥有并且不可拆卸，所以无须关注。Hibernate 使用插件的方式实现二级缓存，在默认情况下，二级缓存是关闭的。合理地使用二级缓存可以有效地减少对数据库的访问次数，提升应用的整体性能。

对于一个版块 Board 对象来说,其实例数目比较少且不常发生更改,User 对象的实例数目比较多但也不经常发生变化,而 Topic 和 Post 的实例数目比较多且较常发生变化。我们将根据这些 PO 的特点使用不同的缓存策略。

配置二级缓存主要有两个步骤。

① 选择需要使用的第三方二级缓存组件(如 EHCache、MemCached 等),在基于 JPA 注解的实体对象或 SessionFactory 的配置中定义缓存策略。

② 配置所选第三方缓存组件的配置文件。每种缓存组件都有自己的配置文件,因此需要手工编辑它们的配置文件,并将它们放置在类路径下。对于 EHCache 来说,其配置文件为 ehcache.xml,而 JBossCache 的配置文件为 treecache.xml。

我们采用 EHCache 缓存实现方案,首先通过 SessionFactory 的配置启用二级缓存并定义缓存策略,这需要调整 xiaochun-dao.xml 对于 sessionFactory Bean 的配置,如代码清单 18-11 所示。

代码清单 18-11 xiaochun-dao.xml

```
...
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
    ...
    <property name="hibernateProperties">
        <props>
            ...
            <!--① 采用EHCache缓存实现方案-->
            <prop key="hibernate.cache.region.factory_class">
                org.hibernate.cache.ehcache.EhCacheRegionFactory
            </prop>
            <prop key="hibernate.cache.use_query_cache">false</prop>
        </props>
    </property>
</bean>
...
```

在①处启用了二级缓存,通过 hibernate.cache.region.factory_class 指定了缓存实现类。在这里,我们采用 EHCache 缓存实现方案。

最后还需要配置 EHCache 的配置文件,将其命名为 ehcache.xml 并放置到类路径下(chapter18/ src/main/resources/)。来看一下 ehcache.xml 的配置内容,如代码清单 18-12 所示。

代码清单 18-12 ehcache.xml

```
<ehcache>
    <diskStore path="java.io.tmpdir" />
    <defaultCache maxElementsInMemory="10000" eternal="false"
        overflowToDisk="false" timeToIdleSeconds="0" timeToLiveSeconds="0"
        diskPersistent="false" diskExpiryThreadIntervalSeconds="120" />
    <!--①存放Board的缓存区-->
    <cache name="fixedRegion" maxElementsInMemory="100"
        eternal="true" overflowToDisk="false"/>
...
```

```
<!--② 存放User、Topic和Post的缓存区-->
<cache name="freqChangeRegion" maxElementsInMemory="5000" eternal="false"
        overflowToDisk="true" timeToIdleSeconds="300" timeToLiveSeconds="1800"/>
</ehcache>
```

在①处定义的 `fixedRegion` 缓存区不使用硬盘缓存，所有对象都在内存中，缓存区中的对象永不过期，这非常适合缓存类似于 `Board` 的实例。在②处定义的 `freqChangeRegion` 缓存区使用硬盘缓存，对象在闲置 300 秒后就从缓存中清除，且对象的最大存活期限为 30 分钟，缓存区中最大的缓存实例个数为 5000 个，超出此限制的实例将被写到硬盘中。这样就完成了 Hibernate 二级缓存的所有配置，当启用 Spring 时，二级缓存就开始工作了。

需要注意的是，上述配置的 EHCACHE 只支持单机缓存（更改 `ehcache.xml` 配置，可以支持分布式集群），也就是在集群环境中，每个应用节点的缓存都是相互独立、无法共享的，这导致缓存的命中率不高。如果 Hibernate 二级缓存要运行在集群环境中，则需要第三方缓存组件支持集群能力，目前比较常用的有 EHCACHE、MemCached、JBossCache。其中 EHCACHE 和 JBossCache 是基于 Java 语言的高效缓存组件，它们都支持分布式集群，支持多种方式（如 JGroup、JMS、RMI）进行应用节点缓存的同步，缓存可以存储在内存或硬盘中。MemCached 服务器端是基于 C 语言的高性能集中式缓存组件，客户端支持多种语言（如 Java、C、PHP 等），缓存只能存储在内存中。与分布式缓存不同的是，集中式缓存为每个应用节点提供了统一的缓存服务，因此，每个应用节点不涉及缓存同步问题。

18.5 对持久层进行测试

按照 Kent Back 的观点，单元测试的重要特性之一应该是可重复性。不可重复的单元测试是没有价值的。因此，好的单元测试应该具备独立性和可重复性。DAO 层因为是和数据库打交道的层，所以 DAO 层的单元测试依赖于数据库中的数据。要实现 DAO 层单元测试的可重复性，就需要对每次因单元测试引起的数据库中的数据变化进行还原，也就是保护单元测试数据库的数据现场。Spring 测试框架并不能很好地解决所有问题，要解决这些问题，必须整合多方资源。下面使用第 20 章介绍的 DbUnit、Unitils 等测试框架来测试论坛 DAO 层。

18.5.1 配置 Unitils 测试环境

首先在测试资源目录 `src/test/resources` 中创建一个项目级别的 `unitils.properties` 配置文件，并对 Unitils 进行相应的配置，如 Unitils 模块、数据库连接信息、数据维护策略等。详细的配置如代码清单 18-13 所示。

代码清单 18-13 unitils.properties

```
#① 启用Unitils所需模块
unitils.modules=database,dbunit,hibernate,spring

#② 配置数据库连接
database.driverClassName=com.mysql.jdbc.Driver
database.url=jdbc:mysql://localhost:3306/sampled?useUnicode=true&characterEncoding=UTF-8
database.dialect = mysql
database.userName=root
database.password=123456
database.schemaNames=sampled

#③ 配置数据库维护策略
updateDataBaseSchema.enabled=true

#④ 配置数据库表创建策略
dbMaintainer.autoCreateExecutedScriptsTable=true
dbMaintainer.script.locations=D:/masterSpring/chapter18/src/test/resources/dbscripts

#⑤配置数据集工厂
DbUnitModule.DataSet.factory.default=com.smart.test.dataset.excel
                                     <img alt="icon" data-bbox="548 443 568 458"/>.MultiSchemaXlsDataSetFactory
DbUnitModule.ExpectedDataSet.factory.default=com.smart.test.dataset.excel
                                     <img alt="icon" data-bbox="548 473 568 488"/>.MultiSchemaXlsDataSetFactory
```

对 DAO 层进行测试,需要与测试数据库、持久层 Hibernate 框架、运行 Spring 容器集成,因此在①处配置 Unitils 的加载模块有 database、dbunit、hibernate、spring。在②处配置测试数据库的连接信息。在③处和④处配置测试数据库的维护策略。在⑤处准备数据集及验证数据集工厂。

18.5.2 准备测试数据库及测试数据

配置好 Unitils 加载模块、测试数据库连接信息、数据库维护策略之后,开始测试数据库及测试数据准备工作。首先在测试目录 src/test/resources/dbscripts 中创建一个数据库创建脚本文件 001_create_sampledb.sql,里面分别是创建论坛版块表 t_board、帖子表 t_post、话题表 t_topic 等创建数据库脚本信息,如代码清单 18-14 所示。

代码清单 18-14 001_create_sampledb.sql

```
CREATE TABLE t_board (
    board_id int(11) NOT NULL auto_increment COMMENT '论坛版块ID',
    board_name varchar(150) NOT NULL default '' COMMENT '论坛版块名',
    board_desc varchar(255) default NULL COMMENT '论坛版块描述',
    topic_num int(11) NOT NULL default '0' COMMENT '帖子数目',
    PRIMARY KEY (board_id),
    KEY AK_Board_NAME (board_name)
) ENGINE=InnoDB AUTO_INCREMENT=9 DEFAULT CHARSET=utf8;

CREATE TABLE t_post (
```



```

post_id int(11) NOT NULL auto_increment COMMENT '帖子ID',
board_id int(11) NOT NULL default '0' COMMENT '论坛ID',
topic_id int(11) NOT NULL default '0' COMMENT '话题ID',
user_id int(11) NOT NULL default '0' COMMENT '发表者ID',
post_type tinyint(4) NOT NULL default '2' COMMENT '帖子类型 1:主帖子 2:回复帖子',
post_title varchar(50) NOT NULL COMMENT '帖子标题',
post_text text NOT NULL COMMENT '帖子内容',
create_time date NOT NULL COMMENT '创建时间',
PRIMARY KEY (post_id),
KEY IDX_POST_TOPIC_ID (topic_id)
) ENGINE=InnoDB AUTO_INCREMENT=25 DEFAULT CHARSET=utf8 COMMENT='帖子';

```

准备好测试数据库的脚本文件之后，接下来使用 Excel 准备测试数据及验证数据（Excel 测试数据的格式要求详见第 17 章）。这里分别对 BoardDao、TopicDao、PostDao、UserDao 4 个 DAO 进行测试，需要为每个 DAO 创建相应的测试数据及验证数据，并放置到 DAO 相应的类路径下，如图 18-22 所示。

下面列举论坛话题的 Excel 测试验证数据集结构，如图 18-23 所示。其他测试数据集结构与论坛话题验证数据集结构相似，这里就不一一列出了。

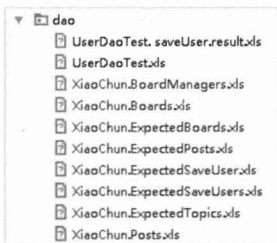


图 18-22 DAO 测试数据目录结构图

	A	B	C	D	E	F	G
	topic_id	board_id	topic_title	user_id	create_time	last_post	topic
1	1	1	AOP背后的故事	1	2011/5/7	2011/5/16	
2	2	4	IOC的原理一	1	2011/5/7	2011/5/16	
3	3	4	IOC的原理二	1	2011/5/7	2011/5/16	
4	4	1	SpringMVC集成	1	2011/5/7	2011/9/18	
5	5	1	SpringMVC集成	1	2011/5/7	2011/9/18	
6							
7							

图 18-23 论坛话题验证数据集结构

18.5.3 编写 DAO 测试基类

配置好 Unitils 测试环境及准备好测试数据之后，开始着手编写 DAO 测试用例。为了简化每个 DAO 测试，我们编写了一个 DAO 测试基类，所有的 DAO 测试用例都需要扩展该基类。具体实现如代码清单 18-15 所示。

代码清单 18-15 BaseDaoTest.java

```

package com.smart.dao;

import org.unitils.UnitilsTestNG;
import org.unitils.spring.annotation.SpringApplicationContext;

@SpringApplicationContext( {"classpath:/xiaochun-dao.xml" })
public class BaseDaoTest extends UnitilsTestNG{

}

...

```

测试 DAO 只需用到 DAO 层的配置信息，因此，在 DAO 测试基类中，只需通过

@SpringApplicationContext 注解加载 xiaochun-dao.xml 配置文件即可。由于我们的 DAO 测试用例基于 Unitils、TestNG 测试框架，因此需要扩展 Unitils 提供的 UnitilsTestNG 测试基类。

18.5.4 编写 BoardDao 测试用例

编写好 DAO 测试基类之后，接下来开始编写每个 DAO 相应的测试用例，首先来看一下 BoardDao 测试用例，具体实现如代码清单 18-16 所示。

代码清单 18-16 BoardDaoTest.java

```
package com.smart.dao;
import org.testng.annotations.Test;
import org.unitils.dbunit.annotation.DataSet;
import com.smart.test.dataset.util.XlsDataSetBeanFactory;
...
public class BoardDaoTest extends BaseDaoTest{

    //① 注入论坛版块DAO
    @SpringBean("boardDao")
    private BoardDao boardDao;

    //② 测试添加版块
    @Test
    @ExpectedDataSet("Xiaochun.ExpectedBoards.xls") //②-1 验证数据
    public void addBoard()throws Exception {
        //②-2 通过XlsDataSetBeanFactory数据集绑定工厂创建测试实体
        List<Board> boards = XlsDataSetBeanFactory.createBeans(BoardDaoTest.class,
            "Xiaochun.SaveBoards.xls", "t_board", Board.class);
        //②-3 持久化Board实例
        for(Board board:boards){
            boardDao.save(board);
        }
    }

    //③ 删除测试版块
    @Test
    @DataSet("Xiaochun.Boards.xls") //③-1 准备测试数据
    @ExpectedDataSet("Xiaochun.ExpectedBoards.xls") //③-2 准备验证数据
    public void removeBoard(){

        //③-3 加载指定的版块
        Board board = boardDao.get(7);

        //③-4 删除指定的版块
        boardDao.remove(board);
    }

    //④ 测试加载版块
    @Test
    @DataSet("Xiaochun.Boards.xls") //④-1准备测试数据
```

```

public void getBoard(){
    //④-2 加载版块
    Board board = boardDao.load(1);

    //④-3 验证结果
    assertNotNull(board);
    assertEquals(board.getBoardName(),"育儿");
}
...

```

在①处，通过 Unitils 提供的 `@SpringBean` 注解，从 Spring 容器中加载 BoardDao 实例。

在②处，测试保存版块功能 `BoardDao#save()`，首先通过 TestNG 提供的 `@Test` 注解将当前方法标志为测试方法。在②-1 处通过 Unitils 提供的 `@ExpectedDataSet` 注解，从当前测试用例所在的类路径中加载 `Xiaochun.ExpectedBoards.xls` 数据集文件。在②-2 处通过 `XlsDataSetBeanFactory#createBeans()` 方法从 `Xiaochun.SaveBoards.xls` 数据集中读取数据并实例化 Board 实体。在②-3 处，通过 `BoardDao#save()` 方法，将所有的 Board 实例持久化到数据库中。最后通过 `@ExpectedDataSet` 注解验证加载的数据是否与数据库中的数据匹配。

在③处，对删除版块 `BoardDao#remove()` 方法进行测试，首先通过 Unitils 提供的 `@DataSet` 注解从当前测试用例所在的类路径中加载 `Xiaochun.Boards.xls` 数据集文件，并将 `Xiaochun.Boards.xls` 数据集中的数据保存到相应的数据库表中。由于我们采用 Unitils 默认数据集先清理后保存策略，也就是先删除当前 `Xiaochun.Boards.xls` 对应的数据库表中的数据，然后将当前 `Xiaochun.Boards.xls` 数据集中的数据同步到数据库中。在③-3 处通过 `BoardDao#get()` 方法加载一个指定版块，并在③-4 处通过 `BoardDao#remove()` 方法将指定版块删除。最后通过 `@ExpectedDataSet` 注解加载的验证数据与当前数据库中的数据进行匹配验证。

在④处，对加载版块 `BoardDao#load()` 方法进行测试，首先通过 Unitils 提供的 `@DataSet` 注解从当前测试用例所在的类路径中加载 `Xiaochun.Boards.xls` 数据集文件，然后通过 `BoardDao#load()` 方法加载一个版块，并通过 TestNG 提供的断言对加载 Board 实体进行验证。最后在 IDE 中执行 `BoardDaoTest` 测试用例，测试结果如图 18-24 所示。

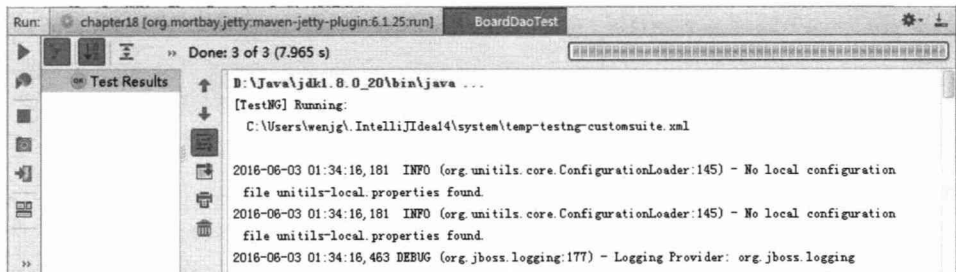


图 18-24 BoardDao 测试结果

至此，我们完成了对 BoardDao 测试用例的编写。由于每个 DAO 测试用例的编写方法都一样，此处就不一一阐述了，感兴趣的读者可以查看本书配套网盘中提供的相应章节案例代码。

18.6 服务层开发

服务层位于 Web 层和 DAO 层之间，服务类调用 DAO 层的类完成各项业务操作，并开放给 Web 层进行调用。我们在服务层中定义了两个服务类，分别是负责用户操作的 UserService 和负责论坛功能的 ForumService。

18.6.1 UserService 的开发

首先来看一下 UserService 的代码，具体实现如代码清单 18-17 所示。

代码清单 18-17 UserService.java

```
package com.smart.service;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
...
//用户管理服务，负责执行查询用户、注册用户、锁定用户等操作
@Service
public class UserService {
    @Autowired
    private UserDao userDao;
    @Autowired
    private LoginLogDao loginLogDao;

    //注册一个新用户。如果用户名已经存在，则抛出UserExistException异常
    public void register(User user) throws UserExistException{
        User u = this.getUserByUserName(user.getUserName());
        if(u != null){
            throw new UserExistException("用户名已经存在");
        }else{
            user.setCredit(100);
            user.setUserType(1);
            userDao.save(user);
        }
    }

    //根据用户名/密码查询User对象
    public User getUserByUserName(String userName){
        return userDao.getUserByUserName(userName);
    }

    //根据userId加载User对象
    public User getUserById(int userId){
```

```

        return userDao.getUser(userId);
    }

    //将用户锁定，锁定的用户不能登录
    public void lockUser(String userName){
        User user = userDao.getUserByUserName(userName);
        user.setLocked(User.USER_LOCK);
        userDao.update(user);
    }

    //解除用户的锁定
    public void unlockUser(String userName){
        User user = userDao.getUserByUserName(userName);
        user.setLocked(User.USER_UNLOCK);
        userDao.update(user);
    }

    ...
}
...

```

UserService 使用 UserDao、LoginLog 完成论坛用户的各项业务操作，这些方法都非常直观，相信读者可以轻松看懂。通过 Spring 提供的 @Autowired 注解，自动从 Spring 容器中加载 UserDao 和 LoginLog 这两个实例。UserService 事务管理通过 Spring 声明式事务管理的功能实现，通过事务的声明性信息，Spring 负责将事务管理增强逻辑动态织入业务方法相应的连接点中。这些逻辑包括获取线程绑定资源、开始事务、提交/回滚事务、进行异常转换和处理等工作。整个论坛服务层的事务管理统一在 xiaochun-service.xml 文件中进行配置，下文将会对其进行详细讲解。

18.6.2 ForumService 的开发

在学习完 UserService 后，再来了解一下提供论坛功能的 ForumService 服务类的代码，具体实现如代码清单 18-18 所示。

代码清单 18-18 ForumService.java

```

package com.smart.service;
import com.smart.dao.*;
import com.smart.domain.*;
...
@Service
public class ForumService {
    @Autowired
    private TopicDao topicDao;
    ...
    //①发表一个主题帖子，用户积分加10，论坛版块的主题帖子数加1
    public void addTopic(Topic topic) {
        Board board = (Board) boardDao.get(topic.getBoardId());
        board.setTopicNum(board.getTopicNum() + 1);
        topicDao.save(topic);
    }
}

```

```

        //①-1 创建主题帖子
        topic.getMainPost().setTopic(topic);
        MainPost post = topic.getMainPost();
        post.setCreateTime(new Date());
        post.setUser(topic.getUser());
        post.setPostTitle(topic.getTopicTitle());
        post.setBoardId(topic.getBoardId());

        //①-2 持久化主题帖子
        postDao.save(topic.getMainPost());

        //①-3 更新用户积分
        User user = topic.getUser();
        user.setCredit(user.getCredit() + 10);
        userDao.update(user);
    }

    //② 删除一个主题帖子，用户积分减50，论坛版块的主题帖子数减1，
    //删除主题帖子所有关联的帖子
    public void removeTopic(int topicId) {
        Topic topic = topicDao.get(topicId);

        // 将论坛版块的主题帖子数减1
        Board board = boardDao.get(topic.getBoardId());
        board.setTopicNum(board.getTopicNum() - 1);

        // 发表该主题帖子的用户扣除50个积分
        User user = topic.getUser();
        user.setCredit(user.getCredit() - 50);

        // 删除主题帖子及其关联的帖子
        topicDao.remove(topic);
        postDao.deleteTopicPosts(topicId);
    }

    //③ 添加一个回复帖子，用户积分加5，主题帖子回复数加1，并更新最后回复时间
    public void addPost(Post post) {
        postDao.save(post);
        User user = post.getUser();
        user.setCredit(user.getCredit() + 5);
        userDao.update(user);
        Topic topic = topicDao.get(post.getTopicId());
        topic.setReplies(topic.getReplies() + 1);
        topic.setLastPost(new Date());

        //topic处于Hibernate受管状态，无须显示更新
        //topicDao.update(topic);
    }
    ...
}
...

```


ForumService 是论坛的核心服务类，它实现了论坛的大部分功能。ForumService 引用了 BoardDao、TopicDao、PostDao 和 UserDao 这 4 个 DAO 类。

大部分服务方法都比较简单，它们完成业务逻辑并进行数据持久化操作。下面仅对一些复杂的方法进行说明。先来看一下①处的 addTopic()方法：在①-1 处通过 Topic 实例创建主题帖子，在①-2 处调用基类的 save()方法进行持久化操作，在①-3 处为发表者添加 10 个积分，并调用 UserDao 的 update()方法更新到数据库中。

③处的 addPost()方法可能也会存在理解上的障碍。在该方法中，我们添加了一个回复帖子，同时更新主题帖子的回复帖子数及主题的最后回复时间，但是并没有调用 TopicDao 的 update()方法更新 Topic。也许会有读者怀疑这里的代码是否有误，其实这个方法是可以正确工作的。因为我们通过 topicDao.get(post.getTopicId())方法从数据表中加载 Topic 实例，所以这个 Topic 实例处于受管的状态，在方法中调整其 replies 和 lastPost 属性，Hibernate 会将 Topic 状态更改自动同步到数据表中，无须显式调用 topicDao.update()方法。



实战经验

由于数据库自增键会带来诸多的不便或潜在的问题，如在一些基于配置型的应用中，经常会涉及一些配置模型数据的导入/导出操作，此时很容易造成主键冲突问题。因此，建议读者在实际开发中尽量采用 UUID 的主键，或使用一个服务在应用层中获取领域对象的主键，省却这种因数据库产生主键而造成的麻烦。

18.6.3 服务类 Bean 的装配

在编写完 UserService 和 ForumService 后，需要在 Spring 配置文件中进行装配，以便注入 DAO Bean 并实施事务管理增强。我们在 src/main/resources 目录下创建一个 Spring 的配置文件 xiaochun-service.xml，该文件的内容如代码清单 18-19 所示。

代码清单 18-19 xiaochun-service.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:tx="http://www.springframework.org/schema/tx"
       ...
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-4.0.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-4.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.0.xsd">
```



```

<!--①扫描com.smart.service包下所有标注@Service的服务组件-->
<context:component-scan base-package="com.smart.service"/>
<!--②事务管理器-->
<bean id="transactionManager"
      class="org.springframework.orm.hibernate4.HibernateTransactionManager"
      p:sessionFactory-ref="sessionFactory" />
<!--③使用强大的切点表达式语言轻松定义目标方法-->
<aop:config>
  <!--③-1通过AOP定义事务增强切面-->
  <aop:pointcut id="serviceMethod"
    expression="execution(* com.smart.service.*Service.*(..))" />
  <!--③-2引用事务增强-->
  <aop:advisor pointcut-ref="serviceMethod" advice-ref="txAdvice" />
</aop:config>
<!--④事务增强-->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
  <!--④-1事务属性定义-->
  <tx:attributes>
    <tx:method name="*" />
  </tx:attributes>
</tx:advice>

<!--⑤ 基于EhCache的系统缓存管理器-->
<bean id="cacheManager"
      class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean"
      p:configLocation="classpath:ehcache.xml"/>
</beans>
...

```

首先在①处扫描 `com.smart.service` 包下所有标注 `@Service` 注解的 `Service Bean`，`Service Bean` 引用 `xiaochun-dao.xml` 扫描 `DAO Bean`。接着对论坛服务层配置事务管理，需要在配置文件中引入 `tx` 命名空间的声明，如 `<beans>` 元素粗体部分所示。采用 `aop/tx` 定义事务方法时，它站在“局外人”的角度对 `IoC` 容器中的 `Bean` 进行事务管理配置定义，再由 `Spring` 将这些配置织入对对应的 `Bean` 中。

在 `aop` 命名空间中，通过切点表达式语言，我们将 `com.smart.service` 包下所有以 `Service` 为后缀的类纳入了需要进行事务增强的范围，并配合 `<tx:advice>` 的 `<aop:advisor>` 完成了事务切面的定义，如③处所示。

`<aop:advisor>` 引用的 `txAdvice` 增强是在 `tx` 命名空间中定义的，如④处所示。事务增强必须有一个事务管理器的支持，`<tx:advice>` 通过 `transaction-manager` 属性引用了在②处定义的事务管理器（默认查找名为 `transactionManager` 的事务管理器，所以如果事务管理器命名为 `transactionManager`，则可以不指定 `transaction-manager` 属性）。

18.7 对服务层进行测试

服务层位于 `Web` 层和 `DAO` 层之间，要对服务层实施测试，需要与 `DAO` 层进行交互。但在测试过程中，这些需要被调用的真实对象常常很难被实例化，或者这些对象在

某些情况下无法被用来测试，例如，真实对象的行为无法预测、真实对象的行为难以触发，或者真实对象的运行速度很慢。这时候就需要使用模拟对象技术（Mock），利用一个模拟对象来模拟代码所依赖的真实对象，以帮助完成测试，或者对服务层和 DAO 层进行集成测试。下面使用第 20 章介绍的 Unitils、TestNG 等框架对论坛服务层进行集成测试。

18.7.1 编写 Service 测试基类

编写好论坛 UserService 和 ForumService 两个服务类之后，开始编写相应的测试用例。为了简化每个 Service 测试，我们编写了一个 Service 测试基类，所有的 Service 测试用例都需要扩展该基类，具体实现如代码清单 18-20 所示。

代码清单 18-20 BaseServiceTest.java

```
package com.smart.service;
import org.springframework.orm.hibernate4.HibernateTemplate;
import org.unitils.UnitilsTestNG;
import org.unitils.spring.annotation.SpringApplicationContext;
import org.unitils.spring.annotation.SpringBean;
@SpringApplicationContext( {"xiaochun-service.xml", "xiaochun-dao.xml" })
public class BaseServiceTest extends UnitilsTestNG {
    @SpringBean("hibernateTemplate")
    public HibernateTemplate hibernateTemplate;
}
...
```

测试 Service 层一般有两种方法：一种是通过 Mockito 等模拟测试框架对 DAO 层进行模拟，完成 Service 层独立性测试；另一种通过 Unitils 等框架对 Service 层和 DAO 层进行集成测试。本案例采用集成测试方法，首先通过 Unitils 提供的 @SpringApplicationContext 注解加载 Service 层和 DAO 层的配置文件 xiaochun-service.xml 和 xiaochun-dao.xml，然后通过 @SpringBean 注解从 Spring 容器中加载 HibernateTemplate 实例。

18.7.2 编写 ForumService 测试用例

编写好 Service 测试基类之后，接下来编写每个 Service 相应的测试用例（在第 20 章中会对 UserService 测试用例进行讲解，这里不再阐述）。下面重点看一下论坛的核心服务 ForumService 测试用例，具体实现如代码清单 18-21 所示。

代码清单 18-21 UserServiceTest.java

```
package com.smart.service;
import static org.hamcrest.Matchers.*;
import com.smart.test.dataset.util.XlsDataSetBeanFactory;
...
public class ForumServiceTest extends BaseServiceTest {
```

```

@SpringBean("forumService")
private ForumService forumService;
@SpringBean("userService")
private UserService userService;

//① 在测试初始化中, 消除Hibernate二级缓存, 避免影响测试
@BeforeMethod
public void init(){
    SessionFactory sf = hibernateTemplate.getSessionFactory();
    Map<String, CollectionMetadata> roleMap = sf.getAllCollectionMetadata();
    for (String roleName : roleMap.keySet()) {
        sf.evictCollection(roleName);
    }
    Map<String, ClassMetadata> entityMap = sf.getAllClassMetadata();
    for (String entityName : entityMap.keySet()) {
        sf.evictEntity(entityName);
    }
    sf.evictQueries();
}

//② 测试新增一个版块
@Test
@DataSet("Xiaochun.DataSet.xls")
public void addBoard() throws Exception {
    Board board = XlsDataSetBeanFactory.createBean(ForumServiceTest.class,
        "Xiaochun.DataSet.xls", "t_board", Board.class);
    forumService.addBoard(board);
    Board boardDb = forumService.getBoardById(board.getBoardId());
    assertEquals (boardDb.getBoardName(), "育儿");
}

//③ 测试新增一个主题帖子
@Test
@DataSet("Xiaochun.DataSet.xls")
public void addTopic() throws Exception {
    Topic topic = XlsDataSetBeanFactory.createBean(ForumServiceTest.class,
        "Xiaochun.DataSet.xls", "t_topic", Topic.class);
    User user = XlsDataSetBeanFactory.createBean(ForumServiceTest.class,
        "Xiaochun.DataSet.xls", "t_user", User.class);
    topic.setUser(user);
    forumService.addTopic(topic);
    Board boardDb = forumService.getBoardById(1);
    User userDb = userService.getUserByUserName("tom");
    assertEquals (boardDb.getTopicNum(), 1);
    assertEquals (userDb.getCredit(), 110);
    assertEquals(topic.getTopicId(), 0);
}
...
}

```

上面的测试用例都比较好理解, 首先使用 Excel 准备测试数据 (格式要求详见第 20 章, 这里不再阐述), 如案例中的 Xiaochun.DataSet.xls 文件。然后通过 Unitils 提供的 @DataSet 注解从当前测试用例的所有类路径中加载 Xiaochun.DataSet.xls 数据集文件并

初始化测试数据库。通过自己编写的 XlsDataSetBeanFactory 工具类，加载 Xiaochun.DataSet.xls 测试数据，并实例化测试对象实例。接着通过 TestNG 和 Hamcrest 提供的断言及匹配方法对结果进行验证。由于在论坛中启用了 Hibernate 二级缓存，为了不影响测试结果，在执行测试方法之前，需要消除 Hibernate 二级缓存，如①处所示。最后在 IDE 中执行 ForumServiceTest 测试用例，可以看到测试运行结果。

至此，我们完成了对 ForumService 和 UserService 测试用例的编写。由于每个 Service 测试用例的编写方法都一样，在这里就不一一阐述了，感兴趣的读者可以查看本书配套网盘中提供的相应章节案例代码。

18.8 Web 层开发

至此，DAO 和服务类都已经准备就绪，并且通过单元测试对其进行了全面测试。接下来需要开发 Web 层将服务和页面关联起来。

18.8.1 BaseController 的基类

由于 Web 层的每个 Controller 都有一些常用的操作，所以我们提供了一个 Controller 的基类，如代码清单 18-22 所示。

代码清单 18-22 BaseController.java

```
package com.smart.web;
import com.smart.domain.User;
import com.smart.exception.NotLoginException;
...
public class BaseController {
    protected static final String ERROR_MSG_KEY = "errorMsg";

    //① 获取保存在Session中的用户对象
    protected User getSessionUser(HttpServletRequest request) {
        return (User) request.getSession().getAttribute(
            CommonConstant.USER_CONTEXT);
    }

    //②将用户对象保存到Session中
    protected void setSessionUser(HttpServletRequest request, User user) {
        request.getSession().setAttribute(CommonConstant.USER_CONTEXT,
            user);
    }

    //③ 获取基于应用程序的URL绝对路径
    public final String getAppbaseUrl(HttpServletRequest request, String url) {
        Assert.hasLength(url, "url不能为空");
        Assert.isTrue(url.startsWith("/"), "必须以/打头");
        return request.getContextPath() + url;
    }
}
```

```

    }
    ...
}

```

由于论坛的 Controller 基于 Spring MVC 注解方式，因此我们无须扩展任何基类。在 BaseController 类中定义了 3 个方法，其中 setSessionUser() 方法负责将 User 对象保存到 Session 中；getSessionUser() 方法负责获取保存在 Session 中的 User 对象；getAppbaseUrl() 方法负责获取基于应用程序的 URL 绝对路径。

Web 层的每个 Controller 都有可能涉及登录验证处理逻辑，如论坛中只有登录用户才能发表新话题，所以我们提供了一个过滤器来进行处理，如代码清单 18-23 所示。

代码清单 18-23 ForumFilter.java

```

package com.smart.web;
import javax.servlet.Filter;
import com.smart.domain.User;
import static com.smart.cons.CommonConstant.*;
...
public class ForumFilter implements Filter {
    private static final String FILTERED_REQUEST = "@@session_context_filtered_request";

    //①不需要登录即可访问的URI资源
    private static final String[] INHERENT_ESCAPE_URIIS = { "/index.jsp", "/index.html",
        "/login.jsp", "/login/doLogin.html", "/register.jsp",
        "/register.html", "/board/listBoardTopics-", "/board/list
TopicPosts-"};

    //②执行过滤
    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {

        //②-1保证该过滤器在一次请求中只被调用一次
        if (request != null && request.getAttribute(FILTERED_REQUEST) != null) {
            chain.doFilter(request, response);
        } else {

            //②-2设置过滤标识，防止一次请求多次过滤
            request.setAttribute(FILTERED_REQUEST, Boolean.TRUE);
            HttpServletRequest httpRequest = (HttpServletRequest) request;
            User userContext = getSessionUser(httpRequest);

            //②-3用户未登录，且当前URI资源需要登录才能访问
            if (userContext == null && ! isURILogin (httpRequest.getRequestURI(),
httpRequest)) {

                String toUrl = httpRequest.getRequestURL(). toString();
                if (!StringUtils.isEmpty(httpRequest.getQueryString())) {
                    toUrl += "?" + httpRequest.getQueryString();
                }

                //②-4将用户的请求URL保存在Session中，用于登录成功后跳到目标URL
                httpRequest.getSession().setAttribute(LOGIN_TO_URL, toUrl);

                //②-5转发到登录页面

```



```

        request.getRequestDispatcher("/login.jsp").forward(request, response);
        return;
    }
    chain.doFilter(request, response);
}
}

//③当前URI资源是否需要登录才能访问
private boolean isURILogin (String requestURI, HttpServletRequest request) {
    if (request.getContextPath().equalsIgnoreCase(requestURI)
        || (request.getContextPath() + "/").equalsIgnoreCase(requestURI))
        return true;
    for (String uri : INHERENT_ESCAPE_URIS) {
        if (requestURI != null && requestURI.indexOf(uri) >= 0) {
            return true;
        }
    }
    return false;
}
...
}

```

在过滤器中，首先设置论坛中所有不需要登录即可访问的 URI 资源，如①处所示。在过滤处理过程中，设置一个当前请求的过滤标识，防止一次请求多次过滤的情况，如②-2 处所示。如果用户未登录，且当前 URI 资源需要登录才能访问，则保存当前请求的 URI 到会话中，并转发到登录页面。

18.8.2 用户登录和注销

用户登录和注销功能由 LoginController 负责，LoginController 通过调用服务层的 UserService 类完成相应的业务操作。来看一下 LoginController 的实现，如代码清单 18-24 所示。

代码清单 18-24 LoginController.java

```

package com.smart.web;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import static com.smart.cons.CommonConstant.*;
...
@Controller
@RequestMapping("/login")
public class LoginController extends BaseController {

    private UserService userService;

    @Autowired
    public void setUserService(UserService userService) {
        this.userService = userService;
    }
    //① 用户登录

```

```

@RequestMapping("/doLogin")
public ModelAndView login(HttpServletRequest request, User user) {
    User dbUser = userService.getUserByUserName(user.getUserName());
    ModelAndView mav = new ModelAndView();
    mav.setViewName("forward:/login.jsp");
    if(dbUser == null){
        mav.addObject("errorMsg", "用户名不存在");
    }else if(!dbUser.getPassword().equals(user.getPassword())){
        mav.addObject("errorMsg", "用户密码不正确");
    }else if(dbUser.getLocked() == User.USER_LOCK){
        mav.addObject("errorMsg", "用户已经被锁定，不能登录。");
    }else{
        dbUser.setLastIp(request.getRemoteAddr());
        dbUser.setLastVisit(new Date());
        userService.loginSuccess(dbUser);
        setSessionUser(request, dbUser);
        String toUrl = (String) request.getSession().getAttribute(LOGIN_TO_URL);
        request.getSession().removeAttribute(LOGIN_TO_URL);

        //如果当前会话中没有保存登录之前的请求URL，则直接跳转到主页
        if(StringUtils.isEmpty(toUrl)){
            toUrl = "/index.html";
        }
        mav.setViewName("redirect:"+toUrl);
    }
    return mav;
}

//② 登录注销
@RequestMapping("/doLogout")
public String logout(HttpSession session) {
    session.removeAttribute(USER_CONTEXT);
    return "forward:/index.jsp";
}
...
}

```

LoginController 直接扩展于 BaseController，login()方法负责处理用户登录操作，当用户名不存在、用户密码不正确或者用户已经被锁定时，都直接转到登录页面并报告相关的错误信息；否则用户添加 5 个积分并将其保存到 HTTP Session 中，然后转向成功页面。

在①处，首先根据用户名获取到 User 对象实例，然后对当前用户实例状态进行判断。如果当前用户实例既不为空，也不是被锁定状态，则说明当前用户合法，表示登录成功，调用 UserService#loginSuccess()方法添加当前用户积分并保存登录日志。最后判断当前会话中是否存在登录之前的请求 URL（这个请求 URL 在论坛的过滤器中设置），如果存在则跳转到这个 URL，如果不存在就直接跳转到主页。

②处的用户登录注销方法 logout()很简单，其主要工作是将 User 从 Session 中移除，并转到论坛主页中。这里使用了 LoginController 中定义的方法，将请求重定向到/index.jsp 页面中。

18.8.3 用户注册

要成为论坛的用户，首先必须进行注册。用户注册是论坛的一个非常重要的功能。下面来看一下负责用户注册的 RegisterController，如代码清单 18-25 所示。

代码清单 18-25 RegisterController.java

```
package com.smart.web;
import com.smart.domain.User;
import com.smart.exception.UserExistException;
import com.smart.service.UserService;
...
@Controller
public class RegisterController extends BaseController {
    @Autowired
    private UserService userService;

    //用户登录
    @RequestMapping(value = "/register", method = RequestMethod.POST)
    public ModelAndView register(HttpServletRequest request, User user) {
        ModelAndView view = new ModelAndView();
        view.setViewName("/success");
        try {
            userService.register(user);
        } catch (UserExistException e) {
            view.addObject("errorMsg", "用户名已经存在，请选择其他的名字。");
            view.setViewName("forward:/register.jsp");
        }
        setSessionUser(request, user);
        return view;
    }
    ...
}
```

在注册用户之前需要判断用户名是否已经存在，如果已经存在，则必须告之用户，以便用户调整注册的用户名。在 Web 2.0 时代，为了增强用户体验，最好在页面端采用 AJAX 技术，当用户输入用户名时即可自动告之用户是否已经存在，而不是等到用户提交注册表单后再进行判断。不过，不管注册用户名是否已经通过 AJAX 校验，仍有必要在服务器端再验证一次。

接下来的工作是配置用户注册的 JSP 页面，如代码清单 18-26 所示。

代码清单 18-26 register.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>用户注册</title>
<script>
function mycheck(){
    if(document.all("user.password").value != document.all("again").value){
```

```

        alert("两次输入的密码不正确, 请更正。");
        return false;
    }else{
        return true;
    }
}
</script>
</head>
<body>
用户注册信息:
<form action="<c:url value="/register.html" />" method="post" onsubmit="return mycheck()">
<c:if test="${!empty errorMsg}">
    <div style="color:red">${errorMsg}</div>
</c:if>
<table border="1px" width="60%">
    <tr>
        <td width="20%">用户名</td>
        <td width="80%"><input type="text" name="userName"/></td>
    </tr>
    <tr>
        <td width="20%">密码</td>
        <td width="80%"><input type="password" name="password"/></td>
    </tr>
    <tr>
        <td width="20%">密码确认</td>
        <td width="80%"><input type="password" name="again"/></td>
    </tr>
    <tr>
        <td colspan="2">
            <input type="submit" value="保存"> <input type="reset" value="重置">
        </td>
    </tr>
</table>
</form>
</body>
</html>

```

用户注册表单的信息非常简单, 仅包含用户名和密码两个信息。当提交表单时, 表单信息填充到 User 属性中。

18.8.4 论坛管理

论坛管理模块对应论坛管理员所使用的各项操作功能, 具体来说, 包括创建论坛版块、指定论坛版块管理员、用户锁定/解锁等功能。ForumManageController 负责处理这些操作请求, 如代码清单 18-27 所示。

代码清单 18-27 ForumManageController.java

```

package com.smart.web;
import com.smart.service.ForumService;
import com.smart.service.UserService;
...

```

//论坛管理, 这部分功能由论坛管理员操作, 包括创建论坛版块、指定论坛版块管理员、用户锁定/解锁

@Controller

public class ForumManageController extends BaseController {

 @Autowired

 private ForumService forumService;

 @Autowired

 private UserService userService;

 //① 列出所有的论坛模块

 @RequestMapping(value = "/index", method = RequestMethod.GET)

 public String listAllBoards() {

 List<Board> boards = forumService.getAllBoards();

 request.setAttribute("boards", boards);

 return "/listAllBoards";

 }

 //② 添加一个主题帖子页面

 @RequestMapping(value = "/forum/addBoardPage", method = RequestMethod.GET)

 public String addBoardPage() {

 return "/addBoard";

 }

 //③ 添加一个主题帖子

 @RequestMapping(value = "/forum/addBoard", method = RequestMethod.POST)

 public String addBoard(Board board) {

 forumService.addBoard(board);

 return "/addBoardSuccess";

 }

 //④ 指定论坛管理员的页面

 @RequestMapping(value = "/forum/setBoardManagerPage", method = RequestMethod.GET)

 public ModelAndView setBoardManagerPage() {

 ModelAndView view = new ModelAndView();

 List<Board> boards = forumService.getAllBoards();

 List<User> users = userService.getAllUsers();

 view.addObject("boards", boards);

 view.addObject("users", users);

 view.setViewName("/setBoardManager");

 return view;

 }

 ...

}

ForumManageController 通过调用服务层的 UserService 和 ForumService 完成相应业务逻辑。由于进行用户锁定、指定论坛版块管理员等操作都需要一个具体的操作页面, 所以 ForumManageController 的另一项工作是将请求导向一个具体的操作页面中, 如②处和④处的 addBoardPage()和 setBoardManagerPage 方法所示。

ForumManageController 共有 4 个转向页面。

- userLockManagePage: 对应 WEB-INF/jsp/userLockManage.jsp 页面, 即用户解锁和锁定的操作页面。
- setBoardManagerPage: 对应 WEB-INF/jsp/setBoardManager.jsp 页面, 这是设置论坛版块管理员的处理页面。

- ❑ listAllBoards: 对应 WEB-INF/jsp/listAllBoards.jsp 页面, 该页面显示论坛版块列表。
- ❑ addBoardPage: 对应 WEB-INF/jsp/addBoard.jsp 页面, 这是新增论坛版块的表单页面。

这些 JSP 页面的代码都比较简单, 读者可自行通过本书配套网盘中的代码进行学习, 这里不再展开阐述。

18.8.5 论坛普通功能

论坛普通功能包括显示论坛版块列表、显示论坛版块主题列表、发表主题帖子、回复帖子、删除帖子、设置精华帖子等, 这些功能由 BoardManageController 负责处理, 具体实现如代码清单 18-28 所示。

代码清单 18-28 BoardManageController.java

```
package com.smart.web;
import com.smart.domain.*;
import com.smart.service.ForumService;
...
@Controller
public class BoardManageController extends BaseController {
    @Autowired
    private ForumService forumService;

    //① 列出论坛模块下的主题帖子
    @RequestMapping(value = "/board/listBoardTopics-{boardId}",
method = RequestMethod.GET)
    public ModelAndView listBoardTopics(@PathVariable Integer boardId,
        @RequestParam(value = "pageNo", required = false) Integer pageNo)
    {
        ModelAndView view =new ModelAndView();
        Board board = forumService.getBoardById(boardId);
        pageNo = pageNo==null?1:pageNo;
        Page pagedTopic = forumService.getPagedTopics(boardId, pageNo,
            CommonConstant.PAGE_SIZE);
        view.addObject("board", board);
        view.addObject("pagedTopic", pagedTopic);
        view.setViewName("/listBoardTopics");
        return view;
    }

    //② 新增主题帖子页面
    @RequestMapping(value = "/board/addTopicPage-{boardId}", method = RequestMethod.GET)
    public ModelAndView addTopicPage(@PathVariable Integer boardId) {
        ModelAndView view =new ModelAndView();
        view.addObject("boardId", boardId);
        view.setViewName("/addTopic");
        return view;
    }
}
```

//③ 添加一个主题帖子

```
@RequestMapping(value = "/board/addTopic", method = RequestMethod.POST)
public String addTopic(HttpServletRequest request, Topic topic) {
    User user = getSessionUser(request);
    topic.setUser(user);
    Date now = new Date();
    topic.setCreateTime(now);
    topic.setLastPost(now);
    forumService.addTopic(topic);
    String targetUrl = "/board/listBoardTopics-" + topic.getBoardId() + ".html";
    return "redirect:"+targetUrl;
}
```

//④ 列出主题的所有帖子

```
@RequestMapping(value = "/board/listTopicPosts-{topicId}", method = RequestMethod.GET)
public ModelAndView listTopicPosts(@PathVariable Integer topicId,
    @RequestParam(value = "pageNo", required = false) Integer pageNo) {
    ModelAndView view = new ModelAndView();
    Topic topic = forumService.getTopicById(topicId);
    pageNo = pageNo == null ? 1 : pageNo;
    Page pagedPost = forumService.getPagedPosts(topicId, pageNo,
        CommonConstant.PAGE_SIZE);

    // 为回复帖子表单准备数据
    view.addObject("topic", topic);
    view.addObject("pagedPost", pagedPost);
    view.setViewName("/listTopicPosts");
    return view;
}
...
}
```

Controller 的主要职责包括页面流程的控制、业务数据的准备，而业务逻辑处理则直接调用服务层的类来完成。如③处的 addTopic()请求处理方法首先应用 Spring MVC 自动绑定功能从表单获取提交的数据并绑定到 Topic 对象实例，接着补充那些需要在服务层产生的数据，如主题帖子的发表人、发表时间、最后回复时间等，然后调用服务层的 ForumService#addTopic()方法新增主题帖子，最后将请求重定向到论坛版块主题帖子列表页面中。

读者可能已经注意到，有很多请求方法处理完成后并未返回一个视图名称，而是直接返回 redirect.targetUrl 完成重定向请求。这是因为目标 URL 是动态的，不方便直接定义返回视图，因此直接利用 Spring MVC 提供的重定向功能进行请求的转向。

BoardManageController 共有 3 个转向页面。

- listBoardTopics: 对应 WEB-INF/jsp/listBoardTopics.jsp 页面，该页面显示论坛版块的所有主题帖子。
- listTopicPosts: 对应 WEB-INF/jsp/listTopicPosts.jsp 页面，该页面显示主题所对应的所有帖子。
- addTopicPage: 对应 WEB-INF/jsp/addTopic.jsp 页面，它是发表新主题帖子的表单页面。

本书配套网盘里包括了所有 JSP 页面的代码，读者可以学习相关知识。

18.8.6 分页显示论坛版块的主题帖子

通过前面的讲述,我们知道论坛分别通过 listBoardTopics.jsp 和 listTopicPosts.jsp 页面分页显示论坛版块下的主题帖子和主题帖子下的所有帖子。本小节通过 listBoardTopics.jsp 页面讲解展现层分页功能的实现过程。先来看一下 listBoardTopics.jsp 页面代码,如代码清单 18-29 所示。

代码清单 18-29 listBoardTopics.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
<!--① 引入一个自定义的Tag, 该Tag用于生成分页导航的代码-->
<%@taglib prefix="xiaochun" tagdir="/WEB-INF/tags" %>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>论坛版块页面</title>
    </head>
    <body>
        <%@ include file="includeTop.jsp"%>
        <div>
            <table border="1px" width="100%">
                <tr>
                    <c:if test="${USER_CONTEXT.userType == 2 || isboardManager}">
                        <td></td>
                    </c:if>
                    <td bgcolor="#EEEEEE">
                        ${board.boardName}
                    </td>
                    <td colspan="4" bgcolor="#EEEEEE" align="right">
                        <a href="<c:url value="/board/addTopicPage-${board.boardId}.html"/>">
                            发表新话题
                        </a>
                    </td>
                </tr>
                <tr>
                    <c:if test="${USER_CONTEXT.userType == 2 || isboardManager}">
                        <td>
                            <script>
                                function switchSelectBox(){
                                    var selectBoxes = document.all("topicIds");
                                    if(!selectBoxes) return ;
                                    if(typeof(selectBoxes.length)=="undefined"){//only one checkbox
                                        selectBoxes.checked = event.srcElement.checked;
                                    }else{//many checkbox ,so is a array
                                        for(var i = 0 ; i < selectBoxes.length ; i++){
                                            selectBoxes[i].checked = event.srcElement.checked;
                                        }
                                    }
                                }
                            </script>
                        </td>
                    </c:if>
                </tr>
            </table>
        </div>
    </body>
</html>
```

```

        <input type="checkbox" onclick="switchSelectBox()" />
    </td>
</c:if>
    <td width="50%">标题</td>
    <td width="10%">发表人</td>
    <td width="10%">回复数</td>
    <td width="15%">发表时间</td>
    <td width="15%">最后回复时间</td>
</tr>
<!--② 判断用户是否是该版块的管理员-->
    <c:set var="isboardManager" value="{false}" />
    <c:forEach items="{USER_CONTEXT.manBoards}" var="manBoard">
        <c:if test="{manBoard.boardId == board.boardId}">
            <c:set var="isboardManager" value="{true}" />
        </c:if>
    </c:forEach>
<!--③ 对保存在Page对象中的分页数据进行渲染，以显示一页的数据-->
    <c:forEach var="topic" items="{pagedTopic.result}">
        <tr>
            <!--③-1 如果是论坛版块管理员或者论坛管理员，则显示批量操作的复选框-->
            <c:if test="{USER_CONTEXT.userType == 2 || isboardManager}">
                <td>
                    <input type="checkbox" name="topicIds" value="{topic.topicId}" />
                </td>
            </c:if>
            <td>
                <a href="{<c:url value="/board/listTopicPosts-${topic.
topicId}.html"/>}">
                    <!--③-2 如果是精华帖子，则附加★号标志-->
                    <c:if test="{topic.digest > 0}">
                        <font color=red>★</font>
                    </c:if>
                    ${topic.topicTitle}
                </a>
            </td>
            <td>${topic.user.userName}<br><br></td>
            <td>${topic.replies}<br><br>
            </td>
            <td><fmt:formatDate pattern="yyyy-MM-dd HH:mm"
                value="{topic.createTime}" /></td>
            <td><fmt:formatDate pattern="yyyy-MM-dd HH:mm"
                value="{topic.lastPost}" /></td>
        </tr>
    </c:forEach>
</table>
</div>
<!--④ 分页显示导航栏-->
<xiaochun:PageBar
    pageUrl="/board/listBoardTopics-${board.boardId}.html"
    pageAttrKey="pagedTopic"/>
<!--⑤ 如果是论坛版块管理员或者论坛管理员，则显示批量操作的按钮-->
<c:if test="{USER_CONTEXT.userType == 2 || isboardManager}">
<script>
    function getSelectedTopicIds(){
        var selectBoxes = document.all("topicIds");

```



```

        if(!selectBoxs) return null;
        if(typeof(selectBoxs.length) == "undefined" && selectBoxs.checked){
            return selectBoxs.value;
        }else{
            var ids = "";
            var split = ""
            for(var i = 0 ; i < selectBoxs.length ; i++){
                if(selectBoxs[i].checked){
                    ids += split+selectBoxs[i].value;
                    split = ",";
                }
            }
            return ids;
        }
    }
}

function deleteTopics(){
    var ids = getSelectedTopicIds();
    if(ids){
        var url = "<c:url value=\"/board/removeTopic.html\"/?topicIds="+
            ids+"&boardId=${board.boardId}";
        location.href = url;
    }
}

function setDefinedTopis(){
    var ids = getSelectedTopicIds();
    if(ids){
        var url = "<c:url value=\"/board/makeDigestTopic.html\"/? topicIds="+
            ids+"&boardId=${board.boardId}";
        location.href = url;
    }
}
}
</script>
<input type="button" value="删除" onclick="deleteTopics()">
<input type="button" value="置精华帖" onclick="setDefinedTopis()">
</c:if>
</body>
</html>

```

在 BoardManageController 中, 通过调用服务层的 ForumService#getPagedTopics() 方法获取了一页的数据, 如果未指定 pageNo 参数, 则默认为第一页数据。在获取页数据对象后, 将其以 “pagedTopic” 为名称放置到 Request 属性列表中, 并调用 listBoardTopics.jsp 页面渲染这一页的数据。最值得我们关注的可能是<xiaochun:PageBar/>这个自定义标签。

<xiaochun:PageBar/>标签接收两个参数, 分别是 pageUrl 和 pageAttrKey。前者是转到其他页面的处理 URL; 而后者是分页对象在 Request 属性列表中的键名称, <xiaochun:PageBar />需要根据这个名称从 Request 中获取 Page 对象。

我们使用 JSP 2.0 的 Tag 语法编写<xiaochun:PageBar/>, 即在 WEB-INF/tags 目录下创建一个名为 PageBar.tag 的标签页面文件, 其标签代码如代码清单 18-30 所示。

代码清单 18-30 PageBar.tag

```
<%@ tag pageEncoding="UTF-8" %>
```


在定义好 PageBar 标签后,不需要定义 TLD 文件,可直接在 JSP 页面中使用该标签,不过在使用之前必须声明标签的前缀: <%@taglib prefix="xiaochun" tagdir="/WEB-INF/tags" %>。

18.8.7 web.xml 配置

至此,论坛应用开发工作已经基本完成,但还未编写 web.xml 这个重要的文件。我们需要在 web.xml 中声明 Spring 配置文件和启动监听器,并配置 Spring 的请求 Servlet。此外,为了避免中文乱码问题,需要提供一个编码转换过滤器。由于论坛应用在 Web 层的 Action 中使用了 Hibernate 延迟加载机制,所以需要配置一个 OpenSessionInViewFilter 过滤器,以便 Hibernate 的 Session 能够在控制器请求方法中保持打开状态。下面是 web.xml 配置文件的所有内容,如代码清单 18-31 所示。

代码清单 18-31 web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns=http://java.sun.com/xml/ns/j2ee
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.4"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <!--① 覆盖default servlet的/, Spring MVC Servlet将处理原来处理静态资源的映射 -->
    <description>小春论坛</description>
    <display-name>xiaochun</display-name>
    <!--② 指定Spring配置文件和初始化监听器-->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:/applicationContext.xml</param-value>
    </context-param>
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener
    </listener-class>
    </listener>
    <filter>
        <!--③ 在Web层打开Hibernate Session, 以便可以在Web层访问到Hibernate延迟加载的数据-->
        <filter-name>hibernateFilter</filter-name>
        <filter-class>org.springframework.orm.hibernate4.support.OpenSessionInView
Filter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>hibernateFilter</filter-name>
        <url-pattern>*.html</url-pattern>
    </filter-mapping>
    <!--④论坛登录验证过滤器-->
    <filter>
        <filter-name>forumFilter</filter-name>
        <filter-class>com.smart.web.ForumFilter</filter-class>
    </filter>
    <filter-mapping>
```

```

        <filter-name>forumFilter</filter-name>
        <url-pattern>*.html</url-pattern>
    </filter-mapping>
    <filter-mapping>
        <filter-name>forumFilter</filter-name>
        <url-pattern>*.jsp</url-pattern>
    </filter-mapping>
    <!--⑤使用Spring的编码转换过滤器，将请求信息的编码统一转换为UTF-8，以避免中文乱码问题-->
    <filter>
        <filter-name>encodingFilter</filter-name>
        <filter-class>org.springframework.web.filter.CharacterEncodingFilter
    </filter-class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
        <init-param>
            <param-name>forceEncoding</param-name>
            <param-value>true</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>encodingFilter</filter-name>
        <url-pattern>*.html</url-pattern>
    </filter-mapping>
    <servlet>
        <servlet-name>xiaochun</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>3</load-on-startup>
    </servlet>
    <!--⑥ 将以.html为后缀的URL由xiaochun Servlet处理-->
    <servlet-mapping>
        <servlet-name>xiaochun</servlet-name>
        <url-pattern>*.html</url-pattern>
    </servlet-mapping>
    <!--⑦ 浏览器不支持PUT、DELETE等方法，由该过滤器将xxx?_method=delete
    或 xxx?_method=put 转换为标准的HTTP DELETE、PUT方法 -->
    <filter>
        <filter-name>HiddenHttpMethodFilter</filter-name>
        <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter
    </filter-class>
    </filter>
    <filter-mapping>
        <filter-name>HiddenHttpMethodFilter</filter-name>
        <servlet-name>xiaochun</servlet-name>
    </filter-mapping>
    <!--⑧ 网站默认的面-->
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>

```

在⑥处对这个 Servlet 的 URL 路径映射进行定义，在这里让所有以.html 为后缀的 URL 都能被 xiaochun Servlet 截获，进而转由 Spring MVC 框架进行处理。由于浏览器不

支持 PUT、DELETE 等方法，我们在⑦处配置一个 HiddenHttpMethodFilter 过滤器，将 xxx?_method=delete 或 xxx?_method=put 转换为标准的 HTTP PUT、DELETE 方法。

为了避免中文乱码问题，一个不错的解决办法是所有的程序文件都采用 UTF-8 编码方式，因为不但中文系统支持 UTF-8，其他类型的操作系统也支持 UTF-8。如果统一采用 UTF-8，则令人反感的中文乱码问题可以得到有效解决。

18.8.8 Spring MVC 配置

编写好控制器、页面及 web.xml 配置之后，剩下的工作就是在 Spring 控制器配置文件 xiaochun-servlet.xml 中扫描 Web 路径，启动 Spring 控制器注解解析，指定 Spring MVC 的视图解析器及配置控制器异常统一拦截处理，如代码清单 18-32 所示。

代码清单 18-32 xiaochun-servlet.xml

```
<beans xmlns="http://www.springframework.org/schema/beans" ...
    xsi:schemaLocation=" ...
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd ">

    <!--① 自动扫描com.smart.web包下标注@Controller注解的类控制器类 -->
    <context:component-scan base-package="com.smart.web" />

    <!--② 启动Spring MVC的注解功能，完成请求和注解POJO的映射 -->
    <mvc:annotation-driven/>

    <!--③ 对模型视图名称的解析，在请求时模型视图名称添加前后缀 -->
    <bean
        class="org.springframework.web.servlet.view.InternalResourceViewResolver"
        p:prefix="/WEB-INF/jsp/" p:suffix=".jsp" />
    <bean id="multipartResolver"

        class="org.springframework.web.multipart.commons.CommonsMultipartResolver"
        p:defaultEncoding="utf-8" />
    <bean id="messageSource"
        class="org.springframework.context.support.ResourceBundleMessageSource"
        p:basename="i18n/messages" />

    <!--④ Web异常解析处理-->
    <bean id="exceptionResolver"
        class="com.smart.web.controller.ForumHandlerExceptionResolver">
        <property name="defaultErrorView">
            <value>fail</value>
        </property>
        <property name="exceptionMappings">
            <props>
                <prop key="java.lang.RuntimeException">fail</prop>
            </props>
        </property>
    </bean>
</beans>
```


Spring MVC 为视图名到具体视图的映射提供了多种可供选择的方法。在这里，我们使用 `InternalResourceViewResolver`，它通过为视图逻辑名添加前后缀的方式进行解析。如视图逻辑名为“login”，将解析为 `/WEB-INF/jsp/login.jsp`；名为“listAllBoards”，将解析为 `/WEB-INF/jsp/listAllBoards.jsp`。

18.9 对 Web 层进行测试

完成 Web 层控制器编写及 Spring MVC 配置之后，接下来开始编写论坛各控制器相应的测试用例。为了提高单元测试的运行效率，使用 Spring 在 `org.springframework.mock` 包中为一些依赖于容器的接口提供模拟类，这样就可以在不启动容器的情况下执行单元测试。

18.9.1 编写 Web 测试基类

为了简化每个控制器测试，我们编写了一个 Web 测试基类，所有的 Web 测试用例都需要扩展该基类，具体实现如代码清单 18-33 所示。

代码清单 18-33 BaseWebTest.java

```
package com.smart.web.controller;
import org.springframework.mock.web.MockHttpServletRequest;
import org.springframework.mock.web.MockHttpServletResponse;
...
@SpringApplicationContext( { "classpath:/applicationContext.xml",
"classpath:/xiaochun-servlet.xml" })
public class BaseWebTest extends UnitilsTestNG {
    //① 根据类型注入Bean
    @SpringBeanByType
    public AnnotationMethodHandlerAdapter handlerAdapter;

    //② 声明模拟对象
    public MockHttpServletRequest request;
    public MockHttpServletResponse response;
    public MockHttpSession session;

    //③ 执行测试前先初始化模拟对象
    @BeforeMethod
    public void before() {
        request = new MockHttpServletRequest();
        response = new MockHttpServletResponse();
        session = new MockHttpSession();
        request.setCharacterEncoding("UTF-8");
    }
}
```

在①处,使用 `Utils` 从 `Spring` 容器中加载 `AnnotationMethodHandlerAdapter` 实例,用于向控制器发送请求。在②处,声明 `Spring` 提供的 `Servlet API` 模拟类 `MockHttpServletRequest`、`MockHttpServletResponse` 及 `MockHttpSession`,并在测试初始化方法中进行实例化,用于发送请求信息及接收响应信息。

18.9.2 编写 ForumManageController 测试用例

编写好 Web 测试基类之后,接下来开始编写每个 Controller 相应的测试用例。在第 20 章中会对 `LoginController` 测试用例进行讲解,这里不再阐述。下面重点看一下论坛的核心控制器 `ForumManageController` 测试用例,如代码清单 18-34 所示。

代码清单 18-34 `ForumManageControllerTest.java`

```
package com.smart.web.controller;
import static org.hamcrest.Matchers.*;
import static org.testng.Assert.*;
...

public class ForumManageControllerTest extends BaseWebTest {

    //① 注入论坛管理控制器
    @SpringBeanByType
    private ForumManageController controller;

    //② 测试论坛首页处理控制器
    @Test
    public void listAllBoards()throws Exception {
        //②-1 设置请求URI及方法
        request.setRequestURI("/index");
        request.setMethod("GET");

        //②-2 调用控制器
        ModelAndView mav = handlerAdapter.handle(request, response, controller);
        List<Board> boards = (List<Board>)request.getAttribute("boards");

        //②-3 验证结果
        assertNotNull(mav);
        assertEquals(mav.getViewName(), "/listAllBoards");

        assertNotNull(boards);
        assertEquals(boards.size(), greaterThan(1));
    }

    //③ 测试跳转添加版块页面
    @Test
    public void addBoardPage()throws Exception {
        //③-1 设置请求URI及方法
        request.setRequestURI("/forum/addBoardPage");

        //请求方法要与控制器中@RequestMapping的设置方法一致
        request.setMethod("GET");
    }
}
```



```
//③-2验证结果
ModelAndView mav = handlerAdapter.handle(request, response, controller);
assertNotNull(mav);
assertEquals(mav.getViewName(), "/addBoard");
}
...
}
```

从 ForumManageController 测试用例来看,所有测试方法都比较好理解。首先通过模拟类 MockHttpServletRequest 设置请求 URI、参数及请求方法(如 POST、GET)。然后通过 Spring 提供的注解方法处理适配器向测试控制器 ForumManageController 发起请求。最后通过 TestNG 提供的断言及 Hamcrest 提供的匹配方法验证返回的结果。验证的返回结果主要有两个:一是验证返回视图的名称;二是验证返回视图的数据,如②-3 处所示。在编写 Web 控制器测试用例的过程中,唯一需要注意的是,测试方法中设置的请求方法与控制器中 @RequestMapping 的设置方法一致。版块管理控制器 BoardManageController 测试用例编写方法与论坛管理控制器一样,这里不再阐述,感兴趣的读者可以参考本书配套网盘中的示例代码。在 IDE 工具中,运行 ForumManageController 测试用例,将看到测试用例可在不启动 Web 容器的情况下顺利执行。

至此,我们全部完成了论坛持久层、服务层、Web 层的开发及单元测试工作,最一步就是部署和运行论坛应用。

18.10 开发环境部署

由于我们是基于 Maven 构建的工程,所以在 pom.xml 文件中配置 Jetty 应用服务器插件,如代码清单 18-35 所示。

代码清单 18-35 spring4x-chapter18 模块pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.
    org/xsd/maven-4.0.0.xsd">
...
<build>
  <plugins>
    <!-- Jetty插件 -->
    <plugin>
      <groupId>org.mortbay.jetty</groupId>
      <artifactId>maven-jetty-plugin</artifactId>
      <version>6.1.25</version>
      <configuration><!--配置说明 -->
        <connectors>
          <connector implementation="org.mortbay.jetty.nio.SelectChannel
            Connector">
            <port>80</port>
```

```
<maxIdleTime>60000</maxIdleTime>
</connector>
</connectors>
<contextPath>/forum</contextPath>
<scanIntervalSeconds>0</scanIntervalSeconds>
</configuration>
</plugin>
</plugins>
</build>
</project>
```

在章节模块中的插件节点 Plugins 下双击 jetty:run 或 jetty:run-exploded, 将以运行模式启动 Jetty 服务器。如果想要以 Debug 模式运行应用, 则通过右键菜单选择 Debug 运行应用即可。启动 Jetty 服务器, 在浏览器地址栏中输入 `http://localhost/forum/`, 将看到如图 18-25 所示的页面。

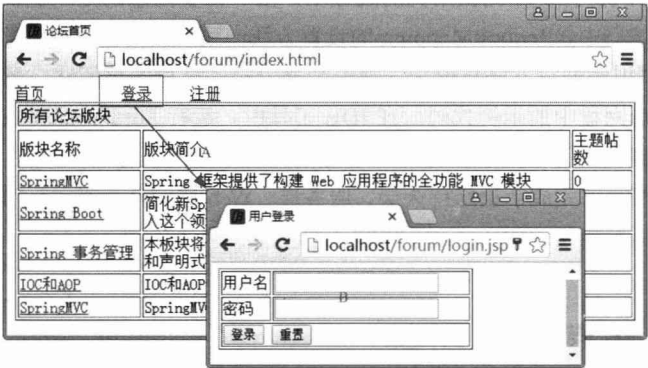


图 18-25 论坛首页及登录页面

单击“登录”链接, 转到 B 页面中, 登录完成后重新返回论坛首页, 这时将看到登录用户的信息, 原来的“登录”、“注册”链接变为“注销”链接。

单击论坛板块的链接, 转到论坛板块的主题帖子列表页面, 列出该版块所有的主题帖子, 如图 18-26 所示。



图 18-26 论坛版块页面

单击某一个主题帖子，将看到该主题帖子及其所有回帖的列表，如图 18-27 所示。

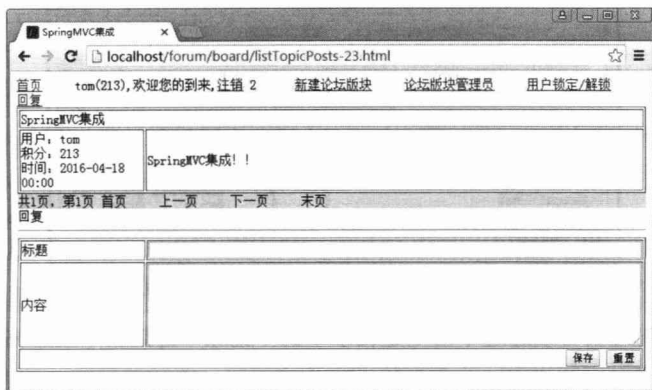


图 18-27 主题帖子页面

在所有回复帖子的下面有一个回复帖子的表单，可以通过这个表单新增回复帖子。如果登录用户是系统管理员，则其对应的操作页面如图 18-28 所示。

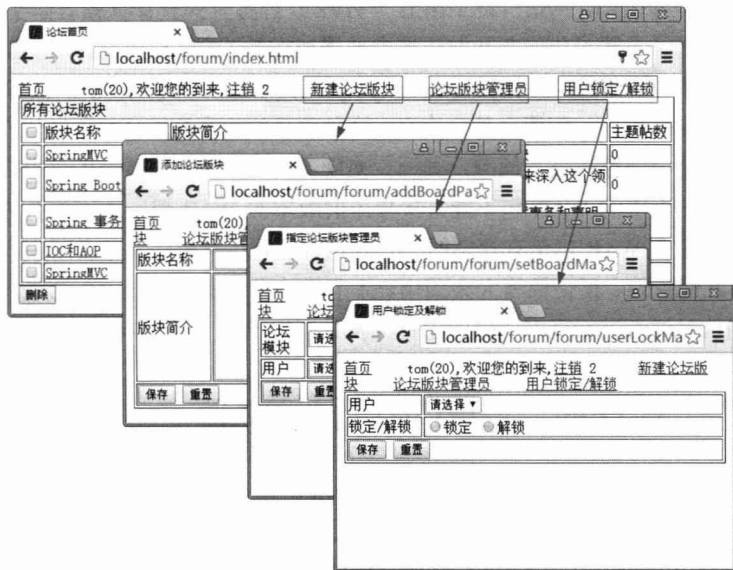


图 18-28 系统管理员的操作页面

论坛管理员登录后，论坛首页的顶部会自动显示出相应操作的链接，如新建论坛版块、论坛版块管理员及用户锁定/解锁，单击这些操作链接可以到达相应的操作页面。

如果用户是某一论坛版块的管理员，则论坛版块的主题帖子列表会自动显示出删除帖子和置精华帖子的操作链接，如图 18-29 所示。



图 18-29 论坛版块管理员所看到的主题帖子列表页面

论坛版块管理员可以通过单击主题帖子下的“删除”链接删除主题帖子，单击“置精华帖”链接将对应的主题帖子置为精华帖子。

18.11 项目配置实战经验

之前我们已经了解了如何在开发环境中部署和运行工程，然而项目的配置文件位于项目的类路径下（如 WEB-INF\classes\conf.properties），数据源也直接采用 DBCP、C3P0 等数据源，直接在 Spring 配置文件中定义。这种项目结构在开发环境下并没有问题，但如果在生产环境中则存在很大问题：其一，实施部署人员需要到 WAR 包中去更改配置信息；其二，新版本的 WAR 包不能直接覆盖旧版本的 WAR 包，否则原有的配置文件会被覆盖。即这个 WAR 包是有状态的，它既包含了程序代码，又包含了生产环境的配置信息，因此配置性、便捷性将大大弱化。本节将给出笔者在实践中总结出来的一个实战方案，相信对读者进行真刀实枪的项目开发具有指导意义。

18.11.1 “传统的” Web 项目属性文件

以下是一个传统的 Web 项目，在其 Spring 配置文件中通过<context:property-placeholder/>引用一个基于类路径的属性文件，如下：

```
<context:property-placeholder location="classpath:conf.properties"/>
<rop:annotation-driven
    core-pool-size="${rop.corePoolSize}"
    max-pool-size="${rop.maxPoolSize}"
    queue-capacity="${rop.queueCapacity}"
    keep-alive-seconds="${rop.keepAliveSeconds}"/>
```

将这个 Web 项目打包成 WAR 包后，在生产环境下如果要更改 conf.properties 配置文件中的参数，则必须在打开的 WAR 包中更改，虽然大部分 Web 应用服务器都会将 WAR 解压到一个文件目录中，但即便如此，对于开发和实施人员来说，仍需担心配置被升级覆盖的问题。

对于开发人员来说,每次打 WAR 包时都需要使用生产环境下的 `conf.properties` 替换开发环境下的 `conf.properties`,而开发人员往往并不清楚生产环境的具体情况。此时,开发人员只能请求实施人员不要用这个 WAR 包直接覆盖生产环境下的 WAR 包。

对于实施人员来说,他得战战兢兢地升级 WAR 包,先将生产环境下 WAR 包的 `conf.properties` 备份出来,然后使用新版 WAR 包,最后再将备份的 `conf.properties` 覆盖回来,以免生产环境下的 `conf.properties` 被覆盖了。当然,平常要更改配置参数也很麻烦,需要到每个 WAR 包中去更改。

18.11.2 如何规划便于部署的 Web 项目属性文件

为了方便 Web 项目的部署,必须实现属性文件和 WAR 包解耦,即 WAR 包中不包含配置文件。但这样问题又来了:难道在开发环境下,需要将配置文件放到一个项目之外的目录下?如此一来,项目团队开发岂不是很麻烦:团队成员从 SVN 迁出项目代码后,还需要手工复制项目的配置文件到一个工程外的指定配置文件目录下。

为了使项目既具有开发便捷性(SVN 一迁出工程, `mvn jetty:run` 即可启动项目),又具有部署便捷性(上传覆盖升级的 WAR 包即可,无须其他动作),只需引入一个 JVM 参数指定配置文件的位置,即可完美地满足开发便捷性和部署便捷性的需求。代码如下:

```
<plugin>
  <groupId>org.mortbay.jetty</groupId>
  <artifactId>maven-jetty-plugin</artifactId>
  <version>6.1.25</version>
  <configuration>
    <webDefaultXml>src/resources/dev/webdefault.xml</webDefaultXml>
    <systemProperties>
      <!-- ①注意这里,我们用一个JVM系统参数指定配置文件的路径 -->
      <systemProperty>
        <name>CONFFILE_PATH</name>
        <value>classpath:conf.properties</value>
      </systemProperty>
    </systemProperties>
    <webAppSourceDirectory>src/main/webapp</webAppSourceDirectory>
    <scanIntervalSeconds>0</scanIntervalSeconds>
    <contextPath>/forum</contextPath>
    <connectors>
      <connector implementation="org.mortbay.jetty.nio.SelectChannelConnector">
        <port>80</port>
      </connector>
    </connectors>
  </configuration>
</plugin>
```

我们在项目工程 `src/resources` 中拥有一个项目配置文件 `conf.properties`,该配置文件是和项目工程在一起的,专门供开发期使用。在 `pom.xml` 中采用 Jetty 插件,在①处通过 `CONFFILE_PATH` 的 JVM 系统参数指向类路径下的 `conf.properties`。

接下来对 Spring 配置文件进行调整,通过这个 CONFFILE_PATH 的 JVM 系统参数引用配置文件。

<!-- ① 注意这里,通过Spring EL表达式访问系统参数的值 -->

```
<context:property-placeholder location="#${systemProperties.CONFFILE_PATH}"/>
<rop:annotation-driven
    core-pool-size="${rop.corePoolSize}"
    max-pool-size="${rop.maxPoolSize}"
    queue-capacity="${rop.queueCapacity}"
    keep-alive-seconds="${rop.keepAliveSeconds}"/>
```

也就是说,在开发环境下,使用工程类路径下的 conf.properties 配置文件。在生产环境下,在 Web 应用服务器中定义 CONFFILE_PATH 的 JVM 系统参数,其值设为一个独立于 WAR 包的系统文件(如 file:/D:/etc/conf.properties),在 WebSphere 中定义 JVM 系统参数的界面如图 18-30 所示。

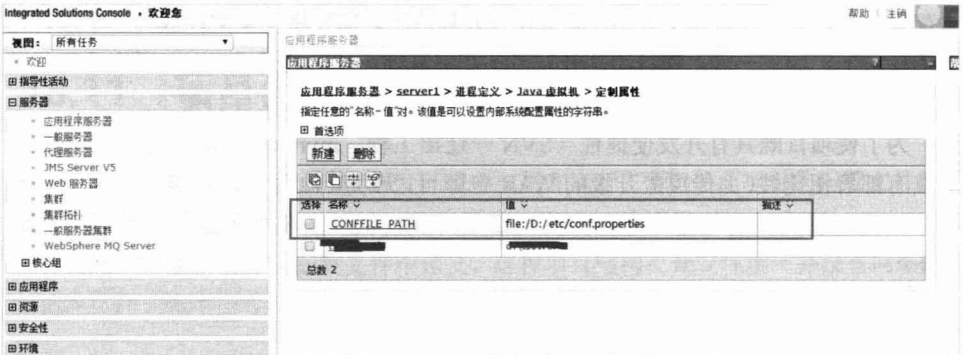


图 18-30 在 WebSphere 中定义 JVM 系统参数界面



提示

在 WebSphere 中定义 JVM 系统参数的操作步骤为:服务器→应用程序服务器(选择对应的应用程序服务器)→服务器基础结构(Java 和进程管理/进程定义)→其他属性(Java 虚拟机)→其他属性(定制属性)。对于 Tomcat, Jetty 都可在应用服务器的配置文件中定义 JVM 参数。

这样,在生产环境下, Spring 将根据 CONFFILE_PATH 的 JVM 系统参数读取 file:/D:/etc/conf.properties 配置文件。由于配置文件已经放在独立于 WAR 包的外部操作系统的文件目录下,如此一来,生产环境下的 WAR 包升级时,直接覆盖旧版 WAR 包即可,不用再考虑新版 WAR 包覆盖配置文件的问题。

18.11.3 数据源的配置

Web 应用基本上离不开数据源,传统的 Web 项目大多数采用 DBCP、C3P0 等数据源直接在 Spring 配置文件中定义,这样会给部署带来问题。

(1) 无法共享数据源。如果在一个 Web 应用服务器中存在多个 WAR 项目，那么这些 WAR 项目都使用同一个数据库。由于采用在各自项目中独立配置数据源的方式，因而会造成创建多个数据源实例的问题，无法实现数据源的共享。

(2) 数据源的优化不方便。如果在 Spring 配置文件中配置 DBCP、C3P0 等数据源，则数据源的优化参数将被抽取到配置文件中。由于数据源的优化参数很多，往往并不能将所有的配置参数都抽取到配置文件中，所以数据源的优化将受限。

因此，应让 Spring 引用 JNDI 数据源，而不应在 Spring 配置文件中通过 DBCP、C3P0 等自行定义数据源。

当然，一个很现实的问题是如何在开发环境下使用 JNDI 数据源。其实这已经不是问题了，Tomcat、Jetty 很早就可以定义 JNDI 数据源，并不是只有 WebSphere、WebLogic 才可以定义 JNDI 数据源。

下面是使用 Maven 的 maven-jetty-plugin 插件定义 JNDI 数据源的配置。

```
<plugin>
  <groupId>org.mortbay.jetty</groupId>
  <artifactId>maven-jetty-plugin</artifactId>
  <version>6.1.25</version>
  <configuration>
    <webDefaultXml>src/resources/dev/webdefault.xml</webDefaultXml>
    <!--①添加一个Jetty的额外配置文件-->
    <jettyEnvXml>src/resources/jetty.xml</jettyEnvXml>
    <systemProperties>
      <systemProperty>
        <name>CONFFILE_PATH</name>
        <value>classpath:conf.properties</value>
      </systemProperty>
    </systemProperties>
    <webAppSourceDirectory>src/main/webapp</webAppSourceDirectory>
    <scanIntervalSeconds>0</scanIntervalSeconds>
    <contextPath>/forum</contextPath>
    <connectors>
      <connector implementation="org.mortbay.jetty.nio.SelectChannelConnector">
        <port>80</port>
      </connector>
    </connectors>
  </configuration>
</plugin>
```

在①处引用了一个与 pom.xml 位于相同目录的 Jetty 配置文件 jetty.xml，其配置内容如下：

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure//EN"
  "http://jetty.mortbay.org/configure.dtd">
<Configure class="org.mortbay.jetty.webapp.WebAppContext">

  <New class="org.mortbay.jetty.plus.naming.Resource">
    <Arg>jdbc/myds</Arg>
    <Arg>
      <New class="org.apache.commons.dbcp.BasicDataSource">
```



```

        <Set name="driverClassName">com.mysql.jdbc.Driver</Set>
        <Set name="url">
            <![CDATA[jdbc:mysql://localhost:3306/sampled]]>
        </Set>
        <Set name="username">root</Set>
        <Set name="password">123456</Set>
    </New>
</Arg>
</New>
</Configure>

```

这里定义了一个名为 jdbc/myds 的 JNDI 数据源，这样项目的 Spring 配置文件就可直接引用这个 JNDI 数据源了。

```

<?xml version="1.0" encoding="UTF-8" ?>
...
<jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/myds"/>
</beans>

```

由于 JNDI 数据源名称不是固定的，比如在某个部署环境下可能为 jdbc/yourds，因此需要将 JNDI 的名字属性化。结合上面所说的配置文件，可以在配置文件中添加一个属性，暂且将该属性命名为 ds.jndi。

```

conf.properties
ds.jndi=java:comp/env/jdbc/yourds

```

然后在 Spring 配置文件中通过参数名引用 JNDI 数据源。

```

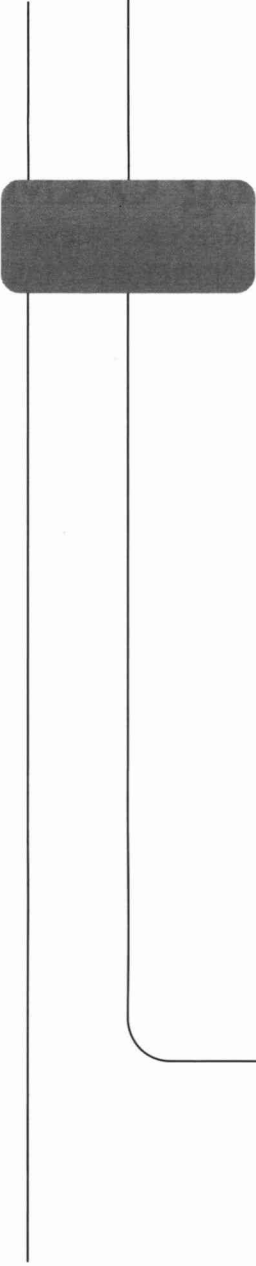
<?xml version="1.0" encoding="UTF-8" ?>
...
<context:property-placeholder location="#{systemProperties.CONFFILE_PATH}"/>
<jee:jndi-lookup id="dataSource" jndi-name="${ds.jndi}"/>
</beans>

```

至此，开发和部署人员都皆大欢喜了。在 JNDI 数据源已经存在的情况下，只需在 conf.properties 中定义好 JNDI 数据源名称，项目即可启动。

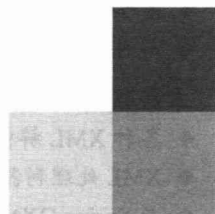
18.12 小结

在本章中，我们开发了一个实际论坛的应用案例，该案例的技术框架采用 Spring+Hibernate 的组合框架。我们从需求、设计、开发、单元测试、部署（开发环境和生产环境部署）典型软件开发过程讲解论坛应用的整体开发过程。当然，由于篇幅的限制，还留有一些未尽的工作，如严格的权限控制、用户帖子管理、投票帖子等，有兴趣的读者可以在本案例的基础上进一步完善这个论坛，让它更接近实际应用。



第 5 篇

提 高 篇



第 19 章

Spring OXM

本章从 XML 各种解析技术的发展历程讲起，并介绍另外一些主流 O/X Mapping 组件的使用方法。通过这些实例讲解，读者可以快速了解目前各种 O/X Mapping 技术。本章筛选了目前一些流行的 O/X Mapping 组件，如 XStream、Castor、JiBX、JAXB 等，从独立使用到与 Spring 整合逐步演进，逐步揭开各 O/X Mapping 组件的神秘面纱。通过本章的学习，读者可以根据需要，选用合适的 O/X Mapping 组件来处理对象 XML 之间的转换，为开发 Web Services 应用打下良好的基础。

本章主要内容：

- ◆ 认识 XML 解析技术
- ◆ XML 处理利器：XStream
- ◆ 基于 Castor 对象的 XML 映射
- ◆ 基于 JiBX 对象的 XML 映射
- ◆ 基于 JAXB 对象的 XML 映射
- ◆ 与 Spring OXM 整合

本章亮点：

- ◆ 各种 XML 解析技术特点分析
- ◆ XML 处理利器：XStream 及其他 O/X Mapping 组件讲解
- ◆ 与 Spring OXM 整合

19.1 认识 XML 解析技术

19.1.1 什么是 XML

XML (Extensible Markup Language) 即可扩展标记语言，从 1996 年开始有其雏形，

并向 W3C（全球信息网联盟）提案，于 1998 年 2 月发布为 W3C 的标准（XML 1.0）。XML 的前身是 SGML（the Standard Generalized Markup Language），是 IBM 从 20 世纪 60 年代就开始发展的 GML（Generalized Markup Language）标准化后的名称。其目的是促进 Internet 上结构化文档的交换。简单地说，XML 是一组规则和准则的集合，用来描述结构化数据。

XML 为描述结构良好的文档提供了一整套灵活的语法。正因为它的这种灵活性，我们需要一些方法来验证 XML 文档是否与我们预计的格式保持一致。于是人们就逐步提出 DTD 和 XML Schema。

DTD 是一套关于标记符的语法规则，它是 XML 1.0 规范的一部分，是 XML 文件的验证机制，属于 XML 文件组成的一部分。DTD 是一种保证 XML 文档格式正确的方法，可以通过比较 XML 文档和 DTD 文件来看文档是否符合规范，以及元素和标签使用是否正确。

XML Schema 指定 XML Schema 定义语言，该语言提供了描述 XML 1.0 文档结构和限制其内容的工具，其中包括那些利用 XML Namespace 的工具。模式语言自身用 XML 1.0 表示并使用名称空间，它在很大程度上重构了 XML 1.0 DTD 的能力，并在一定程度上解决了 DTD 的许多局限性，如：

- ☐ 不支持名称空间。
- ☐ 对模块化和重用的支持非常有限。
- ☐ 不支持对声明的扩展或继承。
- ☐ 编写、维护和读取大型 DTD 以及定义系列相关模式都很困难。
- ☐ 没有嵌入式、结构化自我文档编制。
- ☐ 内容和属性声明不能依靠属性或元素上下文。

XML 不仅仅是一种标记语言，而是一系列技术，如 DTD、XSD、DOM、SAX、XSL、XLink、XPointer、SMIL 等的集合体。这一技术家族为我们开发可扩展性和互操作性的软件提供了一种解决方案，目前广泛应用于系统配置、数据存储及数据交换的格式等，它是 Web 服务的基础，也是现代面向服务的架构（Service-Oriented Architecture, SOA）设计模式的基础。

19.1.2 XML 的处理技术

为了有效使用 XML，需要通过一个 XML 处理器或 XML API 来访问其数据。目前 JAXP 1.6（JSR 206）的两种处理 XML 文档的方法已经得到广泛应用，分别是 DOM（Document Object Model）和 SAX（Simple API for XML）。

DOM 文档对象模型是一种通过编程方式对 XML 文档中的数据及结构进行访问的标准，基于 XML 文档在内存中的树状结构，当一个 XML 文件被装入处理器时，内存中建立起一棵相应的树。DOM 还定义了用来遍历一棵 XML 树及管理各个元素、值和

属性的编程接口（包括方法和属性的名字）。

DOM 标准的一个主要不足在于将整个 XML 文档装入内存所引起的巨大内存开销。当文件的数据量非常大时，这会带来很大的性能瓶颈。于是人们就开始创立一种新的标准，这就是 SAX。

SAX 是一种非常简单的 XML API，它允许开发者使用事件驱动的 XML 解析。与 DOM 不同，SAX 并不要求将整个 XML 文件一起装入内存。它的想法十分简单，一旦 XML 处理器完成对 XML 元素的操作，它就立刻调用一个自定义事件处理器及时处理这个元素及相关数据。

虽然 SAX 解决了 DOM 速度慢、内存占用大的问题，但在灵活性方面受到很大制约，如无法随机访问文档。于是又有一种新的基于流的 Stream API for XML（简称 StAX）逐步出现在人们的视野中，它不仅提高了 XML 的处理速度，而且较好地兼顾了灵活性。StAX 是 JSR 173 标准，目前已经加入 Java 6.0 的 JAXP 1.4 里面。

StAX 如其名称所暗示的那样，把重点放在流上。实际上，StAX 与其他方法的区别就在于应用程序能够把 XML 当作一个事件流来处理。将 XML 当作一个事件流来处理的想法并不新颖（事实上 SAX 已经提出来了），但 StAX 并不使用 SAX 的“推”模型，而是使用“拉”模型进行事件处理。此外，StAX 解析器也不使用回调机制，而是根据应用程序的要求返回事件。StAX 还提供了用户友好的 API，用于读入和写出。

StAX 实际上包括两套处理 XML 的 API，分别提供了不同程度的抽象。基于指针的 API 允许应用程序把 XML 作为一个标记（或事件）流来处理。应用程序可以检查解析器的状态，获得解析的上一个标记信息，然后再处理下一个标记，依此类推。这是一种低层 API，尽管效率高，但是没有提供底层 XML 结构的抽象。较为高级的基于迭代器的 API 允许应用程序把 XML 作为一系列事件对象来处理，每个对象和应用程序交换 XML 结构的一部分。应用程序只需确定解析事件的类型，将其转换成对应的具体类型，然后利用其方法获得属于该事件的信息即可。

DOM、SAX、StAX 技术都是从 XML 的角度来处理文档和建立模型，这对于只关注文档 XML 结构的应用程序来说是适用的，但是很多应用程序仅仅将 XML 作为数据交换的媒介，更多关注的是文档数据本身，为此人们又提出了一种 XML 数据绑定技术，可以使应用程序在很大程度上忽略 XML 文档的实际结构，而直接使用文档的数据内容。

数据绑定是指将数据从一些存储媒介（如 XML 文档和数据库）中取出，并通过程序表示这些数据的过程，即把数据绑定到虚拟机能够理解并且可以操作的某种内存结构中。数据绑定并不是一个新鲜的概念，它在关系数据库上早已得到了广泛的应用，如 Hibernate 就是针对数据库的轻量级数据绑定框架。而针对 XML 数据绑定的 Castor 框架在 2000 年就已经出现，目前已经涌现出许多类似的框架，如 JAXB、JiBX、Quick 和 Zeus 等。

19.2 XML 处理利器：XStream

19.2.1 XStream 概述

XStream 是一套简洁易用的开源类库，用于将 Java 对象序列化为 XML 或者将 XML 反序列化为 Java 对象，是 Java 对象和 XML 之间的一个双向转换器。2004 年 1 月发布了第一个版本 XStream 0.3，2016 年 3 月发布了 XStream 1.4.9 版本。

1. XStream 主要特点

- ☐ 灵活易用：提供了简单、灵活、易用的统一接口，用户无须了解底层细节。
- ☐ 无须映射：大多数对象都可以在无须映射的情况下进行序列化与反序列化的操作。
- ☐ 高速稳定：解析速度快，占用内存少。
- ☐ 灵活转换：转换策略是可以定制的，允许自定义类型存储为指定的 XML 格式。
- ☐ 易于集成：通过实现特定的接口，可以直接与其他任何树形结构进行序列化与反序列化操作。

2. XStream 架构组成

1) Converters（转换器）

当 XStream 遇到需要转换的对象时，它会委派给合适的转换器实现。XStream 为通用类型提供了多种转换器实现，包括基本数据类型、String、Collections、Arrays、null、Date 等。

2) I/O（输入/输出）

XStream 是通过接口 HierarchicalStreamWriter 和 HierarchicalStreamReader 从底层 XML 数据中抽象而来的，分别用于序列化和反序列化操作。

3) Context（上下文引用）

在 XStream 序列化或反序列化对象时，它会创建两个类 MarshallingContext 和 UnmarshallingContext，由它们来处理数据并委派合适的转换器。XStream 提供了三种上下文的默认实现，它们之间存在细微的差别。默认值可以通过 XStream.setMode()方法调整，可选值如下。

- (1) XStream.XPATH_REFERENCES：（默认的）通过 XPath 引用来标识重复的引用。
- (2) XStream.ID_REFERENCES：使用 ID 引用来标识重复的引用。
- (3) XStream.NO_REFERENCES：对象作为树形结构，重复的引用被视为两个不同的对象，循环引用会导致异常产生。这种模式速度快，占用内存少。

4) Facade（统一入口）

作为 XStream 的统一入口点，它将上面所提及的重要组件集成在一起，以统一的接口开放出来。

19.2.2 快速入门

本小节通过一个实例开始 XStream 的应用之旅,采用 User 用户领域对象及 LoginLog 登录日志领域对象作为 XStream 实例转换对象,如代码清单 19-1 和代码清单 19-2 所示。

代码清单 19-1 User.java

```
package com.smart.oxm.domain;
...
public class XStreamSample {
    private int userId;
    private String userName;
    private String password;
    private int credits;
    private String lastIp;
    private Date lastVisit;
    private List logs;

    //set、get方法省略
}
```

代码清单 19-2 LoginLog.java

```
package com.smart.oxm.domain;
...
public class XStreamSample {
    private int loginLogId;
    private int userId;
    private String ip;
    private Date loginDate;

    //set、get方法省略
}
```

在开始实例的开发之前,要先将 XStream 的依赖加入 pom.xml 文件中,本实例采用的是 XStream 1.4.9 版本。代码如下:

```
<dependency>
    <groupId>com.thoughtworks.xstream</groupId>
    <artifactId>xstream</artifactId>
    <version>1.4.9</version>
</dependency>
```

接下来开始使用 XStream 进行对象与 XML 之间的互换,如代码清单 19-3 所示。

代码清单 19-3 XStreamSample.java

```
package com.smart.oxm.xstream;
...
public class XStreamSample {
    private static XStream xstream;
    static{
        //① 创建XStream实例并指定一个XML解析器
        xstream = new XStream(new DomDriver());
    }

    //初始化转换对象
```



```

public static User getUser(){
    LoginLog log1 = new LoginLog();
    log1.setIp("192.168.1.91");
    log1.setLoginDate(new Date());
    User user = new User();
    user.setUserId(1);
    user.setUserName("xstream");
    user.addLoginLog(log1);
    return user;
}

//② Java对象转换为XML
public static void objectToXML()throws Exception{

    //②-1 获取转换的User对象实例
    User user = getUser()

    //②-2 实例化一个文件输出流
    FileOutputStream outputStream = new FileOutputStream("out/XStreamSample.xml");
    //②-3 将User对象转换为XML, 并保存到指定文件
    xstream.toXML(user, outputStream);
}

//③ XML转换为Java对象
public static void XMLToObject()throws Exception{

    //③-1 实例化一个文件输入流
    FileInputStream inputStream= new FileInputStream("out/XStreamSample.xml ");

    //③-2 将XML文件输入流转换为对象
    User user = (User) xstream.fromXML(inputStream);

    for(LoginLog log : user.getLogs()){
        if(log!=null){
            System.out.println("访问IP: " + log.getIp());
            System.out.println("访问时间: " + log.getLoginDate());
        }
    }
}
...
}

```

在①处, 创建一个 XStream 实例, 并指定一个 DOM XML 解析器。如果不指定一个 XML 解析器, 则 XStream 会采用默认的 XPP (XML Pull Parser, 一种高速解析 XML 文件的方式, 速度要比传统方式快很多) 解析器来解析 XML 文件。

在②-1 处, 首先实例化需要转换为 XML 的类, 如实例中的 User、LoginLog。在②-2 处, 创建一个用于输出 XML 文件的文件输出流。在②-3 处, 调用 XStream#toXML() 方法将 User 实例转换为 XML 格式, 并转出到指定的文件流中。同理, 如果将 XML 文件转换为对象, 则需要先读取要转换的 XML 文件到文件流中, 如③-1 处所示, 然后调用 XStream#fromXML() 方法将输入文件流转换为对象。在 IDE 工具中执行当前实例, 在类路径下生成一个 XML 文件, 如下所示:

```

<com.smart.oxm.domain.User>
  <userId>1</userId>
  <userName>xstream</userName>
  <credits>0</credits>
  <logs>
    <com.smart.oxm.domain.LoginLog>
      <loginLogId>0</loginLogId>
      <userId>0</userId>
      <ip>192.168.1.91</ip>
      <loginDate>2016-03-06 21:05:37.879 UTC</loginDate>
    </com.smart.oxm.domain.LoginLog>
    <com.smart.oxm.domain.LoginLog>
      <loginLogId>0</loginLogId>
      <userId>0</userId>
      <ip>192.168.1.92</ip>
      <loginDate>2016-03-06 21:05:37.879 UTC</loginDate>
    </com.smart.oxm.domain.LoginLog>
  </logs>
</com.smart.oxm.domain.User>.

```

19.2.3 使用 XStream 别名

在默认情况下, Java 对象到 XML 的映射是 Java 对象属性名对应 XML 的元素名, Java 类的全名对应 XML 根元素的名字。在实际应用中, 如果 XML 和 Java 类都已经存在相应的名字, 那么, 在进行转换时, 需要设置别名进行映射。

XStream 别名配置包含三种情况。

- ☐ 类别名: 用 `alias(String name, Class type)`。
- ☐ 类成员别名: 用 `aliasField(String alias, Class definedIn, String fieldName)`。
- ☐ 类成员作为属性别名: 用 `aliasAttribute(Class definedIn, String attributeName, String alias)`。单独命名没有意义, 还要通过 `useAttributeFor(Class definedIn, String fieldName)` 应用到某个类上。

从上一个实例生成的 XML 文件来看, 生成的 XML 元素结构不是很友好。接下来使用 XStream 提供的别名机制来修饰生成 XML 元素的结构, 如代码清单 19-4 所示。

代码清单 19-4 XStreamAliasSample.java

```

package com.smart.oxm.xstream.alias;
...
public class XStreamAliasSample{
...

    static{
        ...

        //① 设置类别名, 默认为当前类名称加上包名
        xstream.alias("loginLog", LoginLog.class);
        xstream.alias("user", User.class);
    }
}

```

```

//② 设置类成员别名
xstream.aliasField("id", User.class, "userId");

//③ 把LoginLog的userId属性视为 XML属性, 默认为XML的元素
xstream.aliasAttribute(LoginLog.class, "userId", "id");
xstream.useAttributeFor(LoginLog.class, "userId");

//④ 去掉集合类型生成XML的父节点, 即忽略XML中的 <logs></logs>标记
xstream.addImplicitCollection(User.class, "logs");
}
...
}

```

在①处, 通过 `XStream#alias()` 方法来设置类别名, 默认为当前类的全限定名。在②处, 通过 `XStream#aliasField()` 方法将 `User` 类的 `userId` 属性设置为 `id`。在③处, 通过 `XStream#aliasAttribute()` 和 `XStream#useAttributeFor()` 方法将 `LoginLog` 类的 `userId` 属性设置为 `id`, 并设置为 `LoginLog` 元素的属性, 默认为 `LoginLog` 元素的子元素。在④处, 通过 `XStream#addImplicitCollection()` 方法删除集合节点 `logs`, 即忽略 XML 中的 `<logs></logs>` 标记。

在 IDE 工具中执行当前实例, 在类路径下生成一个 XML 文件, 如下所示:

```

<user>
  <id>1</id>
  <userName>xstream</userName>
  <credits>0</credits>
  <loginLog id="0">
    <loginLogId>0</loginLogId>
    <ip>192.168.1.91</ip>
    <loginDate>2016-03-17 05:12:39.826 UTC</loginDate>
  </loginLog>
  ...
</user>

```

19.2.4 XStream 转换器

在开发过程中, 有时可能需要转换一些自定义类型, 此时默认的映射方式可能无法满足需要。不用担心, `XStream` 已经提供了丰富的扩展点, 用户可以实现自己的转换器, 然后调用 `registerConverter()` 方法注册自定义的转换器。实现自定义的转换器很简单, 只要实现 `XStream` 提供的 `Converter` 接口并实现其方法即可。

上一个实例已成功地应用 `XStream` 进行对象与 XML 的相互转换, 并应用 `XStream` 提供的别名机制设置生成 XML 的结构。下面使用 `XStream` 提供的转换器扩展接口, 对生成的 XML 文件继续优化。

首先需要实现 `Converter` 接口来编写一个日期转换器, 具体的实现如代码清单 19-5 所示。

代码清单 19-5 DateConverter.java

```

package com.smart.oxm.xstream.converters;
...
public class DateConverter implements Converter {
    private Locale locale;
    public DateConverter(Locale locale) {
        super();
        this.locale = locale;
    }
    //①实现该方法，判断要转换的类型
    public boolean canConvert(Class clazz) {
        return Date.class.isAssignableFrom(clazz);
    }

    //②实现该方法，编写Java对象到XML的转换逻辑
    public void marshal(Object value, HierarchicalStreamWriter writer,
        MarshallingContext context){
        DateFormat formatter = DateFormat.getDateInstance(DateFormat.DATE_FIELD,
            this.locale);
        writer.setValue(formatter.format(value));
    }

    //③实现该方法，编写XML到Java对象的转换逻辑
    public Object unmarshal(HierarchicalStreamReader reader,
        UnmarshallingContext context) {
        GregorianCalendar calendar = new GregorianCalendar();
        DateFormat formatter = DateFormat.getDateInstance(DateFormat.DATE_FIELD,
            this.locale);
        try {
            calendar.setTime(formatter.parse(reader.getValue()));
        } catch (ParseException e) {
            throw new ConversionException(e.getMessage(), e);
        }
        return calendar.getGregorianCalendar();
    }
}

```

在①处，通过实现 `ConverterMatcher#canConvert()` 方法来判断转换逻辑。在②处，通过实现 `Converter#marshal()` 方法，完成对象编组（对象转换为 XML）操作时的处理逻辑。在③处，通过实现 `Converter#unmarshal()` 方法，完成对象反编组（XML 转换为对象）操作时的处理逻辑。编写好转换器之后，剩下的工作就是调用 `XStream#registerConverter()` 方法注册自定义的日期转换器。

在 IDE 工具中执行当前实例，将会在类路径下生成一个 XML 文件，如下所示：

```

<user>
  <userId>1</userId>
  <userName>xstream</userName>
  <credits>0</credits>
  <logs>
    <loginLog>
      <loginLogId>0</loginLogId>
      <userId>0</userId>
      <ip>192.168.1.91</ip>
    </loginLog>
  </logs>
</user>

```

```
<loginDate> 2016年3月6日 星期六 </loginDate>
  </loginLog>
  ...
</logs>
</user>
```

19.2.5 XStream 注解

XStream 不但可以通过编码的方式对 XML 进行转换，而且支持基于注解的方式进行转换。下面使用 XStream 提供的注解（见表 19-1）来改造上面的实例。首先需要对 User 和 LoginLog 添加相应的注解，如代码清单 19-6 所示。

代码清单 19-6 User.java

```
package com.smart.oxm.xstream.annotations;
...
@XStreamAlias("user")
public class User {
    @XStreamAsAttribute
    @XStreamAlias("id")
    private int userId;

    @XStreamAlias("username")
    private String userName;

    @XStreamAlias("password")
    private String password;

    @XStreamAlias("credits")
    private int credits;

    @XStreamAlias("lastIp")
    private String lastIp;

    @XStreamConverter(DateConverter.class)
    private Date lastVisit;

    @XStreamImplicit
    private List logs;
    ...
}
```

表 19-1 XStream常用注解

注 解	说 明	作用目标
@XStreamAlias	别名注解	类，字段
@XStreamAsAttribute	转换成属性	字段
@XStreamOmitField	忽略字段	字段
@XStreamConverter	注入转换器	对象
@XStreamImplicit	隐式集合	集合字段

其中, `@XStreamAlias` 为别名注解, 一般作用于目标类或字段, 如实例中的 `@XStreamAlias("user")`、`@XStreamAlias("id")`。`@XStreamConverter` 注解用于注入自定义的转换器, 如实例中的 `@XStreamConverter(DateConverter.class)`, 注入一个日期转换器。对于集合类型, 可以使用 `@XStreamImplicit` 注解来隐藏, 其作用与 `XStream#addImplicitCollection()` 方法一样。`@XStreamAsAttribute` 注解将 Java 对象属性映射为 XML 元素的一个属性。`@XStreamOmitField` 注解标注 Java 对象的属性将不再出现在 XML 中。`LoginLog` 对象添加注解的方式与 `User` 一样, 这里不再阐述。下面来看一下如何应用 `XStream` 提供的注解进行 Java 对象与 XML 的相互转换, 如代码清单 19-7 所示。

代码清单 19-7 XStreamAnnotationSample.java

```
package com.smart.oxm.xstream.annotations;
package com.smart.oxm.xstream.annotations;
import com.thoughtworks.xstream.XStream;
import com.thoughtworks.xstream.io.xml.DomDriver;
...
public class XStreamAnnotationSample {
    private static XStream xstream;
    ...
    static{
        xstream = new XStream(new DomDriver());

        //① 实现该方法, 判断要转换的类型
        xstream.processAnnotations(User.class);
        xstream.processAnnotations(LoginLog.class);
        //② 自动加载注解Bean
        //xstream.autodetectAnnotations(true);
    }

    //Java对象转换为XML
    public static void objectToXml()throws Exception{
        User user = getUser();
        FileOutputStream fs = new FileOutputStream("out/XStreamAnnotationSample.xml");
        OutputStreamWriter writer = new OutputStreamWriter(fs, Charset.forName("UTF-8"));
        xstream.toXML(user, writer);
    }
    ...
}
```

要启用 `XStream` 提供的注解功能, 需要在执行对象与 XML 转换之前先注册标注了 `XStream` 注解的 Java 对象, 如实例中在①处通过 `XStream#processAnnotations()` 方法手工注册 `User` 和 `LoginLog` 对象。`XStream` 除了手工注册外, 还提供一个自动检测标注了 `XStream` 注解的 Java 对象的方法 `XStream#autodetectAnnotations()`, 如实例中的②处所示。自动检测方法不仅使用方便, 而且还提供了缓存机制来缓存所有标注了 `XStream` 注解的 Java 对象。

19.2.6 流化对象

XStream 为 `java.io.ObjectInputStream` 和 `java.io.ObjectOutputStream` 提供替代的实现，允许以对象流方式进行 XML 序列化或反序列化操作。这对于处理集合对象非常有用 (`List<User> users`)，在内存中只保留一个 User 对象流。很显然，我们应该使用基于流而非 DOM 的 XML 解析器读取 XML，以提高性能。

创建一个输出流，至于怎么输出可以使用多种方法，其原理是一样的。在这里就不得不提到 `HierarchicalStreamWriter`。`HierarchicalStreamWriter` 是一个接口，从字面意思上来说它是有层级关系的输出流。`XStream` 默认提供了几个常用的实现类用于输出，如 `CompactWriter`、`PrettyPrintWriter`。下面应用 `XStream` 流化对象来处理 XML 序列化及反序列化，如代码清单 19-8 所示。

代码清单 19-8 ObjectStreamSample.java

```
package com.smart.oxm.xstream;
...
public class ObjectStreamSample {
    private static XStream xstream;
    static{
        xstream = new XStream();
    }

    //Java对象转换为XML
    public void objectToXml() throws Exception{
        //① 实例化序列化对象
        User user = getUser();

        //② 创建一个PrintWriter对象，用于输出
        PrintWriter pw=new PrintWriter("out/ObjectStreamSample.xml");

        //③ 选用一个HierarchicalStreamWriter的实现类来创建输出
        PrettyPrintWriter ppw=new PrettyPrintWriter(pw);
        //CompactWriter cw=new CompactWriter(pw);

        //④ 创建对象输出流
        ObjectOutputStream out = xstream.createObjectOutputStream(ppw);
        out.writeObject(user);
        out.close();
    }

    //XML转换为Java对象
    public User xmlToObject()throws Exception{

        //⑤ 通过对象流进行输入操作
        FileReader reader=new FileReader("out/ObjectStreamSample.xml");
        BufferedReader bufferedReader =new BufferedReader(reader);

        //⑥ 创建对象输入流
        ObjectInputStream input = xstream.createObjectInputStream(bufferedReader);
```



```

//⑦ 从XML文件中读取对象
User user=(User)input.readObject();
return user;
}
...
}

```

通过对象流进行输出操作，首先需要实例化转换的对象，如实例中的①处所示。然后在②处创建一个 `PrintWriter` 对象，将对象序列化到指定的 XML 文件中。在③处，选用一个 `HierarchicalStreamWriter` 的实现类 `PrettyPrintWriter` 来创建输出，也可以采用 `CompactWriter` 实现类。`CompactWriter` 与 `PrettyPrintWriter` 的区别在于，用 `CompactWriter` 方法输出的是连续的没有分隔的 XML 文件，而用 `PrettyPrintWriter` 方法输出的是有分隔的有一定格式的 XML 文件。在④处，调用 `XStream#createObjectOutputStream()` 方法创建对象输出流。

通过对象流进行输入操作，需要使用 `FileReader` 和 `BufferedReader` 读取指定的 XML 文件，如实例中的⑤处所示。在⑥处，调用 `XStream#createObjectInputStream()` 方法创建对象输入流，最后从该输入流读取对象。

19.2.7 持久化 API

如果需要将一个集合中的所有对象持久化到文件系统中，通常会使用 `java.io` 的 API 将集合的对象逐个输入到文件中。虽然这个方法很简单，但过程比较复杂。`XStream` 提供了相关集合类的接口实现类，如 `XmlArrayList`、`XmlSet`、`XmlMap`，用一个简单的方法就可以将集合中的每个对象持久化到文件中。下面应用 `XmlArrayList` 来持久化一个集合中的所有对象，具体实现如代码清单 19-9 所示。

代码清单 19-9 PersistenceSample.java

```

package com.smart.oxm.xstream.persistence;
...
public class PersistenceSample {
    ...
    public void persist() {

        //① 实例化需要持久化的对象
        users = new ArrayList<User>();
        User user = new User();
        user.setUserId(1);
        user.setCredits(10);
        user.setUserName("tom");
        user.setPassword("123456");
        users.add(user);

        //② 创建持久化策略
        File file = new File("out");
        PersistenceStrategy strategy = new FilePersistenceStrategy(file);
    }
}

```

```
//③ 持久化集合对象
```

```
List list = new XmlArrayList(strategy);
list.addAll(users);
```

```
}
```

```
...
```

XStream 为 XmlArrayList、XmlSet、XmlMap 实现类提供统一的创建接口，在创建 XmlArrayList（及相关的集合）时，需要指定一个持久化策略 PersistenceStrategy。XStream 提供一个持久化到文件的策略 FilePersistenceStrategy。这个策略将集合中的每个对象持久化到指定目录不同的文件中，如 int@0.xml、int@1.xml。

19.2.8 额外功能：处理 JSON

目前，在 Web 开发领域，主要的数据交换格式有 XML 和 JSON，相信每个 Web 开发者对二者都不会感到陌生。

JSON (JavaScript Object Notation) 是一种轻量级的数据交换格式，易于阅读和编写，同时也易于机器解析和生成。它是基于 1999 年颁布的 ECMA-262 标准的 JavaScript 语言的一个子集。JSON 采用完全独立于语言的文本格式，但是也使用了类似于高级语言的一些习惯（如 Java、C#等），这些特性使 JSON 成为理想的数据交换语言。虽然目前 XML 是业界数据交换标准，在可扩展性、数据类型的描述方面有着明显的优势，但相比于 JSON 这种轻量级的数据交换格式，XML 显得有些“笨重”。

对于大多数 Web 应用来说，根本不需要复杂的 XML 来传输数据，XML 宣称的扩展性在此没有多大优势。大多数 AJAX 应用直接使用 JSON 发送和接收数据，以构建动态页面的内容，非常灵活易用。如果使用 XML，则程序代码将会复杂不少。当然，在 Web Service 领域中，XML 目前仍有不可动摇的地位。

在 Java 的世界里有不少处理 Java 对象与 JSON、XML 相互转换的组件，如 JSON-lib、XMLBeans。但同时支持两种数据格式转换的轻量级组件并不多，XStream 组件就是少数中的佼佼者。

XStream 的 JettisonMappedXmlDriver 和 JsonHierarchicalStreamDriver 都可以很好地完成 Java 对象和 JSON 的相互转换工作。值得注意的是，在使用 JettisonMappedXmlDriver 时，必须事先将 jettison 依赖包添加到工程 pom 中，如下：

```
<dependency>
  <groupId>org.codehaus.jettison</groupId>
  <artifactId>jettison</artifactId>
  <version>1.3.2</version>
</dependency>
```

下面应用这两个驱动来将对象转换为 JSON，如代码清单 19-10 所示。

代码清单 19-10 XStreamJSONSample.java

```
package com.smart.oxm.xstream.json;
```

```
...
```

```

public class XStreamJSONSample {
    private XStream xstream;
    ...
    //①连续的没有分隔的JSON串
    public static void toJsonByJettisonMappedXmlDriver()throws Exception {
        User user = getUser();
        FileOutputStream outputStream = new
            FileOutputStream("out/JettisonMappedSample.json");
        OutputStreamWriter writer = new
            OutputStreamWriter(outputStream,Charset.forName("UTF-8"));
        xstream = new XStream(new JettisonMappedXmlDriver());
        xstream.setMode(XStream.NO_REFERENCES);
        xstream.alias("user", User.class);
        xstream.toXML(user,writer);
    }

    //② 格式化良好的JSON串
    public void toJsonByJsonHierarchicalStreamDriver()throws Exception {
        User user = getUser();
        FileOutputStream outputStream = new
            FileOutputStream("out/JsonByJsonHierarchicalSample.json");
        OutputStreamWriter writer = new
            OutputStreamWriter(outputStream, Charset.forName("UTF-8"));
        xstream = new XStream(new JsonHierarchicalStreamDriver());
        xstream.alias("user", User.class);
        xstream.toXML(user,writer);
    }
    ...
}

```

JSON 的转换和 XML 的转换用法一样,只需在创建 XStream 实例时,传递一个 XML 到 JSON 映射转换的驱动器如 JettisonMappedXmlDriver、JsonHierarchicalStreamDriver 即可。使用 JettisonMappedXmlDriver 驱动生成的是连续的没有分隔的 JSON 串,而使用 JsonHierarchical StreamDriver 驱动生成一个格式化后的 JSON 串。如果将 JSON 转换为对象,则只能使用 JettisonMappedXmlDriver 驱动。

在 IDE 工具中执行当前实例,在类路径下生成两个 JSON 文件。使用 JettisonMapped XmlDriver 驱动输出的 JSON 如下:

```

{"user":{"userId":1,"userName":"xstream","credits":0,"logs":[{"com.smart.oxm.domai
n.LoginLog":[{"loginLogId":0,"userId":0,"ip":"192.168.1.91","loginDate":"2016-03-1
3 15:56:30.517
UTC"},{"loginLogId":0,"userId":0,"ip":"192.168.1.92","loginDate":"2016-03-13
15:56:30.517 UTC"}]}]}}

```

而使用 JsonHierarch icalStreamDriver 驱动输出的 JSON 如下:

```

{"user": {
  "userId": 1,
  "userName": "xstream",
  "credits": 0,
  "logs": [
    {
      "loginLogId": 0,
      "userId": 0,

```

```

        "ip": "192.168.1.91",
        "loginDate": "2016-03-13 15:41:27 "
    },

    {
        "loginLogId": 0,
        "userId": 0,
        "ip": "192.168.1.92",
        "loginDate": "2016-03-13 15:41:27 "
    }
]
}}

```

19.3 其他常见的 O/X Mapping 开源项目

19.3.1 JAXB

JAXB(Java Architecture for XML Binding)是一个业界的标准,是一项可以根据 XML Schema 产生 Java 类的技术。JAXB 也提供了将 XML 文件反向生成 Java 对象树的方法,并能将 Java 对象树的内容重新写到 XML 文件。从另一方面来讲, JAXB 提供了快速而简便的方法将 XML 模式绑定到 Java 对象,从而使得开发者在 Java 应用程序中能方便地结合 XML 数据和处理函数。

利用 JAXB 技术,无须深入 XML 编程细节,就能在 Java 应用程序中灵活操作 XML 数据,而且可以充分利用 XML 的优势而不用依赖于复杂的 XML 处理模型,如 SAX 或 DOM 等。JAXB 隐藏了底层操作细节,并且取消了 SAX 和 DOM 中无用的关系,生成的 JAXB 类仅描述原始模型中定义的关系。其结果是整合高度可移植的 Java 代码和高度可移植的 XML 文档。通过这些代码可以创建灵活、轻便的应用程序和 Web 服务。

下面使用 JAXB 组件处理 Java 对象与 XML 之间的相互转换,采用 User 和 LoginLog 两个实体来创建 Schema 文档,通过 Schema 文档创建相应的 Java 源代码。在进行 Java 对象编组与反编组操作之前,需要编写 XML Schema 文档,并通过模式文档生成相应的 Java 代码。

1. 编写 XML Schema

XML Schema 文件是一个 XML 的约束文件,它定义了 XML 文件的结构和元素,以及对元素和结构的相关约束, JAXB 使用这个 Schema 文件生成相应的 Java 代码。User 模式结构图如图 19-1 所示,文档结构代码如代码清单 19-11 所示。

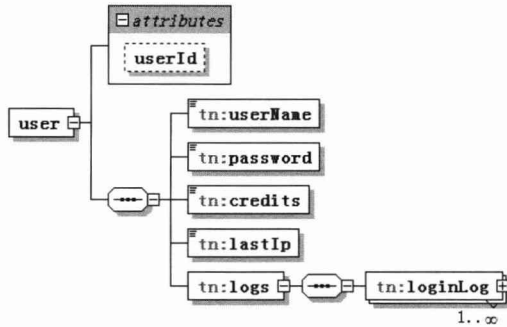


图 19-1 User 模式结构图

代码清单 19-11 user.xsd

```

<?XML version="1.0" encoding="UTF-8"?>
<xs:schema elementFormDefault="qualified" version="1.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:JAXB="http://Java.sun.com/xml/ns/JAXB"
  xmlns:tn="http://www.smart.oxm.com/domain/JAXB"
  targetNamespace="http://www.smart.oxm.com/domain/JAXB"
  JAXB:version="2.0" >
  <xs:annotation>
    <xs:appinfo>
      <JAXB:globalBindings>
        <JAXB:JavaType name="Java.util.Calendar" XMLType="xs:date"
          parseMethod="javax.xml.bind.DatatypeConverter.parseDate"
          printMethod="javax.xml.bind.DatatypeConverter.printDate" />
      </JAXB:globalBindings>
    </xs:appinfo>
  </xs:annotation>
  <xs:element name="user">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="userName" type="xs:string" />
        <xs:element name="password" type="xs:string" />
        <xs:element name="credits" type="xs:int" />
        <xs:element name="lastIp" type="xs:string" />
        <xs:element name="logs">
          <xs:complexType>
            <xs:sequence>
              <xs:element maxOccurs="unbounded" name="loginLog"
                type="tn:LoginLog" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="userId" type="xs:int" />
    </xs:complexType>
  </xs:element>
  <xs:complexType name="LoginLog">
    <xs:sequence>
      <xs:element name="userId" type="xs:string" />
      <xs:element name="ip" type="xs:string" />
    </xs:sequence>
  </xs:complexType>

```

```

        <xs:element name="loginDate" type="xs:date" />
    </xs:sequence>
    <xs:attribute name="loginLogId" type="xs:int" />
</xs:complexType>
</xs:schema>

```

JAXB 在生成代码时，如果元素类型是日期类型，也就是 `type="xs:date"` 时，默认情况下生成的 Java 对象属性类型为 `javax.xml.datatype.XMLGregorianCalendar`。这明显不是我们想要的类型，不过无须担心，JAXB 已经为我们提供了一个全局的类型转换机制，如实例在 Schema 文档中设置一个全局解析日期类型。

```

<xs:annotation>
  <xs:appinfo>
    <JAXB:globalBindings>
      <JAXB:javaType name="java.util.Calendar" xmlType="xs:date"
        parseMethod="javax.xml.bind.DatatypeConverter.parseDate"
        printMethod="javax.xml.bind.DatatypeConverter.printDate" />
    </JAXB:globalBindings>
  </xs:appinfo>
</xs:annotation>

```

此时 JAXB 生成的对应实例类属性类型就变成 `java.util.Calendar`。细心的读者可能会有疑问：如果想要的类型是 `java.util.Date`，那么该如何设置？由于 JAXB 提供的日期解析器只能转换成 `Calendar` 类型，所以我们需要自己编写一个 `Date` 类型的适配器来替换 `javax.xml.bind.DatatypeConverter.parseDate`。接下来编写一个 `Date` 类型的适配器，如代码清单 19-12 所示。

代码清单 19-12 DateAdapter.java

```

package com.smart.oxm.JAXB;

...
public class DateAdapter {
    public static Date parseDate(String s) {
        return DatatypeConverter.parseDate(s).getTime();
    }
    public static String printDate(Date dt) {
        Calendar cal = new GregorianCalendar();
        cal.setTime(dt);
        return DatatypeConverter.printDate(cal);
    }
}

```

2. 生成 Java 类

`xjc` 工具基于 XML Schema 文档定义来绑定一个模式到 Java 类，也就是根据 XML Schema 文档定义生成相应的 Java 实体类。针对当前 XML 的模式来进行绑定的命令是：

```
xjc user.xsd
```

`xjc` 命令行一些常用参数选项说明如下。

- ☐ `-nv`：对于输入的模式不执行严格的 XML 验证。
- ☐ `-b <file>`：指定外部的绑定文件。
- ☐ `-d <dir>`：指定生成文件的存放路径。
- ☐ `-p <pkg>`：指定目标包。

- -classpath <arg>: 指定 classpath。
- -XMLschema: 输入的模式是一个 W3C XML 模式 (默认)。

xjc 命令提供了包参数选项-p, 用来设置生成 Java 实体类的包结构。如果该参数没有输入, 则 xjc 将利用 Schema 文档中定义的 namespace 自动产生包名 (package name)。如实例中设置的 targetNamespace="http://www.smart.oxm.com/domain/jaxb", 生成的包结构为 com.smart.oxm.domain.jaxb。

使用参数-p 指定包名, 生成的源代码结构如下:

```
D:\masterspring\code\chapter19\bin>
                                xjc -p com.smart.oxm.domain.jaxb user.xsd -d src
parsing a schema...
compiling a schema...
com\smart\oxm\domain\ jaxb\Adapter1. java
com\smart\oxm\domain\ jaxb\LoginLog. java
com\smart\oxm\domain\ jaxb\ObjectFactory. java
com\smart\oxm\domain\ jaxb\User. java
com\smart\oxm\domain\ jaxb\package-info. java
```

使用 Schema 中定义的 namespace 作为包名, 生成的源代码结构如下:

```
D:\masterspring\code\chapter19\bin>xjc user.xsd -d src
parsing a schema...
compiling a schema...
com\smart\oxm\domain\ jaxb\LoginLog. java
com\smart\oxm\domain\ jaxb\ObjectFactory. java
com\smart\oxm\domain\ jaxb\User. java
com\smart\oxm\domain\ jaxb\package-info. java
org\w3\_2001\XMLschema\Adapter1. java
```

在使用 JAXB 进行编组与反编组操作之前, 需要创建一个 JAXBContext 对象, 即 JAXB 的上下文, 如实例中的 JAXBContext context = JAXBContext.newInstance(User.class), 通过其 createMarshaller() 方法创建一个编组器, 编组时调用 Marshaller#marshal() 方法完成转换操作。如果希望生成的 XML 文件格式良好, 则需要调用 Marshaller#setProperty() 方法设置输出 XML 文件的格式, 反编组时调用其 createUnmarshaller() 方法创建一个反编组器, 然后调用 Unmarshaller#unmarshal() 方法即可完成 XML 到 Java 对象的转换, 如代码清单 19-13 所示。

代码清单 19-13 JAXBSample.xml

```
package com.smart.oxm.JAXB;
...
public class JAXBSample {

    // Java转换为XML对象
    public static void objectToXML() throws Exception {
        User user = getUser();
        JAXBContext context = JAXBContext.newInstance(User.class);
        Marshaller m = context.createMarshaller();
        m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
        FileWriter writer = new FileWriter("out/JaxbSample.xml");
        m.marshal(user, writer);
    }
}
```



```
// XML转换为Java对象
public static void XMLToObject() throws Exception {
    JAXBContext context = JAXBContext.newInstance(User.class);
    FileReader reader = new FileReader("out/JaxbSample.xml");
    Unmarshaller um = context.createUnmarshaller();
    User u = (User) um.unmarshal(reader);
    ...
}
...
```



实战经验

目前 XML 数据绑定组件基本都支持两种方式：一是映射绑定；二是代码生成。如果采用代码生成，即根据 XML 样本数据文法生成 Java 语言代码，则必须编写 XML Schema 文件。对于熟悉 XML 的开发人员来说，可以自己编写这个 Schema 文件；而对于不熟悉 XML 的开发人员来说，则可以通过一些工具来完成，比较有名的如 Trang、XMLSpy、Stylus Studio 都可以通过 XML 样本数据来生成相应的 Schema 文件。

19.3.2 Castor

Castor 是 ExoLab Group 下一个开放源代码的项目，其主要目标是在 XML 数据、Java 对象和数据库关系数据之间提供一种直接的映射，使得这 3 种对象数据可以相互转换。Castor 项目主要包括 XML 与 Java 对象的映射（Castor XML）、Java 对象与关系数据库表映射（Castor JDO）两个关键功能。在 Castor 中，通过定义数据映射（Mapping）文件，在 XML 文档元素（节点、属性、文本等）、Java 类（类、属性等）和数据库表（关联表、表、字段）之间建立一一映射。通过 XML Schema 定义可以自动生成实体 Java 类定义，通过实体 Java 类定义可以自动生成 XML 元素与 Java 类之间的映射文件。

1. Castor 主要特性

- ☐ 通过定义映射文件在 XML 文档与 Java 对象实体间建立映射。
- ☐ Castor XML 在 Java 对象模型与 XML InfoSet 之间建立相互转换和数据绑定功能。
- ☐ 提供从 XML 文档定义自动生成 Java 类定义文件的功能（Code Generator）。
- ☐ 通过自省方式在 XML 文档与 Java 对象间建立映射。
- ☐ 提供基于 Java 类定义生成映射文件的功能。
- ☐ 提供基于 XML 输入文档生成 Schema 定义文件的功能。
- ☐ Castor JDO 为 Java 对象与数据库关系数据之间提供一种序列化与反序列化的框架。
- ☐ 基于 XML 映射文件的方式定义 3 种不同数据对象模型之间的映射。
- ☐ 提供内存缓存和提交写入方式，减少数据库 JDBC 操作。

- 支持两阶段事务、对象操作回滚和数据库锁定侦测。
- 支持 OQL 与标准 SQL 之间的映射，Castor JDO 对象查询采用 OQL。

Castor 组件提供了几种进行 Java 对象与 XML 相互转换的方式：一是采用 XML Schema 生成代码方式；二是采用自省方式；三是采用映射文件方式。下面通过这几种不同方式体验 Castor 组件的灵活与强大。

在开发实例之前，我们需要在 pom.xml 中引入 Castor 的依赖包，包含 castor-xml 和 castor-core。本实例选用 Castor 1.3 版本，代码如下：

```
<dependency>
  <groupId>org.codehaus.castor</groupId>
  <artifactId>castor-xml</artifactId>
  <version>1.3</version>
</dependency>
<dependency>
  <groupId>org.codehaus.castor</groupId>
  <artifactId>castor-core</artifactId>
  <version>1.3</version>
</dependency>
```

2. XML Schema 生成代码

创建 XML Schema 文件，Castor 使用这个 Schema 文件生成一系列相关的 Java 实体类及相应的类描述符。本例 User 模式结构图如图 19-2 所示。

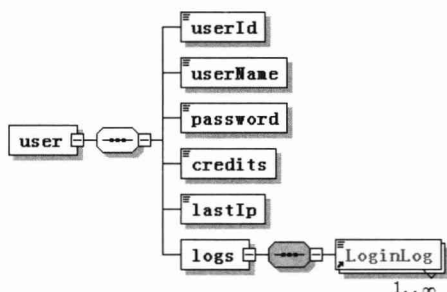


图 19-2 User 模式结构图

为了应用 Castor 生成 Java 实体类，不仅需要把 castor-1.3.jar、castor-1.3-codegen.jar 和 castor-1.3-XML-schema.jar 包放到类路径下，还需要编写一个批处理文件或者 Ant 构建文件。首先编写一个批处理文件 user.bat，如代码清单 19-14 所示。

代码清单 19-14 user.bat

```
@echo off
REM Change the following line to set your JDK path
set JAVA_HOME=%JAVA_HOME%
set JAVA=%JAVA_HOME%\bin\Java
set JAVAC=%JAVA_HOME%\bin\Javac
@echo Create the classpath
set CP=.
for %%i in (lib\*.jar) do call cp.bat %%i
set CP=%CP%;%JDK_BIN%\lib\tools.jar
```

```

@echo.
@echo Using classpath: %CP%
@echo Castor Test Cases
@echo.
@echo Generating classes...
@rem Java 2 style collection types
@rem %JAVA% org.exolab.castor.builder.SourceGeneratorMain -i invoice.xsd -f -types j2
-binding-file bindingInvoice.xml
@rem Java 1.1 collection types
%JAVA% -cp %CP% org.exolab.castor.builder.SourceGeneratorMain -i user.xsd -types j2
-package com.smart.oxm.domain.castor
@echo.
@pause

```

①生成代码入口

其中, `org.exolab.castor.builder.SourceGeneratorMain` 生成代码入口, 参数 `-i` 指定 Schema 文件, 参数 `-package` 指定生成实例类所在的包名。

运行 `user.bat` 文件, 生成一系列 Java 实体类及实体类相应的类描述符。如本实例将生成 3 个实例类 (`User.java`、`LoginLog.java`、`Logs.java`) 及 3 个实体类对应的类描述符 (`UserDescriptor.java`、`LoginLogDescriptor.java`、`LogsDescriptor.java`)。

与 XMLBeans 一样, Castor 也可以通过生成 Java 代码的方式完成对象与 XML 相互转换, 但 Castor 生成实体类及类描述符之后, 转换工作比 XMLBeans 更加简洁, 把一个实体对象转换为 XML, 只要创建当前对象的实例, 调用实体对象 `marshal(Writer out)` 方法, 就完成了对象编组工作, 也就是把当前对象转换成一个 XML 文件。反编组也一样简洁, 只需调用实例类反编组静态方法 `unmarshal(Reader reader)` 就可以完成转换工作, 如代码清单 19-15 所示。

代码清单 19-15 `CastorGeneratorSampe.java`

```

package com.smart.oxm.castor;
...
public class CastorGeneratorSampe {

    // Java对象转换为XML
    public static void objectToXML() throws Exception {
        User user = getUser();
        FileWriter writer = new FileWriter("out/CastorSample.xml");
        user.marshal(writer);
    }

    // XML转换为Java对象
    public static void XMLToObject() throws Exception {
        FileReader reader = new FileReader("out/CastorSample.xml");
        User u = User.unmarshal(reader);
        ...
    }
    ...
}

```

3. 自省方式的编组与反编组

Castor 提供自省的方式在 Java 对象与 XML 文档之间实现转换, 也就是说, 我们不

需要编写 Java 对象与 XML 文档之间的映射文件，Castor 在编组的时候会根据实体对象属性生成相应的 XML 元素。

Castor 提供了 Marshaller 和 Unmarshaller 两个类处理编组与反编组操作，在编组的时候，调用 Marshaller#marshal()方法完成对象转换为 XML 的工作；在反编组的时候，调用 Unmarshaller 静态反编组方法 Unmarshaller#unmarshal()即可。如实例中的 Unmarshaller#unmarshal()方法，通过 FileReader 把 XML 文件内容读入并映射到 User 对象，如代码清单 19-16 所示。

代码清单 19-16 CastorSample.java

```
package com.smart.oxm.castor;
...
public class CastorSample {

    // Java 对象转换为 XML
    public static void objectToXML()throws Exception {
        User user = getUser();
        FileWriter writer = new FileWriter("out/CastorSample.xml");
        Marshaller marshaller = new Marshaller(writer);
        marshaller.setEncoding("GBK");
        marshaller.marshal(user);
    }

    // XML 转换为 Java 对象
    public static void XMLToObject()throws Exception {
        FileReader reader = new FileReader("out/CastorSample.xml");
        User u = (User) Unmarshaller.unmarshal(User.class, reader);
        ...
    }
    ...
}
```

4. 映射文件方式的编组与解编组

Castor 虽然提供自省的方式在 Java 对象与 XML 文档之间进行转换，但当 Java 对象属性与 XML 文档元素名称不一致时，就无法采用自省方式，这时就需要编写它们之间的映射文件，告诉 Castor 它们之间的对应关系。本实例仍然使用 User.java 及 LoginLog.java 作为实例的操作对象。

首先需要编写一个映射文件 mapping.xml，如代码清单 19-17 所示。

代码清单 19-17 mapping.xml

```
<?XML version="1.0"?>
<!DOCTYPE mapping PUBLIC "-//EXOLAB/Castor Mapping DTD Version 1.0//EN"
    "http://castor.org/mapping.dtd">
<mapping>
    <class name="com.smart.oxm.domain.User">
        <map-to xml="user" /><!--①设置XML文件的根元素 -->
        <field name="userId" type="int">
            <bind- xml name="id" node="attribute" />
        </field>
    ...
</mapping>
```

```

        <field name="userName" type="Java.lang.String">
            <bind-xml name="userName" />
        </field>
        <field name="logs" type="com.smart.oxm.domain.LoginLog" collection=
"arraylist">
            <bind-xml name="log" />
        </field>
    </class>

```

<!-- ②通过在该元素中加入auto-complete属性并把值设为true。您可以告诉Castor对于该类的任何属性，只要没有在这个元素中专门列出，就使用默认映射 -->

```

    <class name="com.smart.oxm.domain.LoginLog" identity="userId"
auto-complete="true">
        <map-to xml="logs" />
        <field name="loginLogId" type="int">
            <bind-xml name="id" node="attribute" />
        </field>
        <field name="userId" type="int">
            <bind-xml name="userId" />
        </field>
    </class>
</mapping>

```

与 Castor 自省方式编组及反编组操作方式一样，在编组的时候，也调用 `Marshaller#marshal()` 方法完成 Java 对象转换为 XML 的工作，不同的是在编组之前需要先把映射文件加载到 `Mapping` 中，然后调用 `Marshaller#setMapping()` 方法设置到编组器中。在反编组的时候，与自省方式不同，需要先创建 `Unmarshaller` 实例，并把映射文件加载到其实例中，然后调用 `Unmarshaller` 对象方法 `unmarshal(FileReader reader)` 进行反编组操作，而不是调用静态方法。

Castor 使用 `Marshaller` 和 `Unmarshaller` 类的静态方法以自省的方式实现 XML 文档与 Java 对象之间的编组和反编组，以及使用 `Mapping` 映射文件通过调用 `Marshaller` 和 `Unmarshaller` 类的实例方法实现 XML 文档与 Java 对象之间的编组和反编组两种方式。前一种方式不需要构造 `Marshaller` 和 `Unmarshaller` 类的实例，后一种方式需要构造 `Marshaller` 和 `Unmarshaller` 类的实例，调用非静态方法完成上述功能。在调用 `Marshaller` 和 `Unmarshaller` 类的实例方法时，必须提供 XML 文档结构和 Java 类模型间的映射定义文件。

Castor 通过类描述符和字段描述符几乎可以编组/解编组所有的 Java 对象。当某个 Java 对象类描述符不存在时，编组框架使用反射 (Reflection) 方式获取对象的结构信息，并在内存中为此对象构造类描述符和字段描述符。当实体类的描述符存在时，Castor 使用这些描述符提供的 Java 对象模型信息处理编组和解编组。在编组与解编组过程中，Castor 要求实体类定义必须提供 `public` 类型的默认构造函数（不含参数）声明以及必要的 `getter` 和 `setter` 方法。

19.3.3 JiBX

JiBX 是一款非常优秀的 XML 数据绑定框架。它提供灵活的绑定映射文件，以实现数据对象与 XML 文件之间的转换，并不需要修改既有的 Java 类。不管是在灵活性还是在效率上，JiBX 是目前很多开源项目都无法比拟的，如具有转换效率高、配置绑定文件简单、不需要操作 XPath 文件、不需要写属性的 get/set 方法、对象属性名与 XML 文件元素名不需要相同等优点。

JiBX 使用 Java 字节码的增强技术，即 BCEL (Byte Code Engineering Library)，在编译期，根据绑定配置文件，生成对应对象实例的方法和添加被绑定标记的属性；在运行期，它已经与绑定配置文件无关了，不需要再作任何配置，因为其已经嵌入到相应的 Java 类中。2005 年 12 月发布了 JiBX 1.0 版本，目前的最新版本是 2015 年 1 月发布的 JiBX 1.2.6。

JiBX 具有如下主要特性：

- JiBX 使用 BCEL 技术在编译时对相应的类字节码进行增强。
- JiBX 使用 XPP (XML Pull Parse) 技术，它提供了一个更自然的接口来处理文档中的元素序列，能够使用相对简单的代码来处理数据。
- JiBX 可以自由灵活地定义 XML 中的每个字段和 Java 类的对应关系。

在开发实例之前，需要在 pom.xml 中引入 JiBX 的依赖包，本实例采用 JiBX 1.1.5 版本，代码如下：

```
<dependency>
  <groupId>net.sourceforge.jibx</groupId>
  <artifactId>com.springsource.org.jibx.runtime</artifactId>
  <version>1.1.5</version>
</dependency>
```

在使用 JiBX 进行编组及反编组操作时，需要编写 Java 对象与 XML 之间的绑定映射文件。在运行程序之前，需要先配置绑定文件并进行绑定，在绑定过程中将会动态修改程序中相应的.class 文件，主要是生成对应对象实例的方法和添加被绑定标记的属性 JiBX_bindingList 等。它有如此大的魔法，得益于 BCEL 技术。BCEL 是 Apache Software Foundation 的 Jakarta 项目的一部分，BCEL 在单独的 JVM 指令级别上进行操作，可以深入 Java 类的字节码，用它转换现有的类或者构建新的类，如代码清单 19-18 所示。

代码清单 19-18 binding.xml

```
<binding>
  <mapping name="user" class="com.smart.oxm.domain.User">
    <value name="userName" field="userName"/>
    <!--name 设置XML元素名称，field设置Object对象属性名称 -->
    <value name="password" field="password" usage="optional" />
    <value name="credits" field="credits" usage="optional" />
    <collection field="logs" factory="com.smart.oxm.jibx.JiBXInterfaceFactory.
getArrayListInstance">
      <structure name="log" type="com.smart.oxm.domain.LoginLog">
        <value name="ip" field="ip"/>
      </structure>
    </collection>
  </mapping>
</binding>
```



```

        <value name="loginDate" field="loginDate"/>
    </structure>
</collection>
</mapping>
</binding>

```

在 JiBX 绑定 Java 对象时，如果使用了基本类型，又设置字段为可选值，如 `<value name="credits" field="credits" usage="optional" />` 中的 `usage="optional"`，则 JiBX 根据 `credits` 类型的默认值来决定是否编组当前元素。如果编组的时候 `credits` 的值刚好是 0，则 `credits` 无法显示在 XML 文件中，也就是说基本类型为可选项时，是无法输出默认值的。为了避免出现这个问题，我们在使用基本类型时，需要把字段设置为不可选。

使用接口处理从 XML 到 Java 对象的转换时，先要创建该类的实例。如果转换的是一个实体类，则创建实例不会有什么问题；但如果我们要使用接口编程，则转换的对象必须是一个接口，否则会出现错误，因为 JiBX 并不清楚你需要创建这个接口的哪个实例，所以我们需要在绑定文件中指明该接口的创建工厂方法。例如，本例 `User` 中声明一个 `List` 类型的属性 `logs`，想要让它指向一个 `ArrayList` 的实例，则需要写一个返回 `ArrayList` 实例的工厂类 `JiBXInterfaceFactory`，如代码清单 19-19 所示。

代码清单 19-19 JiBXInterfaceFactory.java

```

package com.smart.oxm.jibx;

...

public class JiBXInterfaceFactory {
    // 获取List实现类实例
    public static List getArrayListInstance(){
        return new ArrayList();
    }
}

```

编写完绑定映射文件后，还要编写一个 Ant 构建文件，来完成 Java 类编译及字节码增强的工作，以增强实体对象编组及反编组的能力，如代码清单 19-20 所示。

代码清单 19-20 build.xml

```

<?XML version="1.0" encoding="gb2312"?>
<project name="oxm" default="run">
    <property name="project" value="D:/masterspring/code/chapter19"/>
    <property name="src" value="D:/masterspring/code /chapter19/src/main/java "/>
    <property name="lib" value="D:/masterspring/code/libs"/>
    <property name="classes"
value="D:/masterspring/code/chapter19/target/classes"/>
    <target name="clean">
        <delete includeEmptyDirs="true">
            <fileset dir="${classes}" includes="**/*" defaultexcludes="no"/>
        </delete>
    </target>
    ...
    <taskdef name="bind" classname="org.jibx.binding.ant.CompileTask"
classpath="${lib}/jibx-bind.jar"/>
    <target name="jibx-binding" depends="compile">
        <bind verbose="false" load="true"

```



```

        binding="${classes}/com/smart/oxm/jibx/binding.xml">
            <classpath>
                <path location="${classes}">
                </path>
                <pathelement location="${lib}/jibx-bind.jar"/>
            </classpath>
        </bind>
    </target>
    <target name="run" depends="jibx-binding" description="run">
        <Java classname="com.smart.oxm.jibx.JibxSample">
            <classpath>
                <path location="${classes}">
                </path>
                ...
            </classpath>
        </Java>
    </target>
</project>

```

用 Ant 运行当前构建文件，在 User.class 同目录下，我们可以看到 JiBX 自动编译生成 JiBX_MungeAdapter.class 等 3 个类，用反编译工具查看 User.class 就会发现，User 已经自动实现 IMarshalleable 和 IUnmarshalleable（编组和反编组）两个接口。

使用 JiBX 进行编组及反编组操作时，需要通过 BindingDirectory 获取一个相应实体类的绑定工厂实例。如实例中的 IBindingFactory bfact = BindingDirectory.getFactory(User.class)，在编组的时候，先调用其 createMarshallingContext() 方法创建一个编组上下文，然后调用编组上下文实例的 marshalDocument() 方法进行编组；在反编组的时候，调用其 createUnmarshallingContext() 方法创建一个反编组上下文，并通过实例的 unmarshalDocument() 方法来进行反编组，如代码清单 19-21 所示。

代码清单 19-21 JibxSample.xml

```

package com.smart.oxm.jibx;
...
public class JibxSample {
    // Java对象转换为XML
    public static void objectToXML() throws Exception {
        User user = getUser();
        IBindingFactory bfact = BindingDirectory.getFactory(User.class);
        IMarshallingContext ctx = bfact.createMarshallingContext();
        FileOutputStream os = new FileOutputStream("out/JibxSample.xml");
        ctx.marshalDocument(user, "UTF-8", null, os);
    }

    // XML转换为Java对象
    public static void XMLToObject() throws Exception {
        IBindingFactory bfact = BindingDirectory.getFactory(User.class);
        IUnmarshallingContext uctx = bfact.createUnmarshallingContext();
        File dataFile = new File("out/JibxSample.xml");
        InputStream in = new FileInputStream(dataFile);
        User user = (User) uctx.unmarshalDocument(in, null);
    }
}

```

```
    }  
    ...  
}
```

JiBX 是一款高性能的数据绑定框架。如果 XML 文件格式比较固定，同时数据转换比较频繁，则可以考虑使用它来助你一臂之力。需要注意的是，JiBX 对中文的支持不是很好，目前 JiBX 只支持 Java 标准的字符集。如果使用的是 UTF-8 编码来处理中文，则没有任何问题；但如果使用 GB2312 或 GBK 编码，则需要自己实现 GB2312 和 GBK 的 Escaper 类，重写 writeAttribute(String, Writer)、writeCData(String, Writer)和 writeContent(String, Writer)方法。

19.3.4 总结比较

上文详细介绍了目前一些主流 O/X Mapping 组件的特点及使用方法，不同 O/X Mapping 组件都有各自的应用场景。如在构建基于 Spring 轻量级 RESTful 服务的过程中，可以使用 XStream 组件处理服务请求响应的报文（XML 或 JSON 数据格式）；在构建基于 Spring 契约优先的 Web Services 时，可以使用 XMLBeans、JAXB 等组件来处理服务请求响应的报文。表 19-2 列举了目前各 O/X Mapping 组件的优势、劣势及其应用场景，读者可以根据实际需求选择合适的 O/X Mapping 组件。

表 19-2 各O/X Mapping组件对比

XML 编组框架	优 势	劣 势	应用场景
XStream	简单易用； 同时支持 XML 和 JSON 数据格式	非标准； 默认 Driver 不支持 CDATA； 映射集合时对 XML 的格式要求很严格； 修改映射就需要修改代码	应用对象各种序列化及反序列化操作； 基于 Spring 轻量级 RESTful 服务中的对象映射处理
Castor	提供丰富的工具，如根据 XML 文档生成 Java 代码、根据 Java 类定义生成映射文件等； 支持自省方式在 XML 与 Java 对象间建立映射； 支持绑定任意类； 完全支持 JDO	不支持使用注解	应用各种数据绑定需求； 应用 JPA 相关实现； 使用 XML 交换数据的应用程序
JiBX	支持 CDATA； 对集合的兼容性很好； 允许定义空标签； 在配置文件中映射，而不是在代码中映射； 基于字节码增强技术，运行速度非常快	需要对 POJO 进行增强，每次重新编译后都必须重新进行增强。不过可以预编译，也可以运行时编译； 配置文件相对复杂些； 不支持灵活的验证机制	广泛应用各种数据绑定需求； Web Services 常用数据绑定框架

续表

XML 编组框架	优 势	劣 势	应用场景
JAXB	业界的标准	XML 模式（XML Schema）发生变化，就必须重新编译；JAXB 重新编译，可能会引起由 XML 模式生成的 JAXB 代码产生变化	广泛应用各种数据绑定需求； Web Services 常用数据绑定框架； 使用 XML 交换数据的应用程序

19.4 与 Spring OXM 整合

19.4.1 Spring OXM 概述

Spring OXM 是 Spring 3.0 的一个新特性，为主流 O/X Mapping 组件提供了统一层抽象和封装，而在 Spring 4.0 中 OXM 没有增加更多新的特性。O/X 映射器这个概念并不新鲜，O 代表 Object，X 代表 XML，目的是在 Java 对象和 XML 之间进行转换操作。Spring OXM 不仅屏蔽了各 O/X Mapping 组件实现的差异性，而且还提供了统一、高效的编程模型。Spring OXM 的结构框架如图 19-3 所示。

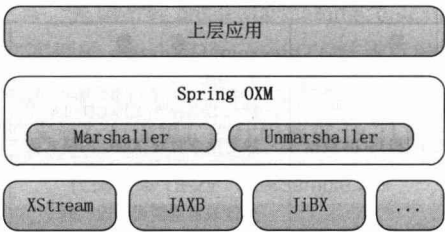


图 19-3 Spring OXM 的结构框架

Marshaller 和 Unmarshaller 是 Spring OXM 的两个核心接口。实现 Marshaller 接口可以实现从 Java 对象到 XML 的映射转换，实现 Unmarshaller 接口可实现从 XML 到 Java 对象的映射转换。下面从 Spring OXM 源代码中摘录部分 Marshaller 接口的定义，如代码清单 19-22 所示。

代码清单 19-22 Marshaller.java

```
package org.springframework.oxm;
import java.io.IOException;
import javax.xml.transform.Result;
public interface Marshaller {

    //① 判断支持的编组类类型
    boolean supports(Class<?> clazz);

    //② 对传入的对象进行编组操作
```

```
void marshal(Object graph, Result result) throws IOException, XmlMappingException;
}
```

Marshaller 接口中提供了两个方法，其中 `Marshaller#supports()` 方法用于判断支持的编组类类型；`Marshaller#marshal()` 方法用于对对象进行编组操作，其中 `graph` 参数为目标转换对象，`result` 参数为 JDK 提供的 XML 转换接口，实现此接口的对象包含构建转换结果树所需的信息。Spring OXM 支持的 `Result` 转换实现类有 `DOMResult`、`SAXResult`、`StreamResult`、`StaxResult`，其中 `DOMResult`、`SAXResult`、`StreamResult` 为 JDK 提供的实现类，`StaxResult` 为 Spring OXM 提供的扩展类。

下面从 Spring OXM 源代码中摘录部分 `Unmarshaller` 接口的定义，如代码清单 19-23 所示。

代码清单 19-23 Unmarshaller.java

```
package org.springframework.oxm;
import java.io.IOException;
import javax.xml.transform.Source;
public interface Unmarshaller{

    //① 判断支持的反编组类类型
    boolean supports(Class<?> clazz);

    //② 对输入的源对象进行反编组操作
    Object unmarshal(Source source) throws IOException, XmlMappingException;
}
```

`Unmarshaller` 接口中提供了两个方法，其中 `Marshaller#supports()` 方法用于判断支持的反编组类类型；`Marshaller#unmarshal()` 方法用于将输入源对象进行反编组操作，其中 `source` 参数为目标输入源对象，`Source` 参数为 JDK 提供的输入源接口，实现此接口的对象包含充当源输入（XML 源或转换指令）所需的信息。Spring OXM 支持的 `Source` 输入源实现类有 `DOMSource`、`SAXSource`、`StreamSource`、`StaxSource`，其中 `DOMSource`、`SAXSource`、`StreamSource` 为 JDK 提供的实现类，`StaxSource` 为 Spring OXM 提供的扩展类。

Spring OXM 统一封装底层的 O/X Mapping 组件异常，将各 O/X Mapping 组件原始异常对象包装到 Spring 自身专为 OXM 建立的运行时异常 `XmlMappingException` 中，并通过 `MarshallingFailureException` 和 `UnmarshallingFailureException` 处理编组和反编组操作之间的区别。Spring OXM 异常层次结构如图 19-4 所示。

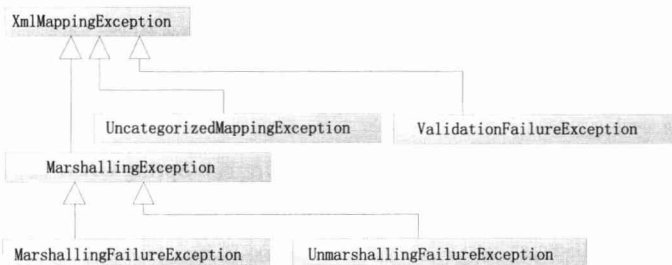


图 19-4 Spring OXM 异常层次结构

19.4.2 整合 OXM 实现者

Spring OXM 默认提供目前几个主流 O/X Mapping 组件的实现，包括 XStream、XMLBeans、Castor、JiBX、JAXB。这些 O/X Mapping 组件统一实现 Spring OXM 的两个核心接口 Marshaller 和 Unmarshaller。具体的实现体系类图如图 19-5 所示。

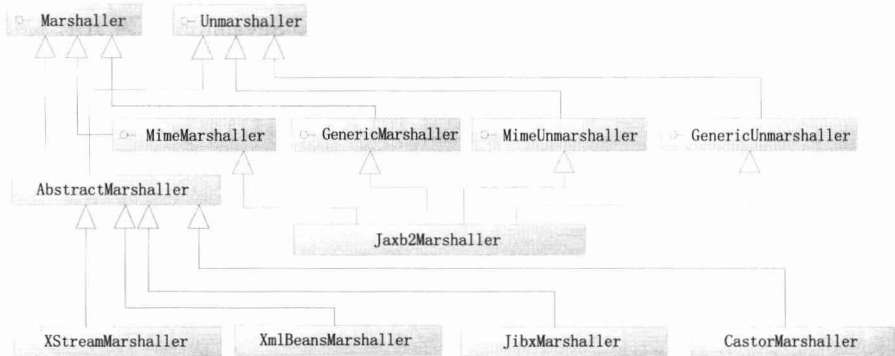


图 19-5 Spring OXM 核心接口实现

要使用 Spring OXM 的 O/X 功能，首先需要有一个在 Java 对象和 XML 之间来回转换的组件，并将上文提到的 XStream、JAXB 等相应的类库添加到工程中。各 O/X Mapping 组件对应的实现类如表 19-3 所示。

表 19-3 各O/X Mapping组件包装器

O/X Mapping 组件	Spring OXM 实现类
XStream	org.springframework.oxm.xstream.XStreamMarshaller
Castor	org.springframework.oxm.castor.CastorMarshaller
JiBX	org.springframework.oxm.jibx.JibxMarshaller
JAXB 2.0	org.springframework.oxm.jaxb.JAXB2Marshaller

19.4.3 如何在 Spring 中进行配置

到目前为止，我们已经介绍了各 O/X Mapping 组件的使用方法以及 Spring OXM 总体框架，接下来介绍如何在 Spring 中整合这些 O/X Mapping 组件。不同的 O/X Mapping 组件在 Spring 中的配置略有不同，下面分别对各主流 O/X Mapping 组件的配置进行讲解。

1. OXM 命名空间

为了简化 Spring OXM 各 O/X Mapping 组件的配置，Spring OXM 提供了一个 OXM 命名空间，目前支持 OXM 命名空间的 O/X Mapping 组件有 3 个，分别是 Jaxb2Marshaller、JibxMarshaller 和 XmlBeansMarshaller。要启用 OXM 命名空间，首先需要在 Spring 配置文件中引用 OXM 模式定义文件，如代码清单 19-24 中的粗体所示。

代码清单 19-24 OXM命名空间

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:oxm="http://www.springframework.org/schema/oxm"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/oxm
http://www.springframework.org/schema/oxm/spring-oxm.xsd">
...
</beans>
```

2. XStreamMarshaller 配置

要启用 Spring OXM 的 XStreamMarshaller 功能，首先需要将 XStream O/X Mapping 组件相应的类库添加到工程 pom.xml 的依赖中，本章中使用 XStream 1.4.9 版本。

```
<dependency>
<groupId>com.thoughtworks.xstream</groupId>
<artifactId>xstream</artifactId>
<version>1.4.9</version>
</dependency>
```

在默认情况下，不需要对 XStreamMarshaller 作任何配置，直接在 Spring 应用程序上下文中配置即可。如果需要自定义的 XML 格式，则可以设置类别名映射及属性别名映射，如代码清单 19-25 所示。

代码清单 19-25 XStreamMarshaller配置实例

```
<bean id="xstreamMarshaller" class="org.springframework.oxm.xstream.XStreamMarshaller"
p:autodetectAnnotations="true">
<!-- 设置类别别名 -->
<property name="aliases">
<map>
<!-- User这个类的别名就变成了user -->
<entry key="user" value="com.smart.oxm.domain.User" />
<entry key="LoginLog" value="com.smart.oxm.domain.LoginLog" />
</map>
</property>
<!-- 设置类属性别名 -->
<property name="fieldAliases">
<map>
<!-- User中的lastVisit属性 -->
<entry key="com.smart.oxm.domain.User.lastVisit" value="lastVisitDate" />
</map>
</property>
</bean>
```

在上面的配置实例中，通过 aliases 属性设置类名别名，通过 fieldAliases 属性设置类属性别名，通过 autodetectAnnotations 属性设置自动加载 XStream 注解 Bean。

3. Jaxb2Marshaller 配置

要启用 Spring OXM 的 Jaxb2Marshaller 功能，首先需要将 Jaxb O/X Mapping 组件相应的类库增加到 pom.xml 的依赖中，如本章中使用 jaxb-api-2.1.4 和 jaxb-impl-2.1.8。

Jaxb2Marshaller 可以使用与 Jaxb1Marshaller 相同配置属性的 contextPath 来指定编组对象。Jaxb2Marshaller 提供了一个 classesToBeBound 属性, 用来设置编组对象数组。Schema 属性指定一个或多个模式资源, 如代码清单 19-26 所示。

代码清单 19-26 Jaxb2Marshaller配置实例

```
<!--① 传统的配置方法 -->
<bean id="jaxb2Marshaller" class="org.springframework.xml.jaxb.Jaxb2Marshaller">
    <!--<property name="contextPath"
value="com.smart.oxm.domain.jaxb"></property>-->
    <property name="classesToBeBound">
        <array>
            <value>com.smart.oxm.domain.jaxb.User</value>
            <value>com.smart.oxm.domain.jaxb.LoginLog</value>
        </array>
    </property>
    <property name="schema" value="classpath:com/smart/oxm/jaxb/user.xsd" />
</bean>
<!--② 基于OXM命名空间的配置方法 -->
<oxm:jaxb2-marshaller id="jaxb2Marshaller2">
    <oxm:class-to-be-bound name="com.smart.oxm.domain.jaxb.User" />
    <oxm:class-to-be-bound name="com.smart.oxm.domain.jaxb.LoginLog" />
</oxm:jaxb2-marshaller>
```

Jaxb1Marshaller 在 Spring OXM 3.0 之后不被支持, 如果没有特殊需求, 则推荐使用 Jaxb2Marshaller 作为 O/X Mapping 编组与反编组包装器。

4. CastorMarshaller 配置

要启用 Spring OXM 的 CastorMarshaller 功能, 首先需要将 Castor O/X Mapping 组件相应的依赖包添加到工程的 pom.xml 中, 如本章中使用 castor-1.3-core、castor-1.3-xml 和 castor-1.3-commons。不需要对 CastorMarshaller 作任何配置, 直接在 Spring 应用程序上下文中配置即可。如果需要自定义的映射配置文件, 则需要配置 mappingLocation 属性, 如代码清单 19-27 所示。

代码清单 19-27 CastorMarshaller配置实例

```
<bean id="castorMarshaller"
    class="org.springframework.xml.castor.CastorMarshaller"
    p:mappingLocation="classpath:/config/mapping.xml" />
```

虽然 Castor 支持自省方式在 XML 与 Java 对象间建立映射, 但有时可能需要自定义设置它们之间的映射关系, 这时就需要使用 Castor 的映射文件。可以通过 mappingLocation 属性来指定自定义的配置文件。

5. JibxMarshaller 配置

要启用 Spring OXM 的 JibxMarshaller 功能, 首先需要将 JiBX O/X Mapping 组件相应的依赖包添加到工程 pom.xml 的依赖中, 如本章中使用 jibx-run、jibx-bind 和 jibx-extras。配置 JibxMarshaller 很简单, 只需在 Spring 应用程序上下文中配置一个 JibxMarshaller, 并通过 targetClass 属性配置目标编组类即可。如果需要设置绑定名称, 则可以通过 bindingName 属性指定一个绑定配置文件, 如代码清单 19-28 所示。

代码清单 19-28 JibxMarshaller配置实例

```

<!--① 传统的配置方法 -->
<bean id="jibxMarshaller" class="org.springframework.oxm.jibx.JibxMarshaller">
    <property name="targetClass" value="com.smart.oxm.domain.User" />
    <!--<property name="bindingName"
        value="classpath:com/smart/oxm/jibx/binding.xml" />-->
</bean>
<!--② 基于OXM命名空间的配置方法 -->
<oxm:jibx-marshaller id="jibxMarshaller2" target-class="com.smart.oxm.domain.User" />

```

一个 JibxMarshaller 只能配置一个目标编组类。如果需要配置多个编组类，则只能为不同的编组类各配置一个 JibxMarshaller。

19.4.4 Spring OXM 简单实例

到目前为止，我们已经介绍了各 O/X Mapping 组件的使用方法及 Spring OXM 框架，接下来通过一个简单的实例来讲解如何应用 Spring OXM 并整合第三方 O/X Mapping 组件。实例使用 Castor 这个 O/X Mapping 组件处理 Java 对象与 XML 之间的相互转换映射。首先在 Spring 上下文（applicationContext.xml）中配置一个 Castor 编组器并注入实例 SpringOxmSample 中，如代码清单 19-29 所示。

代码清单 19-29 applicationContext.xml

```

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
    <bean id="springOxm" class="com.smart.oxm.SpringOxmSample"
        p:marshaller-ref="castorMarshaller"
        p:unmarshaller-ref="castorMarshaller" />
    <bean id="castorMarshaller"
        class="org.springframework.oxm.castor.CastorMarshaller"
        p:mappingLocation="classpath:/config/mapping.xml" />
</beans>

```

Spring OXM 仍然遵守不重造轮子的原则，只是为各个绑定映射组件提供一致的访问接口及异常处理机制。与单独使用 Castor 组件一样，我们需要编写一个映射文件，如代码清单 19-30 所示。

代码清单 19-30 mapping.xml

```

<?XML version="1.0"?>
<mapping>
    <class name="com.smart.oxm.domain.User">
        <map-to XML="user"/>
        <field name="userId" type="integer">
            <bind-XML name="userId" node="element"/>
        </field>
        ...
        <field name="logs" type="com.smart.oxm.domain.User" collection="arraylist">

```

```

        <bind-XML name="logs" />
    </field>
</class>
</mapping>

```

Spring OXM 屏蔽第三方映射组件中编组器及反编组器的实现细节，也就是说映射转换程序与具体映射组件解耦，如果有一天需求突然发生变化，则当前的映射组件不能满足需求，只需修改 Spring 配置文件，换成另一个合适的映射组件即可，而不需要修改映射转换程序，如代码清单 19-31 所示。

代码清单 19-31 SpringOxmSample.java

```

package com.smart.oxm;
...
public class SpringOxmSample {
    private Marshaller marshaller;
    private Unmarshaller unmarshaller;
    //Java转换为XML对象
    public void objectToXML()throws Exception{
        User user = getUser();
        FileOutputStream os = null;
        try {
            os = new FileOutputStream("out/SpringOxmSample.xml");
            this.marshaller.marshal(user, new StreamResult(os));
        } finally {
            ...
        }
    }
    //XML转换为Java对象
    public void XMLToObject()throws Exception{
        FileInputStream is = null;
        User user = null;
        try {
            is = new FileInputStream("out/SpringOxmSample.xml");
            user = (User) this.unmarshaller.unmarshal(new StreamSource(is));
        } finally {
            ...
        }
    }
}

```

Spring OXM 的一个最直接的好处是可以通过使用 Spring 框架的其他特性简化配置。Spring 的 Bean 支持将实例化的 O/X 编组器注入（前面提到过的“依赖项注入”）使用那些编组器的对象。遵循坚实的面向对象的设计实践，Spring OXM 框架只定义两个接口：Marshaller 和 Unmarshaller，它们用于执行 O/X 功能。另一个重大好处是，这些接口的实现完全对开发人员开放，开发人员可以轻松地切换它们而无需修改代码。例如，如果你一开始使用 Castor 进行 O/X 转换，但后来发现它缺乏你需要的某个功能，这时可以切换到 XMLBeans 而无需更改任何代码。唯一需要做的就是更改 Spring 配置文件以使用新的 O/X Mapping 组件。

使用 Spring OXM 的另一个好处是统一的异常层次结构。Spring OXM 遵循使用它的数据访问模块建立的模式，将原始异常对象包装到 Spring 自身专为 OXM 建立的运行时

异常中。由于第三方 O/X Mapping 组件的原始异常被包装到 Spring 运行时异常中,这也是我们能够查明出现异常的根本原因。不必费心修改代码以捕获异常,因为异常已经包装到一个运行时异常中。

19.5 小结

在本章中,首先分析了 XML 及其解析技术的发展过程,详细介绍了 XML 处理利器 XStream O/X Mapping 组件的使用,并介绍了目前流行的另外几个 O/X Mapping 组件的使用方法,并进行了总结比较。接着结合 Spring 提供的 OXM 特性,分析了 Spring OXM 框架及两个重要的接口 Marshaller 和 Unmarshaller,并详细介绍了 O/X Mapping 组件的配置方法。最后通过一个简单的实例,介绍了如何应用 Spring OXM 处理 Java 对象与 XML 之间的相互映射。不管采用何种方式处理,Java 应用程序的 XML 数据绑定可以归纳为两种方式:一是根据 XML 文档文法生成 Java 语言代码(如 JAXB、Castor 等);二是使用某种形式的映射绑定方法,也就是设定 Java 类如何与 XML 进行关联(如 XStream、Castor、JiBX)。

在实际应用过程中,这两种方式都有各自的长处。如果使用由 Schema 或 DTD 定义的稳定文档结构,并且该结构适合应用程序的需要,则代码生成方法可能是最佳的选择。如果使用现有的 Java 实体类,或者希望使用类的结构,该结构反映应用程序对数据的使用法,而不是 XML 结构,则映射方法是最佳的选择。

第 20 章

实战型单元测试

单元测试对保障应用程序正确性而言，其重要性怎样强调都不为过。TestNG 是必须事先掌握的基础测试框架，大多数测试框架和测试工具都在此基础上扩展而来。Spring 测试框架为集成 TestNG、JUnit 等单元测试框架提供了很好的支持，并为测试 Spring 应用提供了许多基础设施。在项目单元测试过程中，不可避免地要涉及测试环境准备，如模拟接口测试、测试数据准备等繁杂工作。Mockito、Unitils、DbUnit 等框架的出现，使得这些问题有了很好的解决方案，特别是 Unitils 结合 DbUnit 对测试 DAO 层提供了强大的支持，大大提高了编写测试用例的效率和质量。

本章主要内容：

- ◆ 概述单元测试相关概念及意义
- ◆ 简要分析对单元测试存在的误解
- ◆ Spring 测试框架简介
- ◆ TestNG、Mockito、Unitils 测试框架简介
- ◆ 使用 TestNG、Mockito、Unitils 及 Spring 进行单元测试
- ◆ 面向数据库应用的测试
- ◆ 测试实战

本章亮点：

- ◆ 对单元测试存在的误解进行全面分析
- ◆ 简明扼要地介绍了 TestNG、Mockito、Unitils 的使用
- ◆ 使用 Excel 准备测试数据及验证数据来简化 DAO 测试

20.1 单元测试概述

一种商品只有通过严格检测才能投放市场，一架飞机只有经过严格测试才能上天，同样，一款软件只有对其各项功能进行严格测试后才能交付使用。不管一款软件多么复杂，它都是由相互关联的方法和类组成的，每个方法和类都可能隐藏着 Bug。只有防微杜渐、小步前进，才可以保证软件大厦的稳固性；否则隐藏在类中的 Bug 随时都有可能像打开的潘多拉魔盒一样让程序陷于崩溃之中，难以驾驭。

按照软件工程思想，软件测试可以分为单元测试、集成测试、功能测试、系统测试等。功能测试和系统测试一般来说是测试人员的职责，但单元测试和集成测试则必须由开发人员保证。

20.1.1 为什么需要单元测试

软件开发的标准过程包括以下几个阶段：『需求分析阶段』→『设计阶段』→『实现阶段』→『测试阶段』→『发布』。其中，测试阶段通过人工或者自动手段来运行或测试某个系统的过程，其目的在于检验它是否满足规定的需求或弄清预期结果与实际结果之间的差别。测试过程按 4 个步骤进行，即单元测试、集成测试、系统测试及发版测试。其中，功能测试主要检查已实现的软件是否满足需求规格说明中确定的各种需求，以及软件功能是否完全、正确。系统测试主要对已经过确认的软件纳入实际运行环境中，与其他系统成分组合在一起进行测试。单元测试、集成测试由开发人员进行，是我们关注的重点，下文对两者进行详细说明。

1. 单元测试

单元测试是开发者编写的一小段代码，用于检验目标代码的一个很小的、很明确的功能是否正确。通常而言，一个单元测试用于判断某个特定条件或特定场景下某个特定函数的行为。例如，用户可能把一个很大的值放入一个有序 List 中，然后确认该值出现在 List 的尾部。或者，用户可能会从字符串中删除匹配某种模式的字符，然后确认字符串确实不再包含这些字符。

单元测试由程序员自己来完成，最终受益的也是程序员自己。可以这么说，程序员有责任编写功能代码，同时也有责任为自己的代码编写单元测试。执行单元测试，就是为了证明这段代码的行为和我们期望的一致。

在一般情况下，一个功能模块往往会调用其他功能模块完成某项功能，如业务层的业务类可能会调用多个 DAO 完成某项业务。在对某个功能模块进行单元测试时，我们希望屏蔽对外在功能模块的依赖，以便将焦点放在目标功能模块的测试上。这时模拟对象将是最有力的工具，它根据外在模块的接口模拟特定的操作行为，这样单元测试就可以在假设关联模块正确工作的情况下验证本模块逻辑的正确性。

2. 集成测试

单元测试和开发工作是并驾齐驱的，甚至是前置性的工作。除了一些显而易见的功能外，大部分功能（类的方法）都必须进行单元测试，通过单元测试可以保障功能模块的正确性。而集成测试则是在功能模块开发完成后，为验证功能模块之间匹配调用的正确性而进行的测试。在单元测试时，往往需要通过模拟对象屏蔽外在模块的依赖，而集成测试恰恰是要验证模块之间集成后的正确性。

举个例子，当对 `UserService` 这个业务层的类进行单元测试时，可以通过创建 `UserDao` 和 `LoginLogDao` 模拟对象，在假设 DAO 类正确工作的情况下对 `UserService` 进行测试。而对 `UserService` 进行集成测试时，则应该注入真实的 `UserDao` 和 `LoginLogDao` 进行测试。

所以，一般来讲，集成测试面向的层面要更高一些，一般对业务层和 Web 层进行集成测试；单元测试则面向一些功能单一的类（如字符串格式化工具类、数据计算类）。当然，我们可能对某个类既进行单元测试又进行集成测试，如 `UserService` 在模块开发期间进行单元测试，而在关联的 DAO 类开发完成后再进行集成测试。

3. 测试的好处

在编写代码的过程中，一定会反复调试以保证它能够编译通过。但代码通过编译，只是说明了它的语法正确，而无法保证它的语义也一定正确，没有任何人可以轻易承诺这段代码的行为一定是正确的。幸运的是，单元测试会为我们的承诺提供保证。编写单元测试就是为了验证这段代码的行为是否与我们期望的一致。有了单元测试，我们就可以自信地交付自己的代码，减少后顾之忧。总之，进行单元测试，会带来以下好处：

- ☐ 软件质量最简单、最有效的保证。
- ☐ 是目标代码最清晰、最有效的文档。
- ☐ 可以优化目标代码的设计。
- ☐ 是代码重构的保障。
- ☐ 是回归测试和持续集成的基石。

20.1.2 单元测试之误解

认为单元测试影响开发进度，一是借口，拒绝对单元测试相关知识进行学习（单元测试、代码重构、版本管理是开发人员的必备技能）；二是单元测试是“先苦后甜”，刚开始搭建环境，引入额外工作，看似“影响进度”，但从长远来看，由于程序质量提升、代码返工减少、后期维护工作量缩小、项目风险降低，从而在整体上赢了回来。

误解一：影响开发进度

一旦编码完成，开发人员总是会迫切希望进行软件的集成工作，这样他们就能够看到系统的实际运行效果。这在外表上看来好像加快了进度，而像单元测试这样的活动被看作影响进度的原因之一，推迟了对整个系统进行集成测试的时间。

在实践中，这种开发步骤常常会导致这样的结果：软件甚至无法运行。更进一步的结果是大量的时间将被花费在跟踪那些包含在独立单元里的简单 Bug 上面。在个别情况下，这些 Bug 也许是琐碎和微不足道的，但总的来说，它们会推迟软件产品交付的时间，而且也无法确保软件产品能够可靠运行。

在实际工作中，进行完整计划的单元测试和编写实际的代码所花费的精力大致上是相同的。一旦完成了这些单元测试工作，很多 Bug 将被纠正，开发人员能够进行更高效的系统集成工作。这才是真实意义上的进步，所以说完整计划下的单元测试是对时间的更高效利用。

误解二：增加开发成本

如果不重视程序中那些未被发现的 Bug 可能带来的后果，则这种后果的严重程度可以从一个 Bug 引起的用户使用不便到系统崩溃。这种后果可能常常会被软件的开发人员所忽视，这种情况会长期损害软件开发商的声誉，并且会对未来的市场产生负面影响。相反，一个可靠的软件系统的良好声誉将有助于一个软件开发商获取未来的市场。

很多研究成果表明，无论什么时候做出修改都要进行完整的回归测试，在生命周期中尽早地对软件产品进行测试将使效率和质量得到最好的保证。Bug 发现得越晚，修改它所需的费用就越高，因此，从经济角度来看，应该尽可能早期地查找和修改 Bug。而单元测试就是一个在早期抓住 Bug 的机会。

相比后阶段的测试，单元测试的创建更简单，且维护更容易，同时可以更方便地进行重构。从全程的费用来考虑，相比于那些复杂且旷日持久的集成测试，或不稳定的软件系统，单元测试所需的费用是很低的。

误解三：我是个编程高手，无须进行单元测试

在每个开发团队中至少有一个这样的开发人员，他非常擅长于编程，他开发的软件总是在第一时间就可以正常运行，因此不需要进行测试。你是否经常听到这样的借口？在现实世界里，每个人都会犯错误。即使某个开发人员可以抱着这种态度在很少的一些简单程序中应付过去，但真正的软件系统是非常复杂的。真正的软件系统不可以寄希望于没有进行广泛的测试和 Bug 修改过程就可以正常工作。编码不是一个可以一次性通过的过程。在现实世界中，软件产品必须进行维护以对功能需求的改变做出及时响应，并且要对最初的开发工作遗留下来的 Bug 进行修改。你希望依靠那些原始作者进行修改吗？这些制造出未经测试的代码的资深工程师还会继续在其他地方制造这样的代码。在开发人员做出修改后进行可重复的单元测试，可以避免产生那些令人不快的副作用。

误解四：测试人员会测出所有 Bug

一旦软件可以运行，开发人员又要面对这样的问题：在考虑软件全局复杂性的前提下对每个单元进行全面的测试。这是一件非常困难的事情，甚至在创造一种单元调用的测试条件时，要全面考虑单元被调用时的各种入口参数。在软件集成阶段，对单元功能全面测试的复杂程度远远超过独立进行的单元测试过程。

最后的结果是测试将无法达到它应有的全面性，一些缺陷将被遗漏，并且很多 Bug 将被忽略。让我们类比一下：假设我们要清理一台电脑主机中的灰尘，如果没有把主机中的各个部件（显卡、内存等）拆开，无论用什么工具，一些灰尘还会隐藏在主机的某些角落无法清理。但我们换个角度想想，如果把主机的每个部件一一拆开，这些死角里的灰尘就容易被发现和接触到了，并且每个部件的灰尘都可以毫不费力地进行清理。

20.1.3 单元测试之困境

测试在软件开发过程中一直都是备受关注的，测试不仅仅局限于软件开发中的一个阶段，它已经开始贯穿于整个软件开发过程。大家普遍认识到，如果测试能在开发阶段进行有效执行，程序的 Bug 就会被及早发现，其质量就能得到有效的保证，从而减少软件开发总成本。但是，相对于测试这个词的流行程度而言，大家对单元测试的认知普遍存在一些偏差，特别是一些程序员很容易陷入一些误区，导致测试并没有在他们所在的开发项目中起到有效的作用。下面对一些比较具有代表性的误区困境进行剖析，并对测试背后所蕴含的一些设计思考进行阐述，希望能够起到抛砖引玉的作用。

误区、困境一：使用 `System.out.print` 跟踪和运行程序就够了

这个误区可以说是程序员的一种通病，认为使用 `System.out.print` 就可以确保编写代码的正确性，无须编写测试用例，他们觉得编写测试用例是在“浪费时间”。使用 `System.out.print` 输出结果，以肉眼观察这种刀耕火种的方式进行测试，不仅效率低下，而且容易出错。

误区、困境二：存在太多无法测试的东西

在编码的时候，确实存在一些看起来比较难测试的代码，但并非无法测试。并且在大多数情况下，是因为被测试的代码在设计时没有考虑到可测试性的问题。编写的程序不仅与第三方一些框架耦合过紧，而且过于依赖其运行环境，从而表现出被测试的代码本身很难测试。

误区、困境三：测试代码可以随意写

编写测试代码时抱着一种随意的态度，没有弄清测试的真正意图。编写测试代码只是为了应付任务而已，先编写程序实现代码，然后再去编写一些单元测试。表现出来的结果是测试过于简单，只走形式和花架，将大量 Bug 传递给系统测试人员。

误区、困境四：不关心测试环境

手工搭建测试环境，测试数据，造成维护困难，占据了大量时间，严重影响效率。对测试产生的“垃圾”不清除、不处理，造成测试不能重复进行，导致脆弱的测试。需要维护好测试环境，做一个“低碳环保”的测试者。

误区、困境五：测试环境依赖性大

测试环境依赖性大，没有有效隔离测试目标及其依赖环境。一是使测试不聚焦；二是常因依赖环境的影响造成失败；三是因依赖环境太厚重从而降低测试的效率（依赖数据库或依赖网络资源，如邮件系统、Web 服务）。

20.1.4 单元测试基本概念

被测系统：SUT (System Under Test)

被测系统 (System Under Test, SUT) 表示正在被测试的系统，目的是测试系统能否正确操作。这一词语常用于软件测试中。软件系统测试的一个特例是对应用程序的测试，称为被测应用程序 (Application Under Test, AUT)。

SUT 也表明软件已经到了成熟期，因为系统测试在测试周期中是集成测试的最后一阶段。

测试替身：Test Double

在进行单元测试时，使用 Test Double 减少对被测对象的依赖，使得测试更加单一。同时，让测试案例执行的时间更短，运行更加稳定，同时能对 SUT 内部的输入/输出进行验证，让测试更加彻底深入。但是，Test Double 也不是万能的，Test Double 不能被过度使用，因为实际交付的产品是使用实际对象的，过度使用 Test Double 会让测试变得越来越脱离实际。

测试夹具：Test Fixture

所谓测试夹具 (Fixture)，就是测试运行程序 (Test Runner) 会在测试方法之前自动执行初始化、回收资源的工作。在 TestNG 中，通过 `@BeforeMethod` 注解标注的方法进行初始化工作；通过 `@AfterMethod` 注解标注的方法进行资源的回收工作。在一个测试类中，甚至可以使用多个 `@BeforeMethod` 注解标注多个方法，这些方法都是在每个测试之前运行的。需要说明一点，`@BeforeMethod` 是在每个测试方法运行前均初始化一次；同理，`@AfterMethod` 是在每个测试方法运行完毕后均执行一次。也就是说，经过这两个注解的初始化和注销，可以保证各个测试之间的独立性而互不干扰，其缺点是效率低。另外，不需要在超类中显式调用初始化和清除方法，只要它们不被覆盖，测试运行程序将根据需要自动调用这些方法。超类中 `@BeforeMethod` 注解标注的方法在子类的 `@BeforeMethod` 方法之前调用（与构造函数调用顺序一致），`@AfterMethod` 注解标注的方法是子类在超类之前运行。

一个测试用例可以包含若干个标注 `@Test` 注解的测试方法，测试用例测试一个或多个类 API 接口的正确性。当然，在调用类 API 时，需要事先创建这个类的对象及一些关联的对象，这组对象就被称为测试夹具，相当于测试用例的“工作对象”。

前面讲过，一个测试用例类可以包含多个标注 `@Test` 注解的测试方法，在运行时，每个测试方法都对应一个测试用例类的实例。当然，用户可以在具体的测试方法中声明并实例化业务类的实例，在测试完成后销毁它们。但是，这样一来就要在每个测试方法中都重复这些代码，因为 `TestCase` 实例依照以下步骤运行。

- ① 创建测试用例的实例。
- ② 使用@BeforeMethod 注解修饰用于初始化夹具的方法。
- ③ 使用@AfterMethod 注解修饰用于注销夹具的方法。
- ④ 保证这两种方法不能带有任何参数。

TestCase 实例运行过程如图 20-1 所示。

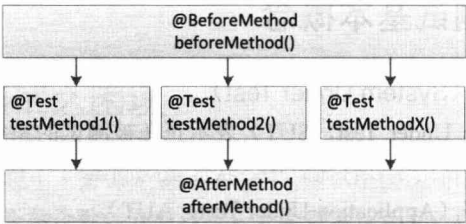


图 20-1 方法级别夹具执行示意图

之所以每个测试方法都需要按以上流程运行，是为了防止测试方法相互之间的影响，因为在同一个测试用例类中不同的测试方法可能会使用相同的测试夹具，前一个测试方法对测试夹具的更改会影响后一个测试方法的现场。而通过如上的运行步骤后，因为每个测试方法在运行前都重建了运行环境，所以测试方法相互之间就不会有影响了。

但是，这种夹具设置方式还是引来了批评，因为它效率低下，特别是在设置 Fixture 非常耗时的情况下（如设置数据库连接）。而且对于不会发生变化的测试环境或者测试数据来说，是不会影响到测试方法的执行结果的，也就没有必要针对每个测试方法重新设置一次夹具。因此，在 TestNG 中引入了类级别的夹具设置方法，编写规范说明如下。

- ① 创建测试用例的实例。
- ② 使用@BeforeClass 注解修饰用于初始化夹具的方法。
- ③ 使用@AfterClass 注解修饰用于注销夹具的方法。
- ④ 保证这两种方法不能带有任何参数。

类级别的夹具仅会在测试类中的所有测试方法执行之前执行初始化，并在全部测试方法测试完毕之后执行注销方法，如图 20-2 所示。

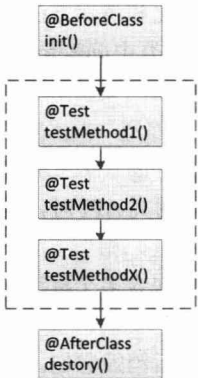


图 20-2 类级别夹具执行示意图

此外, TestNG 还提供了分组级、套件级注解 BeforeGroups、AfterGroups、BeforeSuite、AfterSuite 用来处理测试分组、套件初始化及注销夹具的方法。

测试用例: Test Case

有了测试夹具, 就可以开始编写测试用例的测试方法了。当然也可以不需要测试夹具而直接编写测试用例的测试方法。在 TestNG 中编写测试方法非常简单, 只需在每个测试方法中标注@Test 注解即可。

可以在一个测试用例中添加多个测试方法, 运行器会为每个方法生成一个测试用例实例并分别运行。

测试套件: Test Suite

如果每次只能运行一个测试用例, 那么又陷入了传统测试(使用 main()方法进行测试)的窘境: 手工运行一个个测试用例, 这是非常烦琐和低效的。测试套件专门为解决这一问题而生。它通过 TestSuite 对象将多个测试用例组装成一个测试套件, 则测试套件批量运行。需要特别指出的是, 可以把一个测试套件整个添加到另一个测试套件中, 就像小筐装进大筐里变成一个筐一样。

20.2 TestNG 快速进阶

20.2.1 TestNG 概述

TestNG 是一个设计用来简化广泛的测试需求的测试框架, 其灵感来自 JUnit 和 NUnit, 但引入了一些新的功能, 使其功能更强大, 使用更方便。TestNG 消除了大部分旧框架的限制, 使开发人员能够编写更加灵活和强大的测试。因为它在很大程度上借鉴了 Java 注解(Java 5.0 引入的)来定义的测试, 它也可以告诉你如何在真实的 Java 语言生产环境中使用这个新功能。TestNG 添加了诸如灵活的装置、测试分类、参数测试、依赖方法、数据驱动等特性, 让开发人员编写测试更加灵活、简便。

TestNG 设计的出发点是不仅可以用于单元测试, 而且可以用于集成测试。相比于 JUnit 只适合单元测试, TestNG 无疑会走得更远。

编写一个测试的过程有 3 个典型步骤。

- ① 编写测试的业务逻辑并在代码中插入 TestNG 注解。
- ② 将测试信息添加到 testng.xml 或者 build.xml 文件中。
- ③ 运行 TestNG。

20.2.2 TestNG 生命周期

TestNG 测试用例的完整生命周期要经历以下阶段：类级初始化资源处理、方法级初始化资源处理、执行测试用例中的方法、方法级销毁资源处理、类级销毁资源处理。其中，类级初始化、销毁资源处理方法在一个测试用例类中只能运行一次；方法级初始化、销毁资源处理方法在执行测试用例的每个测试方法中都会运行一次，以防止测试方法相互之间的影响。测试用例的执行过程如图 20-3 所示。

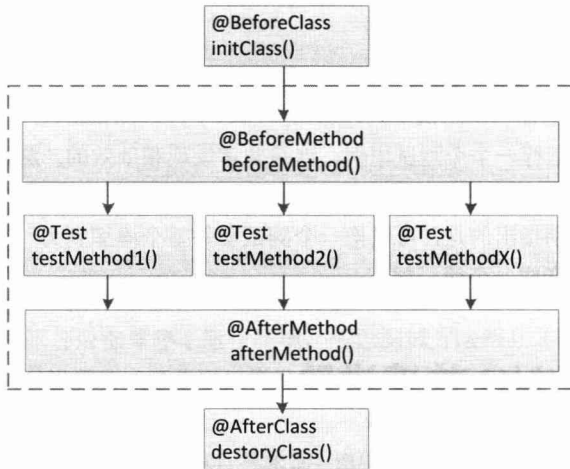


图 20-3 TestNG 测试用例执行示意图

如果在一个测试用例中编写了多个初始化处理方法，则运行时先执行位于最后面的初始化方法，然后往前一个个执行初始化方法。对于多个销毁资源处理方法，则按照方法的顺序一个个往后执行。

20.2.3 使用 TestNG

1. 测试方法

在 TestNG 中使用 `@Test` 注解来标注一个测试方法。此外可以采用 Java 5.0 的静态导入功能导入断言 `Assert` 类，这样就可以很方便地在测试方法中使用断言方法。下面通过一个实例来快速体验 TestNG 测试方法，如代码清单 20-1 所示。

代码清单 20-1 测试方法

```
package sample.testng;
import org.testng.annotations.*;
import static org.testng.Assert.*;
public class MoneyTest{
    private Money f12CHF; //12 瑞士法郎
    private Money f14CHF; //14 瑞士法郎
    private Money f28USD; //28 美国美元
```

```
@BeforeMethod
protected void setUp() {
    f12CHF= new Money(12, "CHF");
    f14CHF= new Money(14, "CHF");
    f28USD= new Money(28, "USD");
}
// 测试方法，在测试方法中标注@Test 注解，方法
// 签名可以任意取名，可以声明抛出异常
@Test
public void moneyBag() { // ①
    Money bag[]= { f26CHF, f28USD };
    MoneyBag expected= new MoneyBag(bag);
    assertEquals(expected, f12CHF.add(f28USD.add(f14CHF))); // ②
}

@AfterMethod
protected void tearDown() {}
}
```

在 TestNG 中，只要在每个测试方法中添加@Test 注解即可。像②处的 assertEquals()断言方法就是测试 Money 的 add()方法功能运行正确性的测试规则。

可以在 MoneyTest 中添加多个测试方法，运行器会为每个方法生成一个测试用例实例并分别运行。

2. @BeforeClass 和@AfterClass

在 TestNG 中加入了两个注解：@BeforeClass 和@AfterClass，使用这两个注解的方法，在一个 Test 类的所有测试方法执行前后各执行一次。这是为了能在 @BeforeClass 中初始化一些昂贵的资源，如数据库连接，然后执行所有的测试方法，最后在@AfterClass 中释放资源。对于初学者来讲，很容易混淆@BeforeClass/@AfterClass 与@BeforeMethod/@AfterMethod，为此表 20-1 对它们作了一下对比。

表 20-1 @BeforeClass/@AfterClass与@BeforeMethod/@AfterMethod的区别

@BeforeClass/@AfterClass	@BeforeMethod/@AfterMethod
在一个类中只能出现一次	在一个类中可以出现多次，执行顺序不确定
方法名不作限制	方法名不作限制
在类中只执行一次	在每个测试方法之前或者之后都会执行一次
@BeforeClass 父类中标识了该注解的方法将会先于当前类中标识了该注解的方法执行。	@BeforeMethod 父类中标识了该注解的方法将会先于当前类中标识了该注解的方法执行。
@AfterClass 父类中标识了该注解的方法将会在当前类中标识了该注解的方法之后执行	@AfterMethod 父类中标识了该注解的方法将会在当前类中标识了该注解的方法之后执行
必须声明为 public static	必须声明为 public，并且非 static
所有标识为@AfterClass 的方法一定会被执行，即使在标识为@BeforeClass 的方法抛出异常的情况下也一样会被执行	所有标识为@AfterMethod 的方法一定会被执行，即使在标识为@BeforeMethod 或者@Test 的方法抛出异常的情况下也一样会被执行

3. 异常测试

因为使用了注解特性，所以 TestNG 测试异常非常简单明了。通过对 `@Test` 传入 `expected` 参数值，即可测试异常。在传入异常类后，测试类如果没有抛出异常或者抛出一个不同的异常，本测试方法就将失败。如代码清单 20-2 所示为一个简单的异常测试实例。

代码清单 20-2 异常测试

```
package sample.testng;
import java.util.*;
...
public class TestNGExceptionTest {
    private User user;

    @BeforeClass
    public void init() {
        user = null;
    }

    @Test(enabled = true, expectedExceptions = NullPointerException.class)
    public void testUser(){
        assertNotNull(user.getUserName());
    }
}
```

预期抛出空指针异常

4. 超时测试

通过在 `@Test` 注解中为 `timeOut` 参数指定时间值，即可进行超时测试。如果测试运行时间超过指定的毫秒数，则测试失败。超时测试对网络链接类非常重要。通过 `timeOut` 进行超时测试非常简单，如代码清单 20-3 所示为一个简单的超时测试实例。

代码清单 20-3 超时测试

```
package sample.testng;
import java.util.*;
...
public class TestNGTimeoutTest {
    ...

    @Test(timeOut = 10)
    public void testUser(){
        assertNotNull(user);
        assertEquals( user.getUserName(), "admin");
    }
}
```

指定测试方法的超时时间

5. 参数化测试

为了测试程序的健壮性，可能需要模拟不同的参数对方法进行测试。如果为每个类型的参数创建一个测试方法，则是一件很难接受的事。幸好 TestNG 提供了参数化测试，它能够创建由参数值供给的通用测试，从而为每个参数都运行一次，而不必创建多个测试方法，如代码清单 20-4 所示。

代码清单 20-4 参数化测试

```

package sample.testng;
import static org.testng.Assert.*;
import org.testng.annotations.*;

public class TestNGParameterTest {
    private SimpleDateFormat simpleDateFormat;

    @DataProvider(name = "testParam")
    public static Object[][] getParamters() {
        String[][] params = {
            { "2016-02-01 00:30:59", "yyyyMMdd", "20160201" },
            { "2016-02-01 00:30:59", "yyyy年MM月dd日", "2016年02月01日" },
            { "2016-02-01 00:30:59", "HH时mm分ss秒", "00时30分59秒" } };
        return params;
    }

    @Test(dataProvider = "testParam")
    public void testSimpleDateFormat() throws ParseException{
        SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        Date d = df.parse(this.date);
        simpleDateFormat = new SimpleDateFormat(this.dateformat);
        String result = simpleDateFormat.format(d);
        assertEquals( result, expectedDate);
    }
}

```

参数数据提供者

@Test 属性 dataProvider 的值要与参数数据提供者的 dataProvider 值保持一致

首先编写测试类的参数数据提供者方法，然后用此方法进行参数初始化。该方法返回一个 `Object[][]` 类型。用 `@DataProvider` 注解来标注该方法，并设置 `name` 值。之后在需要测试的方法中设置 `@Test` 的 `dataProvider` 属性，其值要和上面 `@DataProvider` 修饰的方法中的 `name` 值保持一致。

6. 分组测试

TestNG 支持执行复杂的分组测试。不仅可以声明单个测试用例内的测试方法的分组，而且还可以声明不同测试用例类级的分组。在执行 TestNG 测试时，可以指定要执行的测试分组及排除不执行的测试分组，如代码清单 20-5 所示。

代码清单 20-5 分组测试

```

package sample.testng;
import org.testng.annotations.*;
...
@Test(groups = {"class-group"})
public class TestNGGroupsTest1 {

    @Test(groups = {"group1", "group2"})
    public void testMethod1() {
    }

    @Test(groups = {"group1", "group2"})
}

```

设置用例级分组

设置测试方法级分组，一个方法可以属于多个分组

```

    public void testMethod2() {
    }

    @Test(groups = {"group1"})
    public void testMethod3() {
    }
}

@Test(groups = {"class-group"})
public class TestNGGroupsTest2 {

    @Test(groups = {"group1", "group2"})
    public void testMethod3() {
    }

    ...
}

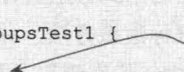
```

7. 依赖测试

有些时候，我们需要测试方法按照一个特定的顺序被调用。这非常有用，比如，在运行某个测试方法前需要先运行特定的测试方法，或希望初始化方法也作为测试方法（因为被标注为@BeforeXXX/@AfterXXX的方法不作为测试报告的一部分）。为了实现这些需求，TestNG为@Test注解提供了dependsOnMethods或dependsOnGroups属性来实现测试方法间的依赖关系，如代码清单20-6所示。

代码清单 20-6 依赖测试

```

package sample.testng;
import org.testng.annotations.*;
...
public class TestNGGroupsTest1 {
    
    @Test(dependsOnMethods= {" testMethod1", " testMethod2"})
    public void testMethod3() {
    }

    @Test
    public void testMethod1() {
    }

    @Test
    public void testMethod2() {
    }
}

```

测试方法 testMethod3() 设置了两个依赖的测试方法 testMethod1() 及 testMethod2()，当执行 testMethod3() 测试方法时，会先调用两个依赖的测试方法。如果 testMethod1() 或 testMethod2() 中的任何一个方法测试失败，则 testMethod3() 将不被执行。只有依赖的方法测试全部通过时，当前方法才会被调用执行，这种依赖关系被称为强依赖，也是 TestNG 默认的依赖关系。可以通过@Test提供的alwaysRun属性来改变这种强依赖，如示例中改成@Test(dependsOnMethods= {" testMethod1", " testMethod2"}, alwaysRun=true)之后，

不管测试方法 `testMethod1()` 或 `testMethod2()` 有没有通过测试, `testMethod3()` 总会被执行, 这种依赖关系被称为软依赖。

20.3 模拟利器 Mockito

20.3.1 模拟测试概述

目前支持 Java 语言的 Mock 测试工具有 EasyMock、JMock、Mockito、MockCreator、Mockrunner、MockMaker 等, 其中 Mockito 是一个针对 Java 的 Mocking 框架。它与 EasyMock 和 JMock 很相似, 是一套通过简单的方法对指定的接口或类生成 Mock 对象的类库, 避免了手工编写 Mock 对象的烦琐。但 Mockito 通过在执行后校验是否已经被调用, 它消除了对期望行为 (Expectations) 的需要。使用 Mockito, 在准备阶段只需花费很少的时间, 可以使用简洁的 API 编写出漂亮的测试, 而且可以对具体的类创建 Mock 对象, 并且有“监视”非 Mock 对象的能力。

Mockito 使用起来非常简单, 学习成本很低, 而且具有非常简洁的 API, 测试代码的可读性很高, 因而十分受欢迎, 用户群越来越多, 很多开源软件也选择了 Mockito。要想了解更多有关 Mockito 的信息, 可以访问其官方网站 <http://www.mockito.org/>。在开始使用 Mockito 之前, 先简单了解一下 Stub 和 Mock 的区别。相比于 EasyMock、JMock, 其编写出来的代码更加容易阅读。无须录制 `mock()` 方法调用就返回默认值是一个很大的优势。目前最新的版本是 1.10.19。

Stub 对象用来提供测试时所需的测试数据, 可以对各种交互设置相应的回应。例如, 可以设置方法调用的返回值等。在 Mockito 中, `when(...).thenReturn(...)` 这样的语法便是设置方法调用的返回值。另外也可以设置方法在何时调用会抛出异常等。

Mock 对象用来验证测试中所依赖对象间的交互是否能够达到预期。在 Mockito 中用 `verify(...).methodXxx(...)` 语法来验证 `methodXxx()` 方法是否按照预期进行了调用。有关 Stub 和 Mock 的详细论述请见 Martin Fowler 的文章 *Mocks Aren't Stub*, 地址为 <http://martinfowler.com/articles/mocksArentStubs.html>。在 Mocking 框架中, 所谓的 Mock 对象实际上是作为上述 Stub 和 Mock 对象同时使用的, 因为它既可以设置方法调用的返回值, 又可以验证方法的调用。

20.3.2 创建 Mock 对象

可以对类和接口进行 Mock 对象的创建, 创建的时候可以为 Mock 对象命名, 也可以忽略命名参数。为 Mock 对象命名的好处就是调试的时候会很方便。比如, 我们 Mock 多个对象, 在测试失败的信息中会把有问题的 Mock 对象打印出来, 有了名字就可以很容易地定位和辨认出是哪个 Mock 对象出现了问题。另外, 其使用也有限制, 对于 `final`

类、匿名类和 Java 的基本类型，是无法进行 Mock 的。除了用 Mock 方法来创建模拟对象，如 `mock(Class<T> classToMock)`，也可以使用 `@mock` 注解定义 Mock。下面通过实例来介绍一下如何创建一个 Mock 对象，如代码清单 20-7 所示。

代码清单 20-7 MockitoSampleTest.java 创建 Mock 对象

```
import com.smart.domain.User;
import com.smart.service.UserService;
import com.smart.service.UserServiceImpl;
import static org.mockito.Mockito.*;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import org.testng.annotations.*;
import static org.testng.Assert.*;
...
public class MockitoSampleTest{

    //① 对接口进行模拟
    UserService mockUserService = mock(UserService.class);

    //② 对类进行模拟
    UserServiceImpl mockServiceImpl = mock(UserServiceImpl.class);

    //③ 基于注解模拟类
    @Mock
    User mockUser;

    @BeforeClass
    public void initMocks() {

        //④ 初始化当前测试类所有@Mock注解模拟对象
        MockitoAnnotations.initMocks(this);
    }
    ...
}
```

在①处和②处，通过 Mockito 提供的 `mock()` 方法创建 `UserService` 用户服务接口、用户服务实现类 `UserServiceImpl` 的模拟对象。在③处，通过 `@Mock` 注解创建用户 `User` 类的模拟对象，并且需要在测试类初始化方法中，通过 `MockitoAnnotations.initMocks()` 方法初始化当前测试类中所有标注 `@Mock` 注解的模拟对象。如果没有执行这一步初始化操作，则测试时会报模拟对象为空对象异常。

20.3.3 设定 Mock 对象的期望行为及返回值

从上文中我们已经知道，可以通过 `when(mock.someMethod()).thenReturn(value)` 来设定 Mock 对象的某个方法调用时的返回值，但它同样有限制条件，即对于 `static` 和 `final` 修饰的方法是无法进行设定的。下面通过实例来介绍一下如何调用方法及设定返回值，如代码清单 20-8 所示。

代码清单 20-8 MockitoSampleTest.java 设定模拟对象的期望行为及返回值

```

import org.testng.annotations.*;
import org.mockito.Mock;
import com.smart.domain.User;
import com.smart.service.UserService;
import com.smart.service.UserServiceImpl;
...
public class MockitoSampleTest {
    ...

    //① 模拟接口UserService测试
    @Test
    public void testMockInterface() {

        //①-1 对方法设定返回值
        when(mockUserService.findUserByUserName("tom")).thenReturn(
            new User("tom", "1234"));

        //①-2 对方法设定返回值
        doReturn(true).when(mockServiceImpl).hasMatchUser("tom", "1234");

        //①-3 对void方法进行方法预期设定
        User u = new User("John", "1234");
        doNothing().when(mockUserService).registerUser(u);

        //①-4 执行方法调用
        User user = mockUserService.findUserByUserName("tom");
        boolean isMatch = mockUserService.hasMatchUser("tom", "1234");
        mockUserService.registerUser(u);

        assertNotNull(user);
        assertEquals(user.getUserName(), "tom");
        assertEquals(isMatch, true);
    }

    //② 模拟实现类UserServiceImpl测试
    @Test
    public void testMockClass() {

        //对方法设定返回值
        when(mockServiceImpl.findUserByUserName("tom"))
            .thenReturn(new User("tom", "1234"));
        doReturn(true).when(mockServiceImpl).hasMatchUser("tom", "1234");

        User user = mockServiceImpl.findUserByUserName("tom");
        boolean isMatch = mockServiceImpl.hasMatchUser("tom", "1234");
        assertNotNull(user);
        assertEquals(user.getUserName(), "tom");
        assertEquals(isMatch, true);
    }

    //③ 模拟User类测试
    @Test
    public void testMockUser() {

```



```

        when(mockUser.getUserId()).thenReturn(1);
        when(mockUser.getUserName()).thenReturn("tom");
        assertEquals(mockUser.getUserId(), 1);
        assertEquals(mockUser.getUserName(), "tom");
    }

```

在①处，模拟测试接口 `UserService` 的 `findUserByUserName()`、`hasMatchUser()` 及 `registerUser()` 方法。在①-1 处通过 `when().thenReturn()` 语法，模拟方法调用及设置方法的返回值，实例通过模拟调用 `UserService` 用户服务接口的查找用户 `findUserByUserName()` 方法，查询用户名为“tom”的详细信息，并设置返回 `User` 对象：`new User("tom", "1234")`。在①-2 处通过 `doReturn().when()` 语法，模拟判断用户 `hasMatchUser()` 方法的调用，判断用户名为“tom”、密码为“1234”的用户是否存在，并设置返回值为 `true`。在①-3 处对 `void` 方法进行方法预期设定，如实例中调用注册用户 `registerUser()` 方法。在设定调用方法及返回值之后，就可以执行接口方法调用验证。在②处和③处，模拟测试用户服务实现类 `UserServiceImpl`，测试的方法与模拟接口一致。

20.3.4 验证交互行为

`Mock` 对象一旦建立便会自动记录自己的交互行为，所以我们可以有选择地对其交互行为进行验证。在 `Mockito` 中验证 `Mock` 对象交互行为的方法是 `verify(mock).xxx()`，于是用此方法验证了 `findUserByUserName()` 方法的调用。因为只调用了一次，所以在 `verify` 中指定了 `times` 或 `atLeastOnce()` 参数。最后验证返回值是否和预期一样，如代码清单 20-9 所示。

代码清单 20-9 MockitoSampleTest.java 验证交互行为

```

import org.testng.annotations.*;
import org.mockito.Mock;
import com.smart.domain.User;
import com.smart.service.UserService;
import com.smart.service.UserServiceImpl;
...
public class MockitoSampleTest {
    ...

    //① 模拟接口UserService测试
    @Test
    public void testMockInterface() {
        ...
        when(mockUserService.findUserByUserName("tom"))
            .thenReturn(new User("tom", "1234"));
        User user = mockServiceImpl.findUserByUserName("tom");

        //①-4 验证返回值
        assertNotNull(user);
        assertEquals(user.getUserName(), "tom");
        assertEquals(isMatch, true);
    }
}

```

```

//①-5 验证交互行为
verify(mockUserService).findUserByUserName("tom");

//①-6 验证方法至少调用一次
verify(mockUserService, atLeastOnce()).findUserByUserName("tom");
verify(mockUserService, atLeast(1)).findUserByUserName("tom");

//①-7 验证方法至多调用一次
verify(mockUserService, atMost(1)).findUserByUserName("tom");
}

```

Mockio 为我们提供了丰富调用方法次数的验证机制，如被调用了特定次数 `verify(xxx, times(x))`、至少 x 次 `verify(xxx, atLeast(x))`、最多 x 次 `verify(xxx, atMost(x))`、从未被调用 `verify(xxx, never())`。在①-6 处，验证 `findUserByUserName()` 方法至少被调用一次。在①-7 处，验证 `findUserByUserName()` 方法至多被调用一次。

20.4 测试整合之王 Unitils

20.4.1 Unitils 概述

Unitils 测试框架的目的是让单元测试变得更加容易和可维护。Unitils 构建在 DbUnit 与 EasyMock 项目之上，并与 JUnit 和 TestNG 相结合。它不仅支持数据库测试，而且支持利用 Mock 对象进行测试，并且可以与 Spring 和 Hibernate 相集成。Unitils 被设计成以一种高度可配置和松散耦合的方式来添加这些服务到单元测试中，目前最新版本是 3.4.2。要使用 Unitils，首先需要在章节模块 `pom.xml` 中添加 Unitils 的相关依赖。

1. Unitils 功能特点

- ☐ 自动维护和强制关闭单元测试数据库（支持 Oracle、HSQLDB、MySQL、DB2）。
- ☐ 简化单元测试数据库连接的设置。
- ☐ 简化利用 DbUnit 测试数据的插入。
- ☐ 简化 Hibernate session 管理。
- ☐ 自动测试与数据库相映射的 Hibernate 映射对象。
- ☐ 易于把 Spring 管理的 Bean 注入单元测试中，支持在单元测试中使用 Spring 容器中的 Hibernate SessionFactory。
- ☐ 简化 EasyMock Mock 对象创建。
- ☐ 简化 Mock 对象注入，利用反射等式匹配 EasyMock 参数。

2. Unitils 模块组件

Unitils 通过模块化的方式来组织各个功能模块，采用类似于 Spring 的模块划分方式，如 `unitils-core`、`unitils-database`、`unitils-mock` 等，比以前整合在一个工程里面显得更加清晰。目前包含的所有模块如下。

- ❑ `unitils-core`: 核心内核包。
- ❑ `unitils-database`: 维护测试数据库及连接池。
- ❑ `unitils-DbUnit`: 使用 DbUnit 来管理测试数据。
- ❑ `unitils-easymock`: 支持创建 Mock 和宽松的反射参数匹配。
- ❑ `unitils-inject`: 支持在一个对象中注入另一个对象。
- ❑ `unitils-mock`: 整合各种 Mock, 在 Mock 的使用语法上进行了简化。
- ❑ `unitils-orm`: 支持 Hibernate、JPA 的配置和自动数据库映射检查。
- ❑ `unitils-spring`: 支持加载 Spring 的上下文配置, 并加载 Spring Bean 注入。

Unitils 的核心架构中包含 Moudule 和 TestListener 两个概念, 类似于 Spring 中粘连其他开源软件中的 FactoryBean 概念, 可以看成第三方测试工具的一个黏合剂。整体框架如图 20-4 所示。

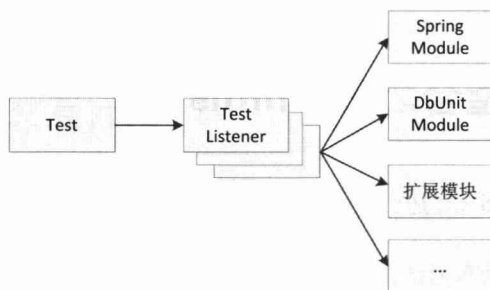


图 20-4 Unitils 架构图

通过 TestListener 可以在测试运行的不同阶段注入某些功能, 同时某个 TestListener 又被一个对应的 Module 所持有。Unitils 也可以被看作一个插件体系结构, TestListener 在整个 Unitils 中又充当了插件中扩展点的角色。从 TestListener 接口中可以看到, 它可以在 `crateTestObject`、`before(after)Class`、`before(after)TestMethod`、`beforeSetup`、`afterTeardown` 的不同切入点添加不同的动作。

3. Unitils 配置文件

- ❑ `unitils-defaults.properties`: 默认配置文件, 开启所有功能。
- ❑ `unitils.properties`: 项目级配置文件, 用于项目通用属性配置。
- ❑ `unitils-local.properties`: 用户级配置文件, 用于个人特殊属性配置。

Unitils 定义了通用的配置文件名 `unitils.properties` 和用户自定义配置文件名 `unitils-local.properties`, 并给出了默认模块及模块对应的 `className`, 便于 Unitils 加载对应的模块。但是如果用户分别在 `unitils.properties` 和 `unitils-local.properties` 文件中对相同属性配置不同值, 则将会以 `unitils-local.properties` 的配置内容为主。

4. Unitils 断言

典型的单元测试一般都包含一个重要的组成部分: 对比实际产生的结果和希望的结果是否一致的方法, 即断言方法 (`assertEquals`)。Unitils 为我们提供了一个非常实用的

断言方法，我们以第 2 章中编写的用户领域对象 `User` 为蓝本，比较两个 `User` 对象的实例，来开始认识 `Unitils` 的断言之旅。

5. `assertReflectionEquals`: 反射断言

在 Java 世界中，要比较现有两个对象实例是否相等，如果类没有重写 `equals()` 方法，则用两个对象的引用是否一致作为判断依据。有时候，我们并不需要关注两个对象是否引用同一个对象，只要两个对象的属性值一样就可以了。在 `TestNG` 单元测试中，有两种测试方式进行这样的场景测试：一是在比较实体类中重写 `equals()` 方法，然后进行对象比较；二是把对象实例的属性一个一个进行比较。不管采用哪种方法，都比较烦琐。`Unitils` 为我们提供了一种非常简单的方法，即使用 `ReflectionAssert.assertReflectionEquals` 方法，如代码清单 20-10 所示。

代码清单 20-10 `assertReflectionEquals` 反射断言测试

```
package sample.unitils;
import org.testng.annotations.*;
import com.smart.domain.User;
import org.unitils.reflectionassert.ReflectionAssert;

public class AssertReflectionEqualsTest {
    @Test
    public void testReflection() {
        User user1 = new User("tom", "1234");
        User user2 = new User("tom", "1234");
        ReflectionAssert.assertReflectionEquals(user1, user2);
    }
}
```

`ReflectionAssert.assertReflectionEquals`(期望值,实际值,比较级别)方法为我们提供了各种级别的比较断言，下面依次介绍这些级别的比较断言。

- ❑ `ReflectionComparatorMode.LENIENT_ORDER`: 忽略要断言的集合 `collection` 或者 `array` 中元素的顺序。
- ❑ `ReflectionComparatorMode.IGNORE_DEFAULTS`: 忽略 Java 类型的默认值。如当引用类型为 `null`，整型类型为 `0`，或者布尔类型为 `false` 时，断言忽略这些值的比较。
- ❑ `ReflectionComparatorMode.LENIENT_DATES`: 比较两个实例的 `Date` 是不是都设置了值或者都为 `null`，而忽略 `Date` 的值是否相等。

6. `assertLenientEquals`: 断言

`ReflectionAssert` 类为我们提供了两种比较断言：一种是可忽略顺序的；另一种是可忽略默认值的断言 `assertLenientEquals`，使用这种断言就可以进行简单比较。下面通过实例来学习其基本用法，如代码清单 20-11 所示。

代码清单 20-11 `assertLenientEquals` 断言测试

```
package sample.unitils;
import java.util.*;
```

```

...
public class AssertReflectionEqualsTest {
    @Test
    public void testLenientEquals(){
        Integer orderList1[] = new Integer[]{1,2,3};
        Integer orderList2[] = new Integer[]{3,2,1};

        //① 测试两个数组的值是否相等，忽略顺序
        //assertReflectionEquals(orderList1, orderList2, LENIENT_ORDER);
        assertLenientEquals(orderList1, orderList2);

        //② 测试两个对象的值是否相等，忽略默认值
        User user1 = new User("tom", "1234");
        User user2 = new User("tom", "1234");
        assertLenientEquals(user1, user2);
    }
}

```

7. assertPropertyXxxEquals: 属性断言

assertLenientEquals 和 assertReflectionEquals 这两个方法是把对象作为整体进行比较，ReflectionAssert 类还为我们提供了只比较对象特定属性的方法：assertPropertyReflectionEquals() 和 assertPropertyLenientEquals()。下面通过实例学习其具体的用法，如代码清单 20-12 所示。

代码清单 20-12 assertPropertyXxxEquals 属性断言

```

package sample.unitils;
import java.util.*;
...
public class AssertReflectionEqualsTest {
    User user = new User("tom", "1234");
    assertPropertyReflectionEquals("userName", "tom", user);
    assertPropertyLenientEquals("lastVisit", null, user);
}

```

assertPropertyReflectionEquals() 断言是默认严格比较模式但是可以手动设置比较级别的断言，assertPropertyLenientEquals() 断言是既可忽略顺序又可忽略默认值的断言。

20.4.2 集成 Spring

Unitils 提供了一些在 Spring 框架下进行单元测试的特性。在基于 Spring 框架的项目开发过程中，编写代码要保持代码的非侵入性，也就是设计的类在非 Spring 容器中仍然易于测试。但是很多时候在 Spring 容器中进行测试还是非常有用的。

Unitils 提供了以下支持 Spring 的特性：

- ApplicationContext 配置的管理。
- 在单元测试代码中注入 Spring 的 Beans。

- ❑ 使用定义在 Spring 配置文件中的 Hibernate SessionFactory。
- ❑ 引用在 Spring 中定义的 Unitils 数据源。

ApplicationContext 配置

可以简单地在 一个类、方法或者属性上标注 `@SpringApplicationContext` 注解，并用 Spring 的配置文件作为参数来加载 Spring 应用上下文。下面通过实例来介绍一下如何创建 ApplicationContext，如代码清单 20-13 所示。

代码清单 20-13 加载 Spring 应用上下文

```
import org.testng.annotations.*;
import org.springframework.context.ApplicationContext;
import org.unitils.UnitilsTestNG;
import org.unitils.spring.annotation.SpringApplicationContext;
import org.unitils.spring.annotation.SpringBean;
import com.smart.service.UserService;
import static org.testng.Assert.*;

//①用户服务测试
public class UserServiceTest extends UnitilsTestNG {

    //①-1 加载Spring配置文件
    @SpringApplicationContext({"smart-service.xml", "smart-dao.xml"})
    private ApplicationContext applicationContext;

    //①-2 加载Spring容器中的Bean
    @SpringBean("userService")
    private UserService userService;

    //①-3 测试Spring容器中的用户服务Bean
    @Test
    public void testUserService () {
        assertNotNull(applicationContext);
        assertNotNull(userService.findUserByUserName("tom"));
    }
}
```

在①-1 处，通过 `@SpringApplicationContext` 注解加载 `smart-service.xml` 和 `smart-dao.xml` 两个配置文件，生成一个 Spring 应用上下文，这样就可以在注解的范围内引用 `applicationContext` 这个上下文。在①-2 处，通过 `@SpringBean` 注解注入当前 Spring 容器中相应的 Bean，如实例中加载 ID 为 `userService` 的 Bean 到当前测试范围。在①-3 处，通过 TestNG 断言验证是否成功加载 `applicationContext` 和 `userService`。Unitils 加载 Spring 上下文的过程是：首先扫描父类的 `@SpringApplicationContext` 注解，如果找到了，就在加载子类的配置文件之前加载父类的配置文件，这样就可以让子类重写配置文件和加载特定的配置文件。

细心的读者可能会发现，采用这种方式加载 Spring 应用上下文，在每次执行测试时，都会重复加载 Spring 应用上下文。Unitils 提供了在类上加载 Spring 应用上下文的能力，以避免重复加载的问题，如代码清单 20-14 所示。

代码清单 20-14 通过基类加载ApplicationContext

```
...
@SpringApplicationContext({"smart-service.xml", "smart-dao.xml"})
public class BaseServiceTest extends UnitilsTestNG {

    //加载Spring应用上下文
    @SpringApplicationContext
    public ApplicationContext applicationContext;

}
```

在父类 `BaseServiceTest` 里指定了 `Spring` 配置文件, `Spring` 应用上下文只会创建一次, 然后在子类 `SimpleUserServiceTest` 里会重用这个应用上下文。加载 `Spring` 应用上下文是一项非常繁重的操作, 如果重用这个 `Spring` 应用上下文, 就会大大提升测试的性能, 如代码清单 20-15 所示。

代码清单 20-15 通过继承使用父类的ApplicationContext

```
...
public class SimpleUserServiceTest extends BaseServiceTest {

    //① Spring容器中加载ID为用户Service的Bean
    @SpringBean("userService")
    private UserService userService1;

    //② 从Spring容器中加载与UserService相同类型的Bean
    @SpringBeanByType
    private UserService userService2;

    //③ 从Spring容器中加载与userService相同名称的Bean
    @SpringBeanByName
    private UserService userService;

    //④ 使用父类的Spring应用上下文
    @Test
    public void testApplicationContext() {
        assertNotNull(applicationContext);
    }

    @Test
    public void testUserService() {
        assertNotNull(userService.findUserByUserName("tom"));
        assertNotNull(userService1.findUserByUserName("tom"));
        assertNotNull(userService2.findUserByUserName("tom"));
    }

}
```

在①处, 使用 `@SpringBean` 注解从 `Spring` 容器中加载一个 ID 为 `userService` 的 Bean。在②处, 使用 `@SpringBeanByType` 注解从 `Spring` 容器中加载一个与 `UserService` 相同类型的 Bean, 如果找不到相同类型的 Bean, 就会抛出异常。在③处, 使用 `@SpringBeanByName` 注解从 `Spring` 容器中加载一个与当前属性名称相同的 Bean。

20.4.3 集成 Hibernate

Hibernate 是一个优秀的 O/R 开源框架，它极大地简化了应用程序的数据访问层开发。虽然我们在使用一个优秀的 O/R 框架，但并不意味着我们无须对数据访问层进行单元测试。单元测试仍然非常重要，它不仅可以确保 Hibernate 映射类的映射正确性，也可以很便捷地测试 HQL 查询等语句。Unitils 为方便测试 Hibernate，提供了许多实用的工具类，如 HibernateUnitils，使用 `assertMappingWithDatabaseConsistent()` 方法就可以方便地测试映射文件的正确性。

SessionFactory 配置

可以简单地在类、方法或者属性上标注 `@HibernateSessionFactory` 注解，并用 Hibernate 的配置文件作为参数来加载 Hibernate 应用上下文。下面通过实例来介绍一下如何创建 SessionFactory，如代码清单 20-16 所示。

代码清单 20-16 通过基类加载 SessionFactory

```
...
@HibernateSessionFactory("hibernate.cfg.xml")
public class BaseDaoTest extends UnitilsTestNG {
    @HibernateSessionFactory
    public SessionFactory sessionFactory;

    @Test
    public void testSessionFactory(){
        assertNotNull(sessionFactory);
    }
}
```

在父类 `BaseDaoTest` 里指定了 Hibernate 配置文件，Hibernate 应用上下文只会创建一次，然后在子类 `SimpleUserDaoTest` 里会重用这个应用上下文。加载 Hibernate 应用上下文是一项非常繁重的操作，如果重用这个 Hibernate 应用上下文，就会大大提升测试的性能，如代码清单 20-17 所示。

代码清单 20-17 通过继承使用父类的 SessionFactory

```
...
public class SimpleUserDaoTest extends BaseDaoTest {
    private UserDao userDao;

    //① 初始化UserDao
    @BeforeClass
    public void init(){
        userDao = new WithoutSpringUserDaoImpl();
        userDao.setSessionFactory(sessionFactory); //使用父类的SessionFactory
    }

    //② Hibernate映射测试
    @Test
    public void testMappingToDatabase() {
```



```
        HibernateUnitils.assertMappingWithDatabaseConsistent();
    }

    //③ 测试 UserDao
    @Test
    public void testUserDao(){
        assertNotNull(userDao);
        assertNotNull(userDao.findUserByUserName("tom"));
        assertEquals("tom", userDao.findUserByUserName("tom").getUserName());
    }
}
...
```

为了更好地演示如何应用 Unitils 测试基于 Hibernate 的数据访问层，在这个实例中不使用 Spring 框架。所以在执行测试时，需要先创建相应的数据访问层实例，如实例中的 userDao。其创建过程如①处所示，先手工实例化一个 UserDao，然后获取父类中创建的 SessionFactory，并设置到 UserDao 中。在②处，使用 Unitils 提供的工具类 HibernateUnitils 中的方法测试 Hibernate 映射文件。在③处，通过 TestNG 的断言验证 UserDao 相关方法，看是否与预期的结果一致。

20.4.4 集成 DbUnit

DbUnit 是一个基于 TestNG 扩展的数据库测试框架。它提供了大量的类，对数据库相关的操作进行了抽象和封装。DbUnit 通过使用用户自定义的数据集及相关操作使数据库处于一种可知的状态，从而使得测试自动化、可重复和相对独立。虽然不用 DbUnit 也可以达到这种目的，但是我们必须为此付出代价（编写大量代码、测试及维护）。既然有了这么优秀的开源框架，我们又何必再造轮子。目前其最新的版本是 2.4.8。

随着 Unitils 的出现，将 Spring、Hibernate、DbUnit 等整合在一起，使得 DAO 层的单元测试变得非常容易。Unitils 采用模块化方式来整合第三方框架，通过实现扩展模块接口 org.unitils.core.Module 来实现扩展功能。在 Unitils 中已经实现了一个 DbUnitModule，很好地整合了 DbUnit。通过这个扩展模块，就可以在 Unitils 中使用 DbUnit 强大的数据集功能，如用于准备数据的 @DataSet 注解、用于验证数据的 @ExpectedDataSet 注解。Unitils 集成 DbUnit 的流程图如图 20-5 所示。

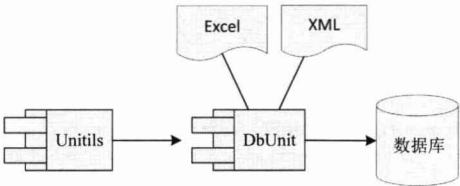


图 20-5 Unitils 集成 DbUnit 示意图

20.4.5 自定义扩展模块

Unitils 通过模块化的方式来组织各个功能模块，对外提供一个统一的扩展模块接口 `org.unitils.core.Module` 来实现与第三方框架的集成及自定义扩展。在 Unitils 中已经实现了目前一些主流框架的模块扩展，如 Spring、Hibernate、DbUnit、TestNG 等。如果这些内置的扩展模块无法满足需求，则可以实现自己的一些扩展模块。扩展 Unitils 模块很简单，如代码清单 20-18 所示。

代码清单 20-18 CustomExtModule

```
package sample.unitils.module;
import java.lang.reflect.Method;
import org.unitils.core.TestListener;
import org.unitils.core.Module;

//① 实现Module接口
public class CustomExtModule implements Module {

    //② 实现获取测试监听的方法
    public TestListener getTestListener() {
        return new CustomExtListener();
    }

    //② 新建监听模块
    protected class CustomExtListener extends TestListener {

        //③ 重写TestListener里的相关方法，完成相关扩展的功能
        @Override
        public void afterTestMethod(Object testObject, Method testMethod,
            Throwable testThrowable) {
            ...
        }

        @Override
        public void beforeTestMethod(Object testObject, Method testMethod) {
            ...
        }
    }
}
```

在①处新建自定义扩展模块 `CustomExtModule` 实现 `Module` 接口；在②处新建自定义监听模块继承 `TestListener`；在③处重写（`@Override`）`TestListener` 里的相关方法完成相关扩展的功能。实现自定义扩展模块之后，剩下的工作就是在 Unitils 配置文件 `unitils.properties` 中注册这个自定义扩展的模块。

```
unitils.modules=...,custom
unitils.module.custom.className= sample.unitils.module.CustomExtModule
```

20.5 使用 Unitils 测试 DAO 层

Spring 的测试框架为我们提供了一个强大的测试环境，解决了日常单元测试中遇到的大部分测试难题，例如：运行多个测试用例和测试方法时，Spring 上下文只需创建一次；数据库现场不受破坏；方便手工指定 Spring 配置文件、手工设定 Spring 容器是否需要重新加载等。但也存在不足的地方，基本上所有的 Java 应用都涉及数据库，带数据库应用系统的测试难点在于数据库测试数据的准备、维护、验证及清理。Spring 测试框架并不能很好地解决所有问题。要解决这些问题，必须整合多方资源，如 DbUnit、Unitils、Mokito 等。其中 Unitils 正是这样的一个测试框架。

20.5.1 数据库测试的难点

按照 Kent Back 的观点，单元测试的重要特性之一应该是可重复性。不可重复的单元测试是没有价值的。因此，好的单元测试应该具备独立性和可重复性。对于业务逻辑层，可以通过 Mockito 底层对象和上层对象来获得这种独立性和可重复性。而 DAO 层因为是和数据库打交道的层，其单元测试依赖于数据库中的数据。要实现 DAO 层单元测试的可重复性，就需要对每次因单元测试引起的数据库中的数据变化进行还原，也就是保护单元测试数据库的数据现场。

20.5.2 扩展 DbUnit 用 Excel 准备数据

在测试数据访问层（DAO）时，通常需要经过测试数据的准备、维护、验证及清理过程。这个过程不仅烦琐，而且容易出错，如数据库现场容易遭受破坏、如何对数据操作正确性进行检查等。虽然 Spring 测试框架在这一方面为我们减轻了很多工作，如通过事务回滚机制来保存数据库现场等，但在测试数据及验证数据准备方面还没有一种很好的处理方式。Unitils 框架的出现改变了难测试 DAO 的局面，它将 SpringModule、DatabaseModule、DbUnitModule 等整合在一起，使得 DAO 的单元测试变得非常容易。基于 Unitils 框架的 DAO 测试过程如图 20-6 所示。

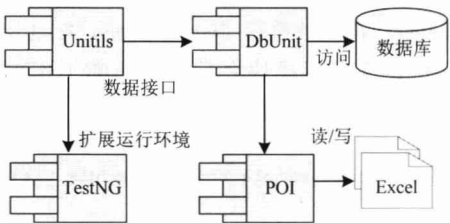


图 20-6 基于 Unitils 框架的 DAO 测试流程

以 TestNG 作为整个测试的基础框架，并采用 DbUnit 作为自动管理数据库的工具，以 XML、Excel 作为测试数据及验证数据准备，最后通过 Unitils 的数据集注解从 Excel、XML 文件中加载测试数据。使用一个注解标签就可以完成加载、删除数据操作。由于 XML 作为数据集的易用性不如 Excel，在这里就不对 XML 数据集进行讲解。下面主要讲解如何应用 Excel 作为准备及验证数据的载体，简化 DAO 单元测试。由于 Unitils 没有提供访问 Excel 的数据集工厂，因此需要编写插件支持 Excel 格式的数据源。Unitils 提供了一个访问 XML 的数据集工厂 `MultiSchemaXmlDataSetFactory`，其继承自 DbUnit 提供的数据集工厂接口 `DataSetFactory`。我们可以参考这个 XML 数据集工厂类，编写一个访问 Excel 的数据集工厂 `MultiSchemaXlsDataSetFactory` 及 Excel 数据集读取器 `MultiSchemaXlsDataSetReader`，然后在数据集读取器中调用 Apache POI 类库来读/写 Excel 文件，如代码清单 20-19 所示。

代码清单 20-19 `MultiSchemaXlsDataSetFactory.java` Excel 数据集工厂

```
import org.unitils.core.UnitilsException;
import org.unitils.DbUnit.datasetfactory.DataSetFactory;
import org.unitils.DbUnit.util.MultiSchemaDataSet;
...
public class MultiSchemaXlsDataSetFactory implements DataSetFactory {
    protected String defaultSchemaName;

    //① 初始化数据集工厂
    public void init(Properties configuration, String defaultSchemaName) {
        this.defaultSchemaName = defaultSchemaName;
    }

    //② 从Excel文件中创建数据集
    public MultiSchemaDataSet createDataSet(File... dataSetFiles) {
        try {
            MultiSchemaXlsDataSetReader xlsDataSetReader =
                new MultiSchemaXlsDataSetReader(defaultSchemaName);
            return xlsDataSetReader.readDataSetXls(dataSetFiles);
        } catch (Exception e) {
            throw new UnitilsException("创建数据集失败: "
                + Arrays.toString(dataSetFiles), e);
        }
    }

    //③ 获取数据集文件的扩展名
    public String getDataSetFileExtension() {
        return "xls";
    }
}
```

与 XML 的数据集工厂 `MultiSchemaXmlDataSetFactory` 一样，Excel 的数据集工厂也需要实现数据集工厂接口 `DataSetFactory` 的 3 个方法：`init(...)`、`createDataSet(File... dataSetFiles)` 和 `getDataSetFileExtension()`。在①处，初始化数据集工厂，需要设置一个默认的数据库表模式名称 `defaultSchemaName`。在②处，执行创建多数据集，具体读取构

建数据集的过程封装在 Excel 读取器 MultiSchemaXlsDataSetReader 中。在③处, 获取数据集文件的扩展名, 对 Excel 文件而言就是“xls”。下面来看一下这个数据集读取器的实现代码, 如代码清单 20-20 所示。

代码清单 20-20 MultiSchemaXlsDataSetReader.java Excel数据集读取器

```
import org.unitils.core.UnitilsException;
import org.unitils.DbUnit.datasetfactory.DataSetFactory;
import org.unitils.DbUnit.util.MultiSchemaDataSet;
...
// Excel 数据集读取器
public class MultiSchemaXlsDataSetReader {
    private String defaultSchemaName;

    public MultiSchemaXlsDataSetReader(String defaultSchemaName) {
        this.defaultSchemaName = defaultSchemaName;
    }
    // Excel 数据集读取器
    public MultiSchemaDataSet readDataSetXls(File... dataSetFiles) {
        try {
            Map<String, List<ITable>> tableMap = getTables(dataSetFiles);
            MultiSchemaDataSet dataSets = new MultiSchemaDataSet();
            for (Entry<String, List<ITable>> entry : tableMap.entrySet()) {
                List<ITable> tables = entry.getValue();
                try {
                    DefaultDataSet ds = new DefaultDataSet(tables
                        .toArray(new ITable[] {}));
                    dataSets.setDataSetForSchema(entry.getKey(), ds);
                } catch (AmbiguousTableNameException e) {
                    throw new UnitilsException("构造DataSet失败!", e);
                }
            }
            return dataSets;
        } catch (Exception e) {
            throw new UnitilsException("解析Excel文件出错: ", e);
        }
    }
    ...
}
```

根据传入的多个 Excel 文件构造一个多数据集。其中, 一个数据集对应一个 Excel 文件, 一个 Excel 的 Sheet 表对应一个数据库 Table。通过 DbUnit 提供的 Excel 数据集构造类 XlsDataSet, 可以很容易地将一个 Excel 文件转换为一个数据集: XlsDataSet(new FileInputStream(xlsFile))。最后将得到的多个 DataSet 用 MultiSchemaDataSet 进行封装。

下面就以一个用户 DAO 的实现类 WithoutSpringUserDaoImpl 为例, 介绍如何使用我们实现的 Excel 数据集工厂。为了让 Unitils 使用自定义的数据集工厂, 需要在 unitils.properties 配置文件中指定自定义的数据集工厂, 如代码清单 20-21 所示。

代码清单 20-21 unitils.properties配置文件

```
...
DbUnitModule.DataSet.factory.default=sample.unitils.dataset.excel.
```

```
MultiSchemaXlsDataSetFactory
DbUnitModule.ExpectedDataSet.factory.default=sample.unitils.dataset.
excel.MultiSchemaXlsData
SetFactory
```

其中, `DbUnitModule.DataSet.factory.default` 是配置数据集工厂类, 在测试方法中可以使用 `@DataSet` 注解加载指定的准备数据。默认是 XML 的数据集工厂, 这里指定自定义数据集工厂全限定类名为 `sample.unitils.dataset.excel.MultiSchemaXlsDataSetFactory`。

`DbUnitModule.ExpectedDataSet.factory.default` 是配置验证数据集工厂类, 也是指定自定义数据集工厂类, 使用 `@ExpectedDataSet` 注解加载验证数据, 如代码清单 20-22 所示。

代码清单 20-22 UserDaoTest.java 用户DAO测试

```
import org.unitils.core.UnitilsException;
import org.unitils.DbUnit.datasetfactory.DataSetFactory;
import org.unitils.DbUnit.util.MultiSchemaDataSet;
...
public class UserDaoTest extends UnitilsTestNG {
    @Test
    @DataSet //① 准备测试数据
    public void getUser() {
        ...
    }

    @Test
    @DataSet("Smart.SaveUser.xls") //② 准备测试数据
    @ExpectedDataSet //③ 准备验证数据
    public void saveUser()throws Exception {
        ...
    }
}
...
```

`@DataSet` 注解表示测试时需要寻找 DbUnit 的数据集文件进行加载。如果没有指明数据集的文件名, 则 Unitils 自动在当前测试用例所在的类路径下加载文件名为测试用例类名的数据集文件, 如实例中的①处, 将到 `UserDaoTest.class` 所在目录加载 `WithExcelUserDaoTest.xls` 数据集文件。

`@ExpectedDataSet` 注解用于加载验证数据集文件。如果没有指明数据集的文件名, 则会在当前测试用例所在的类路径下加载文件名为 `testClassName.methodName-result.xls` 的数据集文件。如实例中的③处将加载 `UserDaoTest.saveUser.result.xls` 数据集文件。

20.5.3 测试实战

使用 TestNG 作为基础测试框架, 结合 Unitils、DbUnit 管理测试数据, 并使用上文编写的 Excel 数据集工厂 (见代码清单 20-19)。从 Excel 数据集文件中获取准备数据及验证数据, 并使用 MySQL 作为测试数据库。下面详细介绍如何应用 Excel 准备数据集及验证数据集来测试 DAO。

在进行 DAO 层的测试之前，先来认识一下需要测试的 UserDaoImpl 用户数据访问类。UserDaoImpl 用户数据访问类中拥有一个获取用户信息和保存注册用户信息的方法，其代码如代码清单 20-23 所示。

代码清单 20-23 UserDaoImpl

```
import java.util.List;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.springframework.orm.hibernate4.HibernateTemplate;
import com.smart.dao.UserDao;
import com.smart.domain.User;

public class UserDaoImpl implements UserDao {

    //通过用户名获取用户信息
    public User findUserByUserName(String userName) {
        String hql = " from User u where u.userName=?";
        List<User> users = getHibernateTemplate().find(hql, userName);
        if (users != null && users.size() > 0)
            return users.get(0);
        else
            return null;
    }

    //保存用户信息
    public void save(User user) {
        getHibernateTemplate().saveOrUpdate(user);
    }
    ...
}
```

在认识了需要测试的 UserDaoImpl 用户数据访问类之后，还需要认识一下用于表示用户领域的对象 User，在演示测试保存用户信息及获取用户信息时需要用到此领域对象，其代码如代码清单 20-24 所示。

代码清单 20-24 User

```
import javax.persistence.Column;
import javax.persistence.Entity;
...
@Entity
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Table(name = "t_user")
public class User implements Serializable{
    @Id
    @Column(name = "user_id")
    protected int userId;

    @Column(name = "user_name")
    protected String userName;

    protected String password;

    @Column(name = "last_visit")
    protected Date lastVisit;
```



```

@Column(name = "last_ip")
protected String lastIp;

@Column(name = "credit")
private int credit;

...
}

```

用户登录日志领域对象 LoginLog 与用户领域对象 Hibernate 的注解配置一致，这里就不再列出，读者可以参考本书配套网盘中的实例代码。在实例测试中，直接使用 Hibernate 进行持久化操作，所以还需要对 Hibernate 进行相应配置，详细的配置如代码清单 20-25 所示。

代码清单 20-25 hibernate.cfg.xml

```

<?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">
<hibernate-configuration>
  <session-factory>
    <!--① SQL方言, 这里设定的是MySQL -->
    <property name="dialect"> org.hibernate.dialect.MySQL5Dialect</property>
    <!--② 数据库连接配置 -->
    <property name="hibernate.connection.driver_class">
      com.mysql.jdbc.Driver
    </property>
    <property name="hibernate.connection.url">
      jdbc:mysql://localhost:3306/sampled?useUnicode=true&characterEncoding=UTF-8
    </property>
    <!--设置连接数据库的用户名-->
    <property name="hibernate.connection.username">root</property>
    <!--设置连接数据库的密码-->
    <property name="hibernate.connection.password">root</property>
    <!--③ 设置显示SQL语句方便调试-->
    <property name="hibernate.show_sql">true</property>
    <!--④ 配置映射 -->
    <property name="configurationClass">
      org.hibernate.cfg.Configuration
    </property>
    <mapping class="com.smart.domain.User" />
    <mapping class="com.smart.domain.LoginLog" />
  </session-factory>
</hibernate-configuration>

```

选用 MySQL 作为测试数据库，在①处，配置 MySQL 的 SQL 方言 MySQL5Dialect。在②处，对连接数据库驱动及数据库连接进行相应的配置。为了方便测试调试，在③处设置显示 Hibernate 生成的 SQL 语句。在④处启用 Hibernate 的注解功能，并配置相应的领域对象，如实例中的 User 和 LoginLog。将配置好的 hibernate.cfg.xml 放在 src 目录下。

1. 配置 Unitils 测试环境

要在单元测试中使用 Unitils，首先需要在测试目录 src\test\resources 中创建一个

项目级 `unitils.properties` 配置文件。实例中 `unitils.properties` 文件的详细配置如代码清单 20-26 所示。

代码清单 20-26 `unitils.properties`

```
#① 启用Unitils所需模块
unitils.modules=database,dbunit,hibernate,spring

#自定义扩展模块，并解决DbUnit对MySQL处理的Bug
unitils.module.dbunit.className=org.dbunit.MySqlDbUnitModule

#② 配置MySQL数据库连接
database.driverClassName=com.mysql.jdbc.Driver
database.url=jdbc:mysql://localhost:3306/sampled?useUnicode=true&characterEncoding=UTF-8
database.dialect =mysql
database.userName=root
database.password=123456
database.schemaNames=sampled

#③ 配置数据库维护策略
updateDataBaseSchema.enabled=true

#④ 配置数据库表创建策略
dbMaintainer.autoCreateExecutedScriptsTable=true
dbMaintainer.script.locations= D:/masterSpring/code/chapter20/schema

#⑤ 数据集加载策略
#CleanInsertLoadStrategy:先删除dateSet中有关表的数据，然后再插入数据
#InsertLoadStrategy: 只插入数据
#RefreshLoadStrategy:有同样key的数据更新，没有的插入
#UpdateLoadStrategy:有同样key的数据更新，没有的不作任何操作
#DbUnitModule.DataSet.loadStrategy.default=org.unitils.dbunit.datasetloadstrategy.
InsertLoadStrategy

#⑥ 配置数据集工厂
DbUnitModule.DataSet.factory.default=sample.unitils.dataset.excel.
MultiSchemaXlsDataSetFactory
DbUnitModule.ExpectedDataSet.factory.default=sample.unitils.dataset.excel.
MultiSchemaXlsDataSetFactory

#⑦ 配置事务策略
#commit是单元测试方法过后提交事务
#rollback是回滚事务
#disabled是没有事务，默认情况下，事务管理是disabled
DatabaseModule.Transactionnal.value.default=disabled

#⑧ 配置数据集结构模式XSD的生成路径
dataSetStructureGenerator.xsd.dirName=resources/xsd
```

我们知道，`unitils.properties` 文件中配置的属性是整个项目级别的，整个项目都可以使用这些全局的属性配置。特定用户使用的属性可以设置在 `unitils-local.properties` 文件中，如 `user`、`password` 和 `schema`，这样每个开发者就可以使用自定义的测试数据库的 `schema`，而且彼此之间不会产生影响，实例的详细配置如代码清单 20-27 所示。

代码清单 20-27 unitils-local.properties

```
...
database.userName=root
database.password=123456
database.schemaNames=sampled
...
```

如果用户分别在 `unitils.properties` 和 `unitils-local.properties` 文件中对相同属性配置不同值, 则将会以 `unitils-local.properties` 配置内容为主。如在 `unitils.properties` 配置文件中也配置了 `database.schemaNames=xxx`, 则测试时启用的是用户自定义配置中的值 `database.schemaNames=sampled`。

2. 配置数据集加载策略

默认的数据集加载机制采用先清理后插入的策略, 也就是数据在被写入数据库的时候会先删除数据集中有对应表的数据, 然后将数据集中的数据写入数据库。这个加载策略是可配置的, 可以通过修改 `DbUnitModule.DataSet.loadStrategy.default` 的属性值来改变加载策略。如代码清单 20-26 中⑤处的配置策略, 这时加载策略就由先清理后插入变成了插入, 数据已经存在于表中将不会被删除, 测试数据只进行插入操作。可选的加载策略列表如下。

- ☐ `CleanInsertLoadStrategy`: 先删除 `dateSet` 中有关表的数据, 然后再插入数据。
- ☐ `InsertLoadStrategy`: 只插入数据。
- ☐ `RefreshLoadStrategy`: 有同样 `key` 的数据更新, 没有的插入。
- ☐ `UpdateLoadStrategy`: 有同样 `key` 的数据更新, 没有的不作任何操作。

3. 配置事务策略

在测试 DAO 的时候都会填写一些测试数据, 每个测试在运行时都会修改或者更新数据, 当下一个测试运行的时候, 都需要将数据恢复到原有状态。如果使用的是 `Hibernate` 或者 `JPA`, 则需要每个测试都运行在事务中, 以保证系统的正常工作。在默认情况下, 事务管理是 `disabled` 的, 可以通过修改 `DatabaseModule.Transaction.value.default` 配置选项, 如代码清单 20-26 中⑧处的配置策略, 这时每个测试都将执行 `commit` 操作。其他可选的配置属性值有 `rollback` 和 `disabled`。

4. 准备测试数据库及测试数据

配置好 `Unitils` 基本配置、加载模块、数据集创建策略、事务策略之后, 我们就着手开始测试数据库及测试数据准备工作, 首先创建测试数据库。

5. 创建测试数据库

在本章节模块 `D:\masterSpring\code\chapter20\` 目录下创建一个 `dbscripts` 文件夹, 且这个文件夹必须与在 `unitils.properties` 文件中 `dbMaintainer.script.locations` 配置项指定的位置一致, 如代码清单 20-26 中的④处所示。

在这个文件夹中创建一个数据库创建脚本文件 `001_sampledb.sql`, 里面包含创建用户表 `t_user` 及登录日志表 `t_login_log`, 详细的脚本如代码清单 20-28 所示。

代码清单 20-28 001_create_sampledb.sql

```

CREATE TABLE t_user (
    user_id INT generated by default as identity (start with 100),
    user_name VARCHAR(30), credit INT,
    credit INT,
    password VARCHAR(32), last_visit timestamp,
    last_ip VARCHAR(23), primary key (user_id));

CREATE TABLE t_login_log (
    login_log_id INT generated by default as identity (start with 1),
    user_id INT,
    ip VARCHAR(23),
    login_datetime timestamp,
    primary key (login_log_id));

```

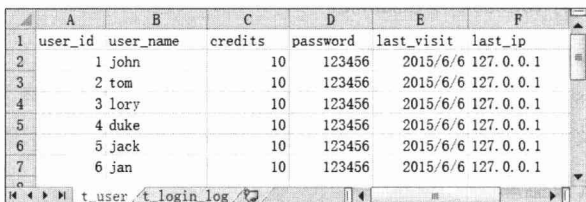
细心的读者可能会发现这个数据库创建脚本文件名好像存在一定的规则，是的，这个数据库创建脚本文件名命名需要按以下规则命名：版本号+“_”+“自定义名称”+“.sql”。

6. 连接到测试数据库

在测试 DAO 时，读者要有个疑问：测试数据库用到的数据源来自哪里？怎么让我们测试的 DAO 类来使用我们的数据源？在执行测试实例的时候，Unitils 会根据我们定义的数据库连接属性来创建一个数据源实例连接到测试数据库。随后的 DAO 测试会重用相同的数据源实例。建立连接的细节定义在 unitils.properties 配置文件中，如代码清单 20-26 中的②处所示。

7. 用 Excel 准备测试数据

准备好测试数据库之后，剩下的工作就是用 Excel 来准备测试数据及验证数据。回顾一下我们要测试的 UserDaoImpl 类（见代码清单 20-23），需要对其中的获取用户信息方法 findUserByUserName()及保存用户信息方法 saveUser()进行测试，所以至少需要准备 3 个 Excel 数据集文件，分别是供查询用户用的数据集文件 UserDao.Users.xls、供保存用户信息用的数据集文件 UserDao.SaveUser.xls 及供保存用户信息用的验证数据集文件 UserDao.ExpectedSaveUser.xls。下面以用户数据集文件 UserDao.Users.xls 为例来进行说明，如图 20-7 所示。



	A	B	C	D	E	F
1	user_id	user_name	credits	password	last_visit	last_ip
2	1	john	10	123456	2015/6/6	127.0.0.1
3	2	tom	10	123456	2015/6/6	127.0.0.1
4	3	lory	10	123456	2015/6/6	127.0.0.1
5	4	duke	10	123456	2015/6/6	127.0.0.1
6	5	jack	10	123456	2015/6/6	127.0.0.1
7	6	jan	10	123456	2015/6/6	127.0.0.1

图 20-7 UserDao.Users.xls 查询用户数据集

其中，Excel 表格底部 Sheet 页 t_user 表示数据库对应的表名称。表格第一行表示数据库中 t_user 表对应的字段名称，从表格第二行开始表示测试的模拟数据。一个数据集文件可以对应多张表，一个 Sheet 就对应一张表。把创建好的数据集文件放到与测试类

相同的目录中，如实例中的 UserDaoTest 类位于 com.smart.dao 包中，则数据集文件需要放到当前包中。

8. 编写 UserDaoImpl 的测试用例

完成了 Unitils 环境配置、准备好测试数据库及测试数据之后，就可以开始编写用户 DAO 单元测试类了。下面为用户数据访问 UserDaoImpl 编写测试用例类，如代码清单 20-29 所示。

代码清单 20-29 UserDaoTest 用户 DAO 测试

```
import org.unitils.core.UnitilsException;
import org.unitils.DbUnit.datasetfactory.DataSetFactory;
import org.unitils.DbUnit.util.MultiSchemaDataSet;
...
@SpringApplicationContext( {"smart-dao.xml" }) //① 初始化Spring容器
public class UserDaoTest extends UnitilsTestNG {

    @SpringBean("jdbcUserDao") //② 从Spring容器中加载DAO
    private UserDao userDao;

    @BeforeClass
    public void init() {

    }

    ...
}
```

在①处，通过 Unitils 提供的 @SpringApplicationContext 注解加载 Spring 配置文件，并初始化 Spring 容器。在②处，通过 @SpringBean 注解从 Spring 容器加载一个用户 DAO 实例。编写好 UserDaoTest 测试基础模型之后，接下来就开始编写查询用户信息的测试方法 findUserByUserName()，如代码清单 20-30 所示。

代码清单 20-30 UserDaoTest.findUserByUserName()测试

```
import org.unitils.core.UnitilsException;
import org.unitils.DbUnit.datasetfactory.DataSetFactory;
import org.unitils.DbUnit.util.MultiSchemaDataSet;
...
public class UserDaoTest extends UnitilsTestNG {

    ...

    @Test //① 标记为测试方法
    @DataSet("Smart.Users.xls") //② 加载准备用户测试数据
    public void findUserByUserName() {
        User user = userDao.findUserByUserName("tony"); //③ 从数据库中加载tony用户
        assertNull("不存在用户名为tony的用户!", user);
        user = userDao.findUserByUserName("jan"); //④ 从数据库中加载jan用户
        assertNotNull("jan用户存在!", user);
        assertEquals("jan", user.getUserName());
        assertEquals("123456", user.getPassword());
        assertEquals(10, user.getCredit());
    }

    ...
}
```

在①处，通过 TestNG 提供的@Test 注解把当前方法标记为测试方法。在②处，通过 Unitils 提供的@DataSet 注解从当前测试类 UserDaoTest.class 所在的目录寻找支持 DbUnit 的数据集文件并进行加载。在执行测试逻辑之前，会把加载的数据集先持久化到测试数据库中（具体加载数据集的策略详见上文“配置数据集加载策略”部分）。实例中采用的是默认加载策略，即先删除测试数据库对应表的数据，再插入数据集内的测试数据。这种策略可以避免不同测试方法加载数据集时的相互干扰。在③处执行查询用户方法时，测试数据库中 t_user 表的数据与图 20-8 中 Smart.Users.xls 的数据一致，因此查询不到“tony”用户信息。在④处，执行查询“jan”用户信息，从测试数据集中可以看出，可以加载到“jan”的详细信息。最后在 IDE 中执行 UserDaoTest.findUserByUserName() 测试方法，按我们的预期通过测试，测试结果如图 20-8 所示。

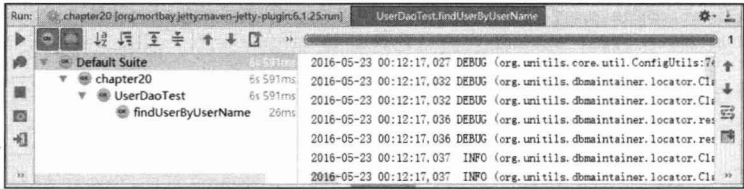


图 20-8 UserDaoTest.findUserByUserName()测试结果

完成了查询用户的测试之后，我们开始着手编写保存用户信息的测试方法，详细的实现代码如代码清单 20-31 所示。

代码清单 20-31 UserDaoTest.saveUser()测试

```
import org.unitils.core.UnitilsException;
import org.unitils.DbUnit.datasetfactory.DataSetFactory;
import org.unitils.DbUnit.util.MultiSchemaDataSet;
...
public class UserDaoTest extends UnitilsTestNG {
    ...

    @Test //① 标记为测试方法
    @ExpectedDataSet("UserDao.ExpectedSaveUser.xls") //准备验证数据
    public void saveUser()throws Exception {
        User u = new User();
        u.setUserid(1);
        u.setUserName("tom");
        u.setPassword("123456");
        u.setLastVisit(getDate("2016-03-06 08:00:00","yyyy-MM-dd HH:mm:ss"));
        u.setCredit(30);
        u.setLastIp("127.0.0.1");
        userDao.save(u); //执行用户信息更新操作
    }
    ...
}
```

在①处，通过 TestNG 提供的@Test 注解把当前方法标记为测试方法。在②处，通过 Unitils 提供的@ExpectedDataSet 注解从当前测试类 UserDaoTest.class 所在的目录寻找支持 DbUnit 的验证数据集文件并进行加载，之后验证数据集里的数据和数据库中的数

据是否一致。在 UserDaoTest.saveUser()测试方法中创建一个 User 实例，并设置验证数据集数据，然后执行保存用户操作。最后在 IDE 中执行 UserDaoTest.saveUser()测试方法，执行结果如图 20-9 所示。

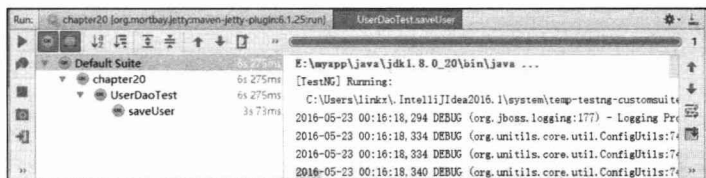


图 20-9 UserDaoTest.saveUser()测试结果 (1)

虽然已经成功地完成了保存用户信息 UserDaoTest.saveUser() 方法测试，但还是存在不足的地方。我们的测试数据通过硬编码方式直接设置在 User 实例中，如果需要更改测试数据，则只能更改测试代码，大大削减了测试的灵活性。如果能直接从 Excel 数据集中获取测试数据，并自动绑定到目标对象，那我们的测试用例就更加完美了。为此，笔者编写了一个获取 Excel 数据集的 Bean 工厂 XlsDataSetBeanFactory，用于自动绑定数据集到测试对象。我们对上面的测试方法进行整改，实现代码如代码清单 20-32 所示。

代码清单 20-32 UserDaoTest.java

```
import org.junitils.core.UnitilsException;
import org.junitils.DbUnit.datasetfactory.DataSetFactory;
import org.junitils.DbUnit.util.MultiSchemaDataSet;
import sample.unitils.dataset.util.XlsDataSetBeanFactory;
...
public class UserDaoTest extends UnitilsTestNG {
    ...

    @Test //① 标记为测试方法
    @ExpectedDataSet("UserDao.ExpectedSaveUser.xls") //准备验证数据
    public void saveUser() throws Exception {

        userDao.clean(); //清理测试数据

        //② 从保存数据集中创建Bean
        User u = XlsDataSetBeanFactory.createBean("UserDao.SaveUser.xls"
                                                , "t_user", User.class);

        userDao.save(u); //③ 执行用户信息更新操作
    }
    ...
}
```

在②处，通过 XlsDataSetBeanFactory.createBean()方法从当前测试类所在目录加载 UserDao.SaveUser.xls 数据集文件。把 UserDao.SaveUser.xls 中名为 t_user 的 Sheet 页中的数据绑定到 User 对象，如果当前 Sheet 页有多条记录，则可以通过 XlsDataSetBeanFactory.createBeans()方法获取用户列表 List<User>。最后在 IDE 中重新执行 UserDaoTest.saveUser()测试方法，执行结果如图 20-10 所示。

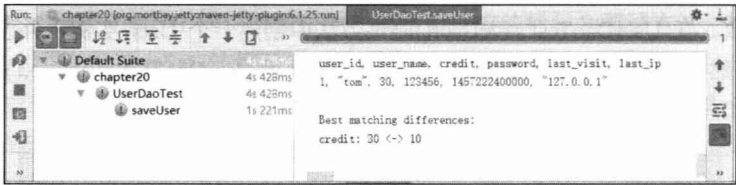


图 20-10 UserDaoTest.saveUser()测试结果（2）

从测试结果可以看出，执行 UserDaoTest.saveUser()测试失败。从右边的失败报告信息可以看出，是由于模拟用户的积分与我们期望的数据不一致造成的，期望用户积分是 30，而我们保存用户的积分是 10。重新对比一下 UserDao.SaveUser.xls 数据集的数据与 UserDao.ExpectedSaveUser.xls 数据集的数据，确实我们准备保存数据集的数据与验证结果的数据不一致。把 UserDao.SaveUser.xls 数据集集中的用户积分更改为 30，最后在 IDE 中重新执行 UserDaoTest.saveUser()测试方法，执行结果如图 20-11 所示。

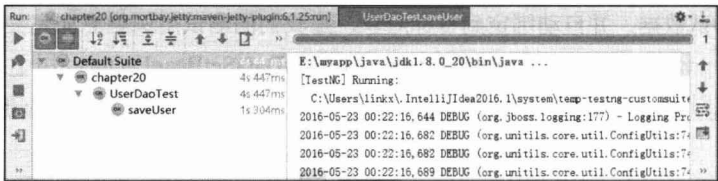


图 20-11 UserDaoTest.saveUser()测试结果（3）

从测试结果可以看出，保存用户通过测试。从上述的测试实战我们已经体验到用 Excel 准备测试数据与验证数据带来的便捷性。至此，我们完成了 DAO 测试的整个过程。对于 XlsDataSetBeanFactory 的具体实现，读者可以查看本章的实例源码，这里就不再详细分析了。代码清单 20-33 是实现的基本骨架。

代码清单 20-33 XlsDataSetBeanFactory

```
import org.dbunit.dataset.Column;
import org.dbunit.dataset.DataSetException;
import org.dbunit.dataset.IDataSet;
import org.dbunit.dataset.ITable;
import org.dbunit.dataset.excel.XlsDataSet;
...
public class XlsDataSetBeanFactory {

    //从Excel数据集文件中创建多个Bean
    public static <T> List<T> createBeans(String file, String tableName,
        Class<T> clazz) throws Exception {
        BeanUtilsBean beanUtils = createBeanUtils();
        List<Map<String, Object>> propsList = createProps(file, tableName);
        List<T> beans = new ArrayList<T>();
        for (Map<String, Object> props : propsList) {
            T bean = clazz.newInstance();
            beanUtils.populate(bean, props);
            beans.add(bean);
        }
        return beans;
    }
}
```



```

    }

    //从Excel数据集文件中创建多个Bean
    public static <T> T createBean(String file, String tableName, Class<T> clazz)
        throws Exception {
        BeanUtilsBean beanUtils = createBeanUtils();
        List<Map<String, Object>> propsList = createProps(file, tableName);
        T bean = clazz.newInstance();
        beanUtils.populate(bean, propsList.get(0));
        return bean;
    }
    ...
}

```

20.6 使用 Unitils 测试 Service 层

在进行服务层的测试之前，先来认识一下需要测试的 UserServiceImpl 服务类。 UserServiceImpl 服务类中有一个处理用户登录的服务方法，其代码如代码清单 20-34 所示。

代码清单 20-34 UserService.java

```

package com.smart.service;
import com.smart.domain.LoginLog;
import com.smart.domain.User;
import com.smart.dao.UserDao;
import com.smart.dao.LoginLogDao;
@Service("userService")
public class UserServiceImpl implements UserService {
    private UserDao userDao;
    private LoginLogDao loginLogDao;

    public void loginSuccess(User user) {
        user.setCredit( 5 + user.getCredit());
        LoginLog loginLog = new LoginLog();
        loginLog.setUserId(user.getUserId());
        loginLog.setIp(user.getLastIp());
        loginLog.setLoginTime(user.getLastVisit());
        userDao.updateLoginInfo(user);
        loginLogDao.insertLoginLog(loginLog);
    }
    ...
}

```

ServiceImpl 需要调用 DAO 层的 UserDao 和 LoginLogDao 以及 User 和 LoginLog 这两个 PO 完成业务逻辑，User 和 LoginLog 分别对应 t_user 和 t_login_log 这两张数据库表。

在用户登录成功后，调用 UserServiceImpl 中的 loginSuccess()方法执行用户登录成功后的业务逻辑。

- ① 登录用户添加 5 个积分 (t_user.credit)。
- ② 将登录用户的最后访问时间 (t_user.last_visit) 和 IP (t_user.last_ip) 更新为当前值。

- ③ 在日志表 (t_login_log) 中为用户添加一条登录日志。

这是一个需要访问数据库并存在数据更改操作的业务方法，它工作在事务环境下。

下面是装配该服务类 Bean 的 Spring 配置文件，如代码清单 20-35 所示。

代码清单 20-35 smart-service.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-4.0.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-4.0.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-4.0.xsd">
    <context:component-scan base-package="com.smart.service"/>
    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
        p:dataSource-ref="dataSource" />
    <tx:annotation-driven />
    <!-- 使用aop/tx命名空间配置事务管理，这里对service包下的服务类方法提供事务-->
    <aop:config>
        <aop:pointcut id="jdbcServiceMethod"
            expression="within(com.smart.service..*)" />
        <aop:advisor pointcut-ref="jdbcServiceMethod" advice-ref="jdbcTxAdvice" />
    </aop:config>
    <tx:advice id="jdbcTxAdvice" transaction-manager="transactionManager">
        <tx:attributes>
            <tx:method name="*" />
        </tx:attributes>
    </tx:advice>
</beans>
```

UserServiceImpl 所关联的 DAO 类和 PO 类都比较简单，这里就不一一列出了，读者可以参考本文配套网盘中的实例代码。

下面为 UserServiceImpl 编写一个简单的测试用例类，此时的目标是让这个基于 Unitils 测试框架的测试类运行起来，并联合 Mockito 框架创建 Dao 模拟对象。首先编写测试 UserService#findUserByUserName() 方法的测试用例，如代码清单 20-36 所示。

代码清单 20-36 UserServiceTest.java

```

package com.smart.service;

import org.junitils.UnitilsTestNG;
import org.junitils.spring.annotation.SpringApplicationContext;
import org.springframework.test.util.ReflectionTestUtils;
import org.junitils.spring.annotation.SpringBean;
import org.testng.annotations.*;
import com.smart.domain.User;
import java.util.Date;

...
@SpringApplicationContext({"smart-service.xml", "smart-dao.xml"}) //①加载Spring配置文件
public class UserServiceTest extends UnitilsTestNG{

    private UserDao userDao; //② 声明用户Dao
    private LoginLogDao loginLogDao;

    @BeforeClass //③ 创建Dao模拟对象
    public void init(){
        userDao = mock(UserDao.class);
        loginLogDao = mock(LoginLogDao.class);
    }

    @Test //④ 设置为TestNG测试方法
    public void findUserByUserName() {

        //④-1 模拟测试数据
        User user = new User();
        user.setUserName("tom");
        user.setPassword("1234");
        user.setCredit(100);
        doReturn(user).when(userDao).findUserByUserName("tom");

        //④-2 实例化用户服务实例类
        UserServiceImpl userService = new UserServiceImpl();

        //④-3 通过Spring测试框架提供的工具类为目标对象私有属性赋值
        ReflectionTestUtils.setField(userService, "userDao", userDao);

        //④-4 验证服务方法
        User u = userService.findUserByUserName("tom");
        assertNotNull(u);
        assertEquals(u.getUserName(), user.getUserName());

        //④-5 验证交互行为
        verify(userDao, times(1)).findUserByUserName("tom");
    }
}

```

在这里，我们让 `UserServiceTest` 直接继承于 `Unitils` 提供的 `UnitilsTestNG` 的抽象测试类，该抽象测试类的作用是让 `Unitils` 测试框架可以在 `TestNG` 测试框架的基础上运行起来。在①处标注了一个类级的 `@SpringApplicationContext` 注解，这里 `Unitils` 将从类路


```
private UserService userService;

@Test
@DataSet("Smart.SaveUsers.xls")//③ 准备验证数据
public void loginSuccess() {
    User user = userService.findUserByUserName("tom"); //④-1 加载tom用户信息
    Date now = new Date();
    user.setLastVisit(now); //④-2 设置当前登录时间
    userService.loginSuccess(user); //④-3 user登录成功,更新其积分及添加日志
    User u = userService.findUserByUserName("tom");
    assertThat(u.getCredit(),is(105)); //⑤ 验证登录成功后用户积分是否是105分
}
}
```

在①处通过加载 Unitils 的 `@SpringApplicationContext` 注解加载 Spring 配置文件,并初始化 Spring 容器。在②处通过 `@SpringBean` 注解从 Spring 容器中获取 `UserService` 实例。在③处通过 `@DataSet` 注解从当前测试用例所在的类路径中加载 `Smart.SaveUsers.xls` 数据集,并将数据集中的数据保存到测试数据库相应的表中。从上面的数据集中可以看出,我们为 `t_user` 表准备了两条用户信息测试数据。在④-1 处从测试数据库中获取“tom”用户信息,模拟当前登录的用户。在④-2 处设置当前“tom”用户的登录时间。在④-3 处调用 `UserService#loginSuccess()` 方法,更新“tom”用户积分,并持久化到测试数据库中。在⑤处,验证“tom”用户当前积分是否是 105 分。至此,完成测试用例的编写,最后在 IDE 中执行 `UserServiceTest` 测试用例,测试结果如图 20-14 所示。

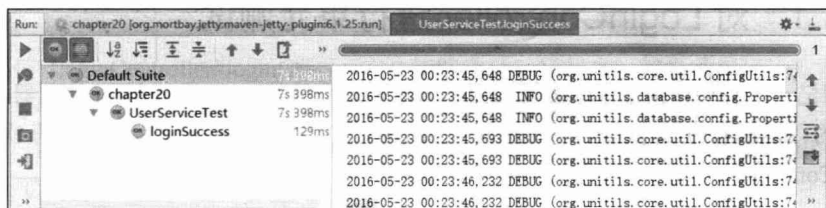


图 20-14 UserServiceTest.loginSuccess()测试结果

从运行结果可以看出,我们已经成功地对 `UserService#loginSuccess()` 执行了单元测试。重复执行当前单元测试,测试结果仍然通过。细心的读者可能会有疑问:没有 `UserService#loginSuccess()` 测试方法实施事务回滚,执行多次之后,“tom”用户的积分不应该是 105 分,那为何测试还是通过呢?这是因为 Unitils 帮我们维护测试数据库中的数据状态。Unitils 这个强大的魔力归根于 Unitils 强大的数据集更新策略。至此,我们成功地完成了 `UserService` 单元测试。从上面为用户服务 `UserService` 编写两个测试方法可以看出,对 Service 层的测试,既可以采用 `TestNG+Unitils+Mockito` 组合,运用 Mockito 强大的模块能力,完成 Service 层的独立性测试;也可以采用 `TestNG+Unitils+DbUnit` 组合,运用 DbUnit 强大的数据库维护能力,完成 Service 层+DAO 层的集成测试。

20.7 测试 Web 层

Spring 在 `org.springframework.mock` 包中为一些依赖于容器的接口提供了模拟类，这样用户就可以在不启动容器的情况下执行单元测试，提高单元测试的运行效率。Spring 提供的模拟类分属于以下 3 个包中。

- ❑ `org.springframework.mock.jndi`: 为 JNDI SPI 接口提供的模拟类，以便可以脱离 Java EE 容器进行测试。
- ❑ `org.springframework.mock.web`: 为 Servlet API 接口（如 `HttpServletRequest`、`ServletContext` 等）提供的模拟类，以便可以脱离 Servlet 容器进行测试。
- ❑ `org.springframework.mock.web.portlet`: 为 Portlet API 接口（如 `PortletRequest`、`PortalContext` 等）提供的模拟类，以便可以脱离 Portlet 容器进行测试。

Spring 将这些模拟类单独打包成 `spring-mock.jar`，读者可以在 Spring 发布包的 `dist` 目录下找到这个类包。

在正常情况下，我们无法获得那些依赖于容器创建的接口实例，如 `HttpServletRequest`、`ServletContext` 等。模拟类实现了这些接口，它在一定程度上模拟了这些接口的输入/输出功能，借助这些模拟类的支持，我们就可以轻松地测试那些依赖于容器的功能模块。

20.7.1 对 LoginController 进行单元测试

回忆一下我们在第 2 章中编写的用于处理用户登录的 `LoginController` 控制器，处理请求的 `handle()` 方法依赖于 Servlet API 接口。为了方便阅读，我们再次列出 `LoginController` 的代码（对原代码进行了细微调整），如代码清单 20-38 所示。

代码清单 20-38 LoginController: 用户登录控制器

```
//①标注为一个Spring MVC的Controller
@Controller
public class LoginController{

    private UserService userService;

    //② 负责处理/index.html的请求
    @RequestMapping(value = "/index.html")
    public String loginPage(){
        return "login";
    }

    //③ 负责处理/loginCheck.html的请求
    @RequestMapping(value = "/loginCheck.html")
    public ModelAndView loginCheck(HttpServletRequest request, LoginCommand
loginCommand){
        boolean isValidUser =
            userService.hasMatchUser(loginCommand.getUserName(),
                                    loginCommand.getPassword());
```



```

    if (!isValidUser) {
        return new ModelAndView("login", "error", "用户名或密码错误。");
    } else {
        User user = userService.findUserByUserName(loginCommand
            .getUserName());
        user.setLastIp(request.getLocalAddr());
        user.setLastVisit(new Date());
        userService.loginSuccess(user);
        request.getSession().setAttribute("user", user);
        return new ModelAndView("main");
    }
}
...
}

```

现在需要对 `LoginController#loginCheck()` 方法进行单元测试，验证以下几种情况的正确性：

- (1) `/loginCheck.html` 的请求路径是否能正确映射到 `LoginController#loginCheck()`。
- (2) 在向 `Request` 添加用户名 `userName` 参数及密码参数，并设置不存在的用户名及密码时，返回 `ModelAndView("login", "error", "用户名或密码错误。")` 对象。
- (3) 在向 `Request` 添加用户名 `userName` 参数及密码参数，并设置正确的用户名及密码时，返回 `ModelAndView("main")` 对象。
- (4) 当登录成功后，检查当前的 `Session` 属性中是否存在 “user”，并验证 `user` 值的正确性。

要使 `LoginController#loginCheck()` 方法能够成功运行起来，就必须保证该方法所依赖的对象事先准备好。我们通过以下方案解决这一问题：

- ❑ 通过 `Unitils` 从 `Spring` 容器中加载 `RequestMappingHandlerAdapter` 和 `LoginController` 实例。
- ❑ 通过 `Spring` 提供的 `Servlet API` 模拟类创建 `HttpServletRequest` 和 `HttpServletResponse` 实例。

20.7.2 使用 Spring Servlet API 模拟对象

下面联合使用 `Spring` 模拟类及 `Unitils` 对 `LoginController` 进行单元测试，具体实现如代码清单 20-39 所示。

代码清单 20-39 `LoginControllerTest`

```

package com.smart.web;
import org.unitils.UnitilsTestNG;
import org.unitils.spring.annotation.SpringApplicationContext;
import org.unitils.spring.annotation.SpringBeanByType;
import static org.hamcrest.Matchers.*;
import com.smart.domain.User;

@SpringApplicationContext({"classpath:applicationContext.xml",

```

```

        "/web/smart-servlet.xml"))
public class LoginControllerTest extends UnitilsTestNG{

    //① 从Spring容器中加载RequestMappingHandlerAdapter
    @SpringBeanByType
    private RequestMappingHandlerAdapter handlerAdapter;

    //② 从Spring容器中加载LoginController
    @SpringBeanByType
    private LoginController controller;

    //③ 声明Request与Response模拟对象
    private MockHttpServletRequest request;
    private MockHttpServletResponse response;

    //④ 执行测试前先初始化模拟对象
    @BeforeClass
    public void before() {
        request = new MockHttpServletRequest();
        request.setCharacterEncoding("UTF-8");
        response = new MockHttpServletResponse();
    }

    //⑤ 测试LoginController#loginCheck()方法
    @Test
    public void loginCheck() throws Exception{

        request.setRequestURI("/loginCheck.html");
        request.addParameter("userName", "tom"); //⑥ 设置请求URL及参数
        request.addParameter("password", "123456");

        //⑦ 向控制发起请求
        ModelAndView mav = controller.loginCheck(request);
        User user = (User) request.getSession().getAttribute("user");

        assertNotNull(mav);
        assertEquals(mav.getViewName(), "main");
        assertNotNull(user);
        assertEquals(user.getUserName(), "tom"); //⑧ 验证返回结果
        assertTrue(user.getCredit() > 5);
    }
}

```

在①处和②处，使用 Unitils 提供的 `@SpringBeanByType` 注解从 Spring 容器中加载 `RequestMappingHandlerAdapter` 和 `LoginController` 实例。在③处，声明 Spring 提供的 Servlet API 模拟类 `MockHttpServletRequest` 及 `MockHttpServletResponse`，并在测试初始化方法中进行实例化。在⑥处，在模拟类 `MockHttpServletRequest` 中设置请求 URL 及参数。在⑦处，通过 Spring 提供的注解方法处理适配器向 `LoginController#loginCheck()` 发起请求。在⑧处，通过 TestNG 提供的断言及 Hamcrest 提供的匹配方法验证返回的结果。注意，当运行 `LoginControllerTest` 测试用例时，并不需要启动 Servlet 容器，用户可以在 IDE 环境下运行该测试用例。

20.7.3 使用 Spring RestTemplate 测试

上文通过 Spring 提供的模拟类并联合 Unitils 测试框架，顺利地完成了 LoginController 的单元测试。下面尝试应用 Spring 提供的 RestTemplate 并联合 Unitils 框架对登录模块的 Web 层进行集成测试。

RestTemplate 是用来在客户端访问 Web 服务的类。和其他 Spring 中的模板类（如 JdbcTemplate、JmsTemplate）很相似，还可以通过提供回调方法和配置 HttpMessageConverter 类来自定义该模板。客户端的操作可以完全使用 RestTemplate 和 HttpMessageConverter 类来执行。要使用 Spring RestTemplate，首先需要在 Spring 应用上下文中进行相应的配置，具体配置如代码清单 20-40 所示。

代码清单 20-40 smart-servlet.xml

```
...
<bean id="restTemplate" class="org.springframework.web.client.RestTemplate">
    <property name="messageConverters">
        <list>
            <bean id="stringHttpMessageConverter"
                class="org.springframework.http.converter.StringHttp
                MessageConverter" />
            <bean id="formHttpMessageConverter"
                class="org.springframework.http.converter.FormHttp
                MessageConverter" />
        </list>
    </property>
</bean>
...
```

发送给 RestTemplate 方法的对象以及从 RestTemplate 方法返回的对象需要使用 HttpMessageConverter 接口转换成 HTTP 消息，因此，我们在上面配置两个消息转换器 StringHttpMessageConverter 和 FormHttpMessageConverter。配置好 RestTemplate 模板操作类之后，就可以开始编写登录控制器 LoginController 测试用例，具体配置如代码清单 20-41 所示。

代码清单 20-41 LoginControllerTest

```
package com.smart.web;
import org.unitils.UnitilsTestNG;
import org.unitils.spring.annotation.SpringApplicationContext;
import org.unitils.spring.annotation.SpringBeanByType;
import static org.hamcrest.Matchers.*;
import com.smart.domain.User;

@SpringApplicationContext({"classpath:applicationContext.xml",
    "/web/smart-servlet.xml"})
public class LoginControllerTest extends UnitilsTestNG{

    //① 从Spring容器中加载restTemplate
    @SpringBeanByType
    private RestTemplate restTemplate;
```

```
//② 从Spring容器中加载LoginController
@SpringBeanByType
private LoginController controller;

//③ 测试LoginController#loginCheck()方法
@Test
public void loginCheck() throws Exception{

    //③-1 构造请求提交参数
    MultiValueMap<String, String> map = new LinkedMultiValueMap<String, String>();
    map.add("userName", "tom");
    map.add("password", "123456");

    //③-2 发送客户访问请求
    result = restTemplate.postForObject(
        "http://localhost:8000/bbs/loginCheck.html", map, String.class);

    //③-3 验证响应结果
    assertNotNull(result);
    assertThat(result, containsString("tom,欢迎您进入小春论坛"));
}
```

在①处和②处，使用 `Unitils` 提供的 `@SpringBeanByType` 注解从 `Spring` 容器中加载 `RestTemplate` 和 `LoginController` 实例。在③-1 处，构造一个提交请求的参数列表，如实例中设置的用于登录的用户名及密码。在③-2 处，使用 `RestTemplate` 模板操作类提供的 `postForObject()` 方法发送访问请求。最后在③-3 处，对响应返回的结果进行验证。在 IDE 工具中，在启动 Web 容器之后，运行 `LoginControllerTest` 测试用例，测试结果如图 20-15 所示。

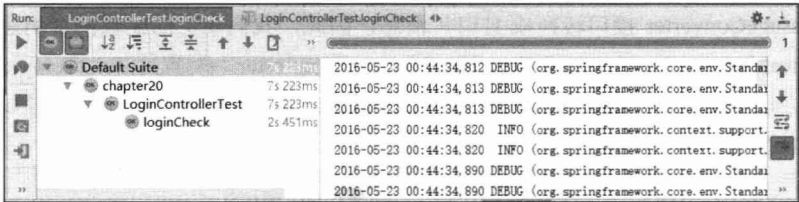


图 20-15 LoginControllerTest.loginCheck()测试结果

20.8 小结

本章讲述了单元测试所需的相关知识，简要分析了目前普遍存在的对单元测试的一些误解、误区及困境。`TestNG` 测试框架是必须掌握的基础内容，我们对 `TestNG` 进行了提纲挈领的学习，并着重学习了 `TestNG` 提供的分组测试、参数测试、依赖测试、数据驱动等特性。

在单元测试中，应该尽量在不依赖外部模块的情况下，重点测试模块内程序逻辑的

正确性。这时，可以通过 Mockito 为测试目标类所依赖的外部模块接口创建模拟对象，通过录制、回放及验证的方式使用模拟对象。通过 Mockito，我们达到了两个看似相对立的目标：一是使测试用例不依赖外部模块；二是使用到外部模块类的目标对象可以正常工作。

Spring 建议用户不要在单元测试时使用 Spring 容器，应该在集成测试时再使用 Spring 容器。手工创建测试夹具或者手工装配测试夹具的工作是单调乏味、没有创意的。通过使用 Unitils，用户就可以享受测试夹具自动装配的好处，将精力集中到目标类逻辑测试编写的工作上。

应该说大部分 Java 应用都是 Web 应用，而大部分 Java Web 应用都是数据库相关的应用，对数据库应用进行测试经常要考虑数据准备、数据库现场恢复、灵活访问数据以验证数据操作正确性等问题。这些问题如果没有一个很好的支持工具，则将给编写测试用例造成挑战。幸好 Unitils 框架为我们搭建好满足这些需求的测试平台，用户仅需在此基础上结合相关框架如 Spring、Hibernate、DbUnit 等就可以满足各种测试场景需要。

为了提高测试 DAO 层的效率，结合 Unitils、DbUnit 框架，编写一个支持 Excel 格式的数据集工厂类，实现使用 Excel 准备测试所需的数据及验证数据，从而大大减少测试 DAO 层的工作量。

对 Service 层的测试，既可以采用 TestNG +Unitils+Mockito 组合，运用 Mockito 强大的模块能力，完成 Service 层独立性测试，也可以采用 TestNG +Unitils+DbUnit 组合，运用 DbUnit 强大的数据库维护能力，完成 Service 层+DAO 层的集成测试。

对 Web 层的测试，我们既可以采用 TestNG +Unitils+Spring Mock 组合，运用 Spring Mock 模拟依赖容器的接口实例，如 HttpServletRequest、ServletContext 等，完成 Web 层中控制器的独立性测试；也可以采用 TestNG +Unitils+Spring RestTemplate，完成 Web 层的集成测试。

《Spring 3.x 企业应用开发实战》读者评价

2012 年 2 月出版

○当当金卡会员 Doray

一直对国内的技术书籍抱有偏见，所以我经常购买外国人写的技术书籍，决定购买这本书时纠结了很久。看完这本书后，我才发现国内的技术书籍也有相当高的水平。

○当当金卡会员 轻声曦语

相当有功底的一本书，以通俗易懂的语言剖析了 Spring 的核心功能。本书不仅有实例代码告诉读者怎么使用 Spring 完成代码，还将 Spring 的构建思想表述出来，让人知其然，亦知其所以然。对我的帮助非常大。

○当当金卡会员 foreveriuu

我越来越喜欢原创作品了，尤其是这种厚积薄发，能够从使用到架构、从相关技术到实际问题的书。

○当当钻石会员 为了找到更好的

这本书可以说很适合初学者，对于熟悉 Spring 的人来说，也可以发现自己想要的。一直以为自己了解 Spring，看完这本书我才明白 Spring 为什么这么火。

○京东金牌会员 边走边忘

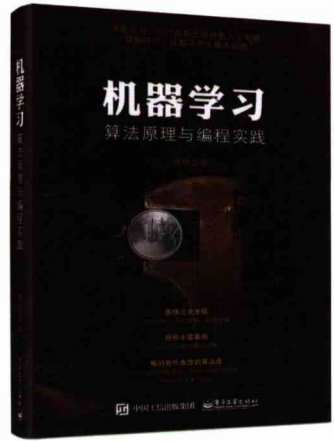
这本书对于理解 Spring 有较大的帮助。作者虽是“程序猿”，但文笔确实不凡，读起来很舒服！

○京东铜牌会员 lijied

经典，值得购买，幽默，有趣，深入浅出。

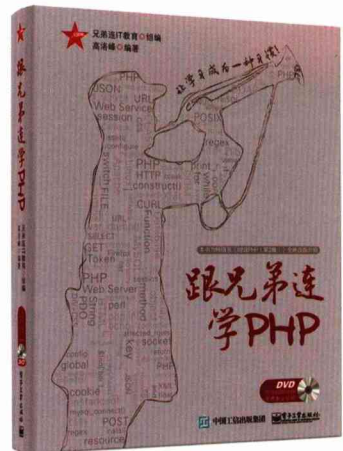
○豆瓣 空气+

无论是从技术的广度还是深度来看，该书都已做到极致，将 Spring 的相关思想、原理、细节描述得深入浅出、淋漓尽致，如最重要的 IoC、复杂抽象的 AOP、很多的实战经验、完整的 Spring MVC 描述，等等，不一而足。该书我仔细读了三遍，作者对 Spring 的相关原理和本质理解得相当深入和透彻。



ISBN : 978-7-121-27367-4

2015年10月出版



ISBN : 978-7-121-27801-3

2015年12月出版

精通Spring 4.x

企业应用开发实战

本书是在《精通Spring 3.x——企业应用开发详解》的基础上，历时一年的重大调整改版而成的，延续了上一版本“追求深度，注重原理，不停留在技术表面”的写作风格，力求使读者在熟练使用Spring的各项功能的同时透彻理解Spring的内部实现，真正做到知其然并知其所以然。此外，本书重点突出了“实战性”的主题，力求使全书内容体现“从实际项目中来，到实际项目中去”的写作原则。

本书核心内容：

第1篇为基础篇，包括第1~3章，讲解Spring概述性知识，以便读者快速建立对Spring的整体认识，能够使用Spring快速开发一个简单的项目，而更多深入的知识则在后续篇章中展开。

第2篇为核心篇，包括第4~9章，讲解Spring的IoC、AOP及SpEL的知识，这些知识是Spring的核心，也是Spring所有衍生服务及功能的基石。

第3篇为数据篇，包括第10~14章，讲解Spring的各种数据访问技术及事务管理的内容，对事务管理的实现机制和各种疑难问题进行剖析。

第4篇为应用篇，包括第15~18章，讲解数据缓存、任务调度、Web开发的内容，对于企业应用开发来说，数据缓存及任务调度是两个无法回避的问题，需要重点学习和掌握；此外，本篇还精心设计了一个实战案例，包含需求分析、数据库设计、项目开发、代码测试、应用部署的整体过程，让读者在项目的实战中整体串接Spring的知识点。

第5篇为提高篇，包括第19章和第20章，讲解Spring OXM及单元测试的内容。

全书工程代码放在本书配套网盘中，请读者下载网盘内容阅读，本书网盘下载地址如下。

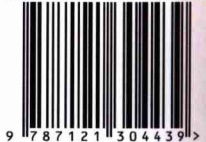
百度云盘：<http://pan.baidu.com/s/1boCI3d1>。



责任编辑：李 冰

封面设计：朝天世纪

ISBN 978-7-121-30443-9



9 787121 304439 >

定价：128.00元