

DefaultPointcutAdvisor 配置一个流程切面来完成这一需求，如代码清单 7-35 所示。

代码清单 7-35 配置控制流程切面

```
<bean id="greetingAdvice" class="com.smart.advisor.GreetingBeforeAdvice" />
<bean id="waiterTarget" class="com.smart.advisor.Waiter"/>
<bean id="controlFlowPointcut"
class="org.springframework.aop.support.ControlFlowPointcut">
    <constructor-arg type="java.lang.Class"
        value="com.smart.advisor.WaiterDelegate" /> ① ← 指定流程切点的类
    <constructor-arg type="java.lang.String"
        value="service" /> ② ← 指定流程切点的方法
</bean>
<bean id="controlFlowAdvisor"
class="org.springframework.aop.support.DefaultPointcutAdvisor"
    p:pointcut-ref="controlFlowPointcut"
    p:advice-ref="greetingAdvice" />

<bean id="waiter3" class="org.springframework.aop.framework.ProxyFactoryBean"
    p:interceptorNames="controlFlowAdvisor"
    p:target-ref="waiterTarget"
    p:proxyTargetClass="true"/>
```

ControlFlowPointcut 有两个构造函数，分别是 ControlFlowPointcut(Class clazz)和 ControlFlowPointcut(Class clazz, String methodName)。第一个构造函数指定一个类作为流程切点；而第二个构造函数指定一个类和某一个方法作为流程切点。

在这里，指定 com.smart.advisor.WaiterDelegate#service()方法作为切点，表示所有通过该方法直接或间接发起的调用匹配切点。说到这里，可能还有一些模糊的地方，下面通过测试代码观察流程切面的运行效果，如代码清单 7-36 所示。

代码清单 7-36 流程切面测试

```
String configPath = "com/smart/advisor/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter waiter = (Waiter) ctx.getBean("waiter3");
WaiterDelegate wd = new WaiterDelegate();
wd.setWaiter(waiter);
waiter.serveTo("Peter");
waiter.greetTo("Peter");
wd.service("Peter");
```

运行上面的代码，在控制台上输出以下信息：

```
waiter serving Peter... ① ← 对应 waiter.serveTo("Peter")
waiter greet to Peter... ② ← 对应 waiter.greetTo("Peter")
③ ← 对应 wd.service("Peter")
com.smart.advisor.Waiter.greetTo
How are you! Mr.Peter.
waiter greet to Peter...
com.smart.advisor.Waiter.serveTo
How are you! Mr.Peter.
waiter serving Peter...
```

①处和②处的信息是直接通过 waiter 调用 serveTo()和 greetTo()方法的输出，此时增强没有起作用；③处是通过 WaiterDelegate#service()调用 Waiter 的 serveTo()和 greetTo()方法的输出，这时发现 Waiter 的两个方法都织入了增强。

流程切面和动态切面从某种程度上说可以算是一类切面，因为二者都需要在运行期判断动态的环境。对于流程切面来说，代理对象在每次调用目标类方法时，都需要判断方法调用堆栈中是否有满足流程切点要求的方法。因此，和动态切面一样，流程切面对性能的影响也很大。在 JVM 1.4 上，流程切点通常比别的切点要慢 5 倍，在 JVM 1.3 上要慢 10 倍。

7.4.7 复合切点切面

在前面的例子中定义的切面仅有一个切点，有时，一个切点可能难以描述目标连接点的信息。比如在前面流程切面的例子中，假设我们希望由 `WaiterDelegate# service()` 发起调用且被调用的方法是 `Waiter#greetTo()` 时才织入增强，这个切点就是复合切点，因为它由两个单独的切点共同确定，第一个切点是代码清单 7-35 所定义的流程切点，而另一个切点是方法名为 “`greetTo`” 的方法名切点。

当然，用户也可以只通过一个切点描述同时满足以上两个匹配条件的连接点，而更好的方法是使用 Spring 提供的 `ComposablePointcut` 把两个切点组合起来，通过切点的复合运算表示。`ComposablePointcut` 可以将多个切点以并集或交集的方式组合起来，提供了切点之间复合运算的功能。

`ComposablePointcut` 本身也是一个切点，它实现了 `Pointcut` 接口。下面先来了解一下 `ComposablePointcut` 的构造函数。

- ❑ `ComposablePointcut()`: 构造一个匹配所有类所有方法的复合切点。
- ❑ `ComposablePointcut(ClassFilter classFilter)`: 构造一个匹配特定类所有方法的复合切点。
- ❑ `ComposablePointcut(MethodMatcher methodMatcher)`: 构造一个匹配所有类特定方法的复合切点。
- ❑ `ComposablePointcut(ClassFilter classFilter, MethodMatcher methodMatcher)`: 构造一个匹配特定类特定方法的复合切点。

`ComposablePointcut` 提供了 3 个交集运算的方法。

- ❑ `ComposablePointcut intersection(ClassFilter filter)`: 将复合切点和一个 `ClassFilter` 对象进行交集运算，得到一个结果复合切点。
- ❑ `ComposablePointcut intersection(MethodMatcher mm)`: 将复合切点和一个 `MethodMatcher` 对象进行交集运算，得到一个结果复合切点。
- ❑ `ComposablePointcut intersection(Pointcut other)`: 将复合切点和一个切点对象进行交集运算，得到一个结果复合切点。

`ComposablePointcut` 提供了两个并集运算的方法。

- ❑ `ComposablePointcut union(ClassFilter filter)`: 将复合切点和一个 `ClassFilter` 对象进行并集运算，得到一个结果复合切点。

- ❑ `ComposablePointcut union(MethodMatcher mm)`: 将复合切点和一个 `MethodMatcher` 对象进行交并集运算, 则得到一个结果复合切点。

`ComposablePointcut` 没有提供直接对两个切点进行交并集运算的方法, 如果需要对两个切点进行交并集运算, 可以使用 Spring 提供的 `org.springframework.aop.support.Pointcuts` 工具类, 该工具类中有两个非常好用的静态方法。

- ❑ `Pointcut intersection(Pointcut a, Pointcut b)`: 对两个切点进行交集运算, 返回一个结果切点, 该切点即 `ComposablePointcut` 对象的实例。
- ❑ `Pointcut union(Pointcut a, Pointcut b)`: 对两个切点进行并集运算, 返回一个结果切点, 该切点即 `ComposablePointcut` 对象的实例。

下面通过 `ComposablePointcut` 创建一个流程切点和方法名切点的相交切点, 程序代码如代码清单 7-37 所示。

代码清单 7-37 GreetingComposablePointcut

```
package com.smart.advisor;
import org.springframework.aop.Pointcut;
import org.springframework.aop.support.ComposablePointcut;
import org.springframework.aop.support.ControlFlowPointcut;
import org.springframework.aop.support.NameMatchMethodPointcut;
public class GreetingComposablePointcut {
    public Pointcut getIntersectionPointcut() {
        ComposablePointcut cp = new ComposablePointcut(); // ① ← 创建一个复合切点

        Pointcut pt1 = new ControlFlowPointcut(WaiterDelegate.class, "service"); // ② ← 创建一个流程切点

        NameMatchMethodPointcut pt2 = new NameMatchMethodPointcut(); // ③ ← 创建一个方法名切点
        pt2.addMethodName("greetTo");
        return cp.intersection(pt1).intersection(pt2); // ④ ← 将两个切点进行交集操作
    }
}
```

通过 `GreetingComposablePointcut#getIntersectionPointcut()` 方法, 即可得到一个相交的复合切点。配置复合切点的切面和配置其他切点一样, 如代码清单 7-38 所示。

代码清单 7-38 配置复合切点切面

```
<bean id="greetingAdvice" class="com.smart.advisor.GreetingBeforeAdvice" />
<bean id="gcp" class="com.smart.advisor.GreetingComposablePointcut" />
<bean id="composableAdvisor"
class="org.springframework.aop.support.DefaultPointcutAdvisor"
p:pointcut="#{gcp.intersectionPointcut}" ① ← 引用 gcp.intersectionPointcut()
p:advice-ref="greetingAdvice"/>
                                     方法所返回的复合切点

<bean id="waiter4" class="org.springframework.aop.framework.ProxyFactoryBean"
p:interceptorNames="composableAdvisor" ② ← 使用复合切点切面
p:target-ref="waiterTarget"
p:proxyTargetClass="true"/>
```

在①处使用 util 命名空间的标签引用另一个 Bean 的属性(关于 util 命名空间标签的详细信息, 请参看 5.4.6 节)。

下面编写相应的测试代码:

```
String configPath = "com/smart/advisor/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter waiter = (Waiter) ctx.getBean("waiter4");
WaiterDelegate wd = new WaiterDelegate();
wd.setWaiter(waiter);
waiter.serveTo("Peter");
waiter.greetTo("Peter");
wd.service("Peter");
```

运行以上代码, 输出以下信息:

```
waiter serving Peter... ① ← 对应 waiter.serveTo("Peter")
waiter greet to Peter... ② ← 对应 waiter.greetTo("Peter")
com.smart.advisor.Waiter.greetTo ③ ← 对应通过 wd.service("Peter")
                                  调用的 Waiter.greetTo() 方法
How are you! Mr.Peter.
waiter greet to Peter...
waiter serving Peter... ④ ← 对应通过 wd.service("Peter")
                                  调用的 Waiter.serveTo() 方法
```

通过以上输出信息发现, 只有通过 `WaiterDelegate#service()` 方法调用 `Waiter#greetTo()` 方法时才织入了增强, 而这正是复合交集切点所描述的接连点。

7.4.8 引介切面

7.3.6 节介绍了引介增强的知识, 引介切面是引介增强的封装器, 通过引介切面, 可以更容易地为现有对象添加任何接口的实现。图 7-9 是引介切面的类继承关系图。

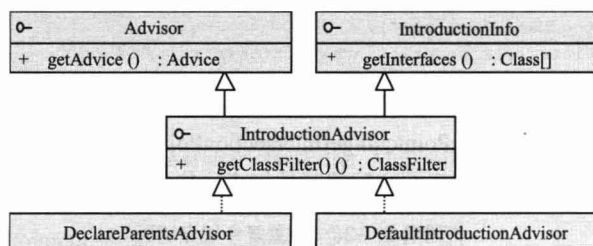


图 7-9 引介切面类继承关系图

从图 7-9 中可以看出, `IntroductionAdvisor` 接口同时继承 `Advisor` 和 `IntroductionInfo` 接口, `IntroductionInfo` 接口描述了目标类需要实现的新接口。 `IntroductionAdvisor` 和 `PointcutAdvisor` 接口不同, 它仅有一个类过滤器 `ClassFilter` 而没有 `MethodMatcher`, 这是因为引介切面的切点是类级别的, 而 `Pointcut` 的切点是方法级别的。

`IntroductionAdvisor` 有两个实现类, 分别是 `DefaultIntroductionAdvisor` 和 `DeclareParentsAdvisor`, 前者是引介切面最常用的实现类, 后者用于实现使用 AspectJ 语言的 `DeclareParent` 注解表示的引介切面。

`DefaultIntroductionAdvisor` 拥有 3 个构造函数。

- ❑ `DefaultIntroductionAdvisor(Advice advice)`: 通过一个增强创建的引介切面, 引介切面将为目标对象新增增强对象中所有接口的实现。
- ❑ `DefaultIntroductionAdvisor(DynamicIntroductionAdvice advice, Class clazz)`: 通过一个增强和一个指定的接口类创建引介切面, 仅为目标对象新增 `clazz` 接口的实现。
- ❑ `DefaultIntroductionAdvisor(Advice advice, IntroductionInfo introductionInfo)`: 通过一个增强和一个 `IntroductionInfo` 创建引介切面, 目标对象需要实现哪些接口由 `introductionInfo` 对象的 `getInterfaces()` 方法表示。

下面通过 `DefaultIntroductionAdvisor` 为代码清单 7-22 中的引介增强配置切面, 会发现这种方式比前面的方式要更简洁、更清晰, 如代码清单 7-39 所示。

代码清单 7-39 配置引介切面

```
<bean id="introduceAdvisor" class="org.springframework.aop.support.
DefaultIntroductionAdvisor">
  <constructor-arg> ① <!-- ControllablePerformanceMonitor 是一个 Advice 对象 -->
    <bean class="com.smart.introduce.ControllablePerformanceMonitor"/>
  </constructor-arg>
</bean>
<bean id="forumServiceTarget" class="com.smart.introduce.ForumService"/>
<bean id="forumService" class="org.springframework.aop.framework.ProxyFactoryBean"
  p:interceptorNames="introduceAdvisor"
  p:target-ref="forumServiceTarget"
  p:proxyTargetClass="true"/>
```

虽然引介切面和其他切面有很大的不同, 但却可以采用相似的 Spring 配置方式配置引介切面。通过这种方式配置的代码在本质上和代码清单 7-23 中配置的效果是一样的。

7.5 自动创建代理

在前面所有的例子中, 都通过 `ProxyFactoryBean` 创建织入切面的代理, 每个需要被代理的 Bean 都需要使用一个 `ProxyFactoryBean` 进行配置, 虽然可以使用父子 `<bean>` 进行改造, 但还是很麻烦。对于小型系统, 可以将就使用, 但对于由拥有众多需要代理 Bean 的系统, 原来的配置显然不尽如人意。

幸运的是, Spring 提供了自动代理机制, 让容器自动生成代理, 把开发人员从烦琐的配置工作中解放出来。在内部, Spring 使用 `BeanPostProcessor` 自动完成这项工作。

7.5.1 实现类介绍

这些基于 `BeanPostProcessor` 的自动代理创建器的实现类, 将根据一些规则自动在容

器实例化 Bean 时为匹配的 Bean 生成代理实例。这些代理创建器可以分为 3 类。

- ❑ 基于 Bean 配置名规则的自动代理创建器：允许为一组特定配置名的 Bean 自动创建代理实例的代理创建器，实现类为 `BeanNameAutoProxyCreator`。
- ❑ 基于 Advisor 匹配机制的自动代理创建器：它会对容器中所有的 Advisor 进行扫描，自动将这些切面应用到匹配的 Bean 中（为目标 Bean 创建代理实例），实现类为 `DefaultAdvisorAutoProxyCreator`。
- ❑ 基于 Bean 中 AspectJ 注解标签的自动代理创建器：为包含 AspectJ 注解的 Bean 自动创建代理实例，实现类为 `AnnotationAwareAspectJAutoProxyCreator`。

图 7-10 是自动代理创建器实现类的类继承图。

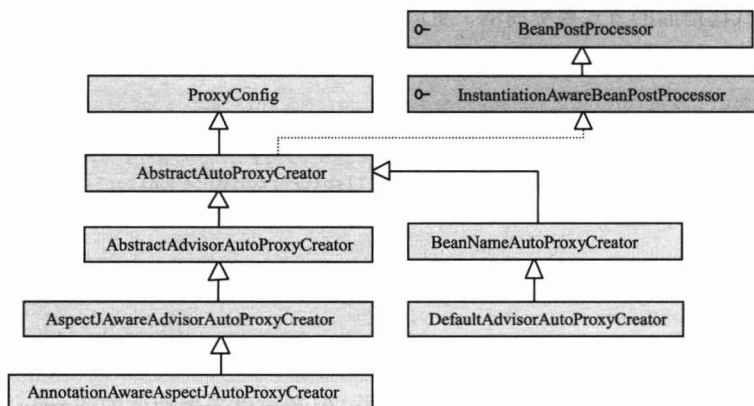


图 7-10 自动代理创建器实现类的类继承图

从图 7-10 中可以清楚地看到所有的自动代理创建器类都实现了 `BeanPostProcessor`，在容器实例化 Bean 时，`BeanPostProcessor` 将对它进行加工处理，所以，自动代理创建器有机会对满足匹配规则的 Bean 自动创建代理对象。

本章暂不涉及 AspectJ 的内容，下面只讲解 `BeanNameAutoProxyCreator` 和 `DefaultAdvisorAutoProxyCreator` 的用法。

7.5.2 BeanNameAutoProxyCreator

在代码清单 7-29 中通过配置两个 `ProxyFactoryBean` 分别为 waiter 和 seller 的 Bean 创建代理对象。下面通过 `BeanNameAutoProxyCreator` 以更优雅、更便捷的方式完成相同的功能，如代码清单 7-40 所示。

代码清单 7-40 使用 Bean 名进行自动代理

```

<bean id="waiter" class="com.smart.advisor.Waiter" />
<bean id="seller" class="com.smart.advisor.Seller" />
<bean id="greetingAdvice" class="com.smart.advisor.GreetingBeforeAdvice" />

<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator"

```

```

p:beanNames="*er" ①
p:interceptorNames="greetingAdvice"
p:optimize="true"/>

```

由于只有一个 Bean 名称，所以直接使用 value 属性进行配置，可以通过<list>子元素设定多个 Bean 名称，或通过逗号、空格、分号的方式设定多个 Bean 名称

BeanNameAutoProxyCreator 有一个 beanNames 属性，它允许用户指定一组需要自动代理的 Bean 名称，Bean 名称可以使用*通配符。假设 Spring 容器中除 waiter 和 seller 外还有其他的 Bean，如果以“er”为后缀的规则可以将这两个 Bean 和容器中其他的 Bean 区分开，就可以通过将 beanNames 属性设定为“*er”使 waiter 和 seller 这两个 Bean 被自动代理。当然，使用通配符会带来一定的风险，在例子中，假设一个其他的 Bean 名称也以“er”结尾，则自动代理创建器也会为该 Bean 创建代理。所以为了保险起见，用户可以使用下面的方式配置 beanNames 属性：value="waiter,seller"。

一般情况下不会为 FactoryBean 的 Bean 创建代理，如果刚好有这样一个需求，则需要在 beanNames 中指定添加\$的 Bean 名称，如<property name="beanNames" value="\$waiter"/>等。

BeanNameAutoProxyCreator 的 interceptorNames 属性指定一个或多个增强 Bean 的名称。此外，还有一个常用的 optimize 属性，如果将此属性设置为 true，则将强制使用 CGLib 动态代理技术。

通过这样的配置后，容器在创建 waiter 和 seller Bean 的实例时，就会自动为它们创建代理对象。而这一操作对于使用者来说是完全透明的，可以通过以下测试证实这一点：

```

String configPath = "com/smart/autoproxy/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter waiter = (Waiter) ctx.getBean("waiter"); ①
Seller seller = (Seller) ctx.getBean("seller"); ②
waiter.greetTo("John");
seller.greetTo("Tom");

```

运行以上代码，在控制台上输出以下信息：

```

com.smart.advisor.Waiter.greetTo ① ← 被自动织入的增强逻辑
How are you! Mr. John.
waiter greet to John...
com.smart.advisor.Seller.greetTo ② ← 被自动织入的增强逻辑
How are you! Mr. Tom.
seller greet to Tom...

```

通过输出信息可以知道，从容器中返回的 waiter Bean 的 greetTo()方法及 seller Bean 的 greetTo()方法都被织入了增强，可见①处返回的 waiter Bean 和②处返回的 seller Bean 都是被代理过的对象。

7.5.3 DefaultAdvisorAutoProxyCreator

我们知道切面 Advisor 是切点和增强的复合体，Advisor 本身已经包含了足够的信息，如横切逻辑（要织入什么）及连接点（织入哪里）。

DefaultAdvisorAutoProxyCreator 能够扫描容器中的 Advisor，并将 Advisor 自动织入匹配的目标 Bean 中，即为匹配的目标 Bean 自动创建代理。

在代码清单 7-30 中通过 ProxyFactoryBean 为 waiter 配置了代理, 在这里, 引入 DefaultAdvisorAutoProxyCreator 为容器中所有带 “greet” 方法名的目标 Bean 自动创建代理。

```
<bean id="waiter" class="com.smart.advisor.Waiter" />
<bean id="seller" class="com.smart.advisor.Seller" />
<bean id="greetingAdvice" class="com.smart.advisor.GreetingBeforeAdvice" />
<bean id="regexAdvisor"
    class="org.springframework.aop.support.RegexpMethodPointcutAdvisor"
    p:patterns=".*greet.*"
    p:advice-ref="greetingAdvice" />

<bean class="org.springframework.aop.framework.autoproxy.
    DefaultAdvisorAutoProxyCreator"/>①
```

在①处, 用 DefaultAdvisorAutoProxyCreator 定义了一个 Bean, 它负责将容器中的 Advisor 织入匹配的目标 Bean 中。通过下面的测试代码检测一下自动代理创建器是否正常工作:

```
String configPath = "com/smart/autoproxy/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter waiter = (Waiter) ctx.getBean("waiter");
Seller seller = (Seller) ctx.getBean("seller");
waiter.serveTo("John");
waiter.greetTo("John");
seller.greetTo("Tom");
```

运行以上代码, 输出以下信息:

```
waiter serving John...
com.smart.advisor.Waiter.greetTo
How are you! Mr.John.
waiter greet to John...
com.smart.advisor.Seller.greetTo
How are you! Mr.Tom.
seller greet to Tom...
```

Waiter#serveTo()方法没有被织入增强, 而 Waiter 和 Seller 的 greetTo()方法都被织入了增强, 由此可见, 增强被正确地织入匹配的接连点中。

7.5.4 AOP 无法增强疑难问题剖析

大家在使用 Spring AOP 时, 或多或少会碰到一些方法无法被增强的问题, 有时同一个类里面的方法有的可以被增强, 有的却无法被增强。要分析其原因, 首先要从 Spring AOP 的实现机制入手。从上文 Spring AOP 基础知识的学习可以知道, AOP 底层实现有两种方法: 一种是基于 JDK 动态代理; 另一种是基于 CGLib 动态代理。

在 JDK 动态代理中通过接口来实现方法拦截, 所以必须确保要拦截的目标方法在接口中有定义, 否则将无法实现拦截。

在 CGLib 动态代理中通过动态生成代理子类来实现方法拦截, 所以必须确保要拦截的目标方法可被子类访问, 也就是目标方法必须定义为非 final, 则非私有实例方法。

是否注意了上面几点, 就可以万事大吉了? 要回答这个问题, 先来看一个实例。首先对 7.5.3 节的示例进行简单的修改, 如代码清单 7-41 所示。

代码清单 7-41 增强问题测试

```
public class AopAwareTest {
    @Test
    public void autoProxy() {
        String configPath = "com/smart/autoproxy/beans-aware.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
        Waiter waiter = (Waiter) ctx.getBean("waiter");
        waiter.serveTo("John");
        waiter.greetTo("John"); //①
    }
}
```

在这里, 引入 DefaultAdvisorAutoProxyCreator 为容器中所有带“To”方法名的目标 Bean 自动创建代理, 如代码清单 7-42 所示。

代码清单 7-42 beans-aware.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.springframework.org/schema/xsi"
    xsi:schemaLocation="http://www.springframework.org/schema/context/spring-context.xsd"
    >
    <context:component-scan base-package="com.smart.aop" />
    <bean id="waiter" class="com.smart.advisor.Waiter" />
    <bean id="greetingAdvice" class="com.smart.advisor.GreetingBeforeAdvice" />
    <!--通过Advisor自动创建代理-->
    <bean id="regexAdvisor"
        class="org.springframework.aop.support.RegexpMethodPointcutAdvisor"
        p:patterns=".*To.*" p:advice-ref="greetingAdvice" />
    <bean
        class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"
        p:proxyTargetClass="true" />
</beans>
```

为了方便测试, 这里只保留服务生。运行以上代码, 输出以下信息:

```
com.smart.advisor.Waiter.serveTo
How are you! Mr.John.
waiter serving John...
com.smart.advisor.Waiter.greetTo
How are you! Mr.John.
waiter greet to John...
```

从运行结果来看, Waiter#serveTo()和 Waiter#greetTo()方法都被织入了增强, 由此可见增强被正确地织入匹配的接连点中。现在改造一下 Waiter#serveTo()方法, 让 Waiter#serveTo()方法内部直接调用 Waiter#greetTo()方法, 并把代码清单 7-41 中①处的方法注释掉。模拟测试在一个类中一个可织入的方法 serveTo()调用另一个可织入的方法 greetTo()时, 这两个方法是否都可以被织入增强, 如代码清单 7-43 所示。

代码清单 7-43 增强问题测试

```

public class AopAwareTest {
    @Test
    public void autoProxy() {
        String configPath = "com/smart/autoproxy/beans-aware.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
        Waiter waiter = (Waiter) ctx.getBean("waiter");
        waiter.serveTo("John");
    }
}

public class Waiter {
    ...

    public void serveTo(String name){
        System.out.println("waiter serving "+name+"...");
        greetTo(name);
    }

    public void greetTo(String name) {
        System.out.println("waiter greet to "+name+"...");
    }
}

```

运行以上代码，输出以下信息：

```

com.smart.advisor.Waiter.serveTo
How are you! Mr.John.
waiter serving John...
waiter greet to John...

```

从运行结果来看，只有 `Waiter#serveTo()` 方法被织入了增强，而 `Waiter#greetTo()` 方法没有被织入增强。原本希望这两个方法都被织入增强，但现在只有从外部调用的方法 `serveTo()` 被增强，在同一个类的内部方法之间调用的方法 `greetTo()` 无法被增强。笔者在实际的项目中也碰到过这个问题，经过反复的测试研究发现，原来在方法内部之间调用的时候，不会使用被增强的代理类，而是直接调用未被增强原类的方法，这也就是方法 `greetTo()` 无法被织入增强的原因。

在实际项目中存在内部两个方法调用的问题，同时又希望它们都能够被增强，例如在一个 `Service` 类的两个方法 `bus1()` 和 `bus2()` 中都使用了缓存注解 `@Cacheable`，也就是希望这两个方法都启用缓存。如果在业务处理过程中，`bus2()` 方法只被 `bus1()` 所调用，则 `bus2()` 方法永远无法启用缓存，这在无形中给系统留下了隐患，而且不易被发现。

笔者为此研究了一套解决办法，就是想办法在内部方法调用时，让其通过代理类调用内部的方法。因此，需要让原来的 `Waiter` 实现一个可注入自身代理类的接口 `BeanSelfProxyAware`，如代码清单 7-44 所示。

代码清单 7-44 BeanSelfProxyAware 接口

```

public interface BeanSelfProxyAware {
    void setSelfProxy(Object object); // 织入自身代理类接口
}

```

有了这个代理类接口之后，需要对所有实现了 `BeanSelfProxyAware` 接口的 `Bean` 执

行自身代理 Bean 的注入，设计一个可复用的注入装配器 BeanSelfProxyAwareMouter，如代码清单 7-45 所示。

代码清单 7-45 BeanSelfProxyAwareMouter

```
package com.smart.aop;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.core.Ordered;
import org.springframework.stereotype.Component;
...
@Component
public class BeanSelfProxyAwareMouter implements SystemBootAddon,
ApplicationContextAware {
    private Logger logger = LoggerFactory.getLogger(this.getClass());
    private ApplicationContext applicationContext;

    //①注入Spring容器
    public void setApplicationContext(ApplicationContext applicationContext) throws
BeansException {
        this.applicationContext = applicationContext;
    }

    public void onReady() { //②实现系统启动器接口中的装配就绪方法

        //②-1从容器中获取所有注入的自动代理Bean
        Map<String, BeanSelfProxyAware> proxyAwareMap =
            applicationContext.getBeansOfType(BeanSelfProxyAware.class);
        if(proxyAwareMap!=null){
            for (BeanSelfProxyAware beanSelfProxyAware : proxyAwareMap.values()) {
                beanSelfProxyAware.setSelfProxy(beanSelfProxyAware);
                if (logger.isDebugEnabled()) {
                    logger.debug("{}注册自身被代理的实例。");
                }
            }
        }

        public int getOrder() {
            return Ordered.HIGHEST_PRECEDENCE;
        }
    }
}
```

在①处通过实现 `ApplicationContextAware#setApplicationContext()` 接口方法来注入 Spring 容器上下文。在②处实现了系统启动器 `SystemBootAddon#onReady()` 接口方法，此接口方法在下文中进行详细说明。这里的实现逻辑是从 Spring 容器中获取所有实现自动代理织入接口 `BeanSelfProxyAware` 的 Bean，循环迭代遍历这些 Bean，并调用 `setSelfProxy()` 方法将自身代理类注入自身。

接下来需要编写一个系统所有组件都装载完成后、准备就绪前调用的插件接口 `SystemBootAddon`，用于在 Spring 容器启动完成之后触发调用注入装配器 `BeanSelfProxyAwareMouter`，如代码清单 7-46 所示。

代码清单 7-46 SystemBootAddon接口

```
public interface SystemBootAddon extends Ordered{

    //在系统就绪后调用的方法
    void onReady();

}
```

这个接口比较简单，只设计了一个接口方法 `onReady()`，用于在系统就绪后调用。同时实现了 Bean 加载顺序接口 `Ordered`，在插件中可以实现 `Ordered#getOrder` 方法，返回一个整型数字来指定插件的执行顺序。值越小优先被加载处理，值越大最后被加载处理。

最后需要设置一个启动管理器，告诉 Spring 什么时候触发 `BeanSelfProxyAwareMounter` 装配器，如代码清单 7-47 所示。

代码清单 7-47 设置启动管理器

```
...
import org.springframework.context.ApplicationListener;
import org.springframework.context.event.ContextRefreshedEvent;
import org.springframework.core.OrderComparator;
import org.springframework.stereotype.Component;
import org.springframework.util.Assert;

@Component
public class SystemBootManager implements ApplicationListener<ContextRefreshedEvent> {
    private Logger logger = LoggerFactory.getLogger(this.getClass());
    private List<SystemBootAddon> systemBootAddons = Collections.EMPTY_LIST;
    private boolean hasRunOnce = false;

    //①注入所有SystemBootAddon插件
    @Autowired(required = false)
    public void setSystemBootAddons(List<SystemBootAddon> systemBootAddons) {
        Assert.notEmpty(systemBootAddons);
        OrderComparator.sort(systemBootAddons);
        this.systemBootAddons =systemBootAddons;
    }

    //②触发所有插件
    public void onApplicationEvent(ContextRefreshedEvent event) {
        if (!hasRunOnce) {
            for (SystemBootAddon systemBootAddon : systemBootAddons) {
                systemBootAddon.onReady();
                if (logger.isDebugEnabled()) {
                    logger.debug("执行插件:{}",systemBootAddon.getClass().getCanonical
Name());
                }
            }
            hasRunOnce = true;
        }else{
            if (logger.isDebugEnabled()) {
                logger.debug("已执行过容器启动插件集，本次忽略之。");
            }
        }
    }
}
```

```

    }
}

```

在①处通过自动注入方式注入所有实现 `SystemBootAddon` 接口的插件，在②处通过监听 Spring 容器的 `ContextRefreshedEvent` 事件调用容器中所有注册的 `SystemBootAddon` 插件。至此，设计的注入 Bean 自身代理类的通用型插件已经全部完成，接下来只需在需要注入自身代理类的 Bean 中实现 `BeanSelfProxyAware` 接口即可，如本示例中的 `Waiter` 类。下面对代码清单 7-43 中的 `Waiter` 类进行简单改造，如代码清单 7-48 所示。

代码清单 7-48 增强问题测试

```

...
public class Waiter implements BeanSelfProxyAware {
    private Waiter waiter;

    public void setSelfProxy(Object object) {
        waiter = (Waiter)object;
    }

    public void serveTo(String name){
        System.out.println("waiter serving "+name+"...");
        greetTo(name);
    }

    public void greetTo(String name) {
        System.out.println("waiter greet to "+name+"...");
    }
}

```

重新运行代码清单 7-43 中的 `AopAwareTest` 测试，输出以下信息：

```

com.smart.advisor.Waiter.serveTo
How are you! Mr.John.
waiter serving John...
com.smart.advisor.Waiter.greetTo
How are you! Mr.John.
waiter greet to John...

```

从运行结果来看，`Waiter#serveTo()`和 `Waiter#greetTo()`方法都被织入了增强，由此可见设计的注入 Bean 自身代理类的通用型插件有效。如果读者在实际项目中遇到同样的问题，则可以借鉴或使用笔者设计的这个插件。

如果不想使用上述方式，则也可以考虑使用 `AspectJ`，因为其支持编译期织入且不需要生成代理类，也就避免了因生成代理类产生的一些制约（`static` 和 `final` 方法不能被覆盖等问题）。关于 `AspectJ` 的用法，将在下一章进行详细讲解。

7.6 小结

AOP 是 OOP 的延伸，它为程序开发提供了一个崭新的思考角度，可以将重复性的

横切逻辑抽取到统一的模块中。通过 OOP 的纵向抽象和 AOP 的横向抽取，程序才可以真正解决重复性代码问题。

Spring 采用 JDK 动态代理和 CGLib 动态代理技术在运行期织入增强，所以不需要装备特殊的编译器或类装载器就可以使用 AOP 的功能。要使用 JDK 动态代理，目标类必须实现接口，而 CGLib 不对目标类作任何限制，它通过动态生成目标类子类的方式提供代理。JDK 在创建代理对象时的性能高于 CGLib，而生成的代理对象的运行性能却比 CGLib 的低。如果是 singleton 的代理，则推荐使用 CGLib 动态代理。

Spring 只能在方法级别上织入增强，Spring 提供了 4 种类型的方法增强，分别是前置增强、后置增强、环绕增强和异常抛出增强，此外还有一种特殊的引介增强。引介增强是类级别的，它为目标类织入新的接口实现。从广义上说，增强其实就是一种最简单的切面，它既包括横切代码又包括切点信息，只不过它的切点只是简单的方法相对位置信息。所以增强一般需要和切点联合才可以表示一个更具实用性的切面。

在 Spring 中，普通的切点通过目标类名和方法名描述连接点的信息。流程切点是比较特殊的切点，它通过方法调用堆栈的运行环境信息来决定连接点。有时需要通过切点的交叉或合并描述一个最终的切点，这时可以使用 ComposablePointcut 的复合切点。

切面是增强和切点的联合体，可以很方便地通过 Spring 提供的 ProxyBeanFactory 将切面织入不同的目标类中。当然，为每个目标类手工配置一个切面是比较烦琐的，Spring 利用 BeanPostProcessor 可干涉 Bean 生命周期的机制，提供了一些可以自动创建代理、织入切面的自动代理创建器，其中 DefaultAdvisorAutoProxyCreator 是功能强大的自动代理创建器，它可以将容器中的所有 Advisor 自动织入目标 Bean 中。

第 8 章

基于@AspectJ 和 Schema 的 AOP

在上一章的学习中我们发现，在 Spring 中定义一个切面是比较烦琐的，需要实现专门的接口，并进行一些较为复杂的配置，Spring AOP 的配置是被批评最多的地方。Spring 听到了这方面的声音，下决心解决这一问题，并取得了很好的突破。如今，Spring AOP 已经焕然一新，用户可以使用@AspectJ 注解非常容易地定义一个切面，而不需要实现任何接口。对于没有使用 Java 5.0 的项目，可以通过基于 Schema 的配置定义切面，其方便程度和基于@AspectJ 注解的配置相差无几。此外，Spring 还可以使用 AspectJ 语言编写切面，发扬二者的长处：用 AspectJ 织入切面，并让 Spring 容器管理这些切面。

本章主要内容：

- ◆ Java 5.0 注解知识
- ◆ 通过@AspectJ 定义切面
- ◆ 切点函数讲解
- ◆ 绑定连接点参数
- ◆ 基于 Schema 配置定义切面
- ◆ Spring LTW

本章亮点：

- ◆ 对切点表达式函数进行深入分析
- ◆ 深入讲解集成 AspectJ 的过程

8.1 Spring 对 AOP 的支持

Spring 在新版本中对 AOP 功能进行了重要的增强，主要表现在以下几个方面：

- 新增了基于 Schema 的配置支持，为 AOP 专门提供了 aop 命名空间。
- 新增了对 AspectJ 切点表达式语言的支持。`@AspectJ` 是 AspectJ 1.5 新增的功能，它通过 Java 5.0 的注解技术，允许开发者在 POJO 中定义切面。Spring 使用和 `@AspectJ` 相同风格的注解，并通过 AspectJ 提供的注解库和解析库处理切点。当然，由于 Spring 只支持方法级的切点，所以仅对 `@AspectJ` 提供了有限的支持。
- 可以无缝地集成 AspectJ。AspectJ 提供了语言级切面的实现，Spring 无意开发一个重复的东西，Spring 对开源世界里一切优秀的东西向来采取兼收并蓄的态度。

这里所说的 Spring AOP 包括基于 XML 配置的 AOP 和基于 `@AspectJ` 注解的 AOP，这两种方法虽然在配置切面时的表现方式不同，但底层都采用了动态代理技术（JDK 或 CGLib 动态代理）。Spring 可以集成 AspectJ，但 AspectJ 本身并不属于 Spring AOP 的范畴。

一般情况下，对于开发 JAVA EE 企业应用的开发者而言，Spring AOP 已经可以满足使用的要求。虽然 AspectJ 提供对 AOP 更为细致的实现，但像实例化切面、属性访问切面、条件切面等功能，在实际应用中并不常用。

如果是基于 Java 5.0 的项目，推荐使用 Spring 提供的 `@AspectJ` 配置方式，因为它能以更简单、更直接的方式应用切面。

8.2 Java 5.0 注解知识快速进阶

在进入本章学习之前，有必要对 Java 5.0 新增的注解（Annotation）技术进行简单的学习。因为 Spring 支持 `@AspectJ`，而 `@AspectJ` 本身就是基于 Java 5.0 的注解技术，所以学习 Java 5.0 的注解知识有助于更好地理解和掌握 Spring 的 AOP 技术。

8.2.1 了解注解

对于 Java 开发人员来说，在编写代码时，除源程序外，还会使用 Javadoc 标签对类、方法或成员变量进行注释，以便使用 Javadoc 工具生成和源码配套的 Javadoc 文档。这些 `@param`、`@return` 等 Javadoc 标签就是注解标签，它们为第三方工具提供了描述程序代码的注释信息。使用过 Xdoclet 的读者对此更有感触，像 Struts、Hibernate 都提供了 Xdoclet 标签，使用它们可以快速地生成对应程序代码的配置文件。

Java 5.0 注解可以看作 Javadoc 和 Xdoclet 标签的延伸与发展。在 Java 5.0 中可以自定义这些标签，并通过 Java 语言的反射机制获取类中标注的注解，完成特定的功能。

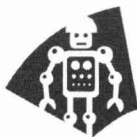
注解是代码的附属信息，它遵循一个基本原则：注解不能直接干扰程序代码的运行，无论增加或删除注解，代码都能够正常运行。Java 语言解释器会忽略这些注解，而由第三方工具负责对注解进行处理。第三方工具可以利用代码中的注解间接控制程序代码的

运行，它们通过 Java 反射机制读取注解的信息，并根据这些信息更改目标程序的逻辑，而这正是 Spring AOP 对@AspectJ 提供支持所采取的方法。



轻松一刻

很多事物的设计都必须遵循最基本的原则。为了防止机器人伤害人类，科幻作家阿西莫夫于 1940 年提出了“机器人三原则”：第一，机器人不能伤害人类；第二，机器人应遵守人类的命令，与第一条违背的命令除外；第三，机器人应能保护自己，与第一条违背的命令除外。这是给机器人赋予的伦理性纲领，机器人学术界一直将这 3 条原则作为机器人开发的准则。



8.2.2 一个简单的注解类

通常情况下，第三方工具不但负责处理特定的注解，其本身还提供了这些注解的定义，所以通常仅需关注如何使用注解即可。但定义注解类本身并不困难，Java 提供了定义注解的语法。下面着手编写一个简单的注解类，如代码清单 8-1 所示。

代码清单 8-1 NeedTest 注解类

```
package com.smart.aspectj.anno;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME) //①声明注解的保留期限
@Target(ElementType.METHOD) //②声明可以使用该注解的目标类型
public @interface NeedTest { //③定义注解
    boolean value() default true; //④声明注解成员
}
```

Java 新语法规定使用@interface 修饰符定义注解类，如③处所示。一个注解可以拥有多个成员，成员声明和接口方法声明类似，这里仅定义了一个成员，如④处所示。成员声明有以下几点限制：

- ❑ 成员以无入参、无抛出异常的方式声明，如 boolean value(String str)、boolean value() throws Exception 等方式是非法的。
- ❑ 可以通过 default 为成员指定一个默认值，如 String level() default "LOW_LEVEL"、int high() default 2 是合法的，当然也可以不指定默认值。
- ❑ 成员类型是受限的，合法的类型包括原始类型及其封装类、String、Class、enums、注解类型，以及上述类型的数组类型，如 ForumService value()、List foo() 是非法的。

在①和②处所看到的注解是 Java 预定义的注解，称为元注解（Meta-Annotation），它们被 Java 编译器使用，会对注解类的行为产生影响。@Retention(RetentionPolicy.

RUNTIME)表示 `NeedTest` 这个注解可以在运行期被 JVM 读取, 注解的保留期限类型在 `java.lang.annotation.Retention` 类中定义, 介绍如下。

- ❑ **SOURCE**: 注解信息仅保留在目标类代码的源码文件中, 但对应的字节码文件将不再保留。
- ❑ **CLASS**: 注解信息将进入目标类代码的字节码文件中, 但类加载器加载字节码文件时不会将注解加载到 JVM 中, 即运行期不能获取注解信息。
- ❑ **RUNTIME**: 注解信息在目标类加载到 JVM 后依然保留, 在运行期可以通过反射机制读取类中的注解信息。

`Target(ElementType.METHOD)`表示 `NeedTest` 这个注解只能应用到目标类的方法上, 注解的应用目标在 `java.lang.annotation.ElementType` 类中定义, 介绍如下。

- ❑ **TYPE**: 类、接口、注解类、Enum 声明处, 相应的注解称为类型注解。
- ❑ **FIELD**: 类成员变量或常量声明处, 相应的注解称为域值注解。
- ❑ **METHOD**: 方法声明处, 相应的注解称为方法注解。
- ❑ **PARAMETER**: 参数声明处, 相应的注解称为参数注解。
- ❑ **CONSTRUCTOR**: 构造函数声明处, 相应的注解称为构造函数注解。
- ❑ **LOCAL_VARIABLE**: 局部变量声明处, 相应的注解称为局域变量注解。
- ❑ **ANNOTATION_TYPE**: 注解类声明处, 相应的注解称为注解类注解, `ElementType.ANNOTATION_TYPE` 包括 `ElementType.ANNOTATION_TYPE`。
- ❑ **PACKAGE**: 包声明处, 相应的注解称为包注解。

如果注解只有一个成员, 则成员名必须取名为 `value()`, 在使用时可以忽略成员名和赋值号 (=), 如 `@NeedTest(true)`。当注解类拥有多个成员时, 如果仅对 `value` 成员进行赋值, 则也可不使用赋值号; 如果同时对多个成员进行赋值, 则必须使用赋值号, 如 `DeclareParents (value = "NaiveWaiter", defaultImpl = SmartSeller.class)`。注解类可以没有成员, 没有成员的注解称为标识注解, 解释程序以标识注解存在与否进行相应的处理; 此外, 所有的注解类都隐式继承于 `java.lang.annotation.Annotation`, 但注解不允许显式继承于其他的接口。

我们希望使用 `NeedTest` 注解对业务类的方法进行标注, 以便测试工具可以根据注解情况激活或关闭对业务类的测试。在编写好 `NeedTest` 注解类后, 就可以在其他类中使用它了。

8.2.3 使用注解

在 `ForumService` 中使用 `NeedTest` 注解, 标注业务方法是否需要测试, 如代码清单 8-2 所示。

代码清单 8-2 ForumService: 使用注解

```
package com.smart.aspectj.anno;
public class ForumService {
    @NeedTest(value=true) //①标注注解
    public void deleteForum(int forumId){
        System.out.println("删除论坛模块: "+forumId);
    }

    @NeedTest(value=false) //②标注注解
    public void deleteTopic(int postId){
        System.out.println("删除论坛主题: "+postId);
    }
}
```

如果注解类和目标类不在同一个包中,则需要通过 `import` 引用注解类。在①和②处,使用 `NeedTest` 分别对 `deleteForum()` 和 `deleteTopic()` 方法进行标注。在标注注解时,可以通过以下格式对注解成员进行赋值:

@<注解名>(<成员名 1>=<成员值 1>,<成员名 2>=<成员值 2>,...)

如果成员是数组类型,则可以通过 `{}` 进行赋值,如 `boolean` 数组的成员可以设置为 `{true,false,true}`。下面是几个注解标注的例子。

示例 1: 多成员的注解。

```
@AnnoExample(id= 2868724, synopsis = "Enable time-travel",
engineer = "Mr. Peabody",date = "4/1/2007")
```

示例 2: 一个成员的注解,成员名为 `value`。可以省略成员名和赋值符号。

```
@Copyright("2011 bookegou.com All Right Reserved")
```

示例 3: 无成员的注解。

```
@Override
```

示例 4: 成员为字符串数组的注解。

```
@SuppressWarnings(value={"unchecked","fallthrough"})
```

示例 5: 成员为注解数组类型的注解。

```
@Reviews({@Review(grade=Review.Grade.EXCELLENT,reviewer="df"),
    @Review(grade=Review.Grade.UNSATISFACTORY,reviewer="eg",
        comment="This method needs an @Override annotation")})
```

`@Reviews` 注解拥有一个 `@Review` 注解数组类型的成员。`@Review` 注解类型有 3 个成员,其中 `reviewer`、`comment` 都是 `String` 类型的成员,但 `comment` 有默认值,而 `grade` 是枚举类型的成员。

由于 `NeedTest` 注解的保留限期是 `RetentionPolicy.RUNTIME` 类型,因此,当 `ForumService` 被加载到 JVM 时,仍可通过反射机制访问到 `ForumService` 各个方法的注解信息。

8.2.4 访问注解

前面提到过,注解不会直接影响程序的运行,但是第三程序或工具可以利用代码

中的注解完成特殊的任务，间接控制程序的运行。对于 RetentionPolicy.RUNTIME 保留期限的注解，可以通过反射机制访问类中的注解。

在 Java 5.0 中，Package、Class、Constructor、Method 及 Field 等反射对象都新增了访问注解信息的方法：<T extends Annotation>T getAnnotation(Class<T> annotationClass)，该方法支持通过泛型直接返回注解对象。

下面通过反射来访问注解，得出 ForumService 类中通过 @NeedTest 注解所承载的测试需求，如代码清单 8-3 所示。

代码清单 8-3 ToolTest: 访问代码中的注解

```
package com.smart.aspectj.anno;
import java.lang.reflect.Method;
import org.testng.annotations.*;

public class ToolTest {
    @Test
    public void tool() {

        //①得到ForumService对应的Class对象
        Class clazz = ForumService.class;

        //②得到 ForumService 对应的 Method 数组
        Method[] methods = clazz.getDeclaredMethods();
        System.out.println(methods.length);
        for (Method method : methods) {

            //③获取方法上所标注的注解对象
            NeedTest nt = method.getAnnotation(NeedTest.class);
            if (nt != null) {
                if (nt.value()) {
                    System.out.println(method.getName() + "()需要测试");
                } else {
                    System.out.println(method.getName() + "()不需要测试");
                }
            }
        }
    }
}
```

在③处通过方法的反射对象，获取了方法上所标注的 NeedTest 注解对象，接着就可以访问注解对象的成员，从而得到 ForumService 类方法的测试需求。运行以上代码，输出以下信息：

```
deleteForum()需要测试
deleteTopic()不需要测试
```

8.3 着手使用@AspectJ

第 7 章分别使用 Pointcut 和 Advice 接口描述切点和增强，并用 Advisor 整合二者描

述切面，@AspectJ 则采用注解来描述切点、增强，二者只是表述方式不同，描述内容的本质是完全相同的，这就好比一个用中文、一个用英文讲述同一则伊索寓言。

8.3.1 使用前的准备

在使用@AspectJ 之前，首先必须保证所使用的 Java 是 5.0 及以上版本，否则无法使用注解技术。

Spring 在处理@Aspect 注解表达式时，需要将 Spring 的 asm 模块添加到类路径中。asm 是轻量级的字节码处理框架，因为 Java 的反射机制无法获取入参名，Spring 就利用 asm 处理@AspectJ 中所描述的方法入参名。

此外，Spring 采用 AspectJ 提供的@AspectJ 注解类库及相应的解析类库，需要在 pom.xml 文件中添加 aspectj.weaver 和 aspectj.tools 类包的依赖。

8.3.2 一个简单的例子

在做好准备工作中，就可以开始编写一个基于@AspectJ 的切面。首先来看一个简单的例子，以便对@AspectJ 有一个切身的体会。

@AspectJ 采用不同的方式对 AOP 进行描述，依旧使用 NaiveWaiter 的例子进行讲解。为了方便阅读，再次给出 NaiveWaiter 类的代码，如下：

```
package com.smart;
public class NaiveWaiter implements Waiter {
    public void greetTo(String clientName) {
        System.out.println("NaiveWaiter:greet to "+clientName+"...");
    }
    public void serveTo(String clientName){
        System.out.println("NaiveWaiter:serving "+clientName+"...");
    }
}
```

下面使用@AspectJ 注解定义一个切面，如代码清单 8-4 所示。

代码清单 8-4 PreGreetingAspect: 切面

```
package com.smart.aspectj.aspectj;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@AspectJ //①通过该注解将PreGreetingAspect标识为一个切面
public class PreGreetingAspect{

    @Before("execution(* greetTo(..))") //②定义切点和增强类型
    public void beforeGreeting(){ //③增强的横切逻辑
        System.out.println("How are you");
    }
}
```

我们“惊奇”地发现这个切面没有实现任何特殊的接口，它只是一个普通的 POJO。它特殊的地方在于使用了 `@AspectJ` 注解。

首先，在 `PreGreetingAspect` 类定义处标注了 `@AspectJ` 注解，这样，第三方处理程序就可以通过类是否拥有 `@AspectJ` 注解判断其是否为一个切面，如①处所示。

其次，在 `beforeGreeting()` 方法定义处标注了 `@Before` 注解，并为该注解提供了成员值 `"execution(* greetTo(..))"`，如②处所示。②处的注解提供了两个信息：`@Before` 注解表示该增强是前置增强，而成员值是一个 `@AspectJ` 切点表达式。它的意思是：在目标类的 `greetTo()` 方法上织入增强，`greetTo()` 方法可以带任意的入参和任意的返回值。

最后，在③处的 `beforeGreeting()` 方法是增强所使用的横切逻辑，该横切逻辑在目标方法前调用，可以通过图 8-1 描述这种关系。

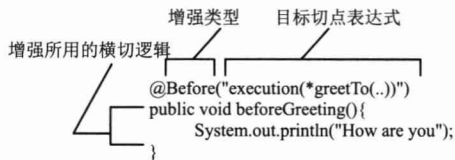


图 8-1 切面的信息构成

`PreGreetingAspect` 类通过注解和代码，将切点、增强类型和增强的横切逻辑糅合到一个类中，使切面的定义浑然天成。`PreGreetingAspect` 类相当于第 7 章中 `BeforeAdvice`、`NameMatchMethodPointcut` 及 `DefaultPointcutAdvisor` 三者联合表达的信息，在掌握 `@AspectJ` 定义切面的技术后，再去使用第 7 章中基于接口的切面技术，一定会顿生“马后桃花马前雪”的感慨。不过，通过基于接口切面的学习可以更加深刻地了解 Spring AOP 的内核技术。

下面通过 `AspectJProxyFactory` 为 `NaiveWaiter` 生成织入 `PreGreetingAspect` 切面的代理，如代码清单 8-5 所示。

代码清单 8-5 AspectJProxyTest

```
package com.smart.aspectj.example;
import org.springframework.aop.aspectj.annotation.AspectJProxyFactory;
import com.smart.NaiveWaiter;
import com.smart.Waiter;
public class AspectJProxyTest {
    Waiter target = new NaiveWaiter();
    AspectJProxyFactory factory = new AspectJProxyFactory();

    //①设置目标对象
    factory.setTarget(target);

    //②添加切面类
    factory.addAspect(PreGreetingAspect.class);

    //③生成织入切面的代理对象
    Waiter proxy = factory.getProxy();
}
```

```

        proxy.greetTo("John");
        proxy.serveTo("John");
    }
}

```

第 7 章通过 ProxyFactory 织入基于接口描述的切面, 此处则使用 AspectJProxyFactory 织入基于@AspectJ 的切面。在①处设置了目标对象; 在②处添加了一个切面类, 该类必须是带@AspectJ 注解的类; 在③处就可以获取织入了切面的代理对象。

接下来直接通过代理对象调用 greetTo()和 serveTo()方法, 将产生以下输出信息:

How are you ①

greet to John...

serving John...

通过①处的输出信息, 可以知道代理对象的 greetTo()方法已经被织入了切面类所定义的增强逻辑。

细心的读者可能会发现, 在 PreGreetingAspect 切面类中定义的前置增强方法和通过 BeforeAdvice 接口定义的前置增强方法不一样, 一个明显的区别就是 PreGreetingAspect 切面类的 beforeGreeting()方法没有任何入参, 而 BeforeAdvice 的接口方法 before(Method method, Object[] args, Object obj)的入参提供了一条访问目标对象方法和入参的途径。难道使用@AspectJ 定义的切面就没有办法访问目标对象连接点的信息了吗? 当然不是, 在本章后面的内容中, 读者将学到相关的知识。

8.3.3 如何通过配置使用@AspectJ 切面

虽然可以通过编程的方式织入切面, 但在一般情况下, 都是通过 Spring 的配置完成切面织入工作的。

```

<!-- ①目标Bean -->
<bean id="waiter" class="com.smart.NaiveWaiter" />
<!-- ②使用了@AspectJ注解的切面类 -->
<bean class="com.smart.aspectj.example.PreGreetingAspect" />
<!-- ③自动代理创建器, 自动将@AspectJ注解切面类织入目标Bean中-->
<bean class="org.springframework.aop.aspectj.annotation.
    &AnnotationAwareAspectJAutoProxyCreator"/>

```

7.5.1 节介绍了几个自动代理创建器, 其中 AnnotationAware AspectJAutoProxy Creator 能够将@AspectJ 注解切面类自动织入目标 Bean 中。在这里, PreGreetingAspect 是使用@AspectJ 注解描述的切面类, 而 NaiveWaiter 是匹配切点的目标类。

如果使用基于 Schema 的 aop 命名空间进行配置, 那么事情就更简单了, 如下:

```

<?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:aop="http://www.springframework.org/schema/aop" ①
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
        http://www.springframework.org/schema/aop ②
        http://www.springframework.org/schema/aop/ spring-aop-4.0.xsd">

```

```

<!--③基于@AspectJ切面的驱动器-->
<aop:aspectj-autoproxy />
<bean id="waiter" class="com.smart.NaiveWaiter" />
<bean class="com.smart.aspectj.example.PreGreetingAspect" />
</beans>

```

首先在配置文件中引入 aop 命名空间，如①和②处所示；然后通过 aop 命名空间的 `<aop:aspectj-autoproxy/>` 自动为 Spring 容器中那些匹配 `@AspectJ` 切面的 Bean 创建代理，完成切面织入。当然，Spring 在内部依旧采用 `AnnotationAwareAspectJAutoProxyCreator` 进行自动代理的创建工作，但具体的实现细节已经被 `<aop:aspectj-autoproxy />` 隐藏起来。

`<aop:aspectj-autoproxy/>` 有一个 `proxy-target-class` 属性，默认为 `false`，表示使用 JDK 动态代理技术织入增强；当配置为 `<aop:aspectj-autoproxy proxy-target-class="true"/>` 时，表示使用 CGLib 动态代理技术织入增强。不过即使 `proxy-target-class` 设置为 `false`，如果目标类没有声明接口，则 Spring 将自动使用 CGLib 动态代理。

8.4 @AspectJ 语法基础

`@AspectJ` 使用 Java 5.0 注解和正规的 AspectJ 的切点表达式语言描述切面，由于 Spring 只支持方法的连接点，所以 Spring 仅支持部分 AspectJ 的切点语言。本节将学习 AspectJ 切点表达式语言，以 AspectJ 5.0 版本为准。

8.4.1 切点表达式函数

AspectJ 5.0 的切点表达式由关键字和操作参数组成，如切点表达式 `execution(* greetTo(..)`，`execution` 为关键字，而 `* greetTo(..)` 为操作参数。在这里，`execution` 代表目标类执行某一方法，而 `* greetTo(..)` 描述目标方法的匹配模式串，二者联合起来表示目标类 `greetTo()` 方法的连接点。为了描述方便，将 `execution()` 称作函数，而将匹配串 `* greetTo(..)` 称作函数的入参。

Spring 支持 9 个 `@AspectJ` 切点表达式函数，它们用不同的方式描述目标类的连接点。根据描述对象的不同，可以大致分为 4 种类型。

- 方法切点函数：通过描述目标类方法的信息定义连接点。
- 方法入参切点函数：通过描述目标类方法入参的信息定义连接点。
- 目标类切点函数：通过描述目标类类型的信息定义连接点。
- 代理类切点函数：通过描述目标类的代理类的信息定义连接点。

这 4 种类型的切点函数通过表 8-1 进行说明。

表 8-1 切点函数

类 别	函 数	入 参	说 明
方法切点函数	execution()	方法匹配模式串	表示满足某一匹配模式的所有目标类方法连接点。如 execution(* greetTo(..))表示所有目标类中的 greetTo()方法
	@annotation()	方法注解类名	表示标注了特定注解的目标类方法连接点。如@annotation(com.smart.anno.NeedTest)表示任何标注了@NeedTest 注解的目标类方法
方法入参切点函数	args()	类名	通过判别目标类方法运行时入参对象的类型定义指定连接点。如 args(com.smart.Waiter)表示所有有且仅有一个按类型匹配于 Waiter 入参的方法
	@args()	类型注解类名	通过判别目标类方法运行时入参对象的类是否标注特定注解来指定连接点。如@args(com.smart.Monitorable)表示任何这样的目标方法：它有一个入参且入参对象的类标注@Monitorable 注解
目标类切点函数	within()	类名匹配串	表示特定域下的所有连接点。如 within(com.smart.service.*)表示 com.smart.service 包中的所有连接点，即包中所有类的所有方法；而 within(com.smart.service.*Service)表示在 com.smart.service 包中所有以 Service 结尾的类的所有连接点
	target()	类名	假如目标类按类型匹配于指定类，则目标类的所有连接点匹配这个切点。如通过 target(com.smart.Waiter)定义的切点、Waiter 及 Waiter 实现类 NaiveWaiter 中的所有连接点都匹配该切点
	@within()	类型注解类名	假如目标类按类型匹配于某个类 A，且类 A 标注了特定注解，则目标类的所有连接点匹配这个切点。如@within(com.smart.Monitorable)定义的切点，假如 Waiter 类标注了@Monitorable 注解，则 Waiter 及 Waiter 实现类 NaiveWaiter 的所有连接点都匹配这个切点
	@target()	类型注解类名	假如目标类标注了特定注解，则目标类的所有连接点都匹配该切点。如@target (com.smart.Monitorable)，假如 NaiveWaiter 标注了@Monitorable，则 NaiveWaiter 的所有连接点都匹配这个切点
代理类切点函数	this()	类名	代理类按类型匹配于指定类，则被代理的目标类的所有连接点都匹配该切点。这个函数比较难以理解，这里暂不举例，留待后面详解

除了表 8-1 中所列的函数外，@AspectJ 还有 call()、initialization()、preinitialization()、staticinitialization()、get()、set()、handler()、adviceexecution()、withincode()、cflow()、cflowbelow()、if()、@this()及@withincode()等函数，这些函数在 Spring 中不能使用，否则会抛出 IllegalArgumentException 异常。在不特别声明的情况下，本书所讲的@AspectJ 函数均指表 8-1 中所列的函数。

8.4.2 在函数入参中使用通配符

- 有些函数的入参可以接受通配符，@AspectJ 支持 3 种通配符。
- ❑ *：匹配任意字符，但它只能匹配上下文中的一个元素。
 - ❑ ..：匹配任意字符，可以匹配上下文中的多个元素，但在表示类时，必须和*联合使用，而在表示入参时则单独使用。

- +: 表示按类型匹配指定类的所有类, 必须跟在类名后面, 如 `com.smart.Car+`。继承或扩展指定类的所有类, 同时还包括指定类本身。
- @AspectJ 函数按其是否支持通配符及支持的程度, 可以为以下 3 类。
 - 支持所有通配符: `execution()` 和 `within()`, 如 `within(com.smart.*)`、`within(com.smart.service..*.*Service+)` 等。
 - 仅支持 “+” 通配符: `args()`、`this()` 和 `target()`, 如 `args(com.smart.Waiter+)`、`target(java.util.List+)` 等。虽然这 3 个函数可以支持 “+” 通配符, 但其意义不大, 因为对于这些函数来说, 使用和不使用 “+” 都是一样的, 如 `target(com.smart.Waiter+)` 和 `target(com.smart.aspectj.Waiter)` 是等价的。
 - 不支持通配符: `@args()`、`@within()`、`@target()` 和 `@annotation()`, 如 `@argscom.smart.anno.NeedTest)` 和 `@within(com.smart.anno.NeedTest)`。

此外, `args()`、`this()`、`target()`、`@args()`、`@within()`、`@target()` 和 `@annotation()` 这 7 个函数除了可以指定类名外, 也可以指定变量名, 并将目标对象中的变量绑定到增强的方法中。关于参数绑定的内容将在 8.6 节介绍, 而函数的其他内容将在 8.5 节详解。

8.4.3 逻辑运算符

切点表达式由切点函数组成, 切点函数之间还可以进行逻辑运算, 组成复合切点。Spring 支持以下切点运算符。

- &&: 与操作符, 相当于切点的交集运算。如果在 Spring 的 XML 配置文件中使用时切点表达式, 由于 & 是 XML 特殊字符, 所以使用转义字符 `&` 表示。为了方便使用, Spring 提供了一个等效的运算符 “and”。如 `within(com.smart.*) and args(String)` 表示在 `com.smart` 包下所有类 (当前包及子孙包) 拥有一个 `String` 入参的方法。
- ||: 或操作符, 相当于切点的并集运算, `or` 是等效的操作符。如 `within(com.smart.*) || args(String)` 表示在 `com.smart` 包下所有类的方法, 或者所有拥有一个 `String` 入参的方法。
- !: 非操作符, 相当于切点的反集运算, `not` 是等效的操作符。如 `!within(com.smart.*)` 表示所有不在 `com.smart` 包下的方法。

在标准的 @AspectJ 中并不提供 `and`、`or` 和 `not` 操作符, 它们是 Spring 为了在 XML 配置文件中方便定义切点表达式而特意添加的等价操作符。有意思的是, 和一般的语法表达不一样, 在 Spring 中使用 `and`、`or` 和 `not` 操作符时, 允许不在前后添加空格, 如 `within(com.smart.*) andnotargs(String)` 和 `within(com.smart.*) and not args(String)` 拥有相同的效果。虽然 Spring 接受这种表示方式, 但为了保证程序的可读性, 最好还是采用传统的习惯, 在操作符的前后添加空格。

**提示**

如果 not 位于切点表达式的开头，则必须在开头添加一个空格，否则将产生解析错误。如“not within(com.smart.*)”将产生解析错误，这应该是 Spring 解析的一个 Bug，在表达式开头添加空格后则可以通过解析，即“ `not within (com.smart.*)`”。

8.4.4 不同增强类型

第 7 章使用不同的接口描述各种增强类型，@AspectJ 也为各种增强类型提供了不同的注解类，它们位于 org.aspectj.lang.annotation.*包中。这些注解类拥有若干个成员，可以通过这些成员完成定义切点信息、绑定连接点参数等操作。此外，这些注解的存留期限都是 RetentionPolicy.RUNTIME，标注目标都是 ElementType.METHOD。下面逐一学习@AspectJ 所提供的几个增强注解。

1. @Before

前置增强，相当于 BeforeAdvice。Before 注解类拥有两个成员。

- ☐ value: 该成员用于定义切点。
- ☐ argNames: 由于无法通过 Java 反射机制获取方法入参名，所以如果在 Java 编译时未启用调试信息，或者需要在运行期解析切点，就必须通过这个成员指定注解所标注增强方法的参数名（注意二者名字必须完全相同），多个参数名用逗号分隔。

2. @AfterReturning

后置增强，相当于 AfterReturningAdvice。AfterReturning 注解类拥有 4 个成员。

- ☐ value: 该成员用于定义切点。
- ☐ pointcut: 表示切点的信息。如果显式指定 pointcut 值，那么它将覆盖 value 的设置值，可以将 pointcut 成员看作 value 的同义词。
- ☐ returning: 将目标对象方法的返回值绑定给增强的方法。
- ☐ argNames: 如前所述。

3. @Around

环绕增强，相当于 MethodInterceptor。Around 注解类拥有两个成员。

- ☐ value: 该成员用于定义切点。
- ☐ argNames: 如前所述。

4. @AfterThrowing

抛出增强，相当于 ThrowsAdvice。AfterThrowing 注解类拥有 4 个成员。

- ☐ value: 该成员用于定义切点。
- ☐ pointcut: 表示切点的信息。如果显式指定 pointcut 值，那么它将覆盖 value 的设置值。可以将 pointcut 成员看作 value 的同义词。

- ☐ throwing: 将抛出的异常绑定到增强方法中。
- ☐ argNames: 如前所述。

5. @After

Final 增强, 不管是抛出异常还是正常退出, 该增强都会得到执行。该增强没有对应的增强接口, 可以把它看成 ThrowsAdvice 和 AfterReturningAdvice 的混合物, 一般用于释放资源, 相当于 try{}finally{} 的控制流。After 注解类拥有两个成员。

- ☐ value: 该成员用于定义切点。
- ☐ argNames: 如前所述。

6. @DeclareParents

引介增强, 相当于 IntroductionInterceptor。DeclareParents 注解类拥有两个成员。

- ☐ value: 该成员用于定义切点, 它表示在哪个目标类上添加引介增强。
- ☐ defaultImpl: 默认的实现类。

除引介增强外, 其他增强都很容易理解, 将在本章后续内容中统一讲述。引介增强的使用比较特别, 为此特别在 8.4.5 节准备了一个实例。

8.4.5 引介增强用法

请看以下两个接口及其实现类, 如图 8-2 所示。

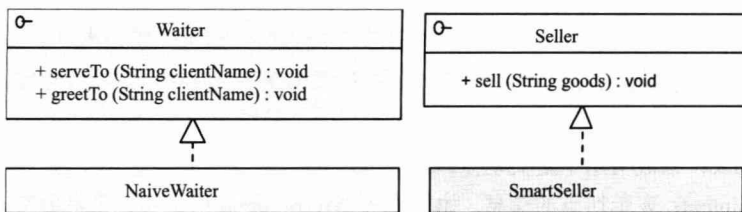


图 8-2 Waiter 和 Seller

假设希望 NaiveWaiter 能够同时充当售货员的角色, 即通过切面技术为 NaiveWaiter 新增 Seller 接口的实现。在第 7 章中已经做过相似的工作, 因此这里不在概念上作更多的解释, 下面着手用 @AspectJ 的引介增强来实现这一功能, 如代码清单 8-6 所示。

代码清单 8-6 EnableSellerAspect

```

package com.smart.aspectj.basic;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.DeclareParents;
import com.smart.Seller;
import com.smart.SmartSeller;

@Aspect
public class EnableSellerAspect {

    @DeclareParents(value="com.smart.NaiveWaiter", //①为NaiveWaiter添加接口实现

```

```

        defaultImpl=SmartSeller.class) //②默认的实现类
    public Seller seller; //③要实现的目标接口
}

```

在 EnableSellerAspect 切面中, 通过@DeclareParents 为 NaiveWaiter 添加了一个需要实现的 Seller 接口, 并指定其默认实现类为 SmartSeller, 然后通过切面技术将 SmartSeller 融合到 NaiveWaiter 中, 这样 NaiveWaiter 就实现了 Seller 接口。

在 Spring 配置文件中配置好切面和 NaiveWaiter Bean, 如下:

```

<aop:aspectj-autoproxy/>
<bean id="waiter" class="com.smart.NaiveWaiter"/>
<bean class="com.smart.aspectj.basic.EnableSellerAspect"/>

```

运行以下测试代码:

```

package com.smart.aspectj.basic;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.smart.Seller;
import com.smart.Waiter;
public class DeclaredParentsTest {
    public static void main(String[] args) {
        String configPath = "com/smart/aspectj/basic/beans.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
        Waiter waiter = (Waiter)ctx.getBean("waiter");
        waiter.greetTo("John");
        Seller seller = (Seller)waiter; //①可以成功地进行强制类型转换
        seller.sell("Beer", "John");
    }
}

```

代码成功执行, 并输出以下信息:

```

NaiveWaiter:greet to John...
SmartSeller: sell Beer to John...

```

可见, NaiveWaiter 已经成功地新增了 Seller 接口的实现。

8.5 切点函数详解

切点函数是 AspectJ 表达式语言的核心, 也是使用@AspectJ 进行切面定义的难点, 本节通过具体实例来学习切点函数的相关知识。为了方便讲解, 假设可供被增强的目标类包括 7 个类, 这些目标类都位于 com.smart.*包中, 如图 8-3 所示。

在这些类中, 除 SmartSeller#showGoods()方法是 protected 外, 其他的方法都是 public。

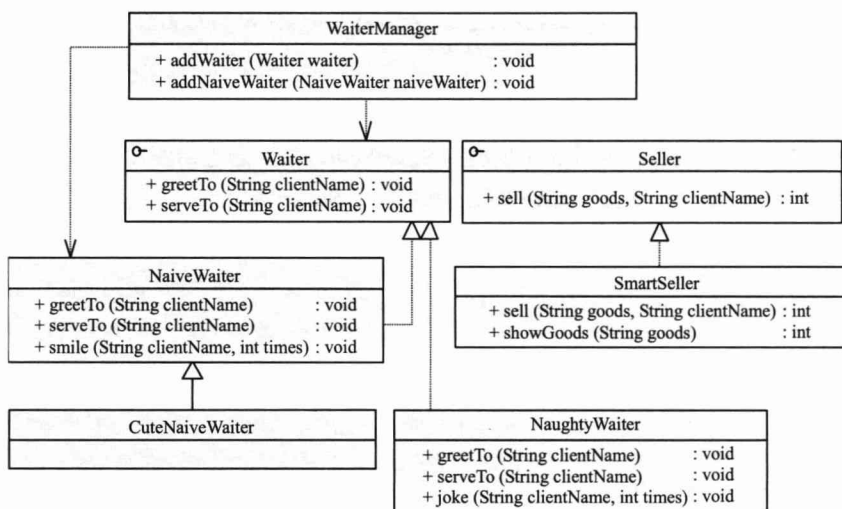


图 8-3 Waiter 和 Seller 类图

8.5.1 @annotation()

@annotation 表示标注了某个注解的所有方法。通过一个实例说明 @annotation() 的用法。TestAspect 定义了一个后置增强切面，该增强将应用到标注了 NeedTest 的目标方法中。

```

package com.smart.aspectj.fun;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.AfterReturning;
@Aspect
public class TestAspect {
    @AfterReturning("@annotation(com.smart.anno.NeedTest)") //①后置增强切面
    public void needTestFun(){
        System.out.println("needTestFun() executed!");
    }
}
  
```

假设 NaughtyWaiter#greetTo() 方法标注了 @NeedTest 注解，而 NaiveWaiter#greetTo() 方法没有标注 @NeedTest 注解，如代码清单 8-7 所示。

代码清单 8-7 标注了 @NeedTest 注解的 NaughtyWaiter

```

package com.smart;
import com.smart.anno.NeedTest;
public class NaughtyWaiter implements Waiter {
    @NeedTest
    public void greetTo(String clientName) {
        System.out.println("NaughtyWaiter:greet to "+clientName+"...");
    }
    ...
}
  
```

通过 Spring 配置自动应用切面。

```
<aop:aspectj-autoproxy />
<bean id="naiveWaiter" class="com.smart.NaiveWaiter" />
<bean id="naughtyWaiter" class="com.smart.NaughtyWaiter" />
<bean class="com.smart.aspectj.fun.TestAspect" />
```

运行代码清单 8-8 中的代码。

代码清单 8-8 PointcutFunTest: 测试代码

```
package com.smart.aspectj.fun;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.smart.Seller;
import com.smart.Waiter;
public class PointcutFunTest{
    @Test
    public void pointcut {
        String configPath = "com/smart/aspectj/fun/beans.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
        Waiter naiveWaiter = (Waiter) ctx.getBean("naiveWaiter");
        Waiter naughtyWaiter = (Waiter) ctx.getBean("naughtyWaiter");
        naiveWaiter.greetTo("John"); //①该方法未被织入增强
        naughtyWaiter.greetTo("Tom"); //②该方法被织入增强
    }
}
```

输出以下信息：

```
NaiveWaiter:greet to John...
NaughtyWaiter:greet to Tom... ① ← 对应NaughtyWaiter 的 greetTo() 方法
needTestFun() executed!
```

从以上信息中可以获知，切面被正确地织入 NaughtyWaiter#greetTo()方法中。

8.5.2 execution()

execution()是最常用的切点函数，其语法如下：

```
execution(<修饰符模式>? <返回类型模式> <方法名模式>(<参数模式>) <异常模式>?)
```

除了返回类型模式、方法名模式和参数模式外，其他项都是可选的。与其直接讲解该方法的使用规则，还不如通过一个个具体的例子来理解。下面给出各种使用 execution()函数的实例。

1. 通过方法签名定义切点

- execution(public * *(..)): 匹配所有目标类的 public 方法，但不匹配 SmartSeller 和 protected void showGoods()方法。第一个*代表返回类型；第二个*代表方法名；而..代表任意入参的方法。
- execution(* *To(..)): 匹配目标类所有以 To 为后缀的方法，它匹配 NaiveWaiter 类和 NaughtyWaiter 的 greetTo()和 serveTo()方法。第一个*代表返回类型；而*To 代表任意以 To 为后缀的方法。

2. 通过类定义切点

- ❑ `execution(* com.smart.Waiter.*(..)`: 匹配 `Waiter` 接口的所有方法，它匹配 `NaiveWaiter` 和 `NaughtyWaiter` 类的 `greetTo()` 和 `serveTo()` 方法。第一个 `*` 代表返回任意类型；`com.smart.Waiter.*` 代表 `Waiter` 接口中的所有方法。
- ❑ `execution(* com.smart.Waiter+.*(..)`: 匹配 `Waiter` 接口及其所有实现类的方法，它不但匹配 `NaiveWaiter` 和 `NaughtyWaiter` 类的 `greetTo()` 和 `serveTo()` 这两个 `Waiter` 接口定义的方法，同时还匹配 `NaiveWaiter#smile()` 和 `NaughtyWaiter#joke()` 这两个不在 `Waiter` 接口中定义的方法。

3. 通过类包定义切点

在类名模式串中，“`.*`”表示包下的所有类，而“`..*`”表示包、子孙包下的所有类。

- ❑ `execution(* com.smart.*(..)`: 匹配 `com.smart` 包下所有类的所有方法。
- ❑ `execution(* com.smart..*(..)`: 匹配 `com.smart` 包、子孙包下所有类的所有方法，如 `com.smart.dao`、`com.smart.service` 及 `com.smart.dao.user` 包下所有类的所有方法都匹配。当“`..`”出现在类名中时，后面必须跟“`*`”，表示包、子孙包下的所有类。
- ❑ `execution(* com..*.*Dao.find*(..)`: 匹配包名前缀为 `com` 的任何包下类名后缀为 `Dao` 的方法，方法名必须以 `find` 为前缀。如 `com.smart.UserDao#findByUserId()`、`com.smart.dao.ForumDao#findById()` 方法都匹配切点。

4. 通过方法入参定义切点

切点表达式中的方法入参部分比较复杂，可以使用“`*`”和“`..`”通配符。其中，“`*`”表示任意类型的参数；而“`..`”表示任意类型的参数且参数个数不限。

- ❑ `execution(* joke(String,int)`: 匹配 `joke(String,int)` 方法，且 `joke()` 方法的第一个入参是 `String`，第二个入参是 `int`。它匹配 `NaughtyWaiter#joke(String,int)` 方法。如果方法中的入参类型是 `java.lang` 包下的类，则可以直接使用类名；否则必须使用全限定类名，如 `joke(java.util.List,int)`。
- ❑ `execution(* joke(String,*))`: 匹配目标类中的 `joke()` 方法，该方法的第一个入参为 `String`，第二个入参可以是任意类型，如 `joke(String s1,String s2)` 和 `joke(String s1,double d2)` 都匹配，但 `joke(String s1,double d2,String s3)` 不匹配。
- ❑ `execution(* joke(String,...))`: 匹配目标类中的 `joke()` 方法，该方法的第一个入参为 `String`，后面可以有任意个入参且入参类型不限，如 `joke(String s1)`、`joke(String s1,String s2)` 和 `joke(String s1,double d2,String s3)` 都匹配。
- ❑ `execution(* joke(Object+))`: 匹配目标类中的 `joke()` 方法，方法拥有一个入参，且入参是 `Object` 类型或该类的子类。它匹配 `joke(String s1)` 和 `joke(Client c)`。如果定义的切点是 `execution(* joke(Object))`，则只匹配 `joke(Object object)`，而不匹配 `joke(String cc)` 或 `joke(Client c)`。

8.5.3 args()和@args()

args()函数的入参是类名，而@args()函数的入参必须是注解类的类名。虽然 args()允许在类名后使用“+”通配符，但该通配符在此处没有意义，添加和不添加效果都一样。

1. args()

该函数接收一个类名，表示目标类方法入参对象是指定类（包含子类）时，切点匹配，如下面的例子：

```
args(com.smart.Waiter)
```

表示运行时入参是 Waiter 类型的方法，它和 execution(* *(com.smart.Waiter))的区别在于后者是针对类方法的签名而言的，而前者则针对运行时的入参类型而言。如 args(com.smart.Waiter)既匹配 addWaiter(Waiter waiter)，又匹配 addNaiveWaiter(NaiveWaiter naiveWaiter)；而 execution(* *(com.smart.Waiter))只匹配 addWaiter(Waiter waiter)。实际上，args(com.smart.Waiter)等价于 execution(* *(com.smart.Waiter+))，当然也等价于 args(com.smart.Waiter+)。

2. @args()

该函数接收一个注解类的类名，当方法的运行时入参对象标注了指定的注解时，匹配切点。这个切点函数的匹配规则不太容易理解，通过图 8-4 对此进行详细讲解。

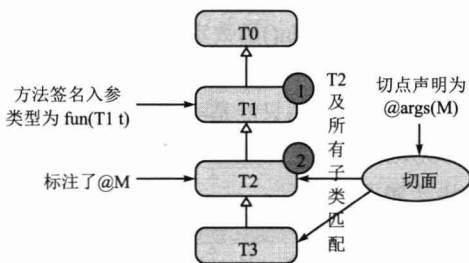


图 8-4 @arg(M)匹配示意图 (1)

T0、T1、T2、T3 具有如图 8-4 所示的继承关系，假设目标类方法的签名为 fun(T1 t)，它的入参为 T1，而切面的切点定义为 @args(M)，T2 类标注了 @M。当 fun(T1 t)的传入对象是 T2 或 T3 时，方法匹配 @args(M) 声明所定义的切点。

假设方法签名是 fun(T1 t)，入参为 T1，而标注 @M 的类是 T0，当 fun(T1 t)传入 T1、T2、T3 的实例时，均不匹配切点 @args(M)。

在类的继承树中，①处为方法签名中入参类型在类继承树中的位置，称之为入参类型点；而②处为标注了 @M 注解的类在类继承树中的位置，称之为注解点。判断方法在运行时是否匹配 @args(M) 切点，可以根据①和②在类继承树中的相对位置来判别。

(1) 如果在类继承树中注解点②高于入参类型点①，则该目标方法不可能匹配切点 @args(M)，如图 8-5 所示。

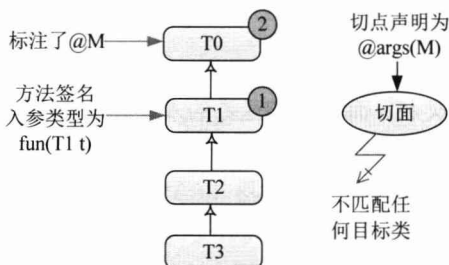


图 8-5 @arg(M)匹配示意图 (2)

(2) 如果在类继承树中注解点②低于入参类型点①，则注解点所在类及其子孙类作为方法入参时，该方法匹配切点@args(M)，如图 8-4 所示。

下面举一个具体的例子。假设定义这样的切点：@args(com.smart.Monitorable)，如果 NaiveWaiter 标注了@Monitorable，则对于 WaiterManager#addWaiter(Waiter w)方法来说，如果入参是 NaiveWaiter 或其子类对象，则该方法匹配切点；如果入参是 NaughtyWaiter 对象，则不匹配切点。如果 Waiter 标注了@Monitorable，但 NaiveWaiter 未标注@Monitorable，则 WaiterManager#addNaiveWaiter(NaiveWaiter w)不匹配切点，这是因为注解点（Waiter）高于入参类型点（NaiveWaiter）。

8.5.4 within()

通过类匹配模式串声明切点，within()函数定义的连接点是针对目标类而言的，而非针对运行期对象的类型而言，这一点和 execution()是相同的。但和 execution()函数不同的是，within()所指定的连接点最小范围只能是类，而 execution()所指定的连接点可以大到包，小到方法入参。所以从某种意义上说，execution()函数的功能涵盖了 within()函数的功能。within()函数的语法如下：

within(<类匹配模式>)

形如 within(com.smart.NaiveWaiter)，是 within()函数所能表达的最小粒度。如果试图用 within()匹配方法级别的连接点，如 within(com.smart.NaiveWaiter.greet*)，那么将会产生解析错误。

下面是一些使用 within()函数的实例。

- within(com.smart.NaiveWaiter): 匹配目标类 NaiveWaiter 的所有方法。如果切点调整为 within(com.smart.Waiter)，则 NaiveWaiter 和 NaughtyWaiter 中的所有方法都不匹配。而 Waiter 本身是接口，不可能实例化，所以 within(com.smart.Waiter)的声明是无意义的。
- within(com.smart.*): 匹配 com.smart 包中的所有类，但不包括子孙包，所以 com.smart.service 包中类的方法不匹配这个切点。
- within(com.smart.*.*): 匹配 com.smart 包及子孙包中的类，所以 com.smart.service、com.smart.dao 及 com.smart.service.fourm 等包中所有类的方法都匹配这个切点。

8.5.5 @within()和@target()

除@annotation()和@args()函数外，还有另外两个用于注解的切点函数，分别是@target()和@within()。和@annotation()及@args()函数一样，它们也只接受注解类名作为入参。其中，@target(M)匹配任意标注了@M的目标类，而@within(M)匹配标注了@M的类及子孙类。

@target(M)切点的匹配规则如图 8-6 所示。

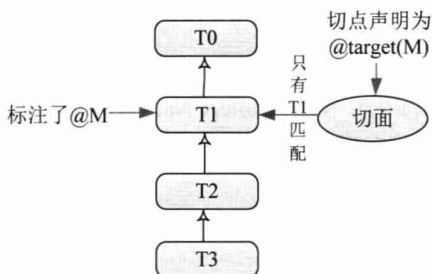


图 8-6 @target(M)匹配目标类示意图

假设 NaiveWaiter 标注了@Monitorable，而其子类 CuteNaiveWaiter 没有标注@Monitorable，则@target(com.smart.Monitorable)匹配 NaiveWaiter 类的所有方法，但不匹配 CuteNaiveWaiter 类的方法。

@within(M)切点的匹配规则如图 8-7 所示。

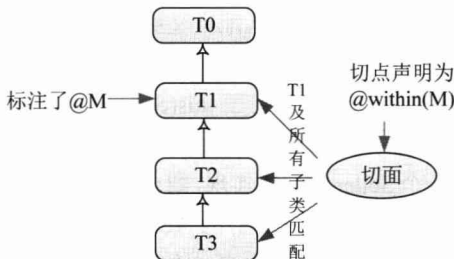


图 8-7 @within(M)匹配目标类示意图

假设 NaiveWaiter 标注了@Monitorable，而其子类 CuteNaiveWaiter 没有标注@Monitorable，则@within(com.smart.Monitorable)不但匹配 NaiveWaiter 类中的所有方法，也匹配 CuteNaiveWaiter 类中的所有方法。

有一个特别值得注意的地方是，如果标注@M注解的是一个接口，则所有实现该接口的类并不匹配@within(M)。假设 Waiter 标注了@Monitorable 注解，但 NaiveWaiter、NaughtyWaiter 及 CuteNaiveWaiter 这些接口实现类都没有标注@Monitorable，则@within(com.smart.Monitorable)和@target(com.smart.Monitorable)都不匹配 NaiveWaiter、NaughtyWaiter 及 CuteNaiveWaiter。这是因为@within()、@target()及@annotation()函数都是针对目标类而言的，而非针对运行时的引用类型而言的，这点区别需要在开发中特别注意。

8.5.6 target()和 this()

target()切点函数通过判断目标类是否按类型匹配指定类来决定连接点是否匹配,而 this()函数则通过判断代理类是否按类型匹配指定类来决定是否和切点匹配。二者都仅接受类名的入参,虽然类名可以带“+”通配符,但对于这两个函数来说,使用与不使用“+”通配符,效果完全相同。

1. target()

target(M)表示如果目标类按类型匹配于 M,则目标类的所有方法都匹配切点。下面通过一些例子理解 target(M)的匹配规则。

- ❑ target(com.smart.Waiter): NaiveWaiter、NaughtyWaiter 及 CuteNaiveWaiter 的所有方法都匹配切点,包括那些未在 Waiter 接口中定义的方法,如 NaiveWaiter#simple() 和 NaughtyWaiter#joke()方法。
- ❑ target(com.smart.Waiter+): 与 target(com.smart.Waiter)是等价的。

2. this()

根据 Spring 的官方文档, this()函数判断代理对象的类是否按类型匹配于指定类,如果匹配,则代理对象的所有连接点匹配切点。但通过实验发现,实际情况和文档有所出入。如声明一个切点 this(com.smart.NaiveWaiter),如果不使用 CGLib 动态代理,则生成的代理对象属于 Waiter 类型,而非 NaiveWaiter 类型,这一点可以简单地通过 instanceof 操作符进行判断。但是,我们发现 NaiveWaiter 中所有的方法都被织入了增强。

一般情况下,使用 this()和 target()来匹配定义切点,二者是等效的。

(1) target(com.smart.Waiter)等价于 this(com.smart.Waiter)。

(2) target(com.smart.NaiveWaiter)等价于 this(com.smart.NaiveWaiter)。

二者的区别体现在通过引介切面产生代理对象时的具体表现。如果通过 8.4.5 节的方法为 NaiveWaiter 引介一个 Seller 接口的实现,则 this(com.smart.Seller)匹配 NaiveWaiter 代理对象的所有方法,包括 NaiveWaiter 本身的 greetTo()、serverTo()方法,以及通过 Seller 接口引入的 sell()方法;而 target(com.smart.Seller)不匹配通过引介切面产生的 NaiveWaiter 代理对象。

下面通过具体的实例来了解这一微妙的区别。EnableSellerAspect 是为 NaiveWaiter 添加 Seller 接口实现的引介切面,如下:

```
package com.smart.aspectj.fun;
...
@Aspect
public class EnableSellerAspect {
    @DeclareParents(value = "com.smart.NaiveWaiter",
                    defaultImpl = SmartSeller.class)
    public static Seller seller;
}
```

TestAspect 是通过判断运行期代理对象所属类型来定义切点的切面,如代码清单 8-9 所示。

代码清单 8-9 TestAspect: 通过this()指定切点

```
package com.smart.aspectj.fun;
...
@Aspect
public class TestAspect {
    //①后置增强, 织入任何运行期对象为Seller类型的Bean中
    @AfterReturning("this(com.smart.Seller)")
    public void thisTest(){
        System.out.println("thisTest() executed!");
    }
}
```

在 Spring 中配置这两个切面和 NaiveWaiter, 如下:

```
<aop:aspectj-autoproxy/>
<bean id="naiveWaiter" class="com.smart.NaiveWaiter" />
<bean class="com.smart.aspectj.fun.EnableSellerAspect"/>
<bean class="com.smart.aspectj.fun.TestAspect" />
```

EnableSellerAspect 切面为 NaiveWaiter 引介 Seller 接口产生一个实现 Seller 接口的代理对象, TestAspect 在判断出 NaiveWaiter 这个代理对象实现 Seller 接口后, 就将其切面织入这个代理对象中, 所以最终 NaiveWaiter 的代理对象其实共织入了两个切面。在这里, 我们忽略了多切面织入顺序的问题, 读者将在本章后续的内容中了解到这方面的知识。

运行以下测试代码:

```
String configPath = "com/smart/aspectj/fun/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter naiveWaiter = (Waiter) ctx.getBean("naiveWaiter");
naiveWaiter.greetTo("John");
naiveWaiter.serveTo("John");
((Seller)naiveWaiter).sell("Beer", "John");
```

输出以下信息:

```
NaiveWaiter:greet to John...
thisTest() executed!
NaiveWaiter:serving John...
thisTest() executed!
SmartSeller: sell Beer to John...
thisTest() executed!
```

可见代理对象的 3 个方法都织入了代码清单 8-9 中通过 this()函数定义的切面。

8.6 @AspectJ 进阶

@AspectJ 可以使用切点函数定义切点, 还可以使用逻辑运算符对切点进行复合运算得到复合切点。为了在切面中重用切点, 还可以对切点进行命名, 以便在其他地方引

用定义过的切点。当一个连接点匹配多个切点时，需要考虑织入顺序的问题，另外一个重要的问题是如何在增强中访问连接点上下文的信息。

8.6.1 切点复合运算

使用切点复合运算符，将拥有强大而灵活的切点表达能力。以下是一个使用了复合切点的切面，如代码清单 8-10 所示。

代码清单 8-10 TestAspect: 切点复合运算

```
package com.smart.aspectj.advanced;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
@Aspect
public class TestAspect {
    @After("within(com.smart.*) "
        + " && execution(* greetTo(..))") //①与运算
    public void greetToFun() {
        System.out.println("--greetToFun() executed!--");
    }

    @Before("!target(com.smart.NaiveWaiter) "+
        "&& execution(* serveTo(..))") //②非与运算
    public void notServeInNaiveWaiter() {
        System.out.println("--notServeInNaiveWaiter() executed!--");
    }

    @AfterReturning("target(com.smart.Waiter) || "+
        " target(com.smart.Seller)") //③或运算
    public void waiterOrSeller(){
        System.out.println("--waiterOrSeller() executed!--");
    }
}
```

在①处，通过&&运算符定义了一个匹配 com.smart 包中所有 greetTo()方法的切点；在②处，通过!和&&运算符定义了一个匹配所有 serveTo()方法并且该方法不位于 NaiveWaiter 目标类的切点；在③处，通过||运算符定义了一个匹配 Waiter 和 Seller 接口实现类所有连接点的切点。

8.6.2 命名切点

在前面所举的例子中，切点直接声明在增强方法处，这种切点声明方式称为匿名切点，匿名切点只能在声明处使用。如果希望在其他地方重用一個切点，则可以通过 @Pointcut 注解及切面类方法对切点进行命名。以下是一个具体的实例，如代码清单 8-11 所示。

代码清单 8-11 TestNamePointcut

```

package com.smart.aspectj.advanced;
import org.aspectj.lang.annotation.Pointcut;
public class TestNamePointcut {
    @Pointcut("within(com.smart.*)") ①
    private void inPackage() { }

    @Pointcut("execution(* greetTo(..)") ②
    protected void greetTo() { }

    @Pointcut("inPackage() and greetTo()") ③
    public void inPkgGreetTo() { }
}

```

通过注解方法 `inPackage()` 对该切点进行命名, 方法可见域修饰符为 `private`, 表明该命名切点只能在本切面类中使用

通过注解方法 `greetTo()` 对该切点进行命名, 方法可见域修饰符为 `protected`, 表明该命名切点可以在当前包中的切面类、子切面类中使用

引用命名切点定义的切点, 本切点也是命名切点, 它对应的可见域为 `public`

在代码清单 8-11 中定义了 3 个命名切点, 命名切点的使用类方法作为切点的名称, 此外方法的访问修饰符还控制了切点的可引用性, 这种可引用性和类方法的可访问性相同, 如 `private` 的切点只能在本类中引用, `public` 的切点可以在任何类中引用。命名切点仅利用方法名及访问修饰符的信息, 所以习惯上方法的返回类型为 `void`, 并且方法体为空。可以通过图 8-8 更直观地了解命名切点的结构。

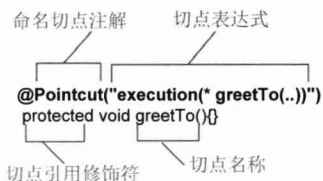


图 8-8 命名切点的结构

在③处, `inPkgGreetTo()` 的切点引用了同类中的 `greetTo()` 切点, 而 `inPkgGreetTo()` 切点可以被任何类引用。用户还可以扩展 `TestNamePointcut` 类, 通过类的继承关系定义更多的切点。

命名切点定义好后, 就可以在定义切面时通过名称引用切点, 请看下面的实例:

```

package com.smart.aspectj.advanced;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
@Aspect
public class TestAspect {
    @Before("TestNamePointcut.inPkgGreetTo()") //①
    public void pkgGreetTo() {
        System.out.println("--pkgGreetTo() executed!--");
    }
    @Before("!target(com.smart.NaiveWaiter) && TestNamePointcut.inPkgGreetTo()") //②
    public void pkgGreetToNotNaiveWaiter() {
        System.out.println("--pkgGreetToNotNaiveWaiter() executed!--");
    }
}

```

在①处, 引用了 `TestNamePointcut.inPkgGreetTo()` 切点; 而在②处, 在复合运算中使用了命名切点。

8.6.3 增强织入的顺序

一个连接点可以同时匹配多个切点，切点对应的增强在连接点上的织入顺序是如何安排的呢？这个问题需要分 3 种情况讨论。

- ❑ 如果增强在同一个切面类中声明，则依照增强在切面类中定义的顺序进行织入。
- ❑ 如果增强位于不同的切面类中，且这些切面类都实现了 `org.springframework.core.Ordered` 接口，则由接口方法的顺序号决定（顺序号小的先织入）。
- ❑ 如果增强位于不同的切面类中，且这些切面类没有实现 `org.springframework.core.Ordered` 接口，则织入的顺序是不确定的。

可以通过图 8-9 描述这种织入的规则。切面类 A 和 B 都实现了 `Ordered` 接口，切面类 A 对应序号为 1，切面类 B 对应序号为 2，切面类 A 按顺序定义了 3 个增强，切面类 B 按顺序定义两个增强，这 5 个增强对应的切点都匹配某个目标类的连接点，则增强织入的顺序为图中虚线所示。

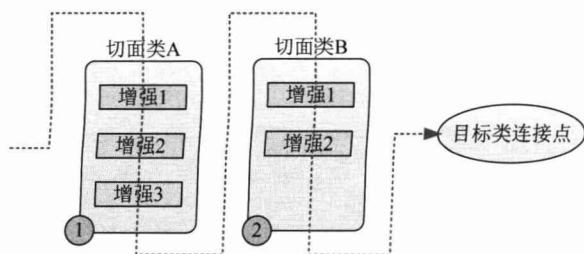


图 8-9 增强织入顺序

8.6.4 访问连接点信息

AspectJ 使用 `org.aspectj.lang.JoinPoint` 接口表示目标类连接点对象。如果是环绕增强，则使用 `org.aspectj.lang.ProceedingJoinPoint` 表示连接点对象，该类是 `JoinPoint` 的子接口。任何增强方法都可以通过将第一个入参声明为 `JoinPoint` 访问连接点上下文信息。先来了解一下这两个接口的主要方法。

1. JoinPoint

- ❑ `java.lang.Object[] getArgs()`: 获取连接点方法运行时的入参列表。
- ❑ `Signature getSignature()`: 获取连接点的方法签名对象。
- ❑ `java.lang.Object getTarget()`: 获取连接点所在的目标对象。
- ❑ `java.lang.Object getThis()`: 获取代理对象本身。

2. ProceedingJoinPoint

`ProceedingJoinPoint` 继承于 `JoinPoint` 子接口，它新增了两个用于执行连接点方法的方法。

- ❑ `java.lang.Object proceed()` throws `java.lang.Throwable`: 通过反射执行目标对象的连接点处的方法。
- ❑ `java.lang.Object proceed(java.lang.Object[] args)` throws `java.lang.Throwable`: 通过反射执行目标对象连接点处的方法，不过使用新的入参替换原来的入参。

来看一个具体的实例，如代码清单 8-12 所示。

代码清单 8-12 TestAspect: 访问连接点对象

```
package com.smart.aspectj.advanced;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
@Aspect
public class TestAspect {
    @Around("execution(* greetTo(..)) && target(com.smart.NaiveWaiter)") ①
    public void joinPointAccess(ProceedingJoinPoint pjp) throws Throwable{ ②
        System.out.println("-----joinPointAccess-----");

        System.out.println("args[0]:"+pjp.getArgs()[0]);
        System.out.println("signature:"+pjp.getTarget().getClass());
        pjp.proceed(); ④
        System.out.println("-----joinPointAccess-----");
    }
}
```

环绕增强
声明连接点入参
访问连接点信息
通过连接点执行
目标对象方法

在①处声明了一个环绕增强；在②处增强方法的第一个入参声明为 `ProceedingJoinPoint` 类型（注意一定要在第一个位置）；在③处通过连接点对象 `pjp` 访问连接点信息；在④处通过连接点调用目标对象的方法。

执行以下测试代码：

```
String configPath = "com/smart/aspectj/advanced/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter naiveWaiter = (Waiter) ctx.getBean("naiveWaiter");
naiveWaiter.greetTo("John");
```

输出以下信息：

```
-----joinPointAccess-----
args[0]:John
signature:class com.smart.NaiveWaiter
NaiveWaiter:greet to John... ①
-----joinPointAccess-----
```

对应 `pjp.proceed()`;

8.6.5 绑定连接点方法入参

在介绍切点函数时说过 `args()`、`this()`、`target()`、`@args()`、`@within()`、`@target()` 和 `@annotation()` 这 7 个函数除了可以指定类名外，还可以指定参数名，将目标对象连接点上的方法入参绑定到增强的方法中。

其中，`args()` 用于绑定连接点方法的入参；`@annotation()` 用于绑定连接点方法的注解

对象；而@args()用于绑定连接点方法入参的注解。来看一个 args()绑定参数的实例，如代码清单 8-13 所示。

代码清单 8-13 TestAspect：绑定连接点参数

```
package com.smart.aspectj.advanced;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
@Aspect
public class TestAspect {
    @Before("target(com.smart.NaiveWaiter) && args(name,num,...)") ①
    public void bindJoinPointParams(int num,String name) ②
    {
        System.out.println("----bindJoinPointParams()----");
        System.out.println("name:"+name);
        System.out.println("num:"+num);
        System.out.println("----bindJoinPointParams()----");
    }
}
```

绑定连接点参数，首先，args(name, num,...) 根据 ②处的增强方法入参找到 name 和 num 对应的类型，以得到真实的切点表达式：target(com.smart.NaiveWaiter) && args(String,int,...)；其次，在该增强方法织入目标连接点时，增强方法可以通过 num 和 name 访问到连接点的方法入参

增强方法接受连接点的参数

在①处，通过 args(name,num,...)进行连接点参数的绑定。和前面讲述的方式不同，当args()函数入参为参数名时，共包括两方面的信息。

(1) 连接点匹配规则信息：连接点方法的第一个入参是 String 类型，第二个入参是 int 类型。

(2) 连接点方法入参和增强方法入参的绑定信息：连接点方法的第一个入参绑定到增强方法的 name 参数上，第二个入参绑定到增强方法的 num 参数上。

切点匹配和参数绑定的过程是这样的：首先，args()根据参数名称在增强方法中查到名称相同的入参并获知对应的类型，这样就知道了匹配连接点方法的入参类型；其次，连接点方法入参类型所在的位置则由参数名在 args()函数中声明的位置决定。代码清单 8-13 中 args(name,num) 匹配的目标类方法的第一个入参必须是 String 类型，第二个入参必须是 int 类型，如 smile(String name,int times)匹配，而 smile(int times ,String anme)不匹配。可以通过图 8-10 详细了解这一有趣的匹配过程。

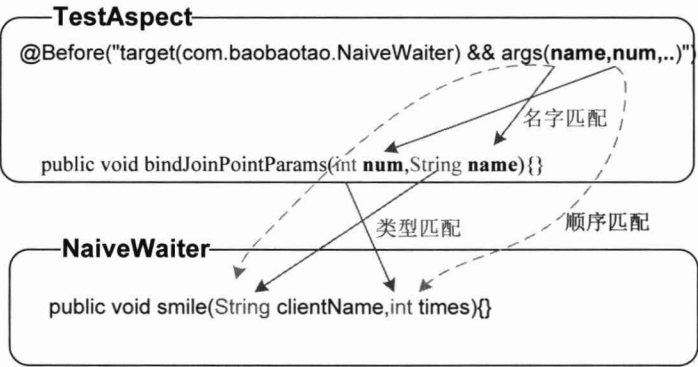


图 8-10 切点匹配和参数绑定过程

和 `args()` 一样, 其他可以绑定连接点参数的切点函数 (如 `@args()` 和 `target()` 等), 当指定参数名时, 就同时具有匹配切点和绑定参数双重功能。

运行下面的测试代码:

```
String configPath = "com/smart/aspectj/advanced/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
NaiveWaiter naiveWaiter = (NaiveWaiter) ctx.getBean("naiveWaiter");
naiveWaiter.smile("John", 2);
```

将看到以下输出信息:

```
----bindJoinPointParams()----
name:John
num:2
----bindJoinPointParams()----
NaiveWaiter:smile to John2times...
```

可见, 增强方法按预期绑定了 `NaiveWaiter.smile(String name,int times)` 方法的运行期入参。



提示

为了保证实例能成功执行, 必须启用 CGLib 动态代理: `<aop:aspectj- autoproxy proxy-target-class="true" />`。因为该实例需要对 `NaiveWaiter` 类进行代理 (因为 `NaiveWaiter#smile()` 方法不是 `Waiter` 接口的方法), 所以必须使用 CGLib 动态代理生成子类的代理方法。

8.6.6 绑定代理对象

使用 `this()` 或 `target()` 函数可绑定被代理对象实例, 在通过类实例名绑定对象时, 依然具有原来连接点匹配的功能, 只不过类名是通过增强方法中同名入参的类型间接决定罢了。这里通过 `this()` 函数来了解对象绑定的用法。

```
package com.smart.aspectj.advanced;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import com.smart.Waiter;
@Aspect
public class TestAspect {
    @Before("this(waiter)") ①
    public void bindProxyObj(Waiter waiter) { ②
        System.out.println("----bindProxyObj()----");
        System.out.println(waiter.getClass().getName());
        System.out.println("----bindProxyObj()----");
    }
}
```

通过②处查找出 `waiter` 对应的类型为 `Waiter`, 因而切点表达式为 `this(Waiter)`。当增强方法织入目标连接点时, 增强方法通过 `waiter` 入参绑定目标对象

①处的切点表达式首先按类变量名查找②处增强方法的入参列表, 进而获取类变量名对应的类为 `com.smart.Waiter`, 这样就知道了切点的定义为 `this(com.smart.Waiter)`, 即所有代理对象为 `Waiter` 类的所有方法匹配该切点。②处的增强方法通过 `waiter` 入参绑定目标对象。

可见 NaiveWaiter 的所有方法都匹配①处的切点。运行以下测试代码：

```
String configPath = "com/smart/aspectj/advanced/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter naiveWaiter = (Waiter) ctx.getBean("naiveWaiter");
naiveWaiter.greetTo("John");
```

可以看到如下输出信息：

```
----bindProxyObj()----
com.smart.NaiveWaiter$$EnhancerByCGLIB$$6758891b
----bindProxyObj()----
NaiveWaiter:greet to John...
```

按相似的方法使用 target()函数进行绑定。

8.6.7 绑定类注解对象

@within()和@target()函数可以将目标类的注解对象绑定到增强方法中，下面通过@within()函数演示注解绑定的操作。

```
package com.smart.aspectj.advanced;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import com.smart.Monitorable;
@Aspect
public class TestAspect {
    @Before("@within(m)") ①
    public void bindTypeAnnoObject(Monitorable m) { ②
        System.out.println("----bindTypeAnnoObject()----");
        System.out.println(m.getClass().getName());
        System.out.println("----bindTypeAnnoObject()----");
    }
}
```

通过②处查找出 m 对应 Monitorable 类型的注解，因而真实的切点表达式为 @within(Monitorable)。当增强方法织入目标连接点时，增强方法通过 m 入参可以引用到连接点

在 NaiveWaiter 类中标注了@Monitorable 注解，所有的 NaiveWaiter Bean 都匹配切点，其 Monitorable 注解对象将绑定到增强方法中。运行以下代码，即可查看绑定注解对象：

```
String configPath = "com/smart/aspectj/advanced/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter naiveWaiter = (Waiter) ctx.getBean("naiveWaiter");
((NaiveWaiter)naiveWaiter).greetTo("John");
```

输出以下信息：

```
----bindTypeAnnoObject()----
$Proxy3
----bindTypeAnnoObject()----
NaiveWaiter:greet to John...
```

从输出信息中我们发现了一个秘密，即使用 CGLib 代理 NaiveWaiter 时，其类的注解 Monitorable 对象也被代理了。

8.6.8 绑定返回值

在后置增强中，可以通过 `returning` 绑定连接点方法的返回值，如下：

```
@AfterReturning(value="target(com.smart.SmartSeller)",returning="retVal") //①
public void bindReturnValue(int retVal){ //②
    System.out.println("----bindException()----");
    System.out.println("returnValue:"+retVal);
    System.out.println("----bindException()----");
}
```

①处和②处的名字必须相同，此外，②处 `retVal` 的类型必须和连接点方法的返回值类型匹配。运行下面的测试代码：

```
String configPath = "com/smart/aspectj/advanced/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
SmartSeller seller = (SmartSeller) ctx.getBean("seller");
seller.sell("Beer","John");
```

可以看到以下输出信息：

```
SmartSeller: sell Beer to John...
----bingReturnValue()----
returnValue:100
----bingReturnValue()----
```

可见目标连接点 `Seller#sell()` 方法返回的入参被成功绑定到增强方法中。

8.6.9 绑定抛出的异常

和通过切点函数绑定连接点信息不同，连接点抛出的异常必须使用 `AfterThrowing` 注解的 `throwing` 成员进行绑定，如代码清单 8-14 所示。

代码清单 8-14 TestAspect: 绑定异常对象

```
package com.smart.aspectj.advanced;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
@Aspect
public class TestAspect {
    @AfterThrowing(value="target(com.smart.SmartSeller)",throwing="iae") //①
    public void bindException(IllegalArgumentException iae){ //②
        System.out.println("----bindException()----");
        System.out.println("exception:"+iae.getMessage());
        System.out.println("----bindException()----");
    }
}
```

①处 `throwing` 指定的异常名和②处入参的异常名相同，这个异常增强只在连接点抛出异常 `instanceof IllegalArgumentException` 时才匹配，增强方法通过 `iae` 参数可以访问抛出的异常对象。

在 `SmartSeller` 中添加一个抛出异常的测试方法，如下：

```
package com.smart;
public class SmartSeller implements Seller {
```



```

public void checkBill(int billId){
    if(billId == 1) throw new IllegalArgumentException("iae Exception");
    else throw new RuntimeException("re Exception");
}
}

```

当 billId 为 1 时抛出 `IllegalArgumentException`，否则抛出 `RuntimeException`。运行以下测试代码：

```

String configPath = "com/smart/aspectj/advanced/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
SmartSeller seller = (SmartSeller) ctx.getBean("seller");
seller.checkBill(1); ①

```

运行该方法将抛出 `IllegalArgumentException`

将看到以下输出信息：

```

----bindException()----
exception:iae Exception
----bindException()----
Exception in thread "main" java.lang.IllegalArgumentException: iae Exception
...

```

可见，当 `seller.checkBill(1)` 抛出异常后，异常增强起作用，处理完成后，再向外抛出 `IllegalArgumentException`。如果将①处的代码调整为 `seller.checkBill(2)`，再次运行代码，将直接抛出 `RuntimeException`，异常增强并没有起作用。这是因为 `RuntimeException` 不按类型匹配于 `IllegalArgumentException`，切点不匹配。

8.7 基于 Schema 配置切面

如果读者的项目不能使用 Java 5.0，那么就无法使用基于 `@AspectJ` 注解的切面。但是使用 `AspectJ` 切点表达式的大门依旧向我们敞开着，因为 Spring 提供了基于 Schema 配置的方法，它完全可以替代基于 `@AspectJ` 注解声明切面的方式。

这是很容易理解的，因为基于 `@AspectJ` 注解的切面，本质上是将切点、增强类型的信息使用注解进行描述，现在把这两个信息移到 Schema 的 XML 配置文件中，只是配置信息所在的地方不一样，表达的信息可以完全相同。XML 和注解，一个是“金刚”，一个是“罗汉”，就像那句俗语“猪往前拱，鸡往后刨”，做的是同一件事，只是方法形式不同而已。

使用基于 Schema 的切面定义后，切点、增强类型的注解信息从切面类中剥离出来，原来的切面类也就蜕变为真正意义上的 POJO。

8.7.1 一个简单切面的配置

首先来配置一个基于 Schema 的切面，它使用了 `aop` 命名空间。为了更准确地了解整个配置文件的全貌，给出一个结构上完整的切面示例，如代码清单 8-15 所示。

代码清单 8-15 基于Schema配置的切面示例

```

<?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: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/aop
                           http://www.springframework.org/schema/aop/spring-aop-4.0.xsd">

  <aop:config proxy-target-class="true">
    <aop:aspect ref="adviceMethods"> ① ← 引用④处的 adviceMethods
      <aop:before pointcut="target(com.smart.NaiveWaiter) and execution (* greetTo(...))" ②
                  method="preGreeting" /> ③ ← 声明切点表达式
      </aop:aspect>                  增强方法使用 adviceMethods Bean 中
    </aop:config>                  的 preGreeting 方法
                                    增强方法所在的 Bean
  <bean id="adviceMethods" class="com.smart.schema.AdviceMethods" /> ④
  <bean id="naiveWaiter" class="com.smart.NaiveWaiter" />
  <bean id="naughtyWaiter" class="com.smart.NaughtyWaiter" />
</beans>

```

使用一个<aop:aspect>元素标签定义切面，其内部可以定义多个增强。在<aop:config>元素中可以定义多个切面。在①处，切面引用了 adviceMethods Bean，该 Bean 是增强方法所在的类。通过<aop:before>声明了一个前置增强，并通过 pointcut 属性定义切点表达式，切点表达式的语法和@AspectJ 中所用的语法完全相同，由于&&在 XML 中使用不便，所以一般用 and 操作符代替之。③处通过 method 属性指定增强的方法，该方法应该是 adviceMethods Bean 中的方法。

<aop:config>拥有一个 proxy-target-class 属性，当设置为 true 时，表示其中声明的切面均使用 CGLib 动态代理技术；当设置为 false 时，使用 Java 动态代理技术。一个配置文件可以同时定义多个<aop:config>，不同的<aop:config>可以采取不同的代理技术。

AdviceMethods 是增强方法所在的类，它是一个普通的 Java 类，没有任何特殊的地方。

```

package com.smart.schema;
public class AdviceMethods{
  public void preGreeting() { ① ← 该方法通过配置被
                               用作增强的方法
    System.out.println("--how are you!--");
  }
}

```

通过以下代码测试这个使用 Schema 配置的切面：

```

String configPath = "com/smart/schema/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter naiveWaiter = (Waiter) ctx.getBean("naiveWaiter");
Waiter naughtyWaiter = (Waiter) ctx.getBean("naughtyWaiter");
naiveWaiter.greetTo("John");
naughtyWaiter.greetTo("Tom");

```

输出以下信息:

```
--how are you!-- ①
NaiveWaiter:greet to John...
NaughtyWaiter:greet to Tom...
```

织入的增强逻辑

可见, 切面被正确地实施到目标 Bean 中。

8.7.2 配置命名切点

在代码清单 8-15 的②处通过 `pointcut` 属性声明的切点是匿名切点, 不能被其他增强或其他切面引用。Spring 提供了命名切点的配置方式, 如代码清单 8-16 所示。

代码清单 8-16 命名切点配置

```
<aop:config proxy-target-class="true">
  <aop:aspect ref="adviceMethods" >

    <aop:pointcut id="greetToPointcut" ①
      expression="target (com.smart.NaiveWaiter)
        and execution(* greetTo(..))" />
    <aop:before method="preGreeting" pointcut-ref="greetToPointcut" />②
  </aop:aspect>
</aop:config>
```

定义切点, 该切点的命名
为 `greetToPointcut`

引用切点

在①处, 使用 `<aop:pointcut>` 定义了一个切点, 并通过 `id` 属性进行了命名; 在②处, 通过 `pointcut-ref` 引用这个命名的切点。和 `<aop:before>` 一样, 除引入增强外, 其他任何增强类型的元素都拥有 `pointcut`、`pointcut-ref` 及 `method` 这 3 个属性。

`<aop:pointcut>` 如果位于 `<aop:aspect>` 元素中, 则命名切点只能被当前 `<aop:aspect>` 内定义的元素访问到。为了能被整个 `<aop:config>` 元素中定义的所有增强访问, 必须在 `<aop:config>` 元素下定义切点。

```
<aop:config proxy-target-class="true">
  <aop:pointcut id="greetToPointcut" ①
    expression="target (com.smart.NaiveWaiter) and execution(* greetTo(..))" />

  <aop:aspect ref="adviceMethods" >
    <aop:before method="preGreeting" pointcut-ref="greetToPointcut" /> ②
  </aop:aspect>
  <aop:aspect ref="adviceMethods" >
    <aop:after method="postGreeting" pointcut-ref="greetToPointcut" /> ③
  </aop:aspect>
</aop:config>
```

当前 `<aop:config>` 下的所有
增强均可以访问该切点

在①处定义的命名切点分别被②和③处不同的切面增强所访问。如果在 `<aop:config>` 元素下直接定义 `<aop:pointcut>`, 则必须保证 `<aop:pointcut>` 在 `<aop:aspect>` 之前定义。在 `<aop:config>` 元素下还可以定义 `<aop:advisor>` (后面介绍), 三者在 `<aop:config>` 中的配置有先后顺序的要求: 首先是 `<aop:pointcut>`, 然后是 `<aop:advisor>`, 最后是 `<aop:aspect>`。而在 `<aop:aspect>` 中定义的 `<aop:pointcut>` 则没有先后顺序的要求, 可以在任何位置定义。可以通过 `spring-aop-4.0.xsd` Schema 样式定义文件了解关于 `config` 元素的配置规则。



实战经验

Schema 样式定义文件比 DTD 样式定义文件具有更强的文档样式定义能力。但由于 Schema 样式定义文件使用基于 XML 的文档样式描述语言定义文档格式, 所以样式定义文件本身的内容比较复杂, 可读性不高。为了快速理解 Spring 所提供的 Schema 样式定义文件(如 spring-aop-4.0.xsd 等)所描述的文档规则, 可以借助一些可视化工具。Altova XMLSpy 就是这样的一款 Schema 可视化工具, 图 8-11 就是从 XMLSpy 中导出的。

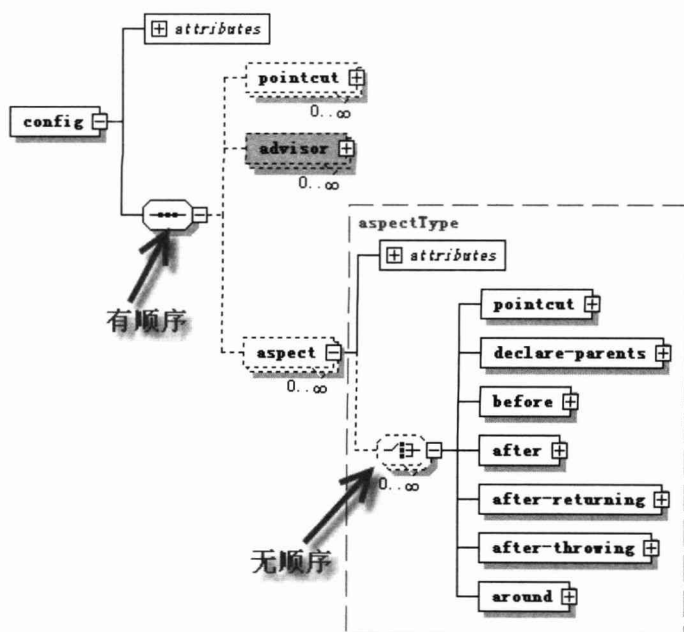


图 8-11 config 元素的 Schema 样式定义

8.7.3 各种增强类型的配置

基于 Schema 定义的切面和基于@AspectJ 定义的切面内容大抵一致, 只是在表现形式上存在差异罢了。在上一节中介绍了前置增强, 本节来学习如何通过 Schema 配置其余的增强类型。

1. 后置增强

通过<aop:after-returning>配置后置增强。

```
<aop:config proxy-target-class="true">
  <aop:aspect ref="adviceMethods">
    <aop:after-returning method="afterReturning"
      pointcut="target(com.smart.SmartSeller)" returning="retVal"/> ①
  </aop:aspect>
</aop:config>
```

```
</aop:config>
...
```

returning 属性必须和增强方法的入参名一致。下面是后置增强对应的方法：

```
package com.smart.schema;
public class AdviceMethods {
    public void afterReturning(int retVal) { ①
        ...
    }
}
```

增强方法，retVal 和配置文件中的 returning 属性值相同

如果增强方法不希望接收返回值，将配置处的<aop:after-returning>的 returning 属性和增强方法的对应入参去除即可。

2. 环绕增强

通过<aop:around>配置环绕增强。

```
<aop:config proxy-target-class="true">
<aop:aspect ref="adviceMethods" >
    <aop:around method="aroundMethod"
        pointcut="execution(* serveTo(..) and within(com. smart.Waiter)"/>
    </aop:aspect>
</aop:config>
...
```

和前面介绍的一样，环绕增强对应的方法，可以将第一个入参声明为 ProceedingJoinPoint。

```
package com.smart.schema;
import org.aspectj.lang.ProceedingJoinPoint;
public class AdviceMethods {
    public void aroundMethod(ProceedingJoinPoint pjp) { ①
        ...
    }
}
```

环绕增强的方法，pjp 可以访问到环绕增强的连接点信息

3. 抛出异常增强

通过<aop:after-throwing>匹配抛出异常的增强。

```
<aop:config proxy-target-class="true">
<aop:aspect ref="adviceMethods" >
    <aop:after-throwing method="afterThrowingMethod"
        pointcut="target(com.smart.SmartSeller) and execution(* checkBill(..))"
        throwing="iae"/> ①
    </aop:aspect>
</aop:config>
...
```

通过 iae 查找增强方法对应名字的入参，进而获取需要增强的连接点的匹配异常类型为 IllegalArgumentException

通过 throwing 属性声明需要绑定的异常对象，指定的异常名必须和增强方法对应的入参名一致。

```
package com.smart.schema;
public class AdviceMethods {
    public void afterThrowingMethod(IllegalArgumentException iae) {
        ...
    }
}
```

4. Final 增强

通过<aop:after>配置 Final 增强。

```
<aop:config proxy-target-class="true">
  <aop:aspect ref="adviceMethods" >
    <aop:after method="afterMethod"
      pointcut="execution(* com.*.Waiter.greetTo(..))"/>
  </aop:aspect>
</aop:config>
...
```

对应的 Final 增强方法如下：

```
package com.smart.schema;
public class AdviceMethods {
  public void afterMethod(){
    ...
  }
}
```

5. 引介增强

通过<aop:declare-parents>配置引介增强。引介增强和其他类型的增强不同，它没有 method、pointcut 和 pointcut-ref 属性。

```
<aop:config proxy-target-class="true">
  <aop:aspect ref="adviceMethods">
    <aop:declare-parents
      implement-interface="com.smart.Seller" ①
      default-impl="com.smart.SmartSeller" ②
      types-matching="com.smart.Waiter+" /> ③
    </aop:aspect>
  </aop:config>
...
```

① 要引介实现的接口
② 默认的实现类
③ 哪些类需要引介接口的实现

<aop:declare-parents>通过 implement-interface 属性声明要实现的接口，通过 default-impl 属性指定默认的实现类，通过 types-matching 属性以 AspectJ 切点表达式语法指定哪些 Bean 需要引介 Seller 接口的实现。可以通过以下代码查看到 NaiveWaiter 已经实施了引介增强：

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter naiveWaiter = (Waiter) ctx.getBean("naiveWaiter");
((Seller)naiveWaiter).sell("Beer", "John"); ①
```

强制类型转换成功

需要注意的是，虽然<aop:declare-parents>没有 method 属性指定增强方法所在的 Bean，但<aop:aspect ref="adviceMethods">中的 ref 属性依然要指定一个增强 Bean。

8.7.4 绑定连接点信息

基于 Schema 配置的增强方法绑定连接点信息和基于@AspectJ 绑定连接点信息所使用的方式没有什么区别。

- 第一，所有增强类型对应的方法的第一个入参都可以声明为 JoinPoint（环绕增强可以声明为 ProceedingJoinPoint）访问连接点信息。

□ 第二, <aop:after-returning> (后置增强) 可以通过 `returning` 属性绑定连接点方法的返回值, <aop:after-throwing> (抛出异常增强) 可以通过 `throwing` 属性绑定连接点方法所抛出的异常。

□ 第三, 所有增强类型都可以通过可绑定参数的切点函数绑定连接点方法的入参。

第一、第二种绑定参数的访问已经在上一节进行了介绍, 下面通过一个实例来了解第三种绑定参数的方法, 如代码清单 8-17 所示。

代码清单 8-17 绑定连接点参数到增强方法

```
<aop:config proxy-target-class="true">
  <aop:aspect ref="adviceMethods">
    <aop:before method="bindParams" ①
      pointcut="target(com.smart.NaiveWaiter) and args(name, num,..)" /> ②
  </aop:aspect>
</aop:config>
```

在②处, 在切点表达式中通过 `args(name,num,..)` 绑定了连接点的两个参数, 对应的增强函数如代码清单 8-18 所示。

代码清单 8-18 AdviceMethods 绑定参数的增强方法

```
package com.smart.schema;
import org.aspectj.lang.ProceedingJoinPoint;
public class AdviceMethods {
  public void bindParams(int num,String name){ //①
    System.out.println("----bindParams()----");
    System.out.println("name:"+name);
    System.out.println("num:"+num);
    System.out.println("----bindParams()----");
  }
}
```

①处的 `bindParams(int num,String name)` 和代码清单 8-17 中的切点函数 `args(name,num,..)` 声明的参数名必须相同。运行以下代码:

```
String configPath = "com/smart/schema/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter naiveWaiter = (Waiter) ctx.getBean("naiveWaiter");
((NaiveWaiter)naiveWaiter).smile("John", 2);
```

输出以下信息:

```
----bindParams()----
name:John
num:2
----bindParams()----
NaiveWaiter:smile to John2times...
```

可见, 目标类 `NaiveWaiter` 的连接点方法 `smile("John", 2)` 的入参被正确地绑定到增强方法中。

8.7.5 Advisor 配置

在第 7 章中学习了 `Advisor` 的知识, 它是 Spring 中切面概念的对应物, 是切点和增

强的复合体，不过它仅包含一个切点和一个增强。在 AspectJ 中没有对应的等价物，在 aop Schema 配置样式中，可以通过<aop:advisor>配置一个 Advisor。通过 advice-ref 属性引用基于接口定义的增强，通过 pointcut 定义切点表达式，或者通过 pointcut-ref 引用一个命名的切点。来看下面的例子：

```
<aop:config proxy-target-class="true">
  <aop:advisor advice-ref="testAdvice" ①  引用③处的增强
               pointcut="execution(* com..*.Waiter.greetTo(..))"/> ②  切点表达式
</aop:config>
<bean id="testAdvice"
      class="com.smart.schema.TestBeforeAdvice"/> ③  前置增强
```

在③处定义了一个前置增强 Bean，而在①处引用这个增强，在②处使用切点表达式配置切点。TestBeforeAdvice 是一个实现了 MethodBeforeAdvice 接口的增强类，如下：

```
package com.smart.schema;
import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;
public class TestBeforeAdvice implements MethodBeforeAdvice {
    public void before(Method method, Object[] args, Object target)
        throws Throwable {
        System.out.println("-----TestBeforeAdvice-----");
        System.out.println("clientName:"+args[0]);
        System.out.println("-----TestBeforeAdvice-----");
    }
}
```

8.8 混合切面类型

现在我们已经掌握了 4 种定义切面的方式：

- (1) 基于@AspectJ 注解的方式。
- (2) 基于<aop:aspect>的方式。
- (3) 基于<aop:advisor>的方式。
- (4) 基于 Advisor 类的方式。

作为开发者，可能会觉得 Spring 在同一个问题上提供了太多的选择，在选择实现方案时反倒让人陷入了困境。其实 Spring 并不是在做一件吃力不讨好的事情，开发人员完全可以根据项目的具体情况做出选择：如果项目采用 Java 5.0，则可以优先考虑使用@AspectJ；如果项目只能使用低版本的 JDK，则可以考虑使用<aop:aspect>；如果正在升级一个基于低版本 Spring AOP 开发的项目，则可以考虑使用<aop:advisor>复用已经存在的 Advice 类；如果项目只能使用低版本的 Spring，那么就只能使用 Advisor 了。此外，值得注意的是，一些切面只能使用基于 API 的 Advisor 方式进行构建，如基于 ControlFlowPointcut 的流程切面。

8.8.1 混合使用各种切面类型

Spring 虽然提供了 4 种定义切面的方式，但其底层的实现技术却是一样的，那就是基于 CGLib 和 JDK 动态代理，所以在同一个 Spring 项目中可以混合使用 Spring 所提供的各种切面定义方式，如代码清单 8-19 所示。

代码清单 8-19 各种切面类型混合使用

```

<bean id="controlFlowAdvisor" ① ← 使用 Advisor API 方式实现的流程控制切面
    class="org.springframework.aop.support.DefaultPointcutAdvisor">
    <property name="pointcut">
        <bean class="org.springframework.aop.support.ControlFlowPointcut">
            <constructor-arg type="java.lang.Class"
                value="com.smart.advisor.WaiterDelegate"/>
            <constructor-arg type="java.lang.String" value="service"/>
        </bean>
    </property>
    <property name="advice">
        <bean class="com.smart.advisor.GreetingBeforeAdvice"/>
    </property>
</bean>

<aop:aspectj-autoproxy />② ← 使用@AspectJ 方式定义的切面

<bean class="com.smart.aspectj.example.PreGreetingAspect" />
    使用基于 Schema 配置方式定义的切面

<aop:config proxy-target-class="true">③ ←
    <aop:advisor advice-ref="testAdvice" pointcut="execution(* com...*.Waiter.
        greetTo(..))"/>④ ← 使用<aop:advisor>配置方式定义的切面

    <aop:aspect ref="adviceMethods">⑤ ← 使用<aop:aspect>配置方式定义的切面
        <aop:before pointcut="target (com.smart.NaiveWaiter) and execution (* greetTo(..))"
            method="preGreeting"/>
    </aop:aspect>
    POJO 的增强类

</aop:config>
<bean id="adviceMethods" class="com.smart.schema.AdviceMethods" />⑥
<bean id="testAdvice" class="com.smart.schema.TestBeforeAdvice"/>⑦ ← 基于特定增强接口的增强类

```

虽然 Spring 可以混合使用各种切面类型达到统一的效果，但在一般情况下并不会在一个项目中同时使用。毕竟项目开发不是时装表演，也不是多军种联合演习，我们应该尽量根据项目的实际需要采用单一的实现方式，以保证技术的单一性。

8.8.2 各种切面类型总结

在学习完 Spring 的 4 种切面定义方式后，有必要对这 4 种实现方式进行比较，它们的本质是相同的，都是定义切点和增强，不同的只是在表现形式上，如表 8-2 所示。

表 8-2 切面不同定义方式具体实现比较

		@AspectJ	<aop:aspect>	Advisor	<aop:advisor>
增强类型	前置增强	@Before	<aop:before>	MethodBeforeAdvice	同 Advisor
	后置增强	@AfterReturning	<aop:after-returning>	AfterReturningAdvice	同 Advisor
	环绕增强	@Around	<aop:around>	MethodInterceptor	同 Advisor
	抛出异常增强	@AfterThrowing	<aop:after-throwing>	ThrowsAdvice	同 Advisor
	final 增强	@After	<aop:after>	无对应接口	同 Advisor
	引介增强	@DeclareParents	<aop:declare-parents>	IntroductionInterceptor	同 Advisor
切点定义		支持 AspectJ 切点表达式语法, 可以通过@Pointcut 注解定义命名切点	支持 AspectJ 切点表达式语法, 可以通过<aop:pointcut>定义命名切点	直接通过基于Pointcut 的实现类定义切点	基本上和<aop:aspect>相同, 不过切点函数不能绑定参数
连接点方法入参绑定		(1) 使用 JoinPoint、ProceedingJoinPoint 连接点对象; (2) 使用切点函数指定参数名绑定	同@AspectJ <aop:after-returning>	通过增强接口方法入参绑定	同 Advisor
连接点方法返回值或抛出异常绑定		(1) 在后置增强中, 使用@AfterReturning 的 returning 成员绑定方法返回值; (2) 在抛出异常增强中, 使用@AfterThrowing 的 throwing 成员绑定方法抛出的异常	(1) 在后置增强中, 使用<aop:after-returning>的 returning 属性绑定方法返回值; (2) 在抛出异常增强中, 使用<aop:after-throwing>的 throwing 属性绑定方法抛出的异常	通过增强接口方法入参绑定	同 Advisor

从表 8-2 中可以看出, <aop:advisor>其实是<aop:aspect>和 Advisor 的“混血儿”, 它的切点表示方式和<aop:aspect>相同, 增强定义方式则和 Advisor 相同。连接点方法入参的绑定方式和 Advisor 一样, 通过增强接口方法入参进行调用, 所以<aop:advisor>在切点表达式中不能使用切点函数绑定连接点方法入参, 否则会产生错误。

在内部, Spring 使用 AspectJExpressionPointcut 为@AspectJ、<aop:aspect>及<aop:advisor>提供具体的切点实现。

8.9 其他

8.9.1 JVM Class 文件字节码转换基础知识

到目前为止, 我们所接触到的 AOP 切面织入都是在运行期通过 JDK 或 CGLib 动态代理的方式实现的。我们知道, 除了运行期织入切面的方式外, 还可以在类加载期通过字节码编辑技术将切面织入目标类中, 这种织入方式称为 LTW (Load Time Weaving)。

AspectJ LTW 使用 Java 5.0 所提供的代理功能 (agent) 完成加载期切面织入工作。JDK 的代理功能能够让代理器访问到 JVM 的底层部件, 借此向 JVM 注册类文件转换器, 在类加载时对类文件的字节码进行转换。AspectJ LTW 由于基于 JDK 动态代理技术工作, 而 JDK 动态代理的作用范围是整个 JVM, 所以这种工作方式比较粗放, 对于单一 JVM 多个应用的情况尤其不适合。

Spring 为 LTW 的过程提供了细粒度的控制, 它支持在单个 ClassLoader 范围内实施类文件转换, 且配置更为简单。

在进行 Spring LTW 的学习之前, 先来了解一下 Java 5.0 中引入的 Instrument 的相关知识, 为后面的学习打好基础。

java.lang.instrument 包的工作原理

Java 5.0 新增了一个 java.lang.instrument 包, 该包中有两个能对 JVM 底层组件进行访问的类。具体地说, 就是通过 JVM 的 -javaagent 代理参数在启动时获取 JVM 内部组件的引用, 以便在后续流程中使用。借助 JDK 动态代理, 可以在 JVM 启动时装配并应用 ClassTransformer, 对类字节码进行转换, 实现 AOP 的功能。

java.lang.instrument 包中定义了两个重要的接口。

□ **ClassFileTransformer**: Class 文件转换器接口, 该接口有一个唯一的方法, 如下:

```
byte[] transform(ClassLoader loader, String className, Class<?> classBeingRedefined,
ProtectionDomain protectionDomain, byte[] classfileBuffer)
```

该接口对 Class 文件的字节码进行转换, classfileBuffer 是类文件对应的字节码数组, 返回的 byte[] 为转换后的字节码。如果返回 null, 则表示不进行字节码处理 (并非将类的字节码数据置空)。

□ **Instrumentation**: 代表 JVM 内部的一个构件。这个术语不好翻译, 它有“仪表、仪器、设备”等意思, 我们不妨将其称为“组件”。

可以通过该接口的方法向 JVM 的内部“组件”注册一些 ClassFileTransformer, 注册转换器的接口方法为 void addTransformer(ClassFileTransformer transformer)。

当 ClassFileTransformer 实例注册到 JVM 中后, JVM 在加载 Class 文件时, 会先调用这个 ClassFileTransformer 的 transform() 方法对 Class 文件的字节码进行转换。如果向 JVM 中注册多个 ClassFileTransformer, 它们将按注册的顺序组成链式的调用。这样 ClassFileTransformer 的实现者就可以从 JVM 层面截获所有类的字节码, 并引入希望添加的逻辑, 如让每个类拥有性能监视的能力、织入特殊用途的增强代码等。

图 8-12 描述了拥有多个转换器的 JVM 从加载类到最终生成对应的类字节码的过程。

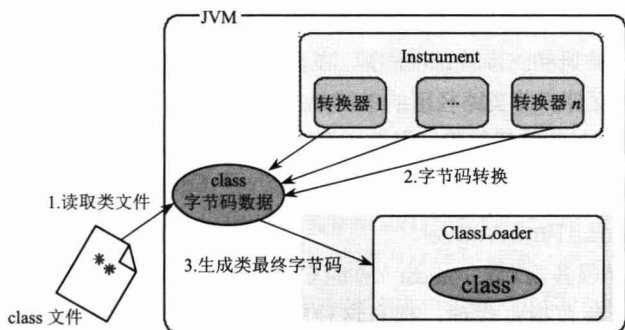


图 8-12 带转换器的 JVM 加载 Class 文件的处理过程

**提示**

目前已经有非常多字节码处理的开源项目，ObjectWeb 的 ASM、Apache 的 BCEL 及 SourceForge 和 SERP 是其中优秀的代表者。TopLink JPA 就使用 ASM 编写它的 ClassTransformer。

8.9.2 使用 LTW 织入切面

Spring 的 LTW 仅支持 AspectJ 定义的切面，既可以是直接采用 AspectJ 语法定义的切面，也可以是采用基于 @AspectJ 注解，通过 Java 类定义的切面。Spring LTW 直接采用了与 AspectJ LTW 相同的基础结构，即它利用类路径下的 META-INF/aop.xml 配置文件找到切面定义及切面所要实施的候选目标类的信息，通过 LoadTimeWeaver 在 ClassLoader 加载类文件时将切面织入目标类中。工作原理如图 8-13 所示。

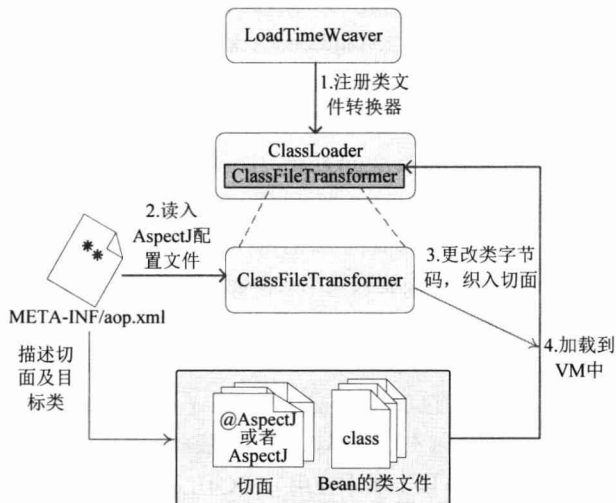


图 8-13 Spring LTW 工作原理

Spring 利用特定 Web 容器的 ClassLoader，通过 LoadTimeWeaver 将 Spring 提供的 ClassFileTransformer 注册到 ClassLoader 中。在类加载期，注册的 ClassFileTransformer 读取 AspectJ 的配置文件，即类路径下的 META-INF/aop.xml 文件，获取切面，对加载到 VM 中的 Bean 类进行字节码转换，织入切面。Spring 容器初始化 Bean 实例时，采用的 Bean 类就是已经被织入了切面的类。下面来了解一下 Spring 的 LoadTimeWeaver。

1. Spring 的 LoadTimeWeaver

大多数 Web 应用服务器（除 Tomcat 外）的 ClassLoader 都支持直接访问 Instrument，无须通过 javaagent 参数指定代理，拥有这种能力的 ClassLoader 称为“组件使能”（instrumentation-capable）。通过“组件使能”功能，可以非常方便地访问 ClassLoader 的 Instrument。Spring 利用 Web 应用服务器类加载器的这个特性，为它们分别提供了专门的 LoadTimeWeaver，以便向特定的 ClassLoader 注册 ClassFileTransformer，对类进行字节码转换，实施切面的织入。

Spring 的 org.springframework.instrument.classloading.LoadTimeWeaver 接口规定了类加载期织入器的高层协议。该接口有 3 个方法。

- ❑ void addTransformer(ClassFileTransformer transformer): 添加一个 ClassFileTransformer 到加载期织入器中。
- ❑ ClassLoader getInstrumentableClassLoader(): 我们知道，JVM 拥有 Instrumentation 组件，但这是 JVM 级别的。Spring 对 ClassLoader 进行扩展，让它也具有 Instrumentation 组件，以便只对 ClassLoader 中的类应用 ClassFileTransformer。
- ❑ ClassLoader getThrowawayClassLoader(): 返回一个“丢弃”的 ClassLoader，目的是使 Instrumentation 的作用范围仅局限在本 ClassLoader 中，而不影响父 ClassLoader。Spring 为 LoadTimeWeaver 提供了多个实现类。
- ❑ InstrumentationLoadTimeWeaver: 该装载期织入器使用 -javaagent JVM 启动参数注册转换器，该类必须和 org.springframework.instrument.InstrumentationSavingAgent 结合使用。InstrumentationSavingAgent 是代理类，它获取 JVM 的 Instrumentation 后，以静态变量的方式保存这个引用，这样其他类就可以通过 getInstrumentation() 方法从 InstrumentationSavingAgent 中获取 JVM 的 Instrumentation。InstrumentationLoadTimeWeaver 即利用 InstrumentationSavingAgent 持有的 Instrumentation 引用完成添加转换器的操作。
- ❑ SimpleLoadTimeWeaver: 该装载期织入器能为当前的 ClassLoader 创建一个相应的 SimpleInstrumentableClassLoader（简单组件使能的 ClassLoader 实现类），一般在测试或 IDE 环境下使用。
- ❑ 特定 Web 服务器的织入器: JBossLoadTimeWeaver（在 JBoss AS 5.0 以上版本使用）、GlassFishLoadTimeWeaver（在 GlassFish 3.0 以下版本使用）、WebLogicLoadTimeWeaver（在 BEA WebLogic 10.0 以上版本使用）、OC4JLoadTimeWeaver（在 Oracle OC4J 10.1.3.1 以上版本使用）。

❑ **ReflectiveLoadTimeWeaver**: 有一些应用服务器不是组件使能的 **ClassLoader**, 或者虽然是组件使能的, 但却无法获取应用服务器的类加载器实例, 用户只能以 **ClassLoader** 类型获取这些类加载器的句柄。在这种情况下, Spring 也没有办法将转换器添加到 **ClassLoader** 中。这时, 只得对目标的 **ClassLoader** 进行改造, 让它实现一些特殊的方法, 并由 **ReflectiveLoadTimeWeaver** 在运行期通过反射的机制调用接口方法, 以便注册 **ClassFileTransformer**。可以通过扩展需要改造的 **ClassLoader** 类, 并实现以下两个特殊方法达到目的。

- **public void addTransformer(java.lang.instrument.ClassFileTransformer)**: 向 **ClassLoader** 中注册转换器。
- **public ClassLoader getThrowawayClassLoader()**: 返回需要丢弃的 **ClassLoader**。

Spring 只需在配置文件中添加一行配置就可以启用 **LoadTimeWeaver**, 如下:

```
<?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">
  <!--①启用Spring装载期织入功能 -->
  <context:load-time-weaver/>
</beans>
```

通过①处的配置, 向 Spring 容器中添加一个 **LoadTimeWeaver**, 它负责向运行期的 **ClassLoader** 注册多个 **ClassFileTransformer**, 以便实施 **LTW** 的功能。

2. 在 Tomcat 下的配置

Tomcat 服务器的 **ClassLoader** 不是组件使能的 **ClassLoader**, Spring 专门为 Tomcat 提供了一个 **TomcatInstrumentableClassLoader**, 它扩展于 Tomcat 服务器的 **org.apache.catalina.loader.WebappClassLoader**, 并实现了 **LoadTimeWeaver** 要求的两个特殊方法。

可以按照以下步骤使 Tomcat Web 应用启动 Spring 的 **LoadTimeWeaver**。

(1) 在 Spring 配置文件中定义 **<context:load-time-weaver/>**。

(2) 将 **org.springframework.instrument.tomcat-{version}.jar** 复制到 **<TOMCAT_HOME>/lib** 下, 该 JAR 包中包含了 **TomcatInstrumentableClassLoader** 类。

(3) 编写以下配置片段, 让 Tomcat 服务器使用 **TomcatInstrumentableClassLoader** 作为其 **ClassLoader**。

```
<Context path="/myWebApp" docBase="/my/webApp/location">
<Loader loaderClass="org.springframework.instrument.classloading.tomcat.
  TomcatInstrumentableClassLoader"
  useSystemClassLoaderAsParent="false"/>
</Context>
```

如何让以上配置片段生效呢? 对于 Tomcat 6.x 和 Tomcat 7.x 来说, 可以使用以下 4 种方式。

(1) 打开<TOMCAT_HOME>/conf/server.xml 文件, 将以上配置片段添加到 server.xml 文件的相应位置上。

(2) 将以上配置片段的内容保存为 context.xml 文件, 并将其复制到<TOMCAT_HOME>/conf/context.xml 中。

(3) 将以上配置片段的内容保存为<webapp_name>.xml 文件, 并将其复制到<TOMCAT_HOME>/conf/Catalina/localhost/<webapp_name>.xml 中。

(4) 将以上配置片段的内容保存为 context.xml 文件, 并将其和 Web 应用程序一起打包在 WAR 包中, 放在 WAR 包的 META-INF 目录下: <webapp_name>.war/META-INF/context.xml。

前两种配置方式会产生全局, 也就是说, Tomcat 应用中所有 Web 应用的类加载器都会调整为 TomcatInstrumentableClassLoader; 而后两种配置方式只会对相应的 Web 应用产生影响, 其他的 Web 应用依旧使用 Tomcat 默认的 WebappClassLoader。从部署的简易性到范围的影响性, 最后一种配置方式是最佳选择。

3. 在其他 Web 应用服务器下的配置

前面说过, 有 4 种类型的 Web 服务器的类加载器支持组件使能功能。因此, 它们既无须使用 javaagent 参数, 也无须更改任何 Web 服务器的配置文件, 只需在 Spring 的配置文件中配置<context:load-time-weaver/>, 就可以使用 Spring 的 LTW。现将这 4 种 Web 服务器及其版本列出。

- ☐ BEA WebLogic (10.0 以上版本)。
- ☐ Oracle Containers for Java EE (OC4J 10.1.3.1 以上版本)。
- ☐ Resin (3.1 以上版本)。
- ☐ JBoss (5.x 以上版本)。

8.10 小结

在本章中, 首先介绍了 Java 5.0 的注解知识, 它是学习@AspectJ 的基础。使用@AspectJ 定义切面比基于接口定义切面更加直观、更加简洁, 成为 Spring 所推荐的切面定义方式。

掌握切点表达式语法和切点函数是学习@AspectJ 的重心, 我们分别对 9 个切点函数进行了详细的讲述。切点表达式非常灵活, 拥有强大的切点表达能力, 读者可以使用通配符、切点函数及切点运算符定义切点。

如果项目因某种原因无法使用 Java 5.0, 则可以采用基于 Schema 配置的方式继续使用 AspectJ 的切点表达式和增强定义, 基于 Schema 的配置采用<aop:aspect>描述@Aspect 类所描述的相同信息, 只是换了一种方法而已。此外, 还可以通过<aop:advisor>复用旧系统已有的 Advice, 并配合使用 AspectJ 的切点表达式。

在切点表达式中，大多数切点函数都可以绑定连接点方法的入参，以便增强方法访问连接点信息。此外，用户也可以简单地将增强方法的第一个入参定义为 JoinPoint 访问连接点的上下文。

Spring AOP 虽然提供了 4 种切面定义方式，但其底层实现却是相同的，也就是说，表象不同，本质归一。所以如果需要，则可以放心地混合使用这些不同的定义方式。

Spring 还支持 LTW 的功能，允许通过 AspectJ 定义切面，在类加载期通过类文件转换器织入切面。

第 9 章

Spring SpEL

Spring 动态表达式语言（简称 SpEL）是一个支持运行时查询和操作对象图的强大动态语言。其语法类似于 EL 表达式，具有诸如显式方法调用和基本字符串模板函数等特性。本章对 JVM 动态语言进行简要概述，并对 SpEL 动态语言特性及其表达式的用法进行详细介绍，通过实例逐步揭开 SpEL 表达式的层层外衣。通过本章的学习，读者不仅可以运用 SpEL 表达式解决实际项目中的一些通用性表达式需求，还可以应用 SpEL 提供的扩展点来解决一些高阶问题。

本章主要内容：

- ◆ JVM 动态语言
- ◆ SpEL 表达式概述
- ◆ SpEL 核心接口
- ◆ SpEL 基础表达式
- ◆ 在 Spring 中使用 SpEL

本章亮点：

- ◆ 对 Java 动态表达式发展的总结
- ◆ 对 SpEL 各种用法进行讲解

9.1 JVM 动态语言

Java 是一门强类型的静态语言，所有代码在运行之前都必须进行严格的类型检查并编译成 JVM 字节码；因此虽然在安全、性能方面得到了保障，但牺牲了灵活性。这个特征就决定了 Java 在语言层面无法直接进行表达式语句的动态解析。而动态语言恰恰相反，其显著的特点是在程序运行时可以改变程序结构或变量类型。典型的动态语言有

Ruby、Python、JavaScript、Perl 等。这些动态语言能够被广泛应用于许多领域，得益于其动态、简单、灵活等特性。因为它们无须编译，即可被解释执行。它们可以在运行时动态改变表达式语句，非常适合编写复杂的动态表达式。

Java 在实现复杂业务系统、大型商业系统、分布式系统及中间件等方面有着非常强的优势。但在开发这些系统的过程中，有时需要引用动态语言的一些特性，以弥补其在动态性方面的不足。特别是在解决一些商业系统中动态规则的解析需求，如积分规则、各类套餐计费、活动促销等规则时，在 Java 的早期版本中，开发人员一般会使用 Rhino、BeanShell、MVEL 等类库实现。

为了简化在 Java 中使用动态脚本语言的难度，Java 6.0 开始提供对 JSR-223 规范的全面支持。JSR-223 中规范了在 Java 虚拟机上运行的动态脚本语言与 Java 程序之间的交互方式，并在 Java 6.0 中内置集成了 Mozilla Rhino 的 JavaScript 解析引擎，因此可以很方便地在 Java 中调用 JavaScript 编写的动态脚本，并通过 JavaScript 解释执行的特性来编写相关动态业务逻辑。

下例在 Java 中使用 Rhino 快速实现一个动态求和函数 sum，只需在脚本引擎中注册一个标准的 JavaScript 函数，就可以在 Java 应用上下文中调用注册的 JavaScript 函数，如代码清单 9-1 所示。

代码清单 9-1 ScriptSample：脚本实现动态函数

```
package com.smart.js;
import javax.script.Invocable;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;

public class ScriptSample {

    public static void main(String[] args) throws Exception{
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("JavaScript");
        String scriptText = "function sum(a,b) { return a+b; } ";
        engine.eval(scriptText);
        Invocable invocable = (Invocable) engine;
        Object result = invocable.invokeFunction("sum", 100,99);
        System.out.println(result); //打印199
    }
}
```

上例只用几行代码就实现了一个动态求和函数。首先使用 ScriptEngineManager 创建一个脚本引擎管理器，并通过其创建脚本引擎，调用 ScriptEngine#eval() 方法注册 JavaScript 求和函数脚本。接下来就可以使用 Invocable#invokeFunction() 方法调用注册的 sum 函数，第一个参数就是要调用的自定义函数名 sum，第二个参数开始对应自定义函数名 sum 的参数列表。

虽然目前在 JVM 中支持很多脚本语言（如 JavaScript、Jruby、Jython 等），但在使用的时候，还是需要对其进行相应的封装。对于仅仅需要一些简单的表达式需求的场景，使用脚本语言显得有些“笨重”，这也就是下文要重点介绍 SpEL 表达式的原因。

9.2 SpEL 表达式概述

Spring 动态语言（简称 SpEL）是一个支持运行时查询和操作对象图的强大的动态语言。其语法类似于 EL 表达式，具有诸如显式方法调用和基本字符串模板函数等特性。

同其他的 Java 动态语言相比（如国外的 OGNL、MVEL 和 JBoss EL，国内的 Aviator、IKExpression 和 FastEL 等），SpEL 不但提供上述表达式的类似功能，而且更加简洁、灵活。加之出自 Spring 社区之手，与 Spring 框架及其子项目的结合显得更加顺畅自然。

SpEL 表达式的语言特性都是经过 Spring 框架及其子项目的需求一步步提炼而成的。在使用方面，IDE 工具可提供很好的代码自动补全功能。在集成方面，SpEL 抽象了一个通用表达式操作 API，因此可以很好地与其他动态语言进行集成。

SpEL 作为 Spring 家族中表达式求值的基础，它不直接依赖于 Spring 框架，可独立使用。只是在基于 Spring 的框架中使用 SpEL 更加便捷，一般情况下，无须手工调用 SpEL 提供的 API。因为 Spring 框架已经提供了许多直接使用 SpEL 表达式的方法，并且在框架层面屏蔽了表达式的运行设施创建过程，直接使用即可。例如，在 Bean 配置定义中，可以直接通过“#{ }”编写 SpEL 表达式。

为了更好地学习 SpEL 动态语言，本章中的示例将 SpEL 作为一个独立的动态语言来使用，通过手工调用 SpEL API 来讲解其各种表达式的用法。下面先通过一个简单的示例来认识一下 SpEL。在使用 SpEL 之前，需要在 pom.xml 文件中添加 spring-expression 模块依赖，如代码清单 9-2 所示。

代码清单 9-2 在 pom.xml 文件中添加表达式模块依赖

```
<?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">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.smart</groupId>
  <artifactId>chapter9</artifactId>
  <version>1.0</version>
  <name>Spring4.x第九章实例</name>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-expression</artifactId>
      <version>${spring.version}</version>
    </dependency>
  </dependencies>
</project>
```

SpEL 被设计成一个可独立运行的模块，可以脱离 Spring 容器直接运行，因此只需引入 SpEL 的模块 spring-expression 即可，无须引入 Spring 框架的其他模块。接下来，在代码中就可以使用 SpEL 提供表达式解析类，如代码清单 9-3 所示。

代码清单 9-3 SpelHello示例

```
package com.smart.spel;
import org.springframework.expression.Expression;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
public class SpelHello {
    public static void main(String[] args) {
        ExpressionParser parser =new SpelExpressionParser();
        Expression exp = parser.parseExpression("'Hello'+ ' World'");
        String message = (String) exp.getValue();
        System.out.println(message);
    }
}
```

在 SpEL 中,使用表达式非常简单,只需创建一个 SpelExpressionParser 实例即可。也就是首先创建一个 SpEL 表达式解析器,然后调用 ExpressionParser#parseExpression() 方法对表达式进行解析。其中,单引号表示 String 类型,如示例中的'Hello'、'World'。



轻松一刻

山东话与河南话的发音大致相同,但也有小小的差别。比如山东人称馒头为干粮,河南人称馒头为馍,发音与山东话的“么”相似。一个河南人在山东上大学,去食堂打饭,向大师傅说:“来个馍。”大师傅听了很是茫然,问:“你要么?”河南人答道:“对呀,我要馍。”大师傅听后就更糊涂了,这怎么来打饭的还不知道自己要什么的,就问道:“你到底要么啊?”河南人有点生气了,心想:你都知道我要什么了怎么还问啊。于是不耐烦地说:“你到底有没有馍?我就是个馍啊!!”

9.3 SpEL 核心接口

上一节介绍了 SpEL 的简单用法,接下来逐步讲解 SpEL 的核心接口。SpEL 的所有类与接口都定义在 org.springframework.expression 包及其子包,以及 spel.support 中。

ExpressionParser 接口用来解析表达式字符串,表达式字符串是一个用单引号标注或者用转义的双引号标注的字符串。Expression 接口用来计算表达式字符串的值,当调用 ExpressionParser#parseExpression() 和 Expression#getValue() 方法时可能抛出解析异常 ParseException 和求值异常 EvaluationException。因此,在手工调用 API 时,需要处理这些可能的异常。

SpEL 支持一系列功能特性,如方法调用、属性调用及构造器调用等。下例是调用字符串类型的 concat() 方法:

```
ExpressionParser parser= new SpelExpressionParser();
```

```
Expression exp=parser.parseExpression("'HelloWorld'.concat('!')");
String message=(String)exp.getValue(); // message的值为 "Hello World!"
```

在上述代码中，`Expression#getValue()`方法返回的是一个 `Object` 对象，需要进行显式类型转换，使用起来感觉有些不够优雅。可以使用泛型方法 `public <T> T getValue(Class<T> desiredResultType)`，这个方法取值的时候无须进行显式类型转换。但需要注意的是，如果返回的类型不能被转换为泛型 `T` 或者被已注册的类型转换器所转换时，则会抛出求值 `EvaluationException`。

```
String message= exp.getValue(String.class);
```

在 SpEL 中更常见的用途是针对特定实例对象的属性进行求值。在下面的例子中，我们通过表达式获取 `User` 实例的 `userName` 属性。

```
User user = new User();
user.setUserName("tom");
user.setCredits(100);
ExpressionParser parser = new SpelExpressionParser();
EvaluationContext context = new StandardEvaluationContext(user);
String username = (String)parser.parseExpression("userName").getValue(context);
```

在创建 `StandardEvaluationContext` 实例时，指定一个根对象作为求值目标对象，这样在求值表达式中就可以引用根对象属性。在求值内部可以使用反射机制从注册对象中获取相应的属性值。

在单独使用 SpEL 时，需要创建一个 `ExpressionParser` 解析器，并提供一个 `EvaluationContext` 求值上下文。在普通的基于 Spring 的应用开发中，这些 API 一般很少会涉及，仅需编写 SpEL 表达式字符串即可。例如，在 Bean 定义的时候使用 SpEL 表达式，只需写好相应的表达式字符串即可，至于解析器、求值上下文、根对象和其他预定义变量等基础设施，Spring 都会创建。

9.3.1 EvaluationContext 接口

`EvaluationContext` 接口提供了属性、方法、字段解析器及类型转换器。默认实现类 `StandardEvaluationContext` 的内部使用反射机制来操作对象。为了提高性能，在其内部会对已获取的 `Method`、`Field` 和 `Constructor` 实例进行缓存。

`StandardEvaluationContext` 类可以通过构造函数传递求值根对象或通过 `setRootObject()` 方法设置求值根对象，通过 `setVariable()` 方法设置相关变量，通过 `registerFunction()` 方法注册自定义函数。还可以向 `StandardEvaluationContext` 类注册自定义构造函数转换器 `ConstructorResolvers`、方法转换器 `MethodResolvers` 及属性转换器 `PropertyAccessors` 来解析 SpEL 表达式。

默认 SpEL 表达式使用的类型转换器是引用 Spring 核心包中的 `ConversionService`，`ConversionService` 转换服务会自动根据源类型与目标类型选择合适的转换器。Spring 内置了一系列常用的类型转换器，如 `StringToBooleanConverter`、`NumberToCharacterConverter` 等。如果内置的类型转换器未能满足需求，则可以编写并注册自定义类型转换器。当处

理表达式中涉及泛型时，SpEL 会尝试将当前值转换为对应的目标类型。例如，在表达式中通过 `setValue()` 方法来设置 `List` 类型的元素时，如果目标类型是 `List<Boolean>`，那么 SpEL 会自动尝试将当前设置值转换为布尔类型，如下：

```
class Simple{
    public List<Boolean> booleanList= new ArrayList<Boolean>();
}
Simple simple= new Simple();
simple.booleanList.add(true);

//创建求值上下文
StandardEvaluationContext simpleContext= new StandardEvaluationContext(simple);

//自动将“false”转换为布尔类型
parser.parseExpression("booleanList[0]").setValue(simpleContext, "false");

//b将被设置为false
Boolean b=simple.booleanList.get(0);
```

在上面的示例中，表达式 `“booleanList[0]”` 是一个 `List<Boolean>` 类型。将一个字符串类型的 `“false”` 值设置给 `booleanList` 集合的第一个元素，由于目标类型是 `Boolean`，因而在 SpEL 内部会自动将原类型为 `“false”` 的字符串值转换为目标布尔类型。如果将 `“false”` 改为 `“false1”`，再次运行的时候，则会报一个转换异常 `ConversionFailedException`，因为 `“false1”` 字符串无法转换为布尔类型。

9.3.2 SpEL 编译器

在默认情况下，SpEL 表达式只有在求值时才进行表达式计算，所以表达式可以在运行时进行动态修改。但对于同一个表达式而言，如果每次取值都要进行动态解析，则势必影响表达式的执行效率。对于一些调用频率不高的场景，对性能的影响不是很明显；反之，如果同一个表达式重复调用比较频繁，那么可能会对性能产生较大的影响。

`SpelCompiler` 编译器就是为了解决这个问题而诞生的，它可以将表达式直接编译成字节码，从而避免每次调用时进行语法解析所产生的时间消耗，有效提高执行效率。由于已经将表达式编译成字节码，如果在后续运行时表达式发生变化，则必须重新编译。因此，`SpelCompiler` 适用于表达式不经常发生变动且重复调用频率较高的场景。下面是基于 SpEL 编码的示例，如代码清单 9-4 所示。

代码清单 9-4 `CompilerSample` 编译表达式示例

```
package com.smart.spel;
import com.smart.User;
import org.springframework.expression.EvaluationContext;
import org.springframework.expression.spel.SpelCompilerMode;
import org.springframework.expression.spel.SpelParserConfiguration;
import org.springframework.expression.spel.standard.SpelExpression;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;
```



```

public class CompilerSample {

    public static void main(String[] args) {
        User user = new User();
        //①创建解析配置
        SpelParserConfiguration configuration = new SpelParserConfiguration(
            SpelCompilerMode.IMMEDIATE,
            CompilerSample.class.getClassLoader());

        //②创建解析器
        SpelExpressionParser parser = new SpelExpressionParser(configuration);

        //③创建取值上下文
        EvaluationContext context = new StandardEvaluationContext(user);

        //④表达式
        String expression = "isVipMember('tom') && isVipMember('jony')";

        //⑤解析表达式
        SpelExpression spelExpression = parser.parseRaw(expression);

        //⑥通过表达式求值
        System.out.println(spelExpression.getValue(context)); //第一次调用
        System.out.println(spelExpression.getValue(context)); //第二次调用
    }
}

```

在 SpEL 中启用编译模式，需要创建一个 `SpelParserConfiguration` 配置解析类，并指定编译模式和类加载器。其中，编译模式 SpEL 提供了一个枚举类型 `SpelCompilerMode`。该枚举共有 3 个值，分别是 `SpelCompilerMode.OFF`、`SpelCompilerMode.MIXED` 和 `SpelCompilerMode.IMMEDIATE`。

编译模式默认采用 `SpelCompilerMode.OFF` 值，表示不启用编译。想要全局开启，则可以通过配置类路径下的 `spring.properties` 配置文件，然后 `SpelParserConfiguration` 会在类加载时读取 `spring.expression.compiler.mode` 属性来进行配置。

`SpelCompilerMode.MIXED` 表示混合模式，表示在解释型和编译型之间进行转换；前面几次执行表达式取值采用解释型处理，直到达到 SpEL 的一个阈值（100）之后，才启用编译处理，也就是对表达式进行编译。

`SpelCompilerMode.IMMEDIATE` 表示立即启用编译。实际上，在 SpEL 内部不会立即启用编译，而是在第二次执行表达式取值时才会启用编译。

在②处通过配置解析对象创建解析器；在③处通过根对象实例 `user` 创建取值上下文；在④处编写一个求值表达式 `"isVipMember('tom') && isVipMember('jony')"`，其中 `isVipMember` 是 `User` 对象中声明的方法，根据用户名判断是否是 VIP 会员，“&&”表示逻辑与操作；在⑤处通过调用 `SpelExpressionParser#parseRaw()` 方法创建 `SpelExpression` 对象；在⑥处调用 `Expression#getValue()` 方法对表达式求值。

9.4 SpEL 基础表达式

9.4.1 文本字符解析

文本表达式支持字符串、日期、数字（正数、实数及十六进制数）、布尔类型及 null，其中字符串需使用单引号或反斜杠+双引号包含起来，如“'Hello World'”、“\"Hello World\"”，而单引号字符可使用“\\”表示。来看一个具体的示例，如代码清单 9-5 所示。

代码清单 9-5 LiteralExprSample 文本解析

```
import org.springframework.expression.EvaluationContext;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;
import java.util.*;

public class LiteralExprSample {
    public static void main(String[] args) {
        ExpressionParser parser = new SpelExpressionParser();

        //①解析字符串
        String helloWorld = (String) parser.parseExpression("\"Hello World\"").getValue();

        //②解析双精度浮点型
        double doubleNumber = (Double) parser.parseExpression("6.0221415E+23").getValue();

        //③解析整型
        int maxValue = (Integer) parser.parseExpression("0x7FFFFFFF").getValue();

        //④解析整型布尔型
        boolean trueValue = (Boolean) parser.parseExpression("true").getValue();

        //⑤解析空值
        Object nullValue = parser.parseExpression("null").getValue();
        ...
    }
}
```

本示例演示了如何使用 SpEL 表达式将文本字面值解析为相应的数据类型。数据类型支持负数、小数、科学计数法及八进制、十六进制数等。默认情况下采用 Double.parseDouble() 方法进行数值转换。

9.4.2 对象属性解析

在 SpEL 中，可使用类似“xxx.yyy.zzz”的对象属性路径轻松地访问对象属性值，如代码清单 9-6 所示。

代码清单 9-6 PropertyExprSample属性解析

```

package com.smart.spel;
import com.smart.User;
import org.springframework.expression.EvaluationContext;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;
import java.util.Date;

public class PropertyExprSample {
    public static void main(String[] args) {

        //①构造一个对象
        User user = new User();
        user.setUserName("tom");
        user.setLastVisit(new Date());
        user.setCredits(100); //设置积分
        user.setPlaceOfBirth(new PlaceOfBirth("中国","厦门")); //设置出生地

        //②构造SpEL解析上下文
        ExpressionParser parser = new SpelExpressionParser();
        EvaluationContext context = new StandardEvaluationContext(user);

        //③基本属性值获取
        String username = (String)parser.parseExpression("userName").getValue(context);
        int credits = (Integer) parser.parseExpression("credits +
10").getValue(context);

        //④嵌套对象属性值获取
        String city = (String)parser.parseExpression("placeOfBirth.city").getValue
(context);
        ...
    }
}

```

从上面的示例中可以看出，对象属性解析与文本字符解析有所不同，对象属性解析需要在取值的时候传递一个计算上下文参数 `EvaluationContext`。在这里，将 `User` 实例作为上下文的根对象传递给 `EvaluationContext`，这样 `SpEL` 表达式解析器就可以根据属性路径表达式获取上下文中根对象的属性值。

9.4.3 数组、集合类型解析

在 `SpEL` 中，支持数组、集合类型（`Map`、`List`）的解析。数组支持标准 Java 语言创建数组的方法，如 “`new int[]{1,2,3}`”。`List` 支持大括号括起来的内容，数据项之间用逗号隔开，如 “`{1,2,3,4}`”、“`{{'a','b'},{'x','y'}}`”。目前 `SpEL` 还不支持多维数组初始化，如 “`new int[2][3][4][5,6]`”。`Map` 采用如下方式表达：`{userName:'tom',credits:100}`，如代码清单 9-7 所示。

代码清单 9-7 CollectionExprSample属性解析

```

package com.smart.spel;
import org.springframework.expression.EvaluationContext;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;
import java.util.List;
import java.util.Map;

public class CollectionExprSample {
    public static void main(String[] args) {
        ...
        ExpressionParser parser = new SpelExpressionParser();
        EvaluationContext context = new StandardEvaluationContext(user);

        //①数组表达式解析
        int[] array1 = (int[]) parser.parseExpression("new int[]{1,2,3}").getValue(
            context);
        int[][] array2 = (int[][]) parser.parseExpression("new int[2][3]").getValue(
            context);

        //目前不支持多维数组初始化, 以下语句将报错
        int[][] array3= (int[][]) parser.parseExpression("new
            int[2][3]{{1,2,3},{4,5,6}}").getValue(context);

        //②List表达式解析
        List list = (List) parser.parseExpression("{1,2,3,4}").getValue(context);
        List listOfLists = (List)
        parser.parseExpression("{{'a','b'},{'x','y'}}").getValue(context);

        //③列表字符串解析
        Map userInfo = (Map) parser.parseExpression("{userName:'tom',credits:100 }").
            getValue(context);
        List userInfo2 = (List) parser.parseExpression("{{userName:'tom',credits:100 },
            {userName:'tom',credits:100 }}").getValue(context);

        //④从数组, List, Map中取值
        String interest1 =
            (String)parser.parseExpression("interestsArray[0]").getValue(context);
        String interest2 =
            (String)parser.parseExpression("interestsList[0]").getValue(context);
        String interest3 =
            (String)parser.parseExpression("interestsMap ['interest1']").getValue(
            context);
        ...
    }
}

```

在 SpEL 中, 要从解析器获取指定属性值, 需要将根对象存储到 EvaluationContext 上下文中, 如示例中的 “new StandardEvaluationContext(user)”。

数组与 List 取值通过指定括号内的索引获取, 如示例中的 “interestsArray[0]”、“interestsList[0]”, 其中 interestsArray、interestsList 均为 User 根对象的属性。

Map 取值通过键名获取，如示例中的 “interestsMap['interest1']”，其中 interestsMap 也是 User 对象的属性，“interest1” 为 Map 存储数据的键值。

9.4.4 方法解析

在 SpEL 中，方法调用支持 Java 可访问的方法，包括对象方法、静态方法，并支持可变方法参数；还可以调用 String 类型的所有可访问的方法，如 String#substring()，如代码清单 9-8 所示。

代码清单 9-8 MethodExprSample 方法解析

```
package com.smart.spel;
import com.smart.User;
import org.springframework.expression.EvaluationContext;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;

public class MethodExprSample {
    public static void main(String[] args) {
        User user = new User();
        ExpressionParser parser = new SpelExpressionParser();
        EvaluationContext context = new StandardEvaluationContext(user);

        //①调用String方法
        String substring = parser.parseExpression(
            "'Spring SpEL'.substring(7)").getValue(String.class);
        Integer index = parser.parseExpression(
            "'Spring SpEL'.indexOf('SpEL')").getValue(Integer.class);

        //②调用实例方法
        boolean isCorrect = parser.parseExpression("validatePassword('123456')").
            getValue(context, Boolean.class);
        //③调用私有方法，将发生错误
        isCorrect = parser.parseExpression("validatePassword2('123456')").getValue(
            context, Boolean.class);
        //④调用静态方法
        isCorrect = parser.parseExpression("validatePassword3('123456')").getValue(
            context, Boolean.class);

        //⑤调用对象方法，可变参数列表
        parser.parseExpression("addInterests('Js','C')").getValue(context, Boolean.
            class);
    }
}
```

在①处，在表达式中直接使用 String 提供的方法对文本字符进行各种操作。在②处，调用自定义类 User 的方法 User#validatePassword()。需要注意的是，私有方法是不能调用的，如示例中③处执行时将会报错。在④处，调用自定义类 User 的静态方法 User#validatePassword3()。在⑤处，调用自定义类 User 的可变参数方法 User# addInterests()。

9.4.5 操作符解析

SpEL 提供了丰富的操作符解析，支持关系操作符、逻辑操作符、算术运算操作符及正则表达式匹配等。

1. 关系操作符

在 SpEL 表达式中，可以使用标准 Java 操作符号，支持关系操作符：等于、不等于、小于、小于或等于、大于、大于或等于、正则表达式及 instanceof 操作符，如代码清单 9-9 所示。

代码清单 9-9 OperatorExprSample 关系操作符解析

```
package com.smart.spel;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;

public class OperatorExprSample {
    public static void main(String[] args) {
        ExpressionParser parser = new SpelExpressionParser();

        //①关系操作符
        boolean trueValue = parser.parseExpression("2 == 2").getValue(Boolean.class);
        boolean falseValue = parser.parseExpression("2 < -5.0").getValue(Boolean.class);

        //②字符串关系比较
        trueValue = parser.parseExpression("\"black\" < \"block\"").getValue(Boolean.class);

        //③instanceof运算符
        falseValue = parser.parseExpression("'xyz' instanceof T(int)").getValue(Boolean.class);

        //④正则匹配运算，前面为字符串，后面为正则表达式
        trueValue = parser.parseExpression("'5.00' matches '^-?\\d+(\\.\\d{2})?$'").getValue(Boolean.class);
        falseValue = parser.parseExpression("'5.0067' matches '^-?\\d+(\\.\\d{2})?$'").getValue(Boolean.class);
    }
}
```

Java 语言本身并未提供字符串关系比较操作符，但 SpEL 则支持字符串关系比较，如示例中②处的 “`\"black\" < \"block\"`”。instanceof 运算符是用来在运行时指出文本字符是否是特定类的一个实例，如示例中的 “`'xyz' instanceof T(int)`” 所示。需要注意的是 instanceof 操作符后面的类型表达格式：T(Java 类型)，如整型 T(int)、T(Integer)，字符类型 T(String)。在④处的表达式中使用 matches 关键字来对文本字符进行正则匹配运算，matches 后面是标准的 Java 正则表达式。

2. 逻辑操作符

逻辑操作符支持与操作 (and 或 &&)、或操作 (or 或 ||)、非操作 (!)。需要指出的是，在 SpEL 表达式中，既可以采用标准 Java 逻辑操作符，还可以采用 “and” 和 “or” 这两个操作符，如代码清单 9-10 所示。

代码清单 9-10 MethodExprSample逻辑操作符解析

```

package com.smart.spel;
import com.smart.User;
import org.springframework.expression.EvaluationContext;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;

public class MethodExprSample {
    public static void main(String[] args) {
        User user = new User();
        ExpressionParser parser = new SpelExpressionParser();
        EvaluationContext context = new StandardEvaluationContext(user);

        //①与操作, 结果为false
        boolean falseValue = parser.parseExpression("true && false").getValue(Boolean.class);

        String expression = "isVipMember('tom') and isVipMember('jony')";
        boolean trueValue = parser.parseExpression(expression).getValue(context, Boolean.class);

        //②或操作, 结果为true
        trueValue = parser.parseExpression("true or false").getValue(Boolean.class);

        //③取非操作, 结果为false
        falseValue = parser.parseExpression("!true").getValue(Boolean.class);
    }
}

```

在 SpEL 表达式中, 通过关系操作符关键字进行与操作, 关键字的前后运算结果必须是 Java 标准的布尔类型。如下列表达式是非法的: `true and 0`; 因为 `and` 操作符后面的“0”是整型值, 而不是一个布尔值。

3. 算术运算操作符

SpEL 表达式支持 Java 标准运算操作, 其中加法运算符可用于数字、字符串和日期, 减法运算符可用于数字和日期, 乘法和除法运算符仅可用于数字。其他支持的数学运算包括取模 (%) 和指数幂 (^), 使用 Java 标准的运算符优先级规则。如代码清单 9-11 所示。

代码清单 9-11 OperatorExprSample算术运算操作符解析

```

package com.smart.spel;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;

public class OperatorExprSample {
    public static void main(String[] args) {
        ExpressionParser parser = new SpelExpressionParser();

        //①加法操作, 运行结果等于1
        int two = parser.parseExpression("1 + 1").getValue(Integer.class);
    }
}

```



```

String testString = parser.parseExpression(
    "\"test\" + ' ' + \"string\"" ).getValue(String.class);

//②减法操作, 运行结果等于4
int four = parser.parseExpression("1 - 3").getValue(Integer.class);

//减法操作, 运行结果等于-9000
double d = parser.parseExpression("1000.00 - 1e4").getValue(Double.class);

//③乘法操作, 运行结果等于6
int six = parser.parseExpression("-2 * -3").getValue(Integer.class);

//乘法操作, 运行结果等于24.0
double twentyFour = parser.parseExpression("2.0 * 3e0 * 4").getValue(Double.
class);

//④除法操作, 运行结果等于-2
int minusTwo = parser.parseExpression("6 / -3").getValue(Integer.class);

//除法操作, 运行结果等于1.0
double one = parser.parseExpression("8.0 / 4e0 / 2").getValue(Double.class);

//⑤求余操作, 运行结果等于3
int three = parser.parseExpression("7 % 4").getValue(Integer.class);

// 求余操作, 运行结果等于1
one = parser.parseExpression("8 / 5 % 2").getValue(Integer.class);

//⑥优先级算术运算, 运行结果等于-21
int minusTwentyOne = parser.parseExpression("1+2-3*8").getValue(Integer.
class);
}
}

```

9.4.6 安全导航操作符

安全导航操作符来源于 Groovy 语言, 它避免了空指针异常。通常在访问对象方法或属性时需要验证该对象是否为空。为了避免烦琐的空对象验证, 可采用安全导航操作符, 它只会返回 `null` 而不是抛出异常。其格式是在获取对象属性操作符“.”前面添加一个“?”, 如代码清单 9-12 所示。

代码清单 9-12 SafeExprSample 安全导航操作符解析

```

package com.smart.spel;
import com.smart.PlaceOfBirth;
import com.smart.User;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;

import java.util.Date;

```

```

public class SafeExprSample {
    public static void main(String[] args) {
        User user = new User();
        user.setUserName("tom");
        user.setLastVisit(new Date());
        user.setCredits(100);
        user.setPlaceOfBirth(new PlaceOfBirth("中国", "厦门"));

        ExpressionParser parser = new SpelExpressionParser();
        StandardEvaluationContext context = new StandardEvaluationContext(user);
        String city = parser.parseExpression("PlaceOfBirth?.City").getValue(context,
String.class);
        System.out.println(city); //打印“厦门”

        //①将PlaceOfBirth对象设置为null
        user.setPlaceOfBirth(null);

        //②不会抛出异常
        city = parser.parseExpression("PlaceOfBirth?.City").getValue(context,
String.class);
        System.out.println(city); //打印“null”
    }
}

```

在上面的示例中，使用了 SpEL 提供的安全导航操作符，如 “PlaceOfBirth?.City”，在获取 PlaceOfBirth 对象属性时，会先判断 PlaceOfBirth 对象是否为空；如果为空则直接返回 null，如果不为空则获取其属性值。在 Java 8.0 中提供了 java.util.Optional 来支持类似的功能。

9.4.7 三元操作符

可以在表达式内使用 if-then-else 条件逻辑三元操作符，也就是 Java 标准的条件表达式：<表达式 1>?<表达式 2>:<表达式 3>。如表达式 “true?'value1': 'value2'”，将返回 “value1”。一个真实的例子如代码清单 9-13 所示。

代码清单 9-13 IfThenElseExprSample 三元操作符解析

```

package com.smart.spel;

import com.smart.PlaceOfBirth;
import com.smart.User;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;

import java.util.Date;

public class IfThenElseExprSample {
    public static void main(String[] args) {

```

```

User user = new User();
user.setUserName("tom");
user.setLastVisit(new Date());
user.setCredits(100);
user.setPlaceOfBirth(new PlaceOfBirth("中国", "厦门"));

ExpressionParser parser = new SpelExpressionParser();
StandardEvaluationContext context = new StandardEvaluationContext(user);

//三元操作表达式
String expression = "UserName == 'tom'? Credits+10:Credits";

Integer credits = parser.parseExpression(expression).getValue(context,
Integer.class);
System.out.println(credits); // 输出 "110"
}
}

```

在示例中，创建了一个用户名为“tom”的用户实例“user”，并设置其积分为100。在三元操作表达式中判断当前是否存在一个“tom”用户，如果存在，就添加10个积分。

9.4.8 Elvis 操作符

Elvis 操作符是 Groovy 语言中使用的三元操作符的缩写。在三元运算符中通常要重复变量两次，例如：

```

String name = "tom";
String displayName = name!=null?name:"Unknown";

```

可以使用 Elvis 操作符替代：name?:"Unknown"，代码非常简洁。下面来看一下具体的实例，如代码清单 9-14 所示。

代码清单 9-14 ElvisExprSample 操作符解析

```

package com.smart.spel;

import com.smart.PlaceOfBirth;
import com.smart.User;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;
import java.util.Date;

public class ElvisExprSample {
    public static void main(String[] args) {
        User user = new User();
        user.setUserName("tom");
        user.setLastVisit(new Date());
        user.setCredits(100);
        user.setPlaceOfBirth(new PlaceOfBirth("中国", "厦门"));

        ExpressionParser parser = new SpelExpressionParser();
        StandardEvaluationContext context = new StandardEvaluationContext(user);
    }
}

```

```
String userName = parser.parseExpression("UserName?:'用户名为空']").getValue
                                (context, String.class);
System.out.println(userName); //输出“tom”
user.setUserName(null);
userName = parser.parseExpression("UserName?:'用户名为空']").getValue(context,
                                String.class);
System.out.println(userName); //输出“用户名为空”
}
}
```

SpEL 提供的 Elvis 操作符非常简洁，其格式为“<var>?:<value>”。如果左边变量取值为 null，就取 value 值；否则就取 var 变量自身的值。

9.4.9 赋值、类型、构造器、变量

1. 赋值

属性设置是通过使用赋值运算符完成的，相当于调用 Expression#setValue()方法，也可以通过调用 Expression#getValue()方法赋值，如代码清单 9-15 所示。

代码清单 9-15 ObjectExprSample赋值

```
package com.smart.spel;
import com.smart.PlaceOfBirth;
import com.smart.User;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;
import java.util.Date;

public class ObjectExprSample {
    public static void main(String[] args) {
        User user = new User();
        user.setUserName("tom");

        ExpressionParser parser = new SpelExpressionParser();
        EvaluationContext context = new StandardEvaluationContext(user);

        //①通过setValue赋值
        parser.parseExpression("userName").setValue(context, "jony");
        System.out.println(user.getUserName()); //输出“jony”

        //②通过表达式赋值
        parser.parseExpression("userName='anyli']").getValue(context);
        System.out.println(user.getUserName()); //输出“anyli”
    }
}
```

在①处通过 Expression 接口提供的 setValue()方法为表达式上下文对象属性赋值；在②处在调用 getValue()方法时通过赋值表达式直接赋值，如示例中的“userName='anyli'”。

2. 类型

T 操作符是一个非常实用的类型操作符，通过它可以从类路径加载指定类名称

(全限定名)的 Class 对象,操作表达式为“T(全限定类名)”;其功能类似于 ClassLoader#loadClass()方法,如代码清单 9-16 所示。

代码清单 9-16 ObjectExprSample类型

```
package com.smart.spel;
import com.smart.PlaceOfBirth;
import com.smart.User;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;

public class ObjectExprSample {
    public static void main(String[] args) {
        ExpressionParser parser = new SpelExpressionParser();

        //①加载java.lang.string
        Class stringClass=parser.parseExpression("T(java.lang.String)").getValue(
            Class.class);
        System.out.println(stringClass==java.lang.String.class); //输出 true

        //②加载com.smart.User
        Class userClass=parser.parseExpression("T(com.smart.User)").getValue(Class.
            class);
        System.out.println(userClass == com.smart.User.class); //输出 true
    }
}
```

在①处通过 T 操作符加载 java.lang.String 类,返回类 String 的 Class 对象;在②处通过 T 操作符加载自定义类 User 的 Class 对象。在 SpEL 内部,通过 StandardTypeLocator#findType()方法加载类。上述类的名称都使用全限定类名。如果加载的目标类位于 java.lang 包下,则可以不带包名;对于其他包下的类,则必须带上完整的包名。

T 操作符还可以直接调用类静态方法,操作表达式为“T(全限定类名).静态方法”。例如,可以使用 T 操作符直接调用 Math# random()方法,获取一个随机数,如下:

```
Object randomValue = parser.parseExpression("T(java.lang.Math).random()").getValue();
System.out.println(randomValue); //返回一个随机数
```

通过 T 操作符可以在 SpEL 表达式中调用工具类中的任意静态方法。这是一个非常实用的特性,如编写一个日期工具类 DateUtils,假设其中有一个解析日期的静态方法 parseDate(),则可以通过 T 操作符直接调用“T(com.smart.DateUtils). parseDate('20160501')”。

3. 构造器

可以使用 new 运算符调用构造器来创建一个对象实例。除了基本类型(如整型、浮点型)和字符串,其他类需要使用全限定类名,如代码清单 9-17 所示。

代码清单 9-17 ObjectExprSample构造器

```

package com.smart.spel;

...

public class ObjectExprSample {
    public static void main(String[] args) {
        ExpressionParser parser = new SpelExpressionParser();

        User user = parser.parseExpression("new com.smart.User('tom')").getValue(User.
class);
        System.out.println(user.getUserName()); //输出“tom”
    }
}

```

4. 变量

变量可以在表达式中使用语法“#变量名”引用，变量设置使用 `EvaluationContext#setVariable(var, value)`方法，其中 `var` 为变量名称，`value` 为变量对应的值，如代码清单 9-18 所示。

代码清单 9-18 ObjectExprSample变量解析

```

package com.smart.spel;

...

public class ObjectExprSample {
    public static void main(String[] args) {
        User user = new User("tom");
        ExpressionParser parser = new SpelExpressionParser();
        EvaluationContext context = new StandardEvaluationContext(user);

        //①为newUserName变量设置新值
        context.setVariable("newUserName", "jony");

        //②取变量值，并赋值
        parser.parseExpression("userName=#newUserName").getValue(context);
        System.out.println(user.getUserName()); //输出“jony”

        //③ this变量值使用
        List<Integer> credits= new ArrayList<Integer>();
        credits.addAll(Arrays.asList(150,100,90,50,110,130,70));
        context.setVariable("credits",credits);
        List<Integer> creditsGreater100=
            (List<Integer>)parser.parseExpression("#credits.?[#this>100]").
getValue(context);
    }
}

```

在①处通过调用 `EvaluationContext#setVariable()`方法在动态上下文中定义一个新的变量“newUserName”。在②处的表达式中通过“#变量名”访问这个变量的值，并将取到的值赋给 `User` 对象的 `userName` 属性。在③处创建了一个积分列表 `credits`，并将积分列表设置到动态上下文中。

9.4.10 集合过滤

集合过滤是动态语言的一个强大功能，它允许通过一个过滤条件获取原集合的子集。集合过滤的语法为“?[selectExpression]”。例如，从一个积分列表中找到大于 100 的积分，代码如下：

```
List<Integer> creditsGreater100=
    (List<Integer>)parser.parseExpression("#credits.?[#this>100]").getValue(context);
```

对于 List 或 Set 来说，过滤条件是针对集合内的每个元素进行比较的；而对于 Map 来说，则是针对每一个条目项（Java 类型的 Map.Entry 对象）进行运算的，如代码清单 9-19 所示。

代码清单 9-19 ObjectExprSample 集合选择

```
package com.smart.spel;
...
public class ObjectExprSample {
    public static void main(String[] args) {
        ExpressionParser parser = new SpelExpressionParser();
        EvaluationContext context = new StandardEvaluationContext();
        Map<String,Integer> creditsMap = new HashMap();
        creditsMap.put("Tom",95);
        creditsMap.put("Jony",110);
        creditsMap.put("Morin",85);
        creditsMap.put("Mose",120);
        creditsMap.put("Morrow",60);
        context.setVariable("credits",creditsMap);

        Map<String,Integer> creditsGreater100=

        (Map<String,Integer>)parser.parseExpression("#credits.?[value>90]").getVal
        ue(context);
    }
}
```

“#credits.?[value>90]”表达式将返回一个新的 Map，即过滤出值大于 100 的条目。默认返回所有匹配的结果，也可以仅返回第一个或最后一个匹配项：用^[...]获得第一个匹配值，如#credits.^[value>90]；用\$[...]获得最后一个匹配值，如#credits.\$[value>90]。

9.4.11 集合转换

集合转换允许使用一个算子对集合内的元素进行运算，得到一个新的集合。集合转换的语法为“![projectionExpression]”。来看一个具体的例子，如代码清单 9-20 所示。

代码清单 9-20 ObjectExprSample 集合转换

```
package com.smart.spel;
...
public class ObjectExprSample {
    public static void main(String[] args) {
```



```

ExpressionParser parser = new SpelExpressionParser();
EvaluationContext context = new StandardEvaluationContext();

List<Integer> credits= new ArrayList<Integer>();
credits.addAll(Arrays.asList(150,100,90,50,110,130,70));
context.setVariable("credits",credits);

List<Boolean> creditsGreater100=
    (List<Boolean>)parser.parseExpression("#credits.![#this>100]").getValue
(context);
}
}

```

在示例中, 对 List 集合元素进行转换, 得到一个判断积分是否大于 100 的结果集合: [true,false,false,true,true,false]。类似的, Map 也可以进行转换, 此时转换表达式是对 Map 中的每个项目运用算子进行求值。

9.5 在 Spring 中使用 SpEL

在 XML 配置方式或注解配置方式中都可以使用 SpEL 表达式进行一些高级配置, 两种方式都采用统一的语法使用 SpEL 表达式: `#{<expression string>}`。

9.5.1 基于 XML 的配置

在 Bean 配置中, 可以使用 SpEL 表达式为 Bean 属性或构造函数入参注入动态值。下面以 Bean 属性使用 SpEL 表达式为例, 如下:

```

<bean id="numberGuess" class="org.springframework.samples.NumberGuess"
    p:randomNumber= "#{T(java.lang.Math).random()*100.0}"
/>

```

在示例中, 通过 SpEL 提供的 T 类型操作符, 直接调用 `java.lang.Math` 的静态方法来生成一个随机数, 并把生成的随机数乘以 100.0 后赋值给 `NumberGuess` 的 `randomNumber` 属性。还可以通过 `systemProperties` 获取各个系统环境变量, 如下:

```

<bean id="systemPropertyBean" class="com.smart.spel.SystemPropertyBean"
    p:osName= "#{systemProperties['os.name']}"
    p:javaHome= "#{systemProperties['os.name']}"
    p:classPath= "#{systemProperties['java.class.path']}"
    p:javaVersion= "#{systemProperties['java.class.path']}"
/>

```

可以在 Bean 配置时引用任何一个在 Spring 容器中已定义的 Bean 的属性(属性提供了相应的 `get` 方法), 通过“Bean 名称.属性名”的方式来引用其他 Bean 的属性, 如下:

```

<bean id="numberGuess" class="org.springframework.samples.NumberGuess"
    p:randomNumber= "#{T(java.lang.Math).random()*100.0}"
/>
<bean id="shapeGuess" class="org.springframework.samples.ShapeGuess"

```

```
p:initialShapeSeed = "#{numberGuess.randomNumber}"
/>
```

在上述代码中，定义了 NumberGuess 和 ShapeGuess 两个 Bean，ShapeGuess 中的 initialShapeSeed 属性通过表达式引用 NumberGuess 中定义的 randomNumber 属性。

9.5.2 基于注解的配置

@Value 注解可以标注在类的属性、方法及构造器函数上，用于从配置文件中加载一个参数值。下面是一个设置属性示例：

```
@Component
public class MyDataSource {

    @Value("#{properties['driverClassName']}")
    private String driverClassName;

    @Value("#{properties['url']}")
    private String url;

    @Value("#{properties['userName']}")
    private String userName;

    @Value("#{properties['password']}")
    private String password;

}
```

上述代码通过 @Value 注解自动注入属性配置文件中的属性选项值，其中 properties 是定义的加载配置文件的 Bean 的名称。要让上述代码能够正常运行，需要在 Spring 中引入 util 工具命名空间，如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
    ...
    http://www.springframework.org/schema/util
    http://www.springframework.org/schema/util/spring-util-4.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.0.xsd">
    <util:properties id="properties" location="classpath:jdbc.properties" />
</beans>
```

上述 jdbc.properties 配置文件的配置选项如下：

```
driverClassName=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/dbName
userName=root
password=1234
```

细心的读者可能会发现，类似 “#{properties['password']}” 这种写法比较容易出错。Spring 提供了一种更简便的写法——属性占位符，只要在 Spring 的配置方法中添加一个 “property-placeholder”，就可以在表达式中使用 “\${属性}”，如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
    ...
    http://www.springframework.org/schema/util
    http://www.springframework.org/schema/util/spring-util-4.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.0.xsd">
    <util:properties id="properties" location="classpath:jdbc.properties" />
    <context:property-placeholder properties-ref="properties" />
</beans>
```

可以把上述代码改为如下形式：

```
@Component
public class MyDataSource {

    @Value("${driverClassName}")
    private String driverClassName;

    @Value("${url}")
    private String url;

    ...
}
```

在 Bean 的方法及构造函数上使用 @Value 注解与类属性的使用方法类似，这里不再赘述。

9.6 小结

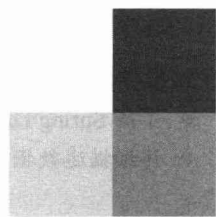
在实际项目开发中，对于动态表达式的需求场景还是比较多的。早期，大家可能会在 Java 中引入一些动态脚本语言如 Rhino、Jython、JRuby 等类库来实现。虽然这些动态脚本语言可以实现各类复杂的表达式需求，但需要自行封装实现，工作量很大。

SpEL 表达式的出现为我们提供了一个轻量级的表达式框架，它实现了一套丰富的操作表达式，支持文本、对象、集合、方法表达式解析，并提供了丰富的表达式操作，可以满足大多数表达式场景需求。SpEL 已深入整合到 Spring 框架的 Bean 配置中，使用 SpEL 可以完成众多高级的 Bean 配置问题。



第 3 篇

数 据 篇



第 10 章

Spring 对 DAO 的支持

随着持久化技术的持续发展，各种持久化框架已趋于成熟，Oracle 也发布了几个持久化规范。Spring 对多个持久化技术提供了集成支持，包括 Hibernate、MyBatis、JPA、JDO；此外，还提供一个简化 JDBC API 操作的 Spring JDBC 框架。Spring 面向 DAO 制定了一个通用的异常体系，屏蔽具体持久化技术的异常，使业务层和具体的持久化技术实现解耦。另外，Spring 提供了模板类简化各种持久化技术的使用。通用的异常体系及模板类是 Spring 整合各种持久化技术的不二法门，Spring 不但借此实现了对多种持久化技术的整合，还可以整合潜在的各种持久化框架，体现了开发模式中“开-闭原则”的经典应用。

本章主要内容：

- ◆ Spring DAO 异常体系
- ◆ Spring 数据访问模板
- ◆ 配置数据源

本章亮点：

- ◆ 了解 Spring DAO 层的设计思想
- ◆ 详细描述数据源的配置

10.1 Spring 的 DAO 理念

DAO (Data Access Object) 是用于访问数据的对象，虽然在大多数情况下将数据保存在数据库中，但这并不是唯一的选择，也可以将数据存储到文件中或 LDAP 中。DAO 不但屏蔽了数据存储的最终介质的不同，也屏蔽了具体的实现技术的不同。

早期，JDBC 是访问数据库的主流选择。近几年，数据持久化技术获得了长足的发

展, Hibernate、MyBatis、JPA、JDO 成为持久层中争放异彩的实现技术。只要为数据访问定义好 DAO 接口, 并使用具体的技术实现 DAO 接口的功能, 就可以在不同的实现技术间平滑地切换。

图 10-1 是一个典型的 DAO 应用实例, 在 UserDao 中定义访问 User 数据对象的接口方法, 业务层通过 UserDao 操作数据, 并使用具体的持久化技术实现 UserDao 接口方法, 这样业务层和具体的持久化技术就实现了解耦。

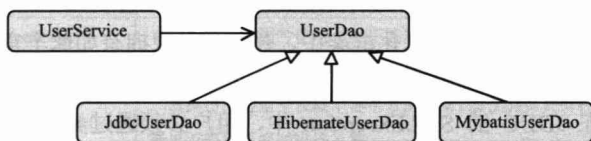


图 10-1 业务层通过 DAO 接口访问数据

提供 DAO 层的抽象可以带来一些好处: 首先, 可以很容易地构造模拟对象, 方便单元测试的开展; 其次, 在使用切面时会有更多的选择, 既可以使用 JDK 动态代理, 又可以使用 CGLib 动态代理。

Spring 本质上希望以统一的方式整合底层的持久化技术, 即以统一的方式进行调用及事务管理, 避免让具体的实现侵入到业务层的代码中。由于每种持久化技术都有各自的异常体系, 所以 Spring 提供了统一的异常体系, 使不同异常体系的阻抗得以消弭, 方便定义出和具体实现技术无关的 DAO 接口, 以及整合到相同的事务管理体系中。

10.2 统一的异常体系

统一的异常体系是整合不同的持久化技术的关键。Spring 提供了一套和实现技术无关的、面向 DAO 层语义的异常体系, 并通过转换器将不同持久化技术的异常转换成 Spring 的异常。

10.2.1 Spring 的 DAO 异常体系

在很多正统 API 或框架中, 检查型异常被过多地使用, 以至在使用 API 时, 代码里充斥着大量 try/catch 样板式的代码。在很多情况下, 除在 try/catch 中记录异常信息外, 并没有做多少实质性的工作。引发异常的问题往往是不可恢复的, 如数据连接失败、SQL 语句存在语法错误等。强制捕捉的检查型异常除限制开发人员的自由外, 并没有提供什么有价值的东西。因此, Spring 的异常体系都是建立在运行期异常的基础上的, 开发者可以根据需要捕捉感兴趣的异常。

JDK 的很多 API 之所以难用, 一个很大的原因就是检查型异常的泛滥, 如 JavaMail、EJB、JDBC 等。使用这些 API, 一堆堆异常处理的代码喧宾夺主地侵入到业务代码中,

破坏了代码的整洁和优雅。

Spring 在 org.springframework.dao 包中提供了一套完备优雅的 DAO 异常体系, 这些异常都继承于 `DataAccessException`, 而 `DataAccessException` 本身又继承于 `NestedRuntimeException`, `NestedRuntimeException` 异常以嵌套的方式封装了源异常。因此, 虽然不同持久化技术的特定异常被转换到 Spring 的 DAO 异常体系中, 但原始的异常信息并不会丢失; 只要用户愿意, 就可以方便地通过 `getCause()` 方法获取原始的异常信息。

Spring 的 DAO 异常体系并不和具体的实现技术相关, 它从 DAO 概念的抽象层面定义了异常的目录树。在所有的持久化框架中, 并没有发现拥有如此丰富语义的异常体系的框架。Spring 的这种设计无疑是独具匠心的, 它使得开发人员关注某一特定语义的异常变得很容易。在 JDBC 的 `SQLException` 中, 用户必须通过异常的 `getErrorCode()` 或 `getSQLState()` 方法获取错误代码, 然后根据这些代码判断错误原因。这种过于底层的 API 不但带来了代码编程上的难度, 而且也使代码的移植变得困难, 因为 `getErrorCode()` 方法是数据库相关的。

Spring 以分类手法建立了异常分类目录, 对于大部分应用来说, 这个异常分类目录对异常类型的划分具有适当的颗粒度。一方面, 使开发者从底层细如针麻的技术细节中脱离出来; 另一方面, 可以从这个语义丰富的异常体系中选择感兴趣的异常加以处理。图 10-2 列出了那些位于 Spring DAO 异常体系第一层次的异常类, 每个异常类下可能拥有众多的子异常类。

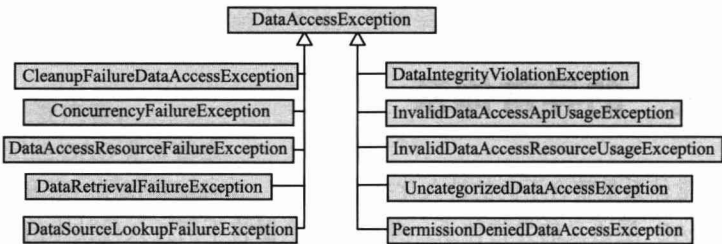


图 10-2 Spring DAO 异常体系

Spring DAO 异常体系类非常丰富, 这里仅列出 `DataAccessException` 异常类下的子类。可以很容易地通过异常类的名字了解异常所代表的语义。下面通过表 10-1 对这些异常进行简单的描述。

表 10-1 Spring DAO异常体系类

异 常	说 明
CleanupFailureDataAccessException	DAO 操作成功执行, 但在释放数据资源时发生异常, 如关闭 Connection 时发生异常等
ConcurrencyFailureException	表示在进行并发数据操作时发生异常, 如乐观锁无法获取、悲观锁无法获取、死锁引发的失败等
DataAccessResourceFailureException	访问数据资源时失败, 如无法获取数据连接、无法获取 Hibernate 的会话等
DataRetrievalFailureException	获取数据失败, 如找不到对应主键的数据、使用了错误的列索引等

续表

异常	说明
DataSourceLookupFailureException	无法从 JNDI 中查找到数据源
DataIntegrityViolationException	当数据操作违反了数据一致性限制时抛出的异常，如插入重复的主键、引用不存在的外键等
InvalidDataAccessApiUsageException	不正确地调用某一持久化技术时抛出的异常，如在 Spring JDBC 中查询对象，在调用前必须进行编译操作，如果忘记这项操作则会产生该异常。这种异常不是由底层数据资源产生的，而是由不正确地使用持久化技术产生的
InvalidDataAccessResourceUsageException	在访问数据源时使用了不正确的方法所抛出的异常，如 SQL 语句错误将抛出该异常
PermissionDeniedDataAccessException	数据访问时由于权限不足引发的异常，如仅拥有只读权限的用户试图进行数据更改操作时将抛出该异常
UncategorizedDataAccessException	其他未分类的异常都归到该异常中

为了进一步细化错误的问题域，Spring 对一级异常类进行了子类的细分，如 InvalidDataAccessResourceUsageException 就拥有十多个子类，下面是其中的两个子类，如图 10-3 所示。

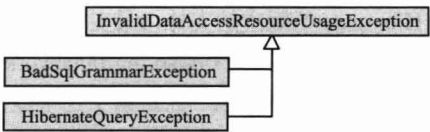


图 10-3 一级异常类的细分

对于 InvalidDataAccessResourceUsageException 异常，不同的持久化技术均有对应的子异常类。如 BadSqlGrammarException 对应 JDBC 实现技术的 SQL 语句语法错误异常，而 HibernateQueryException 对应 Hibernate 实现技术的查询语法异常。

Spring 的这个异常体系具有高度的可扩展性，当 Spring 需要对一种新的持久化技术提供支持时，只要为其定义一个对应的子异常就可以了，这种更改完全满足设计模式中的“开-闭原则”。

虽然 Spring 定义了如此丰富的异常类，但作为开发人员，仅需对感兴趣的异常进行处理即可。假设某个项目要求在发生乐观锁异常时，尝试再次获取乐观锁，而不是直接返回错误；那么，只需在代码中显式捕捉 ConcurrencyFailureException 异常，然后在 catch 代码块中编写满足需求的逻辑即可。其他众多的异常则可以简单地交由框架自动处理，如发生运行期异常时自动回滚事务。

10.2.2 JDBC 的异常转换器

传统的 JDBC API 在发生几乎所有的数据操作问题时都会抛出相同的 SQLException，它将异常的细节性信息封装在异常属性中。所以，如果希望了解异常的具体原因，则必须分析异常对象的信息。

SQLException 拥有两个代表异常具体原因的属性：错误码和 SQL 状态码。前者是数据库相关的，可通过 `getErrorCode()` 方法返回，其值的类型是 `int`；而后者是一个标准的错误代码，可通过 `getSQLState()` 方法返回，是一个 `String` 类型的值，由 5 个字符组成。

Spring 根据错误码和 SQL 状态码信息将 SQLException 译成 Spring DAO 的异常体系所对应的异常。在 `org.springframework.jdbc.support` 包中定义了 `SQLExceptionTranslator` 接口，该接口的两个实现类 `SQLErrorCodeSQLExceptionTranslator` 和 `SQLStateSQLExceptionTranslator` 分别负责处理 SQLException 中错误码和 SQL 状态码的翻译工作。将 SQLException 翻译成 Spring DAO 异常体系的工作是比较困难的，但 Spring 框架替我们完成了这项艰巨的工作并保证了转换的正确性，我们有充分的理由依赖这个转换的正确性。

10.2.3 其他持久化技术的异常转换器

由于各种框架级的持久化技术都拥有一个语义明确的异常体系，所以将这些异常转换为 Spring DAO 的体系相对轻松一些。下面将学习不同持久化技术的异常转换器。

Spring 4.0 移除了对 Hibernate 低版本的支持，只支持 Hibernate 3.6 之后的版本。另外，Spring 4.0 移除了对 TopLink 的支持。在 `org.springframework.orm` 包中，分别为 Spring 所支持的 ORM 持久化技术定义了一个子包，在这些子包中提供相应 ORM 技术的整合类。Spring 为各种 ORM 持久化技术所提供的异常转换器在表 10-2 中说明。

表 10-2 各ORM持久化技术异常转换器

ORM 持久化技术	异常转换器
Hibernate X.0 (X 可为 3,4,5, 下同)	<code>org.springframework.orm.hibernateX.SessionFactoryUtils</code>
JPA	<code>org.springframework.orm.jpa.EntityManagerFactoryUtils</code>
JDO	<code>org.springframework.orm.jdo.PersistenceManagerFactoryUtils</code>

这些工具类除了具有异常转换的功能，在进行事务管理时，还提供了从事务上下文中返回相同会话的功能（将在第 11 章深入讲解事务管理的知识）。

Spring 也支持 MyBatis ORM 持久化技术，由于 MyBatis 抛出的异常是和 JDBC 相同的 SQLException 异常，所以直接采用和 JDBC 相同的异常转换器。

10.3 统一数据访问模板

到一个餐馆用餐，大抵会经历这样一个流程：进入餐馆→迎宾小姐问候并引到适合的位置→拿起菜单点菜→用餐→埋单→离开餐馆。之所以我们喜欢时不时到餐馆用餐，就是因为只要点菜→用餐→埋单就可以了，幕后的烹饪制作、刷锅洗盘等工作完全不用关心，一切已经由餐馆服务人员按照服务流程按部就班、有条不紊地执行了。衡量

一个餐馆服务质量好坏的一个重要标准是我们无须关心他们所负责的流程：不用催问菜为什么还没有上好（不但快而且服务态度佳），不用关心盘子为什么不干净（不但干净而且已经进行了消毒）。

从某种角度看，与其说餐馆为我们提供了服务，还不如说我们参与到餐馆的流程中：不管什么顾客点的菜都由相同的厨师烹制，不管什么顾客都按单付钱。在幕后，餐馆拥有一个服务模板，模板中定义的流程可以用于应付所有的顾客，只要为顾客提供几个专有需求（点的菜可不一样，座位可以自由选择），其他一切都按模板化的方式处理。

在直接使用具体的持久化技术时，大多需要处理整个流程，并没有享受餐馆用餐式的便捷。Spring 为支持的持久化技术分别提供了模板访问的方式，降低了使用各种持久化技术的难度，因此可以大幅度地提高开发效率。

10.3.1 使用模板和回调机制

下面是一段使用 JDBC 进行数据访问操作的简单代码，我们已经尽可能地简化了整个过程的处理，但以下步骤都是不可或缺的，如代码清单 10-1 所示。

代码清单 10-1 JDBC数据访问

```
public void saveCustomer(Customer customer) throws Exception {

    Connection con=null;
    PreparedStatement stmt=null;
    try {

        //①获取资源
        con=getConnection();

        //②启动事务
        con.setAutoCommit(false);

        //③具体的数据访问操作和处理
        stmt=con.prepareStatement("insert into CUSTOMERS(ID,NAME) values(?,?)");
        stmt.setLong(1,customerId);
        stmt.setString(2,customer.getName());
        stmt.execute();
        ...
        stmt.execute();

        //④提交事务
        con.commit();
    }catch(Exception e){
        try{
            //⑤回滚事务
            con.rollback();
        }catch(SQLException sqllex){
            sqllex.printStackTrace(System.out);
        }
        throw e;
    }
}
```

```

}finally{
    //⑥释放资源
    try{
        stmt.close();
        con.close();
    }catch(Exception e){
        e.printStackTrace();
    }
}
}

```

如上述代码所示，JDBC 数据访问操作按以下流程进行：

- (1) 准备资源。
- (2) 启动事务。
- (3) 在事务中执行具体的数据访问操作。
- (4) 提交/回滚事务。
- (5) 关闭资源，处理异常。

按照传统的方式，在编写任何带事务的数据访问程序时，都需要重复编写上面的代码，而其中只有粗体部分所示的代码是业务相关的，其他代码都是在例行公事，因而导致大量八股文式的代码充斥着整个程序。



轻松一刻

八股文的每篇文章均由一定的格式、字数构成，即由破题、承题、起讲、入手、起股、中股、后股、束股八部分组成。破题是用两句话将题目的意义破开；承题是承接破题的意义而说明之；起讲为议论的开始，首二字用“意味”、“若曰”、“以为”、“且夫”、“尝思”等开端；入手为起讲；起股、中股、后股、束股才是正式议论，以中股为全篇重心。在这四股中，每股又有两股排比对称的文字，合共八股，故名八股文。



Spring 将这个相同的数据访问流程固化到模板类中，并将数据访问中固定和变化的部分分开，同时保证模板类是线程安全的，以便多个数据访问线程共享同一个模板实例。固定的部分在模板类中已经准备好，而变化的部分通过回调接口开放出来，用于定义具体数据访问和结果返回的操作。图 10-4 描述了模板类拆分固定和变化部分的逻辑。

这样，只要编写好回调接口，并调用模板类进行数据访问，就可以得到预想的结果：数据访问成功执行，前置和后置的样板化工作也按顺序正确执行，在提高开发效率的同时保证了资源使用的正确性，彻底消除了因忘记进行资源释放而引起的资源泄露问题。

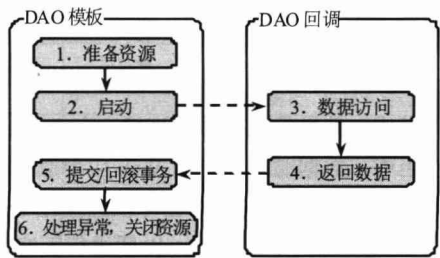


图 10-4 Spring DAO 模板和回调

10.3.2 Spring 为不同持久化技术所提供的模板类

Spring 为各种支持的持久化技术都提供了简化操作的模板和回调，在回调中编写具体的数据操作逻辑，使用模板执行数据操作，在 Spring 中，这是典型的数据操作模式。下面来了解一下 Spring 为不同的持久化技术所提供的模板类，如表 10-3 所示。关于模板类的具体使用，将在本书后续章节介绍。

表 10-3 不同持久化技术对应的模板类

ORM 持久化技术	模 板 类
JDBC	org.springframework.jdbc.core.JdbcTemplate
Hibernate X.0	org.springframework.orm.hibernateX.HibernateTemplate
JPA	org.springframework.orm.jpa.JpaTemplate
JDO	org.springframework.orm.jdo.JdoTemplate

如果直接使用模板类，则一般需要在 DAO 中定义一个模板对象并提供数据资源。Spring 为每种持久化技术都提供了支持类，支持类中已经完成了这样的功能。这样，只需扩展这些支持类，就可以直接编写实际的数据访问逻辑，因此更加方便。

不同持久化技术的支持类如表 10-4 所示。

表 10-4 持久化技术的支持类

ORM 持久化技术	支 持 类
JDBC	org.springframework.jdbc.core.JdbcDaoSupport
Hibernate X.0	org.springframework.orm.hibernateX.HibernateDaoSupport
JPA	org.springframework.orm.jpa.JpaDaoSupport
JDO	org.springframework.orm.jdo.JdoDaoSupport

这些支持类都继承于 dao.support.DaoSupport 类，DaoSupport 类实现了 InitializingBean 接口，在 afterPropertiesSet()接口方法中检查模板对象和数据源是否被正确设置，否则将抛出异常。

所有的支持类都是 abstract 的，其目的是希望被继承使用，而非直接使用。

10.4 数据源

不管采用何种持久化技术，都必须拥有数据连接。在 Spring 中，数据连接是通过数据源获得的。在以往的应用中，数据源一般是由 Web 应用服务器提供的。在 Spring 中，不但可以通过 JNDI 获取应用服务器的数据源，也可以直接在 Spring 容器中配置数据源。此外，还可以通过代码的方式创建一个数据源，以便进行无容器依赖的单元测试。

10.4.1 配置一个数据源

Spring 在第三方依赖包中包含了两个数据源的实现类包：其一是 Apache 的 DBCP；其二是 C3P0。可以在 Spring 配置文件中利用二者中的任何一个配置数据源。

1. DBCP 数据源

DBCP 是一个依赖 Jakarta commons-pool 对象池机制的数据库连接池，所以在类路径下还必须包括 commons-pool 的类包。下面是使用 DBCP 配置 MySQL 数据源的片段：

```
<bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close"
      p: driverClassName="com.mysql.jdbc.Driver"
      p: url="jdbc:mysql://localhost:3309/sampledbs"
      p: username="root"
      p: password="1234"/>
```

BasicDataSource 提供了 close()方法关闭数据源，所以必须设定 destroy-method="close"属性，以便 Spring 容器关闭时，数据源能够正常关闭。

假设数据库是 MySQL，如果数据源配置不当，则将可能发生经典的“8 小时问题”。原因是 MySQL 在默认情况下如果发现一个连接的空闲时间超过 8 小时，则将会在数据库端自动关闭这个连接。而数据源并不知道这个连接已经被数据库关闭了，当它把这个无用的连接返回给某个 DAO 时，DAO 就会报无法获取 Connection 的异常。

除以上必需的数据源属性外，还有一些常用的属性，如表 10-5 所示。

表 10-5 DBCP设置参数说明

分类	属 性	默认值	说 明
事务属性	defaultAutoCommit	true	连接池创建的连接的默认 auto-commit 状态
	defaultReadOnly	驱动默认	连接池创建的连接的默认 read-only 状态。如果没有设置，则 setReadOnly()方法将不会被调用
	defaultTransactionIsolation	驱动默认	连接池创建的连接的默认的 TransactionIsolation 状态，可选值包括：NONE、READ_COMMITTED、READ_UNCOMMITTED、REPEATABLE_READ 和 SERIALIZABLE

续表

分类	属 性	默认值	说 明
数据源连接数量	initialSize	0	初始化连接：连接池启动时创建的初始化连接数量
	maxActive	8	最大活动连接：连接池在同一时间能够分配的最大活动连接数量。如果设置为非正数，则表示不限制
	maxIdle	8	最大空闲连接：连接池中允许保持空闲状态的最大连接数量，超过的空闲连接将被释放。如果设置为负数，则表示不限制
	minIdle	0	最小空闲连接：连接池中允许保持空闲状态的最小连接数量，低于这个数量将创建新的连接。如果设置为 0，则不创建
	maxWait	无限	最大等待时间：当没有可用连接时，连接池等待连接被归还的最大时间（以毫秒计数），超过时间则抛出异常。如果设置为 -1，则表示无限等待
连接健康情况维护和检测	validationQuery	无默认值	SQL 查询语句。在将连接返回给调用者之前，用此 SQL 验证从连接池取出的连接是否可用。如果指定，则查询必须是一个 SQL SELECT，并且必须返回至少一行记录。在 MySQL 中可以设置为“select 1”，在 Oracle 中可以设置为“select 1 from dual”
	testOnBorrow	true	指明是否在从连接池中取出连接前进行检验，如果检验失败，则从连接池中去除该连接并尝试取出另一个新的连接。注意：设置为 true 后如果要生效，则 validationQuery 参数必须正确设置
	testOnReturn	false	指明是否在归还到连接池中前进行检验。注意：设置为 true 后如果要生效，则 validationQuery 参数必须正确设置
	testWhileIdle	false	指明连接是否被空闲连接回收器（如果有）进行检验。如果检测失败，则连接将被从连接池中去除。注意：设置为 true 后如果要生效，则 validationQuery 参数必须正确设置
	timeBetweenEvictionRunsMillis	-1	空闲连接回收器线程运行的周期，以毫秒为单位。如果设置为非正数，则不运行空闲连接回收器线程。注意：启用该参数时，validationQuery 参数必须正确设置
	numTestsPerEvictionRun	3	在每次空闲连接回收器线程（如果有）运行时检查的连接数量
缓存语句	minEvictableIdleTimeMillis	1000 × 60 × 30	连接在可被空闲连接回收器回收前已经在连接池中的空闲时间，以毫秒为单位
	poolPreparedStatements	false	开启连接池的 prepared statement 池功能。当设置为 true 时，所有 CallableStatement 和 PreparedStatement 都会被缓存起来
	maxOpenPreparedStatements	无限制	PreparedStatement 池能够同时分配的打开的 statements 的最大数量。如果设置为 0，则表示不限制
连接泄露回收	removeAbandoned	false	标记是否删除泄露的连接。如果 removeAbandoned 设置为 true，那么“存在泄露嫌疑”的连接可能被连接池主动清除。这个机制在 (getNumIdle() < 2) and (getNumActive() > getMaxActive() - 3) 条件满足时被触发。举例来说：当 maxActive=20，活动连接为 18，空闲连接为 1 时，可以触发 removeAbandoned 动作。但是活动连接只有在未被使用的时间超过 removeAbandoned Timeout 时才被回收，默认为 300 秒。如果应用需要一个进行长操作的连接，则需要考虑将 removeAbandonedTimeout 设置得更长一些，否则可能发生正常连接被强制清除的情况

续表

分类	属 性	默认值	说 明
连接 泄露 回收	removeAbandonedTimeout	300	泄露的连接可以被回收的超时值，单位为秒
	logAbandoned	false	标记当 Statement 或连接被泄露时是否打印程序的 stack traces 日志。被泄露的 Statements 和连接的日志添加在每个连接打开或者生成新的 Statement，因为需要生成 stack trace

如果采用 DBCP 的默认配置，由于 testOnBorrow 属性的默认值为 true，数据源在将连接将交给 DAO 前，会事先检测这个连接是否是好的，如果连接有问题（在数据库端被关闭），则会取一个其他的连接给 DAO，所以并不会会有“8 小时问题”。如果每次将连接交给 DAO 时都检测连接的有效性，那么在高并发的应用中将会带来性能问题，因为它会需要更多的数据库访问请求。

一种推荐的高效方式是：将 testOnBorrow 设置为 false，而将 testWhileIdle 设置为 true，再设置好 timeBetweenEvictionRunsMillis 值。这样，DBCP 将通过一个后台线程定时地对空闲连接进行检测，当发现无用的空闲连接（那些被数据库关闭的连接）时，就会将它们清除掉。只要将 timeBetweenEvictionRunsMillis 值设置为小于 8 小时，那些被 MySQL 关闭的空闲连接就可以被清除出去，从而避免了“8 小时问题”。

当然，MySQL 本身可以通过调整 interactive-timeout（以秒为单位）配置参数，更改空闲连接的过期时间。所以，在设置 timeBetweenEvictionRunsMillis 值时，必须首先获知 MySQL 空闲连接的最大过期时间。

2. C3P0 数据源

C3P0 是一个开放源码的 JDBC 数据源实现项目，实现了 JDBC3 和 JDBC2 扩展规范说明的 Connection 和 Statement 池。下面使用 C3P0 配置一个 Oracle 数据源，代码如下：

```
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource" destroy-  
method="close"  
  p: driverClass=" oracle.jdbc.driver.OracleDriver "  
  p: jdbcUrl=" jdbc:oracle:thin:@localhost:1521:ora9i "  
  p:use="admin"  
  p:password="1234"/>
```

ComboPooledDataSource 和 BasicDataSource 一样提供了一个用于关闭数据源的 close()方法，这样就可以保证 Spring 容器关闭时数据源能够被成功释放。

C3P0 拥有比 DBCP 更丰富的配置属性，通过这些属性，可以对数据源进行各种有效的控制。

- ❑ acquireIncrement: 当连接池中的连接用完时，C3P0 一次性创建新连接的数目。
- ❑ acquireRetryAttempts: 定义在从数据库获取新连接失败后重复尝试获取的次数，默认为 30。
- ❑ acquireRetryDelay: 尝试获取连接的间隔时间，单位为毫秒，默认为 1000。
- ❑ autoCommitOnClose: 连接关闭时默认将所有未提交的操作回滚。默认为 false。

- ❑ **automaticTestTable**: C3P0 将创建一张名为 Test 的空表, 并使用其自带的查询语句进行测试。如果定义了这个参数, 那么属性 **preferredTestQuery** 将被忽略。用户不能在这张 Test 表上进行任何操作, 它仅为 C3P0 测试所用。默认为 null。
- ❑ **breakAfterAcquireFailure**: 获取连接失败将会引起所有等待获取连接的线程抛出异常。但是数据源仍有效保留, 并在下次调用 **getConnection()** 方法时继续尝试获取连接。如果设为 true, 那么在尝试获取连接失败后, 该数据源将声明已断开并永久关闭。默认为 false。
- ❑ **checkoutTimeout**: 当连接池用完时, 客户端调用 **getConnection()** 方法后等待获取新连接的时间, 超时后将抛出 **SQLException**, 如果设为 0 则无限期等待。单位为毫秒, 默认为 0。
- ❑ **connectionTesterClassName**: 通过实现 **ConnectionTester** 或 **QueryConnectionTester** 的类来测试连接, 类名需要设置为全限定名。默认为 **com.mchange.v2.C3P0.impl.DefaultConnectionTester**。
- ❑ **idleConnectionTestPeriod**: 间隔多少秒检查所有连接池中的空闲连接。默认为 0, 表示不检查。
- ❑ **initialPoolSize**: 初始化时创建的连接数, 应在 **minPoolSize** 与 **maxPoolSize** 之间取值。默认为 3。
- ❑ **maxIdleTime**: 最大空闲时间, 超过空闲时间的连接将被丢弃。为 0 或负数则永不丢弃。默认为 0。
- ❑ **maxPoolSize**: 连接池中保留的最大连接数。默认为 15。
- ❑ **maxStatements**: JDBC 的标准参数, 用以控制数据源内加载的 **PreparedStatement** 数量。但由于预缓存的 **Statement** 属于单个 **Connection** 而不是整个连接池, 所以设置这个参数需要考虑多方面的因素, 如果 **maxStatements** 与 **maxStatementsPerConnection** 均为 0, 则缓存被关闭。默认为 0。
- ❑ **maxStatementsPerConnection**: 连接池内单个连接所拥有的最大缓存 **Statement** 数。默认为 0。
- ❑ **numHelperThreads**: C3P0 是异步操作的, 缓慢的 JDBC 操作通过帮助进程完成。扩展这些操作可以有效地提升性能, 通过多线程实现多个操作同时被执行。默认为 3。
- ❑ **preferredTestQuery**: 定义所有连接测试都执行的测试语句。在使用连接测试的情况下, 这个参数能显著提高测试速度。测试的表必须在初始数据源的时候就存在。默认为 null。
- ❑ **propertyCycle**: 用户修改系统配置参数执行前最多等待的秒数。默认为 300。
- ❑ **testConnectionOnCheckout**: 因性能消耗大, 请只在需要的时候使用它。如果设为 true, 那么在每个 **connection** 提交的时候都将校验其有效性。建议使用

idleConnection TestPeriod 或 automaticTestTable 等方法来提升连接测试的性能。默认为 false。

- ☐ testConnectionOnCheckin: 如果设为 true, 那么在取得连接的同时将校验连接的有效性。默认为 false。

对于连接有效性的检测, 请参照我们推荐的 DBCP 配置方式。

3. 使用属性文件

数据源的配置信息有可能经常需要改动, 同时可能被其他工程复用。此外, 用户名/密码等信息比较敏感, 可能需要使用特别的安全措施。所以一般将数据源的配置信息独立到一个属性文件中, 通过<context:property-placeholder>引入属性文件, 以\${xxx}的方式引用属性。示例代码如下:

```
<context:property-placeholder
    location="/WEB-INF/jdbc.properties" />
<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}" />
```

在 jdbc.properties 属性文件中定义属性值, 如下:

```
jdbc.driverClassName= com.mysql.jdbc.Driver
jdbc.url= jdbc:mysql://localhost:3309/sampledb
jdbc.username=root
jdbc.password=1234
```

这里, 属性文件的内容是以明文方式存放的。如果需要对属性文件的内容进行加密, 请参见 6.3 节。



提示

经常有开发者在\${xxx}前后不小心输入一些空格, 这些空格字符将和变量合并后作为属性的值。如 p: username="␣\${jdbc.username}␣"的属性配置项, 在前后都有空格, 被解析后, username 的值为 "␣1234␣", 这将造成最终的错误, 因此需要特别小心。

10.4.2 获取 JNDI 数据源

如果应用配置在高性能的应用服务器(如 WebLogic 或 WebSphere 等)上, 则可能更希望使用应用服务器本身提供的数据源。应用服务器的数据源使用 JNDI 开放调用者使用, Spring 为此专门提供了引用 JNDI 数据源的 JndiObjectFactoryBean 类。下面是一个简单的配置:

```
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean"
    p: jndiName="java:comp/env/jdbc/bbt" />
```

通过 `jndiName` 指定引用的 JNDI 数据源名称。

Spring 为获取 Java EE 资源提供了一个 `jee` 命名空间，通过 `jee` 命名空间，可以有效地简化 Java EE 资源的引用。下面是使用 `jee` 命名空间引用 JNDI 数据源的配置：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
        http://www.springframework.org/schema/jee
        http://www.springframework.org/schema/jee/spring-jee-4.0.xsd">
    <jee:jndi-lookup id="dataSource" jndi-name=" java:comp/env/jdbc/bbt"/>
</beans>
```

10.4.3 Spring 的数据源实现类

Spring 本身也提供了一个简单的数据源实现类 `DriverManagerDataSource`，它位于 `org.springframework.jdbc.datasource` 包中。这个类实现了 `javax.sql.DataSource` 接口，但它并没有提供池化连接的机制；每次调用 `getConnection()` 方法获取新连接时，只是简单地创建一个新的连接。因此，这个数据源类比较适合在单元测试或简单的独立应用中使用，因为它不需要额外的依赖类。

下面来看一下 `DriverManagerDataSource` 类的简单使用，如下：

```
DriverManagerDataSource ds = new DriverManagerDataSource ();
ds.setDriverClassName("com.mysql.jdbc.Driver");
ds.setUrl("jdbc:mysql://localhost:3309/sampledb");
ds.setUsername("root");
ds.setPassword("1234");
Connection actualCon = ds.getConnection();
```

当然，也可以通过配置的方式直接使用 `DriverManagerDataSource` 类。

10.5 小结

Spring 支持目前大多数常用的数据持久化技术。Spring 定义了一套面向 DAO 层的异常体系，并为各种支持的持久化技术提供了异常转换器。这样，在设计 DAO 接口时，就可以抛开具体的实现技术，定义统一的接口。

不管采用何种持久化技术，访问数据的流程是相对固定的。Spring 将数据访问流程划分为固定和变化两部分，并以模板的方式定义好流程，用回调接口将变化的部分开放出来，留给开发者自行定义。这样，仅需提供业务相关的逻辑就可以完成整体的数据访问。Spring 为了进一步简化持久化模板类的使用，为各种持久化技术提供了便捷的支持

类。支持类不但包含数据访问模板，还包含数据源或会话等内容。通过扩展支持类定义自己的数据访问类是最简单的数据访问方式。

不管采用何种持久化技术，都需要定义数据源。在实际部署时，可能会直接采用应用服务器本身提供的数据源，这时可以通过 `JndiObjectFactoryBean` 或 `jee` 命名空间引用 JNDI 数据源。

第 11 章

Spring 的事务管理

在使用 Spring 开发应用时, Spring 的事务管理可能是被使用最多、应用最广的功能。对于大多数应用者来说, Spring 事务管理所带来的好处是实实在在的。学习 Spring 而忽略了事务管理, 就像到了黄山不登天都峰、讲隋唐演义不讲李元霸一样, 最精彩的部分被忽略了。Spring 不但提供了和底层事务源无关的事务抽象, 还提供了声明性事务的功能, 可以让程序从事务代码中解放出来。而这正是 EJB 自鸣得意的地方, 但现在我们有了更好、更便捷的替代方案。

本章主要内容:

- ◆ 事务属性值的实际意义
- ◆ ThreadLocal 的工作机制
- ◆ Spring 事务管理的体系结构
- ◆ 基于 XML 的事务配置
- ◆ 基于注解的事务配置

本章亮点:

- ◆ 介绍数据库事务的基础知识, 它们是掌握事务配置的基础
- ◆ 深入学习 ThreadLocal 知识, 进而揭示 Spring 事务同步管理器的工作原理
- ◆ Spring 注解 AspectJ 织入的过程

11.1 数据库事务基础知识

Spring 虽然提供了灵活方便的事务管理功能, 但这些功能都是基于底层数据库本身的事务处理机制工作的。要深入了解 Spring 的事务管理和配置, 有必要先对数据库事务的基础知识进行学习。

11.1.1 何为数据库事务

“一荣俱荣，一损俱损”这句话很能体现事务的思想，很多复杂的事物要分步进行，但它们组成了一个整体，要么整体生效，要么整体失效。这种思想反映到数据库上，就是多条 SQL 语句，要么所有执行成功，要么所有执行失败。

数据库事务有严格的定义，它必须同时满足 4 个特性：原子性（Atomic）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability），简称为 ACID。下面是对每个特性的说明。

- ❑ 原子性：表示组成一个事务的多个数据库操作是一个不可分割的原子单元，只有所有的操作执行成功，整个事务才提交。事务中的任何一个数据库操作失败，已经执行的任何操作都必须撤销，让数据库返回到初始状态。
- ❑ 一致性：事务操作成功后，数据库所处的状态和它的业务规则是一致的，即数据不会被破坏。如从 A 账户转账 100 元到 B 账户，不管操作成功与否，A 账户和 B 账户的存款总额是不变的。
- ❑ 隔离性：在并发数据操作时，不同的事务拥有各自的数据空间，他们的操作不会对对方产生干扰。准确地说，并非要求做到完全无干扰。数据库规定了多种事务隔离级别，不同的隔离级别对应不同的干扰程度，隔离级别越高，数据一致性越好，但并发性越弱。
- ❑ 持久性：一旦事务提交成功后，事务中所有的数据操作都必须被持久化到数据库中。即使在提交事务后，数据库马上崩溃，在数据库重启时，也必须保证能够通过某种机制恢复数据。

在这些事务特性中，数据“一致性”是最终目标，其他特性都是为达到这个目标而采取的措施、要求或手段。

数据库管理系统一般采用重执行日志来保证原子性、一致性和持久性。重执行日志记录了数据库变化的每一个动作，数据库在一个事务中执行一部分操作后发生错误退出，数据库即可根据重执行日志撤销已经执行的操作。此外，对于已经提交的事务，即使数据库崩溃，在重启数据库时也能够根据日志对尚未持久化的数据进行相应的重执行操作。

和 Java 程序采用对象锁机制进行线程同步类似，数据库管理系统采用数据库锁机制保证事务的隔离性。当多个事务试图对相同的数据进行操作时，只有持有锁的事务才能操作数据，直到前一个事务完成后，后面的事务才有机会对数据进行操作。Oracle 数据库还使用了数据版本的机制，在回滚段为数据的每个变化都保存一个版本，使数据的更改不影响数据的读取。

11.1.2 数据并发的问题

一个数据库可能拥有多个访问客户端，这些客户端都可用并发的方式访问数据库。数据库中的相同数据可能同时被多个事务访问，如果没有采取必要的隔离措施，就会导致各种并发问题，破坏数据的完整性。这些问题可以归结为 5 类，包括 3 类数据读问题（脏读、不可重复读和幻象读）及 2 类数据更新问题（第一类丢失更新和第二类丢失更新）。下面分别通过实例讲解引发问题的场景。

1. 脏读（dirty read）

A 事务读取 B 事务尚未提交的更改数据，并在这个数据的基础上进行操作。如果恰巧 B 事务回滚，那么 A 事务读到的数据根本是不被承认的。来看取款事务和转账事务并发时引发的脏读场景。

时 间	转账事务 A	取款事务 B
T1		开始事务
T2	开始事务	
T3		查询账户余额为 1000 元
T4		取出 500 元，把余额改为 500 元
T5	查询账户余额为 500 元（脏读）	
T6		撤销事务，余额恢复为 1000 元
T7	汇入 100 元，把余额改为 600 元	
T8	提交事务	

在这个场景中，B 希望取款 500 元，而后又撤销了动作，而 A 往相同的账户中转账 100 元，就因为 A 事务读取了 B 事务尚未提交的数据，因而造成账户白白丢失了 500 元。在 Oracle 数据库中，不会发生脏读的情况。



一个有结巴的人在饮料店柜台前转悠，老板很热情地迎上来说：“喝一瓶？”结巴连忙说：“我…喝…喝…”老板麻利地打开易拉罐递给结巴，结巴终于憋出了他的那句话：“我…喝…喝…喝不起啊！”

2. 不可重复读（unrepeatable read）

不可重复读是指 A 事务读取了 B 事务已经提交的更改数据。假设 A 在取款事务的过程中，B 往该账户转账 100 元，A 两次读取账户的余额发生不一致。

时 间	取款事务 A	转账事务 B
T1		开始事务
T2	开始事务	
T3		查询账户余额为 1000 元

续表

时 间	取款事务 A	转账事务 B
T4	查询账户余额为 1000 元	
T5		取出 100 元，把余额改为 900 元
T6		提交事务
T7	查询账户余额为 900 元（和 T4 读取的不一致）	

在同一事务中，T4 时间点和 T7 时间点读取的账户存款余额不一致。

3. 幻象读（phantom read）

A 事务读取 B 事务提交的新增数据，这时 A 事务将出现幻象读的问题。幻象读一般发生在计算统计数据的事务中。举个例子，假设银行系统在同一事务中两次统计存款账户的总金额，在两次统计过程中，刚好新增了一个存款账户，并存入 100 元，这时，两次统计的总金额将不一致。

时 间	统计金额事务 A	转账事务 B
T1		开始事务
T2	开始事务	
T3	统计总存款数为 10000 元	
T4		新增一个存款账户，存款为 100 元
T5		提交事务
T6	再次统计总存款数为 10100 元（幻象读）	

如果新增数据刚好满足事务的查询条件，那么这个新数据就进入了事务的视野，因而产生了两次统计结果不一致的情况。

幻象读和不可重复读是两个容易混淆的概念，前者是指读到了其他已经提交事务的新增数据，而后者是指读到了已经提交事务的更改数据（更改或删除）。为了避免这两种情况，采取的对策是不同的：防止读到更改数据，只需对操作的数据添加行级锁，阻止操作中的数据发生变化；而防止读到新增数据，则往往需要添加表级锁——将整张表锁定，防止新增数据（Oracle 使用多版本数据的方式实现）。

4. 第一类丢失更新

A 事务撤销时，把已经提交的 B 事务的更新数据覆盖了。这种错误可能造成很严重的问题，通过下面的账户取款转账就可以看出来。

时 间	取款事务 A	转账事务 B
T1	开始事务	
T2		开始事务
T3	查询账户余额为 1000 元	
T4		查询账户余额为 1000 元
T5		汇入 100 元，把余额改为 1100 元
T6		提交事务
T7	取出 100 元，把余额改为 900 元	

续表

时 间	取款事务 A	转账事务 B
T8	撤销事务	
T9	余额恢复为 1000 元（丢失更新）	

A 事务在撤销时，“不小心”将 B 事务已经转入账户的金额给抹去了。

5. 第二类丢失更新

A 事务覆盖 B 事务已经提交的数据，造成 B 事务所做操作丢失。

时 间	转账事务 A	取款事务 B
T1		开始事务
T2	开始事务	
T3		查询账户余额为 1000 元
T4	查询账户余额为 1000 元	
T5		取出 100 元，把余额改为 900 元
T6		提交事务
T7	汇入 100 元	
T8	提交事务	
T9	把余额改为 1100 元（丢失更新）	

在上面的例子中，由于支票转账事务覆盖了取款事务对存款余额所做的更新，导致银行最后损失了 100 元。相反，如果转账事务先提交，那么用户账户将损失 100 元。

11.1.3 数据库锁机制

数据并发会引发很多问题，在一些场合下有些问题是允许的，但在另一些场合下可能是致命的。数据库通过锁机制解决并发访问的问题，虽然不同的数据库在实现细节上存在差别，但原理基本上是一样的。

按锁定的对象的不同，一般可以分为表锁定和行锁定。前者对整张表进行锁定，而后者对表中的特定行进行锁定。从并发事务锁定的关系上看，可以分为共享锁定和独占锁定。共享锁定会防止独占锁定，但允许其他的共享锁定。而独占锁定既防止其他的独占锁定，也防止其他的共享锁定。为了更改数据，数据库必须在进行更改的行上施加行独占锁定，INSERT、UPDATE、DELETE 和 SELECT FOR UPDATE 语句都会隐式采用必要的行锁定。下面介绍一下 Oracle 数据库常用的 5 种锁定。

- 行共享锁定：一般通过 SELECT FOR UPDATE 语句隐式获得行共享锁定，在 Oracle 中用户也可以通过 LOCK TABLE IN ROW SHARE MODE 语句显式获得行共享锁定。行共享锁定并不防止对数据行进行更改操作，但是可以防止其他会话获取独占性数据表锁定。允许进行多个并发的行共享和行独占锁定，还允许进行数据表的共享或者采用共享行独占锁定。

- ❑ 行独占锁定：通过一条 INSERT、UPDATE 或 DELETE 语句隐式获取，或者通过一条 LOCK TABLE IN ROW EXCLUSIVE MODE 语句显式获取。这种锁定可以防止其他会话获取一个共享锁定、共享行独占锁定或独占锁定。
- ❑ 表共享锁定：通过 LOCK TABLE IN SHARE MODE 语句显式获得。这种锁定可以防止其他会话获取行独占锁定（INSERT、UPDATE 或 DELETE），或者防止其他表共享行独占锁定或表独占锁定，但它允许在表中拥有多个行共享和表共享锁定。该锁定可以让会话具有对表事务级一致性访问，因为其他会话在用户提交或者回滚该事务并释放对该表的锁定之前不能更改这张被锁定的表。
- ❑ 表共享行独占锁定：通过 LOCK TABLE IN SHARE ROW EXCLUSIVE MODE 语句显式获得。这种锁定可以防止其他会话获取一个表共享、行独占或者表独占锁定，但允许其他行共享锁定。这种锁定类似于表共享锁定，只是一次只能对一张表放置一个表共享行独占锁定。如果 A 会话拥有该锁定，则 B 会话可以执行 SELECT FOR UPDATE 操作；但如果 B 会话试图更新选择的行，则需要等待。
- ❑ 表独占锁定：通过 LOCK TABLE IN EXCLUSIVE MODE 语句显式获得。这种锁定可以防止其他会话对该表的任何其他锁定。

11.1.4 事务隔离级别

尽管数据库为用户提供了锁的 DML 操作方式，但直接使用锁管理是非常麻烦的，因此数据库为用户提供了自动锁机制。只要用户指定会话的事务隔离级别，数据库就会分析事务中的 SQL 语句，然后自动为事务操作的数据资源添加适合的锁。此外，数据库还会维护这些锁，当一个资源上的锁数目太多时，自动进行锁升级以提高系统的运行性能，而这一过程对用户来说完全是透明的。

ANSI/ISO SQL 92 标准定义了 4 个等级的事务隔离级别，在相同的数据环境下，使用相同的输入，执行相同的工作，根据不同的隔离级别，可能导致不同的结果。不同的事务隔离级别能够解决的数据并发问题的能力是不同的，如表 11-1 所示。

表 11-1 事务隔离级别对并发问题的解决情况

隔离级别	脏 读	不可重复读	幻 象 读	第一类丢失更新	第二类丢失更新
READ UNCOMMITTED	允许	允许	允许	不允许	允许
READ COMMITTED	不允许	允许	允许	不允许	允许
REPEATABLE READ	不允许	不允许	允许	不允许	不允许
SERIALIZABLE	不允许	不允许	不允许	不允许	不允许

事务的隔离级别和数据库并发性是对立的。一般来说，使用 READ UNCOMMITTED 隔离级别的数据库拥有最高的并发性和吞吐量，而使用 SERIALIZABLE 隔离级别的数据库并发性最低。

SQL 92 定义 READ UNCOMMITTED 主要是为了提供非阻塞读的能力。Oracle 虽然也支持 READ UNCOMMITTED, 但它不支持脏读; 因为 Oracle 使用多版本机制彻底解决了在非阻塞读时读到脏数据的问题并保证读的一致性, 所以, Oracle 的 READ COMMITTED 隔离级别就已经满足了 SQL 92 标准的 REPEATABLE READ 隔离级别。

SQL 92 推荐使用 REPEATABLE READ 以保证数据的读一致性, 不过用户可以根据应用的需要选择适合的隔离等级。

11.1.5 JDBC 对事务的支持

并不是所有的数据库都支持事务, 即使支持事务的数据库也并非支持所有的事务隔离级别。用户可以通过 `Connection#getMetaData()` 方法获取 `DatabaseMetaData` 对象, 并通过该对象的 `supportsTransactions()`、`supportsTransactionIsolationLevel(int level)` 方法查看底层数据库的事务支持情况。

`Connection` 默认情况下是自动提交的, 即每条执行的 SQL 语句都对应一个事务。为了将多条 SQL 语句当成一个事务执行, 必须先通过 `Connection#setAutoCommit(false)` 阻止 `Connection` 自动提交, 并通过 `Connection#setTransactionIsolation()` 设置事务的隔离级别。`Connection` 中定义了对应 SQL 92 标准 4 个事务隔离级别的常量。通过 `Connection#commit()` 提交事务, 通过 `Connection#rollback()` 回滚事务。下面是典型的 JDBC 事务数据操作的代码, 如代码清单 11-1 所示。

代码清单 11-1 JDBC 事务代码

```
Connection conn ;
try{
    conn = DriverManager.getConnection(); ① ← 获取数据连接
    conn.setAutoCommit(false); ② ← 关闭自动提交机制
    conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE); ③ ← 设置事务隔离级别
    Statement stmt = conn.createStatement();

    int rows = stmt.executeUpdate( "INSERT INTO t_topic VALUES(1,'tom') " );
    rows = stmt.executeUpdate( "UPDATE t_user set topic_nums = topic_nums +1 "+
                                "WHERE user_id = 1");

    conn.commit(); ④ ← 提交事务
}catch(Exception e){
    ...
    conn.rollback(); ⑤ ← 回滚事务
}finally{
    ...
}
```

在 JDBC 2.0 中, 事务最终只能有两个操作: 提交和回滚。但是, 有些应用可能需要对事务进行更多的控制, 而不是简单地提交或回滚。JDBC 3.0 (Java 1.4 及以后的版本) 引入了一个全新的保存点特性, `Savepoint` 接口允许用户将事务分割为多个阶段, 用户可以指定回滚到事务的特定保存点, 而并非像 JDBC 2.0 一样只能回滚到开始事务的点, 如图 11-1 所示。

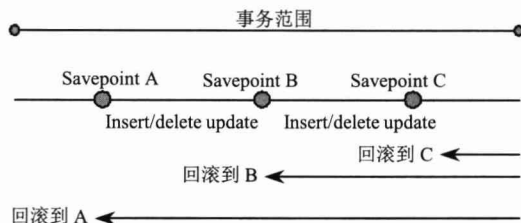


图 11-1 带 Savepoint 的事务

下面的代码使用了保存点功能，在发生特定问题时，回滚到指定的保存点，而非回滚整个事务，如代码清单 11-2 所示。

代码清单 11-2 使用保存点的事务代码

```

...
Statement stmt = conn.createStatement();
int rows = stmt.executeUpdate( "INSERT INTO t_topic VALUES(1,'tom')");

Savepoint svpt = conn.setSavepoint("savePoint1"); ① ← 设置一个保存点
rows = stmt.executeUpdate( "UPDATE t_user set topic_nums = topic_nums +1 "+
                           "WHERE user_id = 1");
...
conn.rollback(svpt); ② ← 回滚到①处的 savePoint1, ①之前的 SQL 操作, 在整个事务提交后依然提交, 但①到②之间的 SQL 操作被撤销了
...
conn.commit(); ③ ← 提交事务

```

并非所有数据库都支持保存点功能，用户可以通过 `DatabaseMetaData#supportsSavepoints()` 方法查看是否支持。

11.2 ThreadLocal 基础知识

在第 10 章中我们知道，Spring 通过各种模板类降低了开发者使用各种数据持久化技术的难度。这些模板类都是线程安全的，也就是说，多个 DAO 可以复用同一个模板实例而不会发生冲突。使用模板类访问底层数据，根据持久化技术的不同，模板类需要绑定数据连接或会话的资源。但这些资源本身是非线程安全的，也就是说它们不能在同一时刻被多个线程共享。虽然模板类通过资源池获取数据连接或会话，但资源池本身解决的是数据连接或会话的缓存问题，并非数据连接或会话的线程安全问题。

按照传统经验，如果某个对象是非线程安全的，在多线程环境下，对对象的访问必须采用 `synchronized` 进行线程同步。但模板类并未采用线程同步机制，因为线程同步会降低并发性，影响系统性能。此外，通过代码同步解决线程安全的挑战性很大，可能会增加几倍的实现难度。那么，模板类究竟仰仗何种“魔法神功”，可以在无须线程同步的情况下就化解线程安全的难题呢？答案就是 `ThreadLocal`！

`ThreadLocal` 在 Spring 中发挥着重要的作用，在管理 request 作用域的 Bean、事务管

理、任务调度、AOP 等模块中都出现了它的身影。要想了解 Spring 事务管理的底层技术，ThreadLocal 是必须攻克的“山头堡垒”。

11.2.1 ThreadLocal 是什么

早在 Java 1.2 版本中就提供了 `java.lang.ThreadLocal`。ThreadLocal 为解决多线程程序的并发问题提供了一种新的思路，使用这个工具类可以很简洁地编写出优美的多线程程序。线程局部变量并不是 Java 的新发明，很多语言（如 IBM XL、FORTRAN）在语法层面就提供了线程局部变量。在 Java 中没有提供语言级支持，而是以一种变通的方法，通过 ThreadLocal 的类提供支持。所以，在 Java 中编写线程局部变量的代码相对来说要笨拙一些，这也是为什么线程局部变量没有在 Java 开发者中得到很好普及的原因。

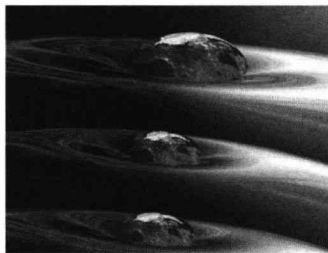
ThreadLocal，顾名思义，它不是一个线程，而是保存线程本地化对象的容器。当运行于多线程环境的某个对象使用 ThreadLocal 维护变量时，ThreadLocal 为每个使用该变量的线程分配一个独立的变量副本。所以每个线程都可以独立地改变自己的副本，而不会影响其他线程所对应的副本。从线程的角度看，这个变量就像线程专有的本地变量，这也是类名中“Local”所要表达的意思。

InheritableThreadLocal 继承于 ThreadLocal，它自动为子线程复制一份从父线程那里继承而来的本地变量：在创建子线程时，子线程会接收所有可继承的线程本地变量的初始值。当必须将本地线程变量自动传送给所有创建的子线程时，应尽可能地使用 InheritableThreadLocal，而非 ThreadLocal。



轻松一刻

ThreadLocal 很像平行宇宙（Parallel Universes）的概念。平行宇宙来自量子力学，是指多元宇宙中所包含的各个宇宙，一个宇宙从某个宇宙中分离出来，与原宇宙平行存在着既相似又不同的其他宇宙。它们可能处于同一空间体系，但时间体系不同，就好像同在一条铁路线上疾驰的先后两列火车。2011 年，邓肯·琼斯执导的《源代码》就是一部关于平行宇宙的科幻影片。



11.2.2 ThreadLocal 的接口方法

ThreadLocal 类接口很简单，只有 4 个方法，先来了解一下。

- ☐ `void set(Object value)`: 设置当前线程的线程局部变量的值。
- ☐ `public Object get()`: 返回当前线程所对应的线程局部变量。

- ❑ `public void remove()`: 将当前线程局部变量的值删除, 目的是为了减少内存的占用。该方法是 Java 5.0 新增的。需要指出的是, 当线程结束后, 对应该线程的局部变量将自动被垃圾回收, 所以显式调用该方法清除线程的局部变量并不是必需的操作, 但它可以加快内存回收的速度。
- ❑ `protected Object initialValue()`: 返回该线程局部变量的初始值。该方法是一个 `protected` 的方法, 显然是为了让子类覆盖而设计的。这个方法是一个延迟调用方法, 在线程第一次调用 `get()` 或 `set(Object)` 时才执行, 并且仅执行一次。
`ThreadLocal` 中的默认实现直接返回一个 `null`。

值得一提的是, 在 Java 5.0 中, `ThreadLocal` 已经支持泛型, 该类的类名已经变为 `ThreadLocal<T>`。API 方法也相应进行了调整, 新版本的 API 方法分别是 `void set(T value)`、`T get()` 及 `T initialValue()`。

`ThreadLocal` 是如何做到为每个线程维护一份独立的变量副本呢? 其实实现的思路很简单: 在 `ThreadLocal` 类中有一个 `Map`, 用于存储每个线程的变量副本, `Map` 中元素的键为线程对象, 值为对应线程的变量副本。下面提供一个简单的实现版本, 如代码清单 11-3 所示。

代码清单 11-3 SimpleThreadLocal

```
public class SimpleThreadLocal {
    private Map valueMap = Collections.synchronizedMap(new HashMap());
    public void set(Object newValue) {
        valueMap.put(Thread.currentThread(), newValue); ①
    }
    public Object get() {
        Thread currentThread = Thread.currentThread();
        Object o = valueMap.get(currentThread); ②
        if (o == null && !valueMap.containsKey(currentThread)) { ③
            o = initialValue();
            valueMap.put(currentThread, o);
        }
        return o;
    }
    public void remove() {
        valueMap.remove(Thread.currentThread());
    }
    public Object initialValue() {
        return null;
    }
}
```

键为线程对象, 值为
本线程的变量副本

返回本线程对应的变量

如果在 Map 中不存在,
则放到 Map 中保存起来

虽然代码清单 11-3 中 `ThreadLocal` 的实现版本显得比较幼稚, 但它和 JDK 所提供的 `ThreadLocal` 类在实现思路上是非常相近的。

11.2.3 一个 ThreadLocal 实例

下面通过一个具体的实例了解一下 `ThreadLocal` 的具体使用方法, 如代码清单 11-4 所示。

代码清单 11-4 SequenceNumber

```

package com.smart.basic;

public class SequenceNumber {
    // 通过匿名内部类覆盖 ThreadLocal
    // 的 initialValue() 方法, 指定初始值
    private static ThreadLocal<Integer> seqNum = new ThreadLocal<Integer>() { ①
        public Integer initialValue() {
            return 0;
        }
    };

    public int getNextNum() { ②
        seqNum.set(seqNum.get()+1);
        return seqNum.get();
    }

    public static void main(String[] args)
    {
        SequenceNumber sn = new SequenceNumber();

        TestClient t1 = new TestClient(sn); ③
        TestClient t2 = new TestClient(sn);
        TestClient t3 = new TestClient(sn);
        t1.start();
        t2.start();
        t3.start();
    }

    private static class TestClient extends Thread
    {
        private SequenceNumber sn;
        public TestClient(SequenceNumber sn) {
            this.sn = sn;
        }
        public void run()
        {
            // 每个线程打出 3 个序列值
            for (int i = 0; i < 3; i++) { ④
                System.out.println("thread["+Thread.currentThread().getName()+
                    "] sn["+sn.getNextNum()+"]");
            }
        }
    }
}

```

① 通过匿名内部类覆盖 ThreadLocal 的 initialValue() 方法, 指定初始值

② 获取下一个序列值

③ 3 个线程共享 sn, 各自产生序列号

④ 每个线程打出 3 个序列值

可通过匿名内部类的方式定义 ThreadLocal 的子类, 提供初始的变量值, 如①处所示。TestClient 线程产生一组序列号, 在③处, 生成 3 个 TestClient, 它们共享同一个 SequenceNumber 实例。运行以上代码, 在控制台上输出以下信息:

```

thread[Thread-2] sn[1]
thread[Thread-0] sn[1]
thread[Thread-1] sn[1]
thread[Thread-2] sn[2]
thread[Thread-0] sn[2]
thread[Thread-1] sn[2]
thread[Thread-2] sn[3]

```

```
thread[Thread-0] sn[3]  
thread[Thread-1] sn[3]
```

考查输出的结果信息，发现每个线程所产生的序号虽然都共享同一个 `SequenceNumber` 实例，但它们并没有相互干扰，而是各自产生独立的序列号，这是因为通过 `ThreadLocal` 为每个线程提供了单独的副本。

11.2.4 与 Thread 同步机制的比较

`ThreadLocal` 和线程同步机制都是为了解决多线程中相同变量的访问冲突问题。那么，`ThreadLocal` 和线程同步机制相比有什么优势呢？

在同步机制中，通过对象的锁机制保证同一时间只有一个线程访问变量。这时该变量是多个线程共享的，使用同步机制要求程序缜密地分析什么时候对变量进行读/写、什么时候需要锁定某个对象、什么时候释放对象锁等繁杂的问题，程序设计和编写难度相对较大。

而 `ThreadLocal` 从另一个角度来解决多线程的并发访问。`ThreadLocal` 为每个线程提供了一个独立的变量副本，从而隔离了多个线程对访问数据的冲突。因为每个线程都拥有自己的变量副本，因而也就没有必要对该变量进行同步。`ThreadLocal` 提供了线程安全的对象封装，在编写多线程代码时，可以把不安全的变量封装进 `ThreadLocal`。

由于 `ThreadLocal` 中可以持有任何类型的对象，低版本 JDK 所提供的 `get()` 方法返回的是 `Object` 对象，需要强制类型转换。但 Java 5.0 通过泛型很好地解决了这个问题，在一定程度上简化了 `ThreadLocal` 的使用，代码清单 11-2 就使用了 Java 5.0 新的 `ThreadLocal<T>` 版本。

概括而言，对于多线程资源共享的问题，同步机制采用了“以时间换空间”的方式：访问串行化，对象共享化；而 `ThreadLocal` 采用了“以空间换时间”的方式：访问并行化，对象独享化。前者仅提供一份变量，让不同的线程排队访问；而后者为每个线程都提供了一份变量，因此可以同时访问而互不影响。

11.2.5 Spring 使用 ThreadLocal 解决线程安全问题

我们知道，在一般情况下，只有无状态的 Bean 才可以在多线程环境下共享。在 Spring 中，绝大部分 Bean 都可以声明为 `singleton` 作用域。正是因为 Spring 对一些 Bean（如 `RequestContextHolder`、`TransactionSynchronizationManager`、`LocaleContextHolder` 等）中非线程安全的“状态性对象”采用 `ThreadLocal` 进行封装，让它们也成为线程安全的“状态性对象”，因此，有状态的 Bean 就能够以 `singleton` 的方式在多线程中正常工作。

一般的 Web 应用划分为展现层、服务层和持久层 3 个层次，在不同的层中编写对应的逻辑，下层通过接口向上层开放功能调用。在一般情况下，从接收请求到返回响应所经过的所有程序调用都同属于一个线程，如图 11-2 所示。

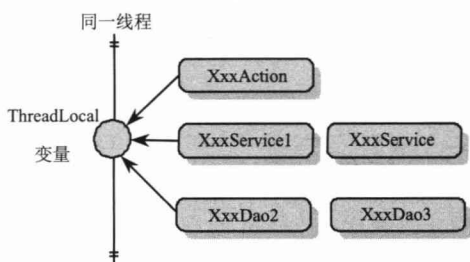


图 11-2 同一线程贯通三层

这样用户就可以根据需要，将一些非线程安全的变量以 ThreadLocal 存放，在同一次请求响应的调用线程中，所有对象所访问的同一 ThreadLocal 变量都是当前线程所绑定的。

下面的实例能够体现 Spring 对有状态 Bean 的改造思路，如代码清单 11-5 所示。

代码清单 11-5 TopicDao: 非线程安全

```
public class TopicDao {

    //①一个非线程安全变量
    private Connection conn;

    public void addTopic(){

        //②引用非线程安全变量
        Statement stat = conn.createStatement();
        ...
    }
}
```

由于①处的 conn 是非线程安全的成员变量，因此 addTopic() 方法也是非线程安全的，所以必须在使用时创建一个新的 TopicDao 实例（非 singleton）。下面使用 ThreadLocal 对 conn 这个非线程安全的“状态”进行改造，如代码清单 11-6 所示。

代码清单 11-6 TopicDao: 线程安全

```
import java.sql.Connection;
import java.sql.Statement;
public class TopicDao {

    //①使用ThreadLocal保存Connection变量
    private static ThreadLocal<Connection> connThreadLocal = new
    ThreadLocal<Connection>();

    public static Connection getConnection(){

        //②如果connThreadLocal没有本线程对应的Connection，则创建一个新的Connection，
        //并将其保存到线程本地变量中
        if (connThreadLocal.get() == null) {
            Connection conn = ConnectionManager.getConnection();
            connThreadLocal.set(conn);
            return conn;
        }else{
```

```

        //③直接返回线程本地变量
        return connThreadLocal.get();
    }

    public void addTopic() {

        //④从ThreadLocal中获取线程对应的Connection
        Statement stat = getConnection().createStatement();
    }
}

```

不同的线程在使用 TopicDao 时，先判断 `connThreadLocal.get()` 是否为 `null`，如果为 `null`，则说明当前线程还没有对应的 `Connection` 对象，这时创建一个 `Connection` 对象并添加到本地线程变量中；如果不为 `null`，则说明当前线程已经拥有了 `Connection` 对象，直接使用就可以了。这样就保证了不同的线程使用自己独立的 `Connection`，而不会使用其他线程的 `Connection`。因此，这个 TopicDao 就可以做成 singleton 的 Bean 了。

当然，这个例子本身很粗糙，将 `Connection` 的 `ThreadLocal` 直接放在 DAO 中只能做到本 DAO 的多个方法共享 `Connection` 时不发生线程安全问题，但无法和其他 DAO 共用同一个 `Connection`。要做到同一事务多 DAO 共享同一个 `Connection`，必须在一个共同的外部类使用 `ThreadLocal` 保存 `Connection`。但这个实例基本上说明了 Spring 对所有状态类线程安全化的解决思路。在本章后面的内容中，将详细说明 Spring 如何通过 `ThreadLocal` 解决事务管理的问题。

11.3 Spring 对事务管理的支持

Spring 为事务管理提供了一致的编程模板，在高层次建立了统一的事务抽象。也就是说，不管是选择 Spring JDBC、Hibernate、JPA 还是选择 MyBatis，Spring 都可以让用户用统一的编程模型进行事务管理。

像 Spring DAO 为不同的持久化实现提供了模板类一样，Spring 事务管理继承了这一风格，也提供了事务模板类 `TransactionTemplate`。通过 `TransactionTemplate` 并配合使用事务回调 `TransactionCallback` 指定具体的持久化操作，就可以通过编程方式实现事务管理，而无须关注资源获取、复用、释放、事务同步和异常处理等操作。

Spring 事务管理的亮点在于声明式事务管理。Spring 允许通过声明方式，在 IoC 配置中指定事务的边界和事务属性，Spring 自动在指定的事务边界上应用事务属性。声明式事务是 EJB 烜赫一时的技术，Spring 让这种技术平民化，甚至可以说，Spring 的声明事务比 EJB 的更为强大。

EJB 事务建立在 JTA 的基础上，而 JTA 又必须通过 JNDI 获取。这意味着不管用户的应用是跨数据源的应用，还是单数据源的应用，EJB 都要求使用全局事务的方式加以处理，即基于 EJB 的应用无法脱离应用服务器所提供的容器环境。这种不加区分一概而论的做法无异于杀鸡杀牛都用一把刀。

Spring 深刻地认识到,大部分应用都是基于单数据源的,只有为数不多的应用需要使用多数据源的 JTA 事务。因此,在单数据源的情况下, Spring 直接使用底层的数据源管理事务。只有在面对多数据源的应用时, Spring 才寻求 Java EE 应用服务器的支持,通过引用应用服务器中的 JNDI 资源完成 JTA 事务。Spring 让人印象深刻的地方在于不管用户使用何种持久化技术,也不管用户是否使用了 JTA 事务,都可以采用相同的事务管理模型。这种统一的处理方式所带来的好处是不可估量的,用户完全可以抛开事务管理的问题编写程序,并在 Spring 中通过配置完成事务的管理工作。

11.3.1 事务管理关键抽象

在 Spring 事务管理 SPI (Service Provider Interface) 的抽象层主要包括 3 个接口,分别是 PlatformTransactionManager、TransactionDefinition 和 TransactionStatus,它们位于 org.springframework.transaction 包中。通过图 11-3 可以描述这三者的关系。

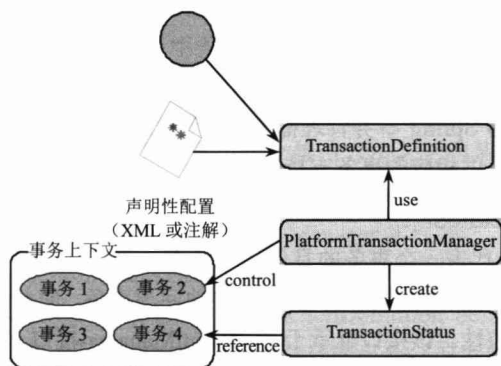


图 11-3 Spring 事务管理 SPI 抽象

TransactionDefinition 用于描述事务的隔离级别、超时时间、是否为只读事务和事务传播规则等控制事务具体行为的事务属性,这些事务属性可以通过 XML 配置或注解描述提供,也可以通过手工编程的方式设置。

PlatformTransactionManager 根据 TransactionDefinition 提供的事务属性配置信息创建事务,并用 TransactionStatus 描述这个激活事务的状态。下面分别来了解这些 SPI 接口内部的组成。

1. TransactionDefinition

TransactionDefinition 定义了 Spring 兼容的事务属性,这些属性对事务管理控制的若干方面进行配置。

- **事务隔离**: 当前事务和其他事务的隔离程度。在 TransactionDefinition 接口中,定义了和 java.sql.Connection 接口中同名的 4 个隔离级别: ISOLATION_READ_UNCOMMITTED、ISOLATION_READ_COMMITTED、ISOLATION_REPEATABLE

READ 和 ISOLATION_SERIALIZABLE（实际上它直接使用 Connection 的同名常量进行赋值），这些常量分别对应于 11.1.4 节所描述的隔离级别。此外，TransactionDefinition 还定义了一个默认的隔离级别——ISOLATION_DEFAULT，它表示使用底层数据库的默认隔离级别。

- ❑ **事务传播：**通常在一个事务中执行的所有代码都会运行于同一事务上下文中。但是 Spring 也提供了几个可选的事务传播类型，例如，简单地参与到现有的事务中，或者挂起当前的事务，创建一个新的事务。在 Spring 事务管理中，传播行为是一个重要的概念，将单独在 11.3.4 节详细讲述。Spring 提供了 EJB CMT 所支持的事务传播类型。
- ❑ **事务超时：**事务在超时前能运行多久，超过时间后，事务被回滚。有些事务管理器不支持事务过期的功能，这时，如果设置 TIMEOUT_DEFAULT 之外的其他值，则将抛出异常。
- ❑ **只读状态：**只读事务不修改任何数据，资源事务管理者可以针对可读事务应用一些优化措施，提高运行性能。只读事务在某些情况下（如使用 Hibernate 时）是一种非常有用的优化，试图在只读事务中更改数据将引发异常。

Spring 允许通过 XML 或注解元数据的方式为一个有事务要求的服务类方法配置事务属性，这些信息作为 Spring 事务管理框架的“输入”，Spring 将自动按事务属性信息的指示，为目标方法提供相应的事务支持。

2. TransactionStatus

TransactionStatus 代表一个事务的具体运行状态。事务管理器可以通过该接口获取事务运行期的状态信息，也可以通过该接口间接地回滚事务，它相比于在抛出异常时回滚事务的方式更具可控性。该接口继承于 SavepointManager 接口，SavepointManager 接口基于 JDBC 3.0 保存点的分段事务控制能力提供了嵌套事务的机制。

SavepointManager 接口拥有以下几个方法。

- ❑ **Object createSavepoint()：**创建一个保存点对象，以便在后面可以利用 rollbackToSavepoint(Object savepoint) 方法使事务回滚到特定的保存点上，也可以通过 releaseSavepoint() 方法释放一个已经不用的保存点。
- ❑ **void rollbackToSavepoint(Object savepoint)：**将事务回滚到特定的保存点上，被回滚的保存点将自动释放。
- ❑ **void releaseSavepoint(Object savepoint)：**释放一个保存点。如果事务提交，则所有的保存点会被自动释放，无须手工清除。

这 3 个方法在底层的资源不支持保存点时，都将抛出 NestedTransactionNotSupported Exception 异常。

TransactionStatus 扩展了 SavepointManager 接口，并提供了以下几个方法。

- ❑ **boolean hasSavepoint()：**判断当前事务是否在内部创建了一个保存点，该保存点是为了支持 Spring 的嵌套事务而创建的。

- ❑ `boolean isNewTransaction()`: 判断当前事务是否是一个新的事务, 如果返回 `false`, 则表示当前事务是一个已经存在的事务, 或者当前操作未运行在事务环境中。
- ❑ `boolean isCompleted()`: 判断当前事务是否已经结束 (已经提交或回滚)。
- ❑ `boolean isRollbackOnly()`: 判断当前事务是否已经被标识为 `rollback-only`。
- ❑ `void setRollbackOnly()`: 将当前事务设置为 `rollback-only`。通过该标识通知事务管理器只能将事务回滚, 事务管理器将通过显式调用回滚命令或抛出异常的方式回滚事务。

3. PlatformTransactionManager

通过 JDBC 的事务管理知识可以知道, 事务只能被提交或回滚 (或回滚到某个保存点后提交), Spring 高层事务抽象接口 `org.springframework.transaction.PlatformTransactionManager` 很好地描述了事务管理这个概念, 如代码清单 11-7 所示。

代码清单 11-7 PlatformTransactionManager

```
public interface PlatformTransactionManager {
    TransactionStatus getTransaction(TransactionDefinition definition)
        throws TransactionException;
    void commit(TransactionStatus status) throws TransactionException;
    void rollback(TransactionStatus status) throws TransactionException;
}
```

`PlatformTransactionManager` 只定义了 3 个接口方法, 它们是 SPI (Service Provider Interface) 高层次的接口方法。这些访问都没有和 JNDI 绑定在一起, 可以像 Spring 容器中普通的 Bean 一样对待 `PlatformTransactionManager` 实现者。

下面来了解一下 `PlatformTransactionManager` 接口方法的功能。

- ❑ `TransactionStatus getTransaction(TransactionDefinition definition)`: 该方法根据事务定义信息从事务环境中返回一个已存在的事务, 或者创建一个新的事务, 并用 `TransactionStatus` 描述这个事务的状态。
- ❑ `commit(TransactionStatus status)`: 根据事务的状态提交事务。如果事务状态已经被标识为 `rollback-only`, 则该方法将执行一个回滚事务的操作。
- ❑ `rollback(TransactionStatus status)`: 将事务回滚。当 `commit()` 方法抛出异常时, `rollback()` 方法会被隐式调用。

11.3.2 Spring 的事务管理器实现类

Spring 将事务管理委托给底层具体的持久化实现框架来完成。因此, Spring 为不同的持久化框架提供了 `PlatformTransactionManager` 接口的实现类, 如表 11-2 所示。

表 11-2 不同持久化技术对应的事务管理器实现类

事 务	说 明
<code>org.springframework.orm.jpa.JpaTransactionManager</code>	使用 JPA 进行持久化时, 使用该事务管理器

续表

事 务	说 明
org.springframework.orm.hibernateX.HibernateTransactionManager	使用 Hibernate X.0 (X 可为 3,4,5, 下同) 版本进行持久化时, 使用该事务管理器
org.springframework.jdbc.datasource.DataSource TransactionManager	使用 Spring JDBC 或 MyBatis 等基于 DataSource 数据源的持久化技术时, 使用该事务管理器
org.springframework.orm.jdo.JdoTransactionManager	使用 JDO 进行持久化时, 使用该事务管理器
org.springframework.transaction.jta.JtaTransactionManager	具有多个数据源的全局事务使用该事务管理器 (不管采用何种持久化技术)

这些事务管理器都是对特定事务实现框架的代理, 这样就可以通过 Spring 所提交的高级抽象对不同种类的事务实现使用相同的方式进行管理, 而不用关心具体的实现。

要实现事务管理, 首先要在 Spring 中配置好相应的事务管理器, 为事务管理器指定数据资源及一些其他事务管理控制属性。下面来看一下几个常见的事务管理器的配置。

1. Spring JDBC 和 MyBatis

如果使用 Spring JDBC 或 MyBatis, 由于它们都基于数据源的 Connection 访问数据库, 所以可以使用 DataSourceTransactionManager, 只要在 Spring 中进行以下配置就可以了, 如代码清单 11-8 所示。

代码清单 11-8 基于数据源的数据管理器

```
...
<bean id="dataSource" ① ← 配置一个数据源
    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}"/>
<bean id="transactionManager" ② ← 基于数据源的事务管理器
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
    p:dataSource-ref="dataSource" ③ ← 引用数据源
/>
</bean>
```

在幕后, DataSourceTransactionManager 使用 DataSource 的 Connection 的 commit()、rollback()等方法管理事务。

2. JPA

JPA 通过 javax.persistence.EntityTransaction 管理 JPA 的事务, EntityTransaction 对象可以通过 javax.persistence.EntityManager#getTransaction()方法获得, 而 EntityManager 又通过一个工厂类方法 javax.persistence.EntityManagerFactory#createEntityManager()获取。

在底层, JPA 依然通过 JDBC 的 Connection 的事务方法完成最终的控制。因此, 要配置一个 JPA 事务管理器, 必须先提供一个 DataSource, 然后配置一个 EntityManagerFactory, 最后才配置 JpaTransactionManager, 如代码清单 11-9 所示。

代码清单 11-9 JPA事务管理器配置

```

...
<bean id="entityManagerFactory"
      class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
      p:dataSource-ref="dataSource"/>① ← 指定一个数据源
...
</bean>
<bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager"
      p:entityManagerFactory-ref="entityManagerFactory"/>② ← 指定实体管理器

```

3. Hibernate

Spring 4.0 已取消了对 Hibernate 3.6 之前版本的支持, 并全面支持 Hibernate 5.0; 因此, 只为 Hibernate 3.6+ 提供了事务管理器。下面以 Hibernate 4.0 为例, 如代码清单 11-10 所示。

代码清单 11-10 Hibernate 4.0 事务管理器配置

```

...
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate4.LocalSessionFactoryBean"
      p:dataSource-ref="dataSource"① ← 指定一个数据源
      p:mappingResources="classpath:bbtForum.hbm.xml">② ← 指定 Hibernate 配置文件

      <property name="hibernateProperties">③ ← Hibernate 其他配置属性
        <props>
          <prop key="hibernate.dialect">${hibernate.dialect}</prop>
          <prop key="hibernate.show_sql">true</prop>
          <prop key="hibernate.generate_statistics">true</prop>
        </props>
      </property>
</bean>
<bean id="transactionManager"
      class="org.springframework.orm.hibernate4.HibernateTransactionManager"
      p:sessionFactory-ref="sessionFactory"④ ← 注入会话工厂

```

大部分 ORM 框架都拥有自己事务管理的 API, 它们对 DataSource 和 Connection 进行了封装。Hibernate 使用 org.hibernate.Session 封装 Connection, 所以需要有一个能够创建 Session 的 SessionFactory (更多内容请参见第 14 章)。

4. JTA

如果希望在 Java EE 容器里使用 JTA, 则将通过 JNDI 和 Spring 的 JtaTransactionManager 获取一个容器管理的 DataSource。JtaTransactionManager 不需要知道 DataSource 和其他特定的资源, 因为它引用容器提供的全局事务管理, 如代码清单 11-11 所示。

代码清单 11-11 JTA事务管理器

```

<?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:jee="http://www.springframework.org/schema/jee"

```

```

xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
http://www.springframework.org/schema/jee
http://www.springframework.org/schema/jee/spring-jee-4.0.xsd">

<jee:jndi-lookup id="accountDs" jndi-name="java:comp/env/jdbc/account"/>①
<jee:jndi-lookup id="orderDs" jndi-name="java:comp/env/jdbc/order"/>

<bean id="transactionManager"
      class="org.springframework.transaction.jta.JtaTransactionManager"/>②
</beans>
```

通过 jee 命名空间获取
Java EE 应用服务器容
器中的数据源

指定 JTA 事务管理器

这里使用了 jee 命名空间从 Java EE 容器中返回 JNDI 管理的资源。①处的配置相当于：

```

<bean id="simple"
      class="org.springframework.jndi.JndiObjectFactoryBean"
      p:jndiName="java:comp/env/jdbc/account"/>
```

显然，使用 jee 命名空间的配置更加简洁。jee 命名空间还可以获取 EJB 本地或远程的无状态 Session Bean（更多信息参见 Spring 参考手册的附录）。

11.3.3 事务同步管理器

Spring 将 JDBC 的 Connection、Hibernate 的 Session 等访问数据库的连接或会话对象统称为资源，这些资源在同一时刻是不能多线程共享的。为了让 DAO、Service 类可能做到 singleton，Spring 的事务同步管理器类 org.springframework.transaction.support.TransactionSynchronizationManager 使用 ThreadLocal 为不同事务线程提供了独立的资源副本，同时维护事务配置的属性和运行状态信息。事务同步管理器是 Spring 事务管理的基石，不管用户使用的是编程式事务管理，还是声明式事务管理，都离不开事务同步管理器。

Spring 框架为不同的持久化技术提供了一套从 TransactionSynchronizationManager 中获取对应线程绑定资源工具类，如表 11-3 所示。

表 11-3 线程绑定资源获取工具

持久化技术	线程绑定资源获取工具
Spring JDBC 或 MyBatis	org.springframework.jdbc.datasource.DataSourceUtils
Hibernate X.0	org.springframework.orm.hibernateX.SessionFactoryUtils
JPA	org.springframework.orm.jpa.EntityManagerFactoryUtils
JDO	org.springframework.orm.jdo.PersistenceManagerFactoryUtils

这些工具类都提供了静态的方法，通过这些方法可以获取和当前线程绑定的资源，如 DataSourceUtils.getConnection(DataSource dataSource)方法可以从指定的数据源中获取和当前线程绑定的 Connection，而 Hibernate 的 SessionFactoryUtils.getSession(SessionFactory

sessionFactory, boolean allowCreate)方法则可以从指定的 SessionFactory 中获取和当前线程绑定的 Session。

当需要脱离模板类,手工操作底层持久化技术的原生 API 时,就需要通过这些工具类获取线程绑定的资源,而不应该直接从 DataSource 或 SessionFactory 中获取。因为后者不能获得与本线程相关的资源,因此无法让数据操作参与到与本线程相关的事务环境中。

这些工具类还有另外一个重要的用途:将特定异常转换为 Spring 的 DAO 异常,已经在第 10 章的表 10-2 中对此进行了说明。

Spring 为不同的持久化技术提供了模板类,模板类在内部通过资源获取工具类间接访问 TransactionSynchronizationManager 中的线程绑定资源。所以,如果 DAO 使用模板类进行持久化操作,这些 DAO 就可以配置成 singleton。如果不使用模板类,也可以直接通过资源获取工具类访问线程相关的资源。

下面,让我们揭开 TransactionSynchronizationManager 的层层面纱,探寻其中的奥秘。

```
public abstract class TransactionSynchronizationManager {

    //①用于保存每个事务线程对应的Connection或Session等类型的资源
    private static final ThreadLocal resources = new ThreadLocal();

    //②用于保存每个事务线程对应事务的名称
    private static final ThreadLocal currentTransactionName = new ThreadLocal();

    //③用于保存每个事务线程对应事务的read-only状态
    private static final ThreadLocal currentTransactionReadOnly = new ThreadLocal();

    //④用于保存每个事务线程对应事务的隔离级别
    private static final ThreadLocal currentTransactionIsolationLevel= new
    ThreadLocal();

    //⑤用于保存每个事务线程对应事务的激活态
    private static final ThreadLocal actualTransactionActive = new ThreadLocal();
    ...
}
```

TransactionSynchronizationManager 将 DAO、Service 类中影响线程安全的所有“状态”统一抽取到该类中,并用 ThreadLocal 进行替换,从此 DAO(必须是基于模板类或资源获取工具类创建的 DAO)和 Service(必须采用 Spring 事务管理机制)摘掉了非线程安全的帽子,完成了脱胎换骨式的身份转变。

11.3.4 事务传播行为

当我们调用一个基于 Spring 的 Service 接口方法(如 UserService#addUser())时,它将运行于 Spring 管理的事务环境中,Service 接口方法可能会在内部调用其他的 Service 接口方法以共同完成一个完整的业务操作,因此就会产生服务接口方法嵌套调用的情

况，Spring 通过事务传播行为控制当前的事务如何传播到被嵌套调用的目标服务接口方法中。事务传播是 Spring 进行事务管理的重要概念，其重要性怎么强调都不为过。但是事务传播行为也是被误解最多的地方，本节将详细分析不同事务传播行为的表现形式，掌握它们之间的区别。

Spring 在 TransactionDefinition 接口中规定了 7 种类型的事务传播行为，它们规定了事务方法和事务方法发生嵌套调用时事务如何进行传播，如表 11-4 所示。

表 11-4 事务传播行为类型

事务传播行为类型	说 明
PROPAGATION_REQUIRED	如果当前没有事务，则新建一个事务；如果已经存在一个事务，则加入到这个事务中。这是最常见的选择
PROPAGATION_SUPPORTS	支持当前事务。如果当前没有事务，则以非事务方式执行
PROPAGATION_MANDATORY	使用当前的事务。如果当前没有事务，则抛出异常
PROPAGATION_REQUIRES_NEW	新建事务。如果当前存在事务，则把当前事务挂起
PROPAGATION_NOT_SUPPORTED	以非事务方式执行操作。如果当前存在事务，则把当前事务挂起
PROPAGATION_NEVER	以非事务方式执行。如果当前存在事务，则抛出异常
PROPAGATION_NESTED	如果当前存在事务，则在嵌套事务内执行；如果当前没有事务，则执行与 PROPAGATION_REQUIRED 类似的操作

在使用 PROPAGATION_NESTED 时，底层的数据源必须基于 JDBC 3.0，并且实现者需要支持保存点事务机制。

11.4 编程式的事务管理

在实际应用中，很少需要通过编程来进行事务管理。即便如此，Spring 还是为编程式事务管理提供了模板类 org.springframework.transaction.support.TransactionTemplate，以满足一些特殊场合的需要。

TransactionTemplate 和那些持久化模板类一样是线程安全的，因此，可以在多个业务类中共享 TransactionTemplate 实例进行事务管理。TransactionTemplate 拥有多个设置事务属性的方法，如 setReadOnly(boolean readOnly)、setIsolationLevel(int isolationLevel)等。

虽然很少需要手工编写事务管理的程序，但作为深入认识 Spring 事务管理的一种方式，了解一下编程式事务也不是坏事。TransactionTemplate 有两个主要的方法：

- ❑ void setTransactionManager(PlatformTransactionManager transactionManager)：设置事务管理器。
- ❑ Object execute(TransactionCallback action)：在 TransactionCallback 回调接口中定义需要以事务方式组织的数据访问逻辑。

TransactionCallback 接口只有一个方法：Object doInTransaction(TransactionStatus status)。如果操作不会返回结果，则可以使用 TransactionCallback 的子接口 TransactionCallbackWithoutResult，如代码清单 11-12 所示。

代码清单 11-12 采用编程式的事务管理

```
...
@Service
public class ForumService1 {
    public ForumDao forumDao;
    public TransactionTemplate template;

    @Autowired ①
    public void setTemplate(TransactionTemplate template) {
        this.template = template;
    }
    public void addForum(final Forum forum) {
        template.execute(new TransactionCallbackWithoutResult() {
            protected void doInTransactionWithoutResult(TransactionStatus status) {
                forumDao.addForum(forum); ②
            }
        });
    }
}
...
```

通过 AOP 主动注入

需要在事务环境中执行的代码

由于 Spring 事务管理基于 TransactionSynchronizationManager 进行工作，所以，如果在回调接口方法中需要显式访问底层数据连接，则必须通过资源获取工具类得到线程绑定的数据连接。这是 Spring 事务管理的底层协议，不容违反。如果 ForumDao 是基于 Spring 提供的模板类构建的，由于模板类已经在内部使用了资源获取工具类获取数据连接，所以用户就不必关心底层数据连接的获取问题。

11.5 使用 XML 配置声明式事务

大多数 Spring 用户选择声明式事务管理的功能，这种方式对代码的侵入性最小，可以让事务管理代码完全从业务代码中移除，非常符合非侵入式轻量级容器的理念。

Spring 的声明式事务管理是通过 Spring AOP 实现的，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的相应连接点中。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。

Spring 提供了和 EJB CMT 相似的声明式事务管理，这不但体现在二者的最终执行效果上，还体现在二者事务声明的语法上。但二者也存在明显的不同，下面通过表 11-5 对二者进行比较。

表 11-5 Spring和EJB CMT声明事务的不同

比较项	EJB CMT	Spring
是否绑定 JTA	绑定在 JTA 上，即使单数据源也是如此。所以 EJB 不能脱离容器运行	可以在任何环境下使用，包括直接在 Spring 中声明的数据源或在应用服务器 JNDI 中的 JTA 数据源
持久化技术支持	采用非开放的 EJB 自制持久化技术	通过少量配置即可和 JDBC、JDO、Hibernate 等持久化技术一起工作
目标类要求	必须是实现特定接口的特殊类	可以是任何 POJO，不过在内部必须使用资源获取工具类操作数据连接或会话。如果 DAO 使用模板类进行构建，则这种要求将自动得到满足
回滚规则	没有提供	提供声明式的回滚规则
开放性控制	使用 EJB CMT，除了使用 setRollbackOnly() 方法，没有办法影响容器的事务管理	Spring 允许用户通过 AOP 定制事务行为。例如，如果需要，用户可以在事务回滚中插入定制的行为，也可以增加任意的增强，就和任何 AOP 的增强一样
分布式事务	支持分布式事务。一般应用并不需要使用这样的功能	Spring 不直接支持高端应用服务器所提供的跨越远程调用的事务上下文传播。此时，可以通过 Spring 的 Java EE 服务集成来提供。此外，在 Spring 中集成 JOTM 后，Spring 也可以提供 JTA 事务的功能

回滚规则的概念比较重要，它使用户能够指定什么样的异常导致自动回滚、什么样的异常不影响事务提交，这些规则可以在配置文件中通过声明的方式指定；同时，仍然可以通过调用 `TransactionStatus#setRollbackOnly()` 方法程式地回滚当前事务。通常我们定义一条规则，如声明 `MyApplicationException` 必须总是导致事务回滚。这种方式带来了显著的好处，它使用户的业务对象不必依赖于事务设施。典型的例子是用户不必在代码中导入 Spring API、事务代码等。

对 EJB 来说，默认的行为是 EJB 容器在遇到系统异常（通常指运行时异常）时自动回滚当前事务。EJB 遇到应用异常（通常指检查型异常）时并不会自动回滚。默认情况下，Spring 的声明式事务管理和 EJB 的相同（只在遇到运行期异常时自动回滚）。



轻松一刻

在讲解 Spring 时，经常会涉及的一个问题是“框架的侵入性”。从广义上讲，“侵入性”就是对应用编程行为的影响，从这个意义上说，任何框架都逃脱不了“侵入性”的控诉。不过反过来一想：框架不对应用产生任何影响，我们还需要框架做什么呢？关键问题是这种影响是怎样施加的。EJB 以一种强权政治的方式强力扭转程序员的固有习惯，让应用开发人员的日子很难过；而 Spring 则以一种春风化雨、喜闻乐见的方式引入它的“侵入性”——这不禁让我们想起了那则著名的“太阳和风”的寓言。



11.5.1 一个将被实施事务增强的服务接口

BbtForum 拥有 4 个方法,我们希望 addTopic()和 updateForum()方法拥有写事务的能力,而其他两个方法只需要拥有读事务的能力就可以了。以下是 BbtForum 的实现代码,如代码清单 11-13 所示。

代码清单 11-13 BbtForum

```
package com.smart.service;
...
@Service
@Transactional
public class BbtForum {
    public ForumDao forumDao;
    public TopicDao topicDao;
    public PostDao postDao;
    public void addTopic(Topic topic) {
        topicDao.addTopic(topic);
        postDao.addPost(topic.getPost());
    }
    public Forum getForum(int forumId) {
        return forumDao.getForum(forumId);
    }
    public void updateForum(Forum forum) {
        forumDao.updateForum(forum);
    }
    public int getForumNum() {
        return forumDao.getForumNum();
    }
    ...
}
```

BbtForum 是一个 POJO,只是简单地使用持久层的多个 DAO 类,通过它们的协作实现业务功能。在这里,我们看不到任何事务操作的代码,所以如果直接使用 BbtForum,那么这些方法都将以无事务的方式运行。现在,我们的任务是通过 Spring 声明式事务配置让这些业务方法拥有适合的事务功能。

11.5.2 使用原始的 TransactionProxyFactoryBean

在 Spring 的早期版本中,用户必须通过 TransactionProxyFactoryBean 代理类对需要事务管理的业务类进行代理,以便实施事务功能的增强。在 Spring 2.0 后,由于可以通过 aop/tx 命名空间声明事务,因此通过该代理类实施声明式事务的方式已经不被推荐。

1. 声明式事务配置

从循序渐进的学习角度来看,了解 TransactionProxyFactoryBean 有助于我们更直观地理解 Spring 实施声明式事务的内部工作原理。所以我们不妨把使用这个代理类作为学习 Spring 事务管理的起始站,后面的章节会陆续介绍 Spring 的配置方式,如代码清单 11-14 所示。

代码清单 11-14 使用TransactionProxyFactoryBean配置

```

<!--①引入DAO和DataSource的配置文件-->
<import resource="classpath:applicationContext-dao.xml" />

<!--②声明事务管理器-->
<bean id="txManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
</bean>

<!--③需要实施事务增强的目标业务Bean -->
<bean id="bbsForumTarget"
      class="package com.smart.service.BbsForum"
      p:forumDao-ref="forumDao"
      p:topicDao-ref="topicDao"
      p:postDao-ref="postDao"/>

<bean id="bbsForum" ④ <----- 使用事务代理工厂类为目标业务 Bean 提供事务增强
      class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean"
      p:transactionManager-ref="txManager" ④-1 <----- 指定事务管理器
      p:target-ref="bbsForumTarget"> ④-2 <----- 指定目标业务 Bean
    <property name="transactionAttributes"> ④-3 <----- 事务属性配置
      <props>
        <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop> ④-4 <----- 只读事务
        <prop key="*">PROPAGATION_REQUIRED</prop> ④-5 <----- 可写事务
      </props>
    </property>
</bean>

```

当然，必须首先配置好 DAO Bean、数据源等基础设施，这些信息统一在 applicationContext-dao.xml 配置文件中定义，如①处所示（读者可以参看本书配套网盘中的具体配置）。在这个示例中，我们使用 Spring JDBC 的持久化技术，所以它的资源池是 JDBC 数据源，而事务管理器是 DataSourceTransactionManager。如果使用其他持久化技术，则需要对基础设施和事务管理器进行相应的调整。

按照约定的习惯，需要事务增强的业务类一般将 id 取名为 xxxTarget，如③处所示，这可以在字面上表示该 Bean 是要被代理的目标 Bean。

通过 TransactionProxyFactoryBean 对业务类进行代理，织入事务增强功能，如④处所示。首先，需要为该代理类指定事务管理器，这些事务管理器实现了 PlatformTransactionManager 接口；其次，通过 target 属性指定需要代理的目标 Bean；最后，为业务 Bean 的不同方法配置事务属性。Spring 允许通过键值配置业务方法的事务属性信息，键可以使用通配符，如 get* 代表目标类中所有以 get 为前缀的方法，它匹配 BbsForum 的 getForum(int forumId) 和 getForumNum() 方法；而 key="*" 匹配 BbsForum 接口的所有方法。



提示

如果将 `TransactionProxyFactoryBean` 的 `optimize` 属性设为 `true`，即添加 `<property name="optimize" value="true"/>`，则 Spring 自动通过 CGLib 动态代理为 `BbtForumImpl` 生成基于子类的代理。这时，`key="**"` 不但匹配 `BbtForum` 接口的所有方法，也匹配 `BbtForumImpl` 中的其他方法，如 `setForumDao()`、`setTopicDao()` 等方法都将织入事务管理的增强。

2. 异常回滚/提交规则

`<prop>` 内的值为事务属性信息，其配置格式如图 11-4 所示。

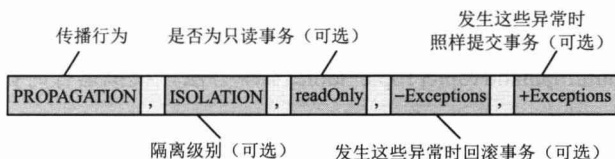


图 11-4 事务属性设置格式

传播行为是唯一必须提供的配置项，当 `<prop>` 的值为空时，也不会发生配置异常，但对应的匹配方法不会应用事务，相当于没有配置。可选的值参看表 11-4 中的定义。

隔离级别配置项是可选的，默认为 `ISOLATION_DEFAULT`，表示使用数据库默认的隔离级别。隔离级别其他可选的设置值为（具体意义参看 11.1.4 节的解释）：

- ☐ `ISOLATION_READ_UNCOMMITTED`。
- ☐ `ISOLATION_READ_COMMITTED`。
- ☐ `ISOLATION_REPEATABLE_READ`。
- ☐ `ISOLATION_SERIALIZABLE`。

如果希望将匹配方法设置为只读事务，则可添加 `readOnly` 配置项，如代码清单 11-14 中的④-4 所示。

当事务运行过程中发生异常时，事务可以被声明为回滚或继续提交。在默认情况下，当发生运行期异常时，事务将被回滚；当发生检查型异常时，既不回滚也不提交，控制权交给外层调用。这种默认的回滚规则在大多数情况下是适用的，不过用户也可以通过配置显式指定回滚规则：指定带正号（+）或负号（-）的异常类名（或异常名匹配片段）。当抛出负号型异常时，将触发事务回滚；当抛出正号型异常时，即使这个异常是检查型异常，事务也会提交。抛出的异常或该异常的祖先类的类名匹配规则中指定了异常类名（或异常名片段），规则就生效，如下面的设置：

```
<prop key="add*">PROPAGATION_REQUIRED, -Exception</prop>
```

只要业务方法运行时抛出的异常或其父类异常的类名包括“Exception”，事务都回滚。以下异常都符合这条规则：`SQLException`、`ParseException`。正因为 Spring 采用名称字符串包含的比较方式，所以用户甚至可以将回滚的规则设置为“-tion”。

因为 Spring 默认的事务回滚规则为“运行期异常回滚，检查型异常不回滚”，所以带负号的异常设置仅对检查型异常有意义。

此外，用户可以指定多个带正号或负号的事务回滚/提交规则，如下：

```
<prop key="add*">
    PROPAGATION_REQUIRED, -XxxException, -YyyException
</prop>
```

3. 异常提交的示例

举一个关于异常回滚的具体例子。假设希望 BbtForum#addTopic(Topic topic)在抛出 PessimisticLockingFailureException 异常时，依旧提交事务，则对应的配置如下：

```
<prop key="addTopic">
    PROPAGATION_REQUIRED, +PessimisticLockingFailureException
</prop>
```

调整 BbtForumImpl#addTopic(Topic topic)实现方法，模拟运行时抛出 PessimisticLockingFailureException。

```
public class BbtForum {
    ...
    public void addTopic(Topic topic) throws Exception {
        topicDao.addTopic(topic); ①
        if(true) throw new PessimisticLockingFailureException("fail");
        postDao.addPost(topic.getPost()); ②
    }
    ...
}
```

依旧会提交此处的数据持久化操作

此处的数据持久化操作无法到达

在运行这个方法时，虽然在①和②之间抛出了异常，但 Spring 最终将使①处的数据持久化操作生效，而②处的数据持久化操作将因为前面抛出了异常而无法执行。

11.5.3 基于 aop/tx 命名空间的配置

使用 TransactionProxyFactoryBean 代理工厂类为业务类添加事务性支持，在学习了动态代理的知识后，理解起来应该比较直观，但它拥有以下一些明显的缺点：

- ❑ 需要对每个需要事务支持的业务类进行单独的配置。
- ❑ 在指定事务方法时，只能通过方法名进行定义，无法利用方法签名的其他信息进行定位（如方法入参、访问域修饰符等）。
- ❑ 事务属性的配置串的规则比较麻烦，规则串虽然包括多项信息，但统一由逗号分隔的字符串来描述，不能利用 IDE 中的诱导输入功能，容易出错。
- ❑ 在为业务类 Bean 添加事务支持时，在容器中既需要定义业务类 Bean（通常命名为 xxxTarget），又需要通过 TransactionProxyFactoryBean 对其进行代理以生成支持事务的代理 Bean。实际上，我们只会从容器中返回代理的 Bean，而业务类 Bean 仅是为了能被代理才定义的；这样就造成相似的东西有两份配置，增加了配置信息量。

这一切都是因为在低版本的 Spring 中没有引入强大的 AOP 切面描述语言而造成的。

Spring 2.0 的一个重要改进是引入了 AspectJ 切面定义语言，借由这股强劲的“东风”，事务方法切面描述的难题就迎刃而解了。

Spring 在基于 Schema 的配置中添加了一个 tx 命名空间，在配置文件中以明确结构化的方式定义事务属性，大大提高了配置事务属性的便利性。配合 aop 命名空间所提供的切面定义这把“利剑”，业务类方法事务配置得到了极大的简化，而在描述能力上也得到了很大的提升。

下面通过 tx 和 aop 命名空间对上一节中基于 FactoryBean 的事务配置方式进行替换，如代码清单 11-5 所示。

代码清单 11-15 applicationContext-tx.xml: 使用 aop/tx 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:aop="http://www.springframework.org/schema/aop"
  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">
  <import resource="classpath:applicationContext-dao.xml" />

  <!--不再需要为了事务AOP增强的实施而改名换姓-->

  <!--①事务管理器-->
  <bean id="txManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
    p:dataSource-ref="dataSource"/>

  <!--②使用强大的切点表达式语言轻松定义目标方法-->
  <aop:config>
    <!--②-1通过aop定义事务增强切面-->
    <aop:pointcut id="serviceMethod"
      expression="execution(* com.smart.service.*Forum.*(..))" />
    <!--②-2引用事务增强-->
    <aop:advisor pointcut-ref="serviceMethod" advice-ref="txAdvice" />
  </aop:config>

  <!--③事务增强-->
  <tx:advice id="txAdvice" transaction-manager="txManager">
    <!--③-1事务属性定义-->
    <tx:attributes>
      <tx:method name="get*" read-only="false"/>
      <tx:method name="add*" rollback-for="Exception" />
      <tx:method name="update*" />
    </tx:attributes>
  </tx:advice>
</beans>
```

首先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 将这些配置织入对应的 Bean 中。

在这一过程中我们看到了 3 种角色：通过 aop/tx 定义的声明式事务配置信息、业务 Bean、Spring 容器。Spring 容器自动将第一者应用于第二者，从容器中返回的业务 Bean 已经是被织入事务增强的代理 Bean，即第一者和第二者在配置时不直接发生关系。

而在使用 TransactionProxyFactoryBean 进行事务配置时，TransactionProxyFactoryBean 需要直接通过 target 属性引用目标业务 Bean，结果造成目标业务 Bean 往往需要使用 target 进行命名（如 userServiceTarget），避免和最终代理 Bean 名称（如 userService）冲突。使用 aop/tx 方式后，业务 Bean 的名称不需要做任何“配合性”的调整，aop 直接通过切点表达式语言就可以对业务 Bean 进行定位。从这个意义上来说，aop/tx 的配置方式对业务 Bean 是“无侵入”的，而 TransactionProxyFactoryBean 的配置显然是“侵入式”的。

在 aop 命名空间中，通过切点表达式语言，将 com.smart.service 包下所有以 Forum 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②处所示。

<aop:advisor>引用的 txAdvice 增强是在 tx 命名空间上定义的，如③处所示。首先，事务增强一定需要一个事务管理器的支持，<tx:advice>通过 transaction-manager 属性引用了①处定义的事务管理器（它默认查找名为 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）。在③-1 中看到，原来掺杂在一起，以逗号分隔字符串定义的事务属性，现在变成了一个结构清晰的 XML 片段。相信无须进行过多的解释，大家就可以明白这个片段中元素及属性的配置语义了。<tx:method>元素拥有的属性如表 11-6 所示。

表 11-6 <tx:method>元素属性表

属 性	是否必需	默 认 值	描 述
name	是	与事务属性关联的方法名，可使用通配符（*）	如“get*”、“handle*”、“on*Event”等
propagation	否	REQUIRED	事务传播行为，可选的值为：REQUIRED、SUPPORTS、MANDATORY、REQUIRES_NEW、NOT_SUPPORTED、NEVER、NESTED
isolation	否	DEFAULT	事务隔离级别，可选的值为：DEFAULT、READ_UNCOMMITTED、READ_COMMITTED、REPEATABLE_READ、SERIALIZABLE
timeout	否	-1	事务超时的时间（以秒为单位）。如果设置为-1，则事务超时的时间由底层的事务系统所决定
read-only	否	false	事务是否只读
rollback-for	否	所有运行期异常回滚	触发事务回滚的 Exception，用异常名称的片段进行匹配。可以设置多个，以逗号分开，如“Exception1, Exception2”

续表

属 性	是否必须	默 认 值	描 述
no-rollback-for	否	所有检查型异常不回滚	不触发事务回滚的 Exception，用异常名称的片段进行匹配。可以设置多个，以逗号分开，如“Exception1, Exception2”

如果需要为不同的业务类 Bean 应用不同的事务管理风格，则可以在<aop:config>中定义另外多套事务切面，具体的配置方法在现有基础上演绎即可，此处不再赘述。关于基于 Schema 配置切面的基本知识，请参见第 8 章。



提示

基于 aop/tx 配置的声明式事务管理是实际应用中最为使用的事务管理方式，它的表达能力最强且使用最为灵活。

11.6 使用注解配置声明式事务

除了基于 XML 的事务配置，Spring 还提供了基于注解的事务配置，即通过 @Transactional 对需要事务增强的 Bean 接口、实现类或方法进行标注；在容器中配置基于注解的事务增强驱动，即可启用基于注解的声明式事务。笔者在实际项目中一般采用这种配置方式。

11.6.1 使用@Transactional 注解

顺着原来的思路，使用@Transactional 对基于 aop/tx 命名空间的事务配置进行改造，比较二者在使用方式上的差异，如代码清单 11-16 所示。

代码清单 11-16 使用@Transactional注解的业务类

```
package com.smart.service;
import org.springframework.transaction.annotation.Transactional;
...
//①对业务类进行事务增强的标注
@Service
@Transactional
public class BbtForum {
    public void addTopic(Topic topic) throws Exception {
        topicDao.addTopic(topic);
        postDao.addPost(topic.getPost());
    }
    public Forum getForum(int forumId) {
        return forumDao.getForum(forumId);
    }
    ...
}
```

因为注解本身具有一组普适性的默认事务属性，所以往往只要在需要事务管理的业务类中添加一个 `@Transactional` 注解，就完成了业务类事务属性的配置。

当然，注解只提供元数据，它本身并不能完成事务切面织入的功能。因此，还需要在 Spring 配置文件中通过一行小小的配置“通知”Spring 容器对标注 `@Transactional` 注解的 Bean 进行加工处理，如代码清单 11-17 所示。

代码清单 11-17 applicationContext-anno.xml: 使事务注解生效

```
<?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">
  <context:component-scan base-package="com.smart"/>
  <import resource="classpath:applicationContext-dao.xml" />
  <bean id="txManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
        p:dataSource-ref="dataSource"/>

  <!--①对标注@Transactional注解的Bean进行加工处理，以织入事务管理切面-->
  <tx:annotation-driven transaction-manager="txManager"/>
</beans>
```

在默认情况下，`<tx:annotation-driven>`会自动使用名为“transactionManager”的事务管理器。所以，如果用户的事务管理器 id 为“transactionManager”，则可以进一步将①处的配置简化为“`<tx:annotation-driven/>`”。

`<tx:annotation-driven>`还有另外两个属性。

- ☐ **proxy-target-class**: 如果为 true，则 Spring 将通过创建子类来代理业务类；如果为 false，则使用基于接口的代理。如果使用子类代理，则需要在类路径中添加 CGLib.jar 类库。
- ☐ **order**: 如果业务类除事务切面外，还需要织入其他的切面，则通过该属性可以控制事务切面在目标连接点的织入顺序。

1. 关于 `@Transactional` 的属性

基于 `@Transactional` 注解的配置和基于 XML 的配置方式一样，它拥有一组普适性很强的默认事务属性，往往可以直接使用这些默认的属性。

- ☐ **事务传播行为**: PROPAGATION_REQUIRED。
- ☐ **事务隔离级别**: ISOLATION_DEFAULT。
- ☐ **读写事务属性**: 读/写事务。
- ☐ **超时时间**: 依赖于底层的事务系统的默认值。
- ☐ **回滚设置**: 任何运行期异常引发回滚，任何检查型异常不会引发回滚。

因为这些默认设置在大多数情况下都是适用的，所以一般不需要手工设置事务注解的属性（见表 11-7）。当然，Spring 允许通过手工设定属性值覆盖默认值。

表 11-7 @Transactional注解属性说明

属 性 名	说 明
propagation	事务传播行为，通过以下枚举类提供合法值： org.springframework.transaction.annotation.Propagation 例如：@Transactional(propagation=Propagation.REQUIRES_NEW)
isolation	事务隔离级别，通过以下枚举类提供合法值： org.springframework.transaction.annotation.Isolation 例如：@Transactional(isolation=Isolation.READ_COMMITTED)
readOnly	事务读写性，布尔型。例如：@Transactional(readOnly=true)
timeout	超时时间，int 型，以秒为单位。例如：@Transactional(timeout=10)
rollbackFor	一组异常类，遇到时进行回滚，类型为：Class<? extends Throwable>[]，默认值为{}。例如： @Transactional(rollbackFor={SQLException.class})。多个异常之间可用逗号分隔
rollbackForClassName	一组异常类名，遇到时进行回滚，类型为 String[]，默认值为{}。例如： @Transactional(rollbackForClassName={"Exception"})
noRollbackFor	一组异常类，遇到时不回滚，类型为 Class<? extends Throwable>[]，默认值为{}
noRollbackForClassName	一组异常类名，遇到时不回滚，类型为 String[]，默认值为{}

2. 在何处标注@Transactional 注解

@Transactional 注解可以被应用于接口定义和接口方法、类定义和类的 public 方法上。

但 Spring 建议在业务实现类上使用@Transactional 注解。当然也可以在业务接口上使用@Transactional 注解，但这样会留下一些容易被忽视的隐患。因为注解不能被继承，所以在业务接口中标注的@Transactional 注解不会被业务实现类继承。如果通过以下配置启用子类代理：

```
<tx:annotation-driven transaction-manager="txManager" proxy-target-class="true"/>
```

那么业务类不会添加事务增强，照样工作在非事务环境下。举一个具体的实例：如果使用子类代理，假设用户为 BbtForum 接口标注了@Transactional 注解，那么其实现类 BbtForumImpl 依旧不会启用事务机制。

因此，Spring 建议在具体业务类上使用@Transactional 注解。这样，不管<tx:annotation-driven>将 proxy-target-class 属性配置为 true 或 false，业务类都会启用事务机制。

3. 在方法处使用注解

方法处的注解会覆盖类定义处的注解。如果有些方法需要使用特殊的事务属性，则可以在类注解的基础上提供方法注解，如下：

```
@Transactional ①  
public class BbtForum {  
    @Transactional(readOnly=true) ②  
    public Forum getForum(int forumId) {
```

类级注解，适用于
类中所有 public 的方法

提供额外的注解信息，它
将覆盖①处的类级注解

```

        return forumDao.getForum(forumId);
    }
    ...
}

```

② 处的方法注解提供了 `readOnly` 事务属性的设置，它将覆盖类级注解中默认的 `readOnly=false` 设置。

4. 使用不同的事务管理器

一般情况下，一个应用仅需使用一个事务管理器。如果希望在不同的地方使用不同的事务管理器，则可以通过如下方式实现：

```

@Service
public class MultiForumService {
    //①使用名为topic的事务管理器
    @Transactional("topic")
    public void addTopic(Topic topic) throws Exception {
        ...
    }

    //②使用名为forum的事务管理器
    @Transactional("forum")
    public void updateForum(Forum forum) {
        ...
    }
}

```

而 `topic` 和 `forum` 的事务管理器可以在 XML 中分别定义，如下：

```

<bean id="forumTxManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
    p:dataSource-ref="forumDataSource">① ← 可以使用不同的数据源
    <qualifier value="forum"/> ② ← 为事务管理器指定一个名字
</bean>

<bean id="topicTxManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
    p:dataSource-ref="topicDataSource">
    <qualifier value="topic"/>
</bean>
<tx:annotation-driven/>

```

在①处，为事务管理器指定了一个数据源，每个事务管理器都可以绑定一个独立的数据源。在②处，指定了一个可被 `@Transactional` 注解引用的事务管理器标识。

在一两处使用带标识的 `@Transactional` 注解也许是合适的，但是如果到处都使用，则显得比较啰唆。可以自定义一个绑定到特定事务管理器的注解，然后直接使用这个自定义的注解进行标识，如下：

```

package com.smart;
...

@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Transactional("forum") //①绑定到forum的事务管理器中
public @interface ForumTransactional {
}

```

按相似的方法，还可以定义一个绑定到 topic 事务管理器的 @TopicTransactional 注解。完成定义后，就可以用以下方式对原来的代码进行调整：

```
public class MultiForumService {
    //①使用名为topic的事务管理器
    @TopicTransactional
    public void addTopic(Topic topic) throws Exception {
        ...
    }

    //②使用名为forum的事务管理器
    @ForumTransactional
    public void updateForum(Forum forum) {
        ...
    }
}
```

11.6.2 通过 AspectJ LTW 引入事务切面

Spring 还提供了对 AspectJ 切面的支持。在 org.springframework.aspects-{version}.jar 中有一个用 AspectJ 语法编写的 AnnotationTransactionAspect 切面类，用于为使用了 @Transactional 注解的业务类提供事务增强。

使用 -javaagent: D:\masterSpring\spring\dist\org.springframework.aspects-{version}.jar 的 JVM 参数，在类路径 META-INF 目录下提供如下 AspectJ 配置文件：

```
<?xml version="1.0"?>
<aspectj>
    <aspects>
        <aspect name="org.springframework.transaction.aspectj.
            AnnotationTransactionAspect" />
    </aspects>
    <weaver
        options="-showWeaveInfo -XmessageHandlerClass:
            org.springframework.aop.aspectj.AspectJWeaverMessageHandler">
        <include within="com.smart.service.impl.*" />
    </weaver>
</aspectj>
```

在类加载期，对标注 @Transactional 注解的类织入 AnnotationTransactionAspect 事务增强切面。

由于 AnnotationTransactionAspect 切面类需要使用事务管理器，所以必须利用 Spring IoC 为切面类提供事务管理器的注入，如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.0.xsd">
    <context:component-scan base-package="com.smart"/>
    <import resource="classpath:applicationContext-dao.xml" />
```

```

<bean id=" transactionManager "①
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
    p:dataSource-ref="dataSource"/>

<tx:annotation-driven/>
</beans>

```

如①处所示，通过 transactionManager 属性注入事务管理器 Bean，这样切面类就能为目标业务类提供事务管理功能。

11.7 集成特定的应用服务器

一般来说，Spring 的事务抽象与应用服务器是无关的。不过，如果用户希望事务管理器使用特定的 UserTransaction 和 TransactionManager 对象（可以被自动探测），以获取更多的事务控制功能（如事务挂起等），则可以采用 Spring 为集成这些应用服务器所提供的适配器。

11.7.1 BEA WebLogic

在一个使用 WebLogic 9.0 或更高版本的环境下，用户一般会优先选用特定于 WebLogic 的 WebLogicJtaTransactionManager 类取代基础的 JtaTransactionManager 类；因为在 WebLogic 环境中，该类提供了对 Spring 事务定义的完全支持，超过了标准的 JTA 语义。用户可以使用以下配置达到目的：

```

<bean id="txManager"
    class="org.springframework.transaction.jta.WebLogicJtaTransactionManager"/>

```

它的特性包括：支持事务名、支持为每个事务定义隔离级别，以及在任何环境下正确地恢复事务的能力。

11.7.2 WebSphere

在 WebSphere 6.1 及以上版本的环境下，用户可以使用 Spring 的 WebSphereUowTransactionManager 类。这个特殊的适配器支持 IBM 的 UOWManager 的 API，在该适配器中，Spring 所支持的事务挂起特性得到了 IBM 官方的支持。

```

<bean id="txManager"
    class="org.springframework.transaction.jta.WebSphereUowTransactionManager"/>

```

11.8 小结

Spring 声明式事务管理是 Spring 中的亮点，也是被使用最多的功能之一。Spring 使

声明式事务平民化,现在可以在 Spring 轻量级容器中享受这项曾经只能在臃肿、厚重的 EJB 应用服务器中才拥有的功能。

Spring 事务管理是 Spring AOP 技术的精彩应用的案例。事务管理作为一个切面织入目标业务方法的周围,业务方法完全从事务代码中解脱出来,代码的复杂度大大降低。被织入的事务代码基于 Spring 事务同步管理器进行工作,事务同步管理器维护着业务类对象线程相关的资源。DAO 类需要利用资源获取工具访问底层数据连接,或者直接建立在相应的持久化模板类的基础上。要了解它们的内部机制,就必须事先了解 ThreadLocal 的工作机制。

Spring 的事务配置相对来说比较简单,这些配置主要提供两方面的信息:其一,切点信息,用于定位实施事务切面的业务类方法;其二,控制事务行为的事务属性,这些属性包括事务隔离级别、事务传播行为、超时时间、回滚规则等。理解事务属性具体配置值的实际意义是非常关键的,因此 11.1 节对此专门进行了讲解。

Spring 通过 aop/tx Schema 命名空间和 @Transactional 注解技术,大大简化了声明式事务配置的难度。同时,了解基于 TransactionProxyFactoryBean 代理类定义声明式事务的工作机制,有助于理解事务增强的内部过程。

第 12 章

Spring 的事务管理难点剖析

Spring 最成功、最吸引人的地方莫过于轻量级的声明式事务管理,仅此一点, Spring 就宣告了重量级 EJB 容器的覆灭。Spring 声明式事务管理将开发者从繁复的事务管理代码中解脱出来,专注于业务逻辑的开发上,这不啻为一件可以被拿来顶礼膜拜的事情。但是,世界并未从此太平,开发人员需要面对的是层出不穷的应用场景,这些场景往往逾越了普通 Spring 技术书籍的理想界定。因此,随着应用开发的深入,在使用经过 Spring 层层封装的声明式事务时,开发人员越来越觉得自己坠入了迷雾、陷入了沼泽,体会不到外界所宣称的那种畅快淋漓。本章的目标旨在整理并剖析实际应用中种种让我们迷茫的场景,让阳光照进云遮雾障的山头。

本章主要内容:

- ◆ DAO、应用分层和事务管理的关系
- ◆ 事务方法嵌套调用的分析
- ◆ 在多线程环境下事务方法的调用问题
- ◆ 使用多种数据访问技术的问题
- ◆ Spring AOP 事务增强有哪些限制
- ◆ 数据连接泄露的原因及规避

本章亮点:

- ◆ 以实战的角度,深入剖析事务管理的众多难点
- ◆ 解构 Spring 数据连接维护的内部原理
- ◆ 如何彻底告别数据连接泄露

12.1 DAO 和事务管理的牵绊

很少有使用 Spring 而不使用 Spring 事务管理器的应用，因此常常有人会问：是否用了 Spring，就一定要用 Spring 事务管理器，否则就无法进行数据的持久化操作呢？事务管理器和 DAO 是什么关系呢？

也许是 DAO 和事务管理如影随形的缘故吧，这个看似简单的问题实实在在地存在着，从初学者心中涌出，萦绕在老把式的脑际。答案当然是否定的。我们都知道，事务管理的目的是保证数据操作的事务性（原子性、一致性、隔离性、持久性，即所谓的 ACID），脱离了事务性，DAO 照样可以顺利地进行数据操作。

12.1.1 JDBC 访问数据库

下面来看一段使用 Spring JDBC 进行数据访问的代码，如代码清单 12-1 所示。

代码清单 12-1 UserJdbcWithoutTransManagerService

```
package com.smart.withouttx.jdbc;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Service;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.apache.commons.dbcp.BasicDataSource;

@Service("userService")
public class UserJdbcWithoutTransManagerService {
    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    public void addScore(String userName, int toAdd) {
        String sql = "UPDATE t_user u SET u.score = u.score + ? WHERE user_name =?";
        jdbcTemplate.update(sql, toAdd, userName);
    }

    public static void main(String[] args) {
        ApplicationContext ctx = new
            ClassPathXmlApplicationContext("com/smart/withouttx/jdbc/jdbcWithoutTx.xml");
        UserJdbcWithoutTransManagerService service =
            (UserJdbcWithoutTransManagerService) ctx.getBean("userService");
        JdbcTemplate jdbcTemplate = (JdbcTemplate) ctx.getBean("jdbcTemplate");
        BasicDataSource basicDataSource = (BasicDataSource) jdbcTemplate.getDataSource();
    }
}
```

```

//①检查数据源autoCommit的设置
System.out.println("autoCommit:"+ basicDataSource.getDefaultAutoCommit());

//②插入一条记录, 初始分数为10
jdbcTemplate.execute("INSERT INTO t_user (user_name,password,score,last_logon_time)
VALUES('tom','123456',10,"+System.currentTimeMillis()+")");

//③调用工作在无事务环境下的服务类方法, 将分数添加20分
service.addScore("tom",20);

//④查看此时用户的分数
int score = jdbcTemplate.queryForInt(
    "SELECT score FROM t_user WHERE user_name ='tom'");
System.out.println("score:"+score);
jdbcTemplate.execute("DELETE FROM t_user WHERE user_name='tom'");
}
}

```

其中, jdbcWithoutTx.xml 配置文件如代码清单 12-2 所示。

代码清单 12-2 jdbcWithoutTx.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:context="http://www.springframework.org/schema/context"
    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
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.0.xsd">
    <context:component-scan base-package="com.smart.withouttx.jdbc"/>
    <context:property-placeholder location="classpath:jdbc.properties"/>
    <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}"/>

    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate"
        p:dataSource-ref="dataSource"/>
</beans>

```

运行 UserJdbcWithoutTransManagerService, 在控制台上打印出如下结果:

```

defaultAutoCommit:true
score:30

```

在 jdbcWithoutTx.xml 中没有配置任何事务管理器, 但是数据已经成功持久化到数据库中。在默认情况下, dataSource 数据源的 autoCommit 被设置为 true——这也意味着所有通过 JdbcTemplate 执行的语句马上提交, 没有事务。如果将 dataSource 的 defaultAutoCommit 设置为 false, 再次运行 UserJdbcWithoutTransManagerService, 将抛出错误, 原因是新增及更改数据的操作都没有提交到数据库, 所以代码清单 12-1 中④处的语句因无法从数据库中查询到匹配的记录而引发异常。

对于强调读速度的应用，数据库本身可能就不支持事务，如使用 MyISAM 引擎的 MySQL 数据库。这时，无须在 Spring 应用中配置事务管理器，因为即使配置了，也是没有实际用处的。

12.1.2 Hibernate 访问数据库

对于 Hibernate 来说，情况就有点复杂了。因为 Hibernate 的事务管理拥有其自身的意义，它和 Hibernate 一级缓存存在密切的关系：在调用 Session 的 save、update 等方法时，Hibernate 并不直接向数据库发送 SQL 语句，只在提交事务（commit）或 flush 一级缓存时才真正向数据库发送 SQL。所以，即使底层数据库不支持事务，Hibernate 的事务管理也是有一定好处的，不会对数据操作的效率造成负面影响。所以，如果使用 Hibernate 数据访问技术，则没有理由不配置 HibernateTransactionManager 事务管理器。

但是，如果不使用 Hibernate 事务管理器，Spring 就会采取默认的事务管理策略搜索（PROPAGATION_REQUIRED, readOnly）。如果有修改操作是不允许的，就会抛出异常。来看下面的例子，如代码清单 12-3 所示。

代码清单 12-3 UserHibernateWithoutTransManagerService

```
package com.smart.withouttx.hiber;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.simple.SimpleJdbcTemplate;
import org.springframework.stereotype.Service;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.orm.hibernate4.HibernateTemplate;
import org.apache.commons.dbcp.BasicDataSource;
import org.springframework.test.jdbc.SimpleJdbcTestUtils;

import com.smart.User;

@Service("hiberService")
public class UserHibernateWithoutTransManagerService {
    private HibernateTemplate hibernateTemplate;

    @Autowired
    public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {
        this.hibernateTemplate = hibernateTemplate;
    }

    public void addScore(String userName, int toAdd) {
        User user = hibernateTemplate.get(User.class, userName);

        //①以下两条语句取消注释后，由于默认事务管理器不支持数据更改，将报异常
```

```

        //user.setScore(user.getScore()+toAdd);
        //hibernateTemplate.update(user);
    }

    public static void main(String[] args) {
        ApplicationContext ctx = new
            ClassPathXmlApplicationContext("com/smart/withouttx/hiber/
hiberWithoutTx.xml");
        UserHibernateWithoutTransManagerService service =
            (UserHibernateWithoutTransManagerService)ctx.getBean("hiberService");

        JdbcTemplate jdbcTemplate = (JdbcTemplate)ctx.getBean("jdbcTemplate");
        BasicDataSource basicDataSource = (BasicDataSource)jdbcTemplate.
getDataSource();

        //②检查数据源autoCommit的设置
        System.out.println("autoCommit:"+ basicDataSource.getDefaultAutoCommit());

        //③插入一条记录, 初始分数为10
        jdbcTemplate.execute("INSERT INTO t_user(user_name,password,score,last_
logon_time)
            VALUES('tom','123456',10,"+System.currentTimeMillis()+")");

        //④调用工作在无事务环境下的服务类方法, 将分数添加20分
        service.addScore("tom",20);

        //⑤查看此时用户的分数
        int score = jdbcTemplate.queryForInt(
            "SELECT score FROM t_user WHERE user_name='tom'");
        System.out.println("score:"+score);
        jdbcTemplate.execute("DELETE FROM t_user WHERE user_name='tom'");
    }
}

```

此时, 采用 hiberWithoutTx.xml 配置文件, 其配置内容如代码清单 12-4 所示。

代码清单 12-4 hiberWithoutTx.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:context="http://www.springframework.org/schema/context"
    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
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.0.xsd">
    <context:component-scan base-package="com.smart.withouttx.hiber"/>
    <context:property-placeholder location="classpath:jdbc.properties"/>
    <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}"/>

```



```

<bean id="jdbcTemplate"
      class="org.springframework.jdbc.core.JdbcTemplate"
      p:dataSource-ref="dataSource"/>

<bean id="sessionFactory"
      class="org.springframework.orm.hibernate4.LocalSessionFactoryBean"
      p:dataSource-ref="dataSource">
  <property name="annotatedClasses">
    <list>
      <value>com.smart.User</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
      <prop key="hibernate.show_sql">true</prop>
    </props>
  </property>
</bean>

<bean id="hibernateTemplate"
      class="org.springframework.orm.hibernate4.HibernateTemplate"
      p:sessionFactory-ref="sessionFactory"/>
</beans>

```

com.smart.User 是使用了 Hibernate 注解的领域对象，如代码清单 12-5 所示。

代码清单 12-5 User

```

package com.smart;

import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.Column;
import javax.persistence.Id;
import java.lang.reflect.Field;
import java.io.Serializable;

@Entity
@Table(name="T_USER")
public class User implements Serializable{
    @Id
    @Column(name = "USER_NAME")
    private String userName;

    private String password;

    private int score;

    @Column(name = "LAST_LOGON_TIME")
    private long lastLogonTime = 0;

    ...
}

```

运行 UserHibernateWithoutTransManagerService，如果代码清单 12-3 的①处取消注

释，那么程序将报错。这说明 Hibernate 在 Spring 中，在没有事务管理器的情况下，只能读数据而无法更改数据。

12.2 应用分层的迷惑

Web、Service 及 DAO 三层是 Web 应用开发常见的模式，但有些开发人员却错误地认为，如果要使用 Spring 的事务管理就一定要先进行三层的划分，更有甚者认为每层一定要先定义一个接口，然后再定义一个实现类。对将“面向接口编程”奉为圭臬，认为放之四海而皆准的论调，笔者并不是很赞同。是的，“面向接口编程”是 Martin Fowler、Rod Johnson 这些大师所提倡的行事原则。如果拿这条原则去开发框架、产品或大型项目，怎么强调都不为过。但是，对于一般的开发人员来说，也许做的是一个普通工程项目，往往只是一些对数据库增、删、查、改的功能。此时，过分强制“面向接口编程”除了会带来更多的类文件，并不会有什么好处。

Spring 框架所提供的各种好处（如 AOP、注解增强、注解 MVC 等）的唯一前提就是让 POJO 的类变成一个受 Spring 容器管理的 Bean。下面的实例用一个 POJO 完成所有的功能，既是 Controller，又是 Service，还是 DAO，如代码清单 12-6 所示。

代码清单 12-6 MixLayerUserService

```
package com.smart.mixlayer;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

//①将POJO类通过注解变成Spring MVC的Controller
@Controller
public class MixLayerUserService {

    //②自动注入JdbcTemplate
    @Autowired
    private JdbcTemplate jdbcTemplate;

    //③通过Spring MVC注解映射成为处理HTTP请求的函数，同时作为一个拥有事务性的方法
    @RequestMapping("/logon.do")
    @Transactional
    public String logon(String userName,String password){
        if(isRightUser(userName,password)){
            String sql = "UPDATE t_user u SET u.score = u.score + ? WHERE user_name =?";
            jdbcTemplate.update(sql,20,userName);
            return "success";
        }else{
            return "fail";
        }
    }
}
```

```
private boolean isRightUser(String userName,String password){
    //do sth
    return true;
}
}
```

通过@Controller 注解将 MixLayerUserService 变成 Web 层的 Controller, 同时也是 Service 层的服务类。此外, 由于直接使用 JdbcTemplate 访问数据, 所以 MixLayerUserService 还是一个 DAO。来看一下对应的 Spring 配置文件, 如代码清单 12-7 所示。

代码清单 12-7 applicationContext.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:context="http://www.springframework.org/schema/context"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:aop="http://www.springframework.org/schema/aop"
       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/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">
    ...
    <!--①事务管理配置-->
    <bean id="transactionManager"
          class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
          p:dataSource-ref="dataSource"/>
    <tx:annotation-driven/>

    <!--②启动Spring MVC的注解功能-->
    <bean class="org.springframework.web.servlet.mvc.annotation.
        ② AnnotationMethodHandlerAdapter"/>
    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"
          p:prefix="/WEB-INF/jsp/" p:suffix=".jsp"/>
</beans>
```

在①处, 通过事务注解驱动使 MixLayerUserService 的 logon()工作在事务环境下; 在②处, 配置了 Spring MVC 的一些基本设施。要使程序能够运行起来, 还必须进行 web.xml 的相关配置, 如代码清单 12-8 所示。

代码清单 12-8 web.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
```

```

<servlet>
    <servlet-name>user</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet
</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:com/smart/mixlayer/applicationContext.xml
</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>user</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
</web-app>

```

这个配置文件很简单，唯一需要注意的是 DispatcherServlet 的配置。默认情况下，Spring MVC 根据 Servlet 的名字查找 WEB-INF 下的 <servletName>-servlet.xml 作为 Spring MVC 的配置文件，在此，通过 contextConfigLocation 参数显式指定 Spring MVC 配置文件的确切位置。

将 org.springframework.jdbc 及 org.springframework.transaction 的日志级别设置为 DEBUG，启动项目，并访问 <http://localhost/forum/logon.do?userName=tom> 应用，MixLayerUserService#logon() 方法将做出响应，查看后台输出日志，如下：

```

Returning cached instance of singleton bean 'transactionManager'
Creating new transaction with name [com.smart.mixlayer.MixLayerUserService.logon]:
PROPAGATION_REQUIRED,ISOLATION_DEFAULT; ''
(DataSourceTransactionManager.java:204) - Acquired Connection [jdbc:mysql://localhost:3306/sampledbs, UserName=root@localhost, MySQL-AB JDBC Driver] for JDBC transaction
(DataSourceTransactionManager.java:221) - Switching JDBC Connection [jdbc:mysql://localhost:3306/sampledbs, UserName=root@localhost, MySQL-AB JDBC Driver] to manual commit
(JdbcTemplate.java:810) - Executing prepared SQL update
(JdbcTemplate.java:569) - Executing prepared SQL statement [UPDATE t_user u SET u.score = u.score + ? WHERE user_name =?]
(JdbcTemplate.java:819) - SQL update affected 0 rows
(AbstractPlatformTransactionManager.java:752) - Initiating transaction commit
(DataSourceTransactionManager.java:264) - Committing JDBC transaction on
Connection [jdbc:mysql://localhost:3306/sampledbs, UserName=root@localhost, MySQL-AB
JDBC Driver]
...

```

日志中的粗体部分说明 MixLayerUserService#logon() 方法已经成功地运行在事务上下文中。

Spring 框架本身不应是代码复杂化的理由，使用 Spring 的开发者应该是无拘无束的：从实际应用出发，根据实际需要进行编程。

12.3 事务方法嵌套调用的迷茫

12.3.1 Spring 事务传播机制回顾

Spring 事务的一个被讹传很广的说法是：一个事务方法不应该调用另一个事务方法，否则将产生两个事务。结果造成开发人员在设计事务方法时束手束脚，生怕一不小心就踩到地雷。

其实这是未正确认识 Spring 事务传播机制而造成的误解。Spring 对事务控制的支持统一在 `TransactionDefinition` 类中描述，该类有以下几个重要的接口方法。

- ☐ `int getPropagationBehavior()`：事务的传播行为。
- ☐ `int getIsolationLevel()`：事务的隔离级别。
- ☐ `int getTimeout()`：事务的过期时间。
- ☐ `boolean isReadOnly()`：事务的读/写特性。

很明显，除了事务的传播行为，对于事务的其他特性，Spring 是借助底层资源的功能来完成的，Spring 无非充当了一个代理的角色。但是事务的传播行为却是 Spring 凭借自身的框架提供的功能，是 Spring 提供给开发者最珍贵的礼物，讹传的说法玷污了 Spring 事务框架最美丽的光环。

所谓事务传播行为，就是多个事务方法相互调用时，事务如何在这些方法间传播。Spring 支持以下 7 种事务传播行为。

- ☐ `PROPAGATION_REQUIRED`：如果当前没有事务，就新建一个事务；如果已经存在一个事务，就加入到这个事务中。这是最常见的选择。
- ☐ `PROPAGATION_SUPPORTS`：支持当前事务。如果当前没有事务，就以非事务方式执行。
- ☐ `PROPAGATION_MANDATORY`：使用当前事务。如果当前没有事务，就抛出异常。
- ☐ `PROPAGATION_REQUIRES_NEW`：新建事务。如果当前存在事务，就把当前事务挂起。
- ☐ `PROPAGATION_NOT_SUPPORTED`：以非事务方式执行操作。如果当前存在事务，就把当前事务挂起。
- ☐ `PROPAGATION_NEVER`：以非事务方式执行。如果当前存在事务，就抛出异常。
- ☐ `PROPAGATION_NESTED`：如果当前存在事务，就在嵌套事务内执行；如果当前没有事务，就执行与 `PROPAGATION_REQUIRED` 类似的操作。

Spring 默认的事务传播行为是 `PROPAGATION_REQUIRED`，它适合绝大多数情况。如果多个 `ServiceX#methodX()` 均工作在事务环境下（均被 Spring 事务增强），且程序中存在调用链 `Service1#method1()->Service2#method2()->Service3#method3()`，那么这 3 个服务类的 3 个方法通过 Spring 的事务传播机制都工作在同一个事务中。

12.3.2 相互嵌套的服务方法

来看一下实例，UserService#login()方法内部调用了 UserService#updateLastLogonTime() 和 ScoreService#addScore()方法，这两个类都继承于 BaseService。它们之间的类结构如图 12-1 所示。

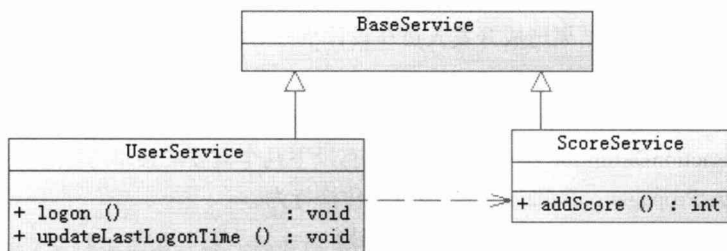


图 12-1 两个相互调用的服务类

UserService#login()方法内部调用了 ScoreService#addScore()方法，二者分别通过 Spring AOP 进行了事务增强，则它们工作在同一事务中。来看具体的代码，如代码清单 12-9 所示。

代码清单 12-9 UserService

```

package com.smart.nestcall;
...
@Service("userService")
public class UserService extends BaseService {
    private JdbcTemplate jdbcTemplate;
    private ScoreService scoreService;

    @Autowired
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    @Autowired
    public void setScoreService(ScoreService scoreService) {
        this.scoreService = scoreService;
    }

    //①该方法嵌套调用了本类的其他方法及其他服务类的方法
    public void login(String userName) {
        System.out.println("before userService.updateLastLogonTime...");
        updateLastLogonTime(userName); //①-1本服务类的其他方法
        System.out.println("after userService.updateLastLogonTime...");

        System.out.println("before scoreService.addScore...");
        scoreService.addScore(userName, 20); //①-2其他服务类的方法
        System.out.println("after scoreService.addScore...");
    }

    public void updateLastLogonTime(String userName) {

```

```
String sql = "UPDATE t_user u SET u.last_logon_time = ? WHERE user_name =?";
jdbcTemplate.update(sql, System.currentTimeMillis(), userName);
}
```

在 UserService 中注入了 ScoreService 的 Bean，而 ScoreService 的代码如代码清单 12-10 所示。

代码清单 12-10 ScoreService

```
package com.smart.nestcall;

...
@Service("scoreService")
public class ScoreService extends BaseService{

    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    public void addScore(String userName, int toAdd) {
        String sql = "UPDATE t_user u SET u.score = u.score + ? WHERE user_name =?";
        jdbcTemplate.update(sql, toAdd, userName);
    }
}
```

通过 Spring 配置为 ScoreService 及 UserService 中的所有公有方法都添加 Spring AOP 的事务增强，让 UserService 的 logon() 和 updateLastLogonTime() 方法及 ScoreService 的 addScore() 方法都工作在事务环境下。下面是关键的配置代码，如代码清单 12-11 所示。

代码清单 12-11 nestcall\applicationContext.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:context="http://www.springframework.org/schema/context"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:aop="http://www.springframework.org/schema/aop"
    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/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.nestcall"/>
    ...
    <bean id="jdbcManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
        p:dataSource-ref="dataSource"/>

    <!--①通过以下配置为所有继承BaseService类的所有子类的所有public方法都添加事务增强-->
    <aop:config proxy-target-class="true">
```



```

        <aop:pointcut id="serviceJdbcMethod"
            expression="within(com.smart.nestcall.BaseService+)" />
        <aop:advisor pointcut-ref="serviceJdbcMethod" advice-ref="jdbcAdvice"
order="0"/>
    </aop:config>
    <tx:advice id="jdbcAdvice" transaction-manager="jdbcManager">
        <tx:attributes>
            <tx:method name="*" />
        </tx:attributes>
    </tx:advice>
</beans>

```

将日志级别设置为 DEBUG，启动 Spring 容器并执行 UserService#logon()方法，仔细观察如下输出日志：

```

before userService.logon method...

//①创建了一个事务
Creating new transaction with name [com.smart.nestcall.UserService.logon]:
PROPAGATION_REQUIRED,ISOLATION_DEFAULT
Acquired Connection [jdbc:mysql://localhost:3306/sampled,
UserName=root@localhost, MySQL-AB JDBC Driver] for JDBC transaction
Switching JDBC Connection [jdbc:mysql://localhost:3306/sampled,
UserName=root@localhost, MySQL-AB JDBC Driver] to manual commit
before userService.updateLastLogonTime...

<!--②updateLastLogonTime()和logon()在同一个Bean中，并未发生加入已存在事务上下文的
动作，而是“天然”地工作在相同的事务上下文中-->
Executing prepared SQL update
Executing prepared SQL statement [UPDATE t_user u SET u.last_logon_time = ? WHERE
user_name =?]
SQL update affected 1 rows
after userService.updateLastLogonTime...
before scoreService.addScore...

//③ScoreService#addScore()方法加入到①处启动的事务上下文中
Participating in existing transaction
Executing prepared SQL update
Executing prepared SQL statement [UPDATE t_user u SET u.score = u.score + ? WHERE
user_name =?]
SQL update affected 1 rows
after scoreService.addScore...
Initiating transaction commit
Committing JDBC transaction on Connection [jdbc:mysql://localhost:3306/sampled,
UserName=root@localhost, MySQL-AB JDBC Driver]
...
after userService.logon method...

```

从上面的输出日志中可以清楚地看到，Spring 为 UserService#logon()方法启动了一个新的事务，而 UserServe#updateLastLogonTime()和 UserService#logon()方法在相同的类中，没有观察到有事务传播行为的发生，其代码块好像“直接合并”到 UserService#logon()方法中。

然而在执行到 ScoreService#addScore()方法时，我们观察到发生了一个事务传播行为：“Participating in existing transaction”；这说明 ScoreService#addScore()方法添加到了

UserService#logon()方法的事务上下文中，二者共享同一个事务。所以最终的结果是 UserService 的 logon()和 updateLastLogonTime()方法及 ScoreService 的 addScore()方法工作在同一个事务中。

12.4 多线程的困惑

12.4.1 Spring 通过单实例化 Bean 简化多线程问题

由于 Spring 的事务管理器是通过线程相关的 ThreadLocal 来保存数据访问基础设施 (Connection 实例) 的，再结合 IoC 和 AOP 实现高级声明式事务的功能，所以 Spring 的事务天然地和线程有着千丝万缕的联系。

我们知道 Web 容器本身就是多线程的，Web 容器为一个 HTTP 请求创建一个独立的线程 (实际上大多数 Web 容器采用共享线程池)，所以由此请求所涉及的 Spring 容器中的 Bean 也运行在多线程环境下。在绝大多数情况下，Spring 的 Bean 都是单实例的 (singleton)，单实例 Bean 的最大好处是线程无关性，不存在多线程并发访问的问题，也就是线程安全的。

一个类能够以单实例的方式运行的前提是“无状态”，即一个类不能拥有状态化的成员变量。我们知道，在传统的编程中，DAO 必须持有一个 Connection，而 Connection 就是状态化的对象。所以传统的 DAO 不能做成单实例的，每次要用时都必须创建一个新的实例。传统的 Service 由于内部包含了若干有状态的 DAO 成员变量，所以其本身也是有状态的。

但在 Spring 中，DAO 和 Service 都以单实例的方式存在。Spring 通过 ThreadLocal 将有状态的变量 (如 Connection 等) 本地线程化，达到另一个层面上的“线程无关”，从而实现线程安全。Spring 不遗余力地将有状态的对象无状态化，就是要达到单实例化 Bean 的目的。

由于 Spring 已经通过 ThreadLocal 的设施将 Bean 无状态化，所以 Spring 中的单实例 Bean 对线程安全问题拥有了一种天生的免疫能力。不但单实例的 Service 可以成功运行在多线程环境中，Service 本身还可以自由地启动独立线程以执行其他的 Service。

12.4.2 启动独立线程调用事务方法

下面对 logon()方法进行改造，让其在方法内部再启动一个新线程，在这个新线程中执行积分添加的操作，看看究竟会发生哪些事务行为，如代码清单 12-12 所示。

代码清单 12-12 UserService

```

package com.smart.multithread;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.stereotype.Service;
import org.apache.commons.dbcp.BasicDataSource;

@Service("userService")
public class UserService extends BaseService {
    private JdbcTemplate jdbcTemplate;
    private ScoreService scoreService;

    @Autowired
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    @Autowired
    public void setScoreService(ScoreService scoreService) {
        this.scoreService = scoreService;
    }

    public void login(String userName) {
        System.out.println("before userService.updateLastLogonTime method...");
        updateLastLogonTime(userName);
        System.out.println("after userService.updateLastLogonTime method...");

        //①在同一个线程中调用scoreService#addScore(), 将运行在同一个事务中
        //scoreService.addScore(userName, 20);

        //②在一个新线程中执行scoreService#addScore(), 将启动一个新的事务
        Thread myThread = new MyThread(this.scoreService, userName, 20);
        myThread.start();
    }

    public void updateLastLogonTime(String userName) {
        String sql = "UPDATE t_user u SET u.last_logon_time = ? WHERE user_name =?";
        jdbcTemplate.update(sql, System.currentTimeMillis(), userName);
    }

    //③负责执行scoreService#addScore()的线程类
    private class MyThread extends Thread {
        private ScoreService scoreService;
        private String userName;
        private int toAdd;

        private MyThread(ScoreService scoreService, String userName, int toAdd) {
            this.scoreService = scoreService;
            this.userName = userName;
            this.toAdd = toAdd;
        }
    }
}

```

```

public void run() {
    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("before scoreService.addScor method...");
    scoreService.addScore(userName, toAdd);
    System.out.println("after scoreService.addScor method...");
}
}
}

```

将日志级别设置为 DEBUG，执行 UserService#logon() 方法，观察以下输出日志：

before userService.logon method...

//①创建一个事务

```

Creating new transaction with name [com.smart.multithread.UserService.logon]:
PROPAGATION_REQUIRED, ISOLATION_DEFAULT
Acquired Connection [jdbc:mysql://localhost:3306/sampled,
UserName=root@localhost, MySQL-AB JDBC Driver] for JDBC transaction
...
SQL update affected 1 rows
after userService.updateLastLogonTime method...
Initiating transaction commit

```

//②提交①处开启的事务

```

Committing JDBC transaction on Connection [jdbc:mysql://localhost:3306/sampled,
UserName=root@localhost, MySQL-AB JDBC Driver]
Releasing JDBC Connection [jdbc:mysql://localhost:3306/sampled, UserName=
root@localhost, MySQL-AB JDBC Driver] after transaction
...
Returning JDBC Connection to DataSource
before scoreService.addScor method...

```

//③创建一个事务

```

Creating new transaction with name [com.smart.multithread.ScoreService.addScore]:
PROPAGATION_REQUIRED, ISOLATION_DEFAULT
Acquired Connection [jdbc:mysql://localhost:3306/sampled,
UserName=root@localhost, MySQL-AB JDBC Driver] for JDBC transaction
...
SQL update affected 0 rows
Initiating transaction commit

```

//④提交③处开启的事务

```

Committing JDBC transaction on Connection [jdbc:mysql://localhost:3306/sampled,
UserName=root@localhost, MySQL-AB JDBC Driver]
Releasing JDBC Connection [jdbc:mysql://localhost:3306/sampled,
UserName=root@localhost, MySQL-AB JDBC Driver] after transaction...

```

在①处，在主线程（main）中执行的 UserService#logon() 方法的事务启动；在②处，其对应的事务提交。而在子线程（Thread-2）中执行的 ScoreService#addScore() 方法的事务在③处启动；在④处提交其对应的事务。

所以可以得出这样的结论：在相同线程中进行相互嵌套调用的事务方法工作在相同的事务中。如果这些相互嵌套调用的方法工作在不同的线程中，则不同线程下的事务方法工作在独立的事务中。

12.5 联合军种作战的混乱

12.5.1 Spring 事务管理器的应对

Spring 抽象的 DAO 体系兼容多种数据访问技术，它们各有特色、各有千秋。如 Hibernate 是一个非常优秀的 ORM 实现方案，但对底层 SQL 的控制不太方便；而 MyBatis 则通过模板化技术让用户方便地控制 SQL，但没有 Hibernate 那样高的开发效率；自由度最高的当然是直接使用 Spring JDBC 了，但它也是底层的，灵活的代价是代码的繁复。很难说哪种数据访问技术是最优秀的，只有在某种特定的场景下才能给出答案。所以在 一个应用中往往采用多种数据访问技术，一般是两种，一种采用 ORM 技术框架，而另一种采用偏 JDBC 的底层技术；二者珠联璧合，形成联合军种，共同御敌。

但是，这种联合军种如何应对事务管理的问题呢？我们知道，Spring 为每种数据访问技术提供了相应的事务管理器，难道需要分别为它们配置对应的事务管理器吗？它们到底是如何协同工作的呢？这些层出不穷的问题往往压制了开发人员使用联合军种的想法。

其实，在这个问题上，我们低估了 Spring 事务管理的能力。如果用户采用了一种高端的 ORM 技术（Hibernate、JPA、JDO），同时还采用了一种 JDBC 技术（Spring JDBC、MyBatis）；由于前者的会话（Session）是对后者连接（Connection）的封装，Spring 会“足够智能地”在同一个事务线程中让前者的会话封装后者的连接。所以，只要直接采用前者的事务管理器就可以了。表 12-1 给出了混合数据访问技术框架所对应的事务管理器。

表 12-1 事务隔离级别对并发问题的解决情况

序 号	混合数据访问技术框架	事务管理器
1	Hibernate+ Spring JDBC 或 MyBatis	org.springframework.orm.hibernateX.HibernateTransactionManager
2	JPA+Spring JDBC 或 MyBatis	org.springframework.orm.jpa.JpaTransactionManager
3	JDO+Spring JDBC 或 MyBatis	org.springframework.orm.jdo.JdoTransactionManager

12.5.2 Hibernate+Spring JDBC 混合框架的事务管理

由于一般不会出现同时使用多个 ORM 框架的情况（如 Hibernate+JPA），所以我们不拟对此命题展开论述，只重点研究 ORM 框架+JDBC 框架的情况。Hibernate+Spring

JDBC 可能是被使用得最多的组合，本节通过实例观察事务管理的运作情况，如代码清单 12-13 所示。

代码清单 12-13 UserService

```
package com.smart.mixedao;

...

@Service("userService")
public class UserService extends BaseService {
    private HibernateTemplate hibernateTemplate;
    private ScoreService scoreService;

    @Autowired
    public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {
        this.hibernateTemplate = hibernateTemplate;
    }

    @Autowired
    public void setScoreService(ScoreService scoreService) {
        this.scoreService = scoreService;
    }

    public void logon(String userName) {

        //①通过Hibernate技术访问数据
        System.out.println("before updateLastLogonTime()..");
        updateLastLogonTime(userName);
        System.out.println("end updateLastLogonTime()..");

        //②通过JDBC技术访问数据
        System.out.println("before scoreService.addScore()..");
        scoreService.addScore(userName, 20);
        System.out.println("end scoreService.addScore()..");
    }

    public void updateLastLogonTime(String userName) {
        User user = hibernateTemplate.get(User.class, userName);
        user.setLastLogonTime(System.currentTimeMillis());
        hibernateTemplate.update(user);

        //③这句很重要，请看下文的分析
        hibernateTemplate.flush();
    }
}
```

在①处使用 Hibernate 技术操作数据，而在②处调用 ScoreService#addScore()，该方法内部使用 Spring JDBC 技术操作数据。

在③处显式调用了 flush()方法，将 Session 中的缓存同步到数据库中（马上向数据库发送一条更新记录的 SQL 语句）。之所以要显式执行 flush()方法，是因为在默认情况下，Hibernate 对数据的更改只记录在一级缓存中，要等到事务提交或显式调用 flush()方法时才会将一级缓存中的数据同步到数据库中，而提交事务的操作发生在 logon()方法

返回前。如果所有针对数据库的更改操作都使用 Hibernate，那么这种数据同步的延迟机制并不会产生任何问题。但是，我们在 `logon()` 方法中同时采用了 Hibernate 和 Spring JDBC 数据访问技术，而 Spring JDBC 无法自动感知 Hibernate 一级缓存；所以如果不及时调用 `flush()` 方法将记录数据更改的一级缓存同步到数据库中，则②处通过 Spring JDBC 进行数据更改的结果将被 Hibernate 一级缓存中的更改覆盖掉，因为 Hibernate 一级缓存要等到 `logon()` 方法返回前才同步到数据库中。

ScoreService 使用了 Spring JDBC 数据访问技术，如代码清单 12-14 所示。

代码清单 12-14 UserService

```
package com.smart.mixdao;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Service;
import org.apache.commons.dbcp.BasicDataSource;

@Service("scoreService")
public class ScoreService extends BaseService{
    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    public void addScore(String userName, int toAdd) {
        String sql = "UPDATE t_user u SET u.score = u.score + ? WHERE user_name =?";
        jdbcTemplate.update(sql, toAdd, userName);
        BasicDataSource basicDataSource = (BasicDataSource)
jdbcTemplate.getDataSource();

        //①查看此处数据库激活的连接数量
        System.out.println("[scoreService.addScore]激活连接数量: "
            +basicDataSource.getNumActive());
    }
}
```

Spring 关键的配置文件代码如代码清单 12-15 所示。

代码清单 12-15 mixdao\applicationContext.xml

```
...
<!--①使用Hibernate事务管理器-->
<bean id="hiberManager"
    class="org.springframework.orm.hibernate4.HibernateTransactionManager"
    p:sessionFactory-ref="sessionFactory"/>

<!--②使UserService及ScoreService的共用方法都拥有事务-->
<aop:config proxy-target-class="true">
    <aop:pointcut id="serviceJdbcMethod"
        expression="within(com.smart.mixdao.BaseService+)" />
    <aop:advisor pointcut-ref="serviceJdbcMethod"
        advice-ref="hiberAdvice"/>
</aop:config>
```



```

</aop:config>
<tx:advice id="hiberAdvice" transaction-manager="hiberManager">
  <tx:attributes>
    <tx:method name="*" />
  </tx:attributes>
</tx:advice>

```

```
</beans>
```

启动 Spring 容器，执行 UserService#logon() 方法，可以看到如下执行日志：

```
before userService.logon()..
```

①在执行 userService.logon() 方法后，Spring 开启一个事务

```

Creating new transaction with name [com.smart.mixdao.UserService.logon]:
PROPAGATION_REQUIRED,ISOLATION_DEFAULT
opened session at timestamp: 13009379637
Opened new Session [org.hibernate.impl.SessionImpl@c5f468] for Hibernate
transaction
...
Exposing Hibernate transaction as JDBC transaction
[jdbc:mysql://localhost:3306/sampledbs, Username=root@localhost, MySQL-AB JDBC Driver]
before userService.updateLastLogonTime()..

```

②userService.updateLastLogonTime() 方法执行时自动绑定到①处开启的 Session 中

```

Found thread-bound Session for HibernateTemplate
loading entity: [com.smart.User#tom]
...

```

```

Not closing pre-bound Hibernate Session after HibernateTemplate
end updateLastLogonTime()..

```

```
before scoreService.addScore()..
```

③scoreService.addScore() 方法执行时绑定到①处开启的 Session 中，并加入其所对应的事务中

```

Found thread-bound Session [org.hibernate.impl.SessionImpl@c5f468] for Hibernate
transaction
Participating in existing transaction
...
SQL update affected 1 rows

```

④此时数据源只打开了一个连接

```

[scoreService.addScore]激活连接数量: 1
end scoreService.addScore()..
Initiating transaction commit

```

⑤提交 Hibernate 的事务，它将触发一级缓存到数据库的同步

```

Committing Hibernate transaction on Session [org.hibernate.impl.SessionImpl
@c5f468]
commit
...
com.smart.User{lastLogonTime=1300937963882,score=10,userName=tom,password=
123456}
re-enabling autocommit

```

⑥提交 Session 底层所绑定的 JDBC Connection 所对应的事务

```
committed JDBC Connection
```



```

transaction completed on session with on_close connection release mode; be sure
to close the session to release JDBC resources!
...
transaction completed on session with on_close connection release mode; be sure
to close the session to release JDBC resources!
after userService.logon()..

```

仔细观察这段输出日志，在①处 `UserService#logon()` 方法开启一个新的事务。②处的 `UserService#updateLastLogonTime()` 方法绑定到事务上下文的 `Session` 中。③处的 `ScoreService#addScore()` 方法加入到①处开启的事务上下文中。④处的输出是 `ScoreService#addScore()` 方法内部的输出信息，汇报此时数据源激活的连接数为 1，这清楚地告诉我们，Hibernate 和 JDBC 这两种数据访问技术在同一事务上下文中“共用”一个连接。在⑤处提交 Hibernate 事务，接着在⑥处触发调用底层的 `Connection` 提交事务。

从以上运行结果可以得出这样的结论：使用 Hibernate 事务管理器后，可以混合使用 Hibernate 和 Spring JDBC 数据访问技术，它们将工作在同一事务上下文中。但是在使用 Spring JDBC 访问数据时，Hibernate 的一级或二级缓存得不到同步。此外，一级缓存延迟数据同步机制可能会覆盖 Spring JDBC 数据更改的结果。

由于混合数据访问技术方案存在“事务同步而缓存不同步”的情况，所以最好用 Hibernate 进行读/写操作，而只用 Spring JDBC 进行读操作。如用 Spring JDBC 进行简要列表的查询，而用 Hibernate 对查询出来的数据进行维护。

如果确实要同时使用 Hibernate 和 Spring JDBC 读/写数据，则必须充分考虑到 Hibernate 缓存机制引发的问题：必须整体分析数据维护逻辑，根据需要及时调用 Hibernate 的 `flush()` 方法，以免覆盖 Spring JDBC 的更改，在 Spring JDBC 更改数据库时，维护 Hibernate 的缓存。由于方法调用顺序的不同可能会影响数据的同步性，因而很容易发生问题，这会极大地提高数据访问程序的复杂性。所以笔者郑重建议不要同时使用 Spring JDBC 和 Hibernate 对数据进行写操作。

可以将以上结论推广到其他混合数据访问技术的方案中，如 Hibernate+MyBatis、JPA+Spring JDBC、JDO+Spring JDBC 等。

12.6 特殊方法成漏网之鱼

12.6.1 哪些方法不能实施 Spring AOP 事务

由于 Spring 事务管理是基于接口代理或动态字节码技术，通过 AOP 实施事务增强的，虽然 Spring 依然支持 AspectJ LTW 在类加载期实施增强，但这种方法很少使用，所以我们不予关注。

对于基于接口动态代理的 AOP 事务增强来说，由于接口的方法都必须是 `public` 的，这就要求实现类的实现方法也必须是 `public` 的（不能是 `protected`、`private` 等），同时不

能使用 `static` 修饰符。所以，可以实施接口动态代理的方法只能是使用 `public` 或 `public final` 修饰符的方法，其他方法不可能被动态代理，相应地也就不能实施 AOP 增强。换句话说，即不能进行 Spring 事务增强。

基于 CGLib 字节码动态代理的方案是通过扩展被增强类，动态创建其子类的方式进行 AOP 增强植入的。由于使用 `final`、`static`、`private` 修饰符的方法都不能被子类覆盖，相应地这些方法将无法实施 AOP 增强。所以方法签名必须特别注意这些修饰符的使用，以免使方法不小心成为事务管理的“漏网之鱼”。

12.6.2 事务增强遗漏实例

本节通过具体的实例说明基于 CGLib 字节码动态代理无法享受 Spring AOP 事务增强的特殊方法，如代码清单 12-16 所示。

代码清单 12-16 UserService

```
package com.smart.special;
import org.springframework.stereotype.Service;
@Service("userService")
public class UserService {
    private void method1() { //① private方法因访问权限的限制，无法被子类覆盖
        System.out.println("method1");
    }
    public final void method2() { //② final方法无法被子类覆盖
        System.out.println("method2");
    }
    public static void method3() { //③ static是类级别的方法，无法被子类覆盖
        System.out.println("method3");
    }
    public void method4() { //④ public方法可以被子类覆盖，因此可以被动态字节码增强
        System.out.println("method4");
    }
    public final void method5() { //⑤ final方法不能被子类覆盖
        System.out.println("in method5");
    }
    protected void method6() { //⑥ protected方法可以被子类覆盖，因此可以被动态字节码增强
        System.out.println("in method6");
    }
}
```

Spring 通过 CGLib 动态代理技术对 UserService Bean 实施 AOP 事务增强的关键配置具体如代码清单 12-17 所示。

代码清单 12-17 applicationContext.xml

```
<aop:config proxy-target-class="true"><!-- ①显式使用CGLib动态代理 -->
    <!-- ②希望对UserService的所有方法实施事务增强 -->
    <aop:pointcut id="serviceJdbcMethod"
        expression="execution(* com.smart.special.UserService.*(..))"/>
```

```

    <aop:advisor pointcut-ref="serviceJdbcMethod" advice-ref="jdbcAdvice"
order="0"/>
  </aop:config>
  <tx:advice id="jdbcAdvice" transaction-manager="jdbcManager">
    <tx:attributes>
      <tx:method name="*" />
    </tx:attributes>
  </tx:advice>

```

在①处通过 `proxy-target-class="true"` 显式使用 CGLib 动态代理技术；在②处通过 AspectJ 切点表达式表达 `UserService` 的所有方法，希望对 `UserService` 的所有方法都实施 Spring AOP 事务增强。

在 `UserService` 中添加一个可执行的方法，如代码清单 12-18 所示。

代码清单 12-18 `UserService`

```

package com.smart.special;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.stereotype.Service;

@Service("userService")
public class UserService {
    ...
    public static void main(String[] args) {
        ApplicationContext ctx =
new ClassPathXmlApplicationContext("user/special/applicationContext.xml");
        UserService service = (UserService) ctx.getBean("userService");

        System.out.println("before method1");
        service.method1();
        System.out.println("after method1");

        System.out.println("before method2");
        service.method2();
        System.out.println("after method2");

        ...

        System.out.println("before method5");
        service.method5();
        System.out.println("after method5");

        System.out.println("before method6");
        service.method6();
        System.out.println("after method6");
    }
}

```

在运行 `UserService` 之前，将 Log4J 日志级别设置为 DEBUG。运行以上代码查看输出日志，如下：

①未启用事务

```
before method1
```

```
in method1
after method1
```

②未启用事务

```
before method2
in method2
after method2
```

③未启用事务

```
before method3
in method3
after method3
```

④启用事务

```
before method4
```

```
Creating new transaction with name [com.smart.special.UserService.method4]:
```

```
PROPAGATION_REQUIRED,ISOLATION_DEFAULT
```

```
...
```

```
in method4
```

```
Initiating transaction commit
```

```
Committing JDBC transaction on Connection [jdbc:mysql://localhost:3306/sampledb,
```

```
UserName=root@localhost, MySQL-AB JDBC Driver]
```

```
Releasing JDBC Connection [jdbc:mysql://localhost:3306/sampledb,
```

```
UserName=root@localhost, MySQL-AB JDBC Driver] after transaction
```

```
Returning JDBC Connection to DataSource
```

```
after method4
```

⑤未启用事务

```
before method5
in method5
after method5
```

@启用事务

```
before method6
```

```
Creating new transaction with name [com.smart.special.UserService.method6]:
```

```
PROPAGATION_REQUIRED,ISOLATION_DEFAULT
```

```
...
```

```
in method6
```

```
Initiating transaction commit
```

```
Committing JDBC transaction on Connection [jdbc:mysql://localhost:3306/sampledb,
```

```
UserName=root@localhost, MySQL-AB JDBC Driver]
```

```
...
```

```
after method6
```

观察以上输出日志，很容易发现 method1 至 method3 及 method5 这 4 个方法都没有被实施 Spring 的事务增强，而 method4 和 method6 被实施了事务增强。该结果验证了我们之前的论述。

表 12-2 描述了哪些特殊方法将成为 Spring AOP 事务增强的“漏网之鱼”。

表 12-2 不能被Spring AOP事务增强的方法

序 号	动态代理策略	不能被事务增强的方法
1	基于接口的动态代理	除 public 外的其他所有方法。此外，public static 也不能被增强
2	基于 CGLib 的动态代理	private、static、final 方法

不过,需要特别指出的是,这些不能被 Spring 事务增强的特殊方法并非就不工作在事务环境下。只要它们被外层的事务方法调用了,由于 Spring 事务管理的传播级别,内部方法也可以工作在外层方法所启动的事务上下文中。我们说这些方法不能被 Spring 进行 AOP 事务增强,是指这些方法不能启动事务,但是外层方法的事务上下文依旧可以顺利地传播到这些方法中。

这些不能被 Spring 事务增强的方法和可被 Spring 事务增强的方法的唯一区别在于“是否可以主动启动一个新事务”:前者不能而后者可以。对于事务传播行为来说,二者是完全相同的,前者也和后者一样不会造成数据连接的泄露问题。换句话说,如果这些“特殊方法”被无事务上下文的方法调用,则它们就工作在无事务上下文中;反之,如果被具有事务上下文的方法调用,则它们就工作在事务上下文中。

对于 private 方法,由于最终都会被 public 方法封装后再开放给外部调用,而 public 方法是可以被事务增强的,所以基本上没有什么问题。在实际开发中,最容易造成隐患的是基于 CGLib 动态代理的 public static 和 public final 这两种特殊方法。原因是它们本身是 public 的,因此可以直接被外部类(如 Web 层的 Controller 类)调用,只要调用者没有事务上下文,这些特殊方法也就以无事务的方式运作。

12.7 数据连接泄露

12.7.1 底层连接资源的访问问题

对于应用开发者来说,数据连接泄露无疑是一个可怕的梦魇。只要你开发的应用存在数据连接泄露的问题,应用系统最终都将因数据连接资源耗尽而崩溃,甚至还可能引起数据库的崩溃。数据连接泄露像一个黑洞那样让开发者避之唯恐不及。

Spring DAO 对所有支持的数据访问技术框架都使用模板化技术进行了薄层封装。只要你的程序都使用 Spring DAO 的模板(如 JdbcTemplate、HibernateTemplate 等)进行数据访问,就一定不会存在数据连接泄露的问题——这是 Spring 给予我们的郑重承诺!如果使用 Spring DAO 模板进行数据操作,则无须关注数据连接(Connection)及其衍生品(如 Hibernate 的 Session 等)的获取和释放操作,模板类已经通过其内部流程替我们完成了,且对开发者是透明的。

但是由于集成第三方产品、整合遗留代码等原因,可能需要直接访问数据源或直接获取数据连接及其衍生品。这时,如果使用不当,就可能在无意中制造出一个魔鬼般的连接泄露问题。

我们知道,当 Spring 事务方法运行时,就产生一个事务上下文,该上下文在本事务执行线程中针对同一个数据源绑定了一个唯一的数据连接(或其衍生品),所有被该事务上下文传播的方法都共享这个数据连接。这个数据连接从数据源获取到返回给数据源

都在 Spring 的掌控之中, 不会发生问题。如果在需要数据连接时能够获取这个被 Spring 管控的数据连接, 则使用者可以放心使用, 无须关注连接释放的问题。

那么, 如何获取这些被 Spring 管控的数据连接呢? Spring 提供了两种方法: 其一是使用数据资源获取工具类; 其二是对数据源(或其衍生品, 如 Hibernate 的 SessionFactory) 进行代理。

12.7.2 Spring JDBC 数据连接泄露

如果从数据源直接获取连接, 且在使用完成后不主动归还给数据源(调用 Connection#close()方法), 则将造成数据连接泄露的问题, 如代码清单 12-19 所示。

代码清单 12-19 JdbcUserService

```
package com.smart.connleak;
...
@Service("jdbcUserService")
public class JdbcUserService {
    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    @Transactional
    public void logon(String userName) {
        try {
            //①直接从数据源获取连接, 后续程序没有显式释放该连接
            Connection conn = jdbcTemplate.getDataSource().getConnection();
            String sql = "UPDATE t_user SET last_logon_time=? WHERE user name =?";
            jdbcTemplate.update(sql, System.currentTimeMillis(), userName);

            //②模拟程序代码的执行时间
            Thread.sleep(1000);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

JdbcUserService 通过 Spring AOP 事务增强的配置, 让所有 public 方法都工作在事务环境中, 即让 logon()和 updateLastLogonTime()方法拥有事务功能。在 logon()方法内部, 在①处通过调用 jdbcTemplate.getDataSource().getConnection()方法显式获取一个连接, 这个连接不是 logon()方法事务上下文线程绑定的连接, 所以如果开发者没有手工释放这个连接(显式调用 Connection#close()方法), 则这个连接将永久被占用(处于 active 状态), 造成连接泄露。下面编写模拟运行的代码, 查看方法执行对数据连接的实际占用情况, 如代码清单 12-20 所示。

代码清单 12-20 JdbcUserService

```

package com.smart.connleak;
...
@Service("jdbcUserService")
public class JdbcUserService {
    ...
    //①以异步线程的方式执行JdbcUserService#login()方法,以模拟多线程的环境
    public static void asynchrLogon(JdbcUserService userService, String userName) {
        UserServiceRunner runner = new UserServiceRunner(userService, userName);
        runner.start();
    }
    private static class UserServiceRunner extends Thread {
        private JdbcUserService userService;
        private String userName;
        public UserServiceRunner(JdbcUserService userService, String userName) {
            this.userService = userService;
            this.userName = userName;
        }
        public void run() {
            userService.login(userName);
        }
    }
    //②让主执行线程睡眠一段指定的时间
    public static void sleep(long time) {
        try {
            Thread.sleep(time);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    //③汇报数据源的连接占用情况
    public static void reportConn(BasicDataSource basicDataSource) {
        System.out.println("连接数[active:idle]-[" +
            basicDataSource.getNumActive() + ":" + basicDataSource.getNumIdle() + "]);
    }
    public static void main(String[] args) {
        ApplicationContext ctx =
            new ClassPathXmlApplicationContext("com/smart/connleak/applicatonContext.xml");
        JdbcUserService userService = (JdbcUserService) ctx.getBean("jdbcUserService");

        BasicDataSource basicDataSource = (BasicDataSource) ctx.getBean("dataSource");

        //④汇报数据源初始连接占用情况
        JdbcUserService.reportConn(basicDataSource);

        JdbcUserService.asynchrLogon(userService, "tom");//启动一个异步线程A
        JdbcUserService.sleep(500);

        //⑤此时线程A正在执行JdbcUserService#login()方法
        JdbcUserService.reportConn(basicDataSource);
    }
}

```



```

JdbcUserService.sleep(2000);

//⑥此时线程A所执行的JdbcUserService#login()方法已经执行完毕
JdbcUserService.reportConn(basicDataSource);

JdbcUserService.asynchrLogon(userService, "john");//启动一个异步线程B
JdbcUserService.sleep(500);

//⑦此时线程B正在执行JdbcUserService#login()方法
JdbcUserService.reportConn(basicDataSource);

JdbcUserService.sleep(2000);

//⑧此时线程A和B都已完成JdbcUserService#login()方法的执行
JdbcUserService.reportConn(basicDataSource);
}

```

在 JdbcUserService 中添加一个可异步执行 login() 方法的 asynchrLogon() 方法, 通过异步执行 login() 方法及让主线程睡眠的方式模拟多线程环境下的执行场景。在不同的执行点, 通过 reportConn() 方法汇报数据源连接的占用情况。

通过 Spring 事务声明, 对 JdbcUserService 的 login() 方法进行事务增强, 配置代码如下代码清单 12-21 所示。

代码清单 12-21 applicationContext.xml

```

<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
    ...
    http://www.springframework.org/schema/tx/spring-tx-4.0.xsd">
<context:component-scan base-package="com.smart.connleak"/>
<context:property-placeholder location="classpath:jdbc.properties"/>
<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}"/>

<bean id="jdbcTemplate"
    class="org.springframework.jdbc.core.JdbcTemplate"
    p:dataSource-ref="dataSource"/>

<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
    p:dataSource-ref="dataSource"/>

<!--①启用注解驱动的事务增强-->
<tx:annotation-driven/>
</beans>

```

然后运行 JdbcUserService, 在控制台上将观察到如下输出信息:

```

连接数[active:idle]-[0:0]
连接数[active:idle]-[2:0]

```

连接数[active:idle]-[1:1]

连接数[active:idle]-[3:0]

连接数[active:idle]-[2:1]

下面通过表 12-3 对数据源连接的占用和泄露情况进行描述。

表 12-3 执行过程数据源连接占用情况

时间	执行线程 1	执行线程 2	数据源连接		
			active	idle	leak
T0	未启动	未启动	0	0	0
T1	正在执行方法	未启动	2	0	0
T2	执行完毕	未启动	1	1	1
T3	执行完毕	正在执行方法	3	0	1
T4	执行完毕	执行完毕	2	1	2

可见,在执行线程 1 执行完毕后,只释放了一个数据连接,还有一个数据连接处于 active 状态,这说明泄露了一个连接。类似的,在执行线程 2 执行完毕后,也泄露了一个连接,原因是直接通过数据源获取连接(jdbcTemplate.getDataSource().getConnection())而没有显式释放。

12.7.3 事务环境下通过 DataSourceUtils 获取数据连接

Spring 提供了一个能从当前事务上下文中获取绑定的数据连接的工具类,即 DataSourceUtils。Spring 强调必须使用 DataSourceUtils 获取数据连接, Spring 的 JdbcTemplate 内部也是通过 DataSourceUtils 来获取连接的。DataSourceUtils 提供了若干获取和释放数据连接的静态方法,说明如下。

- ❑ static Connection doGetConnection(DataSource dataSource): 首先尝试从事务上下文中获取连接,失败后再从数据源获取连接。
- ❑ static Connection getConnection(DataSource dataSource): 和 doGetConnection() 方法的功能一样,实际上,其内部就是通过调用 doGetConnection()方法获取连接的。
- ❑ static void doReleaseConnection(Connection con, DataSource dataSource): 释放连接,放回到连接池中。
- ❑ static void releaseConnection(Connection con, DataSource dataSource): 和 doReleaseConnection()方法的功能一样,实际上,其内部就是通过调用 doReleaseConnection()方法获取连接的。

下面看一下 DataSourceUtils 从数据源获取连接的关键代码,如代码清单 12-22 所示。

代码清单 12-22 DataSourceUtils

```
public abstract class DataSourceUtils {
    ...
    public static Connection doGetConnection(DataSource dataSource) throws SQLException {
```

```
Assert.notNull(dataSource, "No DataSource specified");
```

```
//①首先尝试从事务同步管理器中获取数据连接
```

```
ConnectionHolder conHolder =
    (ConnectionHolder) TransactionSynchronizationManager.getResource(dataSource);
if (conHolder != null && (conHolder.hasConnection() ||
    conHolder.isSynchronizedWithTransaction())) {
    conHolder.requested();
    if (!conHolder.hasConnection()) {
        logger.debug("Fetching resumed JDBC Connection from DataSource");
        conHolder.setConnection(dataSource.getConnection());
    }
    return conHolder.getConnection();
}
```

```
//②如果获取不到连接,则直接从数据源中获取连接
```

```
Connection con = dataSource.getConnection();
```

```
//③如果拥有事务上下文,则将连接绑定到事务上下文中
```

```
if (TransactionSynchronizationManager.isSynchronizationActive()) {
    ConnectionHolder holderToUse = conHolder;
    if (holderToUse == null) {
        holderToUse = new ConnectionHolder(con);
    }
    else {holderToUse.setConnection(con);}
    holderToUse.requested();
    TransactionSynchronizationManager.registerSynchronization(
        new ConnectionSynchronization(holderToUse, dataSource));
    holderToUse.setSynchronizedWithTransaction(true);
    if (holderToUse != conHolder) {
        TransactionSynchronizationManager.bindResource(
            dataSource, holderToUse);
    }
}
return con;
}
```

首先查看当前是否存在事务管理上下文,并尝试从事务管理上下文中获取连接。如果获取失败,则直接从数据源中获取连接。在获取连接后,如果当前拥有事务上下文,则将连接绑定到事务上下文中。

在代码清单 12-23 的 JdbcUserService 中,使用 DataSourceUtils.getConnection()方法替换直接从数据源中获取连接的代码。

代码清单 12-23 JdbcUserService

```
package com.smart.connleak;
...
@Service("jdbcUserService")
public class JdbcUserService {
    private JdbcTemplate jdbcTemplate;

    @Autowired
```

```

public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
    this.jdbcTemplate = jdbcTemplate;
}

@Transactional
public void logon(String userName) {
    try {
        //①使用DataSourceUtils获取数据连接
        Connection conn =
            DataSourceUtils.getConnection(jdbcTemplate.getDataSource());
        //Connection conn = jdbcTemplate.getDataSource().getConnection();

        String sql = "UPDATE t_user SET last_logon_time=? WHERE user_name =?";
        jdbcTemplate.update(sql, System.currentTimeMillis(), userName);
        Thread.sleep(1000);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

重新运行代码，得到如下执行结果：

```

连接数[active:idle]-[0:0]
连接数[active:idle]-[1:0]
连接数[active:idle]-[0:1]
连接数[active:idle]-[1:0]
连接数[active:idle]-[0:1]

```

对照代码清单 12-23 的输出日志，可以看到已经没有连接泄露现象了。一个执行线程在运行 `JdbcUserService#logon()` 方法时只占用一个连接，而且方法执行完毕后，该连接马上释放。这说明通过 `DataSourceUtils.getConnection()` 方法确实获取了方法所在事务上下文绑定的那个连接，而不是像原来那样从数据源中获取一个新的连接。

12.7.4 无事务环境下通过 DataSourceUtils 获取数据连接

然而，是否使用 `DataSourceUtils` 获取数据连接就可以高枕无忧了呢？理想很丰满，现实很骨感：如果 `DataSourceUtils` 在没有事务上下文的方法中使用 `getConnection()` 方法获取连接，那么依然会造成数据连接泄露。

保持代码清单 12-23 的代码不变，将 Spring 配置文件（代码清单 12-20）中①处的 Spring AOP 事务增强配置的代码注释掉，让代码运行在无事务环境下。重新运行代码清单 12-23 的代码，将得到如下输出日志：

```

连接数[active:idle]-[0:0]
连接数[active:idle]-[1:1]
连接数[active:idle]-[1:1]
连接数[active:idle]-[2:1]
连接数[active:idle]-[2:1]

```

下面通过表 12-4 对数据源连接的占用和泄露情况进行描述。

表 12-4 执行过程中数据源连接占用情况

时间	执行线程 1	执行线程 2	数据源连接		
			active	idle	leak
T0	未启动	未启动	0	0	0
T1	正在执行方法	未启动	1	1	0
T2	执行完毕	未启动	1	1	1
T3	执行完毕	正在执行方法	2	1	1
T4	执行完毕	执行完毕	2	1	2

仔细对照表 12-3 的执行过程，我们发现，在 T1 时刻，有事务上下文时的 active 为 2，idle 为 0，而此时由于没有事务管理，则 active 为 1，而 idle 也为 1。这说明有事务上下文时，需要等到整个事务方法（logon()）返回后，事务上下文绑定的连接才会被释放。但在没有事务上下文时，logon()方法调用 JdbcTemplate 执行完数据操作后，会立即释放连接。

在 T2 时刻执行线程完成 logon()方法的调用后，有一个连接没有被释放（active），所以发生了连接泄露。到 T4 时刻，两个执行线程都完成了 logon()方法的调用，但是出现了两个未释放的连接。

要堵上这个连接泄露的漏洞，需要对 logon()方法进行如下改造，如代码清单 12-24 所示。

代码清单 12-24 JdbcUserService

```
package com.smart.connleak;
...
@Service("jdbcUserService")
public class JdbcUserService {
    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    @Transactional
    public void logon(String userName) {
        Connection conn = null;
        try {
            conn = DataSourceUtils.getConnection(jdbcTemplate.getDataSource());
            String sql = "UPDATE t_user SET last_logon_time=? WHERE user_name =?";
            jdbcTemplate.update(sql, System.currentTimeMillis(), userName);
            Thread.sleep(1000);
            //①
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            //②显式使用DataSourceUtils释放连接
            DataSourceUtils.releaseConnection(conn, jdbcTemplate.getDataSource());
        }
    }
}
```

```

    }
}

```

在②处显式调用 `DataSourceUtils.releaseConnection()` 方法释放获取的连接。需要特别指出的是，一定不能在①处释放连接。因为如果 `login()` 方法在获取连接后，①处代码前这段代码执行时发生异常，则①处释放连接的动作将得不到执行。这将是一个非常具有隐蔽性的连接泄露的隐患点。

12.7.5 JdbcTemplate 如何做到对连接泄露的免疫

分析 `JdbcTemplate` 的代码，可以清楚地看到它开放的每个数据操作方法：首先使用 `DataSourceUtils` 获取连接，然后在方法返回之前使用 `DataSourceUtils` 释放连接。

下面看一下 `JdbcTemplate` 最核心的数据操作方法 `execute()`，如代码清单 12-25 所示。

代码清单 12-25 `JdbcTemplate#execute()`

```

public <T> T execute(StatementCallback<T> action) throws DataAccessException {

    //①首先根据DataSourceUtils获取数据连接
    Connection con = DataSourceUtils.getConnection(getDataSource());
    Statement stmt = null;
    try {
        Connection conToUse = con;
        ...
        handleWarnings(stmt);
        return result;
    }
    catch (SQLException ex) {
        JdbcUtils.closeStatement(stmt);
        stmt = null;

        //②发生异常时，使用DataSourceUtils释放数据连接
        DataSourceUtils.releaseConnection(con, getDataSource());
        con = null;
        throw getExceptionTranslator().translate(
            "StatementCallback", getSql(action), ex);
    }
    finally {
        JdbcUtils.closeStatement(stmt);

        //③最后再使用DataSourceUtils释放数据连接
        DataSourceUtils.releaseConnection(con, getDataSource());
    }
}

```

在①处通过 `DataSourceUtils.getConnection()` 方法获取连接，在②和③处通过 `DataSourceUtils.releaseConnection()` 方法释放连接。所有 `JdbcTemplate` 开放的数据访问 API 最终都是直接或间接由 `execute(StatementCallback<T> action)` 方法执行数据访问操作的，因此这个方法代表了 `JdbcTemplate` 数据操作的最终实现方式。

正是因为 JdbcTemplate 严谨的获取连接及释放连接的模式化流程保证了 JdbcTemplate 对数据连接泄露问题的免疫性。所以,如有可能,应尽量使用 JdbcTemplate、HibernateTemplate 等模板进行数据访问操作,避免直接获取数据连接的操作。

12.7.6 使用 TransactionAwareDataSourceProxy

如果不得已要显式获取数据连接,除了可以使用 DataSourceUtils 获取事务上下文绑定的连接,还可以通过 TransactionAwareDataSourceProxy 对数据源进行代理。数据源对象被代理后就具有了事务上下文感知的能力,通过代理数据源的 getConnection()方法获取连接和使用 DataSourceUtils.getConnection()方法获取连接的效果是一样的。

下面是使用 TransactionAwareDataSourceProxy 对数据源进行代理的配置,如代码清单 12-26 所示。

代码清单 12-26 applicatonContext.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"
       ...
       http://www.springframework.org/schema/tx
       http://www.springframework.org/schema/tx/spring-tx-4.0.xsd">
  <context:component-scan base-package="com.smart.connleak"/>
  <context:property-placeholder location="classpath:jdbc.properties"/>

  <!--①未被代理的数据源 -->
  <bean id="originDataSource" 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}"/>

  <!--②对数据源进行代理,使数据源具有事务上下文感知性 -->
  <bean id="dataSource"
        class="org.springframework.jdbc.datasource.TransactionAwareDataSourceProxy"
        p:targetDataSource-ref="originDataSource" />

  <bean id="jdbcTemplate"
        class="org.springframework.jdbc.core.JdbcTemplate"
        p:dataSource-ref="dataSource"/>

  <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
        p:dataSource-ref="dataSource"/>
  <tx:annotation-driven/>
</beans>
```

对数据源进行代理后,就可以通过数据源代理对象的 getConnection()方法获取事务上下文中绑定的数据连接了。因此,如果数据源已经进行了 TransactionAwareDataSourceProxy

代理，而且方法存在事务上下文，那么代码清单 12-19 的代码也不会生产连接泄露的问题。

12.7.7 其他数据访问技术的等价类

理解了 Spring JDBC 的数据连接泄露问题，其中的道理可以平滑地推广到其他框架中。Spring 为每个数据访问技术框架都提供了一个获取事务上下文绑定的数据连接（或其衍生品）的工具类和数据源（或其衍生品）的代理类。

表 12-5 列出了不同数据访问技术框架对应 DataSourceUtils 的等价类。

表 12-5 不同数据访问技术框架DataSourceUtils的等价类

数据访问技术框架	连接（或衍生品）获取工具类
Spring JDBC	org.springframework.jdbc.datasource.DataSourceUtils
Hibernate	org.springframework.orm.hibernateX.SessionFactoryUtils
MyBatis	org.springframework.jdbc.datasource.DataSourceUtils
JPA	org.springframework.orm.jpa.EntityManagerFactoryUtils
JDO	org.springframework.orm.jdo.PersistenceManagerFactoryUtils

表 12-6 列出了不同数据访问技术框架下 TransactionAwareDataSourceProxy 的等价类。

表 12-6 不同数据访问技术框架下TransactionAwareDataSourceProxy的等价类

数据访问技术框架	连接（或衍生品）获取工具类
Spring JDBC	org.springframework.jdbc.datasource.TransactionAwareDataSourceProxy
Hibernate	org.springframework.orm.hibernateX.LocalSessionFactoryBean
MyBatis	org.springframework.jdbc.datasource.TransactionAwareDataSourceProxy
JPA	无
JDO	org.springframework.orm.jdo.TransactionAwarePersistenceManagerFactoryProxy

12.8 小结

Spring 声明式事务是 Spring 最核心、最常用的功能。由于 Spring 通过 IoC 和 AOP 非常透明地实现了声明式事务的功能，所以一般的开发者基本上无须了解 Spring 声明式事务的内部细节，仅需懂得如何配置即可。

但在实际应用开发过程中，Spring 的这种透明的高阶封装在带来便利的同时，也带来了迷惑。就像通过流言传播的消息，最终听众已经不清楚事情的真相，而这对于应用开发来说是很危险的。本章剖析了实际应用中给开发者造成迷惑的各种难点，通过分析 Spring 事务管理的内部运作机制将真相还原出来。

通过本章的学习，我们知道了以下真相。

- ❑ 在没有事务管理的情况下，DAO 照样可以顺利地进行数据操作。
- ❑ 将应用分成 Web、Service 及 DAO 层只是一种参考的开发模式，并非是事务管理工作的前提条件。
- ❑ Spring 通过事务传播机制可以很好地应对事务方法嵌套调用的情况，开发者无须为了事务管理而刻意改变服务方法的设计。
- ❑ 由于单实例的对象不存在线程安全问题，所以经过事务管理增强的单实例 Bean 可以很好地工作在线程安全环境下。
- ❑ 混合使用多个数据访问技术框架的最佳组合是一个 ORM 技术框架（如 Hibernate 或 JPA 等）+ 一个 JDBC 技术框架（如 Spring JDBC 或 MyBatis）。直接使用 ORM 技术框架对应的事务管理器就可以了，但必须考虑 ORM 缓存同步的问题。
- ❑ Spring AOP 事务增强有两个方案：其一为基于接口的动态代理；其二为基于 CGLib 动态生成子类的代理。由于 Java 语法的特性，有些特殊方法不能被 Spring AOP 代理，所以也就无法享受 AOP 织入带来的事务增强；
- ❑ 在使用 Spring JDBC 时如果直接获取 Connection，则可能会造成连接泄露。为了降低连接泄露的可能性，尽量使用 DataSourceUtils 获取数据连接。也可以对数据源进行代理，以便使数据源拥有感知事务上下文的能力。
- ❑ 可以将 Spring JDBC 防止连接泄露的解决方案平滑地应用到其他数据访问技术框架中。

第 13 章

使用 Spring JDBC 访问数据库

Spring JDBC 是 Spring 所提供的持久层技术。它的主要目的是降低使用 JDBC API 的门槛，以一种更直接、更简洁的方式使用 JDBC API。在 Spring JDBC 里，仅需做那些与业务相关的 DML 操作的事，而将资源获取、Statement 创建、资源释放及异常处理等繁杂而乏味的工作交给 Spring JDBC。

虽然 ORM 的框架已经成熟丰富，但 JDBC 的灵活、直接的特性，依然让它拥有自己的用武之地。如在完全依赖查询模型动态产生查询语句的综合查询系统中，Hibernate、MyBatis、JPA 等框架都无法使用，这里 JDBC 是唯一的选择。

本章主要内容：

- ◆ 使用 JdbcTemplate 模板类进行 CRUD 数据操作
- ◆ BLOB 及 CLOB 类型数据的操作
- ◆ 支持命名参数绑定 NamedParameterJdbcTemplate 模拟类的使用
- ◆ 关于主键产生、获取的知识

本章亮点：

- ◆ 详细分析并讨论了主键的产生、获取方式，并比较了它们的优劣
- ◆ 对 LOB 数据操作进行详细分析，这些讨论在其他 ORM 框架中具有普适性
- ◆ 以 MySQL 和 Oracle 两个颇具代表性的数据库为基础进行实例设计

13.1 使用 Spring JDBC

但凡 Java 开发者都有过直接使用 JDBC 编写数据库程序的经历，由于 JDBC API 过于底层，开发者不但需要编写数据操作代码，还需要编写获取 JDBC 连接、处理异常、释放资源等代码。即使一个再简单不过的数据库操作，也需要至少十几行代码。Spring

JDBC 通过模板和回调机制大大降低了使用 JDBC 的复杂度, 借由 JdbcTemplate 的帮助, 仅需要编写那些“必不可少”的代码就可以进行数据库操作。

13.1.1 JdbcTemplate 小试牛刀

几乎可以用 JdbcTemplate 完成任何数据访问操作, 并充分享受 JdbcTemplate 所带来的简洁。下面是使用 JdbcTemplate 创建一张表的例子, 如代码清单 13-1 所示。

代码清单 13-1 使用 JdbcTemplate 创建一张表

```
DriverManagerDataSource ds = new DriverManagerDataSource(); ①
ds.setDriverClassName("com.mysql.jdbc.Driver");
ds.setUrl("jdbc:mysql://localhost:3309/sampledbs");
ds.setUsername("root");
ds.setPassword("1234");

JdbcTemplate jdbcTemplate = new JdbcTemplate(); ②
jdbcTemplate.setDataSource(ds);

String sql = "create table t_user(user_id int primary key,user_name varchar(60));" ③
jdbcTemplate.execute(sql);
```

创建一个数据源

生成一个 JdbcTemplate 实例

创建一张表

在代码清单 13-1 中, 首先使用 DriverManagerDataSource() 方法创建了一个数据源, 接着创建了一个 JdbcTemplate 对象, 最后使用该对象执行 SQL 语句。运行以上代码, 将在 MySQL 数据库中创建一张 t_user 表。

通过以上实例可以感受到 JdbcTemplate 所开创的全新的编程风格, 代码清单 13-1 中粗体显示的部分就是移除闲杂代码后简洁到极致的代码。当然, 从代码行上看, 代码清单 13-1 依旧拥有 9 行代码, 似乎并没有做到真正的简洁。这是因为为了保证实例的完整性, 特意将数据源创建和模板实例创建的代码都列在这个例子中, 但在实际应用中, 用户一般不会在 DAO 中做这些事情。由于 JdbcTemplate 是线程安全的, 因而所有的 DAO 都可以共享同一个 JdbcTemplate 实例, 这样①和②部分的代码就可以从 DAO 中移除了, 转而在 Spring 配置文件中统一定义即可。



提示

本章后面内容所涉及的数据库表的脚本位于本书配套网盘中的 chapter13/schema 目录下, 提供了 MySQL 和 Oracle 两个数据库版本的 SQL 脚本。

13.1.2 在 DAO 中使用 JdbcTemplate

一般情况下都是在 DAO 类中使用 JdbcTemplate, JdbcTemplate 在 XML 配置文件中配置好后, 直接在 DAO 中注入即可。

```
package com.smart.dao;
```

```
...
```

```
//①声明一个 DAO
@Repository
public class ForumDao {
    private JdbcTemplate jdbcTemplate;

    @Autowired //②注入JdbcTemplate实例
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    public void initDb() {

        //注意：在实际项目中，如果SQL不是动态组装的，则应将其定义成一个类级的静态final常量，
        //此处仅为方便代码阅读，所以放在方法内部定义，下同
        String sql = "create table t_user(user_id int primary key,user_name varchar(60))";
        jdbcTemplate.execute(sql);
    }
}
```

按照相同的方式，就可以方便地创建其他的 DAO 类，如 TopicDao、PostDao 等。在 Spring 配置文件中定义 JdbcTemplate 并注入每个 DAO 中。

```
<!--①扫描包以注册注册声明的 Bean -->
<context:component-scan base-package="com.smart"/>

<!--②配置数据源 -->
<context:property-placeholder location="classpath:jdbc.properties" />
<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声明 JdbcTemplate Bean -->
<bean id="jdbcTemplate"
    class="org.springframework.jdbc.core.JdbcTemplate"
    p:dataSource-ref="dataSource"/>
```

在 Spring 配置文件中配置 DAO 一般分为 4 个步骤。

- (1) 定义 DataSource。
- (2) 定义 JdbcTemplate。

JdbcTemplate 拥有几个可用于控制底层 JDBC API 的属性，合理地设置这些属性可以获得良好的程序性能。

- queryTimeout: 设置 JdbcTemplate 所创建的 Statement 查询数据时的最大超时时间。默认为 0，表示使用底层 JDBC 驱动程序的默认设置。
- fetchSize: 设置底层的 ResultSet 每次从数据库返回的行数。该属性对程序的性能影响很大，如果设置过大，因为一次性载入的数据都放到内存中，所以内存的消耗很大；反之，如果设置过小，从数据库读取的次数将增大，也会影响性能。默认值为 0，表示使用底层 JDBC 驱动程序的默认设置。Oracle 驱动程序的 fetchSize 的默认值为 10。

- ❑ `maxRows`: 设置底层的 `ResultSet` 从数据库返回的最大行数。默认值为 0, 表示使用底层 JDBC 驱动程序的默认设置。
 - ❑ `ignoreWarnings`: 是否忽略 SQL 的警告信息。默认为 `true`, 即所有的警告信息都被记录到日志中; 如果设置为 `false`, 则 `JdbcTemplate` 将抛出 `SQLWarningException`。
- (3) 声明一个抽象的 `<bean>`, 以便所有 DAO 复用配置 `JdbcTemplate` 属性的配置。
- (4) 配置具体的 DAO。



实战经验

Spring 几乎为所有的模板类都提供了相应的支持类, 与 `JdbcTemplate` 对应的支持类为 `JdbcDaoSupport`。`JdbcDaoSupport` 内部定义了 `JdbcTemplate` 的成员变量, 开发者可以通过扩展 `JdbcDaoSupport` 定义自己的 DAO, 这也是早期 Spring 开发者常用的做法。但是随着 Bean 的注解配置逐渐成为主流配置方式, 这种做法就显得不合时宜了, 因为直接继承 `JdbcDaoSupport` 无法对 `JdbcTemplate` 成员变量应用 `@Autowired` 注解。所以我们推荐的做法是自己定义一个 `BaseDao`, 在 `BaseDao` 中定义 `JdbcTemplate` 成员变量, 并在其 setter 方法上标注 `@Autowired` 注解。

在实际项目中, 还会在 `BaseDao` 中定义一些通用的功能, 如声明 `JdbcTemplate`、分页查询等。

13.2 基本的数据操作

数据库的增、删、查、改及存储过程调用是最常见的数据库操作, `JdbcTemplate` 提供了众多方法, 通过 `JdbcTemplate` 可以用简单的方法完成这些数据操作。

13.2.1 更改数据

`JdbcTemplate` 提供了若干个 `update()` 方法, 允许对数据表记录进行更改和删除操作。下面的 `ForumDao` 定义了一个新增论坛版块的方法, 如代码清单 13-2 所示。

代码清单 13-2 更新数据

```
package com.smart.dao;
...
@Repository
public class ForumDao {
    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
}
```

```

    }

    public void addForum(Forum forum) {
        String sql = "INSERT INTO t_forum(forum_name,forum_desc) VALUES(?,?)"; ①
        Object[] params = new Object[] {forum.getForumName(), forum.getForumDesc()}; ②
        jdbcTemplate.update(sql, params); ③
    }
    ...
}

```

由于 JdbcTemplate 在内部通过 PreparedStatement 执行 SQL 语句，所以可以使用绑定参数的 SQL 语句，如①处所示，每个“?”占位符可接受一个参数。在②处定义了用于填充占位符的参数数组，在③处直接通过 JdbcTemplate 的 `int update(String sql, Object[] args)` 方法进行表数据的更新。

JdbcTemplate 已经把原来复杂的数据更新操作简化为定义 SQL 语句、准备参数、调用 `update()` 这 3 个至简的步骤，这不由得让我们想起了《登徒子好色赋》中的东家之子——增之一分则太长，减之一分则太短。



实战经验

尽量使用可绑定参数的 SQL 语句，以便数据库可以复用 SQL 的执行计划，提高数据库的执行效率。此外，应尽量在 DAO 中使用类级别的静态常量 (`final static`) 定义 SQL 字符串，不应在方法内部声明 SQL 字符串变量，以提高 JVM 的内存使用效率。

在通过 `update(String sql, Object[] args)` 方法为 SQL 语句的占位符绑定参数时，并没有显式指定对应字段的数据类型，此时，Spring 直接让 PreparedStatement 根据参数的类型进行“猜测”。在上例中，由于 `t_forum` 的 `forum_name` 和 `forum_desc` 都是 `varchar` 类型的，所以并不会发生错误。但一种更好的做法是使用 `int update(String sql, Object[] args, int[] argTypes)` 显式指定每个占位符所对应的字段数据类型，这样就可以保证类型安全，当参数值为 `null` 时，这种形式提供了更好的支持。可以对代码清单 13-2 进行以下调整：

```

package com.smart.dao;
import java.sql.Types; ① ← 使用该类中的常量
                           属性定义参数类型
...
@Repository
public class ForumDao {
    ...
    public void addForum(Forum forum) {
        ...
        jdbcTemplate.update(sql, params, new int[] {Types.VARCHAR, Types.VARCHAR});
    }
}

```

除了以上两个 `update()` 方法外，JdbcTemplate 还提供了以下几个功能相似的重载方法。

- `int update(String sql)`: 为不带占位符的 SQL 语句所提供的便利方法。
- `int update(String sql, Object... args)`: 使用不定参数的方法，和 `update(String sql, Object[] args)` 相似。

- `int update(String sql, PreparedStatementSetter pss)`: `PreparedStatementSetter` 是一个回调接口，它定义了一个 `void setValues(PreparedStatement ps)` 接口方法。`JdbcTemplate` 使用 SQL 语句创建出 `PreparedStatement` 实例后，将调用该回调接口执行绑定参数的操作。可以按如下方式使用该回调接口：

```
//①为了 forum 可以在内部类中使用，必须声明为 final
public void addForum(final Forum forum) {
    ...
    //②通过匿名内部类定义回调实例
    jdbcTemplate.update(sql, new PreparedStatementSetter() {
        public void setValues(PreparedStatement ps) throws SQLException {
            ps.setString(1, forum.getForumName());
            ps.setString(2, forum.getForumDesc());
        }
    });
}
```



提示

`PreparedStatement` 绑定参数时，参数索引从 1 开始而非从 0 开始，第一个参数索引为 1，第二个参数索引为 2，以此类推。

- `int update(PreparedStatementCreator psc)`: `PreparedStatementCreator` 也是一个回调接口，它负责创建一个 `PreparedStatement` 实例。该回调接口定义了一个 `PreparedStatement createPreparedStatement(Connection con)` 方法。同样可以使用匿名内部类定义一个 `PreparedStatementCreator` 实例，如下：

```
public void addForum(final Forum forum) {
    //①由于 sql 变量需要在内部类中使用，所以将其声明为 final
    final String sql = "INSERT INTO t_forum(forum_name,forum_desc) VALUES(?,?)";

    jdbcTemplate.update(new PreparedStatementCreator() {
        public PreparedStatement createPreparedStatement(Connection conn)
            throws SQLException {
            PreparedStatement ps = conn.prepareStatement(sql);
            ps.setString(1, forum.getForumName());
            ps.setString(2, forum.getForumDesc());
            return ps;
        }
    });
}
```

- `protected int update(PreparedStatementCreator psc, PreparedStatementSetter pss)`: 联合使用 `PreparedStatementCreator` 和 `PreparedStatementSetter` 回调。

需要指出的是，在实际使用中，应优先考虑使用不带回调接口的 `JdbcTemplate` 方法。首先，回调使代码显得臃肿；其次，回调并不能带来额外的好处。当使用 `update(String sql, Object[] params)` 等简洁版的方法时，`JdbcTemplate` 会在内部自动创建这些回调实例，我们没有必要越俎代庖。

13.2.2 返回数据库的表自增主键值

有些开发者喜欢使用数据库自增字段作为表主键，即主键值不在应用层产生，而是在新增记录时由数据库产生。这样，应用层在保存对象前并不知道对象主键值，而必须在保存数据后才能从数据库中返回主键值。在很多情况下，我们需要获取新对象持久化后的主键值。在 Hibernate、JPA 等 ORM 框架中，新对象在持久化之后，主键值会自动绑定到对象上，给程序的开发带来了方便。

在 JDBC 3.0 规范中，当新增记录时，允许将数据库自动产生的主键值绑定到 Statement 或 PreparedStatement 中。在使用 Statement 时，可以通过以下方法绑定主键值：

```
int executeUpdate(String sql,int autoGeneratedKeys)
```

也可以通过 Connection 创建绑定自增主键值的 PreparedStatement，如下：

```
PreparedStatement prepareStatement(String sql, int autoGeneratedKeys)
```

当 autoGeneratedKeys 参数设置为 Statement.RETURN_GENERATED_KEYS 时，即可绑定数据库产生的主键值；当设置为 Statement.NO_GENERATED_KEYS 时，不绑定主键值。下面的代码演示了 Statement 绑定并获取数据库产生的主键值的过程：

```
Statement stmt = conn.createStatement();
String sql = "INSERT INTO t_topic(topic_title,user_id) VALUES ('测试主题','123')";

//①指定绑定表自增主键值
stmt.executeUpdate(sql,Statement.RETURN_GENERATED_KEYS);
ResultSet rs = stmt.getGeneratedKeys();
if ( rs.next() ) {
    int key = rs.getInt();//②获取对应的表自增主键值
}
```

Spring 利用这一技术，提供了一个可以返回新增记录对应主键值的方法，如下：

```
int update(PreparedStatementCreator psc, KeyHolder generatedKeyHolder)
```

org.springframework.jdbc.support.KeyHolder 是一个回调接口，Spring 使用它保存新增记录对应的主键，该接口的接口方法描述如下。

- ❑ Number getKey() throws InvalidDataAccessApiUsageException: 当仅插入一行数据，主键不是复合键而是数字类型时，通过该方法可以直接返回新的主键值。如果是复合主键，或者有多个主键返回时，则该方法抛出 InvalidDataAccessApiUsageException。该方法是最常用的方法，因为在一般情况下，一次仅插入一条数据，并且主键字段类型为数字类型。
- ❑ Map<String,Object> getKeys() throws InvalidDataAccessApiUsageException: 如果是复合主键，则列名和列值构成 Map 中的一个 Entry。如果返回的是多个主键，则该方法抛出 InvalidDataAccessApiUsageException 异常。
- ❑ List<Map<String,Object>> getKeyList(): 如果返回多个主键，即 PreparedStatement 新增多条记录，则每个主键对应一个 Map，多个 Map 构成一个 List。

Spring 为 KeyHolder 接口指代了一个通用的实现类 GeneratedKeyHolder，该类返回

新增记录时的自增长主键值。假设希望在新增论坛版块对象后，将主键值加载到对象中，则可以按以下代码进行调整：

```
public void addForum(final Forum forum) {
    final String sql = "INSERT INTO t_forum(forum_name,forum_desc) VALUES(?,?)";

    KeyHolder keyHolder = new GeneratedKeyHolder(); //①创建一个主键持有者
    jdbcTemplate.update(new PreparedStatementCreator() {
        public PreparedStatement createPreparedStatement(Connection conn)
            throws SQLException {
            PreparedStatement ps = conn.prepareStatement(sql);
            ps.setString(1, forum.getForumName());
            ps.setString(2, forum.getForumDesc());
            return ps;
        }
    }, keyHolder);
    forum.setForumId(keyHolder.getKey().intValue()); //②从主键持有者中获取主键
}
```

这样，在调用 addForum(final Forum forum)新增 forum 领域对象后，forum 将拥有对应的主键值，方便后继的使用。

在 JDBC 3.0 之前的版本中，PreparedStatement 不能绑定主键，如果采用表自增键（如 MySQL 的 auto increment 或 SQL Server 的 identity），则将给获取正确的主键值带来挑战，因为用户必须在插入数据后立即执行另一条获取新增主键的查询语句。表 13-1 给出了不同数据库获取最新自增主键值的查询语句。

表 13-1 不同数据库获取新增加的主键值

数 据 库	获取新增主键值的查询语句
DB2	IDENTITY_VAL_LOCAL()
Informix	SELECT dbinfo('sqlca.sqlerrd1') FROM <TABLE>
Sybase	SELECT @@IDENTITY
SQLServer	SELECT SCOPE_IDENTITY()或 SELECT @@IDENTITY
MySQL	SELECT LAST_INSERT_ID()
HSQldb	CALL IDENTITY()
Cloudscape	IDENTITY_VAL_LOCAL()
Derby	IDENTITY_VAL_LOCAL()
PostgreSQL	SELECT nextval('<TABLE>_SEQ')

如果数据库的并发率很高，如在插入记录后执行查询主键前，数据库又执行了若干条插入记录的 SQL 语句，这时，通过表 13-1 返回的主键值就是最后一条插入语句的主键值，而非我们希望的主键值。所以使用查询语句获取表自增键值是不安全的，这也是为什么有些数据库（如 Oracle、Firebird）故意不提供自增键，而只提供序列的原因，序列强制要求用户在新增记录前先获取主键值。Oracle 通过 SELECT <SEQUENCE_NAME>.nextval FROM DUAL 获取序列的下一个值，而 Firebird 通过 SELECT GEN_ID(<SEQUENCE_NAME> 1) FROM RDB\$DATABASE 获取序列的下一个值。13.4.1 节将讲解应用层自增键的相关知识。



实战经验

在实际开发中,我们并不太建议使用表自增键,因为这种方式会让开发更加复杂且降低程序移植性,在应用层中创建主键才是主流的方式,可以使用 UUID 或者通过一个编码引擎获取主键值。我们建议尽量少用数据库的自增键、外键、触发器、存储过程、数据库函数等高级功能,让数据库只负责数据存储和查询(在进行表设计时只应关注表空间、索引、非空限制及唯一值限制等和存储查询相关的特性),不要让它负责业务逻辑,业务逻辑处理应在应用层中进行。

13.2.3 批量更改数据

如果需要一次性插入或更新多条记录,当然可以简单地通过多次调用 `update()` 方法完成任务,但这不是最好的实现方案。更好的选择是使用 `JdbcTemplate` 批量数据更改的方法。一般情况下,后者拥有更好的性能,因为更新的数据将被批量发送到数据库中,它减少了对数据库的访问次数。下面先来认识一下 `JdbcTemplate` 的两个批量数据操作方法。

- ❑ `public int[] batchUpdate(String[] sql)`: 多条 SQL 语句组成一个数组(这些 SQL 语句不带参数),该方法以批量方式执行这些 SQL 语句。Spring 在内部使用 JDBC 提供的批量更新 API 完成操作。如果底层的 Jdbc Driver 不支持批量更新操作,那么 Spring 将采用逐条更新的方式模拟批量更新。
- ❑ `int[] batchUpdate(String sql, BatchPreparedStatementSetter pss)`: 使用该方法对于同一结构的带参 SQL 语句多次进行数据更新操作。通过 `BatchPreparedStatementSetter` 回调接口进行批量参数的绑定工作。`BatchPreparedStatementSetter` 定义了两个方法。
 - `int getBatchSize()`: 指定本批次的大小。
 - `void setValues(PreparedStatement ps, int i)`: 为给定的 `PreparedStatement` 设置参数。

假设需要一次性新增多个论坛版块,则可以通过编写如下方法来满足要求,如代码清单 13-3 所示。

代码清单 13-3 批量插入操作

```
public void addForums(final List<Forum> forums) {
    final String sql = "INSERT INTO t_forum(forum_name,forum_desc) VALUES(?,?)";

    jdbcTemplate.batchUpdate(sql, new BatchPreparedStatementSetter() {

        //①指定该批的记录数
        public int getBatchSize() {
            return forums.size();
        }

        //②绑定插入的参数
```

```

    public void setValues(PreparedStatement ps, int index)
        throws SQLException {
        Forum forum = forums.get(index);
        ps.setString(1, forum.getForumName());
        ps.setString(2, forum.getForumDesc());
    }
});
}

```

有的开发者认为 BatchPreparedStatementSetter 内部会自动将需要更改的数据分多批传送到数据库中, 而每批的大小通过 getBatchSize() 方法指定。其实, 这样理解是错误的。实际上, BatchPreparedStatementSetter 是一次性批量提交数据的, 而不会分批提交, getBatchSize() 是整批的大小。所以, 如果希望将一个 List 中的数据通过 BatchPreparedStatementSetter 批量更新到数据库中, getBatchSize() 就应该设置为 List 的大小。如果 List 非常大, 希望分多次批量提交, 则可以分段读取这个大的 List 并暂存到一个小的 List 中, 再将这个小的 List 通过 BatchPreparedStatementSetter 批量保存到数据库中。

13.2.4 查询数据

请回想一下使用原生 JDBC 查询数据时有哪些操作是必不可少的——是的, 从结果集中获取数据是每个查询功能不可或缺的操作。在 Spring JDBC 中, 仅需要指定 SQL 查询语句并定义好如何从结果集中返回数据就可以了。

1. 使用 RowCallbackHandler 处理结果集

Spring 提供了 org.springframework.jdbc.core.RowCallbackHandler 回调接口, 通过该接口可以定义如何从结果集中获取数据。RowCallbackHandler 接口很简单, 仅有一个方法: void processRow(ResultSet rs) throws SQLException。Spring 会遍历结果集, 对结果集中的每一行调用 RowCallbackHandler 回调接口处理数据。所以无须调用 ResultSet 的 next() 方法, 而只需定义好如何获取结果集数据的逻辑就可以了。

ForumDao 需要提供一个根据 forumId 获取 Forum 对象的方法, 下面是该方法的具体实现, 如代码清单 13-4 所示。

代码清单 13-4 单条数据结果集的处理

```

public Forum getForum(final int forumId) {
    String sql = "SELECT forum_name, forum_desc FROM t_forum WHERE forum_id=?";
    final Forum forum = new Forum();

    // ①将结果集数据行中的数据抽取到 forum 对象中
    jdbcTemplate.query(sql, new Object[]{forumId}, new RowCallbackHandler() {
        public void processRow(ResultSet rs) throws SQLException {
            forum.setForumId(forumId);
            forum.setForumName(rs.getString("forum_name"));
            forum.setForumDesc(rs.getString("forum_desc"));
        }
    });

    return forum;
}

```

虽然 `processRow(ResultSet rs)` 方法会抛出 `SQLException` 异常，但我们无须关注它，因为 `JdbcTemplate` 会将 `SQLException` 转换为 Spring DAO 通用层的运行期异常。

如果需要获取多条记录，依然可以使用 `RowCallbackHandler` 完成任务，只需稍微调整一下结果集的处理逻辑就可以了。下面的代码可以通过论坛版块 ID 的范围值获取 `Forum` 对象的列表，如代码清单 13-5 所示。

代码清单 13-5 多条数据结果集的处理

```
public List<Forum> getForums(final int fromId, final int toId) {
    String sql = "SELECT forum_id, forum_name, forum_desc FROM t_forum WHERE
        forum_id between ? and ?";

    final List forums = new ArrayList ();

    jdbcTemplate.query(sql, new Object[] {fromId, toId}, new RowCallback
Handler() { ① // 将结果集中的数据映射到 List 中
        public void processRow(ResultSet rs) throws SQLException {
            Forum forum = new Forum();
            forum.setForumId(rs.getInt("forum_id"));
            forum.setForumName(rs.getString("forum_name"));
            forum.setForumDesc(rs.getString("forum_desc"));
            forums.add(forum);
        }
    });

    return forums;
}
```

当结果集中没有数据时，并不会抛出异常，只是此时 `RowCallbackHandler` 回调接口中定义的处理逻辑没有得到调用罢了。

代码清单 13-4 和代码清单 13-5 都使用了 `JdbcTemplate` 的 `void query(String sql, Object[] args, RowCallbackHandler rch)` 方法。该方法的参数类型安全的重载版本是 `void query(String sql, Object[] args, int[] argTypes, RowCallbackHandler rch)`。此外还有以下几个功能相似的重载版本：

- ☐ `void query(PreparedStatementCreator psc, RowCallbackHandler rch)`。
- ☐ `void query(String sql, PreparedStatementSetter pss, RowCallbackHandler rch)`。
- ☐ `void query(String sql, RowCallbackHandler rch)`。

2. 使用 `RowMapper<T>` 处理结果集

Spring 还提供了一个和 `RowCallbackHandler` 功能上类似的 `RowMapper<T>` 接口，也可以使用 `RowMapper<T>` 接口定义结果集映射逻辑。在结果集为多行记录时，该接口更容易使用。`RowMapper<T>` 接口只有一个接口方法：`T mapRow(ResultSet rs, int rowNum)`。

`JdbcTemplate` 提供了若干个使用 `RowMapper<T>` 查询数据的方法，这些方法的返回类型为 `List`。下面通过 `JdbcTemplate` 的 `List query(String sql, Object[] args, RowMapper<T> rowMapper)` 方法对代码清单 13-5 进行改造，如代码清单 13-6 所示。

代码清单 13-6 使用 `RowMapper` 映射多行数据

```
public List<Forum> getForums(final int fromId, final int toId) {
    String sql = "SELECT forum_id, forum_name, forum_desc FROM t_forum WHERE
        forum_id between ? and ?";
```



```

return jdbcTemplate.query(sql, new Object[]{fromId, toId}, new RowMapper
<Forum>() {
    public Forum mapRow(ResultSet rs, int index) throws SQLException {
        Forum forum = new Forum();
        forum.setForumId(rs.getInt("forum_id"));
        forum.setForumName(rs.getString("forum_name"));
        forum.setForumDesc(rs.getString("forum_desc"));
        return forum;
    }
});
}

```

粗体部分为 `RowMapper<T>` 的匿名内部类，参照代码清单 13-5，可以发现，在 `RowCallbackHandler` 的 `void processRow(ResultSet rs)` 方法中，需要将行数据映射对象 `Forum` 手工添加到外部定义的 `forums` 列表中；而在 `RowMapper<T>` 的 `T mapRow(ResultSet rs, int index)` 方法中，仅需简单地定义结果集行和对象的映射关系即可，创建 `List<T>` 对象、将行数据映射对象添加到 `List<T>` 中等操作都由 `JdbcTemplate` 代劳了。这也是为什么说 `RowMapper<T>` 更适合在多行结果集时使用的原因。

`JdbcTemplate` 使用 `RowMapper<T>` 的其他几个接口方法说明如下。

- `<T> List<T> query(String sql, Object[] args, int[] argTypes, RowMapper<T> rowMapper)`。
- `<T> List<T> query(String sql, RowMapper<T> rowMapper)`。
- `<T> List<T> query(String sql, PreparedStatementSetter pss, RowMapper<T> rowMapper)`。
- `<T> List<T> query(PreparedStatementCreator psc, RowMapper<T> rowMapper)`。

3. RowCallbackHandler 和 RowMapper<T> 的比较

从功能上来说，`RowCallbackHandler` 和 `RowMapper<T>` 并没有太大的区别，它们都用于定义结果集行的读取逻辑，将 `ResultSet` 中的数据映射到对象或 `List` 中。

Spring 宣称 `RowCallbackHandler` 接口实现类可以有状态的，而 `RowMapper<T>` 的实现类应该是无状态的。如果 `RowCallbackHandler` 接口实现类是有状态的，就不能在多个地方复用，只有无状态的实例才能在不同的地方复用。

这样讲未免过于抽象，来看一下 Spring 自身的几个接口实现类。Spring 中有两个 `RowCallbackHandler` 实现类，分别介绍如下。

- **RowCountCallbackHandler**：计算结果集行数。请看下面的代码：

```

RowCountCallbackHandler countCallback = new RowCountCallbackHandler();
jdbcTemplate.query("select * from user", countCallback);
int rowCount = countCallback.getRowCount();

```

可见，`RowCountCallbackHandler` 包含了一个记录结果集行数的“状态”。在多线程环境下，如果没有进行特殊的处理，就不能在多个地方复用 `countCallback` 实例。

- **SimpleRowCountCallbackHandler**：简单地遍历结果集，不作任何处理，在测试时使用。

Spring 也提供了几个 `RowMapper` 实现类，如 `ColumnMapRowMapper` 和 `SingleColumnRowMapper`。`ColumnMapRowMapper` 将结果集中的每一行映射为一个 `Map`，而

SingleColumnRowMapper 则将结果集中的某一列映射为一个 Object。它们都只定义了映射逻辑，而没有保持状态。

我们知道，通过 JDBC 查询返回一个 ResultSet 结果集时，JDBC 并不会一次性将所有匹配的数据都加载到 JVM 中，而是只返回一批次的数据（由 JDBC 驱动程序决定，如 Oracle 的 JDBC 驱动默认返回 10 行），当通过 ResultSet#next() 游标滚动结果集超过数据范围时，JDBC 再获取一批数据。这样以一种“批量化+串行化”的处理方式避免大结果集处理时 JVM 内存的过大开销。

当处理大结果集时，如果使用 RowMapper，那么采用的方式是将结果集中的所有数据都放到一个 List<T> 对象中，这样将会占用大量的 JVM 内存，甚至可能直接引发 OutOfMemoryException 异常。这时，可使用 RowCallbackHandler 接口，在 processRow() 接口方法内部一边获取数据一边完成处理，这样数据就不会在内存中堆积，可大大减少对 JVM 内存的占用。

举例来说，如果程序要求给所有系统用户发送一封邮件，而系统用户数量为 100 万。一种方案是采用 RowMapper，返回一个 List<User> 的集合，再通过遍历这个 List<User>，逐个发送邮件；而另一种方案是采用 RowCallbackHandler 接口，在 processRow() 接口方法内部逐行获取 User 数据后，立即调用邮件服务发送邮件。虽然这两种方案都达到了相同的目的，但第一种方案会在程序运行过程中，在 JVM 中产生一个系统用户数大小为 100 万条的 List<User>，从而导致极低的系统性能和巨大的内存开销，甚至引起系统崩溃。

采用 RowMapper 的操作方式是先获取数据，然后再处理数据；而 RowCallbackHandler 的操作方式是一边获取数据一边处理，处理完就丢弃之。因此，可以将 RowMapper 看作采用批量化数据处理策略，而 RowCallbackHandler 则采用流化处理策略。



轻松一刻

三国时期，徐庶推荐庞统去辅佐刘备。刘备见庞统貌丑，就派他当耒阳县令。庞统感觉大材小用很不爽，终日醉酒，三月不理政事。百姓状告到刘备那里，刘备立即派张飞去耒阳考察实情。张飞一进县衙，便斥问庞统：“你整天醉酒，哪能不误政事？”庞统说：“百里小县，有啥大事，何难决断！将军少坐，待我发落。”随即唤来公吏，将百余日所积公务都取来剖断。公吏抱来成堆案卷上厅，诉词被告入等，环跪阶下。庞统手中批判，口中发落，耳内听词，曲直分明，毫无差错。

13.2.5 查询单值数据

如果查询的结果集仅有一个值，如 SELECT COUNT(*) FROM t_forum 等，这时可以使用更简单的方式获取结果集的值。JdbcTemplate 为获取结果集中的单值数据提供了

3 组方法，分别用于获取 int、long 的单值，其他类型的单值则以 Object 类型返回。

1. int 类型的单值查询接口

- ☐ int queryForInt(String sql)。
- ☐ int queryForInt(String sql, Object... args)。
- ☐ int queryForInt(String sql, Object[] args, int[] argTypes)。

使用下面简单的代码就可以获取论坛板块的总数：

```
public int getForumNum() {
    String sql = "SELECT COUNT(*) FROM t_forum";
    return getJdbcTemplate().queryForInt(sql);
}
```

值得注意的是，如果返回空结果集，则将抛出 EmptyResultDataAccessException 异常；如果返回的结果集多于一行，则会抛出 IncorrectResultSizeDataAccessException 异常。

2. long 类型的单值查询接口

- ☐ long queryForLong(String sql)。
- ☐ long queryForLong(String sql, Object... args)。
- ☐ long queryForLong(String sql, Object[] args, int[] argTypes)。

当返回的整数超过 int 的值范围时，使用这套接口方法将返回长整型数值。

3. 其他类型的单值查询接口

- ☐ <T> T queryForObject(String sql, Class<T> requiredType)。
- ☐ <T> T queryForObject(String sql, Object[] args, Class<T>requiredType)。
- ☐ <T> T queryForObject(String sql, Object[] args, int[] argTypes, Class<T> requiredType)。
- ☐ <T> T queryForObject(String sql, Object[] args, int[] argTypes, RowMapper<T> rowMapper)。
- ☐ <T> T queryForObject(String sql, Object[] args, RowMapper<T> rowMapper)。
- ☐ <T> T queryForObject(String sql, RowMapper<T> rowMapper)。

在使用带 Class<T> requiredType 参数的方法时，结果集必须仅拥有一行一列，且结果集可以被构造为 T 类型的对象。如果结果集包括多列，则需要使用带 RowMapper<T> rowMapper 参数的方法，在接口方法中手工完成对象构造。

可以通过分析用户主题的回帖率来判断用户的被关注度，被关注度越高的用户是论坛中最有引导力的用户。通过下面的方法来实现这一要求，如代码清单 13-7 所示。

代码清单 13-7 使用 RowMapper 获取单值对象

```
package com.smart.dao;
...
public class TopicDao {
    public double getReplyRate(int userId) {
        String sql = "SELECT topic_replies,topic_views FROM t_topic WHERE user_id=?";
        double rate = jdbcTemplate.queryForObject(sql, new Object[] {userId},
            new RowMapper<Double>() {
                public Double mapRow(ResultSet rs, int index)
```

```

        throws SQLException {
            int replies = rs.getInt("topic_replies");
            int views = rs.getInt("topic_views");
            if (views > 0)
                return new Double((double) replies / views);
            else
                return new Double(0.0);
        }
    });
    return rate;
};
}

```

上面的代码使用 `Object queryForObject(String sql, Object[] args, RowMapper<T> rowMapper)` 接口方法获取某一用户主题的回帖率。粗体部分定义了根据结果集数据构造 `Double` 型返回值的逻辑。在这里，我们把问题复杂化了，主要是为了演示接口方法的使用。实际上，可以通过 SQL 语句，在数据库中直接计算回帖率，如下：

```
String sql = "SELECT topic_replies/topic_views reply_rate FROM t_topic WHERE user_id=?"
```



实战经验

读者可能已经注意到，我们在编写 SQL 语句时，特意用大写形式编写 SQL 语言的关键字和函数，而用小写形式编写表名、字段名等非语义的元素。在 SQL 语句中用大小写区分语义和非语义的元素是比较好的编码习惯。为了防止编写过程中大小写的不断切换所造成的麻烦，用户也可以整句统一先按小写或大写的方式进行编写，然后再对部分代码进行大小写的转换。一般的 IDE 都提供了大小写转换的快捷键，如在 IDEA 中，用户可以通过 `Ctrl+Shift+U` 组合键对所选的代码进行大小写转换。

13.2.6 调用存储过程

`JdbcTemplate` 提供了两个调用存储过程的接口方法，分别介绍如下。

- ❑ `<T> T execute(String callString, CallableStatementCallback<T> action)`: 用户通过 `callString` 参数指定调用存储过程的 SQL 语句；第二个参数 `CallableStatementCallback<T>` 是一个回调接口，该接口只有一个方法 `T doInCallableStatement(CallableStatement cs)`，用户可以在该方法中进行输入参数绑定、输出参数注册及返回数据处理等操作。
- ❑ `<T> T execute(CallableStatementCreator csc, CallableStatementCallback<T> action)`: 该接口方法使用 `CallableStatementCreator` 创建 `CallableStatement`，`CallableStatementCreator` 定义了一个方法 `CallableStatement createCallableStatement(Connection con)`，它使用 `Connection` 实例创建 `CallableStatement` 对象。`CallableStatementCreator` 负责创建 `CallableStatement` 实例、绑定参数、注册输出参数等工作，而

CallableStatementCallback<T>负责处理存储过程的返回结果。Spring 提供了创建 CallableStatementCreator 的工厂类 CallableStatementCreatorFactory，通过该工厂类可以简化 CallableStatementCreator 的实例创建工作。

下面在 MySQL 5.0 中创建一个存储过程，并使用 JdbcTemplate 来调用该存储过程。

```
delimiter // ① ← 将语句的结束符调整为//，否则存储过程中的“;”定义了一个入参和一个出参
                        语句结束符会被错误解析
CREATE PROCEDURE P_GET_TOPIC_NUM(IN in_user_id INT,OUT out_num INT) ② ←
BEGIN
    SELECT COUNT(*) INTO out_num FROM t_topic WHERE user_id = in_user_id;
END
//
delimiter ; ③ ← 重新将语句结束符调整为“;”
```

在 TopicDao 类中，添加一个 getUserTopicNum()方法，执行 P_GET_TOPIC_NUM 存储过程，如代码清单 13-8 所示。

代码清单 13-8 调用存储过程：1

```
public int getUserTopicNum(final int userId) {
    String sql = "{call P_GET_TOPIC_NUM(?,?)}"; //①调用存储过程的SQL语句
    Integer num = jdbcTemplate.execute(sql, new CallableStatementCallback<Integer>() {
        public Integer doInCallableStatement(CallableStatement cs)
            throws SQLException, DataAccessException {
            cs.setInt(1, userId); //②绑定入参
            cs.registerOutParameter(2, Types.INTEGER); //③注册输出参数
            cs.execute();
            return cs.getInt(2); //④获取输出参数的值
        }
    });
    return num;
}
```

使用 JDBC API 调用存储过程的标准 SQL 语句的格式如下：

```
{?= call <procedure-name>[<arg1>,<arg2>, ...]}
```

或

```
{call <procedure-name> [<arg1>,<arg2>, ...]}
```

通过占位符(?)的索引号绑定入参和注册出参。在②处声明了一个入参并绑定入参的值，在③处定义了一个整型的输出参数。如果某个参数既是入参又是出参，则采用相似的方式将该参数声明为入参后，再将其声明为出参就可以了。

调用 CallableStatement#execute()方法执行存储过程，存储过程执行后，就可以通过 CallableStatement 所提供的一套 getXxx()方法获取输出参数的值。值得注意的是，输出参数的位置索引必须和占位符(?)一致。如果存储过程返回结果集，则可以通过 CallableStatement #getResultSet()方法获取结果集对象，并按传统的方式获取结果集中的数据。

如果使用<T> T execute(CallableStatementCreator csc, CallableStatementCallback<T> action)接口，则代码清单 13-8 调整为代码清单 13-9 所示。

代码清单 13-9 调用存储过程: 2

```

public int getUserTopicNum2(final int userId) {
    String sql = "{call P_GET_ TOPIC_NUM(?,?)}" ;

    //①通过 CallableStatementCreatorFactory 创建 CallableStatementCreator
    CallableStatementCreatorFactory fac = new CallableStatementCreatorFactory(sql);
    fac.addParameter(new SqlParameter("userId",Types.INTEGER)); ②
    fac.addParameter(new SqlOutParameter("topicNum",Types.INTEGER)); ③
    Map<String,Integer> paramsMap = new HashMap<String,Integer>();
    paramsMap.put("userId",userId); ④
    CallableStatementCreator csc = fac.newCallableStatementCreator (paramsMap);

    Integer num = jdbcTemplate.execute(csc,new CallableStatementCallback<Integer>(){
        public Integer doInCallableStatement(CallableStatement cs)
            throws SQLException, DataAccessException {
            cs.execute();
            return cs.getInt(2);
        }
    });
    return num;
}

```

通过 CallableStatementCreatorFactory 创建 CallableStatementCreator 时，一般按照以下顺序进行：

(1) 以一条 SQL 语句为入参创建 CallableStatementCreatorFactory 实例。

(2) 设置存储过程的入参、出参或者出入参，注意添加的顺序非常关键，必须和“？”占位符顺序一致。Spring 在 org.springframework.jdbc.core 包中定义了 SqlParameter、SqlOutParameter 和 SqlInOutParameter，分别代表入参、出参和出入参。在②处的 SqlParameter("userId",Types.INTEGER)代表一个入参，“userId”是逻辑名，以便稍后赋值，可以是任意的名称。同理，在③处定义了一个出参。在④处为②处定义的入参指定参数值。虽然不需要在后面为 SqlOutParameter 指定值，但 Spring 却没有提供一个无须指定参数名的构造函数，让人颇为不解。

(3) 通过 newCallableStatementCreator(Map<String,?> map) 方法创建一个 CallableStatementCreator 实例。

13.3 BLOB/CLOB 类型数据的操作

我们已经学习了使用 JdbcTemplate 进行 CRUD (Create Retrieve Update Delete, 增删查改) 操作的知识，本节将进一步学习一些高级的数据库操作知识，包括获取本地数据连接进行数据库相关的操作，此外还要学习如何操作 BLOB 和 CLOB 这些 LOB 数据。

13.3.1 如何获取本地数据连接

从数据源返回的数据连接对象是本地 JDBC 对象（如 `OracleConnection`、`SQLServerConnection`）的代理类，这是因为数据源需要改变数据连接原有的行为以便施加额外的控制。如在调用 `Connection#close()` 方法时，将数据连接返回到连接池中而非将其关闭。在某些情况下，我们希望得到被代理前的本地 JDBC 对象，如 `OracleConnection` 或 `OracleResultSet`，以便调用这些驱动程序厂商相关的 API 完成一些特殊的操作。

为了获取本地 JDBC 对象，Spring 在 `org.springframework.jdbc.support.nativejdbc` 包下定义了 `NativeJdbcExtractor` 接口并提供了实现类。`NativeJdbcExtractor` 接口定义了从数据源的 JDBC 对象抽取本地 JDBC 对象的方法，下面是几个重要的接口方法。

- ❑ `Connection getNativeConnection(Connection con)`：获取本地 `Connection` 对象。
- ❑ `Connection getNativeConnectionFromStatement(Statement stmt)`：获取本地 `Statement` 对象。
- ❑ `PreparedStatement getNativePreparedStatement(PreparedStatement ps)`：获取本地 `PreparedStatement` 对象。
- ❑ `ResultSet getNativeResultSet(ResultSet rs)`：获取本地 `ResultSet` 对象。
- ❑ `CallableStatement getNativeCallableStatement(CallableStatement cs)`：获取本地 `CallableStatement` 对象。

有些简单的数据源仅对 `Connection` 对象进行代理，这时可以直接使用 `SimpleNativeJdbcExtractor`。有些数据源（如 Jakarta Commons DBCP）会对所有的 JDBC 对象进行代理，这时就需要根据具体情况选择适合的抽取器实现类。表 13-2 列出了不同数据源的本地 JDBC 对象抽取器类。

表 13-2 不同数据源的本地JDBC对象抽取类

数据源类型	本地 JDBC 对象抽取类
C3P0 数据源	<code>org.springframework.jdbc.support.nativejdbc.C3P0NativeJdbcExtractor</code>
DBCP 数据源	<code>org.springframework.jdbc.support.nativejdbc.CommonsDbcpNativeJdbcExtractor</code>
JBoss 3.2.4 及以上版本的数据源	<code>org.springframework.jdbc.support.nativejdbc.JBossNativeJdbcExtractor</code>
WebLogic 8.1+及以上版本的数据源	<code>org.springframework.jdbc.support.nativejdbc.WebLogicNativeJdbcExtractor</code>
WebSphere 5.1 及以上版本的数据源	<code>org.springframework.jdbc.support.nativejdbc.WebSphereNativeJdbcExtractor</code>
ObjectWeb'的 XAPool 数据源	<code>org.springframework.jdbc.support.nativejdbc.XAPoolNativeJdbcExtractor</code>

代码清单 13-10 中的代码将从 DBCP 数据源中获取 Oracle 的本地数据连接对象。

代码清单 13-10 获取本地数据连接

```
package com.smart.dao;
...
@Repository
public class PostDao {
    private JdbcTemplate jdbcTemplate;
```



```

@Autowired
public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
    this.jdbcTemplate = jdbcTemplate;
}

public void getNativeConn(){
    try {
        Connection conn =
DataSourceUtils.getConnection(getJdbcTemplate().getDataSource());①

        conn = jdbcTemplate.getNativeJdbcExtractor().②
                        getNativeConnection(conn);
        OracleConnection oconn = (OracleConnection) conn;③
        ...
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

使用 DataSourceUtils 从模板类中获取连接

使用模板类的本地 JDBC 抽取器获取本地连接

这时可以强制进行类型转换了

在①处通过 DataSourceUtils 获取当前线程绑定的数据连接。为了使用线程上下文相关的事务，通过 DataSourceUtils 从数据源中获取连接是正确的做法。如果直接通过 dataSource 获取数据连接，则得到一个和当前线程上下文无关的新的数据连接实例。

JdbcTemplate 可以在配置时注入一个本地 JDBC 对象抽取器，要使代码清单 13-10 正确运行，则必须调整 JdbcTemplate 的配置，如下：

```

<bean id="nativeJdbcExtractor" lazy-init="true"
    class="org.springframework.jdbc.support.nativejdbc.
CommonsDbcpNativeJdbcExtractor"/>①
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource" />
    <property name="nativeJdbcExtractor" ref="nativeJdbcExtractor"/>②
</bean>

```

定义 DBCP 数据源的 JDBC 本地对象抽取器

设置抽取器

在获取 Oracle 的本地 Connection 实例后，就可以使用该对象的一些特有功能了，如使用 OracleConnection 对 LOB 对象进行操作。

13.3.2 相关的操作接口

LOB 代表大对象数据，包括 BLOB 和 CLOB 两种类型，前者用于存储大块的二进制数据，如图片数据、视频数据等（一般不宜将文件存储到数据库中，而应存储到文件服务器中）；而后者用于存储长文本数据，如论坛的帖子内容、产品的详细描述等。值得注意的是，在不同的数据库中，大对象对应的字段类型往往并不相同，如 Oracle 对应 BLOB/CLOB；MySQL 对应 BLOB/LONGTEXT；SQL Server 对应 IMAGE/TEXT。需要指出的是，有些数据库的大对象类型可以像简单类型一样访问，如 MySQL 的 LONGTEXT 的操作方式和 VARCHAR 类型一样。一般情况下，LOB 类型数据的访问方式不同于其他简单类型的数据，用户可能会以流的方式操作 LOB 类型的数据。此外，

LOB 类型数据的访问不是线程安全的，需要分配相应的数据库资源，并在操作完成后释放。另外，Oracle 非常有个性地采用非 JDBC 标准的 API 操作 LOB 数据。所有这些情况给编写 LOB 访问程序带来挑战，Spring 为此在 `org.springframework.jdbc.support.lob` 包中提供了帮助类，以便开发者轻松应对以上问题。

1. LobCreator

虽然 JDBC 定义了两个操作 LOB 类型的接口 `java.sql.Blob` 和 `java.sql.Clob`，但有些厂商的 JDBC 驱动程序并不支持这两个接口。为此，Spring 定义了一个独立于 `java.sql.Blob/Clob` 接口，以统一的方式操作各种数据库 LOB 类型数据的 `LobCreator` 接口。因为 `LobCreator` 本身只有 LOB 所对应的数据库资源，所以它不是线程安全的，一个 `LobCreator` 只能操作一个 LOB 数据。

为了方便在 `PreparedStatement` 中使用 `LobCreator`，用户可以直接使用 `JdbcTemplate#execute(String sql, AbstractLobCreatingPreparedStatementCallback lcpsc)` 方法。下面是 `LobCreator` 接口方法的简要说明。

- ❑ `void close()`: 关闭会话，并释放 LOB 资源。
- ❑ `void setBlobAsBinaryStream(PreparedStatement ps, int paramIndex, InputStream contentStream, int contentLength)`: 通过流填充 BLOB 数据。
- ❑ `void setBlobAsBytes(PreparedStatement ps, int paramIndex, byte[] content)`: 通过二进制数据填充 BLOB 数据。
- ❑ `void setClobAsAsciiStream(PreparedStatement ps, int paramIndex, InputStream asciiStream, int contentLength)`: 通过 ASCII 字符流填充 CLOB 数据。
- ❑ `void setClobAsCharacterStream(PreparedStatement ps, int paramIndex, Reader characterStream, int contentLength)`: 通过 Unicode 字符流填充 CLOB 数据。
- ❑ `void setClobAsString(PreparedStatement ps, int paramIndex, String content)`: 通过字符串填充 CLOB 数据。

2. LobHandler

`LobHandler` 接口为操作大二进制字段和大文本字段提供了统一访问接口，不管底层数据库究竟是以大对象的方式还是以一般数据类型的方式进行操作。此外，`LobHandler` 还充当 `LobCreator` 的工厂类。

Spring 提供 `LobHandler` 类主要是为了迁就 Oracle 特立独行的作风，Oracle 必须使用 `OracleLobHandler` 实现类，而其他数据库使用 `DefaultLobHandler` 就可以了。Oracle 9i 直接使用自己的 API 操作 LOB 数据，并且不允许通过 `PreparedStatement` 的 `setAsciiStream()`、`setBinaryStream()`、`setCharacterStream()` 等方法设置流数据，所以需要使用 Oracle `LobHandler` 调用 Oracle 本地的 JDBC 对象方法完成底层的操作。其他数据库可以统一使用 `DefaultLobHandler`。Oracle 10g 改正了 Oracle 9i 个性化的作风，终于“天下归一”了，所以 Oracle 10g 也可以使用 `DefaultLobHandler`。下面来看一下 `LobHandler` 接口的几个重要方法。

- ❑ `InputStream getBlobAsBinaryStream(ResultSet rs, int columnIndex)`: 从结果集中返回 `InputStream`, 通过 `InputStream` 读取 BLOB 数据。
- ❑ `byte[] getBlobAsBytes(ResultSet rs, int columnIndex)`: 以二进制数据的方式获取结果集中的 BLOB 数据。
- ❑ `InputStream getClobAsAsciiStream(ResultSet rs, int columnIndex)`: 从结果集中返回 `InputStream`, 通过 `InputStream` 以 ASCII 字符流的方式读取 BLOB 数据。
- ❑ `Reader getClobAsCharacterStream(ResultSet rs, int columnIndex)`: 从结果集中获取 Unicode 字符流 `Reader`, 并通过 `Reader` 以 Unicode 字符流的方式读取 CLOB 数据。
- ❑ `String getClobAsString(ResultSet rs, int columnIndex)`: 从结果集中以字符串的方式获取 CLOB 数据。
- ❑ `LobCreator getLobCreator()`: 生成一个会话相关的 `LobCreator` 对象。

13.3.3 插入 LOB 类型的数据

用于保存论坛帖子的 `t_post` 表拥有两个 LOB 字段, 其中 `post_text` 为 CLOB 类型, 而 `post_attach` 为 BLOB 类型。下面来编写插入一个帖子记录的代码, 如代码清单 13-11 所示。

代码清单 13-11 添加 LOB 字段数据

```
package com.smart.dao;
...
import com.smart.domain.Post;

@Repository
public class PostDao {
    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
    //①定义 LobHandler 属性
    private LobHandler lobHandler;

    @Autowired
    public void setLobHandler(LobHandler lobHandler) {
        this.lobHandler = lobHandler;
    }

    public void addPost(final Post post){
        String sql = " INSERT INTO t_post(post_id,user_id,post_text,post_attach)"
            + " VALUES(?,?,?,?)";
        jdbcTemplate.execute(sql,
            new AbstractLobCreatingPreparedStatementCallback(this.lobHandler) { //②
                protected void setValues(PreparedStatement ps,LobCreator lobCreator)
```

```

        throws SQLException {
            ps.setInt(1, 1);
            ps.setInt(2, post.getUserId());

            //③设置 CLOB 字段
            lobCreator.setClobAsString(ps, 3, post.getPostText());
            //④设置 BLOB 字段
            lobCreator.setBlobAsBytes(ps, 4, post.getPostAttach());
        }
    });
}
};
...
}

```

首先在 `PostDao` 中引入一个 `LobHandler` 属性, 如①处所示, 并通过 `Jdbc Template#execute(String sql, AbstractLobCreatingPreparedStatementCallback lcpsc)` 方法完成插入 LOB 数据的操作。然后通过匿名内部类的方式定义 `LobCreatingPreparedStatementCallback` 抽象类的子类, 其构造函数需要一个 `LobHandler` 入参, 如②处所示。最后在匿名类中实现了父类的抽象方法 `setValues(PreparedStatement ps, LobCreator lobCreator)`, 在该方法中, 通过 `lobCreator` 操作 LOB 对象, 如③和④处所示, 分别通过字符串和二进制数组填充 BLOB 和 CLOB 的数据。用户同样可以使用流的方式填充 LOB 数据, 仅需调用 `lobCreator` 相应的流填充方法即可。

需要调整 Spring 的配置文件以配合刚刚定义的 `PostDao`。假设底层数据库是 Oracle, 可以采用以下配置方式:

```

...
<bean id="nativeJdbcExtractor"
    class="org.springframework.jdbc.support.
nativejdbc.CommonsDbcpNativeJdbcExtractor"
    lazy-init="true"/>

<!--①使用设置本地 JDBC 对象抽取-->
<bean id="lobHandler" class="org.springframework.jdbc.support.lob.OracleLobHandler"
    lazy-init="true"
    p:nativeJdbcExtractor-ref="nativeJdbcExtractor"/>

```

读者可能已经注意到 `NativeJdbcExtractor` 和 `JobHandler` Bean 都设置为 `lazy-init="true"`, 这是因为 `NativeJdbcExtractor` 需要通过运行期的反射机制获取底层的 JDBC 对象, 所以需要避免让 Spring 容器启动时就实例化这两个 Bean。

`LobHandler` 需要访问本地 JDBC 对象, 因此需要通过一个 `NativeJdbcExtractor` Bean 来完成任务, 在①处为 `JobHandler` 注入了一个 `NativeJdbcExtractor`。

最后, 可以把 `lobHandler` Bean 设置给需要进行 LOB 数据访问的 `PostDao`。

只要不是 Oracle 9i 的数据库, 即 Oracle 10g 或其他数据库, 则只要简单地配置一个 `DefaultLobHandler` 就可以了, 如下:

```

<bean id="lobHandler "
class="org.springframework.jdbc.support.lob.DefaultLobHandler"
    lazy-init="true"/>

```

DefaultLobHandler 只是简单地代理标准 JDBC 的 PreparedStatement 和 ResultSet 对象,由于并不需要访问数据库驱动本地的 JDBC 对象,所以它不需要 NativeJdbcExtractor 的帮助。

用户可以通过以下代码测试 PostDao 的 addPost()方法,如代码清单 13-12 所示。

代码清单 13-12 测试PostDao的addPost()方法

```
package com.smart.dao;

...

@ContextConfiguration(locations = { "classpath:applicationContext.xml" })
@Rollback
@Transactional
public class TestPostDao extends AbstractTransactionalTestNGSpringContextTests {

    @Autowired
    private PostDao postDao;

    @Test
    public void testAddPost() throws Throwable{
        Post post = new Post();
        post.setUserId(2);

        //①获取图片资源
        ClassPathResource res = new ClassPathResource("temp.jpg");

        //②读取图片文件的数据
        byte[] mockImg = FileCopyUtils.copyToByteArray(res.getFile());
        post.setPostAttach(mockImg);
        post.setPostText("测试帖子的内容");
        postDao.addPost(post);
    }
}
```

在这里,有几个知识点需要解释一下:我们使用 TestNG 编写测试类,通过 @ContextConfiguration 加载 Spring 容器,它能够直接从 IoC 容器中装载 Bean。关于 Spring 中测试的知识,请参见第 20 章。

此外,我们使用 ClassPathResource 加载图片资源,并通过 FileCopyUtils 读取文件的数据。

13.3.4 以块数据方式读取 LOB 数据

用户可以直接以数据块的方式读取 LOB 数据:以 String 读取 CLOB 字段的数据,而以 byte[] 读取 BLOB 字段的数据。在 PostDao 中添加一个 getAttachs()方法,以便获取某一用户所有带附件的帖子,代码如下:

```
public List getAttachs(final int userId){
    String sql = "SELECT post_id,post_attach FROM t_post where user_id=? and
        post_attach is not null";
```



```

return jdbcTemplate.query(sql, new Object[] { userId },
    new RowMapper<Post>() {
        public Post mapRow(ResultSet rs, int rowNum)
            throws SQLException {
            int postId = rs.getInt(1);

            //①以二进制数组方式获取 BLOB 数据
            byte[] attach = lobHandler.getBlobAsBytes(rs, 2);
            Post post = new Post();
            post.setPostId(postId);
            post.setPostAttach(attach);
            return post;
        }
    });
}

```

通过 JdbcTemplate 的 List<T> query(String sql, Object[] args, RowMapper rowMapper) 接口处理行数据的映射。在 RowMapper 回调的 mapRow() 接口方法中, 通过 LobHandler 以 byte[] 获取 BLOB 字段的数据。

13.3.5 以流数据方式读取 LOB 数据

由于 Lob 数据的体积可能很大 (如 100MB), 如果直接以块的方式操作 LOB 数据, 则需要消耗大量的内存, 直接影响到程序的整体运行。对于体积很大的 LOB 数据, 可以使用流的方式进行访问, 以减少内存的占用。JdbcTemplate 为此提供了一个 Object query(String sql, Object[] args, ResultSetExtractor rse) 方法, ResultSetExtractor 接口拥有一个处理流数据的抽象类 org.springframework.jdbc.core.support.AbstractLobStreamingResultSetExtractor, 可以通过扩展此类用流的方式操作 LOB 字段的数据。下面为 PostDao 添加一个以流的方式获取某个帖子附件的方法, 代码如下:

```

...
public void getAttach(final int postId, final OutputStream os) {① ← 用于接收 LOB
    String sql = "SELECT post_attach FROM t_post WHERE post_id=? ";
    jdbcTemplate.query(
        sql, new Object[] {postId},
        new AbstractLobStreamingResultSetExtractor() {②
            protected void handleNoRowFound() throws LobRetrieval FailureException {③
                System.out.println("Not Found result!");
            }
            ← 处理未找到数据行的情况

            public void streamData(ResultSet rs) throws SQLException, IOException {④
                InputStream is = lobHandler.getBlobAsBinaryStream(rs, 1);
                if (is != null) {
                    FileCopyUtils.copy(is, os);
                    ← 以流的方式处理 LOB 字段
                }
            }
        }
    );
}

```

通过扩展 `AbstractLobStreamingResultSetExtractor` 抽象类, 在 `streamData(ResultSet rs)` 方法中以流的方式读取 LOB 字段数据, 如④处所示。这里又用到了 Spring 的工具类 `FileCopyUtils`, 将输入流的数据复制到输出流中。在 `getAttach()` 方法中通过入参 `OutputStream os` 接收 LOB 的数据, 如①处所示。用户可以同时覆盖抽象类中的 `handleNoRowFound()` 方法, 定义未找到数据行时的处理逻辑。

13.4 自增键和行集

Spring JDBC 提供了对自增键及行集的支持, 自增键对象让用户可以不依赖数据库的自增键, 在应用层为新记录提供主键。在 Java 1.4 中引入了 `RowSet`, 它允许在连接断开的情况下操作数据。本节将介绍如何在 Spring JDBC 中使用 `RowSet`。

13.4.1 自增键的使用

一般数据库都提供了自增键的功能, 如 MySQL 的 `auto_increment`、SQL Server 的 `identity` 字段等。Spring 允许用户在应用层产生主键值, 为此定义了 `org.springframework.jdbc.support.incrementer.DataFieldMaxValueIncrementer` 接口, 提供两种产生主键的方案: 第一, 通过序列产生主键; 第二, 通过表产生主键。根据主键产生方式及数据库类型的不同, Spring 提供了若干实现类, 如图 13-1 所示。

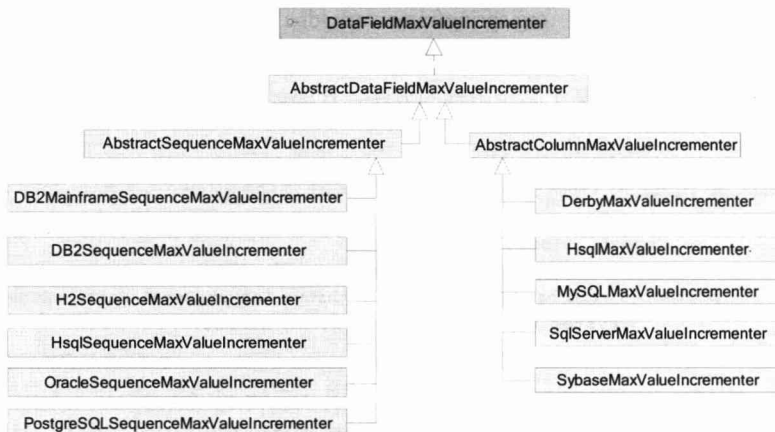


图 13-1 `DateFieldMaxValueIncrementer` 继承类图

根据不同的主键产生方式, 可能需要配置表名、主键字段名或序列名等信息。下面以 Oracle 和 MySQL 数据库为例, 分别讲解使用序列及表字段产生主键值的方式。

`DataFieldMaxValueIncrementer` 接口定义了 3 个获取下一个主键值的方法。

❑ `int nextIntValue()`: 获取下一个主键值, 主键数据类型为 `int`。

- `long nextLongValue()`: 获取下一个主键值, 主键数据类型为 `long`。
- `String nextStringValue()`: 获取下一个主键值, 主键数据类型为 `String`。

在其抽象实现类 `AbstractDataFieldMaxValueIncrementer` 中, 提供了几个重要的属性, 其中 `incrementerName` 属性用于定义序列或模拟序列的名称; 如果返回的主键值是 `String` 类型, 则 `paddingLength` 属性可能会派上用场, 它允许用户指定返回主键值的长度, 不足的部分前面补 0。 `AbstractSequenceMaxValueIncrementer` 类使用标准的数据库序列产生主键值, 而 `AbstractColumnMaxValueIncrementer` 类使用一张模拟序列的表产生主键值。 `AbstractColumnMaxValueIncrementer` 类可以通过 `cacheSize` 属性指定缓存的主键个数, 当内存中的主键值用完时, 递增器将一次性获取 `cacheSize` 个主键值, 这样就可以减少数据访问的次数, 提高应用的性能。

在代码清单 13-13 中, 手工指定 `t_post` 表的主键值为 1, 第二次运行该方法时就会抛出主键冲突的异常, 现在通过 `DateFieldMaxValueIncrementer` 从数据库中获取主键值来弥补这个缺陷。调整 `PostDao` 的代码, 添加 `DateFieldMaxValueIncrementer` 属性, 并通过它从序列中得到下一个主键值。

代码清单 13-13 使用 `DateFieldMaxValueIncrementer` 产生主键

```
@Repository
public class PostDao {
    ...
    private DataFieldMaxValueIncrementer incre; ① ← 主键值产生器

    @Autowired
    public void setIncre(DataFieldMaxValueIncrementer incre) {
        this.incre = incre;
    }

    public void addPost(final Post post) {
        ...
        getJdbcTemplate().execute(
            sql, new AbstractLobCreatingPreparedStatementCallback(
                this.lobHandler) {
                protected void setValues(PreparedStatement ps,
                    LobCreator lobCreator) throws SQLException {
                    ps.setInt(1, incre.nextIntValue()); ② ← 获取下一个主键值
                    ...
                }
            }
        );
    }
    ...//省略get/setter方法
}
```

在②处, 通过 `incre.nextIntValue()` 方法获取下一个主键值。

1. 以序列方式产生主键值

在 Oracle 数据库中创建一个 `seq_post_id` 序列, 使用这个序列为 `t_post` 表提供主键值。以下是创建 `seq_post_id` 序列的脚本:

```
create sequence seq_post_id
increment by 1
start with 1;
```

接着，调整 Spring 的配置，使用 `OracleSequenceMaxValueIncrementer` 作为主键产生器，代码如下：

```
<bean id="incre"
    class="org.springframework.jdbc.support.incrementer.
        OracleSequenceMaxValueIncrementer"
    p:incrementerName="seq_post_id" ① ← 指定序列名
    p:dataSource-ref="dataSource"/> ② ← 设置数据源
```

2. 以表方式产生主键值

在 MySQL 数据库中创建一张用于维护 `t_post` 主键的 `t_post_id` 表。以下是创建该表及插入初始化的 SQL 脚本：

```
create table t_post_id(sequence_id int) type = MYISAM;
insert into t_post_id values(0);
```

由于主键维护表的并发访问量很大，所以最好将其声明为 `MYISAM` 类型。此外，需要为该表提供初始值，以便后续主键值在此之上进行递增。

调整为 MySQL 数据库后，仅需对 Spring 配置文件进行小小的调整就可以了，如下：

```
<bean id="incre"
    class="org.springframework.jdbc.support.incrementer.MySQLMaxValueIncrementer"
    p:incrementerName="t_post_id" ① ← 设置维护主键的表名
    p:columnName="sequence_id" ② ← 用于生成主键值的列名
    p:cacheSize="10" ③ ← 缓存大小
    p:dataSource-ref="dataSource"/>
```

`incrementerName` 和 `columnName` 都很容易理解，`cacheSize` 决定一次返回的主键个数，这里设置为 10。当第一次通过 `MySQLMaxValueIncrementer#nextIntValue()` 方法获取主键值时，`MySQLMaxValueIncrementer` 将使 `t_post_id.sequence_id` 递增 10；而后续 9 次调用 `nextIntValue()` 方法时，都从缓存中获取主键值；直到第 10 次调用 `nextIntValue()` 方法时，才会再次将 `t_post_id.sequence_id` 字段值递增 10，如此循环反复。

13.4.2 如何规划主键方案

在实际应用中必须慎重规划好主键方案，它直接影响到应用系统的运行效率、扩展性及维护性等。一个欠考虑的主键方案不但会带来并发性的问题，还可能造成系统难于编程、不易维护。

从主键创建者的角度，可以粗略地将主键创建方案分为两类：其一为“应用层主键方案”，新数据的主键分配由应用层负责，如采用 `UUID` 或者使用 `DataFieldMaxValueIncrementer` 生成主键都属于这一类型；其二为“数据库层主键方案”，新数据的主键分配由数据库负责，即在定义表结构时将主键列设置为 `auto increment`，或者通过表的触发器分配主键。

笔者认为，“数据库层主键方案”已经成为历史遗留的产物，它的缺点和不足已经随着应用的发展不断凸显：其一，它会给应用开发带来不便，因为必须通过一个查询获取新增数据的主键值；其二，不方便主键值的全局管理和控制，使系统丧失灵活性；其

三，不方便数据的整合和迁移。所以，除非有充足的理由，否则应当尽量避免采用“数据库层主键方案”。

在传统的应用中，数据表主键往往只承担本表内数据唯一性标识的任务。随着应用复杂性的不断提高，数据表主键开始肩负起更多的功能，包括数据库切分、资源定位等。

笔者从事的一个电力生产管理系统，对 100 多种电网设备提供信息管理，包括变电站、线路、标上塔、变压器等，每台设备对应一张表。系统各地区独立部署，但数据必须定时抽取到省公司中汇合。不但要求单个系统内设备表主键是唯一的，还必须要求全局内是唯一的，以便将各地区的设备表数据合并到一起而不产生主键冲突；否则就必须要在数据整合时进行主键转换，而主键转换是极麻烦的事，会给系统带来很大的复杂度。

因此笔者当时采用了“应用层主键方案”，使用 UUID 为各设备表产生主键值，这样就可以保证设备 ID 的全局唯一性，为后期的数据整合带来便利。采用 UUID 的好处是，每台设备的 ID 都是唯一的，不但相同设备各地区的表数据合并不会有主键冲突问题，甚至不同设备的表数据合并也不会有主键冲突问题。

当然，采用 UUID 也有不好的地方，即 UUID 是一个 36 位的字符串，会占用较多的存储空间。所以，另一个候选的主键方案就是采用分段长整型编码方案，将主键编码分为 3 段：第一段代表地区；第二段代表设备类型；第三段代表设备序号，这样就可以创建一个全局唯一的整型主键值。当然，这时我们并不能直接使用 Spring 的 `DataFieldMaxValueIncrementer` 完成主键创建的任务，因为 `DataFieldMaxValueIncrementer` 只能为一张表创建主键。但道理是相通的，我们可以创建一张包含 3 个字段的主键表，这 3 个字段分别存储地区编码、设备类型编码及当前主键最大值，编写一个类似于 `DataFieldMaxValueIncrementer` 的接口以获取对应地区对应设备类型的主键值。



实战经验

在高并发的系统中，如果使用基于序列表的方式创建主键值，则应该考虑两个层面的并发问题：其一是应用层获取主键的并发问题，Spring 的 `DataFieldMaxValueIncrementer` 实现类已经对获取主键值的代码进行了同步，因此保证了同一 JVM 内应用不会产生并发问题；其二是全局的并发问题，如果应用是集群部署的，所有集群节点都通过同一张序列表获取主键值，那么就必须对这张序列表进行乐观锁定（序列表必须添加一个版本或时间戳字段），以防止集群节点的并发问题。但是很可惜，Spring 的 `DataFieldMaxValueIncrementer` 实现类并没有对序列表进行乐观锁定，因此，如果应用需要集群部署，那么 Spring 的 `DataFieldMaxValueIncrementer` 实现类是有全局并发问题的，必须自己实现 `DataFieldMaxValueIncrementer` 接口，以解决全局并发的的问题。

另外，`DataFieldMaxValueIncrementer` 接口只能为一张表提供主键，即每张表都必须配置一个单独的 `DataFieldMaxValueIncrementer`，因此这种方案太死板了。如果需要采用序列表创建主键的方案，笔者建议参照 `DataFieldMaxValueIncrementer` 接口，自行编写一个可为多张表提供主键的接口。

13.4.3 以行集返回数据

行集对象可以绑定一个数据连接并在其整个生命周期中维持该连接，在此情况下，该行集对象被称为“连接的行集”。行集对象还可以先绑定一个数据连接，获取数据后就关闭它，这种行集被称为“非连接行集”。非连接行集可以在断开连接时更改数据，然后重新绑定数据连接，并将对数据的更改同步到数据库中。

由于 `java.sql.ResultSet` 方法抛出的异常是 `SQLException`，所以 Spring 在 `org.springframework.jdbc.support.rowset` 包中提供了 `SqlRowSet` 和 `SqlRowSetMetaData` 接口，对 `RowSet` 和 `RowSetMetaData` 进行了封装，以便将 `SQLException` 转换为 Spring DAO 体系的异常。

`JdbcTemplate` 为了获得基于行集的结果集，提供了以下查询方法：

- ☐ `SqlRowSet queryForRowSet(String sql)`。
- ☐ `SqlRowSet queryForRowSet(String sql, Object... args)`。
- ☐ `SqlRowSet queryForRowSet(String sql, Object[] args, int[] argTypes)`。

下面在 `TopicDao` 中添加一个以 `SqlRowSet` 形式返回用户所有主题的方法，代码如下：

```
public SqlRowSet getTopicRowSet(int userId) {
    String sql = "SELECT topic_id,topic_title FROM t_topic WHERE user_id=?";
    return getJdbcTemplate().queryForRowSet(sql, userId);
}
```

在 `TestTopicDao` 测试类中，通过以下代码对返回的 `SqlRowSet` 进行访问：

```
@Test
public void testGetTopicRowSet() {
    SqlRowSet srs = topicDao.getTopicRowSet(1);

    //①这时，数据连接已经断开
    while (srs.next()) {
        System.out.println(srs.getString("topic_id"));
    }
}
```

在 `topicDao` 查询并返回 `SqlRowSet` 的结果集后，数据连接就已经断开了，但是结果集的数据已经保存在 `SqlRowSet` 中。因此，我们可以访问到 `SqlRowSet` 的结果集数据。值得注意的是，`RowSet` 会一次性装载所有的匹配数据，而不像 `ResultSet` 一样，分批次返回一批数据（一批的行数为 `fetchSize`）。

所以，对于大结果集的数据，使用 `SqlRowSet` 会造成很大的内存消耗，这一点要铭记。不过 `JdbcTemplate` 的 `maxSize` 属性依旧会限制 `SqlRowSet` 的返回记录数。

13.5 NamedParameterJdbcTemplate 模板类

除了标准的 `JdbcTemplate` 外，Spring 还提供了两个易用的 JDBC 模板类，分别是

NamedParameterJdbcTemplate 和 SimpleJdbcTemplate。前者提供命名参数绑定的功能；而后者封装了 JdbcTemplate，将常用的 API 开放出来。

在低版本的 Spring 中，用户只能使用“?”占位符声明参数，并使用索引号绑定参数。在使用这种方法绑定参数时必须足够小心，以保证参数的索引号和 SQL 语句中占位符(?)的位置正确匹配。这种编程模式被认为是弱稳定的，因为当新增一个“?”占位符时，可能导致原来所有的参数绑定方法都需要因此调整索引号，这极有可能引入一些不容易发现的错误。

NamedParameterJdbcTemplate 模板类支持命名参数变量的 SQL，它位于 org.springframework.jdbc.core.namedparam 包中。该包中还定义了一个用于承载命名参数的 SqlParameterSource 接口，该接口拥有两个实现类。

- BeanPropertySqlParameterSource: 该实现类将一个 JavaBean 对象封装成一个参数源，以便通过 JavaBean 属性名和 SQL 语句中命名参数匹配的方式绑定参数。
- MapSqlParameterSource: 该实现类内部有一个 Map 存储参数，可以通过 addValue(String paramName, Object value)或 addValues(Map values)方法添加参数，并通过参数键名和 SQL 语句中命名参数匹配的方式绑定参数。

来看一个使用 NamedParameterJdbcTemplate 的实例，如代码清单 13-14 所示。

代码清单 13-14 使用命名参数绑定的模板类

```
package com.smart.dao;
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
...
@Repository
public class ForumDao {
    ...
    private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

    @Autowired
    public void setNamedParameterJdbcTemplate(NamedParameterJdbcTemplate
        namedParameterJdbcTemplate) {
        this.namedParameterJdbcTemplate = namedParameterJdbcTemplate;
    }

    public void addForumByNamedParams(final Forum forum) {
        final String sql = "INSERT INTO t_forum(forum_name, forum_desc)
            VALUES (:forumName, :forumDesc)"; ① 定义参数源
        SqlParameterSource sps = new BeanPropertySqlParameterSource(forum); ②
        namedParameterJdbcTemplate.update(sql, sps); ③ 定义命名参数
    }
    ...
}
```

使用模板类方法

在 SQL 语句中声明命名参数的格式是“:paramName”，如①处所示。在这个实例中，使用 BeanPropertySqlParameterSource 提供参数源，它接收一个 JavaBean 作为构造函数的入参，如②处所示。和标准的 JdbcTemplate 一样，NamedParameterJdbcTemplate 提供了众多数据访问方法，这些方法大都拥有一个 SqlParameterSource 入参，用以提供参数

源, 这里使用模板类的 `int update(String sql, SqlParameterSource paramSource)` 方法执行插入数据的操作, 如③处所示。

必须在 `applicationContext.xml` 中定义 `NamedParameterJdbcTemplate` 的 Bean, 如下:

```
<bean id="namedParamJdbcTemplate"
      class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate">
    <constructor-arg ref="dataSource"/>
</bean>
```

Forum 领域对象拥有 3 个属性, 其中 `forumName` 和 `forumDesc` 这两个属性和 SQL 语句中的命名参数匹配, 参数即按照这个匹配关系进行绑定。为了更清楚地看出参数绑定规则, 下面给出 Forum 领域对象的代码:

```
package com.smart.domain;
...
public class Forum implements Serializable{
    private int forumId;
    private String forumName;
    private String forumDesc;
    //省略get/setter
}
```

如果数据表记录没有对应的领域对象, 则用户可以直接使用 `MapSqlParameterSource` 达到参数绑定的目的。可以按照如下方式使用 `MapSqlParameterSource`:

```
public void addForum(final Forum forum) {
    final String sql = "INSERT INTO t_forum(forum_name,forum_desc)
                       VALUES (:forumName, :forumDesc)";
    MapSqlParameterSource sps = new MapSqlParameterSource().① ← 提供参数源
        addValue("forumName", forum.getForumName()).
        addValue("forumDesc", forum.getForumDesc());
    namedParameterJdbcTemplate.update(sql, sps);
}
```

由于 `MapSqlParameterSource` 中的大多数方法能够返回对象本身, 所以可以将几个方法的调用串成一个链, 如①处所示。由于这个原因, 使用方法调用链模式设计的 API 受到越来越多开发者的青睐。

值得一提的是, `NamedParameterJdbcTemplate` 可支持 `in`, 而 `PreparedStatement` 对 `in` 的支持只能动态拼接。如下例假设想找 `id` 在 1,2,3,4 范围内的数据:

```
String in_data="1,2,3,4,5,6";
pst = conn.prepareStatement("select id,name from B where id in ( ? )");
pst.setString(1,in_data);
```

使用传统的 `PreparedStatement` 是动态设定参数的, 也就是现在生成 `select id,name from B where id in ( ? )`, 一个“?”代表一个参数, `pst.setString(1,"1,2,3,4")`就相当于 `select id,name from B where id in ('1,2,3,4')`。我们期望找到 `id` 在 1,2,3,4 中的数据, 而这时候生成的 SQL 语句是取查找 `id` 为 1,2,3,4 的记录。使用 `NamedParameterJdbcTemplate` 可以很好地解决这个问题, 如下例:

```
String sql = "select id, name from B where id in (:ids)";
MapSqlParameterSource parameters = new MapSqlParameterSource();
parameters.addValue("ids", ids);
```


这时候使用 `NamedParameterJdbcTemplate` 就相当于 `select id,name from B where id in ('1','2','3','4')`), 符合我们的期望, 而且不需要拼接 SQL 语句。

13.6 小结

本章以 `JdbcTemplate` 模板类为中心, 详细讲解了各种使用 `JdbcTemplate` 对数据进行 CRUD 操作的方法。 `JdbcTemplate` 使用大量的回调接口完成数据的访问操作, `StatementCallback`、`PreparedStatementCallback`、`CallableStatementCallback`、`BatchPreparedStatementSetter` 及 `RowMapper` 是其中常用的回调接口, 一般可以通过匿名内部类实现这些回调接口, 使代码更加紧凑。

此外, 还重点讲述了在 Spring JDBC 中操作 LOB 数据类型的方法。由于有些数据库使用了自己的 API 操作 LOB 数据, 因此需要从数据源的代理 JDBC 对象中抽取本地数据库的 JDBC 对象。LOB 数据可以使用块数据和流数据的方式进行访问。如果 LOB 数据比较大, 则通过流操作方式可以减少内存的占用, 同时保证程序的性能。

在本章中, 我们花费了较大的篇幅讲述主键的知识, 主键的产生方式从产生地点上可以分为应用层产生和数据库产生两种。应用层借助数据库的序列或表产生主键值, 这种方式可以保证程序的可移植性和安全性, 同时可以通过缓存机制提高运行效率。有些数据库支持数据表自增主键值产生机制, 在 JDBC 3.0 以前的版本中, 无法通过 `Statement` 自动获取新增记录对应的主键值。这时需要在插入数据后, 立即执行一条数据库相关的主键获取 SQL 语句以得到对应的主键值。在数据库高并发的情况下, 有可能获取到不正确的主键值。在这种情况下, 在插入数据前在应用层准备好主键值是很好的备选方案。

Spring 持续降低 Spring JDBC 的使用难度, 它提供支持命名参数绑定的 `NamedParameterJdbcTemplate`, 以减少编程的出错概率。

第 14 章

整合其他 ORM 框架

罗素在《西方哲学史》中有一句经典的话：参差多态乃是幸福的本源。按照这样的说法，Spring 开发者是幸福的，因为在 Spring 中不但可以选择 Spring JDBC 作为持久化技术，还可以选择 Hibernate、JPA、JDO 等多种类型的持久化技术。由于篇幅所限，与其大同小异地介绍 Spring 支持的所有 ORM，不如选择其中两个最具代表性的（Hibernate 和 MyBatis）进行重点学习。同时，读者还将学习到 ORM 框架的混用和 DAO 层设计的知识。

本章主要内容：

- ◆ Spring 对 ORM 框架整合的基本思想
- ◆ 在 Spring 中使用 Hibernate
- ◆ 在 Spring 中使用 MyBatis
- ◆ 如何在 Spring 封装的基础上进行 DAO 层的基础设计

本章亮点：

在不同的 ORM 框架中处理 LOB 数据

14.1 Spring 整合 ORM 技术

Spring 的开放性和扩展性在持久化技术领域得到了充分的证明，Spring 不但直接提供了 Spring JDBC，还整合了 JPA、Hibernate、JDO 等 ORM 领域的代表者，使用者完全可以根据需要做出适合自己的选择。

第 13 章详细讲解了 Spring 对 JDBC API 的封装，其中大部分的思路和整合其他 ORM 框架是一脉相通的：在资源管理、DAO 模板类、事务管理、DAO 异常体系等方面，Spring 始终保持整合方式上的统一。在 Spring 中掌握了一种持久化技术后，切换到另一种持久化技术并不需要花费什么代价。

Spring 在集成 ORM 框架时,始终走“民主”、“亲民”的路线。它不仅提供了方便的模板类对原 ORM 进行简化封装,以一种更具 Spring 的风格使用 ORM 技术;同时,在保证继续享受 Spring 通用功能的情况下,开发者还可以使用 ORM 框架原生的 API 编写程序。这让一直使用 ORM 原生 API 编程的开发者感到很自然,不会存在过渡上的障碍。

在使用某种 ORM 框架时,为了让其更适合项目的需要,降低使用难度,一般情况下都会在原有 ORM API 的基础上编写一套封装类。Spring 高屋建瓴地为我们做了类似的工作,所以在决定花费力气并冒着风险去构造类似的底层架构之前,最好事先参考一下 Spring 的解决方案,而不要急着另起炉灶。

使用 Spring 所提供的 ORM 整合方案,可以获得许多好处。

1. 方便基础设施的搭建

不同的 ORM 技术都有一套自己的方案以初始化框架、搭建基础设施等。在搭建基础设施的过程中,数据源是不可或缺的,不同 ORM 框架的实现方式各不相同,如在 JPA 中通过 `persistence.xml` 定义数据源,而 Hibernate 在 `hibernate.cfg.xml` 中配置数据源,目的相同但方法迥异。在数据源的基础上,不同的 ORM 框架拥有自己的基础实例,如 Hibernate 的 `SessionFactory`,它们是 ORM 程序运行时的底层设施,在程序级别代表着 ORM 框架本身。在学习不同的 ORM 框架知识时,往往会有一个独立的篇章专门讲解如何初始化这些实例。在 Spring 中,对于不同的 ORM 框架,首先,始终可以采用相同的方式配置数据源;其次, Spring 为不同的 ORM 框架提供了相应的 `FactoryBean`,用以初始化 ORM 框架的基础设施,可以将它们当成普通的 `Bean` 对待,唯一的差别是属性不同。

2. 异常封装

Spring 能够转换各种 ORM 框架所抛出的异常,将 ORM 框架专有的或检查型异常转换为 Spring DAO 异常体系中的标准异常。这样用户就可以有选择地在适当的地方处理感兴趣的异常,忽略大部分不可恢复的异常,从而避免了很多令人讨厌的异常声明和异常捕捉代码。

3. 统一的事务管理

通过使用基于 Spring DAO 模板的编程风格,甚至使用 ORM 框架原生的 API,只要遵循 Spring 所提出的很少的编程要求,就可以使用 Spring 提供的事务管理功能。Spring 为不同的 ORM 框架提供了对应的事务管理器,可以采用声明的方式进行事务管理,并且可以透明地实现本地事务管理到全局 JTA 事务管理的切换。此外,作为一项附加功能, JDBC 代码能够在事务级别上与用户使用的 ORM 框架一起使用。这一功能对于诸如批量处理、LOB 操作等并不适合单独采用 ORM 完成的地方尤其有用。

4. 允许混合使用多个 ORM 框架

由于 Spring 在 DAO 异常、事务、资源等高级层次建立了抽象,因而可以让业务层

对 DAO 具体实现的技术不敏感。这给开发者带来了一个好处：他们可以在底层选用适合的实现方式，甚至可以混合使用多种 ORM，一般地，CRUD 使用 Hibernate，而数据查询使用 MyBatis 或 Spring JDBC，博采众长，强强联合。同时，用户也可以使用这种特性整合一些遗留的代码，而不必一切从头开始。

5. 方便单元测试

Spring 容器使得替换不同的实现和配置变得非常简单，这些内容包括 Hibernate SessionFactory 的位置、JDBC DataSource、事务管理器及映射对象的实现（如果需要）等，这样就很容易隔离并测试不同的 DAO 类。

14.2 在 Spring 中使用 Hibernate

Hibernate 在 ORM 领域具有广泛的影响，拥有广大的使用群体。它提供了 ORM 最完整、最丰富的实现，在 Spring 4.0 中目前全面支持 Hibernate 5.0，不再支持 Hibernate 3.6 之前的版本。因为 iBatis 的升级版 MyBatis 自身已经提供了对 Spring 整合的支持，所以 Spring 不再为 MyBatis 提供直接支持，大家直接使用 MyBatis 自带的整合功能即可。

14.2.1 配置 SessionFactory

使用 Hibernate 框架的首要工作是编写 Hibernate 的配置文件，其次是如何使用这些配置文件实例化 SessionFactory，创建 Hibernate 的基础设施。

Spring 为创建 SessionFactory 提供了一个好用的 FactoryBean 工厂类：org.springframework.orm.hibernateX.LocalSessionFactoryBean，通过配置一些必要的属性，就可以获取一个 SessionFactory Bean。

LocalSessionFactoryBean 配置灵活度很高，支持开发者的不同习惯，让开发者拥有充分的选择权——这是 Spring 一贯的风格。

1. 零过渡障碍的配置方式

让我们回忆一下使用 Hibernate API 创建一个 SessionFactory 的过程：首先编写好对象关系的映射文件 xxx.hbm.xml；然后通过 Hibernate 的配置文件 hibernate.cfg.xml 将所有的 xxx.hbm.xml 映射文件组装起来；最后通过以下两行经典的代码得到 SessionFactory 的实例：

```
Configuration cfg = new Configuration().configure("hibernate.cfg.xml");  
SessionFactory sessionFactory = cfg.buildSessionFactory();
```

hibernate.cfg.xml 配置文件拥有创建 Hibernate 基础设施所需的配置信息，来看一个最简单的 Hibernate 配置文件，如代码清单 14-1 所示。

代码清单 14-1 hibernate.cfg.xml: Hibernate 配置文件

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-5.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="connection.url">jdbc:mysql://localhost:3309/sampled
  </property>
    <property name="connection.username">root</property>
    <property name="connection.password">1234</property>
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="show_sql">true</property>
    <property name="format_sql">true</property>
    <property name="current_session_context_class">thread</property>
    <mapping resource="com/smart/orm/domain/Forum.hbm.xml"/>
    <mapping resource="com/smart/orm/domain/Topic.hbm.xml"/>
    <mapping resource="com/smart/orm/domain/Post.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

这个配置文件定义了 3 个方面的信息：数据源、映射文件及 Hibernate 控制属性。既然在 Hibernate 中可以使用一个配置文件创建一个 SessionFactory 实例，在 Spring 中也可以顺理成章地通过指定一个 Hibernate 配置文件，利用 LocalSessionFactoryBean 来达到相同的目的。

```
<bean id="sessionFactory"
  class="org.springframework.orm.hibernate5.LocalSessionFactoryBean"
  p:configLocation="classpath:hibernate.cfg.xml"/>①
```

如①处所示，通过 configLocation 属性指定了一个 Hibernate 配置文件。如果有多个 Hibernate 配置文件，则可以通过 configLocations 属性指定，多个文件之间用逗号分隔。

LocalSessionFactoryBean 将利用 Hibernate 配置文件创建一个 SessionFactory 代理对象，以便和 Spring 的事务管理机制配合工作：当数据访问代码使用 SessionFactory 时，可以获取线程绑定的 Session，不管工作在本地或全局的事务，都能正确参与到当前的 Spring 事务管理中。

2. 更具 Spring 风格的配置

Spring 对 ORM 技术的一个重要支持就是提供统一的数据源管理机制，也许更多的开发者更愿意使用 Spring 配置数据源，即在 Spring 容器中定义数据源、指定映射文件、设置 Hibernate 控制属性等信息，完成集成组装的工作，完全抛开 hibernate.cfg.xml 配置文件，如代码清单 14-2 所示。

代码清单 14-2 移除 Hibernate 配置文件的配置

```
<?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"
xmlns:aop="http://www.springframework.org/schema/aop"
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/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:property-placeholder location="classpath:jdbc.properties" />
<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}"/>
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate5.LocalSessionFactoryBean"
    p:dataSource-ref="dataSource">① 数据源

    <!--②指定Hibernate 实体类映射文件-->
    <property name="mappingLocations">
        <list>
            <value>classpath*:com/smart/orm/domain/Forum.hbm.xml</value>
            <value>classpath*:com/smart/orm/domain/Topic.hbm.xml</value>
            <value>classpath*:com/smart/orm/domain/Post.hbm.xml</value>
        </list>
    </property>

    <!--③指定Hibernate配置属性-->
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">
                org.hibernate.dialect.MySQLDialect
            </prop>
            <prop key="hibernate.show_sql">
                true
            </prop>
        </props>
    </property>
</bean>
...
</bean>

```

数据源、映射文件及 Hibernate 控制属性这三方面的信息在 LocalSessionFactoryBean 中得到了完美集成，完全替代了 hibernate.cfg.xml 的作用，但这种配置对于 Spring 开发者而言更加亲切。

首先，①处指定的数据源是 Spring 容器中的数据源，不管是直接在 Spring 容器中配置，还是通过<jee:jndi-lookup>从 EJB 容器中获取，对引用者而言是完全透明的。

其次，凭借 Spring 资源处理的强大功能，指定 Hibernate 映射文件变得相当灵活。

在②处采用了逐个指定映射文件的方法，其实这个方法是最笨拙的。由于 `mappingLocations` 属性的类型是 `Resource[]`，因此它还支持以下简洁的配置方式：

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate5.LocalSessionFactoryBean"
    p:dataSource-ref="dataSource"
    p:mappingLocations="classpath:/com/smart/orm/domain/**/*.hbm.xml">
    ...
</bean>
```

除了 `mappingLocations` 属性外，也可以选择通过其他资源属性指定 Hibernate 映射文件。

- ❑ `mappingJarLocations`：如果映射文件位于 JAR 文件中，则可以通过该属性指定，如 `WEB-INF/lib/example.hbm.jar`。
- ❑ `p:mappingDirectoryLocations`：可以指定多个放置 Hibernate 映射文件的目录，Spring 将加载这些目录下的所有 Hibernate 映射文件，如 `classpath:/com/smart/orm/domain`。
- ❑ `mappingResources`：通过相对于类路径的方式指定映射文件，属性类型为 `String[]`。既可以通过内嵌多个 `<value>` 的 `<list>` 元素指定映射文件，也可以通过逗号分隔的方式指定，如 `"topic.hbm.xml,post.hbm.xml"`。

其他 Hibernate 属性通过键值对的方式提供，如③处所示，键的名称请参考 Hibernate 的说明文档。

由于这种配置方式将所有的配置信息都统一到 Spring 中，给管理、维护和调整工作带来了方便，因而被广泛接受。

14.2.2 使用 HibernateTemplate

基于模板类使用 Hibernate 是最简单的方式，它可以在不牺牲 Hibernate 强大功能的前提下，以一种更简洁的方式使用 Hibernate，极大地降低了 Hibernate 的使用难度。按照 Spring 的风格，它提供了使用模板的支持类 `HibernateDaoSupport`，并通过 `getHibernateTemplate()` 方法向子类开放模板类实例的调用。

为了能够使用注解配置的功能，我们自己编写一个 `BaseDao`，如下：

```
package com.smart.orm.dao.hibernate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.orm.hibernate5.HibernateTemplate;
public class BaseDao {
    @Autowired
    private HibernateTemplate hibernateTemplate;
    ...
}
```

然后通过扩展这个 `BaseDao` 基类创建一个使用 `HibernateTemplate` 的 Dao，如代码清单 14-3 所示。

代码清单 14-3 使用HibernateTemplate的DAO

```

package com.smart.orm.dao.hibernate;

import java.sql.SQLException;
import java.util.List;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.springframework.orm.hibernate5.HibernateCallback;
import org.springframework.stereotype.Repository;
import com.smart.orm.domain.Forum;

@Repository
public class ForumHibernateDao extends BaseDao{

    public void addForum(Forum forum) {
        getHibernateTemplate().save(forum); //①保存实体对象
    }

    public void updateForum(Forum forum) {
        getHibernateTemplate().update(forum); //②更改实体对象
    }

    public Forum getForum(int forumId) {
        return getHibernateTemplate().get(Forum.class, forumId); //③获取实体对象
    }

    public List<Forum> findForumByName(String forumName) { //④使用HQL查询
        return (List<Forum>)getHibernateTemplate().find(
            "from Forum f where f.forumName like ?", forumName+"%");
    }

    public long getForumNum() { //⑤使用Iterate返回结果
        Object obj =getHibernateTemplate()
            .iterate("select count(f.forumId) from Forum f")
            .next();
        return (Long)obj;
    }
}

```

HibernateTemplate 代理了 Hibernate Session 的大多数持久化操作，并以一种更简洁的方式提供调用。HibernateTemplate 所提供的大部分方法对于 Hibernate 开发者来说都是熟悉亲切的，因为模板类的方法大都可以 Session 接口中找到镜像。

1. 常用的 API 方法

下面来了解一下 HibernateTemplate 开放的一些有代表性的 API 方法，其他更多的方法请参见 Spring 的 Javadoc 文档。

- ☐ Serializable save(Object entity): 保存实体对象，并返回主键值。还有一个和该方法功能类似的 void persist(Object entity)方法，后者是 JSR-220 规定的方法。
- ☐ void update(Object entity): 更新实体对象。
- ☐ void saveOrUpdate(Object entity): 保存或更新一个实体。还有一个和该方法功能类似的<T> T merge(T entity)方法，后者是 JSR-220 规定的方法。

- ❑ `void delete(Object entity)`: 删除一个实体。
- ❑ `List find(String queryString)`: 根据 HQL 查询实体。该方法拥有几个带参的重载版本, 如 `find(String queryString, Object value)`、`List find(String queryString, Object... values)`。在内部, 模板类会自动创建 Query 并执行查询。
- ❑ `List findByNameQuery(String queryName)`: 执行命名查询。其也拥有多个可绑定参数的重载版本。
- ❑ `List findByCriteria(DetachedCriteria criteria)`: Criteria 版本的查询。该方法有一个可限定范围的重载版本: `List findByCriteria(org.hibernate.criterion.DetachedCriteria criteria, int firstResult, int maxResults)`。

2. 使用回调接口

一般情况下, 使用模板类的简单代理方法就可以满足要求了, 如果希望使用更多 Hibernate 底层的功能, 则可以使用回调接口。Spring 定义了一个回调接口 `org.springframework.orm.hibernate5.HibernateCallback<T>`, 该接口拥有唯一的方法, 如下:

`T doInHibernate(org.hibernate.Session session) throws HibernateException, SQLException`

该接口配合 `HibernateTemplate` 进行工作, 它无须关心 `Hibernate Session` 的打开/关闭等操作, 仅需定义数据访问逻辑即可。可以通过该接口返回结果, 结果可以是一个实体对象或一个实体对象的 `List`。回调接口中抛出的异常将传播到模板类中并被转换成 Spring DAO 异常体系的对应类。

`HibernateTemplate` 定义了两个使用 `HibernateCallback` 回调接口的方法。

- ❑ `<T> T execute(HibernateCallback<T> action)`: 一般使用该方法执行数据更新、新增等操作。
- ❑ `List executeFind(HibernateCallback<?> action)`: 一般使用该方法执行数据查询操作, 返回的结果是一个 `List`。

使用回调接口对代码清单 14-3 中的 `getForumNum()` 方法提供另一个版本的实现, 如下:

```
public long getForumNum2() {
    Long forumNum = getHibernateTemplate().execute(
        new HibernateCallback<Long>() {
            public Long doInHibernate(Session session)
                throws HibernateException, SQLException {
                Object obj = session
                    .createQuery("select count(f.forumId) from Forum f")
                    .list()
                    .iterator()
                    .next();
                return (Long) obj;
            }
        });
    return forumNum;
}
```

代码中粗体部分以匿名类的方式提供了 `HibernateCallback` 回调的实现，直接使用 `Hibernate Session` 接口完成查询的工作并返回结果。

3. 在 Spring 中配置 DAO

在编写好基于 `HibernateTemplate` 的 DAO 类后，接下来要做的就是 在 Spring 中进行具体配置，使该 DAO 生效，代码如下：

```
<?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"
    xmlns:aop="http://www.springframework.org/schema/aop"
    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/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">

    <!--①扫描类包以启动注解驱动的Bean-->
    <context:component-scan base-package="com.smart"/>
    <context:property-placeholder location="classpath:jdbc.properties" />
    <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}"/>

    <bean id="sessionFactory"
        class="org.springframework.orm.hibernate5.LocalSessionFactoryBean"
        p:dataSource-ref="dataSource"
        p:mappingDirectoryLocations="classpath:/com/smart/orm/domain"
        >
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.dialect">
                    org.hibernate.dialect.MySQLDialect
                </prop>
                <prop key="hibernate.show_sql">
                    true
                </prop>
            </props>
        </property>
    </bean>

    <!--②配置HibernateTemplate Bean-->
    <bean id="hibernateTemplate"
        class="org.springframework.orm.hibernate5.HibernateTemplate"
```

```

    p:sessionFactory-ref="sessionFactory"/>

<!--③配置Hibernate的事务管理器-->
<bean id="transactionManager"
    class="org.springframework.orm.hibernate5.HibernateTransactionManager"
    p:sessionFactory-ref="sessionFactory" />

<tx:annotation-driven transaction-manager="transactionManager"/>
</beans>

```

在 Spring 配置文件中, 首先需要配置一个 `HibernateTemplate` Bean, 它基于 `SessionFactory` 工作, 如②处所示。在①处使用 `<context:component-scan>` 扫描特定的类包以启用注解驱动的 Bean, 这样 `ForumHibernateDao` 类的 `@Repository` 及 `@Autowired` 注解就可以起作用, 将 `ForumHibernateDao` 装配为 Spring 容器中的 Bean。

14.2.3 处理 LOB 类型的数据

对 LOB 类型的数据的处理始终是各种 ORM 框架比较头疼的事, 需要认真对待。Hibernate 为处理特殊数据类型字段定义了一个接口: `org.hibernate.usertype.UserType`。Spring 在 `org.springframework.orm.hibernate5.support` 包中为 BLOB 和 CLOB 类型提供了几个 `UserType` 的实现类。因此, 可以在 Hibernate 的映射文件中直接使用这两个实现类轻松处理 LOB 类型的数据。

- `BlobByteArrayType`: 将 BLOB 数据映射为 `byte[]` 类型的属性。
- `BlobStringType`: 将 BLOB 数据映射为 `String` 类型的属性。
- `BlobSerializableType`: 将 BLOB 数据映射为 `Serializable` 类型的属性。
- `ClobStringType`: 将 CLOB 数据映射为 `String` 类型的属性。

前面使用的 `Post` 领域对象有两个分别对应 CLOB 和 BLOB 字段类型的属性, 下面使用 Spring 的 `UserType` 为 `Post` 配置 Hibernate 的映射文件, 如代码清单 14-4 所示。

代码清单 14-4 post.hbm.xml: LOB数据映射配置

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-5.0.dtd">
<hibernate-mapping auto-import="true" default-lazy="false">
    <class name="com.smart.orm.domain.Post" table="t_post">
        <id name="postId" column="post_id">
            <generator class="identity" />
        </id>
        <property name="userId" column="user_id"/>
        <property name="postText" column="post_text" ①
            type="org.springframework.orm.hibernate5.support.ClobStringType"/>
        <property name="postAttach" column="post_attach" ②
            type="org.springframework.orm.hibernate5.support.BlobByteArrayType"/>
        <property name="postTime" column="post_time" type="date" />
        <many-to-one name="topic" column="topic_id" class="com.smart.orm.domain.Topic" />
    </class>
</hibernate-mapping>

```

对应 CLOB 字段

对应 BLOB 字段

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

postText 为 String 类型的属性，对应数据库的 CLOB 类型；而 postAttach 为 byte[] 类型的属性，对应数据库的 BLOB 类型。分别使用 Spring 所提供的相应 UserType 实现类进行配置，如①和②处所示。

在配置好映射文件后，还需要在 Spring 配置文件中定义 LOB 数据处理器，让 SessionFactory 拥有处理 LOB 数据的能力。

```
<bean id="lobHandler"
    class="org.springframework.jdbc.support.lob.DefaultLobHandler"
    lazy-init="true" />

<bean id="sessionFactory"
    class="org.springframework.orm.hibernate5.LocalSessionFactoryBean"
    p:dataSource-ref="dataSource"
    p:mappingDirectoryLocations="classpath:/com/smart/orm/domain"
    p:lobHandler-ref="lobHandler">①      设置 LOB 数据处理器
    ...
</bean>
```

这样，仅需简单地使用 HibernateTemplate#save(Object entity)等方法，就可以正确地保存 LOB 数据。如果是 Oracle 9i 数据库，还需要配置一个本地 JDBC 抽取器，并使用特定的 LobHandler 实现类。

在使用 LobHandler 操作 LOB 数据时，需要在事务环境下才能工作，所以必须先配置事务管理器，否则会抛出异常。



提示

在使用 Spring JDBC 时，除了可以按 byte[]、String 类型处理 LOB 数据外，还可以使用流的方式操作 LOB 数据，当数据超过一定的大小时（比如 100MB），流操作是唯一可行的方式。可惜，Spring 并未提供以流方式操作 LOB 数据的 UserType（Spring 开发组成员认为在实现上存在难度）。不过，www.atlassian.com 替 Spring 完成了这件难事，用户可以通过以下地址了解到这个满足要求的 BlobInputStream Type：

<http://docs.atlassian.com/atlassian-confluence/latest/org/springframework/orm/hibernate/support/BlobInputStreamType.html>。

14.2.4 添加 Hibernate 事件监听器

Hibernate 设计了一个功能完备的事件模型，允许为事件装配一个或多个事件监听器。通过 Configuration#setListener(String type, Object listener)等方法向 Hibernate 框架注册事件监听器。Hibernate 在 org.hibernate.event 包中定义了事件及对应的事件监听器接口，并在 org.hibernate.event.def 包中提供了事件监听器接口的默认实现。

LocalSessionFactoryBean 允许用户通过 eventListeners 属性向 Hibernate 注册事件监听

器。Spring 本身就提供了一个 Hibernate 的事件监听器 `IdTransferringMergeEventListener`。下面通过简单的配置将其注册到 Hibernate 中。

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
    ...
    <property name="eventListeners">①
        <map>
            <entry key="merge">②-1 ← 事件监听器类型
                <bean class="org.springframework.orm.hibernate5.support.
                    IdTransferringMergeEventListener" />②-2 ← 事件监听器实现类
                </entry>
            </map>
        </property>
    </bean>
```

`eventListeners` 属性是 `Map` 类型的，它以键值对的方式接收事件监听器的注册信息，其中键为事件类型，而值为事件监听器实现类。事件类型必须是 Hibernate 预定义的类型，包括 `auto-flush`、`merge`、`create`、`delete`、`dirty-check`、`evict`、`flush`、`flush-entity`、`load`、`load-collection`、`lock`、`refresh`、`replicate` 和 `save-update` 等。注册监听器时需要指定事件类型，在这一点上，Hibernate 做得实在不够友好。它完全可以利用类反射机制，根据监听器实现类自动判断对应的事件类型，就像在 `web.xml` 中注册 Servlet 容器事件监听器一样，但 Hibernate 却要求必须显式指定事件的名称。

`IdTransferringMergeEventListener` 是 Spring 提供的一个 Hibernate 事件监听器，它必须和 `HibernateTemplate#merge(Object entity)` 方法配合使用。因为 Hibernate 的 `merge()` 方法在对一个新对象进行操作时，并不会将 ID 值传递给原对象，Spring 通过该事件监听器完成这项工作。

14.2.5 使用原生的 Hibernate API

Hibernate 3.0 引入了一个新的特性：通过 `SessionFactory#getCurrentSession()` 方法能够获取和当前线程绑定的 `Session`。这一特性使得 Hibernate 自身具备了获取和事务线程绑定的 `Session` 对象的功能，这与在 Spring 的 `HibernateTemplate` 中使用和事务绑定的 `Session` 是相同。

因此，可以在 Spring 中使用原生的 Hibernate API 编写 DAO，它同样可以正确地和 Spring 的事务管理器一起工作。下面使用原生的 Hibernate API 实现 `ForumDao` 接口，如代码清单 14-5 所示。

代码清单 14-5 `ForumHibernateDao`：基于原生 Hibernate API 的实现

```
package com.smart.orm.dao.hibraw;
...
@Repository
public class ForumHibernateDao {
```

```
//①直接注入Hibernate原生的SessionFactory对象
@Autowired
private SessionFactory sessionFactory;
public void addForum(Forum forum) {
    sessionFactory.getCurrentSession().save(forum);
}
public void updateForum(Forum forum) {
    sessionFactory.getCurrentSession().update(forum);
}
...
}
```

DAO 注入了一个 SessionFactory 对象，如①处所示。这样，DAO 中的所有数据访问方法都可以通过 SessionFactory#getCurrentSession() 方法获取和当前线程绑定的 Session，以便 Spring 的事务管理器能够正确地工作。

和使用 HibernateTemplate 不同的是，使用原生 Hibernate API 所抛出的异常是 Hibernate 异常（也是运行期异常）。这意味着 DAO 的调用者只能以普通的错误来处理这些异常，而无法在声明式事务中使用通用的 Spring DAO 异常体系进行回滚设置。一般情况下，这种差异是可以接受的，并不会带来什么问题。

14.2.6 使用注解配置

和 Spring 类似，Hibernate 不但可以使用 XML 提供 ORM 的配置信息，也可以直接在领域对象类中通过注解定义 ORM 映射信息。Hibernate 不但自己定义了一套注解，还支持 JSR-220 的 JPA 注解。

下面使用注解对 Forum 进行 ORM 的配置，如代码清单 14-6 所示。

代码清单 14-6 使用JPA注解的Forum

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

import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="T_FORUM")
public class Forum implements Serializable{

    @Id
    @Column(name = "FORUM_ID")
    private int forumId;

    @Column(name = "FORUM_NAME")
    private String forumName;

    @Column(name = "FORUM_DESC")
    private String forumDesc;
```

```
public String getForumDesc() {
    return forumDesc;
}
}
```

Hibernate 通过 `AnnotationConfiguration` 的 `addAnnotatedClass()` 或 `addPackage()` 方法加载使用 JPA 注解的实体类, 获取映射的元数据信息, 并在此基础上创建 `SessionFactory` 实例。

需要特别注意的是, 使用 `addPackage` 并不是加载类包下所有标注了 ORM 注解的实体类, 而是加载类包下 `package-info.java` 文件中定义的 `Annotation`, 而该类包下的所有持久化类仍然需要通过 `addAnnotatedClass()` 方法加载。

Spring 专门提供了一个配套的 `AnnotationSessionFactoryBean`, 用于创建基于 JPA 注解的 `SessionFactory`。

```
<!--①通过AnnotationSessionFactoryBean定义SessionFactory -->
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate5.annotation.
        ④AnnotationSessionFactoryBean"
    p:dataSource-ref="dataSource">
    <property name="annotatedClasses"> ②
        <list>
            <value>com.smart.orm.domain.Forum</value>
        </list>
    </property>
    <property name="hibernateProperties">
        ...
    </property>
</bean>
```

`AnnotationSessionFactoryBean` 扩展了 `LocalSessionFactoryBean` 类, 增强的功能是: 可以根据实体类的注解获取 ORM 的配置信息。也可以混合使用 XML 配置和注解配置对象关系映射, Hibernate 内部自动将这些元数据信息进行整合, 并不会产生冲突。

`annotatedClasses` 属性指定使用 JPA 注解的实体类名, 如②处所示。如果实体类比较多, 不要想当然地以为通过 `annotatedPackages` 属性指定实体类所在包名就可以了, `annotatedPackages` 在内部通过调用 Hibernate 的 `AnnotationConfiguration` 的 `addPackage()` 方法加载包中 `package-info.java` 文件定义的 `Annotation`, 而非包中标注注解的实体类。

Spring 为了通过扫描方式加载带注解的实体类, 提供了一个易用的 `packagesToScan` 属性, 可以指定一系列包名, Spring 将扫描并加载这些包路径 (包括子包) 的所有带注解实体类。

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate5.annotation.AnnotationSessionFactoryBean"
    p:dataSource-ref="dataSource">
    <property name="packagesToScan" value="com.smart.orm.domain"/>
    <property name="hibernateProperties">
        ...
    </property>
</bean>
```

`packagesToScan` 属性可接收多个类包路径, 用逗号分隔即可, 例如:

```
<property name="packagesToScan" value="package1,package2"/>
```

14.2.7 事务处理

Spring 的通用事务管理模型对 Hibernate 是完全适用的，包括程式化事务、基于 TransactionProxyFactoryBean、基于 aop/tx 及基于 @Transaction 注解的事务管理。在这里，仅给出基于 @Transaction 注解的事务管理，其他方式请参照第 11 章的内容。

首先，为 BbtForumServe 添加 @Transactional 注解。

```
package com.smart.orm.service.hibernate;

...

@Transactional
@Service
public class BbtForumServe{

    @Autowired
    private ForumHibernateDao forumDao;

    @Autowired
    private TopicHibernateDao topicDao;

    @Autowired
    private PostHibernateDao postDao;
    ...
}
```

其次，在 Spring 配置文件中配置 Hibernate 事务管理器，并启用注解驱动事务。

```
<bean id="transactionManager"
    class="org.springframework.orm.hibernate5.HibernateTransactionManager"
    p:sessionFactory-ref="sessionFactory" />① 为事务管理器指定 sessionFactory

    默认查找名为 transactionManager
    的事务管理器，因此无须显式指定
<tx:annotation-driven />②
```

Hibernate 的事务管理器需要注入一个 SessionFactory 实例，将其命名为 transactionManager 后，在 <tx:annotation-driven />中就无须通过 transaction-manager 默认显式指定了。不管 BbtForumImpl 所用的 DAO 是基于 HibernateTemplate 还是基于 Hibernate 原生的 API，BbtForumImpl 中的所有方法都具有事务性。



轻松一刻

传说晋朝的旌阳县曾有一名道术高深的县令，叫许逊。他能施符作法，替人驱鬼治病，百姓见他像仙人一样神，就称他为“许真君”。一次，由于年成不好，农民缴不起赋税。许逊便叫大家把石头挑来，然后施展法术，用手指一点，使石头都变成了金子。这些金子足以应付百姓拖欠的赋税。成语“点石成金”据此而来。Spring 通过 @Transaction 注解，就可以让业务类这个无事务性的“石头”变成具有事务性的“金子”，Spring 真是一个不折不扣的“许真君”。



14.2.8 延迟加载问题

Hibernate 允许对关联对象、属性进行延迟加载,但是必须保证延迟加载的操作限于同一个 Hibernate Session 范围之内。如果 Service 层返回一个启用了延迟加载功能的领域对象给 Web 层,当 Web 层访问到那些需要延迟加载的数据时,由于加载领域对象的 Hibernate Session 已经关闭,因而将导致延迟加载数据的访问异常。

Spring 为此专门提供了一个 `OpenSessionInViewFilter` 过滤器,它的主要功能是使每个请求过程绑定一个 Hibernate Session,即使最初的事务已经完成,也可以在 Web 层进行延迟加载操作。

`OpenSessionInViewFilter` 过滤器将 Hibernate Session 绑定到请求线程中,它将自动被 Spring 的事务管理器探测到。所以 `OpenSessionInViewFilter` 适用于 Service 层使用 `HibernateTransactionManager` 或 `JtaTransactionManager` 进行事务管理的环境,也可用于非只读事务的数据操作中。

要启用这个过滤器,必须在 `web.xml` 中进行如下配置:

```
...
<filter>
    <filter-name>hibernateFilter</filter-name>
    <filter-class>org.springframework.orm.hibernate5.support.OpenSessionInViewFilter
</filter-class>
</filter>
<filter-mapping>
    <filter-name>hibernateFilter</filter-name>
    <url-pattern>*.html</url-pattern>
</filter-mapping>
...
```

在上面的配置中,假设使用 `.html` 后缀作为 Web 框架的 URL 匹配模式。如果用户使用 Struts 等 Web 框架,则可以将其改为对应的 `*.do` 模型。



实战经验

对于小型应用来说,使用 `OpenSessionInViewFilter` 确实可以降低延迟加载所引发的各种问题,使 Service 层的代码更易开发和维护。但对于大型且高并发的应用来说,笔者强烈建议不要使用 `OpenSessionInViewFilter`。因为 `OpenSessionInViewFilter` 会让每个 Web 请求线程都绑定一个 Hibernate 的 Session,即绑定一个数据连接,直到完成 Web 请求处理后才释放数据连接。这会造成两个性能问题:其一,加大了对数据连接资源访问的并发性(因为原本有些 Web 请求可能并不需要使用数据连接);其二延长了每个 Web 请求对数据连接资源占用的时长。

我们知道,应用系统的瓶颈往往都出现在数据库上,使用 `OpenSessionInViewFilter` 会使系统更容易出现数据库资源的性能瓶颈。如果要抛弃 `OpenSessionInViewFilter`,则开发者必须设计好 Service 层的服务接口,使延迟加载的工作在 Service 层内完成。此外,

还要求提供满足不同需求的 Service 接口。假设 User 和 Dept 两个实体对象是 Many-to-One 的关系，User 中的 Dept 使用了延迟加载方案，那么 UserService 可能就需要提供两个不同的获取 User 对象的服务接口，以应对不同的应用场景：其一为 getUserWithDept()，返回带 Dept 的 User；其二为 getUserWithoutDept()，返回不带 Dept 的 User。如果使用 OpenSessionInViewFilter，则仅需提供一个 getUser() 服务接口，因为延迟加载的操作在 Web 层由具体调用操作自动触发。

14.3 在 Spring 中使用 MyBatis

使用 MyBatis 提供的 ORM 机制，对业务逻辑实现人员而言，面对的是纯粹的 Java 对象，这一点与使用 Hibernate 框架基本一致。对于具体的数据操作，Hibernate 会自动生成 SQL 语句，而 MyBatis 则要求开发者编写具体的 SQL 语句。相对于 Hibernate 等“全自动”的 ORM 机制而言，MyBatis 在 SQL 开发的工作量和数据库移植性上做出了让步，为数据持久化操作提供了更大的自由空间。相对于“全自动”的 ORM 实现方案来说，MyBatis 的出现显得别有创意。

2010 年 5 月，iBATIS 更名为 MyBatis 并迁移到 Google Code，可以把 MyBatis 看作 iBatis 3.0。从公告上可以看出，开发团队想脱离 Apache 基金会寻求独立发展。开发团队保证不会修改授权协议（Apache License）、代码完全兼容、包名不会更改、不会删除 Apache 站上的任何资源。

由于 Spring 越来越成为 Java 事实标准的技术框架，因此 MyBatis 团队开发出整合类，可以让开发者直接在 Spring 中使用 MyBatis。因此，Spring 4.0 移除了对 iBatis 的直接支持，Spring 更乐于让第三方框架自身提供整合支持。

事务管理可以由 Spring 标准机制进行处理。对于 MyBatis 来说，没有特别的事务策略，除了 JDBC Connection 外，也没有特别的事务资源。它和 Spring JDBC 事务管理的方式完全一致，采用和 Spring JDBC 相同的 DataSourceTransactionManager 事务管理器。

14.3.1 配置 SqlMapClient

每个 MyBatis 的应用程序都以一个 SqlSessionFactory 对象的实例为核心。SqlSessionFactory 对象的实例可以通过 SqlSessionFactoryBuilder 对象来获得。SqlSessionFactoryBuilder 对象可以从 XML 配置文件或 Configuration 类的实例中构建 SqlSessionFactory 对象。

和 Hibernate 相似，MyBatis 拥有多个 SQL 映射文件，并通过一个配置文件对这些 SQL 映射文件进行装配，同时在该文件中定义一些控制属性的信息。下面是一个简单的 MyBatis 配置文件，如代码清单 14-7 所示。

代码清单 14-7 mybatisConfig.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
  <settings> ①
    <setting name="lazyLoadingEnabled" value="false" />
  </settings>
  <typeAliases>②
    <typeAlias alias="Forum" type="com.smart.orm.domain.Forum" />
    <typeAlias alias="Topic" type="com.smart.orm.domain.Topic" />
    <typeAlias alias="Post" type="com.smart.orm.domain.Post" />
  </typeAliases>
  <mapper> ③
    <mapper resource="com/smart/orm/domain/mybatis/Forum.xml"/>
    <mapper resource="com/smart/orm/domain/mybatis/Topic.xml"/>
    <mapper resource="com/smart/orm/domain/mybatis/Post.xml"/>
  </mapper>
</configuration>

```

在①处提供可控制 MyBatis 框架运行行为的属性信息。在②处定义全限定类名的别名，在映射文件中可以通过别名代替具体的类名，简化配置。在③处将 MyBatis 的所有映射文件组装起来。

在③处通过<mapper>标签引用了 3 个 SQL 映射文件，下面来了解一下其中的 Forum.xml 文件，如代码清单 14-8 所示。

代码清单 14-8 Forum.xml

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.smart.orm.dao.mybatis.ForumMybatisDao">①
  <select id="getForum" resultType="Forum" parameterType="int" > ②
    SELECT
      forum_id forumId,
      forum_name forumName,
      forum_desc forumDesc
    FROM t_forum
    WHERE forum_id = #{forumId}
  </select>
  ...
  <insert id="addForum" parameterType="Forum">③
    INSERT INTO t_forum(forum_id,forum_name,forum_desc)
    VALUES(#{forumId},#{forumName}, #{forumDesc})
  </insert>
  <update id="updateForum" parameterType="Forum">④
    UPDATE t_forum f
    SET forum_name=#{forumName},forum_desc=#{forumDesc}
    WHERE f.forum_id = #{forumId}
  </update>
</mapper>

```

该文件定义了对 Forum 实体类进行数据操作时所需的 SQL 语句，同时还定义了查询结果和对象属性的映射关系。在①处指定了映射所在的命名空间，每个具体的映射项都有一个 id，可以通过命名空间和映射项的 id 定位到具体的映射项。如通过如下语句可以调用 getForum 的映射语句：

```
SqlSession session = sqlMapper.openSession();
try {
    Forum forum = (Forum) session.selectOne(
        " com.smart.orm.dao.mybatis.ForumMybatisDao. getForum ", 1);
} finally {
    session.close();
}
```

在②、③和④处分别定义了一条 SELECT、INSERT 及 UPDATE 语句映射项，映射项的 parameterType 指定传入的参数对象，可以是全限定名的类，也可以是类的别名，类的别名在 MyBatis 的主配置文件中定义，如代码清单 14-7 中的②处所示。如果映射项的入参是基础类型或 String 类型，则可以使用如 int、long、string 的基础类型名。SELECT 映射项拥有返回类型对象，通过 resultType 指定。在映射项中通过#{xxx} 绑定 parameterType 指定参数类的属性，支持级联属性，如#{topic.forumId}。

14.3.2 在 Spring 中配置 MyBatis

可以使用 MyBatis 提供的 mybatis-spring 整合类包实现 Spring 和 MyBatis 的整合，从功能上来说，mybatis-spring 完全符合 Spring 的风格。要在 Spring 中整合 MyBatis，必须将 mybatis-spring 构件添加到 pom.xml 中（具体参见本章工程的 pom.xml 文件），如代码清单 14-9 所示。

代码清单 14-9 applicationContext-mybatis.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans ...>
    <context:component-scan base-package="com.smart.orm.dao.mybatis" />
    <context:component-scan base-package="com.smart.orm.service.mybatis" />
    <context:property-placeholder location="classpath:jdbc.properties" />
    <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}" />

    <bean id="sqlSessionFactory"
        class="org.mybatis.spring.SqlSessionFactoryBean" ①
        p:dataSource-ref="dataSource"
        p:configLocation="classpath:myBatisConfig.xml"/> ②
</beans>
```

mybatis-spring 类包提供了一个 SqlSessionFactoryBean，以便通过 Spring 风格创建

MyBatis 的 `SqlSessionIoFactory`，如①处所示。只需注入数据源并指定 MyBatis 的总装配文件就可以了，如②处所示。

如果在 MyBatis 的总装配文件 `mybatisConfig.xml` 中指定了 SQL 映射文件，则必须逐个列出所有的 SQL 映射文件，比较烦琐。是否可以像 Spring 加载 Hibernate 映射文件一样按资源路径匹配规则扫描式加载呢？答案是肯定的。`SqlSessionFactoryBean` 提供了 `mapperLocations` 属性，支持扫描式加载 SQL 映射文件。

首先将映射文件匹配从 `mybatisConfig.xml` 中移除，然后通过如下便捷方式加载 SQL 映射文件：

```
<bean id="sqlSessionFactory"
    class="org.mybatis.spring.SqlSessionFactoryBean"
    p:dataSource-ref="dataSource"
    p:configLocation="classpath:myBatisConfig.xml"
    p:mapperLocations="classpath:com/smart/orm/domain/mybatis/*.xml"/>
```

这样，`SqlSessionFactoryBean` 将扫描 `com/smart/orm/domain/mybatis` 类路径并加载所有以.xml 为后缀的映射文件。

14.3.3 编写 MyBatis 的 DAO

1. 使用 `SqlSessionTemplate`

`mybatis-spring` 效仿 Spring 的风格提供了一个模板类 `SqlSessionTemplate`，可以通过模板类轻松地访问数据库。

首先在 `applicationContext-mybatis.xml` 中配置好 `SqlSessionTemplate` Bean。

```
<bean class="org.mybatis.spring.SqlSessionTemplate">
    <constructor-arg ref="sqlSessionFactory"/>
</bean>
```

然后就可以使用 `SqlSessionTemplate` 调用 SQL 映射项完成数据访问操作，如代码清单 14-10 所示。

代码清单 14-10 `ForumMybatisTemplateDao.java`

```
package com.smart.orm.dao.mybatis;

import org.mybatis.spring.SqlSessionTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

import com.smart.orm.domain.Forum;

@Repository
public class ForumMybatisTemplateDao{

    @Autowired
    private SqlSessionTemplate sessionTemplate;

    public Forum getForum(int forumId){ ①
        return (Forum)sessionTemplate.selectOne(
```

```

        "com.smart.orm.dao.mybatis.ForumMybatisDao.getForum",
        forumId);
    }
}

```

在①处, `SqlSessionTemplate` 通过 `selectOne()` 方法调用在 `Forum.xml` 映射文件中定义的命名空间为 `com.smart.orm.dao.mybatis.ForumMybatisDao`、映射项 `id` 为 `getForum` 的 SQL 映射项, 并传入参数, 返回映射成 `Forum` 对象的查询结果。

在 `SqlSessionTemplate` 模板类中提供了多个方便调用的方法, 常用方法介绍如下。

- ❑ `List<?> selectList(String statement, Object parameter)`: 调用 `select` 映射项, 返回一个结果对象集合。其中, `statement` 为映射项全限定名, 即包括命名空间和映射项 `id` (下同); 而 `parameter` 为传递给映射项的入参。
- ❑ `int insert(String statement, Object parameter)`: 调用 `insert` 映射项, 返回插入的记录数。
- ❑ `int update(String statement, Object parameter)`: 调用 `update` 映射项, 返回更改的记录数。

2. 使用映射接口

代码清单 14-10 在①处使用字符串指定映射项, 这种方式很容易引起错误。因为字符串本身没有语义性, 如果存在编写错误, 则在编译期无法识别, 只能等到运行期才能发现。`MyBatis` 为解决这个问题, 特别提供了一种可将 SQL 映射文件中的映射项通过名称匹配接口进行调用的方法: 接口名称和映射命名空间相同, 接口方法和映射元素的 `id` 相同。

下面为 `Forum.xml` 文件的映射项定义一个调用接口, 如代码清单 14-11 所示。

代码清单 14-11 `ForumMybatisDao.java`

```

package com.smart.orm.dao.mybatis;

import java.util.List;

import com.smart.orm.domain.Forum;

public interface ForumMybatisDao{
    void addForum(Forum forum);
    void updateForum(Forum forum) ;
    Forum getForum(int forumId) ;
    long getForumNum() ;
    List<Forum> findForumByName(String forumName);
}

```

类名为 `com.smart.orm.dao.mybatis.ForumMybatisDao`, `Forum.xml` 文件中的每个映射项对应一个接口方法, 接口方法的签名和映射项的声明匹配。

在定义好 `ForumMybatisDao` 接口后, 该如何通过该接口进行数据访问呢? 毕竟 `ForumMybatisDao` 接口没有任何实现类。一种简单的方式是通过 `SqlSessionTemplate` 获取接口的实例。

```

package com.smart.orm.dao.mybatis;
...
@Repository
public class ForumMybatisTemplateDao{

    @Autowired
    private SqlSessionTemplate sessionTemplate;
    public Forum getForum2(int forumId){
        ForumMybatisDao forumMybatisDao =
            sessionTemplate.getMapper(ForumMybatisDao.class);
        return forumMybatisDao.getForum(forumId);
    }
}

```

SqlSessionTemplate 提供了一个可以根据接口类返回接口实例的方法 `getMapper(Class<T> type)`，直接访问接口实例的方法，即可调用 SQL 映射文件定义的映射项。

这种方法虽然比直接通过字符串指定映射项的方法安全便捷，但还不是最优的方法。对于 Spring 应用来说，更希望在 Service 类中通过 `@Autowired` 注解直接注入接口实例，而非每次都通过 `getMapper(Class<T> type)` 方法获取实例。

mybatis-spring 提供了一个“神奇”的转换器 `MapperScannerConfigurer`，它可以将映射接口直接转换为 Spring 容器中的 Bean，这样就可以在 Service 中注入映射接口的 Bean 了。假设已经为 3 个 SQL 映射文件分别定义了对应的接口类，这些接口类位于 `com.smart.orm.dao.mybatis` 包中，接口名分别为 `ForumMybatisDao`、`TopicMybatisDao` 及 `PostMybatisDao`。使用如下配置即可将接口转换为 Bean：

```

<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer"
    p:sqlSessionFactory-ref="sqlSessionFactory"
    p:basePackage="com.smart.orm.dao.mybatis"/>

```

`MapperScannerConfigurer` 将扫描 `basePackage` 所指定的包下的所有接口类（包括子包），如果它们在 SQL 映射文件中定义过，则将它们动态定义为一个 Spring Bean，这样就可以在 Service 中直接注入映射接口的 Bean 了，如代码清单 14-12 所示。

代码清单 14-12 ForumMybatisDao.java

```

package com.smart.orm.service.mybatis;
...
import com.smart.orm.dao.mybatis.ForumMybatisDao;
import com.smart.orm.dao.mybatis.PostMybatisDao;
import com.smart.orm.dao.mybatis.TopicMybatisDao;

@Transactional
@Service
public class BbtForumSerive{

    @Autowired
    private ForumMybatisDao forumDao;

    @Autowired
    private TopicMybatisDao topicDao;
}

```

```
@Autowired
private PostMybatisDao postDao;

public void addForum(Forum forum) {
    forumDao.addForum(forum);
}
...
}
```

如粗体代码所示，BbtForumSevice 可以直接使用 `@Autowired` 注入这些映射接口的 Bean，然后即可顺利地通过接口方法调用 MyBatis 的映射项访问数据。

14.4 DAO 层设计

Spring 为其支持的各种 ORM 框架提供了基于模板模式的 Template 基类，此外还为模板类提供了方便的支持类，其内部包含一个模板类，Spring 推荐开发者直接继承这个 Support 类定义自己的 DAO。但是，在实际应用中，直接继承 Spring 的 Support 定义实体类 DAO 存在一些不足之处，下面将探讨通过引入一个基类简化具体 DAO 类的设计思路。

此外，Spring 提供的各种 ORM 模板类的查询方法使用 `Object[]` 传递查询条件参数。很多开发者发现使用 `Object[]` 传递查询条件参数并非一个最优的办法，因此出现了很多查询方法的设计版本。本节对此进行了汇总，读者可以通过对比，找到自己认为最适合的方式。

14.4.1 DAO 基类设计

虽然 Spring 通过模板类和支持类为各种 ORM 框架提供了出色的支持，但也存在一些不足。首先，这些模板类对泛型的支持是方法级别的，需要通过方法入参指定泛型的类型。对于保存、更改、删除等操作，泛型的意义并不是很大；但对于加载实体、查询实体的操作，泛型可以带来很大的便利。其次，由于支持类通过类似于 `getHibernateTemplate()`、`getJdbcTemplate()` 的方式向子类开放模板类，所以子类在执行数据操作时，必须采取诸如 `getJdbcTemplate().execute()` 的形式，子类代码显得比较拖沓。

在实际应用中，开发者一般会在 Spring 支持类的基础上编写自己的 DAO 基类，进行自己的封装，以获得泛型的支持，并提供自己的代理方法，使子类避免通过显式调用模板类实例进行数据操作。DAO 基类并不需要对模板类的所有方法进行代理，仅需对一些常用的方法（如 CRUD 操作方法）进行代理即可，对不常用的方法则可要求子类显式调用模板实例完成。这样，一方面简化了常用方法的调用，另一方面避免了使基类过于复杂。

由于一般实体类对应的 DAO 都必须拥有 CRUD 操作，与其在每个实体 DAO 接口中重复定义这些方法，不如提供一个通用的 DAO 接口。具体的实体 DAO 接口可以通过扩展这个通用的 DAO 接口并定义实体类相关的其他操作方法来实现。

在没有类级别泛型支持的情况下查询实体，不但要指定主键值，还需要指定实体类型；由于方法入参比较复杂，调用者还需要对返回的结果对象进行强制类型转换。这两种缺点都可以通过在基类中引入泛型得到彻底解决。

BaseDao<T>基类对所有实体 DAO 接口的通用方法进行了抽象并提供了泛型支持，下面以 Hibernate 为例，说明项目 DAO 基类的设计方式，如代码清单 14-13 所示。

代码清单 14-13 BaseDao<T>: DAO基类

```
package com.smart.orm.dao;

import java.io.Serializable;
import java.lang.reflect.ParameterizedType;
import java.lang.reflect.Type;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.orm.hibernate5.HibernateTemplate;

public class BaseDao<T> { //①提供DAO类级别的泛型支持

    @Autowired
    private HibernateTemplate hibernateTemplate; //②注入Hibernate模板类

    private Class entityClass; //③DAO的泛型类，即子类所指定的T所对应的类型

    public BaseDao() { //④通过反射方式获取子类DAO对应的泛型实体类
        Type genType = getClass().getGenericSuperclass();
        Type[] params = ((ParameterizedType) genType).getActualTypeArguments();
        entityClass = (Class) params[0];
    }

    public T get(Serializable id){
        return (T)hibernateTemplate.get(entityClass, id); //⑤直接使用entityClass
    }

    public void save(T entity){
        hibernateTemplate.save(entity);
    }

    public void update(T entity){
        hibernateTemplate.update(entity);
    }

    public HibernateTemplate getHibernateTemplate() {
        return hibernateTemplate;
    }
    ...
}
```

基类 BaseDao<T>通过泛型方式允许子类 DAO 指定操作的实体类，以便简化常用的

数据操作方法。同时，在②处，BaseDao 注入了一个 HibernateTemplate，这样子类只要标注@Repository 注解就会拥有 HibernateTemplate 成员变量，无须各自声明。BaseDao 的构造方法通过 Java 反应机制自动解析出子类 T 所对应的类型，以便 HibernateTemplate 直接利用这个信息进行数据访问操作。如⑤处的 hibernateTemplate.get(entityClass, id) 就使用泛型的类型，得到了一个更为便捷的 get() 方法。

此外，还可以在该类中提供更多方便的操作方法化解子类的实现难度。这样，子类仅需指定泛型对应的实体类，就拥有了各种通用的数据操作能力。如果实体 DAO 的数据操作仅是一些常见的 CRUD 操作，那么子类甚至可以不编写任何代码，DAO 类的编码生产率将得到极大的提高，代码复用率也将上升一个台阶。通过扩展 BaseDao<T> 基类可以得到一个被极大简化的 ForumDao 版本。

```
package com.smart.orm.dao;

import com.smart.orm.domain.Forum;

public class ForumDao extends BaseDao<Forum> { //①通过泛型指定实体类为Forum

    //②子类仅需编写特定的DAO方法就可以了，其他通用的方法从基类继承
    public long getForumNum() {
        Object obj = getHibernateTemplate().iterate(
            "select count(f.forumId) from Forum f").next();
        return (Long) obj;
    }
}
```

通过扩展 BaseDao<T> 基类并借由泛型指定实体类，ForumDao 立即拥有了对 Forum 实例进行 CRUD 操作的功能，并且这些功能都是泛型版本的。如当用户调用 ForumDao#get(Serializable id) 方法加载实体对象时，将直接返回 Forum 类型的实体对象。因此，在 ForumDao 类中，开发者仅需编写个性化的接口方法即可，普通的增、删、查、改方法直接从 BaseDao 中继承。按照相似的方式，还可以轻松完成 TopicDao、PostDao 的编写，用户将发现 DAO 层的代码质量拥有了一个不小的飞跃。

14.4.2 查询接口方法设计

DAO 层除了 CRUD 的数据操作外，另一个重要的操作就是根据查询条件执行数据查询，不同的 ORM 框架都允许用户动态绑定参数确定查询条件。查询条件项的数目往往是不固定的，如既可能要求以 userName 为条件查询 User，也可能要求以 userName+status 等组合条件查询 User。条件项数目的不定性给查询接口方法的设计造成了一定的困难。在实体 DAO 定义带参的查询方法时，一般有 5 种方式，下面分别进行介绍。

方式 1：每个条件项参数对应一个入参

在查询方法中为每个查询条件项定义一个对应的入参，例如：

```
List findOrder(String hql, Date startTime, Date endTime, int deptId)
```

这种方法签名含义清晰，可读性强，内部的逻辑处理简单，但接口稳定性差。如果 `findOrder()` 方法需要添加一个 `userName` 条件，则有两种重构的方式。第一，调整方法的签名，新增一个 `String userName` 入参，这种重构被认为是不健康的重构，因为它违反了软件设计中经典的“开-闭原则”。此外，如果条件项个数很多，方法签名将显得过于拖沓。第二，在 DAO 类中新增一个重载查询方法。但如果查询条件项的组合数过多，则 DAO 类的方法数目将直线上升，整个实体 DAO 类将显得臃肿笨重。

方式 2：使用数组传递条件项参数

即通过数组的方式传递查询条件项参数。由于参数类型具有不一致性，因而要求数组类型采用 `Object[]`。

```
List findOrder(String hql, Object[] params)
```

这种方式可以应对查询条件项参数组合的多样性并保持接口的稳定性，开发者必须在方法内部从数组中获取参数再传递给查询引擎。其缺点是方法的可读性不强，调用者往往需要通过查看接口对应的 Javadoc 文档才能正确使用。此外，在 Java 5.0 以下的版本中，因为没有自动拆/解包的语法特性，所以在调用前必须对基本类型的参数使用封装类进行封装，有时在方法内部还需要进行相反的过程，在使用上较为不便。不过由于 Spring 为支持的 ORM 框架都提供了类似的查询接口（如 `HibernateTemplate#find(String queryString, Object[] values)`），所以 DAO 方法内部的处理相对还是比较简单的。

方式 3：使用 Java 5.0 的不定参数

如果是在 Java 5.0 中，则可以采用不定参数进行方法签名。这种方式在逻辑上和方式 2 并无多大的区别。

```
List findOrder(String hql, Object... params)
```

方式 4：将查询条件项参数封装成对象

为了提高方式 1 中方法签名的简洁性，增强方式 2、3 中方法签名的可读性，方式 4 提出将查询条件项参数封装成一个对象的思路，即查询方法使用查询条件对象传递查询条件。

```
List<Order> findOrder(String hql, OrderQueryParam param)
```

`OrderQueryParam` 查询条件对象封装了 `hql` 查询语句可能会用到的条件项参数，在查询方法内部，开发者必须判断查询条件对象的属性并正确绑定条件项参数。由于需要为条件项参数定义一个类，因此会造成类数目的膨胀，有时甚至一个实体 DAO 需要对多个查询条件参数类。另外，当需要添加一个新的条件项参数时，条件封装类还需要进行相应调整。

方式 5：使用 Map 传递条件项参数

另一种被广泛使用的方式是采用 Map 传递条件项参数，键为参数名，值为参数值。

```
List<Order> findOrder(String hql, Map params)
```

使用这种方式，接口方法签名可以在条件项发生变化的情况下保持稳定，同时通过

键指定条件项参数名在一定程度上解决了接口的健壮性（当调用者指定错误参数名时容易得到错误报警）。但和方式 2、3 一样，调用者必须通过接口 Javadoc 文档才能明白不同条件项要以什么名称进行设置。

14.4.3 分页查询接口设计

绝大多数应用系统都采用分页的方式展现业务数据，并在客户端业务列表页面中提供分页导航栏，用户可以通过分页导航栏查看各页的业务数据。按获取数据方式的不同，分页技术大致可分为以下 3 种。

- 客户端分页：直接将全部或多页结果数据一次性返回给客户端，客户端通过展现组件进行数据分页的控制。
- 数据库分页：在进行数据查询时，数据库仅返回一页数据给客户端。
- 服务器端分页：从数据库返回全部或多页数据，在服务器端缓存多页数据，但只返回一页数据给客户端。

客户端分页可以减小和服务端交互的次数，在进行分页切换时，直接从客户端的缓存中获取数据，无须和服务端进行再次交互，提高了系统交互性；但会增加第一次交互的负荷。数据库分页要求每次切页时都访问数据库，这增加了数据库访问的并发性；但每次从数据库返回的数据较少，当次交互的负荷较轻。而服务器端分页在以上两者之间寻求平衡，它既减少了数据库访问的并发性，同时使服务器端返回给客户端的当次负荷也较小；但服务器端分页技术需要考虑到数据缓存、数据同步等问题，提高了系统的复杂性。

DAO 不可避免地需要涉及分页查询接口，分页查询接口包括查询参数、分页设置及分页结果等对象。下面是一个简单的分页查询接口的设计方案，如图 14-1 所示。

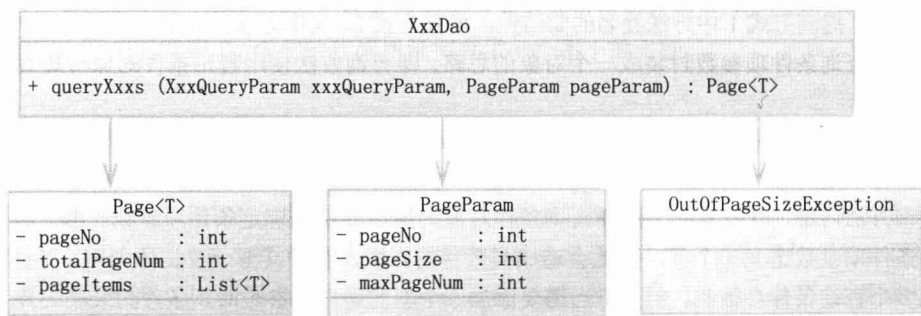


图 14-1 分页查询接口的设计方案

queryXxxs 有两个入参，其中 XxxQueryParam 为查询参数对象，而 PageParam 为分页的设置信息，包括页码（pageNo）、每页记录数（pageSize）及允许查询的最大查询页数（maxPageNum）等。Page<T> 为当前页数据的封装类，其中 pageNo 为页码，totalPageNum 为结果集总页数（客户端的分页导航栏需要用到 pageNo 和 totalPageNum），pageItems 为当

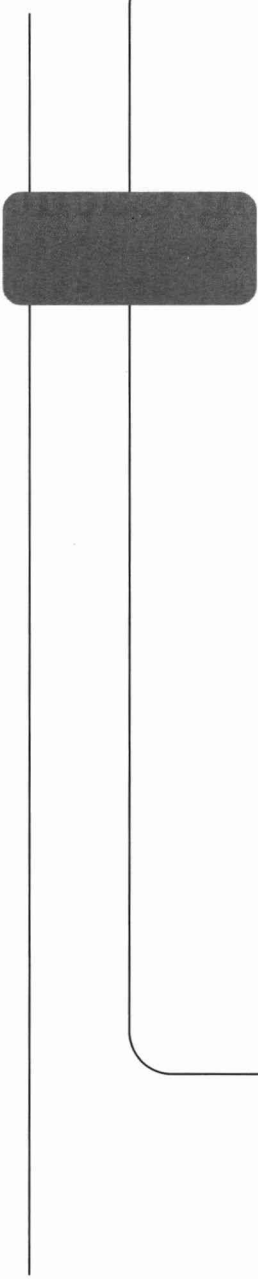
前页的业务数据集合。当需要查询的页码超过结果集的最大页数，或者超过允许查询的最大页数时，将抛出 `OutOfPageSizeException` 运行期异常。

14.5 小结

Spring 为其支持的 ORM 框架提供了方便易用的 `FactoryBean`，用以创建 ORM 框架的基础设施。Spring 通过模板类在不损失框架功能性的前提下大大降低了使用这些 ORM 技术的难度，因此成为 Spring 推荐的使用模式。不过，Spring 并不强制使用模板类，对于任何一种 ORM 技术，Spring 都允许用户使用原生的 API 构造 DAO。在使用原生 API 时，Spring 能保证用户获取到事务绑定的资源，Spring 的事务管理机制同样有效。

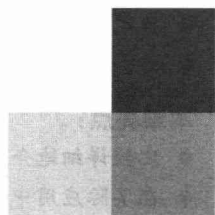
Spring 对 Hibernate 所提供的支持应该是最丰富的，以创建 `SessionFactory` 为例，就有多种配置方式，如直接使用 Hibernate 配置文件、使用混合型的 Spring 风格配置及通过注解的配置方式。而对 MyBatis 的集成，目前只能通过 `myBatis-spring` 来完成，不过依然方便易用。

虽然 Spring 提供了使用各种 ORM 框架的方便类，但这离实际的应用开发还有一段距离。具体的应用一般会定义一个项目级的 DAO 基类，简化接口方法，添加泛型支持。一个设计良好的 DAO 基类可以大大减少 DAO 层的代码总量，提高项目的开发效率。



第 4 篇

应 用 篇



第 15 章

Spring Cache

伴随着信息量的爆炸式增长，以及构建的应用系统越来越多样化、复杂化，特别是伴随着近年来企业级应用互联网化的趋势，缓存（Cache）对应用程序性能的优化变得越来越重要。将所需服务请求的数据放到缓存中，既可以提高应用程序的访问效率，又可以减少数据库服务器的压力，从而让用户获得更为极致的体验。

Spring 开发者正是因为看到了缓存在应用中的重要地位，从 Spring 3.1 开始，就以一贯的优雅风格提供了一种透明的缓存解决方案，这使得 Spring 可以在后台使用不同的缓存框架（如 EhCache、GemFire、HazelCast 和 Guava）时保持编程的一致性。而从 Spring 4.0 开始则全面支持 JSR-107 annotations 和自定义的缓存标签。

本章主要内容：

- ◆ 了解缓存的基本概念
- ◆ Spring 对 Cache 的支持
- ◆ 自定义 Cache 注解的使用
- ◆ 与企业级缓存的集成

本章亮点：

- ◆ 比较详细地介绍了 Cache 在系统中的使用场景和方式
- ◆ 在实际应用中开发缓存程序的实战经验

15.1 缓存概述

15.1.1 缓存的概念

缓存作为系统架构中提升性能的一种重要支撑技术，在企业级应用中的地位越来越

突显。缓存技术日新月异，可选的缓存套件也琳琅满目，让人应接不暇。本节先不急于介绍 Spring Cache 的相关知识，而是先介绍缓存的一些基本概念，再引出 Spring Cache，以求先知“庐山真面目”，再从 Spring 的角度“一览众山小”。

其实在现实生活中也可以找到很多“缓存”的影子。比如，京东的物流为什么那么快，甚至可以实现当日送达，就因为它在全国各地都有分仓库，在发货的时候，会找离客户最近的仓库，如果该仓库有货物，则安排就近送货。用过 Maven 的朋友应该知道，在找依赖构件的时候，先从本机仓库找，如果没有再从本地服务器仓库找，最后才到远程服务器仓库找。

所以可将缓存定义为一种存储机制，它将数据保存在某个地方，并以一种更快的方式提供服务。较为常见的一种情况是在应用中使用缓存机制，以避免方法的多次执行，从而克服性能缺陷，也可减少应用服务器或者数据库的压力。缓存的策略有很多种，在应用系统中可根据实际情况选择，通常会把一些静态数据或者变化频率不高的数据放到缓存中，如配置参数、字典表等。而有些场景可能要寻求替代方案，比如，想提升全文检索的速度，在复杂的场景下建议使用搜索引擎，如 Solr 或 ElasticSearch。

通常在 Web 应用开发中，不同层级对应的缓存要求和缓存策略全然不同。如图 15-1 所示列举了系统不同层级对应的缓存技术选型。

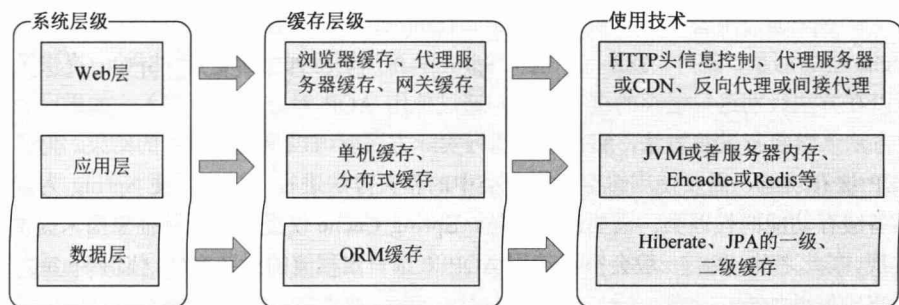


图 15-1 系统层级对应的缓存层级技术

要了解缓存，必须对其中的一些基本概念有所了解。下面先了解一下缓存中两个比较重要的基本概念。

1. 缓存命中率

即从缓存中读取数据的次数与总读取次数的比率。一般来说，命中率越高越好。

$$\text{命中率} = \frac{\text{从缓存中读取的次数}}{(\text{总读取次数}[\text{从缓存中读取的次数} + \text{从慢速设备上读取的次数}])}$$

$$\text{Miss 率} = \frac{\text{没有从缓存中读取的次数}}{(\text{总读取次数}[\text{从缓存中读取的次数} + \text{从慢速设备上读取的次数}])}$$

这是一个非常重要的监控指标，如果要做缓存，就一定要监控这个指标，来看缓存是否工作良好。还是之前举的例子，假设京东的某个分仓库通过监控发现某个仓库的命

中率一直很低，经常无法找到待发货的商品，那么这时候就需要调整策略，否则就会造成搏击里面“出拳百次，无一命中”的笑话。

2. 过期策略

即如果缓存满了，从缓存中移除数据的策略。常见的有 LFU、LRU、FIFO。

- ☐ FIFO (First In First Out): 先进先出策略，即先放入缓存的数据先被移除。
- ☐ LRU (Least Recently Used): 最久未使用策略，即使用时间距离现在最久的那个数据被移除。
- ☐ LFU (Least Frequently Used): 最近最少使用策略，即一定时间段内使用次数（频率）最少的那个数据被移除。
- ☐ TTL (Time To Live): 存活期，即从缓存中创建时间点开始直至到期的一个时间段（不管在这个时间段内有没有访问都将过期）。
- ☐ TTI (Time To Idle): 空闲期，即一个数据多久没被访问就从缓存中移除的时间。

至此，我们基本了解了缓存的一些基本知识。在 Java 中，一般会对调用方法进行缓存控制。比如调用“findUserById(String id)”，应该在调用这个方法之前先从缓存中查找有没有符合查询条件的数据，如果没有，则执行该方法从数据库中查找该用户，然后添加到缓存中，下次调用时将会从缓存中获取该数据。

从 Spring 3.1 开始，提供了类似于@Transactional 事务注解的缓存注解，且提供了 Cache 层的抽象。此外，JSR-107 也从 Spring 4.0 开始得到全面支持。Spring 提供了一种可以在方法级别进行缓存的缓存抽象。通过使用 AOP 对方法进行织入，如果已经为特定方法入参执行过该方法，那么不必执行实际方法就可以返回被缓存的结果。为了启用 AOP 缓存功能，需要使用缓存注解对类中的相关方法进行标记，以便 Spring 为其生成具备缓存功能的代理类。需要注意的是，Spring Cache 仅提供了一种抽象而未提供具体实现。在此之前，我们一般会自己使用 AOP 来做一定程度的封装实现，使用 Spring Cache 带来的好处如下：

- ☐ 支持开箱即用 (Out-Of-The-Box)，并提供基本的 Cache 抽象，方便切换各种底层 Cache。
- ☐ 类似于 Spring 提供的数据库事务管理，通过 Cache 注解即可实现缓存逻辑透明化，让开发者关注业务逻辑。
- ☐ 当事务回滚时，缓存也会自动回滚。
- ☐ 支持比较复杂的缓存逻辑。
- ☐ 提供缓存编程的一致性抽象，方便代码维护。



提示

Java 缓存规范 JCache API (JSR-107) 对 Java 对象缓存进行了标准化，方便高效开发，让程序员摆脱了实现缓存有效期、互斥、假脱机 (Spooling) 和缓存一致性等负担。该规范提供了 API、RI (参考实现) 和 TCK (技术兼容性套件)。读者可以通

过以下地址详细了解该规范：<https://jcp.org/en/jsr/detail?id=107>。同时，本章后面也会详细介绍该规范。有兴趣的读者可以关注 Java 的另一个规范 JSR-347——Java 数据网格规范。这两个规范容易让人混淆，但是 JSR-347 更多关注的是数据处理的分布式特性，如分布式存储、Map/Reduce 分布式计算等。

需要注意的是，Spring Cache 并不针对多进程的应用环境进行专门的处理，也就是说，当应用程序处于分布式或者集群环境下时，需要针对具体的缓存进行相应的配置。如 EhCache 可以通过 RMI、JGroups 及 EhCache Server 等方式来配置其多播集群环境。另外，在 Spring Cache 抽象的操作中没有锁的概念，当多线程并发操作（更新或者删除）同一个缓存项时，将可能得到过期的数据。有些缓存实现提供了锁的功能，如果需要考虑如上场景，则可以详细了解具体缓存的一些相关特性，如 EhCache 就提供了针对缓存元素 key 的 Read（读）、Write（写）锁。



轻松一刻

小情侣在打电话。女：“你说话的声音怎么怪怪的？”男：“在清除呼吸系统的缓存呢。”女：“说人话!!”男：“我在挖鼻孔……”

15.1.2 使用 Spring Cache

这里先展示一个自定义的缓存实现，即不通过任何第三方组件来实现对象的内存缓存，然后通过 Spring Cache 来实现缓存操作，以体会 Spring Cache 所带来的优雅和便捷性。

假设一个场景：在日常所见的项目中，用户查询是一个非常频繁的动作，从性能优化的角度，自然会想到对一个用户的查询方法做缓存，以避免频繁的数据库访问操作，提高页面的响应速度。通常的做法是以用户的 userId 作为 key，以返回的用户信息对象作为 value 值存储。而当以相同的 userId 查询用户时，程序将直接从缓存中获取结果并返回；否则更新缓存。

首先定义一个 User 实体类，该类具备基本的 userId、age 及 userName 属性，且具备 getter 和 setter 方法，如代码清单 15-1 所示。

代码清单 15-1 User.java

```
package com.smart.cache.domain;
import java.io.Serializable;

public class User implements Serializable {
    private String userId;
    private String userName;
    private int age;
```

```
//省略get/setXxx方法
```

Java 对象的缓存和序列化是息息相关的, 一般情况下, 需要被缓存的实体类需要实现 `Serializable`, 只有实现了 `Serializable` 接口的类, JVM 才可以对其对象进行序列化。对于 Redis、EhCache 等缓存套件来说, 被缓存的对象应该是可序列化的, 否则在网络传输、硬盘存储时都会抛出序列化异常。当然, 由于这里直接采用 Map 以内存的方式存储缓存对象, 所以 User 不实现 `Serializable` 接口也不会有问题, 但是实体类始终实现 `Serializable` 接口是一个好的编程习惯。实现 `Serializable` 接口的实体类, 一般要求声明一个 `serialVersionUID` 成员变量, 以表明该实体类的版本。如果实体类的结构发生变化, 则可以修改 `serialVersionUID` 值以支持反序列化工作。关于对象序列化和反序列化的更多知识, 读者可自行查找相关资料学习, 在此不再展开。

接下来定义一个缓存管理器, 这个管理器负责实现缓存逻辑, 支持对象的增加、修改和删除, 并且支持值对象的泛型, 具体实现如代码清单 15-2 所示。

代码清单 15-2 CacheManager.java

```
package com.smart.cache.mycache;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

public class CacheManager<T> {
    private Map<String,T> cache =
        new ConcurrentHashMap<String,T>();

    public T getValue(Object key) {
        return cache.get(key);
    }

    public void addOrUpdateCache(String key,T value) {
        cache.put(key, value);
    }

    public void evictCache(String key) { ① ← 根据 key 来删除缓存中的一条记录
        if(cache.containsKey(key)) {
            cache.remove(key);
        }
    }

    public void evictCache() { ② ← 清空缓存中的记录
        cache.clear();
    }
}
```

现在有了实体类 (User) 和一个缓存管理器 (CacheManager), 还需要一个提供用户查询的服务类, 此服务类使用缓存管理器来支持用户查询, 如代码清单 15-3 所示。

代码清单 15-3 UserService.java

```
package com.smart.cache.mycache;
import com.smart.cache.domain.User;
```



```

public class UserService {
    private CacheManager<User> cacheManager;

    public UserService() {

        //构造一个缓存管理器
        cacheManager = new CacheManager<User>();
    }

    public User getUserById(String userId) {

        //首先查询缓存
        User result = cacheManager.getValue(userId);
        if(result!=null) {
            System.out.println("get from cache..." +userId);

            //如果在缓存中，则直接返回缓存的结果
            return result;
        }

        // 否则到数据库中查询
        result = getFromDB(userId);
        if(result!=null) {

            //将数据库查询的结果更新到缓存中
            cacheManager.addOrUpdateCache(userId, result);
        }
        return result;
    }

    public void reload() {
        cacheManager.evictCache();
    }

    private User getFromDB(String userId) {
        System.out.println("real querying db..." +userId);
        return new User(userId);
    }
}

```

现在开始写一个测试类，用于测试刚才的方法，如代码清单 15-4 所示。

代码清单 15-4 UserMain.java

```

package com.smart.cache.mycache;

public class UserMain {
    public static void main(String[] args) {
        UserService userService = new UserService();

        //开始查询账号
        userService.getUserById("001001");//第一次查询，应该是数据库查询
        userService.getUserById("001001");//第二次查询，应该直接从缓存返回

        //重置缓存
        userService.reload();
    }
}

```

```

        System.out.println("after reload...");

        //应该是数据库查询
        userService.getUserById("001001");

        //第二次查询, 应该直接从缓存返回
        userService.getUserById("001001");
    }
}

```

按照分析, 执行结果应该是: 首先从数据库加载, 然后直接返回缓存中的结果; 重置缓存后, 应该首先从数据库加载, 然后返回缓存中的结果。实际的执行结果如下:

```

从数据库中查询...001001//第一次从数据库加载
从缓存中查询...001001//第二次从缓存加载
重置缓存...//清空缓存
从数据库中查询...001001//又从数据库加载
从缓存中查询...001001//从缓存加载

```

可以看出, 自定义缓存可以正常工作, 但这种实现方式并不优雅: 缓存代码和业务代码高度耦合, 业务代码中穿插着大量的缓存控制逻辑, 并且代码显式依赖缓存的具体实现。此外, 本实现方案并不支持按条件缓存, 比如只有符合某种条件的用户账号才需要缓存。如果要新增这一功能, 代码将进一步复杂化。

下面使用 Spring Cache 来实现上面这个例子, 基于刚才自定义缓存方案的实体类 User, 改造 UserService 和 UserMain, 如代码清单 15-5 所示。因为 Spring 已经提供了默认的缓存管理器, 所以这个例子不用实现自定义的缓存管理器。

代码清单 15-5 UserService.java

```

package com.smart.cache.simplecache;
import com.smart.cache.domain.User;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;

@Service(value = "userServiceBean")
public class UserService {

    //使用一个名为 users 的缓存
    @Cacheable(cacheNames = "users")
    public User getUserById(String userId) {

        //方法内部实现不考虑缓存逻辑, 直接实现业务
        System.out.println("real query user." + userId);
        return getFromDB(userId);
    }

    private User getFromDB(String userId) {
        System.out.println("real querying db..." + userId);
        return new User(userId);
    }
}

```

getUserById()方法标注了一个注解: @Cacheable(cacheNames="users"), 当调用这个方法的时候, 会先从 users 缓存中查询匹配的缓存对象, 如果存在则直接返回; 如果不

存在, 则执行方法体内的逻辑 (查询数据库), 并将返回值放入缓存中。对应缓存的 key 为 userId 的值, value 就是 userId 所对应的 User 对象, 缓存名称需要在 applicationContext.xml 中定义。

现在还需要一个 Spring 配置文件来支持基于注解的缓存, 如代码清单 15-6 所示。

代码清单 15-6 支持基于注解的缓存

```
<?xml version="1.0" encoding="UTF-8" ?>
<cache:annotation-driven />
<bean id="accountServiceBean" class="com.smart.cache.simplecache.UserService"/>
<bean id="cacheManager"
    class="org.springframework.cache.support.SimpleCacheManager">
    <property name="caches">
        <set>
            <bean
                class="org.springframework.cache.concurrent.ConcurrentMapCache
                FactoryBean" p:name="default" />
            <bean
                class="org.springframework.cache.concurrent.ConcurrentMapCache
                FactoryBean" p:name="users" />
        </set>
    </property>
</bean>
</beans>
```

Spring 通过 `<cache:annotation-driven />` 即可启用基于注解的缓存驱动, 这个配置项默认使用了一个定义为 `cacheManager` 的缓存管理器。SimpleCacheManager 是这个缓存管理器的默认实现, 它通过对其属性 `caches` 的配置来实现刚刚自定义的缓存管理器逻辑。从上面的代码中可以看到, 除了默认的 `default` 缓存外, 还自定义了一个名为 `users` 的缓存, 使用了默认的内存存储方案 `ConcurrentMapCacheFactoryBean`, 它是一个基于 `java.util.concurrent.ConcurrentHashMap` 的内存缓存实现方案。

接下来便可以重写 `UserMain`, 以观察使用 Spring Cache 运行的效果, 如代码清单 15-7 所示。

代码清单 15-7 UserMain.java

```
package com.smart.cache.simplecache;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class UserMain {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext(
            "applicationContext.xml"); // 加载 Spring 配置文件
        UserService userService = (UserService) context.getBean("userServiceBean");

        // 第一次查询, 应该查询数据库
        System.out.print("first query...");
        userService.getUserById("somebody");

        // 第二次查询, 应该不查询数据库, 直接返回缓存中的值
```

```

        System.out.print("second query...");
        userService.getUserById("somebody");
    }
}

```

上面的测试代码进行了两次查询，第一次应该查询数据库，第二次应该返回缓存中的值而不再查询数据库。执行上述代码，结果如下：

```

第一次查询...执行用户业务查询方法查找.用户//第一次查询
从数据库中查询...用户//对数据库进行了查询
第二次查询...//第二次查询，没有打印数据库查询日志，直接返回了缓存中的值

```

可以看出，我们配置的基于注解的缓存起作用了，而在 `UserService` 的代码中没有看到任何缓存逻辑代码，只需一个注解 `@Cacheable(cacheNames="users")`，就实现了基本的缓存方案，代码变得非常优雅、简洁。事实上，使用 Spring Cache 非常简单，开发人员只需完成两个步骤。

- ☐ 缓存定义：确定需要缓存的方法和缓存策略。
- ☐ 缓存配置：配置缓存。

接下来我们将深入了解 Spring Cache，以探究其中的奥秘。

15.2 掌握 Spring Cache 抽象

15.2.1 缓存注解

Spring Cache 提供了 5 种可在方法级别或类级别上使用的缓存注解。这些注解定义了哪些方法的返回值会被缓存或者从缓存中移除。注意，只有使用 `public` 定义的方法才可以被缓存，而 `private` 方法、`protected` 方法或者使用 `default` 修饰符的方法都不能被缓存。当在一个类上使用注解时，该类中每个公共方法的返回值都将被缓存到指定的缓存项中或者从中移除。本节将详细介绍这 5 种缓存注解的用法及所支持的属性。

1. @Cacheable

`@Cacheable` 是最主要的注解，它指定了被注解方法的返回值是可被缓存的。其工作原理是 Spring 首先在缓存中查找数据，如果没有则执行方法并缓存结果，然后返回数据。缓存名是必须提供的，可以使用引号、`Value` 或者 `cacheNames` 属性来定义名称。下面的定义展示了 `users` 缓存的声明及其注解的使用：

```

@Cacheable("users")
//Spring 3.x
@Cacheable(value= "users")
//Spring 4.0 新增了 value 的别名 cacheNames，比 value 命名更达意，推荐使用
@Cacheable(cacheNames = "users")

```

此外，还可以以列表的形式提供多个缓存，在该列表中使用逗号分隔缓存名称，并用花括号括起来。以下代码展示了在 `@Cacheable` 注解中定义的两个缓存 `cache1` 和 `cache2`：

```

@Cacheable(cacheNames = {"cache1", "cache2"})

```

下面所示的代码片段展示了如何在方法上应用@Cacheable 注解: getUser(String id) 方法以 id 值为键值将用户缓存到 users 缓存段中。此外,还可以提供自定义键,以便获取存储在某一缓存中的数据。

```
@Cacheable(cacheNames = "users")
public User getUser(String id) {
    return user.get(id);
}
```

1) 键生成器

缓存的本质就是键/值对集合。在默认情况下,缓存抽象使用方法签名及参数值作为一个键值,并将该键与方法调用的结果组成键/值对。如果在 Cache 注解上没有指定 key,则 Spring 会使用 KeyGenerator 来生成一个 key。

```
public interface KeyGenerator {
    Object generate(Object target, Method method, Object... params);
}
```

Spring 默认提供了 SimpleKeyGenerator 生成器。Spring 4.0 废弃了 3.x 的 DefaultKeyGenerator 而用 SimpleKeyGenerator 取代,原因是 DefaultKeyGenerator 在有多个入参时只是简单地把所有入参放在一起使用 hashCode() 方法生成 key 值,这样很容易造成 key 冲突。SimpleKeyGenerator 使用一个复合键 SimpleKey 来解决这个问题。接下来通过 Spring 源码来看看 Spring 生成 key 的规则。

```
public static Object generateKey(Object... params) {
    if (params.length == 0) {
        return SimpleKey.EMPTY;
    }
    if (params.length == 1) {
        Object param = params[0];
        if (param != null && !param.getClass().isArray()) {
            return param;
        }
    }
    return new SimpleKey(params);
}
```

通过源码来学习 Spring 往往简单明了,从上面的代码中可以发现其生成规则如下:

- (1) 如果方法没有入参,则使用 SimpleKey.EMPTY 作为 key。
- (2) 如果只有一个入参,则使用该入参作为 key。
- (3) 如果有多个入参,则返回包含所有入参的一个 SimpleKey。

此外,还可以在声明中指定键值。@Cacheable 注解提供了实现该功能的 key 属性,通过该属性,可以使用 SpEL 指定自定义键,如下:

```
@Cacheable(cacheNames = "users", key = "#user.userCode")
public User getUser(User user, boolean checkLogout)
```

如上所示,入参中有一个 boolean 值用于区分用户是否已经注销,而这个 boolean 值并不想作为 key 的一部分,那么可以根据 key 属性指定使用 userCode 作为缓存键。

当然,如果 key 生成策略涉及一些比较复杂的算法,而这个 key 生成策略又是通用的,则可以通过实现 org.springframework.cache.interceptor.KeyGenerator 接口来定义个性

化的 key 生成器。如自定义了一个 MyKeyGenerator 类并且实现了 KeyGenerator 接口以实现自定义的 key 生成器，那么便可如下使用：

```
@Cacheable(cacheNames = "users", keyGenerator = "myKeyGenerator")
public User getUser(User user, boolean checkWork)
```

2) 带条件的缓存

使用@Cacheable 注解的 condition 属性可按条件进行缓存，condition 属性使用了 SpEL 表达式动态评估方法入参是否满足缓存条件。关于 SpEL 表达式的相关内容，后文会进行介绍。

对于使用了@Cacheable 注解声明的 getUser()方法，我们启用了缓存判断条件：仅对年龄小于 35 周岁的用户启用缓存。

```
@Cacheable(cacheNames = "users", condition = "#user.age < 35")
public User getUser(User user) {
    return users.get(user.getId());
}
```

在上述代码中，#user 引用方法的同名入参变量，接着通过.age 访问 user 入参对象的 age 属性值。在调用方法前，将对注解中声明的条件进行评估，满足条件才缓存。

与 condition 属性相反，可使用 unless 属性排除某些不希望缓存的对象。下面的示例拒绝了对年龄大于或等于 35 周岁的用户进行缓存。

```
@Cacheable(value="users", unless = "#user.age >= 35")
public User getUser(User user) {
    return users.get(user.getId());
}
```

下面通过表 15-1 对@Cacheable 注解的参数进行详细说明。

表 15-1 @Cacheable注解参数说明

参 数	说 明	示 例
value/cacheNames	缓存的名称，在 Spring 配置文件中定义，必须指定至少一个	@Cacheable(cacheNames="cachename") 或者 @Cacheable(cacheNames={"cache1","cache2"})
key	缓存的 key，可以为空。如果指定，则按照 SpEL 表达式编写；如果不指定，则默认按照方法的所有参数进行组合	@Cacheable(cacheNames="cachename",key="#user Name")
condition	缓存的条件，可以为空，使用 SpEL 表达式编写，返回 true 或者 false，只有返回 true 才进行缓存.unless 属性与 condition 属性相反，满足条件则不进行缓存	@Cacheable(cacheNames="cachename",condition= "#userName.length()>2")

2. @CachePut

@CachePut 注解与@Cacheable 注解效果几乎一样，它首先执行方法，然后将返回值放入缓存。当希望使用方法返回值来更新缓存时，便可以选择这种方法，如下：

```
@CachePut(value = "users")
public User getUser(int id) {
    System.out.println("User with id " + id + " requested.");
    return users.get(id);
}
```


在上述代码中,首先执行 `getUser()` 方法,然后将结果值放入 `users` 缓存。与 `@Cacheable` 注解一样, `@CachePut` 注解也提供了 `key`、`condition` 和 `unless` 属性。

下面通过表 15-2 对 `@CachePut` 注解的参数进行详细说明。

表 15-2 `@CachePut` 注解参数说明

参 数	说 明	示 例
value/cacheNames	缓存的名称,在 Spring 配置文件中定义,必须指定至少一个	<code>@CachePut(cacheNames="cachename")</code> 或者 <code>@CachePut (cacheNames={"cache1","cache2"})</code>
key	缓存的 key,可以为空。如果指定,则按照 SpEL 表达式编写;如果不指定,则默认按照方法的所有参数进行组合	<code>@CachePut(cacheNames="cachename",key="#userName")</code>
condition	缓存的条件,可以为空,使用 SpEL 表达式编写,返回 true 或者 false,只有返回 true 才进行缓存。 <code>unless</code> 属性与 <code>condition</code> 属性相反,满足条件则不进行缓存	<code>@CachePut(cacheNames="cachename",condition="#userName.length()>2")</code>

需要注意的是,在同一个方法内不能同时使用 `@CachePut` 和 `@Cacheable` 注解,因为它们拥有不同的特性。当 `@Cacheable` 注解跳过方法直接获取缓存时, `@CachePut` 注解会强制执行方法以更新缓存,这会导致意想不到的情况发生,如当注解都带入了条件属性,就会使得它们彼此排斥。还需要注意的是, `@CachePut` 注解的 `condition` 属性设置的缓存条件也不应该依赖于方法返回的结果(如 `condition="#result"`),因为缓存条件是在方法执行前预先验证的。

3. `@CacheEvict`

`@CacheEvict` 注解是 `@Cacheable` 注解的反向操作,它负责从给定的缓存中移除一个值。大多数缓存框架都提供了缓存数据的有效期,使用该注解可以显式地从缓存中删除失效的缓存数据。该注解通常用于更新或者删除用户的操作。下面的方法定义从数据库中删除一个用户,而 `@CacheEvict` 注解也完成了相同的工作,从 `users` 缓存中删除了被缓存的用户。

```
@CacheEvict("users")
public void removeUser(int id) {
    users.remove(id);
}
```

与 `@Cacheable` 注解一样, `@CacheEvict` 注解也提供了 `key` 和 `condition` 属性,通过这些属性可以使用 SpEL 表达式指定自定义的键和条件。

此外, `@CacheEvict` 注解还具有两个与 `@Cacheable` 注解不同的属性: `allEntries` 属性定义了是否移除缓存的所有条目,其默认行为是不移除这些条目; `beforeInvocation` 属性定义了是在调用方法之前还是在调用方法之后完成移除操作。与 `@Cacheable` 注解不同的是,在默认情况下, `@CacheEvict` 注解在方法调用之后运行。



轻松一刻

五彩缤纷的宏观世界都是由质子、中子和电子所构成的，这些粒子因而被称为基本粒子，意指它们是构造世间万物的基本砖块。事实上，基本粒子世界并没有这么简单，任何基本粒子都存在相应的反粒子。当正反物质相遇时，双方就会相互湮灭抵消，发生爆炸并产生巨大能量。在丹·布朗的小说《天使与魔鬼》里，恐怖分子企图从欧洲核子中心盗取 0.25 克反物质，进而欲炸毁整座梵蒂冈城。

1) allEntries 属性

allEntries 是布尔类型的，用来表示是否需要清除缓存中的所有元素。默认值为 **false**，表示不需要。当指定 **allEntries** 为 **true** 时，Spring Cache 将忽略指定的 **key**，清除缓存中的所有内容。具体使用方法如下：

```
@CacheEvict(cacheNames="users", allEntries=true)
public void delete(String userId) {
    System.out.println("delete user by id: " + userId);
}
```

2) beforeInvocation 属性

需要知道，清除操作默认是在对应方法执行成功后触发的，即方法如果因为抛出异常而未能成功返回时则不会触发清除操作。使用 **beforeInvocation** 属性可以改变触发清除操作的时间。当指定该属性值为 **true** 时，Spring 会在调用该方法之前清除缓存中的指定元素。具体使用方法如下：

```
@CacheEvict(cacheNames="users", beforeInvocation=true)
public void delete(String userId) {
    System.out.println("delete user by id: " + userId);
}
```

需要注意的是，在相同的方法上使用 **@Cacheable** 和 **@CacheEvict** 注解并使用它们指向相同的缓存没有任何意义，因为这相当于数据被缓存之后又被立即移除了，所以需要避免在同一方法上同时使用这两个注解。

下面通过表 15-3 对 **@CacheEvict** 注解的参数进行详细说明。

表 15-3 @CacheEvict注解参数说明

参 数	说 明	示 例
value/cacheNames	缓存的名称，在 Spring 配置文件中定义，必须指定至少一个	@CacheEvict(cacheNames="cachename") 或者 @CacheEvict(cacheNames={"cache1","cache2"})
key	缓存的 key，可以为空。如果指定，则按照 SpEL 表达式编写；如果不指定，则默认按照方法的所有参数进行组合	@CacheEvict(cacheNames="cachename",key="#userName")
condition	缓存的条件，可以为空，使用 SpEL 表达式编写，返回 true 或者 false，只有返回 true 才清空缓存。unless 属性与 condition 属性相反，满足条件则不进行缓存	@CacheEvict(cacheNames="cachename",condition="#userName.length()>2")

续表

参 数	说 明	示 例
allEntries	是否清空所有缓存内容，默认为 false。如果指定为 true，则方法调用后将立即清空所有缓存	@CacheEvict(cacheNames="cachename", allEntries=true)
beforeInvocation	是否在方法执行前就清空，默认为 false。如果指定为 true，则在方法还没有执行的时候就清空缓存。在默认情况下，如果方法执行抛出异常，则不会清空缓存	@CacheEvict(cacheNames="cachename", beforeInvocation=true)

4. @Caching

@Caching 是一个组注解，可以为一个方法定义提供基于@Cacheable、@CacheEvict 或者@CachePut 注解的数组。为了方便说明@Caching 注解的使用方法，示例定义了 User、Member（会员）和 Visitor（游客）3 个实体类，它们彼此之间有一个简单的层次结构：User 是一个抽象类，而 Member 和 Visitor 类扩展了该类。

在代码清单 15-8 中，UserService 类是一个 Spring 服务 Bean，包含了 getUser()方法。同时声明了两个@Cacheable 注解，并使其指向两个不同的缓存项：members 和 visitors。然后根据两个@Cacheable 注解定义中的条件对方法的参数进行检查，并将对象存储在 members 或 visitors 缓存中。

代码清单 15-8 UserService.java

```
package com.smart.cache.cachegroup;
import com.smart.cache.domain.User;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.cache.annotation.Caching;
import org.springframework.stereotype.Service;
import java.util.HashMap;
import java.util.Map;

@Service(value = "cacheGroupUserServcie")
public class UserService {
    private Map<Integer, User> ppl = new HashMap<Integer, User>();
    {
        ppl.put(1, new Member("1", "w1"));
        ppl.put(2, new Visitor("2", "w2"));
    }

    // @Caching 组的使用
    @Caching(cacheable = {
        @Cacheable(value = "members", condition = "#obj instanceof T(com.smart.cache.cachegroup.Member)"),
        @Cacheable(value = "visitors", condition = "#obj instanceof T(com.smart.cache.cachegroup.Visitor)")
    })
    public User getUser(User obj) {
        return ppl.get(Integer.valueOf(obj.getUserId()));
    }
}
```

5. @CacheConfig

在 Spring 4.0 之前并没有类级别的全局缓存注解。前面我们所了解的注解都是基于方法的,如果在同一个类中需要缓存的方法注解属性都相似,则需要一个个地重复增加。Spring 4.0 增加了@CacheConfig 类级别的注解来解决这个问题。一个典型的使用示例如代码清单 15-9 所示。

代码清单 15-9 UserService.java

```
package com.smart.cache.config;
import com.smart.cache.domain.User;
import org.springframework.cache.annotation.CacheConfig;
import org.springframework.cache.annotation.Cacheable;

@CacheConfig(cacheNames = "users",keyGenerator="MyKeyGenerator ")
public class UserService {
    @Cacheable
    public User findA(User user) {
        ...
    }
    @Cacheable
    public User findB(User user) {
        ...
    }
}
```

可以看到,在@CacheConfig 注解中定义了类级别的缓存 users 和自定义 key 生成器,那么在 findA()和 findB()方法中不再需要重复指定,而是默认使用类级别的定义。

15.2.2 缓存管理器

CacheManager 是 SPI (Service Provider Interface, 服务提供程序接口), 提供了访问缓存名称和缓存对象的方法,同时也提供了管理缓存、操作缓存和移除缓存的方法。本节将列举 Spring Cache 框架所提供的不同的缓存管理器实现。

1. SimpleCacheManager

通过使用 SimpleCacheManager 可以配置缓存列表,并利用这些缓存进行相关的操作。因为 SimpleCacheManager 是缓存管理器的简化版本,所以在本章的所有示例中都将使用该实现类。下面的代码片段是针对该缓存管理器的一个配置示例。对应缓存的定义,我们使用了 ConcurrentMapCaheFactoryBean 类来对 ConcurrentMapCache 进行实例化,该实例使用了 JDK 的 ConcurrentMap 实现。

```
<bean id="cacheManager" class="org.springframework.cache.support.SimpleCacheManager">
    <property name="caches">
        <set>
            <bean id="users"
                class="org.springframework.cache.concurrent.ConcurrentMapCache
                FactoryBean" />
        </set>
    </property>
</bean>
```

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

2. NoOpCacheManager

NoOpCacheManager 主要用于测试目的，但实际上它并不缓存任何数据。下面的代码给出了该缓存管理器的配置定义，我们没有为该管理器提供缓存列表，因为它仅仅作为测试目的。

```
<bean id="cacheManager" class="org.springframework.cache.support.NoOpCacheManager" />
```

3. ConcurrentMapCacheManager

ConcurrentMapCacheManager 使用了 JDK 的 ConcurrentMap。它提供了与前面介绍的 SimpleCacheManager 类似的功能，但并不需要像前面那样定义缓存。该缓存定义如下：

```
<bean id="cacheManager"
      class="org.springframework.cache.concurrent.ConcurrentMapCacheManager" />
```

4. CompositeCacheManager

CompositeCacheManager 能够定义多个缓存管理器。当在应用程序上下文中声明 <cache:annotation-driven> 标记时，它只提供一个缓存管理器，这往往并不能满足用户的需求。而 CompositeCacheManager 定义将多个缓存管理器定义组合在一起，从而扩展了该功能。此外，CompositeCacheManager 还提供了一种机制，通过使用 fallbackToNoOpCache 属性回到 NoOpCacheManager。下面所示的定义是一个 CompositeCacheManager 定义，将一个简单的缓存管理器与 HazelCast 缓存管理器捆绑在一起。简单的缓存管理器定义了 members 缓存，而 HazelCast 缓存管理器则为 visitors 定义了缓存管理器。配置 HazelCast 缓存管理器的详细过程将在本章稍后介绍。下面的示例展示了可在不同的缓存中存储不同类型的对象，而不同的缓存则由不同的缓存管理器进行管理。

```
<bean id="cacheManager"
      class="org.springframework.cache.support.CompositeCacheManager">
  <property name="cacheManagers">
    <list>
      <bean class="org.springframework.cache.support.SimpleCacheManager">
        <property name="caches">
          <set>
            <bean id="members" class="org.springframework.cache.concurrent
              Concurrent MapCacheFactoryBean" />
          </set>
        </property>
      </bean>
      <bean class="com.hazelcast.spring.cache.HazelcastCacheManager">
        <constructor-arg ref="hazelcast" />
      </bean>
    </list>
  </property>
</bean>
```

15.2.3 使用 SpEL 表达式

在 Spring Cache 注解属性中（如 key、condition 和 unless），Spring 的缓存抽象使用了 SpEL 表达式，从而提供了属性值的动态生成及足够的灵活性。下面代码所示的方法根据用户编码进行了缓存。对于 key 属性，使用了表达式来自定义键的生成。

```
@Cacheable(cacheNames="users",key = "#user.userId")
public User getUser(User user) {
    return users.get(user.getUserId());
}
```

在下面的代码片段中，还应用了一个条件，以便对那些年龄小于 35 周岁的用户进行缓存。

```
@Cacheable(value = "users", condition = "#user.age < 35")
public User getUser(User user) {
    System.out.println("User with id " + user.getUserId() + " requested.");
    return users.get(user.getUserId());
}
```

SpEL 表达式可基于上下文并通过使用缓存抽象，提供与 root 对象相关联的缓存特定的内置参数。表 15-4 描述了该表达式。

表 15-4 SpEL表达式

名 字	位 置	描 述	示 例
methodName	root 对象	当前被调用的方法名	#root.methodname
method	root 对象	当前被调用的方法	#root.method.name
target	root 对象	当前被调用的目标对象实例	#root.target
targetClass	root 对象	当前被调用的目标对象的类	#root.targetClass
args	root 对象	当前被调用的方法的参数列表	#root.args[0]
caches	root 对象	当前方法调用使用的缓存列表	#root.caches[0].name
Argument name	执行上下文	当前被调用的方法的参数，如 findByUser(User user)，可以通过#user.id 获得参数	#user.id
result	执行上下文	方法执行后的返回值（仅当方法执行之后的判断有效，如'unless'、'cache evict'的 beforeInvocation=false）	#result

15.2.4 基于 XML 的 Cache 声明

如果不想使用注解或者由于其他原因而无法获得项目的源码（如二次开发时经常没有源码的访问权限），也可以用 XML 的方式配置 Spring Cache。其配置方式和 transaction 管理器的 advice 类似，如下例：

```
<!-- 定义需要使用缓存的类 -->
<bean id="userService" class="com.smart.service.UserService"/>
<!-- 缓存定义 -->
<cache:advice id="cacheAdvice" cache-manager="cacheManager">
    <cache:caching cache="users">
        <cache:cacheable method="findUser" key="#userId"/>
    </cache:caching>
</cache:advice>
```



```

<cache:cache-evict method="loadUsers" all-entries="true"/>
</cache:advice>
</cache:advice>
<aop:config>
<aop:advisor advice-ref="cacheAdvice"
    pointcut="execution(*com.smart.service.UserService.*(..))"/>
</aop:config>

```

如上配置文件为 UserService 开启了缓存。使用 cache:advice 定义包装了两个需要使用缓存的方法，其中 findUser() 定义了 Cacheable，而 loadUsers() 定义了 CacheEvict，并且定义了公共的缓存 users。

aop:config 定义了 cacheAdvice 的切入点（关于 AOP 的定义，具体请参考第 8 章）。XML 声明式配置支持所有注解的方法，因此二者之间可以很容易地替换，当然也可以在项目中同时使用这两种方式。

15.2.5 以编程方式初始化缓存

在实际项目中，有时可能需要在完成缓存的初始化。最典型的示例是当启动并且运行应用程序时将数据加载到缓存中。实现该方法很简单，首先访问缓存管理器，然后将数据手工加载到不同的缓存中（这些缓存根据缓存名称进行区分）。在下面的示例中，当初始化应用程序上下文时，将用户列表加载到缓存区域中。

代码清单 15-10 展示了在 Spring Bean 的 @PostConstruct 注解方法中初始化缓存 users。此外，该 Bean 还包含了 getUser() 方法，并且已被注解用于缓存操作。

代码清单 15-10 UserService.java

```

package com.smart.cache.initcache;
import com.smart.cache.domain.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cache.Cache;
import org.springframework.cache.CacheManager;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;
import javax.annotation.PostConstruct;
import java.util.HashMap;
import java.util.Map;

@Service(value = "initUserServcie")
public class UserService {
    private Map<Integer, User> users = new HashMap<Integer, User>();
    {
        users.put(1, new User("1", "w1"));
        users.put(2, new User("2", "w2"));
    }

    private CacheManager cacheManager;

    @Autowired
    public void setCacheManager(CacheManager cacheManager) {

```

```

        this.cacheManager = cacheManager;
    }

    @PostConstruct
    public void setup() {
        Cache usersCache = cacheManager.getCache("users");
        for (Integer key : users.keySet()) {
            usersCache.put(key, users.get(key));
        }
    }

    @Cacheable(value = "users")
    public User getUser(int id) {
        System.out.println("User with id " + id + " requested.");
        return users.get(id);
    }
}

```

接下来创建 `UserService` 类作为 Spring 服务 Bean，并且自动注入 `CacheManager`，然后在 `@PostConstruct` 中通过键值把所有数据放置到缓存中，该键值将用于检索。

下面通过基于 Java 类的配置方式准备好 Spring 容器配置信息，如代码清单 15-11 所示。

代码清单 15-11 ApplicationConfig.java

```

package com.smart.cache.initcache;
import org.springframework.cache.CacheManager;
import org.springframework.cache.annotation.EnableCaching;
import org.springframework.cache.concurrent.ConcurrentMapCache;
import org.springframework.cache.support.SimpleCacheManager;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Configuration;
import java.util.Arrays;

@Configuration
@ComponentScan(basePackages = {"com.smart.cache"})
@EnableCaching
public class ApplicationConfig {
    @Bean
    public CacheManager cacheManager() {
        SimpleCacheManager cacheManager = new SimpleCacheManager();
        cacheManager.setCaches(Arrays.asList(new ConcurrentMapCache("users")));
        return cacheManager;
    }
}

```

用 `UserMain` 测试以上缓存实现的效果，如代码清单 15-12 所示。

代码清单 15-12 UserMain.java

```

package com.smart.cache.initcache;
import com.smart.cache.domain.User;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

```

```

public class UserMain {
    public static void main(String... args) {
        ApplicationContext context = new
            AnnotationConfigApplicationContext(ApplicationConfig.class);
        UserService userService = (UserService) context.getBean("initUserServcie");

        User userFetch1 = userService.getUser(1);
        System.out.println(userFetch1);
        User userFetch2 = userService.getUser(2);
        System.out.println(userFetch2);
    }
}

```

在代码清单 15-12 中，首先从上下文中查找 `initUserServcie` Bean，然后连续调用 `getUser(1)`和 `getUser(2)`方法。可以看到，此时已查找到缓存中的相关信息，如下：

```

User{id=1, name='w1', age=0}
User{id=2, name='w2', age=0}

```

15.2.6 自定义缓存注解

之前介绍的`@Caching` 组合也许会让方法上的注解显得比较杂乱，Spring 提供了自定义注解，可把这些注解组合到一个注解类中，从而解决这个问题，如下代码所示：

```

@Caching(
    put = {
        @CachePut(cacheNames = "user", key = "#user.id"),
        @CachePut(cacheNames = "user", key = "#user.username"),
        @CachePut(cacheNames = "user", key = "#user.email")
    }
)
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
public @interface UserSaveCache {
}

```

这样只需在方法上使用如下代码即可，整个代码显得比较干净。

```

@UserSaveCache
public User save(User user)

```

15.3 配置 Cache 存储

在企业级应用中往往会有更复杂的功能和性能需求，所以在日常开发过程中，大部分情况下会使用第三方的缓存实现，而不是 `SimpleCacheManager` 的简单实现。在企业级 Java 领域，Spring 缓存提供了与不同缓存框架的集成支持。

15.3.1 EhCache

EhCache 是广泛使用的 Java 开源缓存框架之一，本节将介绍如何通过 Spring Cache 来集成 EhCache，以便在项目中便捷地使用。

在项目中增加 EhCache 的依赖。我们选择的是 2.8.3 版本，该版本最值得关注的属性包括 Off-Heap 存储和 JSR-107 兼容。

```
<dependency>
  <groupId>net.sf.ehcache</groupId>
  <artifactId>ehcache</artifactId>
  <version>2.8.3</version>
</dependency>
```

接着需要在 applicationContext-ehcache 配置文件中对 EhCache 进行如下配置：

```
<bean id="cacheManager" class="org.springframework.cache.ehcache.EhCacheCacheManager"
  p:cacheManager="ehcache"/>
<bean id="ehcache" class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean"
  p:configLocation="classpath:ehcache.xml"/>
```

在配置文件中声明了 cacheManager Bean。通过使用 EhCache，定义 cacheManager 是非常简单和直接的。它包含了另外一个名为 ehcache 的 Bean，并使用配置文件 ehcache.xml 对其进行配置。下面给出一个最简单的 ehcache.xml 配置文件，其中仅包含了 users 缓存的定义。

```
<ehcache>
  <cache name="users" maxElementsInMemory="1000" />
</ehcache>
```

Ehcache 还提供了 TTL 或者 Eviction Policy 之类的功能，并且可以对这些功能进行配置。而相关配置应该直接通过缓存提供程序完成，因为缓存抽象没有为这些功能提供任何配置（这些功能不可能被不同的提供程序所支持，如 JDK 的 ConcurrentMap）。如果想进一步配置，请参考 EhCache 的相关文档。

15.3.2 Guava

从 Spring 4.0 开始支持 Guava，Guava 是 Google 贡献的一个开源公共库（Guava 非常好用，建议开发者了解并使用之），它提供了缓存功能。下面的代码片段显示了针对 Guava 的 Maven 依赖项，版本是 18.0。

```
<dependency>
  <groupId>com.google.guava</groupId>
  <artifactId>guava</artifactId>
  <version>18.0</version>
</dependency>
```

配置 GuavaCacheManager 很简单，只需定义 cacheManager Bean 就可以获取配置并运行。同时，没必要定义缓存，因为它们将在需要时被创建。

```
<bean id="cacheManager" class="org.springframework.cache.guava.GuavaCacheManager" />
```

15.3.3 HazelCast

HazelCast 是业界流行的 Java 分布式内存网格框架之一。与其他框架一样, HazelCast 也提供了基于 Spring 缓存抽象的缓存管理器, 这里使用的是其 3.3 版本。

```
<dependency>
  <groupId>com.hazelcast</groupId>
  <artifactId>hazelcast-all</artifactId>
  <version>3.3</version>
</dependency>
```

目前, HazelcastCacheManager 位于 hazelcast-all 的工具包中, 但并没有包括在 spring-context-support 中。如下面的代码所示, HazelcastCacheManager 引用了另一个 Bean 作为其打包的 cacheManager。

```
<bean id="cacheManager" class="com.hazelcast.spring.cache.HazelcastCacheManager">
  <constructor-arg ref="hazelcast" />
</bean>
```

借助 hazelcast Bean, 通过创建普通的 users 缓存为 HazelCast 完成配置。

```
<hz:hazelcast id="hazelcast">
  <hz:config>
    <hz:map name="users">
      <hz:map-store enabled="true" class-name="com.smart.cache.domain.User
        write-delay-seconds="0"/>
    </hz:map>
  </hz:config>
</hz:hazelcast>
```

HazelCast 需要实体类实现 Serializable 接口, 并包含默认的构造函数。

15.3.4 GemFire

GemFire 是一个需要商业许可的数据管理平台, 它提供了基于分布式体系结构的数据访问功能。GemFire 同时提供了一个内存数据网格, 具备极高的吞吐量、低延迟的数据访问及可扩展性。

配置 GemFire 也很简单, 如下代码所示:

```
<gfe:cache id="gemfire-cache"/>
<bean id="cacheManager" class="org.springframework.data.gemfire.support.Gemfire
  CacheManager" p:cache-ref="gemfire-cache">
</bean>
```

关于 GemFire, 如果读者想深入了解, 则可以参考 Spring Data 项目中关于 GemFire 的描述 (可参考官方链接: <http://docs.spring.io/spring-data/gemfire/docs/1.5.2.RELEASE/reference/html/>)。

15.3.5 JSR-107 Cache

从 Spring 4.0 开始全面支持 JSR-107 规范, 该规范包含 @CacheResult、@CacheEvict、

@CacheRemove 和 @CacheRemoveAll 及配套的 @CacheDefaults、@CacheKey 和 @CacheValue 注解。无须把缓存存储迁移到 JSR-107 中，就可以使用 JSR-107 规范的缓存注解，因为 Spring 的缓存抽象为 JSR-107 提供了默认的 CacheResolver 和 KeyGenerator 实现。

表 15-5 梳理了 Spring 缓存注解和相对应的 JSR-107 规范注解之间的一些差异。

表 15-5 Spring Cache和JSR-107 的对比

Spring	JSR-107	备 注
@Cacheable	@CacheResult	基本一致，但是@CacheResult 提出了缓存异常的概念，可以通过 exceptionCacheName 属性指定缓存异常，当异常再次发生时会被抛出，而不会再去调用方法
@CachePut	@CachePut	Spring 会在调用更新方法后缓存结果，而 JCache 需要在方法入参中传递一个@CacheValue 注解参数，JCache 允许通过注解属性 afterInvocation 指定缓存在方法执行前或执行后更新缓存
@CacheEvict	@CacheRemove	@CacheRemove 支持当调用方法抛出异常时，可以按照条件移除缓存
@CacheEvict (allEntries=true)	@CacheRemoveAll	和@CacheRemove 一样
@CacheConfig	@CacheDefaults	类级别的注解，用于配置类级别的通用属性

Java Caching 定义了 4 个核心接口，分别是 CachingProvider、CacheManager、Cache 和 Entry。下面对这 4 个核心接口进行简单介绍。

- ❑ CachingProvider 定义了创建、配置、获取、管理和控制多个 CacheManager。一个应用可以在运行期访问多个 CachingProvider。
- ❑ CacheManager 定义了创建、配置、获取、管理和控制多个唯一命名的 Cache，这些 Cache 存在于 CacheManager 的上下文中。一个 CacheManager 仅被一个 CachingProvider 所拥有。
- ❑ Cache 是一个类似 Map 的数据结构并存储以 key 为索引的值。一个 Cache 仅被一个 CacheManager 所拥有。
- ❑ Entry 是一个存储在 Cache 中的键值对。每个存储在 Cache 中的条目都有一个定义的有效期，即 Expiry Duration。一旦超过这个时间，条目即为过期状态。一旦过期，条目将不可访问。缓存有效期可以通过 ExpiryPolicy 设置。

JCache 的 javax.cache.annotation.CacheResolver 和 Spring Cache 的 CacheResolver 接口类似，唯一不同的是 JCache 只支持单个 Cache，这点从注解属性命名上就可以看出，Spring 是 cacheNames，而 JCache 是 cacheName。一般情况下通过注解参数中的定义去获取缓存；如果未指定，则 JCache 默认取入参的第一个参数作为缓存名称。

CacheResolver 的实例从 CacheResolverFactory 中获取，可以为每个方法定制不同的 CacheResolver，如下：

```
@CacheResult(cacheNames="users",cacheResolverFactory=MyCacheResolverFactory.class)
public User finduser(User user)
```


JCache 的生成器和 Spring 的 KeyGenerator 一样, JCache 定义了 javax.cache.annotation.CacheKeyGenerator 接口。如果没有在参数中注入 @CacheKey, 则默认取所有的参数用于生成 key, 这点和 Spring Cache 也极为类似, 如下:

```
@Cacheable(cacheNames="users", key="#userId")
public User finduser(User user, boolean checkLogout)
@CacheResult(cacheName="users")
public User finduser(@CacheKey user user, boolean checkLogout)
```

JCache 可以管理在注解方法中抛出的异常, 以便阻止缓存的更新。它也可以根据缓存指定的异常, 在下次抛出异常时不再执行方法本身。

假设有一个错误的入参被传入, 而会导致抛出 InvalidUserNotFoundException 异常, 那么在下次该错误入参被再次传入时, Spring 会直接抛出异常而不会再调用方法本身, 如下:

```
@CacheResult(cacheName="users", exceptionCacheName="failures"
    @CacheExceptions = InvalidUserNotFoundException.class)
public User findUser(User user)
```

JSR-107 的缓存实现同样可以使用 Spring 的缓存抽象。JCache 的缓存实现在 org.springframework.cache.jcache 包中。要使用它, 只需要配置相应的缓存管理器即可, 如下:

```
<bean id="cacheManager"
    class="org.springframework.cache.jcache.JCacheCacheManager"
    p:cache-manager-ref="jCacheManager"/>
<!-- JSR-107 cache manager -->
<bean id="jCacheManager" .../>
```

15.4 实战经验

1. 非 public 方法问题

和内部调用问题类似, 非 public 方法如果想实现基于注解的缓存, 则必须采用基于 AspectJ 的 AOP 机制。这里限于篇幅不再详述, 可参考第 8 章。

2. 基于 Proxy 的 Spring AOP 带来的内部调用问题

前面介绍了 Spring Cache 的原理, 即它是基于动态生成子类的代理机制来对方法的调用进行切面的, 这里的关键点是对对象的引用问题。如果对象的方法是内部调用 (this 引用) 而不是外部引用, 则会导致代理失效, 那么切面就失效, 也就是说上面定义的各种注解, 包括 @Cacheable、@CachePut 和 @CacheEvict 都会失效。来看下面的示例:

```
public User getUserByName2(String userName) {
    return getUserByName(userName);
}
```

// 使用了一个缓存, 名为 users

```
@Cacheable(cacheNames = "users")
```

```
public User getUserByName(String userName) {
```

```
//方法内部实现不考虑缓存逻辑，直接实现业务
return getFromDB(userName);
}
```

上面定义了一个新的方法 `getUserByName2()`，其自身调用了 `getUserByName()` 方法，这时发生的是内部调用（`this`），所以没有使用代理类，导致 Spring Cache 失效。要避免这个问题，就要避免对缓存方法的内部调用，或者避免使用基于代理机制的 AOP 模式，也可以使用基于 AspectJ 的 AOP 模式来解决这个问题。如果想使用基于代理机制的 AOP 模式，要想避免此问题，则可以参考 7.5.4 节介绍的方法。

3. @CacheEvict 的可靠性问题

我们看到，`@CacheEvict` 注解有一个属性 `beforeInvocation`，默认为 `false`，即默认情况下都是在实际的方法执行完成后才对缓存进行清空操作。期间如果执行方法出现异常，则会导致缓存清空而不被执行。

```
//清空 users 缓存
@CacheEvict(cacheNames = "users", allEntries=true)
public void reload() {
    throw new RuntimeException();
}
```

注意上面的代码，在 `reload` 的时候抛出了运行期异常，这会导致清空缓存失败。

4. 运行期开发

有的时候，在代码迁移、调试或者部署的时候，恰好没有可运行的缓存服务容器，比如 `MemCache` 还不具备条件、`GemFire` 还没有安装好等，如果这时候想调试代码，基本上非常困难。这里有一个办法，在不具备缓存条件的时候，在不更改代码的前提下，禁用缓存。

方法就是修改 `spring*.xml` 配置文件，设置一个找不到缓存就不执行任何操作的标志位，如下：

```
<bean id="cacheManager"
class="org.springframework.cache.support.CompositeCacheManager">
    <property name="cacheManagers">
        <list>
            <ref bean="jdkCache"/>
            <ref bean="gemfireCache"/>
        </list>
    </property>
    <property name="fallbackToNoOpCache" value="true"/>
</bean>
```

15.5 小结

本章介绍了缓存的基本概念，以及 Spring 通过其缓存抽象所提供的相关功能。同时介绍了 5 个可用来缓存数据或者从缓存中移除数据的重要注解（`@Cacheable`、

@CacheEvict、@CachePut、@Caching、@CacheConfig)。此外还介绍了采用 XML 声明式和注解式两种方式使用 Spring Cache。列举了可抽象使用的不同缓存管理器，同时通过 SpEL 示例强调了定义缓存存储时表达式的重要性。

我们还学习了在应用程序启动期间自动初始化缓存，而这恰恰是企业应用程序中常见的用例。最后一节列举了主流的缓存程序实现，如 EhCache、Guava 和 HazelCast、GemFire、JSR-107，并详细介绍了它们与 Spring Cache 如何集成。最后笔者总结了在实际项目中常见的一些问题，并加以分析。本章对 Spring Cache 的源码和原理也有一定的分析，相信读者在理解之后，也能基于 Spring Cache 抽象定制出符合自己应用的缓存方案。

第 16 章

任务调度和异步执行器

任务调度是大多数应用系统的常见需求之一，直接自己编写基于线程的调度程序，不但容易出错，而且实现难度很大。借助巨人的肩膀，可以站得更高、看得更远。Quartz 是任务调度领域享誉盛名的开源框架，Spring 提供了集成 Quartz 的功能，可以让开发人员以更面向 Spring 的方式创建基于 Quartz 的任务调度应用。此外，Spring 也为 JDK Timer、Java 5.0 和 Executor 提供了有益的支持。

本章主要内容：

- ◆ Quartz 知识和 Spring 的支持
- ◆ JDK Timer 知识和 Spring 的支持
- ◆ Executor 知识和 Spring 所提供的抽象
- ◆ 在实际应用中开发任务调度程序所需注意的问题

本章亮点：

- ◆ 比较详细地讲述了 Quartz 框架
- ◆ 在实际应用中开发任务调度程序的实战经验

16.1 任务调度概述

各种企业应用几乎都会遇到任务调度的需求，以论坛为例，每隔半小时生成精华文章的 RSS 文件，每天凌晨统计论坛用户的积分排名，每隔 30 分钟对锁定到期的用户进行解锁。对于一个典型的企业应用系统来说，每月 1 日凌晨统计上个月各部门的业务数据生成月报表，每隔半小时查询用户是否有快到期的待处理业务等，这样的例子俯拾皆是、不胜枚举。

以上所举调度场景的核心都是以时间为关注点的调度，即在特定的时间点执行指定

的操作。如果将任务调度的范围稍微扩大一些，则还应该包括资源上的调度。如 Web Server 在接收到请求时，会立即创建一个新的线程服务该请求。但资源是有限的，无限制地使用必然耗尽亏空，大多数系统都要对资源使用进行控制。首先必须限制服务线程的最大数目；其次可以考虑使用线程池以便共享服务的线程资源，降低频繁创建、销毁线程的消耗。

任务调度本身涉及多线程并发、运行时间规则制定及解析、运行现场保持与恢复、线程池维护等诸多方面的工作。如果直接使用自定义线程这种刀耕火种的原始办法，则开发任务调度程序是一项颇具挑战性的工作。Java 开源的好处就是，领域问题都能找到现成的解决方案。

OpenSymphony 所提供的 Quartz 自 2001 年发布以来，已经被众多项目作为任务调度的解决方案。Quartz 在提供灵活性的同时并未牺牲其简单性，它所提供的强大功能使开发者可以轻松应对绝大多数任务调度的功能需求。

Sun 从 Java 1.3 开始，通过 `java.util.Timer` 和 `TimerTask` 提供了简单的调度功能，允许用户调度一个按固定时间间隔运行的任务。

针对前面提到的资源调度的要求，Java 5.0 通过 `java.util.concurrent` 这个新包中的若干类和接口，提供了方便的线程池调用功能。

Spring 对以上三者都提供了支持，通过 Spring 所提供的一系列 `FactoryBean`，可以很容易地创建任务调度框架的实例。此外，Spring 还提供了几个工具类，可以将某个具体 Bean 的方法直接作为被调度的任务，从而简化了任务的定义。Spring 还提供了支持线程池的调度执行器，它提供了一个抽象层，屏蔽了 Java 1.3、Java 1.4、Java 5.0 和 Java EE 的环境差异。

本章选取 Quartz 1.8.6 版本进行示例说明（有关 Quartz 的详细说明，可查看其官网 <http://www.quartz-scheduler.org/>）。需要在 `pom.xml` 中加入如下版本依赖：

```
<dependency>
  <groupId>org.quartz-scheduler</groupId>
  <artifactId>quartz</artifactId>
  <version>1.8.6</version>
</dependency>
```

16.2 Quartz 快速进阶

Quartz 是开源任务调度框架中的翘楚，它提供了强大的任务调度机制。Quartz 允许开发人员灵活地定义触发器的调度时间表，并可对触发器和任务进行关联映射。此外，Quartz 提供了调度运行环境的持久化机制，可以保存并恢复调度现场，即使系统因故障关闭，任务调度现场数据也不会丢失。此外，Quartz 还提供了组件式的侦听器、各种插件、线程池等功能。

16.2.1 Quartz 基础结构

Quartz 对任务调度的领域问题进行了高度抽象，提出了调度器、任务和触发器这 3 个核心的概念，并在 `org.quartz` 中通过接口和类对核心概念进行了描述。

- **Job**: 是一个接口，只有一个方法 `void execute(JobExecutionContext context)`，开发者通过实现该接口来定义需要执行的任务，`JobExecutionContext` 类提供了调度上下文的各种信息。`Job` 运行时的信息保存在 `JobDataMap` 实例中。
- **JobDetail**: Quartz 在每次执行 `Job` 时，都重新创建一个 `Job` 实例，所以它不是直接接收一个 `Job` 实例，而是接收一个 `Job` 实现类，以便运行时通过 `newInstance()` 的反射调用机制实例化 `Job`。因此需要通过一个类来描述 `Job` 的实现类及其他相关的静态信息，如 `Job` 名称、描述、关联监听器等信息，而 `JobDetail` 承担了这一角色。通过该类的构造函数 `JobDetail(java.lang.String name, java.lang.String group, java.lang.Class jobClass)`，可以更具具体地了解它的功用。该构造函数要求指定 `Job` 的实现类，以及任务在 `Scheduler` 中的组名和 `Job` 名称。
- **Trigger**: 是一个类，描述触发 `Job` 执行的时间触发规则。主要有 `SimpleTrigger` 和 `CronTrigger` 这两个子类。当仅需要触发一次或者以固定间隔周期性执行时，`SimpleTrigger` 是最适合的选择；而 `CronTrigger` 则可以通过 `Cron` 表达式定义出各种复杂的调度方案，如每天早晨 9:00 执行，每周一、周三、周五下午 5:00 执行等。
- **Calendar**: `org.quartz.Calendar` 和 `java.util.Calendar` 不同，它是一些日历特定时间点的集合（可以简单地将 `org.quartz.Calendar` 看作 `java.util.Calendar` 的集合——`java.util.Calendar` 代表一个日历时间点，若无特殊说明，后面的 `Calendar` 即指 `org.quartz.Calendar`）。一个 `Trigger` 可以和多个 `Calendar` 关联，以便排除或包含某些时间点。假设安排每周一早晨 10:00 执行任务，但是如果遇到法定节假日，则不执行任务，这时就需要在 `Trigger` 触发机制的基础上使用 `Calendar` 进行定点排除。针对不同的时间段类型，Quartz 在 `org.quartz.impl.calendar` 包下提供了若干个 `Calendar` 的实现类，如 `AnnualCalendar`、`MonthlyCalendar`、`WeeklyCalendar` 分别针对每年、每月和每周进行定义。
- **Scheduler**: 代表一个 Quartz 的独立运行容器，`Trigger` 和 `JobDetail` 可以注册到 `Scheduler` 中，二者在 `Scheduler` 中拥有各自的组及名称。组及名称是 `Scheduler` 查找定位容器中某一对象的依据，`Trigger` 的组及名称的组合必须唯一，`JobDetail` 的组及名称的组合也必须唯一（但可以和 `Trigger` 的组及名称相同，因为它们是不同类型的，处在不同的集合中）。`Scheduler` 定义了多个接口方法，允许外部通过组及名称访问和控制容器中的 `Trigger` 和 `JobDetail`。`Scheduler` 可以将 `Trigger` 绑定到某一 `JobDetail` 中，这样，当 `Trigger` 被触发时，对应的 `Job` 就被执行。一个 `Job` 可以对应多个 `Trigger`，但一个 `Trigger` 只能对应一个 `Job`。可以通过

SchedulerFactory 创建一个 Scheduler 实例。Scheduler 拥有一个 SchedulerContext，保存着 Scheduler 上下文信息，可以对照 ServletContext 来理解 SchedulerContext。Job 和 Trigger 都可以访问 SchedulerContext 内的信息。SchedulerContext 内部通过一个 Map，以键值对的方式维护这些上下文数据。SchedulerContext 为保存和获取数据提供了多个 put() 和 getXxx() 方法。可以通过 Scheduler#getContext() 方法获取对应的 SchedulerContext 实例。

- ThreadPool: Scheduler 使用一个线程池作为任务运行的基础设施，任务通过共享线程池中的线程来提高运行效率。

Job 有一个 StatefulJob 子接口，代表有状态的任务。该接口是一个没有方法的标签接口，其目的是让 Quartz 知道任务的类型，以便采用不同的执行方案。无状态任务在执行时拥有自己的 JobDataMap 复制，对 JobDataMap 的更改不会影响下次的执行。而有状态任务共享同一个 JobDataMap 实例，每次任务执行对 JobDataMap 所做的更改会保存下来，后面的执行可以看到这个更改，即每次执行任务后都会对后面的执行产生影响。

正因为这个原因，无状态的 Job 可以并发执行，而有状态的 StatefulJob 不能并发执行。这意味着如果前次的 StatefulJob 还没有执行完毕，则下次的任务将阻塞等待，直到前次任务执行完毕。有状态任务比无状态任务需要考虑更多的因素，程序往往更复杂，因此，除非必要，应尽量避免使用有状态的 Job。

如果 Quartz 使用了数据库持久化任务调度信息，则无状态的 JobDataMap 仅会在 Scheduler 注册任务时保存一次，而有状态任务对应的 JobDataMap 在每次执行任务后都会进行保存。

Trigger 自身也可以拥有一个 JobDataMap，其关联的 JobDataMap 可以通过 JobExecutionContext#getTrigger().getJobDataMap() 方法获取。不管是有状态还是无状态的任务，在任务执行期间对 Trigger 的 JobDataMap 所做的更改都不会进行持久化，即不会对下次任务的执行产生影响。

Quartz 拥有完善的事件和监听体系，大部分组件都拥有事件，如任务执行前事件、任务执行后事件、触发器触发前事件、触发器触发后事件、调度器开始事件、调度器关闭事件等，可以注册相应的监听器处理感兴趣的事件。

图 16-1 描述了 Scheduler 的内部组件结构。SchedulerContext 提供 Scheduler 全局可见的上下文信息，每个任务都对应一个 JobDataMap，虚线框中的 JobDataMap 表示有状态的任务。

一个 Scheduler 可以拥有多个 Trigger 和多个 JobDetail，它们可以分到不同的组中。在注册 Trigger 和 JobDetail 时，如果不显式指定所属的组，那么 Scheduler 将放入默认组中，默认组的组名为 Scheduler.DEFAULT_GROUP。组名和名称组成了对象的全名，同一类型对象（Job 或 Trigger）的全名不能相同。

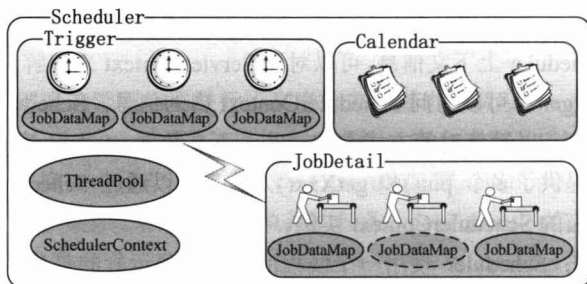


图 16-1 Scheduler 内部组件结构图

Scheduler 本身就是一个容器，它维护着 Quartz 的各种组件并实施调度的规则。Scheduler 还拥有线程池，线程池为任务提供执行线程。这比执行任务时简单地创建一个新线程要拥有更高的效率，同时通过共享机制可以减少资源的占用。基于线程池组件的支持，对于繁忙度高、压力大的任务调度，Quartz 可以提供良好的伸缩性。



提示

Quartz 完整下载包 examples 目录下拥有 10 多个实例，它们是快速掌握 Quartz 应用很好的实例。

16.2.2 使用 SimpleTrigger

SimpleTrigger 拥有多个重载的构造函数，用于在不同场合下构造出对应的实例。

- ❑ SimpleTrigger(String name, String group): 通过该构造函数指定 Trigger 所属组和名称。
- ❑ SimpleTrigger(String name, String group, Date startTime): 除指定 Trigger 所属组和名称外，还可以指定触发的开始时间。
- ❑ SimpleTrigger(String name, String group, Date startTime, Date endTime, int repeatCount, long repeatInterval): 除指定以上信息外，还可以指定结束时间、重复执行次数、时间间隔等参数。
- ❑ SimpleTrigger(String name, String group, String jobName, String jobGroup, Date startTime, Date endTime, int repeatCount, long repeatInterval): 这是最复杂的一个构造函数，在指定触发参数的同时，通过 jobGroup 和 jobName，使该 Trigger 和 Scheduler 中的某个任务关联起来。

通过实现 org.quartz.Job 接口，可以使 Java 类化身为可调度的任务。代码清单 16-1 提供了 Quartz 任务的一个示例。

代码清单 16-1 SimpleJob: 简单的 Job 实现类

```
package com.smart.basic.quartz;
import java.util.Date;
import org.quartz.Job;
```

```

import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;
public class SimpleJob implements Job {
    public void execute(JobExecutionContext jobCtx)
        throws JobExecutionException { ①
        System.out.println(jobCtx.getTrigger().getName()+ " triggered. time is:" +
(new Date()));
    }
}

```

实现 Job 接口方法

这个演示类的 `execute(JobExecutionContext context)` 方法只有一条简单的输出语句，可以在这个方法中包含任何想要执行的代码。下面通过 `SimpleTrigger` 对 `SimpleJob` 进行调度，如代码清单 16-2 所示。

代码清单 16-2 SimpleTriggerRunner: 使用 SimpleTrigger 进行调度

```

package com.smart.basic.quartz;
import java.util.Date;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerFactory;
import org.quartz.SimpleTrigger;
import org.quartz.impl.StdSchedulerFactory;
public class SimpleTriggerRunner {
    public static void main(String args[]) {
        try {
            JobDetail jobDetail = new JobDetail("job1_1", "jgroup1", SimpleJob.class); ①

            SimpleTrigger simpleTrigger = new SimpleTrigger("trigger1_1", "tgroup1"); ②
            simpleTrigger.setStartTime(new Date());
            simpleTrigger.setRepeatInterval(2000);
            simpleTrigger.setRepeatCount(100);

            SchedulerFactory schedulerFactory = new StdSchedulerFactory(); ③
            Scheduler scheduler = schedulerFactory.getScheduler();

            scheduler.scheduleJob(jobDetail, simpleTrigger); ④
            scheduler.start(); ⑤
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

创建一个 JobDetail 实例, 指定 SimpleJob

通过 SimpleTrigger 定义调度规则: 马上启动, 每 2 秒运行一次, 共运行 100 次

通过 SchedulerFactory 获取一个调度器实例

注册并进行调度

调度启动

首先在①处通过 `JobDetail` 封装 `SimpleJob`, 同时指定 `Job` 在 `Scheduler` 中的所属组及名称。在这里, 组名为 `jgroup1`, 而名称为 `job1_1`。

然后在②处创建一个 `SimpleTrigger` 实例, 指定该 `Trigger` 在 `Scheduler` 中的所属组及名称。接着设置调度的时间规则。

最后创建 `Scheduler` 实例, 并将 `JobDetail` 和 `Trigger` 实例注册到 `Scheduler` 中。在这

里, 通过 `StdSchedulerFactory` 获取一个 `Scheduler` 实例, 并通过 `scheduleJob(JobDetail jobDetail, Trigger trigger)` 完成两件事。

(1) 将 `JobDetail` 和 `Trigger` 注册到 `Scheduler` 中。

(2) 用 `Trigger` 对 `JobDetail` 中的任务进行调度。

在 `Scheduler` 启动后, `Trigger` 将定期触发并执行 `SimpleJob` 的 `execute(JobExecutionContext jobCtx)` 方法, 每 10 秒重复一次, 直到任务被执行 100 次。

还可以通过 `SimpleTrigger` 的 `setStartTime(java.util.Date startTime)` 和 `setEndTime(java.util.Date endTime)` 方法指定运行的时间范围。当运行次数和时间范围产生冲突时, 超过时间范围的任务不被执行。如可以通过 `simpleTrigger.setStartTime(new Date(System.currentTimeMillis() + 60000L))` 方法指定 60 秒后开始运行。

除了通过 `scheduleJob(jobDetail, simpleTrigger)` 方法建立 `Trigger` 和 `JobDetail` 的关联外, 还可以通过如下方式建立关联:

```
JobDetail jobDetail = new JobDetail("job1_1", "jgroup1", SimpleJob.class);
SimpleTrigger simpleTrigger = new SimpleTrigger("trigger1_1", "tgroup1");
...
simpleTrigger.setJobGroup("jgroup1"); ①-1 ← 指定关联的 Job 组名
simpleTrigger.setJobName("job1_1"); ①-2 ← 指定关联的 Job 名称
scheduler.addJob(jobDetail, true); ② ← 注册 JobDetail
scheduler.scheduleJob(simpleTrigger); ③ ← 注册指定了关联 JobDetail 的 Trigger
```

在这种方式中, `Trigger` 通过指定 `Job` 所属组及 `Job` 名称, 使用 `Scheduler` 的 `scheduleJob(Trigger trigger)` 方法注册 `Trigger`。在这里, 有两个值得注意的地方。

(1) 通过这种方式注册的 `Trigger` 必须已经指定了 `Job` 组名和 `Job` 名称, 否则调用 `scheduleJob(simpleTrigger)` 方法将抛出异常。

(2) 引用的 `JobDetail` 对象必须已经存在于 `Scheduler` 中, 即代码中①、②和③的顺序不能随意变动。



实战经验

在构造 `Trigger` 实例时, 可以考虑使用 `org.quartz.TriggerUtils` 工具类, 该工具类不但提供了众多获取特定时间的方法, 还拥有多个获取常见 `Trigger` 的方法。如 `makeSecondlyTrigger(String trigName)` 方法将创建一个每秒执行一次的 `Trigger`, `makeWeeklyTrigger(String trigName, int dayOfWeek, int hour, int minute)` 方法将创建一个每星期某一特定时间点执行一次的 `Trigger`, 而 `getEvenMinuteDate(Date date)` 方法将返回某一时间点一分钟以后的时间。

16.2.3 使用 CronTrigger

`CronTrigger` 能够提供比 `SimpleTrigger` 更有具体实际意义的调度方案, 调度规则基

于 Cron 表达式。CronTrigger 支持日历相关的周期时间间隔(比如每月第一个周一执行),而不是简单的周期时间间隔。因此,相对于 SimpleTrigger 而言,CronTrigger 在使用上也要复杂一些。

1. Cron 表达式

Quartz 使用类似于 Linux 下的 Cron 表达式定义时间规则。Cron 表达式由 6 或 7 个空格分隔的时间字段组成,如表 16-1 所示。

表 16-1 Cron 表达式时间字段

位 置	时间域名	允 许 值	允许的特殊字符
1	秒	0-59	,- */
2	分钟	0-59	,- */
3	小时	0-23	,- */
4	日期	1-31	,- * ? / L W C
5	月份	1-12	,- */
6	星期	1-7	,- * ? / L C #
7	年(可选)	空值 1970-2099	,- */

Cron 表达式的时间字段除允许设置数值外,还可以使用一些特殊的字符,提供列表、范围、通配符等功能,详细介绍如下。

- ❑ 星号 (*)：可用在所有字段中,表示对应时间域的每一个时刻。例如,*在分钟字段时,表示“每分钟”。
- ❑ 问号 (?)：该字符只在日期和星期字段中使用,它通常指定为“无意义的值”,相当于占位符。
- ❑ 减号 (-)：表达一个范围。如在小时字段中使用“10-12”,则表示从 10 点到 12 点,即 10,11,12。
- ❑ 逗号 (,)：表示一个列表值。如在星期字段中使用“MON,WED,FRI”,则表示星期一、星期三和星期五。
- ❑ 斜杠 (/)：x/y 表达一个等步长序列,x 为起始值,y 为增量步长值。如在分钟字段中使用 0/15,则表示为 0,15,30 和 45 秒;而 5/15 在分钟字段中表示 5,20,35,50。用户也可以使用*/y,它等同于 0/y。
- ❑ L：该字符只在日期和星期字段中使用,代表“Last”的意思,但它在两个字段中的意思不同。如果 L 用在日期字段中,则表示这个月份的最后一天,如 1 月 31 日,非闰年 2 月 28 日;如果 L 用在星期字段中,则表示星期六,等同于 7 (注意,这里的规则是星期六为一星期的最后一天)。但是,如果 L 出现在星期字段里,而且前面有一个数字 N,则表示“这个月的最后 N 天”。例如,6L 表示该月的最后一个星期五。
- ❑ W：该字符只能出现在日期字段里,是对前导日期的修饰,表示离该日期最近的工作日。例如,15W 表示离该月 15 日最近的工作日,如果该月 15 日是星期

六，则匹配 14 日星期五；如果 15 日是星期日，则匹配 16 日星期一；如果 15 日是星期二，那结果就是 15 日星期二。但必须注意关联的匹配日期不能跨月，如用户指定 1W，如果 1 日是星期六，结果匹配的是 3 日星期一，而非上个月最后一天。W 字符串只能指定单一日期，而不能指定日期范围。

- ❑ LW 组合：在日期字段中可以组合使用 LW，它的意思是当月的最后一个工作日。
- ❑ 井号（#）：该字符只能在星期字段中使用，表示当月的某个工作日。如 6#3 表示当月的第三个星期五（6 表示星期五，#3 表示当前的第三个），而 4#5 表示当月的第五个星期三。假设当月没有第五个星期三，则忽略不触发。
- ❑ C：该字符只在日期和星期字段中使用，代表“Calendar”的意思。它的意思是计划所关联的日期，如果日期没有被关联，则相当于日历中的所有日期。例如，5C 在日期字段中相当于 5 日以后的那一天，1C 在星期字段中相当于星期日后的第一天。

Cron 表达式对特殊字符的大小写不敏感，对代表星期的缩写英文大小写也不敏感。表 16-2 给出了一些完整的 Cron 表示式示例。

表 16-2 Cron表示式示例

表 示 式	说 明
"0 0 12 * * ?"	每天 12:00 运行
"0 15 10 ? * *"	每天 10:15 运行
"0 15 10 * * ?"	每天 10:15 运行
"0 15 10 * * ? *"	每天 10:15 运行
"0 15 10 * * ? 2008"	在 2008 年的每天 10:15 运行
"0 * 14 * * ?"	每天 14 点到 15 点每分钟运行一次，开始于 14:00，结束于 14:59
"0 0/5 14 * * ?"	每天 14 点到 15 点每 5 分钟运行一次，开始于 14:00，结束于 14:55
"0 0/5 14,18 * * ?"	每天 14 点到 15 点每 5 分钟运行一次，此外每天 18 点到 19 点每 5 分钟也运行一次
"0 0-5 14 * * ?"	每天 14:00 到 14:05，每分钟运行一次
"0 10,44 14 ? 3 WED"	3 月每周三的 14:10 到 14:44，每分钟运行一次
"0 15 10 ? * MON-FRI"	每周一、二、三、四、五的 10:15 运行
"0 15 10 15 * ?"	每月 15 日的 10:15 运行
"0 15 10 L * ?"	每月最后一天的 10:15 运行
"0 15 10 ? * 6L"	每月最后一个星期五的 10:15 运行
"0 15 10 ? * 6L 2014-2016"	2014 年、2015 年、2016 年每月最后一个星期五的 10:15 运行
"0 15 10 ? * 6#3"	每月第三个星期五的 10:15 运行



轻松一刻

Cron（译为克龙）代表 100 万年，是英文单词中最大的时间单位。另一个我们熟悉的 google 则来源于 Googol（译为古戈尔），代表 10 的 100 次方，足可穷尽宇宙万物。不过现在最大



的数是 googolplex, 代表 10 的 googol 次方。在汉语里, 在亿、兆之后还有很多大数, 如京 (10 的 16 次方, 后略)、垓 (20)、秭 (24)、穰 (28)、沟 (32)、涧 (36)、正 (40)、载 (44)。此外还有很多来自佛学的大数, 如恒河沙 (10 的 52 次方, 后略)、频波罗 (56)、矧羯罗 (112, 这个数已经大于 googol)、不可说不可说转 (10 的 7×2122 次方)。

2. CronTrigger 实例

下面使用 CronTrigger 对 SimpleJob 进行调度, 通过 Cron 表达式制定调度规则, 让它每 5 秒运行一次, 如代码清单 16-3 所示。

代码清单 16-3 CronTriggerRunner: 使用CronTrigger进行调度

```
package com.smart.basic.quartz;
import org.quartz.CronExpression;
import org.quartz.CronTrigger;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerFactory;
import org.quartz.impl.StdSchedulerFactory;
public class CronTriggerRunner {
    public static void main(String args[]) {
        try {
            JobDetail jobDetail = new JobDetail("job1_2", "jGroup1", SimpleJob.class);

            CronTrigger cronTrigger = new CronTrigger("trigger1_2", "tgroupl"); ①-1
            CronExpression cexp = new CronExpression("0/5 * * * * ?"); ①-2
            cronTrigger.setCronExpression(cexp); ①-3
            SchedulerFactory schedulerFactory = new StdSchedulerFactory();
            Scheduler scheduler = schedulerFactory.getScheduler();
            scheduler.scheduleJob(jobDetail, cronTrigger);
            scheduler.start();
            //②
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

创建 CronTrigger, 指定组及名称 → ①-1
定义 Cron 表达式 → ①-2
设置 Cron 表达式 → ①-3

运行 CronTriggerRunner, 每 5 秒将触发 SimpleJob 运行一次。在默认情况下, Cron 表达式对应当前的时区, 可以通过 CronTriggerRunner 的 setTimeZone(java.util.TimeZone timeZone)方法显式指定时区。用户也可以通过 setStartTime(java.util.Date startTime)和 setEndTime(java.util.Date endTime)方法分别指定开始和结束时间。



实战经验

在代码清单 16-3 的②处需要通过 Thread.currentThread.sleep()方法让主线程睡眠一段时间, 使调度器可以继续执行任务调度的工作; 否则在调度器启动后, 因为主线程立即退出, 寄生于主线程的调度器也将关闭, 调度器中的任务都将相应销毁, 这将导致看

不到实际的运行效果。在单元测试的时候，使主线程睡眠一段时间以便让任务线程不被提前终止是经常使用的测试方法。对于测试某些长周期执行的调度任务，开发者可以简单地调整操作系统时间进行模拟。

16.2.4 使用 Calendar

在实际任务调度中，不可能一成不变地按照某个特定周期调度任务，必须考虑到现实生活中日历上的特殊日期。

下面安排一个任务，每小时运行一次，并将五一劳动节和国庆节排除在外，如代码清单 16-4 所示。

代码清单 16-4 CalendarExample: 使用Calendar

```
package com.smart.basic.quartz;
import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;
import org.quartz.impl.calendar.AnnualCalendar;
import org.quartz.TriggerUtils;
...
public class CalendarExample {
    public static void main(String[] args) throws Exception {
        SchedulerFactory sf = new StdSchedulerFactory();
        Scheduler scheduler = sf.getScheduler();
        AnnualCalendar holidays = new AnnualCalendar(); ①
        Calendar laborDay = new GregorianCalendar(); ②
        laborDay.add(Calendar.MONTH, 5);
        laborDay.add(Calendar.DATE, 1);
        Calendar nationalDay = new GregorianCalendar(); ③
        nationalDay.add(Calendar.MONTH, 10);
        nationalDay.add(Calendar.DATE, 1);
        ArrayList<Calendar> calendars = new ArrayList<Calendar>();
        calendars.add(laborDay);
        calendars.add(nationalDay);
        holidays.setDaysExcluded(calendars); ④
        scheduler.addCalendar("holidays", holidays, false, false); ⑤
        Date runDate = TriggerUtils.getDateOf(0, 0, 10, 1, 4); ⑥
        JobDetail job = new JobDetail("job1", "group1", SimpleJob.class);
        SimpleTrigger trigger = new SimpleTrigger("trigger1", "group1",
            runDate,
            null,
            SimpleTrigger.REPEAT_INDEFINITELY,
            60L * 60L * 1000L);
        trigger.setCalendarName("holidays"); ⑦
```

法定节日是以每年为周期的，所以使用 AnnualCalendar

五一劳动节

国庆节

排除这两个特殊日期

向 Scheduler 注册日历

4月1日上午10点

让 Trigger 应用指定的日历规则

```
scheduler.scheduleJob(job, trigger);
scheduler.start();
//在实际应用中主线程不能停止, 否则Scheduler得不到执行, 此处省略
}
```

由于节日是每年重复的, 所以使用 `org.quartz.Calendar` 的 `AnnualCalendar` 实现类, 通过②和③处的代码, 指定五一劳动节和国庆节这两个节日, 并通过 `AnnualCalendar#setDaysExcluded(ArrayList days)` 方法添加这两个日期。

在定制好 `org.quartz.Calendar` 后, 还需要通过 `Scheduler#addCalendar(String calName, Calendar calendar, boolean replace, boolean updateTriggers)` 方法进行注册。如果 `updateTriggers` 为 `true`, 则 `Scheduler` 中已引用 `Calendar` 的 `Trigger` 将得到更新, 如④处所示。

在⑦处, 让一个 `Trigger` 指定使用 `Scheduler` 中代表节日的 `Calendar`, 这样 `Trigger` 就会避开五一劳动节和国庆节这两个特殊日子了。

16.2.5 任务调度信息存储

在默认情况下, Quartz 将任务调度的运行信息保存在内存中。这种方法提供了最佳的性能, 因为在内存中数据访问速度最快; 不足之处是缺乏数据的持久性, 当程序中途停止或系统崩溃时, 所有运行的信息都会丢失。比如希望安排一个执行 100 次的任务, 如果执行到 50 次时系统崩溃了, 那么系统重启时任务的执行计数器将从 0 开始计数。在大多数实际应用中, 往往并不需要保存任务调度的现场数据, 因为很少需要规划一个指定执行次数的任务。对于仅执行一次的任务来说, 其执行条件信息本身应该是已经持久化的业务数据 (如对锁定到期的用户执行解锁的任务, 锁定到期的时间点应该就是业务数据), 当任务执行完成后, 条件信息也会相应改变。当然, 调度现场信息不仅包括运行次数, 而且包括调度规则、`JobDataMap` 中的数据等。

如果确实需要持久化任务调度信息, 则 Quartz 允许用户通过调整其属性文件, 将这些信息保存到数据库中。在使用数据库保存了任务调度信息后, 即使系统崩溃后重新启动, 任务调度信息仍将得到恢复。如前面所说的例子, 执行 50 次系统崩溃后重新运行, 计数器将从 51 开始计数。使用数据库保存信息的任务称为持久化任务。

1. 通过配置文件调整任务调度信息的保存策略

其实 Quartz JAR 文件的 `org.quartz` 包下就包含了一个 `quartz.properties` 属性配置文件, 并提供了默认设置。如果需要调整默认配置, 则可以在类路径下建立一个新的 `quartz.properties` 属性, 它将自动被 Quartz 加载并覆盖默认的设置。

先来了解一下 Quartz 的默认属性配置文件, 如代码清单 16-5 所示。

代码清单 16-5 quartz.properties: 默认配置

```
org.quartz.scheduler.instanceName = DefaultQuartzScheduler ①
org.quartz.scheduler.rmi.export = false
org.quartz.scheduler.rmi.proxy = false
org.quartz.scheduler.wrapJobExecutionInUserTransaction = false
```

集群的配置, 这里不使用集群

```

org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool ②
org.quartz.threadPool.threadCount = 10
org.quartz.threadPool.threadPriority = 5
org.quartz.threadPool.threadsInheritContextClassLoaderOfInitializingThread = true
org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore ③

```

配置调度器的线程池

配置任务调度现场数据保存机制

Quartz 的属性配置文件主要包括三方面的信息：

- (1) 集群信息。
- (2) 调度器线程池。
- (3) 任务调度现场数据的保存。

如果任务数目很大，则可以通过增大线程池获得更好的性能。在默认情况下，Quartz 采用 `org.quartz.simpl.RAMJobStore` 保存任务的现场数据，顾名思义，信息保存在 RAM 内存中，可以通过以下设置将任务调度现场数据保存到数据库中，如代码清单 16-6 所示。

代码清单 16-6 quartz.properties：使用数据库保存任务调度现场数据

```

...
org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.JobStoreTX
org.quartz.jobStore.tablePrefix = QRTZ_ ①
org.quartz.jobStore.dataSource = qzDS ②
org.quartz.dataSource.qzDS.driver = com.mysql.jdbc.Driver
org.quartz.dataSource.qzDS.URL = jdbc:mysql://localhost:3306/sampledb
org.quartz.dataSource.qzDS.user = stamen
org.quartz.dataSource.qzDS.password = abc
org.quartz.dataSource.qzDS.maxConnections = 10

```

数据库表前缀

数据源名称

定义数据源的具体属性

要将任务调度数据保存到数据库中，就必须使用 `org.quartz.impl.jdbcjobstore.JobStoreTX` 代替原来的 `org.quartz.simpl.RAMJobStore`，并提供相应的数据库配置信息。首先在①处指定了 Quartz 数据库表的前缀，然后在②处定义了一个数据源，最后在③处具体定义这个数据源的连接信息。

用户必须事先在相应的数据库中创建 Quartz 的数据表（共 8 张），在 Quartz 的完整发布包的 `docs/dbTables` 目录下拥有对应不同数据库的 SQL 脚本，我们已经将 Quartz 的不同数据库脚本文件放到 <工程项目>/schema 目录下。对于 MySQL innodb 引擎来说，直接运行 `tables_mysql_innodb.sql` 就可以了。

2. 查询数据库中的运行信息

任务的现场保存对于上层的 Quartz 程序来说是完全透明的，在 `src` 目录下编写一个如代码清单 16-6 所示的 `quartz.properties` 文件后，重新运行代码清单 16-2 或代码清单 16-3 的程序，在数据库表中将看到对应的持久化信息。当调度程序运行过程中途停止后，任务调度的现场数据将记录在数据库表中，在系统重启时就可以在此基础上继续进行任务的调度，如代码清单 16-7 所示。

代码清单 16-7 JDBCJobStoreRunner: 从数据库中恢复任务的调度

```

package com.smart.basic.quartz;
import org.quartz.Scheduler;
import org.quartz.SchedulerFactory;
import org.quartz.SimpleTrigger;
import org.quartz.Trigger;
import org.quartz.impl.StdSchedulerFactory;
public class JDBCJobStoreRunner {
    public static void main(String args[]) {
        try {
            SchedulerFactory schedulerFactory = new StdSchedulerFactory();
            Scheduler scheduler = schedulerFactory.getScheduler();
            String[] triggerGroups = scheduler.getTriggerGroupNames(); ①
            for (int i = 0; i < triggerGroups.length; i++) { ②
                String[] triggers = scheduler.getTriggerNames(triggerGroups[i]);
                for (int j = 0; j < triggers.length; j++) {
                    Trigger tg =
scheduler.getTrigger(triggers[j], triggerGroups[i]);
                    if (tg instanceof SimpleTrigger
                        && tg.getFullName().equals("tgroup1.trigger1_1")) { ②-1
                        scheduler.rescheduleJob(triggers[j], triggerGroups[i],
tg); ②-1
                    }
                }
            }
            scheduler.start();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

获取调度器中所有的触发器组

重新恢复在 tgroup1 组中名为 trigger1_1 的触发器的运行

根据名称判断

恢复运行

当代码清单 16-2 中的 SimpleTriggerRunner 执行到一段时间后非正常退出, 就可以通过 JDBCJobStoreRunner 根据记录在数据库中的现场数据恢复任务的调度。Scheduler 中的所有 Trigger 及 JobDetail 的运行信息都会保存在数据库中, 这里仅恢复 tgroup1 组中名为 trigger1_1 的触发器, 这可以通过如②-1 所示的代码进行过滤, 触发器采用 GROUP.TRIGGER_NAME 的全名格式。通过 Scheduler#rescheduleJob(String triggerName, String groupName, Trigger newTrigger)方法即可重新调度关联某个 Trigger 的任务。

下面来观察一下不同时期 quartz_simple_triggers 表的数据。

(1)运行代码清单 16-2 中的 SimpleTriggerRunner 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	...	100	2000
					4

其中, REPEAT_COUNT 表示需要运行的总次数, 而 TIMES_TRIGGERED 表示已经运行的次数。