

此文档源自 Pro Spring 2.5 Spring 高级程序设计。主要讲述了 Spring 的架构，bean 的配置，属性注入，AOP,对数据库操作的支持 jdbc,iBATIS,hibernate....待续

Pro Spring 2.5 Spring 高级程序设计

DavidWang 精心录制

David Wang

目录

用到的 SQL 语句	2
☞ Spring AOP.....	4
☞ Spring AOP 架构.....	5
☛ Spring 里的通知者和切入点	6
切入点总结.....	8
CGLIB 代理与 JDK 动态代理的性能比较	8
第 6 章 AOP 进阶.....	9
6.1 @AspectJ 注解.....	10
6.2.5 通知的类型.....	10
1 前置通知.....	10
2 后置通知.....	10
3 抛出后通知.....	11
4 后置通知.....	11
5 包围通知.....	11
Spring JDBC	11
9.2 Spring 对数据访问支持的概念.....	11
9.3 Spring 对 JDBC 数据访问的支持.....	12
使用 JdbcTemplate 类 execute()方法.....	12
9.4.2 JdbcTemplate 类的 query 方法和该方法的扩展	16
9.4.3 JdbcTemplate 类的 update 方法	18
9.4.4 JdbcTemplate 类的 batchUpdate 方法.....	18
9.5 RdbmsOperation 子类	19
9.5.2 BatchSqlUpdate.....	22
9.5.3 SqlCall 类和 StoredProcedure 子类.....	23
9.5.4 SqlQuery 类和它的子类.....	25
9.5.5 JdbcTemplate 类和 RdbmsOperation 类的比较	29
9.6 大二进制对象.....	29
9.7 JdbcDaoSupport 类	31
9.8 简单的 Spring JDBC	34
9.8.1 SimpleJdbcTemplate 类.....	34
9.8.2 SimpleJdbcCall 类.....	36
9.8.3 SimpleJdbcInsert 类	39
9.8.4 SimpleJdbcTemplate 类.....	40
9.9 小结.....	40
第十章 集成 iBATIS.....	40
10.1 iBATIS 简述.....	41
10.3 查询数据.....	43
10.3.1 简单查询操作.....	43
10.3.2 一对一查询操作.....	45
10.3.3 一对多查询操作.....	48
10.3.4 多对多查询操作.....	50
10.4 更新数据.....	50

10.5 删除数据.....	51
10.6 插入数据.....	51
10.7 iBATIS 缺少的特性.....	52
10.8 整体性能.....	52
第 11 章 Spring 对 Hibernate 的支持	52
11.3 Hibernate 支持的介绍.....	52
11.3.1 使用 HibernateSession	54
11.3.2 使用 HibernateDaoSupport 类	56
11.3.3 HibernateTemplate 和 Session 之间的选择.....	58
11.4 在企业级应用中使用 Hibernate.....	63
11.4.1 阻止更新脏数据.....	63
11.4.2 对象等价性.....	65
11.4.3 事务支持.....	68
11.4.4 延迟加载.....	75
11.5 处理大数据集.....	89
11.6 处理大对象.....	95
第 16 章 事务管理.....	101
16.1 Spring 事务抽象层简介	101
16.2 分析事务属性.....	101
16.2.1 探索 TransactionDefinition 接口	102
16.2.2 使用 TransactionStatus 接口	102
16.2.3 PlatformTransactionManager 的实现.....	103
16.3 对一个事务管理示例的探索.....	103
1) Obj instanceof SomeType	
只要 SomeType 是 Obj 的类、直接或间接父类或者接口的话都是为 TRUE	
2) 实例内部类	
内部类必须在外部类实例化后才能实例化即 new Outer().new Inner()	
实例内部类中不能定义静态成员，只能定义实例成员。	
3) 静态内部类(成员内部类的一种)	

用到的 SQL 语句

```
create table t_customer(
    id number(19,0) not null,
    first_name varchar2(50) not null,
    last_name varchar2(50) not null,
    last_login timestamp null,
    comments clob null,
    constraint pk_customer primary key(id)
)
```

```
/
create sequence s_customer_id start with 1000
/
create or replace procedure p_actstartled(n number)
is
begin
    dbms_output.put_line(n || '?! Really?');
end;
/
create or replace function f_calculate
return number
as
begin
    return 42;
end;
/

create or replace package simplejdbc as
    type rc_customer_type is ref cursor return t_customer%rowtype;
    procedure p_find_customer(rc_customer_type out rc_customer_type);
end;
/

create or replace package body simplejdbc as
    procedure p_find_customer(rc_customer_type out rc_customer_type) is
    begin
        open rc_customer_type for select * from t_customer;
    end;
end;

insert into t_customer(id,first_name, last_name, last_login, comments)
values (1,'Jan', 'Machacke', null, null);

select table_name from user_tables;

select * from t_customer order by id;

delete from t_customer where id > 3;

log4j config file
#####
# LOGGING LEVELS
#####
```

```
# To turn more verbose logging on - change "WARN" to "DEBUG"
log4j.rootLogger=INFO, console

#####
# LOG FILE LOCATIONS
#####

log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.Threshold=DEBUG
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern= %p [%c{6}] %m%n
```

∞ Spring AOP

1.概念

- 连接点(joinpoint): 一个连接点本身是一个程序执行过程中的特定点。典型的连接点包括对一个方法的调用、一个方法执行的过程本身、类的初始化、对象的实例化等。连接点是 AOP 的核心概念之一,它用来定义在程序的什么地方能通过 AOP 加入额外的逻辑。
- 通知(advice): 在某一特定的连接点处运行的代码称为“通知”。通知有很多种,比如在连接点之前执行的前置通知(before advice)和在连接点之后执行的后置通知(after advice)。
- 切入点(pointcut):切入点是用来定义某一个通知该何时执行的一组连接点。通过创建切入点我们可以精确地控制程序中什么组件接到什么通知。之前我们提到过,一个典型的连接点是方法的调用,而一个典型的切入点就是对某一类的所有方法调用的集合。通常我们会通过复杂的切入点来控制通知什么时候被执行。
- 方面(aspect):通知和切入点的集合叫做方面。这个组合定义了一段程序中应该包括的逻辑以及何时应该执行该逻辑。
- 织入(weaving):织入是将方面真正加入程序代码的过程。对于编译 AOP 方案而言,织入自然是在编译时完成的,它通常是作为编译过程的一个额外步骤。类似地,对于运行时 AOP 方案,织入过程是在程序运行时动态执行的。
- 目标(target):如果一个对象的执行过程受到某个 AOP 操作的修改,那么它就叫做一个目标对象,目标对象通常也称为被通知对象。
- 引入(introduction):通过引入,我们可以在一个对象中加入新的方法或者字段,以改变它的结构,你可以使用引入来让任何对象实现一个特定的接口,而不需要这个对象的类显式的实现这个接口。

2.AOP 的类型

✦ 静态 AOP

用 Java 术语来说,我们可以在一个静态 AOP 实现中通过修改应用程序实际字节码来完成织入过程,从而根据需求修改和扩展程序代码。显然,这是一个达到织入过程的高性能的方式,因为最终结果就是普通的 Java 字节码。

这种方式的缺点是,如果我们想对方面做任何修改,即使只是加入一个新的连接点,都

必须重新编译整个应用程序。

AspectJ 便是静态 AOP 实现的一个绝好示例。

✚ 动态 AOP

动态 AOP 实现(比如 Spring AOP)区别于静态 AOP 实现的地方在于织入过程是在运行时动态进行的。具体如何做到这一点是跟具体实现相关的,但你会看到, Spring 的方法是为所有被通知对象创建代理,以便通知可以按需被调用。动态 AOP 一点小小的不足是,通常它的性能比不上静态静态 AOP,但也在持续地改善中,动态 AOP 实现的主要好处在于,你能轻易地修改一个应用的整个方面的集合而无需重新编译主程序的代码。

∞ Spring AOP 架构

- 📖 Spring AOP 架构的核心是建立在代理上的。当我们建立被通知类的实例时,我们必须使用 ProxyFactory 类加入我们需要织入该类的所有通知。然后为该类的一个对象创建代理。
- 📖 Spring 内部有二种实现代理的方法: JDK 动态代理和 CGLIB 代理。通常 CGLIB 的性能会明显好过 JDK 动态代理。
- 📖 Spring AOP 中最明显的简化之一就是它只支持一种类型的连接点:方法调用。乍看一下,与 AspectJ 比这是一个很大的限制,但在所有的连接点中,方法调用无疑是最有用的一种,我们可以用它来完成大多数用到 AOP 的地方,如果你需要方法调用之外别的连接点,你总可以同时使用 Spring 和 AspectJ。
- 📖 Spring AOP 中,一个方面是由一个实现 Advisor(通知者)接口的类表示的。Spring 中提供了一些方便的 Advisor 接口的实现类。Advisor 有 2 个子接口: IntroductionAdvisor 和 PointcutAdvisor。所有用切入点控制该在哪些连接点运行通知的 Advisor,都应该实现 PointcutAdvisor 接口。
- 📖 ProxyFactory 类控制着 Spring AOP 中的织入和创建代理的过程。在真正创建代理之前,我们必须指定被通知对象或者说目标对象。我们可以通过 setTarget()方法来完成这个步骤。ProxyFactory 内部将生成代理的过程转交给一个 DefaultAopProxyFactory 对象来完成,后者又根据程序中的设置将其转交给一个 Cglib2AopProxy 或者 JdkDynamicAopProxy 来完成。
- 📖 Spring 中通知类型

前置通知 org.springframework.aop.MethodBeforeAdvice

后置通知 org.springframework.aop.AfterReturningAdvice

包围通知 org.springframework.aop.MethodInterceptor

抛出通知 org.springframework.aop.ThrowsAdvice

引入 org.springframework.aop.IntroductionInterceptor

这些通知类型,结合方法调用连接点一起可以完成 90%的 AOP 工作,对于剩下的不常用的 10%,我们可以依赖 AspectJ。

想实现哪种的话，先定义本来的类 **SomeClass** 而后实现 **SomeAdvice implements MethodBeforeAdvice**(或其它)

而后 **ProxyFactory pf = new ProxyFactory()**

bf.setTarget(new SomeClass())

bf.addAdvice(new SomeAdvice())

SomeClass proxy = (SomeClass)bf.getProxy();现在就是代理类了

◇ 抛出通知 此招聘通知类实现 **ThrowsAdvice** 接口，而此接口中并不包含任何方法。但是在此通知类中需要编写以 **afterThrowing()**为名字的方法，参数不定，有二种

- 1) `public void afterThrowing(Exception ex) throws Throwable{`
- 2) `public void afterThrowing(Method method, Object[] args, Object target, IllegalArgumentException ex) throws Throwable{`相信你也可以猜出它们的含义。其中 **IllegalArgumentException** 只是另一个异常，通过程序可以看到，简单地抛出一个 **Exception** 会运行第一个 **afterThrowing()**方法，而抛出的一个 **IllegalArgumentException** 时会运行第二个 **afterThrowing()**方法。对于抛出的每一个异常，**Spring** 只会运行一个 **afterThrowing()**方法，**Spring** 会选择签名与抛出的异常最匹配的一个方法。如果抛出通知定义了两个 **afterThrowing()**方法，他们的异常类型一样，但是一个只有一个参数而另一个有 4 个参数，那么 **Spring** 会运行那个有 4 个参数的 **afterThrowing()**方法。抛出通知在很多情况下都很有用：我们可以用它重新定义整个异常继承结构或者构建集中的异常日志。我们发现在调试运行中的程序时异常通知格外有用，因为我们无需修改程序代码就可以添加新的日志代码。

☺Spring 里的通知者和切入点

到目前为止，以前的示例都用 **ProxyFactory.addAdvice()**方法为代理设定通知。实际上，该方法会在后台委派给 **addAdvisor()**方法，而后者会创建一个 **DefaultPointcutAdvisor** 实例并将切入点设为对所有方法的调用。这样，目标的所有方法就都能被通知到了。在某些情况下，比如用 **AOP** 做日志时，这样做可能正是我们需要的，可能在别的情况下，我们希望只通知相关的而不是所有的方法。

虽然我们可以简单地在通知内检查被通知的方法是不是正确的方法，但将接受的方法名称列表直接写进代码中会降低通知的通用性。使用切入点可以控制哪些方法被通知而无需将该列表写入通知里，显然这样做可重用性比较好。为了确定被调用的方法是否应该被通知，每次目标上的任何一个方法被调用时都需要做一个检查，这显然会影响程序的性能。当使用切入点时，每个方法会被检查一遍，其结果会被缓冲起来供日后使用。

1) 切入点接口

```
public interface Pointcut{
    ClassFilter getClassFilter();
    MethodMatcher getMethodMatcher();
}
public interface ClassFilter{
    Boolean matcher(Class clazz);
}
public interface MethodMatcher{
    Boolean match(Method m,Class targetClass);
}
```



```

    Boolean isRuntime();
    Boolean matches(Method m,Class targetClass,Object[] a
}

```

Spring 支持 2 种不同的 **MethodMatcher**: 静态的和动态的。一个 **MethodMatcher** 具体是哪一种取决于 **isRuntime()** 方法的返回值。在使用一个 **MethodMatcher** 之前, Spring 会调用 **isRuntime()** 如果返回 **FALSE**, 那么该 **MethodMatcher** 就是静态的否则就是动态的。

如果切入点是静态的, 那么 Spring 会针对目标上的每一个方法调用一次 **MethodMatcher** 的 **matches(Method,Class)** 方法, 其返回值被缓存起来以便日后调用该方法时使用。这样, 对每一个方法的适用性测试只会进行一次, 之后调用该方法时不会再调用 **matches()** 方法了。

如果切入点是动态的, Spring 仍然会在目标方法第一次调用时用 **matches(Method,Class)** 进行一个静态的测试来检查其总的适用性。不过, 如果该测试返回 **TRUE**, 那么在此基础上, 每次该方法被调用时 Spring 会再次调用 **matches(Method,Class,Object[])** 方法。这样一个动态的 **MethodMatcher** 可以根据一次具体的方法调用, 而不仅仅是方法本身, 来决定切入点是否适用。

显然, 静态切入点(即 **MethodMatcher** 为静态的切入点)的性能比动态切入点要好得多, 因为它们不需要在每次调用时重新检查, 不过话说回来, 使用动态切入点来决定是否执行通知要比使用静态切入点更加灵活。总的来说, 我们建议尽量使用静态切入点。不过, 如果执行通知的开销非常大的话, 最好使用动态切入点来规避对不必要的调用通知。

通常很少有人自己编写 **Pointcut** 的实现, 因为 Spring 提供了静态切入点和动态切入点的抽象基类。

org.springframework.aop.support 下的

```

ComposablePointcut
ControlFlowPointcut
JdkRegexpMethodPointcut
NameMatchMethodPointcut

```

org.springframework.aop 下的

```

StaticMethodMatcherPointcut
DynamicMethodMatcherPointcut
AnnotationMatchingPointcut
AspectJExpressionPointcut

```

2) 使用 **DefaultPointcutAdvisor**

在使用任何 **Pointcut** 实现之前, 必须先生成一个 **Advisor**, 更准确地说是一个 **PointcutAdvisor**。Spring 用 **Advisor** 来表示方面, 即通知和切入点的结合, 其中切入点定义了哪些地方可以被通知以及如何通知。

3) 用 **StaticMethodMatcherPointcut** 创建静态切入点

以此为基类创建一个简单的静态切入点, 只需要实现 **matches(Method,Class)**

4) 使用 **DynamicMethodMatcherPointcut** 创建动态切入点, 它只有一个抽象方法 **matches(Method,Class,Object[])** 我们必须实现这个方法, 不过我们会看到明智的做法是同时也实现 **matches(Method,Class)** 方法以控制静态检查

5) 使用简单的名称匹配 **NameMatchMethodPointcut** 通常创建一个切入点时, 我们想要基于方法名称来做匹配, 而忽略方法的签名和返回类型。在这种情况下, 我们可以用此类来匹配一组方法名称, 而不需要创建 **StaticMethodMatcherPointcut** 的子类。

6) 用正则表达式创建切入点 我们需要一个正则表达式切入点(**JdkRegexpMethodPointcut** 或

- 者 `Perl5RegexMethodPointcut` 来用正则表达式匹配方法名。
- 7) 通知者的便利实现 对于很多的 `Pointcut`, `Spring` 也提供相对的 `Advisor` 的便利实现, 这些实现也可以当作 `Pointcut` 使用。比如说, 在之前的例子中 (`com.apress.prospring2.ch05.namepc.NamePointcutExample`) 我们同时使用了 `NameMatchMethodPointcut` 和 `DefaultPointcutAdvisor` 也可以直接使用 `NameMatchMethodPointcutAdvisor`, 这样会更加简洁。
 - 8) 使用 `AspectJExpressionPointcut` 此类让你能够编写 `AspectJ` 表达式来定义一个切入点。(参见 `com.apress.prospring2.ch05.aspectj.AspectJExpressionPointcutDemo`)
 - 9) 使用 `AnnotationMatchingPointcut` 考虑这样一种情况: 在测试时对于性能监控的目的, 我们想对若干方法进行通知。但是这些方法都来自于不同的包和类中。另外, 我们还希望通过尽可能少的配置来对要监控的方法或类进行修改。一种解决方案就是实现注解, 并在所有希望通知的类或方法上使用该注解。这便是此类的用武之地。这个类的构造函数有二个参数, 均是传送注解类(`SomeAnnotation.class`), 第一个参数为类级别第二个参数为方法级别, 如果第一个设置而第二个为 `Null` 的话, 则被注解的类的所有方法均被通知, 如果第一个为 `Null` 第二个设置的话, 则被注解的方法被通知, 如果二个都设置的话类级别参数相当于无效, 即仍是只有有注解的方法才会被通知。
 - 10) 使用控制流切入点 `ControlFlowPointcut` `Spring` 的控制流切入点匹配一个类中对某一个方法或所有方法的调用。可参照 `com.apress.prospring2.ch05.cflow.ControlFlowDemo`
 - 11) 使用 `ComposablePointcut` 此类可以将多个切入点组合在一起, 取它们的并集或者交集。它有 2 个方法 `union()` 和 `intersection()` 其传入的参数可以是 `MethodMatcher` 也可以是 `ClassFilter`。参考示例 `com.apress.prospring2.ch05.cflow.ComposablePointcutDemo`

切入点总结

如果你没有找到一个切入点实现满足你的需要, 仍可以在 `Pointcut`、`MethodMatcher` 和 `ClassFilter` 基础上创建自己的实现。

使用组合切入点和通知者有两种办法。一种是分开创建切入点实现和通知者实现。如前面例子中我们都创建了 `Pointcut` 的实现, 然后用 `DefaultPointcutAdvisor` 把通知和切入点加到代理中去。另一种办法在 `Spring` 文档的很多例子中用到, 它将 `Pointcut` 封装到我们自己的 `Advisor` 实现中, 这样, 我们创建的类既实现 `Pointcut` 又实现 `PointcutAdvisor`, 而 `PointcutAdvisor.getPointcut()` 方法简单地返回 `this.Spring` 中的很多类, 如 `StaticMethodMatchPointcutAdvisor` 都采用这种方法。

CGLIB 代理与 JDK 动态代理的性能比较

参照 `com.apress.prospring2.ch05.proxies.ProxyPerfTest` 类。

在上述类代码中, 可以看到测试了 3 种不同的代理类型: 标准 CGLIB 代理、固定通知链的 CGLIB 代理以及 JDK 代理。对于每一种代理我们运行以下 5 种测试。

◆ 被通知方法: 即接受通知的方法, 此例中

```
public class TestPointcut extends StaticMethodMatcherPointcut {

    @Override
    public boolean matches(Method method, Class targetClass) {
        // TODO Auto-generated method stub
    }
}
```

```

        return "advised".equals(method.getName());
    }

    @Override
    public ClassFilter getClassFilter() {
        // TODO Auto-generated method stub
        return new ClassFilter() {
            @Override
            public boolean matches(Class clazz) {
                // TODO Auto-generated method stub
                return clazz == SimpleBean.class;
            }
        };
    }
}

```

此接点标明了只在 SimpleBean 类上的 advised 方法可以被通知到。我们在这个测试中使用的是一个空的前置通知以减小通知本身对性能测试的影响。

- ◆ 未被通知的方法：代理上的一个不接受通知的方法。通常代理上会有很多未被通知的方法。这个测试反映不同的代理处理未被通知方法的性能。此例中为 `unadvise()`
- ◆ `equals()`方法：这个测试反映执行 `equals()`方法的额外开销。这当你的代理在 `HashMap` 或类似集合中作为键值时尤为重要。
- ◆ `hashCode()`方法：和 `equals()`方法一样，`hashCode()`方法在代理作为 `HashMap` 或类似的集合的键值时十分重要。
- ◆ 执行 `Advised` 接口上的方法：之前所说，默认状态下代理会实现 `Advised` 接口，这样就可以在代理生成后修改它，也可以查询有关该代理的信息。这个测试检查各个代理类型处理 `Advised` 接口上方法的速度。

从测试结果可以看出，CGLIB 代理的性能要远远超过 JDK 代理。执行被通知方法时，标准 CGLIB 代理的性能略好过 JDK 代理，不过当我们使用固定通知链 CGLIB 代理时，性能差异就十分明显了。对于未被执行的通知，CGLIB 代理比 JDK 代理快 8 倍以上。至于 `Advised` 接口上的方法，我们注意到 CGLIB 也会快一些，但不会快太多。这是因为 `Advised` 方法在 `intercept()`的较前部分中执行，所以很多别的方法需要的逻辑都被跳过了。

当我们代理类时，CGLIB 代理是默认选项，因为只有它能生成类的代理。如果要在代理接口时使用 CGLIB，我们就必须用 `setOptimize()`设定优化方法将 `ProxyFactory` 类的优化标志设为 `TRUE`。

第 6 章 AOP 进阶

探讨 Spring 中的框架服务如何让我们能透明地使用 AOP，讨论如何使用 Spring 和 AspectJ 的集成来克服 Spring AOP 的限制。

6.1 @AspectJ 注解

@AspectJ 跟 AspectJ 没有关系，Spring 用来解析连接点和通知的一组 Java 5 注解。@AspectJ 简单地告诉 Spring 将这个 bean 视为一个方面，也就是说，从中找出切入点和通知。接着，让 AspectJ 切入点表达式标上 @Before @After @Around 等注解。切入点表达式有

- ◆ **execution** 匹配方法执行连接点
- ◆ **within** 匹配那些在已声明的类型中执行的连接点。例如 `within(com.apress..TestBean)` 匹配从 `TestBean` 的方法中产生的调用
- ◆ **this** 通过用 bean 引用(即 AOP 代理)的类型跟指定的类型作比较来匹配连接点。例如，`this(SimpleBean)` 将只会匹配在 `SimpleBean` 类型的 bean 上的调用
- ◆ **target** 通过用被调用的 bean 的类型和指定的类型作比较来匹配连接点。比如 `target(SimpleBean)` 将只会匹配在 `SimpleBean` 类型 bean 上方法的调用。
- ◆ **args** 通过比较方法的参数类型跟指定的参数类型来匹配连接点。例如，`args(String,String)` 将只会匹配那些有 2 个 `String` 类型参数的方法
- ◆ **@target** 通过检查调用的目标对象是否具有特定注解来匹配连接点。例如 `@target(Magic)` 将只会匹配带有 @Magic 注解的类中的方法调用
- ◆ **@args** 跟 `args` 相似，不过 @args 检查的是方法参数的注解而不是它们的类型。例如 `@args(NotNull)` 会匹配所有那些包含一个被标注了 @NotNull 注解的参数的方法
- ◆ **@within** 跟 `within` 相似，这个表达式匹配那些带有特定注解的类中执行的连接点。例如，表达式 `@within(Magic)` 将会匹配对带有 @Magic 注解的类型的 bean 上方法的调用。
- ◆ **@annotation** 通过检查将被调用的方法上的注解是否为指定的注解来匹配连接点。例如 `@annotation(Magic)` 将会匹配所有标有 @Magic 注解的方法调用
- ◆ **bean** 通过比较 bean 的 ID(或名称)来匹配连接点。我们也可以在 bean 名模式中使用通配符。

****this** 和 **target** 不能使用通配符

我们可以使用 `||` 和 `&&` 运算符来组合多个切入点表达式，并使用 `!(非)` 运算符来对表达式的值取否。

6.2.5 通知的类型

1 前置通知

@Before

参见 `com.apress.prospring2.ch06.services.BeforeAspect` 方面，里面有一些解释容易理解。

2 后置通知

@AfterReturning 参见 `com.apress.prospring2.ch06.services.AfterAspect`

3 抛出后通知

@AfterThrowing 参见 `com.apress.prospring2.ch06.services.AfterThrowingAspect`

4 后置通知

后置通知的最后一类是后置，或更正式来说是最终通知。无论目标方法是正常完成还是抛出异常它都会执行。然而我们得不到有关方法的返回值或任何已抛出异常的任何信息。大多数情况下使用最终通知来释放资源，就像在 Java 中使用 `try/catch/finally` 一样

5 包围通知

包围通知是最强大且最复杂的通知，它围绕着方法调用而执行。因此该通知需要至少一个参数，并且必须返回一个值。这个参数指定被调用的目标，而返回值指定目标的返回值，不用管返回值从何而来。你会发现包围通知一般用于事务的管理。通知会在进行目标调用之前开始一个事务，如果目标正常返回，则通知会提交该事务，如果碰上异常，则会回滚操作。包围通知的另外一个例子是缓存。参见 `com.apress.prospring2.ch06.services.CachingAspect`

Spring JDBC

处理二进制大对象(BLOB)、字符大对象(CLOB)以及从存储过程和函数返回的游标。JDBC 框架的另一个重要特性是批处理更新。此特性允许你在一个集合中执行大量的 JDBC 操作，因此能够极大地改善性能。

Spring 支持在简介中所介绍的所有的的基本 JDBC 操作，并增加了对异常处理和连接管理的良好支持。

9.2 Spring 对数据访问支持的概念

- ◇ 传统 JDBC 操作的缺陷：冗长的代码、受检查异常的缺陷。
- ◇ 打开连接的管理：数据库连接是稀有资源，如果我们打开的太多，数据库的性能将急剧下降，如果我们为整个应用程序维护一个活跃的连接，我们就不能充分使用数据库的潜能。但对连接池进行可靠的管理是很困难的。

- ◇ Spring 的异常体系

所有这些异常都是运行时异常，它比 JDBC 使用的受检查异常更合理。受检查异常应该用来指示应用程序能够修复的错误。但是你怎么可能从格式错误的 SQL 语句引起的异常中恢复呢？

除了丰富的异常体系外，所有的 Spring 数据访问代码使用与 `HibernateTemplate`、`JdoTemplate` 和 `JpaTemplate` 相同的模式。所有这些类维护后台稀有资源的安全(例如数据库连接和 `Hibernate` 会话)。Spring 也为数据访问实现提供方便的超类，它们是 `JdbcDaoSupport`、`HibernateDaoSupport`、`JdoDaoSupport`、`JpaDaoSupport` 等辅助类。

9.3 Spring 对 JDBC 数据访问的支持

Spring 对 JDBC 支持的核心是 `JdbcTemplate` 类。此类允许你直接工作于 `Statement`、`PreparedStatement` 和 `CallableStatement` 类的实例。`JdbcTemplate` 类帮助管理数据库连接和将 `SQLException` 异常转换为 Spring 数据访问异常。

在我们讨论 Spring 对 JDBC 的支持之前，我们先介绍一下 `DataSource`，它是一个 JDBC 接口，是一个可以替代 `DriverManager` 的可选方案。因为它是一个接口，所以有很多不同的实现存在(一般情况下使用 `org.apache.commons.dbcp.BasicDataSource`)

使用 `JdbcTemplate` 类 `execute()` 方法

```
package com.apress.prospring2.ch09.spring;

import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;
import java.util.LinkedList;
import java.util.List;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.dao.DataAccessException;
import org.springframework.jdbc.core.CallableStatementCallback;
import org.springframework.jdbc.core.CallableStatementCreator;
import org.springframework.jdbc.core.ConnectionCallback;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.PreparedStatementCallback;
import org.springframework.jdbc.core.PreparedStatementCreator;
import org.springframework.jdbc.core.StatementCallback;

public class JdbcTemplateDemo {

    private JdbcTemplate jdbcTemplate;

    private static class MyPreparedStatementCreator
        implements PreparedStatementCreator{
```

```
@Override
    public PreparedStatement createPreparedStatement(Connection
conn)

        throws SQLException {
        PreparedStatement ps = conn.prepareStatement("select * from
t_x where id=?");
        ps.setLong(1, 1L);
        return ps;
    }

}

private static class MyPreparedStatementCallback
implements PreparedStatementCallback{
    @Override
    public Object doInPreparedStatement(PreparedStatement ps)
        throws SQLException, DataAccessException {
        // TODO Auto-generated method stub
        ResultSet rs = ps.executeQuery();
        List<Long> ids = new LinkedList<Long>();
        while(rs.next())
            ids.add(rs.getLong(1));
        rs.close();
        return ids;
    }
}

private static class MyCallableStatementCreator
implements CallableStatementCreator{

    @Override
    public CallableStatement createCallableStatement(Connection
conn)

        throws SQLException {
        CallableStatement cs = conn.prepareCall("{? = call
f_calculate}");
        cs.registerOutParameter(1, Types.INTEGER);
        return cs;
    }
}

private static class MyCallableStatementCallback
implements CallableStatementCallback{
```

```

@Override
public Object doInCallableStatement(CallableStatement cs)
    throws SQLException, DataAccessException{
    cs.execute();
    return cs.getLong(1);
}

private void run(){
    //create the table t_x
    this.jdbcTemplate.execute(new ConnectionCallback(){
        @Override
        public Object doInConnection(Connection conn) throws
SQLException,
        DataAccessException {
            PreparedStatement createTable = conn.prepareStatement(
                "create table t_x (id number(19,0) not null, " +
                "constraint pk_x primary key(id))");
            createTable.execute();
            return null;
        }
    });
    //insert a test value
    this.jdbcTemplate.execute(new StatementCallback(){
        @Override
        public Object doInStatement(Statement statement) throws
SQLException,
        DataAccessException {
            return statement.execute("insert into t_x(id) values
(1)");
        }
    });
    this.jdbcTemplate.execute("insert into t_x(id) values (2)");

    //get data id s
    List<Long> ids;
    ids = (List<Long>) this.jdbcTemplate.execute(
        new MyPreparedStatementCreator(),
        new MyPreparedStatementCallback());
    System.out.println(ids);
    //get data with another way
    ids = (List<Long>) this.jdbcTemplate.execute("select id from t_x",
        new MyPreparedStatementCallback());
    System.out.println(ids);
}

```



```

        //execute function
        System.out.println(this.jdbcTemplate.execute(
            new MyCallableStatementCreator(),
            new MyCallableStatementCallback()));
        //execute procedure
        this.jdbcTemplate.execute("{call p_actstartled(42)}",
            new CallableStatementCallback() {
                @Override
                public Object doInCallableStatement(
                    CallableStatement cs) throws
SQLException,DataAccessException{
                    cs.execute();
                    return null;
                }
            });
        //drop the table
        this.jdbcTemplate.execute("drop table t_x");
    }

    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "/jdbcdao-context.xml", JdbcTemplateDemo.class);
        JdbcTemplateDemo demo =
(JdbcTemplateDemo) ac.getBean("jdbcTemplateDemo");
        demo.run();
    }

    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
}

<?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:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

```

```
<bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
    <property name="driverClassName"
value="oracle.jdbc.driver.OracleDriver"/>
    <property name="url"
value="jdbc:oracle:thin:@localhost:1521:XE"/>
    <property name="username" value="system"/>
    <property name="password" value="root"/>
</bean>
<bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource"/>
</bean>
<bean id="jdbcTemplateDemo"
      class="com.apress.prospring2.ch09.spring.JdbcTemplateDemo">
    <property name="jdbcTemplate" ref="jdbcTemplate"/>
</bean>

</beans>
```

[1]

[1, 2]

42

你可以使用各种 `JdbcTemplate.execute` 方法完成所有的数据访问操作。你既可以执行数据定义语句(包括删除表、视图等),也可以进行数据修改语句(数据的插入、更新、删除和查询)。因为 `execute` 方法是通用的,所以你可能认为写代码应该很简单。现在我们看一看如何使用 `JdbcTemplate` 的其他方法来简化 JDBC 代码

9.4.2 JdbcTemplate 类的 query 方法和该方法的扩展

`JdbcTemplate` 类处理行的回调接口

- 📖 **ResultSetExtractor** 此接口抽取从传给它的 `extractData` 方法的公开 `ResultSet` 的单个 `Object`,需要调用得到的 `ResultSet` 对象的 `next` 方法
- 📖 **RowCallbackHandler** `JdbcTemplate` 类为 `ResultSet` 的每一行数据调用此接口的 `processRow(ResultSet)`方法,因此, `processRow` 代码不应该调用 `ResultSet` 对象的 `next` 方法预先获得下一行
- 📖 **RowMapper** 此回调接口用于映射 `ResultSet` 的每一行数据给 `Object` 对象。然后, `JdbcTemplate` 轮循处理 `ResultSet`, 实现不应该调用 `ResultSet.next` 方法。从此方法返回的 `Object` 实例 将被作为一个 `List` 的成员从 `JdbcTemplate` 的方法返回,而方法使用 `RowMapper` 类型作为参数

`JdbcTemplate` 类语句回调接口

- 📖 PreparedStatementCreator 此接口的实现从 createPreparedStatement(Connection)方法返回一个 PreparedStatement 对象
- 📖 PreparedStatementSetter 此接口的 setValues(PreparedStatement) 方法用于设置 PreparedStatement 的参数 (或 从 一个 String 创建 或 从 PreparedStatementCreator.createPreparedStatement 方法返回)

```
//JdbcTemplate.query demo
public void runQuery(){
    //这里的CallbackHandler 可以看到此query不返回值，因为我们
    //只是希望在其processRow方法里面处理业务了，例如对未登录用户发送email
    this.jdbcTemplate.query("select first_name, last_name,
last_login from t_customer "+
        "where last_login is null",new RowCallbackHandler(){
        @Override
        public void processRow(ResultSet resultSet) throws
SQLException{
            //send email to resultSet.getString(1)
            System.out.println("Sending email to user
"+resultSet.getString(1));
        }
    });

    String machaceksName = (String) this.jdbcTemplate.query(
        new PreparedStatementCreator(){
            @Override
            public PreparedStatement createPreparedStatement(
                Connection conn) throws SQLException {
                return conn.prepareStatement(
                    "select first_name from t_customer where "+
                    "last_name like ?");
            }
        },
        new PreparedStatementSetter(){
            @Override
            public void setValues(PreparedStatement ps)
                throws SQLException {
                ps.setString(1, "Mach%");
            }
        },
        new ResultSetExtractor(){
            @Override
            public Object extractData(ResultSet resultSet)
                throws SQLException, DataAccessException {
                if(resultSet.next()){
                    return resultSet.getString(1);
                }
            }
        }
    );
}
```

```

    }
    return null;
}
});
System.out.println("We gotta "+machaceksName);
}

```

与上面代码合在一起的执行结果为

```

[1]
[1, 2]
42
Sending email to user Jan
Sending email to user David
We gotta Jan

```

9.4.3 JdbcTemplate 类的 update 方法

```

private void runUpdate(){
    this.jdbcTemplate.update("update t_customer set first_name = first_name || 'x'");
    this.jdbcTemplate.update("update t_customer set first_name=? where id=?",
        new Object[]{"Jenda",1L}, new int[]{Types.VARCHAR,Types.INTEGER});
}

```

9.4.4 JdbcTemplate 类的 batchUpdate 方法

批量更新(Batch Update)为大量独立的语句执行提供了极大的性能改善。使用批量更新的典型场景是当你需要插入很多行的时候(例如外部数据源导入)。JdbcTemplate 类提供了两个 batchUpdate 方法。一个使用简单 String 数组作为参数，批量执行数组中的 SQL 语句。该方法在执行大量的无参数 SQL 语句时比较有用。第二个 batchUpdate 重载方法使用单个 表示 SQL 语句的 String 和 BatchPreparedStatementSetter 对象两个参数。JdbcTemplate 类以批处理形式为每一条语句调用 BatchPreparedStatementSetter.setValues(PreparedStatement,int)实现。

```

private void runBatch() {
    final int count = 2000;
    final List<String> firstNames = new ArrayList<String>(count);
    final List<String> lastNames = new ArrayList<String>(count);
    for(int i = 0; i < count; i++){
        firstNames.add("First Name "+i);
        lastNames.add("Last Name "+i);
    }
    this.jdbcTemplate.batchUpdate(
        "insert into t_customer ("
        "id, first_name, last_name, last_login, comments) "+

```

```

        "values (?, ?, ?, ?, ?)",
        new BatchPreparedStatementSetter() {
            @Override
            public void setValues(PreparedStatement ps, int i)
                throws SQLException {
                ps.setLong(1, i + 10);
                ps.setString(2, firstNames.get(i));
                ps.setString(3, lastNames.get(i));
                ps.setNull(4, Types.TIMESTAMP);
                ps.setNull(5, Types.CLOB);
            }
            @Override
            public int getBatchSize() {
                return count;
            }
        });
    //执行过上述后 可提前把下行注释掉查看是否成功添加200行数据
    this.jdbcTemplate.batchUpdate(new String[]{
        "update t_customer set first_name = 'FN#'||id",
        "delete from t_customer where id > 2"
    });
}

```

9.5 RdbmsOperation 子类

RdbmsOperation 抽象类是让我们编写子类执行特定操作(更新、插入、删除和查询)的类体系基类。

类体系分为两条路线: SqlCall 体系, 代表无返回值(不是存储过程的输出参数)的语句; SqlOperation 体系, 用于处理有返回值的语句。同时值得注意的是, RdbmsOperation 类在内部使用了 JdbcTemplate

我们将从最简单的 RdbmsOperation 子类——SqlUpdate 类开始介绍, 然后讨论 SqlQuery 的子类, 最后是 StoredProcedure 类。

```

public class SqlUpdateDemo {

    private Insert insert;

    private static class Insert extends SqlUpdate{
        private static final String SQL =
            "insert into t_customer (id,first_name, last_name, last_login,
            comments) "+
            "values (?, ?, ?, ?, ?)";
        Insert(DataSource dataSource) {

```

```

        super(dataSource, SQL);
        declareParameter(new SqlParameter(Types.INTEGER));
        declareParameter(new SqlParameter(Types.VARCHAR));
        declareParameter(new SqlParameter(Types.VARCHAR));
        declareParameter(new SqlParameter(Types.TIMESTAMP));
        declareParameter(new SqlParameter(Types.CLOB));
    }
}

SqlUpdateDemo(DataSource dataSource) {
    this.insert = new Insert(dataSource);
}

private void runInsert() {
    this.insert.update(new Object[] {3L, "J", "Machacke", null, null});
    this.insert.update(new Object[] {4L, "J", "Doe", null, null});
}

public static void main(String[] args) {
    ApplicationContext ac = new ClassPathXmlApplicationContext(
        "/jdbcdao-context.xml", SqlUpdateDemo.class);
    SqlUpdateDemo me = new
SqlUpdateDemo((DataSource) ac.getBean("dataSource"));
    me.runInsert();
}
}

```

类型安全的更新类

```

public class SqlUpdateDemo {

    private Insert insert;

    private static class Insert extends SqlUpdate {
        private static final String SQL =
            "insert into t_customer (id,first_name, last_name, last_login,
comments) "+
            "values (?,?,?,?,?)";
        Insert(DataSource dataSource) {
            super(dataSource, SQL);
            declareParameter(new SqlParameter(Types.INTEGER));
            declareParameter(new SqlParameter(Types.VARCHAR));
            declareParameter(new SqlParameter(Types.VARCHAR));
            declareParameter(new SqlParameter(Types.TIMESTAMP));
            declareParameter(new SqlParameter(Types.CLOB));
        }
    }
}

```

```

        void insert(long id,String firstname,String lastname,Date
lastlogin,String comments){
            update(new
Object[]{id,firstname,lastname,lastlogin,comments});
        }
    }

    SqlUpdateDemo(DataSource dataSource){
        this.insert = new Insert(dataSource);
    }

    private void runInsert(){
        //this.insert.update(new
Object[]{3L,"J","Machacke",null,null});
        //this.insert.update(new Object[]{4L,"J","Doe",null,null});
        this.insert.insert(5L, "Anirvan", "Chakraborty", null, null);
    }

    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "/jdbcdao-context.xml",SqlUpdateDemo.class);
        SqlUpdateDemo me = new
SqlUpdateDemo((DataSource)ac.getBean("dataSource"));
        me.runInsert();
    }
}

```

使用命名参数

```

private static class NamedInsert extends SqlUpdate{
    private static final String SQL =
        "insert into t_customer
(id,first_name,last_name,last_login,comments) "+
        "values(:id,:firstName,:lastName,:lastLogin,:comments)";
    NamedInsert(DataSource dataSource){
        super(dataSource,SQL);
        declareParameter(new SqlParameter(Types.INTEGER));
        declareParameter(new SqlParameter(Types.VARCHAR));
        declareParameter(new SqlParameter(Types.VARCHAR));
        declareParameter(new SqlParameter(Types.TIMESTAMP));
        declareParameter(new SqlParameter(Types.CLOB));
    }
}

```



```
private void runInsert3() {
    Map<String, Object> parameterMap = new HashMap<String, Object>();
    parameterMap.put("id", 6L);
    parameterMap.put("firstName", "John");
    parameterMap.put("lastName", "Appliseed");
    parameterMap.put("lastLogin", null);
    parameterMap.put("comments", null);
    this.namedInsert.updateByNamedParam(parameterMap);
}
```

9.5.2 BatchSqlUpdate

```
package com.apress.prospring2.ch09.spring;

import java.sql.Types;

import javax.sql.DataSource;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.BatchSqlUpdate;

public class BatchSqlUpdateDemo {
    private BatchInsert batchInsert;

    public BatchSqlUpdateDemo(DataSource dataSource) {
        // TODO Auto-generated constructor stub
        batchInsert = new BatchInsert(dataSource);
    }

    private static class BatchInsert extends BatchSqlUpdate{
        private final static String SQL =
            "insert into t_customer (id,first_name, last_name, last_login, "
            + "comments) " +
            "values (?, ?, ?, ?, null)";

        public BatchInsert(DataSource dataSource) {
            // TODO Auto-generated constructor stub
            super(dataSource, SQL);
            declareParameter(new SqlParameter(Types.INTEGER));
        }
    }
}
```

```
        declareParameter(new SqlParameter(Types.VARCHAR));
        declareParameter(new SqlParameter(Types.VARCHAR));
        declareParameter(new SqlParameter(Types.TIMESTAMP));

        batchSize(101);
    }
}

private void run() {
    int count = 5000;
    for(int i = 0; i < count; i++){
        this.batchInsert.update(new Object[]{i+100L, "a" + i, "b" + i,
null});
    }
    this.batchInsert.flush();
}

public static void main(String[] args) {
    ApplicationContext ac = new ClassPathXmlApplicationContext(
        "/jdbcdao-context.xml");
    BatchSqlUpdateDemo me = new BatchSqlUpdateDemo(
        (DataSource)ac.getBean("dataSource"));
    me.run();
}
}
```

9.5.3 SqlCall 类和 StoredProcedure 子类

```
package com.apress.prospring2.ch09.spring;

import java.sql.Types;
import java.util.Collections;
import java.util.Map;

import javax.sql.DataSource;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.jdbc.core.SqlOutParameter;
```

```
import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.StoredProcedure;

public class SqlCallDemo {
    private static final Log logger =
LogFactory.getLog(SqlCallDemo.class);

    private MeaningOfLife meaningOfLife;
    private ActStartled actStartled;

    public SqlCallDemo(DataSource dataSource) {
        meaningOfLife = new MeaningOfLife(dataSource);
        actStartled = new ActStartled(dataSource);
    }

    private static class MeaningOfLife extends StoredProcedure{
        private static final String SQL =
            "f_calculate";
        public MeaningOfLife(DataSource dataSource) {
            // TODO Auto-generated constructor stub
            super(dataSource, SQL);
            setFunction(true);
            declareParameter(new SqlOutParameter("n", Types.INTEGER));
        }
    }

    private static class ActStartled extends StoredProcedure{
        private static final String SQL = "p_actstartled";
        public ActStartled(DataSource dataSource) {
            // TODO Auto-generated constructor stub
            super(dataSource, SQL);
            declareParameter(new SqlParameter("n", Types.INTEGER));
        }
    }

    private void run() {
        Map result =
this.meaningOfLife.execute(Collections.emptyMap());
        logger.info(result.values().iterator().next());
        this.actStartled.execute(result);
    }

    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "/jdbcdao-context.xml");
    }
}
```

```

        SqlCallDemo demo = new SqlCallDemo(
            (DataSource) ac.getBean("dataSource"));
        demo.run();
    }
}

```

注意我们不必为函数([?]=call function-name)考虑 CallableStatement 的语法。我们所需要的就是提供函数名和调用 setFunction(true)。基于上述原因，由于我们只声明了一个输出参数，因此 StoreProcedure 代码为我们处理了所有困难工作，生成了语句 ?=call f_calculate()。因为存储函数或过程可以返回多个输出参数，所以 execute 方法的结果是一个 Map，它的键是参数名键，值为参数的值。

在上述代码中，我们隐式命名了参数“n”，如果没有名称，我们不能为得到 map 键值。因为 f_calculate 存储函数返回一个数值，而存储过程 p_actstartled 使用该数值，在这种情况下我们可以将它命名为“n”。这可以让我们获得从 this.meaningOfLife.execute(Collecitons.emptyMap()) 方法返回的值，在 this.actStartled.execute 调用中使用它。

9.5.4 SqlQuery 类和它的子类

```

public class SqlQueryDemo {
    private final static Log logger =
LogFactory.getLog(SqlQueryDemo.class);
    private static class SelectCustomer extends SqlQuery{
        private final static String SQL =
            "select * from t_customer";
        SelectCustomer(DataSource dataSource){
            super(dataSource, SQL);
        }
        @Override
        protected RowMapper newRowMapper(Object[] parameters, Map context)
        {
            // TODO Auto-generated method stub
            return new ColumnMapRowMapper();
        }
    }

    private SelectCustomer selectCustomer;
    SqlQueryDemo(DataSource dataSource){
        this.selectCustomer = new SelectCustomer(dataSource);
    }

    public void run(){
        List customers = this.selectCustomer.execute();
        logger.info(customers);
    }
}

```

```

    }

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext(
            "/jdbcdao-context.xml", SqlQueryDemo.class);
        SqlQueryDemo me = new SqlQueryDemo(
            (DataSource) context.getBean("dataSource"));
        me.run();
    }
}

```

可以看到结果是 Map 对象的 List，List 中的元素是所查询的行数据，每一行量个 Map，其键是数据行的列名，键值为数据行的列值。判断哪一个对象将会作为 List 的元素成员返回的方法是 newRowMapper(Object[], Map) 方法返回的 RowMapper 对象，我们使用了 ColumnMapRowMapper，它返回一个由列名和列值构造的 Map。

【自己体会：我们可以这样思考，既然我们继承了 SqlQuery 虚基类，我们要用它来完成从数据库中查询数据，既然查询我们就要求其返回数值，但是返回的数值我们是要原生态的还是包装一下呢？毫无疑问，当然要包装一下，因为我们不知道什么原生态的数据用 Java 怎么返回，所以要包装一下，即对返回的列数据进行映射，所以在查询数据后类中要找到用户想采取的 RowMapper，以决定返回什么，RowMap 就是用来包装列名和列值的，这里 ColumnMapRowMapper 是将每一行数据按列名为 k 列值为 v 包装为一个 Map，这样一行一行的堆积起来就是一个 List。我们也可以按其它的堆积方法，即 BeanPropertyRowMapper(SomeClass.class)，这样它会自动的将列名与类中的属性名对应起来，自动调用其 setter 方法设置属性，如果我们想客户化的话，即当某一个列值为空我们不想返回空而返回一个默认的话，我们就要对此类进行定制，即让自己的类继承 MappingSqlQueryParameters 类，在其 mapRow 方法里自己设定，有一个 Map 参数，此参数就是用来放置默认值的地方，见下面的代码，可得到很好的启示】

```
package com.apress.prospring2.ch09.spring;
```

```
import java.sql.Date;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
```

```
import javax.sql.DataSource;
```

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.jdbc.object.MappingSqlQueryWithParameters;
```

```
public class SqlQueryDemo2 {
    private final static Log logger = LoggerFactory.getLog(SqlQueryDemo2.class);
    private final static String LAST_LOGIN_DATE = "last_login";
    private static class MappingSelectCustomer extends MappingSqlQueryWithParameters{
        private final static String SQL =
            "select * from t_customer";
        MappingSelectCustomer(DataSource dataSource){
            super(dataSource,SQL);
        }
        @Override
        protected Object mapRow(ResultSet rs, int rownum, Object[] parameters,
            Map context) throws SQLException {
            // TODO Auto-generated method stub
            Customer customer = new Customer();
            customer.setId(rs.getLong("id"));
            customer.setFirstName(rs.getString("first_name"));
            customer.setLastLogin(rs.getDate("last_login"));
            customer.setLastName(rs.getString("last_name"));
            if(rs.isNull()) customer.setLastLogin(null);
            if(context != null){
                if(context.containsKey(LAST_LOGIN_DATE))
                    customer.setLastLogin((Date)context.get(LAST_LOGIN_DATE));
            }
            return customer;
        }
        public List execute(Date defaultLastLoginDate){
            Map<String,Object> context = new HashMap<String,Object>();
            context.put(LAST_LOGIN_DATE, defaultLastLoginDate);
            return execute(context);
        }
    }

    private MappingSelectCustomer mappingSelectCustomer;
    SqlQueryDemo2(DataSource dataSource){
        this.mappingSelectCustomer = new MappingSelectCustomer(dataSource);
    }
    public void run(){
        List result;
        result = this.mappingSelectCustomer.execute(new Date(System.currentTimeMillis()));
        logger.info(result);
    }

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext(
```

```

        "/jdbcdao-context.xml", SqlQueryDemo2.class);
    SqlQueryDemo2 me = new SqlQueryDemo2(
        (DataSource)context.getBean("dataSource"));
    me.run();
}
}

```

最后,我们有时需要调用返回单值的函数,例如, `select sysdate from dual`.在这种情况下, `SqlFunction` 类是一个很好的选择。实际上,大多数情况,我们甚至无需构建子类来调用此函数。

```

package com.apress.prospring2.ch09.spring;

import javax.sql.DataSource;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.jdbc.object.SqlFunction;

public class SqlFunctionDemo {
    private SqlFunction two;
    private static final Log logger =
        LogFactory.getLog(SqlFunctionDemo.class);

    public SqlFunctionDemo(DataSource dataSource) {
        // TODO Auto-generated constructor stub
        this.two = new SqlFunction(dataSource, "select sysdate from
dual");
    }

    void run() {
        logger.info(this.two.runGeneric());
    }

    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "/jdbcdao-context.xml", SqlFunctionDemo.class);
        SqlFunctionDemo me =
            new SqlFunctionDemo((DataSource)ac.getBean("dataSource"));
        me.run();
    }
}

```


9.5.5 JdbcTemplate 类和 RdbmsOperation 类的比较

实现一个 `RdbmsOperation` 子类比使用 `JdbcTemplate` 类要做更多的工作。多做这些工作的好处是什么呢？想一想为了执行一个 `SQL` 语句数据库必须做的工作：它必须解析语句，校验语句的语法是否正确，创建一个执行计划，最后运行计划。创建执行计划的过程可能相当困难，特别是当 `SQL` 语句包含大量的连接操作并且有很多索引时。数据库检查每一个表和索引的统计数据来决定执行语句的最优策略。所以这些需要做的很多工作，因此大多数数据库缓存了语句的执行计划。

如果你使用 `JdbcTemplate` 类，那么你每次调用它的方法就会创建一个新语句，除非一些数据库识别出了重复的 `SQL` 代码被看作同一个语句执行。对于高性能操作，最好实现一个特定的 `RdbmsOperation` 子类。另一个好处是子类通常是非常适合专用，可以让你以比使用 `Object` 实例数组更优雅的方式传进参数。

9.6 大二进制对象

`LOB`(Large binary object)是大文本块或二进制块，它们的大小从只有几千字节文本到上兆字节二进制数据。每一种数据库都有专有策略来存储这些数据，很不巧，这只是这种数据库的私有访问方式。例如，`PreparedStatement` 类的 `setClob(int,Clob)`方法。如果所有数据遵循 `JDBC` 规范，我们可以毫无问题地使用它。即使所有数据库使用标准 `JDBC` 实现进行工作，但是用此方法也会引起一些与资源管理有关的问题。

幸运的是，`Spring` 使用它的 `LobHandler` 接口和它的实现 `DefaultLobHandler` 类和 `OracleLobHandler` 类来处理该问题。`JDBC` 驱动程序的大多数 `LOB` 支持代码与底层的数据库关系密切。在使用它时，上述情况可能会引起问题，例如，可代替 `JDBC` 实现的连接池实现了 `Connection`，但是不是 `LOB` 代码所希望的本地 `Connection`，因此，`LobHandler` 可以使用一个 `NativeJdbcExtractor` 接口，它从大量的 `Connection` 适配器中返回一个本地的 `Connection`。

下表列出了在 `Spring` 中可用的 `NativeJdbcExtractor`

`C3PONativeJdbcExtractor`

`CommansDbcpNativeJdbcExtractor`

`JBossNativeJdbcExtractor`

`Jdbc4NativeJdbcExtractor` 如果使用 `JDK 1.6` 和 `JDBC 4`，我们可以使用此提取器得到未封装的连接。驱动程序必须是一个 `JDBC4` 的驱动程序，试图在 `JDK 1.6` 中运行 `JDBC 3` 驱动程序将会失败。

`SimpleNativeJdbcExtractor` 此提取器用于从封装的 `Connection` 元数据提取本地本地 `Connection`。封装代码只封装 `Connection` 实例而不是 `DatabaseMetaData` 对象。

`WebLogicNativeJdbcExtractor`

`WebSphereNativeJdbcExtractor`

`XAPoolNativeJdbcExtractor`

见下面的代码

```
package com.apress.prospring2.ch09.spring;
```

```
import java.sql.PreparedStatement;
```

```
import java.sql.SQLException;
import java.sql.Timestamp;

import javax.sql.DataSource;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.PreparedStatementSetter;
import org.springframework.jdbc.support.lob.LobHandler;

public class LobDemo {

    private JdbcTemplate jdbcTemplate;
    private LobHandler lobHandler;

    LobDemo(DataSource dataSource, LobHandler lobHandler) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        this.lobHandler = lobHandler;
    }

    private void runInTemplate() {
        this.jdbcTemplate.update("insert into c_customer "+
            "(first_name,last_name,last_login,comments) values "+
            "(?,?,?, ?,?)",
            new PreparedStatementSetter() {
                @Override
                public void setValues(PreparedStatement ps)
                    throws SQLException {
                    ps.setLong(1, 2L);
                    ps.setString(2, "Jan");
                    ps.setString(3, "Machacek");
                    ps.setTimestamp(4, new
Timestamp(System.currentTimeMillis()));
                    lobHandler.getLobCreator().setClobAsString(ps, 5, "This
is a loooong String");
                }
            });

        logger.info(
            this.jdbcTemplate.queryForObject("select comments from
t_customer where id=?",
            new Object[] {2L}, new RowMapper() {
```

```

        @Override
        public Object mapRow(ResultSet rs, int columnIndex)
            throws SQLException {
            // TODO Auto-generated method stub
            return lobHandler.getClobAsString(rs, 1);
        }
    }));
}

public static void main(String[] args) {
    ApplicationContext ac = new ClassPathXmlApplicationContext(
        "/jdbcdao-context.xml");
    LobDemo demo = new
    LobDemo((DataSource)ac.getBean("dataSource"), (LobHandler)ac.getBean("
    lobHandler"));
    demo.runInTemplate();
}
}
<!-- Large Binary Object -->
<bean id="nativeJdbcExtractor"

    class="org.springframework.jdbc.support.nativejdbc.CommonsDbcpNat
    iveJdbcExtractor"/>
<bean id="lobHandler"
    class="org.springframework.jdbc.support.lob.OracleLobHandler">
    <property name="nativeJdbcExtractor" ref="nativeJdbcExtractor"/>

</bean>

```

9.7 JdbcDaoSupport 类

Spring 提供 JdbcDaoSupport 类的原因: 虽然我们可以在应用程序的类内使用已经介绍过的所有 JDBC 支持类。然而, 很可能将只在应用程序的数据访问层使用它们。如果使用这个类作为你应用程序数据访问层类的超类, 那么我们可以通过调用 getJdbcTemplate 和 getDataSource 方法来访问 JdbcTemplate 和 DataSource 对象。另外 JdbcDaoSupport 实现了 InitializingBean, 甚至这实现了被声明为 final 的 afterPropertiesSet 方法, 也提供了一个可重载的 initDao 方法, 可以使用它的重载方法执行额外的代码。

```
package com.apress.prospring2.ch09.spring;
```

```
import java.sql.Types;
import java.util.List;
import java.util.Map;
```

```
import javax.sql.DataSource;

import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.SqlParameter;
import
org.springframework.jdbc.core.simple.ParameterizedBeanPropertyRowMapper;

import org.springframework.jdbc.core.support.JdbcDaoSupport;
import org.springframework.jdbc.object.SqlQuery;

public class JdbcCustomerDao extends JdbcDaoSupport implements
CustomerDao{

    private SelectCustomerById selectCustomerById;
    private static class SelectCustomerById extends SqlQuery{
        SelectCustomerById(DataSource ds){
            super(ds,"select * from t_customer where id=?");
            declareParameter(new SqlParameter(Types.INTEGER));
        }
        @Override
        protected RowMapper newRowMapper(Object[] parameters, Map context)
        {
            // TODO Auto-generated method stub
            return
ParameterizedBeanPropertyRowMapper.newInstance(Customer.class);
        }
    }

    @Override
    protected void initDao() throws Exception {
        // TODO Auto-generated method stub
        this.selectCustomerById = new
SelectCustomerById(getDataSource());
    }

    @SuppressWarnings("unchecked")
    @Override
    public List<Customer> getAll() {
        // TODO Auto-generated method stub
        return getJdbcTemplate().query("select * from t_customer",

ParameterizedBeanPropertyRowMapper.newInstance(Customer.class));
    }
}
```

```

@Override
public Customer getById(Long id) {
    // TODO Auto-generated method stub
    return (Customer) this.selectCustomerById.findObject(id);
}

}

package com.apress.prospring2.ch09.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class DataAccessTierDemo {

    private static final Log logger =
LogFactory.getLog(DataAccessTierDemo.class);
    private CustomerDao customerDao;
    DataAccessTierDemo(CustomerDao customerDao) {
        this.customerDao = customerDao;
    }
    private void run() {
        logger.info(this.customerDao.getAll());
        logger.info(this.customerDao.getById(1L));
    }

    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "/jdbcdao-context.xml");
        DataAccessTierDemo demo = new DataAccessTierDemo(
            (CustomerDao)ac.getBean("customerDao"));
        demo.run();
    }
}

<!-- JdbcDaoSupport -->
<bean id="abstractJdbcDao" abstract="true"
    class="org.springframework.jdbc.core.support.JdbcDaoSupport">
    <property name="dataSource" ref="dataSource"/>
</bean>
<bean id="customerDao"
    class="com.apress.prospring2.ch09.spring.JdbcCustomerDao"
    parent="abstractJdbcDao"/>

```

CustomerDao 类的作用的确不太重要，但是在设计、实现数据访问时这是推荐的方式。数据访问方法位于在后面要实现的接口内，我们将在下一章介绍如何使用 Hibernate 对象关系映射简化数据访问代码。

9.8 简单的 Spring JDBC

Spring 2.5 有几个已经简化的经典 JDBC 支持核心类。简化有两部分：使用泛型和调用代码更简单安全；简化类只包含最常使用的方法。

9.8.1 SimpleJdbcTemplate 类

```
package com.apress.prospring2.ch09.spring;

import java.sql.Date;

import javax.sql.DataSource;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcTemplate;

public class SimpleJdbcTemplateDemo {

    private static final Log logger =
LogFactory.getLog(SimpleJdbcTemplateDemo.class);
    private SimpleJdbcTemplate jdbcTemplate;

    SimpleJdbcTemplateDemo(DataSource ds) {
        this.jdbcTemplate = new SimpleJdbcTemplate(ds);
    }
    /**
     *使用BeanPropertySqlParameterSource插入数据
     * */
    private void run() {
        Customer customer = new Customer();
```

```
customer.setId(3L);
customer.setFirstName("FN");
customer.setLastLogin(new Date(System.currentTimeMillis()));
customer.setLastName("LN");
SqlParameterSource parameterSource =
    new BeanPropertySqlParameterSource(customer);
this.jdbcTemplate.update("insert into t_customer "+
    "(id,first_name,last_name,last_login,comments) "+
    "values
(:id,:firstName,:lastName,:lastLogin,:comments)",
    parameterSource);
}
/**
 *使用SimpleJdbcTemplate类的批量更新
 */
private void run2() {
    int count = 1000;
    SqlParameterSource[] source = new SqlParameterSource[count];
    for(int i = 0; i < count; i++){
        Customer c = new Customer();
        c.setId(i+100L);
        c.setFirstName("FN #" + i);
        c.setLastName("LN #" + i);
        c.setLastLogin(new Date(System.currentTimeMillis()));
        source[i] = new BeanPropertySqlParameterSource(c);
    }
    this.jdbcTemplate.batchUpdate("insert into t_customer "+
        "(id,first_name,last_name,last_login,comments) "+
        "values(:id,:firstName,:lastName,:lastLogin,:comments)",
        source);
}
/**
 * 使用SimpleJdbcTemplate类查询数据
 */
private void run3() {
    List<Customer> customers = this.jdbcTemplate.query("select * from
t_customer",

    ParameterizedBeanPropertyRowMapper.newInstance(Customer.class));
    logger.info(customers);
}

public static void main(String[] args) {
```

```
ApplicationContext ac = new ClassPathXmlApplicationContext(
    "/jdbcdao-context.xml");
SimpleJdbcTemplateDemo demo = new SimpleJdbcTemplateDemo(
    (DataSource)ac.getBean("dataSource"));
demo.run2();
}
}
```

9.8.2 SimpleJdbcCall 类

```
package com.apress.prospring2.ch09.spring;

import java.math.BigDecimal;
import java.sql.Date;
import java.util.Collections;
import java.util.List;
import java.util.Map;

import javax.sql.DataSource;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
import
org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import
org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import
org.springframework.jdbc.core.simple.ParameterizedBeanPropertyRowMapper;
import org.springframework.jdbc.core.simple.SimpleJdbcCall;
import org.springframework.jdbc.core.simple.SimpleJdbcTemplate;

public class SimpleJdbcTemplateDemo {

    private static final Log logger =
LogFactory.getLog(SimpleJdbcTemplateDemo.class);
    private SimpleJdbcTemplate jdbcTemplate;
    private DataSource dataSource;
```



```
SimpleJdbcTemplateDemo (DataSource ds) {
    this.dataSource = ds;
    this.jdbcTemplate = new SimpleJdbcTemplate(ds);
}

/**
 *使用BeanPropertySqlParameterSource插入数据
 * */
private void run() {
    Customer customer = new Customer();
    customer.setId(3L);
    customer.setFirstName("FN");
    customer.setLastLogin(new Date(System.currentTimeMillis()));
    customer.setLastName("LN");
    SqlParameterSource parameterSource =
        new BeanPropertySqlParameterSource(customer);
    this.jdbcTemplate.update("insert into t_customer "+
        "(id,first_name,last_name,last_login,comments) "+
        "values
(:id,:firstName,:lastName,:lastLogin,:comments)",
        parameterSource);
}

/**
 *使用SimpleJdbcTemplate类的批量更新
 * */
private void run2() {
    int count = 1000;
    SqlParameterSource[] source = new SqlParameterSource[count];
    for(int i = 0; i < count; i++){
        Customer c = new Customer();
        c.setId(i+100L);
        c.setFirstName("FN #" + i);
        c.setLastName("LN #" + i);
        c.setLastLogin(new Date(System.currentTimeMillis()));
        source[i] = new BeanPropertySqlParameterSource(c);
    }
    this.jdbcTemplate.batchUpdate("insert into t_customer "+
        "(id,first_name,last_name,last_login,comments) "+
        "values(:id,:firstName,:lastName,:lastLogin,:comments)",
        source);
}

/**
 * 使用SimpleJdbcTemplate类查询数据
```

```

    * */
    private void run3(){
        List<Customer> customers = this.jdbcTemplate.query("select * from
t_customer",

        ParameterizedBeanPropertyRowMapper.newInstance(Customer.class));
        logger.info(customers);
    }

    /**
    * SimpleJdbcCall类
    * */
    private void runCall(){
        /**
        * 下面几行读起来像看小说，我们得到 五个最初的SimpleJdbcCall实例
        * 调用withFunctionName来设置要使用的函数名，然后调用withReturnValue
        * 来说明它是一个存储函数，而不是一个存储过程
        * 最后我们调用executeObject(Class<T>,Map)来说明我们将接收单一的结果
        * 类型T，所以无需将返回值转型。
        * */
        BigDecimal meaningOfLife = new SimpleJdbcCall(this.dataSource).
            withFunctionName("f_calculate").
            withReturnValue().
            executeObject(BigDecimal.class, Collections.emptyMap());
        logger.info(meaningOfLife);
        /**
        * 需要特别注意SimpleJdbcCall类的最后一个方法
        withoutProcedureColumnMetaDataAccess()
        * 我们从查看存储过程的源代码开始介绍，该存储过程为t_customer
        * 表的所有行返回refcursor
        *
        * create or replace package simplejdbc as
        type rc_customer_type is ref cursor return t_customer%rowtype;
        procedure p_find_customer(rc_customer_type out
rc_customer_type);
        end;

        create or replace package body simplejdbc as
        procedure p_find_customer(rc_customer_type out
rc_customer_type) is
        begin
            open rc_customer_type for select * from t_customer;
        end;
        end;

```

```

    * */
    Map<String, Object> result = new SimpleJdbcCall(this.dataSource).
        withProcedureName("simplejdbc.p_find_customer").
        withoutProcedureColumnMetaDataAccess().
        returningResultSet("rc_customer_type",

ParameterizedBeanPropertyRowMapper.newInstance(Customer.class)).
        execute(new
MapSqlParameterSource("rc_customer_type", null));
    System.out.println(result.keySet().toString());
    logger.info(result);
}

public static void main(String[] args) {
    ApplicationContext ac = new ClassPathXmlApplicationContext(
        "/jdbcdao-context.xml");
    SimpleJdbcTemplateDemo demo = new SimpleJdbcTemplateDemo(
        (DataSource)ac.getBean("dataSource"));
    demo.runCall();
}
}

```

我们必须使用 `withoutProcedureColumnMetaDataAccess()` 调用，因为我们声明了存储过程的 `rc_customer_type` 参数为 `out`。因此，Spring 根本不能发现参数存在。当我们调用此方法时，Spring 将因为调用参数的存在 执行我们的代码。运行代码的结果和我们预料的完全一致：一个 map，它包含一个被映射的基于 `rc_customer_type` 键的 Customer 对象实例。

9.8.3 SimpleJdbcInsert 类

```

/**
 * 最后让我们看一看插入数据的简单方式。
 * 该类提供了与SimpleJdbcCall类相似的功能。
 * 一些方法修改了它们调用的实例，返回this
 * 并且允许我们串联调用。
 * 让我们先开始看一看代码清单中的代码，它展示了
 * 如何 向t_customer表插入一行数据。
 * */
private void runInsert() {
    Customer c = new Customer();
    c.setId(4L);
    c.setFirstName("FN");
    c.setLastName("LN");
    c.setLastLogin(new Date(System.currentTimeMillis()));
    c.setComments("This is a long CLOB string. Mu-har-har!");
}

```

```
SimpleJdbcInsert insert = new SimpleJdbcInsert(this.dataSource);
insert.
    withTableName("t_customer").

    usingColumns("id", "first_name", "last_name", "last_login", "comments
").
    execute(new BeanPropertySqlParameterSource(c));
}
```

9.8.4 SimpleJdbcTemplate 类

9.9 小结

新简化的 JDBC 类提供了一种极好的方式来实现低层次数据访问代码，便利我们无需花费太多时间写冗长的 JDBC 代码。

根据我们以往的经验，大型应用程序数据访问层代码最好的组合方式是：在大多数数据访问代码中使用对象关系映射实现，对于特定目的或性能关键部分的代码使用 Spring JDBC 实现。确实如此，Spring JDBC 的批量处理能力允许我们在很短时间内运行大量的操作。当数据读写间隙运行操作时，Hibernate 和其它 ORM 工具都不能一直运行正确的批量操作。在处理罕见数据类型（例如非常大的 LOB 类型数据）时，使用 Spring JDBC 是非常有用的。在所有这些情况中，大多数 ORM 工具不能使用数据库专有的特性。

第十章 集成 iBATIS

如前一章介绍了如何在 Spring 应用程序中使用 JDBC，尽管 Spring 已经为简化 JDBC 编程作了很多努力，但我们在使用 JDBC 时仍然需要编写大量的代码。

一种避免编写这些代码的方法是使用 ORM（对象关系映射）工具，来映射数据记录并创建合适的 Java 对象。为了让该映射工具以标准方式访问 Java 对象（例如，让此映射工具也能够设置从 join 操作返回的引用对象），它们必须遵守 Java bean 的命名规范。

这里介绍两种基本的 ORM 框架：一种是可以完全依赖它生成 SQL 代码的框架，另一种是需要自己手动编写 SQL 代码的框架。前者可以加快开发速度，但是有时候生成的 SQL 代码不够有效；后者需要手动编写 SQL 代码，确保查询到正确的字段来创建相应的 Java bean。

此章将着眼于使用 iBATIS ORM 框架实现 Spring 应用程序的数据访问层（即 DAO）层。并且我们将讨论以下话题：

1. 配置文件： 在应用程序的数据层如何使用多个配置文件
2. 映射文件： 如何创建映射文件，iBATIS 如何使用这些文件映射所选记录到域对象的属性上。
3. 基本的数据库操作。 如何使用 iBATIS 实现查询、删除和更新操作，如何实现查询操作代表的各类数据关系，如何确保插入操作使用了正确的主键值。

10.1 iBATIS 简述

<http://www.ibatis.com> 是开发者用来编写自定义的 SQL 代码，以匹配 bean 的属性值的 ORM 工具。iBATIS 使用同一种方法插入和更新数据，这确实是一个好消息，因为有时候我们并不想更新整个记录（特别是当记录很大的时候）。大多数情况下，你只需要更新一个字段。可以用 iBATIS 编写自定义代码更新表中指定字段。

```
drop table t_customer;
truncate table t_customer;
create table t_Customer(
id number(19,0) not null,
first_name varchar2(50) not null,
last_name varchar2(50) not null,
last_login timestamp not null,
constraint PK_T_CUSTOMERID primary key (ID)
);
create sequence t_customer_id start with 1000;

select * from t_customer;
select t_customer_id.nextVal from dual;

insert          into          t_customer          values
(1, 'John', 'Smith', TO_DATE('07-08-2010', 'dd-MM-YYYY'));
```

配置文件

customerSqlMap.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE sqlMap PUBLIC "-//iBATIS.com//DTD SQL MAP 2.0//EN"
    "http://www.ibatis.com/dtd/sql-map-2.dtd" >

<sqlMap>
    <typeAlias type="com.apress.prospring2.ch10.domain.Customer"
alias="customer" />
    <resultMap class="customer" id="result">
        <result property="id" column="ID" />
        <result property="firstName" column="first_name" />
        <result property="lastName" column="Last_name" />
        <result property="lastLogin" column="last_login" />
    </resultMap>

    <select id="getAllCustomers" resultMap="result">
        select * from S_CUSTOMER
    </select>
</sqlMap>
```

dataaccess-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <!-- Data Source Bean -->
    <bean id="dataSource"
        class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
        <property name="driverClassName"
value="oracle.jdbc.driver.OracleDriver"/>
        <property name="url"
value="jdbc:oracle:thin:@localhost:1521:XE" />
        <property name="username" value="system" />
        <property name="password" value="root" />
    </bean>

    <!-- SqlMap setup for iBATIS Database Layer -->
    <bean id="sqlMapClient"

        class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
        <property name="configLocation" value="sqlMapConfig.xml" />
        <property name="dataSource" ref="dataSource" />
    </bean>
    <bean id="customerDao"

        class="com.apress.prospring2.ch10.data.SqlMapClientCustomerDao">
        <property name="sqlMapClient" ref="sqlMapClient" />
    </bean>
</beans>
```

sqlMapConfig.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE sqlMapConfig
    PUBLIC "-//ibatis.com//DTD SQL MAP Config 2.0//EN"
    "http://www.ibatis.com/dtd/sql-map-config-2.dtd">

<sqlMapConfig>
    <sqlMap resource="customerSqlMap.xml" />
</sqlMapConfig>
```

测试函数

```
package com.apress.prospring2.ch10.demo;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.apress.prospring2.ch10.dao.CustomerDao;

public class Main {
    private ApplicationContext context;
    private void run() {
        System.out.println("Initializing application");
        context = new
ClassPathXmlApplicationContext("/dataaccess-context.xml");

        System.out.println("Getting customerDao");
        CustomerDao customerDao =
(CustomerDao) context.getBean("customerDao");

        System.out.println("Done");
    }
    public static void main(String[] args) {
        new Main().run();
    }
}
```

10.3 查询数据

我们将介绍如何从数据库中查询记录，以及如何从 DAO 实现中传递参数到查询语句中。我们以讨论对使用域对象的单表查询作为开始，然后接着讨论如何从 1:1 1:N M:N 的数据关系中查询数据

10.3.1 简单查询操作

customerSqlMap.xml 配置文件

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE sqlMap PUBLIC "-//ibatis.com//DTD SQL MAP 2.0//EN"
    "http://www.ibatis.com/dtd/sql-map-2.dtd" >

<sqlMap>
    <typeAlias type="com.apress.prospring2.ch10.domain.Customer"
alias="customer" />
```

```

<resultMap class="customer" id="result">
    <result property="id" column="ID" />
    <result property="firstName" column="first_name" />
    <result property="lastName" column="Last_name" />
    <result property="lastLogin" column="last_login" />
</resultMap>

<select id="getAllCustomers" resultMap="result">
    select * from S_CUSTOMER
</select>
<select id="getCustomerById" resultMap="result"
parameterClass="long">
    select * from S_CUSTOMER where id=#value#
</select>
<select id="getCustomersByLastNameAndLastLoginMap"
    resultMap="result" parameterClass="map">
    select * from s_customer where last_name like #lastName# and
        last_login=#lastLogin#
</select>
<select id="getCustomersByLastNameAndLastLoginDO"
    resultMap="result" parameterClass="customer">
    select * from s_customer where last_name like #lastName# and
        last_login=#lastLogin#
</select>
</sqlMap>

```

关键代码

```

public List<Customer> getByLastNameAndLastLoginMap(String lastName,
    Date lastLogin){
    Map<String,Object> parms = new HashMap<String,Object>();
    parms.put("lastName", lastName);
    parms.put("lastLogin", lastLogin);
    return getSqlMapClientTemplate().queryForList(
        "getCustomersByLastNameAndLastLoginMap",parms);
}

```

```

public List<Customer> getByLastNameAndLastLoginDO(String lastName,
    Date lastLogin){
    Customer c = new Customer();
    c.setLastName(lastName);
    c.setLastLogin(lastLogin);
    return getSqlMapClientTemplate().queryForList(
        "getCustomersByLastNameAndLastLoginDO",c);
}

```



```
}
```

---解释

有些情况下，你可能需要给数据库操作传递多个值。你可以使用任何类型，但是大多数时候，你会发现你要么使用域对象，要么使用`java.util.Map`。当然，使用`Map`和域对象并没有什么区别。然而在Java代码中，若你使用域对象，将会保证类型安全。规则就是，当你只是更新一个有大量列的表的几个字段，最好使用`Map`。当你更新表中的大部分或全部列时，最好使用域对象。需要注意的是，当使用`Map`的`java.util.HashMap`实现时，会碰到一些性能上的损失。我们已经分别测试了使用`java.util.HashMap`和域对象作为参数的25000个查询的iBatis调用。`HashMap`实现需要花费13594ms，而域对象实现花费了12328ms。

```
@Override
```

```
public Customer getById(Long id) {
    // TODO Auto-generated method stub
    return (Customer) getSqlMapClientTemplate().queryForObject(
        "getCustomerById", id );
}
```

---解释

此处的实现，调用了 `queryForObject()` 方法，该方法返回查询语句执行的结果：或是单个 `java.lang.Object` 或是 `null` 值。查询的行数如果超过 1，则会抛出 `DataAccessException` 异常。即使 `queryForObject()` 返回值是 `java.lang.Object`，我们也能安全转型到 `Customer`，因为 `resultMap` 说明对象类型是 `Customer`。同时我们也传递一个 `Long` 对象给 iBatis 调用，它的值设置为主键。

10.3.2 一对一查询操作

我们将增加另一个表及其域对象来演示如何从一对一映射中查询数据。让我们创建一些表和域对象。如下：我们新建了 `T_CUSTOMER` 表和 `T_CUSTOMER_DETAIL` 表

建表语句

```
create table t_customer_detail(
    id number(19,0) not null,
    data varchar2(512) not null,
    constraint pk_customerDetailId primary key(id)
);
drop table t_customer;
alter table s_customer drop constraint PK_CUSTOMERID;
create table t_customer(
    id number(19,0) not null,
    first_name varchar(50) not null,
    last_name varchar(50) not null,
    last_login timestamp not null,
    customer_detail int not null,
    customer_gossip int null,

    constraint PK_CUSTOMERID primary key (id),
```

```
constraint FK_CUSTOMERDetail foreign key(customer_detail)
references t_customer_detail(id) on delete cascade,
constraint FK_customerGossip foreign key(customer_gossip)
references t_customer_detail(id) on delete cascade
);

create sequence s_customer_id start with 1000;
insert into t_customer_detail(id,data) values (100,'Detail 1');
insert into t_customer_detail(id,data) values (101,'Foo');
insert into t_customer_detail(id,data) values (102,'Bar');
insert into t_customer(id,first_name,last_name,last_login,
customer_detail,customer_gossip) values (
1,'John','Smith',TO_DATE('08-08-2007','dd-MM-YYYY'),100,null
);

insert into t_customer(id,first_name,last_name,last_login,
customer_detail,customer_gossip) values (
2,'Jane','Doe',TO_DATE('08-08-2007','dd-MM-YYYY'),101,102
);

#other useful command
select constraint_name,table_name,r_owner, r_constraint_name
from all_constraints
where table_name='t_customer';

---Customer.XML 配置文件
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE sqlMap PUBLIC "-//iBATIS.com//DTD SQL MAP 2.0//EN"
"http://www.ibatis.com/dtd/sql-map-2.dtd" >

<sqlMap>
  <typeAlias type="com.apress.prospring2.ch10.domain.Customer"
alias="customer" />
  <typeAlias type="com.apress.prospring2.ch10.domain.CustomerDetail"
alias="customerDetail"/>
  <resultMap class="customer" id="result">
    <result property="id" column="ID" />
    <result property="firstName" column="first_name" />
    <result property="lastName" column="Last_name" />
    <result property="lastLogin" column="last_login" />
  </resultMap>

  <resultMap class="customerDetail" id="gossipResult">
    <result property="customerDetailId" column="id"/>
  </resultMap>
</sqlMap>
```

```
<result property="data" column="Data"/>
</resultMap>

<resultMap class="customer" id="resultDetail" extends="result">
  <result property="customerDetail.customerDetailId"
    column="Customer_Detail_ID"/>
  <result property="customerDetail.data"
    column="Customer_Detail_Data"/>
  <result property="customerGossip"
select="getCustomerGossipById"
    column="Customer_Gossip" />
</resultMap>

<select id="getCustomerById" resultMap="resultDetail"
parameterClass="int">
  select
  c.Id as Id,
  c.first_name as First_name,
  c.last_name as last_name,
  c.customer_detail as customer_detail,
  c.customer_gossip as Customer_Gossip,
  c.Last_login as Last_login,
  cd.id as Customer_detail_id,
  cd.data as Customer_detail_data
  from
  T_CUSTOMER c inner join T_Customer_Detail cd on
  c.customer_detail=cd.id
  where
  c.Id=#value#
</select>

<select id="getCustomerGossipById" resultMap="gossipResult"
parameterClass="int">
  select * from T_CUSTOMER_DETAIL where id=#value#
</select>
</sqlMap>
```

---解释

此 sqlMap 文件有许多新特性，让我们对它进行更详细的分析。首先，我们使用 resultMap 继承。这在你想新建一个向父 resultMap 添加更多字段的 resultMap 时很有用。例如，你想实现一个搜索方法并根据 lastName 来返回一个 customer 列表。如果你对 customerDetail 和 customerGossip 属性不感兴趣，那么你可以不用管它们，让它们使用默认值—两种情况都是 null。然而，如果你根据 customer.id 返回一个 customer，你将获得该 customer 所有可用信息。这就是我们既创建 resultMap 又创建 resultDetail resultMap 的原因。resultDetail 扩展了 result 并添加了 customerDetail 和 customerGossip 的定义。

我们在 `resultDetail resultMap` 中，说明，`customer_detail_id` 和 `customer_detail_data` 列一定会出现在 `resultSet` 中，它们用于设置 `customerDetail` 对象的 `id` 和 `data` 属性。而 `customerGossip` 对象设置为 `getCustomerGossipById` 查询语句的结果。该语句只接收单个 `long` 参数。参数的值从 `T_CUSTOMER` 表的 `CUSTOMER_GOSSIP` 列得到。换句话说，`customerDetail` 属性永远不会为 `null`，而 `CustomerGossip` 属性可以为 `null`。

一对一查询操作的性能

有一个性能问题需要考虑：使用一条查询语句设置域对象的属性会导致 **N+1** 条查询操作被执行。考虑这种情况，你加载单个 **Customer** 对象：要运行额外的查询来查询出 **Customer_Detail** 行，设置 **CustomerGossip** 属性。这个查询可能成为一个瓶颈。

但是我们有一个办法绕过它。我们可以返回 `customerDetail` 对象作为结果集的一部分，这就不需要执行更多的查询了。然而，一对一关系或是可选的，我们就不能使用这个方法了。解决此问题最好方法是，仅当你真正需要时才设置这些属性。使用一个简化的 `resultMap` 来为返回很多行的查询操作设置基本属性，而使用标准 `resultMap` 设置所有的属性。

10.3.3 一对多查询操作

建表语句

```
create table t_order(
    id number(19,0) not null,
    customer number(19,0) not null,

    constraint PK_ORDERID primary key(id),
    constraint FK_CUSTOMER foreign key(Customer) references
T_CUSTOMER(ID)
);

create sequence s_order_id start with 1000;

create table t_order_line(
    id number(19,0) not null,
    "Order" number(19,0) not null,
    Product varchar(200) not null,
    Price decimal(10,2) not null,

    constraint PK_orderlineId primary key (id),
    constraint FK_order foreign key("Order") references T_Order(id)
);

create sequence s_order_line_id start with 1000;
insert into t_order(id,Customer) values (100,1);
insert into T_Order_line(id,"Order",Product,Price)
    values(200,100,'Punch people over the internet client
```

```
application',19.95);
insert into t_order_line(id,"Order",Product,Price)
    values(201,100,'The managelfreuzer switch',12.95);
```

Order.xml sqlMap 配置文件

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE sqlMap PUBLIC "-//iBATIS.com//DTD SQL MAP 2.0//EN"
    "http://www.ibatis.com/dtd/sql-map-2.dtd" >

<sqlMap>
    <typeAlias type="com.apress.prospring2.ch10.domain.Order"
alias="order"/>
    <typeAlias type="com.apress.prospring2.ch10.domain.OrderLine"
alias="orderLine" />

    <resultMap class="order" id="result">
        <result property="id" column="id" />
        <result property="customer" column="customer"/>
        <result property="orderLines" select="getOrderLinesByOrder"
            column="Id"/>
    </resultMap>

    <resultMap class="orderLine" id="resultLine">
        <result property="id" column="id"/>
        <result property="order" column="Order"/>
        <result property="product" column="Product" />
        <result property="price" column="Price" />
    </resultMap>

    <select id="getOrderById" resultMap="result" parameterClass="long">
        select * from T_Order where id=#value#
    </select>

    <select id="getOrderLinesByOrder" resultMap="resultLine"
parameterClass="long">
        select * from T_ORDER_LINE where "Order"=#value#
    </select>
</sqlMap>
```

我们声明了为了设置 orderLines 属性, iBATIS 需要执行 getOrderLinesByOrder 查询语句, 将查询结果添加到 Order 对象的 orderLines 列表属性中。

测试

```
package com.apress.prospring2.ch10.demo;
```

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
```

```
import com.apress.prospring2.ch10.dao.OrderDao;
import com.apress.prospring2.ch10.domain.Order;

public class Main {
    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "/dataaccess-context.xml");
        OrderDao dao = (OrderDao)ac.getBean("orderDao");
        Order order = dao.getByld(100l);
        System.out.println(order);
    }
}
```

输出:

```
Order {id: 100,customer 1,orderLines: [orderLine {id: 200,order:
100,product: Punch people over the internet client application,price:
19.95},orderLine {id: 201,order: 100,product: The managelfreuzer
switch,price: 12.95},]}
```

一对多查询操作的性能

在一对多关系中，我们没有办法避免 $N+1$ 查询的问题。但是，我们仍可以使用标准 `resultMap` 设置基本属性，然后使用一个个更详细的 `resultMap` 来对标准 `resultMap` 进行扩展来设置所有属性。例如：如果需要显示一列 `Order`，无需知道 `orderLines`。只在显示该 `order` 的更详细的信息时需要这些额外的信息。这意味着通过它的主键查询 `order` 域对象，并产生另一条返回所有 `order` 行的查询语句。

10.3.4 多对多查询操作

最后，多对多查询仅仅是创建两个一对多关系的简单问题。例如，参考 `User` 和 `Role` 对象。一个 `User` 对象可以为多个 `Role`，而一个 `Role` 也可以分配给多个 `User`。我们需要一个关联表 `T_User_Role`，在 `User` 和 `UserRole` 间建立一对多关系，并在 `UserRole` 和 `Role` 之间建立一对多关系，那么 `User` 和 `Role` 之间就建立了多对多关系。

10.4 更新数据

```
<update id="updateCustomer" parameterClass="customer">
    update T_CUSTOMER
        set First_name=#firstName#,
            last_name=#lastName#,
            last_login=#lastLogin#
        where id=#id#
```

```
</update>
```

----性能测试表明，对不到 1000 行的表进行 5000 次的更新操作，Map 实现花费 19063ms，而域对象实现需要花费 15937ms。你不需要太关注是否使用 Map 实现，如果你的应用程序需完成较复杂的更新操作，相比实际的数据库工作开销来说，对象创建和查询开销绝对小得多。

关于数据更新还有一点，每一个 DAO 接口-实现对自己的域对象更新负责。考虑一种情况有两个域对象，Order 和 OrderLine。Order 拥有一个保存保存 OrderLine 对象的列表属性。你可能想在 OrderDao 实现中编写 SQL 语句完成 save() 操作，从而保存所有的 OrderLine 对象，但这绝不是一个好主意。如果这样做了 OrderDao 和 OrderLineDao 责任很模糊。我们需要在文档中写明，一旦 OrderDao.save(Order) 被调用，不需要为每一个 OrderLine 再调用 OrderLineDao.save(OrderLine)。并且，我们可能倾向于把自己的所有操作放到一个事务中，这种做法也不太好，因为不应该在代码里手动控制事务。将事务管理委托给 Spring 完成。

10.5 删除数据

```
<delete id="deleteCustomer" parameterClass="long">
```

```
    delete from T_Customer where id=#value#
```

```
</delete>
```

```
@Override
```

```
public void delete(Long id) {
    // TODO Auto-generated method stub
    getSqlMapClientTemplate().delete("deleteCustomer", id);
}
```

说明：当应用程序代码没有进行严格的责任分层设计（例如数据访问层、商业逻辑层或表现层）使程序的代码不易理解时，关注点溅出很常见。

10.6 插入数据

```
<insert id="insertCustomer" parameterClass="customer">
```

```
    <selectKey keyProperty="id" resultClass="int">
```

```
        select s_customer_id.nextVal from dual
```

```
    </selectKey>
```

```
    insert into T_Customer(Id,first_name,last_name,Last_login)
```

```
        values(#id#,#firstName#,#lastName#,#lastLogin#)
```

```
</insert>
```

```
private void insert(Customer customer){
    getSqlMapClientTemplate().insert("insertCustomer",customer);
}
```

10.7 iBATIS 缺少的特性

尽管 iBATIS 提供很高的性能-代码复杂度比,但仍然缺少几个特性。可能最令人烦恼的就是不支持枚举类型持久化和对象正规化。

以 User 域对象作为例子。一般情况下,系统使用的 role 数目有限。如果每一个 User 对象对应一个 userRole 属性,一个返回 1000 个 User 对象的查询语句也会创建 1000 个 UserRole 对象,即使大多数 UserRole 对象实际上只是代表同一个 Role,这样效率很低,导致大量内存被占用。这非常像 java.lang.Boolean 的情况,你可以创建一千个 Boolean 对象来表示 Boolean.TRUE。即使这 1000 个对象代表相同的值,你也要使用 1000 个独立的对象。缺少对象正规化的另一个弊病就是你必须使用 equals(Object)方法来对值进行比较,如果你有 1000 个相同对象引用的话,你可以使用“==”操作符。

另外,枚举类型的持久化是对 iBATIS 的一个很好的补充。Hibernate 允许你使用持久化枚举。实际上就是引用一个包含开放的 static <T> from(int)和 int toInt()方法的对象,int toInt()方法返回一个枚举中的 Int 值。Hibernate 创建该对象的一个实例。

上述问题只是从 iBATIS 中我们能想到的仅有问题,实际上它是一个非常杰出的 DAO 框架。

10.8 整体性能

当查询域对象属性时,我们需要考虑产生 N+1 性能瓶颈的可能性。但在大多数情况下,可以采用把 resultMap 实现分为一个用于返回大量的行的标准 resultMap 和另一个用于已经行数比较小时而设置比较复杂的属性扩展 resultMap,来规避性能问题。

iBATIS 最大的优势就是它不生成任何代码。尽管如上一小节所述的其缺少的一些特性外,iBATIS 是到目前为止最好的非侵入式 ORM 框架。

第 11 章 Spring 对 Hibernate 的支持

介绍 Hibernate 会话工厂和事务管理的配置。然后,我们将讨论一下如何编写有效的数据维护代码,特别关注 Hibernate 的数据访问方法的使用。接着,我们将讨论延迟加载和会话管理。最后将介绍一些集成测试的测试方法。

11.3 Hibernate 支持的介绍

Hibernate 的主类是 session 类,它提供查找保存和删除映射对象的方法。我们必须先建立 sessionFactory 对象,来得到一个 Hibernate 的 session 对象。创建和配置 SessionFactory 对象来构造 Session 可能相当复杂和笨重。幸运的是, Spring 使用它的 AbstractSessionFactoryBean 的实现子类—— LocalSessionFactoryBean 类和 AnnotationSessionFactoryBean 类提供帮助。LocalSessionFactoryBean 类需要定位映射文件,基于应用程序的考虑这些文件应该位于本地。在大多数情况下,这意味着映射资源在类路径上。第二个子类,即 AnnotationSessionFactoryBean 类使用类中的注解来完成 Hibernate 映射,

这也是我们希望在 Hibernate 中使用的。

LocalSessionFactoryBean 的最简化配置

```
<?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:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/jee
http://www.springframework.org/schema/jee/spring-jee-2.5.xsd">

    <jee:jndi-lookup id="dataSource"
jndi-name="java:comp/env/jdbc/prospring2/ch11"
        expected-type="javax.sql.DataSource" />

    <bean id="hibernateSessionFactory"

class="org.springframework.orm.hibernate3.LocalSessionFactoryBean"
">
        <property name="dataSource">
            <ref bean="dataSource" />
        </property>
        <property name="mappingLocations">
            <list>

                <value>classpath*:/com/apress/prospring2/ch11/dataaccess/hibernat
e/*.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.dialect">
                    org.hibernate.dialect.Oracle9Dialect
                </prop>
            </props>
        </property>
    </bean></beans>
```

11.3.1 使用 HibernateSession

```
package com.apress.prospring2.ch11.dataaccess;

import javax.naming.NamingException;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.mock.jndi.SimpleNamingContextBuilder;

public class DaoDemo {
    @SuppressWarnings("deprecation")
    public static void buildJndi() {
        try {
            SimpleNamingContextBuilder builder;
            builder =
SimpleNamingContextBuilder.emptyActivatedContextBuilder();
            String connectionString =
"jdbc:oracle:thin:@localhost:1521:XE";
            builder.bind("java:comp/env/jdbc/prospring2/ch11",
                new
DriverManagerDataSource("oracle.jdbc.driver.OracleDriver",connectionS
tring,

                "system","root"));
        } catch (NamingException ignored) {}
    }

    public static void main(String[] args) {
        buildJndi();
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "/datasource-context-minimal.xml", DaoDemo.class);
        SessionFactory sessionFactory =
            (SessionFactory)ac.getBean("hibernateSessionFactory");
        Session session = null;
        Transaction transaction = null;
        try {
            session = sessionFactory.openSession();
        }
```

```

        transaction = session.beginTransaction();
        //do work
        transaction.commit();
    } catch (Exception e) {
        if (transaction != null) transaction.rollback();
    } finally {
        if (session != null) session.close();
    }
}
}

```

HibernateTemplate 类的使用方法

```

package com.apress.prospring2.ch11.dataaccess;

import java.sql.SQLException;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.orm.hibernate3.HibernateCallback;
import org.springframework.orm.hibernate3.HibernateTemplate;

public class HibernateTemplateDemo {
    public static void main(String[] args) {
        DaoDemo.buildJndi();
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "/datasource-context-minimal.xml", DaoDemo.class);
        SessionFactory sessionFactory =
            (SessionFactory) ac.getBean("hibernateSessionFactory");
        HibernateTemplate template = new
        HibernateTemplate(sessionFactory);
        template.execute(new HibernateCallback() {
            @Override
            public Object doInHibernate(Session session)
                throws HibernateException, SQLException {
                System.out.println("Execute in doInHibernate");
                return null;
            }
        });
    }
}

```

上面的代码清单说明了大多数我们讨论的要点。使用 `SessionFactory` 实例参数创建

`HibernateTemplate` 类。下一步，我们调用它的 `execute` 方法，提供一个 `HibernateCallback` 匿名实现，处理所有的数据库工作。我们在回调中得到 `session` 并作为 `doInHibernate` 方法的参数。`session` 参数不能是 `null`，因此我们不必担心关闭 `session` 或处理实现中的异常。除此之外，提供给 `doInHibernate` 方法的回调代码可以参与 Spring 事务管理。我们从 `doInHibernate` 方法返回的值简单地委托给调用者，作为 `execute` 方法的执行结果。

在大多数应用程序中，我们设置 `HibernateTemplate` 类作为一个依赖。因为所有的 `HibernateTemplate` 类的方法都是线程安全的，所以我们可以这样做。并且我们在很多 DAO bean 之中分享 `HibernateTemplate` 的单个实例。

如下

配置文件中增加

```
<bean id="hibernateTemplate"

    class="org.springframework.orm.hibernate3.HibernateTemplate">
        <property name="sessionFactory"
ref="hibernateSessionFactory" />
    </bean>
```

测试代码

```
public static void main(String[] args) {
    DaoDemo.buildJndi();
    ApplicationContext ac = new ClassPathXmlApplicationContext(
        "/datasource-context-ht.xml", DaoDemo.class);

    HibernateTemplate template =
(HibernateTemplate) ac.getBean("hibernateTemplate");
    template.execute(new HibernateCallback() {
        @Override
        public Object doInHibernate(Session session)
            throws HibernateException, SQLException {
            System.out.println("Execute in doInHibernate");
            return null;
        }
    });
}
```

11.3.2 使用 HibernateDaoSupport 类

配置文件

```
<?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:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/jee
http://www.springframework.org/schema/jee/spring-jee-2.5.xsd">

<jee:jndi-lookup id="dataSource"
jndi-name="java:comp/env/jdbc/prospring2/ch11"
expected-type="javax.sql.DataSource" />

<bean id="hibernateSessionFactory"

class="org.springframework.orm.hibernate3.LocalSessionFactoryBean"
">
    <property name="dataSource">
        <ref bean="dataSource" />
    </property>
    <property name="mappingLocations">
        <list>

            <value>classpath*:/com/apress/prospring2/ch11/dataaccess/hibernat
e/*.*.hbm.xml</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">
                org.hibernate.dialect.Oracle9Dialect
            </prop>
        </props>
    </property>
</bean>

<bean id="hibernateDaoSupport" abstract="true"

class="org.springframework.orm.hibernate3.support.HibernateDaoSup
port">
    <property name="sessionFactory"
ref="hibernateSessionFactory"/>
</bean>
<bean id="logEntryDao"

class="com.apress.prospring2.ch11.daoimpl.HibernateLogEntryDao"
parent="hibernateDaoSupport" />
</beans>
```

接口与实现类

```
package com.apress.prospring2.ch11.dao;

import java.util.List;

import com.apress.prospring2.ch11.domain.LogEntry;

public interface LogEntryDao {
    void save(LogEntry logEntry);
    List<LogEntry> getAll();
}

package com.apress.prospring2.ch11.daoimpl;

import java.util.List;

import org.springframework.orm.hibernate3.support.HibernateDaoSupport;

import com.apress.prospring2.ch11.dao.LogEntryDao;
import com.apress.prospring2.ch11.domain.LogEntry;

public class HibernateLogEntryDao extends
    HibernateDaoSupport implements LogEntryDao{

    @Override
    @SuppressWarnings({"unchecked"})
    public List<LogEntry> getAll() {
        // TODO Auto-generated method stub
        return getHibernateTemplate().find("from LogEntry");
    }

    @Override
    public void save(LogEntry logEntry) {
        // TODO Auto-generated method stub
        getHibernateTemplate().saveOrUpdate(logEntry);
    }
}
```

11.3.3 HibernateTemplate 和 Session 之间的选择

回忆起 JdbcTemplate 的工作分为 2 部分：它提供普通的资源管理并将 JDBC 检查异常转

换为 Spring 数据访问异常。Hibernate 3 使用运行时异常，Spring [2.5 的@Repository](#) 注解让资源管理与在 HibernateTemplate 所使用的注解非常相似。因此，我们可以说，大多数情况使用 HibernateTemplate 是完全没有必要的

看下面的代码，只是将 SessionFactory 作为其成员，在 Spring 配置文件中对其属性进行注入即可

```
package com.apress.prospring2.ch11.daoimpl;

import java.util.List;

import org.hibernate.SessionFactory;
import org.hibernate.Transaction;

import com.apress.prospring2.ch11.dao.LogEntryDao;
import com.apress.prospring2.ch11.domain.LogEntry;

public class TemplatelessHibernateInvoiceLogEntryDao implements LogEntryDao {
    private SessionFactory sessionFactory;

    @Override
    public List<LogEntry> getAll() {
        // TODO Auto-generated method stub
        Transaction transaction = this.sessionFactory.getCurrentSession().beginTransaction();
        try{
            return this.sessionFactory.getCurrentSession().createCriteria("from
LogEntry").list();
        }finally{
            transaction.commit();
        }
    }

    @Override
    public void save(LogEntry logEntry) {
        // TODO Auto-generated method stub
        Transaction transaction = this.sessionFactory.getCurrentSession().beginTransaction();
        try{
            this.sessionFactory.getCurrentSession().saveOrUpdate(logEntry);
        }finally{
            transaction.commit();
        }
    }
}
```

```
public SessionFactory getSessionFactory() {
    return sessionFactory;
}

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

public LogEntry getByld(Long id){
    Transaction transaction = this.sessionFactory.getCurrentSession().beginTransaction();

    try{
        return (LogEntry) this.sessionFactory.getCurrentSession().get(LogEntry.class, id);
    }finally{
        transaction.commit();
    }
}

}package com.apress.prospring2.ch11.daoimpl;

import java.util.List;

import org.hibernate.SessionFactory;
import org.hibernate.Transaction;

import com.apress.prospring2.ch11.dao.LogEntryDao;
import com.apress.prospring2.ch11.domain.LogEntry;

public class TemplatelessHibernateInvoiceLogEntryDao implements LogEntryDao {
    private SessionFactory sessionFactory;

    @Override
    public List<LogEntry> getAll() {
        // TODO Auto-generated method stub
        Transaction transaction = this.sessionFactory.getCurrentSession().beginTransaction();
        try{
            return this.sessionFactory.getCurrentSession().createCriteria("from
LogEntry").list();
        }finally{
            transaction.commit();
        }
    }

    @Override
```



```

public void save(LogEntry logEntry) {
    // TODO Auto-generated method stub
    Transaction transaction = this.sessionFactory.getCurrentSession().beginTransaction();
    try{
        this.sessionFactory.getCurrentSession().saveOrUpdate(logEntry);
    }finally{
        transaction.commit();
    }

}

public SessionFactory getSessionFactory() {
    return sessionFactory;
}

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

public LogEntry getById(Long id){
    Transaction transaction = this.sessionFactory.getCurrentSession().beginTransaction();

    try{
        return (LogEntry) this.sessionFactory.getCurrentSession().get(LogEntry.class, id);
    }finally{
        transaction.commit();
    }
}
}

```

注意其中的 transaction.beginTransaction()不能少，要不然会报错 creatQuery is not valid without active transaction

配置文件如果不修改的话会抛出

Exception in thread "main" org.hibernate.HibernateException

No Hibernate Session bound to thread, and configuration does not allow creation of non-transactional one here.

很容易理解：SessionFactory.getCurrentSession 方法将返回 session 给当前线程，但是没有那个 Session，SessionFactory 配置也没有包含如何创建一个会话的信息。我们可以告诉 Hibernate 如何通过修改 hibernateSessionFactory bean 定义来创建一个新的线程 session(事务管理也使用它)下面是修改的的配置文件 datasource-context-dao.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"

```

```
xmlns:jee="http://www.springframework.org/schema/jee"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/jee
http://www.springframework.org/schema/jee/spring-jee-2.5.xsd">

<jee:jndi-lookup id="dataSource"
jndi-name="java:comp/env/jdbc/prospring2/ch11"
expected-type="javax.sql.DataSource" />

<bean id="hibernateSessionFactory"

class="org.springframework.orm.hibernate3.LocalSessionFactoryBean"
">
    <property name="dataSource">
        <ref bean="dataSource" />
    </property>
    <property name="mappingLocations">
        <list>

            <value>classpath*:/com/apress/prospring2/ch11/dataaccess/hibernat
e/*.hbm.xml</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">
                org.hibernate.dialect.Oracle9Dialect
            </prop>
            <prop key="hibernate.current_session_context_class">
                thread
            </prop>
            <prop key="hibernate.transaction.factory_class">
                org.hibernate.transaction.JDBCTransactionFactory
            </prop>
        </props>
    </property>
</bean>

<bean id="hibernateDaoSupport" abstract="true"

class="org.springframework.orm.hibernate3.support.HibernateDaoSup
port">
```

```

        <property name="sessionFactory"
ref="hibernateSessionFactory"/>
    </bean>
    <bean id="logEntryDao"

        class="com.apress.prospring2.ch11.daoimpl.HibernateLogEntryDao"
        parent="hibernateDaoSupport" />
    <bean id="templatelessLogEntryDao"

        class="com.apress.prospring2.ch11.daoimpl.TemplatelessHibernateIn
voiceLogEntryDao">
        <property name="sessionFactory"
ref="hibernateSessionFactory"/>
    </bean>
</beans>

```

我们应该对异常进行转换，增加一个 bean post processor，它命令框架解释所有使用 @Repository 注解的 bean 的调用，执行标准的异常转换。转换为 Spring 数据访问异常。

```

<bean
class="org.springframework.dao.annotation.PersistenceExceptionTransal
ationPostProcessor"/>

```

11.4 在企业级应用中使用 Hibernate

前一小节的样例代码是良好的 Hibernate 功能性代码。实际上，完全可以在任何企业级应用程序中使用我们已经编写过的代码。但是它们的工作效率并不会像我们所希望的那样高。

遇到的第一个问题就是无法阻止更新脏数据。我们没有对更新数据库中行数据的代码进行任何形式的检查。

两者，我们也没有考虑事务性行为。若不添加新的代码，doInHibernate 方法的回调代码不会自动在事务中运行。LogEntry 当然需要支持通常意义上的事务，并符合 ACID 规则(原子性，一致性，独立性和持久性)。同时，企业级应用程序在事务中通常包含其它资源(例如 JMS 队列)。

接下来我们的实例工作于 LogEntry 对象。LogEntry 类没有任何关系，它使用来自单个数据库的行构成。通常企业级程序需维护包含很多关联的复杂对象。高效处理这些关联很重要，否则，我们将不得不使用极其复杂的 SQL 语句。

最后如果 t_log_entry 表包含成百上千的行，我们给出的示例就会崩溃，并抛出 java.lang.OutOfMemoryException 异常。现实世界的程序需要高效和优雅地处理超大数据集。

11.4.1 阻止更新脏数据

.....有几种策略阻止这种情况：线程 A 可以为整行加锁，从而使得只有在线程 A 完成更新后线程 B 才可以读取。此方法叫做悲观锁，我们问题假设问题会发生，因此我们将整行数据锁定，以保证该数据的排他访问。但是它是一个严重的性能瓶颈。如果应用程序正在完成很多

更新操作，我们可能将大量的表锁住，系统可能将一直等待直至行数据的锁被释放。

如果执行的读操作比写操作多，最行将行释放，使用乐观锁。我们不会为更新数据显式给行加锁，因为我们假设冲突将不会发生。为了识别脏数据，我们给表和域对象增加 **version** 列。当我们保存数据时，我们松果数据库中的版本是否与对象中的版本保持一致。如果确实一致，那么其他线程不能修改我们可以处理并将更新的行数据。如果域对象中的版本情形数据库行中的版本不一致我们将抛出一个异常提示我们可能保存了脏数据。在 **Hibernate** 中的乐观锁很容易使用。

乐观锁定的 **LogEntry** 对象

.....

```
private Long id;
private String name;
private Date date;
private Long version;
```

.....

LogEntry.hbm.xml 注意 **version** 而不是 **property**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate//Hibernate Mapping DTD
3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping default-lazy="false"
package="com.apress.prospring2.ch11.domain">
  <class name="LogEntry" table="T_LOG_ENTRY">
    <id name="id" type="long" unsaved-value="null">
      <generator class="sequence">
        <param name="sequence">s_log_entry_id</param>
      </generator>
    </id>
    <version name="version" column="version" type="long"
unsaved-value="null"/>
    <property name="name" column="name_" not-null="true"/>
    <property name="date" column="date_" not-null="true" />
  </class>
</hibernate-mapping>
```

测试代码

```
public static void main(String[] args) {
  ApplicationContext ac = new ClassPathXmlApplicationContext(
    "/datasource-context-dao.xml");
  LogEntryDao logEntryDao =
    (LogEntryDao)ac.getBean("logEntryDao");
  //save the original entry
  LogEntry le = new LogEntry();
  le.setName("Name");
```

```

le.setDate(Calendar.getInstance().getTime());
logEntryDao.save(le);

//load two instances of the same LogEntry object
LogEntry le1 = logEntryDao.getById(le.getId());
LogEntry le2 = logEntryDao.getById(le.getId());
//modify and save le1
le1.setName("X");
logEntryDao.save(le1);
//now, let's try to modify and save le2
//remember, le2 represents the same row as le1
le2.setName("Z");
logEntryDao.save(le2);
}

```

这时由于采用了乐观锁，所以会抛出异常。`org.hibernate.StaleObjectStateException`

11.4.3 对象等价性

既然我们可以安全的阻止更新脏数据，那么也需要考虑另一个重要限制。当我们在 Hibernate 中持久化集合时，很可能使用一些集合类。当为一对多关系建立模型时，我们最可能使用的是 Set。一个 Set 是一个集合，它不能保证元素的顺序，也不允许元素重复，它精确地表示了数据库中的一对多关系，我们将域对象增加到 Set 中，因此需要考虑它们的等价性。我们可以实现自然等价或者数据库等价。自然等价意味着，若二个对象从应用逻辑角度包含相同的数据，则它们是相等的。

数据库不关心它是否将二行的所有字段都设置为相等，而是只考虑行的主键是唯一的并且不破坏任何其它约束。如果需要强化自然等价，我们可以考虑使用唯一索引。基于此考虑，我们可以看一看 LogEntry 类的 equals 方法和 hashCode 方法的实现。

```

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((id == null) ? 0 : id.hashCode());
    result = prime * result + ((version == null) ? 0 :
version.hashCode());
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())

```

```

        return false;
    }
    LogEntry other = (LogEntry) obj;
    if (id == null) {
        if (other.id != null)
            return false;
    } else if (!id.equals(other.id))
        return false;
    if (version == null) {
        if (other.version != null)
            return false;
    } else if (!version.equals(other.version))
        return false;
    return true;
}

```

基于上述思路我们实现了数据库等价：如果两个对象拥有相同的 id 和 version，那么它们是相等的。还有一个小问题：除 LogEntry 类外我们的域对象很可能还包含很多其它类，因此我们将重构代码，将通用 equals 和 hashCode 方法移到一个新类，而由 LogEntry 类扩展这个新创建的类。

```
package com.apress.prospring2.ch11.domain;
```

```
import java.io.Serializable;
```

```

public abstract class AbstractIdentityVersionObject<T> implements
Serializable {
    protected Long version;
    protected T id;

    public AbstractIdentityVersionObject() {
    }

    public AbstractIdentityVersionObject(T id) {
        this.id = id;
    }

    protected final boolean idEquals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        AbstractIdentityVersionObject other =
            (AbstractIdentityVersionObject) obj;
    }
}

```

```
        if (id == null) {
            if (other.id != null)
                return false;
        } else if (!id.equals(other.id))
            return false;
        if (version == null) {
            if (other.version != null)
                return false;
        } else if (!version.equals(other.version))
            return false;
        return true;
    }

    protected final int idHashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((id == null) ? 0 : id.hashCode());
        result = prime * result + ((version == null) ? 0 :
version.hashCode());
        return result;
    }

    @Override
    public int hashCode() {
        return idHashCode();
    }

    @Override
    public boolean equals(final Object obj) {
        return idEquals(obj);
    }

    public Long getVersion() {
        return version;
    }

    public void setVersion(final Long version) {
        this.version = version;
    }

    public T getId() {
        return id;
    }

    public void setId(final T id) {
        this.id = id;
    }
}
```

```
}

}

LogEntry 类
package com.apress.prospring2.ch11.domain;

import java.util.Date;

public class LogEntry extends AbstractIdentityVersionObject<Long> {
    private String name;
    private Date date;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Date getDate() {
        return date;
    }
    public void setDate(Date date) {
        this.date = date;
    }
}

}
```

`AbstractIdentityVersionedObject` 类包含 `id` 和 `version` 属性。(id 是一个泛型，允许我们使用任何类型的主键值，甚至使用一个复合类型！)它也实现了数据库等价性。`LogEntry` 对象简单扩展了 `AbstractIdentityVersionedObject(Long)`，并只增加了它声明的列。这让域对象更具可读性，而不必担心会在新域对象中忘记实现 `equals` 和 `hashCode` 方法。这对于希望在数据中持久化的对象需要是非常有用的。它假设一个 `null` 值代表 现代战争不可进行插入操作的对象。另外，它基于不显式修改 `id` 和 `version` 的属性值假设。

上述讨论为延迟加载的讨论构造了一个很好的起点，但是我们开始进入该讨论之前，需要看一看另一个关键的企业级需求。

11.4.3 事务支持

我们将讨论的下一个领域是 `Hibernate` 对事务行为的支持。在 `Spring` 事务支持中使用 `PlatformTransactionManager` 接口，为了能够与 `Hibernate` 一起使用，我们需要一个 `HibernateTransactionManager` 实现。该 bean 需要有一个 `SessionFactory` 引用。

`datasource-context-dao.xml`(新工程)

```
<?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:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/jee
        http://www.springframework.org/schema/jee/spring-jee-2.5.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-2.5.
        xsd">

    <jee:jndi-lookup id="dataSource"
jndi-name="java:comp/env/jdbc/prospring2/ch11"
        expected-type="javax.sql.DataSource" />

    <bean id="hibernateSessionFactory"

class="org.springframework.orm.hibernate3.LocalSessionFactoryBean
">
        <property name="dataSource">
            <ref bean="dataSource" />
        </property>
        <property name="mappingLocations">
            <list>
                <value>classpath*:LogEntry.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.dialect">
                    org.hibernate.dialect.Oracle9Dialect
                </prop>
                <prop key="hibernate.show_sql">true</prop>
                <prop key="hibernate.format_sql">true</prop>
                <prop key="hibernate.current_session_context_class">
                    thread
                </prop>
                <prop key="hibernate.transaction.factory_class">
```

```

        org.hibernate.transaction.JDBCTransactionFactory
    </prop>
</props>
</property>
</bean>

<bean id="hibernateDaoSupport" abstract="true"

    class="org.springframework.orm.hibernate3.support.HibernateDaoSupport">
        <property name="sessionFactory"
ref="hibernateSessionFactory"/>
    </bean>
    <bean id="logEntryDao"

        class="com.apress.prospring2.ch11.dataaccess.hibernate.HibernateLogEntryDao"
        parent="hibernateDaoSupport" />

    <bean id="transactionManager"

        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
            <property name="sessionFactory"
ref="hibernateSessionFactory" />
        </bean>
    </beans>

```

现在我们有 `transactionManager` bean, 可以使用 Spring 事务框架控制 Hibernate 事务。但是, 仅仅声明 `transactionManager` bean 并不意味着我们拥有了事务行为, 正如下面的代码所示

```

package com.apress.prospring2.ch11.dataaccess.demo;

import java.util.Calendar;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.apress.prospring2.ch11.dataaccess.dao.LogEntryDao;
import com.apress.prospring2.ch11.domain.LogEntry;
import com.apress.prospring2.ch11.utils.JNDIUtils;

public class HibernateLogEntryDaoTx1Demo {

```

```
private static LogEntry save(LogEntryDao dao, String name) {
    LogEntry le = new LogEntry();
    le.setName(name);
    le.setDate(Calendar.getInstance().getTime());
    dao.save(le);
    return le;
}

public static void main(String[] args) {
    JNDIUtils.buildJndi();
    ApplicationContext ac = new ClassPathXmlApplicationContext(
        "/datasource-context-dao.xml");
    LogEntryDao dao = (LogEntryDao)ac.getBean("logEntryDao");
    try{
        save(dao, "Hello, this works");
        save(dao, null);
    } catch (Exception e) {
        e.printStackTrace();
        //we don't want to do anything here but alias
        System.out.println(dao.getAll());
    }
}
```

save 方法的第一个 Hibernate 成功了，但第二个操作却失败了。因为我们没有定义事务的边界，所以我们将得到异常捕获的一行消息提示。我们需要重新考虑事务策略，可以让 hibernateLogEntryDao(LogEntry) 方法具有事务性，但是这会带来任何好处。实际上使单个 DAO 方法具有事务性不是一个好的实践，因为它将在服务层之上定义事务的边界。DAO 层应该是一个域对象和一些数据存储方式间的简单转换器。

让我们创建一个简单的服务接口及其实现。服务实现将使用日志，我们将保证服务方法是事务性的。图 11-2 展示了我们将创建的服务的类图。

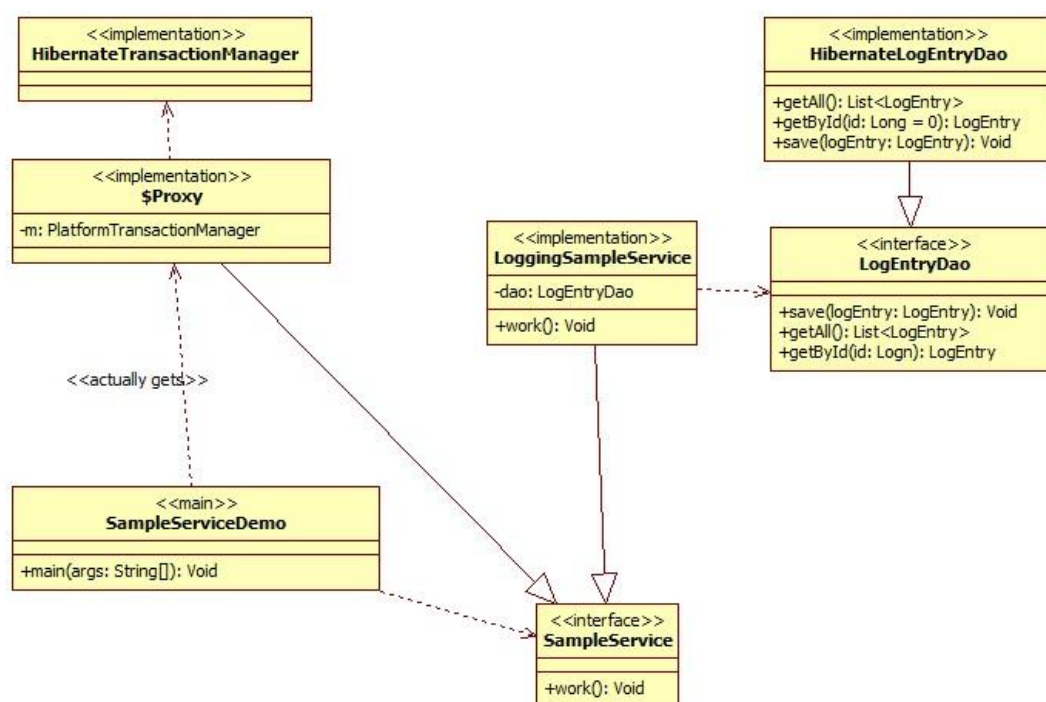


图 11-2 服务的 UML 类图

上图说明 `SampleServiceDemo` 示例的代码并没有使用 `SampleService` 接口的 `LoggingSampleService` 实现，而是使用一个动态生成的代理类。(启示：莫非，所谓的 `transaction` 也只是用代理来实现，即将数据访问代码包围，添加自己的代码如异常捕获等)。该代理类使用 `HibernateTransactionManager` 维护 `work()` 方法的事务行为。为了保持完整性，类图也表示出了 `LogEntryDao` 接口及其 `HibernateLogEntryDao` 实现。对于 AOP 和事务的更深入讨论，请查阅第 5、6、16 章。

接口 `SampleService`

```
package com.apress.prospring2.ch11.dataaccess.service;
```

```
public interface SampleService {
    void work();
}
```

接口实现类 `LoggingSampleService`

```
package com.apress.prospring2.ch11.dataaccess.serviceimpl;
```

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

```
import com.apress.prospring2.ch11.dataaccess.dao.LogEntryDao;
import com.apress.prospring2.ch11.dataaccess.service.SampleService;
import com.apress.prospring2.ch11.domain.LogEntry;
```

```
public class LoggingSampleService implements SampleService {
    private LogEntryDao logEntryDao;
```

```
private void log(String message){
    LogEntry entry = new LogEntry();
    entry.setDate(Calendar.getInstance().getTime());
    entry.setName(message);
    this.logEntryDao.save(entry);
}

@Override
public void work() {
    log("Begin.");
    log("Processing...");

    if(System.currentTimeMillis() % 2 == 0) log(null);
    log("Done");
}

public void setLogEntryDao(LogEntryDao logEntryDao) {
    this.logEntryDao = logEntryDao;
}
}
```

新增配置文件 **datasource-context-tx.xml** 主要添加事务控制项与 aop 项

```
<?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:tx="http://www.springframework.org/schema/tx"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/jee
        http://www.springframework.org/schema/jee/spring-jee-2.5.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-2.5.
        xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

    <tx:advice id="txAdvice" transaction-manager="transactionManager">
        <tx:attributes>
```

```

        <tx:method name="work"/>
    </tx:attributes>
</tx:advice>

<aop:config>
    <aop:pointcut expression="execution(*
com.apress.prospring2.ch11.dataaccess.service.SampleService.*(..))"
id="sampleServiceOperation"/>
    <aop:advisor advice-ref="txAdvice"
pointcut-ref="sampleServiceOperation"/>
</aop:config>

<bean id="sampleService"

    class="com.apress.prospring2.ch11.dataaccess.serviceimpl.LoggingsS
ampleService">
    <property name="logEntryDao" ref="logEntryDao" />
</bean>
</beans>

```

demo 源程序

```
package com.apress.prospring2.ch11.dataaccess.demo;
```

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
```

```
import com.apress.prospring2.ch11.dataaccess.dao.LogEntryDao;
import com.apress.prospring2.ch11.dataaccess.service.SampleService;
import com.apress.prospring2.ch11.utils.JNDIUtils;
```

```
public class SampleServiceDemo {
```

```

    public static void main(String[] args) {
        JNDIUtils.buildJndi();
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            new String[]{
                "classpath*:./datasource-context-dao.xml",
                "classpath*:./datasource-context-tx.xml"
            });
        SampleService sampleService =
        (SampleService)ac.getBean("sampleService");
        LogEntryDao logEntryDao = (LogEntryDao)ac.getBean("logEntryDao");
        int successCount = 0;
        int failureCount = 0;
    }
}

```

```
int before = logEntryDao.getAll().size();
for(int i = 0; i < 10; i++){
    if(tryWork(sampleService))
        successCount++;
    else
        failureCount++;
}
System.out.println("Inserted " + (logEntryDao.getAll().size() - before) +
    ", for " + successCount + " successes and " +
    failureCount + "failures");
}

private static boolean tryWork(SampleService sampleService){
    try{
        sampleService.work();
        return true;
    }catch(Exception e){
        //do nothing (BAD in production
        System.err.println(e.getMessage());
    }
    return false;
}
}
```

执行演示输出结果为:

```
Inserted 15, for 5 successes and 5failures
```

与书本一样，演示成功。

11.4.4 延迟加载

本节与上一节同等重要，实际上两个领域关系很紧密。延迟加载的原理很简单：只在需要时抓取数据。尽管这意味着有更多的路径访问数据库，但是由于 Hibernate 只抓取应用程序需要的数据所以以执行效率会得到改善。延迟加载一般应用于有关系的集合，例如带有 InvoiceLine 对象集合的 Invoice 对象，每一个 InvoiceLine 将拥有一个 Discount 对象集合。

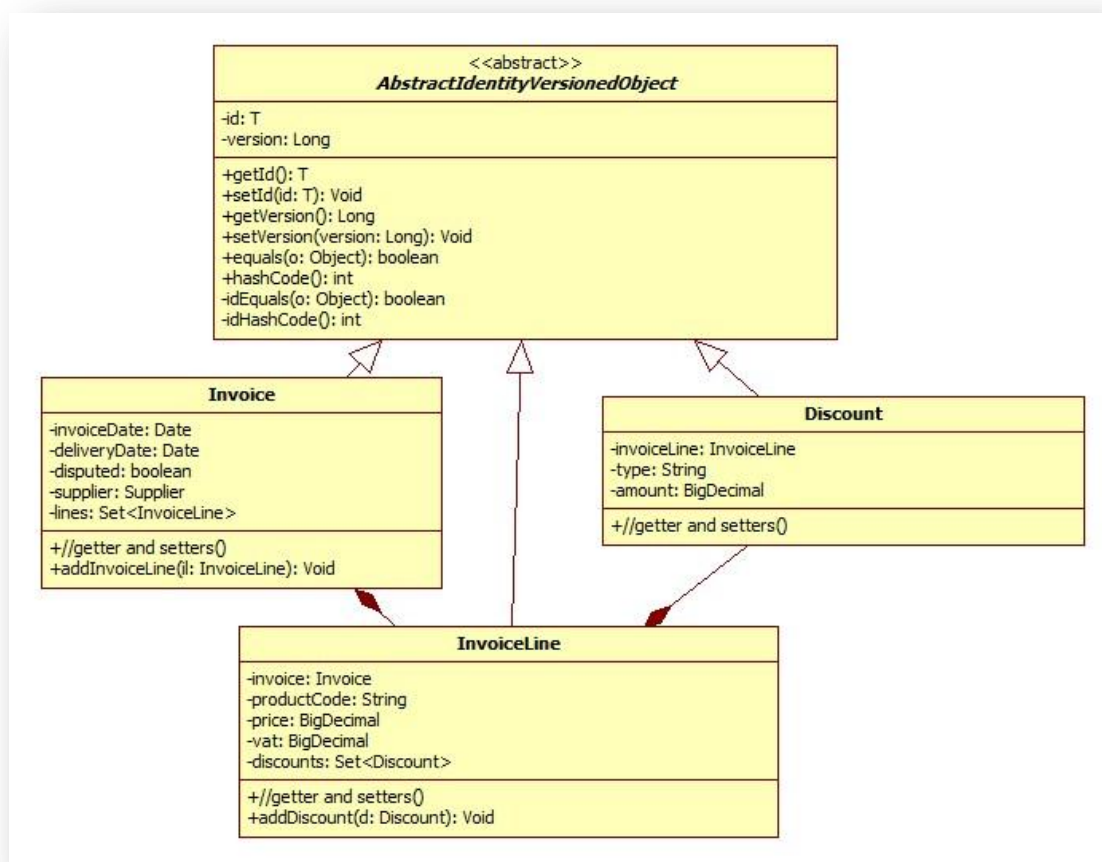
Invoice中

```
public void addInvoiceLine(InvoiceLine invoiceLine){
    invoiceLine.setInvoice(this);
    this.invoiceLines.add(invoiceLine);
}
```

InvoiceLine 中

```
public void addDiscount(Discount discount){
    discount.setInvoiceLine(this);
    this.discounts.add(discount);
}
```

}



代码清晰的说明了 `addInvoiceLine()` 方法和 `InvoiceLine.addDiscount()` 方法在增加的对象和它的容器间建立了双向关联。这对于 **Hibernate** 很重要，它让我们的代码更清晰。

建表语句

```

create table t_supplier(
    id number(19,0) not null,
    version number(19,0) null,
    name varchar2(200) not null,
    constraint pk_supplier primary key (id)
);

create sequence s_supplier_id start with 10000;

create table t_invoice(
    id number(19,0) not null,
    version number(19,0) null,
    invoice_date date not null,
    delivery_date date not null,
    supplier number(19,0) not null,

```



```
constraint pk_invoice primary key (id),
constraint fk_i_supplier foreign key (supplier) references
t_supplier(id)
);
create sequence s_invoice_id start with 10000;

create table t_invoice_line (
  id number(19,0) not null,
  version number(19,0) null,
  invoice number(19,0) not null,
  price number(20,4) not null,
  vat number(20,4) not null,
  product_code varchar2(50) not null,
  constraint pk_invoice_line primary key (id),
  constraint fk_il_invoice foreign key (invoice) references
t_invoice(id)
);
create sequence s_invoice_line_id start with 10000;

create table t_discount (
  id number(19,0) not null,
  version number(19,0) null,
  invoice_line number(19,0) not null,
  type_ varchar2(50) not null,
  amount number(20,4) not null,
  constraint pk_discount primary key (id),
  constraint fk_d_invoice_line foreign key (invoice_line) references
t_invoice_line(id)
);
create sequence s_discount_id start with 10000;
```

Hibernate 映射文件<4 个>

Supplier.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="com.apress.prospring2.ch11.domain"
  default-lazy="true">
  <class name="Supplier" table="t_supplier">
    <id name="id" type="long" unsaved-value="null">
      <generator class="sequence">
        <param name="sequence">s_supplier_id</param>
```

```
        </generator>
    </id>
    <version name="version" column="version" unsaved-value="null"
type="long"/>
    <property name="name" column="name" not-null="true"/>
</class>
</hibernate-mapping>
```

Invoice.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="com.apress.prospring2.ch11.domain"
    default-lazy="true">
    <class name="Invoice" table="t_invoice">
        <id name="id" type="long" unsaved-value="null">
            <generator class="sequence">
                <param name="sequence">s_invoice_id</param>
            </generator>
        </id>
        <version name="version" column="version" unsaved-value="null"
type="long"/>
        <property name="deliveryDate" column="delivery_date"
not-null="true"/>
        <property name="invoiceDate" column="invoice_date"
not-null="true"/>
        <many-to-one name="supplier" not-null="true" class="Supplier" />
        <set name="invoiceLines" cascade="all" inverse="true">
            <key column="invoice" not-null="true"/>
            <one-to-many class="InvoiceLine"/>
        </set>
    </class>
</hibernate-mapping>
```

InvoiceLine.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="com.apress.prospring2.ch11.domain"
    default-lazy="true">
```

```

<class name="InvoiceLine" table="t_invoice_line">
  <id name="id" type="long" unsaved-value="null">
    <generator class="sequence">
      <param name="sequence">s_invoice_line_id</param>
    </generator>
  </id>
  <version name="version" column="version" unsaved-value="null"
type="long"/>
  <property name="price" column="price" not-null="true"/>
  <property name="productCode" column="product_code"
not-null="true"/>
  <property name="vat" column="vat" not-null="true"/>
  <many-to-one name="invoice" class="Invoice" not-null="true"/>
  <set name="discounts" inverse="true" cascade="all">
    <key column="invoice_line" not-null="true"/>
    <one-to-many class="Discount"/>
  </set>
</class>
</hibernate-mapping>

```

Discount.hbm.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

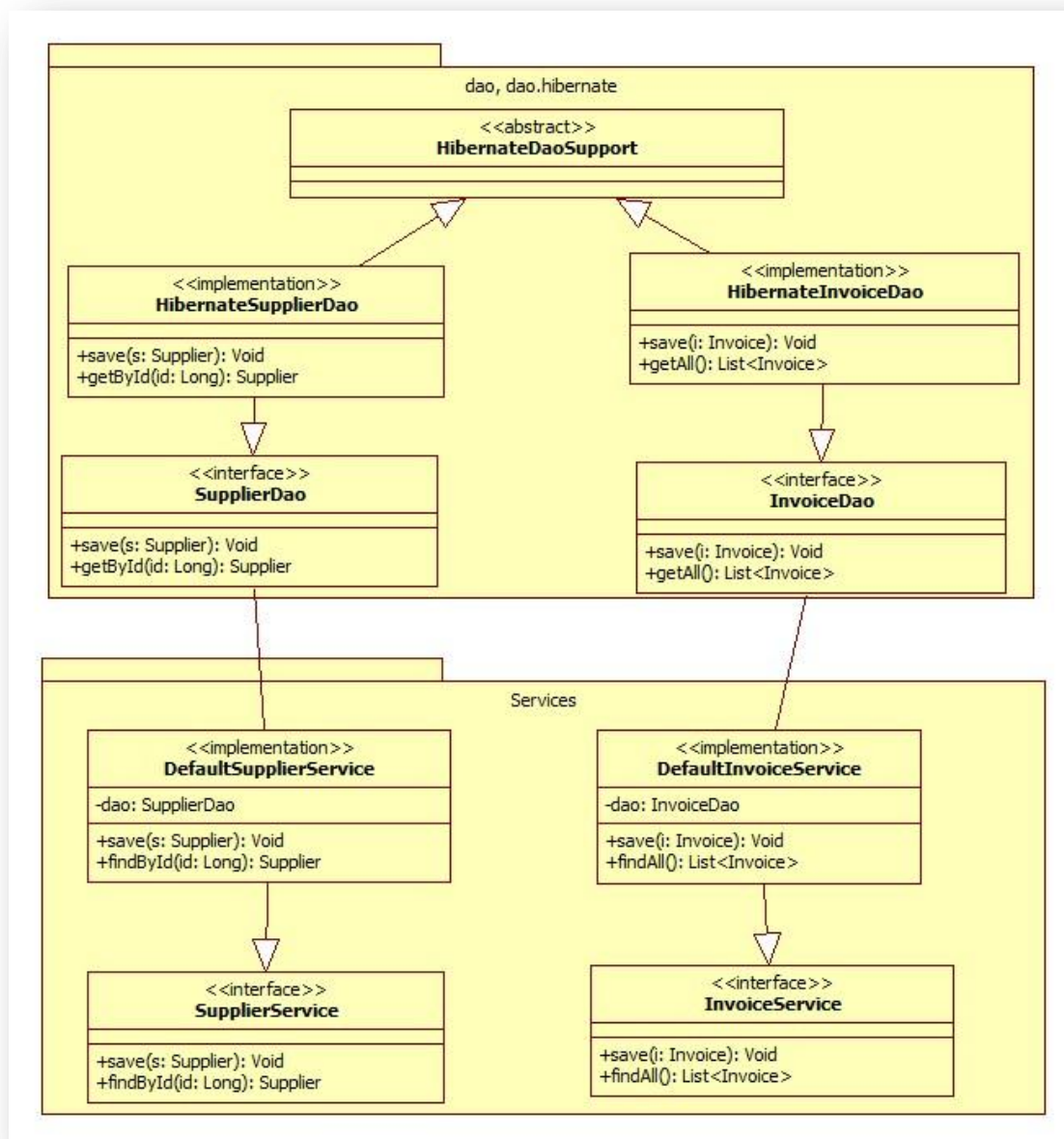
<hibernate-mapping package="com.apress.prospring2.ch11.domain"
  default-lazy="true">
  <class name="Discount" table="t_discount">
    <id name="id" type="long" unsaved-value="null">
      <generator class="sequence">
        <param name="sequence">s_discount_id</param>
      </generator>
    </id>
    <version name="version" column="version" unsaved-value="null"
type="long"/>
    <property name="amount" column="amount" not-null="true"/>
    <property name="type" column="type_" not-null="true"/>
    <many-to-one name="invoiceLine" column="invoice_line"
class="InvoiceLine" not-null="true"/>
  </class>
</hibernate-mapping>

```

说明：列表首先展示了最简单的映射：**Supplier** 对象没有引用其他任何对象。**Invoice** 对象的映射相当复杂：它展示了一个 **Supplier** 对象和行集合的多对多关联映射，行集合代表

InvoiceLine 对象的多对多关联映射。InvoiceLine 对象和 Discount 对象映射使用的模式与我们在 Invoice 对象映射中使用的是一样的。我们也需要关注 hibernate-mapping 元素的 default-lazy 属性的值为 true。

既然我们已经有设置依赖便利方法的域对象和它们的 Hibernate 映射，接下来我们需要创建合适的 DAO 类和服务。图 11-4 展示了我们将要编写的代码的 UML 包图。



这里我们只给出实现的代码(自己编写)
package com.apress.prospring2.ch11.dao;

```
import java.util.List;
```

```
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;
```

```
import com.apress.prospring2.ch11.domain.Invoice;
```

```
import com.apress.prospring2.ch11.interfaces.InvoiceDao;

public class HibernateInvoiceDao extends HibernateDaoSupport implements
    InvoiceDao {

    @SuppressWarnings("unchecked")
    @Override
    public List<Invoice> getAll() {
        // TODO Auto-generated method stub
        return getHibernateTemplate().find("from Invoice");
    }

    @Override
    public void save(Invoice invoice) {
        getHibernateTemplate().save(invoice);
    }

}

package com.apress.prospring2.ch11.dao;

import org.springframework.orm.hibernate3.support.HibernateDaoSupport;

import com.apress.prospring2.ch11.domain.Supplier;
import com.apress.prospring2.ch11.interfaces.SupplierDao;

public class HibernateSupplierDao extends HibernateDaoSupport implements
    SupplierDao {

    @Override
    public Supplier getByld(Long id) {
        // TODO Auto-generated method stub
        return (Supplier) getHibernateTemplate().get(Supplier.class, id);
    }

    @Override
    public void save(Supplier supplier) {
        // TODO Auto-generated method stub
        getHibernateTemplate().save(supplier);
    }

}
```

```
package com.apress.prospring2.ch11.service;

import java.util.List;

import com.apress.prospring2.ch11.domain.Invoice;
import com.apress.prospring2.ch11.interfaces.InvoiceDao;
import com.apress.prospring2.ch11.interfaces.InvoiceService;

public class DefaultInvoiceService implements InvoiceService {

    private InvoiceDao invoiceDao;

    @Override
    public List<Invoice> findAll() {
        // TODO Auto-generated method stub
        return invoiceDao.getAll();
    }

    @Override
    public void save(Invoice invoice) {
        // TODO Auto-generated method stub
        invoiceDao.save(invoice);
    }

    public void setInvoiceDao(InvoiceDao invoiceDao) {
        this.invoiceDao = invoiceDao;
    }
}

package com.apress.prospring2.ch11.service;

import com.apress.prospring2.ch11.domain.Supplier;
import com.apress.prospring2.ch11.interfaces.SupplierDao;
import com.apress.prospring2.ch11.interfaces.SupplierService;

public class DefaultSupplierService implements SupplierService {

    private SupplierDao supplierDao;

    @Override
    public Supplier findById(Long id) {
        // TODO Auto-generated method stub
        return supplierDao.findById(id);
    }
}
```

```
@Override
public void save(Supplier supplier) {
    // TODO Auto-generated method stub
    supplierDao.save(supplier);
}

/**
 * 用 Spring 注入其属性
 */
public void setSupplierDao(SupplierDao supplierDao) {
    this.supplierDao = supplierDao;
}
}
```

示例程序

```
package com.apress.prospring2.ch11.demo;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.apress.prospring2.ch11.interfaces.InvoiceService;
import com.apress.prospring2.ch11.interfaces.SupplierService;
import com.apress.prospring2.ch11.utils.JNDIUtils;

public class InvoiceServiceDemo {
    private SupplierService supplierService;
    private InvoiceService invoiceService;
    private void run() throws Exception{
        JNDIUtils.buildJndi();
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "/datasource-context-tx.xml");
        this.invoiceService = (InvoiceService)ac.getBean("invoiceService");
        this.supplierService = (SupplierService)ac.getBean("supplierService");
        findAllInvoices();
    }
    private void findAllInvoices(){}

    public static void main(String[] args) {
        try {
            new InvoiceServiceDemo().run();
        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

```

    }
}
}

```

示例程序仅验证 **Spring** 配置是否正确，它并不做任何数据库处理。

下面给出配置文件 **datasource-context-tx.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:tx="http://www.springframework.org/schema/tx"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/jee
        http://www.springframework.org/schema/jee/spring-jee-2.5.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-2.5.
        xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

    <jee:jndi-lookup id="dataSource"
        jndi-name="java:comp/env/jdbc/prospring2/ch11"
        expected-type="javax.sql.DataSource" />

    <bean id="hibernateSessionFactory"

        class="org.springframework.orm.hibernate3.LocalSessionFactoryBean"
    >
        <property name="dataSource">
            <ref bean="dataSource" />
        </property>
        <property name="mappingLocations">
            <!-- 本来用通配符表示这些文件，即classpath*:/*.hbm.xml
                发现了一个错误，说是Caused by:
                java.lang.StringIndexOutOfBoundsException: String index out of range: 0
                其实真正的原因是由于，我也定义了一个模板文件 即template.hbm.xml
                这之中没有填写正确的映射信息而发生的错误，删除或者按下面的
                写一个一个写出来就行了
            -->

```



```
-->
    <list>
        <value>classpath*:Supplier.hbm.xml</value>
        <value>classpath*:Invoice.hbm.xml</value>
        <value>classpath*:InvoiceLine.hbm.xml</value>
        <value>classpath*:Discount.hbm.xml</value>
    </list>
</property>
<property name="hibernateProperties">
    <props>
        <prop key="hibernate.dialect">
            org.hibernate.dialect.Oracle9Dialect
        </prop>
        <prop key="hibernate.show_sql">true</prop>
        <prop key="hibernate.format_sql">true</prop>
        <prop key="hibernate.current_session_context_class">
            thread
        </prop>
        <prop key="hibernate.transaction.factory_class">
            org.hibernate.transaction.JDBCTransactionFactory
        </prop>
    </props>
</property>
</bean>

<bean id="transactionManager"

    class="org.springframework.orm.hibernate3.HibernateTransactionMan
ager">
    <property name="sessionFactory"
ref="hibernateSessionFactory" />
</bean>
    <bean id="hibernateDaoSupport" abstract="true"

        class="org.springframework.orm.hibernate3.support.HibernateDaoSup
port">
        <property name="sessionFactory"
ref="hibernateSessionFactory"/>
    </bean>
    <bean id="invoiceDao"
        class="com.apress.prospring2.ch11.dao.HibernateInvoiceDao"
        parent="hibernateDaoSupport" />
    <bean id="supplierDao"
```

```

    class="com.apress.prospring2.ch11.dao.HibernateSupplierDao"
parent="hibernateDaoSupport"/>
    <bean id="invoiceService"
class="com.apress.prospring2.ch11.service.DefaultInvoiceService">
        <property name="invoiceDao" ref="invoiceDao"/>
    </bean>
    <bean id="supplierService"
class="com.apress.prospring2.ch11.service.DefaultSupplierService">
        <property name="supplierDao" ref="supplierDao"/>
    </bean>

</beans>

```

我们将使用 `findAllInvoices()` 方法来探寻 Hibernate 在程序背后所进行的工作。我们先创建一些测试数据来运行代码清单中的代码。

```

create or replace procedure "SYSTEM"."CREATE_SAMPLE_DATA"
is
    i number;
    j number;
    l number;
begin
    dbms_output.put_line('begin');
    for i in 1 .. 50 loop
        insert into t_supplier (id,version,name) values (
            s_supplier_id.nextval,1,'Supplier '||i);
        for j in 1 .. 100 loop
            insert into t_invoice
(id,version,invoice_date,delivery_date,supplier)
                values
(s_invoice_id.nextval,1,sysdate,sysdate,s_supplier_id.currval);
            for l in 1 .. 5 loop
                insert into t_invoice_line
(id,version,invoice,price,vat,product_code)

                values(s_invoice_line_id.nextval,1,s_invoice_id.currval,

                dbms_random.value(1,1000),dbms_random.value(1,100),'Product '||l);
            end loop;
        end loop;
    end loop;
end;

```

#执行存储过程

```
call CREATE_SAMPLE_DATA();
```

代码只是简单的插入了 50 个供应商数据，每一个供应商有 100 张发票，而每一张发票有 5 行数据。这为我们的实验创建了一个示例数据集。让我们修改 findAllInvoices()方法来访问数据库，并返回所有的 Invoice 对象。

修改后的 findAllInvoices()

```
private void findAllInvoices() {
    List<Invoice> invoices = invoiceService.findAll();
    System.out.println("发票List.size() = "+invoices.size());
    //执行成功 size为5000
}
```

我们期望代码打印 5000 行信息，并且运行时确实打印了 5000 行。这可能是计算 t_invoice 表行数的最无效率的方法。然而，好处就是我们不必装载 25000 个 InvoiceLine 对象！Hibernate 只在 t_invoice 表上运行 SQL 语句。

当我们 InvoiceService.findById(Long)方法，装载了 Invoice 对象后，但是当我们试图访问 supplier 和 invoiceLines 属性时，抛出了 LazyInitializationException 异常。原因是装载 Invoice 对象的 Session 已经过期。我们曾经介绍过 HibernateTemplate 在回调方法执行结束后需要保证正确关闭会话。这也是 Hibernate 对事务支持发挥作用的地方。无论什么时候应用程序代码请求 SpringHibernate 支持获取会话，Spring 都将 Session 注册为一个事务同步对象。因此在单个事务中执行的代码总是操作在同一个 Session 上，即使它多次使用了 HibernateTemplate 调用。然而，在处理 bean 事务特性的方式选择上必须小心。看一看表 11-2，它展示了一个应用，该应用声明了它的服务层和 DAO 层是事务性的，更糟的是，它指示 DAO 类和服务 bean 的操作都会请求一个新的事务。

服务调用	DAO 调用	Hibernate Session
Invoice l = findById(1L)		Session 1
	Invoice i= getById(1L)	Session 2
	i.getSupplier()	Session 2
i.getSupplier()		Session 1

因为我们已经使用了 REQUIRES_NEW 事务委托配置了 bean,所以服务调用开始了一个新事务，得到 Hibernate session 1.然后它开始调用 DAO 的 getById()方法。我们已经为每一个调用使用 REQUIRES_NEW 事务委托配置了 DAO bean，它将获得 Hibernate session 2。DAO bean 使用 session 2 装载 Invoice 对象：由于 i 在 session 2 中装载，所以 i.getSupplier()将执行。然而，当 DAO bean 返回到服务层时，在服务层调用 i.getSupplier()将失败。即使我们有一个打开的 Hibernate session,它和装载原始 Invoice 对象的 session 仍不是同一个。第 16 章将对它进行详细介绍。

大家可能争论不会犯那样的错误，但是当服务调用是在 Web 层，并且 DAO 调用是在服务层时我们就有可能犯错误。我们有可能要处理相同的情况。若所有整合和 DAO 测试都工作正常，那问题可能变得更糟。我们采取的解决方案是在我们的 DAO 中需要它或者访问需要的服务层活跃事务中的对象时执行显式预先抓取。预先抓取和延迟抓取正好相反：我们命令 Hibernate 在单个查询语句中查询关系表<体会：即单个的关联抓取就算取完数据量也不会太大，所以内存可以容忍>。服务调用应用总是返回非事务表示层需要的所有数据。

1,显式预先抓取

```
/**
 * 显式预先抓取
```

```

    * */
    @Override
    public Invoice getById(Long id) {
        // TODO Auto-generated method stub

        return (Invoice) DataAccessUtils.uniqueResult(
            getHibernateTemplate().find("from Invoice i inner join
fetch" +
            " i.supplier inner join fetch i.lines il "+
            "left outer join fetch il.discounts where i.id
= ?", id));
    }

```

预先抓取看起来像使用了显式内联和左外连接的 SQL 语句。

这正是我们所需要的：我们有一个 DAO 类，它只在我们需要时返回预先抓取的对象。

2, 延迟加载的其它考虑

即使代码运行在一个干净的服务层事务中，我们也应该考虑因使用延迟加载导致的性能损失。下面的方法返回的是不使用预先抓取的 Invoice 对象

```

/**
 * 延迟抓取
 * */
    @Override
    public Invoice getByIdLazy(Long id) {
        // TODO Auto-generated method stub
        return (Invoice) getHibernateTemplate().get(Invoice.class, id);
    }

```

```

-----
/**Invoice pojo 中*/
    public BigDecimal getInvoiceLinesTotalPrice() {
        BigDecimal total = new BigDecimal(0);
        for (InvoiceLine line : this.invoiceLines) {
            total = total.add(line.getPrice());
        }
        return total;
    }

/**DefaultInvoiceService中*/
    @Override
    public void recalculateDiscounts(Long id) {
        // TODO Auto-generated method stub
        Invoice invoice = this.invoiceDao.getByIdLazy(id);
        BigDecimal total = invoice.getInvoiceLinesTotalPrice();
        if (total.compareTo(BigDecimal.TEN) > 0) {
            //do something special
        }
    }

```

```
}
```

对 `invoice.getInvoiceLinesTotalPrice()` 方法的单纯调用强制 `Hibernate` 抓取了该发票的所有 `InvoiceLines` 对象。

一些应用程序在视图(反)模式中使用开放会话。模式背后的缘由是应用程序在显示一个视图(例如 `JSP` 页面)时保持 `Session` 开放。这种做法简化了延迟抓取和预先抓取间的抉择：视图中的会话是开放的，它可以抓取任何延迟关联。我们有时在一个反模式中调用开放会话，但它可能导致不一致的数据视图。

11.5 处理大数据集

`Web` 应用非常有可能在一个结果页面中抛出大量的结果集显示给用户，我们需要让 `DAO` 层实现分页。`Hibernate` 在它的 `Query` 类中支持分页，该类提供了 `setFirstResult(int)` 和 `setMaxResult(int)` 方法。我们实现分页的第一次尝试可以参照下面的代码：

`HibernateInvoiceDao` 中添加方法

```
/**分页实现*/
@Override
public List<Invoice> search(final int firstResult, final int pageSize)
{
    // TODO Auto-generated method stub

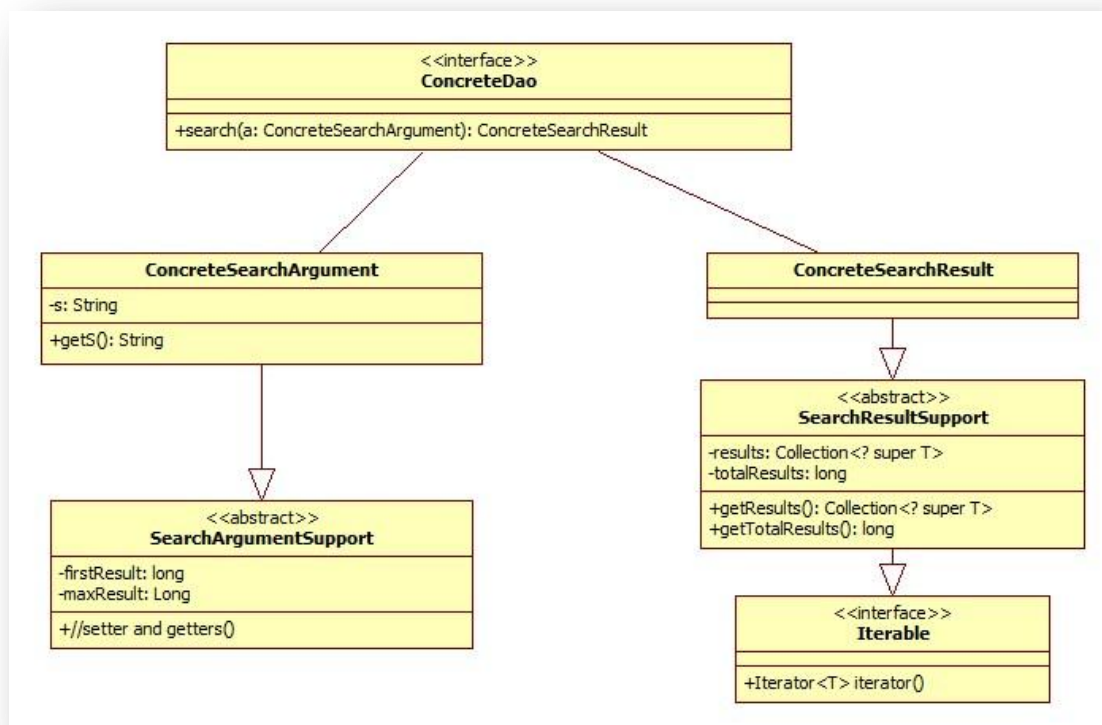
    return (List<Invoice>)getHibernateTemplate().executeFind(
        new HibernateCallback() {
            @Override
            public Object doInHibernate(Session session) throws
                HibernateException,
                SQLException {
                // TODO Auto-generated method stub
                Query query = session.createQuery("from Invoice");
                query.setFirstResult(firstResult);
                query.setMaxResults(pageSize);
                return query.list();
            }
        });
}
```

在 `DefaultInvoiceService` 中也添加方法

```
@Override
public List<Invoice> search(int firstResult, int pageSize) {
    // TODO Auto-generated method stub
    return this.invoiceDao.search(firstResult, pageSize);
}
```

我们现在可以使用搜索方法，设置第一个结果和返回结果集的最大数量。虽然这工作正常，但是我们不能获得结果集的最大数量。因此我们可以显示一个下一页的链接，我们不能显示页面或结果的总共数量。一种解决方案是实现一个返回 `Invoices` 表的总行数的方法。

如果我们可以设置条件来检索所有行的子集让搜索更复杂一些,那会怎么样呢?我们必须为每一个搜索方法实现一个相匹配的计数方法。另外,使用我们的服务的代码必须显式调用两个方法。有一个更好的解决方案,我们在下面的 UML 图对该方案进行说明。



搜索方法执行所有必须的搜索操作,使用 SearchArgumentSupport 子类作为方法的参数,返回 SearchResultSupport 子类。返回的 SearchResultSupport 包含抓取对象的页面和结果的总数。除了包含结果外,它实现了 Iterable<T>。这意味着我们可以在 JSP 的 c:forEach 轮询中使用 SearchResultSupport 子类。

```

<c:forEach items="${invoice}" var="invoice">
    ${invoice.invoiceDate} <!-- etc -->
</c:forEach>
  
```

自己尝试实现 UML 图中的分页方案,编写源码。

如下

抽象类 SearchArgumentSupport

```
package com.apress.prospring2.ch11.search;
```

```

public abstract class SearchArgumentSupport {
    private int firstResult;
    private int maxResult;

    public SearchArgumentSupport(int firstResult, int maxResult) {
        super();
        this.firstResult = firstResult;
    }
  
```

```
        this.maxResult = maxResult;
    }

    public SearchArgumentSupport() {
        super();
        // TODO Auto-generated constructor stub
    }

    public int getFirstResult() {
        return firstResult;
    }

    public void setFirstResult(int firstResult) {
        this.firstResult = firstResult;
    }

    public int getMaxResult() {
        return maxResult;
    }

    public void setMaxResult(int maxResult) {
        this.maxResult = maxResult;
    }
}

实现类 InvoiceSearchArgument
package com.apress.prospring2.ch11.search;

public class InvoiceSearchArgument extends SearchArgumentSupport {
    private String queryString;

    public InvoiceSearchArgument(int firstResult, int maxResult,
        String queryString) {
        super(firstResult, maxResult);
        this.queryString = queryString;
    }

    public String getQueryString() {
        return queryString;
    }

    public void setQueryString(String queryString) {
        this.queryString = queryString;
    }
}
```

抽象类 SearchResultSupport

```
package com.apress.prospring2.ch11.search;
```

```
import java.util.Collection;
```

```
import java.util.Iterator;
```

```
public abstract class SearchResultSupport<T> implements Iterable<T> {
    private Collection<? super T> results;
    private int totalResults;
    public int getTotalResults() {
        return totalResults;
    }

    public SearchResultSupport(Collection<? super T> results, int totalResults) {
        super();
        this.results = results;
        this.totalResults = totalResults;
    }

    public Collection<? super T> getResults() {
        return results;
    }

    @SuppressWarnings("unchecked")
    @Override
    public Iterator<T> iterator() {
        // TODO Auto-generated method stub
        return (Iterator<T>) results.iterator();
    }
}
```

实现类 InvoiceSearchResult

```
package com.apress.prospring2.ch11.search;
```

```
import java.util.Collection;
```

```
import com.apress.prospring2.ch11.domain.Invoice;
```

```
public class InvoiceSearchResult extends SearchResultSupport<Invoice> {

    public InvoiceSearchResult(Collection<? super Invoice> results,
        int totalResults) {
        super(results, totalResults);
    }
}
```



```

        // TODO Auto-generated constructor stub
    }

}

DAO 新增方法 HibernateInvoiceDao 中
/**分页的一个更好的解决方案*/
@SuppressWarnings("unchecked")
public InvoiceSearchResult search(final InvoiceSearchArgument
argument) {

    List<Invoice> lists =
(List<Invoice>)getHibernateTemplate().executeFind(
        new HibernateCallback() {
            @Override
            public Object doInHibernate(Session session) throws
HibernateException,
                SQLException {
                    // TODO Auto-generated method stub
                    Query query =
session.createQuery(argument.getQueryString());
                    query.setFirstResult(argument.getFirstResult());
                    query.setMaxResults(argument.getMaxResult());
                    return query.list();
                }
        });

    Integer totalSize = new Integer(getHibernateTemplate().execute(
        new HibernateCallback() {
            @Override
            public Object doInHibernate(Session session) throws
HibernateException,
                SQLException {
                    // TODO Auto-generated method stub
                    Object obj = session.createQuery("select count(*)
"+argument.getQueryString()).uniqueResult();
                    return obj;
                }
        }).toString());

    InvoiceSearchResult result = new
InvoiceSearchResult(lists,totalSize);

    return result;
}

```

测试类

```
package com.apress.prospring2.ch11.search;
```

```

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.apress.prospring2.ch11.domain.Invoice;
import com.apress.prospring2.ch11.interfaces.InvoiceService;
import com.apress.prospring2.ch11.utils.JNDIUtils;

public class PerfectSearchDemo {
    public static void main(String[] args) {
        JNDIUtils.buildJndi();
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "/datasource-context-tx.xml");
        InvoiceService invoiceService = (InvoiceService)ac.getBean("invoiceService");
        StringBuilder sb = new StringBuilder();
        //分页查询并显示
        final int pageSize = 5;
        for(int i = 0; i < 10; i++){
            //查十页，每一页显示 5 个
            InvoiceSearchResult result =
                invoiceService.search(new InvoiceSearchArgument(i*pageSize, pageSize,
"from Invoice"));
            sb.append(i + " : Get " + result.getTotalResults() + " Objects\n");
            for(Invoice invoice : result){
                sb.append(invoice.toString());
            }
            sb.append("\n");
        }
        System.out.println(sb.toString());
    }
}

```

启示：其中的分页查询底层代码如下：是利用操作 **rownum** 而来的

```

select
    *
from
    ( select
        row_.*,
        rownum rownum_
    from
        ( select
            invoice0_.id as id1_,
            invoice0_.version as version1_,
            invoice0_.delivery_date as delivery3_1_,
            invoice0_.invoice_date as invoice4_1_,

```

```
        invoice0_.supplier as supplier1_
      from
        t_invoice invoice0_ ) row_
    where
      rownum <= ?
  )
where
  rownum_ > ?
```

11.6 处理大对象

使用 Hibernate 从使用了大对象(LOB)的表中抓取对象。有两种类型的 LOB：字符大对象（CLOB）和二进制大对象（BLOB）。CLOB 通常映射为 `String`，而 BLOB 通常映射为 `byte[]`。因为标准的 JDBC 架构不处理大二进制对象，所以我们必须为我们正使用的数据库选择合适的 `LobHandler` 实现。看下面的示例

建表语句

```
create table t_lob_test (
  id number(19,0) not null,
  version number(19,0) not null,
  text_content clob not null,
  binary_content blob not null,
  mime_type varchar2(200) not null,
  constraint pk_lob_test_id primary key (id)
);
create sequence s_lob_test_id start with 10000;
```

t_lob_test 表的域对象

```
package com.apress.prospring2.ch11.domain;
```

```
public class LobTest extends AbstractIdentityVersionObject<Long> {
  private String textContent;
  private byte[] binaryContent;
  private String mimeType;
  public String gettextContent() {
    return textContent;
  }
  public void settextContent(String textContent) {
    this.textContent = textContent;
  }
  public byte[] getBinaryContent() {
    return binaryContent;
  }
}
```

```

    public void setBinaryContent(byte[] binaryContent) {
        this.binaryContent = binaryContent;
    }
    public String getMimeType() {
        return mimeType;
    }
    public void setMimeType(String mimeType) {
        this.mimeType = mimeType;
    }
    @Override
    public String toString() {
        // TODO Auto-generated method stub
        StringBuilder sb = new StringBuilder();
        sb.append("LobTest { id=" + this.id).append(", ");
        sb.append("textContent=").append(this.textContent).append(",
");
        sb.append("binaryContent=");
        for(int i = 0; i < binaryContent.length && i < 50; i++){
            sb.append(String.format("%x", (int) this.binaryContent[i]));
        }
        sb.append("}");
        return sb.toString();
    }
}

```

next->为lobTest域创建配置文件

HIBERNATE 配置文件 LOBTEST.HBM.XML

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="com.apress.prospring2.ch11.domain"
    default-lazy="true">
    <class name="LobTest" table="t_lob_test">
        <id name="id" type="long" unsaved-value="null">
            <generator class="sequence">
                <param name="sequence">s_lob_test_id</param>
            </generator>
        </id>
        <version name="version" column="version" unsaved-value="null"
type="long"/>
        <property name="binaryContent" column="binary_content"
not-null="true"

```

```

        type="org.springframework.orm.hibernate3.support.BlobByteArrayType
    e"/>
        <property name="textContent" column="text_content"
not-null="true"

        type="org.springframework.orm.hibernate3.support.ClobStringType"/
    >
        <property name="mimeType" column="mime_type" not-null="true"/>
    </class>
</hibernate-mapping>

```

使用些域对象的唯一复杂部分是建立 Hibernate 映射和 Spring 配置。我们需要将数据库(oracle 10g)处理 LOB 的方式告诉 Hibernate。为了实现这个操作，我们创建 LobHandler 实现的一个实例并在 HibernateSessionFactoryBean 中引用它。

datasource-context-tx.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:tx="http://www.springframework.org/schema/tx"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/jee
http://www.springframework.org/schema/jee/spring-jee-2.5.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.
xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

    <jee:jndi-lookup id="dataSource"
jndi-name="java:comp/env/jdbc/prospring2/ch11"
        expected-type="javax.sql.DataSource" />

    <bean id="hibernateSessionFactory"

        class="org.springframework.orm.hibernate3.LocalSessionFactoryBean
    ">

```

```

    <property name="dataSource">
        <ref bean="dataSource" />
    </property>
    <property name="mappingLocations">
        <!-- 本来用通配符表示这些文件，即classpath*//*.hbm.xml
            发现了一个错误，说是Caused by:
java.lang.StringIndexOutOfBoundsException: String index out of range: 0
            其实真正的原因是由于，我也定义了一个模板文件 即template.hbm.xml
            这之中没有填写正确的映射信息而发生的错误，删除或者按下面的
            写一个一个写出来就行了
        -->
        <list>
            <value>classpath*/LobTest.hbm.xml</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">
                org.hibernate.dialect.Oracle9Dialect
            </prop>
            <prop key="hibernate.show_sql">true</prop>
            <prop key="hibernate.format_sql">true</prop>
            <prop key="hibernate.current_session_context_class">
                thread
            </prop>
            <prop key="hibernate.transaction.factory_class">
                org.hibernate.transaction.JDBCTransactionFactory
            </prop>
        </props>
    </property>
</bean>

<bean id="transactionManager"

    class="org.springframework.orm.hibernate3.HibernateTransactionMan
ager">
    <property name="sessionFactory"
ref="hibernateSessionFactory" />
</bean>
<!-- 下面2个bean添加Spring事务支持，如果出错的话先删除吧 -->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="*" /><!-- 不知道星号对不对，不对的话引成具体的方法
名 -->

```

```
</tx:attributes>
</tx:advice>

<aop:config>
    <aop:pointcut expression="execution(*
com.apress.prospring2.ch11.service.*(..))"
id="sampleServiceOperation"/>
    <aop:advisor advice-ref="txAdvice"
pointcut-ref="sampleServiceOperation"/>
</aop:config>
    <bean id="hibernateDaoSupport" abstract="true"

        class="org.springframework.orm.hibernate3.support.HibernateDaoSupport">
        <property name="sessionFactory"
ref="hibernateSessionFactory"/>
        </bean>
        <bean id="lobTestDao"
class="com.apress.prospring2.ch11.daoimpl.HibernateLobTestDao"
parent="hibernateDaoSupport"/>
        <bean id="lobTestService"
class="com.apress.prospring2.ch11.serviceimpl.DefaultLobTestService">
            <property name="lobTestDao" ref="lobTestDao"/>
        </bean>

        <!-- 处理LOB需要添加的重要配置 -->
        <bean id="nativeJdbcExtractor"
class="org.springframework.jdbc.support.nativejdbc.SimpleNativeJdbcExtractor"/>
        <bean id="lobHandler"
class="org.springframework.jdbc.support.lob.OracleLobHandler">
            <property name="nativeJdbcExtractor"
ref="nativeJdbcExtractor"/>
        </bean>

</beans>
```

我们需要解决的最后一个难点是：**Lob** 支持要与现有的 **Spring** 事务或 **JTA** 事务同步。因此，我们必须创建一个服务层接口及实现。（请注意：我们甚至不想让 **DAO** 实现具有事务性！）

dao 及 service 源代码这里不再给出，就如前面的例子所述声明 **LobTestDao** 及 **LobTestService** 接口，然后写它们的实现 **HibernateLobTestDao**, **DefaultLobTestService**。其中 **DefaultLobTestService** 里面有一个属性 **lobTestDao** 及其 **setter** 方法。

LOB 示例应用程序

```
package com.apress.prospring2.ch11.demo;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.apress.prospring2.ch11.domain.LobTest;
import com.apress.prospring2.ch11.service.LobTestService;
import com.apress.prospring2.ch11.utils.JNDIUtils;

public class LobTestDemo {
    private void run() {
        JNDIUtils.buildJndi();
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "/datasource-context-tx.xml");
        LobTestService lobTestService =
            (LobTestService)ac.getBean("lobTestService");
        LobTest lobTest = new LobTest();
        lobTest.setTextContent("Hello, world");
        lobTest.setBinaryContent("Hello, world".getBytes());
        lobTest.setMimeType("text/plain");
        lobTestService.save(lobTest);

        LobTest lobTest2 = lobTestService.findById(lobTest.getId());
        System.out.println(lobTest2);
    }

    public static void main(String[] args) throws IOException {
        new LobTestDemo().run();
        new BufferedReader(new
            InputStreamReader(System.in)).readLine();
    }
}
```

是能成功执行并插入数据，但是会出现一点异常

ERROR [org.springframework.jdbc.support.lob.OracleLobHandler] Could not free Oracle LOB

[java.sql.SQLException](#): 关闭的连接

此方法的唯一缺陷是 LOB 列通常包含相当数量的数据，若在列数据上设置 lazy="true"将不

能工作。Hibernate 使用 AbstractLobType 子类抓取列的所有内容。这就是为什么在我们的大型应用程序中使用 Spring JDBC 处理 LOB，而由 Hibernate 处理所有其它数据访问代码的原因。

第 16 章 事务管理

事务是可靠的企业级应用的一个关键组成部分。简单来说，一个事务包含如下内容：开始事务命令，SQL 更新、删除等，以及提交/回滚命令。但实际上，事务所包含的内容要远远超过这些。

Spring 对声明性事务提供了非常棒的支持，这意味着事务管理代码不会再对我们的代码随意切割了。你可以使用传统方式，这时 Spring 会为目标 bean 中的方法创建代理，还可以使用 AOP 以及 tx-advice 标签。

16.1 Spring 事务抽象层简介

无论你是否使用 Spring，当使用事务时我们必须做出一个基本选择——使用全局事务还是局部事务。局部事务特指一个单独的事务性资源(如 JDBC 连接)，而全局事务是由容器管理的，可以包含多个事务性资源。

局部事务的管理很简单，由于大多数的操作仅仅与一个事务性资源（如 JDBC 事务）交互，因此使用局部事务就足够了。然而如果你没有使用 Spring，就需要编写大量的事务管理代码。如果某一天事务范围需要跨越到多个事务资源，你必须放弃局部事务管理代码，而使用全局事务进行重写。

非 Spring 应用的全局事务在大多数情况下是使用 JTA 进行实现的。JTA 是一个复杂的 API，它依赖于 JNDI，这就意味着我们必须使用一个 Java EE 应用服务器。如果不想用编程的方式来实现 JTA 事务，我们可以使用 EJB 的容器管理事务(container managed transaction,CMT)功能，该功能由 Java EE 应用服务器提供。这样你只需要简单地声明哪个操作需要放到一个事务中。只要这么做，容器就会进行事务管理。这是首先的管理策略，因为你编写的代码不包含任何显式的事务管理代码，所有困难的工作都交给了容器。EJB CMT 的另一个优势在于它使你摆脱了直接使用 JTA API 的困境，但根据其定义，你仍必须使用 EJB。

16.2 分析事务属性

事务有 4 个众所周知的 ACID 属性：原子性 atomicity、一致性 consistency、隔离性 isolation 以及持久性 durability，事务性资源必须维护事务的 these 方面。我们无法控制事务的原子性、一致性及持久性，但可以控制超时，设置事务的只读性以指定隔离级别。

在理想情况下，我们无需担心独立性——所有的事务都是完全独立的。实际上，独立的事务意味着顺序执行（也就是说一个接一个的执行）。这确保了完全隔离，但严重限制了系统的吞吐量。我们可以使用隔离级别来控制事务实际的独立程序。

Spring 在 TransactionDefinition 接口封装了所有这些设置。该接口用在 Spring 事务支持的核心接口 PlatformTransactionManager 中，它的实现提供了特定平台上的事务管理支持，如

JDBC 或者 JTA。其核心方法是 `PlatformTransactionManager.getTransaction()` 返回一个 `TransactionStatus` 的引用，它将控制事务的执行，具体地说就是设置事务的结果，并检查事务是否是只读的或者是否是一个新事务。

16.2.1 探索 TransactionDefinition 接口

此接口控制着事务的属性。有四个方法

```
int getPropagationBehavior();
```

```
int getIsolationLevel();
```

```
int getTimeout();
```

```
Boolean isReadOnly();
```

其中 `getTimeout()` 返回一个事务必须完成的时间限制(单位秒)，`isReadOnly()` 它表示事务是否只读。事务管理器的实现可以利用这个值来优化事务的执行，并确保事务只进行读取操作。

隔离级别

`getIsolationLevel()` 对其他事务所能看到的数据变化进行控制。

默认为 `TransactionDefinition.ISOLATION_READ_COMMITTED` 大多数数据库的默认级别。在事务完成前，其他事务无法看到该事务所修改的数据。遗憾的是，在该事务提交后，你就可以查看其他事务插入或更新的数据。这意味着在事务的不同点上，如果其他事务修改了数据，你就会看到不同的数据。

开销最大的是 `TransactionDefinition.ISOLATION_SERIALIZABLE`，所有的事务都是按顺序一个接一个地执行。

传播行为

其中有 `TransactionDefinition.PROPROPAGATION_`

`REQUIRED` 当前如果有事务，就使用该事务，否则会开始一个新事务

`SUPPORTS` 否则不会开始一个新事务

`MANDATORY` 否则会抛出异常

`REQUIRES_NEW` Spring 总是会开始一个新事务。如果当前有事务，则该事务挂起

`NOT_SUPPORTED` Spring 不会执行事务中的代码。如果当前有事务，就挂起它。

`NEVER` 即使当前有事务，Spring 也会在非事务环境下执行。如果当前有事务，则抛出异常

`NESTED` 如果当前有事务，则在嵌套事务中执行。如果没有相当于 `REQUIRED`

16.2.2 使用 TransactionStatus 接口

此接口可以让事务管理器控制事务的执行，可以检查事务是不是一个新事务，或者是否只读。`TransactionStatus` 还可以初始化回滚操作。

其中有 3 个方法

```
Boolean isNewTransaction();
```

```
void setRollbackOnly();
```

```
Boolean isRollbackOnly();
```

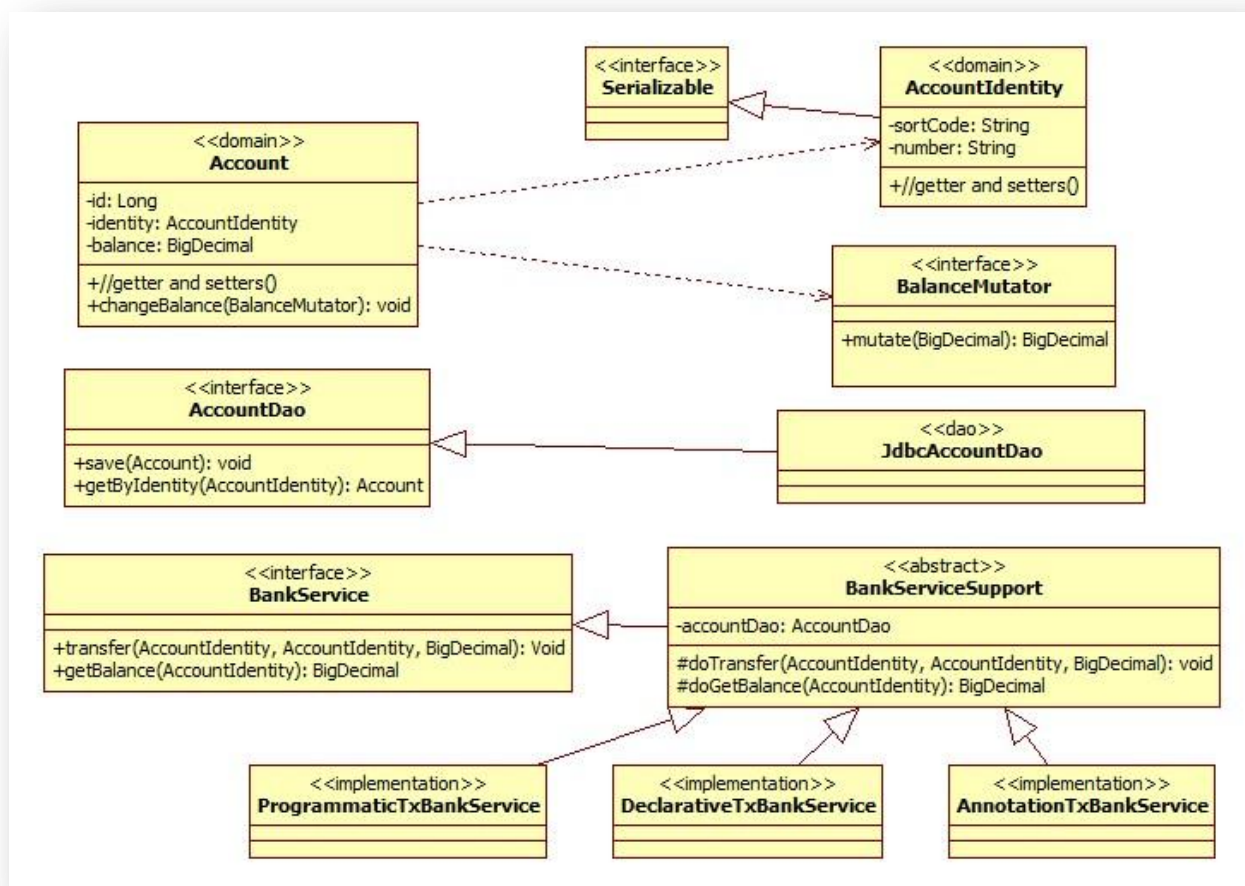
其中的 `setRollbackOnly()` 它将一个事务标识为不可提交的。换句话说，在调用完 `setRollbackOnly` 后你所能执行的唯一操作就是回滚。在大多数情况下，事务管理器会检测到这一点，在它发现事务要提交时会立刻结束事务。

16.2.3 PlatformTransactionManager 的实现

此接口使用 `TransactionDefinition` 和 `TransactionStatus` 接口，创建并管理事务。该接口的实现必须对事务处理器有深入的理解。`DataSourceTransactionManager` 控制着从 `DataSource` 中获得的 JDBC Connection 上的事务的执行；`HibernateTransactionManager` 控制着 `Hibernate Session` 上的事务的执行；`JdoTransactionManager` 管理着 JDO 事务；`JtaTransactionManager` 将事务管理委托给 JTA。

16.3 对一个事务管理示例的探索

使用事务操作的方式有 3 种基本方式：可以使用声明式事务，只需声明某个方法需要事务就行了；可以使用源码级的元数据来说明某个方法需要一个事务；还可以编写事务代码的方式来实现。Spring 对这 3 种方式都提供了支持。



非事务性代码

在讨论事务管理之前，我们得编写必要的支持代码。
创建表 `t_account` 并插入几条数据以供测试之用

createAccount.sql

```
create table t_account (
    id number(19,0) not null,
    sort_code varchar2(6) not null,
    number_ varchar2(8) not null,
    balance number(19,2) not null,
    constraint pk_account primary key (id)
);
insert into t_account (id,sort_code,number_,balance)
values(1,'011001','12345678',1000.0);
insert into t_account (id,sort_code,number_,balance)
values(2,'011001','87654321',100.0);
insert into t_account (id,sort_code,number_,balance)
values(3,'011001','10203040',0.0);
insert into t_account (id,sort_code,number_,balance)
values(4,'011001','50607080',30.0);
insert into t_account (id,sort_code,number_,balance)
values(5,'011001','10000000',1000000.0);
select * from t_account;
```

域对象 `Account.java` 它有一个我们赋予的识别器 (`Long id`)。该识别器就是实现的细节问题了：我们的代码需要这个 `ID`，但它对用户来说是没有任何意义的。用户使用账户号和分类号来标识其账户。`Account` 类只有一个重要的方法：`changeBalance(BalanceMutator)`。这个方法使用 `BalanceMutator` 实例来更新账户余额。

```
package com.apress.prospring2.ch16.domain;
```

```
import java.math.BigDecimal;
```

```
public class Account {
    private Long id;
    private AccountIdentity identity;
    private BigDecimal balance;
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public AccountIdentity getIdentity() {
        return identity;
    }
    public void setIdentity(AccountIdentity identity) {
        this.identity = identity;
    }
}
```

```
public BigDecimal getBalance() {
    return balance;
}

public void setBalance(BigDecimal balance) {
    this.balance = balance;
}

//
public void changeBanlance(BalanceMutator mutator){
    this.balance = mutator.mutate(this.balance);
}
}
```

域对象 AccountIdentity

```
package com.apress.prospring2.ch16.domain;

import org.springframework.util.Assert;

public class AccountIdentity {
    private String sortCode;
    private String number;

    public AccountIdentity() {
        super();
        // TODO Auto-generated constructor stub
    }

    public AccountIdentity(String sortCode, String number) {
        super();
        Assert.notNull(sortCode, "The 'sortCode' argument must not be null.");
        Assert.notNull(number, "The 'number' argument must not be null.");
        this.sortCode = sortCode;
        this.number = number;
    }

    public String getSortCode() {
        return sortCode;
    }

    public void setSortCode(String sortCode) {
        this.sortCode = sortCode;
    }

    public String getNumber() {
        return number;
    }
}
```

```

    }

    public void setNumber(String number) {
        this.number = number;
    }

    @Override
    public String toString() {
        // TODO Auto-generated method stub
        StringBuilder sb = new StringBuilder();
        sb.append("AccountIdentity {").append(this.sortCode);
        sb.append(" ").append(this.number).append("}");
        return sb.toString();
    }
}

```

BalanceMutator 接口

```
package com.apress.prospring2.ch16.domain;
```

```
import java.math.BigDecimal;
```

```
public interface BalanceMutator {
    BigDecimal mutate(BigDecimal balance);
}

```

说明：我们选择使用 BalanceMutator 来更新账户的余额。因为我们觉得管理余额的规则可能会非常复杂。使用一个接口（该接口在服务层实现）会比在 Account 类中简单地定义 credit(BigDecimal) 和 debit(BigDecimal) 方法更灵活

AccountDao 接口和 JdbcAccountDao 实现

```
package com.apress.prospring2.ch16.dao;
```

```
import com.apress.prospring2.ch16.domain.Account;
import com.apress.prospring2.ch16.domain.AccountIdentity;
```

```
public interface AccountDao {
    void save(Account account);
    Account getByIdentity(AccountIdentity accountIdentity);
}

```

```
package com.apress.prospring2.ch16.daoimpl;
```

```
import java.sql.ResultSet;
import java.sql.SQLException;
```

```
import org.springframework.jdbc.core.simple.ParameterizedRowMapper;
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;
```

```
import com.apress.prospring2.ch16.dao.AccountDao;
import com.apress.prospring2.ch16.domain.Account;
import com.apress.prospring2.ch16.domain.AccountIdentity;

public class JdbcAccountDao extends SimpleJdbcDaoSupport implements
AccountDao {

    private static final String INSERT_SQL =
        "insert into t_account (id,sort_code,number_,balance) values
" +
        "(?,?,?,?)";
    private static final String UPDATE_SQL =
        "update t_account set balance=? where id=?";
    private static final String SELECT_SQL =
        "select id,sort_code,number_,balance from t_account "+
        "where sort_code=? and number_=?";
    @Override
    public Account getByIdentity(AccountIdentity accountIdentity)
    {
        // TODO Auto-generated method stub
        return getSimpleJdbcTemplate().queryForObject(SELECT_SQL,
            new ParameterizedRowMapper<Account>() {
                @Override
                public Account mapRow(ResultSet rs, int rowNum)
                    throws SQLException {
                    Account account = new Account();
                    account.setId(rs.getLong(1));
                    AccountIdentity identity = new AccountIdentity(
                        rs.getString(2),
                        rs.getString(3)
                    );
                    account.setIdentity(identity);
                    account.setBalance(rs.getBigDecimal(4));
                    return account;
                }
            },accountIdentity.getSortCode(),accountIdentity.getNumber()
        );
    }

    @Override
    public void save(Account account) {
        // TODO Auto-generated method stub
        if(account.getId() == null){
```

```

        //insert
        getSimpleJdbcTemplate().update(INSERT_SQL,
account.getId(),account.getIdentity().getSortCode(),

        account.getIdentity().getNumber(),account.getBalance());
    }else{
        //update
        getSimpleJdbcTemplate().update(UPDATE_SQL,
account.getBalance(),account.getId());
    }

}

}

```

BankServiceSupport 类，它为抽象类，该类作为事务性的 **BankService** 实现的父类。这样，**BankServiceSupport** 就用以 do 开关的 **protected** 方法去实现 **BankService** 中的方法。

```
package com.apress.prospring2.ch16.service;
```

```
import java.math.BigDecimal;
```

```
import org.springframework.util.Assert;
```

```
import com.apress.prospring2.ch16.dao.AccountDao;
import com.apress.prospring2.ch16.domain.Account;
import com.apress.prospring2.ch16.domain.AccountIdentity;
import com.apress.prospring2.ch16.domain.BalanceMutator;
import com.apress.prospring2.ch16.exceptions.InsufficientFundsException;
import com.apress.prospring2.ch16.exceptions.UnknownAccount;
```

```
public abstract class BankServiceSupport {
    private AccountDao accountDao;
    /**
     * 内部抽象类 BalanceMutatorSupport 提供了 BalanceMutator 接口的部分实现
     */
    protected abstract static class BalanceMutatorSupport implements BalanceMutator{
        private BigDecimal amount;

        public BalanceMutatorSupport(BigDecimal amount) {
            Assert.notNull(amount,"The 'amount' argument must not be null");
            this.amount = amount;
        }
        protected final BigDecimal getAmount(){
            return this.amount;
        }
    }
}

```



```

        protected abstract BigDecimal doMutate(BigDecimal balance);
        public final BigDecimal mutate(BigDecimal balance){
            return doMutate(balance);
        }
    }
}
/**
 * BalanceMutator 接口及 BalanceMutatorSupport 的最终实现类，用于转入转账
中文名字是信用余额改变器
 */
protected static class CreditBalanceMutator
    extends BalanceMutatorSupport{

    public CreditBalanceMutator(BigDecimal amount) {
        super(amount);
    }

    @Override
    protected BigDecimal doMutate(BigDecimal balance) {
        // TODO Auto-generated method stub
        return balance.add(getAmount());
    }
}

/**
 * 又一个实现，用于转出转账，它对余额进行检查，如果不免的话就不能转账
或消费相应的钱数
 */
protected static class NoOverdraftDebitBalanceMutator
    extends BalanceMutatorSupport{

    public NoOverdraftDebitBalanceMutator(BigDecimal amount) {
        super(amount);
    }

    @Override
    protected BigDecimal doMutate(BigDecimal balance) {
        BigDecimal result = balance.subtract(getAmount());
        if(result.compareTo(new BigDecimal(0)) < 0)
            throw new InsufficientFundsException(getAmount().subtract(balance));
        return result;
    }
}

protected void doTransfer(AccountIdentity from,AccountIdentity to,BigDecimal
amount){

```

```

        Account fromAccount = this.accountDao.getByIdentity(from);
        if(fromAccount == null) throw new UnknownAccount(from);
        Account toAccount = this.accountDao.getByIdentity(to);
        if(toAccount == null) throw new UnknownAccount(to);

        fromAccount.changeBanlance(new NoOverdraftDebitBalanceMutator(amount));
        toAccount.changeBanlance(new CreditBalanceMutator(amount));

        this.accountDao.save(fromAccount);
        this.accountDao.save(toAccount);
    }
    protected BigDecimal doGetBalance(AccountIdentity accountIdentity){
        Account account = this.accountDao.getByIdentity(accountIdentity);
        if(account == null) throw new UnknownAccount(accountIdentity);
        return account.getBalance();
    }
    public void setAccountDao(AccountDao accountDao) {
        this.accountDao = accountDao;
    }
}

```

方法 `doTransfer` 和 `doGetBalance` 实现了 `BankService` 的 `transfer` 和 `getBlance` 方法。之所以将其实现为 `protected` 方法的，是因为我们将要创建 `BankServiceSupport` 的不同子类，以说明 Spring 提供的各种事务管理技术。

BankService 接口

```

package com.apress.prospring2.ch16.service;

import java.math.BigDecimal;

import com.apress.prospring2.ch16.domain.AccountIdentity;

public interface BankService {

    void transfer(AccountIdentity from,AccountIdentity to,BigDecimal amount);
    BigDecimal getBalance(AccountIdentity accountIdentity);
}

```

不带事务管理的 BankServiceSupport 类的 Nontransactional 实现

```

package com.apress.prospring2.ch16.serviceimpl;

import java.math.BigDecimal;

import com.apress.prospring2.ch16.domain.AccountIdentity;
import com.apress.prospring2.ch16.service.BankService;
import com.apress.prospring2.ch16.service.BankServiceSupport;

```

```

public class DefaultBankService extends BankServiceSupport implements
    BankService {

    @Override
    public BigDecimal getBalance(AccountIdentity accountIdentity) {
        // TODO Auto-generated method stub
        return doGetBalance(accountIdentity);
    }

    @Override
    public void transfer(AccountIdentity from, AccountIdentity to,
        BigDecimal amount) {
        // TODO Auto-generated method stub
        doTransfer(from, to, amount);
    }

}

```

该示例应用的最后一部分内容就是 Spring 的 XML 配置文件了。我们将其分成两个文件：dao-context.xml 和 svc-context-*.xml。前者定义了 dataSource 和 accountDao bean；后者定义了 bankService bean。

dao-context.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"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="dataSource"
        class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName"
            value="oracle.jdbc.driver.OracleDriver">
        </property>
        <property name="url"
            value="jdbc:oracle:thin:@localhost:1521:XE">
        </property>
        <property name="username" value="system"/>
        <property name="password" value="root"/>
    </bean>

    <bean id="transactionManager"

        class="org.springframework.jdbc.datasource.DataSourceTransactionM
anager">

```

```

        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="accountDao"
        class="com.apress.prospring2.ch16.daoimpl.JdbcAccountDao">
        <property name="dataSource" ref="dataSource"/>
    </bean>

</beans>

```

svc-context-nt.xml 非事务性 bankService bean 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"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
<!-- 非事务性bankService bean的配置 -->
    <bean id="bankService"

        class="com.apress.prospring2.ch16.serviceimpl.DefaultBankService"
    >

        <property name="accountDao" ref="accountDao"/>
    </bean>

</beans>

```

bankService **样例程序**

```
package com.apress.prospring2.ch16.demo;
```

```
import java.math.BigDecimal;
```

```
import org.springframework.context.ApplicationContext;
```

```
import org.springframework.context.support.ClassPathXmlApplicationContext;
```

```
import com.apress.prospring2.ch16.domain.AccountIdentity;
```

```
import com.apress.prospring2.ch16.service.BankService;
```

```
public class Main {
```

```
    /**
```

```
     * 非事务性 bankService 演示
```

```
     */
```

```
    public static void main(String[] args) {
```

```
        ApplicationContext ac = new ClassPathXmlApplicationContext(
```

```
            new String[]{
```

```
                "classpath*:/dao-context.xml",
```

```

        "classpath*:/svc-context-nt.xml"

        });

        BankService bankService = (BankService)ac.getBean("bankService");
        final AccountIdentity a1 = new AccountIdentity("011001","12345678");
        final AccountIdentity a2 = new AccountIdentity("011001","10203040");

        System.out.println("Before");
        System.out.println(a1 + ": " + bankService.getBalance(a1));
        System.out.println(a2 + ": " + bankService.getBalance(a2));
        try{
            bankService.transfer(a1, a2, new BigDecimal("200.00"));
        }catch(Exception e){e.printStackTrace();}
        System.out.println("After");
        System.out.println(a1 + ": " + bankService.getBalance(a1));
        System.out.println(a2 + ": " + bankService.getBalance(a2));
    }
}

```

执行结果

```

Before
AccountIdentity {011001 12345678}: 1000
AccountIdentity {011001 10203040}: 0
After
AccountIdentity {011001 12345678}: 800
AccountIdentity {011001 10203040}: 200

```

完全正确。

该应用能够顺利执行：它可以使用 `AccountIdentity` 对象从数据库中找到对应的账户，修改每个账户的余额，然后将其保存到数据库中。唯一一个问题就是借入与借出操作并未放到一起。如果借出操作失败了，系统将无法取消代入操作。换句话说，我们从支付者账户中取出了钱，却没有将其打入收款者账户中，这样钱就丢失了。我们要确保将借入和借出放到一个单独的操作中。当从支付者的账户中借钱时，我们还需要确保该支付者使用 ATM 卡取出的钱不能超出此次交易后的余额。最后我们需要保证一量交易完成，关于该交易的信息将会保留下来，即使数据库服务器崩溃了。一言以蔽之，我们的 `transfer` 方法需要遵循 ACID 的要求。

16.4 编程式事务管理

首先我们看看如何以编程的方式直接使用 `PlatformTransactionManager`。我们不再使用前一节那些不事事务的代码了，而是实现 `BankServiceSupport` 的另一个子类 `ProgrammaticTxBankService`。

```
package com.apress.prospring2.ch16.serviceimpl;
```

```
import java.math.BigDecimal;
```

```
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.TransactionDefinition;
import org.springframework.transaction.TransactionStatus;
import
org.springframework.transaction.support.DefaultTransactionDefinition;

import com.apress.prospring2.ch16.domain.AccountIdentity;
import com.apress.prospring2.ch16.service.BankService;
import com.apress.prospring2.ch16.service.BankServiceSupport;

public class ProgrammaticTxBankService extends BankServiceSupport
implements
    BankService {
    private PlatformTransactionManager transactionManager;
    @Override
    public BigDecimal getBalance(AccountIdentity accountIdentity) {
        // TODO Auto-generated method stub
        return doGetBalance(accountIdentity);
    }

    @Override
    public void transfer(AccountIdentity from, AccountIdentity to,
        BigDecimal amount) throws Throwable {
        // TODO Auto-generated method stub
        TransactionDefinition transactionDefinition =
            new
DefaultTransactionDefinition(TransactionDefinition.PROPROPAGATION_REQUIRE);
        TransactionStatus transactionStatus =

        this.transactionManager.getTransaction(transactionDefinition);
        try{
            doTransfer(from, to, amount);
            this.transactionManager.commit(transactionStatus);
        }catch(Throwable t){
            this.transactionManager.rollback(transactionStatus);
            throw t;
        }
    }

    public void setTransactionManager(
        PlatformTransactionManager transactionManager) {
        this.transactionManager = transactionManager;
    }
}
```

```
}
```

修改 JdbcAccountDao 中的方法 save 为下:

```
@Override
    public void save(Account account) {
        Grinch.ruin();
        // TODO Auto-generated method stub
        if(account.getId() == null){
            //insert
            getSimpleJdbcTemplate().update(INSERT_SQL,
account.getId(),account.getIdentity().getSortCode(),

            account.getIdentity().getNumber(),account.getBalance());
        }else{
            //update
            getSimpleJdbcTemplate().update(UPDATE_SQL,
account.getBalance(),account.getId());
        }

    }
}
```

其中 Grinch 的代码为 它是模拟异常抛出,以观察事务是否可正确执行回滚

```
package com.apress.prospring2.ch16.utils;

import java.util.Random;

public final class Grinch {
    private static final Random RND = new Random();
    private static final String MESSAGE = "Muhehe! It's broken now.";
    public Grinch() {
        // TODO Auto-generated constructor stub
    }
    public static void ruin(){
        if(RND.nextInt() % 3 ==0){
            System.out.println(MESSAGE);
            throw new RuntimeException(MESSAGE);
        }
    }
}
```

Spring 配置文件 svc-context-ptx.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"
        xsi:schemaLocation="
            http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans.xsd">
```

```

<!-- 非事务性bankService bean的配置 -->
    <bean id="bankService"

        class="com.apress.prospring2.ch16.serviceimpl.ProgrammaticTxBankService">
        <property name="accountDao" ref="accountDao"/>
        <property name="transactionManager" ref="transactionManager"/>
    </bean>

</beans>

```

测试代码仍然为前面的 Main 类...此处不再给出

执行结果如下:

Before

AccountIdentity {011001 12345678}: 600

AccountIdentity {011001 10203040}: 400

Muhehe! It's broken now.

```

java.lang.RuntimeException: Muhehe! It's broken now.
    at com.apress.prospring2.ch16.utils.Grinch.ruin(Grinch.java:14)
    at
com.apress.prospring2.ch16.daoimpl.JdbcAccountDao.save(JdbcAccountDao
.java:47)
    at
com.apress.prospring2.ch16.service.BankServiceSupport.doTransfer(Bank
ServiceSupport.java:78)
    at
com.apress.prospring2.ch16.serviceimpl.ProgrammaticTxBankService.trans
fer(ProgrammaticTxBankService.java:32)
    at com.apress.prospring2.ch16.demo.Main.main(Main.java:29)

```

After

AccountIdentity {011001 12345678}: 600

AccountIdentity {011001 10203040}: 400

解释: 我们的 Grinch.ruin() 方法抛出了 RuntimeException, 然而我们在 ProgrammaticTxBankService.transfer 方法中捕获了该异常, 然后对事务进行了回滚。因此支付者和收款者账户在事务失败前后保持了平衡。

16.4.1 使用 TransactionTemplate 类

修改类 ProgrammaticTxBankService(2)

```
package com.apress.prospring2.ch16.serviceimpl;
```



```
import java.math.BigDecimal;

import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.TransactionStatus;
import
org.springframework.transaction.support.TransactionCallbackWithoutRes
ult;
import org.springframework.transaction.support.TransactionTemplate;

import com.apress.prospring2.ch16.domain.AccountIdentity;
import com.apress.prospring2.ch16.service.BankService;
import com.apress.prospring2.ch16.service.BankServiceSupport;

public class ProgrammaticTxBankService2 extends BankServiceSupport
implements
    BankService {
    private TransactionTemplate transactionTemplate;
    @Override
    public BigDecimal getBalance(AccountIdentity accountIdentity) {
        // TODO Auto-generated method stub
        return doGetBalance(accountIdentity);
    }

    @Override
    public void transfer(final AccountIdentity from, final
AccountIdentity to,
        final BigDecimal amount) throws Throwable {
        this.transactionTemplate.execute(new
TransactionCallbackWithoutResult() {
            @Override
            protected void
doInTransactionWithoutResult(TransactionStatus status) {
                doTransfer(from, to, amount);
            }
        });
    }

    public void setTransactionManager(
        PlatformTransactionManager transactionManager) {
        this.transactionTemplate = new
TransactionTemplate(transactionManager);
    }
}
```

注意其中的属性虽然为 `TransactionTemplate` 但设置方法仍然为 `setTransactionManager`。

16.4.2 编程式事务管理小结

尽管你可以使用编程式事务支持(而且 `TransactionTemplate` 类极大的简化了这一切),但仍然需要完成许多工作。如果你跳回到关于 AOP 的那几章,你会看到事务管理正好是 `around` 通知(advice)的一个极佳应用。

16.5 声明性事务管理

声明性事务管理意味着你无需在 `bean` 中编写任何事务管理代码,你只需将事务配置到 `bean` 上就可以了。因此我们可以使用代码清单 16-8 中的 `DefaultBankService`,在 `bean` 的配置文件中为 `transfer` 方法指定好事务。要想达到这个目的,最简单的方法就是使用代理——代理会拦截所有的方法调用。如果方法名位于事务配置中,代理就会起到 `around` 通知的作用。它会在目标方法调用前开启事务,然后在一个 `try/catch` 块中执行目标方法。如果目标方法正常完成,代理就会提交事务;如果目标方法抛出运行时异常,代理就会进行回滚。为了完成这些事情,代理需要使用配置好的 `PlatformTransactionManager`。这是声明性事务管理的核心概念,其独特之处就是创建代理的方式不同。让我们以遗留下来的 `TransactionProxyFactoryBean` 开始吧。

16.5.1 使用 `TransactionProxyFactoryBean`

简言之,该 `factory bean` 为目标 `bean` 创建了一个 JDK 代理并拦截所有的方法调用。它根据一些配置信息决定如何处理事务性代码。让我们先看一个例子吧。

除了 `dao-context.xml` 配置文件外再增一配置文件

`svc-context-transactionProxyFactoryBean.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"
       xsi:schemaLocation="
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="bankService"

        class="org.springframework.transaction.interceptor.TransactionPro
xyFactoryBean">
        <property name="target">
            <bean
class="com.apress.prospring2.ch16.serviceimpl.DefaultBankService">
                <property name="accountDao" ref="accountDao"/>
            </bean>
        </property>
    </bean>
```

```

    </property>
    <property name="transactionManager" ref="transactionManager"/>
    <property name="transactionAttributes">
        <value>
            *=PROPAGATION_REQUIRED
        </value>
    </property>
</bean>

```

```
</beans>
```

示例程序和前面的示例程序大同小异这里不再给出。

虽然这个配置不那么恰当，但它确实解决了很多问题，DefaultBankService 中的代码不用关注 transfer 方法现在运行在一个事务中。尽管 TransactionProxyFactoryBean 还是 Spring 1.x 式的声明性事务管理。

TransactionProxyFactoryBean 创建了一个 JDK 代理，该代理会拦截所有方法的调用。对于名字出现在 transactionProperties 的 key 中的任何方法，代理会使用指定好的传播级别开启一个事务。配置文件中的 * 号代表所有的方法都带有事务。

首先你可能注意到代理设定了 TransactionDefinition 实例的所有属性，这样即使目标方法抛出了异常（只要发生的异常在允许的异常集合之中），代理仍然会提交事务。你可以在 transactionAttributes 的属性中控制事务处理的方方面面。

最复杂的表达式看起来是这样的：
 transfer=PROPAGATION_REQUIRED,ISOLATION_SERIALIZABLE,readonly,timeout_500,+MyBenignException,+AnotherAllowedException,-BadException

你可能注意到 TransactionProxyFactoryBean 还有其他属性，例如 exposeProxy 和 pointcut，我们在关于 AOP 的那几章提到的一些术语与这些名字很相似。事实上，Spring 2.5 在 TransactionProxyFactoryBean 的实现中使用了 AOP。AOP 事务处理是 Spring 2.5 所推荐的声明性事务处理方式。

16.5.2 在事务管理中使用代理的含义

……代理将拦截对 transfer 方法的调用并开启一个新事务，但如果 transfer 方法内如果还有一个 getBalance 方法的话此方法不会开启一个新事务，因为它不会被代理所执行，而是被代理的目标所执行。

有时嵌套的事务会导致问题，在第 11 章中我们探索了 Hibernate session 上嵌套事务的含义以及由嵌套异常导致的延迟加载出现的问题。

如果确实需要嵌套事务，则使用((BankService)AopContext.currentProxy()).getBalance(to)，但这会将实现绑定到 Spring AOP 上。另外，你还可以使用 Spring AOP 事务管理，通过装载时编织达到目的。

16.6 AOP 事务管理

AOP 事务管理利用了 Spring AOP 的基础设施，在大多数情况下，Spring AOP 会创建一个 JDK

代理以拦截方法调用。你可以使用装载时编织以在装载期编织切面，这样不需要代理了。你有两种方式来配置 Spring AOP 事务管理，基于注解的配置以及 XML 配置。

16.6.1 使用基于注解的 AOP 事务管理

增加注释@Transactional

```
package com.apress.prospring2.ch16.serviceimpl;

import java.math.BigDecimal;

import org.springframework.transaction.annotation.Transactional;

import com.apress.prospring2.ch16.domain.AccountIdentity;
import com.apress.prospring2.ch16.service.BankService;
import com.apress.prospring2.ch16.service.BankServiceSupport;

public class DeclarativeTxBankService extends BankServiceSupport implements
    BankService {

    @Transactional
    @Override
    public BigDecimal getBalance(AccountIdentity accountIdentity) {
        // TODO Auto-generated method stub
        return doGetBalance(accountIdentity);
    }

    @Transactional
    @Override
    public void transfer(AccountIdentity from, AccountIdentity to,
        BigDecimal amount) throws Throwable {
        // TODO Auto-generated method stub
        doTransfer(from, to, amount);
    }

}
```

请注意@Transactional 属性，为了让 Spring 的事务管理基础设施可以利用该属性创建恰当的切入点和通知，我们需要使用 AOP 的自动代理和注解驱动的事务支持。

配置文件 `svc-context-annotationTx.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: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.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx.xsd">
<bean id="bankService"

    class="com.apress.prospring2.ch16.serviceimpl.DeclarativeTxBankSe
rvice">
    <property name="accountDao" ref="accountDao"/>
</bean>
<tx:annotation-driven transaction-manager="transactionManager"/>
<aop:aspectj-autoproxy />

</beans>
```

测试类

```
package com.apress.prospring2.ch16.demo;

import java.math.BigDecimal;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.apress.prospring2.ch16.domain.AccountIdentity;
import com.apress.prospring2.ch16.service.BankService;

public class AnnotationTxDemo {
    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            new String[]{
                "classpath*:./dao-context.xml",
                "classpath*:./svc-context-annotationTx.xml"
            });
        BankService bankService = (BankService)ac.getBean("bankService");
        final AccountIdentity a1 = new AccountIdentity("011001","12345678");
        final AccountIdentity a2 = new AccountIdentity("011001","10203040");

        System.out.println("Before");
        System.out.println(a1 + ": " + bankService.getBalance(a1));
        System.out.println(a2 + ": " + bankService.getBalance(a2));
        try{
```

```
        bankService.transfer(a1, a2, new BigDecimal("200.00"));
    } catch (Exception e) { e.printStackTrace(); } catch (Throwable e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    System.out.println("After");
    System.out.println(a1 + ": " + bankService.getBalance(a1));
    System.out.println(a2 + ": " + bankService.getBalance(a2));
}
}
```

实验结果：运行成功！

`<tx:annotation-driven />` 标签使用 `@Transactional` 注解创建恰当的事务管理切面。接下来由 `<aop:aspect-autoproxy />` 通知匹配的 bean。

1. 探索 tx:annotation-driven 标签

此标签是注解驱动的事务管理支持的核心。

- a) `transactionManager` 指定到现有的 `PlatformTransactionManager` bean 的引用，通知会使用该引用
- b) `mode` 指定 Spring 事务管理框架创建通知 bean 的方式。可用的值有 `proxy` 和 `aspectj`。前者是默认的值，表示通知对象是个 JDK 代理；后者表示 Spring AOP 会使用 `AspectJ` 创建代理。
- c) `order` 指定创建的切面的顺序。只要目标对象有多个通知就可以使用该属性
- d) `proxy-target-class` 该属性如果为 `true` 就表示你想要代理目标类而不是 bean 所实现的所有接口

2. 探索 @Transactional 注解

- a) 凭借 `@Transactional` 注解我们可以控制通知将要创建的事务定义的方方面面。就像使用 `transactionAttributes` 属性表达式一样，你可以指定传播、隔离级别、超时以及允许和不允许的异常。

3. 基于注解的事务管理小结

通过使用 `@Transactional` 注解可以使对应的被注解的方法运行在事务中，如果某个类被它修饰的话，那么该类实例化的 Spring bean 的所有方法都是事务性的。

16.6.2 使用 XML AOP 事务管理

Spring 带有一个 `<tx:advice/>` 标签，该标签会创建一个事务处理通知。我们所需做的就是创建一个切入点，该切入点匹配所有带事务的方法并引用事务性通知。这里仅给出配置文件

```
<?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.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx.xsd">
<bean id="bankService"

class="com.apress.prospring2.ch16.serviceimpl.DefaultBankService"
>
    <property name="accountDao" ref="accountDao"/>
</bean>
<aop:config>
    <aop:pointcut expression="execution(*
com.apress.prospring2.ch16.service.*(..))" id="allServiceMethods"/>
    <aop:advisor advice-ref="defaultTransactionAdvice"
pointcut-ref="allServiceMethods"/>
</aop:config>
<tx:advice id="defaultTransactionAdvice"
transaction-manager="transactionManager" />

</beans>
```

首先我们使用了 `defaultTransactionAdvice`，它指定了代理拦截到方法调用时需要执行的动作。如果我们令该标签为空，它就会假定所有的方法都需要事务。传播级别是 `PROPAGATION_DEFAULT`，隔离级别是 `ISOLATION_DEFAULT`，超时是 `TIMEOUT_DEFAULT`，`read-only` 属性是 `false`。同时所有运行时异常都会导致回滚。

16.7 在多个事务性资源上使用事务

到目前为止，我们所有的例子在事务中都只使用了一个资源：JDBC 连接。如果你要实现的 ứng dụng 的事务中包含多个资源，那就需要使用 `JtaTransactionManager` 了。

如果你使用了应用服务器管理的 `DataSource`，那么你也需要使用 `JtaTransactionManager`

16.8 实现你自己的事务同步

如果你正在开发一个大型应用，你就会发现需要使用事务。本节我们将介绍如何对你自己的对象实现事务同步。

```
package com.apress.prospring2.ch16.interfaces;
```

```
public interface Worker {
    void work(int value);
}
```

```
void commit();  
void rollback();  
}
```

该接口非常简单。理想情况下我们只会在代码中使用 `Worker` 接口实现类的 `work(int)`方法，同时我们希望在一个事务性方法中使用 `Worker` 接口的实例时会自动调用 `commit()`和 `rollback()`方法。为了说明这个问题，我们创建一个简单的 `AccountService` 接口及其实现

```
package com.apress.prospring2.ch16.service;  
  
import com.apress.prospring2.ch16.domain.AccountIdentity;  
  
public interface AccountService {  
    AccountIdentity create();  
}  
package com.apress.prospring2.ch16.serviceimpl;  
  
import java.math.BigDecimal;  
import java.util.Random;  
  
import org.springframework.transaction.annotation.Transactional;  
  
import com.apress.prospring2.ch16.dao.AccountDao;  
import com.apress.prospring2.ch16.domain.Account;  
import com.apress.prospring2.ch16.domain.AccountIdentity;  
import com.apress.prospring2.ch16.interfaces.Worker;  
import com.apress.prospring2.ch16.interfaces.WorkerFactory;  
import com.apress.prospring2.ch16.service.AccountService;  
import com.apress.prospring2.ch16.utils.Grinch;  
  
public class DefaultAccountService implements AccountService {  
    private AccountDao accountDao;  
    private WorkerFactory workerFactory;  
    @Transactional  
    @Override  
    public AccountIdentity create() {  
        // TODO Auto-generated method stub  
        Random random = new Random();  
        StringBuilder number = new StringBuilder(8);  
        for(int i = 0; i < 8; i++){  
            number.append(random.nextInt(9));  
        }  
        AccountIdentity ai = new AccountIdentity("011001",number.toString());  
        Account account = new Account();  
        account.setIdx(System.currentTimeMillis());  
        account.setBalance(BigDecimal.ZERO);  
    }  
}
```



```

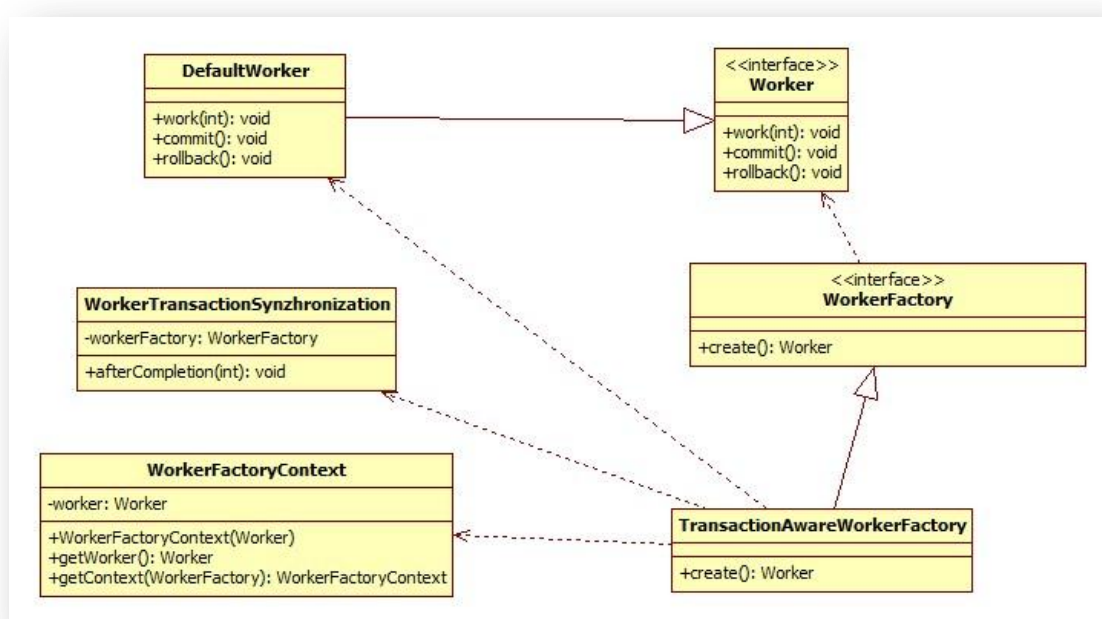
        account.setIdentity(ai);
        Worker worker = this.workerFactory.create();
        worker.work(10);
        this.accountDao.save(account);
        worker.work(20);
        Grinch.ruin();
        return ai;
    }

    public void setAccountDao(AccountDao accountDao) {
        this.accountDao = accountDao;
    }

    public void setWorkerFactory(WorkerFactory workerFactory) {
        this.workerFactory = workerFactory;
    }
}

```

其实现代码说明了其核心概念。我们有一个事务性方法 `AccountIdentity create()`。在该方法体中，我们获得了 `worker` 类的一个实例，它与当前的事务保持同步。接下来调用了 `worker.work()` 方法。然后又调用了 `dao.save` 方法，它会返回新账户的 `AccountIdentity` 对象。此后又调用了 `worker.work` 方法。最后我们使用 `Grinch.ruin()` 来模拟随机失败。我们希望当前事务提交时调用 `worker.commit` 方法，当事务回滚时调用 `worker.rollback` 方法。



可以看到我们有一个名为 `WorkerFactory` 的接口，并将其用在了 `DefaultAccountService` 中。`workerFactory` 的实现负责用 `TransactionSynchronizationManager` 注册一个 `TransactionSynchronization`。接下来当事务完成时，`TransactionSynchronizationManager` 会通知所有已注册的 `TransactionSynchronization` 对象。我们所需做的就是用同步键维护我们正在同步的 `worker` 实例。

WorkerFactoryContext 代码

```
package com.apress.prospring2.ch16.utils;

import org.springframework.transaction.support.TransactionSynchronizationManager;
import org.springframework.util.Assert;

import com.apress.prospring2.ch16.interfaces.Worker;
import com.apress.prospring2.ch16.interfaces.WorkerFactory;

public class WorkerFactoryContext {
    private Worker worker;

    public WorkerFactoryContext(Worker worker) {
        Assert.notNull(worker, "The argument 'worker' must not be null");
        this.worker = worker;
    }

    public Worker getWorker() {
        return worker;
    }

    public static WorkerFactoryContext getContext(WorkerFactory workerFactory){
        if(TransactionSynchronizationManager.isSynchronizationActive() &&
            TransactionSynchronizationManager.hasResource(workerFactory)){
            WorkerFactoryContext context =

            (WorkerFactoryContext)TransactionSynchronizationManager.getResource(workerFactory);
            if(context == null)
                throw new IllegalStateException(String.format(
                    "Null WorkerFactoryContext bound as " +
                    "transactional resource for [%s].",workerFactory ));
            return context;
        }
        throw
            new IllegalStateException(String.format(
                "Cannot access WorkerFactoryContext for [%s] when"+
                " transaction synchronization is not active.",workerFactory));
    }
}
```

上述第二个方法，检查同步是否被激活(也就是说是否有一个活动的事务)以及我们是否用 `workerFactory` 键注册了一个资源(`workerFactoryContext`)。如果是的话，该方法就返回已注册的 `WorkerFactoryContext`。我们将在 `WorkerFactory` 接口的 `TransactionAwareWorkerFactory` 实现中使用该 `WorkerFactoryContext`。

TransactionAwareWorkerFactory 实现

```
package com.apress.prospring2.ch16.isimpl;
```

```
import org.springframework.transaction.support.TransactionSynchronization;
import org.springframework.transaction.support.TransactionSynchronizationManager;

import com.apress.prospring2.ch16.interfaces.Worker;
import com.apress.prospring2.ch16.interfaces.WorkerFactory;
import com.apress.prospring2.ch16.utils.WorkerFactoryContext;
import com.apress.prospring2.ch16.utils.WorkerTransactionSynchronization;

public class TransactionAwareWorkerFactory implements WorkerFactory {

    public TransactionAwareWorkerFactory() {
        // TODO Auto-generated constructor stub
    }

    @Override
    public Worker create() {
        // TODO Auto-generated method stub
        if(TransactionSynchronizationManager.hasResource(this))
            return getTransactionBoundWorker();
        else
            return createNewTransactionBoundWorker();
    }

    private Worker getTransactionBoundWorker(){
        WorkerFactoryContext context = (WorkerFactoryContext)
            TransactionSynchronizationManager.getResource(this);
        return context.getWorker();
    }

    private Worker createNewTransactionBoundWorker(){
        Worker worker = new DefaultWorker();
        WorkerFactoryContext context = new WorkerFactoryContext(worker);
        TransactionSynchronization synchronization =
            new WorkerTransactionSynchronization(this);
        TransactionSynchronizationManager.registerSynchronization(synchronization);
        TransactionSynchronizationManager.bindResource(this, context);
        return worker;
    }
}
```

`create` 方法的实现会检查是否在该事务中使用 `this` 注册了一个资源。如果是的话，就返回绑定到 `WorkerFactoryContext`(我们对其进行了注册以获得活动的事务)上的 `Worker`。如果不是，我们就需要创建一个 `WorkerFactoryContext` 并使用 `TransactionSynchronizationManager` 来注册 `WorkerTransactionSynchronization`。

WorkerTransactionSynchronization 代码

```
package com.apress.prospring2.ch16.utils;
```

```
import org.springframework.transaction.support.TransactionSynchronizationAdapter;
import org.springframework.transaction.support.TransactionSynchronizationManager;

import com.apress.prospring2.ch16.interfaces.Worker;
import com.apress.prospring2.ch16.interfaces.WorkerFactory;

public class WorkerTransactionSynchronization extends
    TransactionSynchronizationAdapter {
    private WorkerFactory workerFactory;

    public WorkerTransactionSynchronization(WorkerFactory workerFactory) {
        this.workerFactory = workerFactory;
    }
    @Override
    public void afterCompletion(int status) {
        // TODO Auto-generated method stub
        if(!TransactionSynchronizationManager.hasResource(this.workerFactory)){
            throw new IllegalStateException(String.format(
                "Required synchronization resource missing under key '%s'.",
                this.workerFactory));
        }
        WorkerFactoryContext context = WorkerFactoryContext.getContext(workerFactory);
        Worker worker = context.getWorker();
        try{
            if(STATUS_COMMITTED == status){
                worker.commit();
            }else{
                worker.rollback();
            }
        }finally{
            TransactionSynchronizationManager.unbindResource(this.workerFactory);
        }
    }
}
```

我们并没有直接实现 `TransactionSynchronization` 接口，而是利用了方便的 `TransactionSynchronizationAdapter` 类，它将 `TransactionSynchronization` 接口中的所有方法进行了空实现并允许子类对这些方法进行重写。凭借这种实现，我们就可以只重写 `afterCompletion(int)` 方法了。在事务完成后，不管是提交还是回滚，`TransactionSynchronizationManager` 都会调用该方法。我们通过 `workerFactory` 键从 `TransactionSynchronizationManager` 中获得 `WorkerFactoryContext`。我们调用了 `context` 的 `getWorker()` 方法，然后根据事务是提交还是回滚来调用 `worker.commit` 还是 `worker.rollback`

DefaultAccountService 源码

```
package com.apress.prospring2.ch16.serviceimpl;
```

```
import java.math.BigDecimal;
import java.util.Random;

import org.springframework.transaction.annotation.Transactional;

import com.apress.prospring2.ch16.dao.AccountDao;
import com.apress.prospring2.ch16.domain.Account;
import com.apress.prospring2.ch16.domain.AccountIdentity;
import com.apress.prospring2.ch16.interfaces.Worker;
import com.apress.prospring2.ch16.interfaces.WorkerFactory;
import com.apress.prospring2.ch16.service.AccountService;
import com.apress.prospring2.ch16.utils.Grinch;

public class DefaultAccountService implements AccountService {
    private AccountDao accountDao;
    private WorkerFactory workerFactory;
    @Transactional
    @Override
    public AccountIdentity create() {
        // TODO Auto-generated method stub
        Random random = new Random();
        StringBuilder number = new StringBuilder(8);
        for(int i = 0; i < 8; i++){
            number.append(random.nextInt(9));
        }
        AccountIdentity ai = new AccountIdentity("011001",number.toString());
        Account account = new Account();
        account.setIdx(System.currentTimeMillis());
        account.setBalance(BigDecimal.ZERO);
        account.setIdentity(ai);
        Worker worker = this.workerFactory.create();
        worker.work(10);
        this.accountDao.save(account);
        worker.work(20);
        Grinch.ruin();
        return ai;
    }
    public void setAccountDao(AccountDao accountDao) {
        this.accountDao = accountDao;
    }
    public void setWorkerFactory(WorkerFactory workerFactory) {
        this.workerFactory = workerFactory;
    }
}
```

配置文件除了 dao-context.xml 外的 svc-context-worker.xml 文件

```
<?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.xsd
            http://www.springframework.org/schema/aop
            http://www.springframework.org/schema/aop/spring-aop.xsd
            http://www.springframework.org/schema/tx
            http://www.springframework.org/schema/tx/spring-tx.xsd">
    <bean id="workerFactory"

        class="com.apress.prospring2.ch16.isimpl.TransactionAwareWorkerFactory" />
    <bean id="accountService"

        class="com.apress.prospring2.ch16.serviceimpl.DefaultAccountService">
        <property name="accountDao" ref="accountDao"/>
        <property name="workerFactory" ref="workerFactory"/>
    </bean>
    <tx:annotation-driven transaction-manager="transactionManager"/>
    <aop:aspectj-autoproxy />
</beans>
```

测试程序

```
package com.apress.prospring2.ch16.demo;
```

```
import org.springframework.context.ApplicationContext;
```

```
import org.springframework.context.support.ClassPathXmlApplicationContext;
```

```
import com.apress.prospring2.ch16.service.AccountService;
```

```
public class TxSynzchronizationMain {
    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            new String[]{
                "classpath*:/dao-context.xml",
                "classpath*:/svc-context-worker.xml"
            });
        AccountService accountService = (AccountService) ac.getBean("accountService");
        try{
```

```
        accountService.create();
    }catch(Exception e){
        //e.printStackTrace();
    }
}
}
```

运行结果解析

结果很清楚的说明了我们已经成功地用 `TransactionSynchronizationManager` 注册了 `TransactionSynchronization` 回调，同时 `workerTransactionSynchronization` 会根据事务的完成状态去调用 `worker.commit` 或者 `worker.rollback`