Spring 数据库访问

> ## 本节目标
>
> 1.理解Spring的数据库访问模板化 2.掌握Spring中使用JDBC 3.掌握Spring中的事务管理

1. Spring的数据库访问

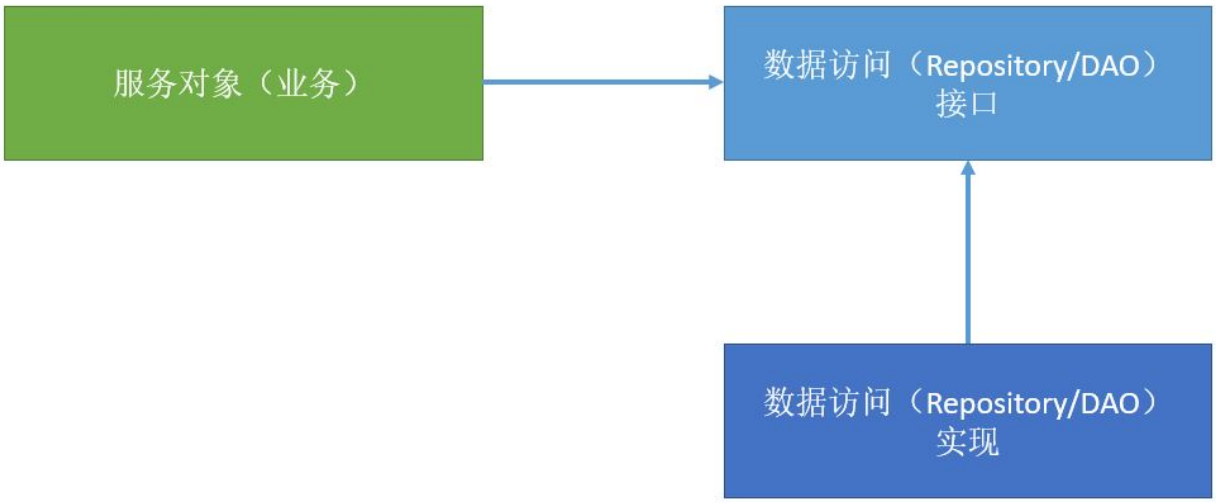
在掌握了Spring容器的核心知识之后，我们就需要把它应用到实际的项目中去。企业级应用项目离不开数据库的访问操作，我们在处理数据库访问的时候必须经过初始化数据库的驱动，打开连接，执行SQL，处理各种异常和关闭连接。 这样的重复操作过程中出现任何问题，都有可能造成数据的损坏。 因此，Spring框架为了解决此类问题，自带了一组数据库访问框架。

1.1 数据访问哲学

Spring的目标之一就是允许我们在开发应用程序时，能够遵循面向对象（OO）原则中的"针对接口编程"。Spring对数据访问的支持也不例外。

为了避免持久化的逻辑分散到应用的各个组件，最好的方式就是将数据访问的功能放到一个或者多个专注于此项任务的组件中，这样的组件通常称之为：数据访问对象（Data Asscess Object DAO）或者 Repository。

为了避免应用于特定的数据访问策略耦合在一起，编写良好的Repository应该以接口的方式暴露功能。



- ◆ 服务对象本身不会处理数据访问，而是将数据访问委托给Repository
- ◆ Repository接口确保其与服务对象的松耦合

服务对象通过接口访问Repository有以下两个好处：

- 第一：

使得服务对象易于测试，服务对象不再与特定的数据访问实现绑定在一起。实际上可以为这些数据访问接口创建Mock实现，这样无需来连接数据库就能测试服务对象，而且显著提升了单元测试的效率，并排除因数据不一致所造成的测试失败。

- 第二：

数据访问层是以持久化技术无关的方式进行访问的。持久化方式的选择独立于Repository，同时只有数据访问相关的方法才通过接口进行暴露。这样实现灵活的设计，并且切换持久化框架对应用程序其它部分所带来的影响最小。

1.2 数据库访问异常体系

1.2.1 JDBC异常

我们在学习JDBC编程时，编写JDBC代码需要强制捕获SQLException。SQLException表示在尝试访问数据库时出现了问题，但是这个异常并没有告我们出什么错误，以及如何进行处理。

可能导致抛出SQLExceptioin的常见问题包括：

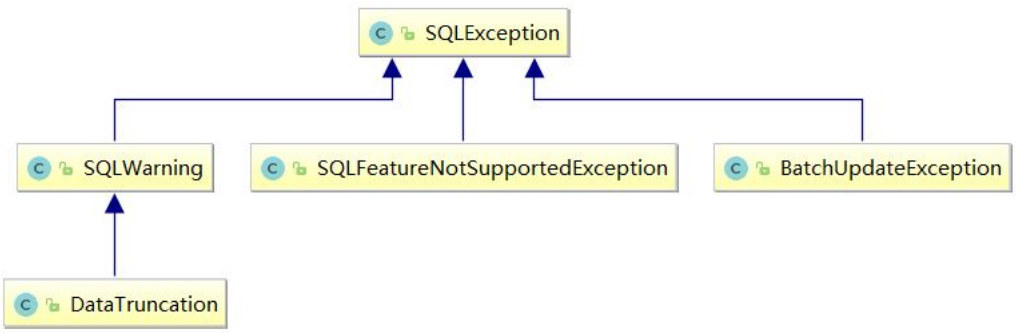
- 应用程序无法连接到数据库
- 要执行的查询存在语法错误
- 查询中所使用的表或列不存在
- 试图插入或者更新的数据违反了数据库表约束

SQLException的问题在于捕获到它的时候如何处理。事实上，能够触发SQLException的问题，通常是不能在catch代码块中解决的。大多数抛出SQLException的情况表明发生了致命性错误。比如：应用程序不能连接到数据库，这意味着应用不能继续使用了。

JDBC异常面临两个问题：

- 对所有数据访问问题都会抛出SQLException
- 不是针对每种可能的问题有不同的异常类型

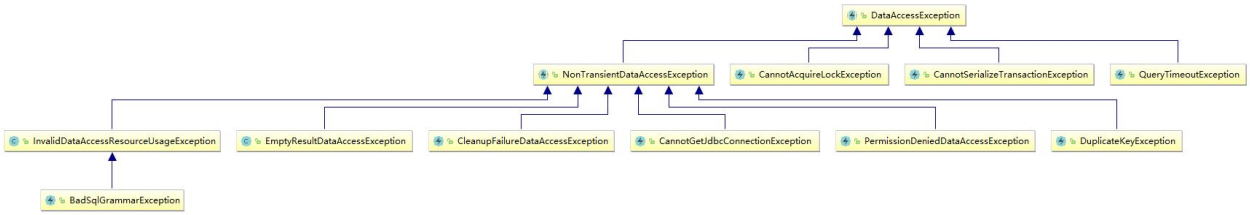
JDBC异常类（部分异常）：



1.2.2 平台无关的持久化异常

Spring JDBC提供的数据访问异常体系解决了以上两个问题。不同于JDBC，Spring提供了多个数据库访问异常，分别描述了它们抛出时所对应的问题。

Spring JDBC异常类（部分异常）：



尽管Spring的异常体系比JDBC简单的SQLException丰富的多，但是它并没有于特定的持久化方式相关联。这意味着我们可以使用Spring抛出一致的异常，而不用关心所选择的持久化方案。这有助于我们将所选择持久化机制与数据库访问层隔离开来。

不再使用catch代码块

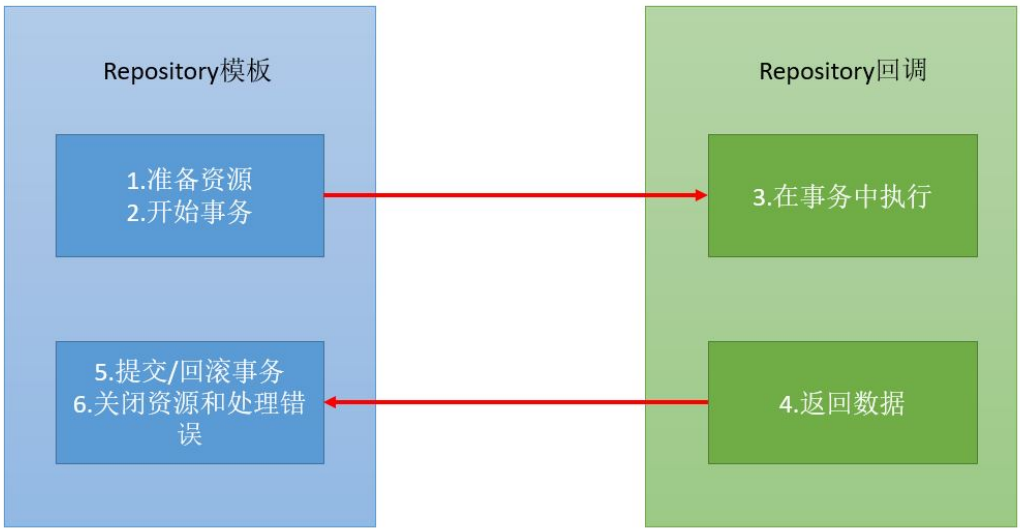
从异常类的继承关系中可以看出这些异常都继承自DataAccessException。 DataAccessException的特殊之处在于它是一个**非检查异常。** 言外之意，没有必要捕获Spring所抛出的数据访问异常（想捕获也是可以的）。

DataAccessException只是Spring处理**检查型异常**和**非检查型异常**哲学的一个范例。Spring认为触发异常的很多问题是不能再catch代码块中修复的。Spring使用了非检查型异常，而不是强制开发人员编写catch代码块。

1.3 数据访问模板化

Spring将数据访问过程中**固定的**和**可变的**的部分明确划分为两个不同的类：模板（template）和回调（callback）。

模板管理过程中固定的风格，而回调处理自定义的数据访问代码。



Spring的数据访问模板类负责通用的数据访问功能，对于应用程序特定任务，则会调用自定义回调对象。

如图所示，Spring的模板类处理数据访问的固定部分——事务控制，管理资源以及处理异常。同时，应用程序相关的数据访问——语句，绑定参数以及处理结果集在回调的实现中处理。事实证明，这是一个优雅的架构，我们只需要关注自己的数据访问逻辑即可。

Spring 提供的数据访问模板，分别适用于不同的持久化机制

编号	模板类（org.springframework.*）	用途
1	jca.cci.core.CciTemplate	JCA CCI连接
2	jdbc.core. JdbcTemplate	JDBC连接
3	jdbc.core.namedparam. NamedParameterJdbcTemplate	支持名命参数的JDBC连接
4	org.ibatis.SqlMapClientTemplate	Mybatis SqlMap客户端
5	orm.jpa.JpaTemplate	Java持久化API实体管理器
6	orm.jdo.JdoTemplate	Java数据对象（Java Data Object）实现
7	orm.hibernate3.HibernateTemplate	Hibernate3.x以上的Session

在本章节中我们重点学习JdbcTemplate模板。我们在使用Spring提供的各种模板类的前提条件是依赖数据源。在声明模板和Repository之前，我们需要在Spring中配置数据源来连接数据库。

2. 配置数据源

无论选择Spring的那种数据访问方式，都需要配置一个数据源的引用。Spring提供了在Spring上下文中配置数据源Bean的多种方式，包括：

- 通过JDBC驱动程序定义数据源
- 通过JNDI查找的数据源
- 连接池的数据源

本章节我们主要介绍基于JDBC驱动数据源和数据源连接池。

2.1 数据源连接池

Spring未提供数据源连接池实现，我们可以通过其它开源实现来进行数据源的配置。下面是常见的几种数据源连接池的开源实现：

- Apache Commons DBCP2 (http://commons.apache.org/proper/commons-dbcp/download_dbcp.cgi)
- c3p0 (http://www.mchange.com/projects/c3p0/)
- BoneCP (http://jolbox.com/)
- **HikariCP** (https://github.com/brettwooldridge/HikariCP)
- Druid （https://github.com/alibaba/druid）

这些数据库连接池中大多数都能配置为Spring的数据源，在一定程度上与Spring自带的DriverManagerDataSource或者SingleConnectionDataSource很类似。如下是我们配置的DruidDataSource的方式：

```
<!-- 数据库连接池 -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.1.2</version>
</dependency>
```

```
<!--阿里巴巴开源的druid数据源连接池-->
<bean id="druidDataSource" class="com.alibaba.druid.pool.DruidDataSource" init-
method="init" destroy-method="close">
    <property name="url"
              value="jdbc:mysql://127.0.0.1:3306/china?
useUnicode=true&amp;characterEncoding=UTF-
8&amp;useSSL=false&amp;createDatabaseIfNotExist=true"/>
    <property name="username" value="root"/>
    <property name="password" value="root"/>

    <!-- 配置初始化大小、最小、最大 -->
    <property name="initialSize" value="10"/>
    <property name="minIdle" value="10"/>
    <property name="maxActive" value="20"/>
    <property name="queryTimeout" value="30000"/>

    <!-- 配置获取连接等待超时的时间 -->
    <property name="maxWait" value="10000"/>
    <!-- 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒 -->
    <property name="timeBetweenEvictionRunsMillis" value="60000"/>
    <!-- 配置一个连接在池中最小生存的时间，单位是毫秒 -->
    <property name="minEvictableIdleTimeMillis" value="300000"/>

    <property name="validationQuery" value="SELECT 'x' FROM DUAL"/>
    <property name="testWhileIdle" value="true"/>
    <property name="testOnBorrow" value="false"/>
    <property name="testOnReturn" value="false"/>

    <!-- 打开PSCache，并且指定每个连接上PSCache的大小 如果用Oracle，则把poolPreparedStatements
配置为true，mysql可以配置为false。 -->
    <property name="poolPreparedStatements" value="false"/>
    <property name="maxPoolPreparedStatementPerConnectionSize" value="20"/>
</bean>
```

2.2 JDBC驱动的数据源

在Spring中，通过JDBC驱动定义数据源是最简单的配置方式。Spring JDBC位于spring-jdbc模块下：

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
</dependency>
```

Spring提供了三个这样的数据源类（位于包：org.springframework.jdbc.datasource下）。

- `DriverManagerDataSource`： 在每个连接请求时都会返回一个新建的连接。与Druid的DruidDataSource不同，由DriverManagerDataSource提供的连接并没有进行池化管理。
- `SimpleDriverDataSource`：与DriverManagerDataSource的工作方式类似，但是直接使用JDBC驱动，来解决在特定环境下的类加载问题，这样的环境包括OSGI容器。
- `SingleConnectionDataSource`：在每个连接请求时都会返回同一个连接，尽管SingleConnectionDataSource不是严格意义上的连接池数据源，但是可以将其视为只有一个连接的池。

如下是`DriverManagerDataSource`的配置：

```
<bean id="managerDataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://127.0.0.1:3306/china?
useUnicode=true&amp;characterEncoding=UTF-
8&amp;useSSL=false&amp;createDatabaseIfNotExist=true"/>
    <property name="username" value="root"/>
    <property name="password" value="root"/>
</bean>
```

与具备池功能的数据源相比，唯一的区别在于这些数据源Bean都没有提供连接池的功能，所以没有可配置的池相关的属性。

3. Spring中使用JDBC

3.1 失控的JDBC代码

我们在使用JDBC提供的直接操作数据库的API，需要负责处理与数据库访问相关的所有事情，其中包括管理数据库资源和处理异常。

下面的一个完整案例，展示了通过JDBC的API向数据库中添加一条记录，修改一条记录，查询一条记录的操作。

```
-- 创建数据库
create database if not exists `library` default character set utf8;
-- 创建表
create table if not exists `soft_bookrack` (
  `book_name` varchar(32) NOT NULL,
  `book_author` varchar(32) NOT NULL,
  `book_isbn` varchar(32) NOT NULL primary key
) ;
```

```
package com.bittech.jdbc.biz;

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

/**
 * Author: Deleteorus
 * Created: 2018/4/15
```

```
 */
public class SimpleJdbcOperation {

    private final DataSource dataSource;

    public SimpleJdbcOperation(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    /**
     * 添加一本书
     */
    public void addBook() {
        Connection connection = null;
        PreparedStatement stmt = null;
        try {
            //获取数据库连接
            connection = dataSource.getConnection();
            //创建语句
            stmt = connection.prepareStatement(
                    "insert into soft_bookrack (book_name, book_author, book_isbn)
values (?,?,?);"
            );
            //参数绑定
            stmt.setString(1, "Spring in Action");
            stmt.setString(2, "Craig Walls");
            stmt.setString(3, "9787115417305");
            //执行语句
            stmt.execute();
        } catch (SQLException e) {
            //处理异常信息
        } finally {
            //清理资源
            try {
                if (stmt != null) {
                    stmt.close();
                }
                if (connection != null) {
                    connection.close();
                }
            } catch (SQLException e) {
                //
            }
        }
    }

    /**
     * 更新一本书
     */
    public void updateBook() {
        Connection connection = null;
        PreparedStatement stmt = null;
        try {
```

```
            //获取数据库连接
            connection = dataSource.getConnection();
            //创建语句
            stmt = connection.prepareStatement(
                    "update soft_bookrack set book_author=? where book_isbn=?;"
            );
            //参数绑定
            stmt.setString(1, "张卫滨");
            stmt.setString(2, "9787115417305");
            //执行语句
            stmt.execute();
        } catch (SQLException e) {
            //处理异常信息
        } finally {
            //清理资源
            try {
                if (stmt != null) {
                    stmt.close();
                }
                if (connection != null) {
                    connection.close();
                }
            } catch (SQLException e) {
                //
            }
        }
    }

    /**
     * 查询一本书
     */
    public void queryBook() {
        Connection connection = null;
        PreparedStatement stmt = null;
        ResultSet rs = null;
        Book book = null;
        try {
            //获取数据库连接
            connection = dataSource.getConnection();
            //创建语句
            stmt = connection.prepareStatement(
                    "select book_name, book_author, book_isbn from soft_bookrack where
book_isbn =?"
            );
            //参数绑定
            stmt.setString(1, "9787115417305");
            //执行语句
            rs = stmt.executeQuery();
            if (rs.next()) {
                book = new Book();
                book.setName(rs.getString("book_name"));
                book.setAuthor(rs.getString("book_author"));
                book.setIsbn(rs.getString("book_isbn"));
```

```
            }
            System.out.println(book);
        } catch (SQLException e) {
            //处理异常信息
        } finally {
            //清理资源
            try {
                if (rs != null) {
                    rs.close();
                }
                if (stmt != null) {
                    stmt.close();
                }
                if (connection != null) {
                    connection.close();
                }
            } catch (SQLException e) {
                //
            }
        }
    }

    public static class Book {
        private String name;
        private String author;
        private String isbn;
        //省略 setter getter 方法
    }
}
```

上面冗长的代码，甚至非常复杂。其中只有20%的代码真正用于业务功能，其余80%的代码都是样板代码。不过，这些样板代码非常重要，清理资源和处理错误确保了数据访问的健壮性，避免了资源的泄露。

基于上面的原因，我们才需要框架来确保这些样板代码只写一次而且是正确的。

3.2 使用JDBC模板

Spring的JDBC框架承担了资源管理和异常处理的工作，从而简化了JDBC代码，让我们只需要编写从数据库读写数据的必须代码。

下面我们把JDBC代码改造为使用JdbcTemplate访问数据。

配置JdbcTemplate的Bean：

```
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="druidDataSource"/>
</bean>
```

改造JDBC代码之后：

```
package com.bittech.jdbc.biz;

import org.springframework.jdbc.core.JdbcTemplate;
```

```
import org.springframework.jdbc.core.RowMapper;

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

/**
 * Author: Deleteorus
 * Created: 2018/4/15
 */
public class JdbcTemplateOperation {
    private final JdbcTemplate jdbcTemplate;
    public JdbcTemplateOperation(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
    /**
     * 添加一本书
     */
    public void addBook() {
        this.jdbcTemplate.update(
                "insert into soft_bookrack (book_name, book_author, book_isbn) values
(?,?,?);",
                "Spring in Action",
                "Craig Walls",
                "9787115417305"
        );
    }

    /**
     * 更新一本书
     */
    public void updateBook() {
        this.jdbcTemplate.update(
                "update soft_bookrack set book_author=? where book_isbn=?;",
                "张卫滨",
                "9787115417305"
        );
    }

    /**
     * 查询一本书
     */
    public void queryBook() {
        /**
        * queryForObject方法有三个参数
        *
        * 第一个参数：String类型，要从数据库中查找数据的SQL
        * 第二个参数：RowMapper类型，用来从ResultSet中提取数据并构建成域对象（示例中的Book对象）
        * 第三个参数：可变参数列表，列出要绑定到查询上的索引参数值
        */
        Book book = jdbcTemplate.queryForObject("select book_name, book_author,
book_isbn from soft_bookrack where book_isbn =?",
                new RowMapper<Book>() {
                    @Override
```

```
                    public Book mapRow(ResultSet rs, int rowNum) throws SQLException {
                        Book book = new Book();
                        book.setName(rs.getString("book_name"));
                        book.setAuthor(rs.getString("book_author"));
                        book.setIsbn(rs.getString("book_isbn"));
                        return book;
                    }
                }, "9787115417305");
        System.out.println(book);
    }

    public static class Book {
        private String name;
        private String author;
        private String isbn;
        //省略 setter getter 方法
    }
}
```

看到改造之后的代码，第一感觉就是**简洁**，代码都是围绕业务编写的。但是需要明确一点是，虽然看不到样板代码了，但不代表其不存在，这样样板代码只是巧妙地隐藏到JDBC模板类中了。

4. Spring的事务管理

在使用Spring开发应用时，Spring的事务管理可能是被使用最多，应用最广的功能。Spring不但提供了和底层事务源无关的事务抽象，还提供了声明式事务的功能，可以让程序从事务代码中解放出来。

4.1 数据库事务回顾

4.1.1 什么是数据库的事务

“一荣俱荣，一损俱损”体现了事务的思想，很多复杂的事物要分步进行，但它们组成了一个整体，要么整体生效，要么整体失效。这种思想反映到数据库上，就是一组SQL语句，要么所有执行成功，要么所有执行失败。

4.1.2 数据库事务的特性

- 原子性（Atomic）
- 一致性（Consistency）
- 隔离性（Isolation）
- 持久性（Durabiliy）

4.1.3 事务的隔离级别

ANSI/ISO SQL 92标准定义了4个等级的事务隔离级别，在相同的数据环境下，使用相同的输入，执行相同的工作，根据不同的隔离级别，可能导致不同的结果。不同的事务隔离级别能够解决数据库并发问题的能力不同。

隔离级别	脏　读	不可重复读	幻　象　读	第一类丢失更新	第二类丢失更新
READ UNCOMMITED	允许	允许	允许	不允许	允许
READ COMMITTED	不允许	允许	允许	不允许	允许
REPEATABLE READ	不允许	不允许	允许	不允许	不允许
SERIALIZABLE	不允许	不允许	不允许	不允许	不允许

事务的隔离级别和数据库的并发性是对立的。一般来说，使用READ UNCOMMITED隔离级别的数据库拥有最高的并发性和吞吐量，而是用SERIALIZABLE隔离级别的数据库并发性最低。

SQL92 推荐使用`REPEATABLE READ`以保证数据库的读一致性，不过用户可以根据应用的需要选择合适的隔离等级。

> 备注：MySQL数据库的默认隔离级别是REPEATABLE READ

4.1.4 JDBC对事务的支持

- 查询数据库事务支持情况

并不是所有的数据库都支持事务，即使支持事务的数据库也并非支持多有的事务隔离级别。用户可以通过`Connection#getMetaData()`方法获取`DatabaseMetaData`对象，并通过该对象的`supportsTransactionIsolationLevel(int level)`方法查看底层数据库的事务支持情况。

检查示例代码如下：

```
Connection connection = dataSource.getConnection();
        DatabaseMetaData databaseMetaData = connection.getMetaData();
        System.out.println("支持事务: " + databaseMetaData.supportsTransactions());
        System.out.println("支持 Savepoint : " + databaseMetaData.supportsSavepoints());
        System.out.println("支持 ANSI92 Full SQL: " +
databaseMetaData.supportsANSI92FullSQL());
        System.out.println("支持 REPEATABLE_READ 隔离级别: " +
databaseMetaData.supportsTransactionIsolationLevel(Connection.TRANSACTION_REPEATABLE_RE
AD));
```

- 编程式处理事务逻辑

Connection默认情况下是自动提交，即每条执行的SQL语句都对应一个事务。为了将一组SQL语句当成一个事务执行，必须先通过Connection#setAutoCommit(false)阻止Connection自动提交，并且通过Connection#setTransactionIsolation()设置事务的隔离级别（注：Connection中定义对应SQL 92标准4个事务隔离级别的常量）。通过Connection#commit()提交事务，Connection#rollback()回滚事务。

编程式控制事务代码示例：

```
Connection connection = null;
try {
    //获取数据库连接
    connection = dataSource.getConnection();
    //关闭自动提交
    connection.setAutoCommit(false);
    //设置事务隔离级别
    connection.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
    //操作一组SQL
```

```
    Statement statement = connection.createStatement();
    int updateEffect = statement.executeUpdate("update memo_group set name='JDBC' where
id = 1");
    int deleteEffect = statement.executeUpdate("delete from memo_group where id=2");
    //提交事务
    if (updateEffect == 1 && deleteEffect == 1) {
        connection.commit();
        System.out.println("SQL commit");
    } else {
        connection.rollback();
        System.out.println("SQL rollback");
    }
} catch (SQLException e) {
    try {
        if (connection != null) {
            //回滚事务
            connection.rollback();
        }
    } catch (SQLException e1) {
        e1.printStackTrace();
    }
} finally {
    //释放资源
}
```

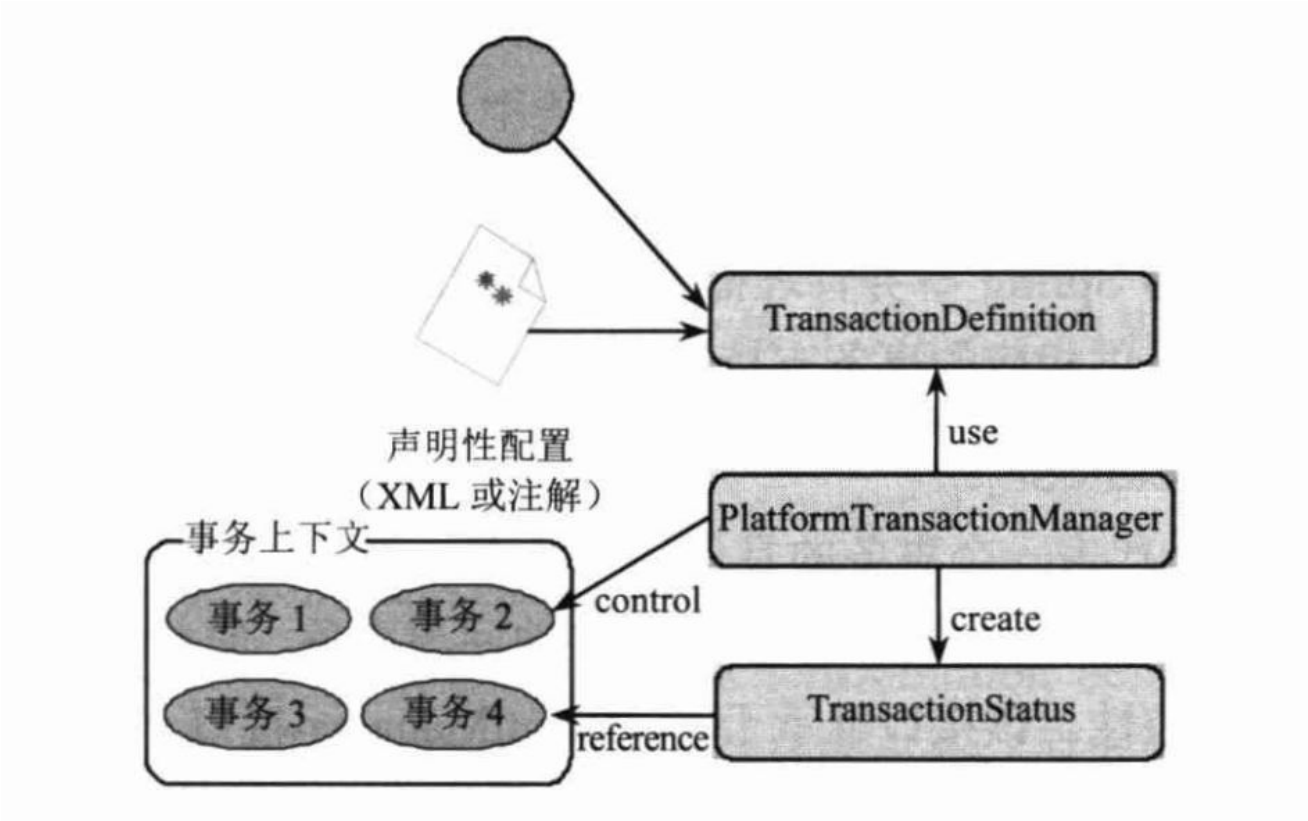
4.2 事务管理器实现

4.2.1 事务管理概述

- Spring为事务管理提供了一致性的编程模版，在高层次建立了统一的事务抽象。也就是说，不管是选择Spring JDBC、Hibernate、JPA还是Mybatis，Spring都可以让用户通过统一的编程模型进行事务管理。
- 像Spring DAO为不同的持久化实现提供了模版类一样，Spring事务管理继承了这一风格，也提供了事务模版类的TransactionTemplate。通过TransactionTemplate并配合使用事务回调TransactionCall指定的具体的持久化操作，就可以通过编程式实现事务的管理（即：**编程式事务管理**），无需关注资源的获取，复用，释放，事务同步和异常处理等操作。
- Spring事务管理的亮点在于**声明式事务管理。**Spring允许通过声明方式，在IoC容器配置中指定事务的边界，事务的属性，Spring自动在指定的事务边界上应用事务属性。
- Spring的事务处理高明之处在于无论使用那种持久化技术，是否使用JTA事务，都可以采用相同的事务管理模型。这种统一的处理方式所带来的好处是不可估量的，开发者完全可以抛开事务管理的问题编写程序，并在Spring中通过配置完成事务的管理工作。

4.2.2 事务管理关键抽象

在Spring的事务管理SPI（Service Provider Interface）的抽象层主要包括3个接口，分别是`PlatformTransactionManager,TransactionDefinition, TransactionStatus`它们位于`org.springframework.transaction`包中属于Spring的`org.springframework:spring-tx:${version}`模块。3个接口的关系如下图所示：



> 备注：结合源码了解事务管理的关键抽象接口

- TransactionDefinition：定义了Spring兼容的事务属性，这些属性对事务管理控制的若干方面进行配置
  - 事务隔离级别：当前事务和其它事务的隔离程度
  - 事务传播：通常在一个事务中执行的所有代码都会位于同一个事务上下文中，Spring提供了几个可选的事务传播类型，在Spring事务管理中，事务传播行为是一个重要概念，稍后详解。（**重要**）
  - 事务超时：事务在超时前能够运行多久，超时时间后，事务回滚。有些事务管理器不支持事务过期的功能。
  - 只读状态：只读事务不修改任何数据，事务管理器可以针对可读事务应用一些优化措施，提高运行性能。

  Spring允许通过**XML**或者**注解**元数据的方式为一个**有事务要求的服务类方法**配置事务属性，这些信息作为Spring事务框架的输入，Spring会自动按事务属性信息的指示，为目标方法提供相应的事务支持。
- TransactionStatus : 代表一个事务的具体运行状态。事务管理器可以通过该接口获取事务运行期的状态信息，也可以通过该接口间接地回滚事务，它相比于在抛出异常时回滚事务的方式更具可控制性。
- PlatformTransactionManager：通过JDBC的事务管理知识可以知道，事务只能被提交或者回滚（回滚到某个保存点后提交），Spring高层事务抽象接口 `org.springframework.transaction.PlatformTransactionManager` 很好地描述了事务管理这个概念。

4.2.3 事务管理器实现类

Spring将事务管理委托给底层具体的持久化实现框架来完成。因此，Spring为不同的持久化框架提供了 `PlatformTransactionManager` 接口的实现类。

下表是不同持久化技术对应的事务管理器实现类。

事务管理器实现类	说明
org.springframework.orm.jpa.JpaTransactionManager	使用JPA（Java Persistence API）进行持久化时，使用该事务管理器
org.springframework.orm.hibernate4.HibernateTransactionManager	使用Hibernate4版本进行持久化，使用该事务管理器
org.springframework.orm.hibernate5.HibernateTransactionManager	使用Hibernate5版本进行持久化，使用该事务管理器
org.springframework.jdbc.datasource.DataSourceTransactionManager	使用Spring JDBC或者Mybatis等基于DataSource数据源进行持久化技术，使用该事务管理器
org.springframework.orm.jdo.JdoTransactionManager	使用JDO进行持久化时，使用该事务管理器
org.springframework.transaction.jta.JtaTransactionManager	具有多个数据源的全局事务使用该事务管理器（不管采用何种持久化技术）

这些事务管理器都是对特定事务实现框架的代理，这样就可以通过Spring所提交的高级抽象对不同种类的事务实现使用相同的方式进行管理，而不用关系具体的实现。

要实现事务管理，首先要在Spring中配置好响应的事务管理器，为事务管理器指定数据源以及一些其它的事务管理控制属性。下面以Spring JDBC和Mybatis为例子，配置事务管理器：

```
<!-- 第一步：配置数据源 -->
<bean id="druidDataSource" class="com.alibaba.druid.pool.DruidDataSource" init-
method="init" destroy-method="close">
    <property name="url" value="jdbc:mysql://127.0.0.1:3306/china"/>
    <property name="username" value="root"/>
    <property name="password" value="root"/>
</bean>

<!-- 第二步：配置基于数据源的事务管理器 -->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <!-- 第三步：配置引用数据源 -->
    <property name="dataSource" ref="dataSource"/>
</bean>
```

在幕后DataSourceTransactionManager使用DataSource的Connection的commit(),rollback()等方法管理事务。

4.2.4 事务传播行为

当我们调用一个基于Spring的Service接口方法（如：UserServer#addUser()）时，它将运行于Spring管理的事务环境中，Service接口方法可能会在内部调用其它的Service接口方法以共同完成一个完整的业务操作，因此就会产生服务接口方法嵌套调用的情况。Spring通过事务传播行为控制当前的事务如何传播到被嵌套调用的目标服务接口方法中。

下面时Spring在TransactionDefinition接口中规定的7中事务传播行为类型：

事务传播行为类型	说　明
PROPAGATION_REQUIRED	如果当前没有事务，则新建一个事务；如果已经存在一个事务，则加入到这个事务中。这是最常见的选择
PROPAGATION_SUPPORTS	支持当前事务。如果当前没有事务，则以非事务方式执行
PROPAGATION_MANDATORY	使用当前的事务。如果当前没有事务，则抛出异常
PROPAGATION_REQUIRES_NEW	新建事务。如果当前存在事务，则把当前事务挂起
PROPAGATION_NOT_SUPPORTED	以非事务方式执行操作。如果当前存在事务，则把当前事务挂起
PROPAGATION_NEVER	以非事务方式执行。如果当前存在事务，则抛出异常
PROPAGATION_NESTED	如果当前存在事务，则在嵌套事务内执行；如果当前没有事务，则执行与PROPAGATION_REQUIRED 类似的操作

4.3 编程式事务（了解）

在实际应用中，很少需要通过编程来进行事务管理。即便如此，Spring还是为编程式事务管理提供了模板类`org.springframework.transaction.support.TransactionTemplate`，以满足一些特殊场合的需要。

TransactionTemplate和持久化模板了一样是线程安全的，因此可以在多个业务类中共享TransactionTemplate实例进行事务管理。

下面是TransactionTemplate使用示例：

```
<bean id="transactionTemplate"
class="org.springframework.transaction.support.TransactionTemplate">
    <!-- 使用上文中已经配置好的事务管理器 -->
    <property name="transactionManager" ref="transactionManager"/>
    <property name="isolationLevelName" value="ISOLATION_REPEATABLE_READ"/>
</bean>
```

```
TransactionTemplate transactionTemplate = context.getBean(TransactionTemplate.class);
transactionTemplate.execute(new TransactionCallback<Integer>() {
    @Override
    public Integer doInTransaction(TransactionStatus status) {
        //需要在事务中执行的代码
        //进行数据库访问操作
        int effect = memoGroupDao.update("1","DAO");
        return effect;
    }
});
```

由于Spring事务管理基于`TransactionSynchronizationManager`进行工作，所以如果在回调接口方法中需要显式访问底层数据连接，则必须通过资源获取工具类（比如：`org.springframework.jdbc.datasource.DataSourceUtils`）得到线程绑定的数据库连接。这是Spring事务管理的底层协议，不容违反。如果`memoGroupDao`是基于Spring提供的模板类构建，由于模板类已经在内部使用了资源获取工具类获取数据库连接，所以就不必关系底层数据库连接的获取问题。

4.4 声明式事务

大多数Spring用户会选择声明式事务管理的功能，这种方式对代码的侵入性最小，可以让事务管理代码完全从业务代码中移除，非常符合非侵入式轻量级容器的理念。

Spring的声明式事务管理是通过Spring AOP实现的，通过事务的声明信息，Spring负责将事务管理增强逻辑动态织入到业务方法的相应连接点中。这些逻辑包括获取线程绑定资源，开始事务，提交/回滚事务，进行异常转换和处理等工作。

声明式事务通过Spring AOP实现，我们通过切面的方式配置需要添加相应的依赖。

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aspects</artifactId>
</dependency>
```

4.4.1 XML配置声明式事务

Spring引用Aspect语言，通过`aop/tx`名命空间进行事务配置。

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

    <!-- 启用注解扫描，自动注入 -->
    <contex:component-scan base-package="com.bittech.jdbc"/>

    <!-- 0. 配置数据源 -->
    <bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource" destroy-
method="close">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost:3306/memo"/>
        <property name="username" value="root"/>
        <property name="password" value="root"/>
        <property name="maxTotal" value="20"/>
        <property name="maxIdle" value="10"/>
    </bean>

    <!-- 1.事务管理器 -->
    <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>
```

```
    <!-- 2. 事务增强 -->
    <tx:advice id="txAdvice" transaction-manager="transactionManager">
        <!-- 2.1 事务属性定义 -->
        <tx:attributes>
            <tx:method name="get*" read-only="true"/>
            <tx:method name="query*" read-only="true"/>
            <tx:method name="update*" rollback-for="Exception"/>
            <tx:method name="add*" rollback-for="Exception"/>
        </tx:attributes>
    </tx:advice>

    <!-- 3. 使用强大的切点表达式轻松定义目标方法 -->
    <aop:config>
        <!-- 3.1 通过AOP定义事务的增强切面 -->
        <aop:pointcut id="serviceMethod" expression="execution(*
com.bittech.jdbc.service.*.*.*(..))"/>
        <!-- 3.2 引用事务增强 -->
        <aop:advisor advice-ref="txAdvice" pointcut-ref="serviceMethod"/>
    </aop:config>
</beans>
```

在上述配置`<aop:config>`中的配置式切面的XML配置，事务配置的重点在于事务属性`<tx:attributes>`中的`<tx:method>`的配置。

下面是`<tx:method>`的属性信息描述：

属　性	是否必需	默 认 值	描　述
name	是	与事务属性关联的方法名，可使用通配符（*）	如“get*”、“handle*”、“on*Event”等
propagation	否	REQUIRED	事务传播行为，可选的值为：REQUIRED SUPPORTS、MANDATORY、REQUIRES_NEW、NOT_SUPPORTED、NEVER、NESTED
isolation	否	DEFAULT	事务隔离级别，可选的值为：DEFAULT、READ_UNCOMMITTED READ_COMMITTED、REPEATABLE_READ SERIALIZABLE
timeout	否	-1	事务超时的时间（以秒为单位）。如果设置为-1，则事务超时的时间由底层的事务系统所决定
read-only	否	false	事务是否只读
rollback-for	否	所有运行期异常回滚	触发事务回滚的 Exception，用异常名称的片段进行匹配。可以设置多个，以逗号分开，如“Exception1，Exception2”

属　性	是否必须	默 认 值	描　述
no-rollback-for	否	所有检查型异常不回滚	不触发事务回滚的 Exception，用异常名称的片段进行匹配。可以设置多个，以逗号分开，如“Exception1, Exception2”

基于`aop/tx`配置的声明式事务管理是实际应用中最常用的使用的事务管理方式，它的表达能力最强且使用最为灵活。

4.4.2 注解配置声明式事务

除了基于XML的事务配置，Spring还提供了基于注解的事务配置，即通过@Transactional对需要事务增强的Bean接口，实现类或方法进行标注；在容器中配置基于注解的事务增强驱动，即可启用基于注解的声明式事务。一般开启注解装配之后多才采用这种方式。

4.4.2.1 配置案例

- 配置中启用事务注解自动处理

```
<!-- 启用事务注解驱动 -->
<tx:annotation-driven transaction-manager="transactionManager"/>
```

- 在实现类上标注@Transactional注解

```
@Service
@Transactional
public class MemoGroupServiceImpl implements MemoGroupService {

    @Autowired
    private MemoGroupDao memoGroupDao;

    public List<MemoGroup> queryMemoGroup(int id) {
        return memoGroupDao.queryMemoGroupById(id);
    }

    public boolean addMemoGroup(MemoGroup memoGroup) {
        return memoGroupDao.insertMemoGroup(memoGroup) == 1;
    }
}
```

4.4.2.2 @Transactional注解属性

基于@Transactional注解的配置和基于XML的配置方式一样，它拥有一组普适性很强的默认事务属性，通常直接使用这些默认属性即可。当然也可以通过手工设定属性值覆盖默认值

下面是@Transactional注解属性说明（更多内容可以查看源代码）：

属 性 名	说 明
propagation	事务传播行为，通过以下枚举类提供合法值： org.springframework.transaction.annotation.Propagation 例如：@Transactional(propagation=Propagation.REQUIRES_NEW)
isolation	事务隔离级别，通过以下枚举类提供合法值： org.springframework.transaction.annotation.Isolation 例如：@Transactional(isolation=Isolation.READ_COMMITTED)
readOnly	事务读写性，布尔型。例如：@Transactional(readOnly=true)
timeout	超时时间，int 型，以秒为单位。例如：@Transactional(timeout=10)
rollbackFor	一组异常类，遇到时进行回滚，类型为：Class<? extends Throwable>[]，默认值为{}。例如： @Transactional(rollbackFor={SQLException.class})。多个异常之间可用逗号分隔
rollbackForClassName	一组异常类名，遇到时进行回滚，类型为 String[]，默认值为{}。例如： @Transactional(noRollbackForClassName={"Exception"})
noRollbackFor	一组异常类，遇到时不回滚，类型为 Class<? extends Throwable>[]，默认值为{}
noRollbackForClassName	一组异常类名，遇到时不回滚，类型为 String[]，默认值为{}

4.4.2.3 @Transactional注解位置

@Transactional注解可以被应用于：

- 接口定义
- 接口方法
- 类定义
- 类的public方法

但是Spring建议在**业务实现类**上使用@Transactional注解。因为注解不能被继承，所以在业务接口中标注的@Transactional注解不会被业务实现了继承。如果通过以下配置启用子类代理：

```
<tx:annotation-driven transaction-manager="transactionManager"
                      proxy-target-class="true"/>
```

那么业务类不会添加事务增强，照样工作在非事务环境下。因此标注到实现类上，无论是否启用子类代理都能够正常启用事务机制。

4.4.2.4 在方法处使用注解

方法处的注解会覆盖类定义处的注解。如果有写方法需要使用特殊的事务属性，则可以在类注解的基础上提供方法注解。

```
@Service
//1.类级注解，适用于类中所有public的方法
@Transactional
public class MemoGroupServiceImpl implements MemoGroupService {

    @Autowired
    private MemoGroupDao memoGroupDao;

    //2.提供额外的注解信息，它将覆盖1处的类级别注解
    @Transactional(transactionManager = "transactionManager",readOnly = true)
```

```
    public List<MemoGroup> queryMemoGroup(int id) {
        return memoGroupDao.queryMemoGroupById(id);
    }

    public boolean addMemoGroup(MemoGroup memoGroup) {
        return memoGroupDao.insertMemoGroup(memoGroup) == 1;
    }
}
```

总结

知识块	知识点	分类	掌握程度
AOP思想	1.异常体系 2.数据库访问模版类	实战型	掌握
事务管理	1.事务管理器 2.编程和声明式事务	实战型	掌握

实践

基于SpringCore搭建数据库项目，通过上面使用`JdbcTemplate`类的示例，改造便签应用。

架构：

- 业务层
- 数据层

功能：

- 便签
  - 创建便签
  - 根据编号删除便签
  - 根据时间查询便签
  - 根据内容查询便签
  - 分页查询便签
  - 设置指定便签为私密
  - 设置指定便签为提醒
- 便签组
  - 创建便签组
  - 修改便签组
  - 删除便签组
  - 查询指定组中的便签