一、使用 `JdbcTemplate`

`JdbcTemplate` 是 JDBC 核心包中的中心类. 它处理资源的创建和释放,帮助避免常见错误,例如忘记关闭连接. 它执行核心 JDBC 工作流的基本任务(例如语句创建和执行) ,并根据应用程序提供的 SQL 代码提取结果. `JdbcTemplate` 类:

- 执行 SQL 查询
- 更新语句和存储过程调用
- 对 `ResultSet` 实例执行迭代并提取返回的参数值.
- 捕获 JDBC 异常并将它们转换为 `org.springframework.dao` 包中定义的通用的、信息量更大的异常层次结构。

当为代码使用 `JdbcTemplate` 时,只需要实现回调接口,为它们提供明确定义的合同. 给定 `JdbcTemplate` 类提供的 `Connection`,`PreparedStatementCreator` 回调接口创建一个预准备语句,提供 SQL 和任何必要的参数. `CallableStatementCreator` 接口也是如此,它创建了可调用语句. `RowCallbackHandler` 接口从 `ResultSet` 的每一行中提取值.

可以通过引用 `DataSource` 的直接实例,在 DAO 实现中使用 `JdbcTemplate`, 也可以在 Spring IoC 容器中对其进行配置,并将其作为 bean 引用提供给 DAO.

引入依赖:

```
<dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
            <version>1.2.18</version>
        </dependency>

        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.30</version>
        </dependency>

        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjweaver</artifactId>
            <version>1.9.8</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aspects</artifactId>
            <version>5.3.28</version>
        </dependency>

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

```
            <version>5.3.28</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-orm</artifactId>
            <version>5.3.28</version>
        </dependency>
```

1.查询 (`SELECT`)

以下查询获取相关的行数:

```
int rowCount = this.jdbcTemplate.queryForObject("select count(*) from t_actor",
Integer.class);
```

以下查询使用绑定变量:

```
int countOfActorsNamedJoe = this.jdbcTemplate.queryForObject(
        "select count(*) from t_actor where first_name = ?", Integer.class, "Joe");
```

以下查询查找 `String`:

```
String lastName = this.jdbcTemplate.queryForObject(
        "select last_name from t_actor where id = ?",
        String.class, 1212L);
```

以下查询查找并填充单个 domain 对象:

```
Actor actor = jdbcTemplate.queryForObject(
        "select first_name, last_name from t_actor where id = ?",
        (resultSet, rowNum) -> {
            Actor newActor = new Actor();
            newActor.setFirstName(resultSet.getString("first_name"));
            newActor.setLastName(resultSet.getString("last_name"));
            return newActor;
        },
        1212L);
```

以下查询查找并填充多个 domain 对象:

```
List<Actor> actors = this.jdbcTemplate.query(
        "select first_name, last_name from t_actor",
        (resultSet, rowNum) -> {
            Actor actor = new Actor();
            actor.setFirstName(resultSet.getString("first_name"));
            actor.setLastName(resultSet.getString("last_name"));
            return actor;
        });
```

如果最后两个代码片段实际存在于同一个应用程序中,那么删除两个 `RowMapper` lambda 表达式中重复的代码段并将他们提取到一个字段 ,当需要时通过 DAO 方法引用它. 例如,最好重写前面的代码片段,如下所示:

```
private final RowMapper<Actor> actorRowMapper = (resultSet, rowNum) -> {
    Actor actor = new Actor();
    actor.setFirstName(resultSet.getString("first_name"));
    actor.setLastName(resultSet.getString("last_name"));
    return actor;
};

public List<Actor> findAllActors() {
    return this.jdbcTemplate.query( "select first_name, last_name from t_actor",
actorRowMapper);
}
```

2. 使用 `JdbcTemplate` 更新 (`INSERT`, `UPDATE`, and `DELETE`)

可以使用 `update(..)` 方法执行插入,更新和删除操作. 参数值通常作为变量参数提供,或者作为对象数组提供.

下面的例子是插入一个新的纪录:

```
this.jdbcTemplate.update(
        "insert into t_actor (first_name, last_name) values (?, ?)",
        "Leonor", "Watling");
```

以下例子是更新已存在的纪录:

```
this.jdbcTemplate.update(
        "update t_actor set last_name = ? where id = ?",
        "Banjo", 5276L);
```

以下例子是删除一条纪录:

```
this.jdbcTemplate.update(
        "delete from t_actor where id = ?",
        Long.valueOf(actorId));
```

3. `JdbcTemplate` 最佳实践

一旦配置, `JdbcTemplate` 类的实例就是线程安全的. 这很重要,因为这意味着可以配置 `JdbcTemplate` 的单个实例,然后将此共享引用安全地注入多个 DAO (或存储库) . `JdbcTemplate` 是有状态的,因为它维护对 `DataSource` 的引用,但此状态不是会话状态.

使用 `JdbcTemplate` 类时的常见做法是在 Spring 配置文件中配置 `DataSource` , 然后将共享 `DataSource` bean 依赖注入到 DAO 类中. `JdbcTemplate` 是在 `DataSource` 的 setter 中创建的. 这导致 DAO 类似于以下内容:

```
public class JdbcCorporateEventDao implements CorporateEventDao {

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    // JDBC-backed implementations of the methods on the CorporateEventDao follow...
}
```

使用 `@Repository` 注解类
使用 `@Autowired` 注解 `DataSource` 的 setter 方法
使用 `DataSource` 创建一个新的 `JdbcTemplate` .

以下示例显示了相应的 XML 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:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <bean id="corporateEventDao" class="com.example.JdbcCorporateEventDao">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
        <property name="driverClassName" value="${jdbc.driverClassName}"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
```

```
    </bean>

    <context:property-placeholder location="jdbc.properties"/>

</beans>
```

显式配置的替代方法是使用组件扫描和注解支持依赖注入. 在这种情况下,可以使用 `@Repository` 注解该类(这使其成为组件扫描的候选者) 并使用 `@Autowired` 注解 `DataSource` 的 setter 方法. 以下示例显示了如何执行此操作:

```
@Repository
public class JdbcCorporateEventDao implements CorporateEventDao {

    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    // JDBC-backed implementations of the methods on the CorporateEventDao follow...
}
```

使用 `@Repository` 注解类
使用 `@Autowired` 注解 `DataSource` 的 setter 方法
使用 `DataSource` 创建一个新的 `JdbcTemplate`.

以下示例显示了相关的 XML 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:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <!-- Scans within the base package of the application for @Component classes to
configure as beans -->
    <context:component-scan base-package="org.springframework.docs.test" />

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
        <property name="driverClassName" value="${jdbc.driverClassName}"/>
        <property name="url" value="${jdbc.url}"/>
```

```
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>

    <context:property-placeholder location="jdbc.properties"/>

</beans>
```

无论选择使用(或不使用) 上述哪种模板初始化样式,每次要运行 SQL 时,很少需要创建 `JdbcTemplate` 类的新实例. 配置完成后, `JdbcTemplate` 实例是线程安全的. 如果的应用程序访问多个数据库, 可能需要多个 `JdbcTemplate` 实例,这需要多个 `DataSources`, 随后需要多个不同配置的 `JdbcTemplate` 实例.

二、使用 `NamedParameterJdbcTemplate`

`NamedParameterJdbcTemplate` 类通过使用命名参数添加了对编写 JDBC 语句的支持, 而不是仅使用经典占位符(`'?'`) 参数编写 JDBC 语句. `NamedParameterJdbcTemplate` 类包装了 `JdbcTemplate` 并委托包装的 `JdbcTemplate` 来完成其大部分工作. 本节仅描述 `NamedParameterJdbcTemplate` 类与 `JdbcTemplate` 本身不同的区域 - 即使用命名参数编写 JDBC 语句. 以下示例显示如何使用 `NamedParameterJdbcTemplate`:

```
// some JDBC-backed DAO class...
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
}

public int countOfActorsByFirstName(String firstName) {

    String sql = "select count(*) from T_ACTOR where first_name = :first_name";

    SqlParameterSource namedParameters = new MapSqlParameterSource("first_name",
firstName);

    return this.namedParameterJdbcTemplate.queryForObject(sql, namedParameters,
Integer.class);
}
```

请注意在分配给 `sql` 变量的值中使用命名参数表示法以及插入 `namedParameters` 变量 (`MapSqlParameterSource` 类型) 的相应值.

或者,可以使用基于 `Map` 的样式将命名参数及其对应值传递给 `NamedParameterJdbcTemplate` 实例. `NamedParameterJdbcOperations` 暴露并由 `NamedParameterJdbcTemplate` 类实现的其余方法遵循类似的模式,此处不再介绍.

以下示例显示了基于 `Map` 的样式的使用:

```
// some JDBC-backed DAO class...
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;
```

```
public void setDataSource(DataSource dataSource) {
    this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
}

public int countOfActorsByFirstName(String firstName) {

    String sql = "select count(*) from T_ACTOR where first_name = :first_name";

    Map<String, String> namedParameters = Collections.singletonMap("first_name",
firstName);

    return this.namedParameterJdbcTemplate.queryForObject(sql, namedParameters,
 Integer.class);
}
```

与 `NamedParameterJdbcTemplate` 相关的一个很好的功能(并且存在于同一个 Java 包中) 是 `SqlParameterSource` 接口. 已经在之前的一个代码片段(`MapSqlParameterSource` 类) 中看到了此接口的实现示例. `SqlParameterSource` 是 `NamedParameterJdbcTemplate` 的命名参数值的来源. `MapSqlParameterSource` 类是一个简单的实现,它是 `java.util.Map` 的适配器,其中键是参数名称,值是参数值.

三、Spring的声明式事务

大多数 Spring Framework 用户选择声明式事务管理. 此选项对应用程序代码的影响最小,因此与非侵入式轻量级容器的理想最为一致.

随着 Spring 面向切面编程(AOP) 的使用,Spring Framework 的声明式事务管理成为可能. 但是,由于事务切面代码随 Spring Framework 发行版一起提供, 并且提供一些样板代码,因此可以不需要理解 AOP 概念也能使用此代码.

1.理解 Spring Framework 的声明式事务的实现

仅仅通过 `@Transactional` 注解是不够的,将 `@EnableTransactionManagement` 添加到您的配置中,并期望您了解它是如何工作的. 为了更深入地理解,本节解释了在发生与事务相关的问题时 Spring Framework 的声明式事务基础结构的内部工作原理.

关于 Spring Framework 的声明式事务支持,最重要的概念是通过 AOP 代理启用此支持,并且事务性的通知由元数据(当前基于 XML或基于注解) 驱动. AOP 与事务元数据的组合产生一个 AOP 代理,该代理使用 `TransactionInterceptor` 和适当的 `TransactionManager` 实现来驱动方法调用事务.

2.声明式事务实现的例子

以转账例子为例，转账接口设计两个功能，分别是转出功能和转入功能。在上一章使用JdbcTemplate的基础上完成DAO功能。

- 创建DAO接口

```
public interface AccountDao {
    void add();

    void reduce();
}
```

- 创建DAO接口实现类 用JdbcTemplate实现数据操作

```
/**
 * @Author: 索尔 VX: 214490523
 * @技术交流社区:  qfjava.cn
 */
@Repository
public class AccountDaoImpl implements AccountDao {

    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void setDataSource(DataSource dataSource){
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    /**
     * 小红转出
     */
    @Override
    public void reduce() {
        String sql = "update t_account set money = money-? where id=?";
        jdbcTemplate.update(sql,500,2);
    }

    /**
     * 小明转入
     */
    @Override
    public void add() {
        String sql = "update t_account set money = money+? where id=?";
        jdbcTemplate.update(sql,500,1);
    }


}
```

- 创建Service业务处理接口，定义转账功能

```
public interface AccountService {

    void transfer();

}
```

- 实现Service接口

```
/**
 * @Author: 索尔 VX：214490523
 * @技术交流社区： qfjava.cn
 */
@Service
public class AccountServiceImpl implements AccountService {

    @Autowired
    private AccountDao accountDao;

    @Override
    public void transfer() {
        System.out.println("小红转出");
        accountDao.reduce();
        int i = 100/0;
        System.out.println("小明转入");
        accountDao.add();
    }
}
```

在没有事务控制的情况下，当小红转出钱后程序出现异常，于是小明转入操作没办法执行。

3.基于XML的声明式事务的实现

- 在spring.xml配置文件中配置事务

```
<?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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
https://www.springframework.org/schema/context/spring-context.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
```

```
    <!--引入数据库配置文件-->
    <context:property-placeholder location="db.properties"></context:property-
placeholder>

    <!--配置datasource-->
    <bean class="com.alibaba.druid.pool.DruidDataSource" id="dataSource">
        <property name="username" value="${db.username}"></property>
        <property name="password" value="${db.password}"></property>
        <property name="url" value="${db.url}"></property>
        <property name="driverClassName" value="${db.driverClassName}"></property>

    </bean>

    <!--配置jdbc template-->
    <bean class="org.springframework.jdbc.core.JdbcTemplate" id="jdbcTemplate">
        <property name="dataSource" ref="dataSource"></property>
    </bean>

    <!--配置包扫描路径-->
    <context:component-scan base-package="com.qf"></context:component-scan>

    <!--开启事务控制-->
    <tx:annotation-driven transaction-manager="transactionManager"></tx:annotation-
driven>

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

- 在业务层使用`@Transactional`注解

```
@Service
public class AccountServiceImpl implements AccountService {

    @Autowired
    private AccountDao accountDao;

    @Transactional
    @Override
    public void transfer() {
        System.out.println("小红转出");
        accountDao.reduce();
        int i = 100/0;
        System.out.println("小明转入");
        accountDao.add();
```

```
    }
}
```

Transactional注解可以标注在类上或方法上。

- 标记在类上：适用于类中所有方法
- 标记在方法上：适用于当前方法

四、事务的特性

1.什么是事务

事务(Transaction)是用户定义的一个数据库操作序列，这些操作要么全做，要么全不做，是一个不可分割的工作单位。事务和程序是两个概念在关系数据库中，一个事务可以是一条SQL语句，一组SQL语句或整个程序一个程序通常包含多个事务。

2.定义事务

- 显式定义方式

通过begin-commit或者begin-rollback来实现事务的提交或回滚操作。

```
BEGIN TRANSACTION
        SQL 语句1
        SQL 语句2
        。。。。。
    COMMIT
```

```
BEGIN TRANSACTION
        SQL 语句1
        SQL 语句2
        。。。。。
    ROLLBACK
```

- 隐式定义方式

  当用户没有显式地定义事务时，数据库管理系统按缺省规定自动划分事务

3.事务的特性

事务的ACID特性：

- 原子性（Atomicity）： 事务是数据库的逻辑工作单位，事务中包括的诸操作要么都做，要么都不做
- 一致性（Consistency）： 事务执行的结果必须是使数据库从一个一致性状态变到另一个一致性状态
- 隔离性（Isolation）： 一个事务内部的操作及使用的数据对其他并发事务是隔离的
- 持续性（Durability ）： 一个事务一旦提交，它对数据库中数据的改变就应该是永久性的。

4.事务的隔离级别

不同的隔离级别会带来的不同的效果

- read uncommitted: 一个事务中会读到另一个事务未提交的数据。会出现脏读的现象
- read committed: 虽然解决了脏读，但是没有解决不可重复读——多次读到的数据是不一致的。
- repeatable read: 解决了不可重复读问题，会出现幻读，什么是幻读?
- serializable： 串行化，相当于是锁表，不存在并发

5.设置@Transactional的隔离级别

- isolation = Isolation.READ_UNCOMMITTED
- isolation = Isolation.READ_COMMITTED
- isolation = Isolation.REPEATABLE_READ
- isolation = Isolation.SERIALIZABLE

五、Spring事务的传播行为

事务的传播行为描述了在一个方法中调用另一个方法时，事务是如何传播的。

Spring提供了7种不同的传播行为：

事务传播类型	外部不存在事务	外部存在事务	使用方式	使用场景
REQUIRED（默认）	开启新的事务	当前事务加入到外部事务中	Propagation.REQUIRED	适用于增删改查
SUPPORTS	不开启新的事务	当前事务加入到外部事务中	Propagation.SUPPORTS	适用于查询
REQUIRES_NEW	开启新的事务	把外部事务挂起，创建新的事务	Propagation.REQUIRES_NEW	适用于内外事务不关联的场景
NOT_SUPPORTED	不开启新的事务	把外部事务挂起	Propagation.NOT_SUPPORTED	不常用
NEVER	不开启新的事务	抛出异常	Propagation.NEVER	不常用
MANDATORY	抛出异常	当前事务加入到外部事务中	Propagation.MANDATORY	不常用