

Spring数据源码环境搭建与核心类介绍

本章将讲解对 Spring 与数据源码环境搭建相关操作，包含 Spring Data Access 中的 spring-jdbc 模块和 spring-orm 模块的基础操作内容。

1.1 spring-jdbc 测试环境搭建

本节将对 spring-jdbc 测试环境的搭建进行说明，通过本节的操作将会得到一个 spring-jdbc 源码分析的测试环境。在本节中会对 spring-jdbc 的两种环境搭建进行说明：第一种是 SpringXML 模式；第二种是 Spring 注解模式。

1.1.1 spring-jdbc 基于 SpringXML 环境搭建

本节将搭建基于 SpringXML 模式的 spring-jdbc 环境，首先在 IDEA 中右击项目顶层，选择 New→Module 选项，如图 1.1 所示。

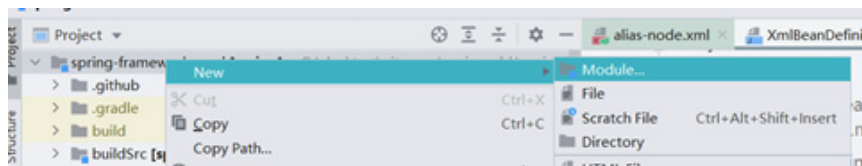


图 1.1 选择 Module 工程

选择 Module 选项后会弹出如图 1.2 所示的对话框，单击 Gradle 选项，勾选 Java 复选框，再单击 Next 按钮进入下一步。

单击 Next 按钮后弹出如图 1.3 所示的对话框，输入 Name、GroupId、ArtifactId 和 Version 这四项数据内容。

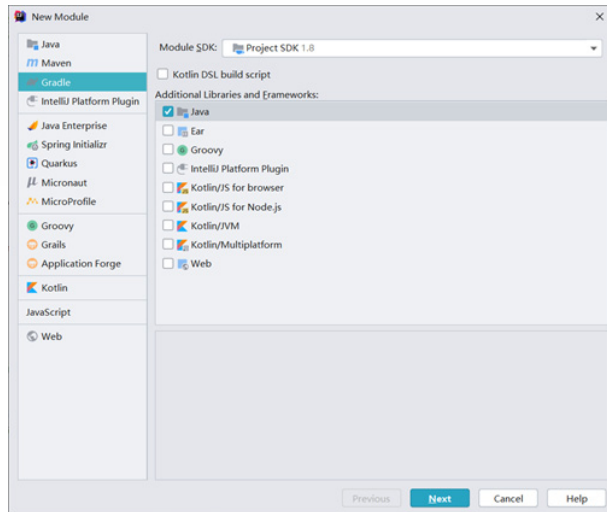


图 1.2 选择 Gradle 和 Java 工程

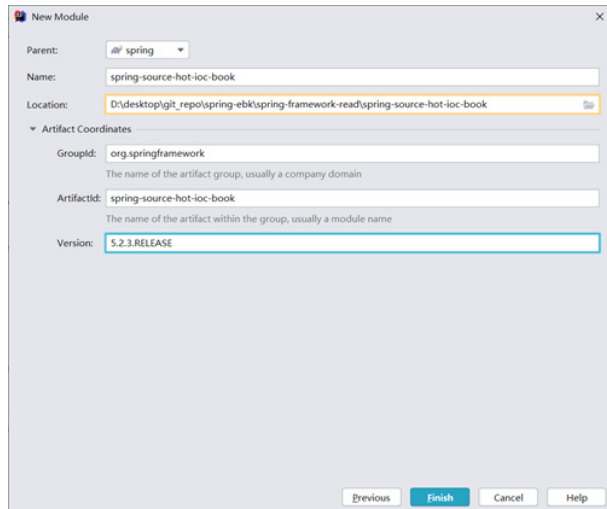


图 1.3 输入 Name 等属性

输入完成后单击 **Finish** 按钮完成工程创建。在工程创建完成以后需要对 `build.gradle` 文件进行修改，主要修改的是 `dependencies` 节点相关数据，向 `dependencies` 节点添加如下数据内容。

```
dependencies {
    compile(project(":spring-tx"))
    compile(project(":spring-jdbc"))
    compile(project(":spring-context"))
    compile(project(":spring-orm"))
    implementation 'mysql:mysql-connector-java:5.1.46'
    implementation 'org.apache.tomcat:tomcat-dbcp:9.0.1'
    implementation 'org.hibernate:hibernate-core:5.4.2.Final'
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.7.0'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.7.0'
}
```

在本章中所涉及的 `spring-jdbc` 和 `spring-orm` 环境都将基于该工程进行。在本节使用的是

SpringXML 模式搭建 spring-jdbc 测试环境，在 resources 文件夹下创建 jdbcTemplateConfiguration.xml 文件，并且向该文件中添加如下内容。

```
<?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.tomcat.dbcp.dbcp2.BasicDataSource">
        <property name="driverClassName" value=""/>
        <property name="url"
            value=""/>
        <property name="username" value=""/>
        <property name="password" value=""/>
    </bean>
    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource"/>
    </bean>
</beans>
```

在上述 SpringXML 配置文件中对 dataSource 和 jdbcTemplate 进行了自定义，当填写完上述内容之后需要填写一些自定义的配置，需要填写的内容有如下 4 项。

- (1) driverClassName: 表示数据库驱动名称。
- (2) url: 表示 JDBC 链接地址。
- (3) username: 表示数据库用户名。
- (4) password: 表示数据库密码。

读者根据自身环境配置完成上述四项参数信息后就可以编写具体的测试代码。创建一个 Java 类，类名为 TUserEntity，具体代码如下。

```
@Entity
@Table(name = "t_user")
public class TUserEntity {
    private Long id;

    private String name;
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name = "id", nullable = false)
    public Long getId() {
        return id;
    }

    @Basic
    @Column(name = "name", nullable = true, length = 255)
    public String getName() {
        return name;
    }
}
```

在完成 TUserEntity 类的编写操作后需要新建数据库表，关于 t_user 的建表语句如下。

```
CREATE TABLE t_user (
    id bigint(20) NOT NULL AUTO_INCREMENT,
    name varchar(255) DEFAULT NULL,
    PRIMARY KEY (id)
) ENGINE=MyISAM AUTO_INCREMENT=3 DEFAULT CHARSET=utf8mb4;
```

在完成 `t_user` 建表语句的编写后需要将其放在数据库中执行，本例中采用的是 MySQL 数据库，直接将其复制到 MySQL 命令行中执行即可。完成数据表的创建后需要向该表中添加数据，添加数据的 SQL 语句代码如下。

```
INSERT INTO t_user(id, name) VALUES (1, 'ac');
```

在完成数据表数据初始化后需要进行最后的 `JdbcTemplate` 使用，创建一个 Java 类，类名为 `JdbcTemplateWithXmlDemo`，该类具体代码如下。

```
public class JdbcTemplateWithXmlDemo {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext context =
            new ClassPathXmlApplicationContext("jdbcTemplateConfiguration.xml");
        JdbcTemplate bean = context.getBean(JdbcTemplate.class);
        List<TUserEntity> query = bean.query("select * from t_user", new
            RowMapper<TUserEntity>() {
                @Override
                public TUserEntity mapRow(ResultSet rs, int rowNum) throws SQLException {
                    long id = rs.getLong("id");
                    String name = rs.getString("name");
                    TUserEntity tUserEntity = new TUserEntity();
                    tUserEntity.setId(id);
                    tUserEntity.setName(name);
                    return tUserEntity;
                }
            });
        System.out.println();
    }
}
```

完成 `JdbcTemplateWithXmlDemo` 类的编写后需要将这个类启动，将断点放在 `System.out.println()` 这一行，查看 `query` 的数据信息，具体信息如图 1.4 所示。

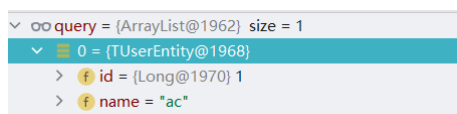


图 1.4 查询结果 1

从图 1.4 中可以发现，`query` 的数据信息和数据库中的数据信息相同，此时完成了 `spring-jdbc` 基于 `SpringXML` 模式的测试环境搭建全过程。

1.1.2 spring-jdbc 基于 Spring 注解模式环境搭建

本节将介绍 `spring-jdbc` 基于 `Spring` 注解模式的测试环境。首先创建一个 `Spring` 配置类，类名为 `SpringJdbcConfig`，向该类中添加如下代码。


```
@Configuration
@ComponentScan("com.github.source.hot.data.jdbc")
public class SpringJdbcConfig {
    @Bean
    public DataSource mysqlDataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("");
        dataSource.setUrl("");
        dataSource.setUsername("");
        dataSource.setPassword("");

        return dataSource;
    }

    @Bean
    public JdbcTemplate jdbcTemplate(
        DataSource mysqlDataSource
    ) {
        JdbcTemplate jdbcTemplate = new JdbcTemplate();
        jdbcTemplate.setDataSource(mysqlDataSource);
        return jdbcTemplate;
    }
}
```

在完成 SpringJdbcConfig 类的编写后需要根据当前所在环境修改下面 4 个方法的参数。

- (1) 方法 **setDriverClassName**: 设置数据库驱动名称。
- (2) 方法 **setUrl**: 设置数据库链接地址。
- (3) 方法 **setUsername**: 设置数据库用户名。
- (4) 方法 **setPassword**: 设置数据库密码。

读者根据自身环境配置完成上述 4 个方法的参数信息后就可以编写具体的测试代码。创建一个 Java 类, 类名为 JdbcTemplateWithAnnotationDemo, 具体代码如下。

```
public class JdbcTemplateWithAnnotationDemo {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext(SpringJdbcConfig.class);
        JdbcTemplate bean = context.getBean(JdbcTemplate.class);
        List<TUserEntity> query = bean.query("select * from t_user",
            new RowMapper<TUserEntity>() {
                @Override
                public TUserEntity mapRow(ResultSet rs, int rowNum) throws SQLException {
                    long id = rs.getLong("id");
                    String name = rs.getString("name");
                    TUserEntity tUserEntity = new TUserEntity();
                    tUserEntity.setId(id);
                    tUserEntity.setName(name);
                    return tUserEntity;
                }
            });
        System.out.println();
    }
}
```

```

    }

```

完成 `JdbcTemplateWithAnnotationDemo` 类的编写后需要将这个类启动, 将断点放在 `System.out.println()` 这一行, 查看 `query` 的数据信息, 具体信息如图 1.5 所示。

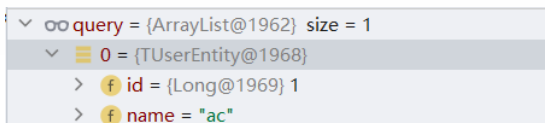


图 1.5 查询结果 2

从图 1.5 中可以发现, `query` 的数据信息和数据库中的数据信息相同, 此时完成了 `spring-jdbc` 基于 `Spring` 注解模式的测试环境搭建全过程。

1.2 spring-orm 测试环境搭建

在本节将对 `spring-orm` 测试环境的搭建进行说明, 通过本节将会得到一个 `spring-orm` 源码分析的测试环境。在本节中会对 `spring-orm` 的两种环境搭建进行说明: 第一种是 `Spring XML` 模式; 第二种是 `Spring` 注解模式。

1.2.1 spring-orm 基于 SpringXML 环境搭建

本节将介绍 `spring-orm` 基于 `SpringXML` 模式的测试环境。首先需要在 `resources` 文件夹下创建 `hibernate5Configuration.xml` 文件, 并且向该文件中添加如下内容。

```

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

    <bean id="sessionFactory"
        class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSource"/>
        <property name="packagesToScan" value="com.github.source.hot.data.model"/>
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.hbm2ddl.auto">
                    update
                </prop>
                <prop key="hibernate.dialect">
                    org.hibernate.dialect.MySQL5Dialect
                </prop>
                <prop key="hibernate.hbm2ddl.auto">
                    update
                </prop>
            </props>
        </property>
    </bean>

```

```
<bean id="dataSource"
      class="org.apache.tomcat.dbcp.dbcp2.BasicDataSource">
  <property name="driverClassName" value=""/>
  <property name="url"
            value=""/>
  <property name="username" value=""/>
  <property name="password" value=""/>
</bean>

<bean id="txManager"
      class="org.springframework.orm.hibernate5.HibernateTransactionManager">
  <property name="sessionFactory" ref="sessionFactory"/>
</bean>
</beans>
```

在本节中使用的是 Hibernate 框架作为 ORM 框架，在 hibernate5Configuration.xml 文件中关于 sessionFactory 的配置信息有如下内容。

(1) packagesToScan: 表示实体对象的扫描路径。

(2) hibernateProperties: 表示 hibernate 的属性值。

在 hibernate5Configuration.xml 文件中关于 hibernateProperties 属性做出了 3 个数据的设置。

(1) hibernate.hbm2ddl.auto: 表示启动时的操作。

(2) hibernate.dialect: 表示数据库方言，在本例中采用的是 MySQL5。数据库方言是指各个数据库厂商提供的 SQL。使用数据库方言的意义在于让 ORM 框架适应不同的厂商数据库，属于一个转译模块。

(3) hibernate.hibernate.hbm2ddl.auto: 表示启动时应该进行的操作。

上述第 1 个属性和第 3 个属性都表示启动时的操作，具体操作符有 4 种。

(1) create: 表示启动的时候先进行删除操作，再进行创建操作。

(2) create-drop: 表示创建，只不过在系统关闭前执行删除操作。

(3) update: 这个操作在启动时会检查 schema 是否一致，如果不一致会做 scheme 更新。

(4) validate: 在启动时验证现有 schema 与用户配置的 hibernate 是否一致，如果不一致就抛出异常，并不做更新。

本例中使用的是 update 操作，在按需填写完成 packagesToScan 属性和 hibernateProperties 属性后还需要对 dataSource 中的 ([相关信息进行配置，需要进行配置的内容有如下 4 项。

(1) driverClassName: 表示数据库驱动名称。

(2) url: 表示 JDBC 链接地址。

(3) username: 表示数据库用户名。

(4) password: 表示数据库密码。

按需填写完成上述 4 项配置后需要编写主体测试类，创建类名为 SpringOrmXMLDemo 的 Java 文件，向 SpringOrmXMLDemo 类中添加如下代码。

```
public class SpringOrmXMLDemo {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext context =
            new ClassPathXmlApplicationContext("hibernate5Configuration.xml");
        SessionFactory bean = context.getBean(SessionFactory.class);
        Session session = bean.openSession();
        TUserEntity tUserEntity = session.get(TUserEntity.class, 1L);
    }
}
```

```

        System.out.println();
    }
}

```

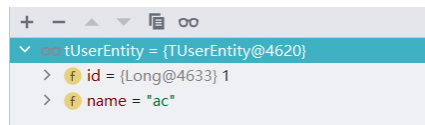


图 1.6 查询结果 3

编写完成 `SpringOrmXMLDemo` 类后将断点放在 `System.out.println()` 这一行，调试该项目查看 `tUserEntity` 变量的数据信息，具体信息如图 1.6 所示。

从图 1.6 中可以发现，`tUserEntity` 和数据库中的数据信息相同，至此 `spring-orm` 基于 `SpringXML` 模式环境搭建完成。

1.2.2 spring-orm 基于 Spring 注解模式环境搭建

本节将介绍 `spring-orm` 基于 `Spring` 注解模式的测试环境。首先需要编写一个 `Spring` 配置类，创建一个类，类名为 `HibernateConf`，具体代码如下。

```

@Configuration
@EnableTransactionManagement
public class HibernateConf {

    @Bean
    public LocalSessionFactoryBean sessionFactory() {
        LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();
        sessionFactory.setDataSource(dataSource());
        sessionFactory.setPackagesToScan(
            "");
        sessionFactory.setHibernateProperties(hibernateProperties());

        return sessionFactory;
    }

    @Bean
    public DataSource dataSource() {
        BasicDataSource dataSource = new BasicDataSource();
        dataSource.setDriverClassName("");
        dataSource.setUrl("");
        dataSource.setUsername("");
        dataSource.setPassword("");
        return dataSource;
    }

    @Bean
    public PlatformTransactionManager hibernateTransactionManager() {
        HibernateTransactionManager transactionManager
            = new HibernateTransactionManager();
        transactionManager.setSessionFactory(sessionFactory().getObject());
        return transactionManager;
    }

    private Properties hibernateProperties() {

```

```

        Properties hibernateProperties = new Properties();
        hibernateProperties.setProperty(
            "hibernate.hbm2ddl.auto", "update");
        hibernateProperties.setProperty(
            "hibernate.dialect", "org.hibernate.dialect.H2Dialect");
        hibernateProperties.setProperty(
            "hibernate.hbm2ddl.auto", "update");
        return hibernateProperties;
    }
}

```

在编写完成 `HibernateConf` 类后还需要进行自定义配置填写，需要修改的信息如下。

- (1) 修改方法 `setPackagesToScan` 的参数，将其设置为实体类所在的包路径。
- (2) 修改方法 `setDriverClassName` 的参数，将其设置为数据库驱动名称。
- (3) 修改方法 `setUrl` 的参数，将其设置为数据库链接地址。
- (4) 修改方法 `setUsername` 的参数，将其设置为数据库用户名。
- (5) 修改方法 `setPassword` 的参数，将其设置为数据库密码。
- (6) 修改 `hibernateProperties` 方法中的数据。

完成 `HibernateConf` 类中的自定义配置修改后需要进行启动类的编写，创建一个 Java 类，类名为 `SpringOrmAnnotationDemo`，具体代码如下。

```

@Configuration
@EnableTransactionManagement
@Import(HibernateConf.class)
public class SpringOrmAnnotationDemo {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext(SpringOrmAnnotationDemo.class);
        SessionFactory bean = context.getBean(SessionFactory.class);
        Session session = bean.openSession();
        TUserEntity tUserEntity = session.get(TUserEntity.class, 1L);
        System.out.println();
    }
}

```

编写完 `SpringOrmAnnotationDemo` 类后将断点放在 `System.out.println()` 这一行，调试该项目查看 `tUserEntity` 变量的数据信息，具体信息如图 1.7 所示。

从图 1.7 中可以发现，`tUserEntity` 和数据库中的数据信息相同，至此 `spring-orm` 基于 Spring 注解模式环境搭建完成。

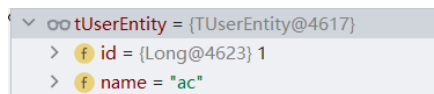


图 1.7 查询结果 4

1.3 Spring 数据操作中的核心类

本节将介绍 Spring 数据操作中的核心类。在前面关于测试环境搭建过程中所使用的类都可以视作核心类，例如在 `spring-jdbc` 环境搭建过程中使用的 `DriverManagerDataSource` 类和

JdbcTemplate 类等，下面将列出作者认为的核心类。

- (1) DriverManagerDataSource，它是一个基于 JDBC 的标准实现，作用是进行数据源管理。
- (2) JdbcTemplate，它是一个基于 JDBC 的实现，作用是简化传统 JDBC 的操作流程。
- (3) JdbcOperations，封装了 JDBC 的数据操作。
- (4) ResultSetExtractor，它是 JdbcTemplate 查询方法使用时的回调接口，此接口的实现从 ResultSet 中获取实际数据。接口的作用是提取数据。
- (5) RowCallbackHandler，它用于一行一行地处理 ResultSet 数据。
- (6) RowMapper，用于按行进行映射处理，通过 ResultSet 进行实际类转换。
- (7) SqlProvider，用于提取 SQL 语句。
- (8) HibernateOperations，基于 Hibernate 进行封装的操作 API 接口，定义了数据库操作接口。
- (9) SessionHolder，用于持有 Hibernate 中的 session 对象的对象。
- (10) TransactionManager 事务管理器接口，子类有 PlatformTransactionManager 和 ReactiveTransactionManager，在非响应式编程的情况下常用的是 PlatformTransactionManager。
- (11) TransactionExecution 事务状态接口，子类有 TransactionStatus 和 ReactiveTransaction，前者适用于非响应式编程，后者适用于响应式编程。

1.4 总结

在本章讲述了 spring-jdbc 和 spring-orm 的基本环境搭建，主要围绕 SpringXML 模式和 Spring 注解模式进行，此外介绍了 Spring 数据操作中的核心类。

JdbcTemplate类分析

本章将对 JdbcTemplate 类进行分析。JdbcTemplate 类是一个基于 JDBC 进行封装的类，常见的 JDBC 操作有新增、修改、删除和查询，本章将围绕这 4 个操作进行分析。

2.1 初识 JdbcTemplate 类

在本节将对 JdbcTemplate 类做一个简单的介绍，首先需要查看 JdbcTemplate 的类图，详细信息如图 2.1 所示。

在图 2.1 中可以发现，JdbcTemplate 类实现了两个接口 JdbcOperations 和 InitializingBean，在 spring-jdbc 中 JdbcOperations 接口定义了 JDBC 相关操作，按照操作行为分类可以分为下面 3 类。

- (1) 执行类，以 "execute" 字符串开头的方法。
- (2) 查询类，以 "query" 字符串开头的方法。
- (3) 更新类，包含 "update" 字符串的方法，在 JdbcOperations 中有批量提交的方法 batchUpdate。

关于 JdbcOperations 接口的实现会在本章的后续部分进行分析，在 JdbcTemplate 类中还实现了 InitializingBean，需要在 JdbcTemplate 的类图中找到具体的实现方法，实现方法在 JdbcAccessor 中，具体代码如下。

```
@Override
public void afterPropertiesSet() {
    if (getDataSource() == null) {
        throw new IllegalArgumentException("Property 'dataSource' is required");
    }
    if (!isLazyInit()) {
        getExceptionTranslator();
    }
}
```

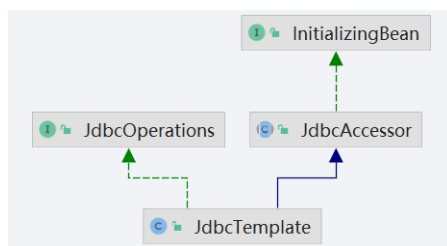


图 2.1 JdbcTemplate 类图

在上述代码中主要处理流程如下。

- (1) 判断数据源 (**DataSource**) 对象是否为空, 如果数据源对象为空则会抛出异常。
 - (2) 判断是不是懒加载的, 如果不是懒加载的会获取 **SQLExceptionTranslator** 对象。
- 在上述两个操作流程中, 需要关注 **getExceptionTranslator** 方法, 具体处理代码如下。

```
public SQLExceptionTranslator getExceptionTranslator() {
    SQLExceptionTranslator exceptionTranslator = this.exceptionTranslator;
    if (exceptionTranslator != null) {
        return exceptionTranslator;
    }
    synchronized(this) {
        exceptionTranslator = this.exceptionTranslator;
        if (exceptionTranslator == null) {
            DataSource dataSource = getDataSource();
            if (dataSource != null) {
                exceptionTranslator =
                    new SQLErrorCodeSQLExceptionTranslator(dataSource);
            }
            else {
                exceptionTranslator = new SQLStateSQLExceptionTranslator();
            }
            this.exceptionTranslator = exceptionTranslator;
        }
        return exceptionTranslator;
    }
}
```

这段代码的主要目的是将 **exceptionTranslator** 成员变量进行初始化, 从上述代码中可以发现, 初始化方法有两种方式。

- (1) 当数据源对象存在时, 通过 **SQLErrorCodeSQLExceptionTranslator** 的构造方法+数据源对象进行创建。

- (2) 当数据源对象为空时, 通过 **SQLStateSQLExceptionTranslator** 的构造方法进行创建。

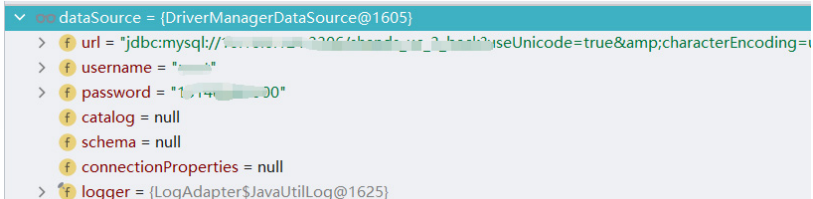
2.1.1 DataSource 分析

在 **afterPropertiesSet** 处理方法中提到了一个接口 **DataSource**, **DataSource** 是 Java 提供的接口, 具体包路径是 **javax.sql.DataSource**, 在 **spring-jdbc** 中由 **DriverManagerDataSource** 实现, 关于数据源可以简单地理解为存储了数据库账号、数据库密码和数据库地址的存储对象。在前面搭建的测试用例中关于数据源的定义代码如下。

```
@Bean
public DataSource mysqlDataSource() {
    DriverManagerDataSource dataSource = new DriverManagerDataSource();
    dataSource.setDriverClassName("com.mysql.jdbc.Driver");
    dataSource.setUrl("");
    dataSource.setUsername("");
    dataSource.setPassword("");

    return dataSource;
}
```


在这段代码中定义的 `DataSource` 对象信息如图 2.2 所示。



```
> dataSource = {DriverManagerDataSource@1605}
> f url = "jdbc:mysql://localhost:3306/test?useUnicode=true&characterEncoding=utf8"
> f username = "root"
> f password = "1234567890"
> f catalog = null
> f schema = null
> f connectionProperties = null
> f logger = {LogAdapter$JavaUtilLog@1625}
```

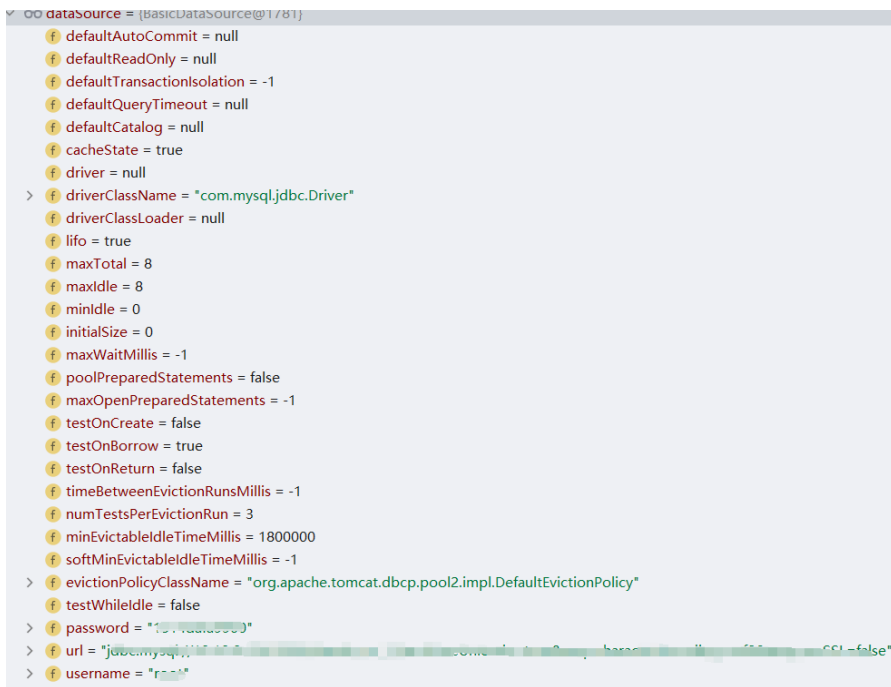
注：在实际开发过程中，链接地址、账户和密码会因开发者的配置不同而不同，具体由开发者配置决定。由于作者采用的是一个线上环境的数据库，因此书中将链接地址、账户和密码信息隐藏。

图 2.2 `DataSource` 对象信息

通过图 2.2 可以发现，通过 `set` 方法进行设置的数据已经完成设置。在 `mysqlDataSource` 方法中所使用的 `DataSource` 实现类是 `spring-jdbc` 自带的，在第 1 章中搭建的测试用例还引入了 `tomcat-dbcp` 相关依赖，并对其进行了使用，关于 `tomcat-dbcp` 中的数据源配置代码如下。

```
@Bean
public DataSource dataSource() {
    BasicDataSource dataSource = new BasicDataSource();
    dataSource.setDriverClassName("com.mysql.jdbc.Driver");
    dataSource.setUrl("");
    dataSource.setUsername("");
    dataSource.setPassword("");
    return dataSource;
}
```

在这个方法中所得到的数据源对象信息如图 2.3 所示。



```
> dataSource = {BasicDataSource@1781}
> f defaultAutoCommit = true
> f defaultReadOnly = null
> f defaultTransactionIsolation = -1
> f defaultQueryTimeout = null
> f defaultCatalog = null
> f cacheState = true
> f driver = null
> f driverClassName = "com.mysql.jdbc.Driver"
> f driverClassLoader = null
> f lifo = true
> f maxTotal = 8
> f maxIdle = 8
> f minIdle = 0
> f initialSize = 0
> f maxWaitMillis = -1
> f poolPreparedStatements = false
> f maxOpenPreparedStatements = -1
> f testOnCreate = false
> f testOnBorrow = true
> f testOnReturn = false
> f timeBetweenEvictionRunsMillis = -1
> f numTestsPerEvictionRun = 3
> f minEvictableIdleTimeMillis = 1800000
> f softMinEvictableIdleTimeMillis = -1
> f evictionPolicyClassName = "org.apache.tomcat.dbcp.pool2.impl.DefaultEvictionPolicy"
> f testWhileIdle = false
> f password = "1234567890"
> f url = "jdbc:mysql://localhost:3306/test?useUnicode=true&characterEncoding=utf8"
> f username = "root"
```

图 2.3 `BasicDataSource` 对象信息

从图 2.3 中可以发现，基本的四个元素（数据库链接、数据库账号、数据库密码和数据库驱动）都是存在的，相比 `spring-jdbc` 提供的数据库源对象，`tomcat-dbcp` 提供的数据库源对象有更多属性，有兴趣的读者可以进入源码查看具体作用。

2.1.2 JdbcTemplate 的初始化

在前面对 `DataSource` 进行了相关分析，接下来将进入 `JdbcTemplate` 的初始化分析。通常对于 `JdbcTemplate` 初始化是指进行数据库源对象的设置，在第 1 章的测试用例中可以看到两种设置方式。

(1) 通过 `JavaBean` 进行设置。

(2) 通过 `SpringXML` 配置进行设置。

下面以通过 `JavaBean` 进行设置作为分析目标，具体处理代码如下。

```
@Bean
public JdbcTemplate jdbcTemplate(
    DataSource mysqlDataSource
) {
    JdbcTemplate jdbcTemplate = new JdbcTemplate();
    jdbcTemplate.setDataSource(mysqlDataSource);
    return jdbcTemplate;
}
```

在这段代码中所得到的 `JdbcTemplate` 对象信息如图 2.4 所示。

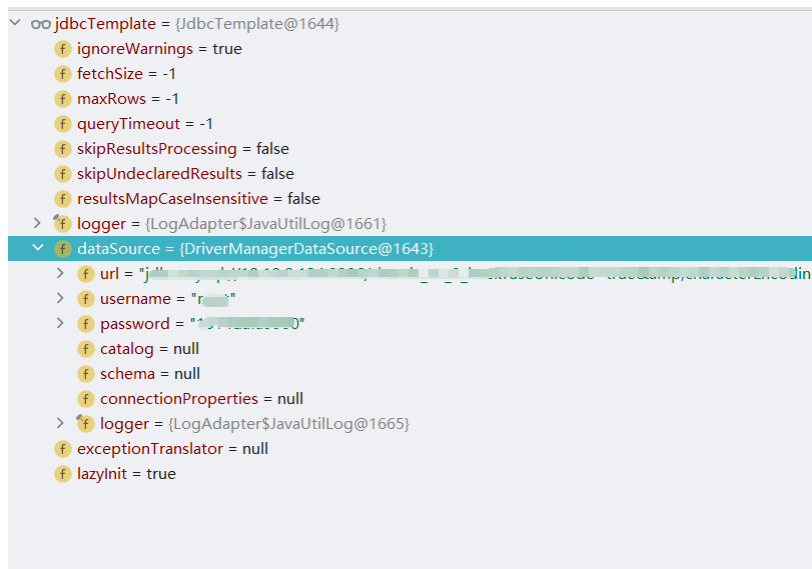


图 2.4 JdbcTemplate 对象信息

在图 2.4 中可以发现数据库源对象被成功设置，此外还有一些默认值在 `JdbcTemplate` 对象中。在图 2.4 中需要注意 `exceptionTranslator` 对象为 `null`，同时还需要注意 `lazyInit` 变量信息为 `true`。再继续向下执行代码，由于此时的数据库源对象不为空，因此不会出现 `IllegalArgumentException` 异常信息。由于此时 `lazyInit` 变量为 `true`，因此不会将 `exceptionTranslator` 变量信息进行初始化。如果需要查看 `exceptionTranslator` 变量的初始化操作，则需要将代码进行修改，修改后代码如下。

```

@Bean
public JdbcTemplate jdbcTemplate(
    DataSource mysqlDataSource
) {
    JdbcTemplate jdbcTemplate = new JdbcTemplate();
    jdbcTemplate.setDataSource(mysqlDataSource);
    jdbcTemplate.setLazyInit(false);
    return jdbcTemplate;
}

```

经过修改后将断点放在 `JdbcAccessor` 对象中的 `afterPropertiesSet` 方法上就可以查看到 `exceptionTranslator` 的数据信息，具体信息如图 2.5 所示。

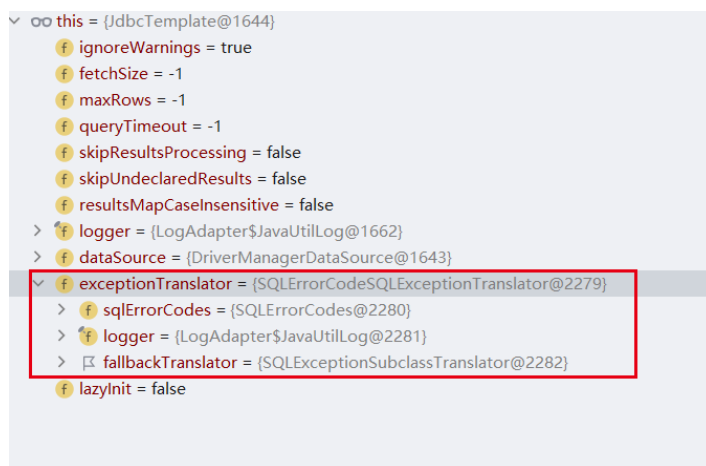


图 2.5 修改后 `exceptionTranslator` 的数据信息

2.2 JdbcTemplate 中的执行操作分析

本节将对 `JdbcTemplate` 中的执行操作进行分析。在 `JdbcTemplate` 中关于执行操作可以使用 `execute` 方法，比如可以使用 “`jdbcTemplate.execute("select * from t_user");`”，在 `JdbcTemplate` 类中使用 `execute` 方法是最常见的，这段代码对应的源码内容如下。

```

@Override
public void execute(final String sql) throws DataAccessException {
    class ExecuteStatementCallback implements StatementCallback<Object>, SqlProvider {
        @Override
        @Nullable
        public Object doInStatement(Statement stmt) throws SQLException {
            stmt.execute(sql);
            return null;
        }
    }
    @Override
    public String getSql() {
        return sql;
    }
}

execute(new ExecuteStatementCallback());

```

```
}
```

在这段代码中可以看到关于执行的方法调用会寻找另一个 `execute` 方法，在分析最终的 `execute` 方法前需要先对 `ExecuteStatementCallback` 对象进行分析。从上述代码中可以发现 `ExecuteStatementCallback` 实现了两个接口：`StatementCallback` 接口和 `SqlProvider` 接口。`StatementCallback` 接口的作用是通过 `Statement` 调用 `execute` 方法执行 SQL 语句，`SqlProvider` 接口的作用是获取当前的 SQL 语句。下面查看最终的 `execute` 方法，具体代码如下。

```
@Override
@Nullable
public <T> T execute(StatementCallback<T> action) throws DataAccessException {

    // 获取链接对象
    Connection con = DataSourceUtils.getConnection(observeOnDataSource());
    Statement stmt = null;
    try {
        stmt = con.createStatement();
        // 应用 Statement 配置
        applyStatementSettings(stmt);
        // 执行
        T result = action.doInStatement(stmt);
        // 处理警告信息
        handleWarnings(stmt);
        return result;
    }
    catch (SQLException ex) {
        // 获取 SQL 语句
        String sql = getSql(action);
        // 关闭 Statement 对象
        JdbcUtils.closeStatement(stmt);
        stmt = null;
        // 释放链接
        DataSourceUtils.releaseConnection(con, getDataSource());
        con = null;
        // 抛出异常
        throw translateException("StatementCallback", sql, ex);
    }
    finally {
        // 关闭 Statement 对象
        JdbcUtils.closeStatement(stmt);
        // 释放链接
        DataSourceUtils.releaseConnection(con, getDataSource());
    }
}
```

上述代码中主要处理逻辑如下。

- (1) 获取数据库链接对象。
- (2) 从数据库链接对象中获取 `Statement` 对象。
- (3) 应用 `Statement` 的配置数据。
- (4) 执行 `Statement` 中的 SQL 语句。
- (5) 处理警告信息。

(6) 将第(4)步中得到的结果返回。

(7) 关闭 **Statement** 对象。

(8) 释放链接对象。

上述8个操作步骤是正常处理过程中需要进行的操作，其中操作步骤(4)就是和数据库进行直接交互的操作（执行 SQL 语句）。当遇到异常时会进行下面4个步骤的操作。

(1) 获取 SQL 语句。

(2) 关闭 **Statement** 对象。

(3) 释放链接对象。

(4) 抛出异常。

下面将对上述提到的一些操作流程进行细节分析。

2.2.1 获取数据库链接对象

本节将对获取数据库链接对象进行分析，处理代码如下。

```
Connection con = DataSourceUtils.getConnection(observeOnDataSource());
```

在这段代码中可以看到具体负责获取数据库链接对象的类是 **DataSourceUtils**，在获取数据库链接对象时需要传递参数，参数为数据源对象（**DataSource**），数据源对象的获取要依靠 **observeOnDataSource** 方法，具体代码如下。

```
protected DataSource obtainDataSource() {  
    DataSource dataSource = getDataSource();  
    Assert.state(dataSource != null, "No DataSource set");  
    return dataSource;  
}
```

在这段代码中会从成员变量中获取数据源对象，如果数据源对象为空将抛出异常。在得到数据源对象后就会直接调用 **getConnection** 方法，具体处理代码如下。

```
public static Connection getConnection(DataSource dataSource)  
    throws CannotGetJdbcConnectionException {  
    try {  
        // 执行获取链接的方法  
        return doGetConnection(dataSource);  
    } catch (SQLException ex) {  
        throw new CannotGetJdbcConnectionException("Failed to obtain JDBC Connection", ex);  
    } catch (IllegalStateException ex) {  
        throw new CannotGetJdbcConnectionException(  
            "Failed to obtain JDBC Connection: " + ex.getMessage());  
    }  
}
```

在这段代码中可以看到具体的获取方法是由 **doGetConnection** 提供的，该方法就是获取数据库链接对象的核心方法，具体代码如下。

```
public static Connection doGetConnection(DataSource dataSource) throws SQLException {  
    Assert.notNull(dataSource, "No DataSource specified");  
  
    // 事务同步管理器中获取链接持有器对象
```

```

        ConnectionHolder conHolder =
            (ConnectionHolder) TransactionSynchronizationManager
                .getResource(dataSource);
        // 从链接持有器中获取链接对象
        if (conHolder != null && (conHolder.hasConnection() || conHolder
            .isSynchronizedWithTransaction())) {
            conHolder.requested();
            if (!conHolder.hasConnection()) {
                logger.debug("Fetching resumed JDBC Connection from DataSource");
                conHolder.setConnection(fetchConnection(dataSource));
            }
            return conHolder.getConnection();
        }

        logger.debug("Fetching JDBC Connection from DataSource");
        // 从数据源中获取链接对象
        Connection con = fetchConnection(dataSource);

        // 判断是否处于同步状态
        if (TransactionSynchronizationManager.isSynchronizationActive()) {
            try {
                ConnectionHolder holderToUse = conHolder;
                // 如果 holderToUse 对象为空将进行初始化操作
                if (holderToUse == null) {
                    holderToUse = new ConnectionHolder(con);
                }
                // 不为空将对其进行链接对象设置
                else {
                    holderToUse.setConnection(con);
                }
                // 请求数量加 1
                holderToUse.requested();
                // 注册 ConnectionSynchronization 对象
                TransactionSynchronizationManager.registerSynchronization(
                    new ConnectionSynchronization(holderToUse, dataSource));
                // 设置事务同步标记为 true
                holderToUse.setSynchronizedWithTransaction(true);
                if (holderToUse != conHolder) {
                    // 资源绑定
                    TransactionSynchronizationManager.bindResource(dataSource, holderToUse);
                }
            } catch (RuntimeException ex) {
                // Unexpected exception from external delegation call -> close Connection and rethrow
                // 释放链接
                releaseConnection(con, dataSource);
                throw ex;
            }
        }

        return con;
    }
}

```

在上述代码中关于数据库链接对象的获取提供了两种方式。

- (1) 从链接持有器对象中获取数据库链接对象。
- (2) 从零开始创建一个数据库链接对象。

下面对第一种获取方式进行细节说明。在 `TransactionSynchronizationManager` 对象中通过数据源对象获取链接持有器 (`ConnectionHolder`)，当链接持有器不为空并且满足如下两种情况中的一种时就从链接持有器中获取数据库链接对象，两种情况具体如下。

- (1) 链接持有器中存在数据库链接对象。
- (2) 链接持有器中的 `synchronizedWithTransaction` 变量为 `true`。

变量 `synchronizedWithTransaction` 表示资源是否和事务是同步的，默认值为 `false`。在通过上述两个条件判断后在链接持有器中还可能没有数据库链接对象，此时需要进行手动创建，创建方法是 `fetchConnection`。通过 `setConnection` 方法和 `fetchConnection` 方法创建后，链接持有器就具备了数据库链接对象，最终可以通过 `conHolder.getConnection()` 获取并返回数据库链接对象。接下来对第二种获取方式进行分析，第二种方式最开始会执行 `fetchConnection(dataSource)` 方法将数据库链接对象提前准备好，在准备完成后会通过 `TransactionSynchronizationManager.isSynchronizationActive` 方法判断是否需要进行额外处理，该方法用于判断当前线程中的事务同步状态是否处于激活状态，如果该方法的判断结果为 `false` 则直接返回数据库链接对象，如果该方法的判断结果为 `true` 则需要进行下面的操作。

- (1) 判断从事务同步管理器中获取的链接持有器对象是否存在，如果不存在则需要通过 `ConnectionHolder` 构造方法和数据库链接对象进行创建，如果存在则需要将数据库链接对象设置给链接持有器。
- (2) 注册 `ConnectionSynchronization` 对象。
- (3) 将数据源对象和链接持有器进行绑定。

2.2.2 应用 Statement 的配置数据

本节将对应用 `Statement` 的配置数据进行源码分析，具体处理代码如下。

```
protected void applyStatementSettings(Statement stmt) throws SQLException {
    int fetchSize = getFetchSize();
    if (fetchSize != -1) {
        stmt.setFetchSize(fetchSize);
    }
    int maxRows = getMaxRows();
    if (maxRows != -1) {
        stmt.setMaxRows(maxRows);
    }
    DataSourceUtils.applyTimeout(stmt, getDataSource(), getQueryTimeout());
}
```

在上述代码中会涉及下面 3 个数据。

- (1) `fetchSize`，该变量表示调用 `rs.next` 时，`ResultSet` 会一次性从数据库中取得多少行数据。
- (2) `maxRows`，该变量表示 `Statement` 对象生成的所有 `ResultSet` 对象可以包含的最大行数，设置为给定数。
- (3) `queryTimeout`，该变量表示执行超时时间。

在上述 3 个变量设置中, 关于 `queryTimeOut` 的数据设置需要通过 `applyTimeout` 方法进行处理, 关于 `applyTimeout` 的代码如下。

```
public static void applyTimeout(Statement stmt, @Nullable DataSource dataSource, int
timeout)
    throws SQLException {
    Assert.notNull(stmt, "No Statement specified");
    ConnectionHolder holder = null;
    if (dataSource != null) {
        holder =
            (ConnectionHolder) TransactionSynchronizationManager.getResource(dataSource);
    }
    if (holder != null && holder.hasTimeout()) {
        stmt.setQueryTimeout(holder.getTimeToLiveInSeconds());
    } else if (timeout >= 0) {
        stmt.setQueryTimeout(timeout);
    }
}
```

在 `applyTimeout` 方法中关于 `queryTimeOut` 数据的来源有两个。

- (1) 链接持有器中的 `deadline` 变量。
- (2) 方法参数 `timeout`。

2.2.3 处理警告信息

本节将对处理警告信息进行源码分析, 负责处理警告信息的代码如下。

```
protected void handleWarnings(Statement stmt) throws SQLException {
    // 是否需要忽略警告信息
    if (isIgnoreWarnings()) {
        if (logger.isDebugEnabled()) {
            SQLWarning warningToLog = stmt.getWarnings();
            // 通过日志将警告信息输出
            while (warningToLog != null) {
                logger.debug("SQLWarning ignored: SQL state '" +
                    warningToLog.getSQLState() + "', error code '" +
                        warningToLog.getErrorCode() + "', message [" +
                            warningToLog.getMessage() + "];");
                warningToLog = warningToLog.getNextWarning();
            }
        }
    }
    else {
        // 抛出异常
        handleWarnings(stmt.getWarnings());
    }
}
```

在这段代码中会根据 `ignoreWarnings` 变量进行两种处理, 变量 `ignoreWarnings` 表示是否忽略警告。当不忽略警告时会执行 `handleWarnings` 方法, 具体处理代码如下。


```
protected void handleWarnings(@Nullable SQLWarning warning) throws
SQLException {
    if (warning != null) {
        throw new SQLException("Warning not ignored", warning);
    }
}
```

在这段代码中可以看到，当异常信息不为空时会抛出 `SQLException` 异常，当忽略警告时会进行日志输出。注意，这里输出的日志级别是 `debug` 级别并非 `warn` 级别。

2.2.4 释放链接对象

本节将对释放链接对象进行源码分析，负责释放链接对象的代码如下。

```
public static void releaseConnection(@Nullable Connection con,
    @Nullable DataSource dataSource) {
    try {
        doReleaseConnection(con, dataSource);
    } catch (SQLException ex) {
        logger.debug("Could not close JDBC Connection", ex);
    } catch (Throwable ex) {
        logger.debug("Unexpected exception on closing JDBC Connection", ex);
    }
}
```

在上述代码中最终的调用方法是 `doReleaseConnection`，具体处理代码如下。

```
public static void doReleaseConnection(@Nullable Connection con,
    @Nullable DataSource dataSource) throws SQLException {
    if (con == null) {
        return;
    }
    if (dataSource != null) {
        ConnectionHolder conHolder = (ConnectionHolder)
            TransactionSynchronizationManager
                .getResource(dataSource);
        if (conHolder != null && connectionEquals(conHolder, con)) {
            // 释放计数器减1
            conHolder.released();
            return;
        }
    }
    // 关闭链接
    doCloseConnection(con, dataSource);
}
```

在上述代码的处理过程中会有 3 种处理情况。

- (1) 数据库链接对象为空，不做处理直接返回。
- (2) 数据源对象不为空，从 `TransactionSynchronizationManager` 对象中根据数据源对象获取链接持有器，通过链接持有器的 `released` 方法将链接数减 1 后结束操作。
- (3) 进行数据库链接对象关闭操作。

2.2.5 配合 PreparedStatementCreator 和 PreparedStatementCallback 的执行操作

本节将对 execute 方法族中 PreparedStatementCreator 和 PreparedStatementCallback 的执行操作进行分析，具体处理代码如下。

```

@Override
@Nullable
public <T> T execute(PreparedStatementCreator psc, PreparedStatementCallback<T>
action)
    throws DataAccessException {

    Assert.notNull(psc, "PreparedStatementCreator must not be null");
    Assert.notNull(action, "Callback object must not be null");
    if (logger.isDebugEnabled()) {
        String sql = getSql(psc);
        logger.debug("Executing prepared SQL statement" + (sql != null ? " [" + sql +
"]" : ""));
    }

    // 获取数据库链接对象
    Connection con = DataSourceUtils.getConnection(observeOnDataSource());
    PreparedStatement ps = null;
    try {
        // 创建 PreparedStatement 对象
        ps = psc.createPreparedStatement(con);
        // 应用 PreparedStatement 属性
        applyStatementSettings(ps);
        // 通过 PreparedStatementCallback 执行 SQL 语句
        T result = action.doInPreparedStatement(ps);
        // 处理警告信息
        handleWarnings(ps);
        // 返回值处理
        return result;
    }
    catch (SQLException ex) {
        if (psc instanceof ParameterDisposer) {
            ((ParameterDisposer) psc).cleanupParameters();
        }
        // 获取 SQL 语句
        String sql = getSql(psc);
        psc = null;
        // 关闭 PreparedStatement 对象
        JdbcUtils.closeStatement(ps);
        ps = null;
        // 释放数据库链接
        DataSourceUtils.releaseConnection(con, getDataSource());
        con = null;
        // 抛出异常
        throw translateException("PreparedStatementCallback", sql, ex);
    }
    finally {

```

```

        if (psc instanceof ParameterDisposer) {
            // 清理参数
            ((ParameterDisposer) psc).cleanupParameters();
        }
        // 关闭 Statement
        JdbcUtils.closeStatement(ps);
        // 释放链接
        DataSourceUtils.releaseConnection(con, getDataSource());
    }
}

```

在这段代码中主要处理操作如下。

(1) 对参数 `PreparedStatementCreator` 和 `PreparedStatementCallback` 进行非空判断，如果是空将抛出异常。

(2) 获取数据库链接对象。

(3) 通过参数 `PreparedStatementCreator` 创建 `PreparedStatement` 对象。

(4) 应用 `PreparedStatement` 相关参数信息。共有 3 个参数信息需要初始化，分别是 `fetchSize`、`maxRows` 和 `timeOut`。

(5) 执行 `PreparedStatementCallback` 中的处理方法。

(6) 处理警告信息。

(7) 将第 (5) 步中得到的数据作为返回值返回。

(8) 如果 `PreparedStatementCreator` 类型是 `ParameterDisposer` 则需要进行参数清理操作。

(9) 关闭 `Statement`。

(10) 释放数据库链接对象。

上述 10 个操作为正常处理流程中的处理，但是在程序执行过程中难免出现异常，在这个方法中当出现 `SQLException` 异常时将会进行如下操作。

(1) 如果 `PreparedStatementCreator` 类型是 `ParameterDisposer` 则需要进行参数清理操作。

(2) 从 `PreparedStatementCreator` 中获取 SQL 语句。

(3) 关闭 `Statement`。

(4) 释放数据库链接对象。

(5) 抛出异常。

在这个执行方法中可以发现，最终与数据库交互的对象是 `PreparedStatementCallback` 接口，具体是由 `doInPreparedStatement` 方法负责处理的。

2.2.6 配合 ConnectionCallback 的执行操作

本节将分析配合 `ConnectionCallback` 的执行操作代码，具体处理代码如下。

```

@Override
@Nullable
public <T> T execute(ConnectionCallback<T> action) throws DataAccessException {
    Assert.notNull(action, "Callback object must not be null");

    // 获取数据库链接对象
    Connection con = DataSourceUtils.getConnection(observeOnDataSource());
}

```

```

    try {
        // 创建代理链接对象
        Connection conToUse = createConnectionProxy(con);
        // 通过 action 对象对数据进行处理
        return action.doInConnection(conToUse);
    }
    catch (SQLException ex) {
        // 获取 SQL 语句
        String sql = getSql(action);
        // 释放数据库链接对象
        DataSourceUtils.releaseConnection(con, getDataSource());
        con = null;
        // 抛出异常
        throw translateException("ConnectionCallback", sql, ex);
    }
    finally {
        // 释放数据库链接对象
        DataSourceUtils.releaseConnection(con, getDataSource());
    }
}

```

在这个方法中主要处理流程如下。

- (1) 对参数 `ConnectionCallback` 进行非空判断，如果为空则抛出异常。
- (2) 获取数据库链接对象。
- (3) 创建数据库链接对象的代理对象。
- (4) 通过参数 `ConnectionCallback` 和数据库代理链接对象进行数据库交互。
- (5) 释放数据库链接对象。

上述 5 个处理操作为基本操作，但是在实际处理过程中会出现异常，当出现异常时会进行如下操作。

- (1) 从参数 `ConnectionCallback` 中获取 SQL 语句。
- (2) 释放数据库链接对象。
- (3) 抛出异常。

2.3 JdbcTemplate 中的查询操作分析

本节将对 `JdbcTemplate` 中的查询操作进行分析，在第 1 章中所搭建的用例中有关于 `JdbcTemplate` 的查询使用例子，在当时所使用的方法签名为 `org.springframework.jdbc.core.JdbcTemplate#query(java.lang.String, org.springframework.jdbc.core.RowMapper<T>)`，本节将从该方法开始对 `JdbcTemplate` 中的一些查询操作进行源码分析，下面是 `org.springframework.jdbc.core.JdbcTemplate#query(java.lang.String, org.springframework.jdbc.core.RowMapper<T>)` 的源码。

```

@Override
public <T> List<T> query(String sql, RowMapper<T> rowMapper) throws
    DataAccessException {
    return result(query(sql, new RowMapperResultSetExtractor<>(rowMapper)));
}

```

在这段代码中需要分为3个方法进行分析：

- (1) result 方法。
 - (2) query 方法。
 - (3) RowMapperResultSetExtractor 构造方法。
- 上述3个方法就是本节源码分析的主要目标对象。

2.3.1 RowMapperResultSetExtractor 对象分析

在 query 方法的代码中可以看到 RowMapperResultSetExtractor 对象的创建需要使用 RowMapper 对象，在 RowMapperResultSetExtractor 的成员变量中也存在该对象，下面关注 RowMapper 的作用，RowMapper 可以将 ResultSet 对象转换为一个实体对象，围绕这个功能在 RowMapperResultSetExtractor 对象中定义了 extractData 方法，具体代码如下。

```
@Override
public List<T> extractData(ResultSet rs) throws SQLException {
    List<T> results = (this.rowsExpected > 0 ? new ArrayList<>(this.rowsExpected) : new
    ArrayList<>());
    int rowNum = 0;
    while (rs.next()) {
        results.add(this.rowMapper.mapRow(rs, rowNum++));
    }
    return results;
}
```

在上述代码中主要处理操作如下。

- (1) 创建存储容器。
- (2) 通过 ResultSet.next 方法判断 ResultSet 是否还有下一个数据，如果存在下一个数据则通过成员变量 rowMapper 进行转换，将转换后的数据放入第(1)步中的存储容器。

综上所述，RowMapperResultSetExtractor 的核心作用有如下两个。

- (1) 存储 RowMapper 对象。
- (2) 提供 extractData 方法，配合 RowMapper 从 ResultSet 对象中获取数据。

2.3.2 query 方法分析

下面将对 query 方法进行分析，query 方法的处理代码如下。

```
@Override
@Nullable
public <T> T query(final String sql, final ResultSetExtractor<T> rse) throws
DataAccessException {

    class QueryStatementCallback implements StatementCallback<T>, SqlProvider {
        @Override
        @Nullable
        public T doInStatement(Statement stmt) throws SQLException {
            ResultSet rs = null;
            try {
```

```

        rs = stmt.executeQuery(sql);
        return rse.extractData(rs);
    }
    finally {
        JdbcUtils.closeResultSet(rs);
    }
}

@Override
public String getSql() {
    return sql;
}
}

return execute(new QueryStatementCallback());
}

```

通过对 `JdbcTemplate` 执行操作的源码分析可以知道负责进行交互的方法是 `execute`，参数是 `StatementCallback`，在 `query` 方法中 `StatementCallback` 接口的实现类是 `QueryStatementCallback` 类，在 `QueryStatementCallback` 类中对 `doInStatement` 的实现逻辑如下。

- (1) 通过 `Statement` 对象将 SQL 语句提交，得到返回值 `ResultSet` 对象。
- (2) 将第 (1) 步中得到的 `ResultSet` 对象通过 `ResultSetExtractor` 接口提供的 `extractData` 方法进行处理得到返回值。

前面配合的是 `RowMapperResultSetExtractor` 对象，在该方法中参数 `rse` 就是 `RowMapperResultSetExtractor` 对象，具体的 `extractData` 方法就会执行 `RowMapperResultSetExtractor` 中的处理逻辑。

2.3.3 RowMapper 接口分析

在 `query` 方法中可以发现重度依赖参数 `ResultSetExtractor`，而参数 `ResultSetExtractor` 又需要依赖于 `RowMapper` 接口，因此对于 `RowMapper` 接口的分析很重要。`RowMapper` 接口的作用是从 `ResultSet` 对象中提取数据，这个提取数据操作需要通过实现 `mapRow` 方法来自定义。在 `spring-jdbc` 中提供的实现有如下 3 个。

- (1) `SingleColumnRowMapper`。
- (2) `BeanPropertyRowMapper`。
- (3) `ColumnMapRowMapper`。

接下来将对 `RowMapper` 的上述 3 个实现进行说明。

1. SingleColumnRowMapper 对象分析

下面将对 `SingleColumnRowMapper` 对象进行分析，由于对象 `SingleColumnRowMapper` 是 `RowMapper` 接口的实现类，因此可以直接查看 `mapRow` 方法，具体代码如下。

```

@Override
@SuppressWarnings("unchecked")
@Nullable
public T mapRow(ResultSet rs, int rowNum) throws SQLException {
    // 获取 ResultSet 的元数据
    ResultSetMetaData rsmd = rs.getMetaData();
}

```

```

// 获取总列数
int nrOfColumns = rsmd.getColumnCount();
// 如果列数不等于1就抛出异常
if (nrOfColumns != 1) {
    throw new IncorrectResultSetColumnCountException(1, nrOfColumns);
}

// 将 rs 中的数据进行转换, 转换目标是 requiredType 类型
Object result = getColumnValue(rs, 1, this.requiredType);
if (result != null && this.requiredType != null && !this.requiredType.isInstance(
(result))) {
    try {
        return (T) convertValueToRequiredType(result, this.requiredType);
    }
    catch (IllegalArgumentException ex) {
        throw new TypeMismatchDataAccessException(
            "Type mismatch affecting row number " + rowNum + " and column type '" +
            rsmd.getColumnTypeName(1) + "': " + ex.getMessage());
    }
}
return (T) result;
}

```

在上述代码中主要处理流程如下。

- (1) 通过 **ResultSet** 从中获取 **ResultSet** 元数据。
- (2) 通过 **ResultSet** 元数据获取总列数, 此时获取的总列数如果数量不等于 1 将抛出异常。
- (3) 将 **ResultSet** 中的数据进行转换, 转换目标是成员变量 **requiredType** 的类型。当同时满足下面 3 个条件时则需要再次进行转换, 如果不满足条件将直接返回。条件如下。

- ① 转换结果不为空。
- ② 转换类型 (**requiredType** 变量) 不为空。
- ③ 转换类型不是转换结果的接口类型。

在上述 3 个处理流程中需要关注两个方法。

(1) 方法 **getColumnValue**, 作用是从 **ResultSet** 中根据 **requiredType** 变量获取数据值。

(2) 方法 **convertValueToRequiredType**, 作用是进行进一步转换。

下面先对 **getColumnValue** 方法进行分析, 主要处理代码如下。

```

@Nullable
protected Object getColumnValue(ResultSet rs, int index, @Nullable Class<?> requiredType)
throws SQLException {
    if (requiredType != null) {
        return JdbcUtils.getResultSetValue(rs, index, requiredType);
    }
    else {
        return getColumnValue(rs, index);
    }
}

```

在上述方法中有两个处理操作。

(1) 根据索引和目标类型从 `ResultSet` 中获取数据结果。

(2) 根据索引从 `ResultSet` 中获取数据结果。

上述两个操作的分支是判断 `requiredType` 是否为空，此处具体的获取方式为 `ResultSet.getXXX`，当 `requiredType` 不为空时 `XXX` 将转换为具体的数据类型，例如 `requiredType` 为 `String` 类型，`getXXX` 会转换为 `getString`，当 `requiredType` 为空时会变成 `getObject`。对此有兴趣的读者可以继续学习 `JdbcUtils.getResultSetValue` 方法和 `getColumnValue` 方法。

接下来对 `convertValueToRequiredType` 方法进行分析，主要处理代码如下。

```
@SuppressWarnings("unchecked")
@Nullable
protected Object convertValueToRequiredType(Object value, Class<?> requiredType) {
    if (String.class == requiredType) {
        return value.toString();
    }
    else if (Number.class.isAssignableFrom(requiredType)) {
        if (value instanceof Number) {
            return NumberUtils.convertNumberToTargetClass(((Number) value),
                (Class<Number>) requiredType);
        }
        else {
            return NumberUtils.parseNumber(value.toString(), (Class<Number>)
                requiredType);
        }
    }
    else if (this.conversionService != null &&
        this.conversionService.canConvert(value.getClass(), requiredType)) {
        return this.conversionService.convert(value, requiredType);
    }
    else {
        throw new IllegalArgumentException(
            "Value [" + value + "] is of type [" + value.getClass().getName() +
            "] and cannot be converted to required type [" + requiredType.getName() + "]");
    }
}
```

在上述代码中提到的转换方式有 3 种。

(1) 当 `requiredType` 类型为 `String` 时的 `toString` 转换。

(2) 当 `requiredType` 类型为 `Number` 的子类时的转换。

(3) 当转换服务可以进行 `value` 类型和 `requiredType` 类型转换时的转换。

在上述 3 种转换方式中第 1 种方式的转换内容不多，主要关注后两种转换方式。在 `Number` 相关转换中提出了两种类型转换：第 1 种是 `convertNumberToTargetClass`；第 2 种是 `parseNumber`。两者的差异为前者不需要进行 `value` 转换为字符串，后者需要进行 `value` 先转换为字符串再进行目标类型的转换。最后是第 3 种转换方式，在第 3 种转换方式中需要使用转换服务（`ConversionService`），通过转换服务来进行对象转换。此处属于 `SpringIoC` 相关内容，本章不做描述。

2. BeanPropertyRowMapper 对象分析

下面将对 BeanPropertyRowMapper 对象进行分析，由于对象 BeanPropertyRowMapper 是 RowMapper 接口的实现类，因此可以直接查看 mapRow 方法，具体代码如下。

```
@Override
public T mapRow(ResultSet rs, int rowNum) throws SQLException {
    Assert.state(this.mappedClass != null, "Mapped class was not specified");
    // 通过 BeanUtils 将 mappedClass 类型实例化
    T mappedObject = BeanUtils.instantiateClass(this.mappedClass);
    // 将 mappedObject 用 BeanWrapper 进行包装
    BeanWrapper bw = PropertyAccessorFactory.forBeanPropertyAccess(mappedObject);
    // 初始化 BeanWrapper
    initBeanWrapper(bw);

    // 获取 ResultSet 元数据
    ResultSetMetaData rsmd = rs.getMetaData();
    // 获取总列数
    int columnCount = rsmd.getColumnCount();
    Set<String> populatedProperties = (isCheckFullyPopulated() ? new HashSet<>() : null);

    for (int index = 1; index <= columnCount; index++) {
        // 列名
        String column = JdbcUtils.lookupColumnName(rsmd, index);
        // 字段名
        String field = lowerCaseName(StringUtils.delete(column, " "));
        // 属性描述对象
        PropertyDescriptor pd=(this.mappedFields!=null?this.mappedFields.get(field):null);
        if (pd != null) {
            try {
                // 获取数据值
                Object value = getColumnValue(rs, index, pd);
                if (rowNum == 0 && logger.isDebugEnabled()) {
                    logger.debug("Mapping column '" + column + "' to property '" + pd.getName()
+ "' of type '" + ClassUtils.getQualifiedName(pd.getPropertyType()) + "'");
                }
                try {
                    // 通过 BeanWrapper 设置属性值
                    bw.setPropertyValue(pd.getName(), value);
                }
                catch (TypeMismatchException ex) {
                    if (value == null && this.primitivesDefaultedForNullValue) {
                        if (logger.isDebugEnabled()) {
                            logger.debug("Intercepted TypeMismatchException for row " +
rowNum + " and column '" + column + "' with null value when setting property '" + pd.getName()
+ "' of type '" + ClassUtils.getQualifiedName(pd.getPropertyType()) +
"' on object: " + mappedObject, ex);
                        }
                    }
                }
            }
            else {

```

```

        throw ex;
    }
}
if (populatedProperties != null) {
    // 在属性值名称列表中添加数据
    populatedProperties.add(pd.getName());
}
}
catch (NotWritablePropertyException ex) {
    throw new DataRetrievalFailureException(
        "Unable to map column '" + column + "' to property '" + pd.getName()
+ "'", ex);
}
}
else {
    if (rowNumber == 0 && logger.isDebugEnabled()) {
        logger.debug("No property found for column '" + column + "' mapped to field '" + field + "'");
    }
}
}

if (populatedProperties != null && !populatedProperties.equals(this.mappedProperties)) {
    throw new InvalidDataAccessApiUsageException("Given ResultSet does not contain all fields " + "necessary to populate object of class [" + this.mappedClass.getName() + "]: " + this.mappedProperties);
}

return mappedObject;
}

```

从上述代码中可以发现，`BeanPropertyRowMapper` 的 `mapRow` 方法主要围绕 `BeanWrapper` 进行处理，具体处理细节如下。

- (1) 通过 `BeanUtils` 将 `mappedClass` 对象的实际类型进行实例化。
- (2) 通过 `PropertyAccessorFactory` 将第 (1) 步中实例化的结果进行 `BeanWrapper` 包装。
- (3) 初始化 `BeanWrapper` 对象，主要目的是进行转换服务的设置。
- (4) 从 `ResultSet` 对象中获取 `ResultSet` 元数据和总列数。
在得到总列数后会进行如下操作。
 - (1) 从 `ResultSet` 元数据中获取当前处理的列名。
 - (2) 将得到的列名进行首字母小写处理并删除多余的空格。
 - (3) 获取属性描述符对象，获取方式是从 `mappedFields` 对象中根据首字母小写的列名获取。
 - (4) 根据索引和属性描述符从 `ResultSet` 中获取数据。
 - (5) 通过 `BeanWrapper` 的 `setProperty` 将第 (4) 步得到的数据进行属性设置。
 - (6) 将 `BeanWrapper` 对象中的实际对象返回。

在 `BeanPropertyRowMapper` 的 `mapRow` 方法处理过程中主要依赖于 `BeanWrapper`，通过 `BeanWrapper` 对象来对属性进行赋值从而将数据对象包装完成，主要方法是 `setProperty`，该方法可以理解为已知字段名称和数据通过反射进行字段赋值。

3. ColumnMapRowMapper 对象分析

下面将对 ColumnMapRowMapper 对象进行分析。由于对象 ColumnMapRowMapper 是 RowMapper 接口的实现类，因此可以直接查看 mapRow 方法，具体代码如下。

```
@Override
public Map<String, Object> mapRow(ResultSet rs, int rowNum) throws SQLException {
    ResultSetMetaData rsmd = rs.getMetaData();
    int columnCount = rsmd.getColumnCount();
    Map<String, Object> mapOfColumnValues = createColumnMap(columnCount);
    for (int i = 1; i <= columnCount; i++) {
        String column = JdbcUtils.lookupColumnName(rsmd, i);
        mapOfColumnValues.putIfAbsent(getColumnKey(column), getColumnValue(rs, i));
    }
    return mapOfColumnValues;
}
```

上述代码的主要操作流程如下。

- (1) 获取 ResultSet 元数据。
- (2) 获取总列数。
- (3) 创建返回值存储容器。
- (4) 循环处理每列数据，单列的处理逻辑如下。
 - ① 通过 JdbcUtils 从 ResultSet 元数据中获取列名。
 - ② 通过索引从 ResultSet 中获取数据值。
 - ③ 将列名和数据值放入存储容器中。

2.3.4 ResultSetExtractor 对象分析

在分析 query 方法时可以看到第三个参数是 ResultSetExtractor，它是一个接口类型，在 spring-jdbc 中存在 4 个子类，分别如下。

- (1) AbstractLobStreamingResultSetExtractor。
- (2) SqlRowSetResultSetExtractor。
- (3) RowCallbackHandlerResultSetExtractor。
- (4) RowMapperResultSetExtractor。

在前面已经分析了 RowMapperResultSetExtractor，它是通过 RowMapper 接口从 ResultSet 中将数据进行转换并返回的。下面将对其他 3 个子类进行分析。

1. AbstractLobStreamingResultSetExtractor 对象分析

下面将对 AbstractLobStreamingResultSetExtractor 对象进行分析，由于对象 AbstractLobStreamingResultSetExtractor 是 ResultSetExtractor 接口的实现类，因此可以直接查看 extractData 方法，具体代码如下。

```
@Override
@Nullable
public final T extractData(ResultSet rs) throws SQLException, DataAccessException {
    if (!rs.next()) {
        handleNoRowFound();
    }
}
```

```

        else {
            try {
                streamData(rs);
                if (rs.next()) {
                    handleMultipleRowsFound();
                }
            }
            catch (IOException ex) {
                throw new LobRetrievalFailureException("Could not stream LOB content", ex);
            }
        }
        return null;
    }
}

```

上述代码主要处理逻辑如下。

- (1) 判断 **ResultSet** 中是否还有数据需要处理，如果没有则抛出异常。
- (2) 如果 **ResultSet** 中有需要处理的数据将通过 **streamData** 方法进行处理。处理完毕后判断 **ResultSet** 中是否还有下一个数据需要处理，如果有则抛出异常。

在这个处理流程中需要注意两点。

- (1) **streamData** 为抽象方法，在 **spring-jdbc** 项目中，目前 **AbstractLobStreamingResultSetExtractor** 类并未出现子类实现（非测试类）。
- (2) 返回值为 **null**。

2. SqlRowSetResultSetExtractor 对象分析

下面将对 **SqlRowSetResultSetExtractor** 对象进行分析，由于对象 **SqlRowSetResultSetExtractor** 是 **ResultSetExtractor** 接口的实现类，因此可以直接查看 **extractData** 方法，具体代码如下。

```

@Override
public SqlRowSet extractData(ResultSet rs) throws SQLException {
    return createSqlRowSet(rs);
}

```

在上述代码中主要处理流程是创建 **SqlRowSet** 接口的实现类，实现类具体类型是 **ResultSetWrappingSqlRowSet**，具体创建代码如下。

```

protected SqlRowSet createSqlRowSet(ResultSet rs) throws SQLException {
    CachedRowSet rowSet = newCachedRowSet();
    rowSet.populate(rs);
    return new ResultSetWrappingSqlRowSet(rowSet);
}

```

在这个创建过程中最终要的是 **ResultSetWrappingSqlRowSet** 对象的构造方法，具体代码如下。

```

public ResultSetWrappingSqlRowSet(ResultSet resultSet) throws
InvalidResultSetAccessException {
    this.resultSet = resultSet;
    try {
        this.rowSetMetaData = new
ResultSetWrappingSqlRowSetMetaData(resultSet.getMetaData());
    }
}

```

```

        catch (SQLException se) {
            throw new InvalidResultSetAccessException(se);
        }
        try {
            ResultSetMetaData rsmd = resultSet.getMetaData();
            if (rsmd != null) {
                int columnCount = rsmd.getColumnCount();
                this.columnLabelMap = new HashMap<>(columnCount);
                for (int i = 1; i <= columnCount; i++) {
                    String key = rsmd.getColumnLabel(i);
                    if (!this.columnLabelMap.containsKey(key)) {
                        this.columnLabelMap.put(key, i);
                    }
                }
            }
            else {
                this.columnLabelMap = Collections.emptyMap();
            }
        }
        catch (SQLException se) {
            throw new InvalidResultSetAccessException(se);
        }
    }
}

```

在这个构造方法中，主要目的是完成 3 个成员变量的初始化。

- (1) 成员变量 `resultSet`，用于存储结果集合。
- (2) 成员变量 `rowSetMetaData`，用于存储 `ResultSet` 元数据。
- (3) 成员变量 `columnLabelMap`，用于存储列名和索引之间的映射关系。

3. RowCallbackHandlerResultSetExtractor 对象分析

下面将对 `RowCallbackHandlerResultSetExtractor` 对象进行分析，由于对象 `RowCallbackHandlerResultSetExtractor` 是 `ResultSetExtractor` 接口的实现类，因此可以直接查看 `extractData` 方法，具体代码如下。

```

@Override
@Nullable
public Object extractData(ResultSet rs) throws SQLException {
    while (rs.next()) {
        this.rch.processRow(rs);
    }
    return null;
}

```

在这段代码中，主要依赖 `rch` 进行 `ResultSet` 的处理，需要注意的是返回值为 `null`。

2.4 JdbcTemplate 中的更新操作分析

本节将对 `JdbcTemplate` 中的更新操作进行分析，下面先来看一个最为简单的 `update` 方法调

用，该方法只需要将更新的 SQL 语句作为参数进行传递即可，具体处理代码如下。

```
@Override
public int update(final String sql) throws DataAccessException {

    class UpdateStatementCallback implements StatementCallback<Integer>, SqlProvider {
        @Override
        public Integer doInStatement(Statement stmt) throws SQLException {
            int rows = stmt.executeUpdate(sql);
            if (logger.isTraceEnabled()) {
                logger.trace("SQL update affected " + rows + " rows");
            }
            return rows;
        }
        @Override
        public String getSql() {
            return sql;
        }
    }

    return updateCount(execute(new UpdateStatementCallback()));
}
```

上述代码的主要处理流程如下。

(1) 创建 `UpdateStatementCallback` 对象，该对象的 `doInStatement` 方法是通过 `Statement` 对象进行数据更新的。

(2) 通过 `execute` 方法传递 `UpdateStatementCallback` 对象。

(3) 通过 `updateCount` 返回处理数量。

在这个处理过程中，已经了解的方法有 `execute`，下面对 `updateCount` 方法进行分析，具体代码如下。

```
private static int updateCount(@Nullable Integer result) {
    Assert.state(result != null, "No update count");
    return result;
}
```

在上述代码中，主要处理操作为判断处理数量是否为空，如果不为空则返回，否则则抛出异常。

2.4.1 配合 `PreparedStatementCreator` 和 `PreparedStatementSetter` 的更新操作

本节将对 `update` 方法族中 `PreparedStatementCreator` 和 `PreparedStatementCallback` 的更新操作进行分析，具体处理代码如下。

```
protected int update(final PreparedStatementCreator psc, @Nullable final
PreparedStatementSetter pss)
    throws DataAccessException {

    logger.debug("Executing prepared SQL update");
```

```

return updateCount(execute(psc, ps -> {
    try {
        if (pss != null) {
            pss.setValues(ps);
        }
        int rows = ps.executeUpdate();
        if (logger.isTraceEnabled()) {
            logger.trace("SQL update affected " + rows + " rows");
        }
        return rows;
    }
    finally {
        if (pss instanceof ParameterDisposer) {
            ((ParameterDisposer) pss).cleanupParameters();
        }
    }
}));
}

```

在这段代码中可以发现，`execute` 方法使用的是 `PreparedStatementCreator` 参数和 `PreparedStatementCallback` 参数，在分析 `execute` 方法时可以确认核心处理是通过第二个参数 `PreparedStatementCallback` 进行处理的，在该方法中主要处理流程如下。

- (1) 参数 `PreparedStatementSetter` 不为空的情况下将 `ps` 设置为 `PreparedStatementSetter`。
- (2) 通过 `ps` 进行更新操作，更新完成后获取更新行号并将更新行号返回。
- (3) 当参数 `PreparedStatementSetter` 类型为 `ParameterDisposer` 时进行参数清理操作。

注意，上述提到的 `ps` 对象的实际类型是 `PreparedStatement`。

2.4.2 配合 SQL 的批量更新

本节将对配合 SQL 的批量更新进行源码分析，主要处理代码如下。

```

@Override
public int[] batchUpdate(final String... sql) throws DataAccessException {
    Assert.notEmpty(sql, "SQL array must not be empty");
    if (logger.isDebugEnabled()) {
        logger.debug("Executing SQL batch update of " + sql.length + " statements");
    }

    class BatchUpdateStatementCallback implements StatementCallback<int[]>, SqlProvider {

        @Nullable
        private String currSql;

        @Override
        public int[] doInStatement(Statement stmt) throws SQLException,
            DataAccessException {
            int[] rowsAffected = new int[sql.length];
            if (JdbcUtils.supportsBatchUpdates(stmt.getConnection())) {
                // 循环需要处理的 SQL 语句列表
            }
        }
    }
}

```

```

        for (String sqlStmt : sql) {
            // 进行 SQL 语句组合
            this.currSql = appendSql(this.currSql, sqlStmt);
            // 添加需要批量执行的 SQL 语句
            stmt.addBatch(sqlStmt);
        }
        try {
            // 执行 SQL 语句
            rowsAffected = stmt.executeBatch();
        }
        catch (BatchUpdateException ex) {
            String batchExceptionSql = null;
            for (int i = 0; i < ex.getUpdateCounts().length; i++) {
                if (ex.getUpdateCounts()[i] == Statement.EXECUTE_FAILED) {
                    batchExceptionSql = appendSql(batchExceptionSql, sql[i]);
                }
            }
            if (StringUtils.hasLength(batchExceptionSql)) {
                this.currSql = batchExceptionSql;
            }
            throw ex;
        }
    }
    else {
        for (int i = 0; i < sql.length; i++) {
            this.currSql = sql[i];
            // 执行单条 SQL 语句
            if (!stmt.execute(sql[i])) {
                rowsAffected[i] = stmt.getUpdateCount();
            }
            else {
                throw new InvalidDataAccessApiUsageException("Invalid batch SQL
statement: " + sql[i]);
            }
        }
    }
    return rowsAffected;
}

private String appendSql(@Nullable String sql, String statement) {
    return (StringUtils.hasLength(sql) ? sql + "; " + statement : statement);
}

@Override
@Nullable
public String getSql() {
    return this.currSql;
}
}

```



```
int[] result = execute(new BatchUpdateStatementCallback());
Assert.state(result != null, "No update counts");
return result;
}
```

在这段代码中主要关注的是 `BatchUpdateStatementCallback` 对象的 `doInStatement` 方法，`BatchUpdateStatementCallback` 对象实现了 `StatementCallback` 接口，在这个基础上会调用 `execute` 和 `StatementCallback` 的方法进行处理。下面对 `doInStatement` 方法进行分析，在该方法中主要处理了两种批量更新模式。

(1) 数据库支持批量更新通过 SQL 语句的组合放入 `Statement` 对象中，再一次性提交所有 SQL 语句。

(2) 数据库不支持批量更新通过循环 SQL 语句列表一条一条执行 SQL 语句。

2.5 PreparedStatementCreator 和 PreparedStatementSetter 接口分析

在分析 `JdbcTemplate` 对象的更新操作和执行操作时遇到了 `PreparedStatementCreator` 接口和 `PreparedStatementSetter` 接口。

在 `spring-jdbc` 中，关于 `PreparedStatementCreator` 接口的实现类有如下两个。

(1) `PreparedStatementCreatorImpl`。

(2) `SimplePreparedStatementCreator`。

在 `spring-jdbc` 中，关于 `PreparedStatementSetter` 接口的实现类有如下 3 个。

(1) `ArgumentTypePreparedStatementSetter`。

(2) `PreparedStatementCreatorImpl`。

(3) `ArgumentPreparedStatementSetter`。

本节后续将会对上述 5 个类进行分析。

2.5.1 PreparedStatementCreatorImpl 对象分析

本节将分析 `PreparedStatementCreatorImpl` 对象，首先查看 `PreparedStatementCreatorImpl` 对象的成员变量，`PreparedStatementCreatorImpl` 中有如下两个成员变量。

(1) 成员变量 `actualSql`，用于存储需要执行的 SQL 语句，类型是字符串。

(2) 成员变量 `parameters`，用户存储的参数列表，类型是列表。

在 `PreparedStatementCreatorImpl` 对象中第一个需要关注的是构造方法，具体代码如下。

```
public PreparedStatementCreatorImpl(String actualSql, List<?> parameters) {
    this.actualSql = actualSql;
    Assert.notNull(parameters, "Parameters List must not be null");
    this.parameters = parameters;
    if (this.parameters.size() != declaredParameters.size()) {
        Set<String> names = new HashSet<>();
        for (int i = 0; i < parameters.size(); i++) {
            Object param = parameters.get(i);
            if (param instanceof SqlParameterValue) {
```

```

        names.add(((SqlParameterValue) param).getName());
    }
    else {
        names.add("Parameter #" + i);
    }
}
if (names.size() != declaredParameters.size()) {
    throw new InvalidDataAccessApiUsageException(
        "SQL [" + sql + "]: given " + names.size() +
        " parameters but expected " + declaredParameters.size());
}
}
}

```

在这个构造方法中，除了两个成员变量的赋值以外，还会进行一个额外的参数验证，执行验证的前提是当参数 `parameters` 列表中的元素数量和 `declaredParameters` 的元素数量不相同才需要处理，具体验证处理过程如下。

(1) 提取参数 `parameters` 中的参数名称。当元素类型是 `SqlParameterValue` 时通过 `getName` 方法获取参数名称，反之则进行字符串拼接并放入 `names` 中。

(2) 在处理完成所有的 `parameters` 元素后判断 `names` 列表中的元素数量是否和 `declaredParameters` 的元素数量相同，若不相同则抛出异常。

接下来对 `createPreparedStatement` 方法进行分析，该方法是 `PreparedStatementCreator` 的方法，具体实现代码如下。

```

@Override
public PreparedStatement createPreparedStatement(Connection con) throws SQLException
{
    PreparedStatement ps;
    if (generatedKeysColumnNames != null || returnGeneratedKeys) {
        if (generatedKeysColumnNames != null) {
            ps = con.prepareStatement(this.actualSql, generatedKeysColumnNames);
        }
        else {
            ps = con.prepareStatement(this.actualSql,
PreparedStatement.RETURN_GENERATED_KEYS);
        }
    }
    else if (resultSetType == ResultSet.TYPE_FORWARD_ONLY && !updatableResults) {
        ps = con.prepareStatement(this.actualSql);
    }
    else {
        ps = con.prepareStatement(this.actualSql, resultSetType,
            updatableResults ? ResultSet.CONCUR_UPDATABLE :
ResultSet.CONCUR_READ_ONLY);
    }
    setValues(ps);
    return ps;
}

```

在 `createPreparedStatement` 方法处理过程中，主要目的有如下两个。

(1) 通过数据库链接对象创建 `PreparedStatement` 对象用于返回值。

(2) 为 PreparedStatement 进行数据设置。

在第一个操作过程中, 关于 PreparedStatement 对象的创建使用的是 prepareStatement 方法, prepareStatement 方法和 createPreparedStatement 方法的差异是不同的参数信息, 具体差异参数如下。

- (1) SQL 语句+列名。
- (2) SQL 语句+autoGeneratedKeys。
- (3) 只有 SQL 语句。
- (4) SQL 语句+返回值类型+resultSetConcurrency。

在这 4 个不同的参数列表中需要关注下面两个变量表, 关于 autoGeneratedKeys 的数据选项可以查看表 2.1 所示的 autoGeneratedKeys 数据说明, 关于 resultSetConcurrency 的数据选项可以查看表 2.2 所示的 resultSetConcurrency 说明。

表 2.1 autoGeneratedKeys 数据说明

变 量 名 称	变 量 含 义
java.sql.Statement#CLOSE_ALL_RESULTS	ResultSet 在调用时应关闭所有先前保持打开状态的常量 getMoreResults
java.sql.Statement#CLOSE_CURRENT_RESULT	ResultSet 在调用时应关闭当前对象的常数 getMoreResults
java.sql.Statement#EXECUTE_FAILED	执行批处理语句时发生错误的常量
java.sql.Statement#KEEP_CURRENT_RESULT	ResultSet 在调用时不应关闭当前对象的常数 getMoreResults
java.sql.Statement#NO_GENERATED_KEYS	生成的密钥不应用于检索的常量
java.sql.Statement#RETURN_GENERATED_KEYS	生成的密钥可应用于检索的常数
SUCCESS_NO_INFO	该常量指示批处理语句已成功执行, 但没有影响到行数的计数

表 2.2 resultSetConcurrency 说明

变 量 名 称	变 量 含 义
CLOSE_CURSORS_AT_COMMIT	在提交事务后, ResultSet 将保存相关数据
CONCUR_READ_ONLY	只允许读取 ResultSet 对象, 无法对其进行更新
CONCUR_UPDATABLE	允许更新 ResultSet 对象
FETCH_FORWARD	结果集中的行将向前处理的常数; 从前向后处理
FETCH_REVERSE	结果集中的行将以相反方向处理的常数; 从后向前处理
FETCH_UNKNOWN	结果集中的行将被处理的顺序的常数是未知的
HOLD_CURSORS_OVER_COMMIT	与 CLOSE_CURSORS_AT_COMMIT 不同的是, 在提交事务后 ResultSet 可用
TYPE_FORWARD_ONLY	ResultSet 支持 forward 操作
TYPE_SCROLL_INSENSITIVE	ResultSet 支持 backforward、random、last 和 first 操作, 对其他会话中的数据库做出的操作不敏感
TYPE_SCROLL_SENSITIVE	ResultSet 支持 backforward、random、last 和 first 操作, 对其他会话中的数据库做出的操作敏感

下面将对 `setValues` 方法进行分析，具体处理代码如下。

```
@Override
public void setValues(PreparedStatement ps) throws SQLException {
    int sqlColIndx = 1;
    for (int i = 0; i < this.parameters.size(); i++) {
        Object in = this.parameters.get(i);
        SqlParameter declaredParameter;
        // 推论 SqlParameter 对象
        if (in instanceof SqlParameterValue) {
            SqlParameterValue paramValue = (SqlParameterValue) in;
            in = paramValue.getValue();
            declaredParameter = paramValue;
        }
        else {
            if (declaredParameters.size() <= i) {
                throw new InvalidDataAccessApiUsageException(
                    "SQL [" + sql + "]: unable to access parameter number " + (i + 1) +
                    " given only " + declaredParameters.size() + " parameters");
            }
            declaredParameter = declaredParameters.get(i);
        }
        // 进行 SqlStatement 对象赋值
        if (in instanceof Iterable && declaredParameter.getSqlType() != Types.ARRAY) {
            Iterable<?> entries = (Iterable<?>) in;
            for (Object entry : entries) {
                if (entry instanceof Object[]) {
                    Object[] valueArray = (Object[]) entry;
                    for (Object argValue : valueArray) {
                        StatementCreatorUtils.setParameterValue(ps, sqlColIndx++,
declaredParameter, argValue);
                    }
                }
                else {
                    StatementCreatorUtils.setParameterValue(ps, sqlColIndx++,
declaredParameter, entry);
                }
            }
        }
        else {
            StatementCreatorUtils.setParameterValue(ps, sqlColIndx++, declaredParameter, in);
        }
    }
}
```

在这个方法中，主要围绕 `parameters` 进行单个元素的处理，处理目标是为 `SqlStatement` 进行数据值的设置，具体单个元素的处理细节如下。

(1) 推论需要进行设置的数据对象值。

(2) 通过 `StatementCreatorUtils` 中提供的 `setParameterValue` 方法进行数据值设置。

下面对于推论数据值进行分析。这个推论流程为：从 `parameters` 集合中提取一个元素，判断该元素是不是 `SqlParameterValue` 类型，如果是 `SqlParameterValue` 类型则直接作为推论结果，如果不是则需要从 `declaredParameters` 对象的集合中获取。注意，`declaredParameters` 对象的集合数据可能存在也可能不存在，如果 `declaredParameters` 对象中集合元素数量少于当前处理的索引值将抛出异常。

在完成需要设置的数据值推论后就需要进行数据值设置，在进行设置前还需要进行一次判断，通过这个判断会进行 `Iterable` 相关的数据值设置操作，不通过这个判断将直接进行数据设置。具体判断内容有两个，需要同时满足才会进行 `Iterable` 相关的数据值设置操作。

(1) 当前正在处理的对象是 `Iterable` 类型，当前正在处理的对象是指 `parameters` 中的元素。

(2) 推论值 `declaredParameter` 的 SQL 类型不是 `java.sql.Types#ARRAY`。

在进行数据设置时都需要使用 `StatementCreatorUtils.setParameterValue` 方法，这个方法主要是进行 `PreparedStatement` 设置，具体方法可以有如下 3 种（不完全举例）。

(1) 方法 `setString`，用于设置 `String` 类型的数据。

(2) 方法 `setBigDecimal`，用于设置 `BigDecimal` 类型的数据。

(3) 方法 `setBoolean`，用于设置 `Boolean` 类型的数据。

2.5.2 SimplePreparedStatementCreator 对象分析

本节将分析 `SimplePreparedStatementCreator` 对象，该对象是 `CallableStatementCreator` 的实现类，具体实现代码如下。

```
@Override
public CallableStatement createCallableStatement(Connection con) throws SQLException {
    return con.prepareCall(this.callString);
}
```

在这段代码中，通过数据库链接对象的 `prepareCall` 方法创建 `CallableStatement` 对象，创建所需的参数是需要执行的 SQL 语句。

2.5.3 ArgumentTypePreparedStatementSetter 对象分析

本节将分析 `ArgumentTypePreparedStatementSetter` 对象，在这个对象中有如下两个成员变量。

(1) 成员变量 `args`，用于存储参数值列表。

(2) 成员变量 `argTypes`，用于存储参数类型列表。

对象 `ArgumentTypePreparedStatementSetter` 是 `PreparedStatementSetter` 的实现类。关于 `setValues` 方法的处理代码如下。

```
@Override
public void setValues(PreparedStatement ps) throws SQLException {
    int parameterPosition = 1;
    // 判断数据值列表是否为空，判断参数列表是否为空
    if (this.args != null && this.argTypes != null) {
        for (int i = 0; i < this.args.length; i++) {
            Object arg = this.args[i];
            // 判断单个元素是否为集合
        }
    }
}
```

```

        if (arg instanceof Collection && this.argTypes[i] != Types.ARRAY) {
            Collection<?> entries = (Collection<?>) arg;
            for (Object entry : entries) {
                if (entry instanceof Object[]) {
                    Object[] valueArray = ((Object[]) entry);
                    for (Object argValue : valueArray) {
                        doSetValue(ps, parameterPosition, this.argTypes[i], argValue);
                        parameterPosition++;
                    }
                }
                else {
                    doSetValue(ps, parameterPosition, this.argTypes[i], entry);
                    parameterPosition++;
                }
            }
        }
        else {
            doSetValue(ps, parameterPosition, this.argTypes[i], arg);
            parameterPosition++;
        }
    }
}

```

在上述代码中主要进行设置的操作是由 `doSetValue` 方法进行的，在 `doSetValue` 方法中负责设置的操作具体依赖于 `StatementCreatorUtils.setParameterValue` 方法，该方法在前面已经讲述，其作用在本处不做说明。在这个方法处理过程中提供了两种设置方式，这两种方式分别如下。

- (1) 集合类型的设置。
- (2) 非集合类型的设置。

2.5.4 ArgumentPreparedStatementSetter 对象分析

本节将分析 `ArgumentPreparedStatementSetter` 对象，在这个对象中有一个成员变量 `args`，用于存储参数值列表。

对象 `ArgumentPreparedStatementSetter` 是 `PreparedStatementSetter` 的实现类。关于 `setValues` 方法的处理代码如下。

```

@Override
public void setValues(PreparedStatement ps) throws SQLException {
    if (this.args != null) {
        for (int i = 0; i < this.args.length; i++) {
            Object arg = this.args[i];
            doSetValue(ps, i + 1, arg);
        }
    }
}

```

将 `ArgumentPreparedStatementSetter` 和 `ArgumentTypePreparedStatementSetter` 对比，在 `ArgumentTypePreparedStatementSetter` 中关于 `setValues` 的处理操作增加了类型相关的处理，在 `ArgumentPreparedStatementSetter` 处理中仅仅是 `StatementCreatorUtils.setParameterValue` 方法的调用。

2.6 NamedParameterJdbcTemplate 分析

本节将分析 `NamedParameterJdbcTemplate`。`NamedParameterJdbcTemplate` 与 `JdbcTemplate` 相比，它在原有 `JdbcTemplate` 的基础上增加了支持命名参数的特性。接下来将对 `NamedParameterJdbcTemplate` 的使用做简单说明。在 `NamedParameterJdbcTemplate` 中提供了如下两种构造方法。

(1) 通过 `DataSource` 进行构造，具体代码如下。

```
public NamedParameterJdbcTemplate(DataSource dataSource) {
    Assert.notNull(dataSource, "DataSource must not be null");
    this.classicJdbcTemplate = new JdbcTemplate(dataSource);
}
```

(2) 通过 `JdbcOperations` 进行构造，具体代码如下。

```
public NamedParameterJdbcTemplate(JdbcOperations classicJdbcTemplate) {
    Assert.notNull(classicJdbcTemplate, "JdbcTemplate must not be null");
    this.classicJdbcTemplate = classicJdbcTemplate;
}
```

在这两种构造方法中可以发现，前者通过 `DataSource` 在构造方法内部创建了 `JdbcTemplate` 对象，后者中 `JdbcOperations` 作为参数，在 `spring-jdbc` 中最为熟知且常用的对象是 `JdbcTemplate`。在了解了这两项内容后开始编写测试用例，首先创建一个类，类名为 `NamedParameterJdbcTemplateDemo`，具体代码如下。

```
@Configuration
public class NamedParameterJdbcTemplateDemo {

    @Bean
    public DataSource mysqlDataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("");
        dataSource.setUrl("");
        dataSource.setUsername("");
        dataSource.setPassword("");
        return dataSource;
    }

    @Bean
    public NamedParameterJdbcTemplate namedParameterJdbcTemplateDatasource(
        @Qualifier("mysqlDataSource") DataSource dataSource
    ){
        return new NamedParameterJdbcTemplate(dataSource);
    }

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new
        AnnotationConfigApplicationContext(NamedParameterJdbcTemplateDemo.class);
        NamedParameterJdbcTemplate template =
        context.getBean("namedParameterJdbcTemplateDatasource", NamedParameterJdbcTemplate.class);
        Map<String, Object> paramMap = new HashMap<>();
        paramMap.put("id", "10");
        paramMap.put("name", "小明");
        template.update(
```

```

        "insert into t_user(id,name) values (:id,:name)",
        paramMap
    );
}
}

```

在这段代码中，需要读者填写的内容如下。

- (1) 数据库链接地址。
- (2) 数据库驱动。
- (3) 数据库用户名称。
- (4) 数据库密码。

在上述代码中演示了通过 `DataSource` 进行 `NamedParameterJdbcTemplate` 创建的过程，此外还对一个 `insert` 语句配合参数表进行简单使用。下面将介绍通过 `JdbcOperations` 构造 `NamedParameterJdbcTemplate`，具体代码如下。

```

@Bean
public JdbcTemplate jdbcTemplate(
    DataSource mysqlDataSource
) {
    JdbcTemplate jdbcTemplate = new JdbcTemplate();
    jdbcTemplate.setDataSource(mysqlDataSource);
    jdbcTemplate.setLazyInit(false);
    return jdbcTemplate;
}

@Bean
public NamedParameterJdbcTemplate namedParameterJdbcTemplateJdbcOperation(
    @Qualifier("jdbcTemplate") JdbcTemplate jdbcTemplate
){
    return new NamedParameterJdbcTemplate(jdbcTemplate);
}

```

在 `NamedParameterJdbcTemplate` 对象中提供了 `execute`（执行）、`update`（更新）和 `query`（查询）3 个方法，下面将对这 3 个方法进行相关分析。

2.6.1 NamedParameterJdbcTemplate 中的执行操作

接下来将对执行操作进行相关分析，首先查看下面代码。

```

@Override
@Nullable
public <T> T execute(String sql, SqlParameterSource paramSource,
    PreparedStatementCallback<T> action)
    throws DataAccessException {

    return getJdbcOperations().execute(getPreparedStatementCreator(sql,paramSource),action);
}

```

在这段代码中可以发现，需要通过 `JdbcOperations` 提供的 `execute` 方法完成具体的 SQL 语句执行，具体需要关注第一个参数 `PreparedStatementCreator`，该参数在这段代码中通过 `getPreparedStatementCreator` 方法生成，具体处理代码如下。


```

protected PreparedStatementCreator getPreparedStatementCreator(String sql,
    SqlParameterSource paramSource) {
    return getPreparedStatementCreator(sql, paramSource, null);
}

protected PreparedStatementCreator getPreparedStatementCreator(String sql,
    SqlParameterSource paramSource, @Nullable Consumer<PreparedStatementCreatorFactory>
    customizer) {

    ParsedSql parsedSql = getParsedSql(sql);
    PreparedStatementCreatorFactory pscf =
    getPreparedStatementCreatorFactory(parsedSql, paramSource);
    if (customizer != null) {
        customizer.accept(pscf);
    }
    Object[] params = NamedParameterUtils.buildValueArray(parsedSql, paramSource,
    null);
    return pscf.newPreparedStatementCreator(params);
}

```

在 `getPreparedStatementCreator` 代码中，主要处理流程如下。

- (1) 进行 SQL 语句解析得到 `ParsedSql` 对象。
- (2) 通过 `ParsedSql` 对象和参数列表创建 `PreparedStatementCreatorFactory` 对象。
- (3) 通过 `ParsedSql` 对象和参数列表求得最终的参数值。
- (4) 通过 `PreparedStatementCreatorFactory` 对象提供的 `newPreparedStatementCreator` 方法配合第 (3) 步的参数值得到 `PreparedStatementCreator` 对象。

接下来将对上述流程中所用到的方法进行分析。

- (1) 方法 `getParsedSql`，作用是对 SQL 语句进行解析。
- (2) 方法 `getPreparedStatementCreatorFactory`，作用是创建 `PreparedStatementCreatorFactory` 对象。
- (3) 方法 `NamedParameterUtils.buildValueArray`，作用是提取参数值。
- (4) 方法 `newPreparedStatementCreator`，作用是创建 `PreparedStatementCreator` 接口的实现类。

1. `getParsedSql` 方法分析

下面将对 `getParsedSql` 方法进行分析，具体处理代码如下。

```

protected ParsedSql getParsedSql(String sql) {
    if (getCacheLimit() <= 0) {
        return NamedParameterUtils.parseSqlStatement(sql);
    }
    synchronized (this.parsedSqlCache) {
        ParsedSql parsedSql = this.parsedSqlCache.get(sql);
        if (parsedSql == null) {
            parsedSql = NamedParameterUtils.parseSqlStatement(sql);
            this.parsedSqlCache.put(sql, parsedSql);
        }
        return parsedSql;
    }
}

```

在这段代码中，可以分为两类执行逻辑。

- (1) 存在缓存的处理。
- (2) 不存在缓存的处理。

关于上述两者的处理逻辑判断条件是成员变量 `cacheLimit`（缓存容量）是否大于 0，若大于 0 则需要进行缓存处理，若小于或等于 0 则不需要。注意，`cacheLimit` 的默认值是 256。在这段代码中，核心的处理代码是 `NamedParameterUtils.parseSqlStatement`。下面将前文的 SQL 语句单独使用该方法进行处理，编写下面代码。

```
ParsedSql parsedSql = NamedParameterUtils.parseSqlStatement("insert into t_user(id,name) values (:id,:name)");
```

通过执行上述代码可以得到 `parsedSql` 对象，具体信息如图 2.6 所示。

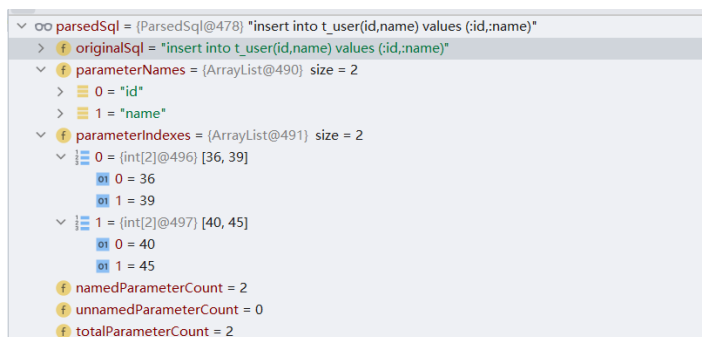


图 2.6 `parsedSql` 对象信息

在图 2.6 中可以发现，`ParsedSql` 对象存在 6 个成员变量，关于这 6 个成员变量的信息可查看表 2.3 所示的 `ParsedSql` 成员变量表。

表 2.3 `ParsedSql` 成员变量表

属性名称	属性类型	属性说明
<code>originalSql</code>	<code>String</code>	用于存储待解析的 SQL 语句
<code>parameterNames</code>	<code>List<String></code>	用于存储 SQL 语句中的参数名称
<code>parameterIndexes</code>	<code>List<int[]></code>	用于存储 SQL 语句中的参数索引。注意，元素是 <code>int</code> 数组，一般长度为 2，第一位存储冒号索引，第二位存储参数名称最后一个字符索引
<code>namedParameterCount</code>	<code>int</code>	参数名称数量
<code>unnamedParameterCount</code>	<code>int</code>	未知参数名称数量
<code>totalParameterCount</code>	<code>int</code>	参数名称总数量

2. `getPreparedStatementCreatorFactory` 方法分析

下面将对 `getPreparedStatementCreatorFactory` 方法进行分析，具体处理代码如下。

```
protected PreparedStatementCreatorFactory getPreparedStatementCreatorFactory(
    ParsedSql parsedSql, SqlParameterSource paramSource) {

    String sqlToUse = NamedParameterUtils.substituteNamedParameters(parsedSql, paramSource);
    List<SqlParameter> declaredParameters =
        NamedParameterUtils.buildSqlParameterList(parsedSql, paramSource);
```

```
return new PreparedStatementCreatorFactory(sqlToUse, declaredParameters);
}
```

在上述代码中，主要处理流程有如下 3 步。

- (1) 将带有参数名称的 SQL 语句进行替换，替换目标是问号。
- (2) 提取参数名称。
- (3) 创建 `PreparedStatementCreatorFactory` 对象。

下面先来看看第 (1) 步中得到的数据信息，具体如图 2.7 所示。

```
> sqlToUse = "insert into t_user(id,name) values (?,?)"
```

图 2.7 SQL 语句替换结果

第 (2) 步中得到的数据信息如图 2.8 所示。

```

declaredParameters = {ArrayList@1600} size = 2
  0 = {SqlParameter@1610}
    name = "id"
    sqlType = -2147483648
    typeName = null
    scale = null
  1 = {SqlParameter@1611}
    name = "name"
    sqlType = -2147483648
    typeName = null
    scale = null

```

图 2.8 参数名提取结果

第 (3) 步中得到的数据信息如图 2.9 所示。

```

new PreparedStatementCreatorFactory(sqlToUse, declaredParameters) = {PreparedStatementCreatorFactory@1600}
  sql = "insert into t_user(id,name) values (?,?)"
  declaredParameters = {ArrayList@1600} size = 2
  resultSetType = 1003
  updatableResults = false
  returnGeneratedKeys = false
  generatedKeysColumnNames = null

```

图 2.9 PreparedStatementCreatorFactory 对象信息

下面对第 (1) 步的细节进行分析，在前面对 `ParsedSql` 分析时已经确认了参数名称的索引位，具体参数是 `parameterIndexes`，既然是进行替换那么还需要原有 SQL 语句，通过原有 SQL 语句配合索引区间将索引区间值替换为问号“?”即可完成处理。这部分的具体处理代码如下。

```

public static String substituteNamedParameters(ParsedSql parsedSql, @Nullable
SqlParameterSource paramSource) {
    // 提取带有参数名称的 SQL 语句
    String originalSql = parsedSql.getOriginalSql();
    // 提取参数名称
    List<String> paramNames = parsedSql.getParameterNames();
    if (paramNames.isEmpty()) {
        return originalSql;
    }
    StringBuilder actualSql = new StringBuilder(originalSql.length());
    int lastIndex = 0;

```

```

// 循环参数名称列表
for (int i = 0; i < paramNames.size(); i++) {
    String paramName = paramNames.get(i);
    // 提取索引区间
    int[] indexes = parsedSql.getParameterIndexes(i);
    int startIndex = indexes[0];
    int endIndex = indexes[1];
    actualSql.append(originalSql, lastIndex, startIndex);
    // 进行索引区间字符串的替换操作
    if (paramSource != null && paramSource.hasValue(paramName)) {
        Object value = paramSource.getValue(paramName);
        if (value instanceof SqlParameterValue) {
            value = ((SqlParameterValue) value).getValue();
        }
        if (value instanceof Iterable) {
            Iterator<?> entryIter = ((Iterable<?>) value).iterator();
            int k = 0;
            while (entryIter.hasNext()) {
                if (k > 0) {
                    actualSql.append(", ");
                }
                k++;
                Object entryItem = entryIter.next();
                if (entryItem instanceof Object[]) {
                    Object[] expressionList = (Object[]) entryItem;
                    actualSql.append('(');
                    for (int m = 0; m < expressionList.length; m++) {
                        if (m > 0) {
                            actualSql.append(", ");
                        }
                        actualSql.append('?');
                    }
                    actualSql.append(')');
                }
                else {
                    actualSql.append('?');
                }
            }
        }
        else {
            actualSql.append('?');
        }
    }
    lastIndex = endIndex;
}
actualSql.append(originalSql, lastIndex, originalSql.length());
return actualSql.toString();
}

```

在方法 `getPreparedStatementCreatorFactory` 中得到替换参数名称的 SQL 语句后需要进行参

数名称提取，具体处理代码如下。

```
public static List<SqlParameter> buildSqlParameterList(ParsedSql parsedSql,
SqlParameterSource paramSource) {
    List<String> paramNames = parsedSql.getParameterNames();
    List<SqlParameter> params = new ArrayList<>(paramNames.size());
    for (String paramName : paramNames) {
        params.add(new SqlParameter(
            paramName, paramSource.getSqlType(paramName),
            paramSource.getTypeName(paramName)));
    }
    return params;
}
```

在这段代码中主要处理操作如下。

- (1) 从参数 `ParsedSql` 提取参数名称列表。
- (2) 创建存储返回值的容器。
- (3) 循环处理参数名称列表，单个处理操作是通过 `SqlParameter` 的构造方法进行初始化的。

在第三个处理过程中需要使用到 `SqlParameterSource` 接口，该接口可以用于提取 `sqlType` 变量和 `typeName` 变量，关于 `SqlParameterSource` 的分析将在本书后面介绍。当得到替换参数名称的 SQL 语句和 SQL 语句参数列表后，通过 `PreparedStatementCreatorFactory` 的构造函数将 `PreparedStatementCreatorFactory` 对象进行初始化，完成 `getPreparedStatementCreatorFactory` 方法处理。初始化过程仅为成员变量赋值，没有其他操作。

3. NamedParameterUtils.buildValueArray 方法分析

下面将对 `NamedParameterUtils.buildValueArray` 方法进行分析，具体处理代码如下。

```
public static Object[] buildValueArray(
    ParsedSql parsedSql, SqlParameterSource paramSource, @Nullable
    List<SqlParameter> declaredParams) {

    // 创建参数值存储容器
    Object[] paramArray = new Object[parsedSql.getTotalParameterCount()];
    // 异常处理
    if (parsedSql.getNamedParameterCount() > 0 &&
        parsedSql.getUnnamedParameterCount() > 0) {
        throw new InvalidDataAccessApiUsageException(
            "Not allowed to mix named and traditional ? placeholders. You have " +
            parsedSql.getNamedParameterCount() + " named parameter(s) and " +
            parsedSql.getUnnamedParameterCount() + " traditional placeholder(s) in
statement: " + parsedSql.getOriginalSql());
    }
    // 参数名称列表
    List<String> paramNames = parsedSql.getParameterNames();
    for (int i = 0; i < paramNames.size(); i++) {
        String paramName = paramNames.get(i);
        try {
            // 从参数表中获取数据
            Object value = paramSource.getValue(paramName);
            // 确认具体的 SQL 语句参数
            SqlParameter param = findParameter(declaredParams, paramName, i);
```

```

        // 将参数列表中的元素进行赋值
        paramArray[i] = (param != null ? new SqlParameterValue(param, value) : value);
    }
    catch (IllegalArgumentException ex) {
        throw new InvalidDataAccessApiUsageException(
            "No value supplied for the SQL parameter '" + paramName + "': " +
            ex.getMessage());
    }
}
return paramArray;
}

```

在上述代码中，主要处理的操作流程如下。

(1) 创建存储参数值的容器。

(2) 进行可能的异常处理，当 `namedParameterCount` 大于 0 并且 `unnamedParameterCount` 大于 0 时抛出异常。

(3) 从 `SqlParameterSource` 中提取实际的参数值，将提取的实际值放入第 (1) 步的容器中。

在上述操作流程中需要重点关注的是第 (3) 步，在第 (3) 步中需要使用 `SqlParameterSource` 接口提供的 `getValue` 方法，该方法用于获取参数名称对应的具体参数值，还需要关注 `findParameter` 方法的处理，它会从方法参数 `declaredParams` 中寻找参数名称对应的参数值，在寻找完成后进行元素填充，具体填充方式有两种：第一种是直接对 `getValue` 的值进行填充；第二种是通过 `SqlParameterValue` 构造方法进行对象创建后填充。关于 `SqlParameterSource` 提供的 `getValue` 方法将在本章后续进行分析，下面对 `findParameter` 方法进行分析，具体处理代码如下。

```

@Nullable
private static SqlParameter findParameter(
    @Nullable List<SqlParameter> declaredParams, String paramName, int paramIndex)
{
    if (declaredParams != null) {
        // 判断名称是否相同，如果名称相同则获取
        for (SqlParameter declaredParam : declaredParams) {
            if (paramName.equals(declaredParam.getName())) {
                return declaredParam;
            }
        }
    }
    if (paramIndex < declaredParams.size()) {
        // 通过索引获取，并且参数名称不存在才可作为返回值
        SqlParameter declaredParam = declaredParams.get(paramIndex);
        if (declaredParam.getName() == null) {
            return declaredParam;
        }
    }
    return null;
}

```

在 `findParameter` 方法中提供了两种搜索方式。

(1) 通过参数名称进行搜索。

(2) 通过索引值进行搜索。

4. newPreparedStatementCreator 方法分析

下面将对 newPreparedStatementCreator 方法进行分析，具体处理代码如下。

```
public PreparedStatementCreator newPreparedStatementCreator(@Nullable Object[] params)
{
    return new PreparedStatementCreatorImpl(params != null ? Arrays.asList(params) :
Collections.emptyList());
}
```

在这段代码中可以看到会进行 PreparedStatementCreatorImpl 对象的实例化操作，具体操作是对参数列表进行设置。

至此，对于 getPreparedStatementCreator 方法中所涉及的方法都已经分析完成，同时完成了执行方法的分析。

2.6.2 NamedParameterJdbcTemplate 中的更新操作

接下来将分析 NamedParameterJdbcTemplate 中的更新操作，具体处理代码如下。

```
@Override
public int update(String sql, Map<String, ?> paramMap) throws DataAccessException {
    return update(sql, new MapSqlParameterSource(paramMap));
}
```

在这段代码中还需要进一步调用更新方法，具体调用如下。

```
@Override
public int update(String sql, SqlParameterSource paramSource) throws DataAccessException
{
    return getJdbcOperations().update(getPreparedStatementCreator(sql, paramSource));
}
```

在这个方法中可以看到熟悉的方法 getPreparedStatementCreator，该方法在分析执行操作时已经做过详细分析，此处不做分析。在 NamedParameterJdbcTemplate 对象中的更新方法族中处理逻辑都相似，都需要通过 JdbcOperations 进行更新操作。接下来对于批量更新进行分析，具体处理代码如下。

```
@Override
public int[] batchUpdate(String sql, SqlParameterSource[] batchArgs) {
    if (batchArgs.length == 0) {
        return new int[0];
    }

    ParsedSql parsedSql = getParsedSql(sql);
    PreparedStatementCreatorFactory pscf = getPreparedStatementCreatorFactory(parsedSql,
batchArgs[0]);

    return getJdbcOperations().batchUpdate(
        pscf.getSql(),
        new BatchPreparedStatementSetter() {
```

```

        @Override
        public void setValues(PreparedStatement ps, int i) throws SQLException {
            Object[] values = NamedParameterUtils.buildValueArray(parsedSql,
                batchArgs[i], null);
            pscf.newPreparedStatementSetter(values).setValues(ps);
        }
        @Override
        public int getBatchSize() {
            return batchArgs.length;
        }
    }
});
}

```

在这个批量更新操作中，具体处理流程如下。

- (1) 进行 SQL 语句解析得到 `ParsedSql` 对象。
- (2) 通过 `SqlParameterSource` 和 `ParsedSql` 创建 `PreparedStatementCreatorFactory` 对象。
- (3) 创建 `BatchPreparedStatementSetter` 对象，在这个对象中需要重写 `setValues` 方法和 `getBatchSize` 方法。

在上述 3 个步骤中，最关键的是 `BatchPreparedStatementSetter` 的重写，在这个重写过程中处理方法和执行操作中相同之处不做分析。

2.6.3 NamedParameterJdbcTemplate 中的查询操作

接下来将分析 `NamedParameterJdbcTemplate` 中的查询操作，具体处理代码如下。

```

@Override
@Nullable
public <T> T query(String sql, SqlParameterSource paramSource, ResultSetExtractor<T>
    rse)
    throws DataAccessException {

    return getJdbcOperations().query(getPreparedStatementCreator(sql, paramSource),
        rse);
}

```

在这段代码中可以发现一个熟悉的方法 `getPreparedStatementCreator`，该方法在分析执行方法时已做分析，本处不做分析。对于 `NamedParameterJdbcTemplate` 中的查询操作，可以这样理解：在原本具有参数名称的 SQL 语句中将参数名称进行替换，替换内容从 `SqlParameterSource` 对象中获取，替换后得到一个需要执行的 SQL 语句，在得到语句后执行该 SQL 语句，通过 `ResultSetExtractor` 处理返回值。

2.7 SqlParameterSource 接口分析

在分析 `NamedParameterJdbcTemplate` 对象时曾提到 `SqlParameterSource` 接口，本节将对 `SqlParameterSource` 接口进行分析，首先对接口进行简单认识，具体代码如下。

```

public interface SqlParameterSource {
    int TYPE_UNKNOWN = JdbcUtils.TYPE_UNKNOWN;
}

```



```
boolean hasValue(String paramName);

@Nullable
Object getValue(String paramName) throws IllegalArgumentException;

default int getSqlType(String paramName) {
    return TYPE_UNKNOWN;
}

@Nullable
default String getTypeName(String paramName) {
    return null;
}

@Nullable
default String[] getParameterNames() {
    return null;
}
}
```

在 `SqlParameterSource` 接口中定义了 5 个方法，关于这 5 个方法的介绍参见表 2.4 所示的 `SqlParameterSource` 方法。

表 2.4 `SqlParameterSource` 方法

名 称	作 用
<code>hasValue</code>	根据参数名称判断是否具备数据值
<code>getValue</code>	根据参数名称获取对应的参数值
<code>getSqlType</code>	根据参数名称获取该参数对应的 SQL 语句类型
<code>getTypeName</code>	根据参数名称获取该参数对应的 SQL 语句类型名称
<code>getParameterNames</code>	获取所有的参数名称

在 `spring-jdbc` 中，关于 `SqlParameterSource` 接口的子类实现有如下 4 个。

- (1) 类 `EmptySqlParameterSource`。
- (2) 类 `AbstractSqlParameterSource`。
- (3) 类 `BeanPropertySqlParameterSource`。
- (4) 类 `MapSqlParameterSource`。

下面将对这 4 个类进行分析。

2.7.1 `EmptySqlParameterSource` 类分析

`EmptySqlParameterSource` 类是 `SqlParameterSource` 接口的实现类，从名字可以知道 `EmptySqlParameterSource` 类中的实现基本上都是空实现，下面查看源码，具体代码如下。

```
public class EmptySqlParameterSource implements SqlParameterSource {
```

```
public static final EmptySqlParameterSource INSTANCE = new
EmptySqlParameterSource();

@Override
public boolean hasValue(String paramName) {
    return false;
}

@Override
@Nullable
public Object getValue(String paramName) throws IllegalArgumentException {
    throw new IllegalArgumentException("This SqlParameterSource is empty");
}

@Override
public int getSqlType(String paramName) {
    return TYPE_UNKNOWN;
}

@Override
@Nullable
public String getTypeName(String paramName) {
    return null;
}

@Override
@Nullable
public String[] getParameterNames() {
    return null;
}
}
```

在这段代码中提供了一个单例对象 INSTANCE，该对象是 EmptySqlParameterSource 实例，用于全局。在方法实现中都采用了默认值策略，因此这个类的处理细节不多。

2.7.2 AbstractSqlParameterSource 类分析

AbstractSqlParameterSource 类是 SqlParameterSource 接口的实现类，在这个类中存在两个成员变量。

- (1) 成员变量 sqlTypes，用于存储参数名称和 SQL 语句类型的关系。
 - (2) 成员变量 typeNames，用于存储参数名称和 SQL 语句类型名称的关系。
- 关于这两个成员变量的具体定义如下。

```
private final Map<String, Integer> sqlTypes = new HashMap<>();
private final Map<String, String> typeNames = new HashMap<>();
```

既然存在绑定关系，必然会出现关系绑定的方法和获取绑定关系的方法，在 AbstractSqlParameterSource 中提供的绑定方法代码如下。

```
public void registerSqlType(String paramName, int sqlType) {
    Assert.notNull(paramName, "Parameter name must not be null");
    this.sqlTypes.put(paramName, sqlType);
}

public void registerTypeName(String paramName, String typeName) {
    Assert.notNull(paramName, "Parameter name must not be null");
    this.typeNames.put(paramName, typeName);
}
```

在 `AbstractSqlParameterSource` 中提供的获取绑定关系的方法代码如下。

```
@Override
public int getSqlType(String paramName) {
    Assert.notNull(paramName, "Parameter name must not be null");
    return this.sqlTypes.getDefault(paramName, TYPE_UNKNOWN);
}

@Override
@Nullable
public String getTypeName(String paramName) {
    Assert.notNull(paramName, "Parameter name must not be null");
    return this.typeNames.get(paramName);
}
```

上述这两个方法不只提供了绑定关系获取的操作，同时它们还是 `SqlParameterSource` 接口的实现方法。

2.7.3 BeanPropertySqlParameterSource 类分析

下面将对 `BeanPropertySqlParameterSource` 类进行分析，`BeanPropertySqlParameterSource` 类是 `AbstractSqlParameterSource` 类的子类，在 `BeanPropertySqlParameterSource` 类中存在如下两个成员变量。

- (1) 成员变量 `beanWrapper`，用于存储 `Bean` 的包装对象。
- (2) 成员变量 `propertyNames`，用于存储属性值名称。

介绍完成员变量后，下面将对各个实现的方法进行分析，首先分析的是 `hasValue` 方法，具体处理代码如下。

```
@Override
public boolean hasValue(String paramName) {
    return this.beanWrapper.isReadableProperty(paramName);
}
```

在 `hasValue` 方法中，通过 `BeanWrapper` 对象的 `isReadableProperty`（是否可读）方法判断是否存在 `value` 值。继续向下分析 `getValue` 方法，具体处理代码如下。

```
@Override
@Nullable
public Object getValue(String paramName) throws IllegalArgumentException {
    try {
        return this.beanWrapper.getPropertyValue(paramName);
    }
}
```

```

        catch (NotReadablePropertyException ex) {
            throw new IllegalArgumentException(ex.getMessage());
        }
    }
}

```

在 `getValue` 方法中通过 `BeanWrapper` 对象的 `getPropertyValue`（获取属性）方法从 `BeanWrapper` 中获取参数名称对应的属性值。继续向下分析 `getSqlType` 方法，具体处理代码如下。

```

@Override
public int getSqlType(String paramName) {
    int sqlType = super.getSqlType(paramName);
    if (sqlType != TYPE_UNKNOWN) {
        return sqlType;
    }
    Class<?> propType = this.beanWrapper.getPropertyType(paramName);
    return StatementCreatorUtils.javaTypeToSqlParameterType(propType);
}

```

可以看到在 `getSqlType` 方法中会从父类方法中进行一次 SQL 语句类型的获取，当类型是未知时会通过 `BeanWrapper` 中的 `getPropertyType`（获取数据类型）方法得到一个 `Class`，在得到 `Class` 后会通过 `StatementCreatorUtils.javaTypeToSqlParameterType` 方法进一步获取一个数据。关于这个数据信息可以查看下面代码。

```

javaTypeToSqlTypeMap.put(boolean.class, Types.BOOLEAN);
javaTypeToSqlTypeMap.put(Boolean.class, Types.BOOLEAN);
javaTypeToSqlTypeMap.put(byte.class, Types.TINYINT);
javaTypeToSqlTypeMap.put(Byte.class, Types.TINYINT);
javaTypeToSqlTypeMap.put(short.class, Types.SMALLINT);
javaTypeToSqlTypeMap.put(Short.class, Types.SMALLINT);
javaTypeToSqlTypeMap.put(int.class, Types.INTEGER);
javaTypeToSqlTypeMap.put(Integer.class, Types.INTEGER);
javaTypeToSqlTypeMap.put(long.class, Types.BIGINT);
javaTypeToSqlTypeMap.put(Long.class, Types.BIGINT);
javaTypeToSqlTypeMap.put(BigInteger.class, Types.BIGINT);
javaTypeToSqlTypeMap.put(float.class, Types.FLOAT);
javaTypeToSqlTypeMap.put(Float.class, Types.FLOAT);
javaTypeToSqlTypeMap.put(double.class, Types.DOUBLE);
javaTypeToSqlTypeMap.put(Double.class, Types.DOUBLE);
javaTypeToSqlTypeMap.put(BigDecimal.class, Types.DECIMAL);
javaTypeToSqlTypeMap.put(java.sql.Date.class, Types.DATE);
javaTypeToSqlTypeMap.put(java.sql.Time.class, Types.TIME);
javaTypeToSqlTypeMap.put(java.sql.Timestamp.class, Types.TIMESTAMP);
javaTypeToSqlTypeMap.put(Blob.class, Types.BLOB);
javaTypeToSqlTypeMap.put(Clob.class, Types.CLOB);

```

最后对 `getParameterNames` 方法进行分析，具体处理代码如下。

```

@Override
@NotNull
public String[] getParameterNames() {
    return getReadablePropertyNames();
}

public String[] getReadablePropertyNames() {

```

```

    if (this.propertyNames == null) {
        List<String> names = new ArrayList<>();
        PropertyDescriptor[] props = this.beanWrapper.getPropertyDescriptors();
        for (PropertyDescriptor pd : props) {
            if (this.beanWrapper.isReadableProperty(pd.getName())) {
                names.add(pd.getName());
            }
        }
        this.propertyNames = StringUtils.toStringArray(names);
    }
    return this.propertyNames;
}

```

在这段代码中可以看到，关于参数名称的获取会通过 `BeanWrapper` 中的 `getPropertyDescriptors`（获取属性描述符对象）方法进行，在获取到属性描述符列表后判断是否可读，如果可读就会成为返回值中的一个。至此，关于 `BeanPropertySqlParameterSource` 的分析也就完成了。

2.7.4 MapSqlParameterSource 类分析

下面将对 `MapSqlParameterSource` 类进行分析。`MapSqlParameterSource` 类是 `AbstractSqlParameterSource` 类的子类，在 `MapSqlParameterSource` 类中存在一个成员变量 `values`，用于存储参数名称和参数值之间的关系。

关于 `values` 成员变量的定义代码如下。

```
private final Map<String, Object> values = new LinkedHashMap<>();
```

在 `MapSqlParameterSource` 对象中第一个需要关注的方法是 `addValue`，具体代码如下。

```

public MapSqlParameterSource addValue(String paramName, @Nullable Object value) {
    Assert.notNull(paramName, "Parameter name must not be null");
    this.values.put(paramName, value);
    if (value instanceof SqlParameterValue) {
        registerSqlType(paramName, ((SqlParameterValue) value).getSqlType());
    }
    return this;
}

```

在 `addValue` 方法中会进行如下处理。

- (1) 参数名称和参数值放入 `values` 容器中。
- (2) 如果 `value` 的类型是 `SqlParameterValue`，则会进行类型注册。

在 `MapSqlParameterSource` 中还有其他 `addValue` 方法，主要处理流程如下。

- (1) 将参数名称和参数值进行绑定。
- (2) 进行类型注册。
- (3) 进行类型名称注册。

在不同的 `addValue` 方法中会根据不同的参数进行上述 3 个方法的组合操作，第一个处理操作是必须执行的。接下来对 `hasValue` 方法进行分析，具体处理代码如下。

```

@Override
public boolean hasValue(String paramName) {
    return this.values.containsKey(paramName);
}

```

方法 `hasValue` 的判断逻辑是通过 `values` 容器中是否存在参数名称作为返回值。接下来对 `getValue` 方法进行分析，具体处理代码如下。

```
@Override
@Nullable
public Object getValue(String paramName) {
    if (!hasValue(paramName)) {
        throw new IllegalArgumentException("No value registered for key '" + paramName + "'");
    }
    return this.values.get(paramName);
}
```

在 `getValue` 方法中，对于数据的获取需要执行如下操作。

- (1) 判断容器中是否存在参数名称对应的数据，如果不存在则抛出异常。
- (2) 从容器中获取对应的数据。

最后对 `getParameterNames` 方法进行分析，具体处理代码如下。

```
@Override
@NonNull
public String[] getParameterNames() {
    return StringUtils.toStringArray(this.values.keySet());
}
```

在 `getParameterNames` 中提供的获取参数名称方式是从 `values` 中获取所有的键。

2.8 JdbcTemplate 中的 call 方法分析

本节将对 `JdbcTemplate` 中的 `call` 方法进行分析。该方法主要用于执行数据库中的函数，具体处理代码如下。

```
@Override
public Map<String, Object> call(CallableStatementCreator csc, List<SqlParameter>
    declaredParameters)
    throws DataAccessException {

    final List<SqlParameter> updateCountParameters = new ArrayList<>();
    final List<SqlParameter> resultSetParameters = new ArrayList<>();
    final List<SqlParameter> callParameters = new ArrayList<>();

    for (SqlParameter parameter : declaredParameters) {
        if (parameter.isResultsParameter()) {
            if (parameter instanceof SqlReturnResultSet) {
                resultSetParameters.add(parameter);
            }
            else {
                updateCountParameters.add(parameter);
            }
        }
        else {
            callParameters.add(parameter);
        }
    }
}
```

```

Map<String, Object> result = execute(csc, cs -> {
    boolean retVal = cs.execute();
    int updateCount = cs.getUpdateCount();
    if (logger.isTraceEnabled()) {
        logger.trace("CallableStatement.execute() returned '" + retVal + "'");
        logger.trace("CallableStatement.getUpdateCount() returned " + updateCount);
    }
    Map<String, Object> resultsMap = createResultsMap();
    if (retVal || updateCount != -1) {
        resultsMap.putAll(extractReturnedResults(cs, updateCountParameters,
resultSetParameters, updateCount));
    }
    resultsMap.putAll(extractOutputParameters(cs, callParameters));
    return resultsMap;
});

Assert.state(result != null, "No result map");
return result;
}

```

上述代码的主要操作流程如下。

(1) 将方法参数 `declaredParameters` 根据不同数据类型分类存储到 `updateCountParameters`、`resultSetParameters` 和 `callParameters` 容器中。

(2) 调用 `execute` 方法执行获取处理结果。

先对第一个操作流程进行细节说明。当 SQL 语句中的参数类型是 `CallableStatement` `.getMoreResults/getUpdateCount` 方法的处理结果时,如果类型是 `SqlReturnResultSet` 将放入 `resultSetParameters` 容器中,其他类型放入 `updateCountParameters` 容器中。当 SQL 语句中的参数类型不是 `CallableStatement.getMoreResults/getUpdateCount` 方法的处理结果时,将数据放入 `callParameters` 容器中。

其次对第二个操作流程中的细节进行说明,在第二个操作流程中主要关注 `extractReturnedResults` 方法和 `extractOutputParameters` 方法的处理,下面将对这两个方法分别分析。

2.8.1 extractReturnedResults 方法分析

本节将对 `extractReturnedResults` 方法进行分析,具体处理代码如下。

```

protected Map<String, Object> extractReturnedResults(CallableStatement cs,
    @Nullable List<SqlParameter> updateCountParameters, @Nullable
List<SqlParameter> resultSetParameters,
    int updateCount) throws SQLException {

    Map<String, Object> results = new LinkedHashMap<>(4);
    int rsIndex = 0;
    int updateIndex = 0;
    boolean moreResults;
    if (!this.skipResultsProcessing) {
        do {
            if (updateCount == -1) {

```

```

        // 通过 processResultSet 处理数据值
        if (resultSetParameters != null && resultSetParameters.size() > rsIndex) {
            SqlReturnResultSet declaredRsParam = (SqlReturnResultSet)
resultSetParameters.get(rsIndex);
            results.putAll(processResultSet(cs.getResultSet(), declaredRsParam));
            rsIndex++;
        }
        else {
            if (!this.skipUndeclaredResults) {
                String rsName = RETURN_RESULT_SET_PREFIX + (rsIndex + 1);
                SqlReturnResultSet undeclaredRsParam = new
SqlReturnResultSet(rsName, getColumnMapRowMapper());
                if (logger.isTraceEnabled()) {
                    logger.trace("Added default SqlReturnResultSet parameter named '" +
rsName + "'");
                }
                results.putAll(processResultSet(cs.getResultSet(), undeclaredRsParam));
                rsIndex++;
            }
        }
    }
    else {
        // 非批量添加
        if (updateCountParameters != null && updateCountParameters.size() >
updateIndex) {
            SqlReturnUpdateCount ucParam = (SqlReturnUpdateCount)
updateCountParameters.get(updateIndex);
            String declaredUcName = ucParam.getName();
            results.put(declaredUcName, updateCount);
            updateIndex++;
        }
        else {
            if (!this.skipUndeclaredResults) {
                String undeclaredName = RETURN_UPDATE_COUNT_PREFIX + (updateIndex + 1);
                if (logger.isTraceEnabled()) {
                    logger.trace("Added default SqlReturnUpdateCount parameter named
'" + undeclaredName + "'");
                }
                results.put(undeclaredName, updateCount);
                updateIndex++;
            }
        }
    }
    moreResults = cs.getMoreResults();
    updateCount = cs.getUpdateCount();
    if (logger.isTraceEnabled()) {
        logger.trace("CallableStatement.getUpdateCount() returned " + updateCount);
    }
}
while (moreResults || updateCount != -1);
}
return results;
}

```


这段代码的主要目的是从参数中提取数据，具体的提取方式如下。

- (1) 从参数 `CallableStatement` 的结果集和参数 `resultSetParameters` 中提取。
- (2) 从参数 `CallableStatement` 的结果集和返回值前缀及索引提取。
- (3) 从参数 `updateCountParameters` 中配合索引提取。
- (4) 从更新值前缀和索引提取。

2.8.2 extractOutputParameters 方法分析

本节将对 `extractOutputParameters` 方法进行分析，具体处理代码如下。

```
protected Map<String, Object> extractOutputParameters(CallableStatement cs,
List<SqlParameter> parameters)
    throws SQLException {

    Map<String, Object> results = new LinkedHashMap<>(parameters.size());
    int sqlColIndex = 1;
    for (SqlParameter param : parameters) {
        if (param instanceof SqlOutParameter) {
            SqlOutParameter outParam = (SqlOutParameter) param;
            Assert.state(outParam.getName() != null, "Anonymous parameters not allowed");
            SqlReturnType returnType = outParam.getSqlReturnType();
            if (returnType != null) {
                Object out = returnType.getTypeValue(cs, sqlColIndex,
outParam.getSqlType(), outParam.getTypeName());
                results.put(outParam.getName(), out);
            }
            else {
                Object out = cs.getObject(sqlColIndex);
                if (out instanceof ResultSet) {
                    if (outParam.isResultSetSupported()) {
                        results.putAll(processResultSet((ResultSet) out, outParam));
                    }
                    else {
                        String rsName = outParam.getName();
                        SqlReturnResultSet rsParam = new SqlReturnResultSet(rsName,
getColumnMapRowMapper());
                        results.putAll(processResultSet((ResultSet) out, rsParam));
                        if (logger.isTraceEnabled()) {
                            logger.trace("Added default SqlReturnResultSet parameter named '" +
rsName + "'");
                        }
                    }
                }
            }
        }
        else {
            results.put(outParam.getName(), out);
        }
    }
    if (!(param.isResultsParameter())) {
        sqlColIndex++;
    }
}
```

```
    }  
    return results;  
}
```

在这段代码中，主要处理操作是将参数 `parameters` 进行重整，得到输出结果，在处理 `parameters` 的过程中只会对类型是 `SqlOutParameter` 的数据进行处理，处理逻辑如下。

- (1) 获取 SQL 语句的返回值类型，数据来源是 `SqlOutParameter` 对象。
- (2) 通过 `SqlReturnType` 获取数据值，将数据值和参数名称放入返回值容器。
- (3) 通过处理索引位从 `CallableStatement` 对象中获取数据值，将参数名称和数据值放入返回值容器中。

2.9 总结

本章围绕 `JdbcTemplate` 进行了相关分析，从 `JdbcTemplate` 出发先搭建了 `JdbcTemplate` 的测试环境，在测试环境搭建后对 `JdbcTemplate` 中的执行操作、查询操作和更新操作进行了源码分析，完成 `JdbcTemplate` 对象分析后进一步对 `JdbcTemplate` 对象中的 `PreparedStatementCreator` 接口和 `PreparedStatementSetter` 接口进行了分析。完成 `PreparedStatementCreator` 接口和 `PreparedStatementSetter` 接口分析后进行 `NamedParameterJdbcTemplate` 分析，在 `NamedParameterJdbcTemplate` 分析中包含了执行操作、查询操作和更新操作的分析，然后对 `SqlParameterSource` 接口进行了实现类相关分析。最后进行 `JdbcTemplate` 中的 `call` 方法分析。