03 规范兼容：JDBC 规范与 ShardingSphere 是什么关系?

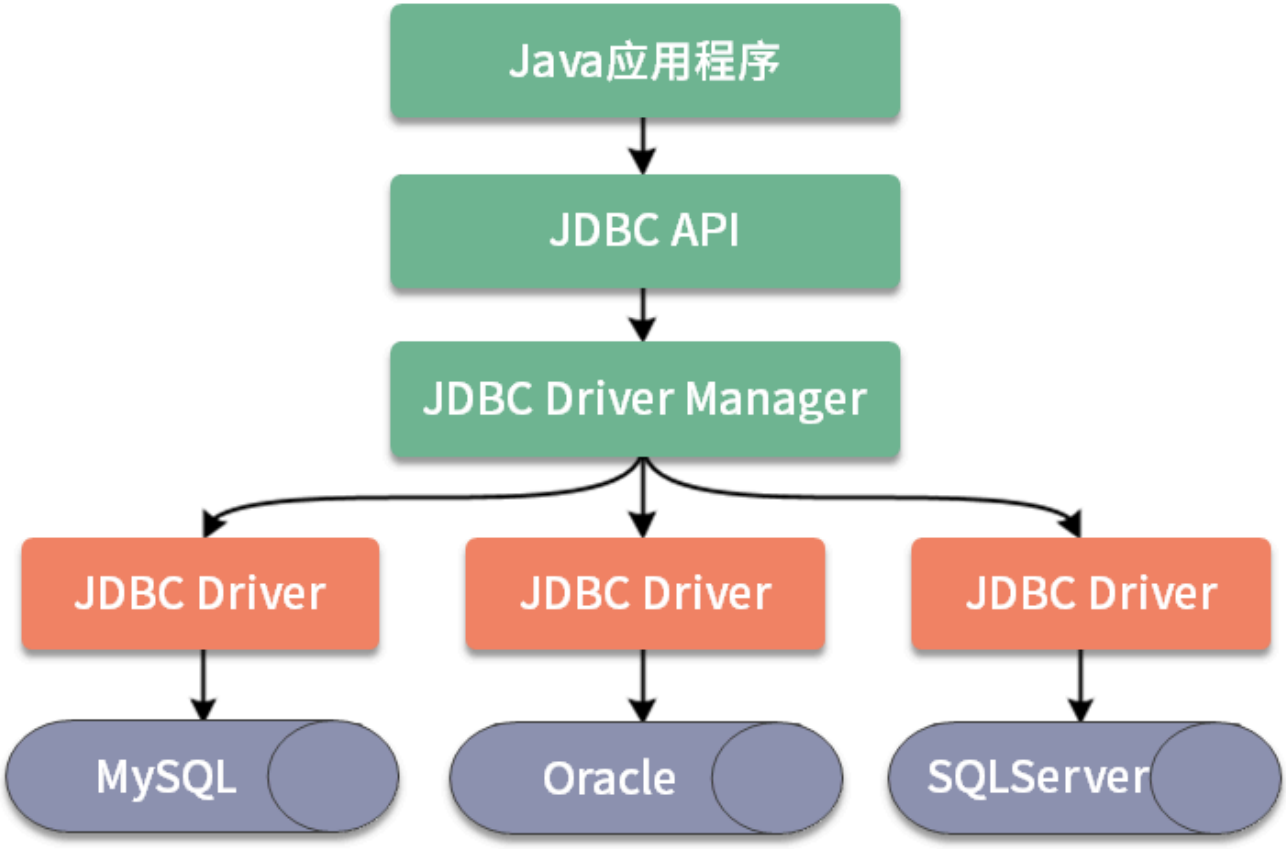
我们知道 ShardingSphere 是一种典型的客户端分片解决方案，而客户端分片的实现方式之一就是重写 JDBC 规范。在上一课时中，我们也介绍了，ShardingSphere 在设计上从一开始就完全兼容 JDBC 规范，ShardingSphere 对外暴露的一套分片操作接口与 JDBC 规范中所提供的接口完全一致。

讲到这里，你可能会觉得有点神奇，**ShardingSphere 究竟是通过什么方式，实现了与 JDBC 规范完全兼容的 API 来提供分片功能呢?**

这个问题非常重要，值得我们专门花一个课时的内容来进行分析和讲解。可以说，**理解 JDBC 规范以及 ShardingSphere 对 JDBC 规范的重写方式，是正确使用 ShardingSphere 实现数据分片的前提**。今天，我们就深入讨论 JDBC 规范与 ShardingSphere 的这层关系，帮你从底层设计上解开其中的神奇之处。

JDBC 规范简介

ShardingSphere 提供了与 JDBC 规范完全兼容的实现过程，在对这一过程进行详细展开之前，先来回顾一下 JDBC 规范。**JDBC（Java Database Connectivity）的设计初衷是提供一套用于各种数据库的统一标准**，而不同的数据库厂家共同遵守这套标准，并提供各自的实现方案供应用程序调用。作为统一标准，JDBC 规范具有完整的架构体系，如下图所示：



JDBC规范整体架构图

JDBC 架构中的 Driver Manager 负责加载各种不同的驱动程序（Driver），并根据不同的请求，向调用者返回相应的数据库连接（Connection）。而应用程序通过调用 JDBC API 来实现对数据库的操作。**对于开发人员而言，JDBC API 是我们访问数据库的主要途径，也是 ShardingSphere 重写 JDBC 规范并添加分片功能的入口**。如果我们使用 JDBC 开发一个访问数据库的处理流程，常见的代码风格如下所示：

```
// 创建池化的数据源
PooledDataSource dataSource = new PooledDataSource ();
// 设置MySQL Driver
dataSource.setDriver ("com.mysql.jdbc.Driver");
// 设置数据库URL、用户名和密码
dataSource.setUrl ("jdbc:mysql://localhost:3306/test");
dataSource.setUsername ("root");
dataSource.setPassword ("root");
// 获取连接
Connection connection = dataSource.getConnection();
// 执行查询
PreparedStatement statement = connection.prepareStatement ("select * from use
// 获取查询结果应该处理
ResultSet resultSet = statement.executeQuery();
while (resultSet.next()) {
        …
}
// 关闭资源
statement.close();
resultSet.close();
connection.close();
```

这段代码中包含了 JDBC API 中的核心接口，使用这些核心接口是我们基于 JDBC 进行数据访问的基本方式，这里有必要对这些接口的作用和使用方法做一些展开。事实上，随着课程内容的不断演进，你会发现在 ShardingSphere 中，完成日常开发所使用的也就是这些接口。

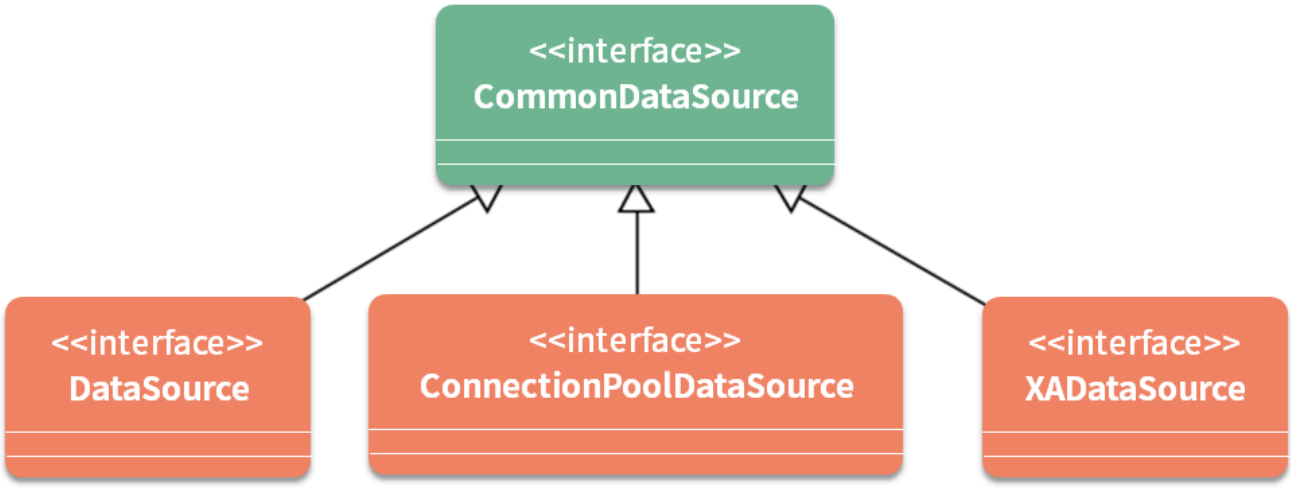
DataSource

DataSource 在 JDBC 规范中代表的是一种数据源，核心作用是获取数据库连接对象 Connection。在 JDBC 规范中，实际可以直接通过 DriverManager 获取 Connection。我们知道获取 Connection 的过程需要建立与数据库之间的连接，而这个过程会产生较大的系统开销。

为了提高性能，通常会建立一个中间层，该中间层将 DriverManager 生成的 Connection 存放到连接池中，然后从池中获取 Connection，可以认为，DataSource 就是这样一个中间层。在日常开发过程中，我们通常都会基于 DataSource 来获取 Connection。而在 ShardingSphere 中，暴露给业务开发人员的同样是一个经过增强的 DataSource 对象。DataSource 接口的定义是这样的：

```
public interface DataSource  extends CommonDataSource, Wrapper {
  Connection getConnection() throws SQLException;
  Connection getConnection(String username, String password)
    throws SQLException;
}
```

可以看到，DataSource 接口提供了两个获取 Connection 的重载方法，并继承了 CommonDataSource 接口，该接口是 JDBC 中关于数据源定义的根接口。除了 DataSource 接口之外，它还有两个子接口：

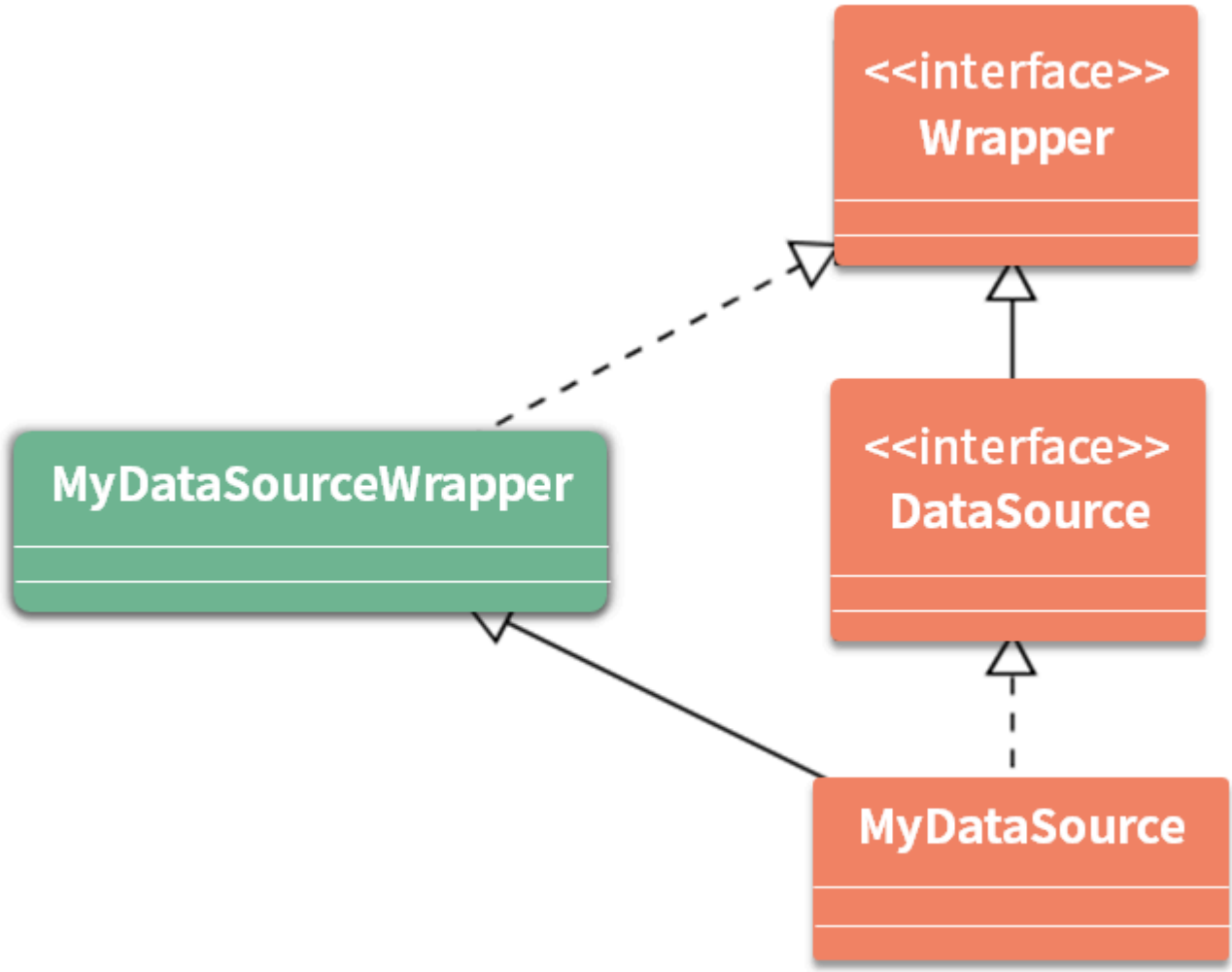


CommonDataSource 类层结构图

其中，DataSource 是官方定义的获取 Connection 的基础接口，ConnectionPoolDataSource 是从连接池 ConnectionPool 中获取的 Connection 接口。而 XADataSource 则用来实现在分

布式事务环境下获取 Connection，我们在讨论 ShardingSphere 的分布式事务时会接触到这个接口。

请注意，DataSource 接口同时还继承了一个 Wrapper 接口。从接口的命名上看，可以判断该接口应该起到一种包装器的作用，事实上，由于很多数据库供应商提供了超越标准 JDBC API 的扩展功能，所以，Wrapper 接口可以把一个由第三方供应商提供的、非 JDBC 标准的接口包装成标准接口。以 DataSource 接口为例，如果我们想要实现自己的数据源 MyDataSource，就可以提供一个实现了 Wrapper 接口的 MyDataSourceWrapper 类来完成包装和适配：



Wrapper 接口的使用方式图

在 JDBC 规范中，除了 DataSource 之外，Connection、Statement、ResultSet 等核心对象也都继承了这个接口。显然，ShardingSphere 提供的就是非 JDBC 标准的接口，所以也应该会用到这个 Wrapper 接口，并提供了类似的实现方案。

Connection

DataSource 的目的是获取 Connection 对象，我们可以把 Connection 理解为一种**会话（Session）机制**。Connection 代表一个数据库连接，负责完成与数据库之间的通信。所有 SQL 的执行都是在某个特定 Connection 环境中进行的，同时它还提供了一组重载方法，分别用于创建 Statement 和 PreparedStatement。另一方面，Connection 也涉及事务相关的操

作，为了实现分片操作，ShardingSphere 同样也实现了定制化的 Connection 类 ShardingConnection。

Statement

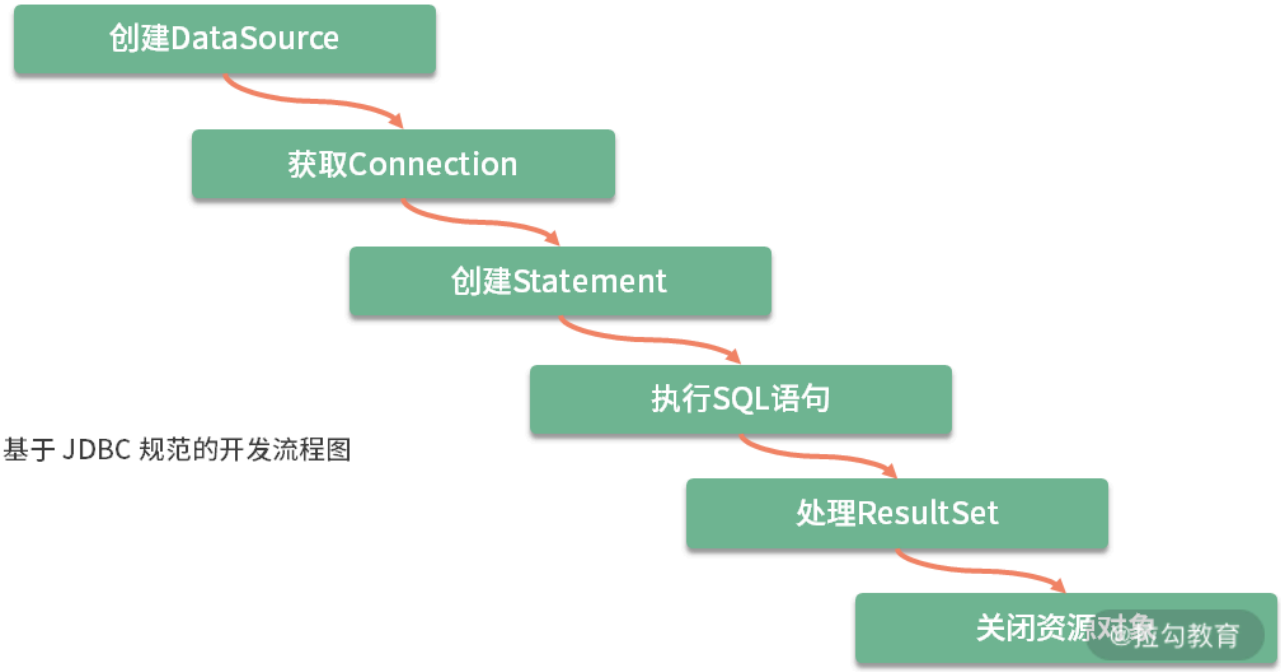
JDBC 规范中的 Statement 存在两种类型，一种是**普通的 Statement**，一种是**支持预编译的 PreparedStatement**。所谓预编译，是指数据库的编译器会对 SQL 语句提前编译，然后将预编译的结果缓存到数据库中，这样下次执行时就可以替换参数并直接使用编译过的语句，从而提高 SQL 的执行效率。当然，这种预编译也需要成本，所以在日常开发中，对数据库只执行一次性读写操作时，用 Statement 对象进行处理比较合适；而当涉及 SQL 语句的多次执行时，可以使用 PreparedStatement。

如果需要查询数据库中的数据，只需要调用 Statement 或 PreparedStatement 对象的 executeQuery 方法即可，该方法以 SQL 语句作为参数，执行完后返回一个 JDBC 的 ResultSet 对象。当然，Statement 或 PreparedStatement 中提供了一大批执行 SQL 更新和查询的重载方法。在 ShardingSphere 中，同样也提供了 ShardingStatement 和 ShardingPreparedStatement 这两个支持分片操作的 Statement 对象。

ResultSet

一旦通过 Statement 或 PreparedStatement 执行了 SQL 语句并获得了 ResultSet 对象后，那么就可以通过调用 Resulset 对象中的 next() 方法遍历整个结果集。如果 next() 方法返回为 true，就意味结果集中存在数据，则可以调用 ResultSet 对象的一系列 getXXX() 方法来取得对应的结果值。对于分库分表操作而言，因为涉及从多个数据库或数据表中获取目标数据，势必需要对获取的结果进行归并。因此，ShardingSphere 中也提供了分片环境下的 ShardingResultSet 对象。

作为总结，我们梳理了基于 JDBC 规范进行数据库访问的开发流程图，如下图所示：



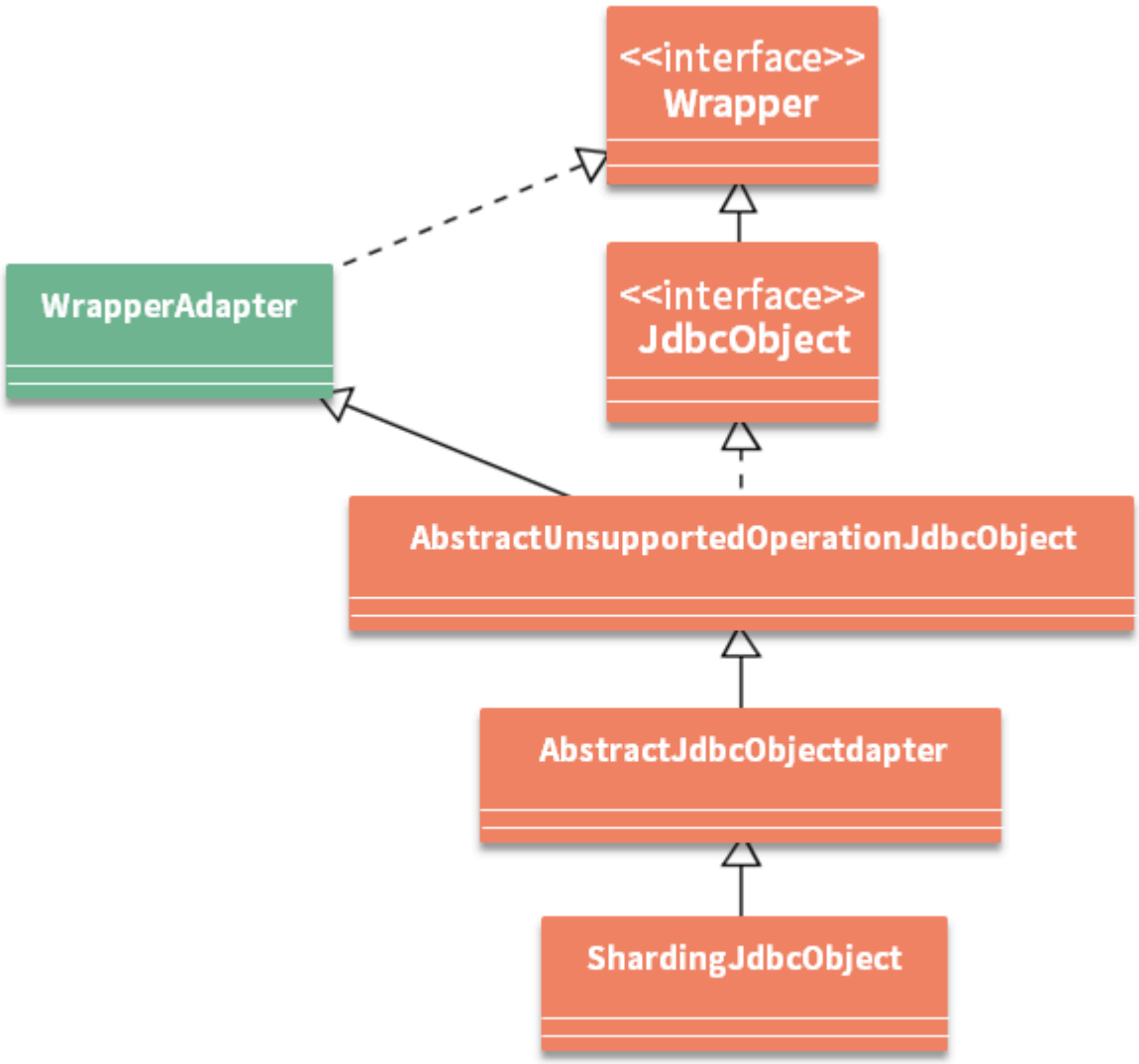
基于 JDBC 规范的开发流程图

ShardingSphere 提供了与 JDBC 规范完全兼容的 API。也就是说，开发人员可以基于这个开发流程和 JDBC 中的核心接口完成分片引擎、数据脱敏等操作，我们来看一下。

基于适配器模式的 JDBC 重写实现方案

在 ShardingSphere 中，实现与 JDBC 规范兼容性的基本策略就是采用了设计模式中的适配器模式（Adapter Pattern）。**适配器模式通常被用作连接两个不兼容接口之间的桥梁，涉及为某一个接口加入独立的或不兼容的功能。**

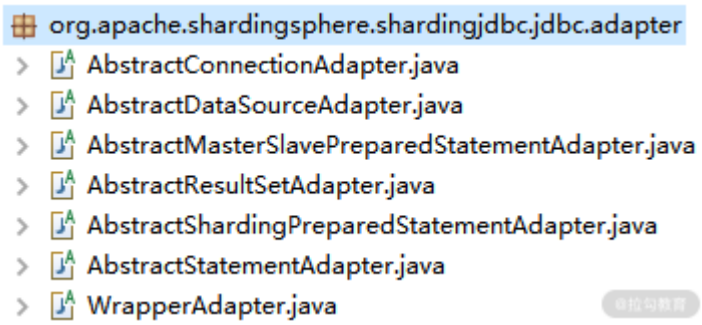
作为一套适配 JDBC 规范的实现方案，ShardingSphere 需要对上面介绍的 JDBC API 中的 DataSource、Connection、Statement 及 ResultSet 等核心对象都完成重写。虽然这些对象承载着不同功能，但重写机制应该是共通的，否则就需要对不同对象都实现定制化开发，显然，这不符合我们的设计原则。为此，ShardingSphere 抽象并开发了一套基于适配器模式的实现方案，整体结构是这样的，如下图所示：



ShardingSphere 基于适配器模式的实现方案图

首先，我们看到这里有一个 JdbcObject 接口，这个接口泛指 JDBC API 中的 DataSource、Connection、Statement 等核心接口。前面提到，这些接口都继承自包装器 Wrapper 接口。

ShardingSphere 为这个 Wrapper 接口提供了一个实现类 WrapperAdapter，这点在图中得到了展示。在 ShardingSphere 代码工程 sharding-jdbc-core 的 org.apache.shardingsphere.shardingjdbc.jdbc.adapter 包中包含了所有与 Adapter 相关的实现类：



在 ShardingSphere 基于适配器模式的实现方案图的底部，有一个 ShardingJdbcObject 类的定义。这个类也是一种泛指，代表 ShardingSphere 中用于分片的 ShardingDataSource、ShardingConnection、ShardingStatement 等对象。

最后发现 ShardingJdbcObject 继承自一个 AbstractJdbcObjectAdapter，而 AbstractJdbcObjectAdapter 又继承自 AbstractUnsupportedOperationJdbcObject，这两个类都是抽象类，而且也都泛指一组类。两者的区别在于，AbstractJdbcObjectAdapter 只提供了针对 JdbcObject 接口的一部分实现方法，这些方法是我们完成分片操作所需要的。而对于那些我们不需要的方法实现，则全部交由 AbstractUnsupportedOperationJdbcObject 进行实现，这两个类的所有方法的合集，就是原有 JdbcObject 接口的所有方法定义。

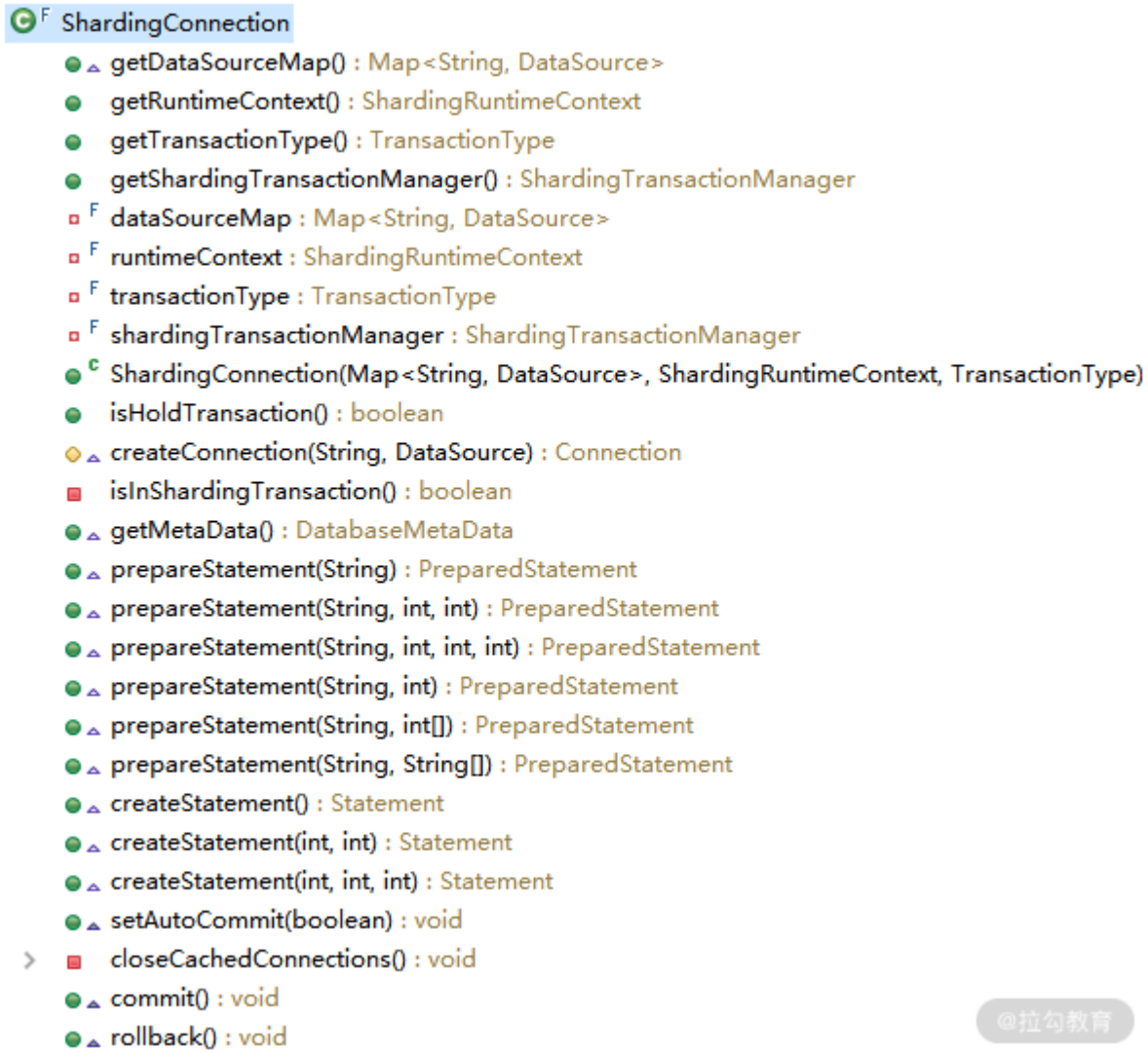
这样，我们大致了解了 ShardingSphere 对 JDBC 规范中核心接口的重写机制。这个重写机制非常重要，在 ShardingSphere 中应用也很广泛，我们可以通过示例对这一机制做进一步理解。

ShardingSphere 重写 JDBC 规范示例：ShardingConnection

通过前面的介绍，我们知道 ShardingSphere 的分片引擎中提供了一系列 ShardingJdbcObject 来支持分片操作，包括 ShardingDataSource、ShardingConnection、ShardingStatement、ShardingPreparedStament 等。这里以最具代表性的 ShardingConnection 为例，来讲解它的实现过程。请注意，今天我们关注的还是重写机制，不会对 ShardingConnection 中的具体功能以及与其他类之间的交互过程做过多展开讲解。

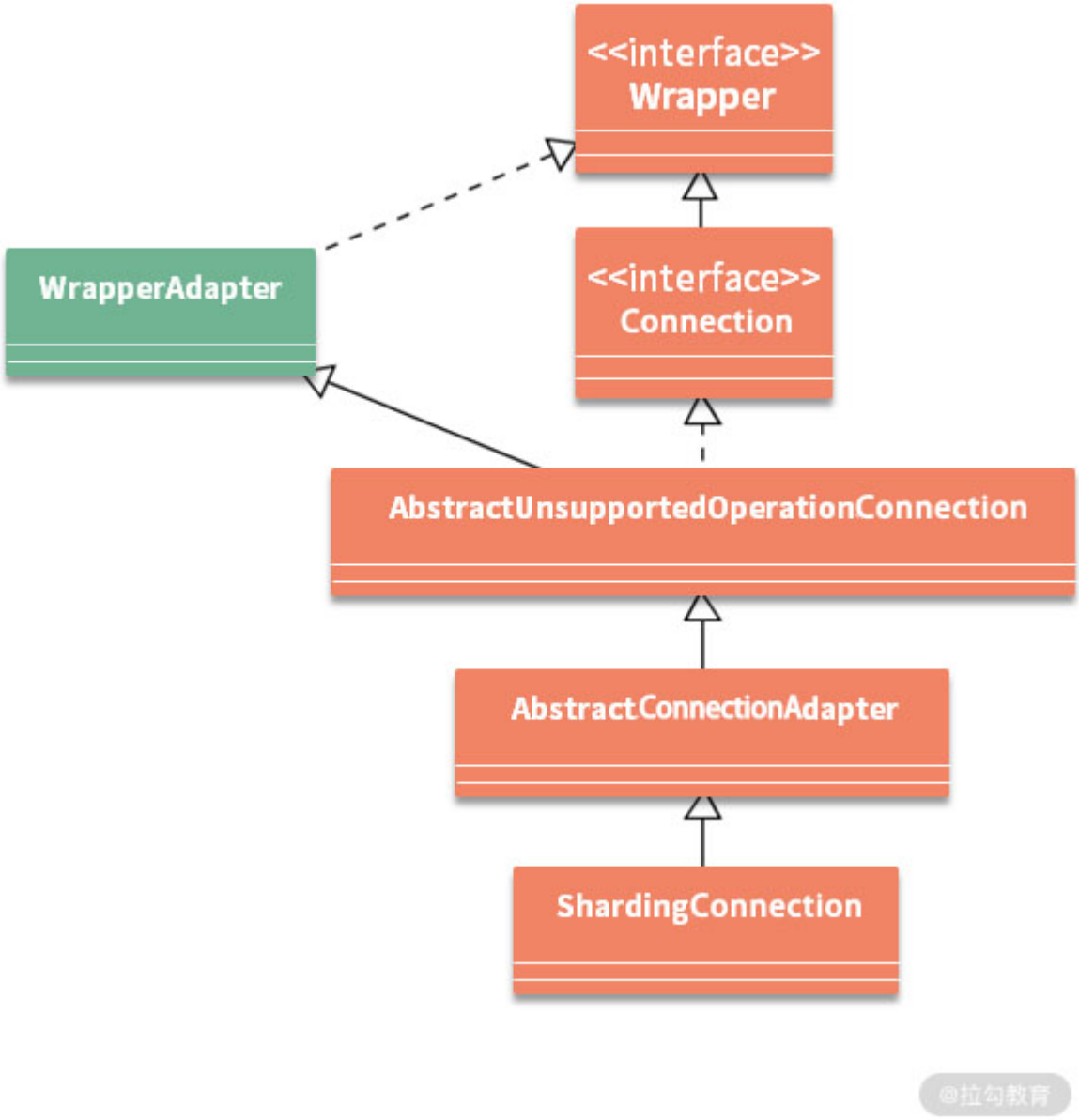
ShardingConnection 类层结构

ShardingConnection 是对 JDBC 中 Connection 的适配和包装，所以它需要提供 Connection 接口中定义的方法，包括 createConnection、getMetaData、各种重载的 prepareStatement 和 createStatement 以及针对事务的 setAutoCommit、commit 和 rollback 方法等。ShardingConnection 对这些方法都进行了重写，如下图所示：



ShardingConnection 中的方法列表图

ShardingConnection 类的一条类层结构支线就是适配器模式的具体应用，这部分内容的类层结构与前面介绍的重写机制的类层结构是完全一致的，如下图所示：



AbstractConnectionAdapter

我们首先来看看 AbstractConnectionAdapter 抽象类，ShardingConnection 直接继承了它。在 AbstractConnectionAdapter 中发现了一个 cachedConnections 属性，它是一个 Map 对象，该对象其实缓存了这个经过封装的 ShardingConnection 背后真实的 Connection 对象。如果我们对一个 AbstractConnectionAdapter 重复使用，那么这些 cachedConnections 也会一直被缓存，直到调用 close 方法。可以从 AbstractConnectionAdapter 的 getConnections 方法中理解具体的操作过程：

```
public final List<Connection> getConnections(final ConnectionMode connectionM
        //获取DataSource
        DataSource dataSource = getDataSourceMap().get(dataSourceName);
        Preconditions.checkState(null != dataSource, "Missing the data source
        Collection<Connection> connections;

        //根据数据源从cachedConnections中获取connections
        synchronized (cachedConnections) {
            connections = cachedConnections.get(dataSourceName);
        }
```

```
        //如果connections多于想要的connectionSize，则只获取所需部分
        List<Connection> result;
        if (connections.size() >= connectionSize) {
            result = new ArrayList<>(connections).subList(0, connectionSize);
        } else if (!connections.isEmpty()) {//如果connections不够
            result = new ArrayList<>(connectionSize);
            result.addAll(connections);
            //创建新的connections
            List<Connection> newConnections = createConnections(dataSourceNam
            result.addAll(newConnections);
            synchronized (cachedConnections) {
                //将新创建的connections也放入缓存中进行管理
                cachedConnections.putAll(dataSourceName, newConnections);
            }
        } else {//如果缓存中没有对应dataSource的Connections，同样进行创建并放入缓存中
            result = new ArrayList<>(createConnections(dataSourceName, connec
            synchronized (cachedConnections) {
                cachedConnections.putAll(dataSourceName, result);
            }
        }
        return result;
    }
```

这段代码有三个判断，流程上比较简单，参考注释即可，需要关注的是其中的 createConnections 方法：

```
    private List<Connection> createConnections(final String dataSourceName, final
        if (1 == connectionSize) {
            Connection connection = createConnection(dataSourceName, dataSour
            replayMethodsInvocation(connection);
            return Collections.singletonList(connection);
        }
        if (ConnectionMode.CONNECTION_STRICTLY == connectionMode) {
            return createConnections(dataSourceName, dataSource, connectionSi
        }
        synchronized (dataSource) {
            return createConnections(dataSourceName, dataSource, connectionSi
        }
    }
```

这段代码涉及了 ConnectionMode（连接模式），这是 ShardingSphere 执行引擎中的重要概念，今天我们先不展开，将在第 21 课时“执行引擎：分片环境下SQL执行的整体流程应该如何进行抽象？”中详细讲解。这里，可以看到 createConnections 方法批量调用了一个 createConnection 抽象方法，该方法需要 AbstractConnectionAdapter 的子类进行实现：

```
    protected abstract Connection createConnection(String dataSourceName, DataSou
```

同时，我们看到对于创建的 Connection 对象，都需要执行这样一个语句：

```
replayMethodsInvocation(connection);
```

这行代码比较难以理解，让我们来到定义它的地方，即 WrapperAdapter 类。

WrapperAdapter

从命名上看，WrapperAdapter 是一个包装器的适配类，实现了 JDBC 中的 Wrapper 接口。我们在该类中找到了这样一对方法定义：

```
    //记录方法调用
    public final void recordMethodInvocation(final Class<?> targetClass, fina
        jdbcMethodInvocations.add(new JdbcMethodInvocation(targetClass.getMet
        }

        //重放方法调用
    public final void replayMethodsInvocation(final Object target) {
        for (JdbcMethodInvocation each : jdbcMethodInvocations) {
            each.invoke(target);
        }
        }
```

这两个方法都用到了 JdbcMethodInvocation 类：

```
public class JdbcMethodInvocation {

    @Getter
    private final Method method;

    @Getter
    private final Object[] arguments;

    public void invoke(final Object target) {
        method.invoke(target, arguments);
    }
}
```

显然，JdbcMethodInvocation 类中用到了反射技术根据传入的 method 和 arguments 对象执行对应方法。

了解了 JdbcMethodInvocation 类的原理后，我们就不难理解 recordMethodInvocation 和 replayMethodsInvocation 方法的作用。其中，recordMethodInvocation 用于记录需要执行的方法和参数，而 replayMethodsInvocation 则根据这些方法和参数通过反射技术进行执行。

对于执行 replayMethodsInvocation，我们必须先找到 recordMethodInvocation 的调用入口。通过代码的调用关系，可以看到在 AbstractConnectionAdapter 中对其进行了调用，具体来说就是 setAutoCommit、setReadOnly 和 setTransactionIsolation 这三处方法。这里以 setReadOnly 方法为例给出它的实现：

```
    @Override
    public final void setReadOnly(final boolean readOnly) throws SQLException
        this.readOnly = readOnly;
        //调用recordMethodInvocation方法记录方法调用的元数据
        recordMethodInvocation(Connection.class, "setReadOnly", new Class[]{b

        //执行回调
        forceExecuteTemplate.execute(cachedConnections.values(), new ForceExe

            @Override
            public void execute(final Connection connection) throws SQLExcept
                connection.setReadOnly(readOnly);
            }
        });
    }
```

AbstractUnsupportedOperationConnection

另一方面，从类层关系上，可以看到 AbstractConnectionAdapter 直接继承的是 AbstractUnsupportedOperationConnection 而不是 WrapperAdapter，而在 AbstractUnsupportedOperationConnection 中都是一组直接抛出异常的方法。这里截取部分代码：

```
public abstract class AbstractUnsupportedOperationConnection extends WrapperA

    @Override
    public final CallableStatement prepareCall(final String sql) throws SQLEx
        throw new SQLFeatureNotSupportedException("prepareCall");
    }

    @Override
    public final CallableStatement prepareCall(final String sql, final int re
        throw new SQLFeatureNotSupportedException("prepareCall");
}
…
}
```

AbstractUnsupportedOperationConnection 这种处理方式的目的就是明确哪些操作是 AbstractConnectionAdapter 及其子类 ShardingConnection 所不能支持的，属于职责分离的一种具体实现方法。

小结

JDBC 规范是理解和应用 ShardingSphere 的基础，ShardingSphere 对 JDBC 规范进行了重写，并提供了完全兼容的一套接口。在这一课时中，我们首先给出了 JDBC 规范中各个核心接口的介绍；正是在这些接口的基础上，ShardingSphere 基于适配器模式对 JDBC 规范进行了重写；我们对这一重写方案进行了抽象，并基于 ShardingConnectin 类的实现过程详细阐述了 ShardingSphere 重写 Connection 接口的源码分析。

这里给你留一道思考题：ShardingSphere如何基于适配器模式实现对JDBC中核心类的重写?

JDBC 规范与 ShardingSphere 的兼容性概念至关重要。在掌握了这个概念之后，下一课时将介绍应用集成方面的话题，即在业务系统中使用 ShardingSphere 的具体方式。

上一页 下一页