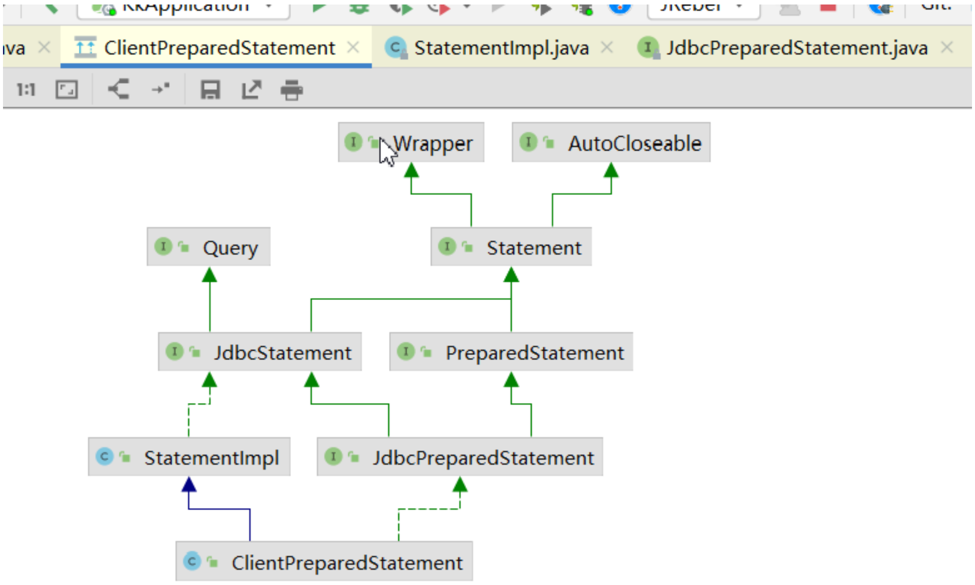
关于PreparedStatement

MySQL驱动包中提供了JdbcPreparedStatement 接口作为PreparedStatement子接口

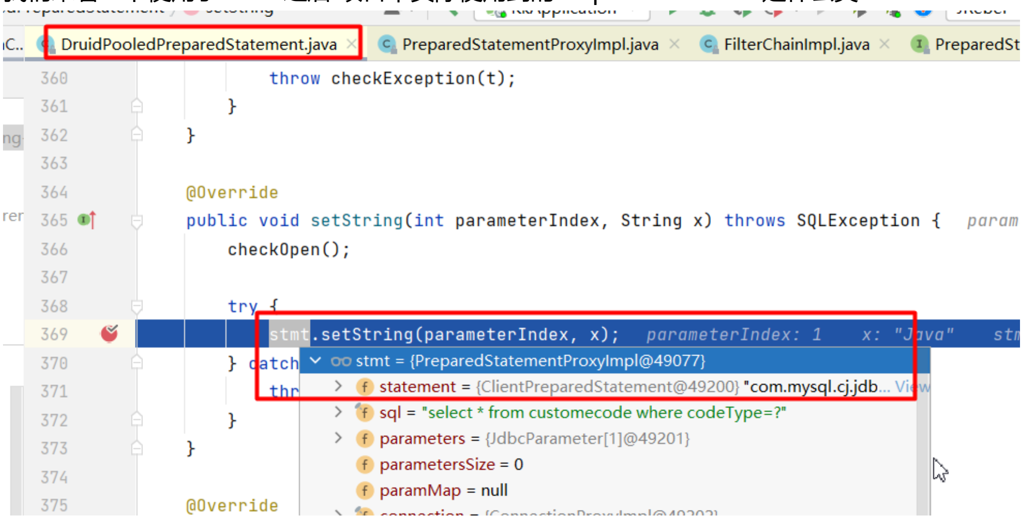
public interface JdbcPreparedStatement extends java.sql.PreparedStatement, JdbcStatement；

其中JdbcStatement也是MySQL驱动包中的

public interface JdbcStatement extends java.sql.Statement, Query {}



我们来看一下使用了Druid之后 项目中实际使用到的PrepareStatement是什么类


DruidPooledPreparedStatement.java
PreparedStatementProxyImpl.java
FilterChainImpl.java
PreparedSt

```
            throw checkException(t);
        }
    }

    @Override
    public void setString(int parameterIndex, String x) throws SQLException {   param
        checkOpen();

        try {
            stmt.setString(parameterIndex, x);   parameterIndex: 1    x: "Java"    stm
        } catch
            thr
        }
    }

    @Override
```

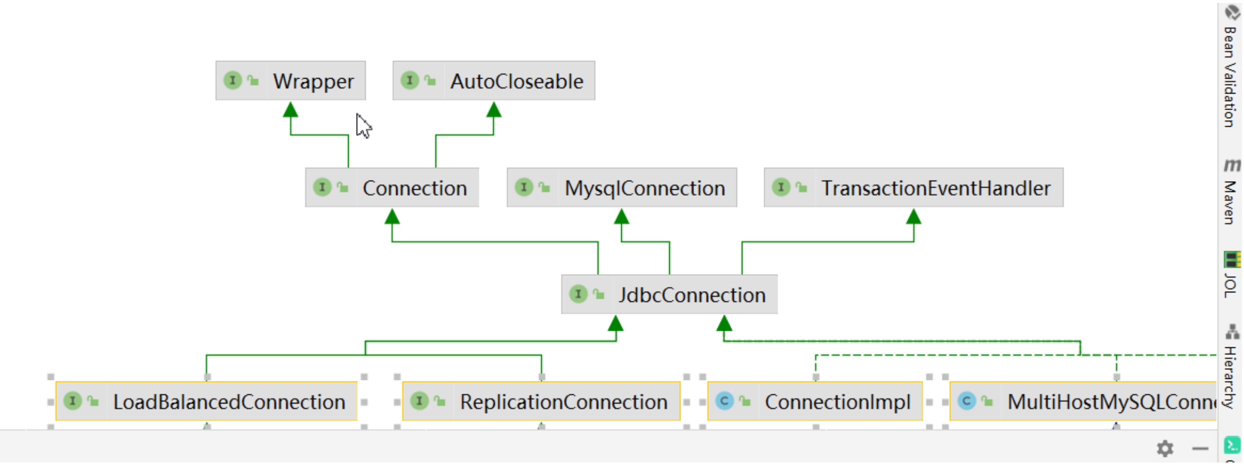
```
stmt = {PreparedStatementProxyImpl@49077}
  statement = {ClientPreparedStatement@49200} "com.mysql.cj.jdb... View
  sql = "select * from customecode where codeType=?"
  parameters = {JdbcParameter[1]@49201}
  parametersSize = 0
  paramMap = null
```

Druid包中提供了 PreparedStatementProxyImpl作为 PreparedStatement，但是PreparedStatementProxyImpl并不实际实现PreparedStatement的功能。 PreparedStatementProxyImpl中存在成员属性PreparedStatement，这个属性引用的对象是mysql驱动包中的ClientPreparedStatement。

但是Druid对外开放的不是PreparedStatementProxyImpl ，而是DruidPooledPreparedStatement， 也就是Mybatis中持有的PrepareStatement就是 DruidPooledPreparedStatement，这个DruidPooledPreparedStatement内存在成员属性PreparedStatement，这个成员属性引用的对象就是PreparedStatementProxyImpl

关于Connection

出了Sql中定义的PreparedStatement之外我们 还会关心Sql包中定义的Connection接口。 在MySQL的jdbc驱动包中 存在如下类层次结构。



从上面的 图片继承体系中我们看到对于MySQL数据库驱动而言，他提供了JdbcConnection接口作为Connection接口的子类。 在Jdbc的驱动中使用ConnectionImpl实现类作为Connection的实现类。
Sql包中定义了Connection接口，JDBC驱动中提供了JDBCConnection作为子接口，驱动中提供了ConnectionImpl 作为Connection的最终实现类。

关于DataSource
Sql包提供了DataSource接口用于获取Connection，Druid中提供了DruidDataSource作为实现类。
DruidDataSource同时实现了ConnectionPoolDataSource接口。
ConnectionPoolDataSource是sql包提供的接口。 这个ConnectionPoolDataSource接口中提供了 PooledConnection getPooledConnection()，返回值不是Connection，而是sql包中的PooledConnection。
因此DruidDataSource 具有池化思想。具体如何体现池的呢？

DruidDataSource中存在如下属性：

```
private volatile DruidConnectionHolder[] connections;
private DruidConnectionHolder[] evictConnections;
private DruidConnectionHolder[] keepAliveConnections;
```

在DruidDataSource的int方法中会执行初始化连接池。
比如：

```
 ds = new DruidDataSource();
 ds.setPassword(password);
 ds.setDriverClassName(driverName);
 druidDataSourceConfigUtils.
 fillDruidDataSourceParams(ds,
 BafConstants.DRUID_MAX_ACTIVE, BafConstants.DRUID_MAX_WAITE, true);
 //init方法会执行DruidDataSource的初始化方法，这会导致为池创建connection
 ds.init();
```

在init中initialSize表示连接池初始化大小

```
for (int i = 0; i < initialSize; ++i) {
//提交任务创建connection 存放在池中
submitCreateTask(true);
}
```

在submitCreateTask中使用CreateConnectionTask 来创建connection，所以核心创建connection的逻辑在于CreateConnectionTask

```
public Connection createPhysicalConnection(String url, Properties info)
throws SQLException {
Connection conn;
if (getProxyFilters().size() == 0) {
conn = getDriver().connect(url, info);
} else {
conn = new FilterChainImpl(this).connection_connect(info);
}
createCountUpdater.incrementAndGet(this);
 return conn;
 }
```

在创建物理连接的逻辑中我们看到如果 getProxyFilters为空，则直接使用 conn = getDriver().connect(url, info);创建连接。

getDriver 是返回DruidAbstractDataSource 中的Driver属性 这个Driver属性就是com.mysql.jdbc.Driver，这是Mysql驱动的类。也就是说使用了 Driver类中的connect方法返回connection

public java.sql.Connection connect(String url, Properties info) throws SQLException

如果getProxyFilter不为空，则 conn = new FilterChainImpl(this).connection_connect(info);

创建FilterChainIMpl的时候传入了this，通过this可以得到ProxyFilter

Druid中的FilterChainImpl

在FilterChainImpl的 connection_connect方法中 首先 获取nextFilter，然后执行filter的connection_connect

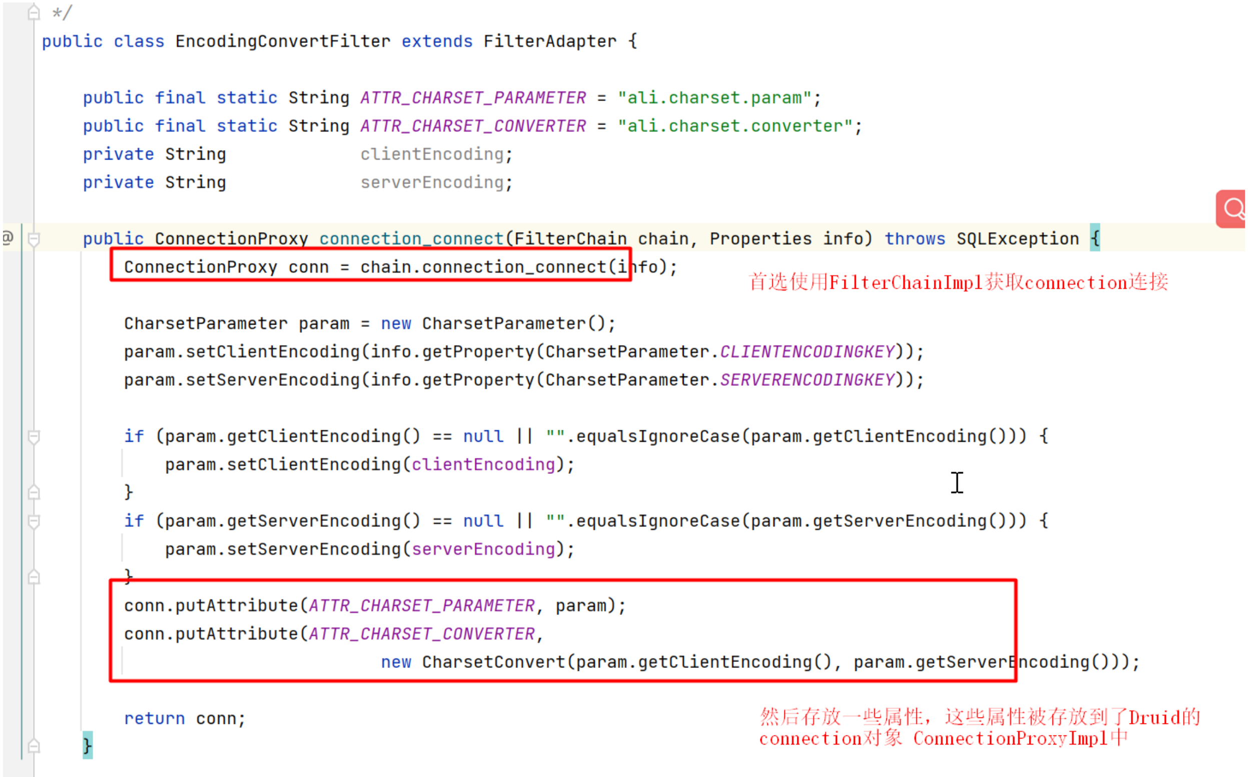
```
public ConnectionProxy connection_connect(Properties info)
throws SQLException {
 /**
 *注意this是调用链对象FilterChainImpl。 如果当前pos小于filterSize，
 * 则会使用nextFilter获取下一个filter，其中nextFilter会执行pos++
 * 然后得到nextFilter之后执行其 connection_connect方法。
 * 从概念上说FilterChainImpl 也是Filter接口的实现类。
 因此FilterChainImpl和 nextFilter都有connection_connect方法
 *
  *
  * 我们知道在SpringMvc的filter中， 我们一般会首先执行filter的逻辑
  然后filter的逻辑执行完了之后才会执行
  filterChain.doFilter(request, response);
```

```
* 但是在Druid的filter中，比如statFilter中，
他首先执行的是 connection = chain.connection_connect(info);
* 其中chain就是当前的FilterChainImpl对象，
因此执行逻辑又进入当前connection_connect方法中
*
* 因此if 中的逻辑会执行到最后一个Filter，
在最后一个Filter中执行 chain.connection_connect
获取到连接connection，然后再执行Filter中的其他逻辑，比如对connection
* 进行包装、修改等。
*
*
*
*/
if (this.pos < filterSize) {
return nextFilter()
.connection_connect(this, info);
}

/**
* 经过上面的分析我们知道 在先执行Filter的逻辑之前会
先执行chain.connection_connect(info)
* 也就是下面的逻辑，这段逻辑 dataSource.getRawDriver
会返回我们配置文件中指定的驱动类 也就是驱动包中的
* com.mysql.cj.jdbc.Driver
*/
Driver driver = dataSource.getRawDriver();
String url = dataSource.getRawJdbcUrl();

/**
* 使用Driver获取connection。 在MySQL中驱动就
是com.mysql.cj.jdbc.Driver， connection就是ConnectionImpl
*/
Connection nativeConnection = driver.connect(url, info);

if (nativeConnection == null) {
return null;
}

/**
* 但是在这里我们发现Druid返回的并不是原生MySQL驱动中
```

```
的Connection对象ConnectionImpl，而是返回了Druid中的ConnectionProxyImpl
*/
return new ConnectionProxyImpl(dataSource, nativeConnection,
info, dataSource.createConnectionId());
}
```

假设我们的Filter只有StatFilter，其实现如下， connection_connect 方法的第一个参数 FilterChain在上面传入的是this，也就是FilterChainImpl对象， 在下面的方法中我们看到connection_connect内部首先是执行了chain.connection_connect，因此这将会执行FilterChainImpl中跳过if (this.pos < filterSize) 之后的逻辑，也就是使用driver.connect(url, info);创建了connection，然后在statFilter中就拿到了物理connection。
值得注意到是hi FilterChainImpl中 使用Driver.connect创建connection之后 并不是直接返回原生的物理connection，而是返回了ConnectionProxyImpl， return new ConnectionProxyImpl(dataSource, nativeConnection, info, dataSource.createConnectionId());
因此StatFilter中实际获取到的connection是ConnectionProxyImpl。

为什么返回的是ConnectionProxyImpl 呢而不是原生的ConnectionImpl对象呢?
因为在有些Filter的connection_connect 方法中使用 FilterChainImpl 获取到connection后会执行一些逻辑： 存放一些属性，比如EncodingConvertFilter的connection_connect

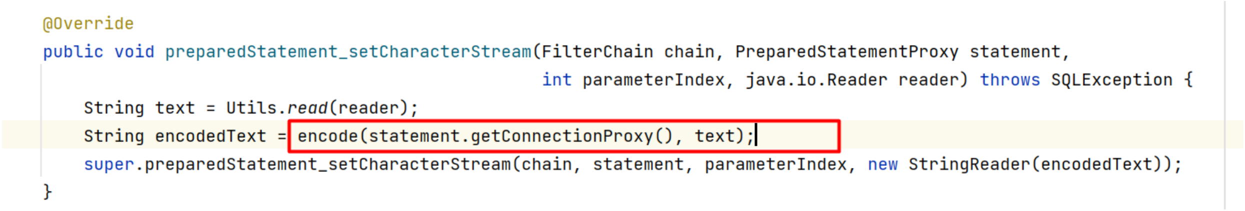


```
 */
public class EncodingConvertFilter extends FilterAdapter {

    public final static String ATTR_CHARSET_PARAMETER = "ali.charset.param";
    public final static String ATTR_CHARSET_CONVERTER = "ali.charset.converter";
    private String             clientEncoding;
    private String             serverEncoding;

    public ConnectionProxy connection_connect(FilterChain chain, Properties info) throws SQLException {
        ConnectionProxy conn = chain.connection_connect(info);

        CharsetParameter param = new CharsetParameter();
        param.setClientEncoding(info.getProperty(CharsetParameter.CLIENTENCODINGKEY));
        param.setServerEncoding(info.getProperty(CharsetParameter.SERVERENCODINGKEY));

        if (param.getClientEncoding() == null || "".equalsIgnoreCase(param.getClientEncoding())) {
            param.setClientEncoding(clientEncoding);
        }
        if (param.getServerEncoding() == null || "".equalsIgnoreCase(param.getServerEncoding())) {
            param.setServerEncoding(serverEncoding);
        }
        conn.putAttribute(ATTR_CHARSET_PARAMETER, param);
        conn.putAttribute(ATTR_CHARSET_CONVERTER,
                          new CharsetConvert(param.getClientEncoding(), param.getServerEncoding()));

        return conn;
    }
```

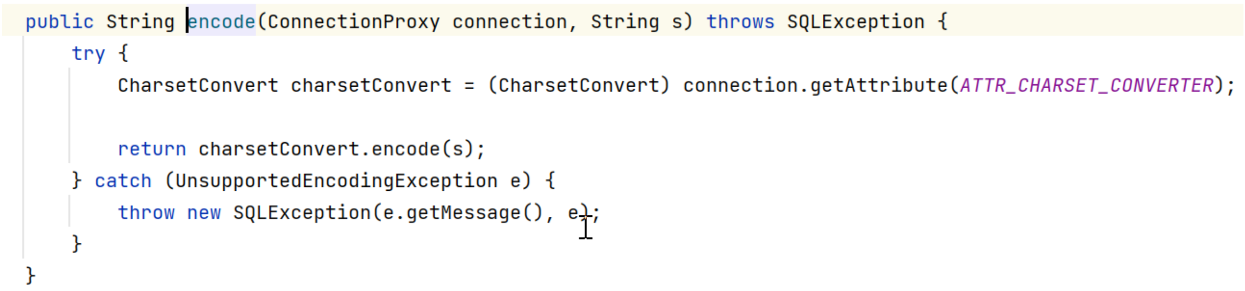
首选使用FilterChainImpl获取connection连接

然后存放一些属性，这些属性被存放到了Druid的connection对象 ConnectionProxyImpl中

ConnectionProxyImpl中通过putAttribute设置的属性有什么作用呢？ 在FilterChainImpl的preparedStatement_setCharacterStream方法中就会调用每一个filter的preparedStatement_setCharacterStream，在com.alibaba.druid.filter.encoding.EncodingConvertFilter#preparedStatement_setCharacterStream的方法中就会从ConnectinProxyImp中getAttribute，然后用取出的convert charsetConvert.encode(s);



```
@Override
public void preparedStatement_setCharacterStream(FilterChain chain, PreparedStatementProxy statement,
                                                 int parameterIndex, java.io.Reader reader) throws SQLException {
    String text = Utils.read(reader);
    String encodedText = encode(statement.getConnectionProxy(), text);
    super.preparedStatement_setCharacterStream(chain, statement, parameterIndex, new StringReader(encodedText));
}
```



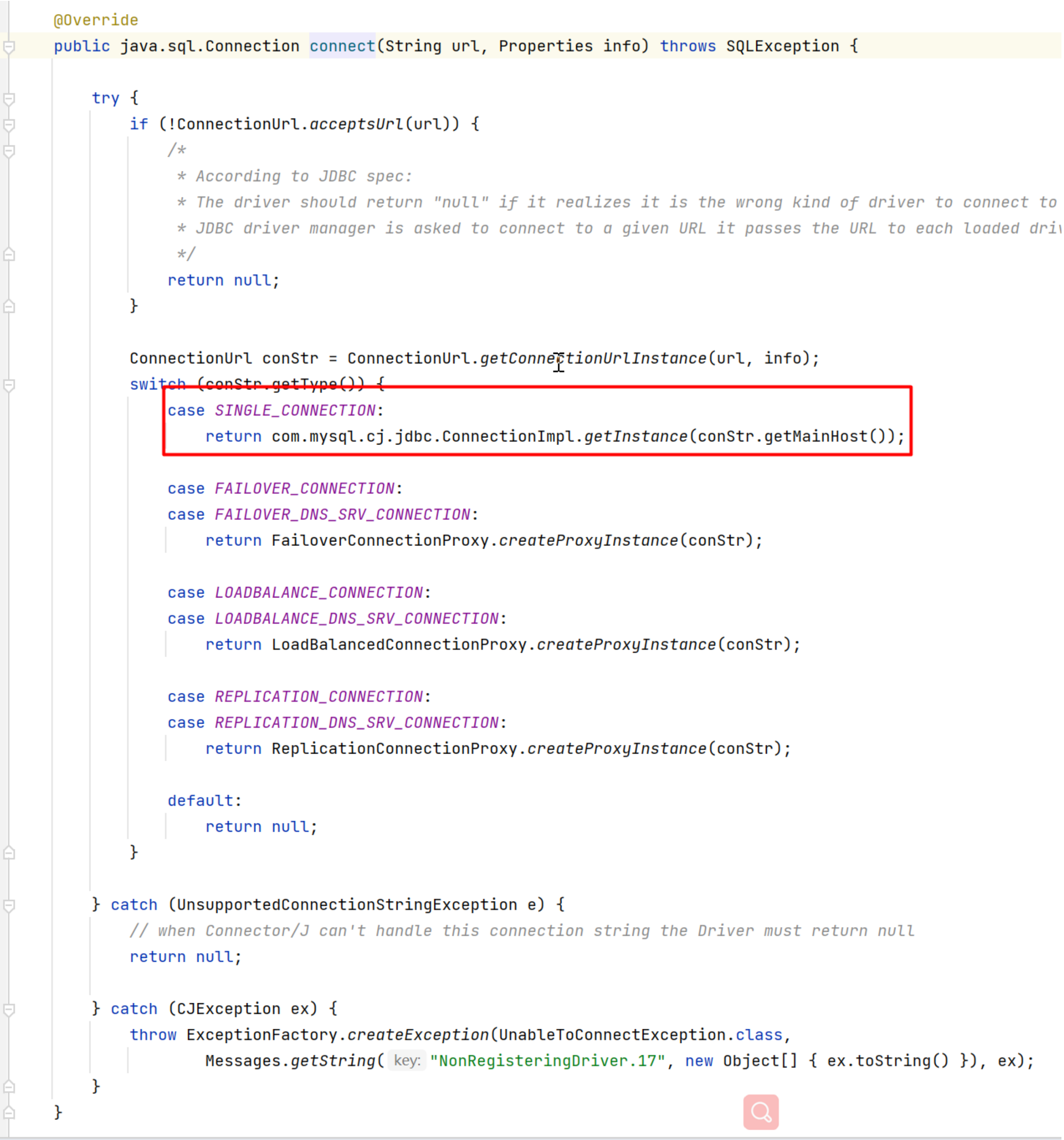
```
public String encode(ConnectionProxy connection, String s) throws SQLException {
    try {
        CharsetConvert charsetConvert = (CharsetConvert) connection.getAttribute(ATTR_CHARSET_CONVERTER);

        return charsetConvert.encode(s);
    } catch (UnsupportedEncodingException e) {
        throw new SQLException(e.getMessage(), e);
    }
}
```

使用Driver创建Connection

从上面的分析中我们看到FilterChainImpl中会主动使用驱动类Driver的connect方法创建connection对象。

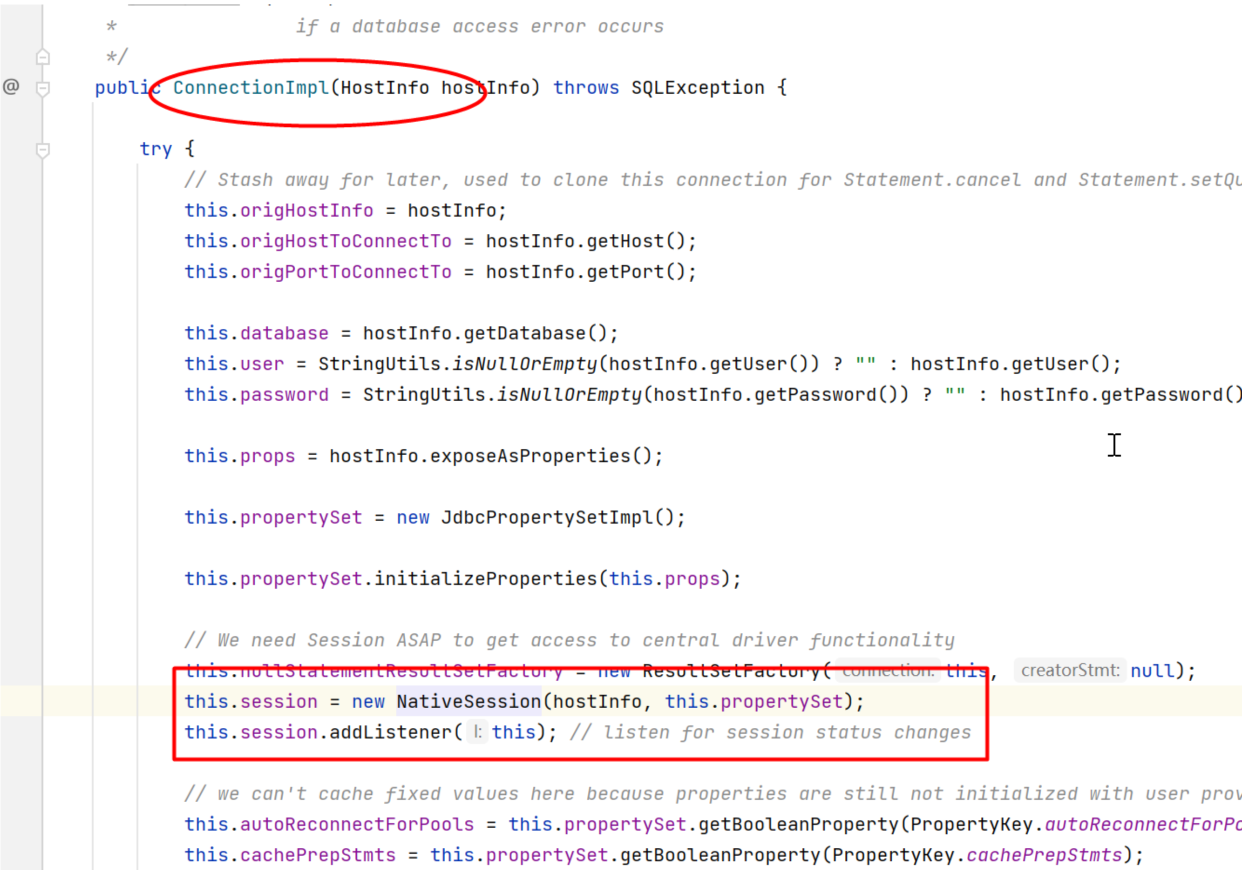
```
/**
 * 使用Driver获取connection。
 在MySQL中驱动就是com.mysql.cj.jdbc.Driver，  connection就是ConnectionImpl
 */
Connection nativeConnection = driver.connect(url, info);
```

使用mysql驱动包中的Driver的connect方法 最终会执行com.mysql.cj.jdbc.NonRegisteringDriver#connect



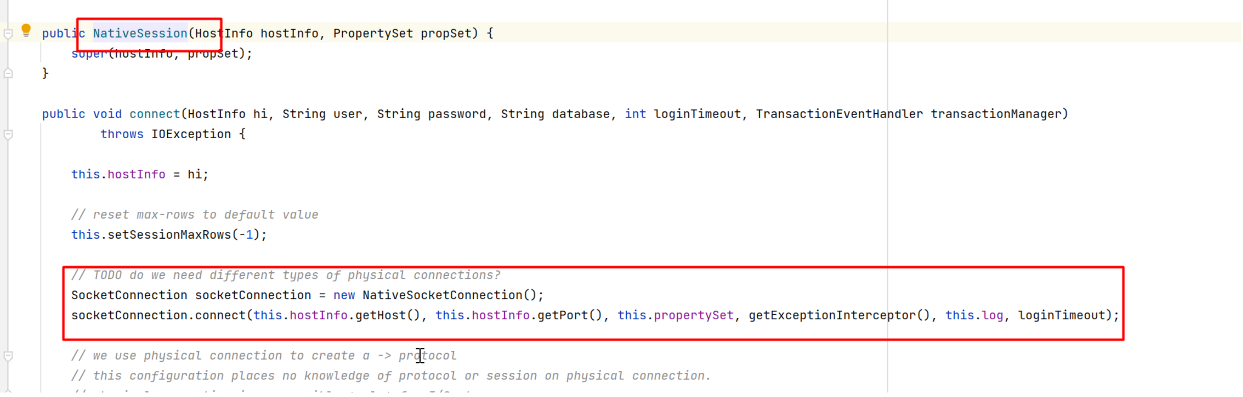
```java
@Override
public java.sql.Connection connect(String url, Properties info) throws SQLException {

    try {
        if (!ConnectionUrl.acceptsUrl(url)) {
            /*
             * According to JDBC spec:
             * The driver should return "null" if it realizes it is the wrong kind of driver to connect to
             * JDBC driver manager is asked to connect to a given URL it passes the URL to each loaded dri
             */
            return null;
        }

        ConnectionUrl conStr = ConnectionUrl.getConnectionUrlInstance(url, info);
        switch (conStr.getType()) {
            case SINGLE_CONNECTION:
                return com.mysql.cj.jdbc.ConnectionImpl.getInstance(conStr.getMainHost());

            case FAILOVER_CONNECTION:
            case FAILOVER_DNS_SRV_CONNECTION:
                return FailoverConnectionProxy.createProxyInstance(conStr);

            case LOADBALANCE_CONNECTION:
            case LOADBALANCE_DNS_SRV_CONNECTION:
                return LoadBalancedConnectionProxy.createProxyInstance(conStr);

            case REPLICATION_CONNECTION:
            case REPLICATION_DNS_SRV_CONNECTION:
                return ReplicationConnectionProxy.createProxyInstance(conStr);

            default:
                return null;
        }

    } catch (UnsupportedConnectionStringException e) {
        // when Connector/J can't handle this connection string the Driver must return null
        return null;

    } catch (CJException ex) {
        throw ExceptionFactory.createException(UnableToConnectException.class,
                Messages.getString( key: "NonRegisteringDriver.17", new Object[] { ex.toString() }), ex);
    }
}
```

对于ConnectionImpl来说 代表着一个连接，ConnectionImpl 中持有一个NativeSession对象



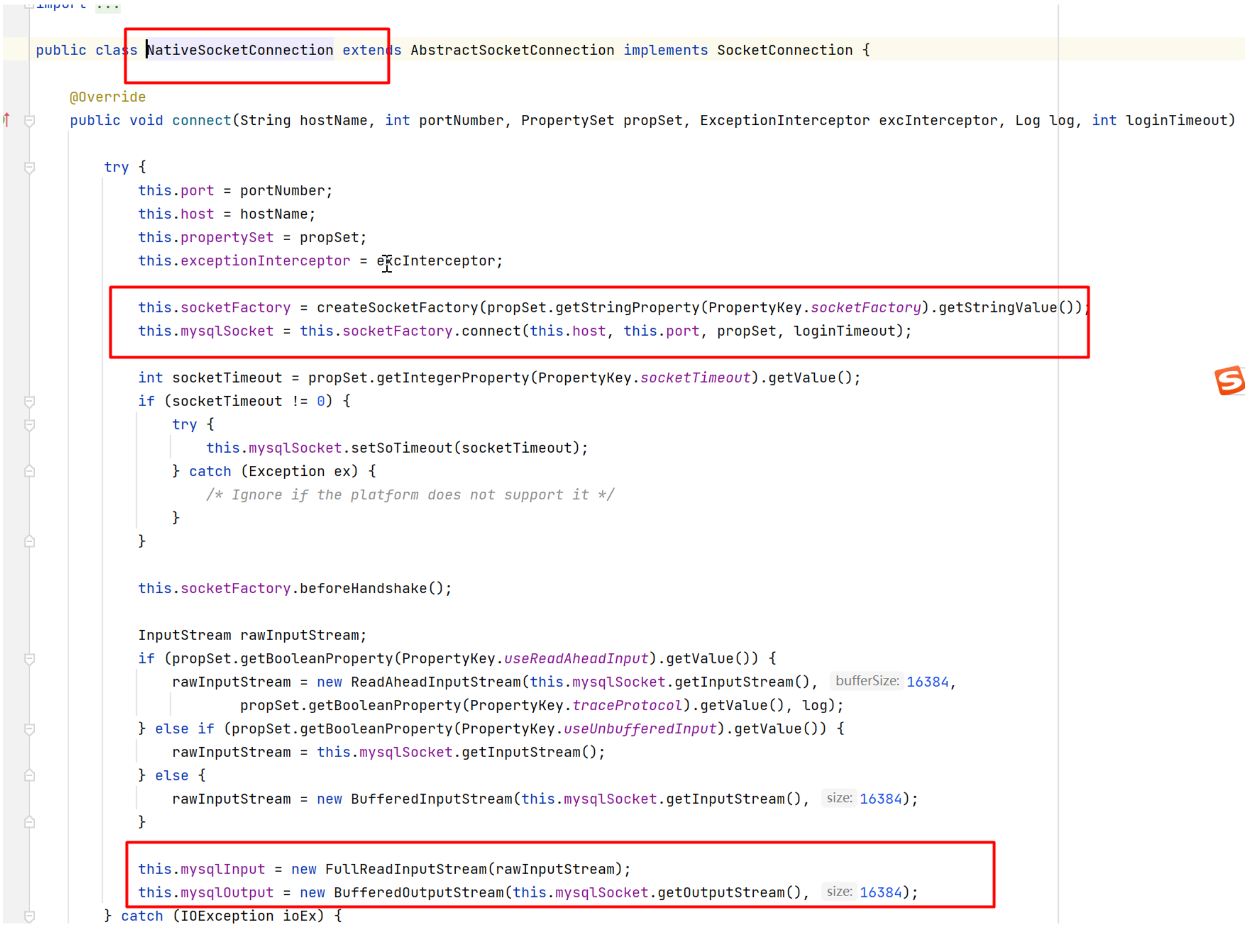
```java
     *                if a database access error occurs
     */
    public ConnectionImpl(HostInfo hostInfo) throws SQLException {

        try {
            // Stash away for later, used to clone this connection for Statement.cancel and Statement.setQu
            this.origHostInfo = hostInfo;
            this.origHostToConnectTo = hostInfo.getHost();
            this.origPortToConnectTo = hostInfo.getPort();

            this.database = hostInfo.getDatabase();
            this.user = StringUtils.isNullOrEmpty(hostInfo.getUser()) ? "" : hostInfo.getUser();
            this.password = StringUtils.isNullOrEmpty(hostInfo.getPassword()) ? "" : hostInfo.getPassword()

            this.props = hostInfo.exposeAsProperties();

            this.propertySet = new JdbcPropertySetImpl();

            this.propertySet.initializeProperties(this.props);

            // We need Session ASAP to get access to central driver functionality
            this.nullStatementResultSetFactory = new ResultSetFactory( connection: this,  creatorStmt: null);
            this.session = new NativeSession(hostInfo, this.propertySet);
            this.session.addListener( l: this); // listen for session status changes

            // we can't cache fixed values here because properties are still not initialized with user prov
            this.autoReconnectForPools = this.propertySet.getBooleanProperty(PropertyKey.autoReconnectForPc
            this.cachePrepStmts = this.propertySet.getBooleanProperty(PropertyKey.cachePrepStmts);
```

对于NativeSession对象来说，在其connect方法会创建 NativeSocketConnection对象。



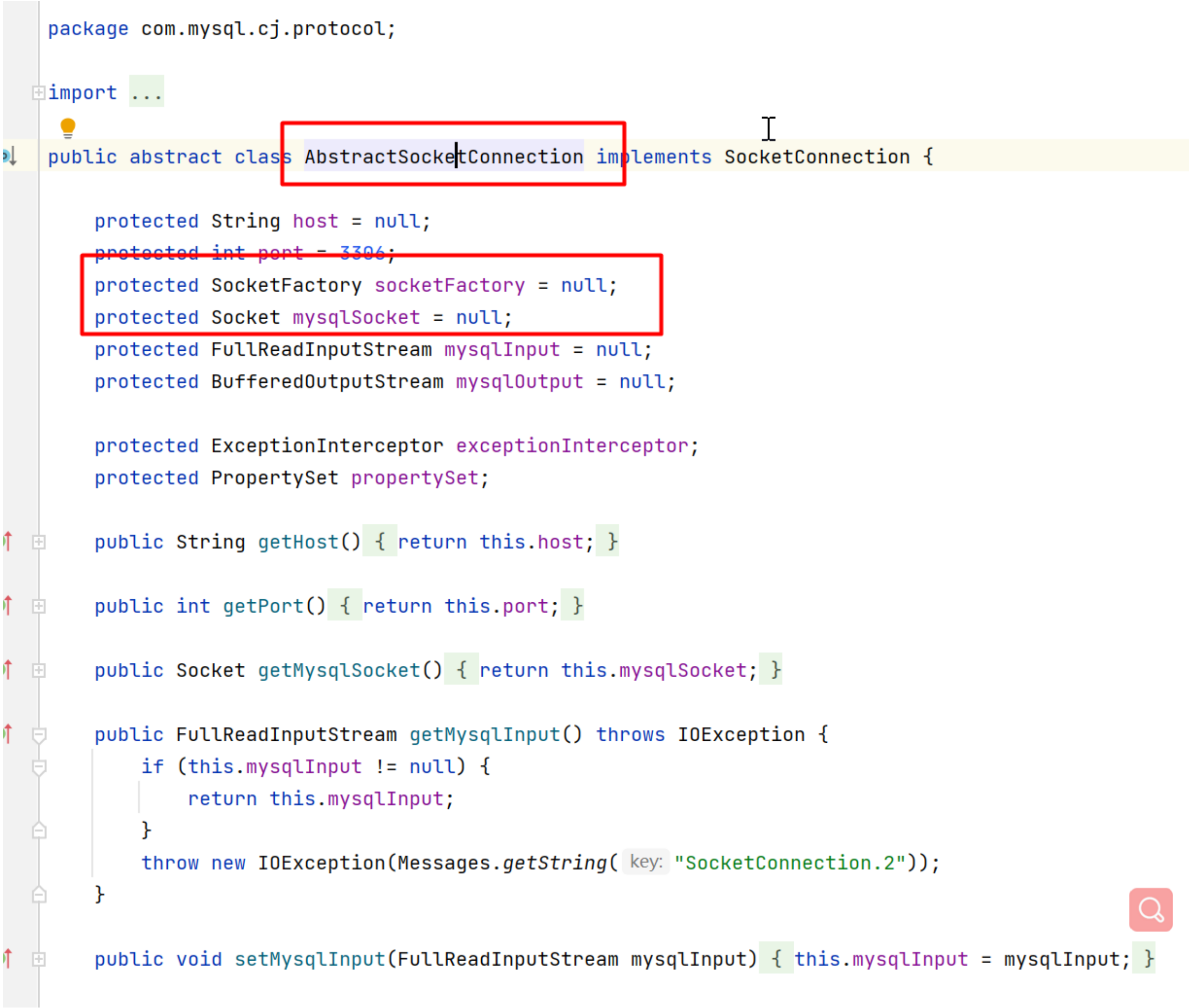
```java
    public NativeSession(HostInfo hostInfo, PropertySet propSet) {
        super(hostInfo, propSet);
    }

    public void connect(HostInfo hi, String user, String password, String database, int loginTimeout, TransactionEventHandler transactionManager)
            throws IOException {

        this.hostInfo = hi;

        // reset max-rows to default value
        this.setSessionMaxRows(-1);

        // TODO do we need different types of physical connections?
        SocketConnection socketConnection = new NativeSocketConnection();
        socketConnection.connect(this.hostInfo.getHost(), this.hostInfo.getPort(), this.propertySet, getExceptionInterceptor(), this.log, loginTimeout);

        // we use physical connection to create a -> protocol
        // this configuration places no knowledge of protocol or session on physical connection.
```

NativeSocketConnection对象的connect方法中会创建socketFactory， 通过socketFactory 的connect方法创建java中的Socket对象mysqlSocket

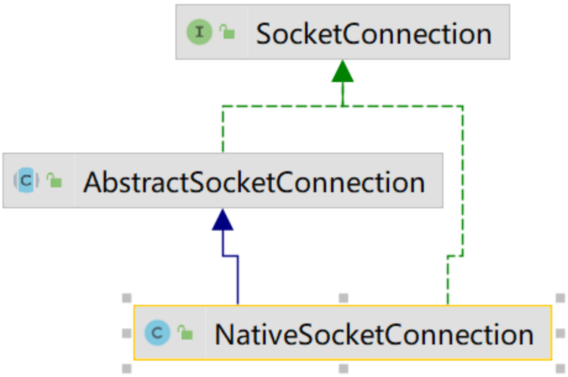


```
import ...

public class NativeSocketConnection extends AbstractSocketConnection implements SocketConnection {

    @Override
    public void connect(String hostName, int portNumber, PropertySet propSet, ExceptionInterceptor excInterceptor, Log log, int loginTimeout)

        try {
            this.port = portNumber;
            this.host = hostName;
            this.propertySet = propSet;
            this.exceptionInterceptor = excInterceptor;

            this.socketFactory = createSocketFactory(propSet.getStringProperty(PropertyKey.socketFactory).getStringValue());
            this.mysqlSocket = this.socketFactory.connect(this.host, this.port, propSet, loginTimeout);

            int socketTimeout = propSet.getIntegerProperty(PropertyKey.socketTimeout).getValue();
            if (socketTimeout != 0) {
                try {
                    this.mysqlSocket.setSoTimeout(socketTimeout);
                } catch (Exception ex) {
                    /* Ignore if the platform does not support it */
                }
            }

            this.socketFactory.beforeHandshake();

            InputStream rawInputStream;
            if (propSet.getBooleanProperty(PropertyKey.useReadAheadInput).getValue()) {
                rawInputStream = new ReadAheadInputStream(this.mysqlSocket.getInputStream(),  bufferSize: 16384,
                        propSet.getBooleanProperty(PropertyKey.traceProtocol).getValue(), log);
            } else if (propSet.getBooleanProperty(PropertyKey.useUnbufferedInput).getValue()) {
                rawInputStream = this.mysqlSocket.getInputStream();
            } else {
                rawInputStream = new BufferedInputStream(this.mysqlSocket.getInputStream(),  size: 16384);
            }

            this.mysqlInput = new FullReadInputStream(rawInputStream);
            this.mysqlOutput = new BufferedOutputStream(this.mysqlSocket.getOutputStream(),  size: 16384);
        } catch (IOException ioEx) {
```

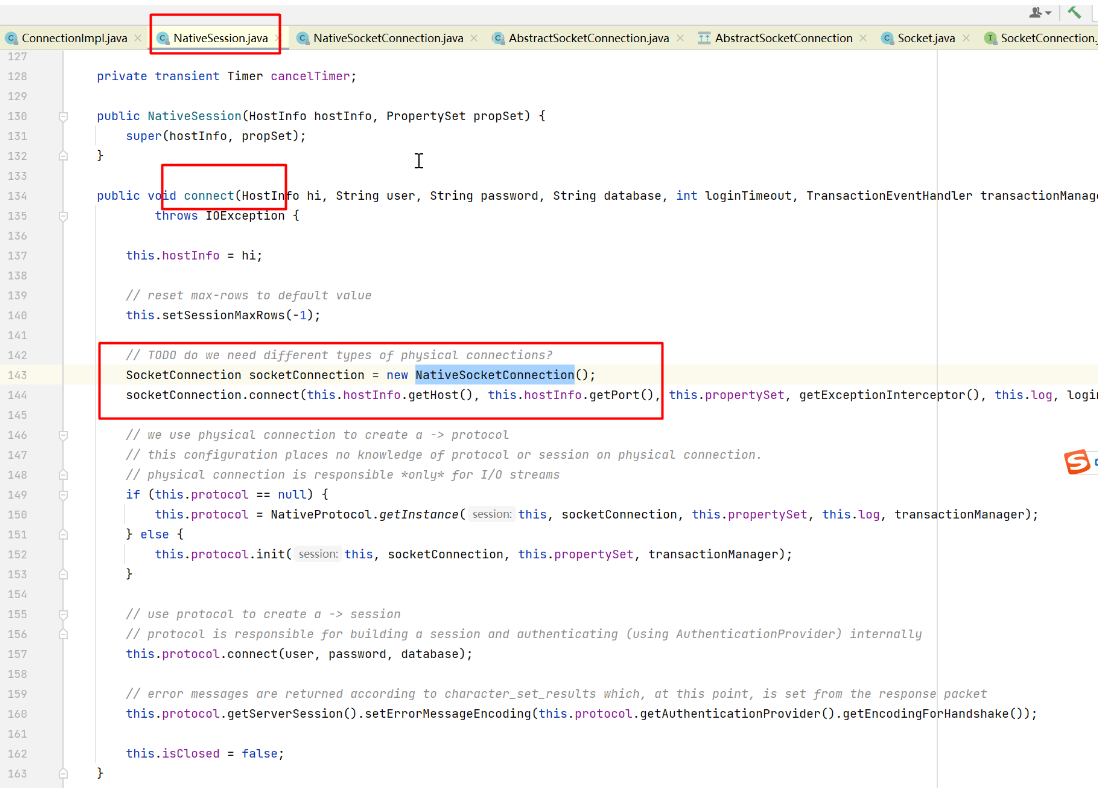
至此我们了解到 mysql驱动包中使用AbstractSocketConnection建立连接， AbstractSocketConnection的子类是NativeSocketConnection



```
package com.mysql.cj.protocol;

import ...

public abstract class AbstractSocketConnection implements SocketConnection {

    protected String host = null;
    ~~protected int port = 3306;~~
    protected SocketFactory socketFactory = null;
    protected Socket mysqlSocket = null;
    protected FullReadInputStream mysqlInput = null;
    protected BufferedOutputStream mysqlOutput = null;

    protected ExceptionInterceptor exceptionInterceptor;
    protected PropertySet propertySet;

    public String getHost() { return this.host; }

    public int getPort() { return this.port; }

    public Socket getMysqlSocket() { return this.mysqlSocket; }

    public FullReadInputStream getMysqlInput() throws IOException {
        if (this.mysqlInput != null) {
            return this.mysqlInput;
        }
        throw new IOException(Messages.getString( key: "SocketConnection.2"));
    }

    public void setMysqlInput(FullReadInputStream mysqlInput) { this.mysqlInput = mysqlInput; }
```



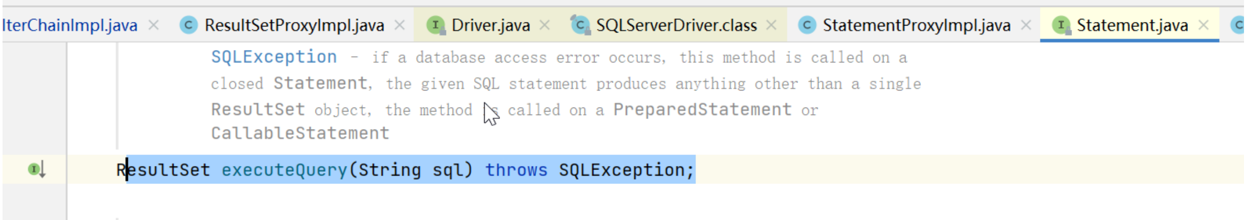
那么我们不禁要问Mysql驱动中的ConnectionImpl和NativeSocketConnection之间的关系是什么?
ConnectionImpl作为Java 标准sql包中的Connection的实现类。 他有成员属性NativeSession 。
NativeSession中通过成员属性保存了host信息。 NativeSession的connect方法 每次执行都会创建一个
NativeSocketConnection，NativeSession对象中不存在NativeSocketConnection成员属性。



```java
    private transient Timer cancelTimer;

    public NativeSession(HostInfo hostInfo, PropertySet propSet) {
        super(hostInfo, propSet);
    }

    public void connect(HostInfo hi, String user, String password, String database, int loginTimeout, TransactionEventHandler transactionManag
            throws IOException {

        this.hostInfo = hi;

        // reset max-rows to default value
        this.setSessionMaxRows(-1);

        // TODO do we need different types of physical connections?
        SocketConnection socketConnection = new NativeSocketConnection();
        socketConnection.connect(this.hostInfo.getHost(), this.hostInfo.getPort(), this.propertySet, getExceptionInterceptor(), this.log, logi

        // we use physical connection to create a -> protocol
        // this configuration places no knowledge of protocol or session on physical connection.
        // physical connection is responsible *only* for I/O streams
        if (this.protocol == null) {
            this.protocol = NativeProtocol.getInstance( session: this, socketConnection, this.propertySet, this.log, transactionManager);
        } else {
            this.protocol.init( session: this, socketConnection, this.propertySet, transactionManager);
        }

        // use protocol to create a -> session
        // protocol is responsible for building a session and authenticating (using AuthenticationProvider) internally
        this.protocol.connect(user, password, database);

        // error messages are returned according to character_set_results which, at this point, is set from the response packet
        this.protocol.getServerSession().setErrorMessageEncoding(this.protocol.getAuthenticationProvider().getEncodingForHandshake());

        this.isClosed = false;
    }
```

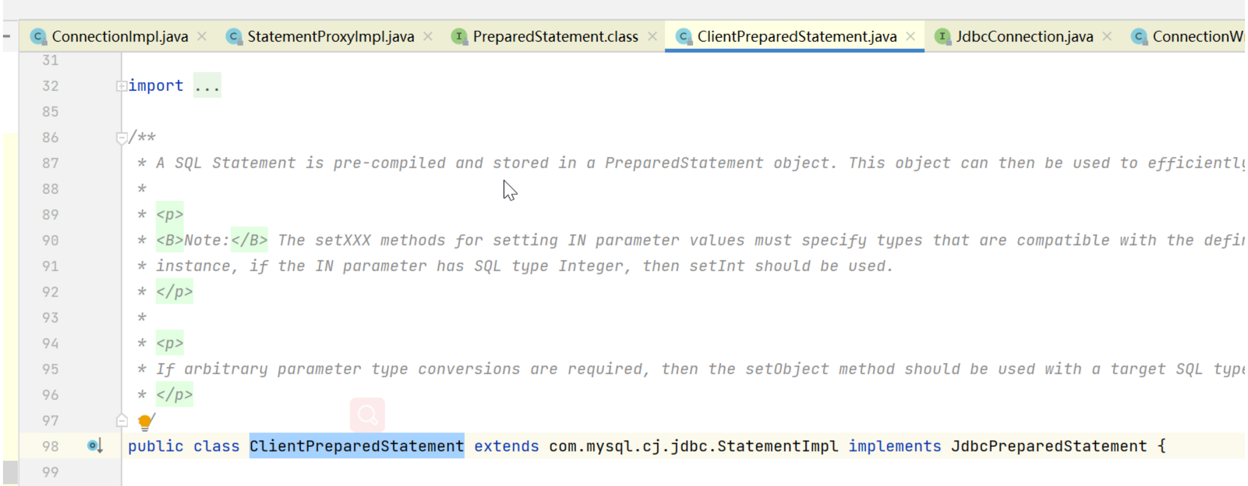
Druid中的Statement

Java表中中的statement定义了如下方法executeQuery

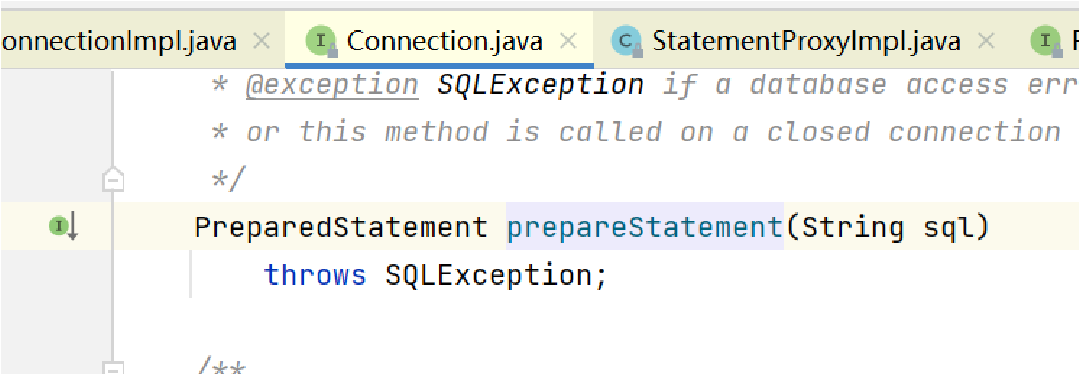


```
SQLException - if a database access error occurs, this method is called on a
closed Statement, the given SQL statement produces anything other than a single
ResultSet object, the method called on a PreparedStatement or
CallableStatement

ResultSet executeQuery(String sql) throws SQLException;
```

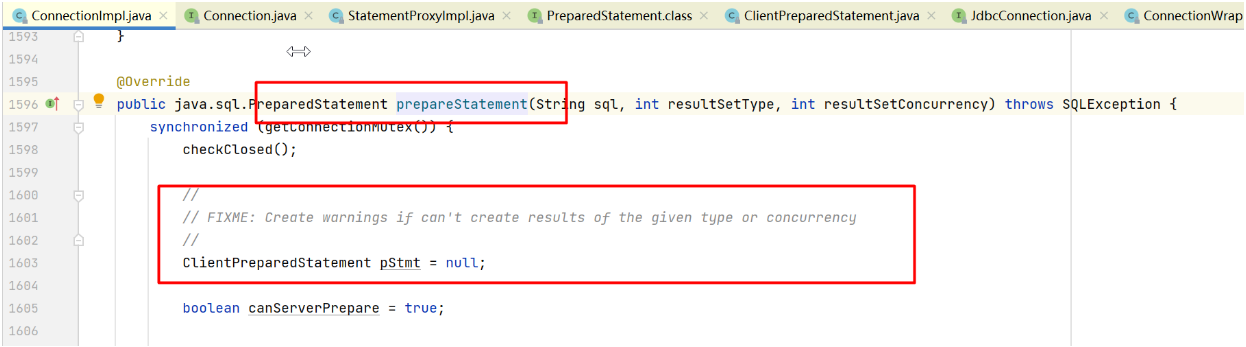
在jdbc的驱动包中提供了ClientPreparedStatement作为PreparedStatement的默认实现，Sql标准包中的Connection接口定义了prepareStatement方法返回一个PreparedStatement对象，这在Connection的默认实现ConnectionImpl 类中的prepareStatement方法中就可以看到返回都是ClientPreparedStatement对象



```java
import ...

/**
 * A SQL Statement is pre-compiled and stored in a PreparedStatement object. This object can then be used to efficiently
 *
 * <p>
 * <B>Note:</B> The setXXX methods for setting IN parameter values must specify types that are compatible with the defi
 * instance, if the IN parameter has SQL type Integer, then setInt should be used.
 * </p>
 *
 * <p>
 * If arbitrary parameter type conversions are required, then the setObject method should be used with a target SQL type
 * </p>
 */
public class ClientPreparedStatement extends com.mysql.cj.jdbc.StatementImpl implements JdbcPreparedStatement {
```

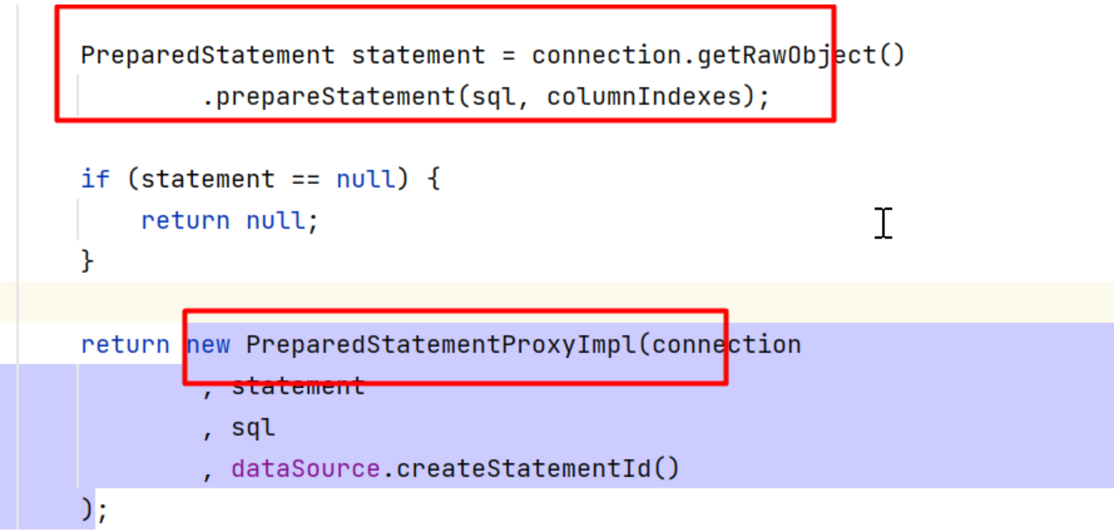


```
     * @exception SQLException if a database access err
     * or this method is called on a closed connection
     */
    PreparedStatement prepareStatement(String sql)
        throws SQLException;

    /**
```



```
    }

    @Override
    public java.sql.PreparedStatement prepareStatement(String sql, int resultSetType, int resultSetConcurrency) throws SQLException {
        synchronized (getConnectionMutex()) {
            checkClosed();

            //
            // FIXME: Create warnings if can't create results of the given type or concurrency
            //
            ClientPreparedStatement pStmt = null;

            boolean canServerPrepare = true;

```

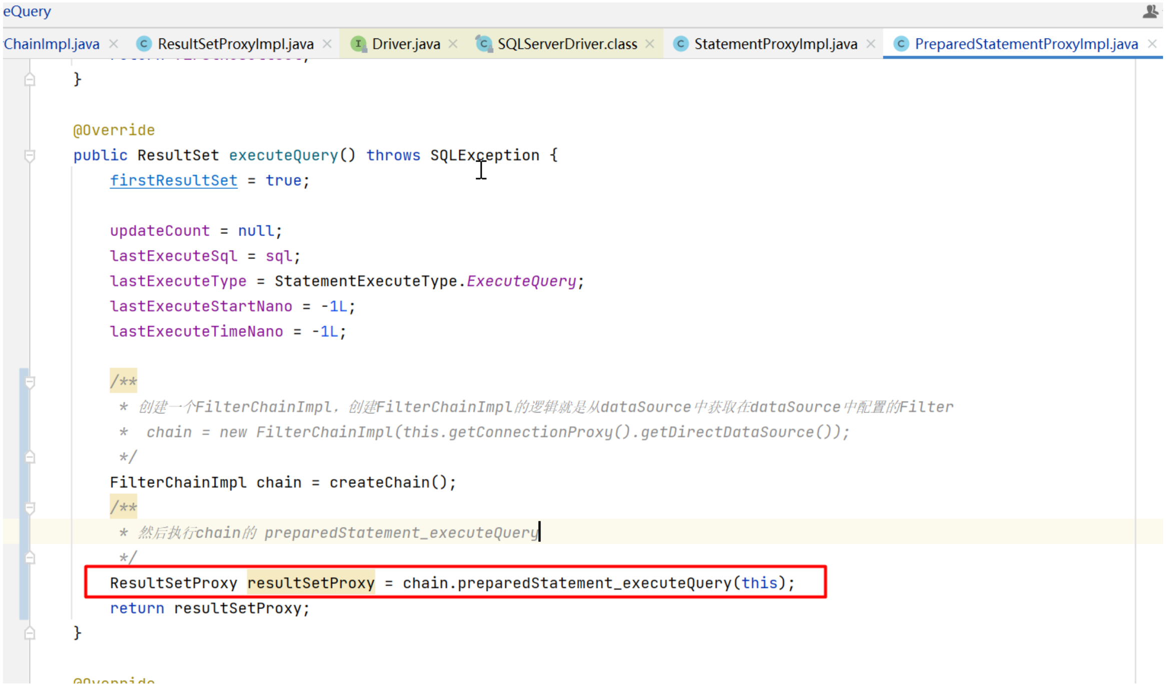
在Druid中，针对底层connection的prepareStatement返回的PreparedStatement对象会使用PreparedStatementProxyImpl 作为代理包装



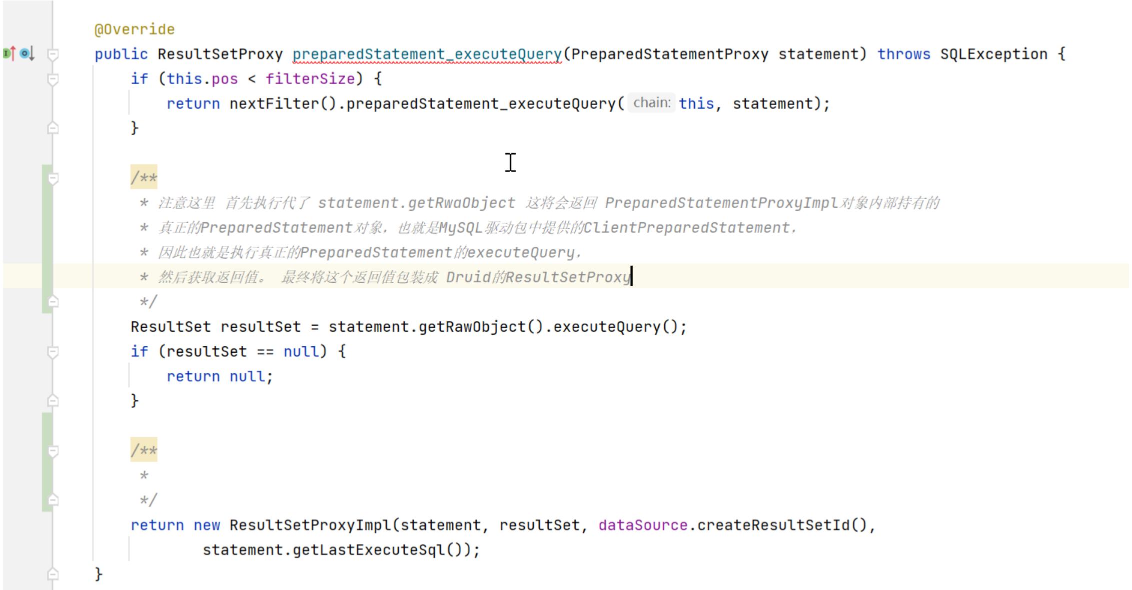
```
PreparedStatement statement = connection.getRawObject()
        .prepareStatement(sql, columnIndexes);

if (statement == null) {
    return null;
}

return new PreparedStatementProxyImpl(connection
        , statement
        , sql
        , dataSource.createStatementId()
);
```

因此在Druid中我们获取到的PrepareStatement就是PreparedStatementProxyImpl对象，对于这个对象他内部持有标准的PreparedStatement对象的引用（也就是Mysql驱动包中提供的ClientPreparedStatement）PreparedStatementProxyImpl 也是实现了PreparedStatement接口的，只不过其内部逻辑将会委托给内部的属性PreparedStatement statement;来完成。

我们分析一下PreparedStatementProxyImpl的executeQuery方法



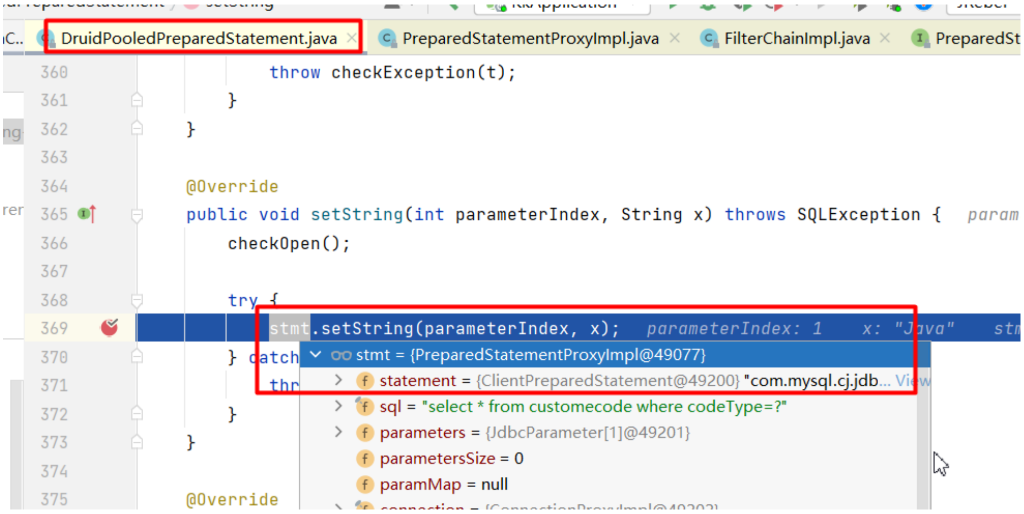
```java
}

@Override
public ResultSet executeQuery() throws SQLException {
    firstResultSet = true;

    updateCount = null;
    lastExecuteSql = sql;
    lastExecuteType = StatementExecuteType.ExecuteQuery;
    lastExecuteStartNano = -1L;
    lastExecuteTimeNano = -1L;

    /**
     * 创建一个FilterChainImpl，创建FilterChainImpl的逻辑就是从dataSource中获取在dataSource中配置的Filter
     *  chain = new FilterChainImpl(this.getConnectionProxy().getDirectDataSource());
     */
    FilterChainImpl chain = createChain();
    /**
     * 然后执行chain的 preparedStatement_executeQuery
     */
    ResultSetProxy resultSetProxy = chain.preparedStatement_executeQuery(this);
    return resultSetProxy;
}
```



```java
@Override
public ResultSetProxy preparedStatement_executeQuery(PreparedStatementProxy statement) throws SQLException {
    if (this.pos < filterSize) {
        return nextFilter().preparedStatement_executeQuery( chain: this, statement);
    }

    /**
     * 注意这里 首先执行代了 statement.getRwaObject 这将会返回 PreparedStatementProxyImpl对象内部持有的
     * 真正的PreparedStatement对象，也就是MySQL驱动包中提供的ClientPreparedStatement.
     * 因此也就是执行真正的PreparedStatement的executeQuery.
     * 然后获取返回值。 最终将这个返回值包装成 Druid的ResultSetProxy
     */
    ResultSet resultSet = statement.getRawObject().executeQuery();
    if (resultSet == null) {
        return null;
    }

    /**
     *
     */
    return new ResultSetProxyImpl(statement, resultSet, dataSource.createResultSetId(),
            statement.getLastExecuteSql());
}
```

最终我们通过 PreparedStatement的executeQuery方法得到的就是Druid的ResultSetProxyImpl对象，这个ResultSetProxyImpl对象中持有mysql驱动包中定义的ResultSet接口的真正实现类对象。

从这里我们发现Druid总是对 底层SQL 组件对象进行代理。正如下图


DruidPooledPreparedStatement.java
PreparedStatementProxyImpl.java
FilterChainImpl.java
PreparedSt

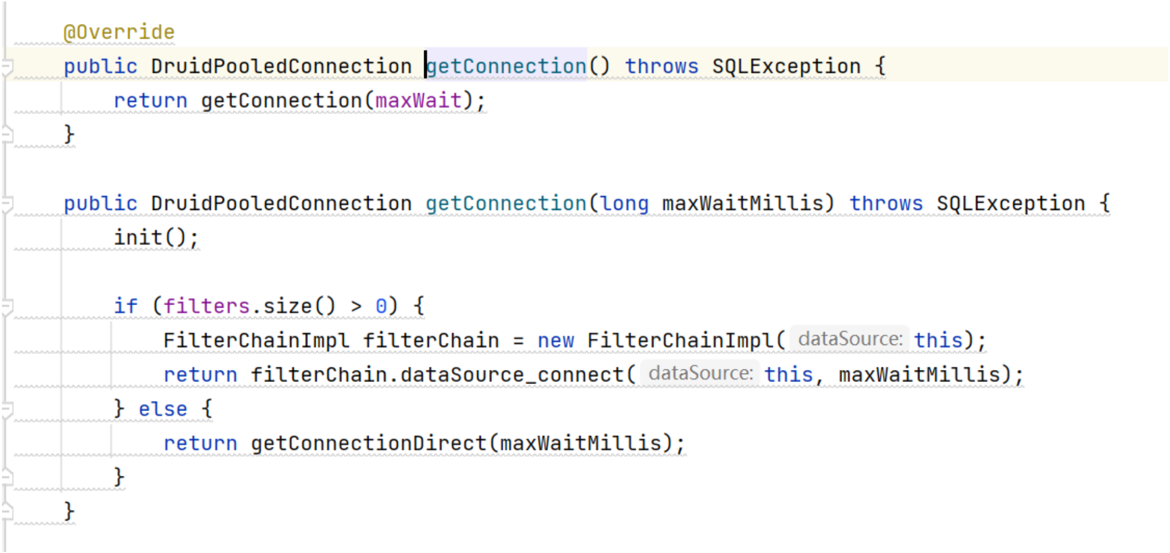
```java
            throw checkException(t);
        }
    }

    @Override
    public void setString(int parameterIndex, String x) throws SQLException {   param
        checkOpen();

        try {
            stmt.setString(parameterIndex, x);   parameterIndex: 1    x: "Java"    stm
        } catch
            thr
        }
    }

    @Override
```

```
stmt = {PreparedStatementProxyImpl@49077}
  statement = {ClientPreparedStatement@49200} "com.mysql.cj.jdb... View
  sql = "select * from customecode where codeType=?"
  parameters = {JdbcParameter[1]@49201}
  parametersSize = 0
  paramMap = null
```

创建Connection的另一种场景

另外一种获取连接的场景是使用DataSource获取连接 Connection con = DataSourceUtils.getConnection(obtainDataSource()); 也就是执行 Connection con = dataSource.getConnection();

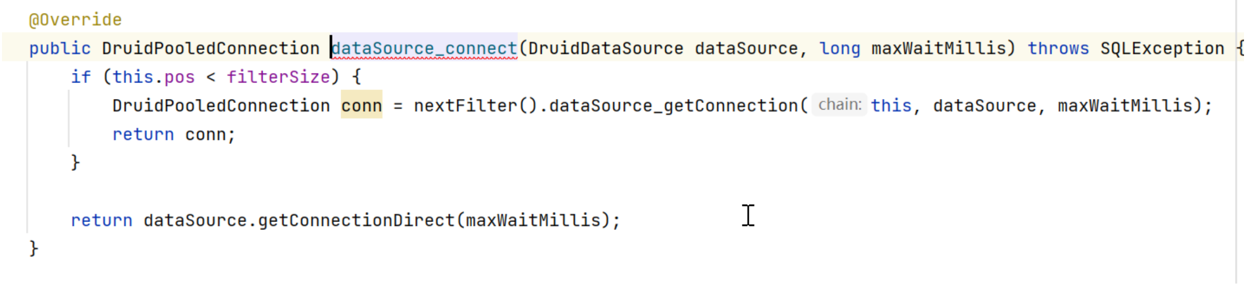
在DruidDataSource的getConnection中实现如下



```
@Override
public DruidPooledConnection getConnection() throws SQLException {
    return getConnection(maxWait);
}

public DruidPooledConnection getConnection(long maxWaitMillis) throws SQLException {
    init();

    if (filters.size() > 0) {
        FilterChainImpl filterChain = new FilterChainImpl( dataSource: this);
        return filterChain.dataSource_connect( dataSource: this, maxWaitMillis);
    } else {
        return getConnectionDirect(maxWaitMillis);
    }
}
```

在这个getConnection中是使用了FilterChainImpl的dataSource_connect

FilterChainImpl的dataSource_connect 如下：



```
@Override
public DruidPooledConnection dataSource_connect(DruidDataSource dataSource, long maxWaitMillis) throws SQLException {
    if (this.pos < filterSize) {
        DruidPooledConnection conn = nextFilter().dataSource_getConnection( chain: this, dataSource, maxWaitMillis);
        return conn;
    }

    return dataSource.getConnectionDirect(maxWaitMillis);
}
```

从这个dataSource_connect 中我们可以看到创建连接的操作是使用了getConnectionDirect

在这个getConnectionDriect方法中首先获取物理连接

PhysicalConnectionInfo pyConnInfo = DruidDataSource.this.createPhysicalConnection();

holder = new DruidConnectionHolder(this, pyConnInfo);

物理connection连接放置到Holder中，然后放置到DruidPooledConnection，也就是dataSource.connect返回的就是DruidPooledConnection

DruidPooledConnection poolalbeConnection = new DruidPooledConnection(holder);