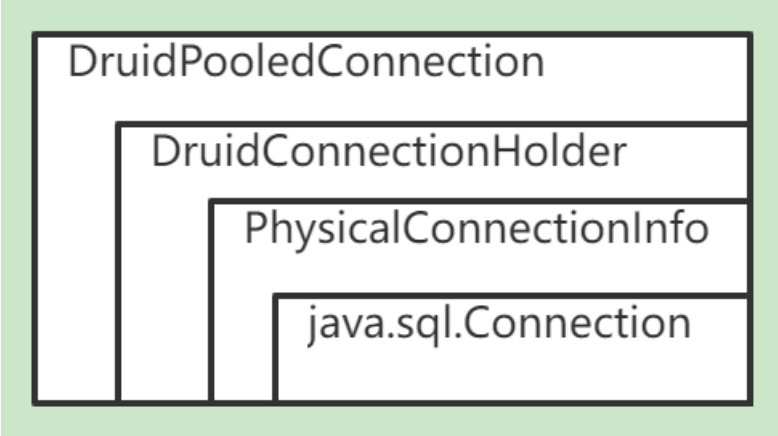
# 研究问题

## 为什么要有`DruidConnectionHolder`？

完成对“池化技术”的部分了解后，选择了研究`DruidConnectionHolder`的业务意义，看能否从中学习到一些设计模式或者“如何扩展、增强”等知识。
因此在研究holder的前，我划分好要了解的内容：**1.**类的组成结构（指的是所在包、接口、继承等）。**2.**类本身有哪些变量？**3.**该类与哪些类相关联？
带着以上三个内容，我从前面章节中提到的“connection的组成”开始研究，即下图所示结构：

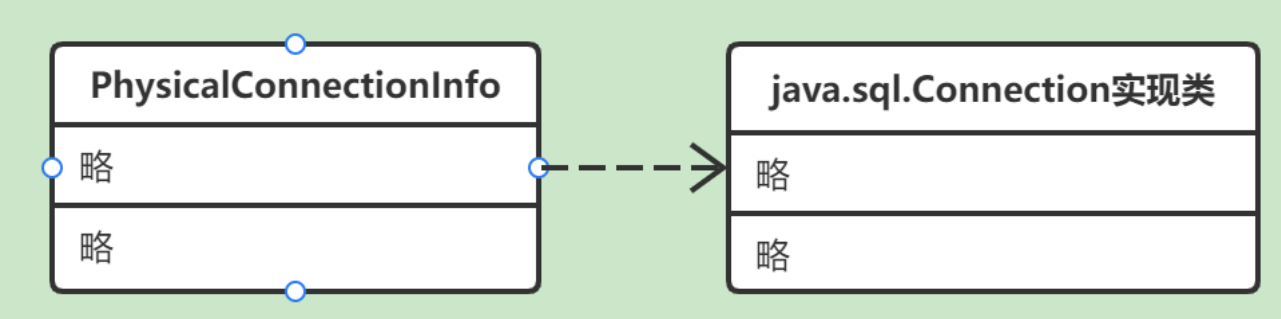



## java.sql.Connection

JDBC相关接口，具体内容不展开讨论

## PhysicalConnectionInfo

`PhysicalConnectionInfo`中持有`java.sql.Connection`实现类的实例，关系如下图



### 成员变量

这里指出我所清楚的几个变量，`connectStartNanos`、`connectedNanos`、`initedNanos`、`validatedNanos`。这四个变量分别的含义是“建立连接前的系统当前时间”、“建立连接后的系统当前时间”、“初始化连接后的系统当前时间”、“检验连接后的系统当前时间”。形如用`connectedNanos`减去`connectStartNanos`，便可以知道与数据库建立连接时所消耗的时间。其他变量同理。具体代码在`DruidAbstractDataSource.createPhysicalConnection()`中。

### 方法

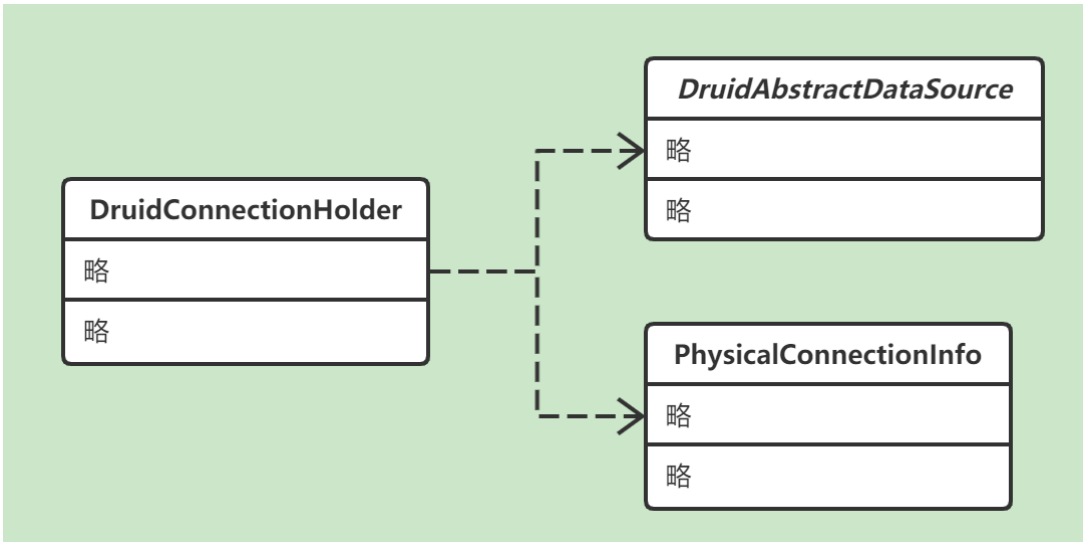
`DruidAbstractDataSource.initPhysicalConnection(Connection, Map<String, Object>, Map<String, Object>)`和`DruidAbstractDataSource.validateConnection(Connection)`。若在创建连接池时配置了创建物理连接的初始化方法和检验业务，这两个方法分别会执行对用的业务。

### 意义

不难看出`PhysicalConnectionInfo`存在的目的是扩展JDBC的Connection的功能。通过关联java.sql.Connection实现类，增加了“检验物理连接”、“初始化物理连接”的业务方法。因此我对`PhysicalConnectionInfo`的理解是：“存放与物理连接相关的内容”。

## DruidConnectionHolder

holder持有着`PhysicalConnectionInfo`和`DruidAbstractDataSource`的实例引用，关系如下图



成员变量

目前了解到的有`lastActiveTimeMillis`、`lastKeepTimeMillis`、`statementPool`。这三个变量分别的作用是“最后活跃时间”、“最后保活时间（存疑）”、“当前连接缓存的执行语句”

**ConnectionEventListeners**

在了解javax.sql.PooledConnection后，发现在holder中有成员变量connectionEventListeners，研究该变量，以下代码块中仅展现相关部分。

```
public interface PooledConnection {
    void addConnectionEventListener(ConnectionEventListener
listener);
    void removeConnectionEventListener(ConnectionEventListener
listener);
}

public class DruidPooledConnection extends PoolableWrapper
implements javax.sql.PooledConnection, Connection {
    @Override
    public void addConnectionEventListener(ConnectionEventListener
listener) {
        if (holder == null) {
            throw new IllegalStateException();
        }

        holder.getConnectionEventListeners().add(listener);
    }

    @Override
    public void
removeConnectionEventListener(ConnectionEventListener listener) {
        if (holder == null) {
            throw new IllegalStateException();
        }

        holder.getConnectionEventListeners().remove(listener);
    }

    @Override
    public void close() throws SQLException {
        if (this.disable) {
```

```
            return;
        }

        DruidConnectionHolder holder = this.holder;
        if (holder == null) {
            if (dupCloseLogEnable) {
                LOG.error("dup close");
            }
            return;
        }

        DruidAbstractDataSource dataSource =
holder.getDataSource();
        boolean isSameThread = this.getOwnerThread() ==
Thread.currentThread();

        if (!isSameThread) {
            dataSource.setAsyncCloseConnectionEnable(true);
        }

        if (dataSource.isAsyncCloseConnectionEnable()) {
            syncClose();
            return;
        }

        if (!CLOSING_UPDATER.compareAndSet(this, 0, 1)) {
            return;
        }

        try {
            // 在这里通知对应的监听
            for (ConnectionEventListener listener :
holder.getConnectionEventListeners()) {
                listener.connectionClosed(new
ConnectionEvent(this));
            }

            List<Filter> filters = dataSource.getProxyFilters();
            if (filters.size() > 0) {
                FilterChainImpl filterChain = new
FilterChainImpl(dataSource);
                filterChain.dataSource_recycle(this);
            } else {
```

```
                recycle(); // 留意此方法
            }
        } finally {
            CLOSING_UPDATER.set(this, 0);
        }

        this.disable = true;
    }
}
```

由以上源码看出，`DruidPooledConnection`和`DruidConnectionHolde`配合实现了`javax.sql.PooledConnection`接口中的方法。Holder在上述过程中负责保存对应的监听器集合。

响应完`ConnectionEventListener`之后，接着会执行`DruidPooledConnection.recycle()`方法以及`DruidDataSource.recycle(DruidPooledConnection)`，以下给出这部分代码。

```
// DruidPooledConnection
public void recycle() throws SQLException {
    if (this.disable) {
        return;
    }

    DruidConnectionHolder holder = this.holder;
    if (holder == null) {
        if (dupCloseLogEnable) {
            LOG.error("dup close");
        }
        return;
    }

    if (!this.abandoned) {
        DruidAbstractDataSource dataSource =
holder.getDataSource();
        dataSource.recycle(this);
    }
    // 解绑数据
    this.holder = null;
    conn = null;
    transactionInfo = null;
    closed = true;
}
```

```
// DruidDataSource
protected void recycle(DruidPooledConnection pooledConnection)
throws SQLException {
    // 忽略部分代码
    // reset holder, restore default settings, clear warnings
    boolean isSameThread = pooledConnection.ownerThread ==
Thread.currentThread();
    if (!isSameThread) {
        final ReentrantLock lock = pooledConnection.lock;
        lock.lock();
        try {
            holder.reset();
        } finally {
            lock.unlock();
        }
    } else {
        holder.reset();
    }
}

// DruidPooledConnection
public void reset() throws SQLException {
    // reset default settings
    if (underlyingReadOnly != defaultReadOnly) {
        conn.setReadOnly(defaultReadOnly);
        underlyingReadOnly = defaultReadOnly;
    }

    if (underlyingHoldability != defaultHoldability) {
        conn.setHoldability(defaultHoldability);
        underlyingHoldability = defaultHoldability;
    }

    if (underlyingTransactionIsolation !=
defaultTransactionIsolation) {
        conn.setTransactionIsolation(defaultTransactionIsolation);
        underlyingTransactionIsolation =
defaultTransactionIsolation;
    }

    if (underlyingAutoCommit != defaultAutoCommit) {
        conn.setAutoCommit(defaultAutoCommit);
        underlyingAutoCommit = defaultAutoCommit;
```

```
        }

        connectionEventListeners.clear(); // 会话级别
        statementEventListeners.clear(); // 会话级别

        lock.lock();
        try {
            for (Object item : statementTrace.toArray()) {
                Statement stmt = (Statement) item;
                JdbcUtils.close(stmt);
            }

            statementTrace.clear();
        } finally {
            lock.unlock();
        }

        conn.clearWarnings();
    }
```

从“回收”业务可知，在`DruidPooledConnection.close()`且进入`recycle()`，会重置holder部分数据，并且`DruidPooledConnection`会解绑Connection实例、holder实例。而在holder的reset()中，只会重置部分会话级别的数据，统计数据是不会重置的。如何理解“会话级别”呢，我的理解是与业务相关的阶段，例如执行业务上特殊的SQL。因此`connectionEventListener`也只是“会话级别”的。

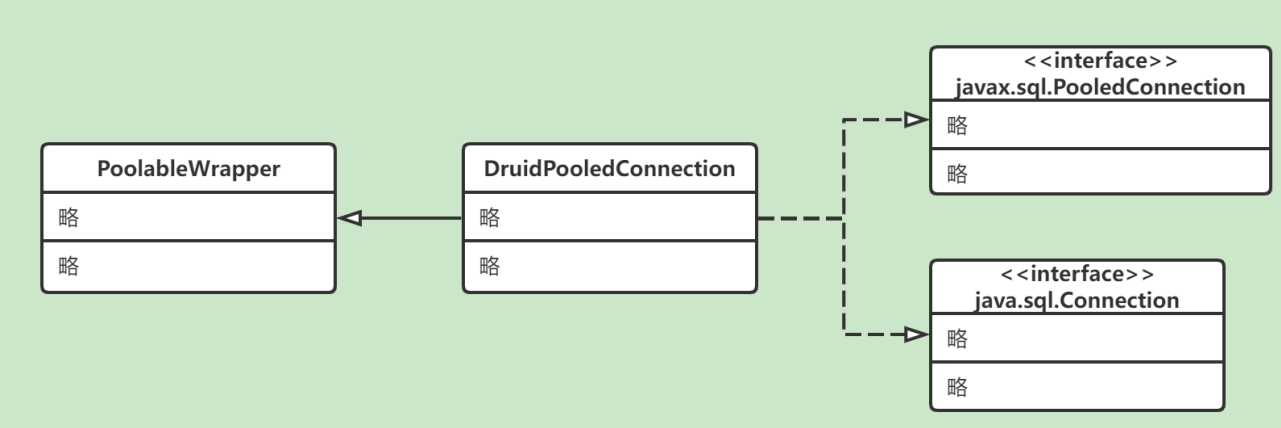
因此，从`DruidDataSource.getConnection()`~`DruidPooledConnection.close()`期间，我理解为“客户线程拥有连接的阶段。”，即连接在池外的一个生命周期。这个生命周期也可理解holder是的一个生命周期。利用这个概念，知道holder负责的业务了。可以回答“为什么要有DruidConnectionHolder” - 负责收集连接在池外的信息。

## 意义

其实在涉及到的成员变量中，就能够看出holder主要是负责保存跟执行SQL相关的信息。换句话来说，holder专门用于跟踪应用程序业务信息。

# DruidPooledConnection

先给出这个类的关系图




根据UML图的关系，DruidPooledConnection既然实现了javax.sql.PooledConnection这个接口，说明DruidPooledConnection拥有了javax.sql.PooledConnection的特性。因此我选择分析javax.sql.PooledConnection这个接口。

## javax.sql.PooledConnection

引用openjdk11中该接口的描述：

```
An object that provides hooks for connection pool management.A
PooledConnection object represents a physical connection to a data
source. The connectioncan be recycled rather than being closed when
an application is finished with it, thus reducing the number of
connections that need to be made.
```

翻译的大体意思是：`javax.sql.PooledConnection`为连接池的管理提供了“钩子（hook）”。该接口让数据库连接在调用`close()`方法时会被连接池回收而不是真的关闭与数据库的物理连接。
其中涉及到的"hook"，在网上定于为“钩子函数”。其目的是在目标函数被调用前，先调用该钩子函数。其实在Java中常见的代理模式与这个hook有着相似的功能，例如触发某个时间后，代理类先处理非业务逻辑（例如写日志），再调用目标业务方法。

这个接口涉及到的方法如下：

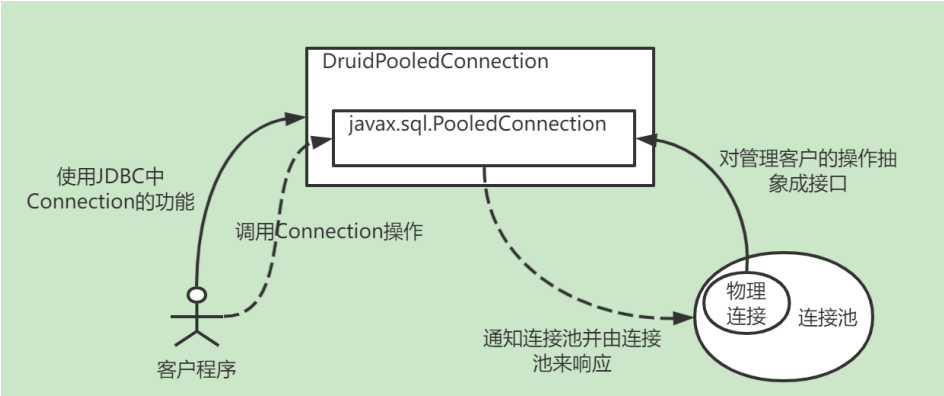
```
public interface PooledConnection {
    void close() throws SQLException; // App调用该方法并不是关闭物理连
接。
                                      // 关闭物理连接的操作应该由连接池来
完成。
    Connection getConnection() throws SQLException;
    void addConnectionEventListener(ConnectionEventListener
listener);
    void removeConnectionEventListener(ConnectionEventListener
listener);
    public void addStatementEventListener(StatementEventListener
listener);
    public void removeStatementEventListener(StatementEventListener
listener);
}
```

看`close()`和`getConnection()`这两个方法，可以知道要代理`java.sql.Connection`中对应的方法。这里就通过方法覆盖和动态链接实现了与hook类似的效果。以下是`DruidPooledConnection`重写后的`close()`。

```
    @Override
    public void close() throws SQLException {
        if (this.disable) {
            return;
        }

        DruidConnectionHolder holder = this.holder;
        if (holder == null) {
            if (dupCloseLogEnable) {
                LOG.error("dup close");
            }
            return;
        }

        DruidAbstractDataSource dataSource =
holder.getDataSource();
        boolean isSameThread = this.getOwnerThread() ==
Thread.currentThread();

        if (!isSameThread) {
            dataSource.setAsyncCloseConnectionEnable(true);
        }
```

```
        if (dataSource.isAsyncCloseConnectionEnable()) {
            syncClose(); // 根据条件选择关闭连接
            return;
        }

        if (!CLOSING_UPDATER.compareAndSet(this, 0, 1)) {
            return;
        }

        try {
            for (ConnectionEventListener listener :
holder.getConnectionEventListeners()) {
                listener.connectionClosed(new
ConnectionEvent(this));
            }

            List<Filter> filters = dataSource.getProxyFilters();
            if (filters.size() > 0) {
                FilterChainImpl filterChain = new
FilterChainImpl(dataSource);
                filterChain.dataSource_recycle(this);
            } else {
                recycle(); // 回收该连接
            }
        } finally {
            CLOSING_UPDATER.set(this, 0);
        }

        this.disable = true;
    }
```

至此，实现了`javax.sql.PooledConnection`接口的类就代表着该类的数据库连接具备着池化属性。对于客户程序来讲是透明的。而对连接池而言，该接口则是一层抽象框架，目的是抽象出来连接池中的连接应该有的行为以及告诉连接池如何响应客户在连接上做出的一些行为（该接口的监听方法）。`javax.sql.PooledConnection`像是"客户 <---> 连接池"之间的一个桥梁，用下图来表示：




## 意义

目前了解到在`DruidPooledConnection`大部分的成员变量与holder相对应，其行为也类似。比较新颖的是在`DruidPooledConnection`还会记录着拥有该连接的线程。DruidPooledConnection最主要的功能还是为客户程序提供了与`java.sql.Connection`相同的功能，同时又为了连接池的管理重写了部分功能。

## 信息划分图