#### 一、初始化数据源

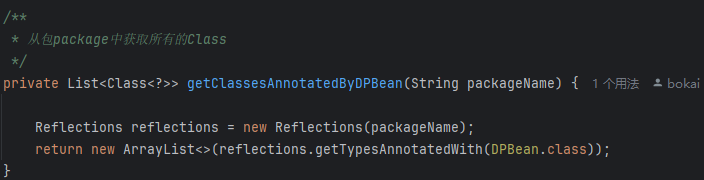
##### 1. 获取BeanFactory



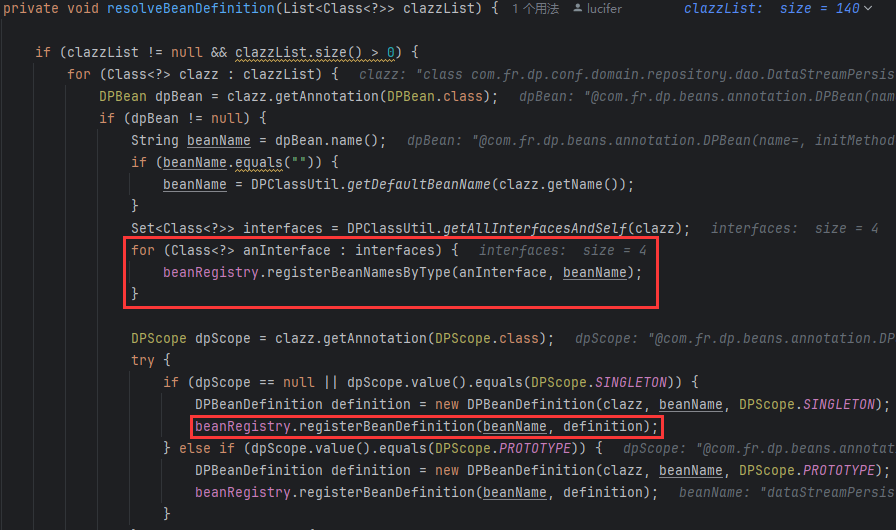
构造函数中指定需要扫描的目录：



获得所有使用了@DPBean注解的类：



将这些类以及他们实现的接口注册到容器中：



registerBeanNamesByType注册所有类/接口到allBeanNamesByType中

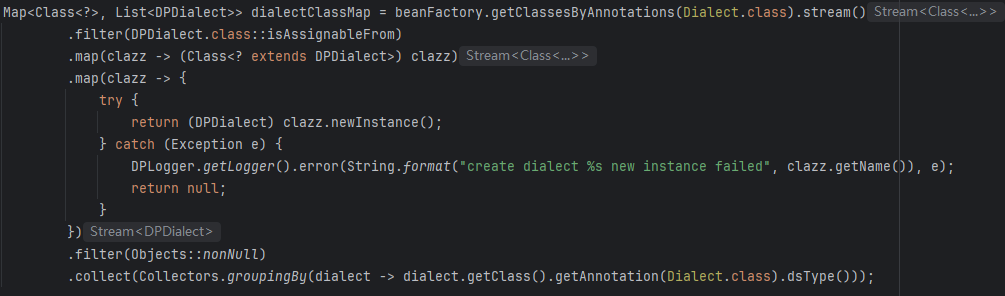
registerBeanDefinition注册实现类到swiftBeanDefinitionMap中

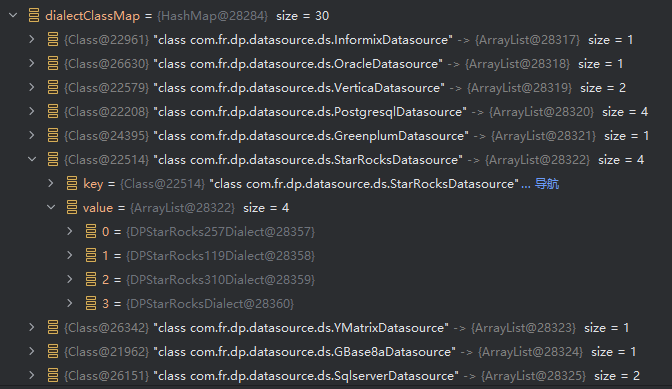
##### 2. 根据@Datasource注解获取数据源类





##### 3. 收集数据源类对应的方言。一种数据源可能有多个版本的方言。





##### 4. 获取各个数据源的配置信息，包含datasource、reader、writer等类全限定名，以及数据源名称





#### 二、获取所有数据库连接信息

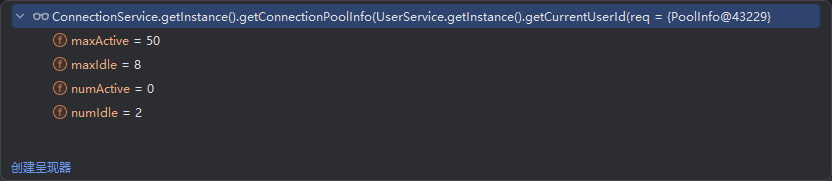
##### 1. 配置数据连接后获取数据连接信息

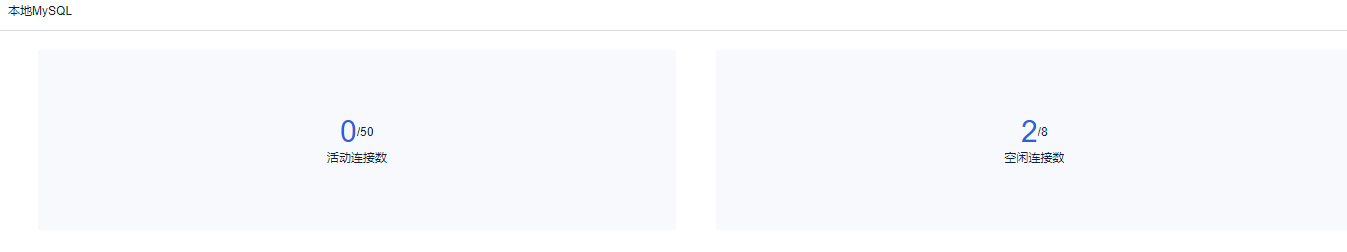




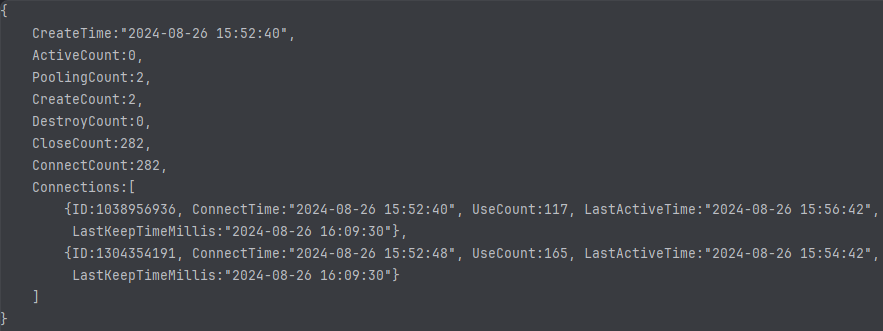
##### 2. 获取连接池信息





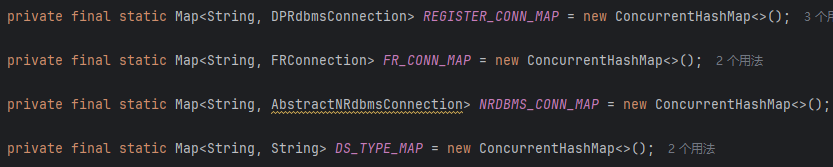


底层是查询druid连接池：



##### 3. FDL内部保存连接名到数据连接映射关系的数据结构DPConnectionPool

保存了4个Map：



① REGISTER\_CONN\_MAP：目前只在测试里面用，用register方法直接注册数据连接

② FR\_CONN\_MAP：关系型数据源的数据连接映射，采用缓存机制，在启动的时候加载，之后使用直接使用缓存，提高速度。

③ NRDBMS\_CONN\_MAP：非关系型数据源的数据连接映射

④ DS\_TYPE\_MAP：没看懂

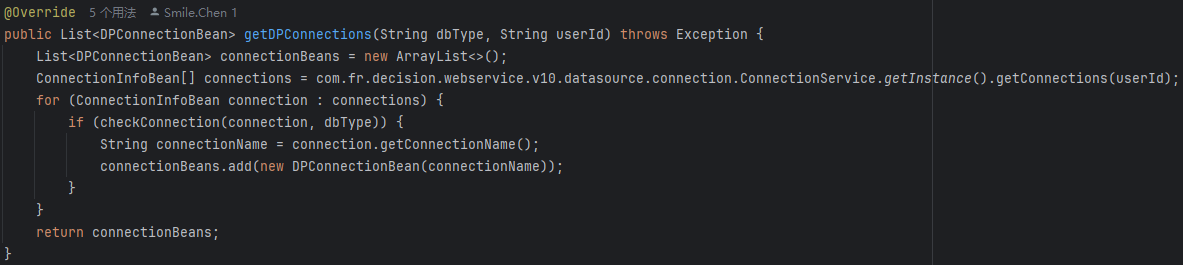
##### 4. 获取方言信息



#### 三、配置和获取数据源相关信息

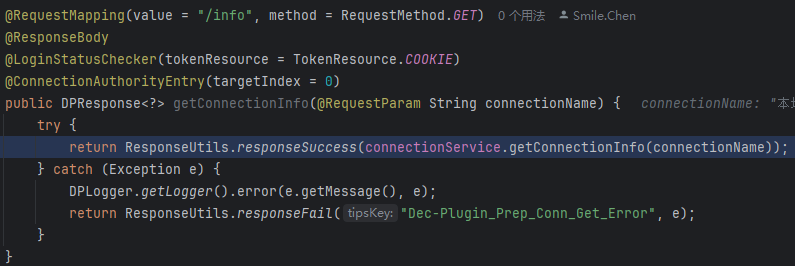
##### 1. 获取指定数据源的数据连接列表

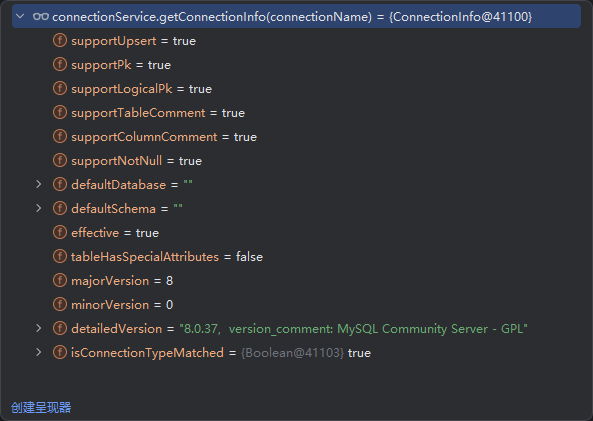






##### 2. 获取指定数据源下指定数据连接的信息





##### 3. 获取所有schema / database、测试连接



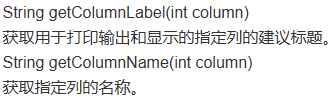
底层：

① 通过JDBC方法connection.getMetaData() 获得DatabaseMetaData类型元数据

② 用metaData.getSchemas() / metaData.getCatalogs() 方法获得包含schema/database名的表

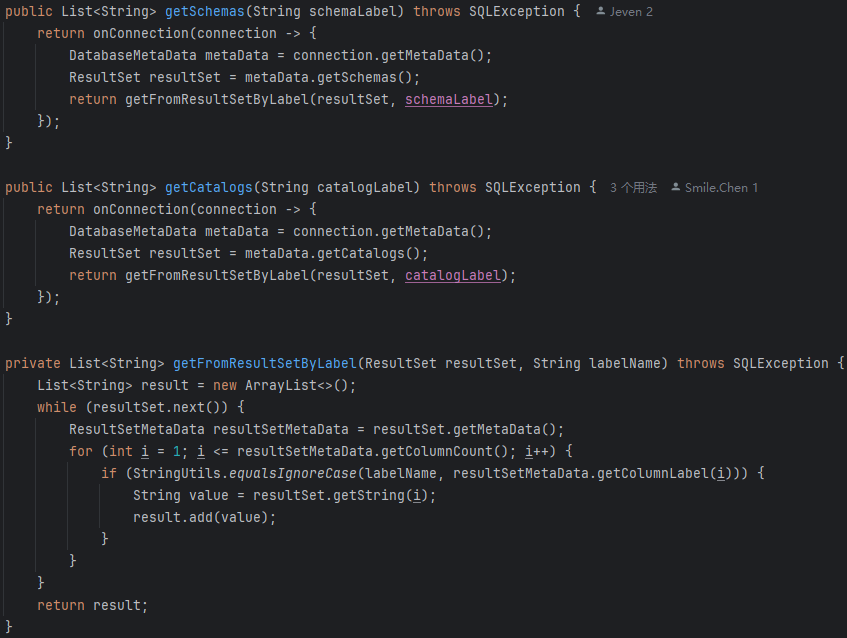
③ 从②中的表中通过resultSetMetaData.getColumnLabel() 方法选取Label为TABLE\_CAT / TABLE\_SCHEM 的列，其中就包含了所有schema/database名

★在用这种方法获取名称时候，尽量使用getColumnLabel而不是getColumnName！



参考：[getColumnLabel VS getColumnName \_getcolumnlabel getcolumnname-CSDN博客](https://blog.csdn.net/shuchengzhang/article/details/3621364)

[getColumnLabel and getColumnName\_getcolumnlabel和getcolumnname-CSDN博客](https://blog.csdn.net/xiangxueping80/article/details/43061989)

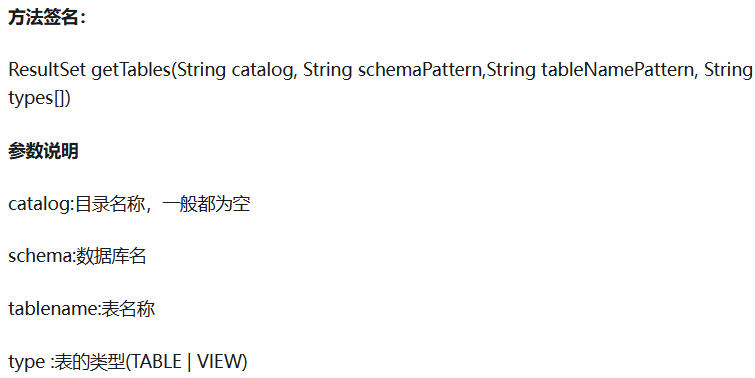


##### 4. 获取指定数据连接的某个schema/database下的所有table/view

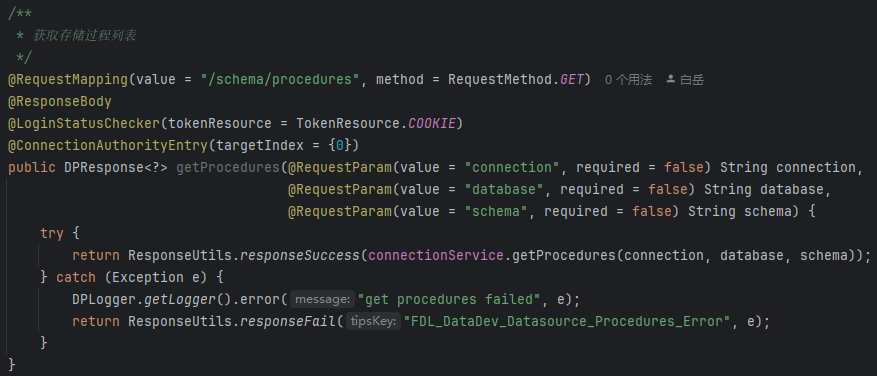


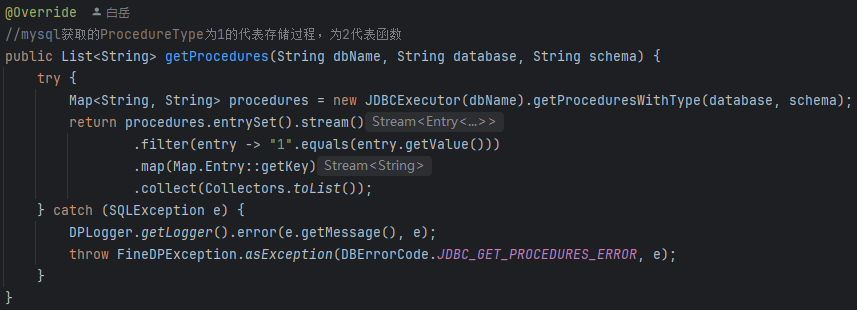
底层使用JDBC的getTables方法



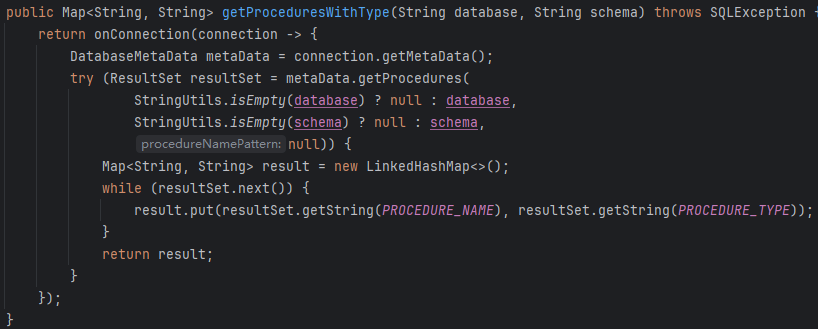


##### 5. 获取指定schema/database下的所有存储过程

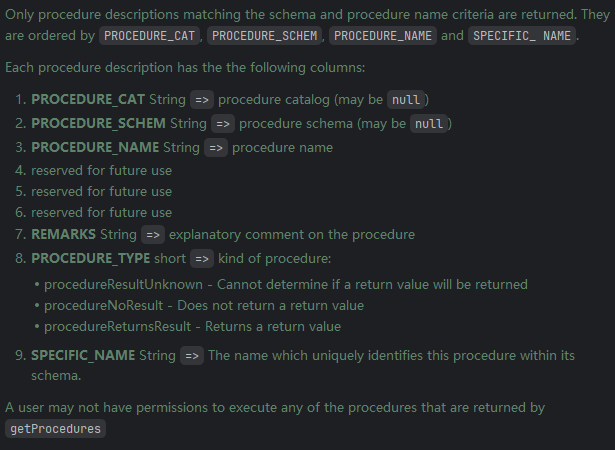




底层使用metaData.getProcedures() 方法



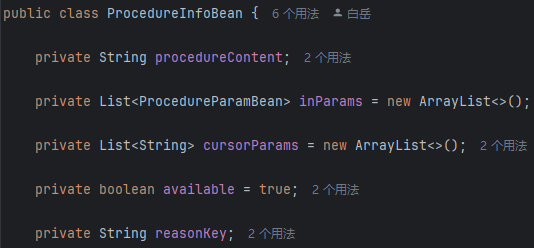
该方法返回的ResultSet的列：



##### 6. 获取指定存储过程具体信息（参数信息+内容）

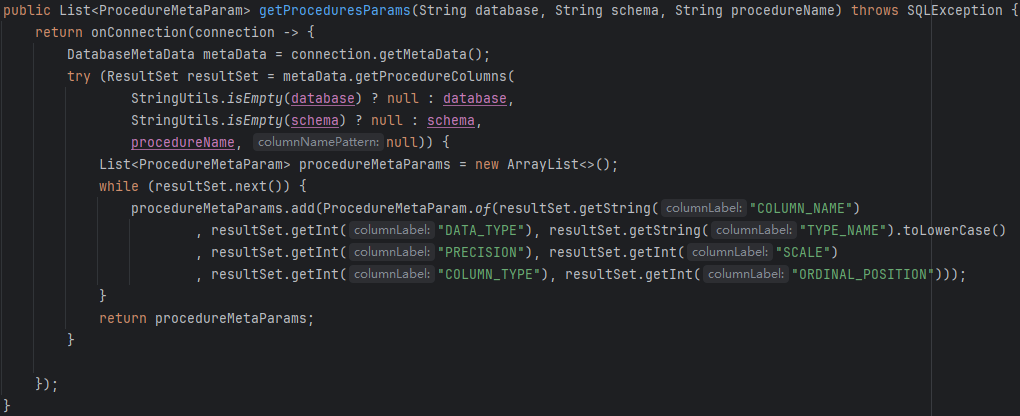


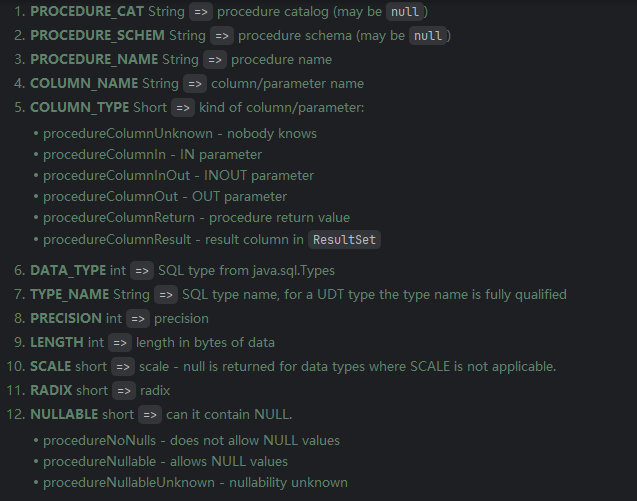
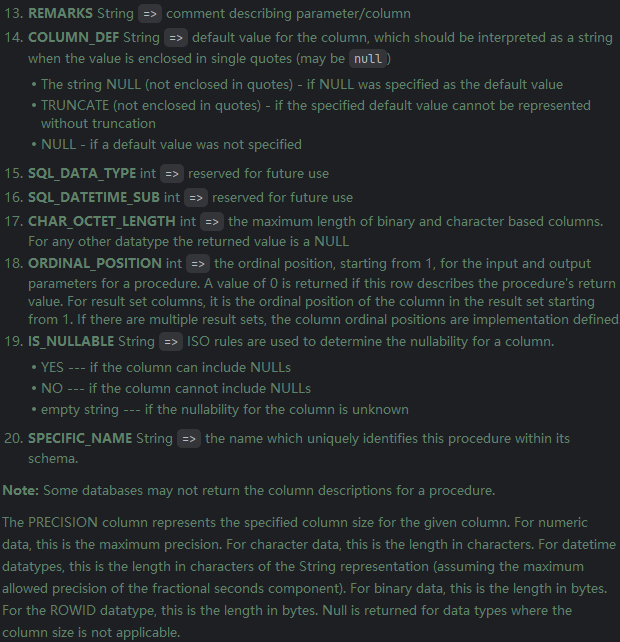
信息保存在ProcedureInfoBean中：



① 参数信息

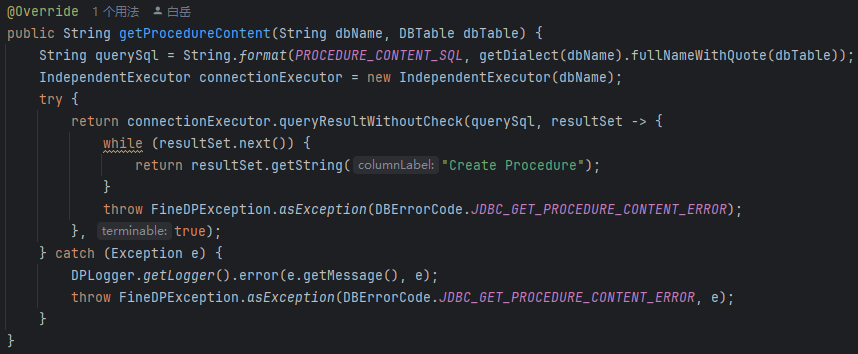
底层使用getProcedureColumns() 方法



② 存储过程内容

没有现成的JDBC接口，需要自己写SQL获取。使用一个SQL执行器IndependentExecutor（继承自JDBCExecutor）。



queryResultWithoutCheck方法底层把一个queryResult任务提交给SqlExecutorPool（一个sql执行线程池），返回FutureTask

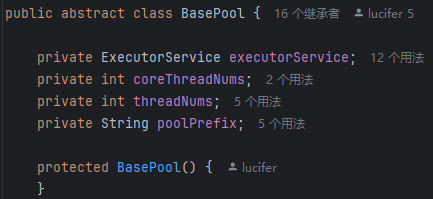


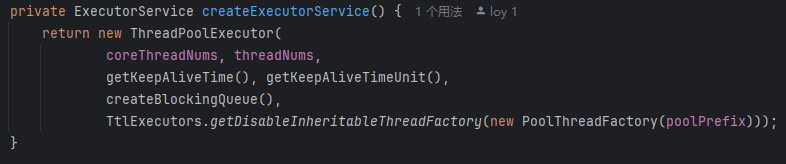
queryResult底层使用JDBC的statement.executeQuery方法，并用ResultSetVisitor（一个自定义的lambda函数）处理结果集

#### 四、FDL线程池及实现

##### 1. BasePool

底层封装了ExecutorService：

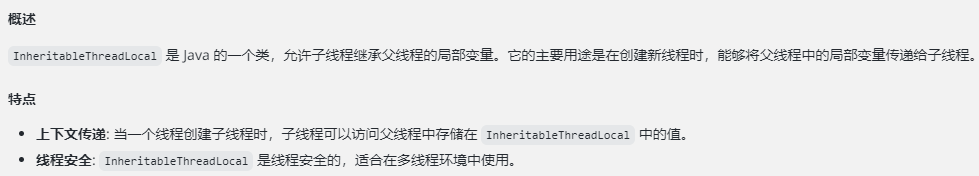


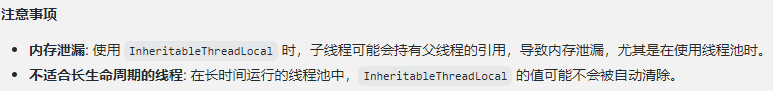


创建ThreadPoolExecutor的时候，自定义了阻塞队列和ThreadFactory

其中阻塞队列自定义了一个FIFOPriorityBlockingQueue用来保持FIFO（但实际未做到）

ThreadFactory先使用了自定义的PoolThreadFactory（主要用来给线程命名，便于发生错误后定位），然后使用getDisableInheritableThreadFactory方法，清除**InheritableThreadLocal**变量，避免内存泄露。





##### 2. SqlExecutorPool



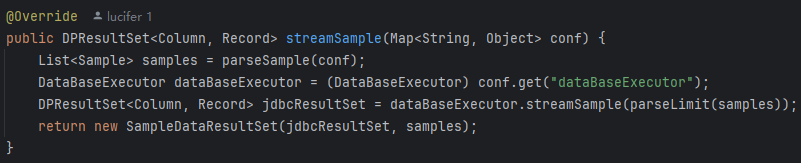
提供一个单例，同时指定线程前缀

#### 五、存储过程预览和执行

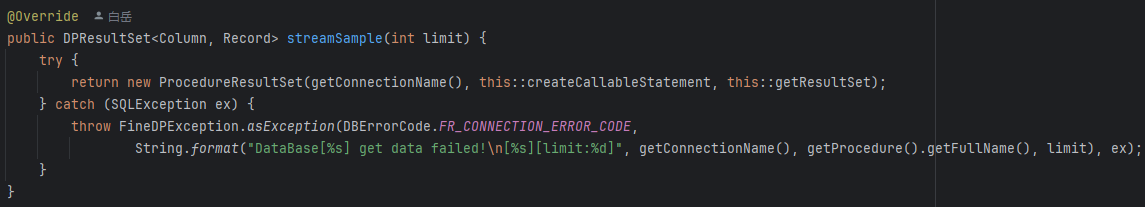
##### 1. 预览



用异步的方式，调用Datasource的streamSample方法获取结果集。



根据节点的配置信息，选择是查询结果集还是游标结果集，然后创建CallableStatement类型对象并调用execute方法，最后根据结果集类型调用相应的getResultSet方法获得结果集。

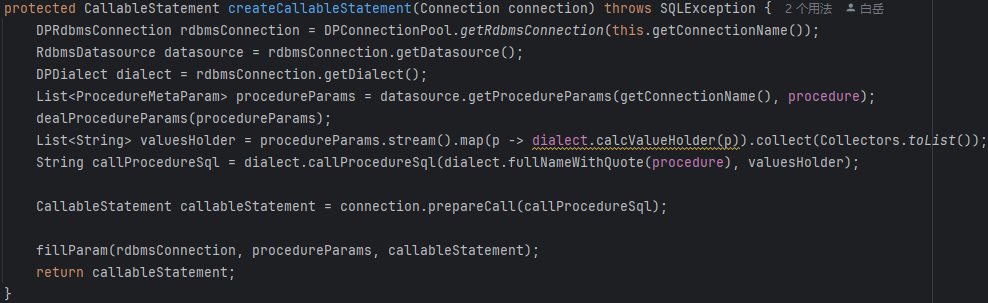


创建CallableStatement类型对象流程：

① 构造调用语句（一般是call …，特殊的如GaussDB上是借助select语句）

② 使用prepareCall方法创建CallableStatement类型对象

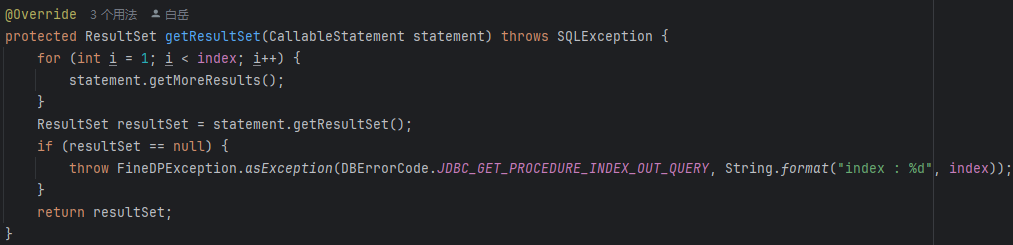
③ fillParam中调用statement.registerOutParameter方法，为OUT类型参数绑定类型



getResultSet方法：

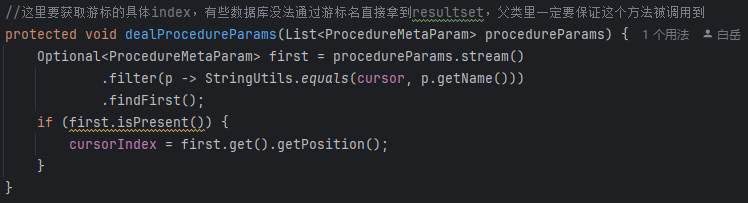
① 查询结果集

statement.getMoreResults()使迭代器指向下一个结果集，再用statement.getResultSet()获取结果集。

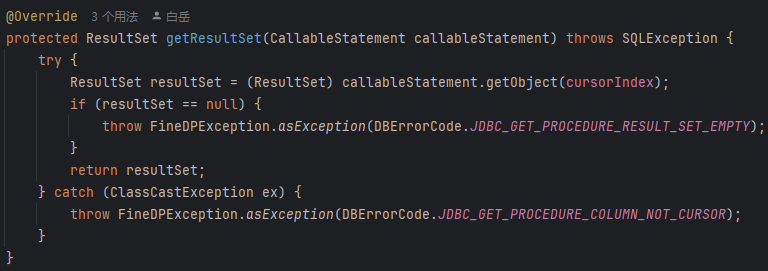


② 游标结果集

由于要用游标名获取指定结果集，先用游标名获得游标的位置



再通过callableStatement.getObject方法获取指定位置游标参数的输出结果（因为是通过游标类型参数返回结果集）



##### 2. 执行

预览只需要简单获取存储过程结果即可。执行需要走统一的节点执行方法，需要从前端获取配置信息后，判断是执行存储过程，然后再执行存储过程的逻辑。底层逻辑和预览基本相同。

区别：

预览时，为了简便起见，没有异常回滚的逻辑

执行时，会设置中断方法，在出现异常时调用statement.cancel()方法，取消调用。此外，还会关闭事务自动提交，在执行完之后手动提交事务，最后再打开，并使用connection.rollback()进行回滚。

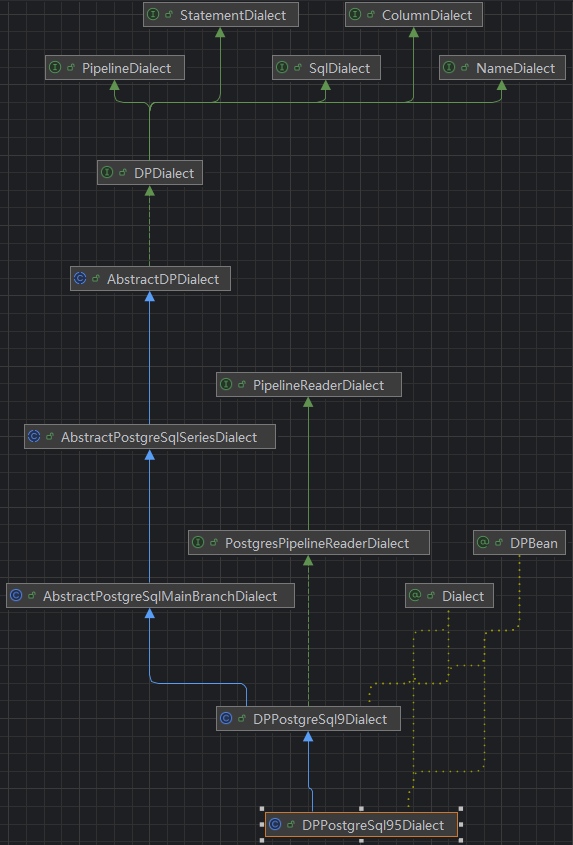
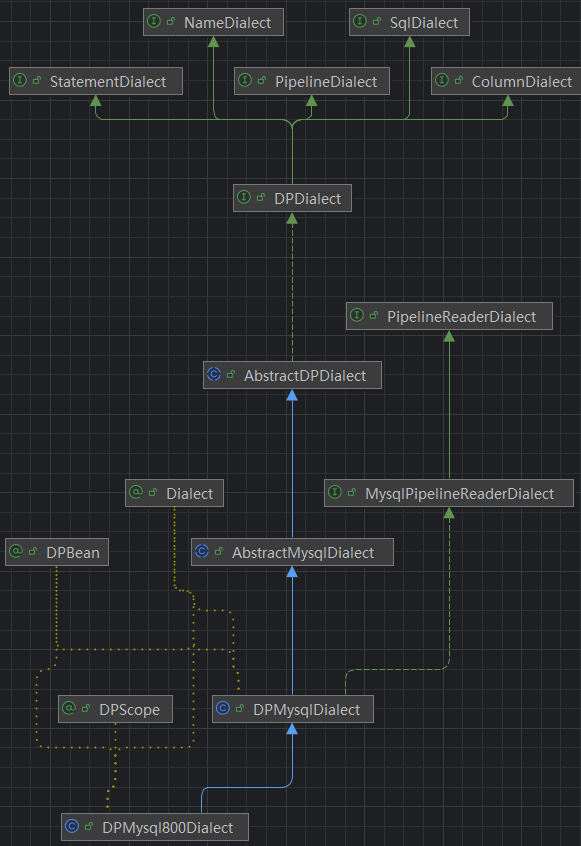


#### 六、FDL方言处理

##### 1. 数据结构

类图：

Postgre系（PostgreSql95为例）、MySQL系（Mysql800为例）

顶层接口：

① PipelineDialect：

就一个抽象方法utcTimestampMillisExpression，用来获取单位为毫秒的utc时间戳

② StatementDialect：

就一个抽象方法queryStatement，用来生成Statement类型对象。目前实现的方法里面都是用JDBC的createStatement方法生成的。

③ SqlDialect：

包含很多SQL语法相关的抽象方法，涵盖各种DDL、DML及其他

④ ColumnDialect：

包含DPMetaColumn转DialectMetaColumn的几种抽象方法，以及calcValueHolder方法（用来根据类型得到各种SQL语句中字段名称的格式，一般是显式指定类型，极个别特殊的例如vertica的geography/geometry类型用于加上数据库的处理函数）、useLabelAsName方法（部分数据库用label作为名字）

⑤ NameDialect

字段名称拼接、封闭符/连接符的定义和封装/解封等

二级接口

DPDialect：

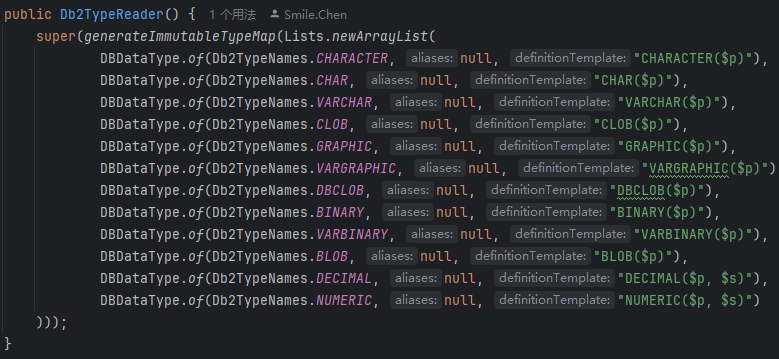
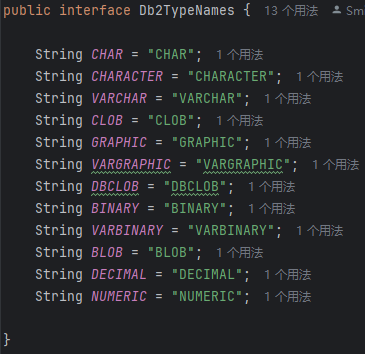
和数据库特性相关。包含返回字段方言类型对象方法、表/视图/主键/注释/窗口函数/upsert等较为高级的特性的支持判断等方法、方言版本匹配判断方法

此外支持管道取数的任务还需要实现PipelineReaderDialect接口

##### 2. 字段类型

通过JDBC获取的参数信息，其中包含的数据类型是标准SQL类型，需要转换成方言类型用来前端展示。

① 适配了TypeReader的数据库



直接用JDBC获取的字段名从TypeReader的Map中获取Template，然后填入precision、scale

② 未适配TypeReader的数据库

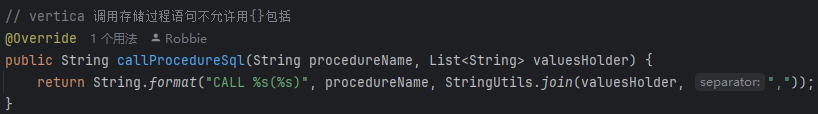


用JDBC获取的标准类型（一个int值）、precision、scale去Definition的Map中做匹配，获取Template。如果设置了precisionCompleteMatch / scaleCompleteMatch，需要找到precision/scale完全匹配的才行，否则获取到最合适的（模板precision/scale大于等于字段中，最小的那个）即可。

##### 3. 语法

① 存储过程调用语句

call语句template



参数显示指定类型，以及特殊格式

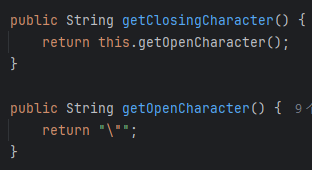


② 数据库封闭符

大部分使用双引号 “ ”

少部分使用反引号 ` `

个别使用中括号 [ ]，以及不能使用封闭符（返回空字符串）



##### 4. 各种类型Map

① Definition的map

标准SQL类型到前端展示格式的映射。给未适配TypeReader的数据库使用



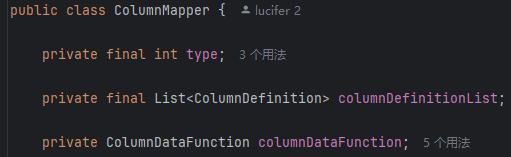


用JDBC获取的标准类型（一个int值）、precision、scale去Definition的Map中做匹配，获取Template。如果设置了precisionCompleteMatch / scaleCompleteMatch，需要找到precision/scale完全匹配的才行，否则获取到最合适的（模板precision/scale大于等于字段中，最小的那个）即可。

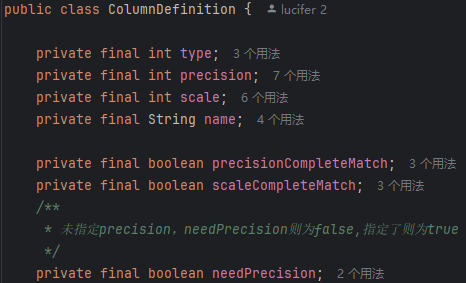
stream中分组，一个type对应多个格式，用的时候根据precision、scale匹配



ColumnMapper：除了type和对应的columndefinition列表，还有对应的字段读写处理函数（包括ReaderFunction和WriterFunction，在definition类加载的时候注册）

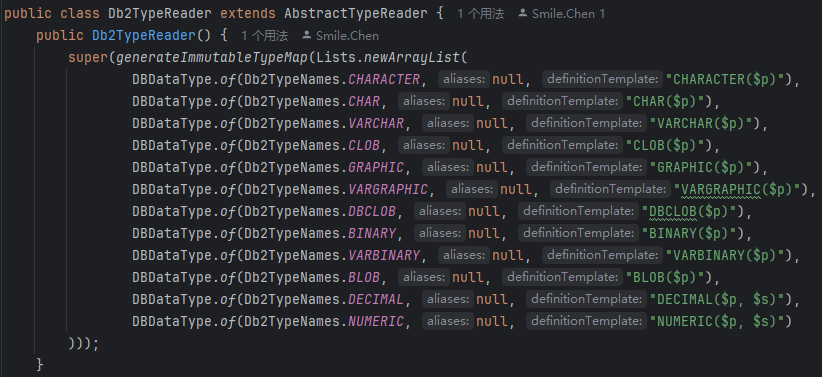


ColumnDefinition：负责匹配逻辑、占位符替换



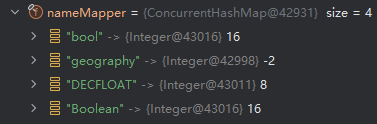
② TypeReader的Map

标准SQL类型到前端展示格式的映射。适配了TypeReader的数据库使用



③ ColumnDialect的Map（以GaussDB为例）

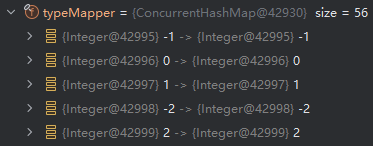
nameMapper：名称到type的映射，优先级最高



labelMapper：label到type的映射，优先级次高



typeMapper：type到type的映射，优先级最低。用标准type到自己的映射初始化，可以被子类覆盖

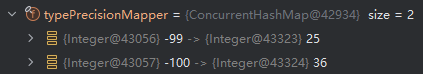


以上三种的用法：通过这三个mapper获得type之后，再去definition或typereader的map里面获取template

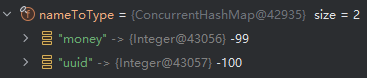
typeToJdbcTypeMapper：数据源的特殊type到jdbcType的映射



typePrecisionMapper：特殊type到指定precision的映射



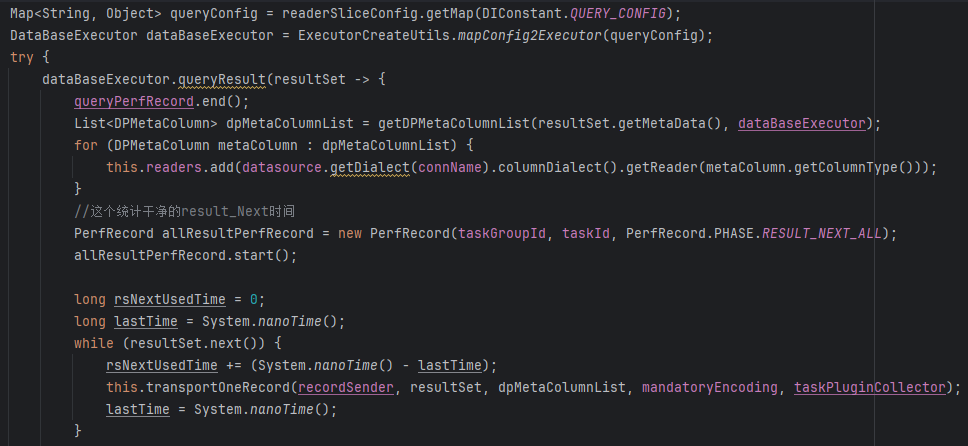
nameToType：给数据库自身读取完类型使用



#### 七、reader / writer底层传递数据逻辑

##### 1. reader

总体流程：



① 从reader的配置信息中获取查询语句信息

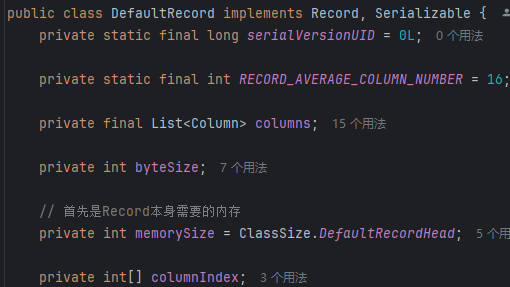
② 通过JDBC的getMetaData函数获取结果集元信息，并从中提取所有字段信息

③ 遍历所有字段，根据字段type获取相应的ReaderFunction，保存在readers中

④ 调用transportOneRecord方法传递结果集。底层实现借助自己写的Record（记录）和Column（字段）类型，初始时Record为空，使用注册好的ReaderFunction将DPMetaColumn类型转化为Column类型后，逐个加入Record中。

Record类型（以DefaultRecord为例）：

包含增加/修改/获取列、获取字段所占内存、复制列等方法



Column类型：

包含获取列数据、获取类型、与其他字段的类型转换方法和类型特有方法等

##### 2. BufferedRecordExchanger

reader和writer之间传递数据的类，实际上是一个自己写的缓冲区

顶层接口：



##### 3. reader

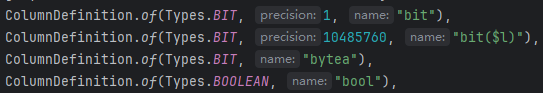
基本上是和writer相反的流程

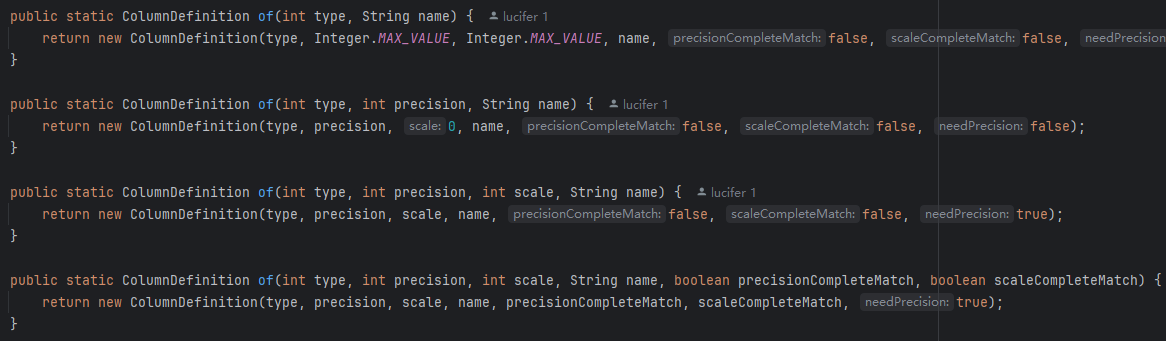
#### 八、涉及到的设计模式

##### 1. 创建型模式

① 工厂模式 / 抽象工厂模式

自定义的各种of方法

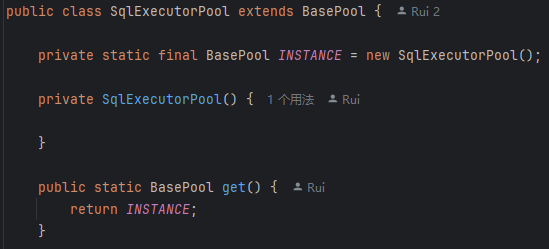




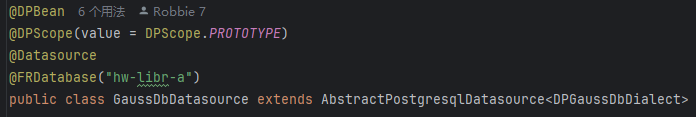
② 单例模式

各种地方都有用到

线程池：

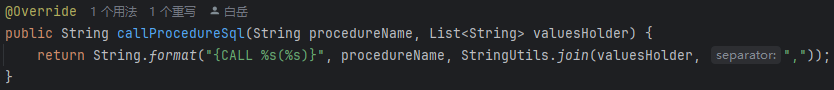


@DPBean注解（类似于原生的@Bean）

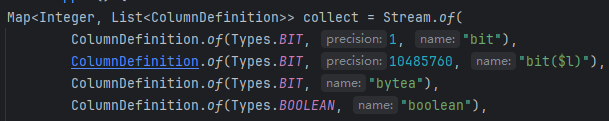


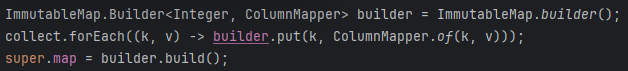
③ 建造者模式

构造call语句时参数部分用StringUtils.join方法建造。类似于StringBuilder的append方法。



创建ImmutableMap时使用建造者模式





④ 原型模式

暂时没用到。

##### 2. 结构型模式

① 适配器模式

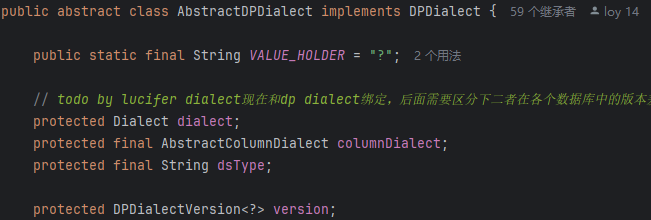
两个类分别实现了两个不同的接口。如果想要实现用其中一个接口统一调用这两个类的方法，可以考虑适配器模式。

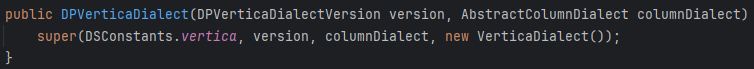
例如主工程中的PluginLoopContainerChecker实现LoopContainerChecker接口，插件工程中的LoopContainerCheckerProvider实现其他接口（或者没有实现接口）。采用一个PluginLoopContainerChecker类作为适配器类，在实现LoopContainerChecker接口的同时，内部调用LoopContainerCheckerProvider的check方法

② 桥接模式

本质上是组合，很多地方用到

AbstractDPDialect中，桥接了columnDialect、version两个属性



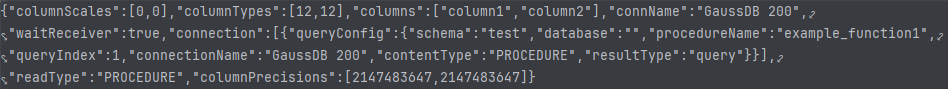


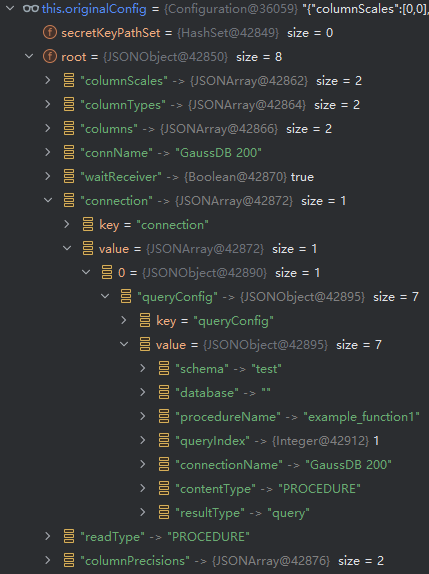
③ 过滤器模式（标准模式）

没用到，一般都是用stream流的filter方法

④ 组合模式（部分整体模式）

存放前端配置信息的Configuration类型对象，使用树形结构保存JSON对象，内部使用的封装了JSONObject类型





⑤ 装饰器模式

暂时没看到。和桥接模式/代理模式很接近。

装饰器模式强调对象自身功能的增强和扩展；代理模式强调核心功能和非核心功能的解耦；桥接模式强调对象自身和对象具有的属性的解耦。

⑥ 外观模式

数据源使用了外观模式，每个数据源都有一个Façade类继承自AbstractDatasourceFacade，主要包括获取数据源名称、获取支持的功能等等

⑦ 享元模式

主要用于各种池：

数据库连接池（使用DruidDatasource）、SQL执行线程池、预览任务线程池

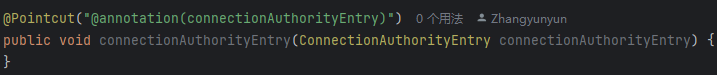
⑧ 代理模式

主要在Spring AOP中体现，目前暂时只看到鉴权时候用到

Controller层：



切面类：

##### 3. 行为型模式

① 责任链模式

SpringMVC的拦截器、Servlet的过滤器、log4j

② 命令模式

暂时没看到，如果硬要说的话，编写SQL语句交给IndependentExecutor执行应该算一种。

只不过这里的命令是直接调用JDBC的接口生成的，而不是自己定义的类

③ 解释器模式

一般不会自己写。通常使用已有方法：编译器、SQL解释器、正则匹配等等。

④ 迭代器模式

一般不会自己写。用过的：JDBC的statement.getMoreResults()（指向下一个结果集）、resultSet.next()（指向下一行结果）、增强for循环（底层是Iterator）

⑤ 中介者模式

FDL中比较少见。硬要说的话reader和writer之间的数据传递可以算，中介者是一个缓冲区模型

⑥ 备忘录模式

任务的配置点击保存后会持久化到配置文件中，每次打开任务可以读取。

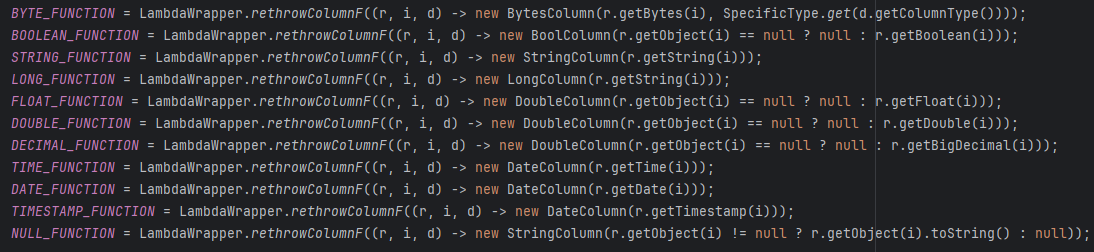
⑦ 观察者模式

暂时没看到。但事件监听等功能上应该用的很多

⑧ 状态模式/策略模式/访问者模式

其实都很类似。代码中一般策略模式用的最多。

例如每一种字段类型都写一个Column类型，以及各种用到lambda函数的地方，实际上都用了策略模式



⑨ 空对象模式

基本不会用，会让bug排查变得很困难

⑩ 模板模式

暂时没看到。执行任务的类ReaderRunner中的run方法有点类似，规定了由init / prepare / startRead / post构成的运行流程，但这几个方法并不是在ReaderRunner这个类中定义的。