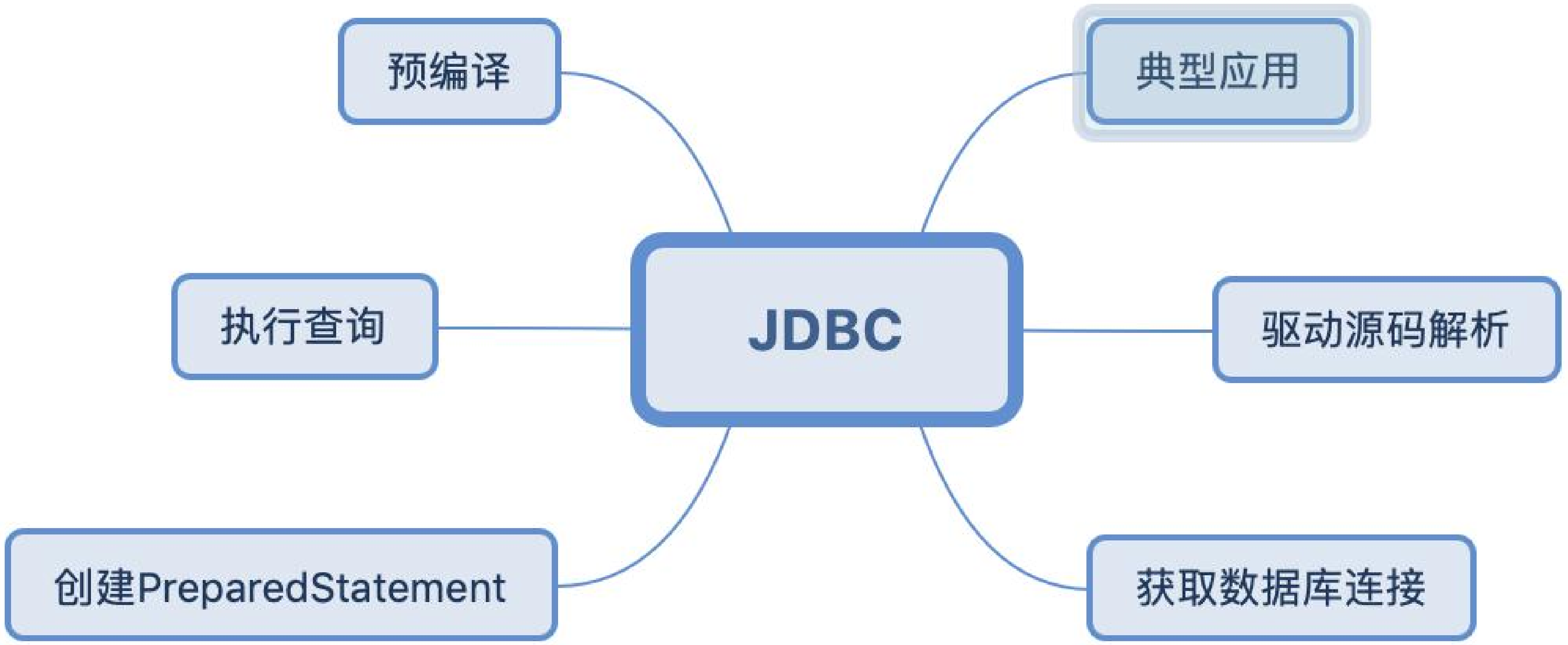
Java高级工程师

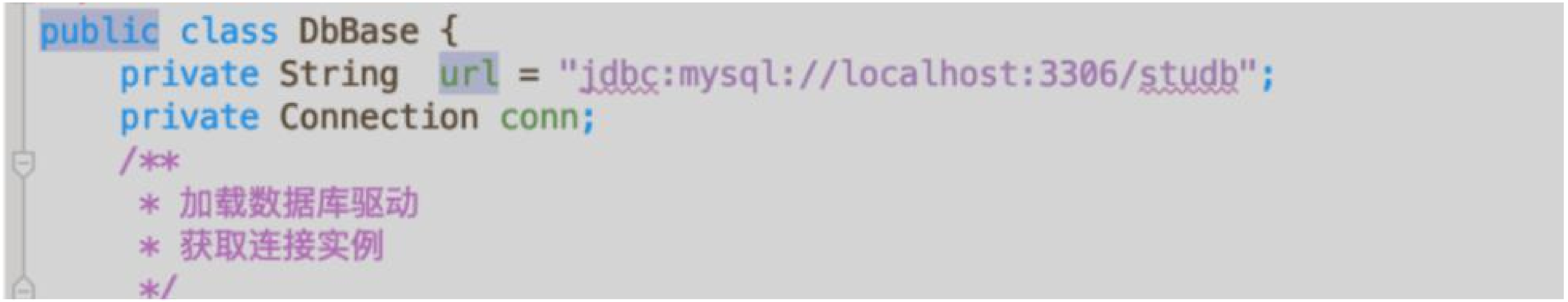
JDBC篇

课程大纲

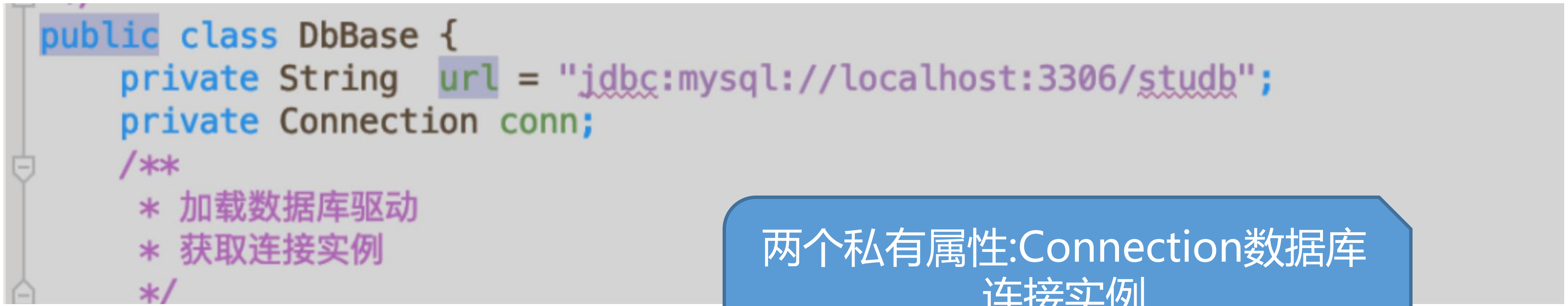


JDBC典型应用

建一个类,把数据库操作封到底层


```
public class DbBase {
    private String  url = "jdbc:mysql://localhost:3306/studb";
    private Connection conn;
    /**
     * 加载数据库驱动
     * 获取连接实例
     */
```

建一个类,把数据库操作封到底层



```
public class DbBase {
    private String  url = "jdbc:mysql://localhost:3306/studb";
    private Connection conn;
    /**
     * 加载数据库驱动
     * 获取连接实例
     */
```

两个私有属性:Connection数据库连接实例
url 数据库连接字符串

Java是怎样连接数据库的?

频度:高

难度:中

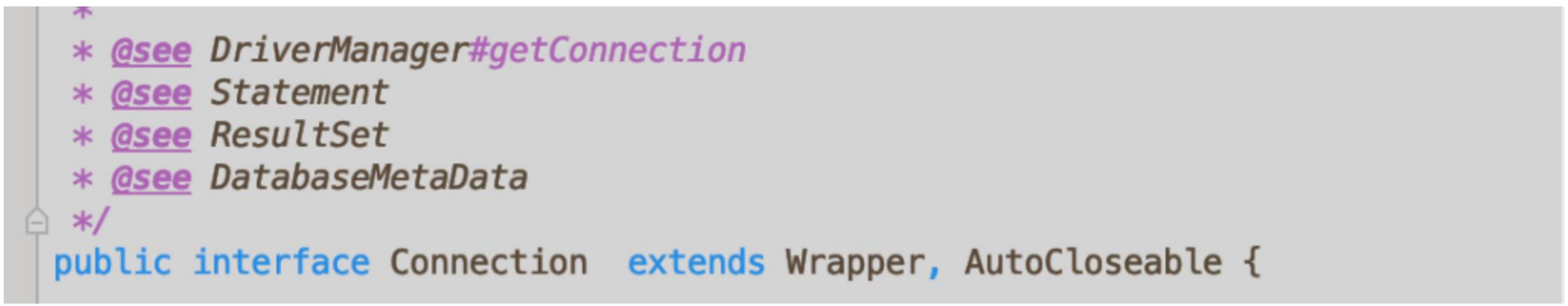
通过率:低

答案

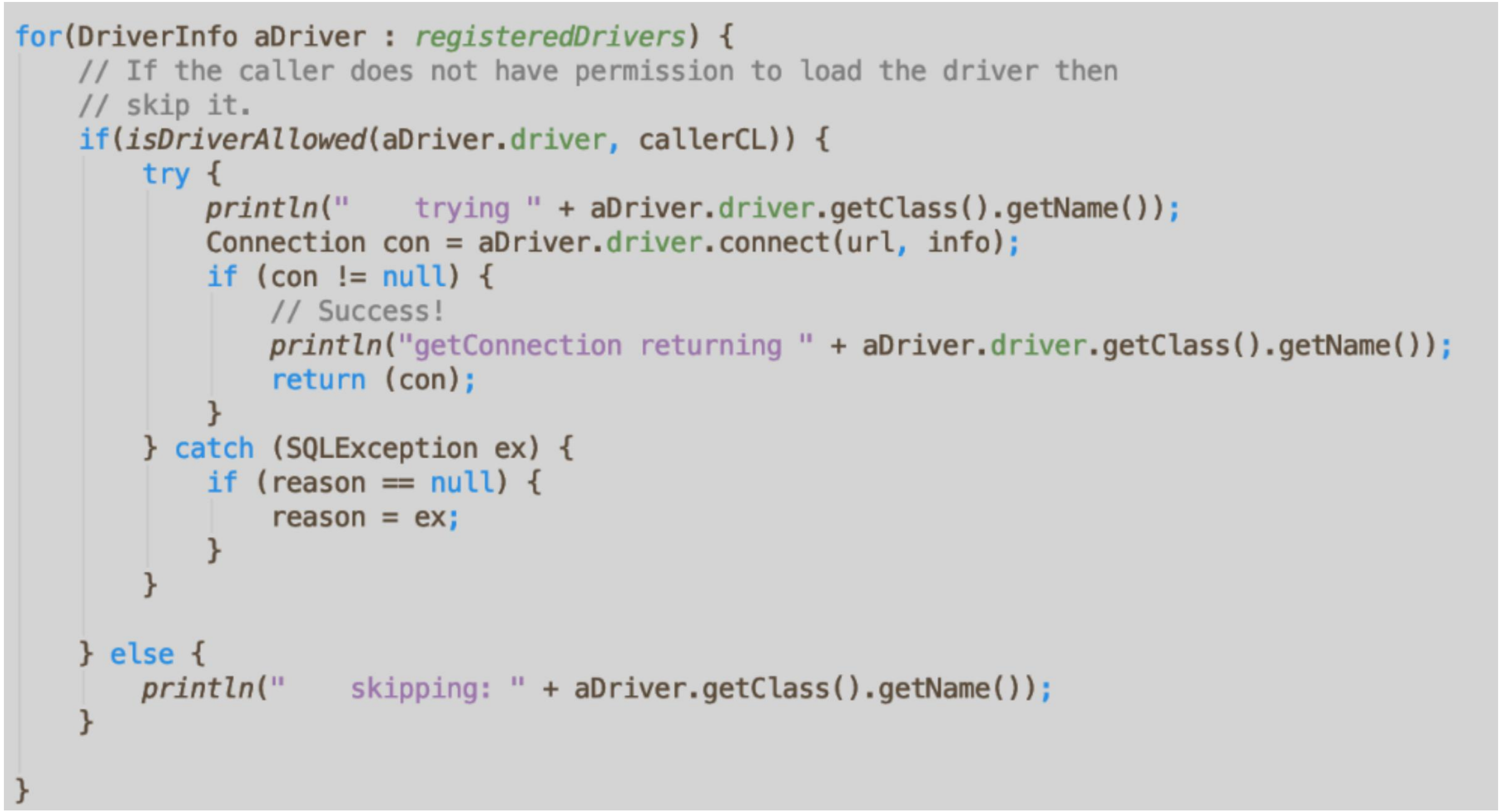
通过DirveManager加载符合Jdbc标准的驱动(反射)

通过connection接口约定获取数据库连接实例

Connection被作为接口约定抽象出来

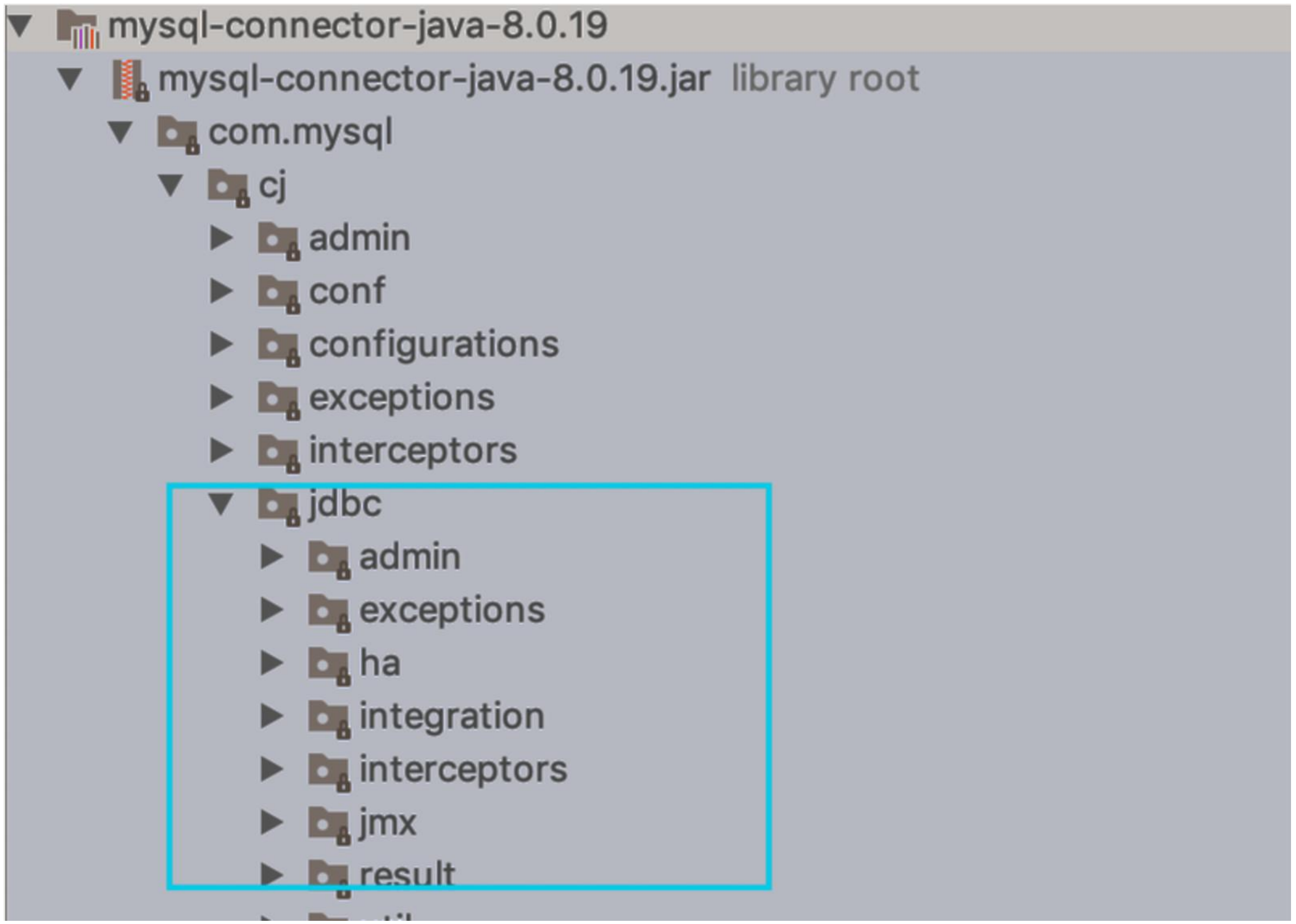

```
 * @see DriverManager#getConnection
 * @see Statement
 * @see ResultSet
 * @see DatabaseMetaData
 */
public interface Connection  extends Wrapper, AutoCloseable {
```

通过Driver接口调用厂商驱动获取数据库连接实例



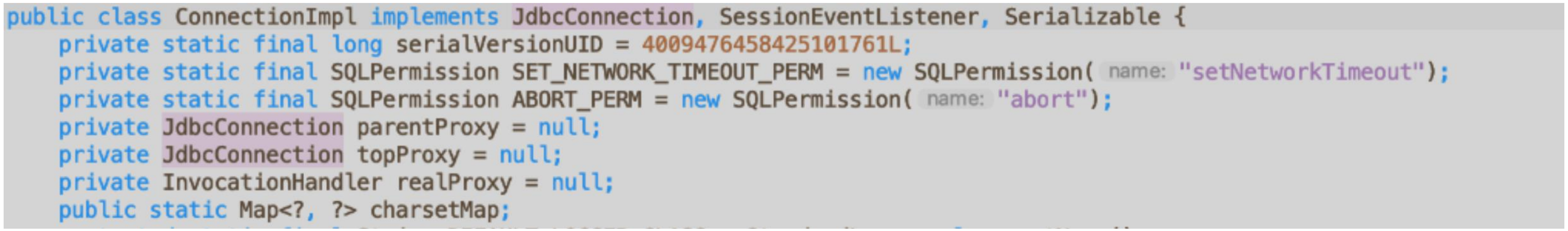
```
for(DriverInfo aDriver : registeredDrivers) {
    // If the caller does not have permission to load the driver then
    // skip it.
    if(isDriverAllowed(aDriver.driver, callerCL)) {
        try {
            println("    trying " + aDriver.driver.getClass().getName());
            Connection con = aDriver.driver.connect(url, info);
            if (con != null) {
                // Success!
                println("getConnection returning " + aDriver.driver.getClass().getName());
                return (con);
            }
        } catch (SQLException ex) {
            if (reason == null) {
                reason = ex;
            }
        }

    } else {
        println("    skipping: " + aDriver.getClass().getName());
    }

}
```

其实jdbc是厂商的驱动实现的



```
▼ mysql-connector-java-8.0.19
  ▼ mysql-connector-java-8.0.19.jar  library root
    ▼ com.mysql
      ▼ cj
        ▶ admin
        ▶ conf
        ▶ configurations
        ▶ exceptions
        ▶ interceptors
        ▼ jdbc
          ▶ admin
          ▶ exceptions
          ▶ ha
          ▶ integration
          ▶ interceptors
          ▶ jmx
          ▶ result
```

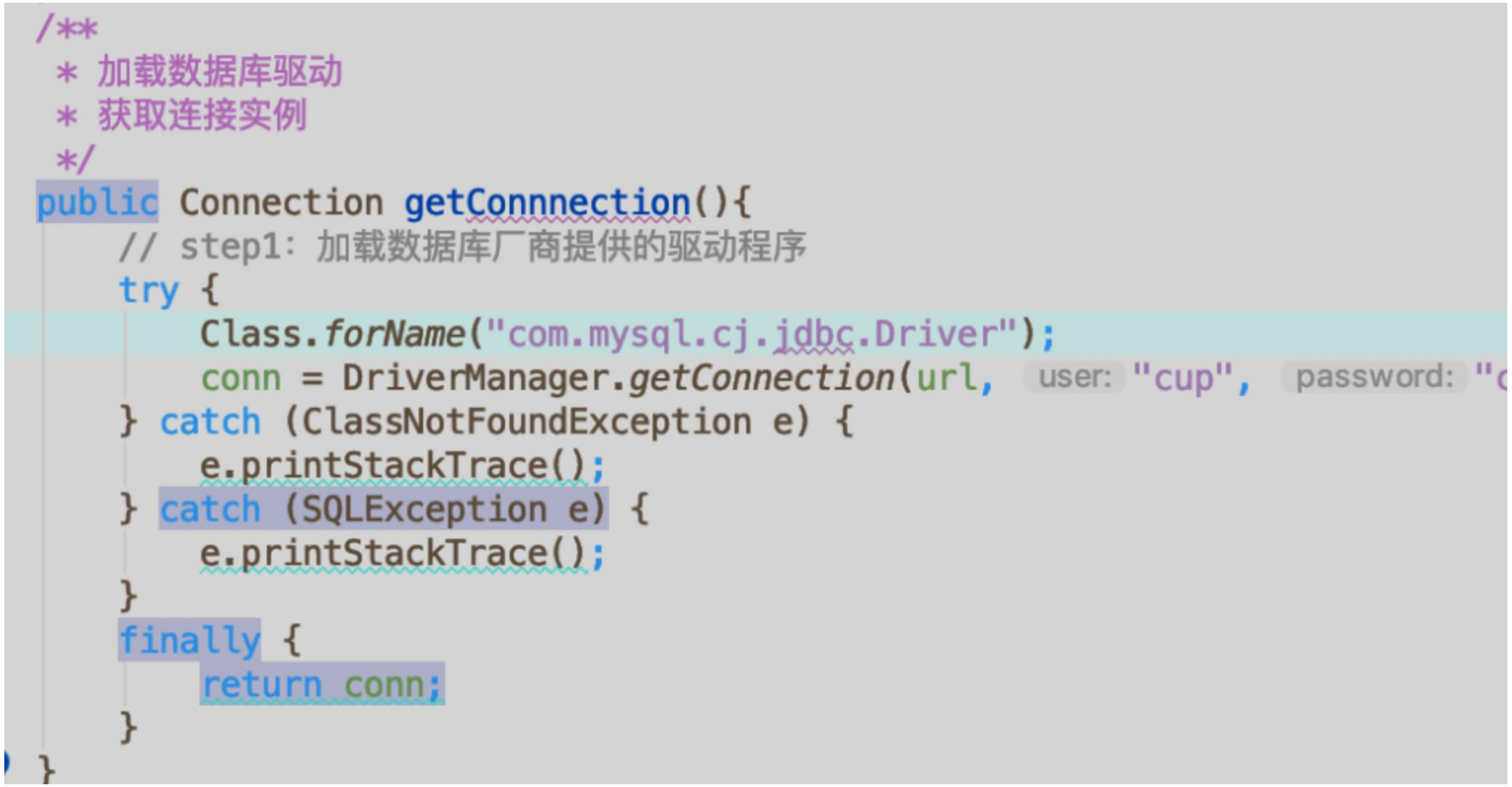
厂商的驱动实现connection接口



```
public class ConnectionImpl implements JdbcConnection, SessionEventListener, Serializable {
    private static final long serialVersionUID = 4009476458425101761L;
    private static final SQLPermission SET_NETWORK_TIMEOUT_PERM = new SQLPermission( name: "setNetworkTimeout");
    private static final SQLPermission ABORT_PERM = new SQLPermission( name: "abort");
    private JdbcConnection parentProxy = null;
    private JdbcConnection topProxy = null;
    private InvocationHandler realProxy = null;
    public static Map<?, ?> charsetMap;
```

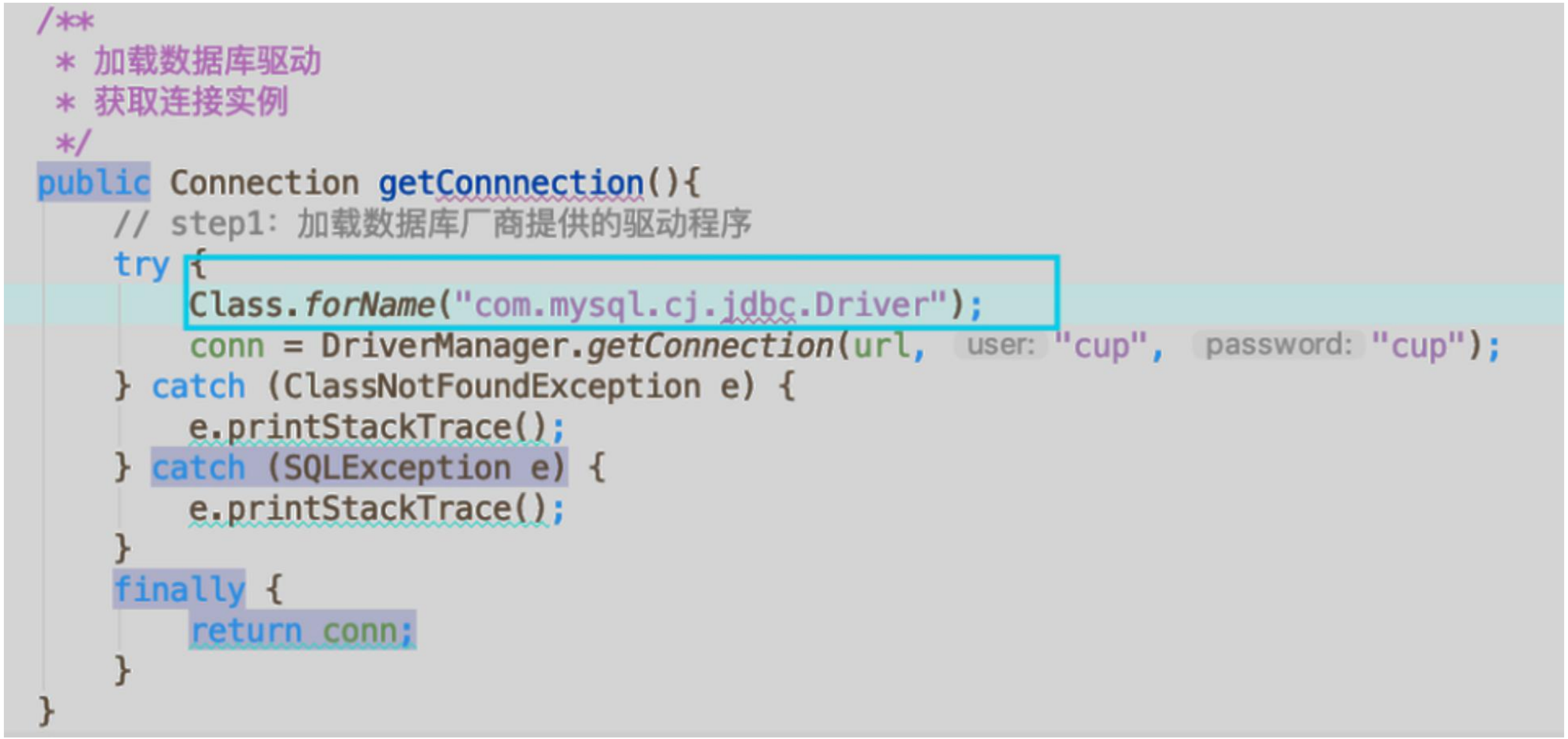
加载厂商驱动

注册驱动,用连接字符串和用户名/密码获取连接器实例



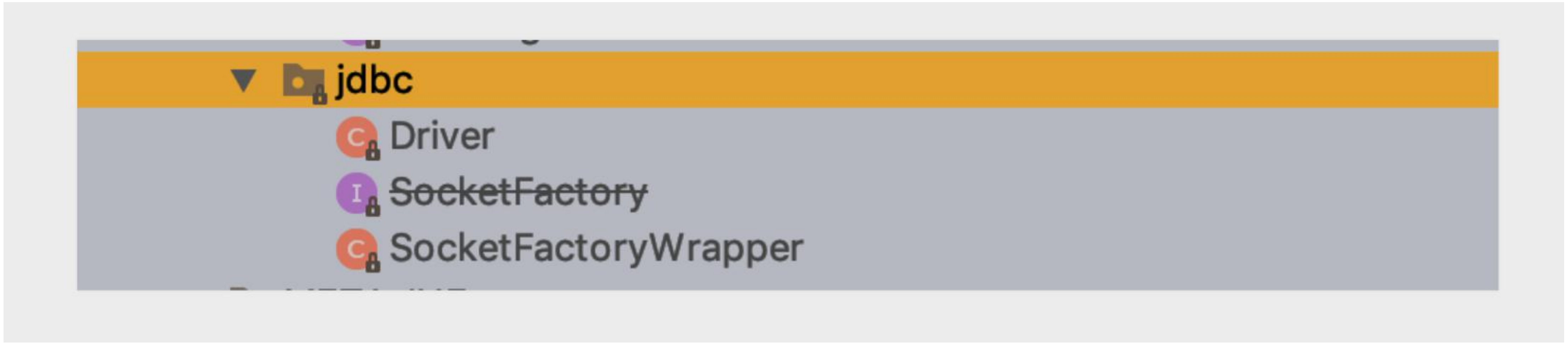
```
/**
 * 加载数据库驱动
 * 获取连接实例
 */
public Connection getConnnection(){
    // step1: 加载数据库厂商提供的驱动程序
    try {
        Class.forName("com.mysql.cj.jdbc.Driver");
        conn = DriverManager.getConnection(url, user: "cup", password: "c
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    } catch (SQLException e) {
        e.printStackTrace();
    }
    finally {
        return conn;
    }
}
```

这个驱动名怎么来?



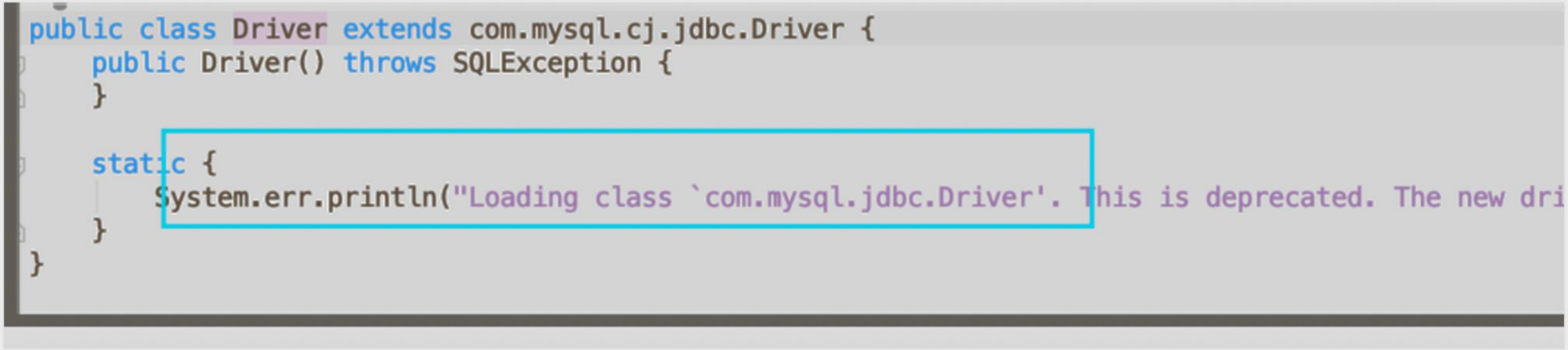
```
/**
 * 加载数据库驱动
 * 获取连接实例
 */
public Connection getConnnection(){
    // step1: 加载数据库厂商提供的驱动程序
    try {
        Class.forName("com.mysql.cj.jdbc.Driver");
        conn = DriverManager.getConnection(url, user: "cup", password: "cup");
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    } catch (SQLException e) {
        e.printStackTrace();
    }
    finally {
        return conn;
    }
}
```

从jdbc包中找



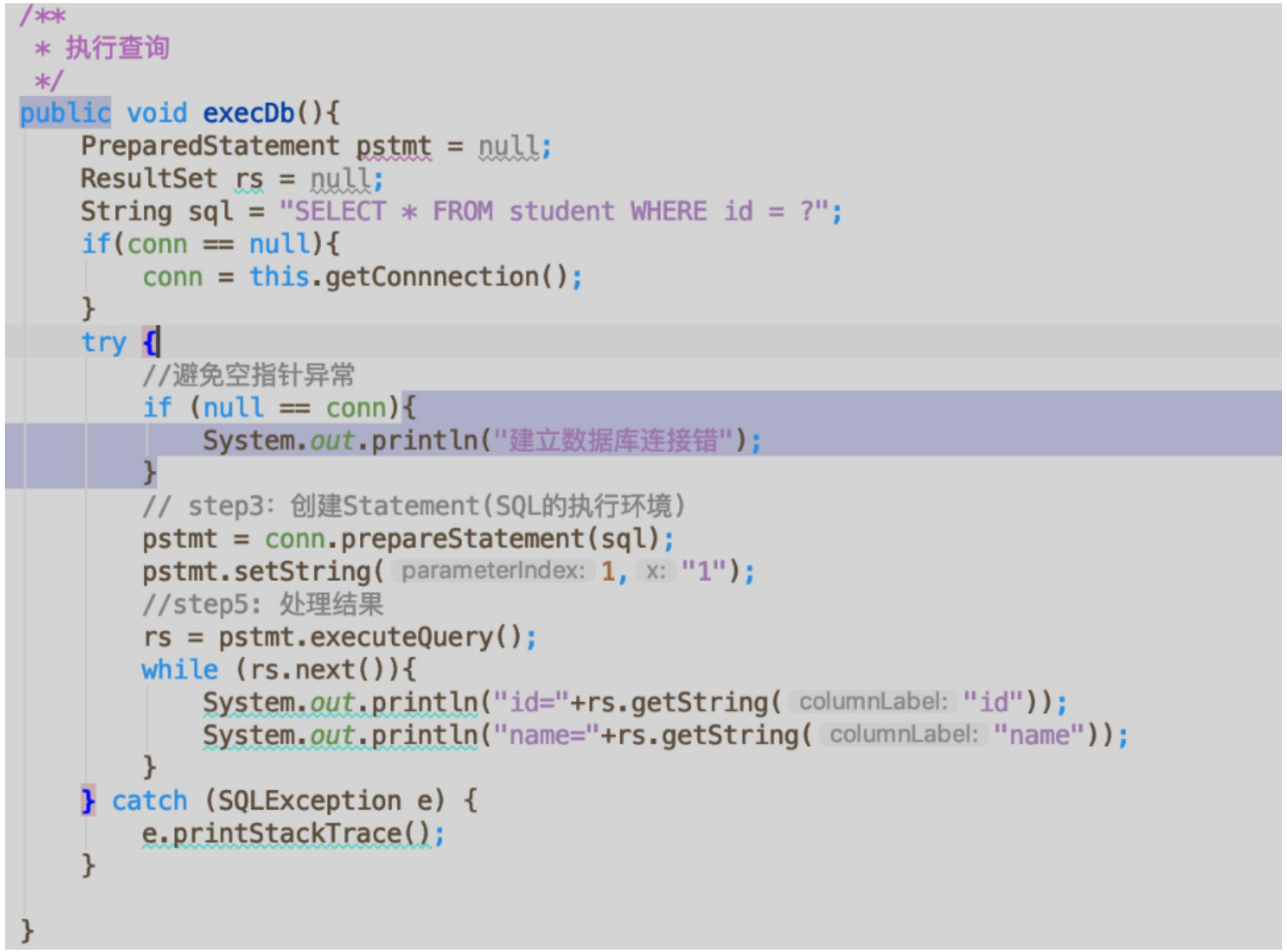
jdbc
Driver
~~SocketFactory~~
SocketFactoryWrapper

从jdbc包中找



```
public class Driver extends com.mysql.cj.jdbc.Driver {
    public Driver() throws SQLException {
    }

    static {
        System.err.println("Loading class `com.mysql.jdbc.Driver'. This is deprecated. The new dri
    }
}
```

执行一个简单的查询



```
/**
 * 执行查询
 */
public void execDb(){
    PreparedStatement pstmt = null;
    ResultSet rs = null;
    String sql = "SELECT * FROM student WHERE id = ?";
    if(conn == null){
        conn = this.getConnnection();
    }
    try {
        //避免空指针异常
        if (null == conn){
            System.out.println("建立数据库连接错");
        }
        // step3: 创建Statement(SQL的执行环境)
        pstmt = conn.prepareStatement(sql);
        pstmt.setString( parameterIndex: 1, x: "1");
        //step5: 处理结果
        rs = pstmt.executeQuery();
        while (rs.next()){
            System.out.println("id="+rs.getString( columnLabel: "id"));
            System.out.println("name="+rs.getString( columnLabel: "name"));
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }

}
```

PreparedStatement是什么?

和Statement有什么区别?

频度:高

难度:中

通过率:低

Statement

The object used for executing a static SQL statement and returning the results it produces.

解读

执行的是静态的sql语句

解读

不会把用户非法输入的单引号用\反斜杠做转义

解读

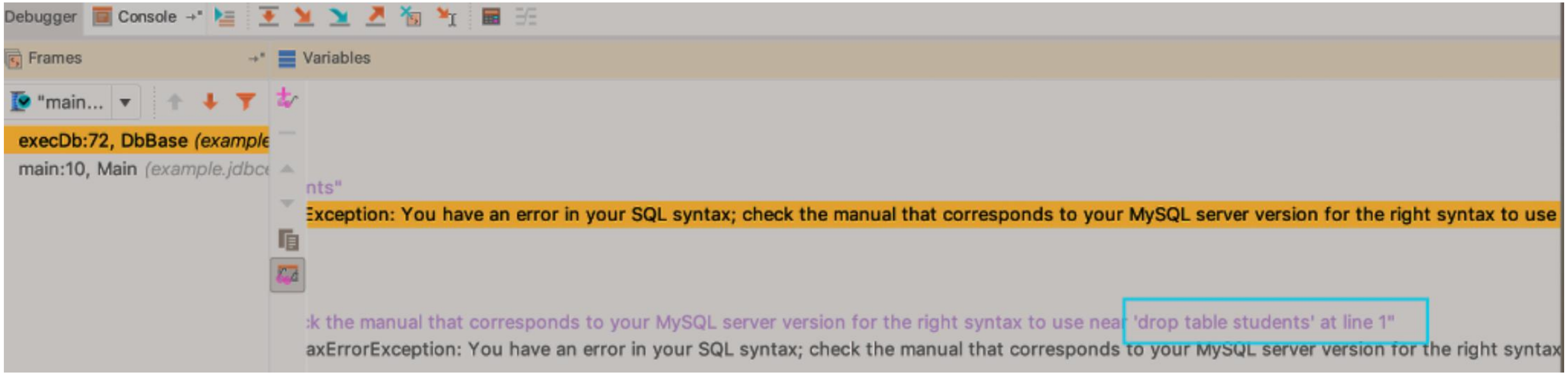
有sql 注入的风险

看一个例子

- 注入一个危险的sql脚本
- 特意写错表名,防止真删了

```
sqlx = "SELECT * FROM student WHERE id ='1"+"';drop table students";
stm = conn.createStatement();
stm.execute(sqlx);
```

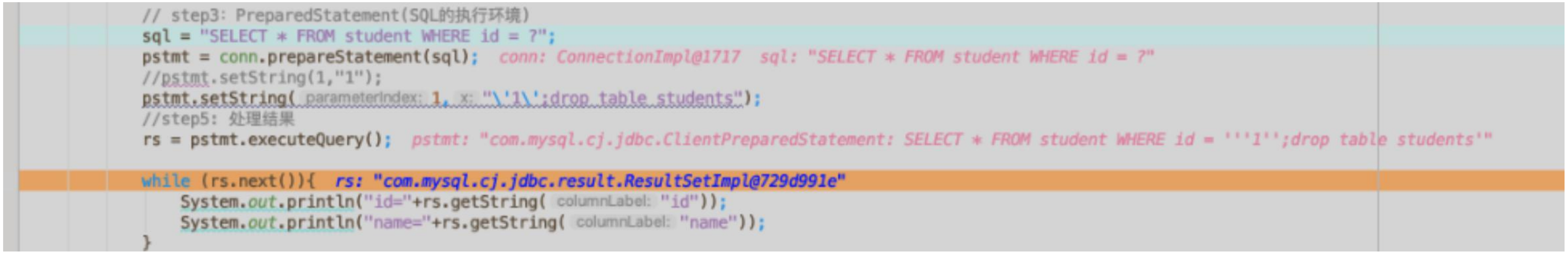
看看结果,其实已经去执行了



```
execDb:72, DbBase (example
main:10, Main (example.jdbc
```

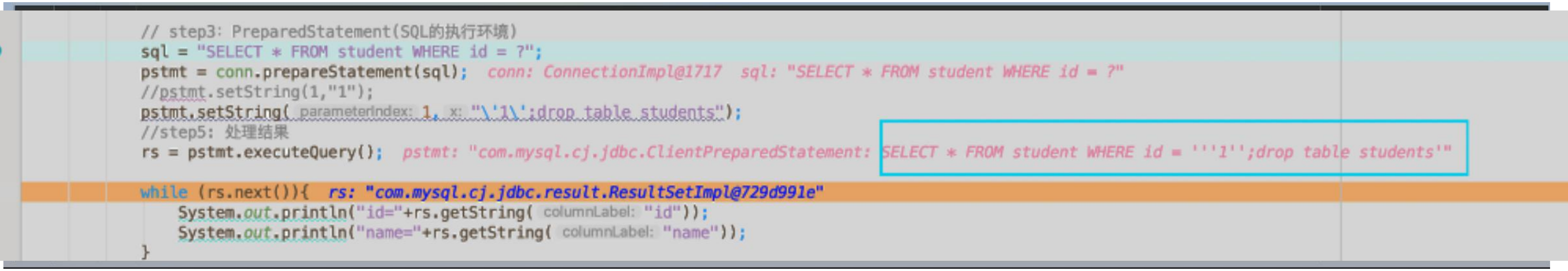
```
nts"
Exception: You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use
:k the manual that corresponds to your MySQL server version for the right syntax to use near 'drop table students' at line 1"
axErrorException: You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax
```

PreparedStatement防止注入的机制



```
// step3: PreparedStatement(SQL的执行环境)
sql = "SELECT * FROM student WHERE id = ?";
pstmt = conn.prepareStatement(sql);  conn: ConnectionImpl@1717  sql: "SELECT * FROM student WHERE id = ?"
//pstmt.setString(1,"1");
pstmt.setString( parameterIndex: 1, x: "\'1\';drop table students");
//step5: 处理结果
rs = pstmt.executeQuery();  pstmt: "com.mysql.cj.jdbc.ClientPreparedStatement: SELECT * FROM student WHERE id = '''1'';drop table students'"

while (rs.next()){  rs: "com.mysql.cj.jdbc.result.ResultSetImpl@729d991e"
    System.out.println("id="+rs.getString( columnLabel: "id"));
    System.out.println("name="+rs.getString( columnLabel: "name"));
}
```

通过转义的方式是无效的



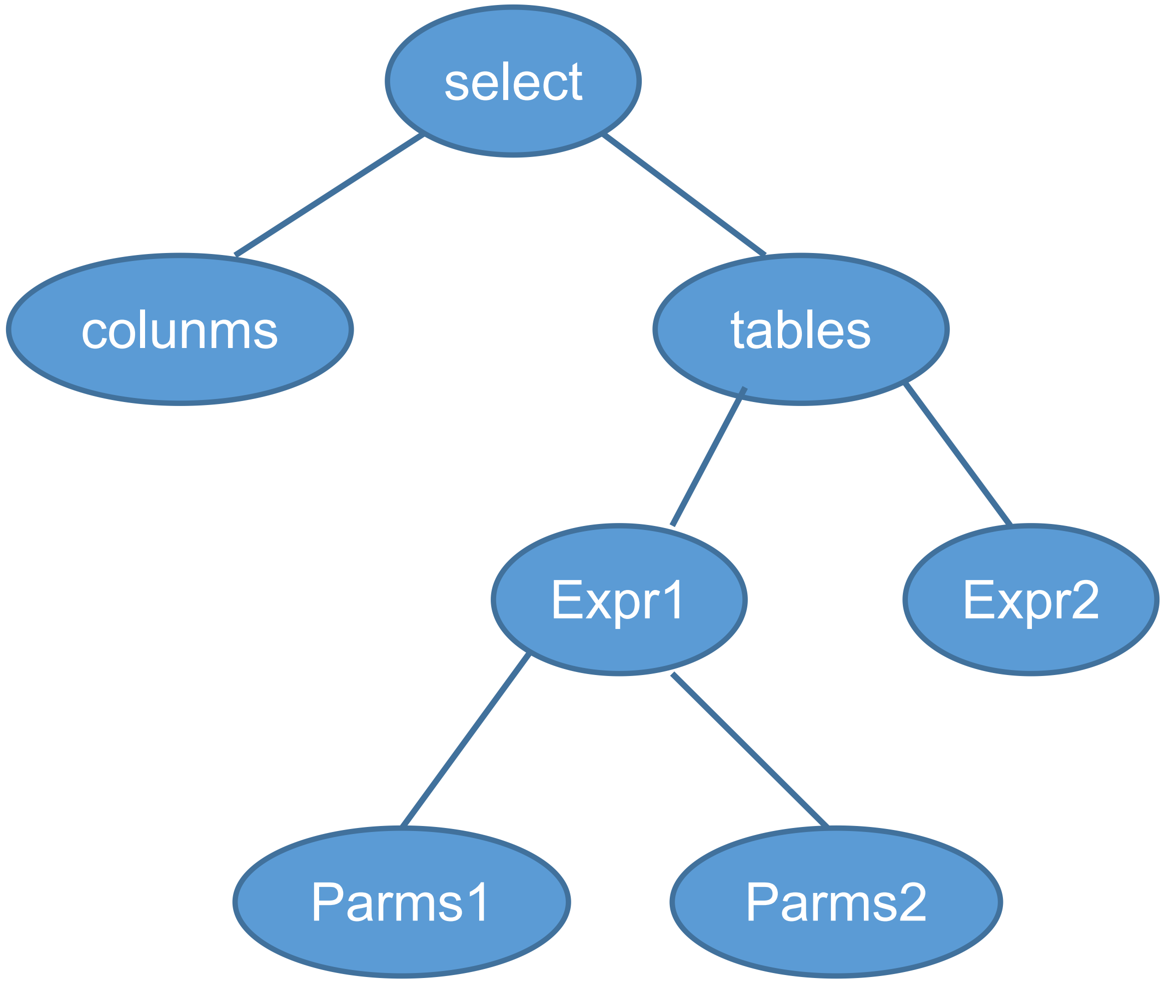
```
// step3: PreparedStatement(SQL的执行环境)
sql = "SELECT * FROM student WHERE id = ?";
pstmt = conn.prepareStatement(sql);  conn: ConnectionImpl@1717  sql: "SELECT * FROM student WHERE id = ?"
//pstmt.setString(1,"1");
pstmt.setString( parameterIndex: 1, x: "\'1\';drop table students");
//step5: 处理结果
rs = pstmt.executeQuery();  pstmt: "com.mysql.cj.jdbc.ClientPreparedStatement: SELECT * FROM student WHERE id = '''1'';drop table students'"

while (rs.next()){  rs: "com.mysql.cj.jdbc.result.ResultSetImpl@729d991e"
    System.out.println("id="+rs.getString( columnLabel: "id"));
    System.out.println("name="+rs.getString( columnLabel: "name"));
}
```

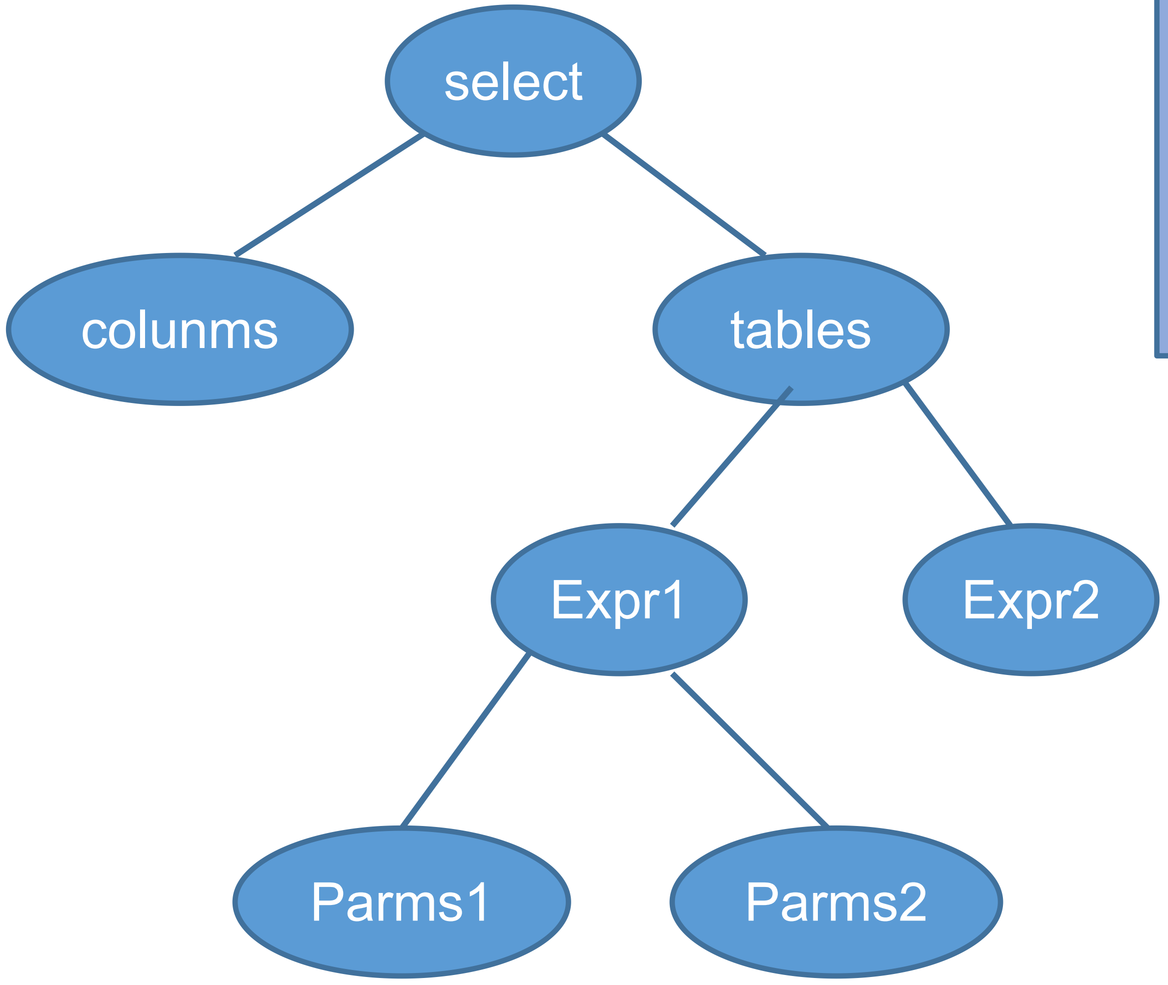
区别二

PreparedStatement是预编译的,对于批量处理可以大大提高效率. 也叫**JDBC**存储过程

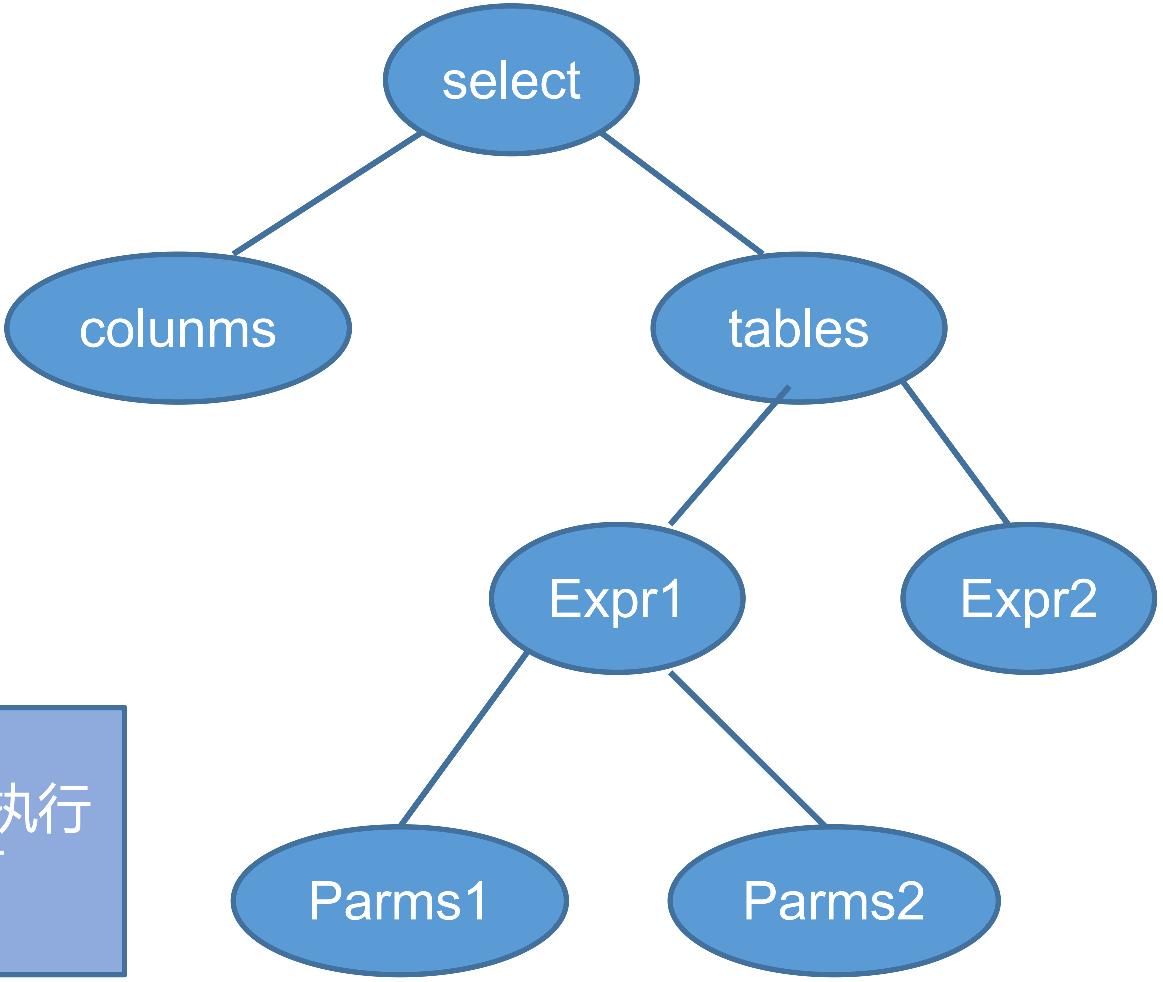
什么是预编译

通常的实现,sql在编译时形成一个语法树,例如,按左规则范式的方式:


select
colunms
tables
Expr1
Expr2
Parms1
Parms2

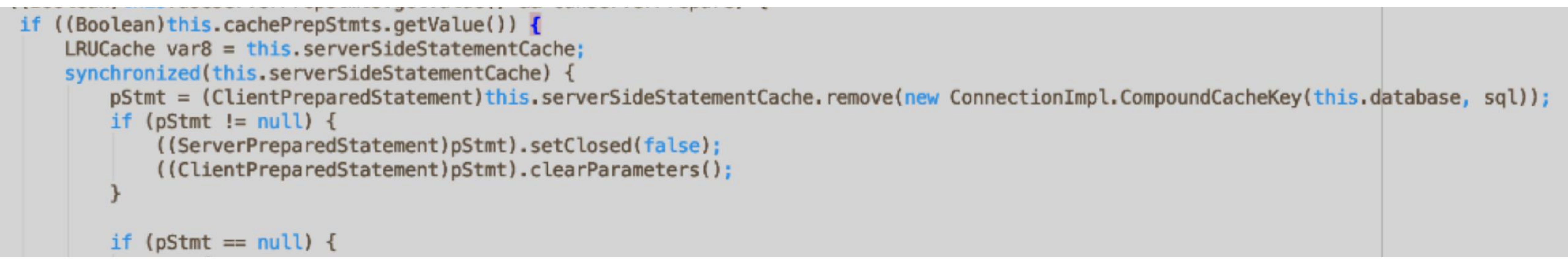


开销主要在建树上



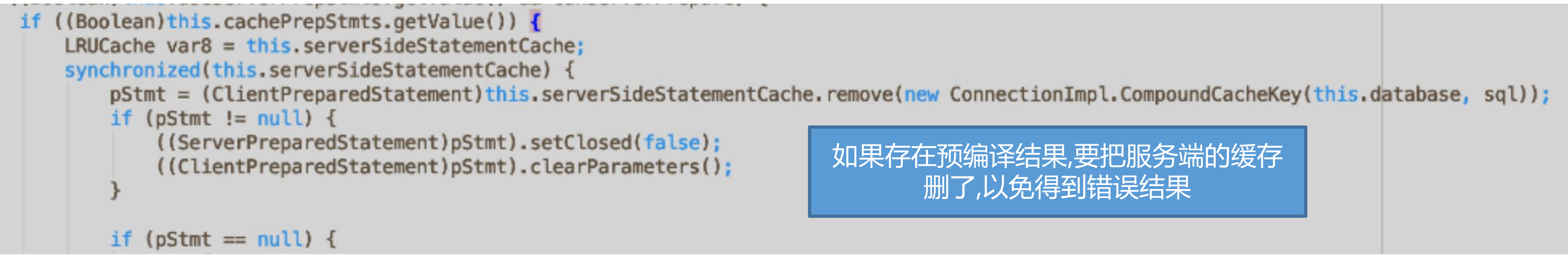
预编译就是建好了树,执行时给参数赋值即可

初始化时,先进行缓存查找



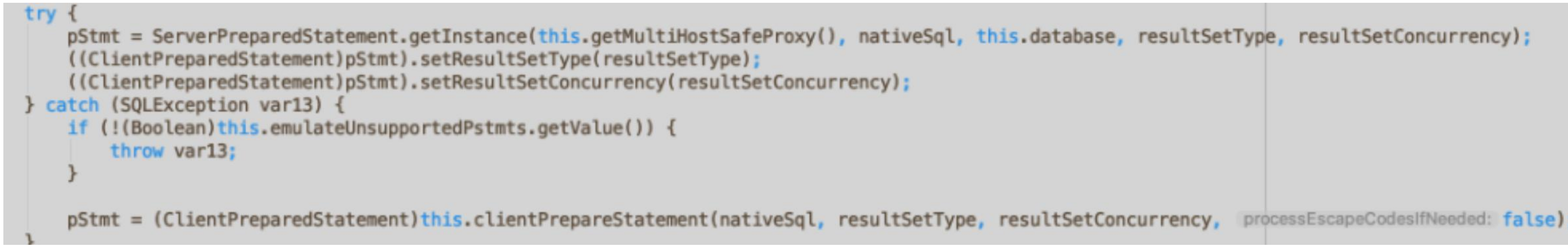
```
if ((Boolean)this.cachePrepStmts.getValue()) {
    LRUCache var8 = this.serverSideStatementCache;
    synchronized(this.serverSideStatementCache) {
        pStmt = (ClientPreparedStatement)this.serverSideStatementCache.remove(new ConnectionImpl.CompoundCacheKey(this.database, sql));
        if (pStmt != null) {
            ((ServerPreparedStatement)pStmt).setClosed(false);
            ((ClientPreparedStatement)pStmt).clearParameters();
        }

        if (pStmt == null) {
```

初始化时,先进行缓存查找


```
if ((Boolean)this.cachePrepStmts.getValue()) {
    LRUCache var8 = this.serverSideStatementCache;
    synchronized(this.serverSideStatementCache) {
        pStmt = (ClientPreparedStatement)this.serverSideStatementCache.remove(new ConnectionImpl.CompoundCacheKey(this.database, sql));
        if (pStmt != null) {
            ((ServerPreparedStatement)pStmt).setClosed(false);
            ((ClientPreparedStatement)pStmt).clearParameters();
        }

        if (pStmt == null) {
```

如果存在预编译结果,要把服务端的缓存删了,以免得到错误结果

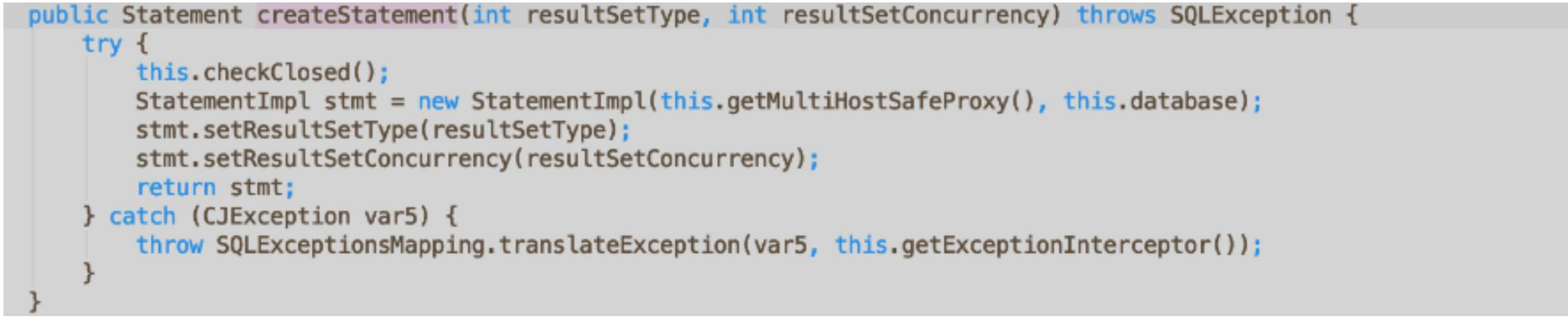
清理干净现场后,执行初始化



```
try {
    pStmt = ServerPreparedStatement.getInstance(this.getMultiHostSafeProxy(), nativeSql, this.database, resultSetType, resultSetConcurrency);
    ((ClientPreparedStatement)pStmt).setResultSetType(resultSetType);
    ((ClientPreparedStatement)pStmt).setResultSetConcurrency(resultSetConcurrency);
} catch (SQLException var13) {
    if (!(Boolean)this.emulateUnsupportedPstmts.getValue()) {
        throw var13;
    }

    pStmt = (ClientPreparedStatement)this.clientPrepareStatement(nativeSql, resultSetType, resultSetConcurrency,  processEscapeCodesIfNeeded: false)
}
```

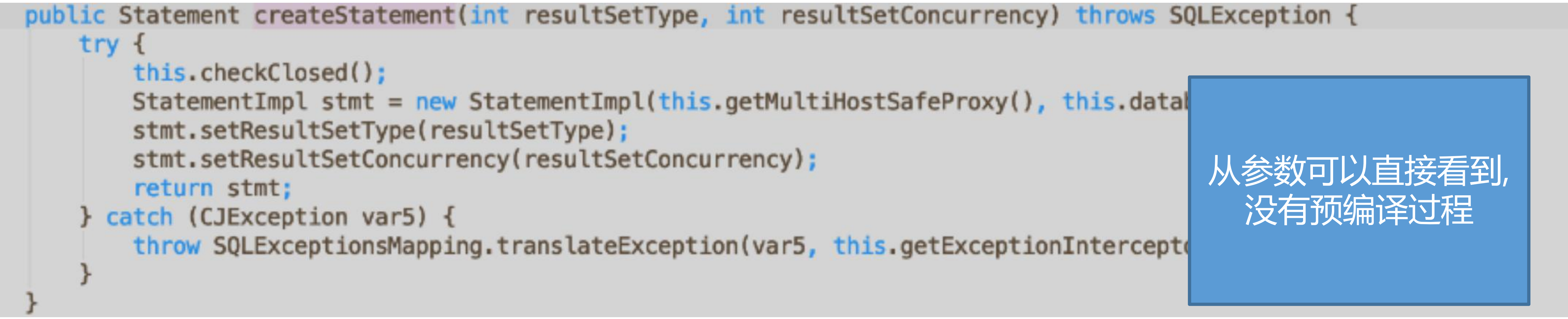
服务端初始化(预编译)异常,则走客户端预编译的过程

对比Statements



```
public Statement createStatement(int resultSetType, int resultSetConcurrency) throws SQLException {
    try {
        this.checkClosed();
        StatementImpl stmt = new StatementImpl(this.getMultiHostSafeProxy(), this.database);
        stmt.setResultSetType(resultSetType);
        stmt.setResultSetConcurrency(resultSetConcurrency);
        return stmt;
    } catch (CJException var5) {
        throw SQLExceptionsMapping.translateException(var5, this.getExceptionInterceptor());
    }
}
```

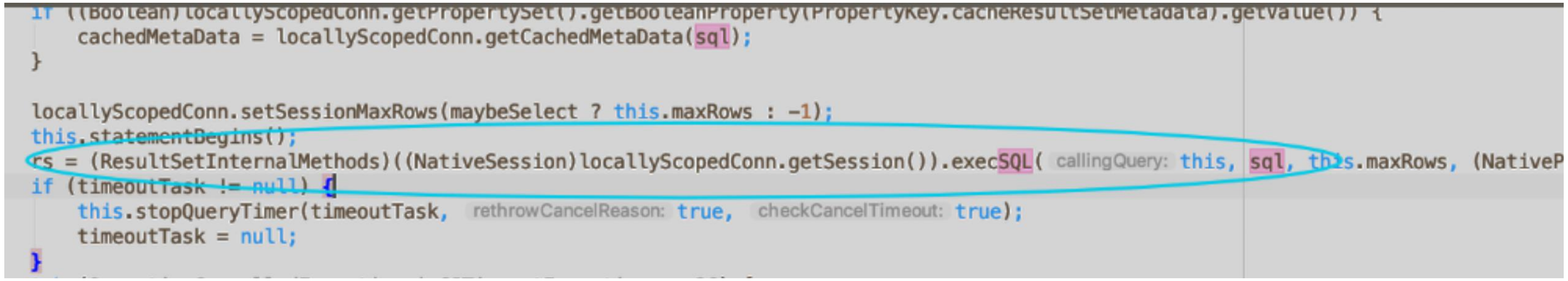
对比Statements



```
public Statement createStatement(int resultSetType, int resultSetConcurrency) throws SQLException {
    try {
        this.checkClosed();
        StatementImpl stmt = new StatementImpl(this.getMultiHostSafeProxy(), this.datal
        stmt.setResultSetType(resultSetType);
        stmt.setResultSetConcurrency(resultSetConcurrency);
        return stmt;
    } catch (CJException var5) {
        throw SQLExceptionsMapping.translateException(var5, this.getExceptionIntercept
    }
}
```

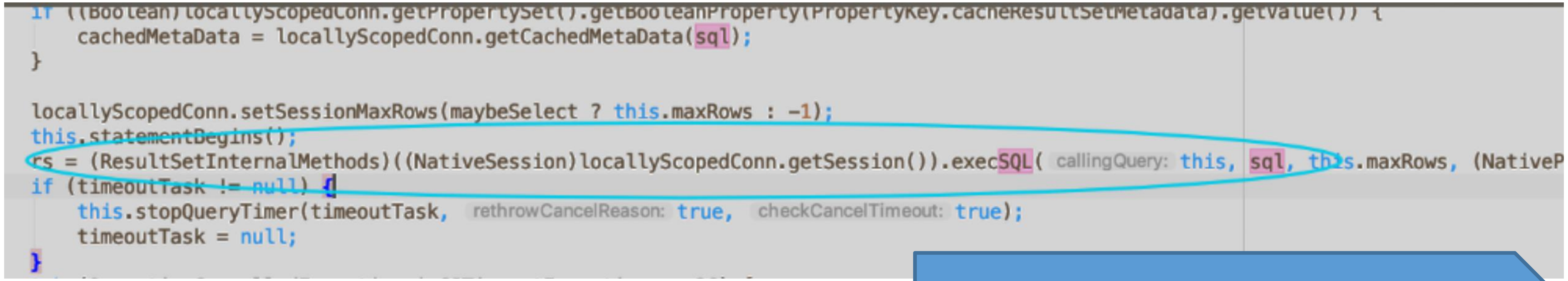
从参数可以直接看到,
没有预编译过程

再看一下执行sql语句的过程



```
if ((Boolean)locallyScopedConn.getPropertySet().getBooleanProperty(PropertyKey.cacheResultSetMetadata).getValue()) {
    cachedMetaData = locallyScopedConn.getCachedMetaData(sql);
}

locallyScopedConn.setSessionMaxRows(maybeSelect ? this.maxRows : -1);
this.statementBegins();
rs = (ResultSetInternalMethods)((NativeSession)locallyScopedConn.getSession()).execSQL( callingQuery: this, sql, this.maxRows, (NativeP
if (timeoutTask != null) {
    this.stopQueryTimer(timeoutTask, rethrowCancelReason: true, checkCancelTimeout: true);
    timeoutTask = null;
}
```

再看一下执行sql语句的过程



```
if ((Boolean)locallyScopedConn.getPropertySet().getBooleanProperty(PropertyKey.cacheResultSetMetadata).getValue()) {
    cachedMetaData = locallyScopedConn.getCachedMetaData(sql);
}

locallyScopedConn.setSessionMaxRows(maybeSelect ? this.maxRows : -1);
this.statementBegins();
rs = (ResultSetInternalMethods)((NativeSession)locallyScopedConn.getSession()).execSQL( callingQuery: this, sql, this.maxRows, (NativeP
if (timeoutTask != null) {
    this.stopQueryTimer(timeoutTask, rethrowCancelReason: true, checkCancelTimeout: true);
    timeoutTask = null;
}
```

没有预编译过程,
执行一次,
编译一次

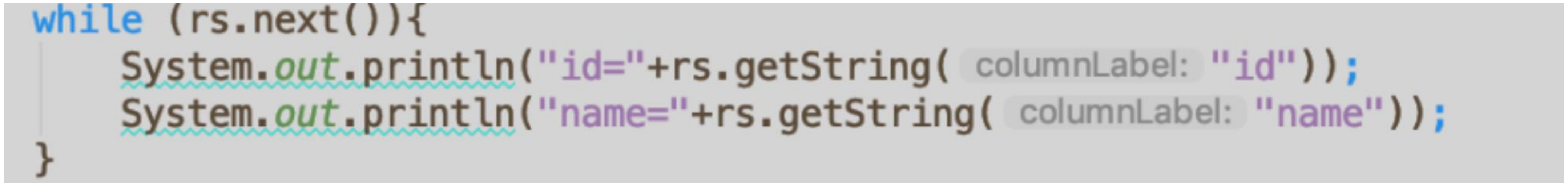
预编译的好处

Sql结构不变,只有参数变时,效率很高

如下面的语句:

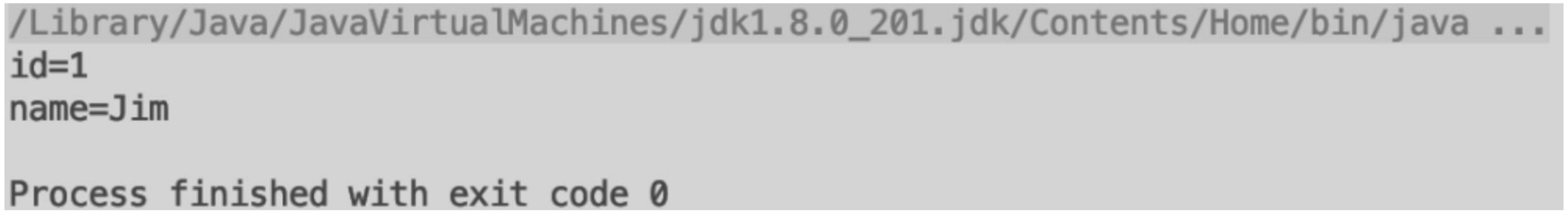
```
stmt.addBatch("INSERT INTO  TABLE1  VALUES("11","12","13","",""));
stmt.addBatch("INSERT INTO  TABLE1  VALUES("12","12","13","",""));
stmt.addBatch("INSERT INTO  TABLE1  VALUES("13","12","13","",""));
stmt.addBatch("INSERT INTO  TABLE1  VALUES("14","12","13","",""));
stmt.addBatch("INSERT INTO  TABLE1  VALUES("15","12","13","",""));
stmt.addBatch("INSERT INTO  TABLE1  VALUES("16","12","13","",""));
stmt.addBatch("INSERT INTO  TABLE1  VALUES("17","12","13","",""));
stmt.addBatch("INSERT INTO  TABLE1  VALUES("18","12","13","",""));
```

执行,获取结果



```
while (rs.next()){
    System.out.println("id="+rs.getString( columnLabel: "id"));
    System.out.println("name="+rs.getString( columnLabel: "name"));
}
```

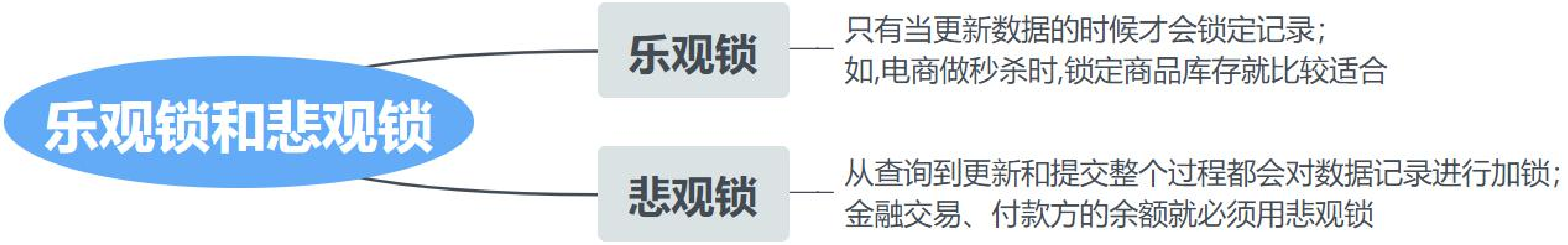
执行,获取结果



```
/Library/Java/JavaVirtualMachines/jdk1.8.0_201.jdk/Contents/Home/bin/java ...
id=1
name=Jim

Process finished with exit code 0
```

执行过程的几个知识点

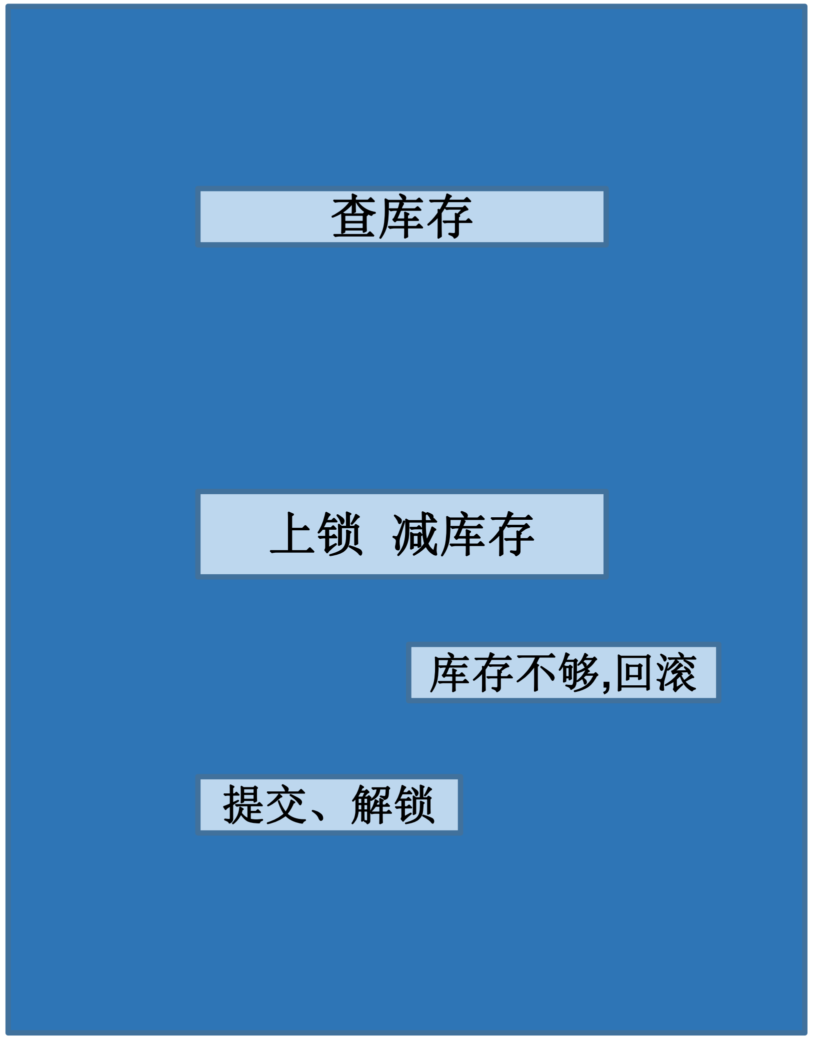
JDBC中存在哪些不同类型的锁?



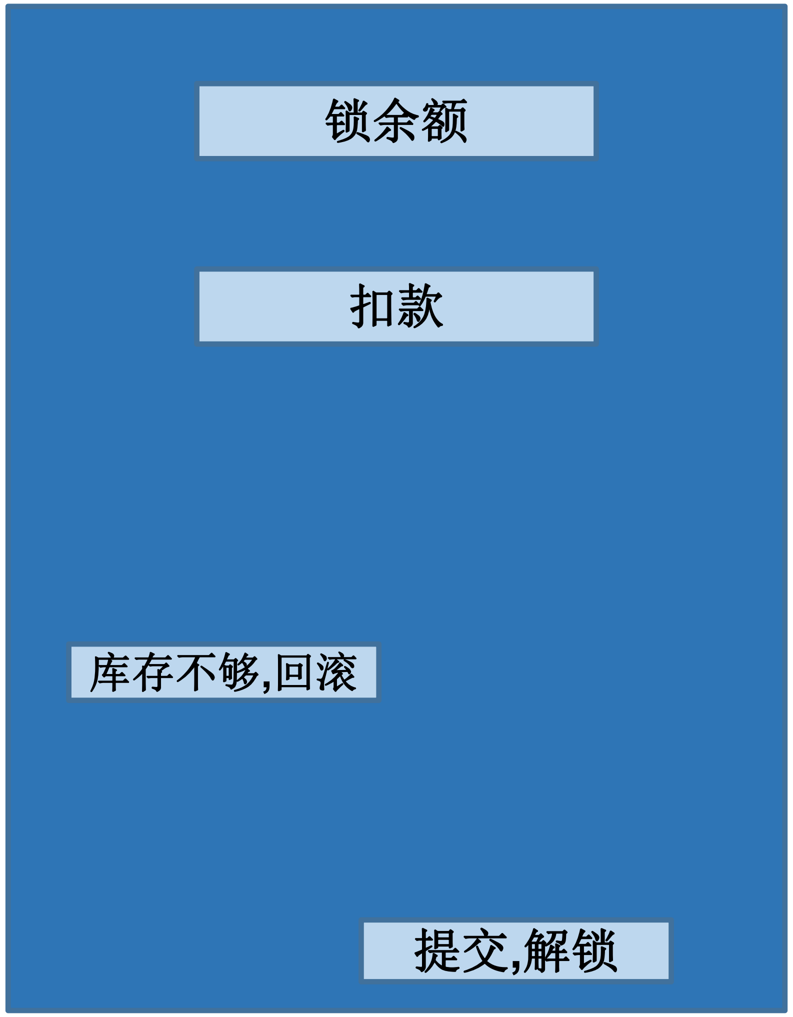
只有当更新数据的时候才会锁定记录;
如,电商做秒杀时,锁定商品库存就比较适合

从查询到更新和提交整个过程都会对数据记录进行加锁;
金融交易、付款方的余额就必须用悲观锁

乐观锁

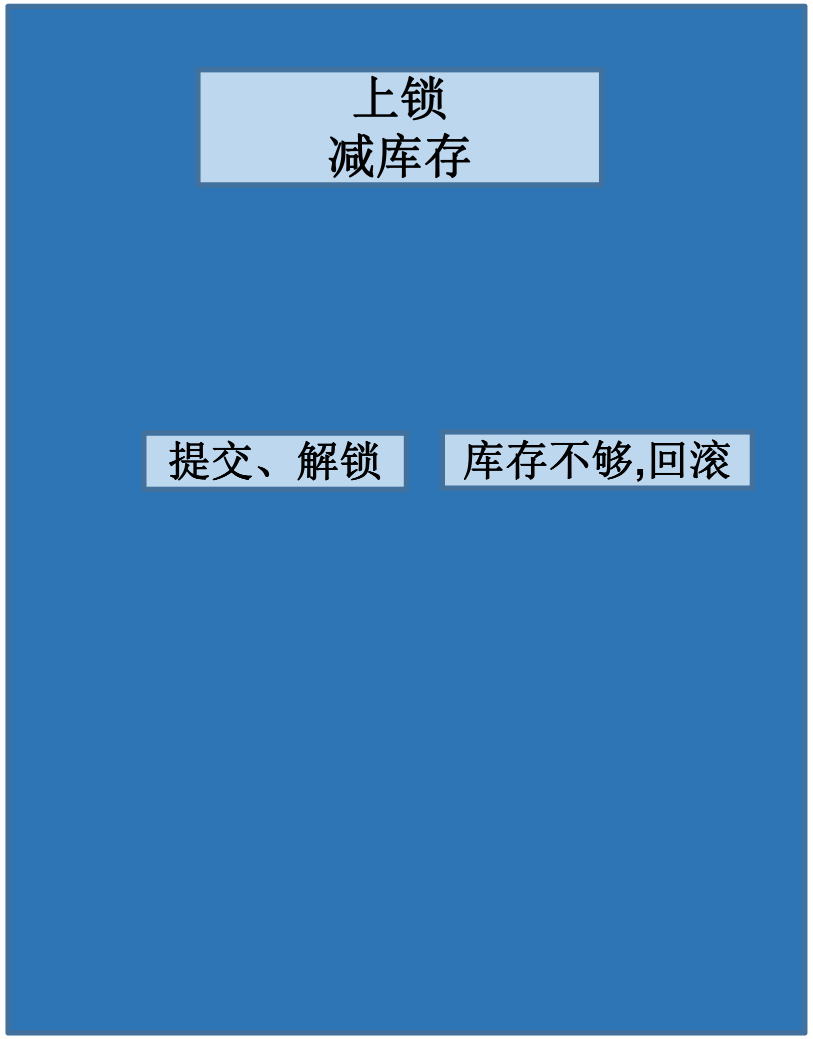


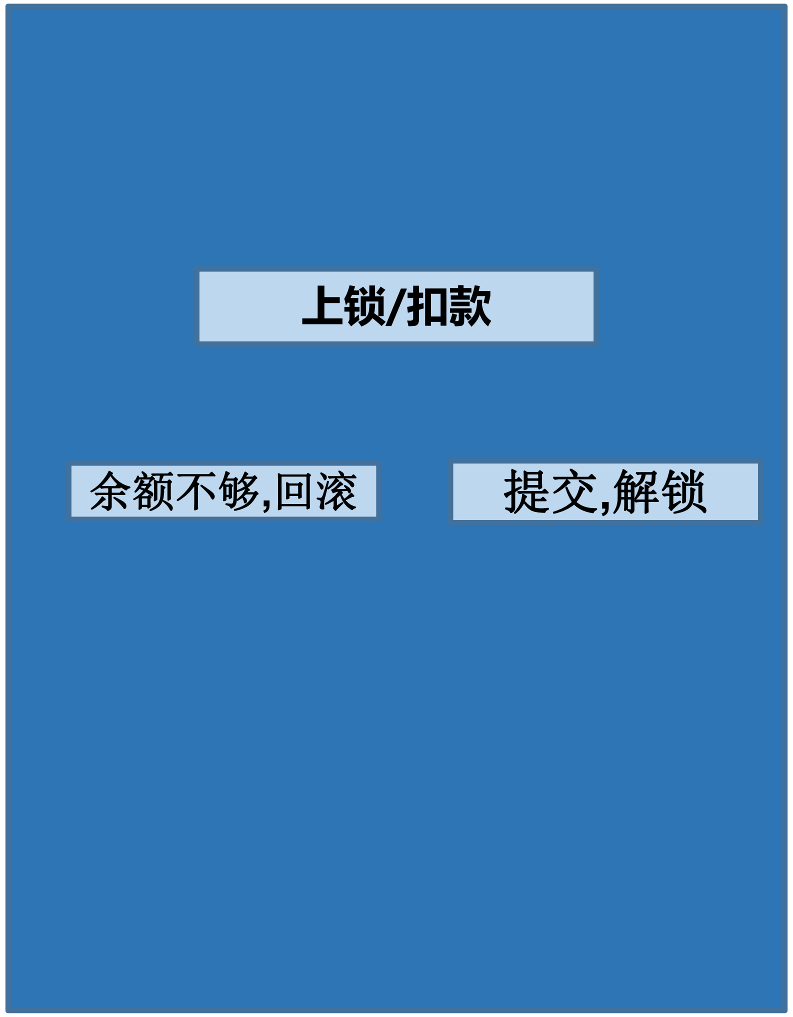
悲观锁

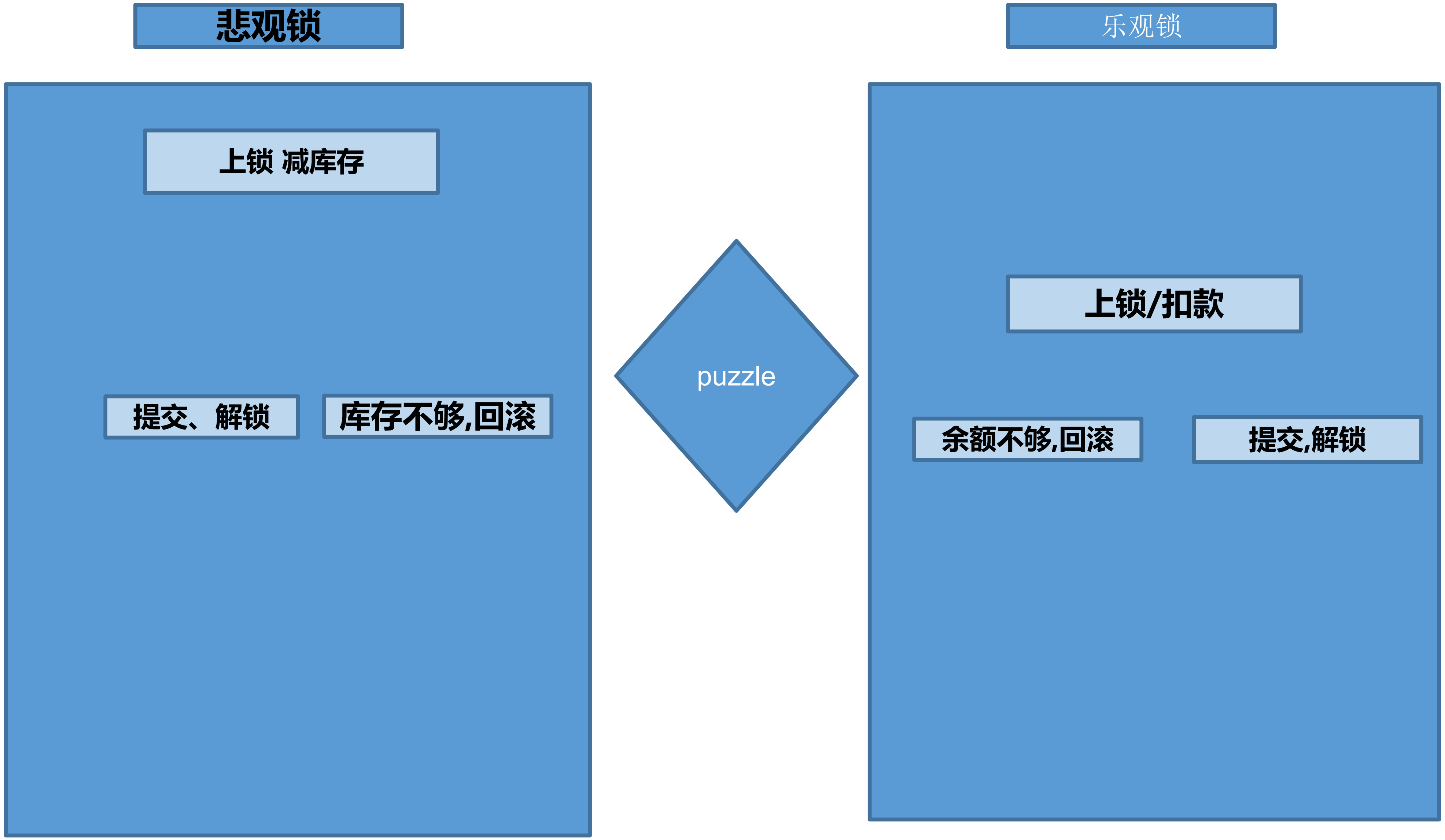


悲观锁

乐观锁






悲观锁
上锁 减库存
提交、解锁
库存不够,回滚
puzzle
乐观锁
上锁/扣款
余额不够,回滚
提交,解锁

悲观锁的实现

String sql="select job,ename,sal from emp where job=? for update";

乐观锁的实现

在数据表中添加version字段

乐观锁的实现

update order set price = 1, version = version + 1 where id = 1 and version = 0

乐观锁的实现

执行完成后，version字段值将变成1, 第二人执行update:

乐观锁的实现

update order set price = 1, version = version + 1 where id = 1 and version = 0

乐观锁的实现

此时的version的值已经被修改为1，所以第二人修改失败，实现乐观锁控制。

乐观锁存在的问题

死锁

死锁(事务A)

```
update order set price = 1 where id = 1
update order set price = 2 where id = 2
```

死锁(事务B)

update order set price = 1 where id = 2
update order set price = 2 where id = 1

死锁

事务A在执行完第一条update的时候，刚好事务B也执行完第一条update

死锁

此时， 事务A中order表中的id = 1的行被锁住， 事务B中order表中id = 2的行被锁住，两个事务继续往下执行

死锁

事务A中第二条update执行需要order表中id = 2的行数据，而事务B中第二条update执行需要id = 1的行数据， 两条update往下执行的条件都需要对方事务中已经被锁住的行，于是陷入无限等待，形成死锁。

解决死锁

指定锁的执行顺序

事务A

```
update order set price = 2 where id = 2
update order set price = 1 where id = 1
```

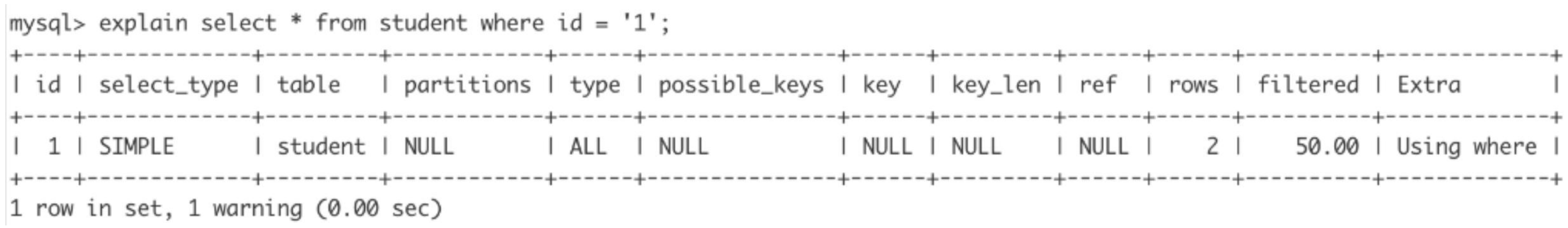
事务B

```
update order set price = 1 where id = 2
update order set price = 2 where id = 1
```

总结

并发的事务锁记录时都按一样的顺序去上锁

数据库调优篇



```
mysql> explain select * from student where id = '1';
+----+-------------+---------+------------+------+---------------+------+---------+------+------+----------+-------------+
| id | select_type | table   | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra       |
+----+-------------+---------+------------+------+---------------+------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | student | NULL       | ALL  | NULL          | NULL | NULL    | NULL |    2 |    50.00 | Using where |
+----+-------------+---------+------------+------+---------------+------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

type列

连接类型。一个好的SQL语句至少要达到range级别。杜绝出现all级别。

key列

使用到的索引名。如果没有选择索引，值是NULL。

key_len列

索引长度。

rows列

扫描行数。该值是个预估值。

extra列

详细说明。注意，常见的不太友好的值，如下：Using filesort，Using temporary。

Sql语句优化

SQL语句中IN包含的值不应过多

Sql语句优化

例如：select id from t where num in(1,2,3) 对于连续的数值，能用between就不要用in了

Sql语句优化

SELECT语句务必指明字段名称

Sql语句优化

SELECT*增加很多不必要的消耗（CPU、IO、内存、网络带宽）；增加了使用覆盖索引的可能性；当表结构发生改变时，应用也需要更新。所以要求直接在select后面接上字段名。

Sql语句优化

当只需要一条数据的时候，使用limit 1

Sql语句优化

如果排序字段没有用到索引，就尽量少排序

Sql语句优化

如果限制条件中其他字段没有索引，尽量少用or

Sql语句优化

尽量用union all代替union

Sql语句优化

区分in和exists、not in和not exists

区分in和exists主要是造成了驱动顺序的改变（这是性能变化的关键），如果是exists，那么以外层表为驱动表，先被访问，如果是IN，那么先执行子查询。所以IN适合于外表大而内表小的情况；EXISTS适合于外表小而内表大的情况。

关于not in和not exists，推荐使用not exists，不仅仅是效率问题，
not in可能存在逻辑问题。

原sql语句

select colname ... from A
表 where a.id not in (select b.id from B表)

高效的sql语句

select colname … from A表 Left join B表 on a.id = b.id where b.id is null

使用合理的分页方式以提高分页的效率

```
select id,name from product limit 866613, 20
```

随着表数据量的增加，直接使用limit分页查询会越来越慢。

以取前一页的最大行数的id，然后根据这个最大的id来限制下一页的起点。

优化的sql语句

上一页最大的id是866612则:

```
select id,name from product where id> 866612 limit 20
```

避免在where子句中对字段进行null值判断

对于null的判断会导致引擎放弃使用索引而进行全表扫描。

不建议使用%前缀模糊查询

LIKE “%name”或者LIKE “%name%”，这种查询会导致索引失效而进行全表扫描。但是可以使用LIKE “name%”。

使用全文索引解决

```
ALTER TABLE `dynamic_201606` ADD FULLTEXT INDEX `idx_user_name` (`user_name`);
```

使用全文索引解决

```
select id,fnum,fdst from dynamic_201606 where match(user_name) against('zhangsan' in boolean mode);
```

避免在where子句中对字段进行表达式操作

```
select user_id,user_project from user_base where age*2=36;
```

优化sql

select user_id,user_project from user_base where age=36/2;

总结

今天两个专题

在不用框架的情况下纯编码接入mysql

数据库调优:sql语句的优化

作业

1、从建库、建表、到纯手工代码实践一个数据库连接的项目

2、用sql优化专题中的正例,反例,在作业一实现的项目中采用打印耗时日志的方法分析你的调优效果