

5.3 Java 数据库基础

- JDBC
- 参数化查询
- 数据库事务
- DAO 与 DTO

目录

1

JDBC

2

参数化查询

3

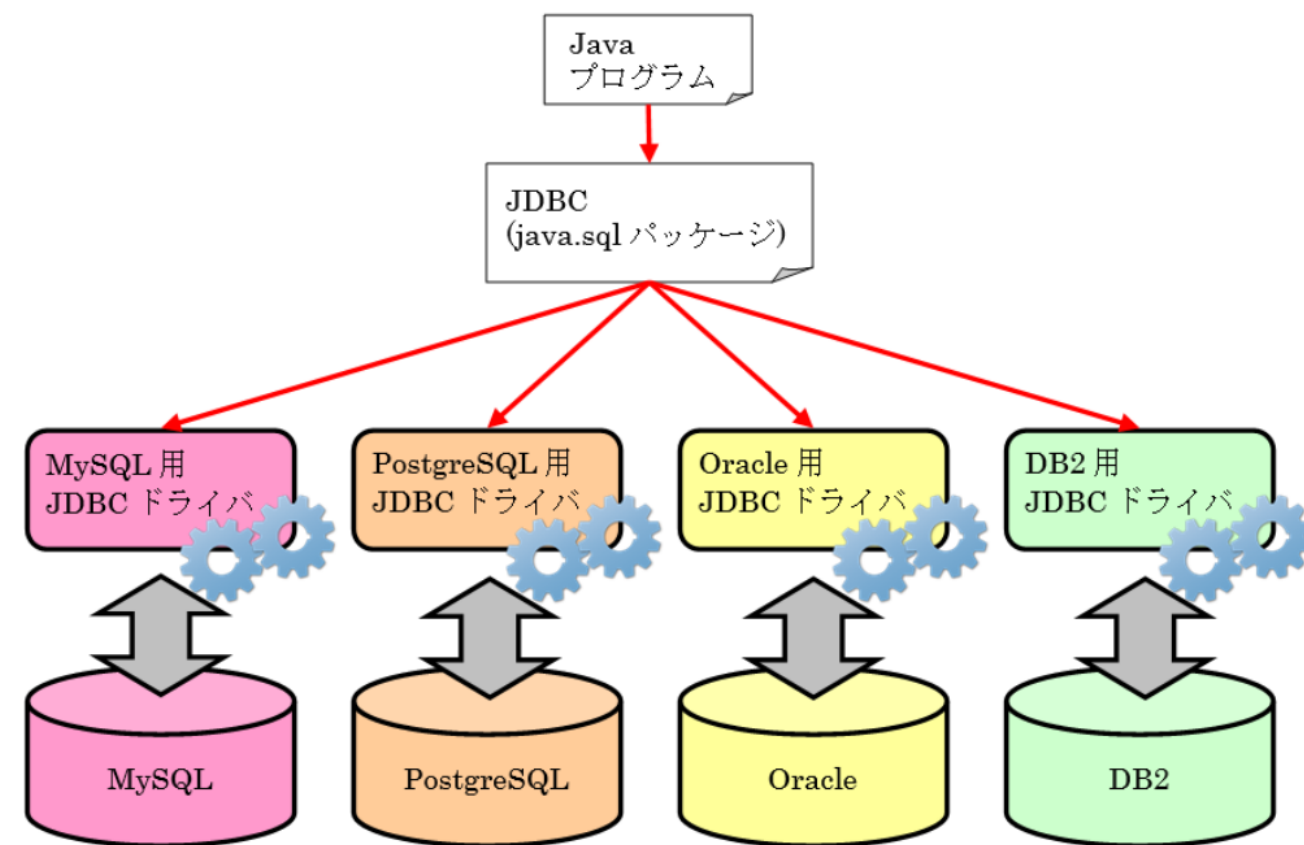
数据库事务

4

DAO 与 DTO

JDBC 概述

- **JDBC** (Java Database Connectivity) 是 Java 为了访问不同的数据库系统提供的统一接口。各个数据库厂商将为我们提供该接口的具体实现（类）。包含这些实现的 Jar 包被称为“**数据库驱动** [ドライバ]”。
- 只要安装了对应驱动，无论连接哪一个 DBMS 的数据库，我们都能书写同样的 Java 代码处理数据。



JDBC 的使用步骤

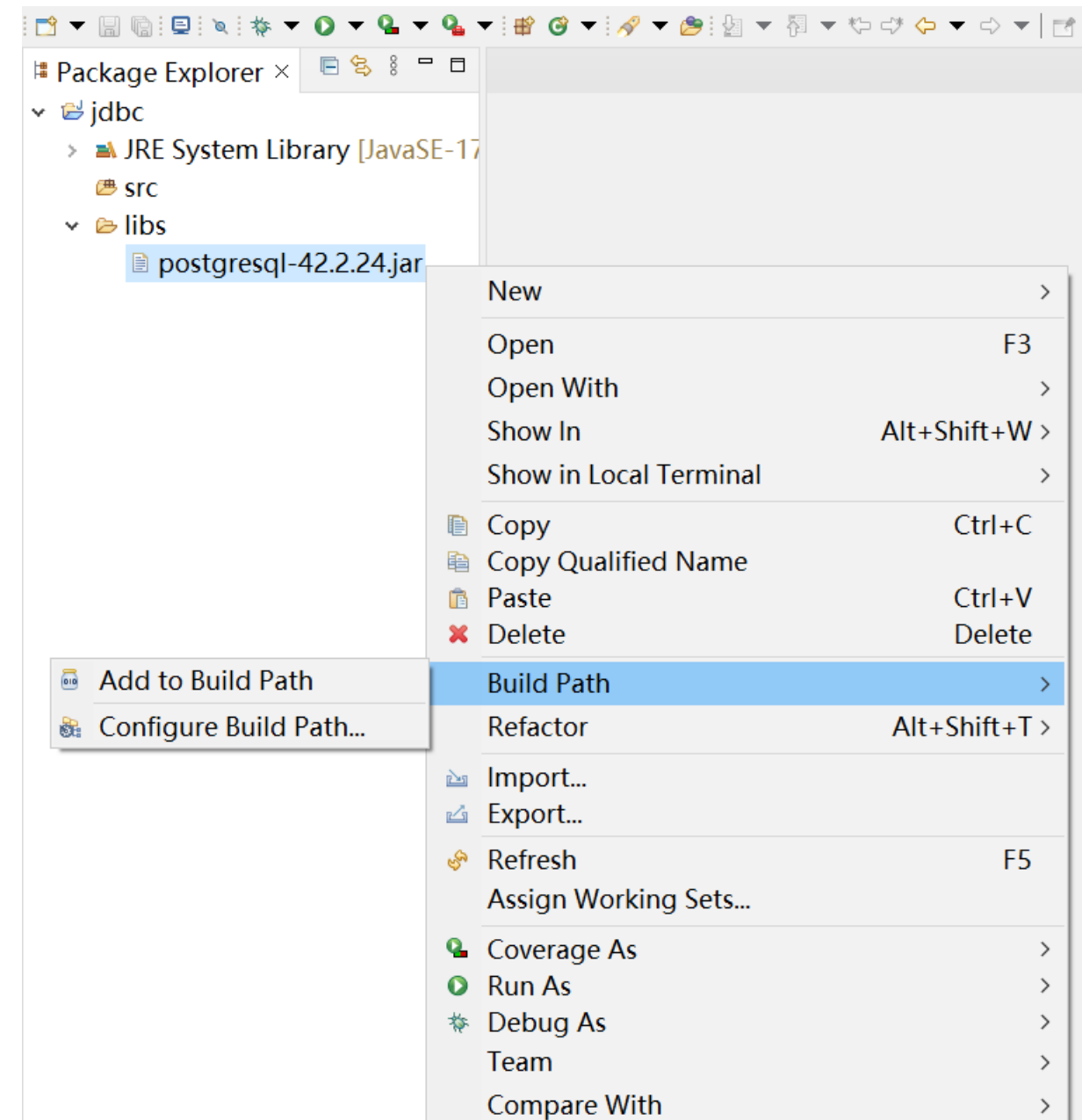
- JDBC 的使用可以分为 4 步：
 - 注册驱动：加载 Driver 类。
 - 获取链接：得到数据库链接。
 - 执行数据库操作：发送 SQL 语句至数据库。
 - 释放资源：关闭到数据库的链接。

Try 

JDBCExample.java

注册驱动

- 第一步：我们需要导入驱动对应的 Jar 包，以便代码中加载对应的 Driver 类：
 1. 复制 postgresql-42.2.24.jar 到项目的 libs 文件夹（如果没有，就创建一个）中。
 2. 右键点击 Build Path → Add to Build Path 将驱动添加到项目当中。



连接数据库

- 第二步：我们需要让 Java 连接上数据库，获得一个 **Connection**（链接）对象：
 1. 准备好数据库链接需要的信息，包括数据库的 **URL**、用户名、密码等。
 - 数据库的 URL 为："jdbc:postgresql://localhost:端口号/数据库名"。其中，端口号是注册时设定的数字（默认为 5432）。比如，之前的 hello 数据库的 URL 应该是：

```
String url = "jdbc:postgresql://localhost:5432/hello";
```
 - 用户名、密码都是注册时设定的内容（默认为 postgres 和 123456）。

接次页 ➤



2. 创建 **Driver** 对象，使用其 **connect** 方法创建链接：

```
1 // 1. 注册驱动, 创建 Driver 对象
2 Driver driver = new Driver();
3
4 // 2. 取得数据库链接 (第一种方法: 直接在代码里输入配置)
5 String url = "jdbc:postgresql://localhost:5432/hello"; // 设定数据库地址
6 Properties info = new Properties();
7 info.setProperty("user", "postgres"); // 设定用户名
8 info.setProperty("password", "123456"); // 设定密码
9
10 Connection con = driver.connect(url, info); // 建立链接
```

配置信息的书写方法

- 当我们通过 JDBC 连接数据库时，常常需要在代码中直接用字符串书写配置内容：

```
String url = "jdbc:postgresql://localhost:5432/hello";  
  
info.setProperty("password", "123456");
```

- 将类似这样的配置信息直接写在代码中的做法有两个问题：
 - 重要信息暴露在代码中，信息安全存在隐患。
 - 当我们想要修改一些配置时，必须修改源代码并编译，费时费力。
- 对此的解决方法，便是把这些配置信息写在外部文件中保存。

Properties 的使用

- Java 提供了 **Properties** 类，通过把配置信息写在一种特殊的 **.properties** 文件中，你可以简单地实现配置信息的获取、保存或修改。
- .properties 文件中，每行都代表 1 个配置，使用“配置名=配置值”的形式书写：

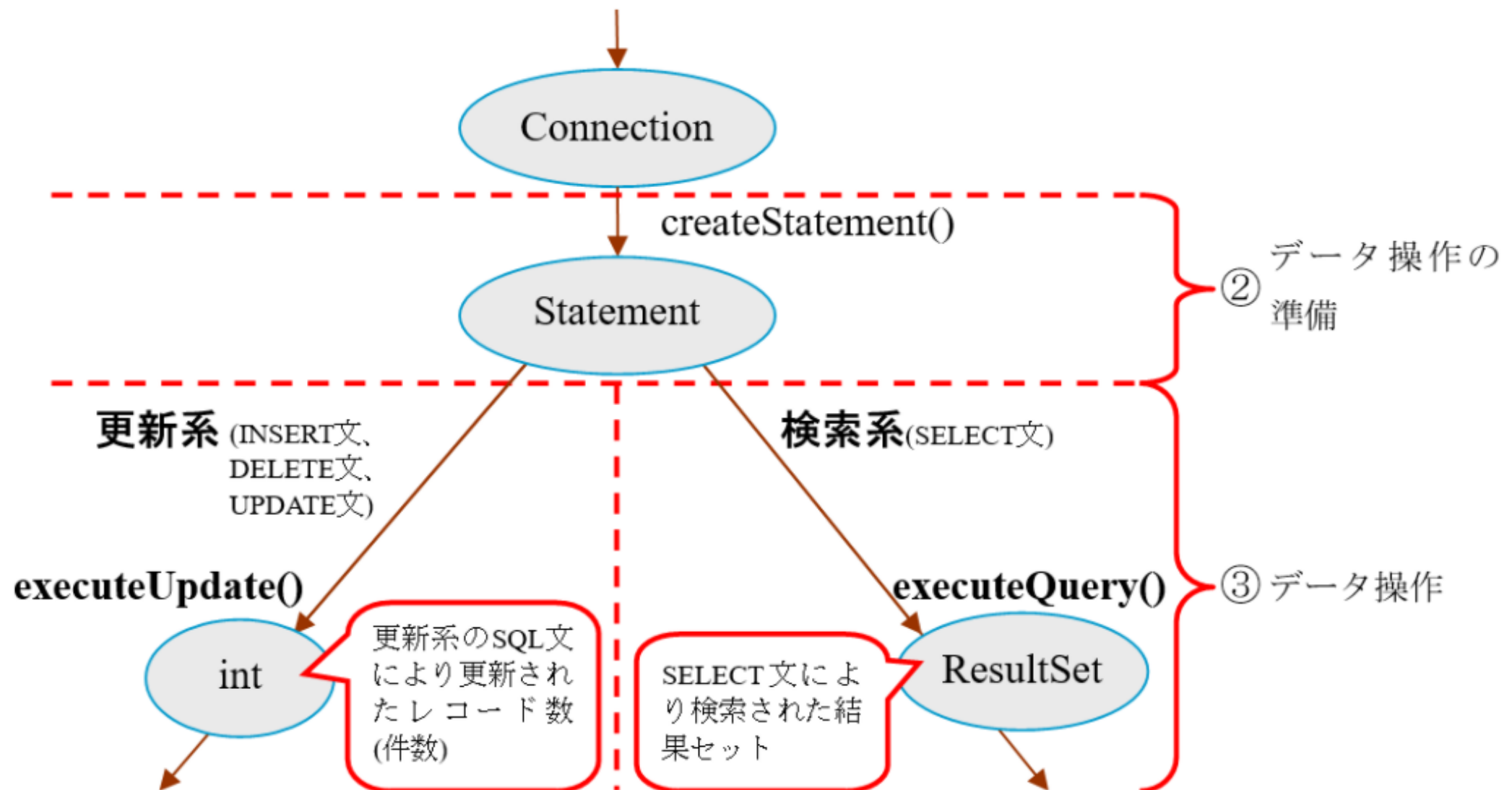
```
1 url=jdbc:postgresql://localhost:5432/hello
2 user=postgres
3 password=123456
```

- 这样的配置信息，我们可以直接通过 Properties 类提供的方法读取和使用：

```
properties.load(new FileReader("hello.properties"));
```

执行数据库操作

- 第三步：在连接上数据库后，我们就可以利用 Connection 类的方法执行增删改查操作。



执行 SQL 语句

- 要执行 SQL 语句，首先需要使用 Connection 类的 **createStatement()** 方法创建一个 **Statement** 对象：

```
Statement smt = con.createStatement();
```

- 接下来，书写需要执行的 SQL 语句的字符串：

```
String sql = "SELECT * FROM student";
```

接次页 

- 最后，将这个字符串传入 Statement 对象的 **executeQuery()** 方法或 **executeUpdate()** 方法执行。其中：
 - **executeQuery()** 方法用于执行**查询**（SELECT）语句。它将返回一个保存了查询结果的 **ResultSet** 对象。
 - **executeUpdate()** 方法用于执行**更新**（插入、删除或修改）语句。它将返回一个整数，代表更新了多少行记录。
- 比如，刚刚的 SQL 语句是一个查询语句，因此要使用 **executeQuery()** 方法：

```
ResultSet rs = smt.executeQuery(sql);
```


查询数据

- **executeQuery()** 将会返回一个 **ResultSet** 对象。ResultSet 有很多用于查询的功能，我们目前只要记住两个：
 - **getXxx("column")**: 用于获得当前“查看”记录的 column 字段。Xxx 是该字段的数据类型，比如 Int、String 等。
 - **next()**: 开始“查看”下一行记录。
- 开始时，我们会“查看”“第 0 行”记录（即还未开始查看）。我们可以使用 **next()** 方法移动到下一行记录，再使用 **getXxx()** 方法从中取出想要的字段，然后重复此流程：

```
1 while (rs.next()) {  
2     System.out.println("[id: " + rs.getInt("id")  
3         + ", name: " + rs.getString("name")  
4         + ", score: " + rs.getInt("score") + "]);  
5 }
```

数据的更新

- 数据的插入 (INSERT)、删除 (DELETE) 和修改 (UPDATE) 统一使用 **executeUpdate()** 方法执行。
- 该方法返回一个整数值，代表总共有多少行记录被更新（比如 DELETE 命令会返回有多少行记录被删除）：

```
1 Statement smt = con.createStatement();  
2  
3 String sql = "DELETE FROM student WHERE name = 'Alice'";  
4 int result = smt.executeUpdate(sql);  
5  
6 System.out.println(result); // => 1
```


释放资源

- 第四步：和文件读写等操作一样，我们需要关闭创建的链接以释放系统资源。要关闭链接，使用 Statement 和 Connection 类的 **close()** 方法：

```
smt.close();  
con.close();
```

- 为了保证资源的释放，可以使用之前 (← § 2.5.3) 学习的 **try-with-resources** 语句（此法不需要使用 close() 方法）：

```
1 try (  
2     Connection con = driver.connect(url, info);  
3     Statement smt = con.createStatement();  
4 ) {  
5     // codes...  
6 } catch (SQLException e) {  
7     // Exception handling...  
8 }
```

Q & A

Question and answer

目录

1

JDBC

2

参数化查询

3

数据库事务

4

DAO 与 DTO

Statement

- 我们之前使用了一个 Statement 对象执行 SQL 语句，传入其方法的是整个语句的字符串。然而，直接把整个语句用字符串运算创建出来有以下弊端：
 1. 需要使用多个加号连接字符串，SQL 代码和 Java 语句混在一起，可读性低，容易出错。
 2. 直接把参数写入字符串，会导致 **SQL 注入风险**。恶意用户可以传入特殊的参数破坏数据库或泄露其数据。
 3. 无法预处理 SQL 语句，影响执行效率。

SQL 注入

- 在实际对数据库进行操作时，很多时候我们会使用用户提供的数据作为参数。比如，下面这个语句使用用户提供的用户名查询数据库中的密码：

```
SELECT password FROM account WHERE username = '用户输入的参数';
```

- 此时，如果恶意用户了解 SQL 语言并猜出了服务器创建语句的方法，就可以通过书写特殊的参数改变原语句的意思：

```
1 SELECT password FROM account  
2 WHERE username = ' '; UPDATE account SET password = 'HACKED' -- ';
```

- 这就是所谓的 **SQL 注入** [SQL インジェクション]。



PreparedStatement

- 要回避这些问题，可以改用 **PreparedStatement** 类来执行的 SQL 语句。
- 要创建一个 PreparedStatement 对象，使用 Connection 对象的 **prepareStatement()** 方法，并传入 SQL 语句的“模板”：

```
1 // 准备 SQL 语句的“模板”  
2 String sql = "SELECT password FROM account WHERE username = ?";  
3 // 创建 PreparedStatement 对象  
4 PreparedStatement smt = con.prepareStatement(sql);
```

UPDATE bookinfo SET price = ? WHERE isbn = ?

先頭から順に「?」
に 1 と 2 の番号が
割り当てられる。

PreparedStatement 的使用

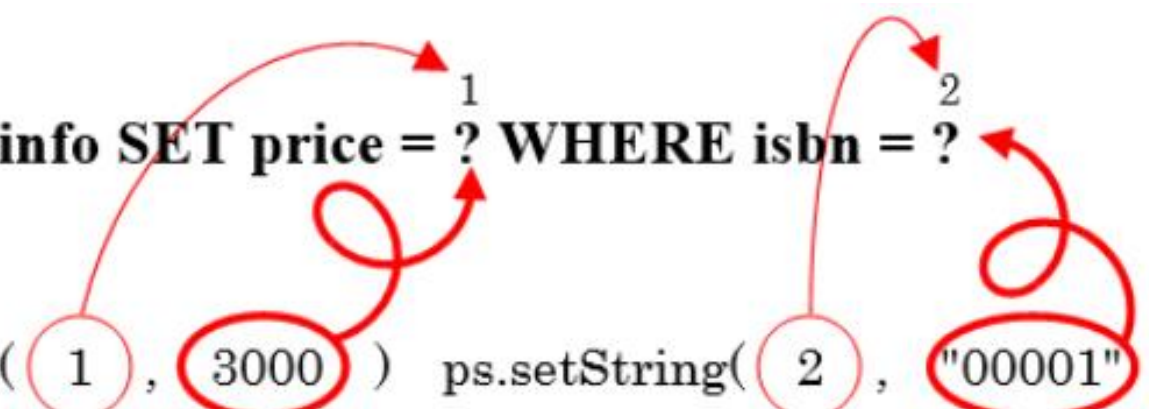
- 在（如通过接受用户输入）获得实际使用的数据后，调用 PreparedStatement 对象的 **setXxx()** 方法来设置这些参数。其中，Xxx 的部分是对应变量的类型，如 Int、String 等。
- setXxx() 方法接受 2 个参数，其中第 1 个指定设定哪一个 SQL 语句中的参数，第 2 个指定参数值：

```
smt.setString(1, "Alice");
```

Note  参数序号从 1 开始。

UPDATE bookinfo SET price = ? WHERE isbn = ?

ps.setInt(1, 3000) ps.setString(2, "00001")



接次页 

 接前页

- 在设定好参数之后，我们就可以直接使用它的 `executeQuery()` 或 `executeUpdate()` 方法：

```
ResultSet rs = smt.executeQuery();
```

Try 

PreparedExample.java

- `PreparedStatement` 会在设置 SQL 中的参数前检查数据的类型，从而避免 SQL 注入问题。

Q & A

Question and answer

目录

1

JDBC

2

参数化查询

3

数据库事务

4

DAO 与 DTO

数据库事务

- **事务** [トランザクション] 是指一系列连续、不可分割的 SQL 操作的集合。

Example ✓

我们在操作交易数据时，需要用多个语句控制账户上的金钱。比如可以按照以下顺序：

1. 使购买方账户上的金钱减少。
2. 使销售方账户增加等量金钱。

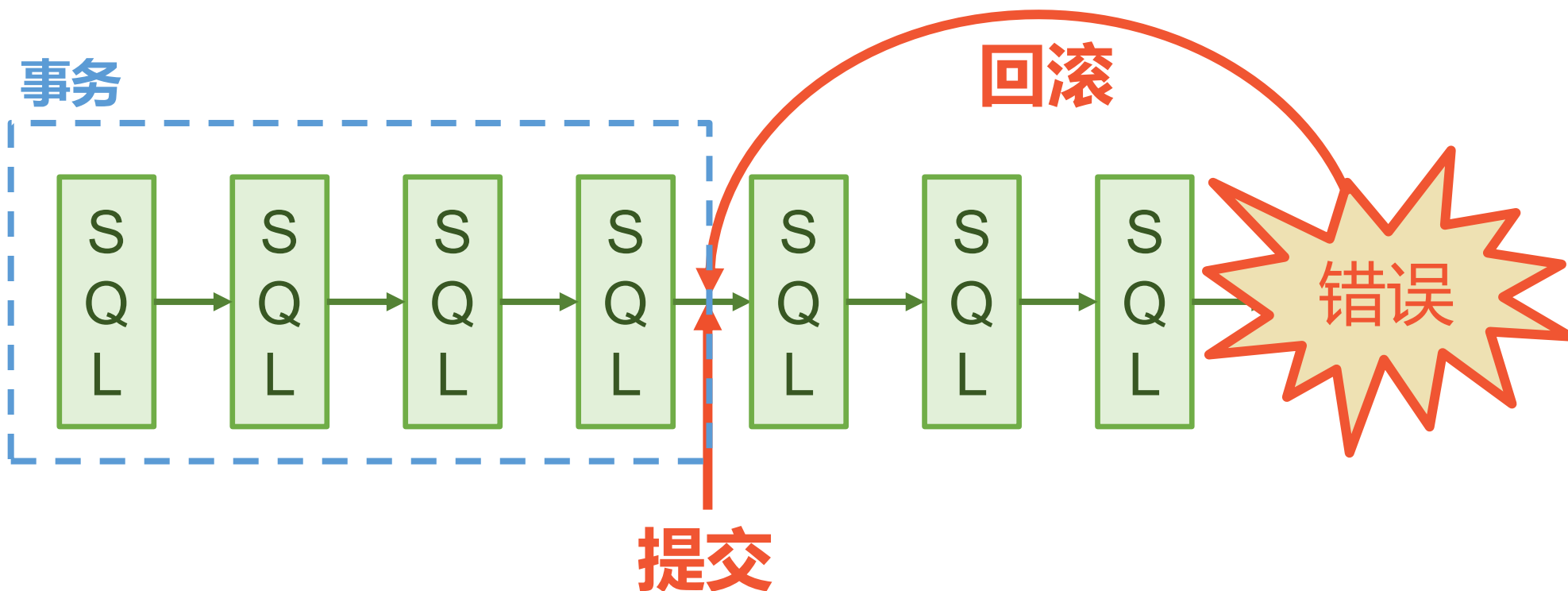
这些操作应该是**不可分割**的。否则，如果操作执行到一半时失败了，可能导致买方损失金钱（如果不撤销该操作）；或卖方获得额外金钱（如果撤销了该操作）。

事务遵循的原则：ACID

- 把一系列语句整理成事务后，它应该遵循以下 4 个原则：
 1. **原子性**[不可分性] (Atomicity)：所有操作不可分割，要么全部完成，要么全部不完成。
 2. **一致性**[一贯性·整合性] (Consistency)：事务中的每一步操作都必须满足数据库设定的条件（比如约束）。
 3. **事务隔离**[独立性] (Isolation)：其它操作只能访问到整个事务开始前的状态或结束后的状态，不能“插队”。
 4. **持久性**[永续性] (Durability)：事务处理结束后，对数据的修改将保存在数据库中，结果不会丢失。（比如存到 ROM 中。）
- 这些原则被统称为 **ACID** 原则。

事务的提交和回滚

- 一般事务的操作流程如下：我们先将一个事务中的所有 SQL 语句依次执行，然后**提交**[コミット]它。换句话说，每两次提交操作之间的语句就构成一次事务。
- 如果事务的执行中出现了某些问题，就需要**回滚**[ロールバック]该事务。回滚将撤销所有**未提交的 SQL** 语句，也就是上次提交之后的操作。



JDBC 中的事务

- 默认情况下，JDBC 中的每一个操作都是一个单独的事务：每一个语句在执行后都会**自动提交**。
- 要让多个语句组成一个事务，我们要先调用 Connection 对象的 **setAutoCommit(false)** 方法，取消语句的自动提交功能。
- 之后，我们就可以不提交而执行一个事务中的所有 SQL 语句。在你需要提交这些语句的结果时，调用 **commit()** 方法。
- 在某个操作失败或出现异常时，可以调用 **rollback()** 方法回滚整个事务。

Try 
TransactionExample.java

Q & A

Question and answer

目录

1

JDBC

2

参数化查询

3

数据库事务

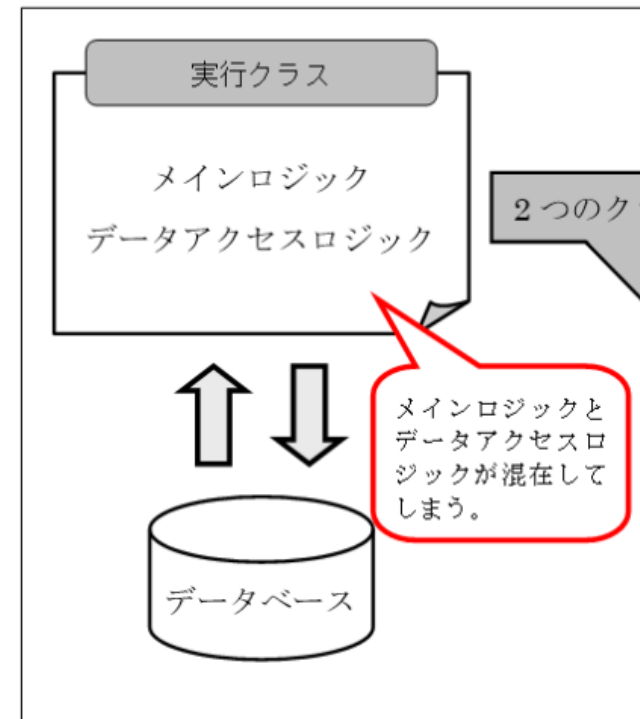
4

DAO 与 DTO

DAO 模式

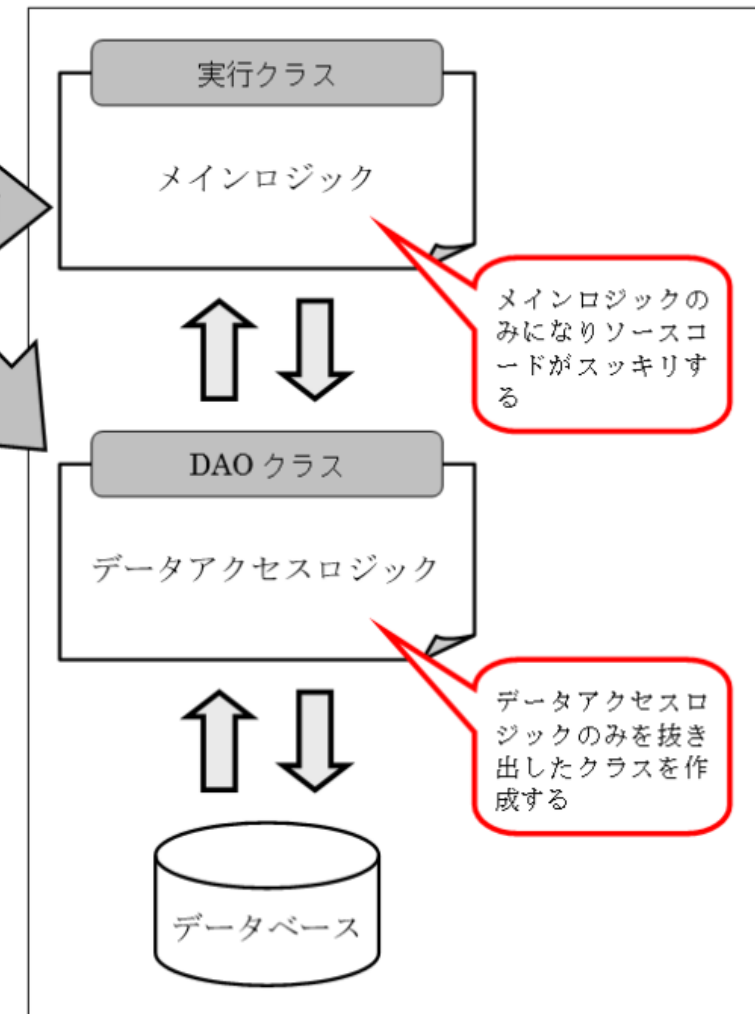
- 在创建使用数据库的应用程序时，经常使用称为 **DAO 模式** 和 **DTO 模式** 的两种程序设计模式。
- **DAO** (Data Access Object, 数据访问对象) 模式指创建一个**专门访问数据库的类**，将访问数据库的代码从程序主逻辑分离开来。可以把它理解成一个数据库访问“窗口”。

DAO を利用しない場合



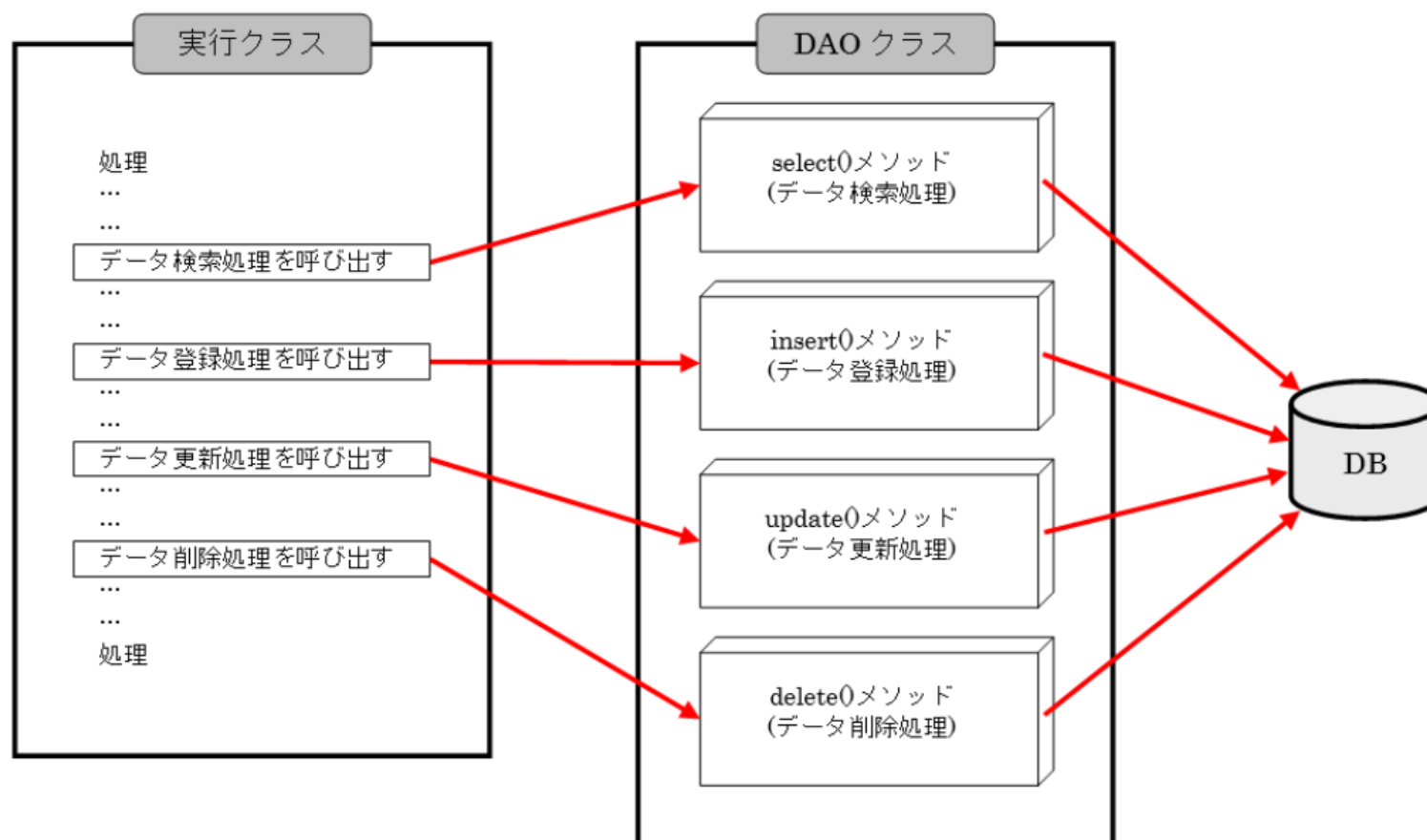
2つのクラスに分割

DAO を利用する場合



DAO 模式的优点

- 使用 DAO 模式的程序在设计时明确区分了主逻辑和数据访问过程。因此，它具有以下优点：
 1. 提高了**独立性和可扩展性**，修改某个类不会影响到其它类（解耦）。
 2. 类似的数据库访问功能被集成到一个类中，提高了**可读性和结构性**。
 3. 防止重复描述数据库访问处理，**简化源代码**。



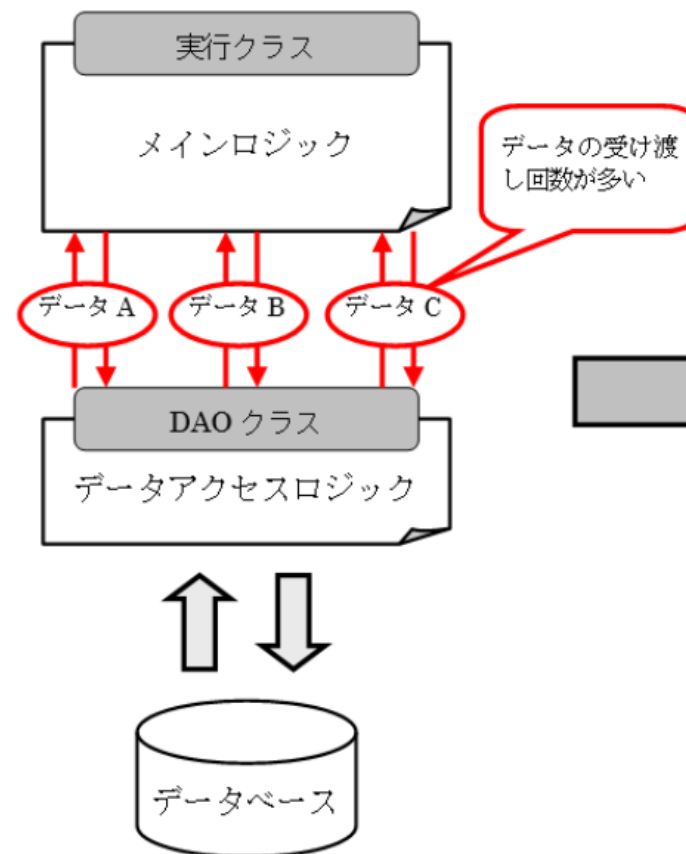
DTO 模式

- 如果只使用 DAO 模式，在主逻辑和数据访问中会出现一些低效的部分，比如说我们需要针对**每一个**数据库中的字段定义方法。
- **DTO** (Data Transfer Object, 数据传输对象) 模式是基于 *JavaBeans* 概念的“专用于数据传输的类”。
- 通常我们为一个完整的数据（比如用户、博客等）定义一个类，再为它的每个字段定义一个相应的变量，以及它们的 Getter 和 / 或 Setter。数据传输时，我们不再分别查询、传输原始数据，而是**将这个类作为整体**进行传输。

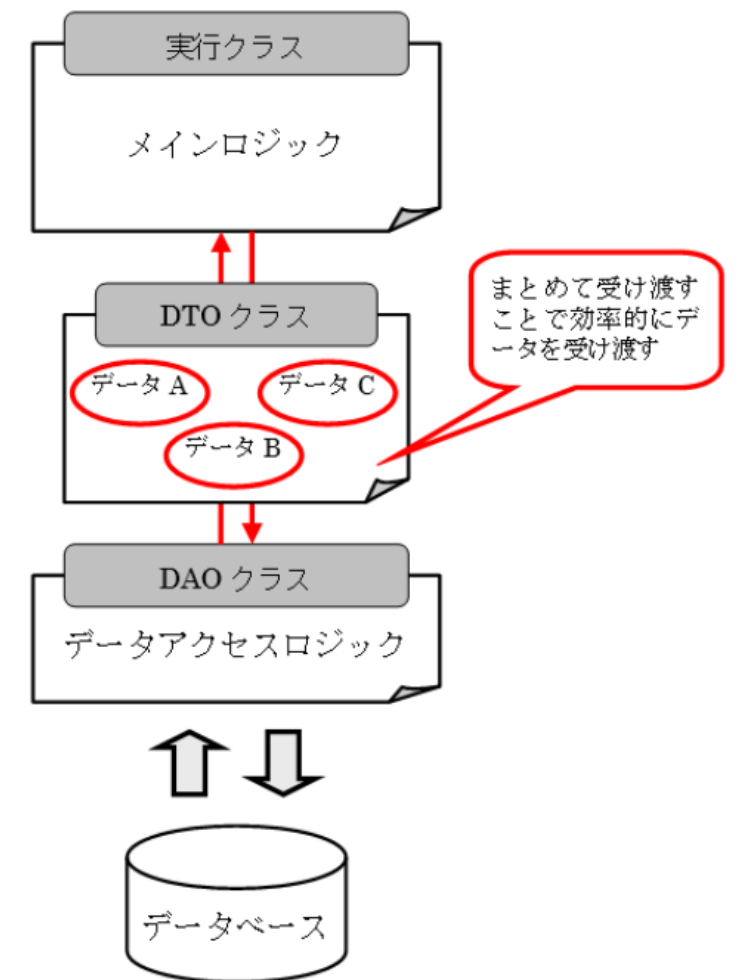
DTO 模式的优点

1. 使用 DTO 模式，数据的传输变更更容易，我们只需要传输一个个**对象**。
2. 我们可以通过把对象放入列表或数组等**数据结构**中来管理一系列数据。
3. 数据库的整体访问次数将会降低，提高了**运行效率**。

DTO を利用しない場合



DTO を利用する場合



Try 01011
11010
01011
student 包

Q & A

Question and answer

总结

Sum Up

1. Java 中访问数据库的方法：JDBC。
 - ① 驱动的安装和导入。
 - ② 创建链接（Connection）、Statement、执行 SQL 语句的完整流程。
2. SQL 注入的概念和它在 JDBC 中的解决方法：PreparedStatement 类。
3. 事务：ACID 原则、提交和回滚的概念和语法。
4. DAO 与 DTO 模式的基本概念。

THANK YOU!