

Java^{SE8}

技術手冊

- 涵蓋 OCP/JP (原 SCJP) 考試範圍
- Lambda 專案、新時間日期 API、等 Java SE 8 新功能詳細介紹
- JDK 基礎與 IDE 操作交相對照
- 提供實作檔案與操作錄影教學

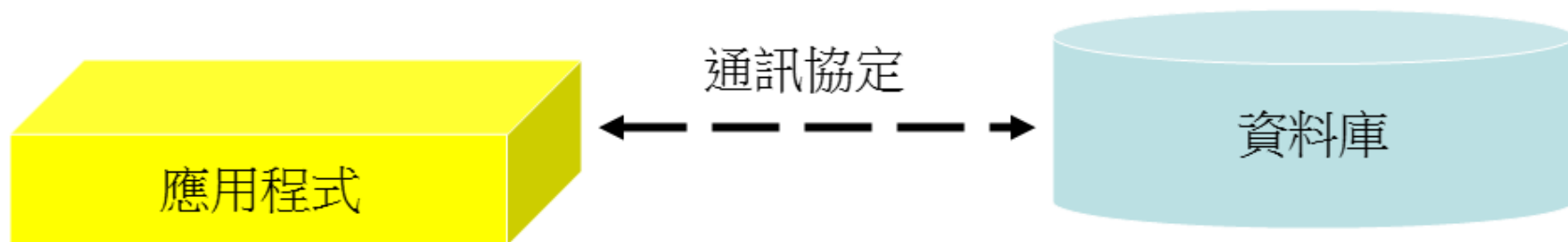
整合資料庫

學習目標

- 了解 JDBC 架構
- 使用 JDBC API
- 瞭解交易與隔離層級
- 認識 RowSet

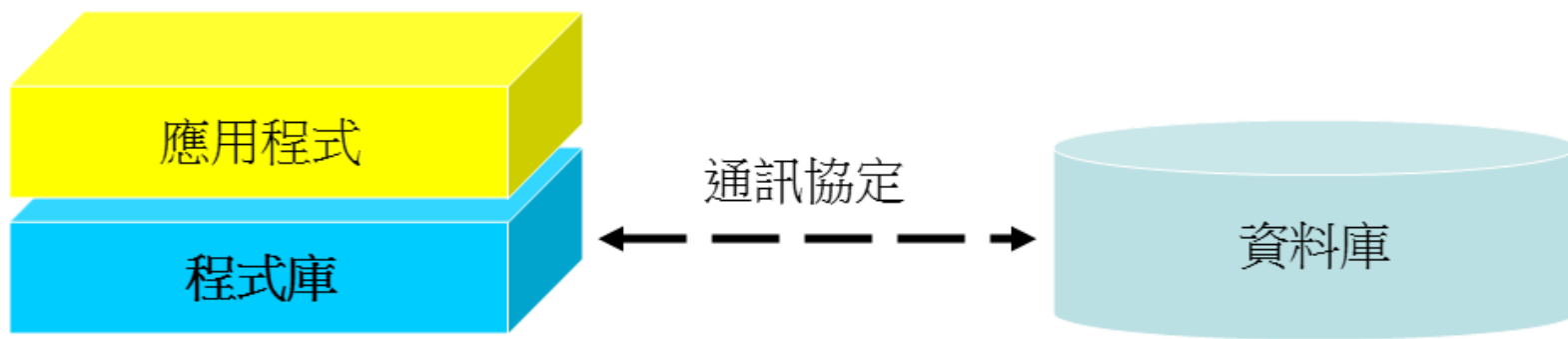
簡介 JDBC

- 資料庫本身是個獨立運行的應用程式
- 撰寫的應用程式是利用網路通訊協定與資料庫進行指令交換，以進行資料的增刪查找



簡介 JDBC

- 應用程式會利用一組專門與資料庫進行通訊協定的程式庫



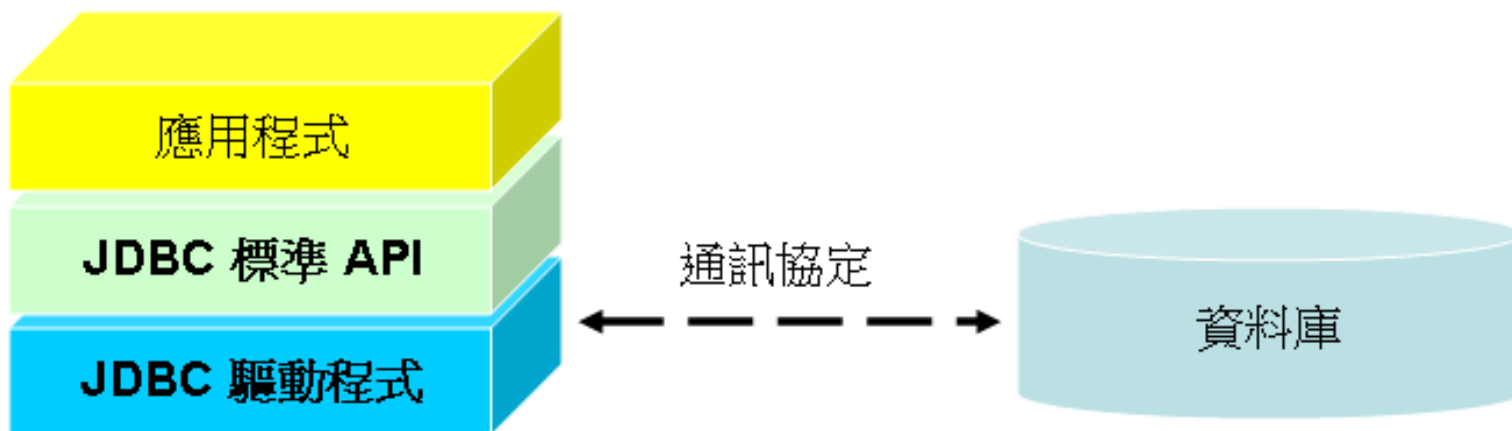
簡介 JDBC

- 應用程式如何呼叫這組程式庫？
- 不同的資料庫通常會有不同的通訊協定
- 用以連線不同資料庫的程式庫在 API 上也會有所不同

```
XySqlConnection conn = new XySqlConnection("localhost", "root", "1234");  
conn.selectDB("gossip");  
XySqlQuery query = conn.query("SELECT * FROM T_USER");
```

簡介 JDBC

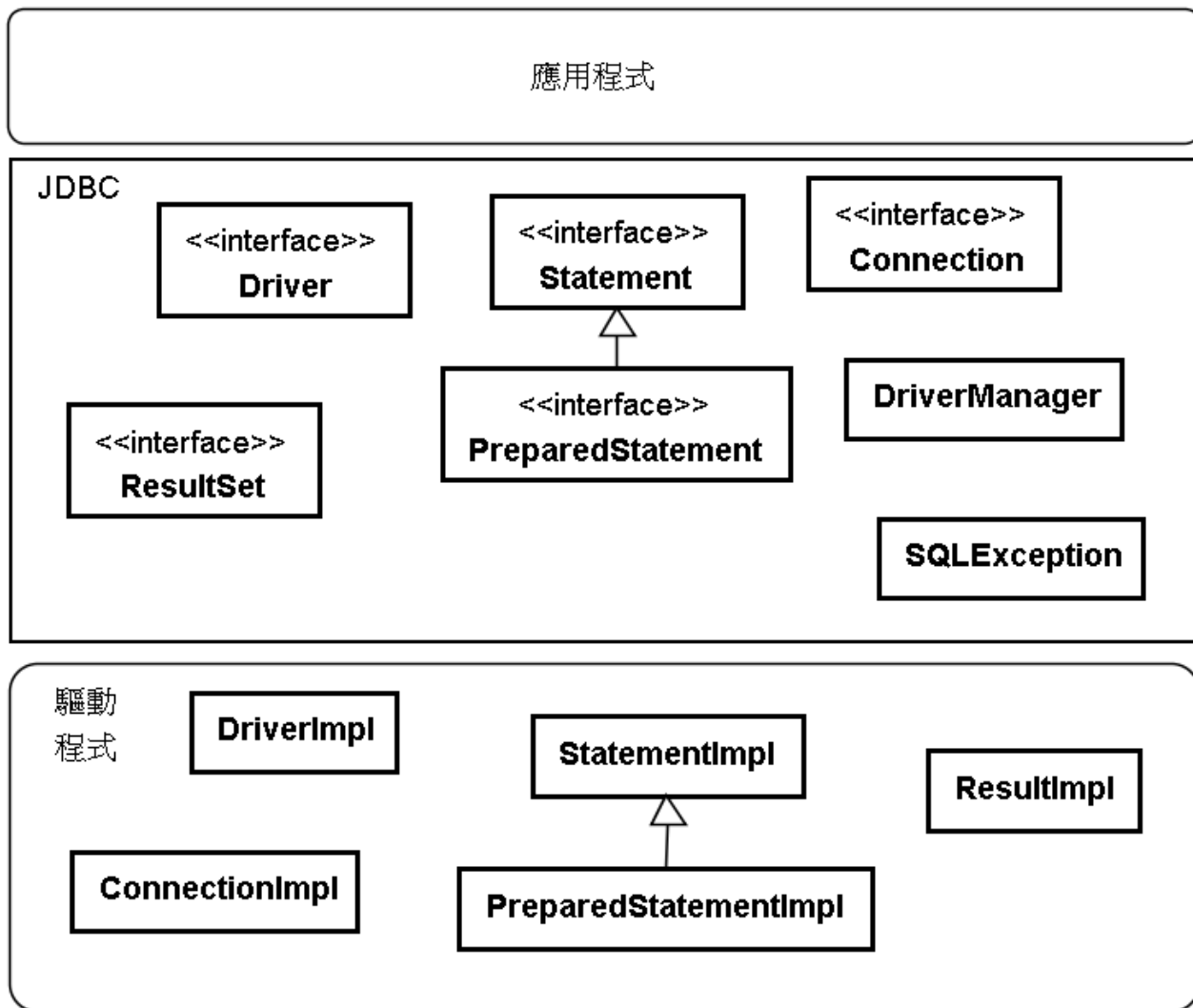
- JDBC 全名 Java DataBase Connectivity
- Java 連線資料庫的標準規範，定義一組標準類別與介面：
 - 應用程式需要連線資料庫時就呼叫這組標準 API，而標準 API 中的介面會由資料庫廠商實作，通常稱之為 JDBC 驅動程式（Driver）



簡介 JDBC

- JDBC 應用程式開發者介面（Application Developer Interface）
 - 應用程式需要連線資料庫
 - 相關 API 主要是座落於 `java.sql` 與 `javax.sql`
- JDBC 驅動程式開發者介面（Driver Developer Interface）
 - 資料庫廠商要實作驅動程式時的規範，一般開發者並不用瞭解

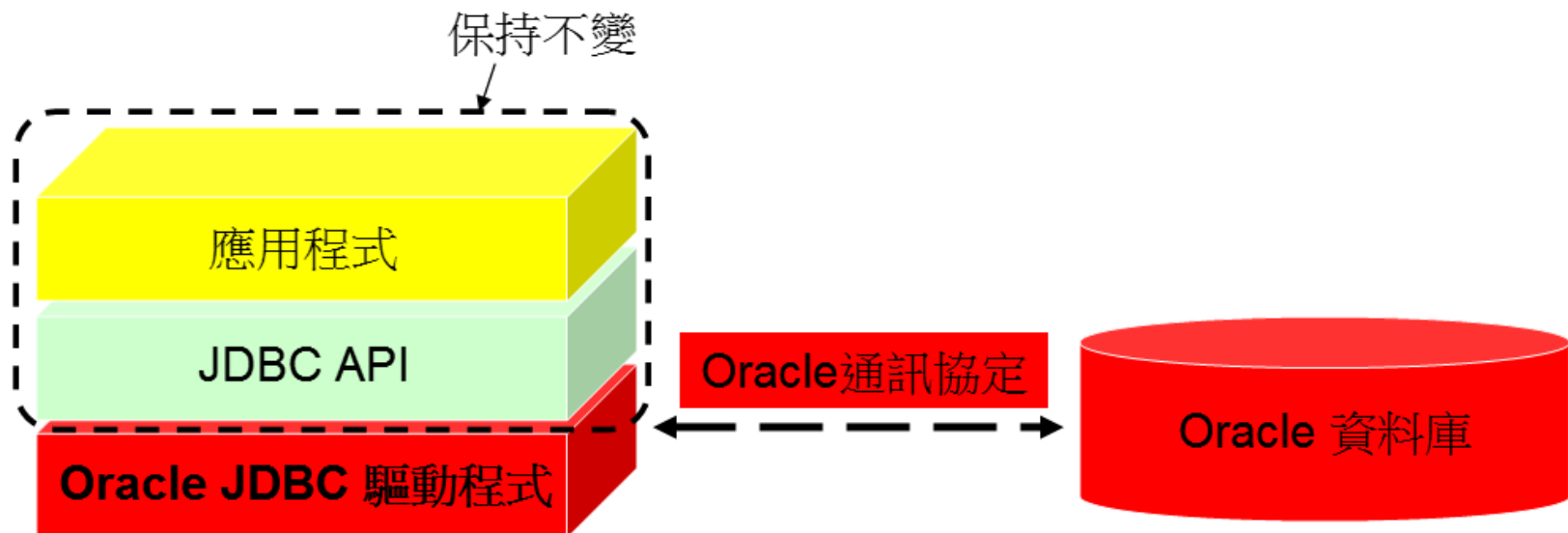
簡介 JDBC



簡介 JDBC

- 應用程式會使用 JDBC 連線資料庫

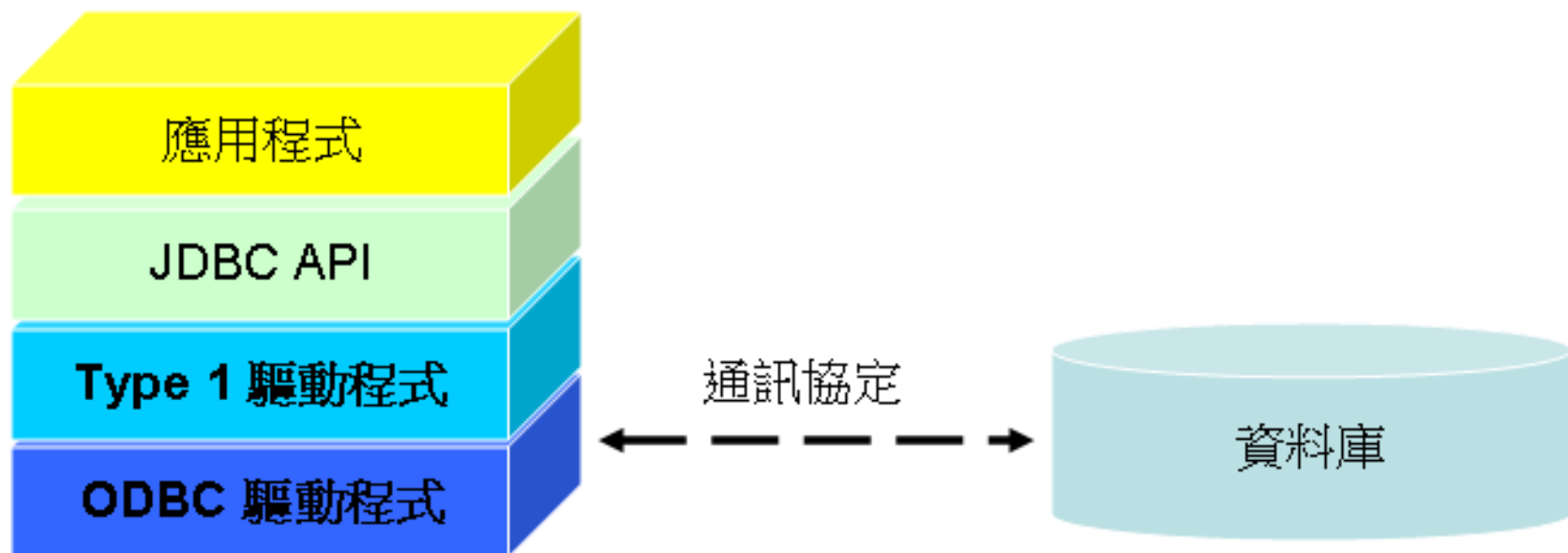
```
Connection conn = DriverManager.getConnection(...);  
Statement st = conn.createStatement();  
ResultSet rs = st.executeQuery("SELECT * FROM T_USER");
```



簡介 JDBC

- 廠商在實作 JDBC 驅動程式時，依方式可將驅動程式分作四種類型
 - Type 1 : JDBC-ODBC Bridge Driver
 - Type 2 : Native API Dirver
 - Type 3 : JDBC-Net Driver
 - Type 4 : Native Protocol Driver

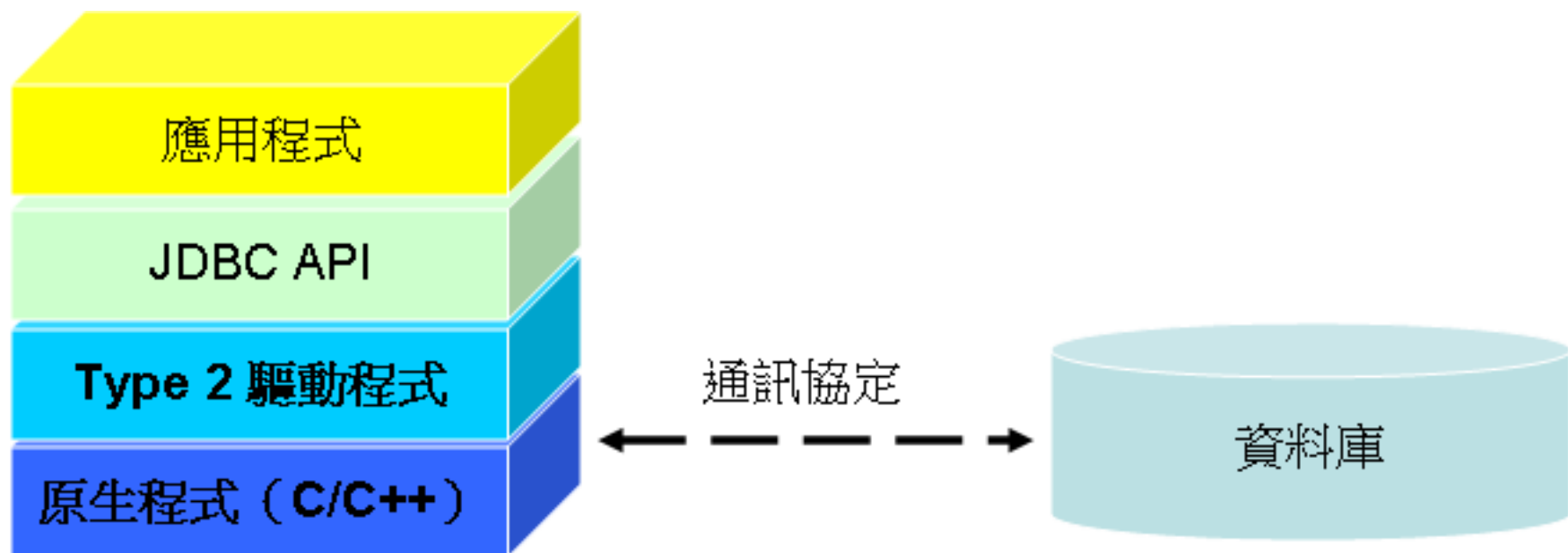
Type 1 : JDBC-ODBC Bridge Driver



Type 1 : JDBC-ODBC Bridge Driver

- 實作這種驅動程式非常簡單
- JDBC 與 ODBC 並非一對一的對應，所以部份呼叫無法直接轉換，因此有些功能是受限的
- 多層呼叫轉換結果，存取速度也會受到限制
- ODBC 本身需在平台上先設定好，彈性不足，ODBC 驅動程式本身也有跨平台的限制

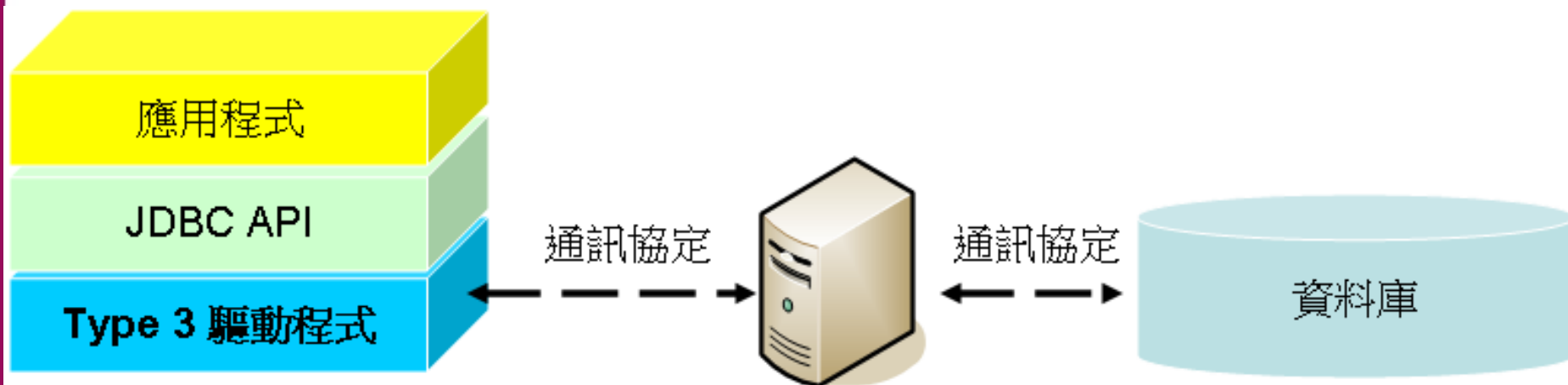
Type 2 : Native API Dirver



Type 2 : Native API Dirver

- 驅動程式本身與平台相依，沒有達到 JDBC 驅動程式的目標之一：跨平台
- 直接呼叫資料庫原生 API，因此在速度上，有機會成為四種類型中最快的驅動程式
 - 速度的優勢是在於獲得資料庫回應資料後，建構相關 JDBC API 實作物件時
- 使用前必須先在各平台進行驅動程式的安裝設定（像是安裝資料庫專屬的原生程式庫）

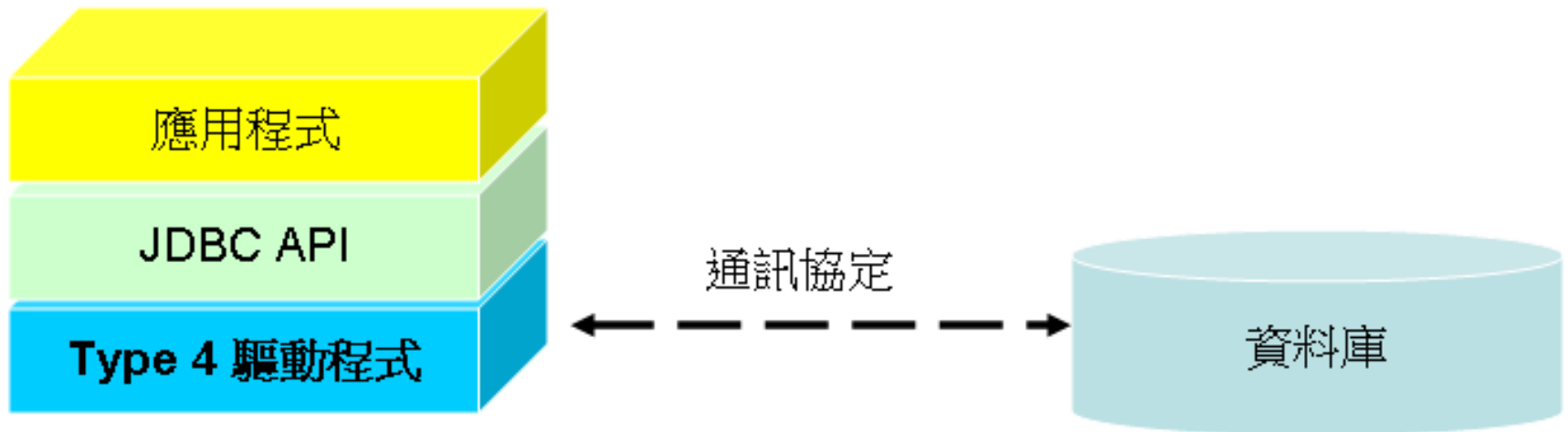
Type 3 : JDBC-Net Driver



Type 3 : JDBC-Net Driver

- 可使用純粹的 Java 技術來實現，可以跨平台
- 架構彈性高，客戶端不受影響
- 經由中介伺服器轉換，速度較慢，獲得架構上的彈性是使用這類型驅動程式的目的

Type 4 : Native Protocol Driver



Type 4 : Native Protocol Driver

- 驅動程式可以使用純粹 Java 技術來實現，可以跨平台
- 效能上也能有不錯的表現
- 不需要如 Type 3 獲得架構上的彈性時，通常會使用這類型驅動程式

連接資料庫

- 為了要連接資料庫系統，必須要有廠商實作的 JDBC 驅動程式，必須在 CLASSPATH 中設定驅動程式 JAR 檔案

連接資料庫

- 基本資料庫操作相關的 JDBC 介面或類別是位於 `java.sql` 套件中
- 要取得資料庫連線，必須有幾個動作：
 - 註冊 `Driver` 實作物件
 - 取得 `Connection` 實作物件
 - 關閉 `Connection` 實作物件

連接資料庫

- 以 MySQL 實作的驅動程式為例，`com.mysql.jdbc.Driver` 類別實作了 **`java.sql.Driver`** 介面
- 管理 `Driver` 實作物件的類別是 **`java.sql.DriverManager`**

```
DriverManager.registerDriver(new com.mysql.jdbc.Driver());
```

連接資料庫

- 實際上很少自行撰寫程式碼進行這個動作
- 只要想辦法載入 `Driver` 介面的實作類別 `.class` 檔案，就會完成註冊
- 可以透過 `java.lang.Class` 類別的 `forName()`，動態載入驅動程式類別

```
try {  
    Class.forName("com.mysql.jdbc.Driver");  
}  
catch(ClassNotFoundException e) {  
    throw new RuntimeException("找不到指定的類別");  
}
```

連接資料庫

```
package com.mysql.jdbc;
import java.sql.SQLException;
public class Driver extends NonRegisteringDriver
                                implements java.sql.Driver {
    static {
        try {
            java.sql.DriverManager.registerDriver(new Driver());
        } catch (SQLException E) {
            throw new RuntimeException("Can't register driver!");
        }
    }

    public Driver() throws SQLException {}
}
```

連接資料庫

- 使用 JDBC 時，要求載入 .class 檔案的方式有四種：

- 使用 `Class.forName()`
- 自行建立 `Driver` 介面實作類別的實例

```
java.sql.Driver driver = new com.mysql.jdbc.Driver();
```

- 啟動 JVM 時指定 `jdbc.drivers` 屬性


```
> java -Djdbc.drivers=com.mysql.jdbc.Driver;ooo.XXXDriver YourProgram
```

- 設定 JAR 中 `/services/java.sql.Driver` 檔案
(JDK6)

連接資料庫

```
Connection conn = DriverManager.getConnection(  
    jdbcUrl, username, password);
```

協定:子協定:資料來源識別



- 除了「協定」在 JDBC URL 中總是 jdbc 開始之外，JDBC URL 格式各家資料庫都不相同，必須查詢資料庫產品使用手冊

取得 Connection 實作物件

- 以 MySQL 為例：

`jdbc:mysql://主機名稱:連接埠/資料庫名稱?參數=值&參數=值`

`jdbc:mysql://localhost:3306/demo?user=root&password=123456`

`jdbc:mysql://localhost:3306/demo?user=root&password=123&useUnicode=true&characterEncoding=UTF8`

- XML 中：

`jdbc:mysql://localhost:3306/demo?user=root&password=123&useUnicode=true&characterEncoding=UTF8`

```
Connection conn = null;
SQLException ex = null;
try {
    String url = "jdbc:mysql://localhost:3306/demo";
    String user = "root";
    String password = "123456";
    conn = DriverManager.getConnection(url, user, password);
    ....
}
catch(SQLException e) {
    ex = e;
}
finally {
    if(conn != null) {
        try {
            conn.close();
        }
        catch(SQLException e) {
            if(ex == null) {
                ex = e;
            }
        }
    }
    if(ex != null) {
        throw new RuntimeException(ex);
    }
}
```

連接資料庫

- 在處理 JDBC 時很常遇到的 `SQLException` 例外物件，為資料庫操作過程發生錯誤時的代表物件
- 受檢例外（`Checked Exception`），必須使用 `try...catch` 明確處理，在例外發生時嘗試關閉相關資源

連接資料庫

- 取得 `Connection` 物件之後，可以使用 **`isClosed()`** 方法測試與資料庫的連接是否關閉
- 若確定不再需要連接，則必須使用 `close()` 來關閉與資料庫的連接

連接資料庫

- 從 JDK7 之後，JDBC 的 Connection、Statement、ResultSet 等介面，都是

java.lang.AutoCloseable 子介面

```
String url = "jdbc:mysql://localhost:3306/demo";
String user = "root";
String password = "123456";
try(Connection conn = DriverManager.getConnection(url, user, password)) {
    ....
}
catch(SQLException e) {
    throw new RuntimeException(e);
}
```

連接資料庫

- DriverManager 如何進行連線？

```
SQLException reason = null;
for (int i = 0; i < drivers.size(); i++) { // 逐一取得 Driver 實例
    ...
    DriverInfo di = (DriverInfo)drivers.elementAt(i);
    ...
    try {
        Connection result = di.driver.connect(url, info); // 嘗試連線
        if (result != null) {
            return (result); // 取得 Connection 就傳回
        }
    } catch (SQLException ex) {
        if (reason == null) { // 記錄第一個發生的例外
            reason = ex;
        }
    }
}
if (reason != null) {
    println("getConnection failed: " + reason);
    throw reason; // 如果有例外物件就丟出
}
throw new SQLException( // 沒有適用的 Driver 實例，丟出例外
    "No suitable driver found for "+ url, "08001");
```


連接資料庫

```
public class ConnectionDemo {  
    public static void main(String[] args)  
        throws ClassNotFoundException, SQLException {  
        Class.forName("com.mysql.jdbc.Driver"); ← ❶ 載入驅動程式  
        String jdbcUrl = "jdbc:mysql://localhost:3306/demo";  
        String user = "root";  
        String passwd = "openhome";  
        try(Connection conn =  
            DriverManager.getConnection(jdbcUrl, user, passwd)) {  
            out.printf("已%s 資料庫連線%n",  
                conn.isClosed() ? "關閉" : "開啟");  
        }  
    }  
}
```

使用 Statement、ResultSet

- `java.sql.Statement` 物件是 SQL 陳述的代表物件
- 可以使用 `Connection` 的 **`createStatement()`** 來建立

```
Statement stmt = conn.createStatement();
```

- 可以使用 **`executeUpdate()`**、**`executeQuery()`** 等方法來執行 SQL

使用 Statement、ResultSet

- `executeUpdate()` 主要是用來執行 CREATE TABLE、INSERT、DROP TABLE、ALTER TABLE 等會改變資料庫內容的 SQL

```
stmt.executeUpdate("INSERT INTO t_message VALUES(1, 'justin', " +  
    "'justin@mail.com', 'message...')");
```

- `executeQuery()` 方法則是用於 SELECT 等查詢資料庫的 SQL

使用 Statement、ResultSet

- `executeUpdate()` 會傳回 `int` 結果，表示資料變動的筆數
- `executeQuery()` 會傳回 `java.sql.ResultSet` 物件，代表查詢的結果

```
ResultSet result = stmt.executeQuery("SELECT * FROM t_message");
while(result.next()) {
    int id = result.getInt("id");
    String name = result.getString("name");
    String email = result.getString("email");
    String msg = result.getString("msg");
    // ...
}
```

使用 Statement、ResultSet

```
ResultSet result = stmt.executeQuery("SELECT * FROM t_message");
while(result.next()) {
    int id = result.getInt(1);
    String name = result.getString(2);
    String email = result.getString(3);
    String msg = result.getString(4);
    // ...
}
```

使用 Statement、ResultSet

- Statement 的 **execute()** 可以用來執行 SQL
 - 傳回 `true` 的話表示將傳回查詢結果，可以使用 **getResultSet()** 取得 `ResultSet` 物件
 - 傳回 `false`，表示傳回更新筆數或沒有結果，可以使用 **getUpdateCount()** 取得更新筆數。

```
if(stmt.execute(sql)) {  
    ResultSet rs = stmt.getResultSet(); // 取得查詢結果 ResultSet  
    ...  
}  
else { // 這是個更新操作  
    int updated = stmt.getUpdateCount(); // 取得更新筆數  
    ...  
}
```

```
public class MessageDAO {  
    private String url;  
    private String user;  
    private String passwd;
```

```
    public MessageDAO(String url, String user, String passwd) {  
        this.url = url;  
        this.user = user;  
        this.passwd = passwd;  
    }
```

❶ 這個方法會在資料庫中新增留言

```
    public void add(Message message) {
```

❷ 取得 Connection 物件

```
        try(Connection conn = DriverManager.getConnection(url, user, passwd);
```

```
            Statement statement = conn.createStatement()) { ← ❸ 建立 Statement 物件
```

```
            String sql = String.format(
```

```
                "INSERT INTO t_message(name, email, msg) VALUES ('%s', '%s', '%s')",  
                message.getName(), message.getEmail(), message.getMsg());
```

```
            statement.executeUpdate(sql); ← ❹ 執行 SQL 陳述句
```

```
        } catch(SQLException ex) {
```

```
            throw new RuntimeException(ex);
```

```
        }
```

```
    }
```

```
public List<Message> get() { ← ❸ 這個方法會從資料庫中查詢所有留言
    List<Message> messages = new ArrayList<>();
    try(Connection conn = DriverManager.getConnection(url, user, passwd);
        Statement statement = conn.createStatement()) {
        ResultSet result =
            statement.executeQuery("SELECT * FROM t_message");
        while (result.next()) {
            Message message = toMessage(result);
            messages.add(message);
        }
    } catch(SQLException ex) {
        throw new RuntimeException(ex);
    }
    return messages;
}
```


使用 PreparedStatement

```
Statement statement = conn.createStatement();
String sql = String.format(
    "INSERT INTO t_message(name, email, msg) VALUES ('%s', '%s', '%s')",
    message.getName(), message.getEmail(), message.getMsg());
statement.executeUpdate(sql);
```

```
PreparedStatement stmt = conn.prepareStatement(
    "INSERT INTO t_message VALUES (?, ?, ?, ?)");

stmt.setInt(1, 2);
stmt.setString(2, "momor");
stmt.setString(3, "momor@mail.com");
stmt.setString(4, "message2...");
stmt.executeUpdate();
stmt.clearParameters();
```

- 使用 PreparedStatement 改寫先前 MessageDAO 中 add() 執行 SQL 語句的部份

```
public void add(Message message) {  
    try(Connection conn = DriverManager.getConnection(url, user, passwd);  
        PreparedStatement statement = conn.prepareStatement(  
            "INSERT INTO t_message(name, email, msg) VALUES (?, ?, ?)")) {  
        statement.setString(1, message.getName());  
        statement.setString(2, message.getEmail());  
        statement.setString(3, message.getMsg());  
        statement.executeUpdate();  
    } catch(SQLException ex) {  
        throw new RuntimeException(ex);  
    }  
}
```

使用 PreparedStatement

- 安全 ...

```
Statement statement = connection.createStatement();
String queryString = "SELECT * FROM user_table WHERE username='" +
    username + "' AND password='" + password + "'";
ResultSet resultSet = statement.executeQuery(queryString);
```

```
SELECT * FROM user_table
WHERE username='caterpillar' AND password='123456'
```

```
SELECT * FROM user_table
WHERE username='caterpillar' AND password=' ' OR '1'='1'
```

```
SELECT * FROM user_table
WHERE username='caterpillar' AND password='' OR '1'='1'
```

使用 PreparedStatement

- 以下 username 與 password 將被視作是 SQL 中純粹的字串，而不會當作 SQL 語法來

左刀金四

```
PreparedStatement stmt = conn.prepareStatement(  
    "SELECT * FROM user_table WHERE username=? AND password=?");  
stmt.setString(1, username);  
stmt.setString(2, password);
```

使用 CallableStatement

- 呼叫資料庫的預存程序（Stored Procedure）

```
{?= call <程序名稱>[<引數 1>,<引數 2>, ...]}  
{call <程序名稱>[<引數 1>,<引數 2>, ...]}
```

- 必須呼叫 **prepareCall()** 建立
CallableStatement 實例

Java 型態與 SQL 型態對應

Java 型態	SQL 型態
boolean	BIT
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
float	FLOAT
double	DOUBLE
byte[]	BINARY、VARBINARY、LONGBINARY
java.lang.String	CHAR、VARCHAR、LONGVARCHAR
java.math.BigDecimal	NUMERIC、DECIMAL
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP

Java 型態與 SQL 型態對應

- 日期時間在 JDBC 中，並不是使用 `java.util.Date`
 - 年、月、日、時、分、秒、毫秒
- 在 JDBC 中要表示日期，是使用 `java.sql.Date`
 - 年、月、日
- 要表示時間的話則是使用 `java.sql.Time`
 - 時、分、秒
- 使用 `java.sql.Timestamp`
 - 時、分、秒、微秒

Java 型態與 SQL 型態對應

- JDK8 新時間日期 API

- 對於 Timestamp 實例，你可以使用
toInstant() 方法將之轉為 Instant 實例
- 如果有個 Instant 實例，可以透過
Timestamp 的 from() 靜態方法，將之轉為
Timestamp 實例

```
Instant instant = timestamp.toInstant();
```

```
Timestamp timestamp2 = Timestamp.from(instant);
```


使用 DataSource 取得連線

- 實際應用程式開發時，JDBC URL、使用者名稱、密碼等資訊是很敏感的資訊，有些開發人員根本無從得知
- 如果 MessageDAO 的使用者無法告知這些資訊，你如何改寫 MessageDAO ？

使用 DataSource 取得連線

- 可以讓 MessageDAO 依賴於 **javax.sql.DataSource** 介面 …

```
public class MessageDAO3 {  
    private DataSource dataSource;  
    public MessageDAO3(DataSource dataSource) {  
        this.dataSource = dataSource;  
    }  
}
```

使用 DataSource 取得連線

```
public void add(Message message) {
    try(Connection conn = dataSource.getConnection();
        PreparedStatement statement = conn.prepareStatement(
            "INSERT INTO t_message(name, email, msg) VALUES (?, ?, ?)") {
        ...略
    } catch(SQLException ex) {
        throw new RuntimeException(ex);
    }
}

public List<Message> get() {
    List<Message> messages = null;
    try(Connection conn = dataSource.getConnection();
        Statement statement = conn.createStatement()) {
        ...略
    } catch(SQLException ex) {
        throw new RuntimeException(ex);
    }
    return messages;
}
```

使用 DataSource 取得連線

- 實作具簡單連接池的 DataSource...

```
import java.util.logging.Logger;
import javax.sql.DataSource;
```

❶ 實作 DataSource

```
public class SimpleConnectionPoolDataSource implements DataSource {
    private Properties props;
    private String url;
    private String user;
    private String passwd;
    private int max; // 連接池中最大 Connection 數目
    private List<Connection> conns; ← ❷ 維護可重用的 Connection 物件
```

```
    public SimpleConnectionPoolDataSource()
        throws IOException, ClassNotFoundException {
        this("jdbc.properties");
    }
```

❸ 可指定 properties 檔案

```
    public SimpleConnectionPoolDataSource(String configFile)
        throws IOException, ClassNotFoundException {
        props = new Properties();
        props.load(new FileInputStream(configFile));
```

```
url = props.getProperty("cc.openhome.url");
user = props.getProperty("cc.openhome.user");
passwd = props.getProperty("cc.openhome.password");
max = Integer.parseInt(props.getProperty("cc.openhome.poolmax"));

conns = Collections.synchronizedList(new ArrayList<Connection>());
}

public synchronized Connection getConnection() throws SQLException {
    if(conns.isEmpty()) { ← ❹ 如果 List 為空就建立新的 ConnectionWrapper
        return new ConnectionWrapper(
            DriverManager.getConnection(url, user, passwd),
            conns,
            max
        );
    }
    else { ← ❺ 否則傳回 List 中一個 Connection
        return conns.remove(conns.size() - 1);
    }
}
```

```
private class ConnectionWrapper implements Connection {
    private Connection conn;
    private List<Connection> conns;
    private int max;

    public ConnectionWrapper(Connection conn,
                             List<Connection> conns, int max) {
        this.conn = conn;
        this.conns = conns;
        this.max = max;
    }

    @Override
    public void close() throws SQLException {
        if (conns.size() == max) { ← ⑦ 如果超出最大可維護 Connection
            conn.close();          數量就關閉 Connection
        }
        else {
            conns.add(this); ← ⑧ 否則放入 List 中以備重用
        }
    }

    @Override
    public Statement createStatement() throws SQLException {
        return conn.createStatement();
    }
    ...略
}
...略
```

使用 DataSource 取得連線

```
JDBCDemo jdbc.properties
```

```
cc.openhome.url=jdbc:mysql://localhost:3306/demo  
cc.openhome.user=root  
cc.openhome.password=123456  
cc.openhome.poolmax=10
```

```
MessageDAO3 dao =  
    new MessageDAO3(new SimpleConnectionPoolDataSource());
```

使用 ResultSet 捲動、更新資料

- 從 JDBC 2.0 開始，ResultSet 並不僅可以使用 **previous()**、**first()**、**last()** 等方法前後移動資料游標，還可以呼叫 **updateXXX()**、**updateRow()** 等方法進行資料修改

使用 ResultSet 捲動、更新資料

- 建立 Statement 或 PreparedStatement 實例時，可以指定結果集類型與並行方式

```
createStatement(int resultSetType, int resultSetConcurrency)  
prepareStatement(String sql,  
                  int resultSetType, int resultSetConcurrency)
```

- 結果集類型可以指定三種設定：
 - ResultSet.TYPE_FORWARD_ONLY (預設)
 - ResultSet.TYPE_SCROLL_INSENSITIVE
 - ResultSet.TYPE_SCROLL_SENSITIVE

使用 ResultSet 捲動、更新資料

- 更新設定可以有兩種指定：
 - ResultSet.CONCUR_READ_ONLY (預設)
 - ResultSet.CONCUR_UPDATABLE

```
Statement stmt = conn.createStatement(  
    ResultSet.TYPE_SCROLL_INSENSITIVE,  
    ResultSet.CONCUR_UPDATEABLE);
```

```
PreparedStatement stmt = conn.prepareStatement(  
    "SELECT * FROM t_message",  
    ResultSet.TYPE_SCROLL_INSENSITIVE,  
    ResultSet.CONCUR_UPDATEABLE);
```

使用 ResultSet 捲動、更新資料

- 資料游標移動 ...

```
Statement stmt = conn.createStatement("SELECT * FROM t_message",
                                         ResultSet.TYPE_SCROLL_INSENSITIVE,
                                         ResultSet.CONCUR_READ_ONLY);

ResultSet rs = stmt.executeQuery();
rs.absolute(2);           // 移至第 2 列
rs.next();                // 移至第 3 列
rs.first();               // 移至第 1 列
boolean b1 = rs.isFirst(); // b1 是 true
```

使用 ResultSet 捲動、更新資料

- 使用 ResultSet 進行資料修改
 - 必須選取單一表格
 - 必須選取主鍵
 - 必須選取所有 NOT NULL 的值
- 更新資料 ...

```
Statement stmt = conn.prepareStatement("SELECT * FROM t_message",
                                         ResultSet.TYPE_SCROLL_INSENSITIVE,
                                         ResultSet.CONCUR_READ_ONLY);

ResultSet rs = stmt.executeQuery();
rs.next();
rs.updateString(3, "caterpillar@openhome.cc");
rs.updateRow();
```

使用 ResultSet 捲動、更新資料

- 新增資料 ...

```
Statement stmt = conn.prepareStatement("SELECT * FROM t_message",
                                         ResultSet.TYPE_SCROLL_INSENSITIVE,
                                         ResultSet.CONCUR_READ_ONLY);

ResultSet rs = stmt.executeQuery();
rs.moveToInsertRow();
rs.updateString(2, "momor");
rs.updateString(3, "momor@openhome.cc");
rs.updateString(4, "blah..blah");
rs.insertRow();
rs.moveToCurrentRow();
```

使用 ResultSet 捲動、更新資料

- 刪除資料 ...

```
Statement stmt = conn.prepareStatement("SELECT * FROM t_message",
                                         ResultSet.TYPE_SCROLL_INSENSITIVE,
                                         ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery();
rs.absolute(3);
rs.deleteRow();
```

批次更新

- 以下每一次執行 `executeUpdate()`，其實都會向資料庫發送一次 SQL

```
Statement stmt = conn.createStatement();
while(someCondition) {
    stmt.executeUpdate(
        "INSERT INTO t_message(name,email,msg) VALUES('...','...','...')");
}
```

批次更新

- 可以使用 **addBatch()** 方法來收集 SQL
- 使用 **executeBatch()** 方法將所收集的 SQL 傳送出去

```
Statement stmt = conn.createStatement();
while(someCondition) {
    stmt.addBatch(
        "INSERT INTO t_message(name,email,msg) VALUES('...', '...', '...')");
}
stmt.executeBatch();
```


批次更新

- 以 MySQL 驅動程式的 Statement 實作為

例

```
public synchronized void addBatch(String sql) throws SQLException {  
    if (this.batchedArgs == null) {  
        this.batchedArgs = new ArrayList();  
    }  
    if (sql != null) {  
        this.batchedArgs.add(sql);  
    }  
}
```

批次更新

- 使用 `executeBatch()` 時，SQL 的執行順序，就是 `addBatch()` 時的順序
- `executeBatch()` 會傳回 `int[]`，代表每筆 SQL 造成的資料異動列數
- 先前已開啟的 `ResultSet` 會被關閉，執行過後收集 SQL 用的 `List` 會被清空
- 任何的 SQL 錯誤，會丟出 **`BatchUpdateException`**
- 可以使用這個物件的 **`getUpdateCounts()`** 取得 `int[]`，代表先前執行成功的 SQL 所造成的異動筆數

批次更新

- PreparedStatement 使用批次更新

```
PreparedStatement stmt = conn.prepareStatement(
    "INSERT INTO t_message(name,email,msg) VALUES(?, ?, ?)");
while(someCondition) {
    stmt.setString(1, "..");
    stmt.setString(2, "..");
    stmt.setString(3, "..");
    stmt.addBatch(); // 收集參數
}
stmt.executeBatch(); // 送出所有參數
```

批次更新

- 以 MySQL 的 PreparedStatement 實作類別為例

```
public void addBatch() throws SQLException {
    if (this.batchedArgs == null) {
        this.batchedArgs = new ArrayList();
    }
    this.batchedArgs.add(new BatchParams(this.parameterValues,
        this.parameterStreams, this.isStream, this.streamLengths,
        this.isNull));
}
```

Blob 與 Clob

- BLOB 全名 Binary Large Object，用於儲存大量的二進位資料，像是圖檔、影音檔等
- CLOB 全名 Character Large Object，用於儲存大量的文字資料
- `java.sql.Blob` 與 `java.sql.Clob` 兩個類別分別代表 BLOB 與 CLOB 資料

Blob 與 Clob

- Blob 擁有
getBinaryStream()、getBytes() 等方法，可以取得代表欄位來源的
InputStream 或欄位的 byte[] 資料
- Clob 擁有
getCharacterStream()、getAsciiStream() 等方法，可以取得 Reader 或
InputStream 等資料

Blob 與 Clob

- 也可以把 BLOB 欄位對應 `byte[]` 或輸入 / 輸出串流
- 使用 `PreparedStatement` 的 **`setBytes()`** 來設定要存入的 `byte[]` 資料，使用 **`setBinaryStream()`** 來設定代表輸入來源的 `InputStream`
- 使用 `ResultSet` 的 **`getBytes()`** 以 `byte[]` 取得欄位中儲存的資料，或以 **`getBinaryStream()`** 取得代表欄位來源的 `InputStream`

Blob 與 Clob

```
InputStream in = readFileAsInputStream(".....");
PreparedStatement stmt = conn.prepareStatement(
    "INSERT INTO IMAGES(src, img) VALUE(?, ?)");
stmt.setString(1, "...");
stmt.setBinaryStream(2, in);
stmt.executeUpdate();
```

```
PreparedStatement stmt = conn.prepareStatement(
    "SELECT img FROM IMAGES");
ResultSet rs = stmt.executeQuery();
while(rs.next()) {
    InputStream in = rs.getBinaryStream(1);
    //..使用 InputStream 作資料讀取
}
```


簡介交易

- 原子性 (Atomicity)
- 一致性 (Consistency)
- 隔離行為 (Isolation behavior)
- 持續性 (Durability)

```
Connection conn = null;
try {
    conn = dataSource.getConnection();
    conn.setAutoCommit(false); // 取消自動提交
    Statement stmt = conn.createStatement();
    stmt.executeUpdate("INSERT INTO ...");
    stmt.executeUpdate("INSERT INTO ...");
    conn.commit(); // 提交
}
catch(SQLException e) {
    e.printStackTrace();
    if(conn != null) {
        try {
            conn.rollback(); // 撤回
        }
        catch(SQLException ex) {
            ex.printStackTrace();
        }
    }
}
finally {
    ...
    if(conn != null) {
        try {
            conn.setAutoCommit(true); // 回復自動提交
            conn.close();
        }
        catch(SQLException ex) {
            ex.printStackTrace();
        }
    }
}
```

```
conn.setAutoCommit(false);
Statement stmt = conn.createStatement();
stmt.executeUpdate("INSERT INTO ...");
...
point = conn.setSavepoint(); // 設定儲存點
stmt.executeUpdate("INSERT INTO ...");
...
conn.commit();
}
catch(SQLException e) {
    e.printStackTrace();
    if(conn != null) {
        try {
            if(point == null) {
                conn.rollback();
            }
            else {
                conn.rollback(point); // 撤回儲存點
                conn.releaseSavepoint(point); // 釋放儲存點
            }
        }
        catch(SQLException ex) {
            ex.printStackTrace();
        }
    }
}
finally {
    ...
    if(conn != null) {
        try {
            conn.setAutoCommit(true);
        }
    }
}
```

```
try {
    conn.setAutoCommit(false);
    stmt = conn.createStatement();
    while(someCondition) {
        stmt.addBatch("INSERT INTO ...");
    }
    stmt.executeBatch();
    conn.commit();
} catch(SQLException ex) {
    ex.printStackTrace();
    if(conn != null) {
        try {
            conn.rollback();
        } catch(SQLException e) {
            e.printStackTrace();
        }
    }
} finally {
    ...
    if(conn != null) {
        try {
            conn.setAutoCommit(true);
            conn.close();
        }
        catch(SQLException ex) {
            ex.printStackTrace();
        }
    }
}
```

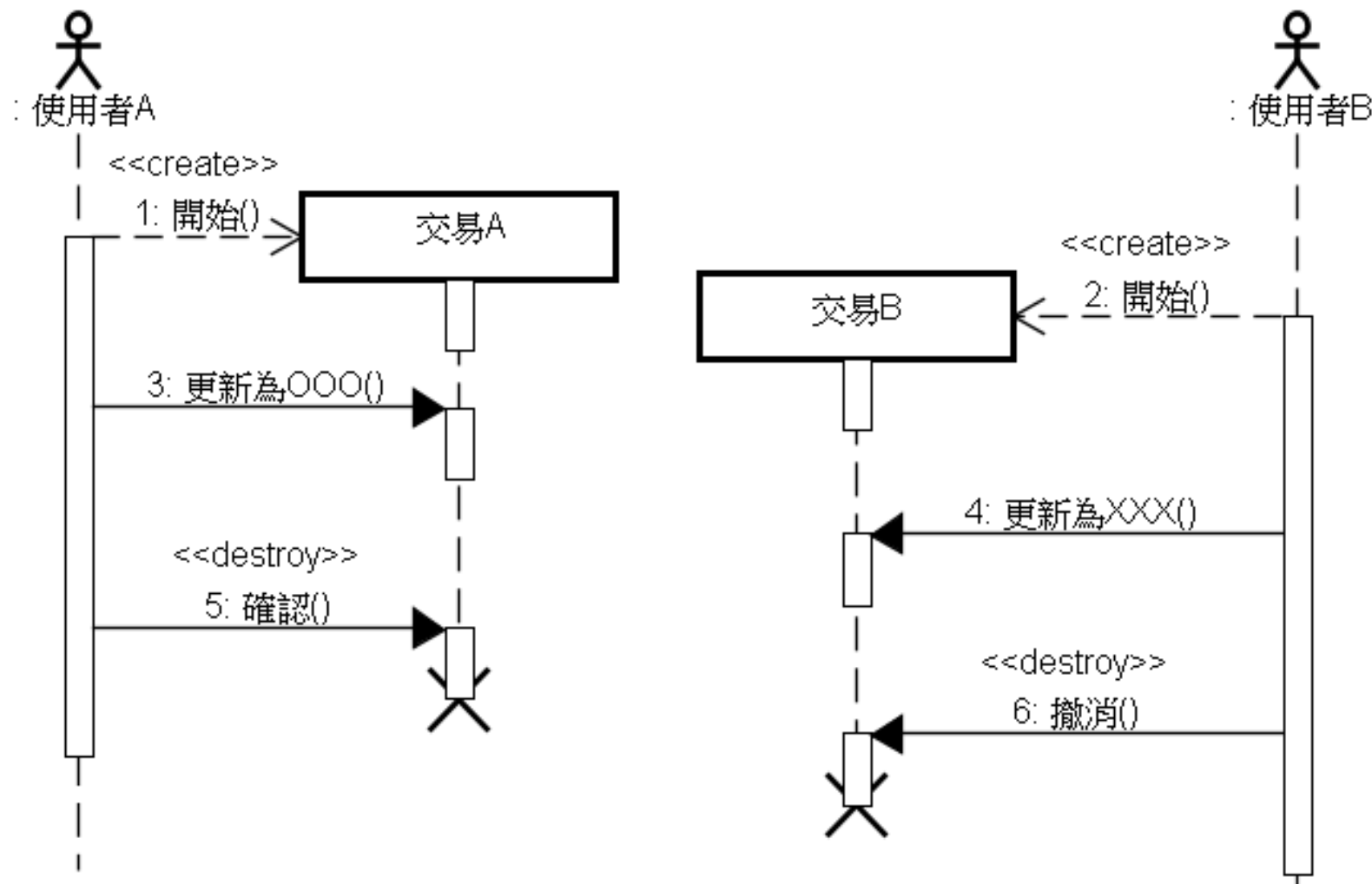
隔離行為

- 可以透過 `Connection` 的 `getTransactionIsolation()` 取得資料庫目前的隔離行為設定
- 透過 `setTransactionIsolation()` 可提示資料庫設定指定的隔離行為

隔離行為

- 可設定常數是定義在 `Connection` 上
 - `TRANSACTION_NONE`
 - `TRANSACTION_UNCOMMITTED`
 - `TRANSACTION_COMMITTED`
 - `TRANSACTION_REPEATABLE_READ`
 - `TRANSACTION_SERIALIZABLE`

更新遺失 (Lost update)



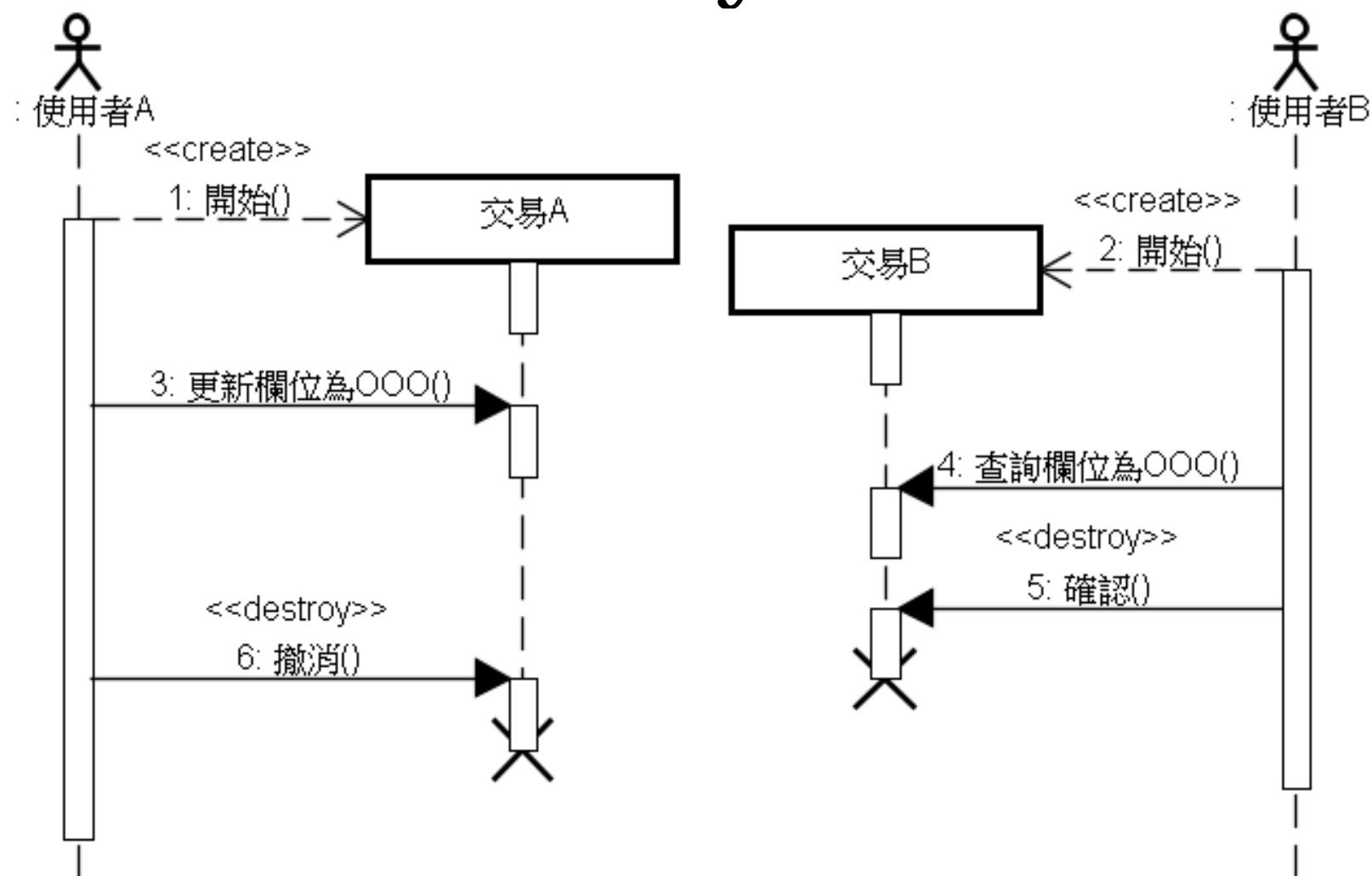
```

sequenceDiagram
    actor UserA as 使用者A
    participant TransactionA as 交易A
    participant TransactionB as 交易B
    actor UserB as 使用者B

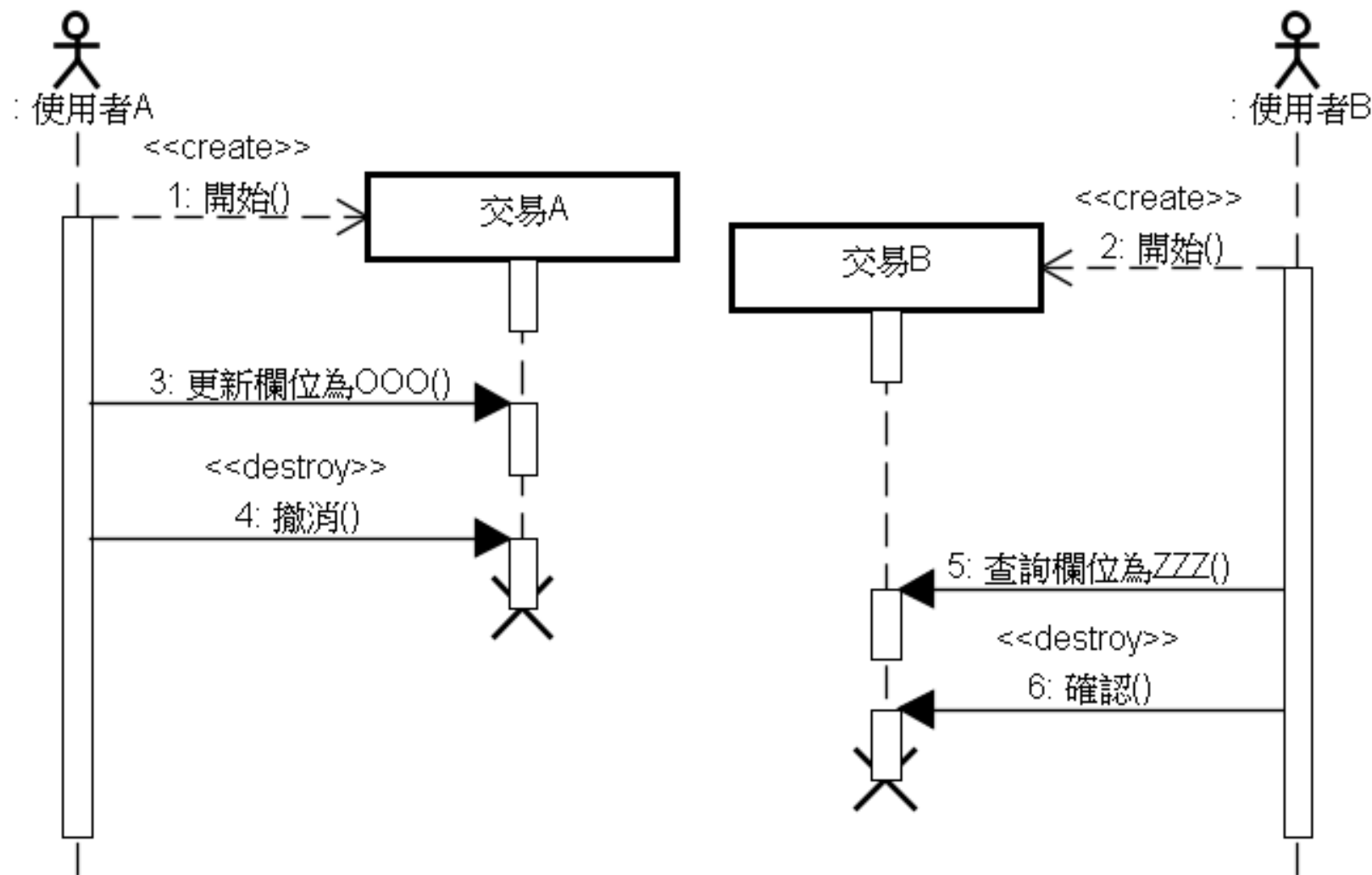
    UserA->>TransactionA: <<create>>
    UserA->>TransactionA: 1: 開始()
    TransactionA->>TransactionA: 3: 更新為000()
    UserA->>TransactionA: <<destroy>>
    UserA->>TransactionA: 4: 確認()
    TransactionA->>TransactionA: [End]

    UserB->>TransactionB: <<create>>
    UserB->>TransactionB: 2: 開始()
    TransactionB->>TransactionB: 5: 更新為XXX()
    UserB->>TransactionB: <<destroy>>
    UserB->>TransactionB: 6: 撤消()
    TransactionB->>TransactionB: [End]
  
```

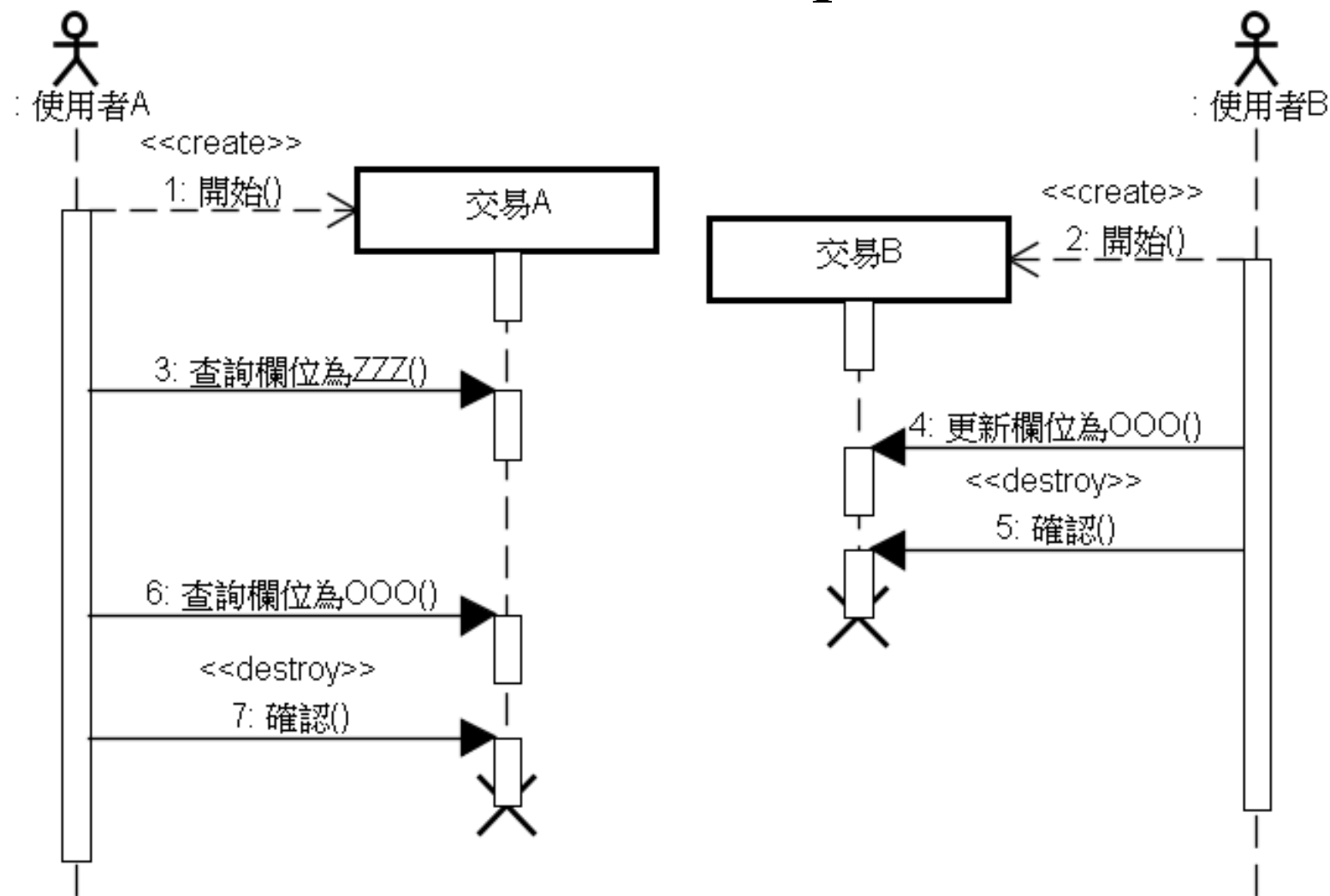

髒讀 (Dirty read)



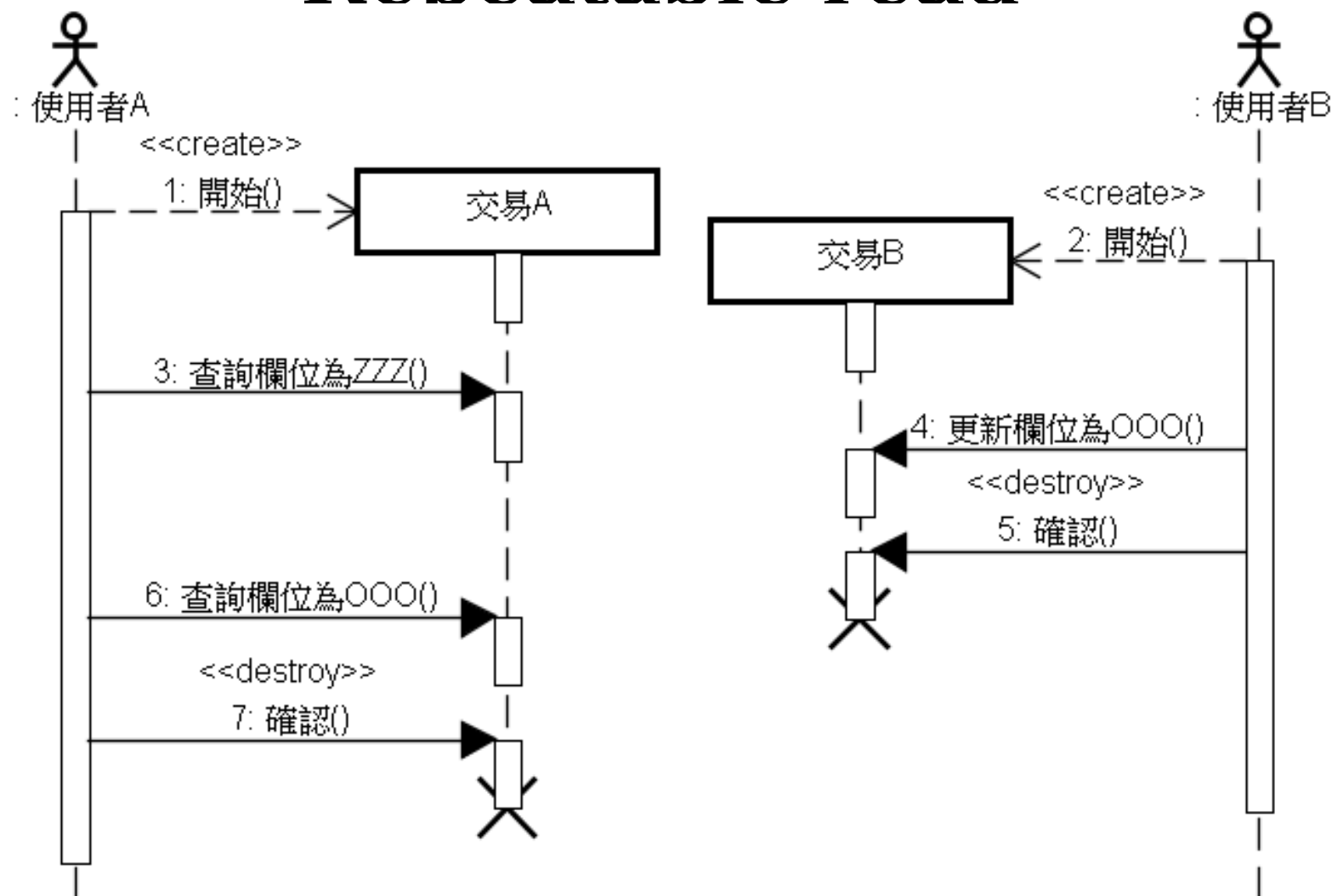
Read committed



無法重複的讀取（Unrepeatable read）



Repeatable read



幻讀（Phantom read）

- 同一交易期間，讀取到的資料筆數不一致。例如交易 A 第一次讀取得到五筆資料，此時交易 B 新增了一筆資料，導致交易 B 再次讀取得到六筆資料
- 如果隔離行為設定為可重複讀取，但發生幻讀現象，可以設定隔離層級為「可循序」（Serializable），也就是在有交易時若有資料不一致的疑慮，交易必須可以照順序逐一進行

隔離行為與可預防之問題

隔離行為	Lost update	Dirty read	Unrepeatable read	Phantom read
Read uncommitted	預防			
Read committed	預防	預防		
Repeatable read	預防	預防	預防	
Serializable	預防	預防	預防	預防

隔離行為

- 得知是否支援某個隔離行為

```
DatabaseMetadata meta = conn.getMetaData();  
boolean isSupported = meta.supportsTransactionIsolationLevel(  
    Connection.TRANSACTION_READ_COMMITTED);
```

簡介 metadata

- 詮讀資料的資料 (Data about data)
- 可以透過 `Connection` 的 `getMetaData()` 方法取得 `DatabaseMetaData` 物件
- 可以透過 `ResultSet` 的 `getMetaData()` 方法，取得 `ResultSetMetaData` 物件


```
public List<ColumnInfo> getAllColumnInfo() {  
    List<ColumnInfo> infos = null;  
    try(Connection conn = dataSource.getConnection()) {  
        DatabaseMetaData meta = conn.getMetaData();  
        ResultSet crs = meta.getColumns("demo", null, "t_message", null);  
        infos = new ArrayList<>();  
        while(crs.next()) {  
            ColumnInfo info = new ColumnInfo();  
            info.setName(crs.getString("COLUMN_NAME"));  
            info.setType(crs.getString("TYPE_NAME"));  
            info.setSize(crs.getInt("COLUMN_SIZE"));  
            info.setNullable(crs.getBoolean("IS_NULLABLE"));  
            info.setDef(crs.getString("COLUMN_DEF"));  
            infos.add(info);  
        }  
    }  
    catch(SQLException ex) {  
        throw new RuntimeException(ex);  
    }  
    return infos;  
}
```

❶ 查詢 t_files
↓ 表格所有欄位

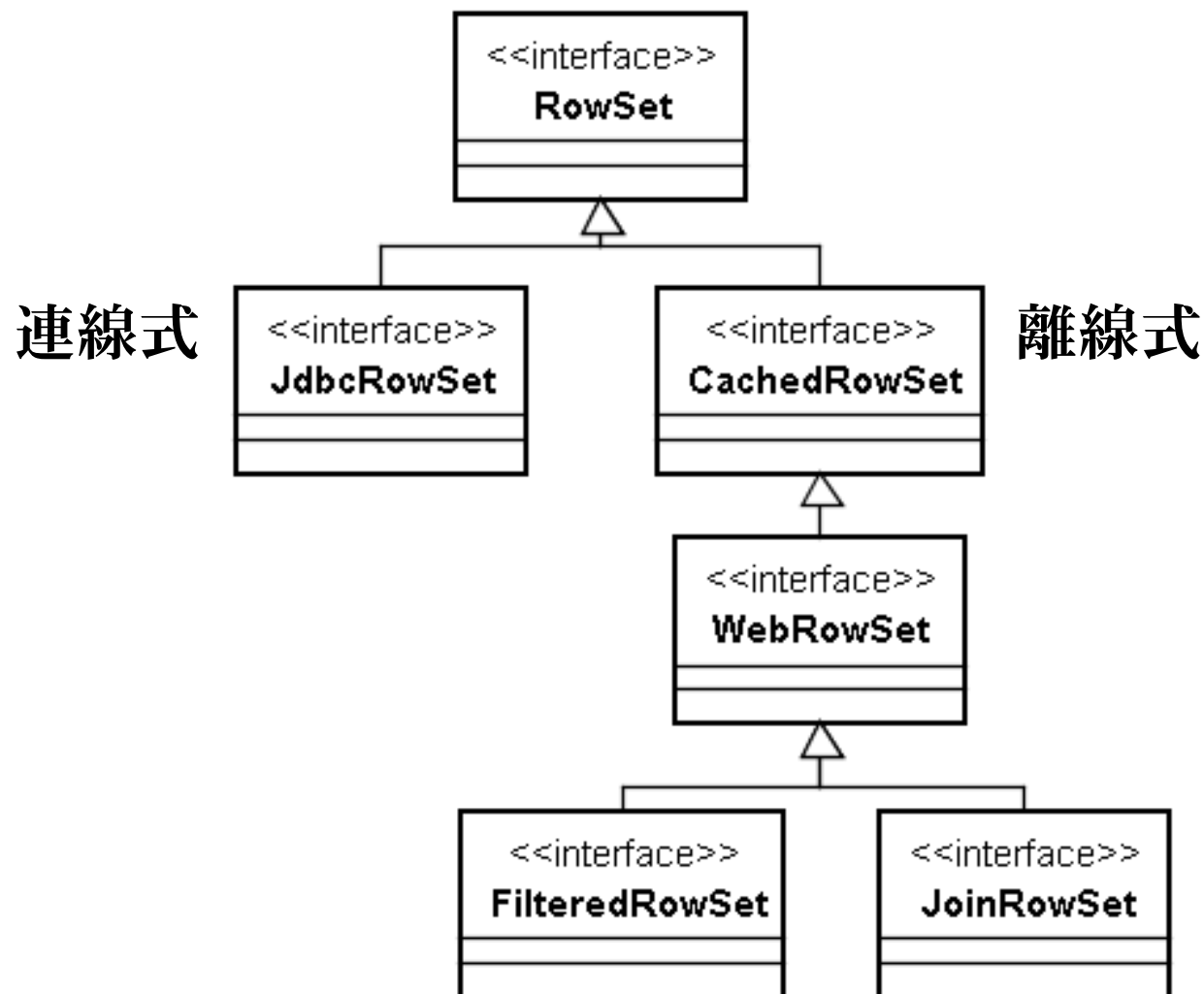
❷ 用來收集欄位資訊

❸ 封裝欄位名稱、型態、大小、可否為空、預設值等資訊

簡介 RowSet

- **javax.sql.RowSet** 代表資料的列集合
 - 這邊的資料並不一定是資料庫中的資料，可以是試算表資料、XML 資料或任何具有列集合概念的資料來源
- 是 **ResultSet** 的子介面，新增了一些行為，像是透過 **setCommand()** 設定查詢指令、透過 **execute()** 執行查詢指令以填充資料等

簡介 RowSet



簡介 RowSet

- JdbcRowSet 是連線式的 RowSet
 - 操作 JdbcRowSet 期間，會保持與資料庫的連線，可視為取得、操作 ResultSet 行為封裝，可簡化 JDBC 程式的撰寫，或作為 JavaBean 使用
- CachedRowSet 為離線式的 RowSet
 - 查詢並充填完資料後，就會斷開與資料來源的連線，而不用佔據相關連線資源，必要也可以再與資料來源連線進行資料同步

簡介 RowSet

- 若使用 Oracle/Sun JDK 附帶的 `JdbcRowSetImpl`，在 JDK6 之前，可以如下建
`JdbcRowSet rowset = new JdbcRowSetImpl();`
- 在 JDK7 之後，新增了 `javax.sql.rowset.RowSetFactory` 介面與 `javax.sql.rowset.RowSetProvider` 類別
`RowSetFactory rowSetFactory = RowSetProvider.newFactory();`
`JdbcRowSet rowset = rowSetFactory.createJdbcRowSet();`

簡介 RowSet

- 如果使用 Oracle/Sun JDK，以上程式片段會取得 `JdbcRowSetImpl` 實例
- 可以在啟動 JVM 時，利用系統屬性 "`javax.sql.rowset.RowSetFactory`" 指定其它廠商實作

簡介 RowSet

- 使用 RowSet 查詢資料

```
rowset.setUrl("jdbc:mysql://localhost:3306/demo");  
rowset.setUsername("root");  
rowset.setPassword("123456");  
rowset.setCommand("SELECT * FROM t_messages WHERE id = ?");  
rowset.setInt(1, 1);  
rowset.execute();
```

```
public class MessageDAO4 {
    private JdbcRowSet rowset;

    public MessageDAO4(
        String url, String user, String passwd) throws SQLException {
        JdbcRowSet rowset = RowSetProvider.newFactory().createJdbcRowSet();
        rowset.setUrl(url);
        rowset.setUsername(user);
        rowset.setPassword(passwd);
        rowset.setCommand("SELECT * FROM t_message");
        rowset.execute();
    }

    public void add(Message message) throws SQLException {
        rowset.moveToInsertRow();
        rowset.updateString(2, message.getName());
        rowset.updateString(3, message.getEmail());
        rowset.updateString(4, message.getMsg());
        rowset.insertRow();
    }
}
```



```
public List<Message> get() throws SQLException {
    List<Message> messages = new ArrayList<>();
    rowset.beforeFirst();
    while (rowset.next()) {
        Message message = new Message();
        message.setId(rowset.getLong(1));
        message.setName(rowset.getString(2));
        message.setEmail(rowset.getString(3));
        message.setMsg(rowset.getString(4));
        messages.add(message);
    }
    return messages;
}

public void close() throws SQLException {
    if (rowset != null) {
        rowset.close();
    }
}
}
```

簡介 RowSet

- 在查詢之後，想要離線進行操作，則可以使用 `CachedRowSet` 或其子介面實作物件
- 使用 **`close()`** 關閉 `CachedRowSet`，若在相關更新操作之後，想與再與資料來源進行同步，則可以呼叫 **`acceptChanges()`** 方法

```
conn.setAutoCommit(false); // conn 是 Connection
rowSet.acceptChanges(conn); // rowSet 是 CachedRowSet
conn.setAutoCommit(true);
```

簡介 RowSet

- **WebRowSet** 是 `CachedRowSet` 的子介面，其不僅具備離線操作，還能進行 XML 讀寫

```
public static void writeXml(OutputStream outputStream)
    throws Exception {
    try(WebRowSet rowset =
        RowSetProvider.newFactory().createWebRowSet()) {
        rowset.setUrl("jdbc:mysql://localhost:3306/demo");
        rowset.setUsername("root");
        rowset.setPassword("123456");
        rowset.setCommand("SELECT * FROM t_message");
        rowset.execute();
        rowset.writeXml(outputStream);
    }
}

public static void main(String[] args) throws Exception {
    TMessageUtil.writeXml(System.out);
}
```

簡介 RowSet

...略

```
<data>
  <currentRow>
    <columnValue>1</columnValue>
    <columnValue>良葛格</columnValue>
    <columnValue>caterpillar@openhome.cc</columnValue>
    <columnValue>這是一篇測試留言！</columnValue>
  </currentRow>
  <currentRow>
    <columnValue>2</columnValue>
    <columnValue>毛美眉</columnValue>
    <columnValue>momor@openhome.cc</columnValue>
    <columnValue>我來留言囉！</columnValue>
  </currentRow>
</data>
</webRowSet>
```

簡介 RowSet

- **FilteredRowSet** 可以對列集合進行過濾，實現類似 SQL 中 WHERE 等條件式的功能
- **JoinRowSet** 則可以讓你結合兩個 RowSet 物件，實現類似 SQL 中 JOIN 的功能
- <http://docs.oracle.com/javase/tutorial/jdbc/basics/gettingstarted.html>