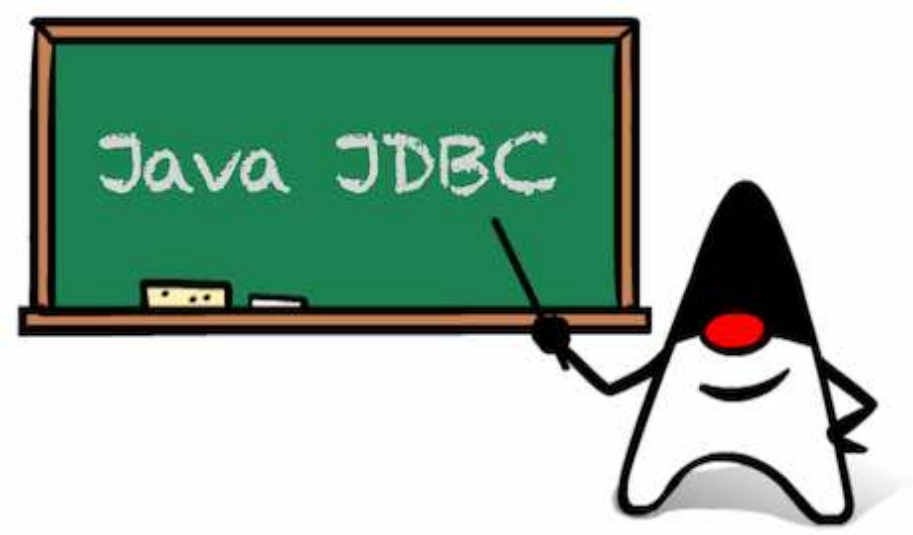
17 JDBC编程

程序运行的时候，往往需要存取数据。现代应用程序最基本，也是使用最广泛的数据存储就是关系数据库。

Java为关系数据库定义了一套标准的访问接口：JDBC（Java Database Connectivity），本章我们介绍如何在Java程序中使用JDBC。



JDBC简介

在介绍JDBC之前，我们先简单介绍一下关系数据库。

程序运行的时候，数据都是在内存中的。当程序终止的时候，通常都需要将数据保存到磁盘上，无论是保存到本地磁盘，还是通过网络保存到服务器上，最终都会将数据写入磁盘文件。

而如何定义数据的存储格式就是一个大问题。如果我们自己来定义存储格式，比如保存一个班级所有学生的成绩单：

名字	成绩
Michael	99
Bob	85
Bart	59
Lisa	87

你可以用一个文本文件保存，一行保存一个学生，用`,`隔开：

```
Michael,99
Bob,85
Bart,59
Lisa,87
```

你还可以用JSON格式保存，也是文本文件：

```
[
    {"name":"Michael","score":99},
    {"name":"Bob","score":85},
    {"name":"Bart","score":59},
    {"name":"Lisa","score":87}
]
```

你还可以定义各种保存格式，但是问题来了：

存储和读取需要自己实现，JSON还是标准，自己定义的格式就各式各样了；

不能做快速查询，只有把数据全部读到内存中才能自己遍历，但有时候数据的大小远远超过了内存（比如蓝光电影，40GB的数据），根本无法全部读入内存。

为了便于程序保存和读取数据，而且，能直接通过条件快速查询到指定的数据，就出现了数据库（Database）这种专门用于集中存储和查询的软件。

数据库软件诞生的历史非常久远，早在1950年数据库就诞生了。经历了网状数据库，层次数据库，我们现在广泛使用的关系数据库是20世纪70年代基于关系模型的基础上诞生的。

关系模型有一套复杂的数学理论，但是从概念上是十分容易理解的。举个学校的例子：

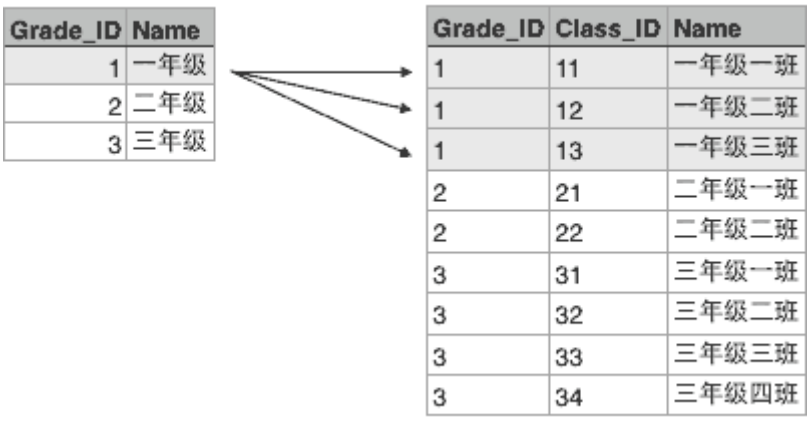
假设某个XX省YY市ZZ县第一实验小学有3个年级，要表示出这3个年级，可以在Excel中用一个表格画出来：

Grade_ID	Name
1	一年级
2	二年级
3	三年级

每个年级又有若干个班级，要把所有班级表示出来，可以在Excel中再画一个表格：

Grade_ID	Class_ID	Name
1	11	一年级一班
1	12	一年级二班
1	13	一年级三班
2	21	二年级一班
2	22	二年级二班
3	31	三年级一班
3	32	三年级二班
3	33	三年级三班
3	34	三年级四班

这两个表格有个映射关系，就是根据Grade_ID可以在班级表中查找到对应的所有班级：



Grade_ID	Name
1	一年级
2	二年级
3	三年级

Grade_ID	Class_ID	Name
1	11	一年级一班
1	12	一年级二班
1	13	一年级三班
2	21	二年级一班
2	22	二年级二班
3	31	三年级一班
3	32	三年级二班
3	33	三年级三班
3	34	三年级四班

也就是Grade表的每一行对应Class表的多行，在关系数据库中，这种基于表（Table）的一对多的关系就是关系数据库的基础。

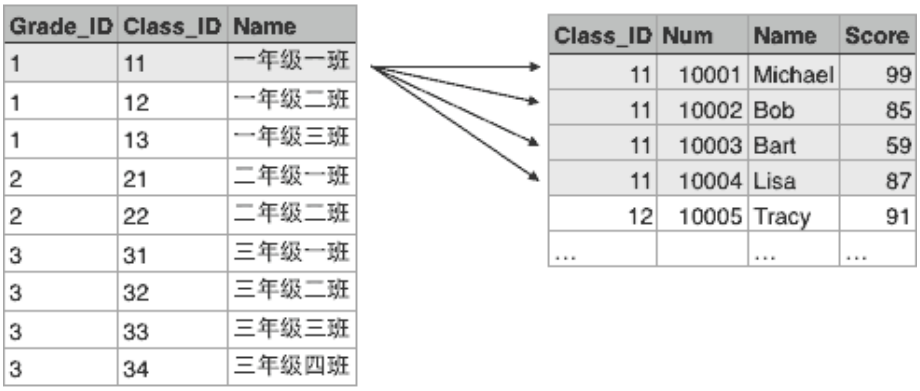
根据某个年级的ID就可以查找所有班级的行，这种查询语句在关系数据库中称为SQL语句，可以写成：

```
SELECT * FROM classes WHERE grade_id = '1';
```

结果也是一个表：

```
---------+----------+----------
grade_id | class_id | name
---------+----------+----------
1        | 11       | 一年级一班
---------+----------+----------
1        | 12       | 一年级二班
---------+----------+----------
1        | 13       | 一年级三班
---------+----------+----------
```

类似的，Class表的一行记录又可以关联到Student表的多行记录：



Grade_ID	Class_ID	Name
1	11	一年级一班
1	12	一年级二班
1	13	一年级三班
2	21	二年级一班
2	22	二年级二班
3	31	三年级一班
3	32	三年级二班
3	33	三年级三班
3	34	三年级四班

Class_ID	Num	Name	Score
11	10001	Michael	99
11	10002	Bob	85
11	10003	Bart	59
11	10004	Lisa	87
12	10005	Tracy	91
...		...	...

由于本教程不涉及到关系数据库的详细内容，如果你想从零学习关系数据库和基本的SQL语句，请参考SQL课程。

NoSQL

你也许还听说过NoSQL数据库，很多NoSQL宣传其速度和规模远远超过关系数据库，所以很多同学觉得有了NoSQL是否就不需要SQL了呢？千万不要被他们忽悠了，连SQL都不明白怎么可能搞明白NoSQL呢？

数据库类别

既然我们要使用关系数据库，就必须选择一个关系数据库。目前广泛使用的关系数据库也就这么几种：

付费的商用数据库：

- Oracle，典型的高富帅；
- SQL Server，微软自家产品，Windows定制专款；
- DB2，IBM的产品，听起来挺高端；
- Sybase，曾经跟微软是好基友，后来关系破裂，现在家境惨淡。

这些数据库都是不开源而且付费的，最大的好处是花了钱出了问题可以找厂家解决，不过在Web的世界里，常常需要部署成千上万的数据库服务器，当然不能把大把大把的银子扔给厂家，所以，无论是Google、Facebook，还是国内的BAT，无一例外都选择了免费的开源数据库：

- MySQL，大家都在用，一般错不了；
- PostgreSQL，学术气息有点重，其实挺不错，但知名度没有MySQL高；
- sqlite，嵌入式数据库，适合桌面和移动应用。

作为一个Java工程师，选择哪个免费数据库呢？当然是MySQL。因为MySQL普及率最高，出了错，可以很容易找到解决方法。而且，围绕MySQL有一大堆监控和运维的工具，安装和使用很方便。

安装MySQL

为了能继续后面的学习，你需要从MySQL官方网站下载并安装MySQL Community Server 5.6，这个版本是免费的，其他高级版本是要收钱的（请放心，收钱的功能我们用不上）。MySQL是跨平台的，选择对应的平台下载安装文件，安装即可。

安装时，MySQL会提示输入`root`用户的口令，请务必记清楚。如果怕记不住，就把口令设置为`password`。

在Windows上，安装时请选择`UTF-8`编码，以便正确地处理中文。

在Mac或Linux上，需要编辑MySQL的配置文件，把数据库默认的编码全部改为UTF-8。MySQL的配置文件默认存放在`/etc/my.cnf`或者`/etc/mysql/my.cnf`：

```
[client]
default-character-set = utf8

[mysqld]
default-storage-engine = INNODB
character-set-server = utf8
collation-server = utf8_general_ci
```

重启MySQL后，可以通过MySQL的客户端命令行检查编码：

```
$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor...
...

mysql> show variables like '%char%';
+--------------------------+------------------------------
--------------------------+
| Variable_name            | Value
                         |
+--------------------------+------------------------------
--------------------------+
| character_set_client     | utf8
                         |
| character_set_connection | utf8
                         |
| character_set_database   | utf8
                         |
| character_set_filesystem | binary
                         |
| character_set_results    | utf8
                         |
| character_set_server     | utf8
                         |
| character_set_system     | utf8
                         |
| character_sets_dir       | /usr/local/mysql-5.1.65-
osx10.6-x86_64/share/charsets/ |
+--------------------------+------------------------------
--------------------------+
8 rows in set (0.00 sec)
```

看到`utf8`字样就表示编码设置正确。

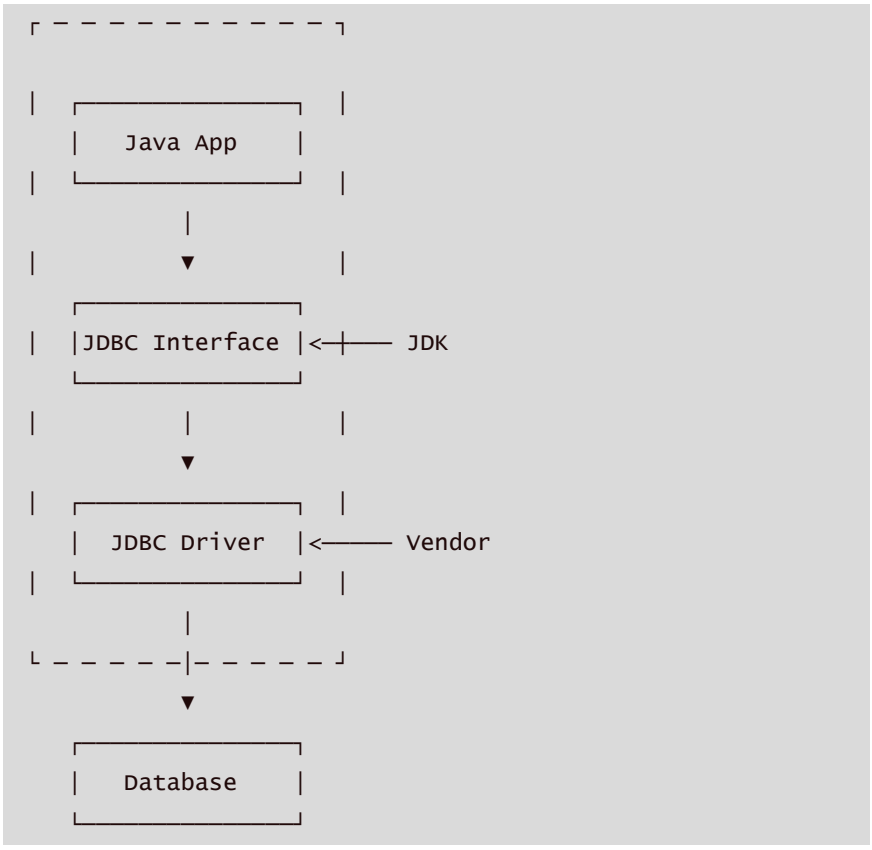
注：如果MySQL的版本≥5.5.3，可以把编码设置为`utf8mb4`，`utf8mb4`和`utf8`完全兼容，但它支持最新的Unicode标准，可以显示emoji字符。

JDBC

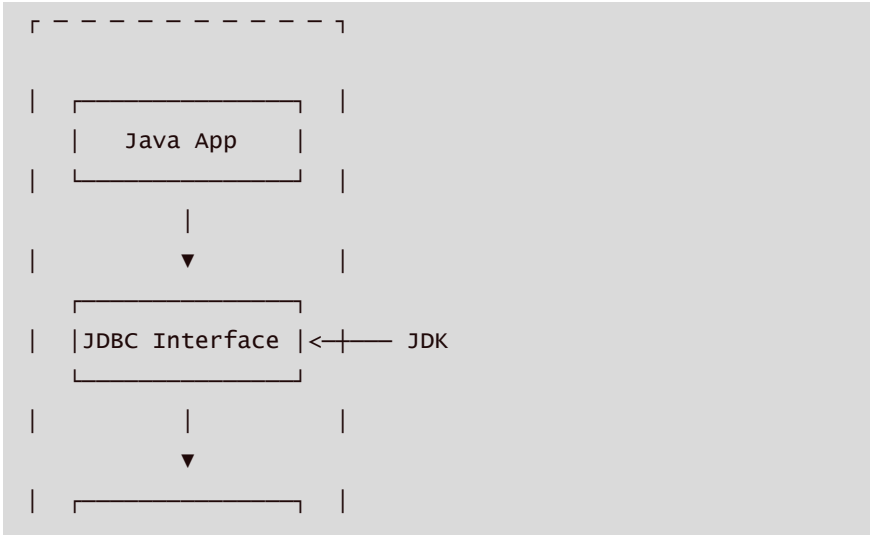
什么是JDBC？JDBC是Java DataBase Connectivity的缩写，它是Java程序访问数据库的标准接口。

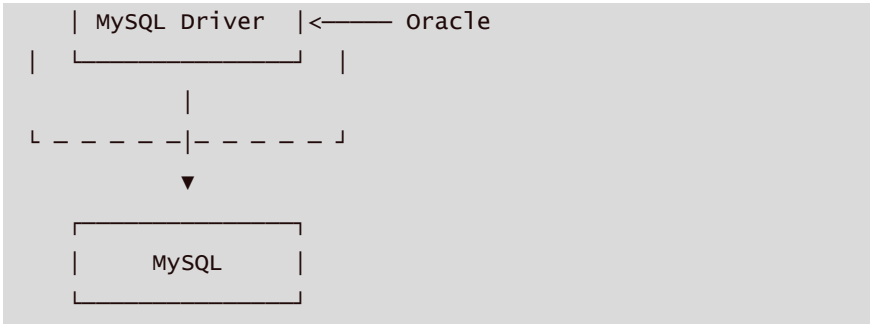
使用Java程序访问数据库时，Java代码并不是直接通过TCP连接去访问数据库，而是通过JDBC接口来访问，而JDBC接口则通过JDBC驱动来实现真正对数据库的访问。

例如，我们在Java代码中如果要访问MySQL，那么必须编写代码操作JDBC接口。注意到JDBC接口是Java标准库自带的，所以可以直接编译。而具体的JDBC驱动是由数据库厂商提供的，例如，MySQL的JDBC驱动由Oracle提供。因此，访问某个具体的数据库，我们只需要引入该厂商提供的JDBC驱动，就可以通过JDBC接口来访问，这样保证了Java程序编写的是一套数据库访问代码，却可以访问各种不同的数据库，因为他们都提供了标准的JDBC驱动：

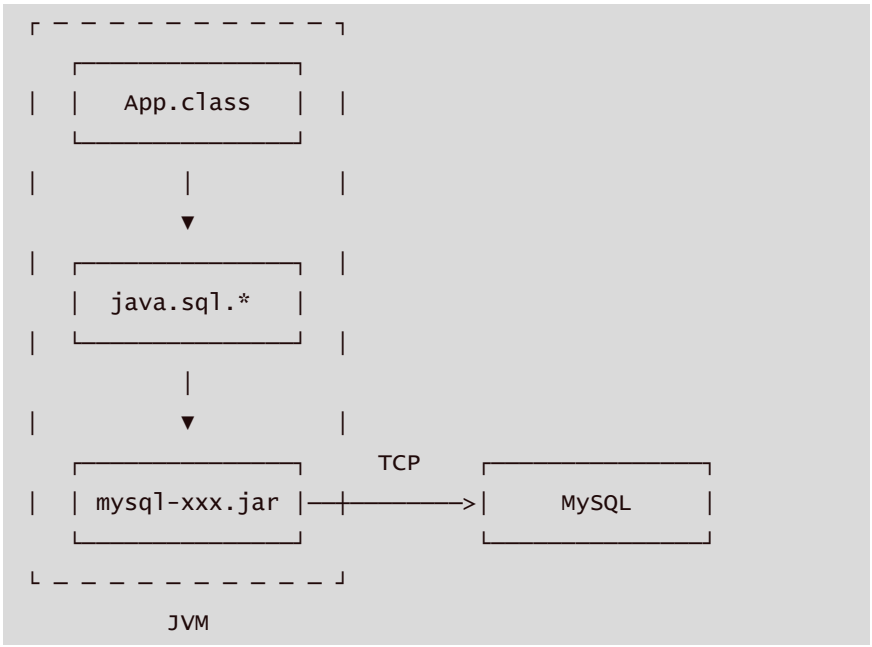


从代码来看，Java标准库自带的JDBC接口其实就是定义了一组接口，而某个具体的JDBC驱动其实就是实现了这些接口的类：





实际上，一个MySQL的JDBC的驱动就是一个jar包，它本身也是纯Java编写的。我们自己编写的代码只需要引用Java标准库提供的java.sql包下面的相关接口，由此再间接地通过MySQL驱动的jar包通过网络访问MySQL服务器，所有复杂的网络通讯都被封装到JDBC驱动中，因此，Java程序本身只需要引入一个MySQL驱动的jar包就可以正常访问MySQL服务器：



小结

使用JDBC的好处是：

- 各数据库厂商使用相同的接口，Java代码不需要针对不同数据库分别开发；
- Java程序编译期仅依赖java.sql包，不依赖具体数据库的jar包；
- 可随时替换底层数据库，访问数据库的Java代码基本不变。

JDBC查询

前面我们讲了Java程序要通过JDBC接口来查询数据库。JDBC是一套接口规范，它在哪呢？就在Java的标准库`java.sql`里放着，不过这里面大部分都是接口。接口并不能直接实例化，而是必须实例化对应的实现类，然后通过接口引用这个实例。那么问题来了：JDBC接口的实现类在哪？

因为JDBC接口并不知道我们要使用哪个数据库，所以，用哪个数据库，我们就去使用哪个数据库的“实现类”，我们把某个数据库实现了JDBC接口的jar包称为JDBC驱动。

因为我们选择了MySQL 5.x作为数据库，所以我们首先得找一个MySQL的JDBC驱动。所谓JDBC驱动，其实就是一个第三方jar包，我们直接添加一个Maven依赖就可以了：

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.47</version>
    <scope>runtime</scope>
</dependency>
```

注意到这里添加依赖的`scope`是`runtime`，因为编译Java程序并不需要MySQL的这个jar包，只有在运行期才需要使用。如果把`runtime`改成`compile`，虽然也能正常编译，但是在IDE里写程序的时候，会多出来一大堆类似`com.mysql.jdbc.Connection`这样的类，非常容易与Java标准库的JDBC接口混淆，所以坚决不要设置为`compile`。

有了驱动，我们还要确保MySQL在本机正常运行，并且还需要准备一点数据。这里我们用一个脚本创建数据库和表，然后插入一些数据：

```
-- 创建数据库learjdbc:
DROP DATABASE IF EXISTS learnjdbc;
CREATE DATABASE learnjdbc;

-- 创建登录用户learn/口令learnpassword
CREATE USER IF NOT EXISTS learn@'%' IDENTIFIED BY
'learnpassword';
GRANT ALL PRIVILEGES ON learnjdbc.* TO learn@'%' WITH
GRANT OPTION;
FLUSH PRIVILEGES;

-- 创建表students:
USE learnjdbc;
CREATE TABLE students (
  id BIGINT AUTO_INCREMENT NOT NULL,
  name VARCHAR(50) NOT NULL,
  gender TINYINT(1) NOT NULL,
  grade INT NOT NULL,
  score INT NOT NULL,
  PRIMARY KEY(id)
) Engine=INNODB DEFAULT CHARSET=UTF8;

-- 插入初始数据:
INSERT INTO students (name, gender, grade, score) VALUES
('小明', 1, 1, 88);
INSERT INTO students (name, gender, grade, score) VALUES
('小红', 1, 1, 95);
INSERT INTO students (name, gender, grade, score) VALUES
('小军', 0, 1, 93);
INSERT INTO students (name, gender, grade, score) VALUES
('小白', 0, 1, 100);
```

```
INSERT INTO students (name, gender, grade, score) VALUES
('小牛', 1, 2, 96);
INSERT INTO students (name, gender, grade, score) VALUES
('小兵', 1, 2, 99);
INSERT INTO students (name, gender, grade, score) VALUES
('小强', 0, 2, 86);
INSERT INTO students (name, gender, grade, score) VALUES
('小乔', 0, 2, 79);
INSERT INTO students (name, gender, grade, score) VALUES
('小青', 1, 3, 85);
INSERT INTO students (name, gender, grade, score) VALUES
('小王', 1, 3, 90);
INSERT INTO students (name, gender, grade, score) VALUES
('小林', 0, 3, 91);
INSERT INTO students (name, gender, grade, score) VALUES
('小贝', 0, 3, 77);
```

在控制台输入`mysql -u root -p`，输入`root`口令后以`root`身份，把上述SQL贴到控制台执行一遍就行。如果你运行的是最新版MySQL 8.x，需要调整一下`CREATE USER`语句。

JDBC连接

使用JDBC时，我们先了解什么是Connection。Connection代表一个JDBC连接，它相当于Java程序到数据库的连接（通常是TCP连接）。打开一个Connection时，需要准备URL、用户名和口令，才能成功连接到数据库。

URL是由数据库厂商指定的格式，例如，MySQL的URL是：

```
jdbc:mysql://<hostname>:<port>/<db>?
key1=value1&key2=value2
```

假设数据库运行在本机`localhost`，端口使用标准的`3306`，数据库名称是`learnjdbc`，那么URL如下：

```
jdbc:mysql://localhost:3306/learnjdbc?
useSSL=false&characterEncoding=utf8
```

后面的两个参数表示不使用SSL加密，使用UTF-8作为字符编码（注意MySQL的UTF-8是`utf8`）。

要获取数据库连接，使用如下代码：

```
// JDBC连接的URL, 不同数据库有不同的格式:
String JDBC_URL = "jdbc:mysql://localhost:3306/test";
String JDBC_USER = "root";
String JDBC_PASSWORD = "password";
// 获取连接:
Connection conn = DriverManager.getConnection(JDBC_URL,
JDBC_USER, JDBC_PASSWORD);
// TODO: 访问数据库...
// 关闭连接:
conn.close();
```

核心代码是`DriverManager`提供的静态方法`getConnection()`。`DriverManager`会自动扫描classpath，找到所有的JDBC驱动，然后根据我们传入的URL自动挑选一个合适的驱动。

因为JDBC连接是一种昂贵的资源，所以使用后要及时释放。使用`try (resource)`来自动释放JDBC连接是一个好方法：

```
try (Connection conn =
DriverManager.getConnection(JDBC_URL, JDBC_USER,
JDBC_PASSWORD)) {
    // ...
}
```

JDBC查询

获取到JDBC连接后，下一步我们就可以查询数据库了。查询数据库分以下几步：

第一步，通过`Connection`提供的`createStatement()`方法创建一个`Statement`对象，用于执行一个查询；

第二步，执行`Statement`对象提供的`executeQuery("SELECT * FROM students")`并传入SQL语句，执行查询并获得返回的结果集，使用`ResultSet`来引用这个结果集；

第三步，反复调用`ResultSet`的`next()`方法并读取每一行结果。

完整查询代码如下：

```
try (Connection conn =
DriverManager.getConnection(JDBC_URL, JDBC_USER,
JDBC_PASSWORD) {
    try (Statement stmt = conn.createStatement()) {
        try (ResultSet rs = stmt.executeQuery("SELECT id,
grade, name, gender FROM students WHERE gender=\'M\'")) {
            while (rs.next()) {
                long id = rs.getLong(1); // 注意：索引从1开
始
                long grade = rs.getLong(2);
                String name = rs.getString(3);
                String gender = rs.getString(4);
            }
        }
    }
}
```

注意要点：

`Statment`和`ResultSet`都是需要关闭的资源，因此嵌套使用`try (resource)`确保及时关闭；

`rs.next()`用于判断是否有下一行记录，如果有，将自动把当前行移动到下一行（一开始获得`ResultSet`时当前行不是第一行）；

`ResultSet`获取列时，索引从`1`开始而不是`0`；

必须根据`SELECT`的列的对应位置来调用`getLong(1)`，`getString(2)`这些方法，否则对应位置的数据类型不对，将报错。

SQL注入

使用`Statement`拼字符串非常容易引发SQL注入的问题，这是因为SQL参数往往是从方法参数传入的。

我们来看一个例子：假设用户登录的验证方法如下：

```
User login(String name, String pass) {
    // ...
    stmt.executeQuery("SELECT * FROM user WHERE login='" +
name + "' AND pass='" + pass + "'");
    // ...
}
```

其中，参数`name`和`pass`通常都是Web页面输入后由程序接收到的。

如果用户的输入是程序期待的值，就可以拼出正确的SQL。例如：name = `"bob"`，pass = `"1234"`：

```
SELECT * FROM user WHERE login='bob' AND pass='1234'
```

但是，如果用户的输入是一个精心构造的字符串，就可以拼出意想不到的SQL，这个SQL也是正确的，但它查询的条件不是程序设计的意图。例如：name = `"bob' OR pass="`, pass = `" OR pass='"`：

```
SELECT * FROM user WHERE login='bob' OR pass=' AND pass='
OR pass=''
```

这个SQL语句执行的时候，根本不用判断口令是否正确，这样一来，登录就形同虚设。

要避免SQL注入攻击，一个办法是针对所有字符串参数进行转义，但是转义很麻烦，而且需要在任何使用SQL的地方增加转义代码。

还有一个办法就是使用`PreparedStatement`。使用`PreparedStatement`可以*完全避免SQL注入*的问题，因为`PreparedStatement`始终使用`?`作为占位符，并且把数据连同SQL本身传给数据库，这样可以保证每次传给数据库的SQL语句是相同的，只是占位符的数据不同，还能高效利用数据库本身对查询的缓存。上述登录SQL如果用`PreparedStatement`可以改写如下：

```
User login(String name, String pass) {
    // ...
    String sql = "SELECT * FROM user WHERE login=? AND
pass=?";
    PreparedStatement ps = conn.prepareStatement(sql);
    ps.setObject(1, name);
    ps.setObject(2, pass);
    // ...
}
```

所以，`PreparedStatement`比`Statement`更安全，而且更快。

使用Java对数据库进行操作时，必须使用PreparedStatement，严禁任何通过参数拼字符串的代码！

我们把上面使用`Statement`的代码改为使用`PreparedStatement`：

```
try (Connection conn =
DriverManager.getConnection(JDBC_URL, JDBC_USER,
JDBC_PASSWORD) {
    try (PreparedStatement ps =
conn.prepareStatement("SELECT id, grade, name, gender FROM
students WHERE gender=? AND grade=?")) {
        ps.setObject(1, "M"); // 注意：索引从1开始
        ps.setObject(2, 3);
        try (ResultSet rs = stmt.executeQuery()) {
            while (rs.next()) {
                long id = rs.getLong("id");
                long grade = rs.getLong("grade");
                String name = rs.getString("name");
                String gender = rs.getString("gender");
            }
```

```
        }
    }
}
```

使用`PreparedStatement`和`Statement`稍有不同，必须首先调用`setObject()`设置每个占位符`?`的值，最后获取的仍然是`ResultSet`对象。

另外注意到从结果集读取列时，使用`String`类型的列名比索引要易读，而且不易出错。

注意到JDBC查询的返回值总是`ResultSet`，即使我们写这样的聚合查询`SELECT SUM(score) FROM ...`，也需要按结果集读取：

```
ResultSet rs = ...
if (rs.next()) {
    double sum = rs.getDouble(1);
}
```

数据类型

有的童鞋可能注意到了，使用JDBC的时候，我们需要在Java数据类型和SQL数据类型之间进行转换。JDBC在`java.sql.Types`定义了一组常量来表示如何映射SQL数据类型，但是平时我们使用的类型通常也就以下几种：

SQL数据类型	JAVA数据类型
BIT, BOOL	boolean
INTEGER	int
BIGINT	long
REAL	float
FLOAT, DOUBLE	double
CHAR, VARCHAR	String
DECIMAL	BigDecimal
DATE	java.sql.Date, LocalDate
TIME	java.sql.Time, LocalTime

注意：只有最新的JDBC驱动才支持`LocalDate`和`LocalTime`。

练习

下载练习：使用JDBC查询数据库 （推荐使用IDE练习插件快速下载）

小结

- JDBC接口的`Connection`代表一个JDBC连接；
- 使用JDBC查询时，总是使用`PreparedStatement`进行查询而不是`Statement`；
- 查询结果总是`ResultSet`，即使使用聚合查询也不例外。

JDBC更新

数据库操作总结起来就四个字：增删改查，行话叫CRUD：Create，Retreive，Update和Delete。

查就是查询，我们已经讲过了，就是使用`PreparedStatement`进行各种`SELECT`，然后处理结果集。现在我们来看看如何使用JDBC进行增删改。

插入

插入操作是`INSERT`，即插入一条新记录。通过JDBC进行插入，本质上也是用`PreparedStatement`执行一条SQL语句，不过最后执行的不是`executeQuery()`，而是`executeUpdate()`。示例代码如下：

```
try (Connection conn =
DriverManager.getConnection(JDBC_URL, JDBC_USER,
JDBC_PASSWORD) {
    try (PreparedStatement ps = conn.prepareStatement(
            "INSERT INTO students (id, grade, name,
gender) VALUES (?,?,?,?)")) {
        ps.setObject(1, 999); // 注意：索引从1开始
        ps.setObject(2, 1); // grade
        ps.setObject(3, "Bob"); // name
        ps.setObject(4, "M"); // gender
        int n = ps.executeUpdate(); // 1
    }
}
```

设置参数与查询是一样的，有几个`?`占位符就必须设置对应的参数。虽然`Statement`也可以执行插入操作，但我们仍然要严格遵循*绝不能手动拼SQL字符串*的原则，以避免安全漏洞。

当成功执行`executeUpdate()`后，返回值是`int`，表示插入的记录数量。此处总是`1`，因为只插入了一条记录。

插入并获取主键

如果数据库的表设置了自增主键，那么在执行`INSERT`语句时，并不需要指定主键，数据库会自动分配主键。对于使用自增主键的程序，有个额外的步骤，就是如何获取插入后的自增主键的值。

要获取自增主键，不能先插入，再查询。因为两条SQL执行期间可能有别的程序也插入了同一个表。获取自增主键的正确写法是在创建`PreparedStatement`的时候，指定一个`RETURN_GENERATED_KEYS`标志位，表示JDBC驱动必须返回插入的自增主键。示例代码如下：

```
try (Connection conn =
DriverManager.getConnection(JDBC_URL, JDBC_USER,
JDBC_PASSWORD) {
    try (PreparedStatement ps = conn.prepareStatement(
```

```
            "INSERT INTO students (grade, name, gender)
VALUES (?,?,?)",
            Statement.RETURN_GENERATED_KEYS)) {
        ps.setObject(1, 1); // grade
        ps.setObject(2, "Bob"); // name
        ps.setObject(3, "M"); // gender
        int n = ps.executeUpdate(); // 1
        try (ResultSet rs = ps.getGeneratedKeys()) {
            if (rs.next()) {
                long id = rs.getLong(1); // 注意：索引从1开
始
            }
        }
    }
}
```

观察上述代码，有两点注意事项：

一是调用`prepareStatement()`时，第二个参数必须传入常量`Statement.RETURN_GENERATED_KEYS`，否则JDBC驱动不会返回自增主键；

二是执行`executeUpdate()`方法后，必须调用`getGeneratedKeys()`获取一个`ResultSet`对象，这个对象包含了数据库自动生成的主键的值，读取该对象的每一行来获取自增主键的值。如果一次插入多条记录，那么这个`ResultSet`对象就会有多行返回值。如果插入时有多列自增，那么`ResultSet`对象的每一行都会对应多个自增值（自增列不一定必须是主键）。

更新

更新操作是`UPDATE`语句，它可以一次更新若干列的记录。更新操作和插入操作在JDBC代码的层面上实际上没有区别，除了SQL语句不同：

```
try (Connection conn =
DriverManager.getConnection(JDBC_URL, JDBC_USER,
JDBC_PASSWORD) {
    try (PreparedStatement ps =
conn.prepareStatement("UPDATE students SET name=? WHERE
id=?")) {
        ps.setObject(1, "Bob"); // 注意：索引从1开始
        ps.setObject(2, 999);
        int n = ps.executeUpdate(); // 返回更新的行数
    }
}
```

`executeUpdate()`返回数据库实际更新的行数。返回结果可能是正数，也可能是0（表示没有任何记录更新）。

删除

删除操作是`DELETE`语句，它可以一次删除若干列。和更新一样，除了SQL语句不同外，JDBC代码都是相同的：

```
try (Connection conn =
DriverManager.getConnection(JDBC_URL, JDBC_USER,
JDBC_PASSWORD) {
    try (PreparedStatement ps =
conn.prepareStatement("DELETE FROM students WHERE id=?"))
{
        ps.setObject(1, 999); // 注意：索引从1开始
        int n = ps.executeUpdate(); // 删除的行数
    }
}
```

练习

下载练习：使用JDBC更新数据库 （推荐使用IDE练习插件快速下载）

小结

- 使用JDBC执行`INSERT`、`UPDATE`和`DELETE`都可视为更新操作；
- 更新操作使用`PreparedStatement`的`executeUpdate()`进行，返回受影响的行数。

JDBC事务

数据库事务（Transaction）是由若干个SQL语句构成的一个操作序列，有点类似于Java的`synchronized`同步。数据库系统保证在一个事务中的所有SQL要么全部执行成功，要么全部不执行，即数据库事务具有ACID特性：

- Atomicity：原子性
- Consistency：一致性
- Isolation：隔离性
- Durability：持久性

数据库事务可以并发执行，而数据库系统从效率考虑，对事务定义了不同的隔离级别。SQL标准定义了4种隔离级别，分别对应可能出现的数据不一致的情况：

ISOLATION LEVEL	脏读（DIRTY READ）	不可重复读（NON REPEATABLE READ）	幻读（PHANTOM READ）
Read Uncommitted	Yes	Yes	Yes
Read Committed	-	Yes	Yes
Repeatable Read	-	-	Yes
Serializable	-	-	-

对应用程序来说，数据库事务非常重要，很多运行着关键任务的应用程序，都必须依赖数据库事务保证程序的结果正常。

举个例子：假设小明准备给小红支付100，两人在数据库中的记录主键分别是`123`和`456`，那么用两条SQL语句操作如下：

```
UPDATE accounts SET balance = balance - 100 WHERE id=123
AND balance >= 100;
UPDATE accounts SET balance = balance + 100 WHERE id=456;
```

这两条语句必须以事务方式执行才能保证业务的正确性，因为一旦第一条SQL执行成功而第二条SQL失败的话，系统的钱就会凭空减少100，而有了事务，要么这笔转账成功，要么转账失败，双方账户的钱都不变。

这里我们不讨论详细的SQL事务，如果对SQL事务不熟悉，请参考SQL事务。

要在JDBC中执行事务，本质上就是如何把多条SQL包裹在一个数据库事务中执行。我们来看JDBC的事务代码：

```
Connection conn = openConnection();
try {
    // 关闭自动提交:
    conn.setAutoCommit(false);
    // 执行多条SQL语句:
    insert(); update(); delete();
    // 提交事务:
    conn.commit();
} catch (SQLException e) {
    // 回滚事务:
    conn.rollback();
} finally {
    conn.setAutoCommit(true);
    conn.close();
}
```

其中，开启事务的关键代码是`conn.setAutoCommit(false)`，表示关闭自动提交。提交事务的代码在执行完指定的若干条SQL语句后，调用`conn.commit()`。要注意事务不是总能成功，如果事务提交失败，会抛出SQL异常（也可能在执行SQL语句的时候就抛出了），此时我们必须捕获并调用`conn.rollback()`回滚事务。最后，在`finally`中通过`conn.setAutoCommit(true)`把`Connection`对象的状态恢复到初始值。

实际上，默认情况下，我们获取到`Connection`连接后，总是处于“自动提交”模式，也就是每执行一条SQL都是作为事务自动执行的，这也是为什么前面几节我们的更新操作总能成功的原因：因为默认有这种“隐式事务”。只要关闭了`Connection`的`autoCommit`，那么就可以在一个事务中执行多条语句，事务以`commit()`方法结束。

如果要设定事务的隔离级别，可以使用如下代码：

```
// 设定隔离级别为READ COMMITTED:
conn.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
```

如果没有调用上述方法，那么会使用数据库的默认隔离级别。MySQL的默认隔离级别是`REPEATABLE READ`。

练习

下载练习：使用数据库事务 （推荐使用IDE练习插件快速下载）

小结

数据库事务（Transaction）具有ACID特性：

- Atomicity：原子性
- Consistency：一致性
- Isolation：隔离性
- Durability：持久性

JDBC提供了事务的支持，使用Connection可以开启、提交或回滚事务。

JDBC Batch

使用JDBC操作数据库的时候，经常会执行一些批量操作。

例如，一次性给会员增加可用优惠券若干，我们可以执行以下SQL代码：

```
INSERT INTO coupons (user_id, type, expires) VALUES (123, 'DISCOUNT', '2030-12-31');
INSERT INTO coupons (user_id, type, expires) VALUES (234, 'DISCOUNT', '2030-12-31');
INSERT INTO coupons (user_id, type, expires) VALUES (345, 'DISCOUNT', '2030-12-31');
INSERT INTO coupons (user_id, type, expires) VALUES (456, 'DISCOUNT', '2030-12-31');
...
```

实际上执行JDBC时，因为只有占位符参数不同，所以SQL实际上是一样的：

```
for (var params : paramsList) {
    PreparedStatement ps = conn.preparedStatement("INSERT INTO coupons (user_id, type, expires) VALUES (?,?,?)");
    ps.setLong(params.get(0));
    ps.setString(params.get(1));
    ps.setString(params.get(2));
    ps.executeUpdate();
}
```

类似的还有，给每个员工薪水增加10%～30%：

```
UPDATE employees SET salary = salary * ? WHERE id = ?
```

通过一个循环来执行每个`PreparedStatement`虽然可行，但是性能很低。SQL数据库对SQL语句相同，但只有参数不同的若干语句可以作为batch执行，即批量执行，这种操作有特别优化，速度远远快于循环执行每个SQL。

在JDBC代码中，我们可以利用SQL数据库的这一特性，把同一个SQL但参数不同的若干次操作合并为一个batch执行。我们以批量插入为例，示例代码如下：

```
try (PreparedStatement ps = conn.prepareStatement("INSERT
INTO students (name, gender, grade, score) VALUES (?, ?,
?, ?)")) {
    // 对同一个PreparedStatement反复设置参数并调用addBatch():
    for (String name : names) {
        ps.setString(1, name);
        ps.setBoolean(2, gender);
        ps.setInt(3, grade);
        ps.setInt(4, score);
        ps.addBatch(); // 添加到batch
    }
    // 执行batch:
    int[] ns = ps.executeBatch();
    for (int n : ns) {
        System.out.println(n + " inserted."); // batch中每
个SQL执行的结果数量
    }
}
```

执行batch和执行一个SQL不同点在于，需要对同一个`PreparedStatement`反复设置参数并调用`addBatch()`，这样就相当于给一个SQL加上了多组参数，相当于变成了“多行”SQL。

第二个不同点是调用的不是`executeUpdate()`，而是`executeBatch()`，因为我们设置了多组参数，相应地，返回结果也是多个`int`值，因此返回类型是`int[]`，循环`int[]`数组即可获取每组参数执行后影响的结果数量。

练习

使用Batch操作

小结

- 使用JDBC的batch操作会大大提高执行效率，对内容相同，参数不同的SQL，要优先考虑batch操作。

JDBC连接池

我们在讲多线程的时候说过，创建线程是一个昂贵的操作，如果有大量的小任务需要执行，并且频繁地创建和销毁线程，实际上会消耗大量的系统资源，往往创建和消耗线程所耗费的时间比执行任务的时间还长，所以，为了提高效率，可以用线程池。

类似的，在执行JDBC的增删改查的操作时，如果每一次操作都来一次打开连接，操作，关闭连接，那么创建和销毁JDBC连接的开销就太大了。为了避免频繁地创建和销毁JDBC连接，我们可以通过连接池（Connection Pool）复用已经创建好的连接。

JDBC连接池有一个标准的接口`javax.sql.DataSource`，注意这个类位于Java标准库中，但仅仅是接口。要使用JDBC连接池，我们必须选择一个JDBC连接池的实现。常用的JDBC连接池有：

- HikariCP
- C3P0
- BoneCP
- Druid

目前使用最广泛的是HikariCP。我们以HikariCP为例，要使用JDBC连接池，先添加HikariCP的依赖如下：

```
<dependency>
    <groupId>com.zaxxer</groupId>
    <artifactId>HikariCP</artifactId>
    <version>2.7.1</version>
</dependency>
```

紧接着，我们需要创建一个`DataSource`实例，这个实例就是连接池：

```
HikariConfig config = new HikariConfig();
config.setJdbcUrl("jdbc:mysql://localhost:3306/test");
config.setUsername("root");
config.setPassword("password");
config.addDataSourceProperty("connectionTimeout", "1000");
// 连接超时：1秒
config.addDataSourceProperty("idleTimeout", "60000"); //
空闲超时：60秒
config.addDataSourceProperty("maximumPoolSize", "10"); //
最大连接数：10
DataSource ds = new HikariDataSource(config);
```

注意创建`DataSource`也是一个非常昂贵的操作，所以通常`DataSource`实例总是作为一个全局变量存储，并贯穿整个应用程序的生命周期。

有了连接池以后，我们如何使用它呢？和前面的代码类似，只是获取`Connection`时，把`DriverManage.getConnection()`改为`ds.getConnection()`：

```
try (Connection conn = ds.getConnection()) { // 在此获取连接
    // ...
} // 在此“关闭”连接
```

通过连接池获取连接时，并不需要指定JDBC的相关URL、用户名、口令等信息，因为这些信息已经存储在连接池内部了（创建`HikariDataSource`时传入的`HikariConfig`持有这些信息）。一开始，连接池内部并没有连接，所以，第一次调用`ds.getConnection()`，会迫使连接池内部先创建一个`Connection`，再返回给客户端使用。当我们调用`conn.close()`方法时（在`try(resource){...}`结束处），不是真正“关闭”连接，而是释放到连接池中，以便下次获取连接时能直接返回。

因此，连接池内部维护了若干个`Connection`实例，如果调用`ds.getConnection()`，就选择一个空闲连接，并标记它为“正在使用”然后返回，如果对`Connection`调用`close()`，那么就把连接再次标记为“空闲”从而等待下次调用。这样一来，我们就通过连接池维护了少量连接，但可以频繁地执行大量的SQL语句。

通常连接池提供了大量的参数可以配置，例如，维护的最小、最大活动连接数，指定一个连接在空闲一段时间后自动关闭等，需要根据应用程序的负载合理地配置这些参数。此外，大多数连接池都提供了详细的实时状态以便进行监控。

练习

下载练习：使用HikariCP连接池 （推荐使用IDE练习插件快速下载）

小结

- 数据库连接池是一种复用`Connection`的组件，它可以避免反复创建新连接，提高JDBC代码的运行效率；
- 可以配置连接池的详细参数并监控连接池。