数据库连接

在本活动中，您将学习使用 Java 和 JDBC 类连接到 SQL 数据库。JDBC（Java 数据库连接）是一种低级的、不是特别面向对象的数据库访问机制，可以在其之上构建其他系统（例如 Hibernate ORM）。

JDBC 接口

JDBC 类位于 `java.sql` 包中。这些类上的大多数方法都可以抛出一个 `SQLException` 受检查的异常，这意味着如果您不处理它，您的代码将无法编译。对于简单的程序，这意味着 `RuntimeException` 如果出现问题，请包装在 a 中以终止程序：

```
try {
    // do stuff with JDBC
} catch (SQLException e) {
    throw new RuntimeException(e);
}
```

对此模式的两个评论：

1. 在真实的程序（例如 Web 服务器）中，您当然不希望每当单个方法导致错误时就关闭服务器。在这里，您需要执行一些操作，例如记录错误，并向特定调用者显示错误消息（例如 HTTP 500 内部服务器错误），同时保持服务器的其余部分运行。如何实现这一目标取决于您使用的库或框架。
2. 大多数 JDBC 工作实际上发生在 try-with-resources 块中，就异常处理而言，这些块的工作方式相同，但在 `try` .

尝试资源

资源是您在使用完它后需要关闭一次的东西，但前提是它首先被正确打开。例如，在 C 中，堆内存是一种资源：您可以使用 获取（打开）它 `malloc` ，完成后您必须 `free` 在内存上调用一次，除非 `malloc` 首先返回 NULL（分配失败），在这种情况下调用 `free` 是一个错误。（您确实检查了 `malloc` 返回值是否为 NULL，不是吗？） `free` 在同一内存上调用两次也是一个错误，并且根本不调用它会导致内存泄漏。

在Java中，我们不必手动管理内存，但还有其他种类的资源：

- 文件。

- 网络连接。
- 某些绘图/窗口系统中的图形对象。
- 数据库连接。

为了帮助管理这些，Java 提供了一个 `java.lang.AutoCloseable` 具有单一方法 `void close()` 和 try-with-resources 构造的接口：

```
try (Resource r = ...) {
    // do things with r here
}
```

只要资源实现 `AutoCloseable` （否则使用此模式是语法错误），此模式就保证

1. 如果初始化语句（在圆括号中）失败（返回 null 或抛出异常），则该块将永远不会被执行。
2. 如果初始化成功，那么当块退出时， `r.close()` 无论块到达末尾、提前退出（例如 return 语句）还是引发异常，都将被调用一次。

try-with-resources 块可以（但不一定）包含一个或多个 `catch` 语句，在这种情况下，它们适用于该块、初始化语句和隐含的 `close()` .

您还可以在 try 语句中包含多个资源，方法是在括号内的术语内用分号分隔它们。

Advanced note

早期版本的 java 使用 `finally` 关键字来实现类似的功能，但正确执行起来更具挑战性，特别是当 close 函数也可能抛出异常时。从 Java 7 开始，try-with-resources 是正确的使用模式，您几乎永远不需要块 `finally` 。 `AutoCloseable` 您可以在自己的类上实现来支持这一点。

打开连接

您可以通过调用打开连接

```
try(Connection c = DriverManager.getConnection(connection_string)) {
    // do stuff with connection
} catch (SQLException e) {
    // handle exception, for example by wrapping in RuntimeException
}
```

连接字符串是包含 URL 的字符串，例如

```
jdbc:mariadb://localhost:3306/DATABASE?
user=USER&localSocket=/var/run/mysqld/mysqld.sock
```

当您尝试打开连接时，Java 在类路径上查找实现 `mariadb` 您所请求的寻址方案（例如 ）的驱动程序。这使得设置类路径有点棘手，但我们有 Maven 来帮我们管理它。

然而，我们需要了解一些有关网络和安全的知识才能理解该 URL。

在传统的数据库设置中，数据库位于其自己的计算机（或计算机集群）上，应用程序通过网络连接到它。为此，默认情况下，数据库侦听 TCP 端口 3306。

出于安全原因，称职的管理员会进行设置，以便有一个防火墙阻止除应用程序（可能还包括开发人员和管理员）之外的任何计算机访问数据库，数据库计算机肯定无法直接从互联网访问。然后，应用程序还需要用户名和密码才能连接到数据库。由于这些密码是由应用程序使用的，不需要人类记住，因此绝对没有理由选择弱密码：绝对最小值是具有 128 位熵的密码。计算机记住这么长时间的东西没有任何问题！在这种情况下，您可以将额外的 `pass=` 参数添加到连接字符串中，并且为了防止密码通过网络（即使是您的内部网络）以未加密的方式发送，您还可以设置 TLS 或类似的隧道技术。

Advanced note

学习如何保护事物需要时间，但执行长密码和默认加密等操作只是一个开始。如果你想变得聪明，你可以进行行为检查，这样如果有人开始做他们通常不做的事情，就会触发日志。当您的数据库受到攻击时，您将如何发现？您如何判断它何时受到攻击但尚未被闯入？在有人过来对你大喊大叫之前，你如何让它恢复正常运行？

虽然聪明的东西都很有趣而且很好，但其中*很多*最终都归结为老式的系统管理。定期备份所有内容。注意发生了什么变化。尽快将您的日志从计算机上删除，以免它们被篡改。定期轮换您的密钥，因为您可以为其编写 shell 脚本，同时它可以使任何事情变得更加安全，并且会让合规人员感到高兴。

有关管理计算机的更完整指南，请参阅 Michael W Lucas 的任何书籍。

在我们的虚拟机上，当您默认设置 mariadb 时，数据库服务器和客户端都在同一台计算机上运行，因此您可以通过根本不使用网络连接来获得安全性和性能 - 相反，当您键入时， `mysql` 它会通过网络连接进行连接POSIX 套接字，另一种特殊类型的文件（键入 `s` ） `ls -l` ，在本例中为 `/var/run/mysqld/mysqld.sock` 。POSIX 套接字就像一对管道，它允许不同进程之间进行双向并发通信，但其 API 更接近于网络 (TCP) 套接字。

所有这些讨论的要点是，对于您的虚拟机，您的连接字符串将如下所示（全部在一行上，没有换行符）：

```
jdbc:mariadb://localhost:3306/DATABASE?
user=USER&localSocket=/var/run/mysqld/mysqld.sock
```

该 `localSocket` 选项会覆盖开始时的主机/端口。为此，您需要 mariadb 驱动程序和类路径上名为 JNA（Java Native Access）的库，当然您的系统需要支持套接字。

TCP 连接的更标准的连接字符串如下所示：

```
jdbc:mariadb://localhost:3306/DATABASE?user=USER
```

它确实连接到本地主机上的 TCP 端口 3306。

Advanced note

如果您愿意，可以在虚拟机上进行配置：主 mariadb 配置文件 `/etc/my.cnf` 在我们的示例中仅包含一条包含 `/etc/my.cnf.d/` 文件夹中所有文件的语句；在那里我们有 `mariadb-server.cnf` 包含行

```
[mysqld]
skip-networking
```

删除最后一行并重新启动服务器（ `systemctl restart mariadb` ），然后您的 mariadb 服务器（ `mysqld` ）将真正监听端口 3306。

我们在控制台客户端中没有注意到这一点 `mysql` ，因为默认情况下，它会首先尝试套接字，如果套接字不存在，则再尝试端口 3306。然而，JDBC 驱动程序只会尝试您提供的选项，如果您不告诉它使用套接字，它会尝试端口 3306，如果没有任何监听，它会抛出异常。

POM文件

在本单元的存储库中， `code/jdbc/` 您可以找到使用 `elections` 数据库的最小 JDBC 应用程序。将其下载到您的虚拟机 `wget` （或者从单元存储库中获取它，如果您已将其克隆到那里）并将其解压到一个空文件夹（ `tar -xvf jdbc-example.tar` ）。它包含一个文件 `pom.xml` 和一个文件 `src/main/java/org/example/Example.java` 。

在 POM 文件中，我们注意到以下依赖关系：

```
<dependencies>
    <dependency>
        <groupId>org.mariadb.jdbc</groupId>
        <artifactId>mariadb-java-client</artifactId>
        <version>2.7.1</version>
    </dependency>
    <dependency>
        <groupId>net.java.dev.jna</groupId>
        <artifactId>jna</artifactId>
        <version>5.6.0</version>
    </dependency>
    <dependency>
        <groupId>net.java.dev.jna</groupId>
        <artifactId>jna-platform</artifactId>
        <version>5.6.0</version>
    </dependency>
</dependencies>
```

第一个是 mariadb JDBC 驱动程序。maven运行程序时，会自动将依赖项放到classpath中；如果您将其关闭，那么创建 `Connection` 将引发异常。

另外两个是适用于您的平台的 JNA（Java Native Access）库，驱动程序使用它们连接到 POSIX 套接字。如果您关闭这些，驱动程序将忽略该 `localSocket` 选项，尝试连接到端口 3306，并抛出异常，因为那里没有任何监听（除非您已配置）。

- `mvn compile` 在POM文件所在的文件夹中运行以下载依赖项并编译示例程序。
- 运行 `mvn exec:java` 来运行程序。这将使用稍后在 POM 文件中配置的内容来启动具有主类和正确类路径的 `exec-maven-plugin` 程序。 `org.example.Example` 它应该打印出各方名单。
- 如果需要，您可以使用 `mvn package` 在 中构建两个 JAR 文件 `target`： 一个只是已编译的示例类，但更有趣的一个 `jar-with-dependencies` 在其名称中包含已编译的类和所有依赖项，即 JDBC mariadb 驱动程序和 JNA（以及所有他们的依赖）。 `java -cp jdbc-example-0.1-jar-with-dependencies.jar org.example.Example` 如果您愿意，您可以在不使用 Maven 的情况下运行这个 jar 。该文件是由 `maven-assembly-plugin` 我们在 POM 文件中配置的构建的。

来自 Java 的 SQL

在 `Example.java` 课程中，我们可以看到 JDBC 的实际应用示例：

```
private void readData(Connection c) {
    String SQL = "SELECT id, name FROM Party";
    try (PreparedStatement s = c.prepareStatement(SQL)) {
        ResultSet r = s.executeQuery();
        while (r.next()) {
            int id = r.getInt("id");
            String name = r.getString("name");
            System.out.println("Party #" + id + " is: " + name);
        }
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
```

- SQL 命令以字符串形式出现。如果我们想为准备好的语句添加参数，我们在这里打上问号。
- 我们在另一个 try-with-resources 块中创建一个 `PreparedStatement`： 我们需要在完成语句后立即关闭它们，但是程序只要运行就可以保持数据库连接打开。
- 在这种情况下，我们没有要传递的参数，因此我们执行查询来获取 `ResultSet` 结果上的“游标”（一种迭代器）。（A `ResultSet` 也是一个资源，但是当它的语句关闭时它会自动关闭，所以我们不必自己处理这个。）
- 我们迭代结果的行并对它们执行一些操作，在本例中打印到标准输出。
- 所有这些操作都可以抛出一个 `SQLException` ，所以我们在最后捕获它，并且因为这只是一个小示例程序，所以我们通过抛出异常来终止整个程序来处理它。

结果集

Java 是一种面向对象的语言，具有编译时类型系统：如果您声明 a `class Person` 有一个字段 `int age` ，那么编译器将阻止您尝试将字符串放入其中。JDBC 无法为您提供这种级别的保护，因为当您编译 Java 程序时，您不知道数据库中存在哪些表和字段（即使您知道，有人也可以在程序编译后更改它们）。因此，您必须退回到一些更类似于 C 的模式来使用数据库。

结果集可以指向数据库中的行，也可以不指向数据库中的行。如果它指向一行，您可以使用 `get...` 方法读取值。如果结果集未指向行，则尝试读取任何内容都会引发 SQLException。这里的规则是：

- 当您返回结果集时，它开始指向*第一行之前*，因此立即读取会引发错误。
- 跟注 `boolean next()` 试图推进一行。如果返回 true，那么您就有了一个新行并且可以从中读取；如果你得到错误，那么你已经超出了最后一行，这将是一个阅读错误。（如果结果中根本没有行，则第一次调用 `next()` 将返回 false。）

使用结果集的正确模式通常是 while 循环，因为每次进入循环体时，都保证找到一行：

```
while (r.next()) {
    // we have a row, do something with it
}
```

然而，这种模式有一些例外。首先，像这样的一些语句 `select count(...)` 将始终返回一行 - 也许该行中的值为零，但这与根本没有行不同 - 所以在这种情况下你可以这样做

```
if (r.next()) {
    // this should always happen
} else {
    // this should never happen
    // throw an exception or something
}
```

在调用之前访问一行仍然是一个错误 `next()` ，而且我们不是那种忽略 API 调用返回值的人。

另一个特殊情况是如果您想对第一行做一些特殊的事情：

```
if (r.next()) {
    // if we get here then there was at least one row
    // we're on the first row and can do something special with it
    do {
        // this block will be called exactly once for every row
        // including the first one
    } while (r.next())
} else {
    // if we get here then there were no rows at all
}
```

该 `do-while` 循环允许我们编写一个为每一行调用一次的块，同时仍然允许我们对第一行执行一些特殊的操作，而不需要丑陋的 `boolean isFirstRow` 标志或类似的东西。

在结果集中，只要我们确定位于一行，就可以通过声明列的名称和类型来读取列中的值：

```
int id = r.getInt("id");
```

这告诉 JDBC 有一个名为 `id` type 的列 `int` ，并获取其值。（如果名称或类型错误，则会出现异常。）其他方法包括等 `getString` 。 `getDouble`

因此，在 SQL 语句中，您需要清楚要获取的列的名称和顺序：

- 如果列比字段更复杂，则给它一个别名，例如，那么 `SELECT COUNT(1) AS c` 您可以这样做 `getInt("c")` 。
- 切勿 `SELECT *` 从 JDBC 执行此操作，始终准确列出您需要的列。这既可以修复您获取它们的顺序，而且如果您不需要所有它们，效率会更高。

练习1

修改示例程序，使其采用参与方 ID 作为参数，并仅显示该参与方的名称或字符串“No party with this ID”。如果数据库中没有。

为此，您必须更改以下内容：

- `main` 读取参数 off `args` ，或者如果未传递参数则打印错误消息。
- `main` 将参数作为额外的 int 参数传递给 `readData` .
- 在准备好的语句中将参数设置为问号，然后将参数绑定到语句中。

绑定参数的命令是 `s.setInt(pos, value)` where `pos` 是参数（问号）在字符串中的索引，*从 1 而不是 0 开始计数*。所以你只是想要 `s.setInt(1, value)` 。当然还有 `setString` 等等，这些方法在声明上采用适当类型的第二个参数。

使用参数运行命令的最简单方法是构建一个具有依赖项的 jar，然后在 `java -cp JARFILE MAINCLASS PARAMETERS` 主类作为参数传递给此类后，调用 Java 的任何其他命令行参数。

练习2

服务是一段可以由其他代码调用的代码，通常可以为它们访问资源。本练习是关于编写一个 `DataService` 抽象程序来抽象程序其余部分的 JDBC 访问。

在自己的文件中创建以下类（如果您愿意，可以将私有字段与公共构造函数/getters/setters一起使用）：

```
public class Party {
    public int id;
    public String name;
}

public class Ward {
    public int id;
    public String name;
    public int electorate;
}

public class Candidate {
    public int id;
    public String name;
    public Party party;
    public Ward ward;
    public int votes;
}
```

`DataService` 现在，编写一个具有以下描述的类：

- DataService 实现 `AutoCloseable` . 它有一个公共构造函数，它接受一个连接字符串并 `Connection` 使用该字符串创建一个，并将其存储在私有字段中。该 `close()` 方法关闭连接。
- 一种 `public List<Party> getParties()` 通过在提供的连接上使用 JDBC 返回数据库中所有各方的列表的方法。
- `public Party getParty(int id)` 一种返回此 id 的参与方（如果有）的方法，否则返回 null。

这些方法应该处理所有可能的情况（例如， `getWards` 如果数据库中没有病房，则必须仍然有效）。但他们不应该抛出 SQLException。相反，创建您自己的异常类，称为派生类 `DataServiceException` （如果您愿意， `RuntimeException` 这可以是内部类 `DataService` ）并将其包装 `SQLException` 在其中。

顺便说一句，这并不是马虎程序员忽略异常的借口！

现在，调整 `Example` 程序，以便

- 主程序使用 `DataService` 和 域类（例如 `Party` ）并且不直接了解 JDBC。
- 如果您传递一个 id 作为参数，它会获取具有该 id 的政党（如果有的话）并显示实例中的政党信息 `Party` 。
- 如果您不传递 id，则会显示所有各方的列表。

Advanced note

该类 `DataService` 是一种资源，它在其构造函数中打开一个连接，并在您关闭实例时关闭它。这是一种标准模式，如果使用它的程序员在 try-with-resources 块中使用它，则可以完美地工作。

但是，有人可以创建一个实例，手动关闭它，然后尝试继续使用它，这会导致 `SQLException` 连接中断。要通过防御性编程来处理这种情况，您可以：

1. 在 close 方法中，关闭后将连接字段设置为 null，作为此实例已关闭的标志。
2. 在所有其他方法中，首先检查连接字段是否为空，如果是则抛出异常（您可以为此编写自己的私有方法）。根据 Java 约定，在这种情况下抛出的正确异常是 a `java.lang.IllegalStateException`，这大致意味着*您在错误的时间调用了方法*- 在这种情况下是在关闭有问题的资源之后。

练习3

`Candidate getCandidate(int id)` 在数据服务上也实现一个方法。这将要求您在 SQL 中使用 JOIN 并创建所有三个域类的实例。