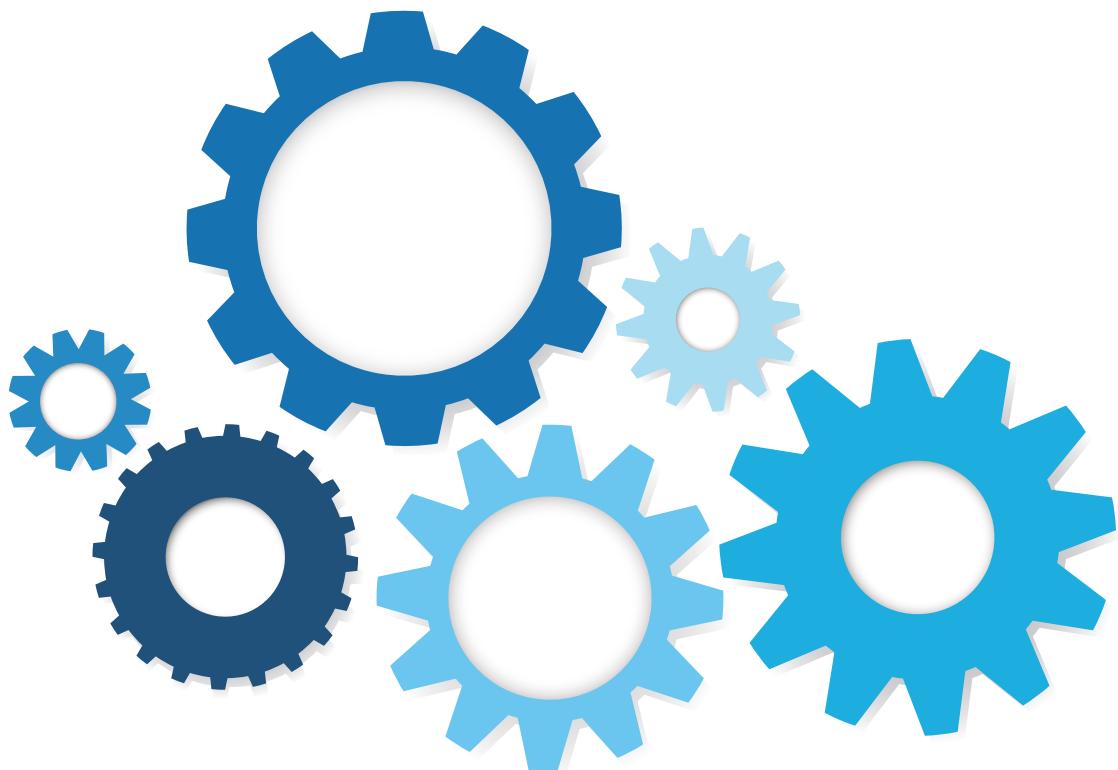


SQL 基础教程



目录

第 1 章 环境搭建	1
1.1 MySQL 8.0 的安装（强烈建议使用 8.x 版本）	1
1.1.1 Windows 下 MySQL 8.0 的下载安装	1
1.1.2 macOS 下 MySQL 8.0 的下载安装	24
1.1.3 Linux 下 MySQL 8.0 的下载安装	35
1.3.3 设置远程连接:	41
1.2 连接 MySQL 并执行 SQL 查询	41
1.2.1 使用命令行方式连接 MySQL 服务	41
1.2.2 使用 MySQL Workbench 连接 MySQL	42
1.2.3 [选学] 使用 HeidiSQL 连接 MySQL	46
1.2.4 [选学] 使用 DBeaver 连接 MySQL	48
1.2.5 [选学] 使用 Navicat 连接 MySQL	54
1.2.6 [选学] 使用 SQLyog 连接 MySQL	54
1.2.7 [选学] DataGrip 的安装和连接 MySQL	55
1.3 创建学习用的数据库	55
第 2 章 初识数据库	56
2.1 初始数据库	56
2.1.1 DBMS 的种类	56
2.1.2 RDBMS 的常见系统结构	56
2.2 初识 SQL	57
2.2.1 SQL 的基本书写规则	58
2.2.2 数据库的创建（CREATE DATABASE 语句）	59
2.2.3 表的创建（CREATE TABLE 语句）	59
2.2.4 命名规则	59
2.2.5 数据类型的指定	60
2.2.6 约束的设置	60
2.2.7 表的删除和更新	60
2.2.8 向 product 表中插入数据	63
练习题	65
2.1	65
2.2	65
2.3	65
2.4	66
第 3 章 SELECT 语句基础	67
3.1 SELECT 语句基础	67
3.1.1 从表中选取数据	67
3.1.2 从表中选取符合条件的数据	67
3.1.3 相关法则	67
3.2 算术运算符和比较运算符	68
3.2.1 算术运算符	68

3.2.2 比较运算符	68
3.2.3 常用法则	69
3.3 逻辑运算符	69
3.3.1 NOT 运算符	69
3.3.2 AND 运算符和 OR 运算符	70
3.3.3 真值表	71
练习题-第一部分	73
3.1	73
3.2	73
3.3	74
3.4	74
3.4 对表进行聚合查询	74
3.4.1 聚合函数	74
3.4.2 常用法则	75
3.5 对表进行分组	75
3.5.1 GROUP BY 语句	75
3.5.2 常见错误	77
3.6 为聚合结果指定条件	77
3.6.1 用 HAVING 得到特定分组	77
3.6.2 HAVING 特点	77
3.7 对查询结果进行排序	78
3.7.1 ORDER BY	78
3.7.2 ORDER BY 中列名可使用别名	78
练习题-第二部分	79
3.5	79
3.6	79
3.7	79
第 4 章 复杂一点的查询	81
4.1 视图	81
4.1.1 什么是视图	81
4.1.2 视图与表有什么区别	81
4.1.3 为什么会有视图	81
4.1.4 如何创建视图	82
4.1.5 如何修改视图结构	85
4.1.6 如何更新视图内容	86
4.1.7 如何删除视图	87
4.2 子查询	87
4.2.1 什么是子查询	88
4.2.2 子查询和视图的关系	88
4.2.3 嵌套子查询	88
4.2.4 标量子查询	88
4.2.5 标量子查询有什么用	89
4.2.6 关联子查询	89
小结	90

练习题-第一部分	91
4.1	91
4.2	91
4.3	91
4.4	92
4.3 各种各样的函数	92
4.3.1 算数函数	93
4.3.2 字符串函数	95
4.3.3 日期函数	97
4.3.4 转换函数	99
4.4 谓词	100
4.4.1 什么是谓词	100
4.4.2 LIKE 谓词 – 用于字符串的部分一致查询	100
4.4.3 BETWEEN 谓词 – 用于范围查询	102
4.4.4 IS NULL、IS NOT NULL – 用于判断是否为 NULL	103
4.4.5 IN 谓词 – OR 的简便用法	104
4.4.6 使用子查询作为 IN 谓词的参数	105
4.4.7 EXIST 谓词	108
4.5 CASE 表达式	111
4.5.1 什么是 CASE 表达式?	111
4.5.2 CASE 表达式的使用方法	111
练习题-第二部分	114
4.5	114
4.6	114
4.7	115
第 5 章 集合运算	116
5.1 表的加减法	116
5.1.1 什么是集合运算	116
5.1.2 表的加法–UNION	117
5.1.3 MySQL 8.0 不支持交运算 INTERSECT	122
5.1.4 差集, 补集与表的减法	123
5.1.5 对称差	124
5.2 连结 (JOIN)	125
5.2.1 内连结 (INNER JOIN)	126
5.2.2 外连结 (OUTER JOIN)	137
5.2.3 多表连结	141
5.2.4 ON 子句进阶–非等值连结	145
5.2.5 交叉连结 - CROSS JOIN(笛卡尔积)	149
5.2.6 连结的特定语法和过时语法	152
练习题	153
5.1	153
5.2	153
5.3	153
5.4	153

5.5	153
第 6 章 SQL 高级处理	154
6.1 窗口函数	154
6.1.1 窗口函数概念及基本的使用方法	154
6.2 窗口函数种类	155
6.2.1 专用窗口函数	155
6.2.2 聚合函数在窗口函数上的使用	156
6.3 窗口函数的应用 - 计算移动平均	157
6.3.1 窗口函数适用范围和注意事项	159
6.4 GROUPING 运算符	159
6.4.1 ROLLUP - 计算合计及小计	159
练习题	160
6.1	160
6.2	161
6.3	161
第 7 章 综合练习	162
练习一: 各部门工资最高的员工 (难度: 中等)	162
练习二: 换座位 (难度: 中等)	162
练习三: 分数排名 (难度: 中等)	163
练习四: 连续出现的数字 (难度: 中等)	164
练习五: 树节点 (难度: 中等)	164
练习六: 至少有五名直接下属的经理 (难度: 中等)	165
练习七: 分数排名 (难度: 中等)	166
练习八: 查询回答率最高的问题 (难度: 中等)	166
练习九: 各部门前 3 高工资的员工 (难度: 中等)	167
练习十: 平面上最近距离 (难度: 困难)	168
练习十一: 行程和用户 (难度: 困难)	168
第 8 章 附录 1 - SQL 语法规范	170
语法规范	170
参考资料:	171
第 9 章 附录 2 - [选学] 使用 Python 连接 MySQL	172
9.1 使用 Python 连接 MySQL	172
9.1.1 安装 PyMySQL 模块	172
9.1.2 导入模块	172
9.1.3 使用 PyMySQL 的 connect 函数建立到 MySQL 的连接	172
9.1.4 创建用于执行 SQL 语句的游标对象	173
9.1.5 执行 SQL 语句	174
9.1.6 取回 SQL 语句的执行结果	174
9.1.7 使用 pandas 的 read_sql 函数执行 SQL 查询并取回结果为 DataFrame	174
9.1.8 关闭游标和数据库连接	175
9.1.9 使用函数封装数据库连接	176

第 1 章 环境搭建

本章重点内容：

- 在电脑上安装 MySQL 数据库系统
- 安装客户端并连接到本机上的 MySQL 数据库
- 使用提供的脚本创建本教程所使用的示例数据库

1.1 MySQL 8.0 的安装（强烈建议使用 8.x 版本）

考虑到大家所使用的操作系统的不同，本教程分别提供了 Windows 10, macOS 和 CentOS 平台上的 MySQL 8.0 的安装流程，你可根据自己所使用电脑的操作系统选择以下三节中的一节进行学习和安装。

1.1.1 Windows 下 MySQL 8.0 的下载安装

首先以最常见的 win10 为例，介绍 MySQL8.0 的下载安装。

1.1.1 下载 MySQL 针对个人用户和商业用户提供了不同的版本，MySQL 社区版 (MySQL Community Edition) 是供个人用户免费下载的开源数据库，本教程将以 MySQL 社区版为例进行安装连接和 SQL 查询的讲解。

MySQL 官网上的社区版软件的下载地址 <https://dev.mysql.com/downloads/>,

选择 MySQL Community Server 可以下载 windows 操作系统下的最新版 MySQL 安装文件。如果需要安装历史版本，可以选择最后的 Download Archives 后选择 MySQL Community Server，然后在新页面里选择所需历史版本的社区版。

如果想下载本教程所使用的 MySQL 8.0.21.0，也可以在百度网盘下载，

下载链接：<https://pan.baidu.com/s/1S0tMoVqqRXwa2qD0siHcIg>

提取码：80lf

备用下载链接：<https://pan.baidu.com/s/1zK2vj50DvuAee-EqAcl-0A>

提取码：80lf

我们接下来以文档写作时的最新版 MySQL 8.0.21 为例，进行下载安装的介绍。

进入到 MySQL Installer for Windows 页面后，选择下载下方的完整安装程序。

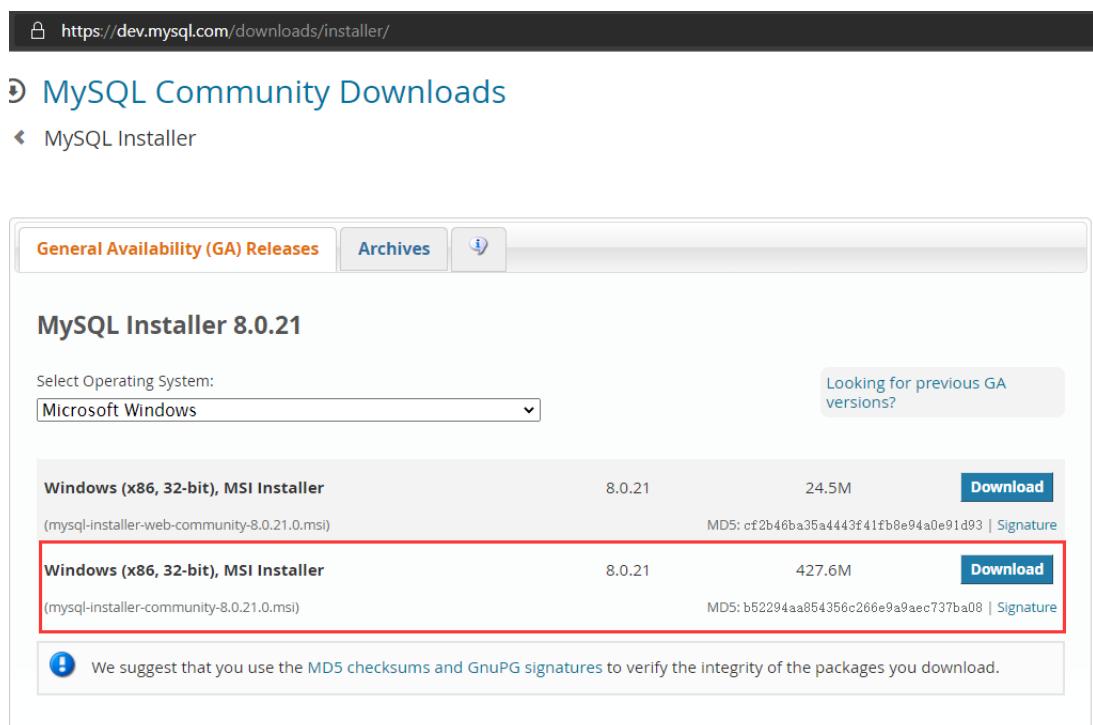


图 1.1: 下载页面 1

在下载页面选择下方的不注册，仅下载。

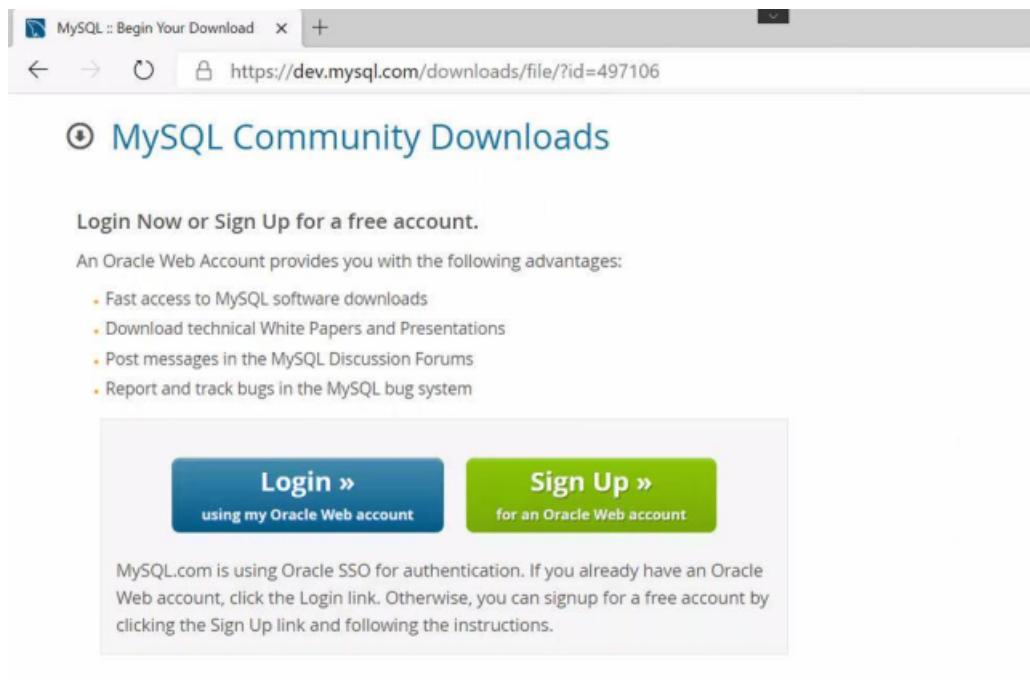


图 1.2: 下载页面 2

完成下载后，得到一个后缀为 msi 的安装文件。

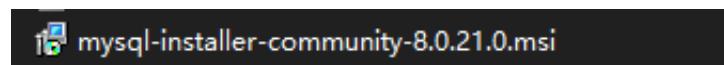


图 1.3: msi 安装包

1.1.2 安装 找到刚才下载好的 msi 文件，双击开始安装。初学者建议采用完全安装模式 (Full) 进行安装：

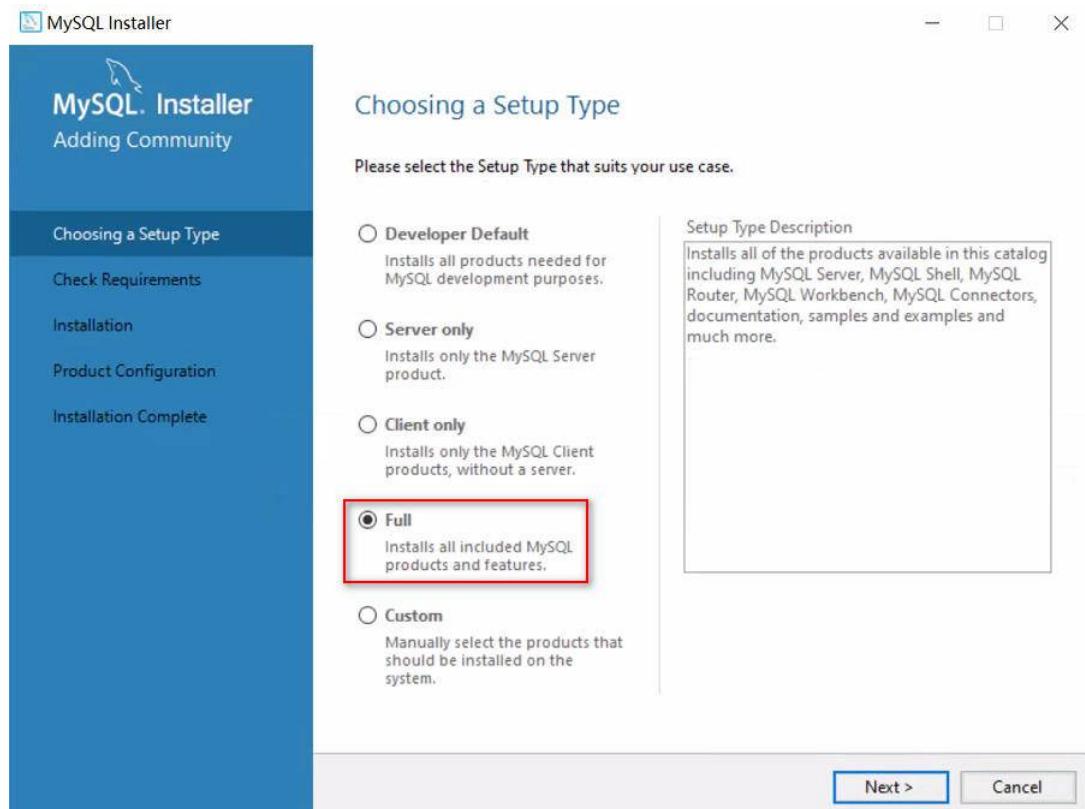


图 1.4: Win_setup_1

完全安装模式下，部分模块会依赖其他其他组件（每台电脑上列出的依赖项很可能会有不同）。

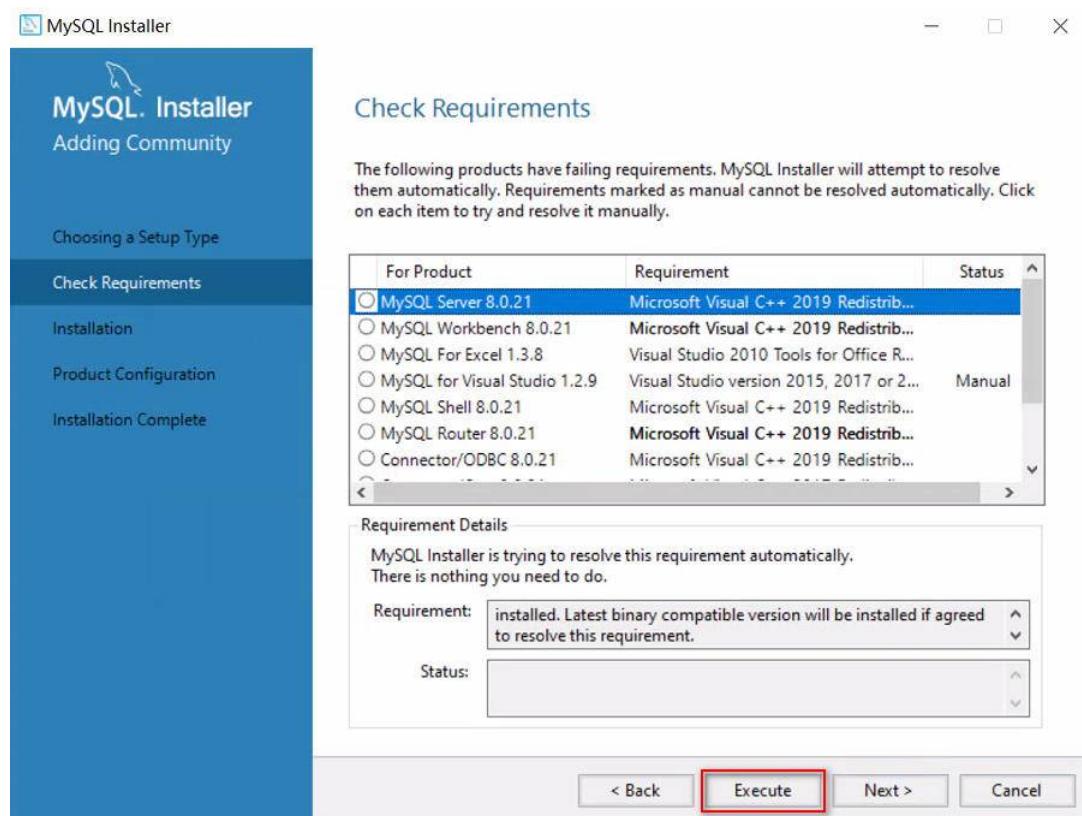


图 1.5: Win_setup_2

如果你的电脑之前没有安装过这些组件，则需要额外进行安装，此处点击 Execute 按钮即可。在这些所依赖的组件的安装过程中，只需要一路选择“同意”并逐个安装就可以了。

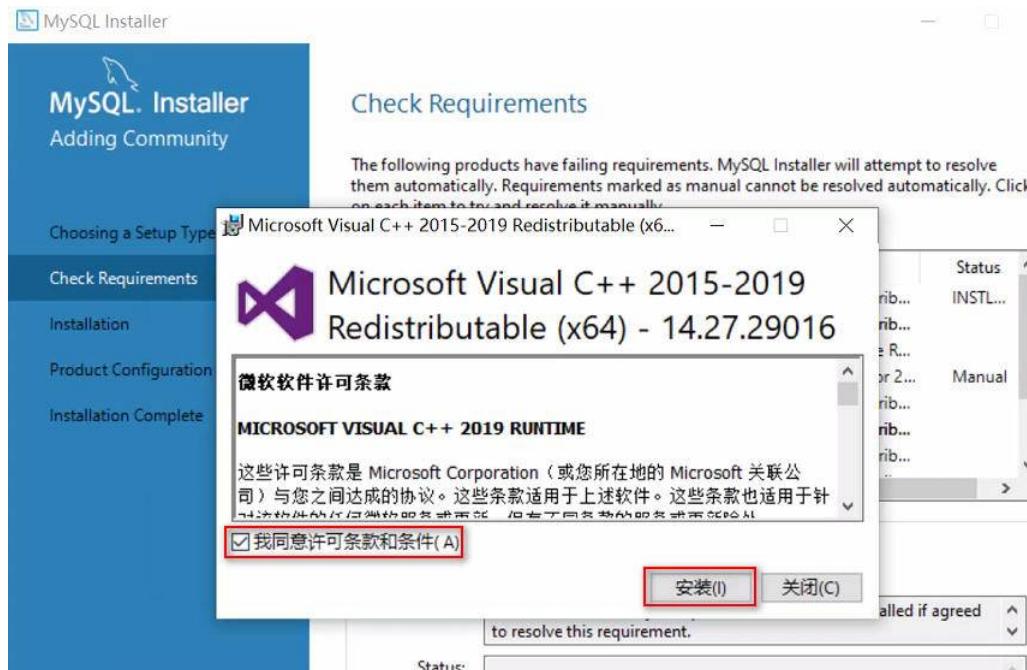


图 1.6: Win_setup_3

安装好一个组件后，点击关闭按钮，自动开始安装下一个组件（这一步根据操作系统版本可能会略有不同）

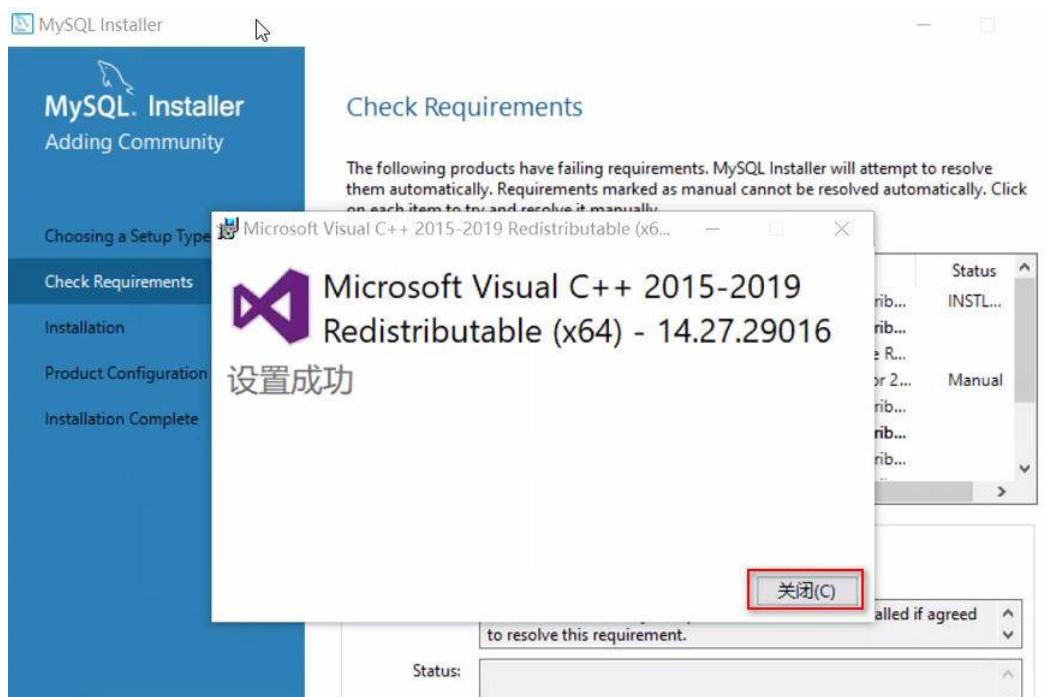


图 1.7: Win_setup_4

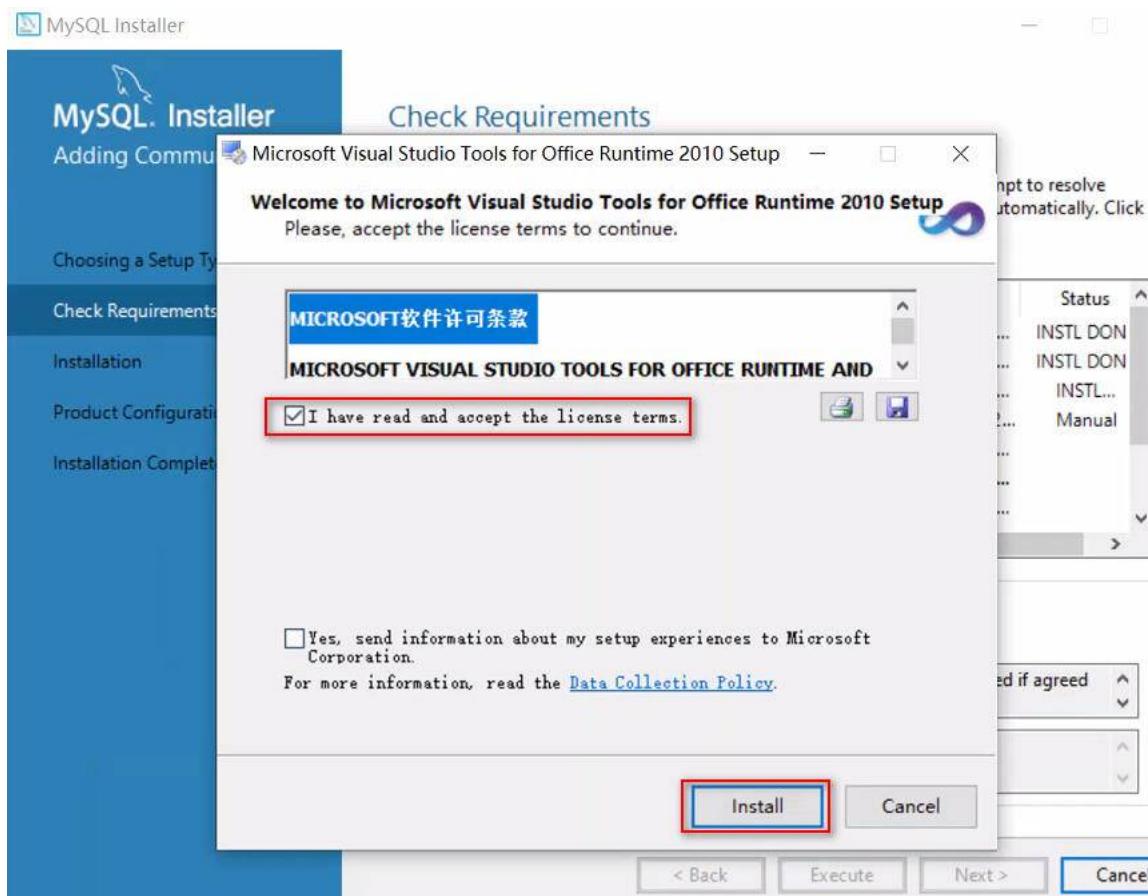


图 1.8: Win_setup_5

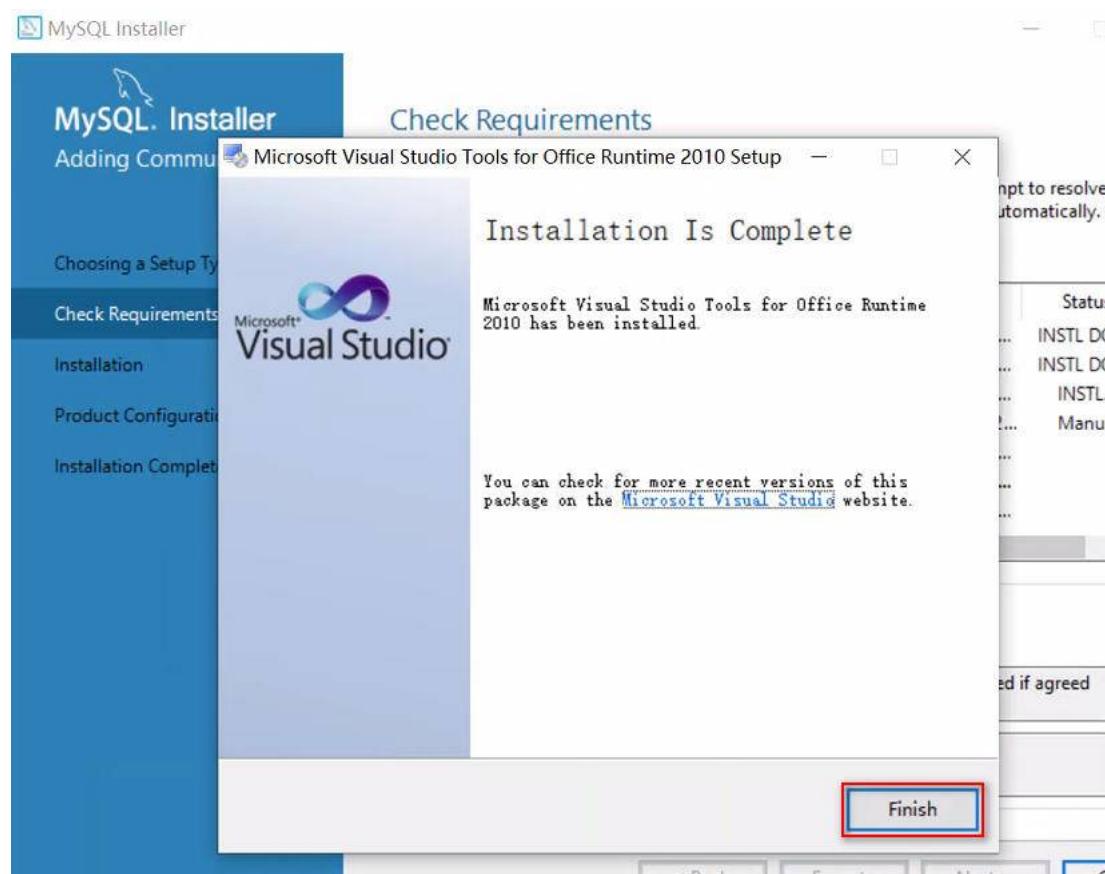


图 1.9: Win_setup_6

正常情况下，会将所有组件安装成功。但可能会有个别组件未安装成功。个别组件没有呈现绿色是因为你的电脑中缺少某个程序，例如，如果你的电脑没有安装 Python 环境，则该项目就不会呈现绿色。待下边剩下 3 个按钮且上方大部分组件为绿色时，即可点击 Next：

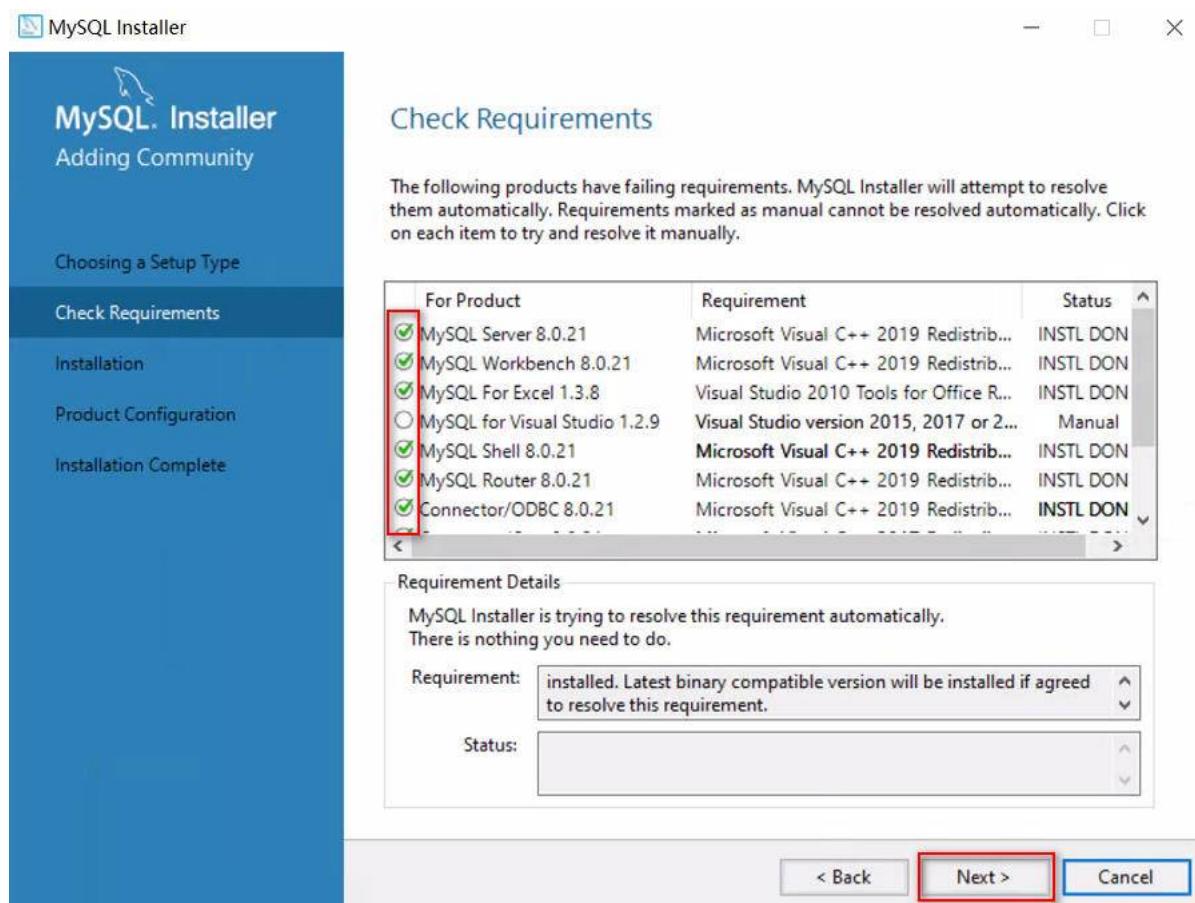


图 1.10: Win_setup_7

如果有个别组件未安装成功，此时可以先选择 Yes，忽略个别组件的安装。

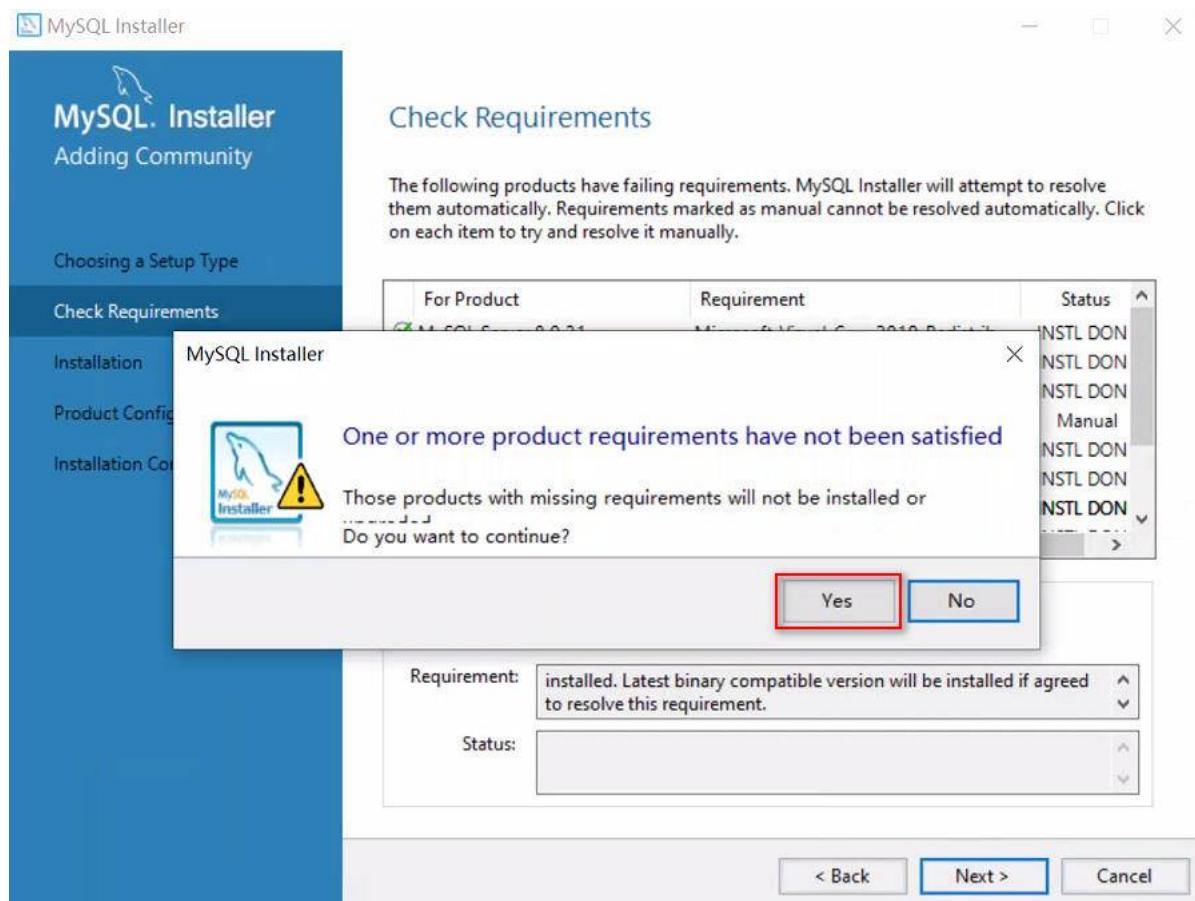


图 1.11: Win_setup_8

点击 Execute，开始安装服务器软件 MySQL Server，连接和查询软件 MySQL Workbench 及其他相关软件等内容。

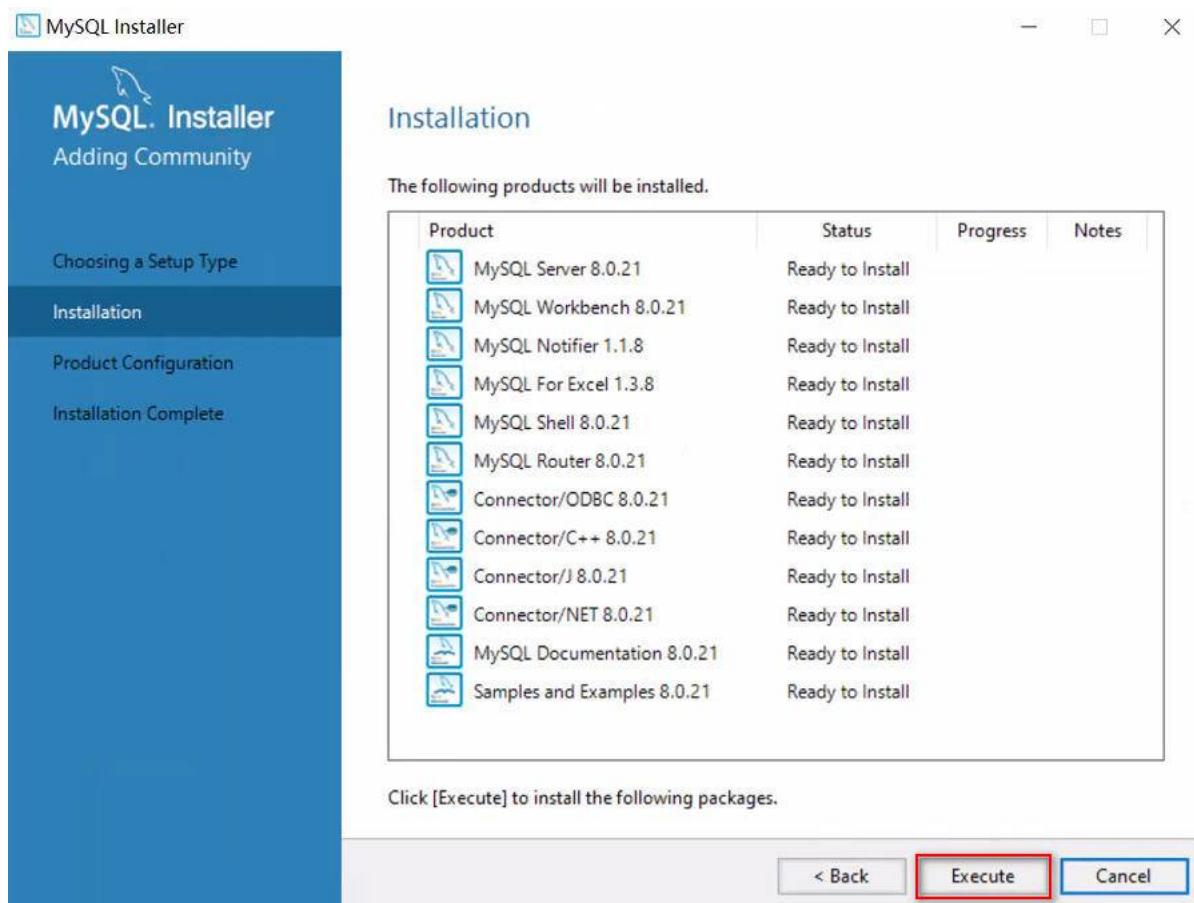


图 1.12: Win_setup_9

稍等片刻，安装完成后，点击 Next

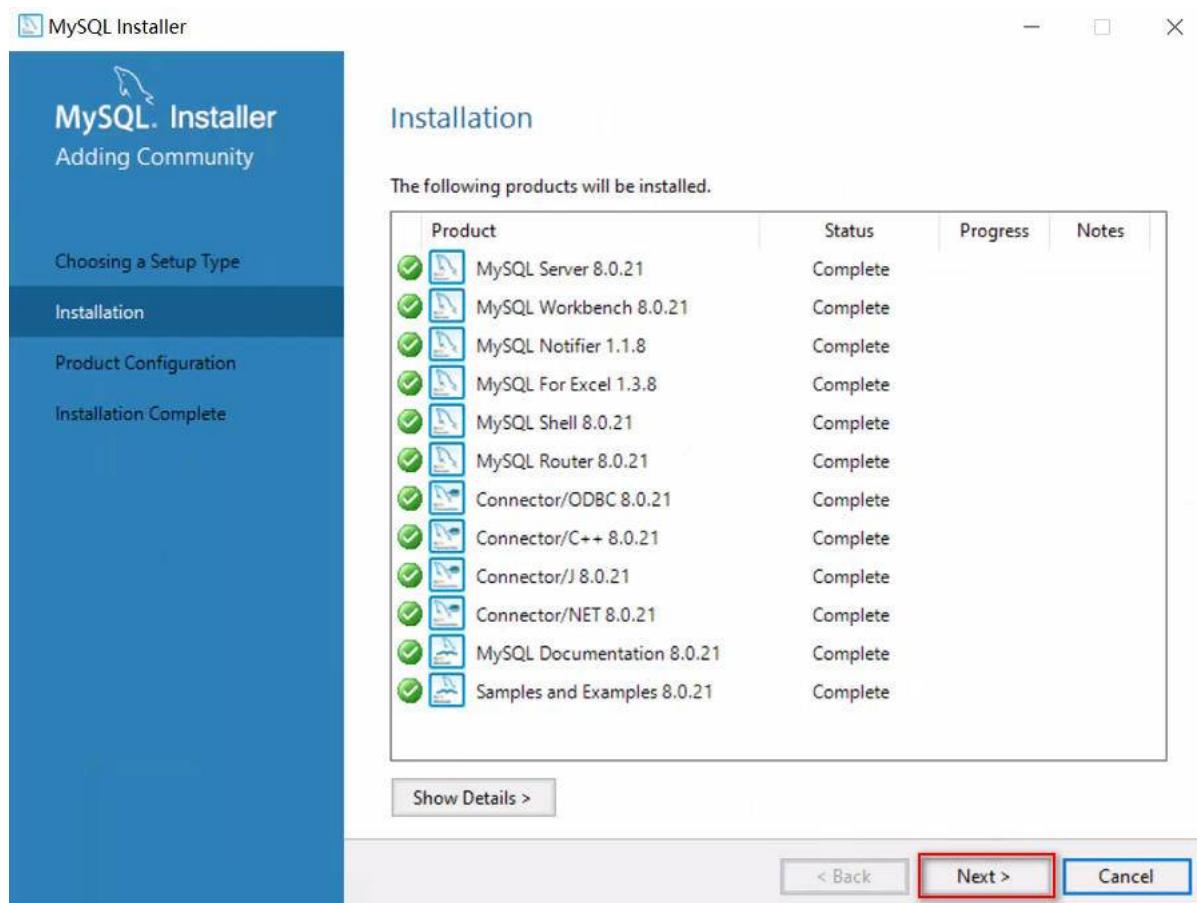


图 1.13: Win_setup_10

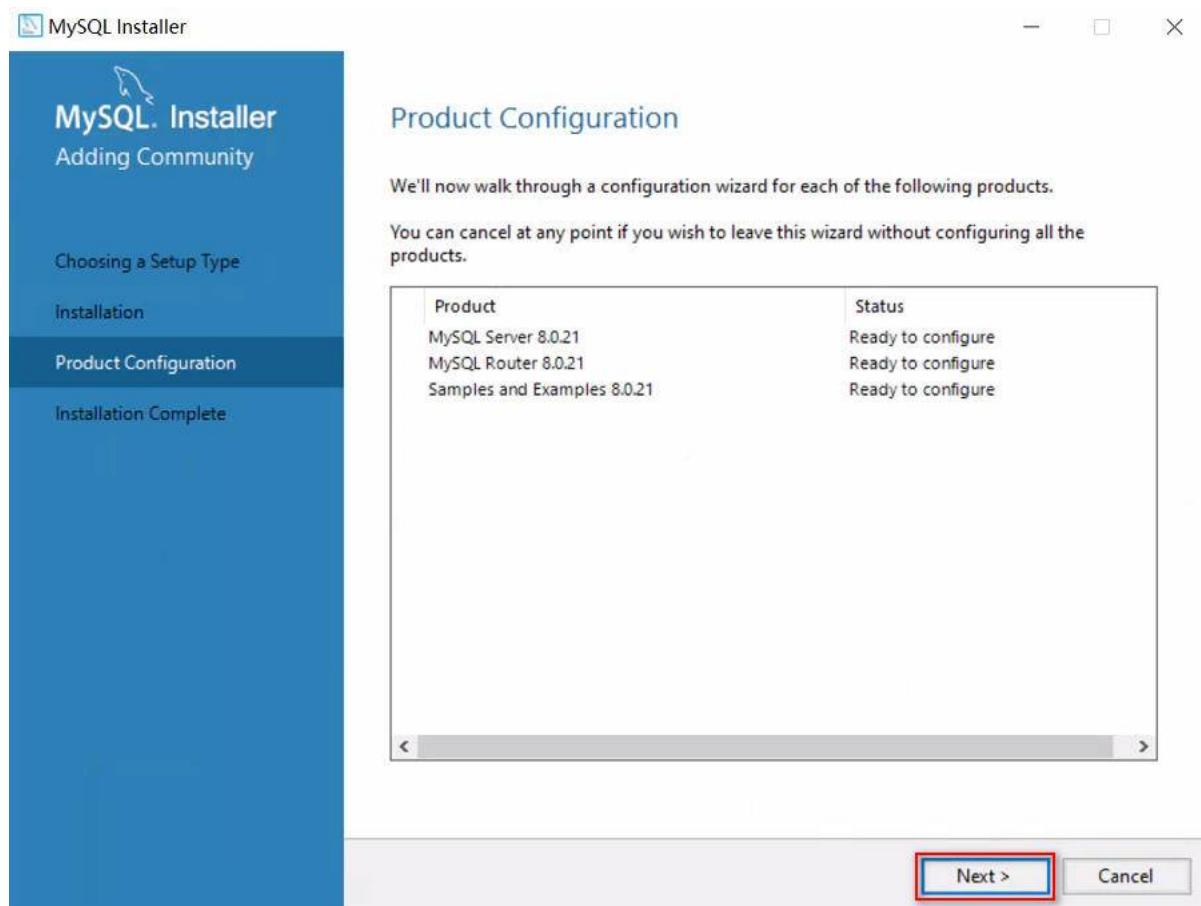


图 1.14: Win_setup_11

下图这一步是选是否以集群方式安装 MySQL，我们选择默认的第一个，然后点击 Next：

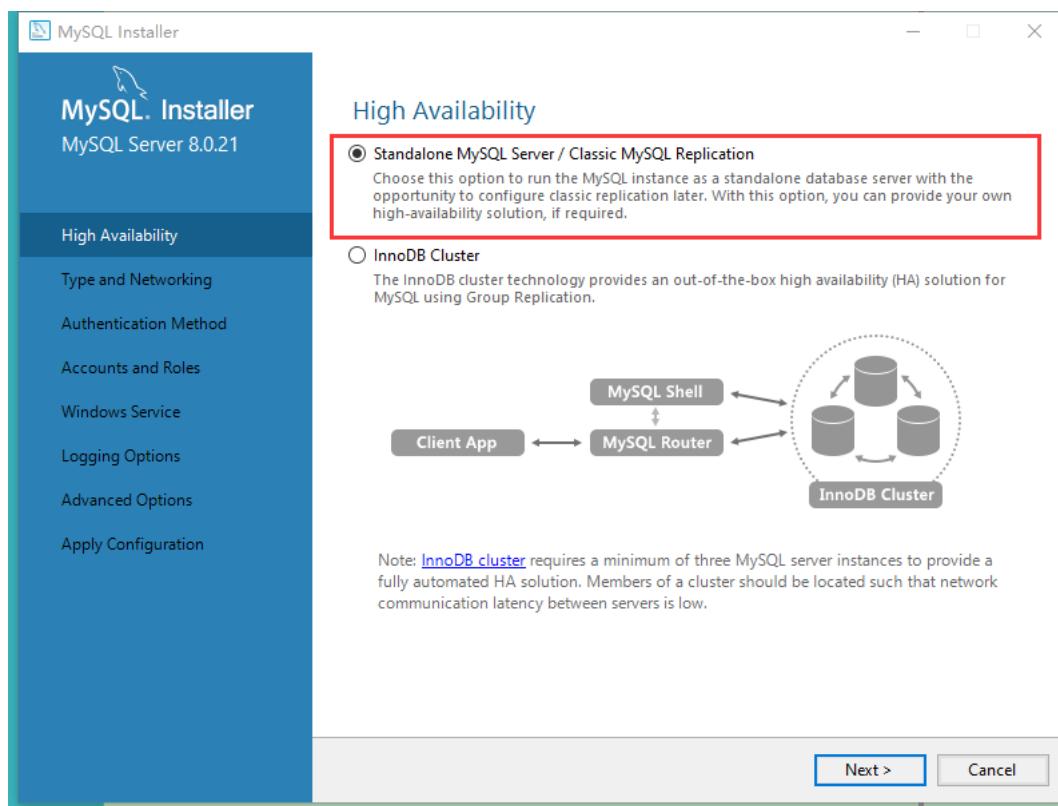


图 1.15: Win_setup_12

此处上边的各种相关配置保持默认即可，勾选最下边的“Show Advanced and Logging Options”框，然后点击 Next：

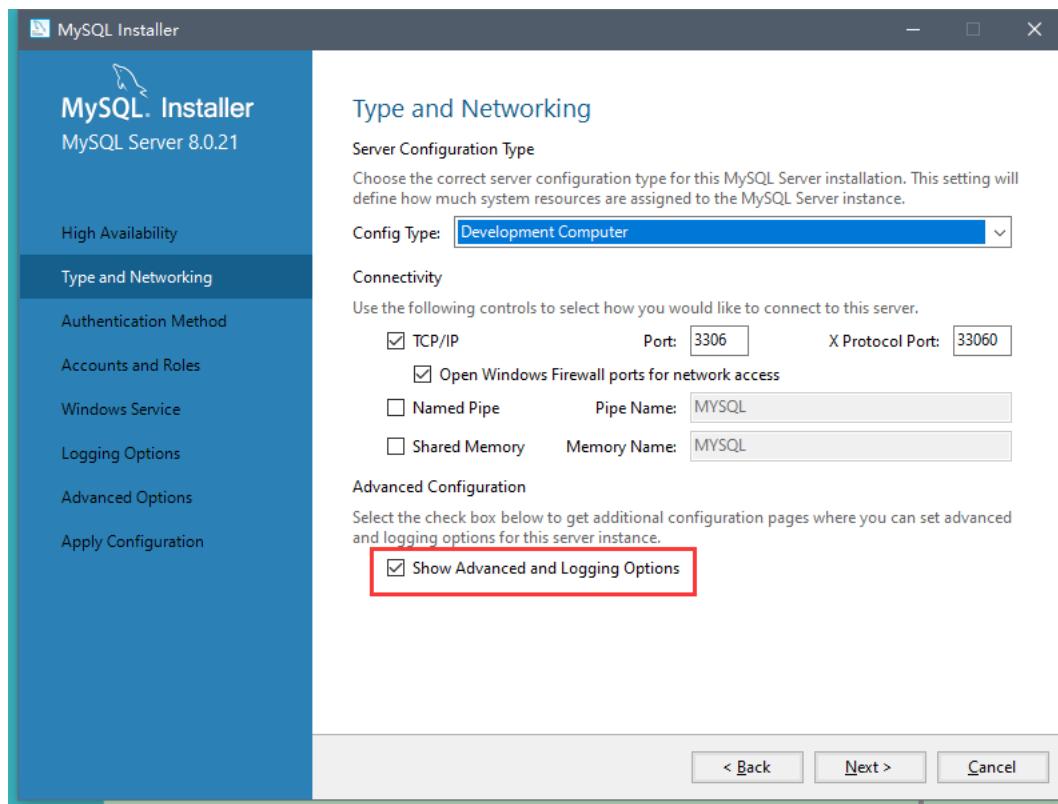


图 1.16: Win_setup_13

下图是密码强度的设置，第一种模式为强密码校验模式，MySQL 8.0 推荐使用最新的数据库和客户端，更换了加密插件，者可能导致第三方客户端工具无法连接数据库。

第二种加密方式沿袭了 MySQL 5.x 的加密方式，对第三方工具连接不敏感，我们仅为了学习 SQL 查询，并不需要很高的安全性，因此**此处请务必选择第二种方式（非常重要）**

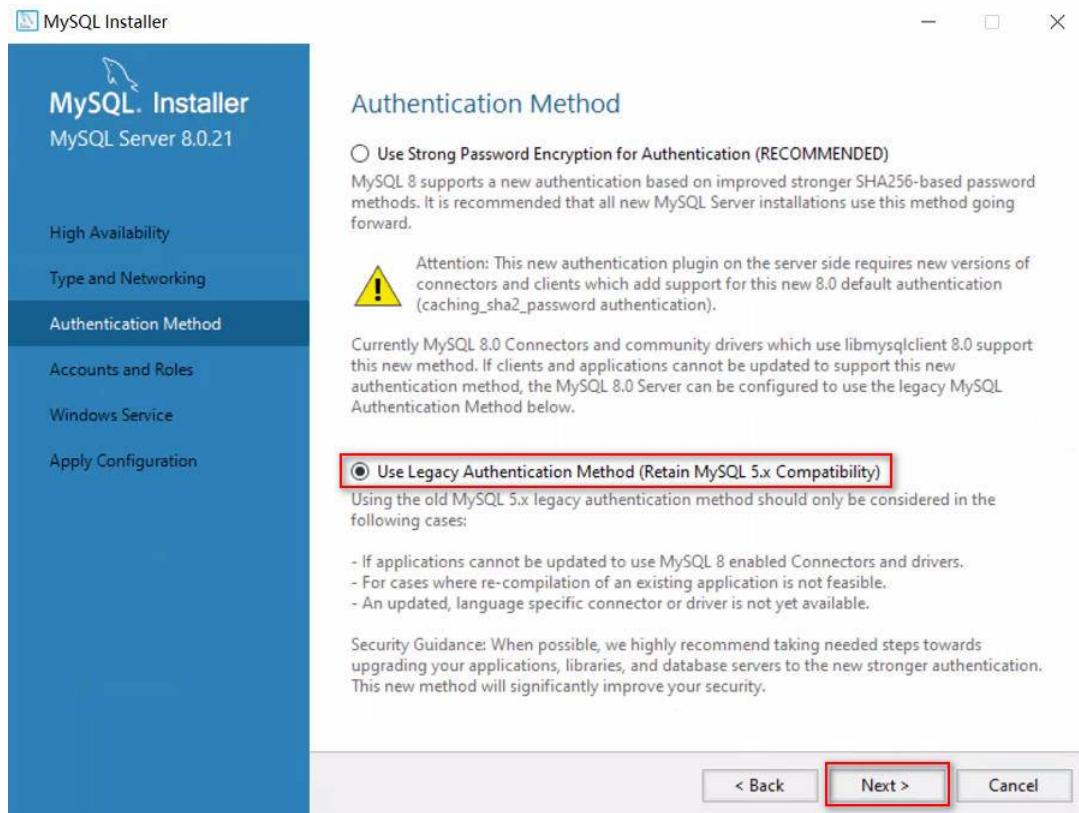


图 1.17: Win_setup_14

在这一步设置 MySQL 的 root 账户密码，由于上一步选择了第二个选项，因此这里可以设置为较简单容易记忆的而密码，例如"123456"。建议设置比较简单的密码，并将密码记录下来以防遗忘，忘记密码是一件麻烦事。

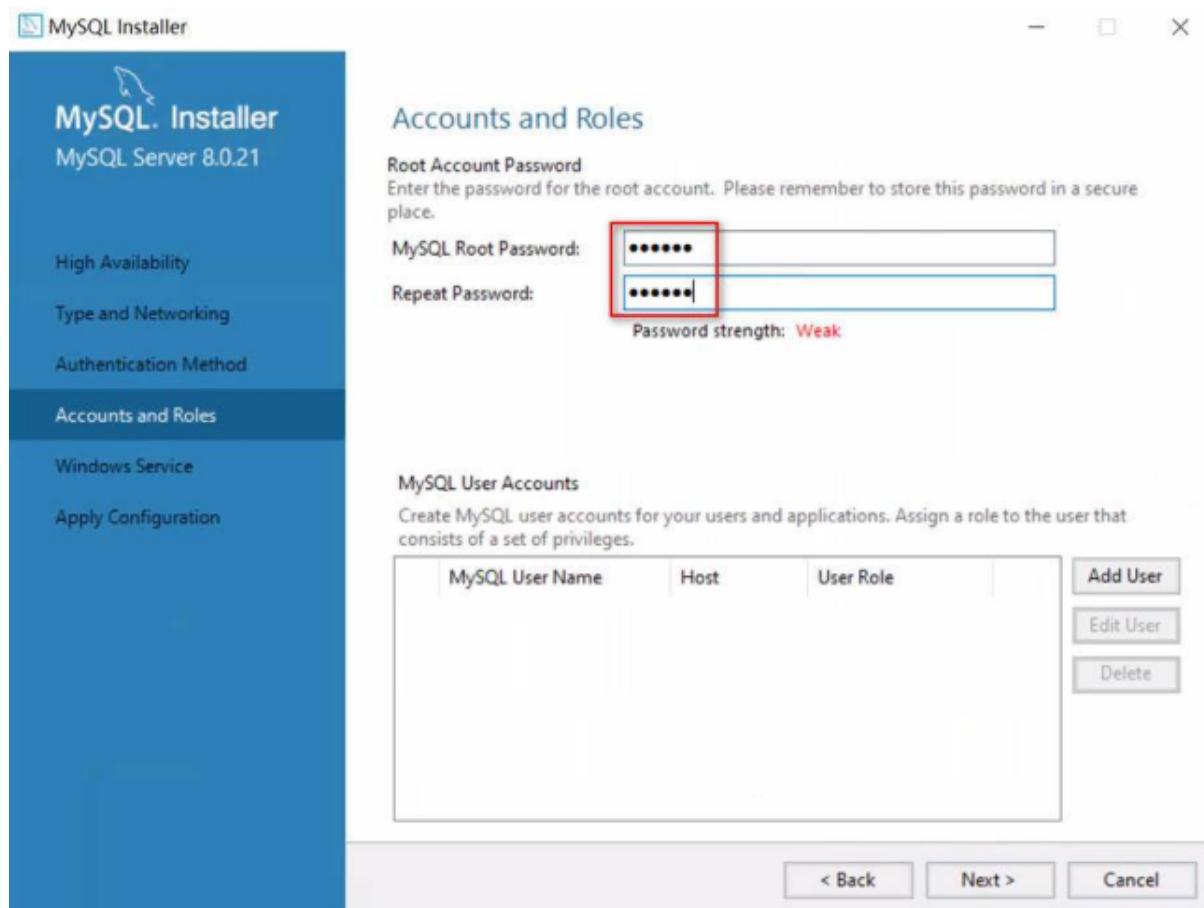


图 1.18: Win_setup_15

此处保持默认即可，如果 windows service name 右侧有**!**色警告图标显示，表示名称重复，手动更换一个名称即可，然后点击 Next：

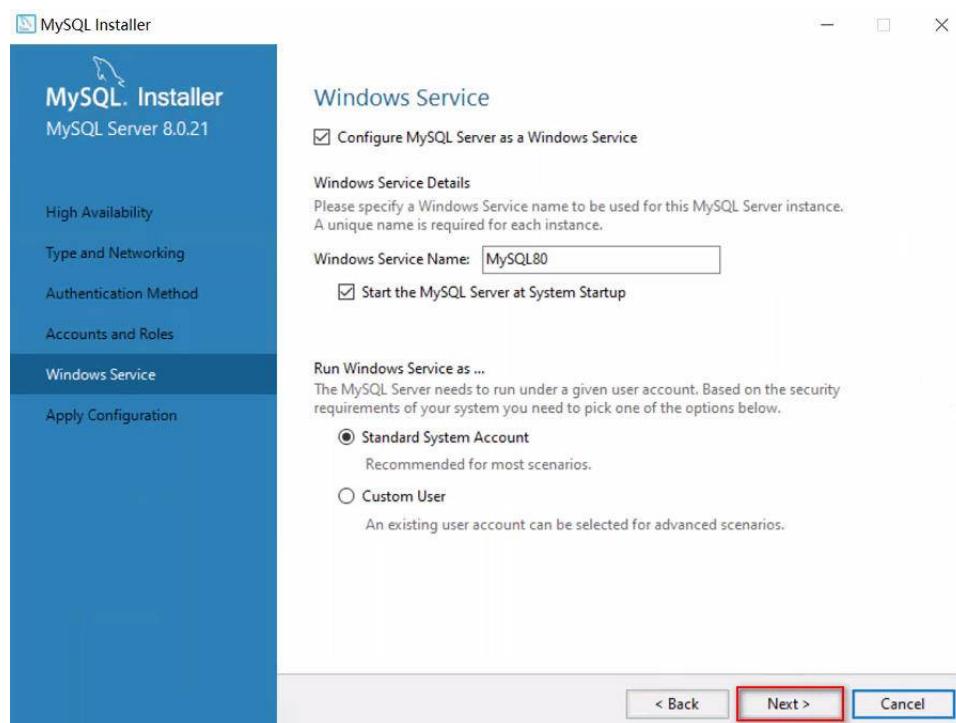


图 1.19: Win_setup_16

Logging Options 这里使用默认设置即可，我们的学习中暂时用不到这些设置，直接点击 Next：

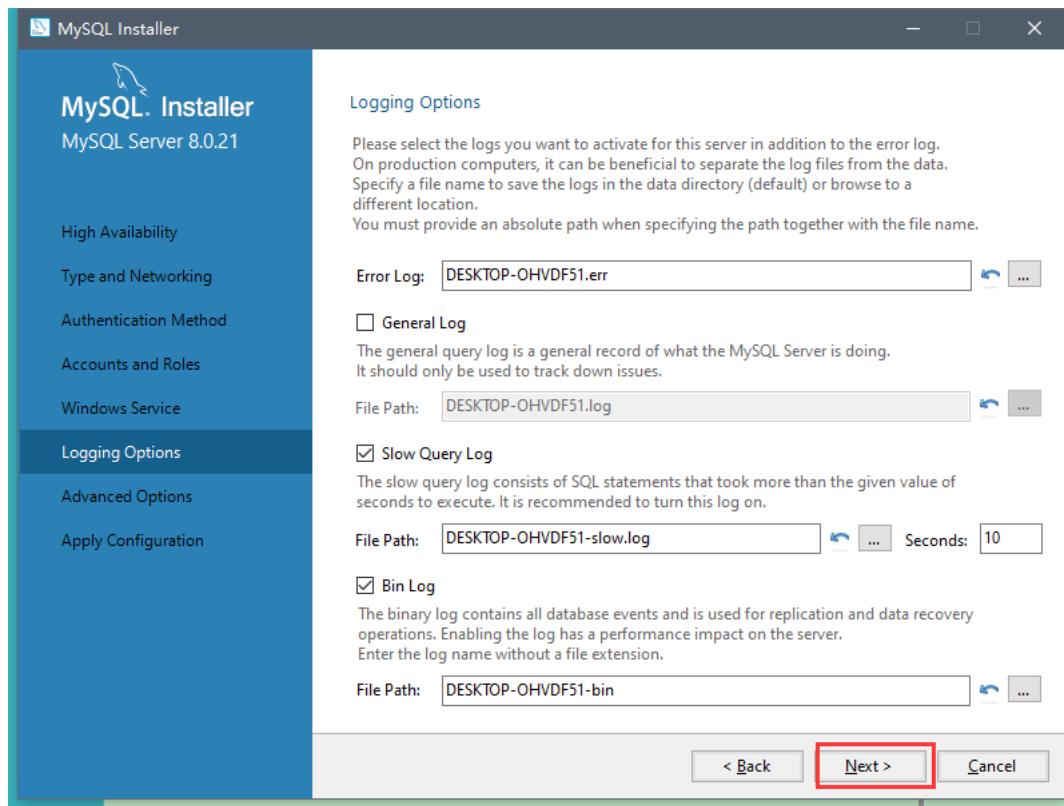


图 1.20: Win_setup_17

下图是设置是否大小写敏感的。这一步非常重要，由于 windows 系统是大小写不敏感的，请大家务必使用第一个选项 Lower Case。

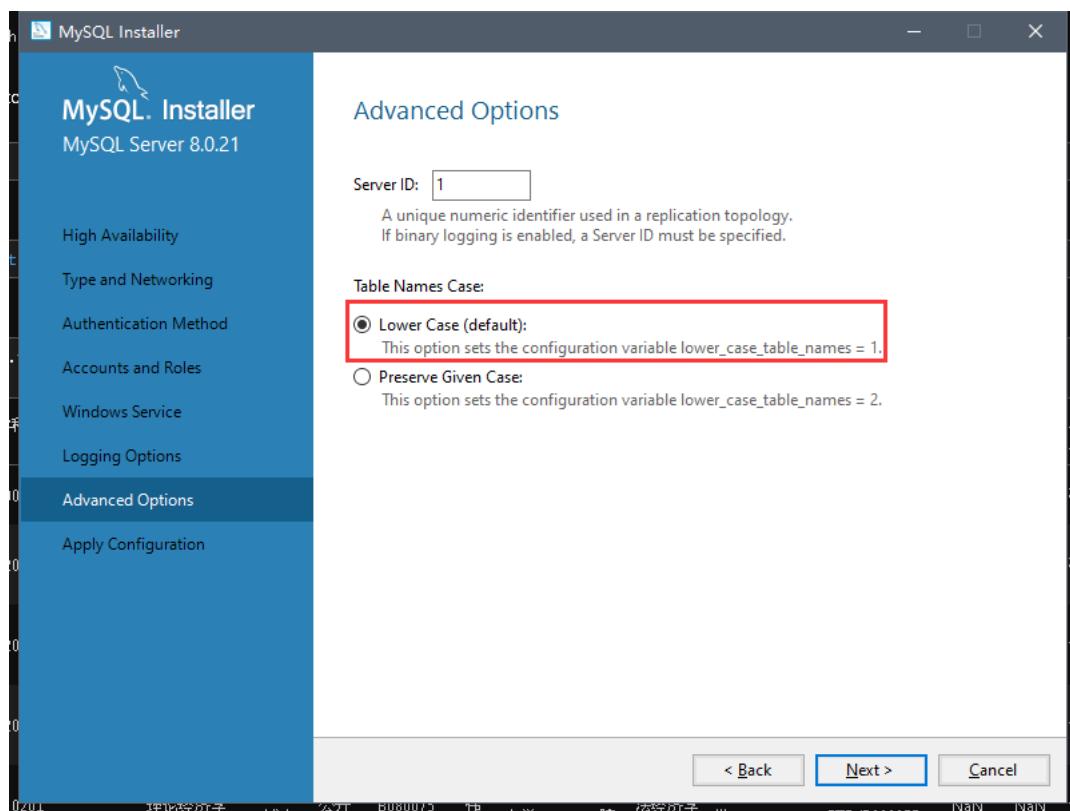


图 1.21: Win_setup_18

点击 Execute

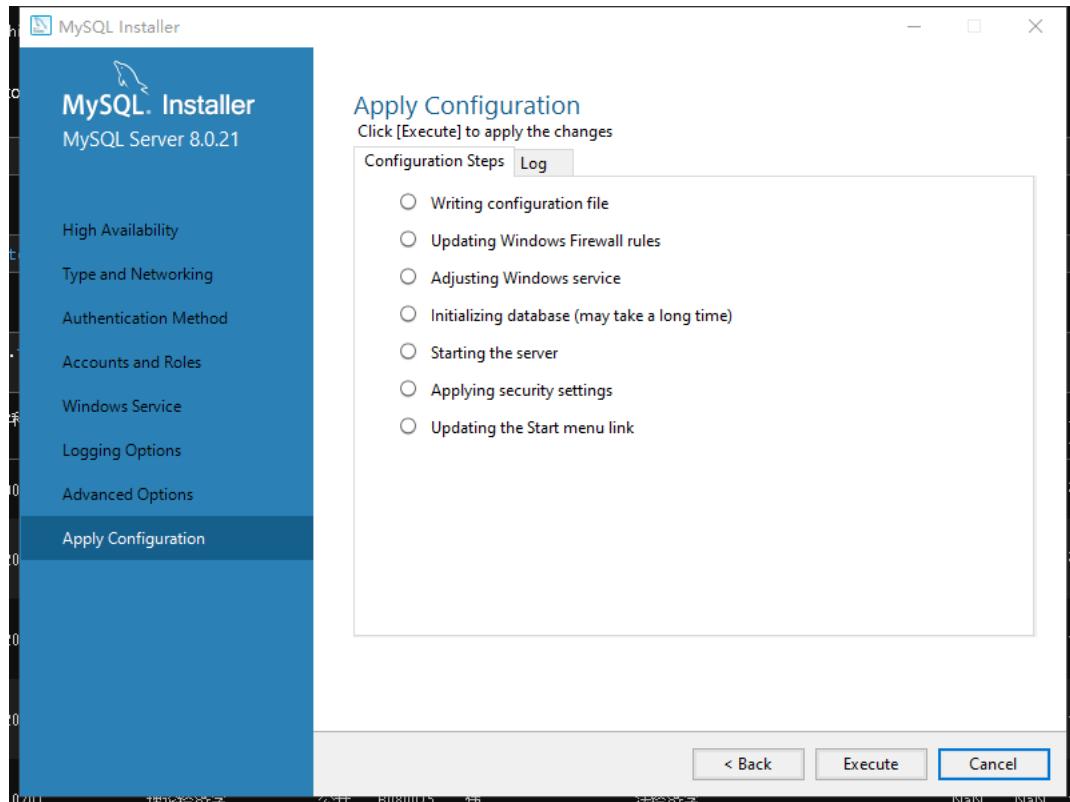


图 1.22: Win_setup_19

完成安装后，在下图中点击 Finish 回到安装的主进程：

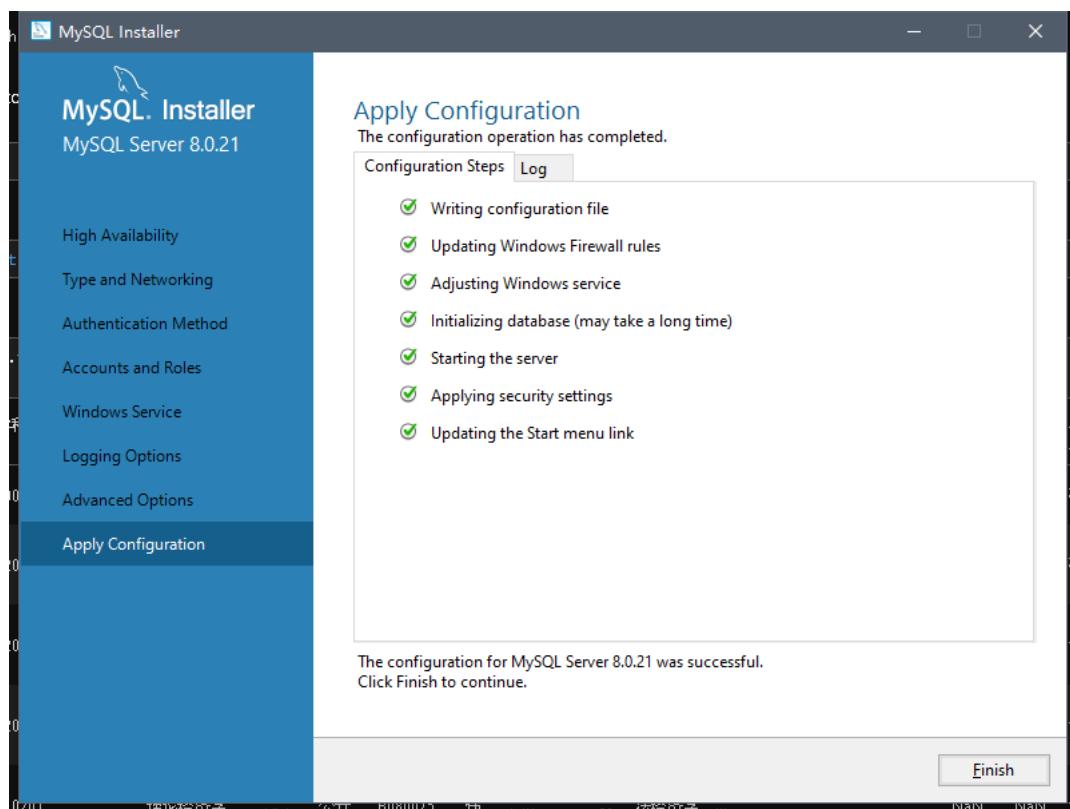


图 1.23: Win_setup_20

在主进程界面点击 Next

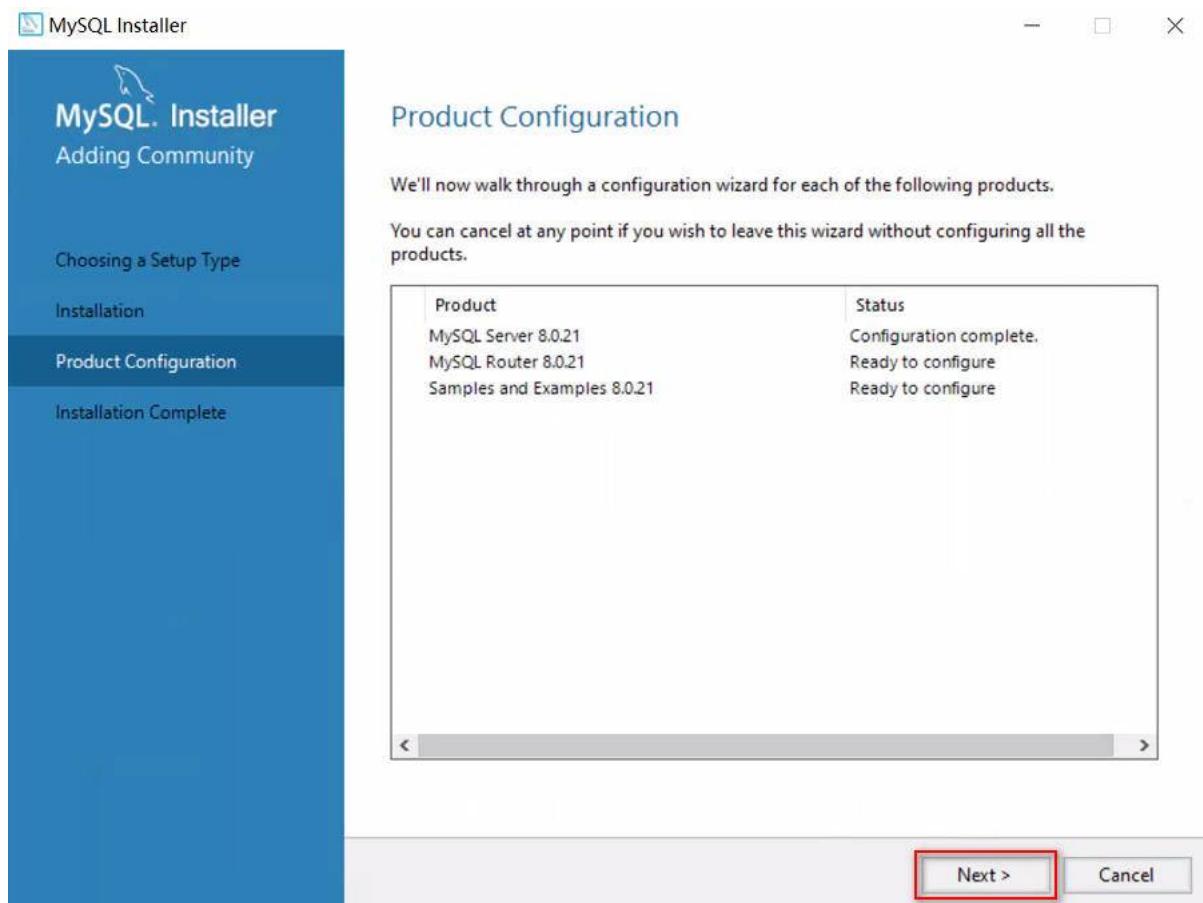


图 1.24: Win_setup_21

这一步无需任何选择，直接点击 Finish

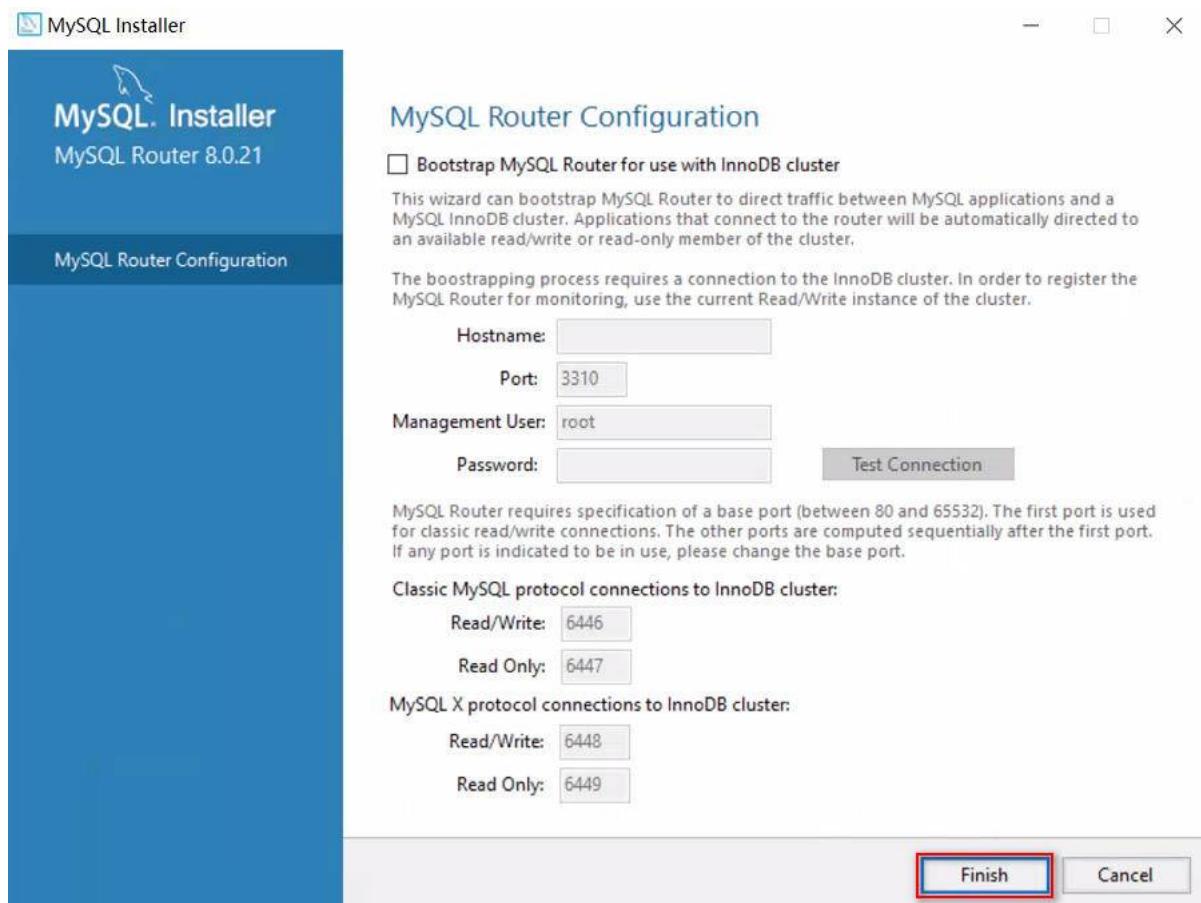


图 1.25: Win_setup_22

进入到 Connect To Server 界面后，输入刚才设置的密码，点击 check 进行校验，校验通过后 Status 会显示连接成功，然后点击 Next

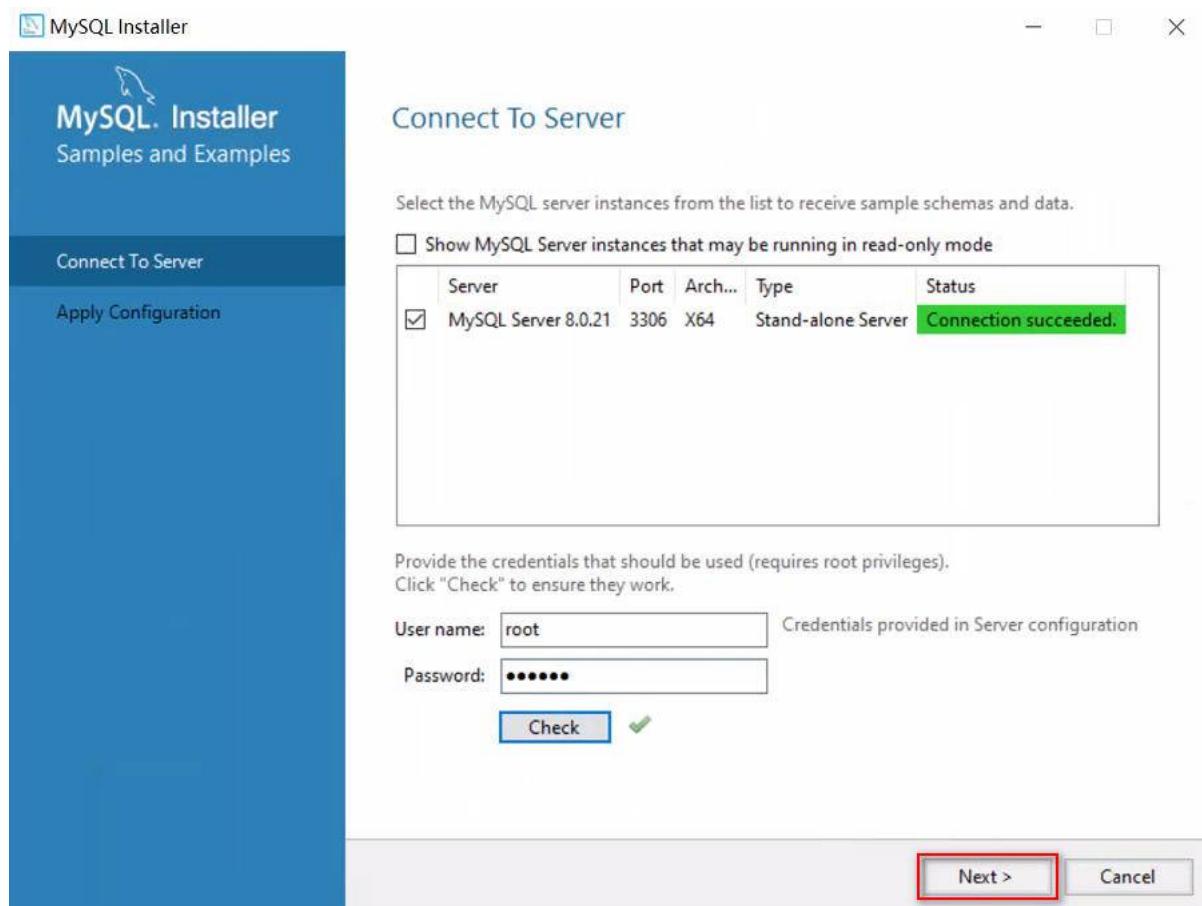


图 1.26: Win_setup_23

点击 Execute 应用设置：

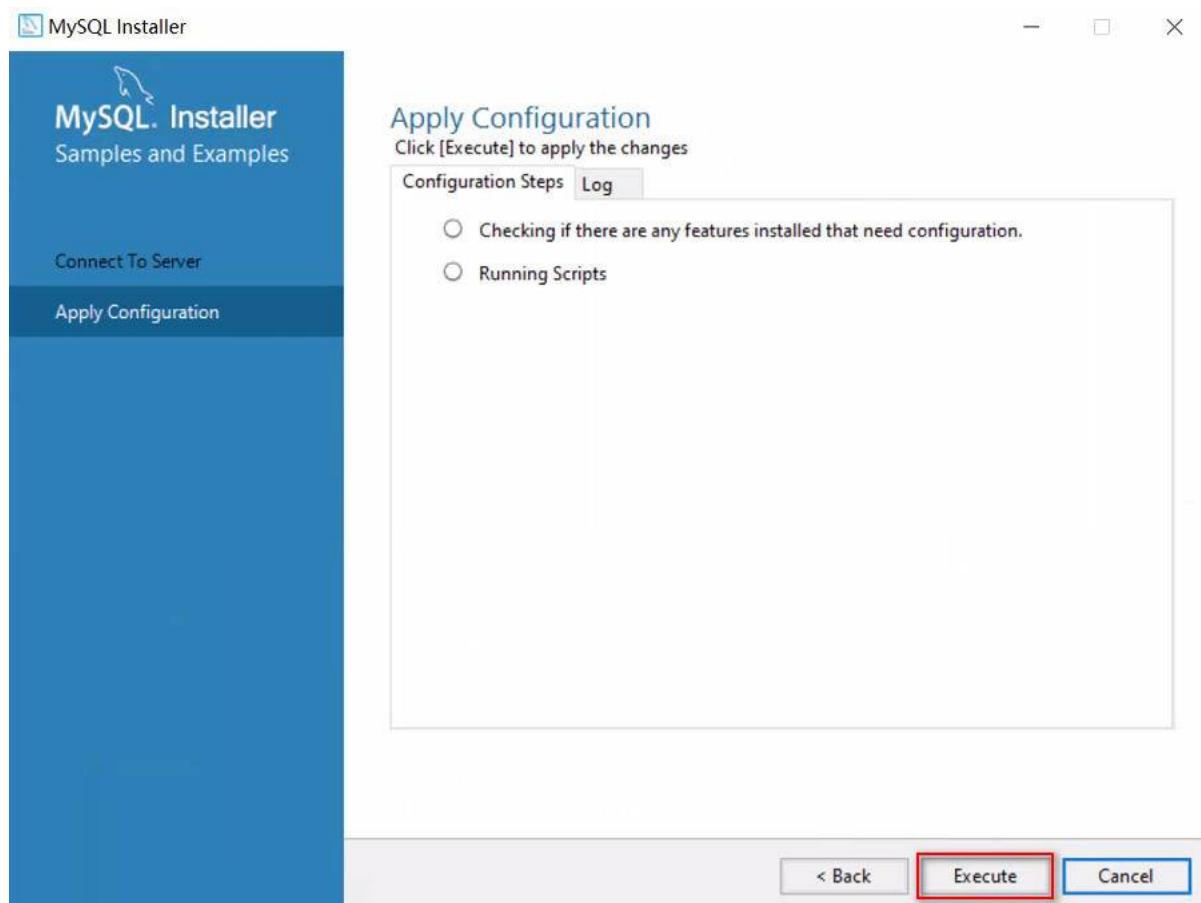


图 1.27: Win_setup_24

上述步骤完成后，点击 Finish

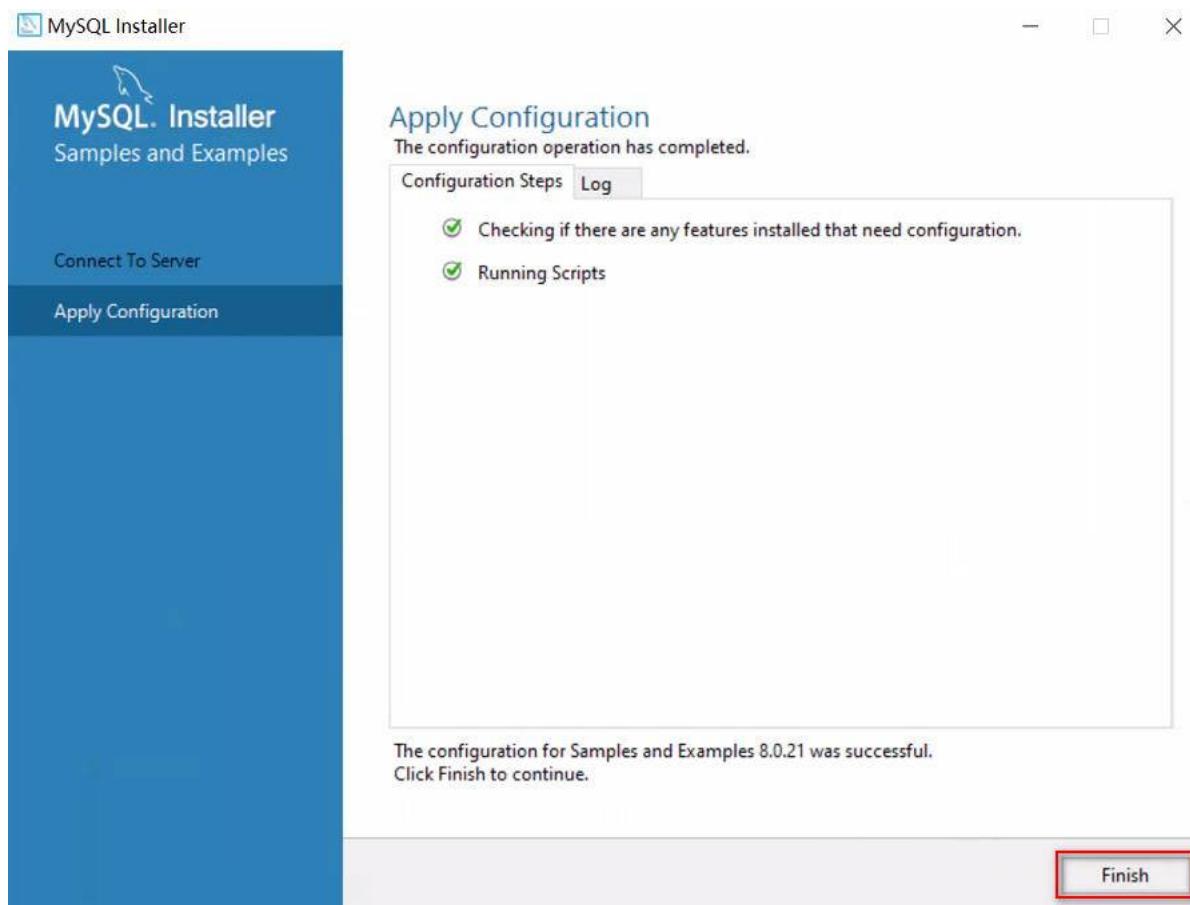


图 1.28: Win_setup_25

回到安装主进程后，点击 Next

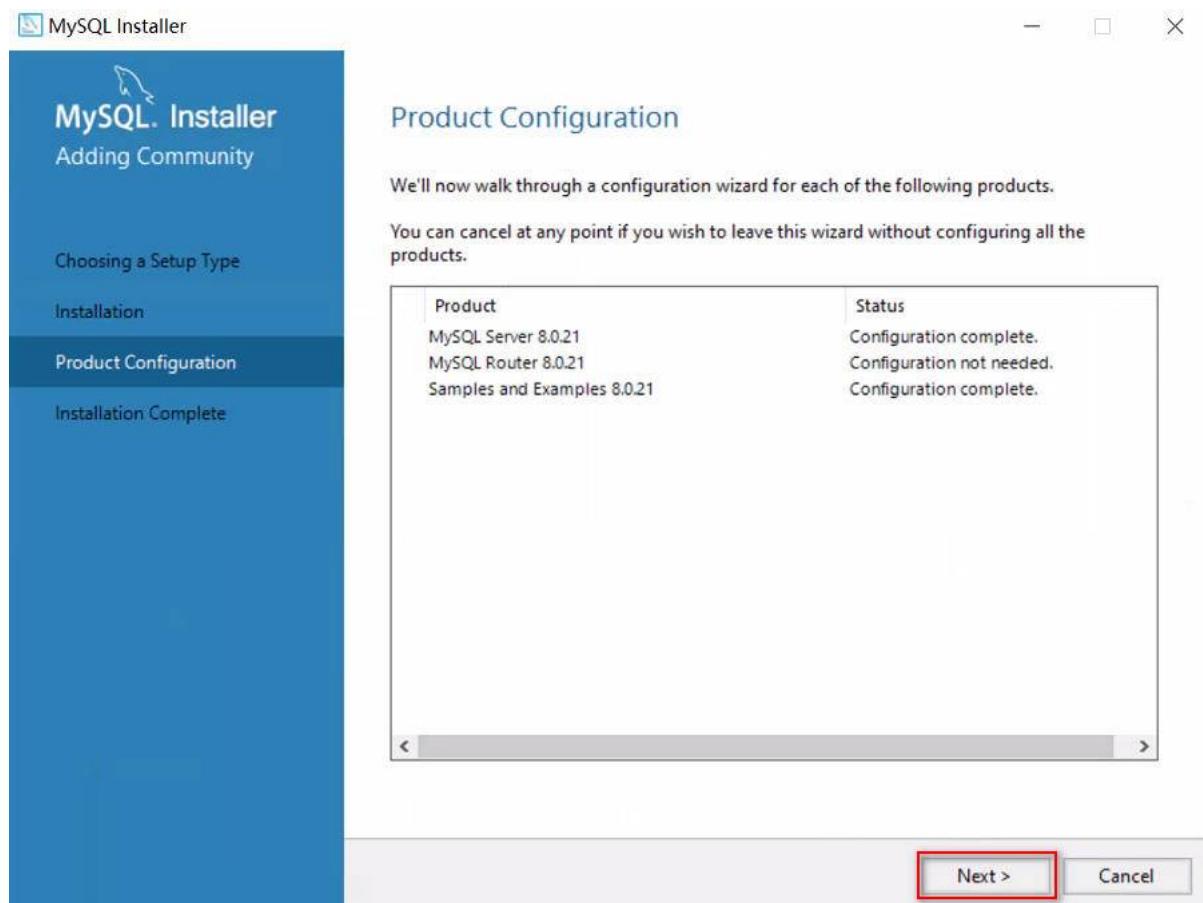


图 1.29: Win_setup_26

点击 Finish，完成安装。

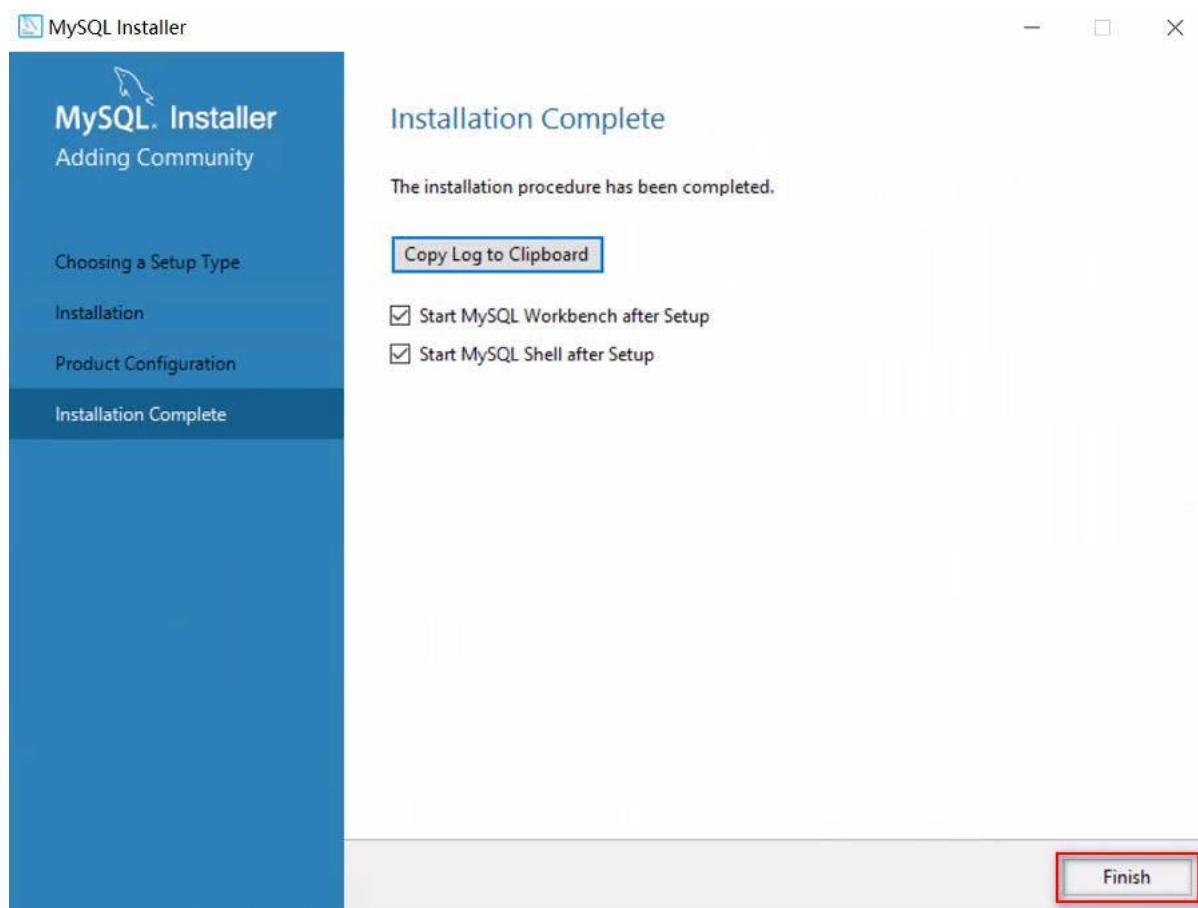


图 1.30: Win_setup_27

现在，你的电脑上就已经安装了 MySQL 的服务器软件，用于连接服务器进行查询的 MySQL Workbench，以及其他程序语言连接 MySQL 的驱动，此外还安装了几个示例数据库，但本教程将采用《SQL 基础教程》一书中的示例数据库，该数据库的创建和数据导入将在本章第三节介绍。

1.1.2 macOS 下 MySQL 8.0 的下载安装

MySQL 官网上社区版软件的下载地址 <https://dev.mysql.com/downloads/>，选择 MySQL Community Server，可以进一步选择下载 macOS 操作系统下的最新版 MySQL。

如果需要安装历史版本，可以选择最后的 Download Archives，之后再选择 MySQL Installer，然后在新页面里选择所需历史版本的社区版。

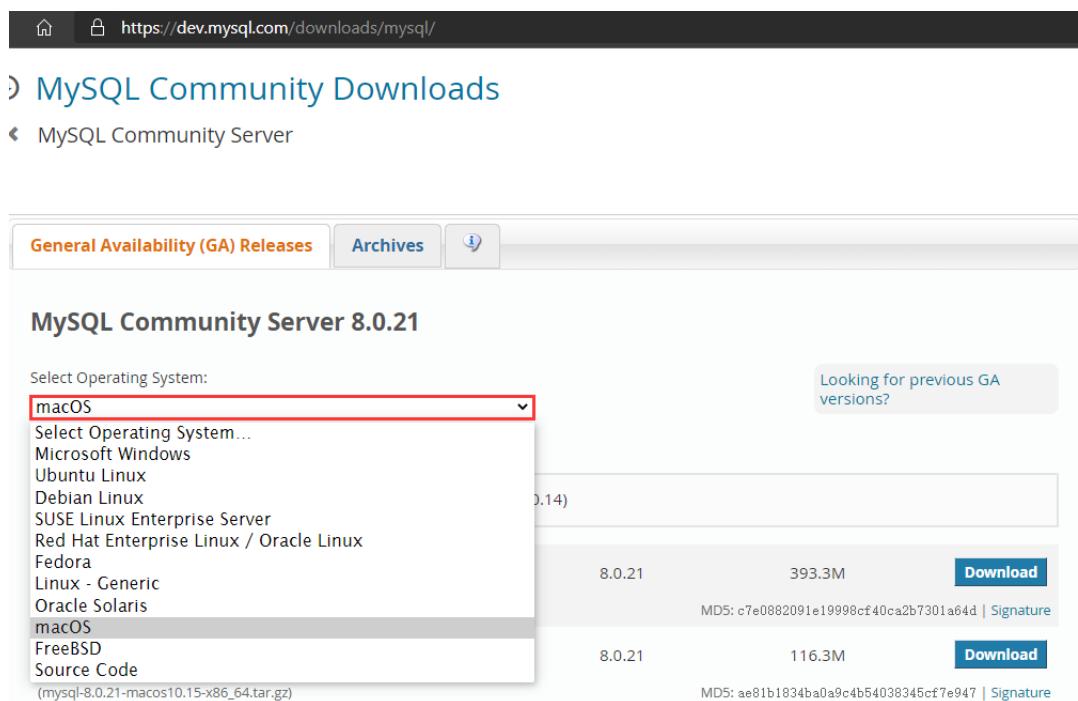


图 1.31: macOS_setup_1

然后选择下载 **DMG Archive** 安装文件。

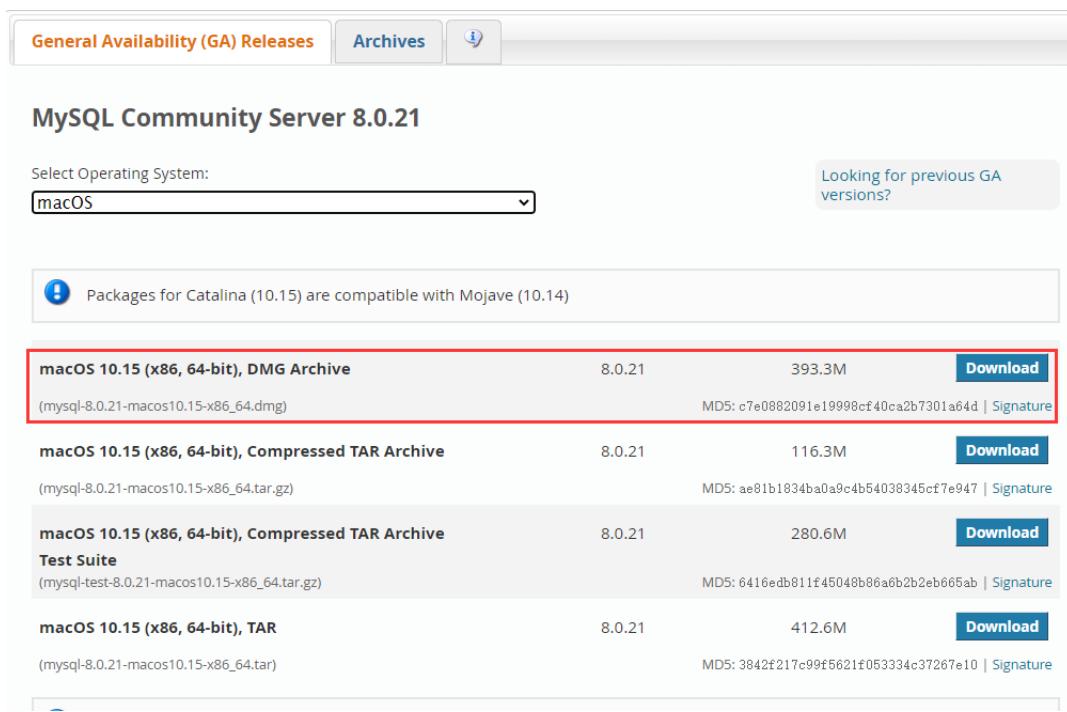


图 1.32: macOS_setup_2

点击下方的 **No thanks, just start my download** 直接下载。

④ MySQL Community Downloads

Login Now or Sign Up for a free account.

An Oracle Web Account provides you with the following advantages:

- Fast access to MySQL software downloads
- Download technical White Papers and Presentations
- Post messages in the MySQL Discussion Forums
- Report and track bugs in the MySQL bug system

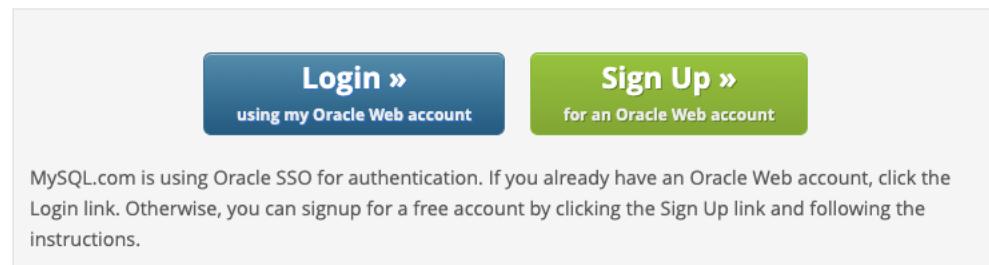


图 1.33: macOS_setup_3

下载的文件为：



图 1.34: macOS_setup_4

如果官网下载速度很慢，或者希望下载本教程所使用的 MySQL 8.0.21.0，也可以在百度网盘下载，下载链接：<https://pan.baidu.com/s/1ka22UtzqFdOaIosrpKz92w>

提取码：8xh4

备用下载链接：https://pan.baidu.com/s/1XeA_8PQvvRePEdZ5ayOT-Q

提取码：8xh4

我们接下来以文档写作时的最新版为 8.0.21 为例，进行下载安装讲解。

在 mac 上直接双击 dmg 文件就可以开始安装了。我们只选择了安装 MySQL Community Server，因此安装过程较为简单。

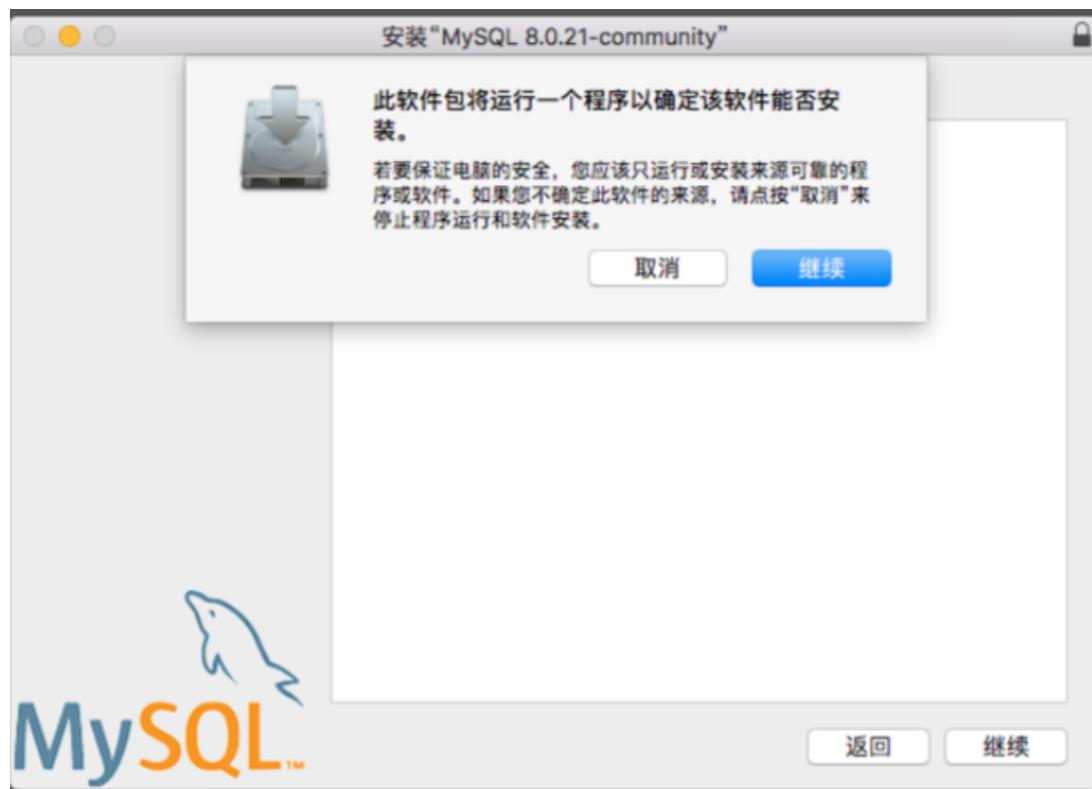


图 1.35: macOS_setup_5

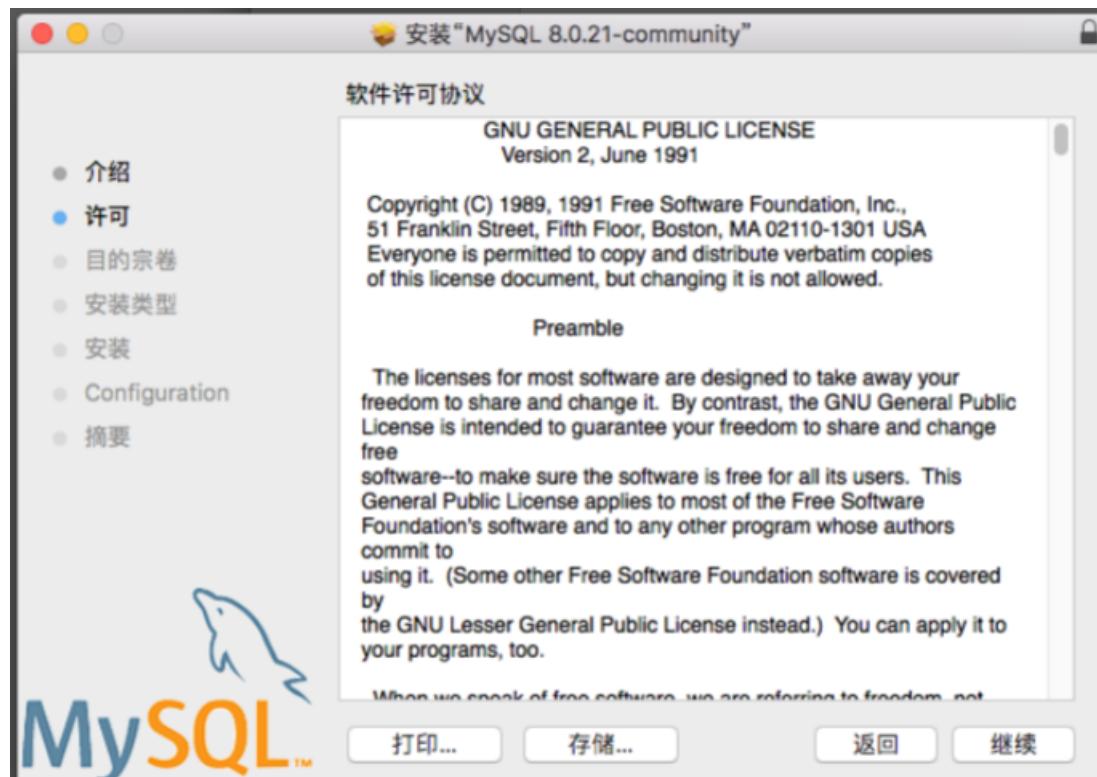


图 1.36: macOS_setup_6



图 1.37: macOS_setup_7

注意，在配置 MySQL 服务器这一步，最好选择下边的选项，这样就可以使用简单密码。



图 1.38: macOS_setup_8

在 Configuration 接下来的步骤需要设置密码，如果上一步选择了第二个选项，这里就可以使用比较简单的密码。请务必牢记自己设置的密码。

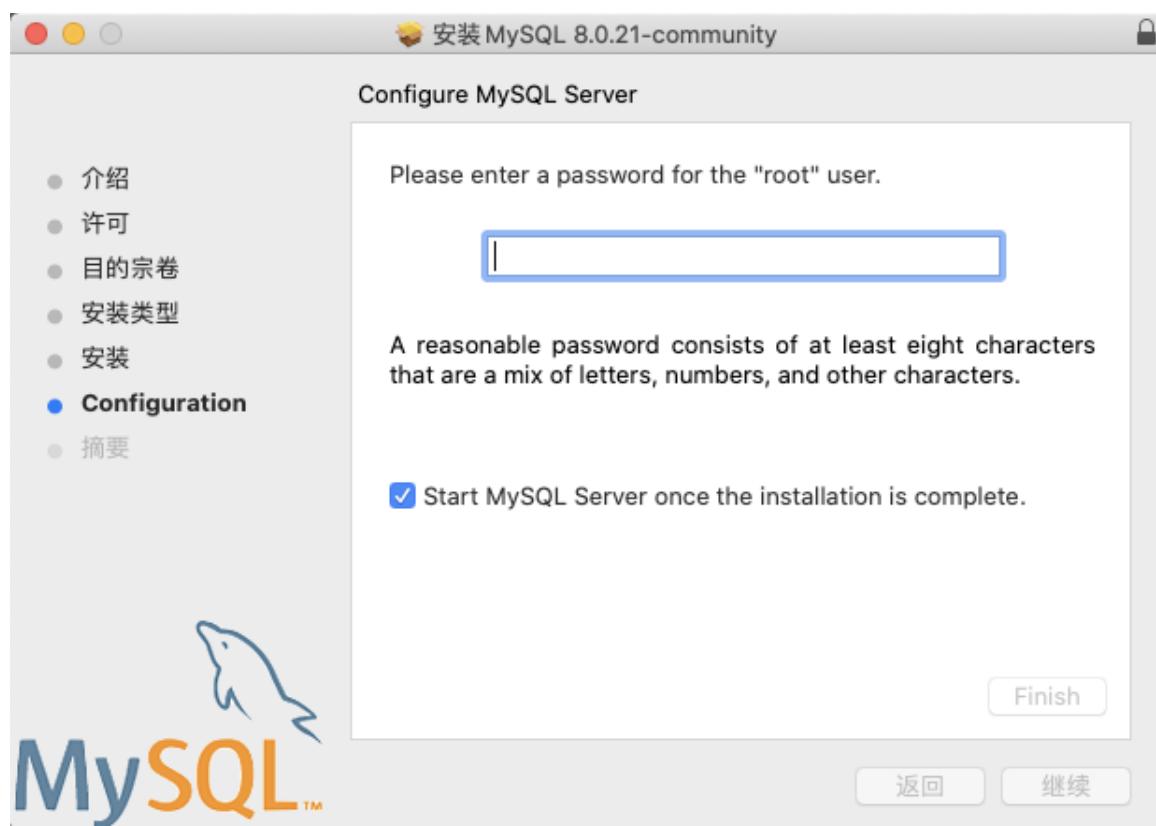


图 1.39: macOS_setup_9

注意，如果在前一步选择使用了 MySQL8.0 强密码，MySQL 安装过程中会自动生成一个随机密码，类似于下图这种形式：

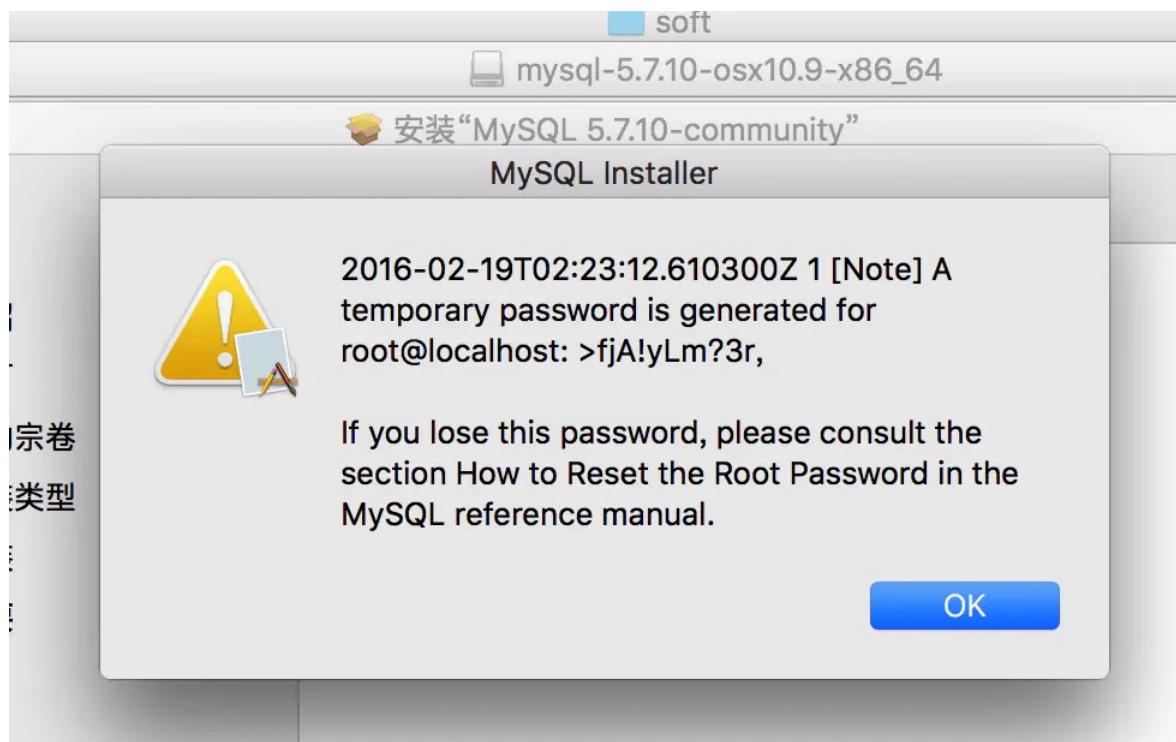


图 1.40: macOS_setup_10

此时请截图保留该密码，并在安装完成后，使用该密码登录并重新设置密码。由于选择使用了强密码，

此时设置密码必须使用大小写字母数字和特殊符号的组合。

完成安装后，打开电脑的系统偏好设置，会出现 MySQL 的服务标识。

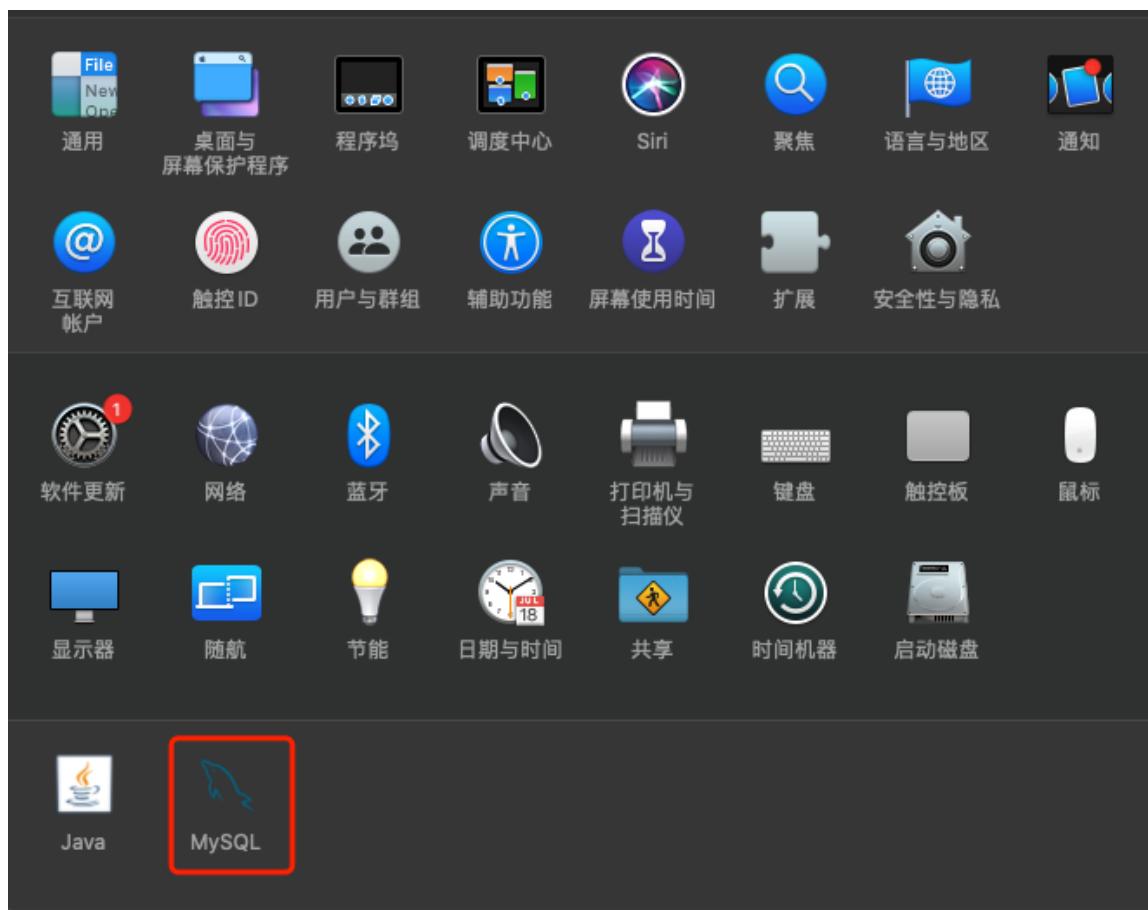


图 1.41: macOS_setup_11

正确安装 MySQL 后，打开上述截图中的 MySQL 之后会看到如下界面，在这里可以启动和停止 MySQL 服务，以及配置 MySQL 是否随电脑启动而自动启动。

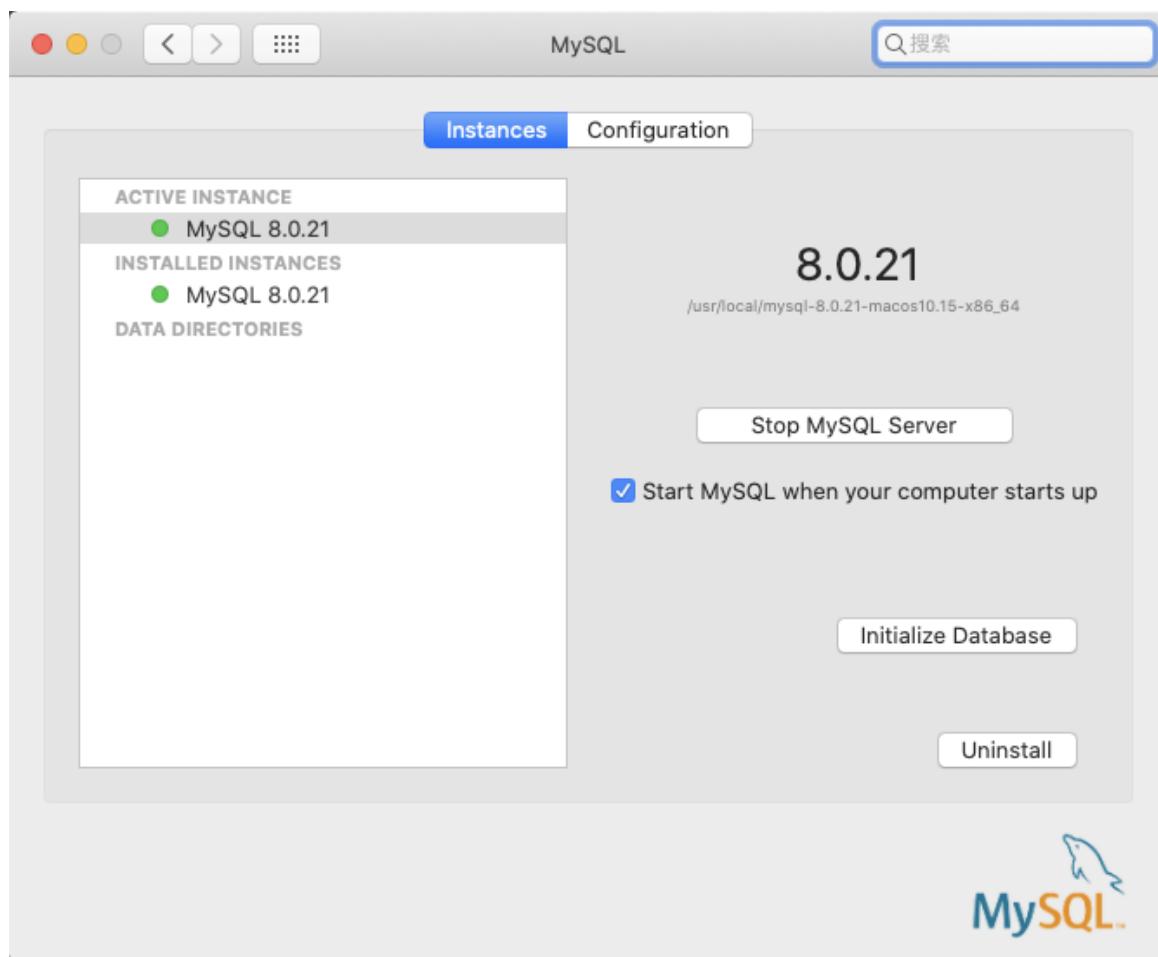
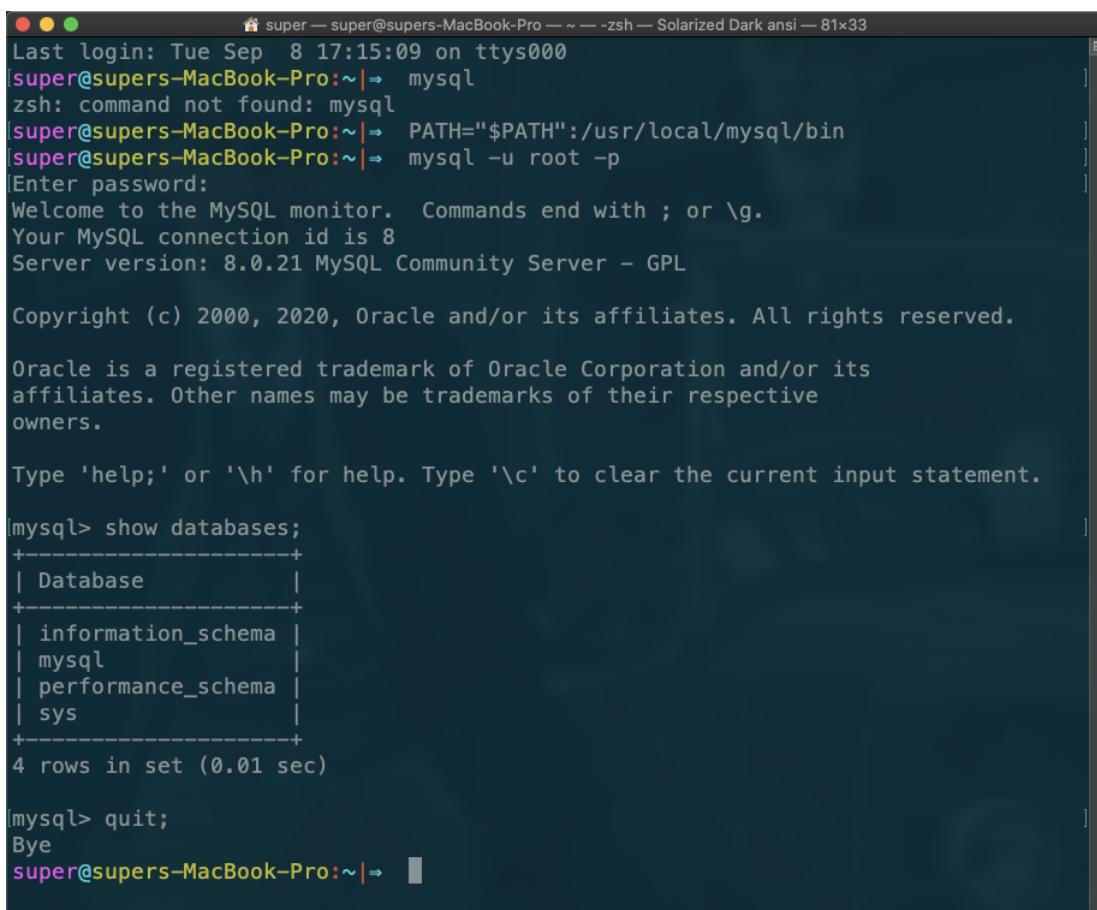


图 1.42: macOS_setup_12

安装之后，如果我们直接在终端输入 mysql，会提示找不到命令，需要配置一下环境变量才可以，输入以下命令：

```
PATH="$PATH": /usr/local/mysql/bin
```

再通过终端输入 mysql 登录命令后，就可以看到 mysql 的交互式界面了。



```

super — super@supers-MacBook-Pro — ~ — zsh — Solarized Dark ansi — 81x33
Last login: Tue Sep  8 17:15:09 on ttys000
super@supers-MacBook-Pro:~|> mysql
zsh: command not found: mysql
super@supers-MacBook-Pro:~|> PATH="$PATH":/usr/local/mysql/bin
super@supers-MacBook-Pro:~|> mysql -u root -p
[Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 8
Server version: 8.0.21 MySQL Community Server - GPL

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+-----+
| Database      |
+-----+
| information_schema |
| mysql          |
| performance_schema |
| sys            |
+-----+
4 rows in set (0.01 sec)

mysql> quit;
Bye
super@supers-MacBook-Pro:~|>

```

图 1.43: macOS_setup_13

然后输入以下内容，将自己电脑的 mysql 路径配置到环境变量中，如果在安装过程中你没修改过安装路径，那么你的电脑上 MySQL 的路径应该和下述代码中所使用的路径是一致的：

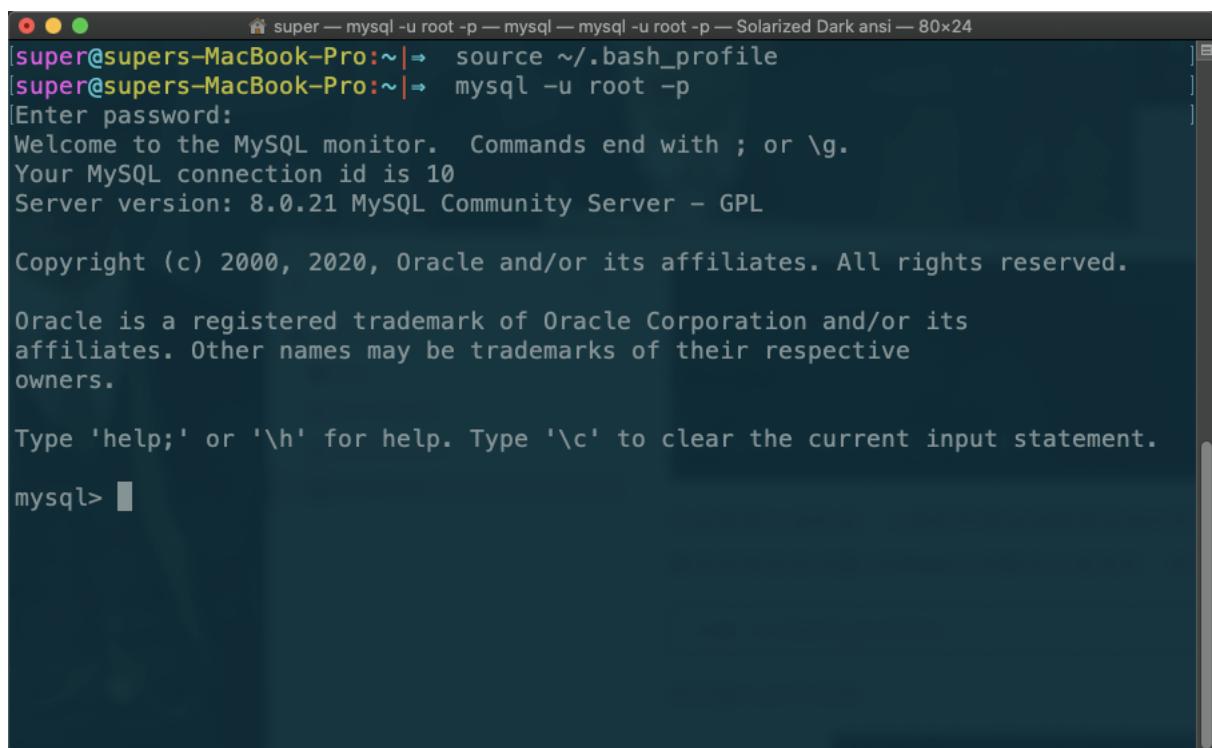
“plain export PATH=\$PATH:/usr/local/mysql/bin ““

```
export PATH=$PATH:/usr/local/mysql/bin
```

图 1.44: macOS_setup_14

运行上述配置后，通过输入下面这条命令使修改生效

“plain source /.bash_profile ““ 接下来，在你的终端中输入命令来登录到 MySQL “‘plain mysql -u root -p ““ 然后需要输入你刚才设置的密码（如果你选择了强密码，则这里需要输入你查到的随机密码），如果看到以下界面，则表示你的电脑上：1.MySQL8.0 已经正确安装，2. 已经正确配置了 MySQL8.0 的环境变量，并且 3. 已经启动 MySQL 服务器程序。如果并没有出现下属界面，请按照上述三个顺序逐个检查。



The screenshot shows a terminal window titled 'super — mysql -u root -p — mysql — mysql -u root -p — Solarized Dark ansi — 80x24'. The window displays the MySQL monitor welcome message, including the server version (8.0.21 MySQL Community Server - GPL), copyright information (Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.), and trademark notices (Oracle is a registered trademark of Oracle Corporation and/or its affiliates). It also includes help instructions and a prompt: 'mysql>'. The background of the terminal is dark, and the text is white.

图 1.45: macOS_setup_15

使用终端(或者 windows 下的命令行)与 MySQL 进行交互是非常便捷和高效的，但是对于平时不怎么使用终端的普通人来说，使用终端在做数据查询时，在查询结果的显示和导出方面有诸多不便，特别是当我们对 SQL 查询不熟练的时候，这种方式很不利于我们进行查询语句的调试。因此本教程将选择查询界面更加友好的客户端工具来连接数据库，这种通过终端连接 MySQL 的方式暂时不再使用。

接下来请安装跳转到 2.1 节安装 MySQL Workbench 并连接本机的 MySQL 服务。

> 注：

>macOS 上的 SQL 查询工具也有很多其他的选择，比如 DBeaver(开源，免费)，DataGrip(需付费购买，提供 30 天的免费试用期，学生使用学校的邮箱可以申请到一年的免费使用权) 等，可参考知乎上的问题：

>Mac 平台上有哪些好的 SQL 数据库开发工具？<https://www.zhihu.com/question/20498949>

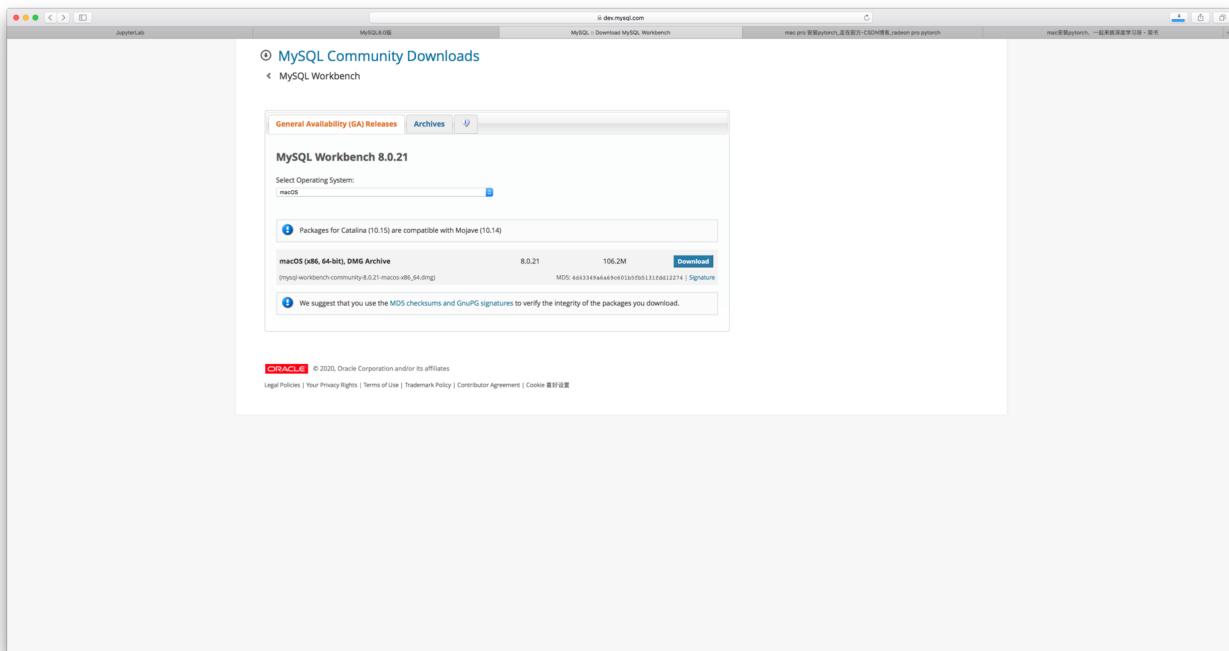


图 1.46: macOS_setup_16

1.1.3 Linux 下 MySQL 8.0 的下载安装

本节我们以 centos 版本的 Linux 操作系统为例，介绍下载安装 MySQL8.0.21 的过程。示例操作系统的 linux 版本：centos-release-7-7。1908.0.el7。centos。x86_64

在 CentOS 系统中默认是安装了 MariaDB 的，但是我们需要的是 MySQL，我们可以直接下载安装 MySQL，安装 MySQL 可以覆盖 MariaDB。

> 关于 MariaDB：

> MariaDB 数据库管理系统是 MySQL 的一个分支，主要由开源社区在维护，采用 GPL 授权许可。开发这个分支的原因之一是：甲骨文公司收购了 MySQL 后，有将 MySQL 闭源的潜在风险，因此社区采用分支的方式来避开这个风险。MariaDB 的目的是完全兼容 MySQL，包括 API 和命令行，使之能轻松成为 MySQL 的代替品。但在两个分支经过了几年的各自迭代之后，在一些方面二者出现了一些差异。

1.1.3.1 安装步骤：首先，从 MySQL 官网 下载 MySQL 的 Yum Repository。根据 CentOS 和 MySQL 的版本，选择下载相对应的文件。本文选择红色方框的文件。

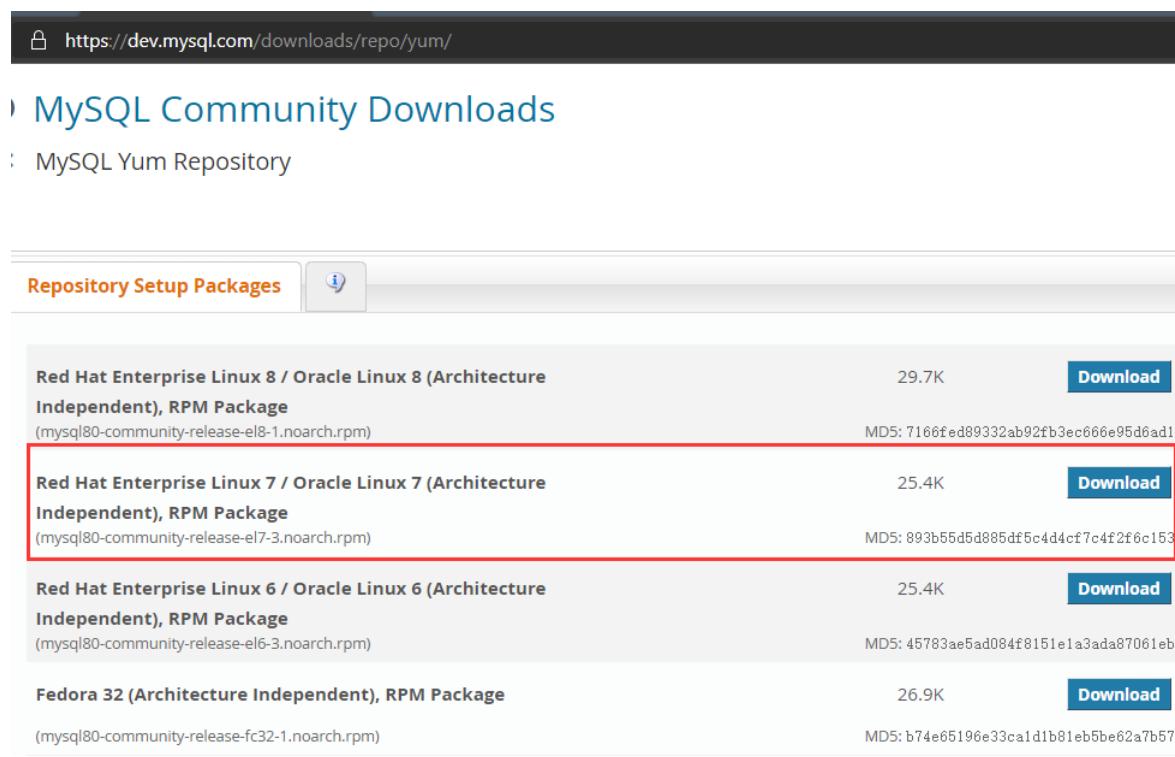


图 1.47: Linux_setup_1

Yum 是一个基于 rpm 的软件包管理器，它可以从指定的服务器自动下载 RPM 包并且安装，可以自动处理依赖性关系，并且一次安装所有依赖的软体包，无须繁琐地一次次下载、安装。

下载命令：

源码地址 [sql](#)

```
Line 1 wget https://dev.mysql.com/get/mysql80-community-release-el7-2.noarch.rpm
```

用 yum 命令安装下载好的 rpm 包。

源码地址 [sql](#)

```
Line 1 yum -y install mysql80-community-release-el7-2.noarch.rpm
```

安装 MySQL 服务器。

源码地址 [sql](#)

```
Line 1 yum -y install mysql-community-server
```

这一步由于要下载安装文件，可能会花一些时间。安装完成后就会覆盖掉之前的 mariadb。当出现如下图所示的内容，则代表 MySQL 就安装完成了。

```

Installed:
  mysql-community-devel.x86_64 0:8.0.15-1.el7
  mysql-community-libs.x86_64 0:8.0.15-1.el7
  mysql-community-libs-compat.x86_64 0:8.0.15-1.el7
  mysql-community-server.x86_64 0:8.0.15-1.el7

Dependency Installed:
  mysql-community-client.x86_64 0:8.0.15-1.el7
  mysql-community-common.x86_64 0:8.0.15-1.el7

Replaced:
  mariadb-devel.x86_64 1:5.5.60-1.el7_5    mariadb-libs.x86_64 1:5.5.60-1.el7_5

Complete!

```

覆盖掉之前的mariadb

图 1.48: Linux_setup_2

1.1.3.2 MySQL 数据库设置 启动 MySQL

源码地址 [sql](#)

```
Line 1  systemctl start mysqld.service
```

查看 MySQL 运行状态, Active 后面的状态代表启动服务后为 active (running), 停止后为 inactive (dead), 运行状态如图:

源码地址 [sql](#)

```
Line 1  systemctl status mysqld.service
```

```
[root@iZwz9awjmpog49qwoc2i0yZ local]# systemctl status mysqld.service
● mysqld.service - MySQL Server
  Loaded: loaded (/usr/lib/systemd/system/mysqld.service; enabled; vendor present)
  Active: active (running) since Tue 2019-04-16 16:14:22 CST; 12s ago
    Docs: man:mysqld(8)
          http://dev.mysql.com/doc/refman/en/using-systemd.html
   Process: 25565 ExecStartPre=/usr/bin/mysqld_pre_systemd (code=exited, status=0/SUCCESS)
   Main PID: 25640 (mysqld)
     Status: "SERVER_OPERATING"
    CGroup: /system.slice/mysqld.service
             └─25640 /usr/sbin/mysqld

Apr 16 16:14:02 iZwz9awjmpog49qwoc2i0yZ systemd[1]: Starting MySQL Server...
Apr 16 16:14:22 iZwz9awjmpog49qwoc2i0yZ systemd[1]: Started MySQL Server.
```

图 1.49: Linux_setup_3

重新启动 Mysql 和停止 Mysql 的命令如下:

源码地址 [sql](#)

```
Line 1  service mysqld restart #重新启动 Mysql
-  systemctl stop mysqld.service #停止 Mysql
```

此时 MySQL 已经开始正常运行, 不过要想进入 MySQL 还得先找出此时 root 用户的密码, 通过如下命令可以在日志文件中找出密码: 为了加强安全性, MySQL8.0 为 root 用户随机生成了一个密码, 在

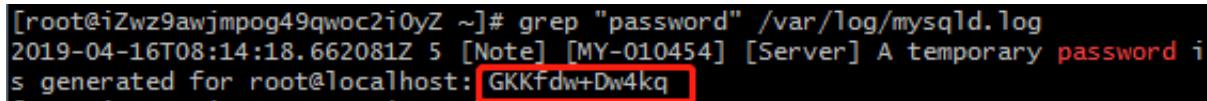
error log 中，关于 error log 的位置，如果安装的是 RPM 包，则默认是 /var/log/mysqld.log。只有启动过一次 mysql 才可以查看临时密码

使用命令：

源码地址 [sql](#)

```
Line 1 grep 'temporary_password' /var/log/mysqld.log
```

查看初始的随机密码：



```
[root@iZwz9awjmpog49qwoc2i0yZ ~]# grep "password" /var/log/mysqld.log
2019-04-16T08:14:18.662081Z 5 [Note] [MY-010454] [Server] A temporary password i
s generated for root@localhost: GKKfdw+Dw4kq
```

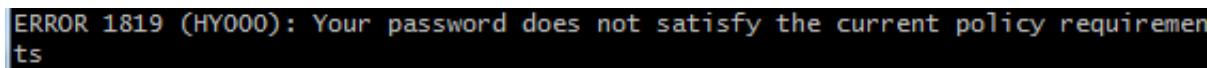
图 1.50: Linux_setup_4

登录 root 用户

源码地址 [sql](#)

```
Line 1 mysql -u root -p
```

然后输入上述查到的初始密码。登录后还不能做任何查询或建库操作，因为 MySQL 默认必须修改初始的随机密码之后才能操作数据库，否则会报错：



```
ERROR 1819 (HY000): Your password does not satisfy the current policy requirements
```

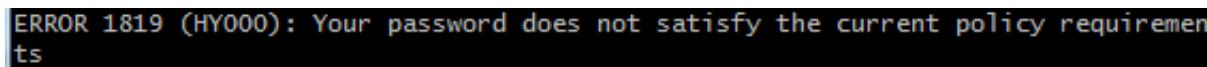
图 1.51: Linux_setup_5

修改密码为"123456"，注意结尾要有分号，表示语句的结束。

源码地址 [sql](#)

```
Line 1 ALTER USER 'root'@'localhost' IDENTIFIED BY '123456';
```

这里有个问题，新密码设置的时候如果设置的过于简单会报错：



```
ERROR 1819 (HY000): Your password does not satisfy the current policy requirements
```

图 1.52: Linux_setup_6

原因是因为 MySQL 有密码设置的规范，具体是与 validate_password_policy 的值有关：

Policy	Tests Performed
0 or LOW	Length
1 or MEDIUM	Length; numeric, lowercase/uppercase, and special characters
2 or STRONG	Length; numeric, lowercase/uppercase, and special characters; dictionary file

图 1.53: Linux_setup_7

MySQL 完整的初始密码规则可以通过如下命令查看：

Variable_name	Value
validate_password.check_user_name	ON
validate_password.dictionary_file	
validate_password.length	8
validate_password.mixed_case_count	1
validate_password.number_count	1
validate_password.policy	MEDIUM
validate_password.special_char_count	1

图 1.54: Linux_setup_8

密码的长度是由 validate_password_length 决定的，而 validate_password_length 的计算公式是：

源码地址 [sql](#)

```
Line 1 validate_password_length = validate_password_number_count +
    validate_password_special_char_count + (2 *
    validate_password_mixed_case_count)
```

如果想要设置简单的密码必须要修改约束，修改两个全局参数：

- **validate_password_policy** 代表密码策略，默认是 1：符合长度，且必须含有数字，小写或大写字母，特殊字符。设置为 0 判断密码的标准就基于密码的长度了。一定要先修改两个参数再修改密码

源码地址 [sql](#)

```
Line 1 mysql> set global validate_password.policy=0;
```

- **validate_password_length** 代表密码长度，最小值为 4

源码地址 [sql](#)

```
Line 1 mysql> set global validate_password.length=4;
```

修改完，如图

Variable_name	Value
validate_password.check_user_name	ON
validate_password.dictionary_file	
validate_password.length	4
validate_password.mixed_case_count	1
validate_password.number_count	1
validate_password.policy	LOW
validate_password.special_char_count	1

图 1.55: Linux_setup_9

此时密码就可以设置的很简单，例如 1234 之类的。到此数据库的密码设置就完成了。

但此时还有一个问题，就是因为安装了 Yum Repository，以后每次 yum 操作都会自动更新，如不需要更新，可以把这个 yum 卸载掉：

源码地址 [sql](#)

```
Line 1 [root@localhost ~]\$ section{yum -y remove mysql80-community-release-el7-2.  
noarch
```

在 CentOS 中 MySQL 的主要配置所在的目录:

源码地址 [sql](#)

```
Line 1 1 /etc/my.cnf 这是 mysql 的主配置文件  
- 2 /var/lib/mysql mysql 数据库的数据库文件存放位置  
- 3 /var/log mysql 数据库的日志输出存放位置
```

一些可能会用到的设置:

设置表名为大小写不敏感:

1. 用 root 帐号登录后, 使用命令

```
Line 1  
-  
- systemctl stop mysqld.service
```

停止 MySQL 数据库服务, 修改 vi /etc/my.cnf, 在 [mysqld] 下面添加

源码地址 [sql](#)

```
Line 1 lower_case_table_names=1
```

这里的参数 0 表示区分大小写, 1 表示不区分大小写。

2. 做好数据备份, 然后使用下述命令删除数据库数据 (删除后, 数据库将恢复到刚安装的状态)

源码地址 [sql](#)

```
Line 1 rm -rf /var/lib/mysql
```

3. 重启数据库, 此时数据库恢复到初始状态。

源码地址 [sql](#)

```
Line 1 service mysql start
```

4. 重复安装时的操作, 查找临时密码

源码地址 [sql](#)

```
Line 1 grep 'temporary_password' /var/log/mysqld.log
```

5. 修改密码。密码 8 位以上, 大小写符号数据。如想要使用简单密码, 需按照上述安装流程中的步骤进行操作。

源码地址 [sql](#)

```
Line 1 ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password BY '  
*****';update user set host = "%" where user='root';
```

6. 刷新 MySQL 的系统权限相关表

源码地址 [sql](#)

```
Line 1 FLUSH PRIVILEGES;
```

此时，MySQL 的表名的大小写不再敏感。

1.3.3 设置远程连接：

如果你想要在另外一台电脑上连接 centos 上安装的 MySQL，那么还需要一些其他的设置。

首先需要将 MySQL 设置为可以远程连接，设置 mysql 库的 user 表中帐号的 host 为

其次，MYSQL 8.0 内新增加 mysql_native_password 函数，通过更改这个函数密码来进行远程连接。

例如，更改 root 用户的 native_password 密码

源码地址 [sql](#)

```
Line 1 ALTER USER 'root'@'%' IDENTIFIED WITH mysql_native_password BY 'MyPass@123';
```

接下来为 centos 的防火墙开启 MySQL 所使用的 3306 端口，并重新加载防火墙：

源码地址 [sql](#)

```
Line 1 firewall-cmd --zone=public --add-port=3306/tcp --permanent  
- firewall-cmd --reload
```

完成上述设置后，重新启动 MySQL 服务：

源码地址 [sql](#)

```
Line 1 service mysqld restart #重新启动 Mysql
```

最后，在另外一台电脑上，使用下一节介绍的各类客户端工具进行连接测试。

1.2 连接 MySQL 并执行 SQL 查询

在安装成功 MySQL 后，我们可以通过开始菜单-> 控制面板-> 管理工具-> 服务中查找并开启或关闭 MySQL 服务。开启服务后，MySQL Server 将以后台服务的形式在你的电脑上运行。如果想要进行查询，还需要使用命令行工具或者其他更加友好的应用程序连接到 MySQL 服务。

1.2.1 使用命令行方式连接 MySQL 服务

连接数据库的最基本的方式，是使用命令行。以 win10 为例，在完成 MySQL 服务的安装后，在开始菜单找到刚才安装好的 MySQL 8.0 命令行工具，单击即可打开：

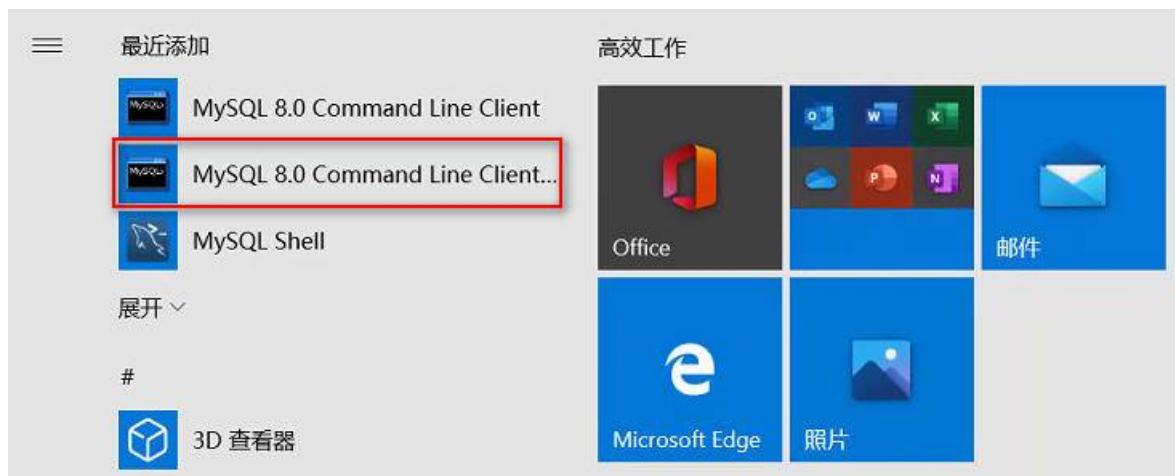


图 1.56: 使用命令行方式连接 MySQL 服务 1

输入你在安装过程中为 root 用户设置的密码即可连接到数据库，看到如下界面，表示数据库安装成功，并且你已经使用命令行连接到了数据库（图中红框表示数据库版本），在最后的"mysql>" 后即可输入 SQL 命令执行各种数据库操作。

```

MySQL 8.0 Command Line Client - Unicode
Enter password: *****
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 17
Server version: 8.0.21 MySQL Community Server - GPL

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> -

```

图 1.57: 使用命令行方式连接 MySQL 服务 2

但是使用命令行在做数据查询时，在查询结果的显示和导出方面有诸多不便，特别是当我们对 SQL 查询不熟练的时候，这种方式不利于我们进行查询语句的修改和调试。因此本教程将选择查询界面更加友好的客户端工具（使用 MySQL Workbench）来连接数据库，这种通过命令行连接的方式暂时不再使用。

> 常见的可用于连接 MySQL 数据库并进行查询等操作的管理工具包括开源软件 MySQL Workbench, HeidiSQL 和 DBeaver，以及商业软件 Navicat(有免费版和 14 天试用版)，SQLyog(有免费的社区版) 和 DataGrip 等等。

1.2.2 使用 MySQL Workbench 连接 MySQL

MySQL Workbench 是 MySQL 官方的客户端工具，支持 windows, macOS 和 Linux。对于 windows 用户，我们在安装 MySQL 的时候由于选择的是完整安装模式，因此也同时安装了这个工具，对于 macOS 的用户，可以在<https://dev.mysql.com/downloads/workbench/>选择 macOS 版本进行下载安装。MySQL

Workbench 是一款功能强大的数据库管理工具，它既可以用于设计数据库，也可以用于连接数据库进行查询，我们这个课程主要使用它的连接数据库进行查询的功能。

如果你使用的是 windows 7 操作系统，打开 MySQL Workbench 后，会有如下提示：



图 1.58: MySQL Workbench 1

这是说 MySQL 不再维护 win7 系统下的 MySQL Workbench，但并不影响使用。打开 MySQL Workbench 后，使用快捷键 **ctrl+u**，或者点击菜单栏里的 Database->Connect to Database，进入数据库连接的设置界面，如下图所示：

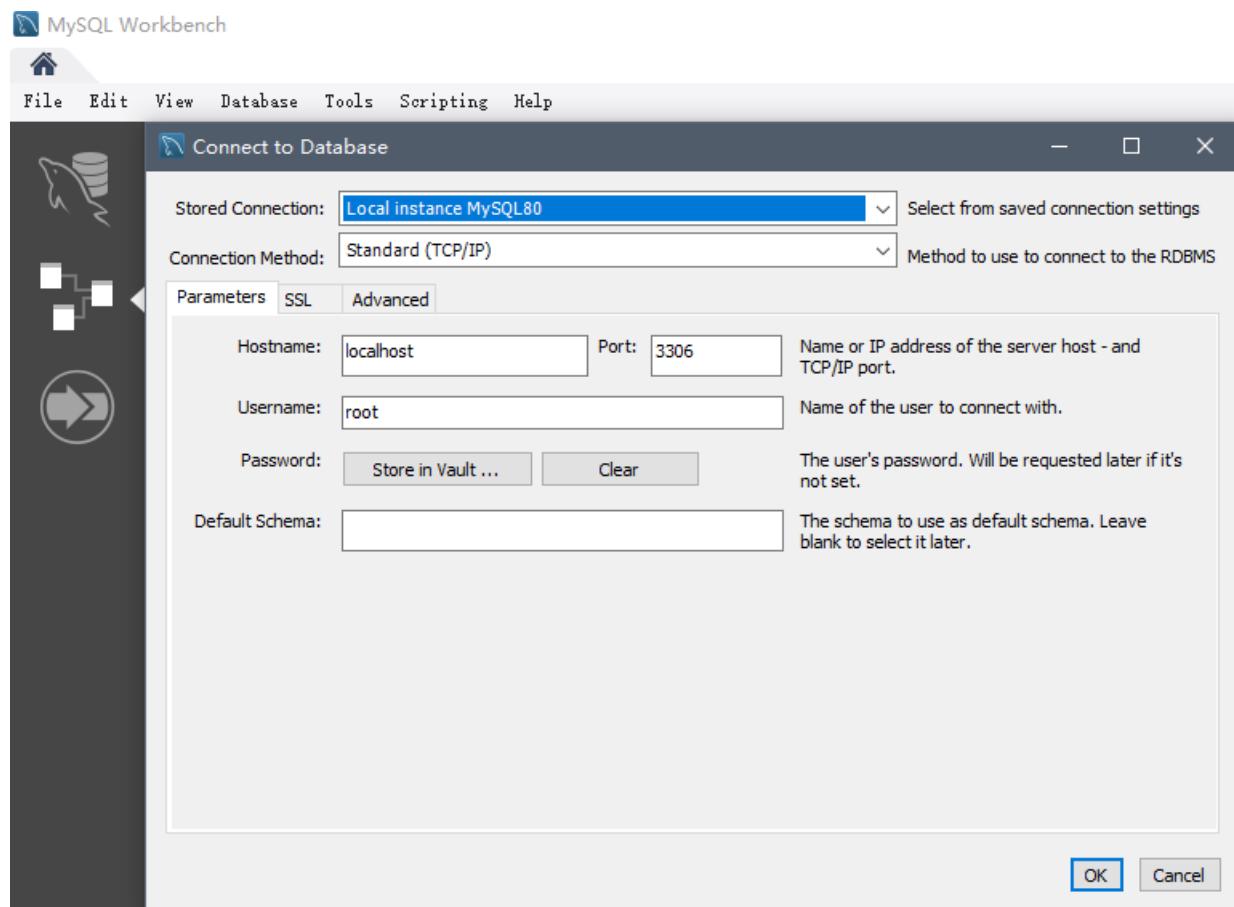


图 1.59: MySQL Workbench 2

然后如下图红框中设置连接本机的 MySQL8.0，输入密码并点击 OK：

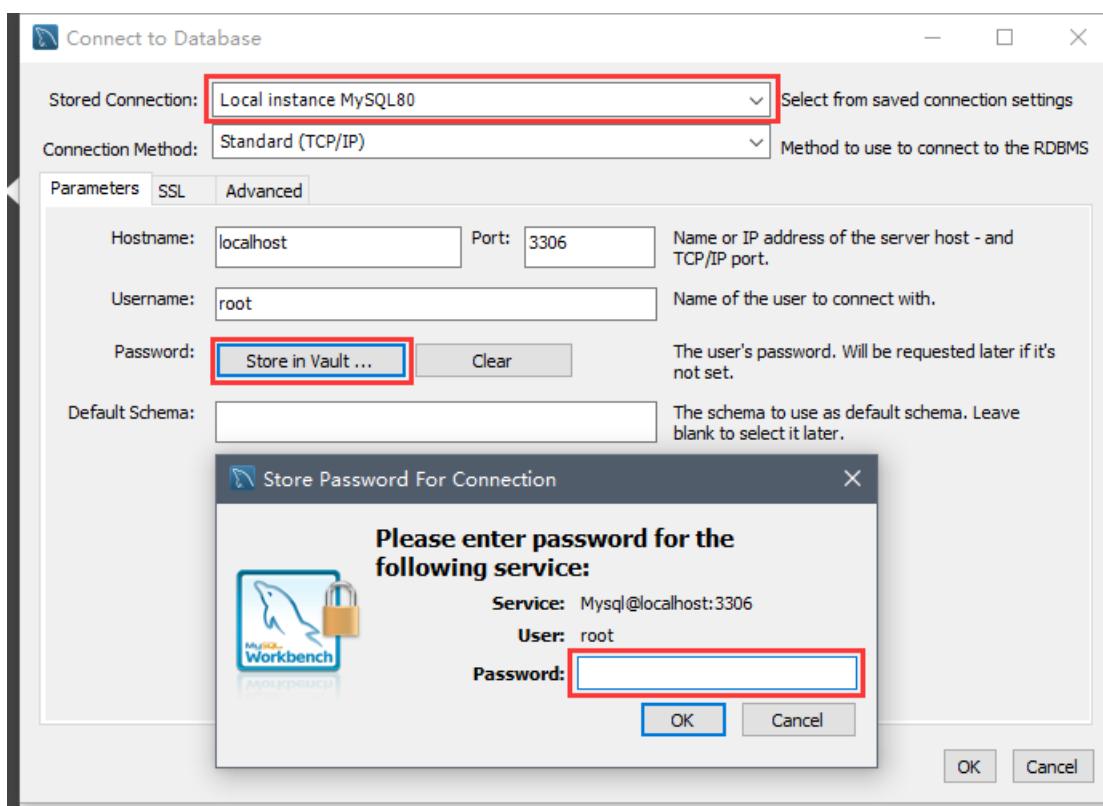


图 1.60: MySQL Workbench 3

如果弹出如下对话框，表示 MySQL 服务器设置为对于表名和列名的大小写敏感，但由于 windows 系统默认的是大小写不敏感，因此当表名或列名有大写字母时，即使写的 SELECT 语句中正确地使用了大写的表名，程序也无法正确执行。这时候建议大家在开始学习查询之前，把表名逐一更改为小写。

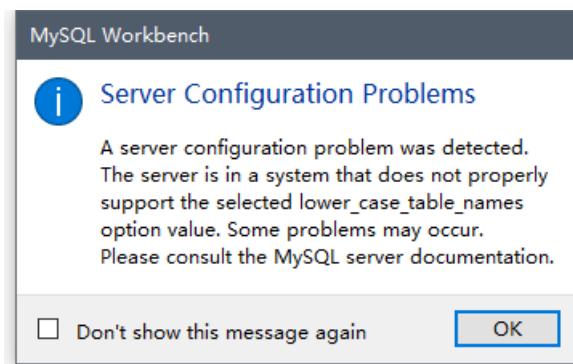


图 1.61: MySQL Workbench 4

打开后的界面

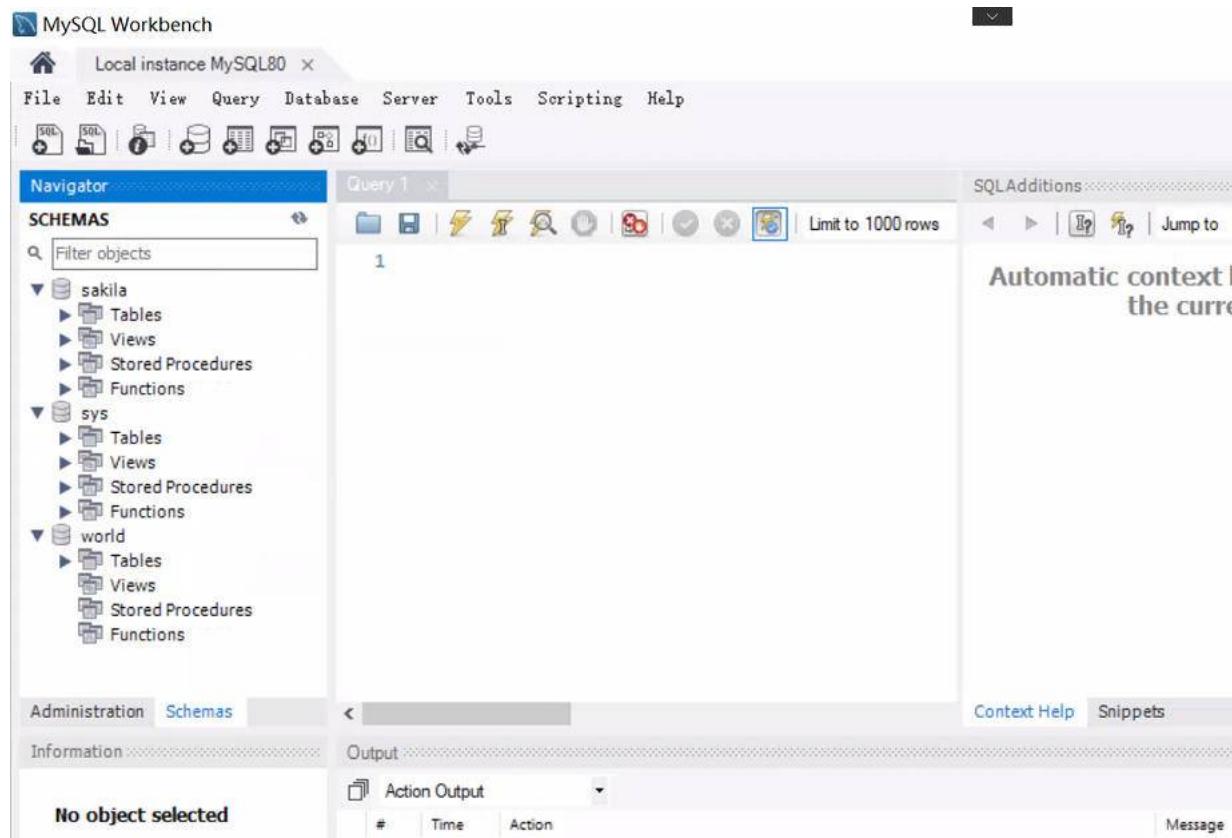


图 1.62: MySQL Workbench 5

在中间的编辑器里，就可以写 SQL 查询了。

下一次打开软件时，可以直接点击保存的数据库连接，不需要先进行设置后再连接了：

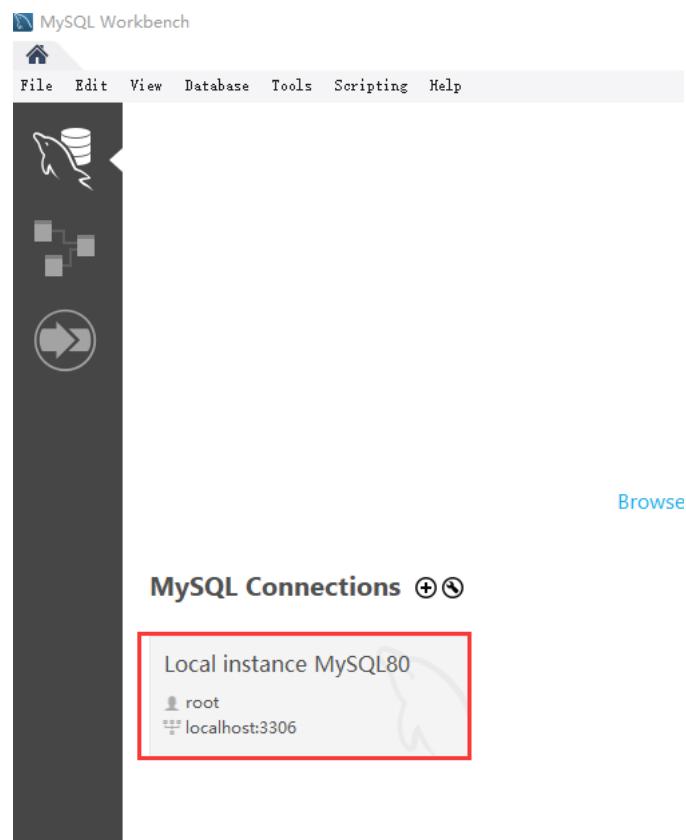


图 1.63: MySQL Workbench 6

1.2.3 [选学] 使用 HeidiSQL 连接 MySQL

HeidiSQL 是一款功能非常强大的开源免费的数据库客户端软件，采用 Delphi 语言开发，支持 Windows 操作系统。支持 MySQL、Postgresql、MariaDB、Percona Server 和微软的 SQL Server，官网下载地址：<https://www.heidisql.com/download.php>，下载 portable 版本后，解压缩就可以使用。

名称	修改日期	类型	大小
Backups	2020-09-07 18:40	文件夹	
plugins	2020-09-07 18:39	文件夹	
Snippets	2020-09-07 18:39	文件夹	
gpl.txt	2017-11-02 11:51	文本文档	18 KB
heidisql.exe	2020-03-17 19:05	应用程序	9,010 KB
libcrypto-1_1-x64.dll	2020-02-03 20:57	应用程序扩展	2,639 KB
libiconv-2.dll	2020-02-03 20:57	应用程序扩展	1,651 KB
libintl-8.dll	2020-02-03 20:57	应用程序扩展	670 KB
libmariadb.dll	2020-02-04 19:21	应用程序扩展	1,050 KB
libmysql.dll	2018-01-09 21:04	应用程序扩展	4,637 KB
libmysql-6.1.dll	2019-07-23 18:43	应用程序扩展	4,765 KB
libpq-10.dll	2020-02-03 20:57	应用程序扩展	278 KB
libpq-12.dll	2020-02-03 20:57	应用程序扩展	287 KB
libssl-1_1-x64.dll	2020-02-03 20:57	应用程序扩展	628 KB
license.txt	2020-02-03 20:57	文本文档	2 KB
portable.lock	2020-03-17 19:20	LOCK 文件	0 KB
portable_settings.txt	2020-09-07 18:45	文本文档	4 KB
sqlite3.dll	2019-12-24 14:32	应用程序扩展	1,855 KB
tabs.ini	2020-09-07 18:42	配置设置	1 KB

图 1.64: 使用 HeidiSQL 连接 MySQL 1

点击文件夹中的 heidisql.exe，在弹出对话框里点击“新建”，然后填写密码，再点击“打开”即可连接到本机上安装的 MySQL 数据库。

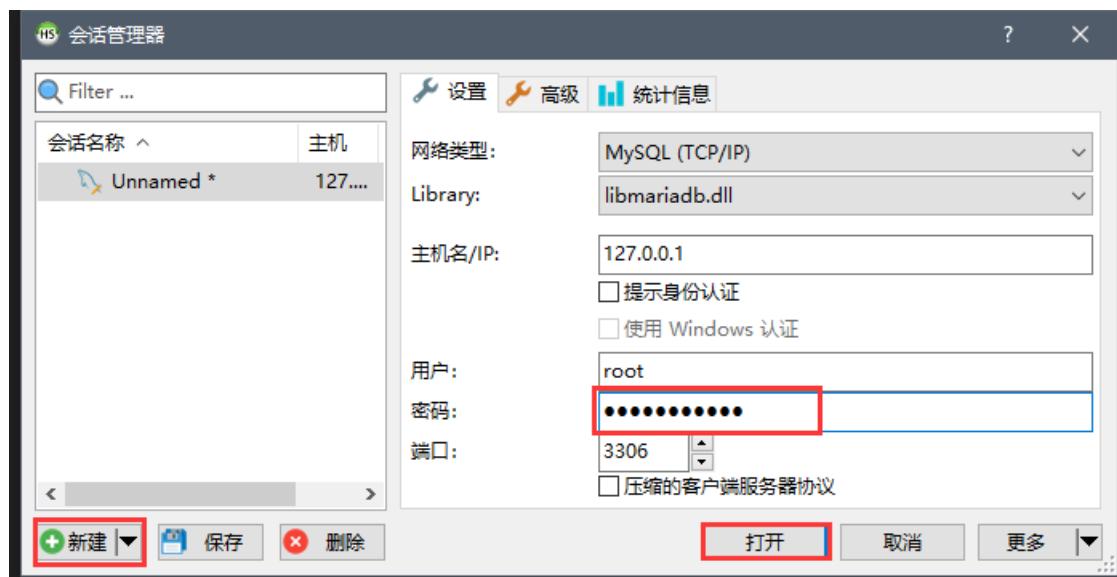


图 1.65: 使用 HeidiSQL 连接 MySQL 2

软件的主界面如图所示：

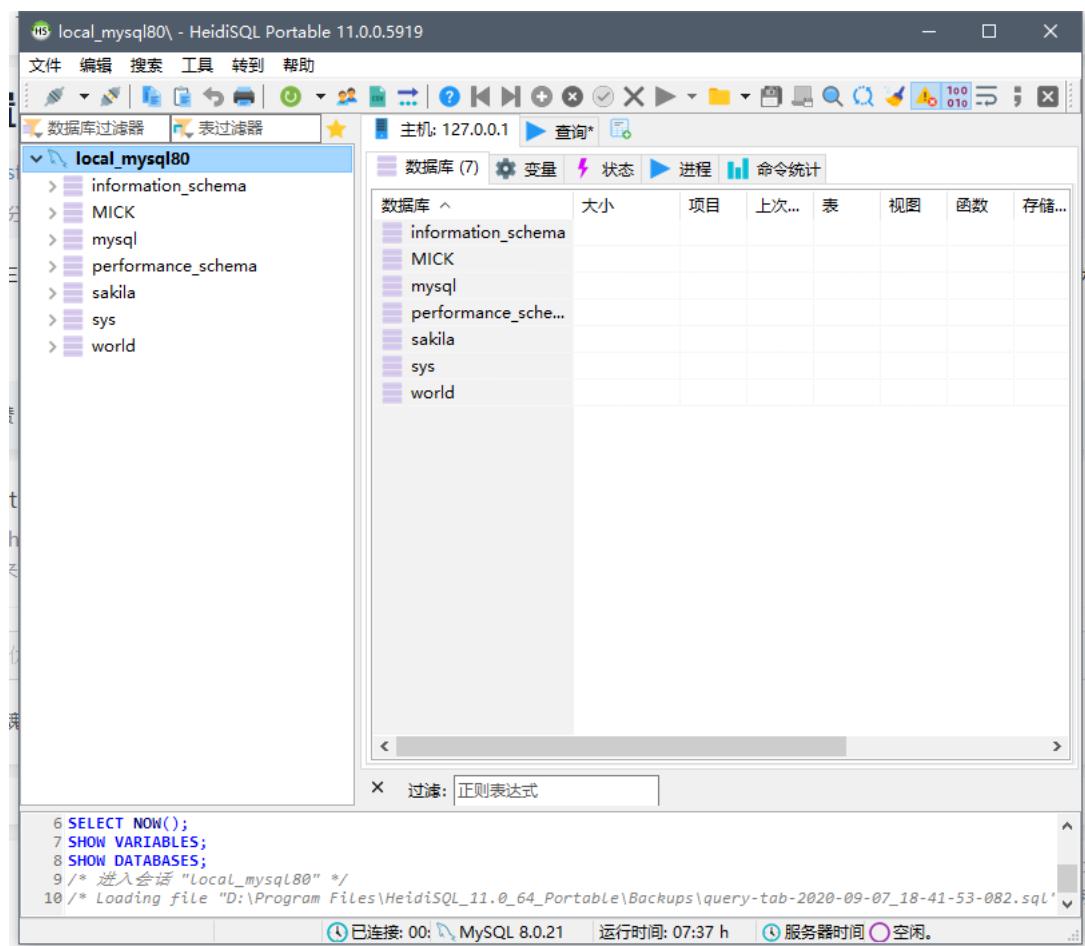


图 1.66: 使用 HeidiSQL 连接 MySQL 3

1.2.4 [选学] 使用 DBeaver 连接 MySQL

DBeaver 是一款基于 JAV 开发的**免费和开源 (GPL)** 的通用数据库管理工具和 SQL 客户端，提供 windows, macOS 和 Linux 全平台支持，能够连接包括 MySQL, PostgreSQL, Oracle, DB2, MSSQL, Sybase, Mimer, HSQLDB, Derby 等主流数据库软件在内的绝大多数兼容 JDBC 驱动的数据库。DBeaver 提供一个图形界面用来查看数据库结构、执行 SQL 查询和脚本，浏览和导出数据，处理 BLOB/CLOB 数据，修改数据库结构等等。

由于是开源软件，大家可直接从官网<https://dbeaver.io/>下载，安装完成后，打开软件，点击“文件”菜单下的“新建连接”图标，并选择 MySQL：

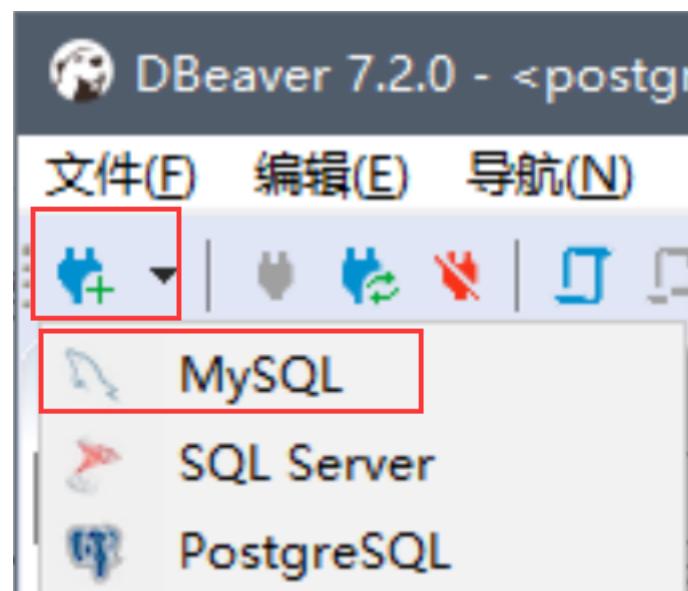


图 1.67: 使用 DBeaver 连接 MySQL 1

然后点击下一步：

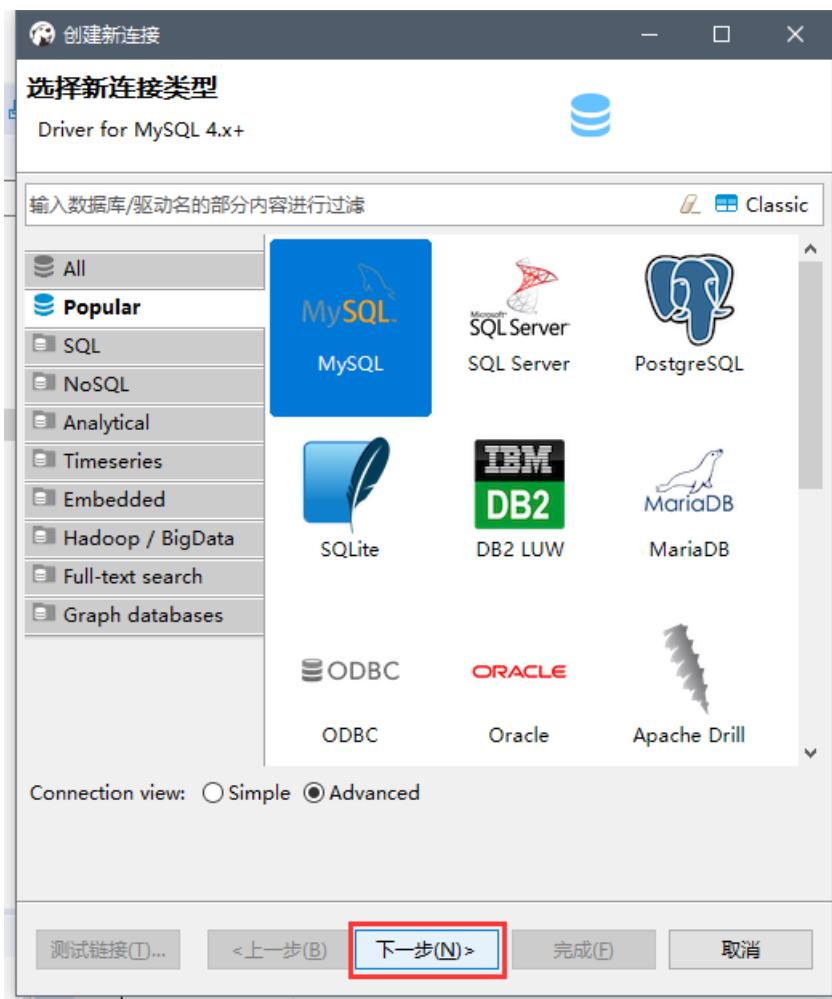


图 1.68: 使用 DBeaver 连接 MySQL 2

在下方弹出的设置窗口中，"服务器地址" 使用默认的 localhost，"数据库" 暂时留空，然后在下方填

入自己设置的密码，最后点击测试连接。

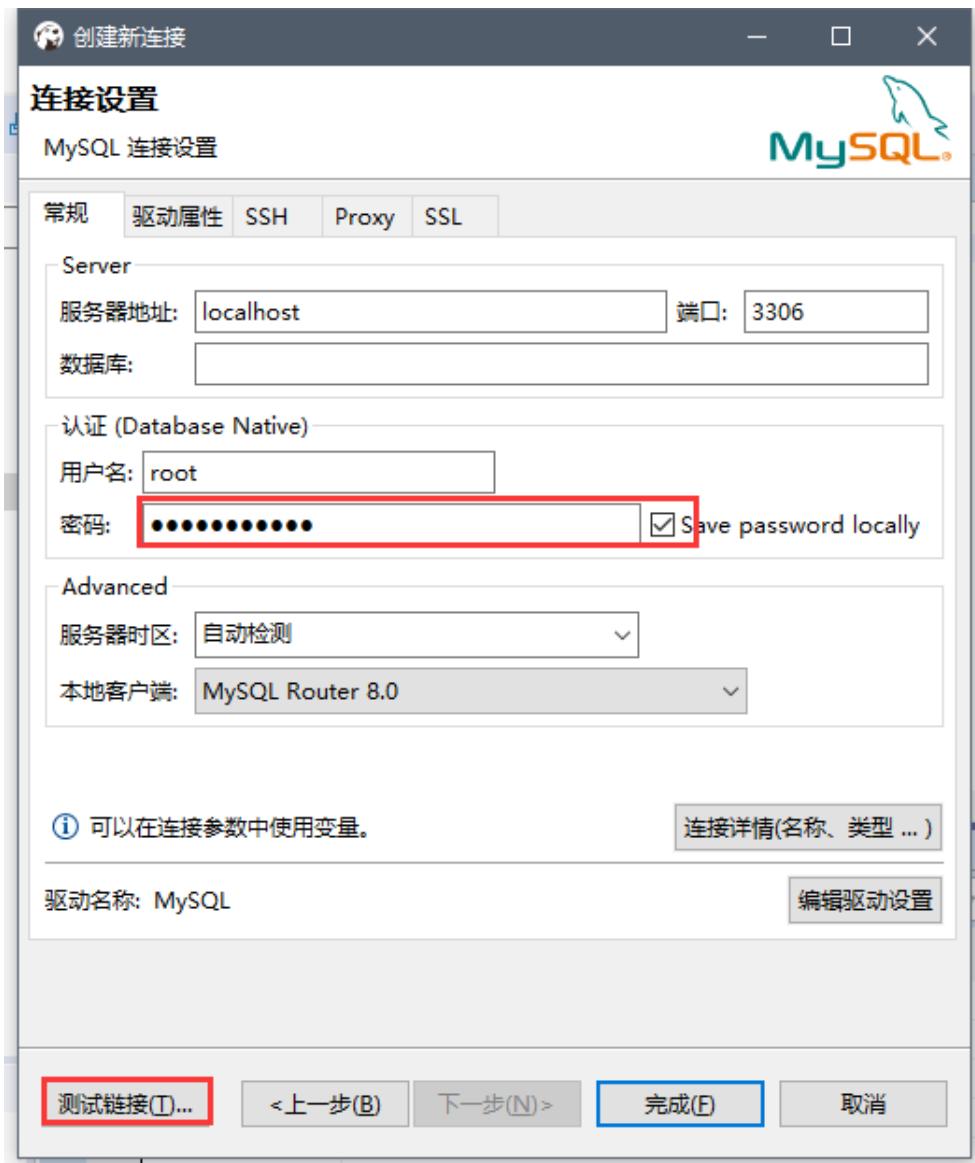


图 1.69: 使用 DBeaver 连接 MySQL 3

本次学习主要使用一个新建的 shop 数据库，但因为相关数据我们还没导入（建库建表及数据导入的脚本在本章第三节），因此数据库这里先留空。等下一步导入数据后，再进一步修改连接参数。

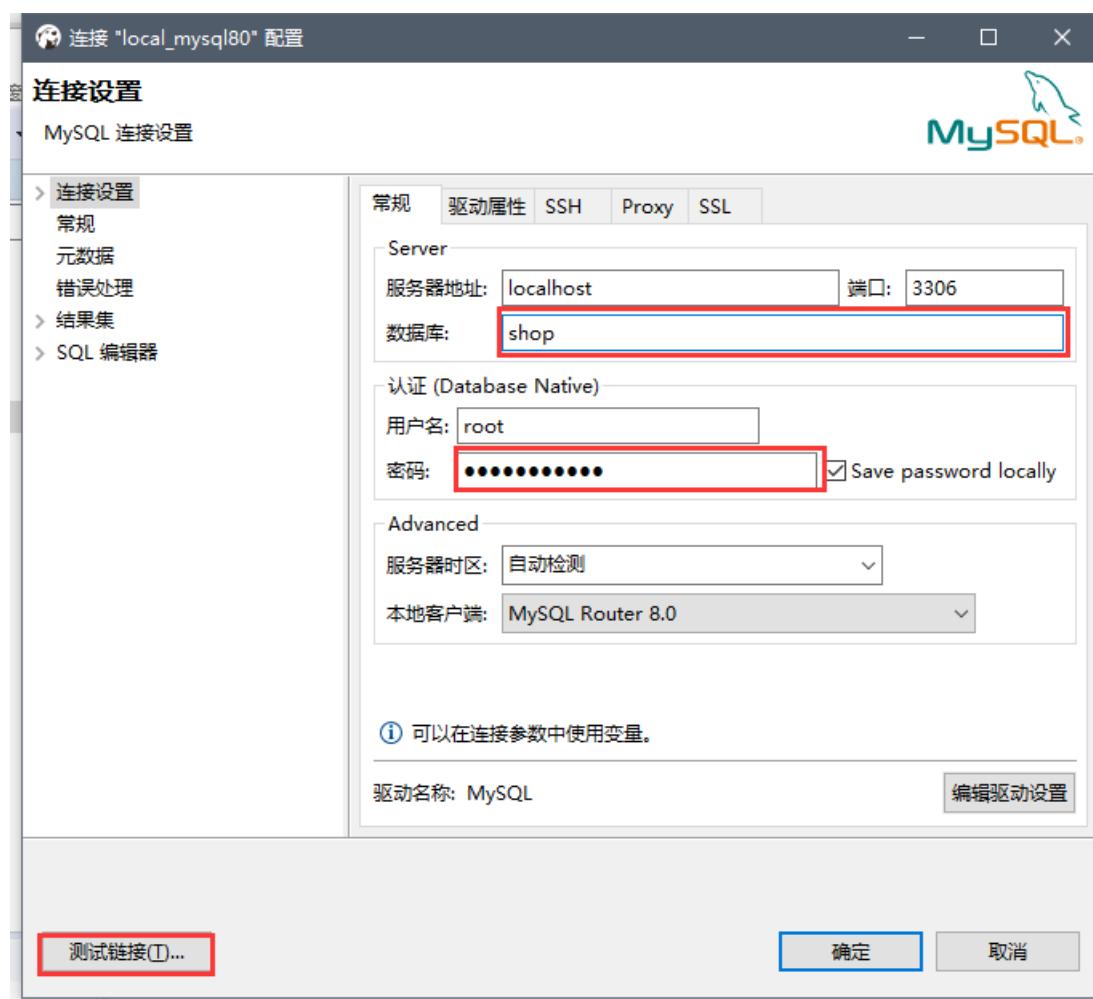


图 1.70: 使用 DBeaver 连接 MySQL 4

首次连接时会提示需要下载驱动程序，

完成驱动程序的下载后，再次点击“测试连接”，如果弹出如下对话框，则表示连接成功：

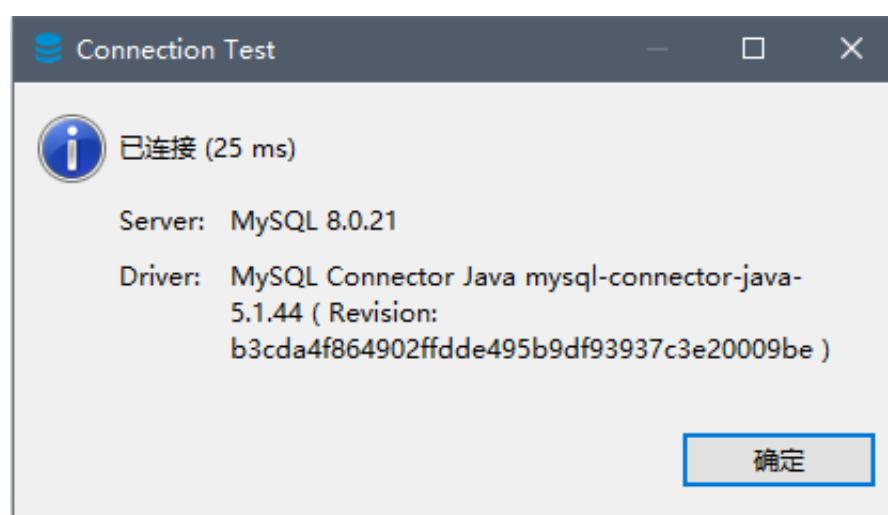


图 1.71: 使用 DBeaver 连接 MySQL 5

点击确定并保存连接后，以后就可以直接双击这个连接进行数据库连接了。如果需要保存多个数据库连接，可以使用快捷键 F2 或选中当前连接并点击鼠标右键后选择“重命名”为当前连接起一个便于识别

的名称。

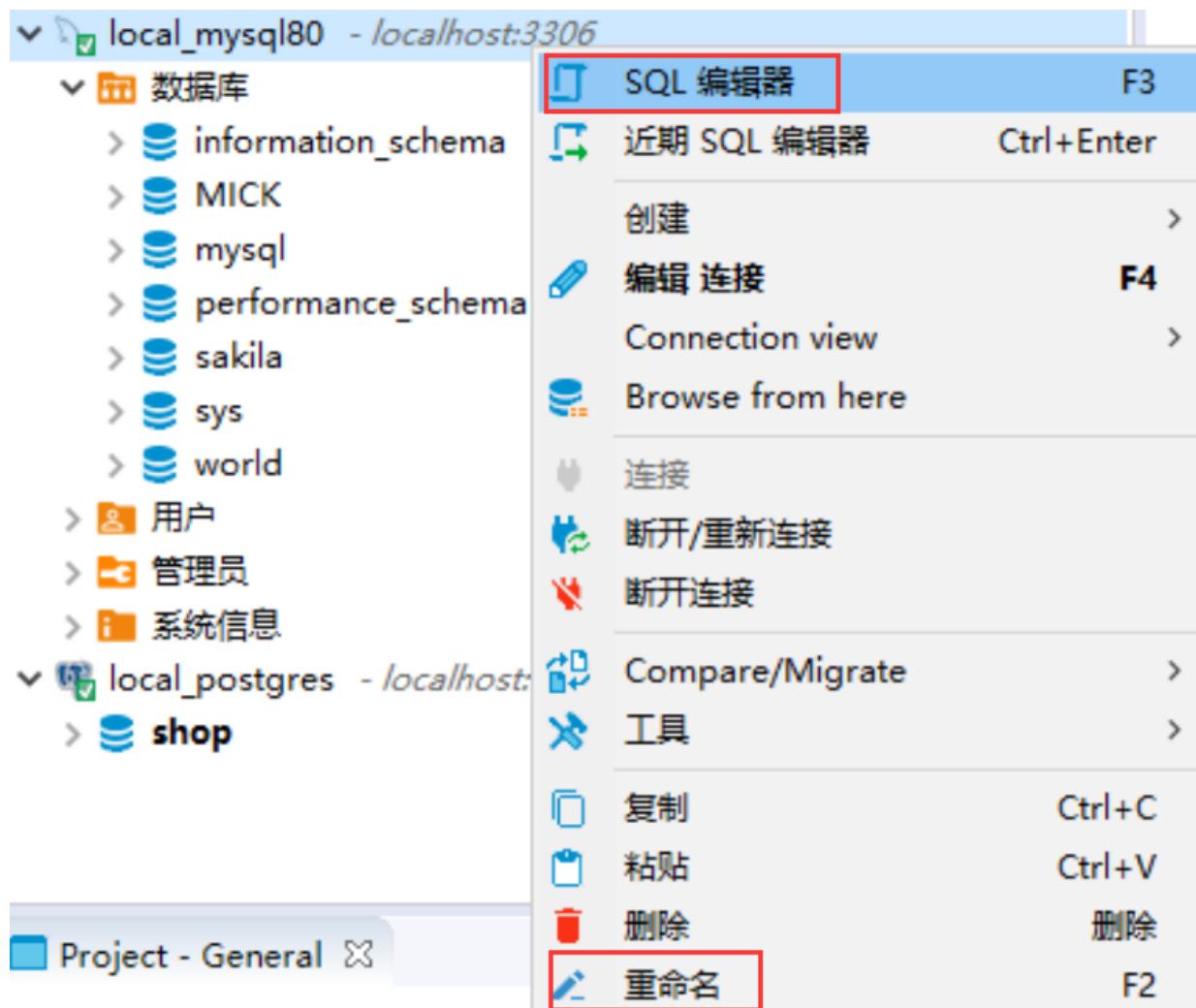


图 1.72: 使用 DBeaver 连接 MySQL 6

接下来就可以开始写你的第一个 SQL 语句了：如上图这样，选中刚刚保存的连接，鼠标右键选第一个按钮“SQL 编辑器”，或者直接使用快捷键 F3，就会打开一个 SQL 编辑器，然后就可以在这里编写 SQL 语句了。

例如我们可以使用如下语句完成本教程所使用的示例数据库的创建：

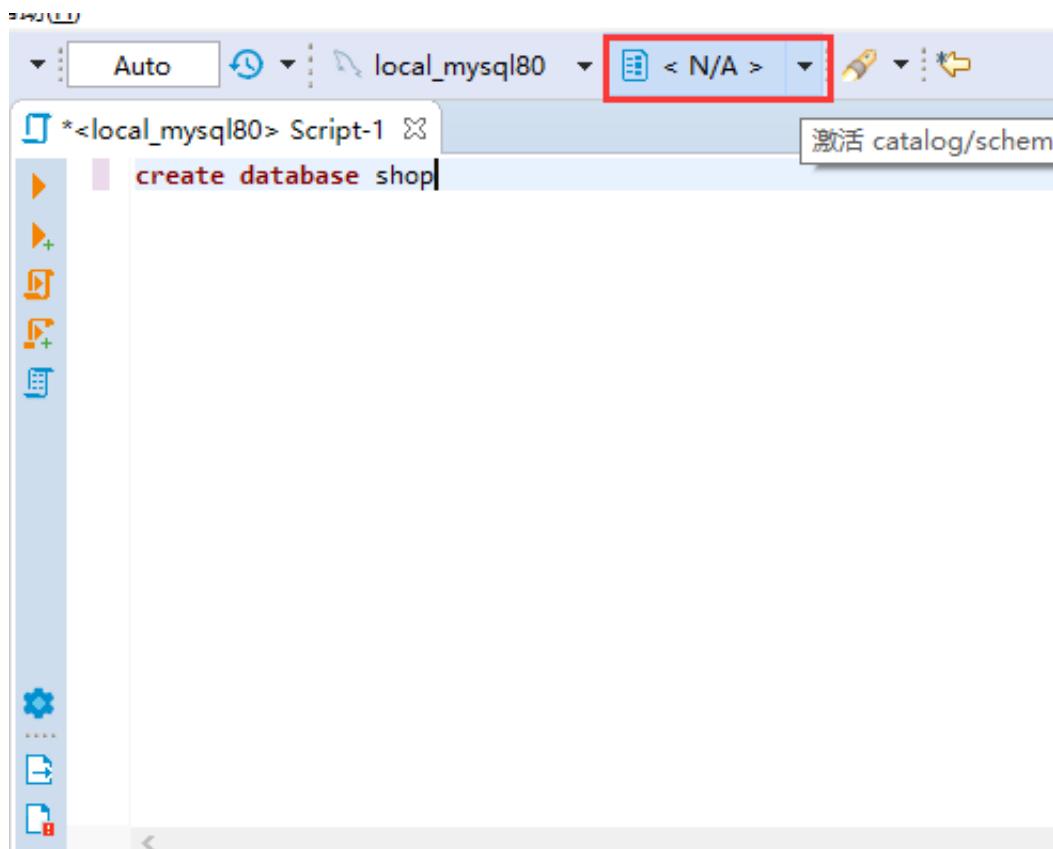


图 1.73: 使用 DBeaver 连接 MySQL 7

注意，在使用创建数据库的语句时，是无需在红框中选中数据库的，但其他所有的创建表，导入数据，和最常用的查询语句，都需要选中相应的数据库。在执行上述语句后，我们可以回到连接设置里，把默认数据库连接改为刚才创建的 shop。

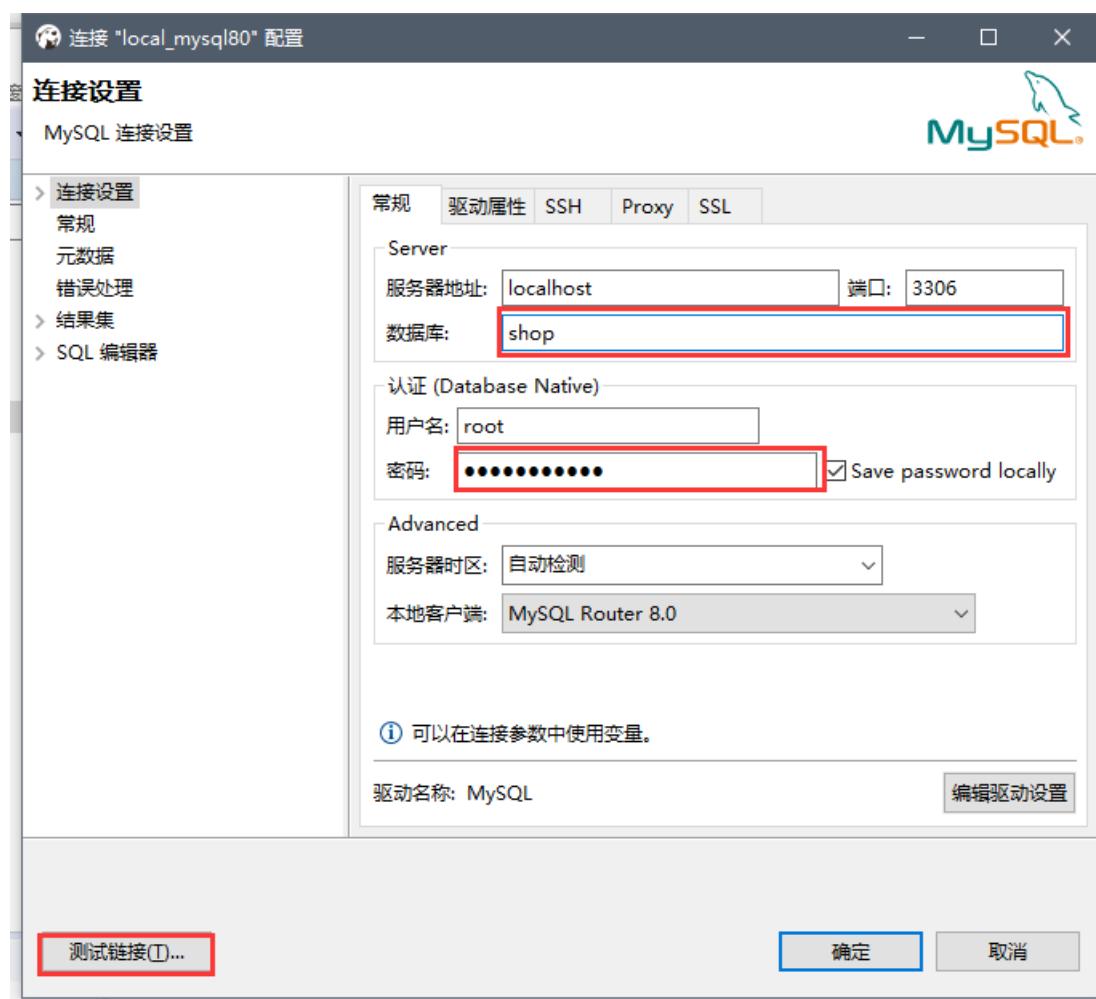


图 1.74: 使用 DBeaver 连接 MySQL 8

1.2.5 [选学] 使用 Navicat 连接 MySQL

Navicat 包含了一系列的功能强大的数据库管理软件，主要有：完整功能版的 Navicat Premium，以及专门用于 MySQL 数据库管理的 Navicat for MySQL，用于 PostgreSQL 数据库管理的 Navicat for PostgreSQL，用于 SQL Server 数据库管理的 Navicat for SQL Server，用于 Oracle 数据库管理的 Navicat for Oracle，等等，但它提供的免费的版本 Navicat Lite 已足够本次课程使用，除此之外，上述的其他软件均为收费软件。

> 注：

> Navicat 官网已经不再提供 Navicat Lite 的下载了，但可以通过搜索引擎找到 Navicat Lite 的历史版本的下载链接。此外，也可以从 Navicat 官网下载 Navicat Premium 或 Navicat for MySQL 的 14 天试用版。

1.2.6 [选学] 使用 SQLyog 连接 MySQL

SQLyog 是业界著名的 Webyog 公司出品的一款简洁高效、功能强大的图形化 MySQL 数据库管理工具。SQLyog 的企业版是收费软件，但该软件也提供了社区版供大家使用，虽然在功能上有些限制，但对于本课程已经足够用了。

SQLyog 社区版的下载地址为：<https://github.com/webyog/sqlyog-community/wiki/Downloads>

1.2.7 [选学]DataGrip 的安装和连接 MySQL

DataGrip 是大名鼎鼎的 JetBrains 出品的数据库工具，支持 windows, macOS 和 Linux 操作系统。

1.3 创建学习用的数据库

根据《SQL 基础教程》提供的 MySQL 版本的数据库，数据表的创建以及数据导入的代码，经过一些修改，创建了一份 sql 脚本，运行该脚本可以一步到位地创建本文档所需的数据库 shop 及其中所有的表，并插入本教程所需要的所有数据

由于本教程聚焦于面向初学者介绍 SQL 查询，对于数据库的创建，表的创建和数据导入，以及数据更新，暂时不做深入介绍，有兴趣和需要的读者可参考《SQL 基础教程》1-4, 1-5，以及第四章。

下述 SQL 脚本可用于创建本教程所使用的示例数据库 shop 以及数据库中表的创建和数据的插入。

见《附录 3 - shop.sql》

下载链接: <https://pan.baidu.com/s/1FOsWKC8Jd-dbSRKFap588Q>

提取码: gxzt

SQL 脚本的一些要点 – v 9.08

1. 存储引擎使用 InnoDB，字符集改为 utf8mb4 以更好地支持中文。
2. 所有表名所使用的英文字母都改为小写 (后续章节中，SQL 查询中的表名也需要相应修改为小写)
3. 所有列名所使用的英文字母确认为小写 (后续章节中，SQL 查询中的列名也需要相应修改为小写)
4. 存在问题的数据，例如 inventoryproduct 表的 inventory_id 列应为 P 开头的，已修正为正确的数据。
5. 需测试 SQL 脚本在命令行及各个客户端中是否能被正确执行。
 - MySQL Workbench 已测试通过 (在 win10 使用 MySQL Workbench 8.0.21)
 - DBeaver 已测试通过 (使用" 执行 SQL 脚本 (CTR+x)") (在 win10 使用 DBeaver7.2.0)
 - HeidiSQL 已测试通过 (在 win10 使用 HeidiSQL 11.0.0)
 - navicat 已测试通过 (在 win10&win7 使用 navicat premium 15.0.17)
 - sqlyog 已测试通过 (在 win10 使用 SQLyog 12.09)
 - 命令行 win10 下测试未通过。插入中文数据时提示" Data too long for column 'product_name' at row 1"

第 2 章 初识数据库

【声明】本教程有些截图来自图灵出版社出版的《SQL 基础教程第 2 版》，截图仅用来学习使用，已获得出版社授权。

本章主要对数据库进行基本介绍，考虑易用性及普及度，课程主要使用 MySql 进行介绍。

2.1 初始数据库

数据库是将大量数据保存起来，通过计算机加工而成的可以进行高效访问的数据集合。该数据集合称为数据库（Database, DB）。用来管理数据库的计算机系统称为数据库管理系统（Database Management System, DBMS）。

2.1.1 DBMS 的种类

DBMS 主要通过数据的保存格式（数据库的种类）来进行分类，现阶段主要有以下 5 种类型。

- 层次数据库（Hierarchical Database, HDB）
- 关系数据库（Relational Database, RDB）
 - Oracle Database: 甲骨文公司的 RDBMS
 - SQL Server: 微软公司的 RDBMS
 - DB2: IBM 公司的 RDBMS
 - PostgreSQL: 开源的 RDBMS
 - MySQL: 开源的 RDBMS

如上是 5 种具有代表性的 RDBMS，其特点是由行和列组成的二维表来管理数据，这种类型的 DBMS 称为关系数据库管理系统（Relational Database Management System, RDBMS）。

- 面向对象数据库（Object Oriented Database, OODB）
- XML 数据库（XML Database, XMLDB）
- 键值存储系统（Key-Value Store, KVS），举例：MongoDB

本课程将向大家介绍使用 SQL 语言的数据库管理系统，也就是关系数据库管理系统（RDBMS）的操作方法。

2.1.2 RDBMS 的常见系统结构

使用 RDBMS 时，最常见的系统结构就是客户端 / 服务器类型（C/S 类型）这种结构。

图 1-3 使用 RDBMS 时的系统结构

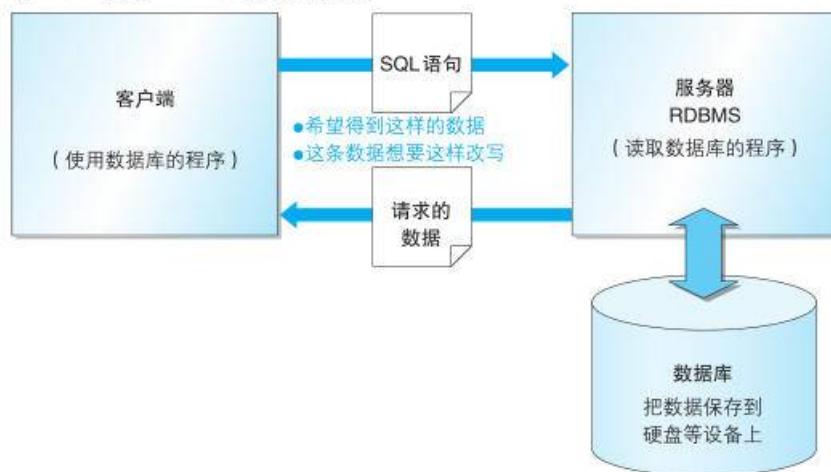


图 2.1: 使用 RDBMS 时的系统结构

2.2 初识 SQL

图 1-6 表的示例(商品表)

商品编号	商品名称	商品种类	销售单价	进货单价	登记日期	列名 (数据的项目名称)
0001	T恤衫	衣服	1000	500	2009-09-20	
0002	打孔器	办公用品	500	320	2009-09-11	行 (记录)
0003	运动T恤	衣服	4000	2800		
0004	菜刀	厨房用具	3000	2800	2009-09-20	
0005	高压锅	厨房用具	6800	5000	2009-01-15	
0006	叉子	厨房用具	500		2009-09-20	
0007	擦菜板	厨房用具	880	790	2008-04-28	
0008	圆珠笔	办公用品	100		2009-11-11	

图 2.2: 表的示例

数据库中存储的表结构类似于 excel 中的行和列，在数据库中，行称为 **记录**，它相当于一条记录，列称为 **字段**，它代表了表中存储的数据项目。

行和列交汇的地方称为单元格，一个单元格中只能输入一条记录。

SQL 是为操作数据库而开发的语言。国际标准化组织 (ISO) 为 SQL 制定了相应标准，以此为基准的 SQL 称为标准 SQL (相关信息请参考专栏——标准 SQL 和特定的 SQL)。

完全基于标准 SQL 的 RDBMS 很少，通常需要根据不同的 RDBMS 来编写特定的 SQL 语句，原则上，本课程介绍的是标准 SQL 的书写方式。

根据对 RDBMS 赋予的指令种类的不同，SQL 语句可以分为以下三类：

- DDL

DDL (Data Definition Language, 数据定义语言) 用来创建或者删除存储数据用的数据库以及数据库中的表等对象。DDL 包含以下几种指令。

- CREATE : 创建数据库和表等对象
 - DROP : 删除数据库和表等对象
 - ALTER : 修改数据库和表等对象的结构
- DML
- DML (Data Manipulation Language, 数据操纵语言) 用来查询或者变更表中的记录。DML 包含以下几种指令。
- SELECT : 查询表中的数据
 - INSERT : 向表中插入新数据
 - UPDATE : 更新表中的数据
 - DELETE : 删除表中的数据
- DCL

DCL (Data Control Language, 数据控制语言) 用来确认或者取消对数据库中的数据进行的变更。除此之外，还可以对 RDBMS 的用户是否有权限操作数据库中的对象（数据库表等）进行设定。DCL 包含以下几种指令。

- COMMIT : 确认对数据库中的数据进行的变更
- ROLLBACK : 取消对数据库中的数据进行的变更
- GRANT : 赋予用户操作权限
- REVOKE : 取消用户的操作权限

实际使用的 SQL 语句当中有 90

2.2.1 SQL 的基本书写规则

- SQL 语句要以分号 (;) 结尾
- SQL 不区分关键字的大小写，但是插入到表中的数据是区分大小写的
- Win 系统默认不区分表名及字段名的大小写
- Linux / Mac 默认严格区分表名及字段名的大小写
 - 本教程已统一调整表名及字段名的为小写，以方便初学者学习使用。…
- 常数的书写方式是固定的
`'abc', 1234, '26 Jan 2010', '10/01/26', '2010-01-26'.....`
- 单词需要用半角空格或者换行来分隔

SQL 语句的单词之间需使用半角空格或换行符来进行分隔，且不能使用全角空格作为单词的分隔符，否则会发生错误，出现无法预期的结果。

请大家认真查阅《附录 1 - SQL 语法规规范》，养成规范的书写习惯。

2.2.2 数据库的创建 (CREATE DATABASE 语句)

语法：

源码地址 [sql](#)

Line 1 CREATE DATABASE < 数据库名称 >

创建本课程使用的数据库

源码地址 [sql](#)

Line 1 CREATE DATABASE shop;

2.2.3 表的创建 (CREATE TABLE 语句)

语法：

源码地址 [sql](#)

```
Line 1 CREATE TABLE < 表名 >
      ( < 列名 1> < 数据类型 > < 该列所需约束 > ,
        < 列名 2> < 数据类型 > < 该列所需约束 > ,
        < 列名 3> < 数据类型 > < 该列所需约束 > ,
        < 列名 4> < 数据类型 > < 该列所需约束 > ,
      .
      .
      .
      < 该表的约束 1> , < 该表的约束 2> , ... );
```

创建本课程用到的商品表

源码地址 [sql](#)

```
Line 1 CREATE TABLE product
      - (product_id CHAR(4) NOT NULL,
      - product_name VARCHAR(100) NOT NULL,
      - product_type VARCHAR(32) NOT NULL,
5       sale_price INTEGER ,
      - purchase_price INTEGER ,
      - regist_date DATE ,
      - PRIMARY KEY (product_id));
```

2.2.4 命名规则

- 只能使用半角英文字母、数字、下划线作为数据库、表和列的名称
 - 名称必须以半角英文字母开头

商品表中的列名	Product 表定义的列名
商品编号	product_id
商品名称	product_name
商品种类	product_type
销售单价	sale_price
进货单价	purchase_price
登记日期	regist_date

图 2.3: 商品表和 product 表列名对应关系

2.2.5 数据类型的指定

数据库创建的表，所有的列都必须指定数据类型，每一列都不能存储与该列数据类型不符的数据。

四种最基本的数据类型

- INTEGER 型

用来指定存储整数的列的数据类型（数字型），不能存储小数。

- CHAR 型

用来存储定长字符串，当列中存储的字符串长度达不到最大长度的时候，使用半角空格进行补足，由于会浪费存储空间，所以一般不使用。

- VARCHAR 型

用来存储可变长度字符串，定长字符串在字符数未达到最大长度时会用半角空格补足，但可变长字符串不同，即使字符数未达到最大长度，也不会用半角空格补足。

- DATE 型

用来指定存储日期（年月日）的列的数据类型（日期型）。

2.2.6 约束的设置

约束是除了数据类型之外，对列中存储的数据进行限制或者追加条件的功能。

NOT NULL 是非空约束，即该列必须输入数据。

PRIMARY KEY 是主键约束，代表该列是唯一值，可以通过该列取出特定的行的数据。

2.2.7 表的删除和更新

- 删除表的语法：

源码地址 [sql](#)

Line 1 **DROP TABLE < 表名 >** ;

– 删除 product 表

需要特别注意的是，删除的表是无法恢复的，只能重新插入，请执行删除操作时无比要谨慎。

源码地址 [sql](#)

```
Line 1 DROP TABLE product;
```

- 添加列的 ALTER TABLE 语句

源码地址 [sql](#)

```
Line 1 ALTER TABLE < 表名 > ADD COLUMN < 列的定义 >;
```

- 添加一列可以存储 100 位的可变长字符串的 product_name_pinyin 列

源码地址 [sql](#)

```
Line 1 ALTER TABLE product ADD COLUMN product_name_pinyin VARCHAR(100);
```

- 删除列的 ALTER TABLE 语句

源码地址 [sql](#)

```
Line 1 ALTER TABLE < 表名 > DROP COLUMN < 列名 >;
```

- 删除 product_name_pinyin 列

源码地址 [sql](#)

```
Line 1 ALTER TABLE product DROP COLUMN product_name_pinyin;
```

ALTER TABLE 语句和 DROP TABLE 语句一样，执行之后无法恢复。误添的列可以通过 ALTER TABLE 语句删除，或者将表全部删除之后重新再创建。

【扩展内容】

- 清空表内容

源码地址 [sql](#)

```
Line 1 TRUNCATE TABLE TABLE_NAME;
```

优点：相比 Drop / Delete，Truncate 用来清除数据时，速度最快。

- 数据的更新

基本语法：

源码地址 [sql](#)

```
Line 1 UPDATE <表名>
  - SET <列名> = <表达式> [, <列名2>=<表达式2>...];
  - WHERE <条件>; -- 可选，非常重要。
  - ORDER BY 子句; -- 可选
  5 LIMIT 子句; -- 可选
```

使用 update 时要注意添加 where 条件，否则将会将所有的行按照语句修改

[源码地址 sql](#)

```
Line 1  -- 修改所有的注册时间
- UPDATE product
-   SET regist_date = '2009-10-10';
- -- 仅修改部分商品的单价
5 UPDATE product
-   SET sale_price = sale_price * 10
- WHERE product_type = '厨房用具';
```

使用 UPDATE 也可以将列更新为 NULL (该更新俗称为 NULL 清空)。此时只需要将赋值表达式右边的值直接写为 NULL 即可。

[源码地址 sql](#)

```
Line 1  -- 将商品编号为0008的数据(圆珠笔)的登记日期更新为NULL
- UPDATE product
-   SET regist_date = NULL
- WHERE product_id = '0008';
```

和 INSERT 语句一样, UPDATE 语句也可以将 NULL 作为一个值来使用。** 但是, 只有未设置 NOT NULL 约束和主键约束的列才可以清空为 NULL。** 如果将设置了上述约束的列更新为 NULL, 就会出错, 这点与 INSERT 语句相同。

- 多列更新

UPDATE 语句的 SET 子句支持同时将多个列作为更新对象。

[源码地址 sql](#)

```
Line 1  -- 基础写法, 一条UPDATE语句只更新一列
- UPDATE product
-   SET sale_price = sale_price * 10
- WHERE product_type = '厨房用具';
5 UPDATE product
-   SET purchase_price = purchase_price / 2
- WHERE product_type = '厨房用具';
```

该写法可以得到正确结果, 但是代码较为繁琐。可以采用合并的方法来简化代码。

[源码地址 sql](#)

```
Line 1  -- 合并后的写法
- UPDATE product
-   SET sale_price = sale_price * 10,
-       purchase_price = purchase_price / 2
5 WHERE product_type = '厨房用具';
```

需要明确的是, SET 子句中的列不仅可以是两列, 还可以是三列或者更多。

2.2.8 向 product 表中插入数据

为了学习 ‘INSERT’语句用法，我们首先创建一个名为 ‘productins‘的表，建表语句如下：

[源码地址 sql](#)

```
Line 1 CREATE TABLE productins
- (product_id      CHAR(4)      NOT NULL,
- product_name    VARCHAR(100) NOT NULL,
- product_type    VARCHAR(32)   NOT NULL,
5  sale_price      INTEGER      DEFAULT 0,
- purchase_price  INTEGER ,
- regist_date     DATE ,
- PRIMARY KEY (product_id));
```

基本语法：

[源码地址 sql](#)

```
Line 1 INSERT INTO <表名> (列1, 列2, 列3, ……) VALUES (值1, 值2, 值3, ……);
```

对表进行全列 INSERT 时，可以省略表名后的列清单。这时 VALUES 子句的值会默认按照从左到右的顺序赋给每一列。

[源码地址 sql](#)

```
Line 1 -- 包含列清单
- INSERT INTO productins (product_id, product_name, product_type,
- sale_price, purchase_price, regist_date) VALUES ('0005', '高压锅', '厨房用具',
6800, 5000, '2009-01-15');
- -- 省略列清单
5 INSERT INTO productins
- VALUES ('0005', '高压锅', '厨房用具', 6800, 5000, '2009-01-15');
```

原则上，执行一次 INSERT 语句会插入一行数据。插入多行时，通常需要循环执行相应次数的 INSERT 语句。其实很多 RDBMS 都支持一次插入多行数据

[源码地址 sql](#)

```
Line 1 -- 通常的INSERT
- INSERT INTO productins VALUES ('0002', '打孔器',
'办公用品', 500, 320, '2009-09-11');
- INSERT INTO productins VALUES ('0003', '运动T恤',
5 '衣服', 4000, 2800, NULL);
- INSERT INTO productins VALUES ('0004', '菜刀',
'厨房用具', 3000, 2800, '2009-09-20');
- -- 多行INSERT ( DB2、SQL、SQL Server、PostgreSQL 和 MySQL多行插入)
- INSERT INTO productins VALUES ('0002', '打孔器',
10 '办公用品', 500, 320, '2009-09-11'),
('0003', '运动T恤', '衣服', 4000, 2800, NULL),
('0004', '菜刀', '厨房用具', 3000, 2800, '2009-09-20');
```

```

- -- Oracle中的多行INSERT
- INSERT ALL INTO productins VALUES ('0002', '打孔器', '办公用品', 500, 320, '
2009-09-11')
15 INTO productins VALUES ('0003', '运动T恤', '衣服', 4000, 2800, NULL)
- INTO productins VALUES ('0004', '菜刀', '厨房用具', 3000, 2800, '2009-09-20')
- SELECT * FROM DUAL;
- -- DUAL是Oracle特有（安装时的必选项）的一种临时表A。因此“SELECT *FROM DUAL”
部分也只是临时性的，并没有实际意义。

```

INSERT 语句中想给某一列赋予 NULL 值时，可以直接在 VALUES 子句的值清单中写入 NULL。想要插入 NULL 的列一定不能设置 NOT NULL 约束。

[源码地址 sql](#)

```

Line 1 INSERT INTO productins (product_id, product_name, product_type,
- sale_price, purchase_price, regist_date) VALUES ('0006', '叉子',
- '厨房用具', 500, NULL, '2009-09-20');

```

还可以向表中插入默认值（初始值）。可以通过在创建表的 CREATE TABLE 语句中设置 DEFAULT 约束来设定默认值。

[源码地址 sql](#)

```

Line 1 CREATE TABLE productins
- (product_id CHAR(4) NOT NULL,
- (略)
- sale_price INTEGER
5 (略) DEFAULT 0, -- 销售单价的默认值设定为0;
- PRIMARY KEY (product_id));

```

可以使用 INSERT ... SELECT 语句从其他表复制数据。

[源码地址 sql](#)

```

Line 1 -- 将商品表中的数据复制到商品复制表中
- INSERT INTO productcopy (product_id, product_name, product_type, sale_price,
purchase_price, regist_date)
- SELECT product_id, product_name, product_type, sale_price,
purchase_price, regist_date
5 FROM Product;

```

本课程用表插入数据 sql 如下：

[源码地址 sql](#)

```

Line 1 - DML : 插入数据
- STARTTRANSACTION;
- INSERT INTO product VALUES('0001', 'T恤衫', '衣服', 1000, 500, '2009-09-20');
- INSERT INTO product VALUES('0002', '打孔器', '办公用品', 500, 320, '2009-09-11
');
5 INSERT INTO product VALUES('0003', '运动T恤', '衣服', 4000, 2800, NULL);

```

```

- INSERT INTO product VALUES('0004', '菜刀', '厨房用具', 3000, 2800, '2009-09-20
  ');
- INSERT INTO product VALUES('0005', '高压锅', '厨房用具', 6800, 5000, '
  2009-01-15');
- INSERT INTO product VALUES('0006', '叉子', '厨房用具', 500, NULL, '2009-09-20'
  );
- INSERT INTO product VALUES('0007', '擦菜板', '厨房用具', 880, 790, '2008-04-28
  ');
10 INSERT INTO product VALUES('0008', '圆珠笔', '办公用品', 100, NULL, '
  2009-11-11');
- COMMIT;

```

练习题

2.1

编写一条 CREATE TABLE 语句，用来创建一个包含表 1-A 中所列各项的表 Addressbook（地址簿），并为 regist_no（注册编号）列设置主键约束

列的含义	列的名称	数据类型	约束
注册编号	regist_no	整数型	不能为NULL、主键
姓名	name	可变长字符串类 型(长度为 128)	不能为NULL
住址	address	可变长字符串类 型(长度为 256)	不能为NULL
电话号码	tel_no	定长字符串类型 (长度为 10)	
邮箱地址	mail_address	定长字符串类型 (长度为 20)	

图 2.4: 表 Addressbook（地址簿）中的列

2.2

假设在创建练习 2.1 中的 Addressbook 表时忘记添加如下一列 postal_code（邮政编码）了，请把此列添加到 Addressbook 表中。

列名: postal_code

数据类型: 定长字符串类型 (长度为 8)

约束: 不能为 NULL

2.3

编写 SQL 语句来删除 Addressbook 表。

2.4

编写 SQL 语句来恢复删除掉的 Addressbook 表。

第 3 章 SELECT 语句基础

3.1 SELECT 语句基础

3.1.1 从表中选取数据

SELECT 语句 从表中选取数据时需要使用 SELECT 语句，也就是只从表中选出（SELECT）必要数据的意思。通过 SELECT 语句查询并选取出必要数据的过程称为匹配查询或查询（query）。

基本 SELECT 语句包含了 SELECT 和 FROM 两个子句（clause）。示例如下：

源码地址 [sql](#)

```
Line 1  SELECT <列名>,
-      FROM <表名>;
```

其中，SELECT 子句中列举了希望从表中查询出的列的名称，而 FROM 子句则指定了选取出数据的表的名称。

3.1.2 从表中选取符合条件的数据

WHERE 语句 当不需要取出全部数据，而是选取出满足“商品种类为衣服”“销售单价在 1000 日元以上”等某些条件的数据时，使用 WHERE 语句。

SELECT 语句通过 WHERE 子句来指定查询数据的条件。在 WHERE 子句中可以指定“某一列的值和这个字符串相等”或者“某一列的值大于这个数字”等条件。执行含有这些条件的 SELECT 语句，就可以查询出只符合该条件的记录了。

源码地址 [sql](#)

```
Line 1  SELECT <列名>, … …
-      FROM <表名>
-      WHERE <条件表达式>;
```

比较下面两者输出结果的不同：

源码地址 [sql](#)

```
Line 1  -- 用来选取product type列为衣服' 的记录的SELECT语句
-  SELECT product_name, product_type
-  FROM product
-  WHERE product_type = '衣服';
5   -- 也可以选取出不是查询条件的列（条件列与输出列不同）
-  SELECT product_name
-  FROM product
-  WHERE product_type = '衣服';
```

3.1.3 相关法则

- 星号 (*) 代表全部列的意思。
- SQL 中可以随意使用换行符，不影响语句执行（但不可插入空行）。
- 设定汉语别名时需要使用双引号 ("") 括起来。

含义	运算符
加法	+
减法	-
乘法	*
除法	/

- 在 SELECT 语句中使用 DISTINCT 可以删除重复行。
- 注释是 SQL 语句中用来标识说明或者注意事项的部分。分为 1 行注释"-" 和多行注释"/* */"。

源码地址 [sql](#)

```
Line 1 -- 想要查询出全部列时，可以使用代表所有列的星号 (*) 。
- SELECT *
-   FROM <表名>;
- -- SQL语句可以使用AS关键字为列设定别名（用中文时需要双引号（“”））。
5  SELECT product_id      As id,
-     product_name    As name,
-     purchase_price AS "进货单价"
-   FROM product;
- -- 使用DISTINCT删除product_type列中重复的数据
10 SELECT DISTINCT product_type
-   FROM product;
```

3.2 算术运算符和比较运算符

3.2.1 算术运算符

SQL 语句中可以使用的四则运算的主要运算符如下：

3.2.2 比较运算符

源码地址 [sql](#)

```
Line 1 -- 选取出sale_price列为500的记录
- SELECT product_name, product_type
-   FROM product
- WHERE sale_price = 500;
```

SQL 常见比较运算符如下：

运算符	含义
=	相等
<>	不相等
>=	大于等于
>	大于
<=	小于等于
<	小于

3.2.3 常用法则

- SELECT 子句中可以使用常数或者表达式。
- 使用比较运算符时一定要注意不等号和等号的位置。
- 字符串类型的数据原则上按照字典顺序进行排序，不能与数字的大小顺序混淆。
- 希望选取 NULL 记录时，需要在条件表达式中使用 IS NULL 运算符。希望选取不是 NULL 的记录时，需要在条件表达式中使用 IS NOT NULL 运算符。

相关代码如下：

源码地址 [sql](#)

```

Line 1  -- SQL语句中也可以使用运算表达式
- SELECT product_name, sale_price, sale_price * 2 AS "sale_price_x2"
-   FROM product;
- -- WHERE子句的条件表达式中也可以使用计算表达式
5  SELECT product_name, sale_price, purchase_price
-   FROM product
- WHERE sale_price-purchase_price >= 500;
- /* 对字符串使用不等号
- 首先创建chars并插入数据
10 选取出大于‘2’的SELECT语句*/
- -- DDL: 创建表
- CREATE TABLE chars
-   (chr CHAR (3) NOT NULL,
-   PRIMARY KEY (chr) );
15 -- 选取出大于'2'的数据的SELECT语句('2'为字符串)
- SELECT chr
-   FROM chars
- WHERE chr > '2';
- -- 选取NULL的记录
20 SELECT product_name, purchase_price
-   FROM product
- WHERE purchase_price IS NULL;
- -- 选取不为NULL的记录
- SELECT product_name, purchase_price
-   FROM product
- WHERE purchase_price IS NOT NULL;

```

3.3 逻辑运算符

3.3.1 NOT 运算符

想要表示“不是……”时，除了前文的 $<>$ 运算符外，还存在另外一个表示否定、使用范围更广的运算符：NOT。

NOT 不能单独使用，如下例：

源码地址 [sql](#)

```
Line 1 -- 选取出销售单价大于等于1000日元的记录
- SELECT product_name, product_type, sale_price
-   FROM product
- WHERE sale_price >= 1000;
5 -- 向代码清单2-30的查询条件中添加NOT运算符
- SELECT product_name, product_type, sale_price
-   FROM product
- WHERE NOT sale_price >= 1000;
```

3.3.2 AND 运算符和 OR 运算符

当希望同时使用多个查询条件时，可以使用 AND 或者 OR 运算符。

AND 相当于“并且”，类似数学中的取交集；

OR 相当于“或者”，类似数学中的取并集。

如下图所示：

AND 运算符工作效果图

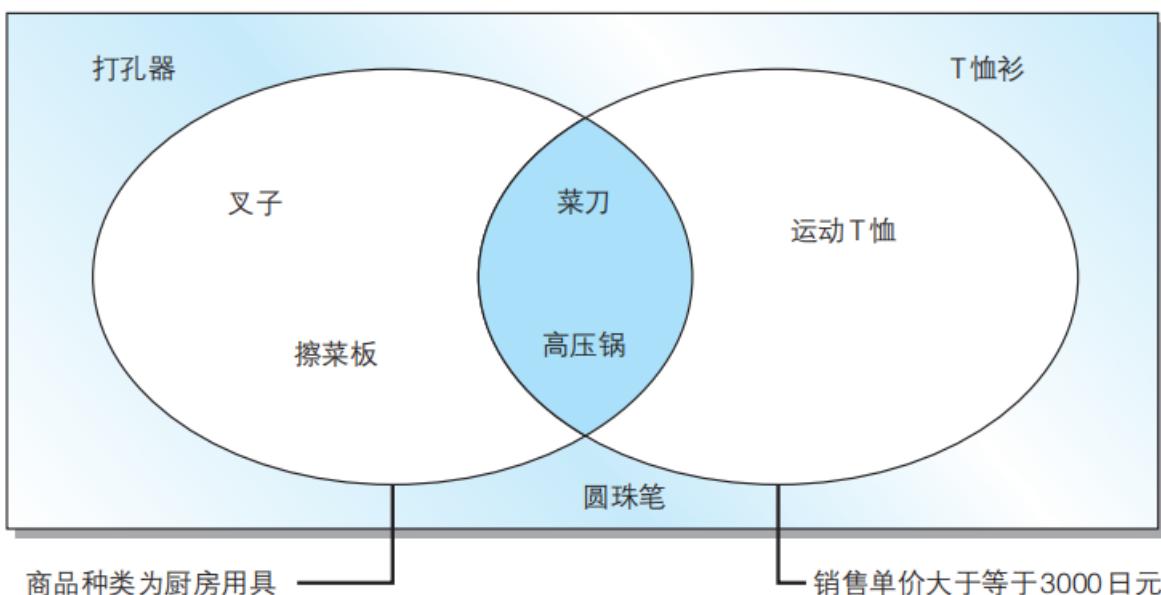


图 3.1: and

OR 运算符工作效果图

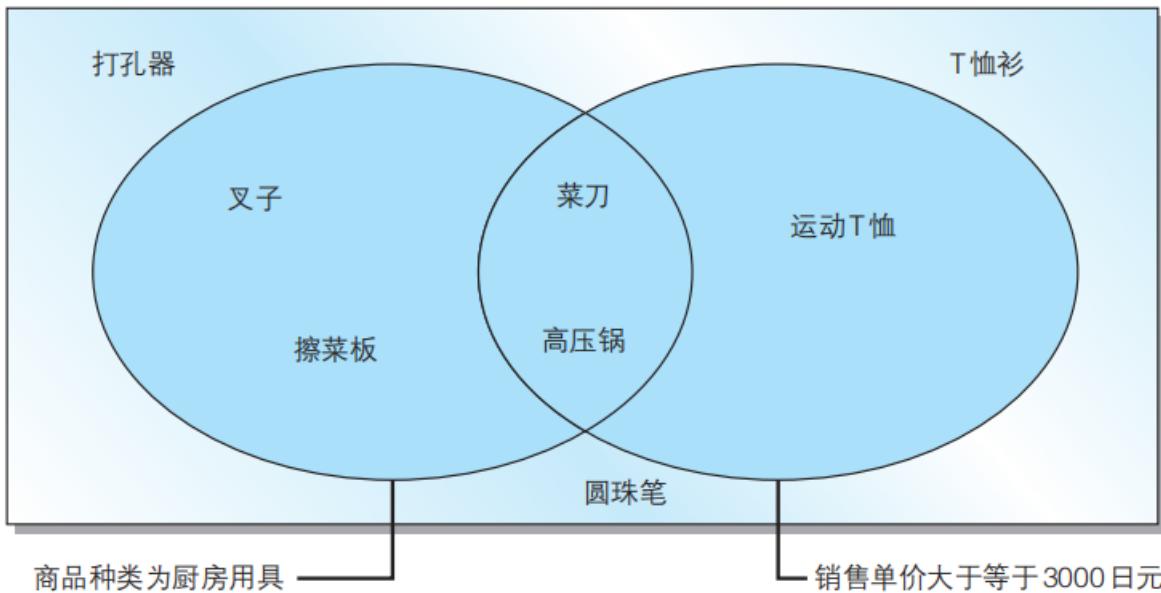


图 3.2: or

通过括号优先处理 如果要查找这样一个商品，该怎么处理？

“商品种类为办公用品”并且“登记日期是 2009 年 9 月 11 日或者 2009 年 9 月 20 日”理想结果为“打孔器”，但当你输入以下信息时，会得到错误结果

源码地址 [sql](#)

```
Line 1 -- 将查询条件原封不动地写入条件表达式，会得到错误结果
- SELECT product_name, product_type, regist_date
-   FROM product
- WHERE product_type = '办公用品'
5   AND regist_date = '2009-09-11'
- OR regist_date = '2009-09-20';
```

错误的原因是是 AND 运算符优先于 OR 运算符，想要优先执行 OR 运算，可以使用**括号**：

源码地址 [sql](#)

```
Line 1 -- 通过使用括号让OR运算符先于AND运算符执行
- SELECT product_name, product_type, regist_date
-   FROM product
- WHERE product_type = '办公用品'
5   AND ( regist_date = '2009-09-11'
-        OR regist_date = '2009-09-20' );
```

3.3.3 真值表

复杂运算时该怎样理解？

当碰到条件较复杂的语句时，理解语句含义并不容易，这时可以采用**真值表**来梳理逻辑关系。

什么是真值？

本节介绍的三个运算符 NOT、AND 和 OR 称为逻辑运算符。这里所说的逻辑就是对真值进行操作的意思。**真值**就是值为真 (TRUE) 或假 (FALSE) 其中之一的值。

例如，对于 $sale_price >= 3000$ 这个查询条件来说，由于 $product_name$ 列为‘运动 T 恤’的记录的 $sale_price$ 列的值是 2800，因此会返回假 (FALSE)，而 $product_name$ 列为‘高压锅’的记录的 $sale_price$ 列的值是 5000，所以返回真 (TRUE)。

AND 运算符两侧的真值都为真时返回真，除此之外都返回假。

OR 运算符两侧的真值只要有一个不为假就返回真，只有当其两侧的真值都为假时才返回假。

NOT 运算符只是单纯的将真转换为假，将假转换为真。

真值表

AND			OR			NOT	
P	Q	P AND Q	P	Q	P OR Q	P	NOT P
真	真	真	真	真	真	真	假
真	假	假	真	假	真	假	真
假	真	假	假	真	真	真	假
假	假	假	假	假	假	假	真

图 3.3: true

查询条件为 $P \text{ AND } (Q \text{ OR } R)$ 的真值表

P AND (Q OR R)				
P	Q	R	Q OR R	P AND (Q OR R)
真	真	真	真	真
真	真	假	真	真
真	假	真	真	真
真	假	假	假	假
假	真	真	真	假
假	真	假	真	假
假	假	真	真	假
假	假	假	假	假

P : 商品种类为办公用品
 Q : 登记日期是 2009 年 9 月 11 日
 R : 登记日期是 2009 年 9 月 20 日
 Q OR R : 登记日期是 2009 年 9 月 11 日或者 2009 年 9 月 20 日
 P AND (Q OR R) : 商品种类为办公用品，并且，登记日期是 2009 年 9 月 11 日或者 2009 年 9 月 20 日

图 3.4: true2

含有 NULL 时的真值 NULL 的真值结果既不为真，也不为假，因为并不知道这样一个值。

那该如何表示呢？

这时真值是除真假之外的第三种值——**不确定** (UNKNOWN)。一般的逻辑运算并不存在这第三种值。SQL 之外的语言也基本上只使用真和假这两种真值。与通常的逻辑运算被称为二值逻辑相对，只有 SQL 中的逻辑运算被称为三值逻辑。

三值逻辑下的 AND 和 OR 真值表为：

AND			OR		
P	Q	P AND Q	P	Q	P OR Q
真	真	真	真	真	真
真	假	假	真	假	真
真	不确定	不确定	真	不确定	真
假	真	假	假	真	真
假	假	假	假	假	假
假	不确定	假	假	不确定	不确定
不确定	真	不确定	不确定	真	真
不确定	假	假	不确定	假	不确定
不确定	不确定	不确定	不确定	不确定	不确定

图 3.5: true3

练习题-第一部分

3.1

编写一条 SQL 语句，从 product (商品) 表中选取出“登记日期 (regist 在 2009 年 4 月 28 日之后”的商品，查询结果要包含 product_name 和 regist_date 两列。

3.2

请说出对 product 表执行如下 3 条 SELECT 语句时的返回结果。

源码地址 [sql](#)

```
Line 1  SELECT *
-      FROM product
- WHERE purchase_price = NULL;
```

源码地址 [sql](#)

```
Line 1  SELECT *
-      FROM product
- WHERE purchase_price <> NULL;
```

源码地址 [sql](#)

```
Line 1  SELECT *
-      FROM product
- WHERE product_name > NULL;
```

3.3

代码清单 2-22 (2-2 节) 中的 SELECT 语句能够从 product 表中取出“销售单价 (saleprice) 比进货单价 (purchase price) 高出 500 日元以上”的商品。请写出两条可以得到相同结果的 SELECT 语句。执行结果如下所示。

源码地址 [sql](#)

Line	product_name	sale_price	purchase_price
-	T恤衫	1000	500
-	运动T恤	4000	2800
5	高压锅	6800	5000

3.4

请写出一条 SELECT 语句，从 product 表中选取出满足“销售单价打九折之后利润高于 100 日元的办公用品和厨房用具”条件的记录。查询结果要包括 product_name 列、product_type 列以及销售单价打九折之后的利润（别名设定为 profit）。

提示：销售单价打九折，可以通过 saleprice 列的值乘以 0.9 获得，利润可以通过该值减去 purchase_price 列的值获得。

3.4 对表进行聚合查询

3.4.1 聚合函数

SQL 中用于汇总的函数叫做聚合函数。以下五个是最常用的聚合函数：

- COUNT：计算表中的记录数（行数）
- SUM：计算表中数值列中数据的合计值
- AVG：计算表中数值列中数据的平均值
- MAX：求出表中任意列中数据的最大值
- MIN：求出表中任意列中数据的最小值

请沿用第一章的数据，使用以下操作熟练函数：

源码地址 [sql](#)

```
Line 1 -- 计算全部数据的行数 (包含NULL)
- SELECT COUNT(*)
  FROM product;
-- 计算NULL以外数据的行数
5 SELECT COUNT(purchase_price)
  FROM product;
-- 计算销售单价和进货单价的合计值
- SELECT SUM(sale_price), SUM(purchase_price)
  FROM product;
```

```

10  -- 计算销售单价和进货单价的平均值
- SELECT AVG(sale_price), AVG(purchase_price)
-   FROM product;
- -- MAX和MIN也可用于非数值型数据
- SELECT MAX(regist_date), MIN(regist_date)
15   FROM product;

```

[源码地址 sql](#)

使用聚合函数删除重复值

```

Line 1  -- 计算去除重复数据后的数据行数
- SELECT COUNT(DISTINCT product_type)
-   FROM product;
- -- 是否使用DISTINCT时的动作差异 (SUM函数)
5  SELECT SUM(sale_price), SUM(DISTINCT sale_price)
-   FROM product;

```

3.4.2 常用法则

- COUNT 函数的结果根据参数的不同而不同。COUNT(*) 会得到包含 NULL 的数据行数，而 COUNT(<列名>) 会得到 NULL 之外的数据行数。
- 聚合函数会将 NULL 排除在外。但 COUNT(*) 例外，并不会排除 NULL。
- MAX/MIN 函数几乎适用于所有数据类型的列。SUM/AVG 函数只适用于数值类型的列。
- 想要计算值的种类时，可以在 COUNT 函数的参数中使用 DISTINCT。
- 在聚合函数的参数中使用 DISTINCT，可以删除重复数据。

3.5 对表进行分组

3.5.1 GROUP BY 语句

之前使用聚合函数都是会整个表的数据进行处理，当你想将进行分组汇总时（即：将现有的数据按照某列来汇总统计），GROUP BY 可以帮助你：

[源码地址 sql](#)

```

Line 1  SELECT <列名1>, <列名2>, <列名3>, ...
-     FROM <表名>
-     GROUP BY <列名1>, <列名2>, <列名3>, ... ;

```

看一看是否使用 GROUP BY 语句的差异：

[源码地址 sql](#)

```

Line 1  -- 按照商品种类统计数据行数
- SELECT product_type, COUNT(*)
-   FROM product

```

```

- GROUP BY product_type;
5 -- 不含GROUP BY
- SELECT product_type, COUNT(*)
- FROM product

```

按照商品种类对表进行切分



图 3.6: cut

这样，GROUP BY 子句就像切蛋糕那样将表进行了分组。在 GROUP BY 子句中指定的列称为 **聚合键** 或者 **分组列**。

聚合键中包含 NULL 时 将进货单价 (purchase_price) 作为聚合键举例：

源码地址 [sql](#)

```

Line 1 SELECT purchase_price, COUNT(*)
-   FROM product
- GROUP BY purchase_price;

```

此时会将 NULL 作为一组特殊数据进行处理

GROUP BY 书写位置 GROUP BY 的子句书写顺序有严格要求，不按要求会导致 SQL 无法正常执行，目前出现过的子句顺序为：

1 SELECT 2. FROM 3. WHERE 4. GROUP BY

其中前三项用于筛选数据，GROUP BY 对筛选出的数据进行处理

源码地址 [sql](#)

在 WHERE 子句中使用 GROUP BY

```

Line 1 SELECT purchase_price, COUNT(*)
-   FROM product
- WHERE product_type = '衣服'
- GROUP BY purchase_price;

```

3.5.2 常见错误

在使用聚合函数及 GROUP BY 子句时，经常出现的错误有：

1. 在聚合函数的 SELECT 子句中写了聚合键以外的列使用 COUNT 等聚合函数时，SELECT 子句中如果出现列名，只能是 GROUP BY 子句中指定的列名（也就是聚合键）。
2. 在 GROUP BY 子句中使用列的别名 SELECT 子句中可以通过 AS 来指定别名，但在 GROUP BY 中不能使用别名。因为在 DBMS 中，SELECT 子句在 GROUP BY 子句后执行。
3. 在 WHERE 中使用聚合函数原因是聚合函数的使用前提是结果集已经确定，而 WHERE 还处于确定结果集的过程中，所以相互矛盾会引发错误。如果想指定条件，可以在 SELECT，HAVING（下面马上会讲）以及 ORDER BY 子句中使用聚合函数。

3.6 为聚合结果指定条件

3.6.1 用 HAVING 得到特定分组

将表使用 GROUP BY 分组后，怎样才能只取出其中两组？

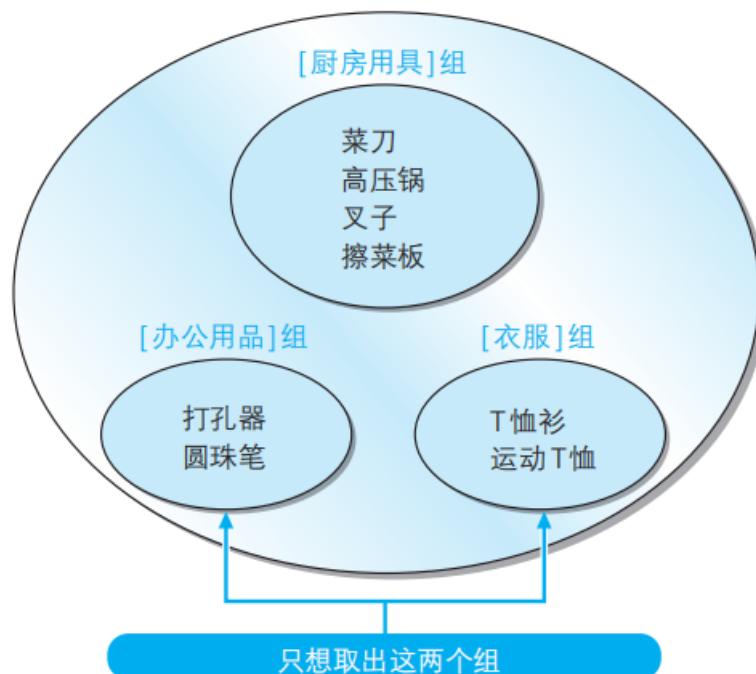


图 3.7: groupby

这里 WHERE 不可行，因为，WHERE 子句只能指定记录（行）的条件，而不能用来指定组的条件（例如，“数据行数为 2 行”或者“平均值为 500”等）。

可以在 GROUP BY 后使用 HAVING 子句。

HAVING 的用法类似 WHERE

3.6.2 HAVING 特点

HAVING 子句用于对分组进行过滤，可以使用数字、聚合函数和 GROUP BY 中指定的列名（聚合键）。

[源码地址 sql](#)

```

Line 1  -- 数字
- SELECT product_type, COUNT(*)
-   FROM product
- GROUP BY product_type
5 HAVING COUNT(*) = 2;
- -- 错误形式 (因为product_name不包含在GROUP BY聚合键中)
- SELECT product_type, COUNT(*)
-   FROM product
- GROUP BY product_type
10 HAVING product_name = '圆珠笔';

```

3.7 对查询结果进行排序

3.7.1 ORDER BY

SQL 中的执行结果是随机排列的，当需要按照特定顺序排序时，可已使用 **ORDER BY** 子句。

[源码地址 sql](#)

```

Line 1  SELECT <列名1>, <列名2>, <列名3>, ...
-   FROM <表名>
- ORDER BY <排序基准列1>, <排序基准列2>, ...

```

默认为升序排列，降序排列为 DESC

[源码地址 sql](#)

```

Line 1  -- 降序排列
- SELECT product_id, product_name, sale_price, purchase_price
-   FROM product
- ORDER BY sale_price DESC;
5 -- 多个排序键
- SELECT product_id, product_name, sale_price, purchase_price
-   FROM product
- ORDER BY sale_price, product_id;
- -- 当用于排序的列名中含有NULL时，NULL会在开头或末尾进行汇总。
10 SELECT product_id, product_name, sale_price, purchase_price
-   FROM product
- ORDER BY purchase_price;

```

3.7.2 ORDER BY 中列名可使用别名

前文讲 GROUP BY 中提到，GROUP BY 子句中不能使用 SELECT 子句中定义的别名，但是在 ORDER BY 子句中却可以使用别名。为什么在 GROUP BY 中不可以而在 ORDER BY 中可以呢？

这是因为 SQL 在使用 HAVING 子句时 SELECT 语句的顺序为：

FROM WHERE GROUP BY HAVING SELECT ORDER BY。

其中 SELECT 的执行顺序在 GROUP BY 子句之后, ORDER BY 子句之前。也就是说, 当在 ORDER BY 中使用别名时, 已经知道了 SELECT 设置的别名存在, 但是在 GROUP BY 中使用别名时还不知道别名的存在, 所以不能在 ORDER BY 中可以使用别名, 但是在 GROUP BY 中不能使用别名

练习题-第二部分

3.5

请指出下述 SELECT 语句中所有的语法错误。

[源码地址 sql](#)

```
Line 1  SELECT product_id, SUM (product_name)
- --本SELECT语句中存在错误。
-   FROM product
- GROUP BY product_type
5 WHERE regist_date > '2009-09-01';
```

3.6

请编写一条 SELECT 语句, 求出销售单价 (‘sale_price’ 列) 合计值大于进货单价 (‘purchase_price’ 列) 合计值 1.5 倍的商品种类。执行结果如下所示。

[源码地址 sql](#)

```
Line 1  product_type | sum | sum
- -----+-----+-----
- 衣服      | 5000 | 3300
- 办公用品  | 600  | 320
```

product_type	sum	sum
衣服	5000	3300
办公用品	600	320

SUM (sale_price) 的结果

← SUM (purchase_price) 的结果

图 3.8: test26

3.7

此前我们曾经使用 SELECT 语句选取出了 product (商品) 表中的全部记录。当时我们使用了 ORDERBY 子句来指定排列顺序, 但现在已经无法记起当时如何指定的了。请根据下列执行结果, 思考 ORDERBY 子句的内容。

product_id	product_name	product_type	sale_price	purchase_price	regist_date
0003	运动T恤	衣服	4000	2800	
0008	圆珠笔	办公用品	100		2009-11-11
0006	叉子	厨房用具	500		2009-09-20
0001	T恤衫	衣服	1000	500	2009-09-20
0004	菜刀	厨房用具	3000	2800	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11
0005	高压锅	厨房用具	6800	5000	2009-01-15
0007	擦菜板	厨房用具	880	790	2008-04-28

图 3.9: test27

第 4 章 复杂一点的查询

之前介绍了 SQL 基本的查询用法，接下来介绍一些相对复杂的用法。

4.1 视图

我们先来看一个查询语句（仅做示例，未提供相关数据）

源码地址 [sql](#)

```
Line 1 SELECT stu_name FROM view_students_info;
```

单从表面上看起来这个语句是和正常的从数据表中查询数据是完全相同的，但其实我们操作的是一个视图。所以从 SQL 的角度来说操作视图与操作表看起来是完全相同的，那么为什么还会有视图的存在呢？视图到底是什么？视图与表有什么不同呢？

4.1.1 什么是视图

视图是一个虚拟的表，不同于直接操作数据表，视图是依据 SELECT 语句来创建的（会在下面具体介绍），所以操作视图时会根据创建视图的 SELECT 语句生成一张虚拟表，然后在这张虚拟表上做 SQL 操作。

4.1.2 视图与表有什么区别

《SQL 基础教程第 2 版》用一句话非常凝练的概括了视图与表的区别——“是否保存了实际的数据”。所以视图并不是数据库真实存储的数据表，它可以看作是一个窗口，通过这个窗口我们可以看到数据库表中真实存在的数据。所以我们要区别视图和数据表的本质，即视图是基于真实表的一张虚拟的表，其数据来源均建立在真实表的基础上。

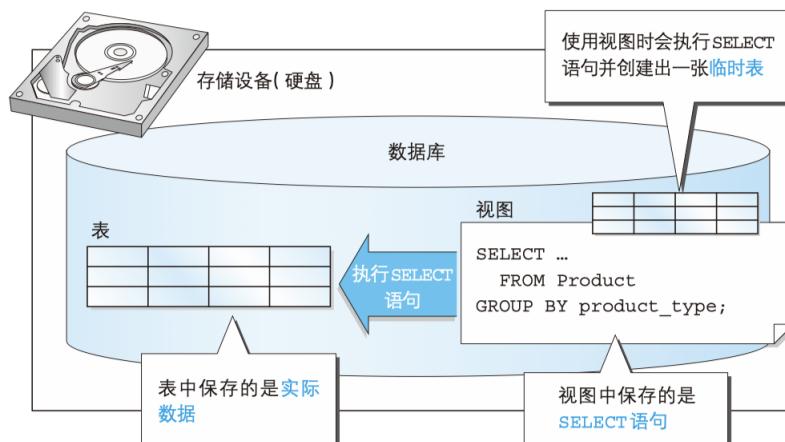


图 4.1: 视图和表

下面这句顺口溜也方便大家记忆视图与表的关系：“视图不是表，视图是虚表，视图依赖于表”。

4.1.3 为什么会有视图

那既然已经有数据表了，为什么还需要视图呢？主要有以下几点原因：

1. 通过定义视图可以将频繁使用的 SELECT 语句保存以提高效率。
2. 通过定义视图可以使用户看到的数据更加清晰。

3. 通过定义视图可以不对外公开数据表全部字段，增强数据的保密性。
4. 通过定义视图可以降低数据的冗余。

4.1.4 如何创建视图

说了这么多视图与表的区别，下面我们就一起来看一下如何创建视图吧。

创建视图的基本语法如下：

源码地址 [sql](#)

```
Line 1 CREATE VIEW <视图名称>(<列名1>,<列名2>,...) AS <SELECT语句>
```

其中 SELECT 语句需要书写在 AS 关键字之后。SELECT 语句中列的排列顺序和视图中列的排列顺序相同，SELECT 语句中的第 1 列就是视图中的第 1 列，SELECT 语句中的第 2 列就是视图中的第 2 列，以此类推。而且视图的列名是在视图名称之后的列表中定义的。需要注意的是视图名在数据库中需要是唯一的，不能与其他视图和表重名。

视图不仅可以基于真实表，我们也可以在视图的基础上继续创建视图。

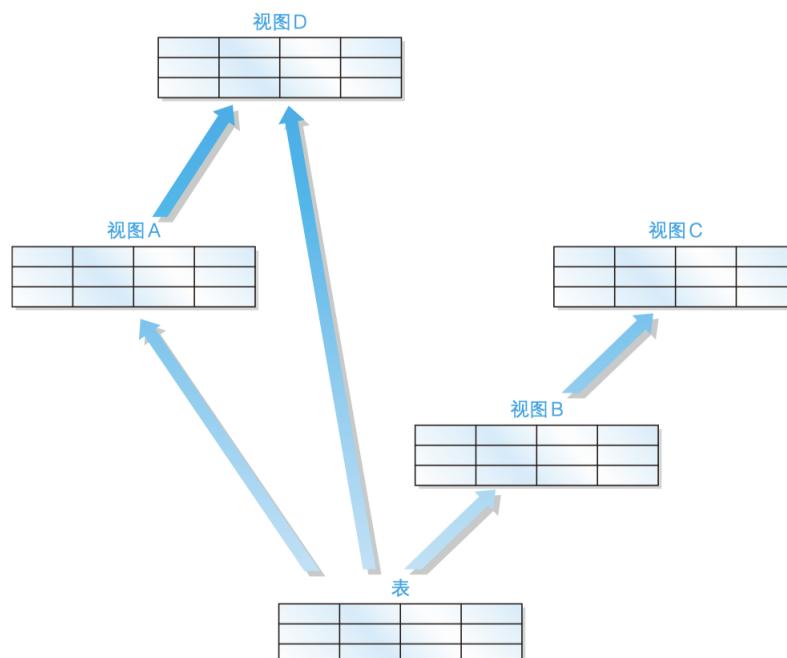


图 4.2: 可以在视图的基础上创建视图

虽然在视图上继续创建视图的语法没有错误，但是我们还是应该尽量避免这种操作。这是因为对多数 DBMS 来说，多重视图会降低 SQL 的性能。

- 注意事项

需要注意的是在一般的 DBMS 中定义视图时不能使用 ORDER BY 语句。下面这样定义视图是错误的。

源码地址 [sql](#)

```
Line 1 CREATE VIEW productsum (product_type, cnt_product)
- AS
- SELECT product_type, COUNT(*)
```

```

-   FROM product
5  GROUP BY product_type
-   ORDER BY product_type;

```

为什么不能使用 ORDER BY 子句呢？这是因为视图和表一样，数据行都是没有顺序的

在 MySQL 中视图的定义是允许使用 ORDER BY 语句的，但是若从特定视图进行选择，而该视图使用了自己的 ORDER BY 语句，则视图定义中的 ORDER BY 将被忽略。

- 基于单表的视图

我们在 product 表的基础上创建一个视图，如下：

源码地址 [sql](#)

```

Line 1 CREATE VIEW productsum (product_type, cnt_product)
- AS
- SELECT product_type, COUNT(*)
-   FROM product
5  GROUP BY product_type ;

```

创建的视图如下图所示：

product_type	cnt_product
1 衣服	2
2 办公用品	2
3 厨房用具	4

图 4.3: 视图查询结果

- 基于多表的视图

为了学习多表视图，我们再创建一张表，相关代码如下：

源码地址 [sql](#)

```

Line 1 CREATE TABLE shop_product
- (shop_id      CHAR(4)      NOT NULL,
-  shop_name    VARCHAR(200)  NOT NULL,
-  product_id   CHAR(4)      NOT NULL,
5  quantity     INTEGER      NOT NULL,
-  PRIMARY KEY (shop_id, product_id));
- INSERT INTO shop_product (shop_id, shop_name, product_id, quantity) VALUES (
-   '000A', '东京',      '0001', 30);
- INSERT INTO shop_product (shop_id, shop_name, product_id, quantity) VALUES (
-   '000A', '东京',      '0002', 50);
- INSERT INTO shop_product (shop_id, shop_name, product_id, quantity) VALUES (
-   '000A', '东京',      '0003', 15);
10 INSERT INTO shop_product (shop_id, shop_name, product_id, quantity) VALUES (
-   '000B', '名古屋',    '0002', 30);

```

```
- INSERT INTO shop_product (shop_id, shop_name, product_id, quantity) VALUES ('000B', '名古屋', '0003', 120);
- INSERT INTO shop_product (shop_id, shop_name, product_id, quantity) VALUES ('000B', '名古屋', '0004', 20);
- INSERT INTO shop_product (shop_id, shop_name, product_id, quantity) VALUES ('000B', '名古屋', '0006', 10);
- INSERT INTO shop_product (shop_id, shop_name, product_id, quantity) VALUES ('000B', '名古屋', '0007', 40);
15 INSERT INTO shop_product (shop_id, shop_name, product_id, quantity) VALUES ('000C', '大阪', '0003', 20);
- INSERT INTO shop_product (shop_id, shop_name, product_id, quantity) VALUES ('000C', '大阪', '0004', 50);
- INSERT INTO shop_product (shop_id, shop_name, product_id, quantity) VALUES ('000C', '大阪', '0006', 90);
- INSERT INTO shop_product (shop_id, shop_name, product_id, quantity) VALUES ('000C', '大阪', '0007', 70);
- INSERT INTO shop_product (shop_id, shop_name, product_id, quantity) VALUES ('000D', '福冈', '0001', 100);
```

我们在 product 表和 shop_product 表的基础上创建视图。

源码地址 [sql](#)

```
Line 1 CREATE VIEW view_shop_product(product_type, sale_price, shop_name)
- AS
- SELECT product_type, sale_price, shop_name
- FROM product,
5     shop_product
- WHERE product.product_id = shop_product.product_id;
```

创建的视图如下图所示

	product_type	sale_price	shop_name
1	衣服	1000	东京
2	办公用品	500	东京
3	衣服	4000	东京
4	办公用品	500	名古屋
5	衣服	4000	名古屋
6	厨房用具	3000	名古屋
7	厨房用具	500	名古屋
8	厨房用具	880	名古屋
9	衣服	4000	大阪
10	厨房用具	3000	大阪
11	厨房用具	500	大阪
12	厨房用具	880	大阪
13	衣服	1000	福冈

图 4.4: 视图查询结果

我们可以在这个视图的基础上进行查询

源码地址 [sql](#)

```
Line 1  SELECT sale_price, shop_name
-      FROM view_shop_product
-      WHERE product_type = '衣服';
```

查询结果为：

	sale_price	shop_name
1	1000	东京
2	4000	东京
3	4000	名古屋
4	4000	大阪
5	1000	福冈

图 4.5: 视图查询结果

4.1.5 如何修改视图结构

修改视图结构的基本语法如下：

源码地址 [sql](#)

```
Line 1  ALTER VIEW <视图名> AS <SELECT语句>
```

其中视图名在数据库中需要是唯一的，不能与其他视图和表重名。当然也可以通过将当前视图删除然后重新创建的方式达到修改的效果。（对于数据库底层是不是也是这样操作的呢，你可以自己探索一下。）

- 修改视图

我们修改上方的 productSum 视图为

源码地址 [sql](#)

```
Line 1 ALTER VIEW productSum
      AS
      -
      -
      -
5       SELECT product_type, sale_price
            FROM Product
           WHERE regist_date > '2009-09-11';
```

此时 productSum 视图内容如下图所示

	product_type	sale_price
1	衣服	1000
2	厨房用具	3000
3	厨房用具	500
4	办公用品	100

图 4.6: 视图查询结果

4.1.6 如何更新视图内容

因为视图是一个虚拟表，所以对视图的操作就是对底层基础表的操作，所以在修改时只有满足底层基本表的定义才能成功修改。

对于一个视图来说，如果包含以下结构的任意一种都是不可以被更新的：

- 聚合函数 SUM()、MIN()、MAX()、COUNT() 等。
- DISTINCT 关键字。
- GROUP BY 子句。
- HAVING 子句。
- UNION 或 UNION ALL 运算符。
- FROM 子句中包含多个表。

视图归根结底还是从表派生出来的，因此，如果原表可以更新，那么视图中的数据也可以更新。反之亦然，如果视图发生了改变，而原表没有进行相应更新的话，就无法保证数据的一致性了。

- 更新视图

因为我们刚刚修改的 productSum 视图不包括以上的限制条件，我们来尝试更新一下视图

源码地址 [sql](#)

```
Line 1 UPDATE productsum
      -
      -
      -
      SET sale_price = '5000'
      WHERE product_type = '办公用品';
```

此时我们再查看 productSum 视图，可以发现数据已经更新了

	product_type	sale_price
1	衣服	1000
2	厨房用具	3000
3	厨房用具	500
4	办公用品	5000

图 4.7: 视图查询结果

此时观察原表也可以发现数据也被更新了

product_id	product_name	product_type	sale_price	purchase_price	regist_date
1 0001	T恤	衣服	1000	500	2009-09-20
2 0002	打孔器	办公用品	500	320	2009-09-11
3 0003	运动T恤	衣服	4000	2800	<null>
4 0004	菜刀	厨房用具	3000	2800	2009-09-20
5 0005	高压锅	厨房用具	6800	5000	2009-01-15
6 0006	叉子	厨房用具	500	<null>	2009-09-20
7 0007	擦菜板	厨房用具	880	790	2008-04-28
8 0008	圆珠笔	办公用品	5000	<null>	2009-11-11

图 4.8: 视图查询结果

不知道大家看到这个结果会不会有疑问，刚才修改视图的时候是设置 `product_type='办公用品'` 的商品的 `sale_price=5000`，为什么原表的数据只有一条做了修改呢？

还是因为视图的定义，视图只是原表的一个窗口，所以它修改也只能修改透过窗口能看到的内容。

注意：这里虽然修改成功了，但是并不推荐这种使用方式。而且我们在创建视图时也尽量使用限制不允许通过视图来修改表

4.1.7 如何删除视图

删除视图的基本语法如下：

源码地址 [sql](#)

```
Line 1  DROP VIEW <视图名1> [ , <视图名2> ... ]
```

注意：需要有相应的权限才能成功删除。

- 删除视图

我们删除刚才创建的 `productSum` 视图

源码地址 [sql](#)

```
Line 1  DROP VIEW productSum;
```

如果我们继续操作这个视图的话就会提示当前操作的内容不存在。

4.2 子查询

我们先来看一个语句（仅做示例，未提供相关数据）

源码地址 [sql](#)

```
Line 1  SELECT stu_name
      - FROM (
```

```

-     SELECT stu_name, COUNT(*) AS stu_cnt
-         FROM students_info
5          GROUP BY stu_age) AS studentSum;

```

这个语句看起来很好理解，其中使用括号括起来的 sql 语句首先执行，执行成功后再执行外面的 sql 语句。但是我们上一节提到的视图也是根据 SELECT 语句创建视图然后在这个基础上再进行查询。那么什么是子查询呢？子查询和视图又有什么关系呢？

4.2.1 什么是子查询

子查询指一个查询语句嵌套在另一个查询语句内部的查询，这个特性从 MySQL 4.1 开始引入，在 SELECT 子句中先计算子查询，子查询结果作为外层另一个查询的过滤条件，查询可以基于一个表或者多个表。

4.2.2 子查询和视图的关系

子查询就是将用来定义视图的 SELECT 语句直接用于 FROM 子句当中。其中 AS studentSum 可以看作是子查询的名称，而且由于子查询是一次性的，所以子查询不会像视图那样保存在存储介质中，而是在 SELECT 语句执行之后就消失了。

4.2.3 嵌套子查询

与在视图上再定义视图类似，子查询也没有具体的限制，例如我们可以这样

源码地址 [sql](#)

```

Line 1  SELECT product_type, cnt_product
-      FROM (SELECT *
-              FROM (SELECT product_type,
-                          COUNT(*) AS cnt_product
-                  FROM product
-                 GROUP BY product_type) AS productsum
-      WHERE cnt_product = 4) AS productsum2;

```

其中最内层的子查询我们将其命名为 productSum，这条语句根据 product_type 分组并查询个数，第二层查询中将个数为 4 的商品查询出来，最外层查询 product_type 和 cnt_product 两列。虽然嵌套子查询可以查询出结果，但是随着子查询嵌套的层数的叠加，SQL 语句不仅会难以理解而且执行效率也会很差，所以要尽量避免这样的使用。

4.2.4 标量子查询

标量就是单一的意思，那么标量子查询也就是单一的子查询，那什么叫做单一的子查询呢？

所谓单一就是要求我们执行的 SQL 语句只能返回一个值，也就是要返回表中具体的某一行的某一列。例如我们有下面这样一张表

源码地址 [sql](#)

product_id	product_name	sale_price
0003	运动T恤	4000
0004	菜刀	3000

5	0005	高压锅	6800
---	------	-----	------

那么我们执行一次标量子查询后是要返回类似于，“0004”，“菜刀”这样的结果。

4.2.5 标量子查询有什么用

我们现在已经知道标量子查询可以返回一个值了，那么它有什么作用呢？

直接这样想可能会有些困难，让我们看几个具体的需求：

1. 查询出销售单价高于平均销售单价的商品
2. 查询出注册日期最晚的那个商品

你有思路了吗？

让我们看如何通过标量子查询语句查询出销售单价高于平均销售单价的商品。

[源码地址 sql](#)

```
Line 1 SELECT product_id, product_name, sale_price
-      FROM product
- WHERE sale_price > (SELECT AVG(sale_price) FROM product);
```

上面的这条语句首先后半部分查询出 product 表中的平均售价，前面的 sql 语句在根据 WHERE 条件挑选出合适的商品。由于标量子查询的特性，导致标量子查询不仅仅局限于 WHERE 子句中，通常任何可以使用单一值的位置都可以使用。也就是说，能够使用常数或者列名的地方，无论是 SELECT 子句、GROUP BY 子句、HAVING 子句，还是 ORDER BY 子句，几乎所有的地方都可以使用。

我们还可以这样使用标量子查询：

[源码地址 sql](#)

```
Line 1 SELECT product_id,
-          product_name,
-          sale_price,
-          (SELECT AVG(sale_price)
-             FROM product) AS avg_price
-      FROM product;
```

你能猜到这段代码的运行结果是什么吗？运行一下看看与你想象的结果是否一致。

4.2.6 关联子查询

- 什么是关联子查询

关联子查询既然包含关联两个字那么一定意味着查询与子查询之间存在着联系。这种联系是如何建立起来的呢？

我们先看一个例子：

[源码地址 sql](#)

```
Line 1 SELECT product_type, product_name, sale_price
-      FROM product AS p1
- WHERE sale_price > (SELECT AVG(sale_price)
-                         FROM product AS p2
-                         WHERE p1.product_type = p2.product_type
- GROUP BY product_type);
```

你能理解这个例子在做什么操作么？先来看一下这个例子的执行结果

product_type	product_name	sale_price
1 办公用品	打孔器	500
2 衣服	运动T恤	4000
3 厨房用具	菜刀	3000
4 厨房用具	高压锅	6800

图 4.9: 视图查询结果

通过上面的例子我们大概可以猜到吗，关联子查询就是通过一些标志将内外两层的查询连接起来起到过滤数据的目的，接下来我们就一起看一下关联子查询的具体内容吧。

- 关联子查询与子查询的联系

还记得我们之前的那个例子么‘查询出销售单价高于平均销售单价的商品’，这个例子的 SQL 语句如下

源码地址 [sql](#)

```
Line 1 SELECT product_id, product_name, sale_price
  -   FROM product
  - WHERE sale_price > (SELECT AVG(sale_price) FROM product);
```

我们再来看一下这个需求‘选取各商品种类中高于该商品种类的平均销售单价的商品’。SQL 语句如下：

源码地址 [sql](#)

```
Line 1 SELECT product_type, product_name, sale_price
  -   FROM product ASp1
  - WHERE sale_price > (SELECT AVG(sale_price)
  -   FROM product ASp2
  -           WHERE p1.product_type =p2.product_type
  - GROUP BY product_type);
```

可以看出上面这两个语句的区别吗？在第二条 SQL 语句也就是关联子查询中我们将外面的 product 表标记为 p1，将内部的 product 设置为 p2，而且通过 WHERE 语句连接了两个查询。

但是如果刚接触的话一定会比较疑惑关联查询的执行过程，这里有一个 [博客](<https://zhuanlan.zhihu.com/p/4184>) 讲的比较清楚。在这里我们简要的概括为：

1. 首先执行不带 WHERE 的主查询
2. 根据主查询结果匹配 product_type，获取子查询结果
3. 将子查询结果再与主查询结合执行完整的 SQL 语句

在子查询中像标量子查询，嵌套子查询或者关联子查询可以看作是子查询的一种操作方式即可。

小结

视图和子查询是数据库操作中较为基础的内容，对于一些复杂的查询需要使用子查询加一些条件语句组合才能得到正确的结果。但是无论如何对于一个 SQL 语句来说都不应该设计的层数非常深且特别复杂，不仅可读性差而且执行效率也难以保证，所以尽量有简洁的语句来完成需要的功能。

练习题-第一部分

4.1

创建出满足下述三个条件的视图（视图名称为 ViewPractice5_1）。使用 product（商品）表作为参照表，假设表中包含初始状态的 8 行数据。

- 条件 1：销售单价大于等于 1000 日元。
- 条件 2：登记日期是 2009 年 9 月 20 日。
- 条件 3：包含商品名称、销售单价和登记日期三列。

对该视图执行 SELECT 语句的结果如下所示。

[源码地址 sql](#)

```
Line 1  SELECT * FROM ViewPractice5_1;
```

执行结果

[源码地址 sql](#)

product_name	sale_price	regist_date
T恤衫	1000	2009-09-20
菜刀	3000	2009-09-20

4.2

向习题一中创建的视图 ViewPractice5_1 中插入如下数据，会得到什么样的结果呢？

[源码地址 sql](#)

```
Line 1  INSERT INTO ViewPractice5_1 VALUES ('刀子', 300, '2009-11-02');
```

4.3

请根据如下结果编写 SELECT 语句，其中 sale_price_all 列为全部商品的平均销售单价。

[源码地址 sql](#)

product_id	product_name	product_type	sale_price	sale_price_all
0001	T恤衫	衣服	1000	2097.5000000000000000
0002	打孔器	办公用品	500	
				2097.5000000000000000
0003	运动T恤	衣服	4000	2097.5000000000000000
0004	菜刀	厨房用具	3000	
				2097.5000000000000000
0005	高压锅	厨房用具	6800	
				2097.5000000000000000

-	0006	叉子	厨房用具	500	
	2097.	500000000000000000			
-	0007	擦菜板	厨房用具	880	
	2097.	500000000000000000			
10	0008	圆珠笔	办公用品	100	
	2097.	500000000000000000			

4.4

请根据习题一中的条件编写一条 SQL 语句, 创建一幅包含如下数据的视图(名称为 AvgPriceByType)。

源码地址 [sql](#)

Line	product_id	product_name	product_type	sale_price	avg_sale_price
-	0001	T恤衫	衣服	1000	2500.0000000000000000
-	0002	打孔器	办公用品	500	300.0000000000000000
5	0003	运动T恤	衣服	4000	2500.0000000000000000
-	0004	菜刀	厨房用具	3000	
		2795.0000000000000000			
-	0005	高压锅	厨房用具	6800	
		2795.0000000000000000			
-	0006	叉子	厨房用具	500	
		2795.0000000000000000			
-	0007	擦菜板	厨房用具	880	
		2795.0000000000000000			
10	0008	圆珠笔	办公用品	100	
		300.0000000000000000			

提示: 其中的关键是 avg_sale_price 列。与习题三不同, 这里需要计算出的是各商品种类的平均销售单价。这与使用关联子查询所得到的结果相同。也就是说, 该列可以使用关联子查询进行创建。问题就是应该在什么地方使用这个关联子查询。

4.3 各种各样的函数

sql 自带了各种各样的函数, 极大提高了 sql 语言的便利性。

所谓函数, 类似一个黑盒子, 你给它一个输入值, 它便按照预设的程序定义给出返回值, 输入值称为参数。

函数大致分为如下几类:

- 算术函数 (来进行数值计算的函数)
- 字符串函数 (来进行字符串操作的函数)
- 日期函数 (来进行日期操作的函数)
- 转换函数 (来进行转换数据类型和值的函数)
- 聚合函数 (来进行数据聚合的函数)

函数总个数超过 200 个，不需要完全记住，常用函数有 30~50 个，其他不常用的函数使用时查阅文档即可。

4.3.1 算数函数

+ - * / 四则运算在之前的章节介绍过，此处不再赘述。

为了演示其他的几个算数函数，在此构造 ‘samplemath’ 表

源码地址 [sql](#)

```
Line 1  -- DDL : 创建表
- USE shop;
- DROP TABLE IF EXISTS samplemath;
- CREATE TABLE samplemath
5   (m float(10,3),
- n INT,
- p INT);

-
- -- DML : 插入数据
10 START TRANSACTION; -- 开始事务
- INSERT INTO samplemath(m, n, p) VALUES (500, 0, NULL);
- INSERT INTO samplemath(m, n, p) VALUES (-180, 0, NULL);
- INSERT INTO samplemath(m, n, p) VALUES (NULL, NULL, NULL);
- INSERT INTO samplemath(m, n, p) VALUES (NULL, 7, 3);
15 INSERT INTO samplemath(m, n, p) VALUES (NULL, 5, 2);
- INSERT INTO samplemath(m, n, p) VALUES (NULL, 4, NULL);
- INSERT INTO samplemath(m, n, p) VALUES (8, NULL, 3);
- INSERT INTO samplemath(m, n, p) VALUES (2.27, 1, NULL);
- INSERT INTO samplemath(m, n, p) VALUES (5.555, 2, NULL);
20 INSERT INTO samplemath(m, n, p) VALUES (NULL, 1, NULL);
- INSERT INTO samplemath(m, n, p) VALUES (8.76, NULL, NULL);
- COMMIT; -- 提交事务
- -- 查询表内容
- SELECT * FROM samplemath;

25 +-----+-----+-----+
- | m      | n      | p      |
- +-----+-----+-----+
- | 500.000 | 0     | NULL   |
- | -180.000 | 0     | NULL   |
30 |    NULL | NULL   | NULL   |
- |    NULL | 7     | 3      |
- |    NULL | 5     | 2      |
- |    NULL | 4     | NULL   |
- | 8.000  | NULL   | 3      |
35 | 2.270  | 1     | NULL   |
- | 5.555  | 2     | NULL   |
```

```

- |      NULL |      1 |  NULL |
- |  8.760 |  NULL |  NULL |
- +-----+-----+-----+
40 11 rows in set (0.00 sec)

```

- ABS – 绝对值

语法：‘ABS(数值)‘

ABS 函数用于计算一个数字的绝对值，表示一个数到原点的距离。

当 ABS 函数的参数为 ‘NULL‘ 时，返回值也是 ‘NULL‘。

- MOD – 求余数

语法：‘MOD(被除数, 除数)‘

MOD 是计算除法余数（求余）的函数，是 modulo 的缩写。小数没有余数的概念，只能对整数列求余数。

注意：主流的 DBMS 都支持 MOD 函数，只有 SQL Server 不支持该函数，其使用 ‘

- ROUND – 四舍五入

语法：‘ROUND(对象数值, 保留小数的位数)‘

ROUND 函数用来进行四舍五入操作。

注意：当参数 **保留小数的位数** 为变量时，可能会遇到错误，请谨慎使用变量。

源码地址 [sql](#)

```

Line 1  SELECT m,
- ABS(m) AS abs_col ,
- n, p,
- MOD(n, p) AS mod_col,
5 ROUND(m,1) AS round_cols
- FROM samplemath;
- +-----+-----+-----+-----+-----+
- | m      | abs_col | n      | p      | mod_col | round_col |
- +-----+-----+-----+-----+-----+
10 | 500.000 | 500.000 |    0 |  NULL   |  NULL   |  500.0 |
- | -180.000 | 180.000 |    0 |  NULL   |  NULL   | -180.0 |
- |      NULL |      NULL |  NULL |  NULL   |  NULL   |      NULL |
- |      NULL |      NULL |    7 |     3 |      1 |      NULL |
- |      NULL |      NULL |    5 |     2 |      1 |      NULL |
15 |      NULL |      NULL |    4 |  NULL   |  NULL   |      NULL |
- |  8.000  |  8.000  |  NULL |     3 |  NULL   |     8.0 |
- |  2.270  |  2.270  |     1 |  NULL   |  NULL   |     2.3 |
- |  5.555  |  5.555  |     2 |  NULL   |  NULL   |     5.6 |
- |      NULL |      NULL |     1 |  NULL   |  NULL   |      NULL |
20 |  8.760  |  8.760  |  NULL |  NULL   |  NULL   |     8.8 |
- +-----+-----+-----+-----+-----+
- 11 rows in set (0.08 sec)

```

4.3.2 字符串函数

字符串函数也经常被使用，为了学习字符串函数，在此我们构造 ‘samplestr’ 表。

源码地址 [sql](#)

```
Line 1  -- DDL : 创建表
- USE shop;
- DROP TABLE IF EXISTS samplestr;
- CREATE TABLE samplestr
5 (str1 VARCHAR (40),
- str2 VARCHAR (40),
- str3 VARCHAR (40)
- );
- -- DML: 插入数据
10 START TRANSACTION;
- INSERT INTO samplestr (str1, str2, str3) VALUES ('opx', 'rt', NULL);
- INSERT INTO samplestr (str1, str2, str3) VALUES ('abc', 'def', NULL);
- INSERT INTO samplestr (str1, str2, str3) VALUES ('太阳', '月亮', '火星');
- INSERT INTO samplestr (str1, str2, str3) VALUES ('aaa', NULL, NULL);
15 INSERT INTO samplestr (str1, str2, str3) VALUES (NULL, 'xyz', NULL);
- INSERT INTO samplestr (str1, str2, str3) VALUES ('@!#$%', NULL, NULL);
- INSERT INTO samplestr (str1, str2, str3) VALUES ('ABC', NULL, NULL);
- INSERT INTO samplestr (str1, str2, str3) VALUES ('aBC', NULL, NULL);
- INSERT INTO samplestr (str1, str2, str3) VALUES ('abc哈哈', 'abc', 'ABC');
20 INSERT INTO samplestr (str1, str2, str3) VALUES ('abcdefabc', 'abc', 'ABC');
- INSERT INTO samplestr (str1, str2, str3) VALUES ('micmic', 'i', 'I');
- COMMIT;
- -- 确认表中的内容
- SELECT * FROM samplestr;
25 +-----+-----+-----+
- | str1      | str2 | str3 |
- +-----+-----+-----+
- | opx       | rt   | NULL  |
- | abc       | def  | NULL  |
30 | 太阳     | 月亮 | 火星  |
- | aaa       | NULL | NULL  |
- | NULL      | xyz  | NULL  |
- | @!#$%    | NULL | NULL  |
- | ABC       | NULL | NULL  |
35 | aBC       | NULL | NULL  |
- | abc哈哈  | abc  | ABC   |
- | abcdefabc | abc  | ABC   |
- | micmic   | i    | I     |
- +-----+-----+-----+
40 11 rows in set (0.00 sec)
```

- CONCAT – 拼接

语法: ‘CONCAT(str1, str2, str3)‘

MySQL 中使用 CONCAT 函数进行拼接。

- LENGTH – 字符串长度

语法: ‘LENGTH(字符串)‘

- LOWER – 小写转换

LOWER 函数只能针对英文字母使用, 它会将参数中的字符串全都转换为小写。该函数不适用于英文字母以外的场合, 不影响原本就是小写的字符。

类似的, UPPER 函数用于大写转换。

- REPLACE – 字符串的替换

语法: ‘REPLACE(对象字符串, 替换前的字符串, 替换后的字符串)‘

- SUBSTRING – 字符串的截取

语法: ‘SUBSTRING (对象字符串 FROM 截取的起始位置 FOR 截取的字符数)‘

使用 SUBSTRING 函数可以截取出字符串中的一部分字符串。截取的起始位置从字符串最左侧开始计算, 索引值起始为 1。

The screenshot shows the MySQL Workbench interface. The top window is titled '*<MySQL8.0> Script' and contains the following SQL code:

```

SELECT
    str1,
    str2,
    str3,
    CONCAT(str1, str2, str3) AS str_concat,
    LENGTH(str1) AS len_str,
    LOWER(str1) AS low_str,
    REPLACE(str1,str2,str3) AS rep_str,
    SUBSTRING(str1 FROM 3 FOR 2) AS sub_str
FROM
    samplestr;
  
```

The bottom window is titled 'samplestr' and displays the results of the query. The results are presented in a table with the following data:

	ABC str1	ABC str2	ABC str3	str_concat	len_str	low_str	rep_str	sub_str
1	opx	rt	[NULL]	[NULL]	3	opx	[NULL]	x
2	abc	def	[NULL]	[NULL]	3	abc	[NULL]	c
3	太阳	月亮	火星	太阳月亮火星	6	太阳	太阳	
4	aaa	[NULL]	[NULL]	[NULL]	3	aaa	[NULL]	a
5	[NULL]	xyz	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]
6	@!#\$%	[NULL]	[NULL]	[NULL]	5	@!#\$%	[NULL]	#\$
7	ABC	[NULL]	[NULL]	[NULL]	3	abc	[NULL]	C
8	aBC	[NULL]	[NULL]	[NULL]	3	abc	[NULL]	C
9	abc哈哈	abc	ABC	abc哈哈abcABC	9	abc哈哈	ABC哈哈	c哈
10	abcdefabc	abc	ABC	abcdefabcabcABC	9	abcdefabc	ABCdefABC	cd
11	micmic	i	I	micmicil	6	micmic	mlcmic	cm

图 4.10: 字符串函数应用示例

【扩展内容】SUBSTRING_INDEX – 字符串按索引截取

语法: SUBSTRING_INDEX (原始字符串, 分隔符, n)

该函数用来获取原始字符串按照分隔符分割后, 第 n 个分隔符之前 (或之后) 的子字符串, 支持正向和反向索引, 索引起始值分别为 1 和 -1。

[源码地址 sql](#)

```
Line 1 SELECT SUBSTRING_INDEX('www.mysql.com', '.', 2);
+-----+
| SUBSTRING_INDEX('www.mysql.com', '.', 2) |
+-----+
5 | www.mysql                                     |
+-----+
- 1 row in set (0.00 sec)
SELECT SUBSTRING_INDEX('www.mysql.com', '.', -2);
+-----+
10 | SUBSTRING_INDEX('www.mysql.com', '.', -2) |
+-----+
| mysql.com                                      |
+-----+
- 1 row in set (0.00 sec)
```

获取第 1 个元素比较容易，获取第 2 个元素/第 n 个元素可以采用二次拆分的写法。

[源码地址 sql](#)

```
Line 1 SELECT SUBSTRING_INDEX('www.mysql.com', '.', 1);
+-----+
| SUBSTRING_INDEX('www.mysql.com', '.', 1) |
+-----+
5 | www                                         |
+-----+
- 1 row in set (0.00 sec)
SELECT SUBSTRING_INDEX(SUBSTRING_INDEX('www.mysql.com', '.', 2), '.', -1);
+-----+
10 | SUBSTRING_INDEX(SUBSTRING_INDEX('www.mysql.com', '.', 2), '.', -1) |
+-----+
| mysql                                       |
+-----+
- 1 row in set (0.00 sec)
```

4.3.3 日期函数

不同 DBMS 的日期函数语法各有不同，本课程介绍一些被标准 SQL 承认的可以应用于绝大多数 DBMS 的函数。特定 DBMS 的日期函数查阅文档即可。

- CURRENT_DATE – 获取当前日期

[源码地址 sql](#)

```
Line 1 SELECT CURRENT_DATE;
+-----+
| CURRENT_DATE |
```

```

- +-----+
5 | 2020-08-08 |
- +-----+
- 1 row in set (0.00 sec)

```

- CURRENT_TIME – 当前时间

[源码地址 sql](#)

```

Line 1 SELECT CURRENT_TIME;
- +-----+
- | CURRENT_TIME |
- +-----+
5 | 17:26:09 |
- +-----+
- 1 row in set (0.00 sec)

```

- CURRENT_TIMESTAMP – 当前日期和时间

[源码地址 sql](#)

```

Line 1 SELECT CURRENT_TIMESTAMP;
- +-----+
- | CURRENT_TIMESTAMP |
- +-----+
5 | 2020-08-08 17:27:07 |
- +-----+
- 1 row in set (0.00 sec)

```

- EXTRACT – 截取日期元素

语法: ‘EXTRACT(日期元素 FROM 日期)’

使用 EXTRACT 函数可以截取出日期数据中的一部分，例如“年”
“月”，或者“小时”“秒”等。该函数的返回值并不是日期类型而是数值类型

[源码地址 sql](#)

```

Line 1 SELECT CURRENT_TIMESTAMP as now,
- EXTRACT(YEAR    FROM CURRENT_TIMESTAMP) AS year,
- EXTRACT(MONTH   FROM CURRENT_TIMESTAMP) AS month,
- EXTRACT(DAY     FROM CURRENT_TIMESTAMP) AS day,
5 EXTRACT(HOUR    FROM CURRENT_TIMESTAMP) AS hour,
- EXTRACT(MINUTE  FROM CURRENT_TIMESTAMP) AS MINute,
- EXTRACT(SECOND  FROM CURRENT_TIMESTAMP) AS second;
- +-----+-----+-----+-----+-----+-----+
- | now          | year | month | day   | hour | MINute | second |
10 +-----+-----+-----+-----+-----+-----+

```

```

- | 2020-08-08 17:34:38 | 2020 |     8 |     8 |    17 |    34 |    38 |
- +-----+-----+-----+-----+-----+-----+
- 1 row in set (0.00 sec)

```

4.3.4 转换函数

“转换”这个词的含义非常广泛，在 SQL 中主要有两层意思：一是数据类型的转换，简称为类型转换，在英语中称为‘cast’；另一层意思是值的转换。

- CAST – 类型转换

语法：‘CAST (转换前的值 AS 想要转换的数据类型) ‘

[源码地址 sql](#)

```

Line 1 -- 将字符串类型转换为数值类型
- SELECT CAST('0001' AS SIGNED INTEGER) AS int_col;
- +-----+
- | int_col |
5 +-----+
- |      1 |
- +-----+
- 1 row in set (0.00 sec)
- -- 将字符串类型转换为日期类型
10 SELECT CAST('2009-12-14' AS DATE) AS date_col;
- +-----+
- | date_col   |
- +-----+
- | 2009-12-14 |
15 +-----+
- 1 row in set (0.00 sec)

```

- COALESCE – 将 NULL 转换为其他值

语法：‘COALESCE(数据 1, 数据 2, 数据 3……)‘

COALESCE 是 SQL 特有的函数。该函数会返回可变参数 A 中左侧开始第 1 个不是 NULL 的值。参数个数是可变的，因此可以根据需要无限增加。

在 SQL 语句中将 NULL 转换为其他值时就会用到转换函数。

[源码地址 sql](#)

```

Line 1 SELECT COALESCE(NULL, 11) AS col_1,
- COALESCE(NULL, 'hello_world', NULL) AS col_2,
- COALESCE(NULL, NULL, '2020-11-01') AS col_3;
- +-----+-----+-----+
5 | col_1 | col_2       | col_3       |
- +-----+-----+-----+
- |    11 | hello world | 2020-11-01 |

```

```
- +-----+-----+-----+
- |      |
- | 1 row in set (0.00 sec)
```

4.4 谓词

4.4.1 什么是谓词

谓词就是返回值为真值的函数。包括‘TRUE / FALSE / UNKNOWN’。

谓词主要有以下几个：

- LIKE
- BETWEEN
- IS NULL、IS NOT NULL
- IN
- EXISTS

4.4.2 LIKE 谓词 – 用于字符串的部分一致查询

当需要进行字符串的部分一致查询时需要使用该谓词。

部分一致大体可以分为前方一致、中间一致和后方一致三种类型。

首先我们来创建一张表

源码地址 [sql](#)

```
Line 1 -- DDL : 创建表
- CREATE TABLE samplelike
- ( strcol VARCHAR(6) NOT NULL,
- PRIMARY KEY (strcol)
5 samplelike);
- -- DML : 插入数据
- START TRANSACTION; -- 开始事务
- INSERT INTO samplelike (strcol) VALUES ('abcd');
- INSERT INTO samplelike (strcol) VALUES ('ddabc');
10 INSERT INTO samplelike (strcol) VALUES ('abddc');
- INSERT INTO samplelike (strcol) VALUES ('abcdd');
- INSERT INTO samplelike (strcol) VALUES ('dabc');
- INSERT INTO samplelike (strcol) VALUES ('abddc');
- COMMIT; -- 提交事务
15 SELECT * FROM samplelike;
- +-----+
- | strcol |
- +-----+
- | abcd   |
- | abddd  |
20 | abddc  |
- | abddc  |
```

```

- | abdddc |
- | ddabc |
- | dddabc |
25 +-----+
- 6 rows in set (0.00 sec)

```

- 前方一致：选取出“dddabc”

前方一致即作为查询条件的字符串（这里是“ddd”）与查询对象字符串起始部分相同。

[源码地址 sql](#)

```

Line 1 SELECT *
- FROM samplelike
- WHERE strcol LIKE 'ddd%';
- +-----+
5 | strcol |
- +-----+
- | dddabc |
- +-----+
- 1 row in set (0.00 sec)

```

其中的‘

- 中间一致：选取出“abcd”，“dddabc”，“abdddc”

中间一致即查询对象字符串中含有作为查询条件的字符串，无论该字符串出现在对象字符串的最后还是中间都没有关系。

[源码地址 sql](#)

```

Line 1 SELECT *
- FROM samplelike
- WHERE strcol LIKE '%ddd%';
- +-----+
5 | strcol |
- +-----+
- | abcd |
- | abdddc |
- | dddabc |
- +-----+
10 +-----+
- 3 rows in set (0.00 sec)

```

- 后方一致：选取出“abcd”

后方一致即作为查询条件的字符串（这里是“ddd”）与查询对象字符串的末尾部分相同。

[源码地址 sql](#)

```
Line 1 SELECT *
```

```

- FROM samplelike
- WHERE strcol LIKE '%ddd';
- +-----+
5 | strcol |
- +-----+
- | abcdyy |
- +-----+
- 1 row in set (0.00 sec)

```

综合如上三种类型的查询可以看出，查询条件最宽松，也就是能够取得最多记录的是‘中间一致’。这是因为它同时包含前方一致和后方一致的查询结果。

- _ 下划线匹配任意 1 个字符

使用 _（下划线）来代替

[源码地址 sql](#)

```

Line 1 SELECT *
- FROM samplelike
- WHERE strcol LIKE 'abc__';
- +-----+
5 | strcol |
- +-----+
- | abcdy |
- +-----+
- 1 row in set (0.00 sec)

```

4.4.3 BETWEEN 谓词 – 用于范围查询

使用 BETWEEN 可以进行范围查询。该谓词与其他谓词或者函数的不同之处在于它使用了 3 个参数。

[源码地址 sql](#)

```

Line 1 -- 选取销售单价为100 ~ 1000元的商品
- SELECT product_name, sale_price
- FROM product
- WHERE sale_price BETWEEN 100 AND 1000;
5 +-----+-----+
- | product_name | sale_price |
- +-----+-----+
- | T恤          |      1000 |
- | 打孔器        |       500 |
10 | 叉子         |       500 |
- | 擦菜板        |      880 |
- | 圆珠笔        |       100 |
- +-----+-----+
- 5 rows in set (0.00 sec)

```

BETWEEN 的特点就是结果中会包含 100 和 1000 这两个临界值，也就是闭区间。如果不想让结果中包含临界值，那就必须使用 < 和 >。

[源码地址 sql](#)

```
Line 1 SELECT product_name, sale_price
- FROM product
- WHERE sale_price > 100
- AND sale_price < 1000;
5 +-----+-----+
- | product_name | sale_price |
- +-----+-----+
- | 打孔器       |      500 |
- | 叉子         |      500 |
10 | 擦菜板       |     880 |
- +-----+-----+
- 3 rows in set (0.00 sec)
```

4.4.4 IS NULL、IS NOT NULL – 用于判断是否为 NULL

为了选取出某些值为 NULL 的列的数据，不能使用 =，而只能使用特定的谓词 IS NULL。

[源码地址 sql](#)

```
Line 1 SELECT product_name, purchase_price
- FROM product
- WHERE purchase_price IS NULL;
5 +-----+-----+
- | product_name | purchase_price |
- +-----+-----+
- | 叉子         |      NULL |
- | 圆珠笔       |      NULL |
- +-----+-----+
10 2 rows in set (0.00 sec)
```

与此相反，想要选取 NULL 以外的数据时，需要使用 IS NOT NULL。

[源码地址 sql](#)

```
Line 1 SELECT product_name, purchase_price
- FROM product
- WHERE purchase_price IS NOT NULL;
5 +-----+-----+
- | product_name | purchase_price |
- +-----+-----+
- | T恤          |      500 |
- | 打孔器       |      320 |
- | 运动T恤     |     2800 |
10 | 菜刀        |     2800 |
```

```

- | 高压锅      |      5000 |
- | 擦菜板      |       790 |
- +-----+
- 6 rows in set (0.00 sec)

```

4.4.5 IN 谓词 – OR 的简便用法

多个查询条件取并集时可以选择使用‘or’语句。

[源码地址 sql](#)

```

Line 1 -- 通过OR指定多个进货单价进行查询
- SELECT product_name, purchase_price
- FROM product
- WHERE purchase_price = 320
5 OR purchase_price = 500
- OR purchase_price = 5000;
- +-----+
- | product_name | purchase_price |
- +-----+
10 | T恤          |      500 |
- | 打孔器        |       320 |
- | 高压锅        |      5000 |
- +-----+
- 3 rows in set (0.00 sec)

```

虽然上述方法没有问题，但还是存在一点不足之处，那就是随着希望选取的对象越来越多，SQL 语句也会越来越长，阅读起来也会越来越困难。这时，我们就可以使用 IN 谓词 ‘IN(值 1, 值 2, 值 3,)’ 来替换上述 SQL 语句。

[源码地址 sql](#)

```

Line 1 SELECT product_name, purchase_price
- FROM product
- WHERE purchase_price IN (320, 500, 5000);
- +-----+
5 | product_name | purchase_price |
- +-----+
- | T恤          |      500 |
- | 打孔器        |       320 |
- | 高压锅        |      5000 |
10 +-----+
- 3 rows in set (0.00 sec)

```

上述语句简洁了很多，可读性大幅提高。反之，希望选取出“进货单价不是 320 元、500 元、5000 元”的商品时，可以使用否定形式 NOT IN 来实现。

[源码地址 sql](#)

```

Line 1  SELECT product_name, purchase_price
-   FROM product
- WHERE purchase_price NOT IN (320, 500, 5000);
- +-----+
5 | product_name | purchase_price |
- +-----+
- | 运动T恤      |        2800 |
- | 菜刀        |        2800 |
- | 擦菜板      |         790 |
10 +-----+
- 3 rows in set (0.00 sec)

```

需要注意的是，在使用 IN 和 NOT IN 时是无法选取出 NULL 数据的。实际结果也是如此，上述两组结果中都不包含进货单价为 NULL 的叉子和圆珠笔。NULL 只能使用 IS NULL 和 IS NOT NULL 来进行判断。

4.4.6 使用子查询作为 IN 谓词的参数

- IN 和子查询

IN 谓词 (NOT IN 谓词) 具有其他谓词所没有的用法，那就是可以使用子查询作为其参数。我们已经在 5-2 节中学习过了，子查询就是 SQL 内部生成的表，因此也可以说“能够将表作为 IN 的参数”。同理，我们还可以说“能够将视图作为 IN 的参数”。

在此，我们创建一张新表 ‘shopproduct’ 显示出哪些商店销售哪些商品。

源码地址 [sql](#)

```

Line 1  -- DDL : 创建表
- DROP TABLE IF EXISTS shopproduct;
- CREATE TABLE shopproduct
- ( shop_id CHAR(4)      NOT NULL,
5  shop_name VARCHAR(200) NOT NULL,
- product_id CHAR(4)      NOT NULL,
- quantity INTEGER       NOT NULL,
- PRIMARY KEY (shop_id, product_id) -- 指定主键
- );
10 -- DML : 插入数据
- START TRANSACTION; -- 开始事务
- INSERT INTO shopproduct (shop_id, shop_name, product_id, quantity) VALUES ('000A', '东京', '0001', 30);
- INSERT INTO shopproduct (shop_id, shop_name, product_id, quantity) VALUES ('000A', '东京', '0002', 50);
- INSERT INTO shopproduct (shop_id, shop_name, product_id, quantity) VALUES ('000A', '东京', '0003', 15);
15 INSERT INTO shopproduct (shop_id, shop_name, product_id, quantity) VALUES ('000B', '名古屋', '0002', 30);
- INSERT INTO shopproduct (shop_id, shop_name, product_id, quantity) VALUES ('000B', '名古屋', '0003', 120);

```

```

- INSERT INTO shopproduct (shop_id, shop_name, product_id, quantity) VALUES ('000B', '名古屋', '0004', 20);
- INSERT INTO shopproduct (shop_id, shop_name, product_id, quantity) VALUES ('000B', '名古屋', '0006', 10);
- INSERT INTO shopproduct (shop_id, shop_name, product_id, quantity) VALUES ('000B', '名古屋', '0007', 40);
20 INSERT INTO shopproduct (shop_id, shop_name, product_id, quantity) VALUES ('000C', '大阪', '0003', 20);
- INSERT INTO shopproduct (shop_id, shop_name, product_id, quantity) VALUES ('000C', '大阪', '0004', 50);
- INSERT INTO shopproduct (shop_id, shop_name, product_id, quantity) VALUES ('000C', '大阪', '0006', 90);
- INSERT INTO shopproduct (shop_id, shop_name, product_id, quantity) VALUES ('000C', '大阪', '0007', 70);
- INSERT INTO shopproduct (shop_id, shop_name, product_id, quantity) VALUES ('000D', '福冈', '0001', 100);
25 COMMIT; -- 提交事务
- SELECT * FROM shopproduct;
- +-----+-----+-----+-----+
- | shop_id | shop_name | product_id | quantity |
- +-----+-----+-----+-----+
30 | 000A    | 东京      | 0001      |      30 |
- | 000A    | 东京      | 0002      |      50 |
- | 000A    | 东京      | 0003      |      15 |
- | 000B    | 名古屋    | 0002      |      30 |
- | 000B    | 名古屋    | 0003      |     120 |
35 | 000B    | 名古屋    | 0004      |      20 |
- | 000B    | 名古屋    | 0006      |      10 |
- | 000B    | 名古屋    | 0007      |      40 |
- | 000C    | 大阪      | 0003      |      20 |
- | 000C    | 大阪      | 0004      |      50 |
40 | 000C    | 大阪      | 0006      |      90 |
- | 000C    | 大阪      | 0007      |      70 |
- | 000D    | 福冈      | 0001      |     100 |
- +-----+-----+-----+-----+
- 13 rows in set (0.00 sec)

```

由于单独使用商店编号 (shop_id) 或者商品编号 (product_id) 不能区分表中每一行数据，因此指定了 2 列作为主键 (primary key) 对商店和商品进行组合，用来唯一确定每一行数据。

假设我么需要取出大阪在售商品的销售单价，该如何实现呢？

第一步，取出大阪门店的在售商品 ‘product_id’；

第二步，取出大阪门店在售商品的销售单价 ‘sale_price’

[源码地址 sql](#)

Line 1 -- step1: 取出大阪门店的在售商品 product_id

```

Line 2  SELECT product_id
-   FROM shopproduct
- WHERE shop_id = '000C';
5 +-----+
- | product_id |
- +-----+
- | 0003      |
- | 0004      |
10 | 0006      |
- | 0007      |
- +-----+
- 4 rows in set (0.00 sec)

```

上述语句取出了大阪门店的在售商品编号，接下来，我么可以使用上述语句作为第二步的查询条件来使用了。

[源码地址 sql](#)

```

Line 1  -- step2: 取出大阪门店在售商品的销售单价 sale_price
Line 2  SELECT product_name, sale_price
-   FROM product
- WHERE product_id IN (SELECT product_id
5    FROM shopproduct
-                   WHERE shop_id = '000C');
- +-----+-----+
- | product_name | sale_price |
- +-----+-----+
10 | 运动T恤     |      4000 |
- | 菜刀         |      3000 |
- | 叉子         |       500 |
- | 擦菜板       |       880 |
- +-----+-----+
15 | 4 rows in set (0.00 sec)

```

根据之前学习的知识，子查询是从最内层开始执行的（由内而外），因此，上述语句的子查询执行之后，sql 展开成下面的语句

[源码地址 sql](#)

```

Line 1  -- 子查询展开后的结果
-   SELECT product_name, sale_price
-   FROM product
- WHERE product_id IN ('0003', '0004', '0006', '0007');
5 +-----+-----+
- | product_name | sale_price |
- +-----+-----+
- | 运动T恤     |      4000 |
- | 菜刀         |      3000 |

```

```

10 | 叉子          |      500 |
- | 擦菜板        |      880 |
- +-----+
- 4 rows in set (0.00 sec)

```

可以看到，子查询转换之后变为 in 谓词用法，你理解了吗？或者，你会疑惑既然 in 谓词也能实现，那为什么还要使用子查询呢？这里给出两点原因：

- 实际生活中，某个门店的在售商品是不断变化的，使用 in 谓词就需要经常更新 sql 语句，降低了效率，提高了维护成本；
- 实际上，某个门店的在售商品可能有成百上千个，手工维护在售商品编号真是个大工程。

使用子查询即可保持 sql 语句不变，极大提高了程序的可维护性，这是系统开发中需要重点考虑的内容。

- NOT IN 和子查询

NOT IN 同样支持子查询作为参数，用法和 in 完全一样。

[源码地址 sql](#)

```

Line 1 -- NOT IN 使用子查询作为参数，取出未在大阪门店销售的商品的销售单价
- SELECT product_name, sale_price
-   FROM product
- WHERE product_id NOT IN (SELECT product_id
-                            FROM shopproduct
-                            WHERE shop_id = '000A');
- +-----+
- | product_name | sale_price |
- +-----+
- | 菜刀          |      3000 |
- | 高压锅        |      6800 |
- | 叉子          |      500 |
- | 擦菜板        |      880 |
- | 圆珠笔        |      100 |
- +-----+
- 5 rows in set (0.00 sec)

```

4.4.7 EXIST 谓词

EXIST 谓词的用法理解起来有些难度。

EXIST 的使用方法与之前的都不相同

语法理解起来比较困难

实际上即使不使用 EXIST，基本上也都可以使用 IN（或者 NOT IN）来代替

这么说的话，还有学习 EXIST 谓词的必要吗？答案是肯定的，因为一旦能够熟练使用 EXIST 谓词，就能体会到它极大的便利性。

不过，你不用过于担心，本课程介绍一些基本用法，日后学习时可以多多留意 EXIST 谓词的用法，以期能够在达到 SQL 中级水平时掌握此用法。

- EXIST 谓词的使用方法

谓词的作用就是 “判断是否存在满足某种条件的记录”

如果存在这样的记录就返回真 (TRUE)，如果不存在就返回假 (FALSE)。

EXIST (存在) 谓词的主语是 “记录”。

我们继续以 IN 和子查询中的示例，使用 EXIST 选取出大阪门店在售商品的销售单价。

[源码地址 sql](#)

```
Line 1  SELECT product_name, sale_price
      -   FROM product AS p
      - WHERE EXISTS (SELECT *
      -                   FROM shopproduct AS sp
      -                   WHERE sp.shop_id = '000C'
      -                   AND sp.product_id = p.product_id);
      - +-----+-----+
      - | product_name | sale_price |
      - +-----+-----+
10  | 运动T恤       |     4000 |
      - | 菜刀         |     3000 |
      - | 叉子         |      500 |
      - | 擦菜板       |      880 |
      - +-----+-----+
15  4 rows in set (0.00 sec)
```

- EXIST 的参数

之前我们学过的谓词，基本上都是像“列 LIKE 字符串”或者“列 BETWEEN 值 1 AND 值 2”这样需要指定 2 个以上的参数，而 EXIST 的左侧并没有任何参数。因为 EXIST 是只有 1 个参数的谓词。所以，EXIST 只需要在右侧书写 1 个参数，该参数通常都会是一个子查询。

[源码地址 sql](#)

```
Line 1  (SELECT *
      -   FROM shopproduct AS sp
      -   WHERE sp.shop_id = '000C'
      -   AND sp.product_id = p.product_id)
```

上面这样的子查询就是唯一的参数。确切地说，由于通过条件 “SP.product_id = P.product_id” 将 product 表和 shopproduct 表进行了联接，因此作为参数的是关联子查询。EXIST 通常会使用关联子查询作为参数。

- 子查询中的 SELECT

由于 EXIST 只关心记录是否存在，因此返回哪些列都没有关系。EXIST 只会判断是否存在满足子查询中 WHERE 子句指定的条件 “商店编号 (shop_id) 为'000C'，商品 (product) 表和商店

商品 (shopproduct) 表中商品编号 (product_id) 相同” 的记录，只有存在这样的记录时才返回真 (TRUE)。

因此，使用下面的查询语句，查询结果也不会发生变化。

源码地址 [sql](#)

```

Line 1  SELECT product_name, sale_price
      -   FROM product AS p
      - WHERE EXISTS (SELECT 1 -- 这里可以书写适当的常数
      -                   FROM shopproduct AS sp
      -                   WHERE sp.shop_id = '000C'
      -                   AND sp.product_id = p.product_id);
      - +-----+-----+
      - | product_name | sale_price |
      - +-----+-----+
10  | 运动T恤       |     4000 |
      - | 菜刀         |     3000 |
      - | 叉子         |      500 |
      - | 擦菜板       |     880 |
      - +-----+-----+
15  4 rows in set (0.00 sec)

```

大家可以把在 EXIST 的子查询中书写 SELECT * 当作 SQL 的一种习惯。

- 使用 NOT EXIST 替换 NOT IN

就像 EXIST 可以用来替换 IN 一样，NOT IN 也可以用 NOT EXIST 来替换。

下面的代码示例取出，不在大阪门店销售的商品的销售单价。

源码地址 [sql](#)

```

Line 1  SELECT product_name, sale_price
      -   FROM product AS p
      - WHERE NOT EXISTS (SELECT *
      -                   FROM shopproduct AS sp
      -                   WHERE sp.shop_id = '000A'
      -                   AND sp.product_id = p.product_id);
      - +-----+-----+
      - | product_name | sale_price |
      - +-----+-----+
10  | 菜刀         |     3000 |
      - | 高压锅       |     6800 |
      - | 叉子         |      500 |
      - | 擦菜板       |     880 |
      - | 圆珠笔       |      100 |
      - +-----+-----+
15  5 rows in set (0.00 sec)

```

NOT EXIST 与 EXIST 相反，当“不存在”满足子查询中指定条件的记录时返回真 (TRUE)。

4.5 CASE 表达式

4.5.1 什么是 CASE 表达式？

CASE 表达式是函数的一种。是 SQL 中数一数二的重要功能，有必要好好学习一下。

CASE 表达式是在区分情况时使用的，这种情况的区分在编程中通常称为（条件）分支。

CASE 表达式的语法分为简单 CASE 表达式和搜索 CASE 表达式两种。由于搜索 CASE 表达式包含简单 CASE 表达式的全部功能。本课程将重点介绍搜索 CASE 表达式。

语法：

源码地址 [sql](#)

```
Line 1 CASE WHEN <求值表达式> THEN <表达式>
      - WHEN <求值表达式> THEN <表达式>
      - WHEN <求值表达式> THEN <表达式>
      -
      5   .
      .
      - ELSE <表达式>
      - END
```

上述语句执行时，依次判断 when 表达式是否为真值，是则执行 THEN 后的语句，如果所有的 when 表达式均为假，则执行 ELSE 后的语句。无论多么庞大的 CASE 表达式，最后也只会返回一个值。

4.5.2 CASE 表达式的使用方法

假设现在要实现如下结果：

源码地址 [sql](#)

```
Line 1 A : 衣服
      - B : 办公用品
      - C : 厨房用具
```

因为表中的记录并不包含“A :”或者“B :”这样的字符串，所以需要在 SQL 中进行添加。并将“A :”“B :”“C :”与记录结合起来。

应用场景 1：根据不同分支得到不同列值

源码地址 [sql](#)

```
Line 1 SELECT product_name,
      - CASE WHEN product_type = '衣服' THEN CONCAT('A_: ',product_type)
            - WHEN product_type = '办公用品' THEN CONCAT('B_: ',product_type)
            - WHEN product_type = '厨房用具' THEN CONCAT('C_: ',product_type)
            - ELSE NULL
      - END AS abc_product_type
      - FROM product;
      - +-----+
      - | product_name | abc_product_type |
      10 | +-----+-----+
```

```

- | T恤          | A : 衣服      |
- | 打孔器        | B : 办公用品   |
- | 运动T恤        | A : 衣服      |
- | 菜刀          | C : 厨房用具   |
15 | 高压锅        | C : 厨房用具   |
- | 叉子          | C : 厨房用具   |
- | 擦菜板        | C : 厨房用具   |
- | 圆珠笔        | B : 办公用品   |
- +-----+
20 8 rows in set (0.00 sec)

```

ELSE 子句也可以省略不写，这时会被默认为 ELSE NULL。但为了防止有人漏读，还是希望大家能够显示地写出 ELSE 子句。此外，CASE 表达式最后的“END”是不能省略的，请大家特别注意不要遗漏。忘记书写 END 会发生语法错误，这也是初学时最容易犯的错误。

应用场景 2：实现列方向上的聚合

通常我们使用如下代码实现行的方向上不同种类的聚合（这里是 sum）

源码地址 [sql](#)

```

Line 1  SELECT product_type,
-         SUM(sale_price) AS sum_price
-     FROM product
-   GROUP BY product_type;
5  +-----+
-  | product_type | sum_price |
-  +-----+
-  | 衣服          |      5000 |
-  | 办公用品      |       600 |
10 | 厨房用具      |    11180 |
-  +-----+
- 3 rows in set (0.00 sec)

```

假如要在列的方向上展示不同种类额聚合值，该如何写呢？

源码地址 [sql](#)

```

Line 1  sum_price_clothes | sum_price_kitchen | sum_price_office
-  +-----+-----+-----+
-      5000 |        11180 |        600

```

聚合函数 + CASE WHEN 表达式即可实现该效果

源码地址 [sql](#)

```

Line 1  -- 对按照商品种类计算出的销售单价合计值进行行列转换
-  SELECT SUM(CASE WHEN product_type = '衣服' THEN sale_price ELSE 0 END) AS
-         sum_price_clothes,
-         SUM(CASE WHEN product_type = '厨房用具' THEN sale_price ELSE 0 END) AS
-         sum_price_kitchen,

```

```

        SUM(CASE WHEN product_type = '办公用品' THEN sale_price ELSE 0 END) AS
        sum_price_office
5   FROM product;
+-----+-----+-----+
| sum_price_clothes | sum_price_kitchen | sum_price_office |
+-----+-----+-----+
|      5000 |          11180 |          600 |
10 +-----+-----+-----+
- 1 row in set (0.00 sec)

```

【扩展内容】应用场景 3：实现行转列

假设有如下图表的结构

name	subject	score
张三	语文	93
张三	数学	88
张三	外语	91
李四	语文	87
李四	数学	90
李四	外语	77

图 4.11: 表数据示例

计划得到如下的图表结构

name	chinese	math	english
张三	93	88	91
李四	87	90	77

图 4.12: 预期结果

聚合函数 + CASE WHEN 表达式即可实现该转换

源码地址 [sql](#)

```

Line 1 -- CASE WHEN 实现数字列 score 行转列
- SELECT name,
-         SUM(CASE WHEN subject = '语文' THEN score ELSE null END) as chinese,
-         SUM(CASE WHEN subject = '数学' THEN score ELSE null END) as math,
5          SUM(CASE WHEN subject = '外语' THEN score ELSE null END) as english
-     FROM score
-    GROUP BY name;
+-----+-----+-----+
- | name | chinese | math | english |
10 +-----+-----+-----+
- | 张三 |      93 |    88 |      91 |

```

```

- | 李四 |     87 |    90 |     77 |
- +-----+-----+-----+
- 2 rows in set (0.00 sec)

```

上述代码实现了数字列 score 的行转列，也可以实现文本列 subject 的行转列

[源码地址 sql](#)

```

Line 1 -- CASE WHEN 实现文本列 subject 行转列
- SELECT name,
-       MAX(CASE WHEN subject = '语文' THEN subject ELSE null END) as chinese,
-       MAX(CASE WHEN subject = '数学' THEN subject ELSE null END) as math,
5      MIN(CASE WHEN subject = '外语' THEN subject ELSE null END) as english
- FROM score
- GROUP BY name;
- +-----+-----+-----+
- | name | chinese | math | english |
10 +-----+-----+-----+
- | 张三 | 语文     | 数学   | 外语     |
- | 李四 | 语文     | 数学   | 外语     |
- +-----+-----+-----+
- 2 rows in set (0.00 sec)

```

总结：

- 当待转换列为数字时，可以使用 ‘SUM AVG MAX MIN‘等聚合函数；
- 当待转换列为文本时，可以使用 ‘MAX MIN‘等聚合函数

练习题-第二部分

4.5

运算或者函数中含有 NULL 时，结果全都会变为 NULL ? (判断题)

4.6

对本章中使用的 product (商品) 表执行如下 2 条 SELECT 语句，能够得到什么样的结果呢？

[源码地址 sql](#)

```

Line 1 SELECT product_name, purchase_price
-   FROM product
- WHERE purchase_price NOT IN (500, 2800, 5000);

```

[源码地址 sql](#)

```

Line 1 SELECT product_name, purchase_price
-   FROM product
- WHERE purchase_price NOT IN (500, 2800, 5000, NULL);

```

4.7

按照销售单价 (sale_price) 对练习 6.1 中的 product (商品) 表中的商品进行如下分类。

- 低档商品：销售单价在 1000 日元以下 (T 恤衫、办公用品、叉子、擦菜板、圆珠笔)
- 中档商品：销售单价在 1001 日元以上 3000 日元以下 (菜刀)
- 高档商品：销售单价在 3001 日元以上 (运动 T 恤、高压锅)

请编写出统计上述商品种类中所包含的商品数量的 SELECT 语句，结果如下所示。

执行结果

源码地址 [sql](#)

```
Line 1  low_price | mid_price | high_price
- - - - + - - - - + - - - -
-      5 |          1 |          2
```

第 5 章 集合运算

5.1 表的加减法

5.1.1 什么是集合运算

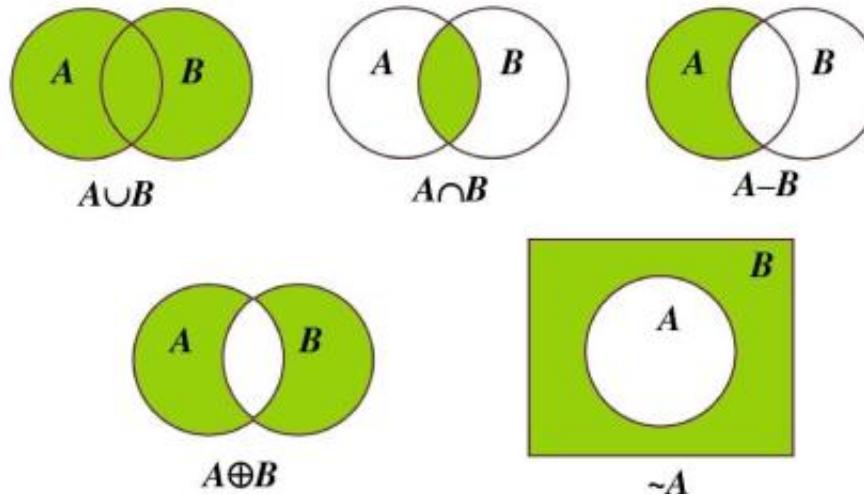
集合在数学领域表示“各种各样的事物的总和”，在数据库领域表示记录的集合。具体来说，表、视图和查询的执行结果都是记录的集合，其中的元素为表或者查询结果中的每一行。

在标准 SQL 中，分别对检索结果使用 UNION, INTERSECT, EXCEPT 来将检索结果进行并、交和差运算，像 UNION, INTERSECT, ‘EXCEPT’这种用来进行集合运算的运算符称为集合运算符。

以下的文氏图展示了几种集合的基本运算。

文氏图

集合运算的表示



7

图 5.1: 文氏图 1

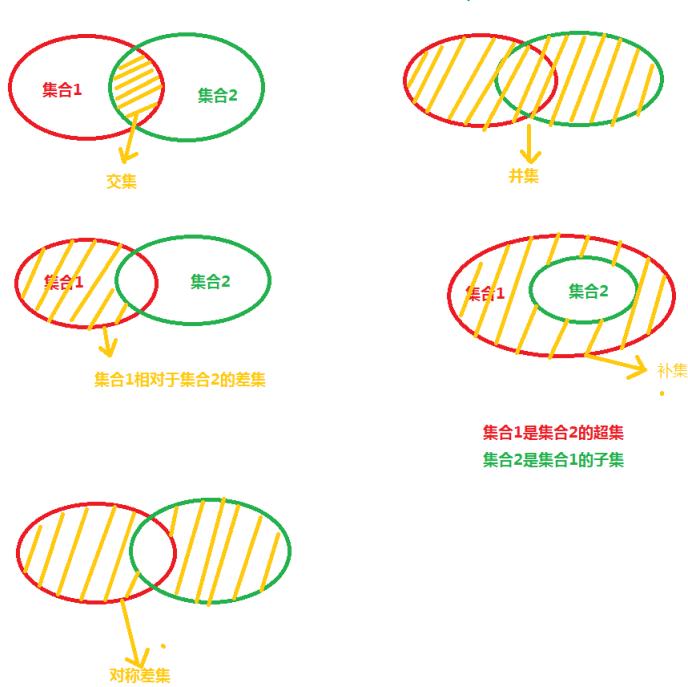


图 5.2: 文氏图 2

在数据库中, 所有的表--以及查询结果--都可以视为集合, 因此也可以把表视为集合进行上述集合运算, 在很多时候, 这种抽象非常有助于对复杂查询问题给出一个可行的思路.

5.1.2 表的加法—UNION

- UNION

建表代码及数据导入请使用第一章提供的代码。

接下来我们演示 UNION 的具体用法及查询结果:

源码地址 [sql](#)

```
Line 1  SELECT product_id, product_name
      -   FROM product
      - UNION
      - SELECT product_id, product_name
      5    FROM product2;
```

上述结果包含了两张表中的全部商品. 你会发现, 这就是我们在学校学过的集合中的并集运算, 通过文氏图会看得更清晰 (图 4-3):

图7-1 使用UNION对表进行加法(并集)运算的图示

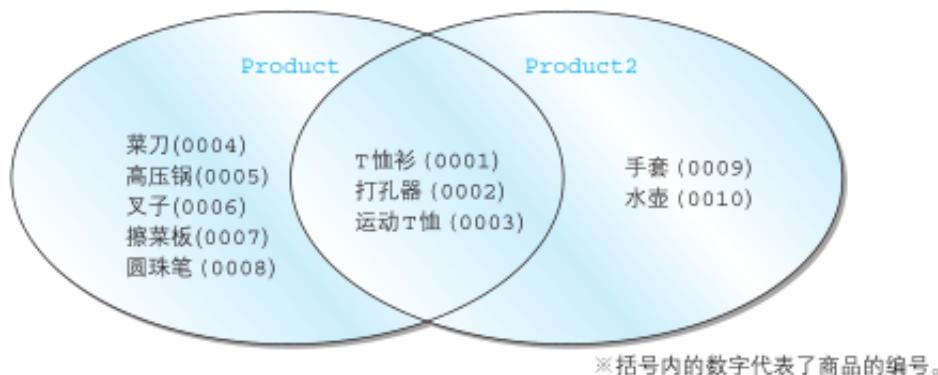


图 5.3: 使用 UNION 对表进行加法运算的图示

通过观察可以发现,商品编号为“0001”“0003”的3条记录在两个表中都存在,因此大家可能会认为结果中会出现重复的记录,但是 UNION 等集合运算符通常都会除去重复的记录。

上述查询是对不同的两张表进行求并集运算.对于同一张表,实际上也是可以进行求并集的.

练习题假设连锁店想要增加毛利率超过 50

结果应该类似于:

product_id	product_name	product_type	sale_price	purchase_price
0002	打孔器	办公用品	500	320
0006	叉子	厨房用具	500	(NULL)
0008	圆珠笔	办公用品	100	(NULL)
0001	T恤	衣服	1000	500

图 5.4: 查询结果

源码地址 [sql](#)

```
Line 1 -- 参考答案:
- SELECT product_id,product_name,product_type
-     ,sale_price,purchase_price
-   FROM product
5 WHERE sale_price<800

-
- UNION

-
- SELECT product_id,product_name,product_type
10    ,sale_price,purchase_price
-   FROM product
- WHERE sale_price>1.5*purchase_price;
```

思考:如果不使用 UNION 该怎么写查询语句?

源码地址 [sql](#)

```
Line 1 -- 参考答案:
```

```

- SELECT product_id,product_name,product_type
- ,sale_price,purchase_price
- FROM product
5 WHERE sale_price < 800
- OR sale_price > 1.5 * purchase_price;

```

- UNION 与 OR 谓词

对于上边的练习题, 如果你已经正确地写出来查询, 你会发现, 使用 UNION 对两个查询结果取并集, 和在一个查询中使用 WHERE 子句, 然后使用 OR 谓词连接两个查询条件, 能够得到相同的结果.

那么是不是就没必要引入 UNION 了呢? 当然不是这样的. 确实, 对于同一个表的两个不同的筛选结果集, 使用 UNION 对两个结果集取并集, 和把两个子查询的筛选条件用 OR 谓词连接, 会得到相同的结果, 但倘若要将两个不同的表中的结果合并在一起, 就不得不使用 UNION 了.

而且, 即便是对于同一张表, 有时也会出于查询效率方面的因素来使用 UNION.

练习题

分别使用 UNION 或者 OR 谓词, 找出毛利率不足 30

参考答案:

[源码地址 sql](#)

```

Line 1 -- 使用 OR 谓词
- SELECT *
- FROM product
- WHERE sale_price / purchase_price < 1.3
5 OR sale_price / purchase_price IS NULL;

```

[源码地址 sql](#)

```

Line 1 -- 使用 UNION
- SELECT *
- FROM product
- WHERE sale_price / purchase_price < 1.3
5
- UNION
- SELECT *
- FROM product
- WHERE sale_price / purchase_price IS NULL;

```

- 包含重复行的集合运算 UNION ALL

SQL 语句的 UNION 会对两个查询的结果集进行合并和去重, 这种去重不仅会去掉两个结果集相互重复的, 还会去掉一个结果集中的重复行. 但在实践中有时候需要不去重的并集, 在 UNION 的结果中保留重复行的语法其实非常简单, 只需要在 UNION 后面添加 ALL 关键字就可以了.

例如, 想要知道 product 和 product2 中所包含的商品种类及每种商品的数量, 第一步, 就需要将两个表的商品种类字段选出来, 然后使用 UNION ALL 进行不去重地合并. 接下来再对两个表的结果按 product_type 字段分组计数.

源码地址 [sql](#)

```
Line 1 -- 保留重复行
- SELECT product_id, product_name
-   FROM product
- UNION ALL
5  SELECT product_id, product_name
-   FROM product2;
```

查询结果如下：

product_id	product_name
0001	T恤
0002	打孔器
0003	运动T恤
0004	菜刀
0005	高压锅
0006	叉子
0007	擦菜板
0008	圆珠笔
0001	T恤
0002	打孔器
0003	运动T恤
0009	手套
0010	水壶

图 5.5: 查询结果

练习题

商店决定对 product 表中利润低于 50

product_id	product_name	product_type	sale_price	purchase_price	regist_date
0002	打孔器	办公用品	500	320	2009-09-11
0006	叉子	厨房用具	500	(NULL)	2009-09-20
0007	擦菜板	厨房用具	880	790	2008-04-28
0008	圆珠笔	办公用品	100	(NULL)	2009-11-11
0001	T恤	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11

图 5.6: 查询结果

参考答案

源码地址 [sql](#)

```
Line 1 SELECT *
-   FROM product
- WHERE sale_price < 1000
- UNION ALL
5  SELECT *
```

```
- FROM product
- WHERE sale_price > 1.5 * purchase_price
```

- 【扩展阅读】bag 模型与 set 模型

在高中数学课上我们就学过, 集合的一个显著的特征就是集合中的元素都是互异的. 当我们把数据库中的表看作是集合的时候, 实际上存在一些问题的: 不论是有意的设计或无意的过失, 很多数据库中的表包含了重复的行.

Bag 是和 set 类似的一种数学结构, 不一样的地方在于: bag 里面允许存在重复元素, 如果同一个元素被加入多次, 则袋子里就有多个该元素.

通过上述 bag 与 set 定义之间的差别我们就发现, 使用 bag 模型来描述数据库中的表很多时候更加合适.

是否允许元素重复导致了 set 和 bag 的并交差等运算都存在一些区别. 以 bag 的交为例, 由于 bag 允许元素重复出现, 对于两个 bag, 他们的并运算会按照:

1. 该元素是否至少在一个 bag 里出现过;

2. 该元素在两个 bag 中的最大出现次数这两个方面来进行计算. 因此对于 $A = 1,1,1,2,3,5,7$, $B = 1,1,2,2,4,6,8$ 两个 bag, 它们的并就等于 $1,1,1,2,2,3,4,5,6,7,8$.

- 隐式类型转换

通常来说, 我们会把类型完全一致, 并且代表相同属性的列使用 UNION 合并到一起显示, 但有时候, 即使数据类型不完全相同, 也会通过隐式类型转换来将两个类型不同的列放在一列里显示, 例如字符串和数值类型:

源码地址 [sql](#)

```
Line 1  SELECT product_id, product_name, '1'
-      FROM product
-      UNION
-  SELECT product_id, product_name,sale_price
5     FROM product2;
```

上述查询能够正确执行, 得到如下结果:

product_id	product_name	1
0001	T恤	1
0002	打孔器	1
0003	运动T恤	1
0004	菜刀	1
0005	高压锅	1
0006	叉子	1
0007	擦菜板	1
0008	圆珠笔	1
0001	T恤	1000
0002	打孔器	500
0003	运动T恤	4000
0009	手套	800
0010	水壶	2000

图 5.7: 查询结果

练习题

使用 SYSDATE() 函数可以返回当前日期时间, 是一个日期时间类型的数据, 试测试该数据类型和数值, 字符串等类型的兼容性.

例如, 以下代码可以正确执行, 说明时间日期类型和字符串, 数值以及缺失值均能兼容.

[源码地址 sql](#)

```
Line 1 SELECT SYSDATE(), SYSDATE(), SYSDATE()
-
- UNION
-
5 SELECT 'chars', 123, null
```

上述代码的查询结果:

SYSDATE()	SYSDATE() (1)	SYSDATE() (2)
2020-08-25 15:51:48	2020-08-25 15:51:48	2020-08-25 15:51:48
chars	123	(Null)

图 5.8: 查询结果

5.1.3 MySQL 8.0 不支持交运算 INTERSECT

集合的交, 就是两个集合的公共部分, 由于集合元素的互异性, 集合的交只需通过文氏图就可以很直观地看到它的意义.

虽然集合的交运算在 SQL 标准中已经出现多年了, 然而很遗憾的是, 截止到 MySQL 8.0 版本, MySQL 仍然不支持 INTERSECT 操作.

[源码地址 sql](#)

```
Line 1 SELECT product_id, product_name
  FROM product
-
- INTERSECT
5 SELECT product_id, product_name
  FROM product2
```

> 错误代码: 1064

> You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'SELECT product_id, product_name

> FROM product2

【扩展阅读】bag 的交运算

对于两个 bag, 他们的交运算会按照:

1. 该元素是否同时属于两个 bag,
2. 该元素在两个 bag 中的最小出现次数

这两个方面来进行计算. 因此对于 $A = \{1, 1, 1, 2, 3, 5, 7\}$, $B = \{1, 1, 2, 2, 4, 6, 8\}$ 两个 bag, 它们的交运算结果就等于 $\{1, 1, 2\}$.

5.1.4 差集, 补集与表的减法

求集合差集的减法运算和实数的减法运算有些不同, 当使用一个集合 A 减去另一个集合 B 的时候, 对于只存在于集合 B 而不存在于集合 A 的元素, 采取直接忽略的策略, 因此集合 A 和 B 做减法只是将集合 A 中也同时属于集合 B 的元素减掉。

图 7-3 使用 EXCEPT 对记录进行减法运算的图示

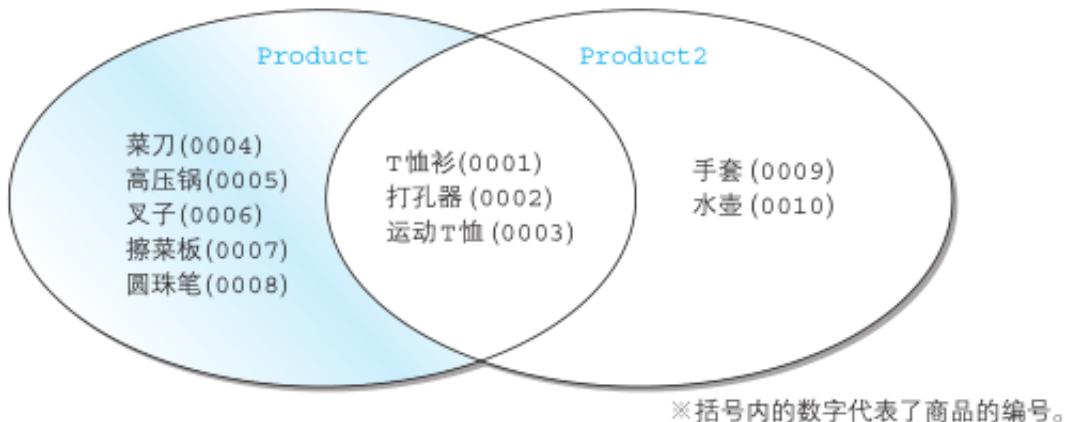


图 5.9: 使用 EXCEPT 对记录进行减法运算的图示

- MySQL 8.0 还不支持 EXCEPT 运算

MySQL 8.0 还不支持表的减法运算符 EXCEPT. 不过, 借助第六章学过的 NOT IN 谓词, 我们同样可以实现表的减法.

练习题

找出只存在于 product 表但不存在于 product2 表的商品.

[源码地址 sql](#)

```
Line 1 -- 使用 IN 子句的实现方法
- SELECT *
-   FROM product
- WHERE product_id NOT IN (SELECT product_id
-                            FROM product2)
5
```

- EXCEPT 与 NOT 谓词

通过上述练习题的 MySQL 解法, 我们发现, 使用 NOT IN 谓词, 基本上可以实现和 SQL 标准语法中的 EXCEPT 运算相同的效果.

练习题

使用 NOT 谓词进行集合的减法运算, 求出 product 表中, 售价高于 2000, 但利润低于 30

product_id	product_name	product_type	sale_price	purchase_price	regist_date
0003	运动T恤	衣服	4000	2800	(NULL)
0005	高压锅	厨房用具	6800	5000	2009-01-15

图 5.10: 查询结果

参考答案:

[源码地址 sql](#)

```
Line 1 SELECT *
-   FROM product
- WHERE sale_price > 2000
- AND product_id NOT IN (SELECT product_id
5
-                           FROM product
- WHERE sale_price<1.3*purchase_price)
```

- EXCEPT ALL 与 bag 的差

类似于 UNION ALL, EXCEPT ALL 也是按出现次数进行减法, 也是使用 bag 模型进行运算.

对于两个 bag, 他们的差运算会按照:

- 该元素是否属于作为被减数的 bag;
- 该元素在两个 bag 中的出现次数。

这两个方面来进行计算. 只有属于被减数的 bag 的元素才参与 EXCEPT ALL 运算, 并且差 bag 中的次数, 等于该元素在两个 bag 的出现次数之差 (差为零或负数则不出现). 因此对于 A = 1,1,1,2,3,5,7, B = 1,1,2,2,4,6,8 两个 bag, 它们的差就等于 1,3,5,7.

- INTERSECT 与 AND 谓词

对于同一个表的两个查询结果而言, 他们的交 INTERSECT 实际上可以等价地将两个查询的检索条件用 AND 谓词连接来实现.

练习题

使用 AND 谓词查找 product 表中利润率高于 50%, 并且售价低于 1500 的商品, 查询结果如下所示

product_id	product_name	product_type	sale_price	purchase_price	regist_date
0001	T恤	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11

图 5.11: 查询结果

参考答案

[源码地址 sql](#)

```
Line 1 SELECT *
-   FROM product
- WHERE sale_price > 1.5 * purchase_price
- AND sale_price < 1500
```

5.1.5 对称差

两个集合 A,B 的对称差是指那些仅属于 A 或仅属于 B 的元素构成的集合. 对称差也是个非常基础的运算, 例如, 两个集合的交就可以看作是两个集合的并去掉两个集合的对称差. 上述方法在其他数据

库里也可以用来简单地实现表或查询结果的对称差运算：首先使用 UNION 求两个表的并集，然后使用 INTERSECT 求两个表的交集，然后用并集减去交集，就得到了对称差。

但由于在 MySQL 8.0 里，由于两个表或查询结果的并不能直接求出来，因此并不适合使用上述思路来求对称差。好在还有差集运算可以使用。从直观上就能看出来，两个集合的对称差等于 A-B 并上 B-A，因此实践中可以用这个思路来求对称差。

练习题

使用 product 表和 product2 表的对称差来查询哪些商品只在其中一张表，结果类似于：

product_id	product_name	product_type	sale_price	purchase_price	regist_date
0004	菜刀	厨房用具	3000	2800	2009-09-20
0005	高压锅	厨房用具	6800	5000	2009-01-15
0006	叉子	厨房用具	500	(NULL)	2009-09-20
0007	擦菜板	厨房用具	880	790	2008-04-28
0008	圆珠笔	办公用品	100	(NULL)	2009-11-11
0009	手套	衣服	800	500	(NULL)
0010	水壶	厨房用具	2000	1700	2009-09-20

图 5.12: 查询结果

提示：使用 NOT IN 实现两个表的差集。

参考答案

源码地址 [sql](#)

```
Line 1 -- 使用 NOT IN 实现两个表的差集
- SELECT *
-   FROM product
- WHERE product_id NOT IN (SELECT product_id FROM product2)
5 UNION
- SELECT *
-   FROM product2
- WHERE product_id NOT IN (SELECT product_id FROM product)
```

- 借助并集和差集迂回实现交集运算 INTERSECT

通过观察集合运算的文氏图，我们发现，两个集合的交可以看作是两个集合的并去掉两个集合的对称差。

5.2 连结 (JOIN)

前一节我们学习了 UNION 和 INTERSECT 等集合运算，这些集合运算的特征就是以行方向为单位进行操作。通俗地说，就是进行这些集合运算时，会导致记录行数的增减。使用 UNION 会增加记录行数，而使用 INTERSECT 或者 EXCEPT 会减少记录行数。

但这些运算不能改变列的变化，虽然使用函数或者 CASE 表达式等列运算，可以增加列的数量，但仍然只能从一张表中提供的基础信息列中获得一些“引申列”，本质上并不能提供更多的信息。如果想要从多个表获取信息，例如，如果我们想要找出某个商店里的衣服类商品的名称，数量及价格等信息，则必须分别从 shopproduct 表和 product 表获取信息。

图 7-5 联结的图示



图 5.13: 联结的图示

> 注:

> 截至目前, 本书中出现的示例 (除了关联子查询) 基本上都是从一张表中选取数据, 但实际上, 期望得到的数据往往会分散在不同的表之中, 这时候就需要使用连结了。

> 之前在学习关联子查询时我们发现, 使用关联子查询也可以从其他表获取信息, 但 ** 连结 ** 更适合从多张表获取信息。

连结 (JOIN) 就是使用某种关联条件 (一般是使用相等判断谓词"`=`"), 将其他表中的列添加过来, 进行“添加列”的集合运算。可以说, 连结是 SQL 查询的核心操作, 掌握了连结, 能够从两张甚至多张表中获取列, 能够将过去使用关联子查询等过于复杂的查询简化为更加易读的形式, 以及进行一些更加复杂的查询.

SQL 中的连结有多种分类方法, 我们这里使用最基础的内连结和外连结的分类方法来分别进行讲解.

5.2.1 内连结 (INNER JOIN)

内连结的语法格式是:

源码地址 [sql](#)

```
Line 1 -- 内连结
- FROM <tb_1> INNER JOIN <tb_2> ON <condition(s)>
```

其中 `INNER` 关键词表示使用了内连结, 至于内连结的涵义, 目前暂时可以不必细究. 例如, 还是刚才那个问题:

找出某个商店里的衣服类商品的名称, 数量及价格等信息.

我们进一步把这个问题明确化:

找出东京商店里的衣服类商品的商品名称, 商品价格, 商品种类, 商品数量信息.

- 使用内连结从两个表获取信息

我们先来分别观察所涉及的表, `product` 表保存了商品编号, 商品名称, 商品种类等信息, 这个表可以提供关于衣服种类的衣服的详细信息, 但是不能提供商店信息。

表 7-1 Product(商品)表

product_id (商品编号)	product_name (商品名称)	product_type (商品种类)	sale_price (销售单价)	purchase_price (进货单价)	regist_date (登记日期)
0001	T恤衫	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11
0003	运动T恤	衣服	4000	2800	
0004	菜刀	厨房用具	3000	2800	2009-09-20
0005	高压锅	厨房用具	6800	5000	2009-01-15
0006	叉子	厨房用具	500		2009-09-20
0007	擦菜板	厨房用具	880	790	2008-04-28
0008	圆珠笔	办公用品	100		2009-11-11

图 5.14: Product (商品) 表

我们接下来观察 shopproduct 表, 这个表里有商店编号名称, 商店的商品编号及数量. 但要想获取商品的种类及名称售价等信息, 则必须借助于 product 表.

表 7-2 ShopProduct(商店商品)表

shop_id (商店编号)	shop_name (商店名称)	product_id (商品编号)	quantity (数量)
000A	东京	0001	30
000A	东京	0002	50
000A	东京	0003	15
000B	名古屋	0002	30
000B	名古屋	0003	120
000B	名古屋	0004	20
000B	名古屋	0006	10
000B	名古屋	0007	40
000C	大阪	0003	20
000C	大阪	0004	50

图 5.15: ShopProduct (商店商品) 表

所以问题的关键是, 找出一个类似于“轴”或者“桥梁”的公共列, 将两张表用这个列连结起来. 这就是连结运算所要作的事情.

我们来对比一下上述两张表, 可以发现, 商品编号列是一个公共列, 因此很自然的事情就是用这个商品编号列来作为连接的“桥梁”, 将 product 和 shopproduct 这两张表连接起来。

表 7-3 两张表及其包含的列

	Product	ShopProduct
商品编号	○	○
商品名称	○	
商品种类	○	
销售单价	○	
进货单价	○	
登记日期	○	
商店编号		○
商店名称		○
数量		○

图 5.16: 两张表及其包含的列

> 注:

> 如果你使用过 excel 的 vlookup 函数, 你会发现这个函数正好也能够实现这个功能. 实际上, 在思路上, 关联子查询更像是 vlookup 函数: 以表 A 为主表, 然后根据表 A 的关联列的每一行的取值, 逐个到表 B 中的关联列中去查找取值相等的行。

> 当数据量较小时, 这种方式并不会有什么性能问题, 但数据量较大时, 这种方式将会导致较大的计算开销: 对于外部查询返回的每一行数据, 都会向内部的子查询传递一个关联列的值, 然后内部子查询根据传入的值执行一次查询然后返回它的查询结果. 这就使得, 例如外部主查询的返回结果有一万行, 那么子查询就会执行一万次, 这将会带来非常恐怖的时间消耗.

我们把上述问题进行分解:

首先, 找出每个商店的商店编号, 商店名称, 商品编号, 商品名称, 商品类别, 商品售价, 商品数量信息.

按照内连结的语法, 在 FROM 子句中使用 INNER JOIN 将两张表连接起来, 并为 ON 子句指定连结条件为 shopproduct.product_id=product.product_id, 就得到了如下的查询语句:

源码地址 [sql](#)

```
Line 1  SELECT SP.shop_id
      ,SP.shop_name
      ,SP.product_id
      ,P.product_name
      ,P.product_type
      ,P.product_type
```

```

    ,P.sale_price
    ,SP.quantity
FROM shopproduct AS SP
INNER JOIN product AS P
ON SP.product_id = P.product_id;
10

```

在上述查询中, 我们分别为两张表指定了简单的别名, 这种操作在使用连结时是非常常见的, 通过别名会让我们在编写查询时少打很多字, 并且更重要的是, 会让查询语句看起来更加简洁. 上述查询将会得到如下的结果:

shop_id	shop_name	product_id	product_name	product_type	sale_price	quantity
000A	东京	0001	T恤	衣服	1000	30
000A	东京	0002	打孔器	办公用品	500	50
000A	东京	0003	运动T恤	衣服	4000	15
000B	名古屋	0002	打孔器	办公用品	500	30
000B	名古屋	0003	运动T恤	衣服	4000	120
000B	名古屋	0004	菜刀	厨房用具	3000	20
000B	名古屋	0006	叉子	厨房用具	500	10
000B	名古屋	0007	擦菜板	厨房用具	880	40
000C	大阪	0003	运动T恤	衣服	4000	20
000C	大阪	0004	菜刀	厨房用具	3000	50
000C	大阪	0006	叉子	厨房用具	500	90
000C	大阪	0007	擦菜板	厨房用具	880	70
000D	福冈	0001	T恤	衣服	1000	100

图 5.17: 查询结果

观察查询结果, 我们看到, 这个结果里的列已经包含了所有我们需要的信息.

关于内连结, 需要注意以下三点:

要点一: 进行连结时需要在 FROM 子句中使用多张表

之前的 FROM 子句中只有一张表, 而这次我们同时使用了 shopproduct 和 product 两张表, 使用关键字 INNER JOIN 就可以将两张表连结在一起了:

源码地址 [sql](#)

Line 1 FROM shopproduct AS SP INNER JOIN product AS P

要点二: 必须使用 ON 子句来指定连结条件

在进行内连结时 ON 子句是必不可少的 (大家可以试试去掉上述查询的 ON 子句后会有什么结果).

ON 子句是专门用来指定连结条件的, 我们在上述查询的 ON 之后指定两张表连结所使用的列以及比较条件, 基本上, 它能起到与 WHERE 相同的筛选作用, 我们会在本章的结尾部分进一步探讨这个话题.

要点三: SELECT 子句中的列最好按照表名. 列名的格式来使用

当两张表的列除了用于关联的列之外, 没有名称相同的列的时候, 也可以不写表名, 但表名使得我们能够在今后的任何时间阅读查询代码的时候, 都能马上看出每一列来自于哪张表, 能够节省我们很多时间.

但是, 如果两张表有其他名称相同的列, 则必须使用上述格式来选择列名, 否则查询语句会报错.

我们回到上述查询所回答的问题. 通过观察上述查询的结果, 我们发现, 这个结果离我们的目标: 找出东京商店的衣服类商品的基础信息已经很接近了. 接下来, 我们只需要把这个查询结果作为一张表, 给它增加一个 WHERE 子句来指定筛选条件.

- 结合 WHERE 子句使用内连结

如果需要在使用内连结的时候同时使用 WHERE 子句对检索结果进行筛选, 则需要把 WHERE 子句写在 ON 子句的后边.

例如, 对于上述查询问题, 我们可以在前一步查询的基础上, 增加 WHERE 条件.

增加 WHERE 子句的方式有好几种, 我们先从最简单的说起.

第一种增加 WHERE 子句的方式, 就是把上述查询作为子查询, 用括号封装起来, 然后在外层查询增加筛选条件.

[源码地址 sql](#)

```
Line 1  SELECT *
-      FROM ( -- 第一步查询的结果
-          SELECT SP.shop_id
-                  ,SP.shop_name
-                  ,SP.product_id
-                  ,P.product_name
-                  ,P.product_type
-                  ,P.sale_price
-                  ,SP.quantity
10     FROM shopproduct AS SP
-      INNER JOIN product AS P
-          ON SP.product_id = P.product_id) AS STEP1
-      WHERE shop_name = '东京'
-      AND product_type = '衣服' ;
```

还记得我们学习子查询时的认识吗? 子查询的结果其实也是一张表, 只不过是一张虚拟的表, 它并不真实存在于数据库中, 只是数据库中其他表经过筛选, 聚合等查询操作后得到的一个"视图". 这种写法能很清晰地分辨出每一个操作步骤, 在我们还不十分熟悉 SQL 查询每一个子句的执行顺序的时候可以帮到我们.

但实际上, 如果我们熟知 WHERE 子句将在 FROM 子句之后执行, 也就是说, 在做完 INNER JOIN ... ON 得到一个新表后, 才会执行 WHERE 子句, 那么就得到标准的写法:

[源码地址 sql](#)

```
Line 1  SELECT  SP.shop_id
-          ,SP.shop_name
-          ,SP.product_id
-          ,P.product_name
-          ,P.product_type
-          ,P.sale_price
-          ,SP.quantity
-      FROM shopproduct AS SP
-      INNER JOIN product AS P
-          ON SP.product_id = P.product_id
-      WHERE SP.shop_name = '东京'
-      AND P.product_type = '衣服' ;
```

我们首先给出上述查询的执行顺序: FROM 子句->WHERE 子句->SELECT 子句
也就是说, 两张表是先按照连结列进行了连结, 得到了一张新表, 然后 WHERE 子句对这张新表的行
按照两个条件进行了筛选, 最后, SELECT 子句选出了那些我们需要的列.

此外, 一种不是很常见的做法是, 还可以将 WHERE 子句中的条件直接添加在 ON 子句中, 这时候
ON 子句后最好用括号将连结条件和筛选条件括起来.

[源码地址 sql](#)

```
Line 1  SELECT SP.shop_id
      - ,SP.shop_name
      - ,SP.product_id
      - ,P.product_name
      5 ,P.product_type
      - ,P.sale_price
      - ,SP.quantity
      - FROMshopproduct AS SP
      - INNER JOINproduct AS P
      10    ON (SP.product_id = P.product_id
      -     AND SP.shop_name = '东京'
      -     AND P.product_type = '衣服') ;
```

但上述这种把筛选条件和连结条件都放在 ON 子句的写法, 不太容易阅读, 不建议大家使用. 另外,
先连结再筛选的标准写法的执行顺序是, 两张完整的表做了连结之后再做筛选, 如果要连结多张表, 或者需要
做的筛选比较复杂时, 在写 SQL 查询时会感觉比较吃力. 在结合 WHERE 子句使用内连结的时候, 我们
也可以更改任务顺序, 并采用任务分解的方法, 先分别在两个表使用 WHERE 进行筛选, 然后把上述两
个子查询连结起来.

[源码地址 sql](#)

```
Line 1  SELECT SP.shop_id
      - ,SP.shop_name
      - ,SP.product_id
      - ,P.product_name
      5 ,P.product_type
      - ,P.sale_price
      - ,SP.quantity
      - FROM ( -- 子查询 1:从shopproduct 表筛选出东京商店的信息
      -         SELECT *
      -             FROMshopproduct
      -             WHERE shop_name = '东京' ) AS SP
      - INNER JOIN -- 子查询 2:从 product 表筛选出衣服类商品的信息
      -         (SELECT *
      -             FROMproduct
      -             WHERE product_type = '衣服') AS P
      -             ON SP.product_id = P.product_id;
```

先分别在两张表里做筛选, 把复杂的筛选条件按表分拆, 然后把筛选结果 (作为表) 连接起来, 避免了
写复杂的筛选条件, 因此这种看似复杂的写法, 实际上整体的逻辑反而非常清晰. 在写查询的过程中, 首先

要按照最便于自己理解的方式来写, 先把问题解决了, 再思考优化的问题. **练习题**

找出每个商店里的衣服类商品的名称及价格等信息. 希望得到如下结果:

shop_id	shop_name	product_id	product_name	product_type	purchase_price
000A	东京	0001	T恤	衣服	500
000A	东京	0003	运动T恤	衣服	2800
000B	名古屋	0003	运动T恤	衣服	2800
000C	大阪	0003	运动T恤	衣服	2800
000D	福冈	0001	T恤	衣服	500

图 5.18: 查询结果

源码地址 [sql](#)

```

Line 1  -- 参考答案 1--不使用子查询
-   SELECT SP.shop_id,SP.shop_name,SP.product_id
-     ,P.product_name, P.product_type, P.purchase_price
-   FROM shopproduct AS SP
5  INNER JOIN product AS P
-     ON SP.product_id = P.product_id
- WHERE P.product_type = '衣服';

-
- -- 参考答案 2--使用子查询
10 SELECT SP.shop_id, SP.shop_name, SP.product_id
-     ,P.product_name, P.product_type, P.purchase_price
-   FROM shopproduct AS SP
- INNER JOIN --从 product 表找出衣服类商品的信息
-   (SELECT product_id, product_name, product_type, purchase_price
15   FROM product
-     WHERE product_type = '衣服')AS P
-   ON SP.product_id = P.product_id;

```

上述第二种写法虽然包含了子查询, 并且代码行数更多, 但由于每一层的目的很明确, 更适于阅读, 并且在外连结的情形下, 还能避免错误使用 WHERE 子句导致外连结失效的问题, 相关示例见后文中的"结合 WHERE 子句使用外连结" 章节。

练习题

分别使用连结两个子查询和不使用子查询的方式, 找出东京商店里, 售价低于 2000 的商品信息, 希望得到如下结果.

shop_id	shop_name	product_id	quantity	product_id(1)	product_name	product_type	sale_price
000A	东京	0001	30	0001	T恤	衣服	1000
000A	东京	0002	50	0002	打孔器	办公用品	500

图 5.19: 查询结果

源码地址 [sql](#)

```

Line 1  -- 参考答案

```

```

- -- 不使用子查询
- SELECT SP.* , P.*
-   FROM shopproduct AS SP
- INNER JOIN product AS P
-     ON SP.product_id = P.product_id
- WHERE shop_id = '000A'
-   AND sale_price < 2000;

```

- 结合 GROUP BY 子句使用内连结

结合 GROUP BY 子句使用内连结, 需要根据分组列位于哪个表区别对待.

最简单的情形, 是在内连结之前就使用 GROUP BY 子句.

但是如果分组列和被聚合的列不在同一张表, 且二者都未被用于连结两张表, 则只能先连结, 再聚合.

练习题

每个商店中, 售价最高的商品的售价分别是多少?

[源码地址 sql](#)

```

Line 1  -- 参考答案
- SELECT SP.shop_id
-       ,SP.shop_name
-       ,MAX(P.sale_price) AS max_price
5   FROM shopproduct AS SP
- INNER JOIN product AS P
-     ON SP.product_id = P.product_id
- GROUP BY SP.shop_id,SP.shop_name

```

思考题

上述查询得到了每个商品售价最高的商品, 但并不知道售价最高的商品是哪一个. 如何获取每个商店里售价最高的商品的名称和售价?

> 注:

这道题的一个简易的方式是使用下一章的窗口函数。当然, 也可以使用其他我们已经学过的知识来实现, 例如, 在找出每个商店售价最高商品的价格后, 使用这个价格再与 product 列进行连结, 但这种做法在价格不唯一时会出现问题。

- 自连结 (SELF JOIN)

之前的内连结, 连结的都是不一样的两个表. 但实际上一张表也可以与自身作连结, 这种连接称之为自连结. 需要注意, 自连结并不是区分于内连结和外连结的第三种连结, 自连结可以是外连结也可以是内连结, 它是不同于内连结外连结的另一个连结的分类方法.

- 内连结与关联子查询

回忆第五章第三节关联子查询中的问题: 找出每个商品种类当中售价高于该类商品的平均售价的商品. 当时我们是使用关联子查询来实现的.

[源码地址 sql](#)

```
Line 1  SELECT product_type, product_name, sale_price
```

```

-   FROM product AS P1
- WHERE sale_price > (SELECT AVG(sale_price)
-                         FROM product AS P2
-                         WHERE P1.product_type = P2.product_type
-                         GROUP BY product_type);

```

使用内连结同样可以解决这个问题：首先，使用 GROUP BY 按商品类别分类计算每类商品的平均价格。

[源码地址 sql](#)

```

Line 1  SELECT product_type
        ,AVG(sale_price) AS avg_price
-   FROM product
- GROUP BY product_type;

```

接下来，将上述查询与表 product 按照 product_type (商品种类) 进行内连结。

[源码地址 sql](#)

```

Line 1  SELECT P1.product_id
        ,P1.product_name
        ,P1.product_type
        ,P1.sale_price
        ,P2.avg_price
-   FROM product AS P1
- INNER JOIN
-     (SELECT product_type,AVG(sale_price) AS avg_price
-      FROM product
-      GROUP BY product_type) AS P2
-     ON P1.product_type = P2.product_type;

```

最后，增加 WHERE 子句，找出那些售价高于该类商品平均价格的商品。完整的代码如下：

[源码地址 sql](#)

```

Line 1  SELECT P1.product_id
        ,P1.product_name
        ,P1.product_type
        ,P1.sale_price
        ,P2.avg_price
-   FROM product AS P1
- INNER JOIN
-     (SELECT product_type,AVG(sale_price) AS avg_price
-      FROM product
-      GROUP BY product_type) AS P2
-     ON P1.product_type = P2.product_type
- WHERE P1.sale_price > P2.avg_price;

```

仅仅从代码量上来看，上述方法似乎比关联子查询更加复杂，但这并不意味着这些代码更难理解。通过上述分析，很容易发现上述代码的逻辑实际上更符合我们的思路，因此尽管看起来复杂，但思路实际上更加清晰。作为对比，试分析如下代码：

[源码地址 sql](#)

```

Line 1  SELECT P1.product_id
      ,P1.product_name
      ,P1.product_type
      ,P1.sale_price
      ,AVG(P2.sale_price) AS avg_price
  FROM product AS P1
 INNER JOIN product AS P2
   ON P1.product_type=P2.product_type
 WHERE P1.sale_price > P2.sale_price
GROUP BY P1.product_id,P1.product_name,P1.product_type,P1.sale_price,P2.
      product_type

```

虽然去掉了子查询, 查询语句的层次更少, 而且代码行数似乎更少, 但实际上这个方法可能更加难以写出来. 在实践中, 一定要按照易于让自己理解的思路去分层次写代码, 而不要花费很长世间写出一个效率可能更高但自己和他人理解起来难度更高的代码。

- 自然连结 (NATURAL JOIN)

自然连结并不是区别于内连结和外连结的第三种连结, 它其实是内连结的一种特例--当两个表进行自然连结时, 会按照两个表中都包含的列名来进行等值内连结, 此时无需使用 ON 来指定连接条件.

[源码地址 sql](#)

```

Line 1  SELECT * FROM shopproduct NATURAL JOIN product

```

上述查询得到的结果, 会把两个表的公共列 (这里是 product_id, 可以有多个公共列) 放在第一列, 然后按照两个表的顺序和表中列的顺序, 将两个表中的其他列都罗列出来。

product_id	shop_id	shop_name	quantity	product_name	product_type	sale_price	purchase_price	regist_date
0001	000A	东京	30	T恤	衣服	1000	500	2009-09-20
0002	000A	东京	50	打孔器	办公用品	500	320	2009-09-11
0003	000A	东京	15	运动T恤	衣服	4000	2800	(NULL)
0002	000B	名古屋	30	打孔器	办公用品	500	320	2009-09-11
0003	000B	名古屋	120	运动T恤	衣服	4000	2800	(NULL)
0004	000B	名古屋	20	菜刀	厨房用具	3000	2800	2009-09-20
0006	000B	名古屋	10	叉子	厨房用具	500	(NULL)	2009-09-20
0007	000B	名古屋	40	擦菜板	厨房用具	880	790	2008-04-28
0003	000C	大阪	20	运动T恤	衣服	4000	2800	(NULL)
0004	000C	大阪	50	菜刀	厨房用具	3000	2800	2009-09-20
0006	000C	大阪	90	叉子	厨房用具	500	(NULL)	2009-09-20
0007	000C	大阪	70	擦菜板	厨房用具	880	790	2008-04-28
0001	000D	福冈	100	T恤	衣服	1000	500	2009-09-20

图 5.20: 查询结果

练习题

试写出与上述自然连结等价的内连结.

[源码地址 sql](#)

```

Line 1  -- 参考答案
      - SELECT SP.product_id,SP.shop_id,SP.shop_name,SP.quantity

```

```

    ,P.product_name,P.product_type,P.sale_price
    ,P.purchase_price,P.regist_date
5   FROM shopproduct AS SP
- INNER JOIN product AS P
-     ON SP.product_id = P.product_id

```

使用自然连结还可以求出两张表或子查询的公共部分, 例如教材中 7-1 选取表中公共部分-INTERSECT 一节中的问题: 求表 product 和表 product2 中的公共部分, 也可以用自然连结来实现:

[源码地址 sql](#)

Line 1 `SELECT * FROM product NATURAL JOIN product2`

product_id	product_name	product_type	sale_price	purchase_price	regist_date
0001	T恤	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11

图 5.21: 查询结果

这个结果和书上给的结果并不一致, 少了运动 T 恤, 这是由于运动 T 恤的 regist_date 字段为空, 在进行自然连结时, 来自于 product 和 product2 的运动 T 恤这一行数据在进行比较时, 实际上是在逐字段进行等值连结, 两个缺失值用等号进行比较, 结果不为真. 而连结只会返回对连结条件返回为真的那些行.

如果我们将查询语句进行修改:

[源码地址 sql](#)

Line 1 `SELECT *`
- `FROM (SELECT product_id, product_name`
- `FROM product) AS A`
- `NATURAL JOIN`
5 `(SELECT product_id, product_name`
- `FROM product2) AS B;`

那就可以得到正确的结果了:

product_id	product_name
0001	T恤
0002	打孔器
0003	运动T恤

图 5.22: 查询结果

- 使用连结求交集

我们在上一节表的加减法里知道, MySQL 8.0 里没有交集运算, 我们当时是通过并集和差集来实现求交集的. 现在学了连结, 让我们试试使用连结来实现求交集的运算.

练习题: 使用内连结求 product 表和 product2 表的交集.

[源码地址 sql](#)

```

Line 1  SELECT P1.*
-      FROM product AS P1
-      INNER JOIN product2 AS P2
-          ON (P1.product_id = P2.product_id)
-          AND P1.product_name = P2.product_name
-          AND P1.product_type = P2.product_type
-          AND P1.sale_price = P2.sale_price
-          AND P1.regist_date = P2.regist_date)

```

得到如下结果：

product_id	product_name	product_type	sale_price	purchase_price	regist_date
0001	T恤	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11

图 5.23: 查询结果

注意上述结果和 P230 的结果并不一致--少了 `product_id='0001'` 这一行, 观察源表数据可发现, 少的这行数据的 `regist_date` 为缺失值, 回忆第六章讲到的 `IS NULL` 谓词, 我们得知, 这是由于缺失值是不能用等号进行比较导致的.

如果我们仅仅用 `product_id` 来进行连结:

源码地址 [sql](#)

```

Line 1  SELECT P1.*
-      FROM product AS P1
-      INNER JOIN product2 AS P2
-          ON P1.product_id = P2.product_id

```

查询结果:

product_id	product_name	product_type	sale_price	purchase_price	regist_date
0001	T恤	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11
0003	运动T恤	衣服	4000	2800	(Null)

图 5.24: 查询结果

这次就一致了。

5.2.2 外连结 (OUTER JOIN)

内连结会丢弃两张表中不满足 ON 条件的行, 和内连结相对的就是外连结. 外连结会根据外连结的种类有选择地保留无法匹配到的行.

按照保留的行位于哪张表, 外连结有三种形式: 左连结, 右连结和全外连结.

左连结会保存左表中无法按照 ON 子句匹配到的行, 此时对应右表的行均为缺失值; 右连结则会保存右表中无法按照 ON 子句匹配到的行, 此时对应左表的行均为缺失值; 而全外连结则会同时保存两个表中无法按照 ON 子句匹配到的行, 相应的另一张表中的行用缺失值填充.

三种外连结的对应语法分别为:

[源码地址 sql](#)

```
Line 1 -- 左连结
- FROM <tb_1> LEFT OUTER JOIN <tb_2> ON <condition(s)>
- -- 右连结
- FROM <tb_1> RIGHT OUTER JOIN <tb_2> ON <condition(s)>
5 -- 全外连结
- FROM <tb_1> FULL OUTER JOIN <tb_2> ON <condition(s)>
```

- 左连结与右链接

由于连结时可以交换左表和右表的位置, 因此左连结和右连结并没有本质区别. 接下来我们先以左连结为例进行学习. 所有的内容在调换两个表的前后位置, 并将左连结改为右连结之后, 都能得到相同的结果. 稍后再介绍全外连结的概念.

- 使用左连结从两个表获取信息

如果你仔细观察过将 shopproduct 和 product 进行内连结前后的结果的话, 你就会发现, product 表中有两种商品并未在内连结的结果里, 就是说, 这两种商品并未在任何商店有售 (这通常意味着比较重要的业务信息, 例如, 这两种商品在所有商店都处于缺货状态, 需要及时补货). 现在, 让我们先把之前内连结的 SELECT 语句转换为左连结试试看吧.

练习题: 统计每种商品分别在哪些商店有售, 需要包括那些在每个商店都没货的商品.

使用左连结的代码如下 (注意区别于书上的右连结):

[源码地址 sql](#)

```
Line 1 SELECT SP.shop_id
- ,SP.shop_name
- ,SP.product_id
- ,P.product_name
- ,P.sale_price
5 FROM product AS P
- LEFT OUTER JOIN shopproduct AS SP
- ON SP.product_id = P.product_id;
```

上述查询得到的检索结果如下 (由于并未使用 ORDER BY 子句指定顺序, 你执行上述代码得到的结果可能顺序与下图不同):

shop_id	shop_name	product_id	product_name	sale_price
000A	东京	0001	T恤	1000
000D	福冈	0001	T恤	1000
000A	东京	0002	打孔器	500
000B	名古屋	0002	打孔器	500
000A	东京	0003	运动T恤	4000
000B	名古屋	0003	运动T恤	4000
000C	大阪	0003	运动T恤	4000
000B	名古屋	0004	菜刀	3000
000C	大阪	0004	菜刀	3000
(NULL)	(NULL)	(NULL)	高压锅	6800
000B	名古屋	0006	叉子	500
000C	大阪	0006	叉子	500
000B	名古屋	0007	擦菜板	880
000C	大阪	0007	擦菜板	880
(NULL)	(NULL)	(NULL)	圆珠笔	100

图 5.25: 查询结果

我们观察上述结果可以发现, 有两种商品: 高压锅和圆珠笔, 在所有商店都没有销售. 由于我们在 SELECT 子句选择列的显示顺序以及未对结果进行排序的原因, 这个事实需要你仔细地进行观察.

外连结要点 1: 选取出单张表中全部的信息

与内连结的结果相比, 不同点显而易见, 那就是结果的行数不一样. 内连结的结果中有 13 条记录, 而外连结的结果中有 15 条记录, 增加的 2 条记录到底是什么呢? 这正是外连结的关键点. 多出的 2 条记录是高压锅和圆珠笔, 这 2 条记录在 shopproduct 表中并不存在, 也就是说, 这 2 种商品在任何商店中都没有销售. 由于内连结只能选取出同时存在于两张表中的数据, 因此只在 product 表中存在的 2 种商品并没有出现在结果之中. 相反, 对于外连结来说, 只要数据存在于某一张表当中, 就能够读取出来. 在实际的业务中, 例如想要生成固定行数的单据时, 就需要使用外连结. 如果使用内连结的话, 根据 SELECT 语句执行时商店库存状况的不同, 结果的行数也会发生改变, 生成的单据的版式也会受到影响, 而使用外连结能够得到固定行数的结果. 虽说如此, 那些表中不存在的信息我们还是无法得到, 结果中高压锅和圆珠笔的商店编号和商店名称都是 NULL (具体信息大家都不知道, 真是无可奈何). 外连结名称的由来也跟 NULL 有关, 即“结果中包含原表中不存在 (在原表之外) 的信息”. 相反, 只包含表内信息的连结也就被称为内连结了.

外连结要点 2: 使用 LEFT、RIGHT 来指定主表

外连结还有一点非常重要, 那就是要把哪张表作为主表. 最终的结果中会包含主表内所有的数据. 指定主表的关键字是 LEFT 和 RIGHT. 顾名思义, 使用 LEFT 时 FROM 子句中写在左侧的表是主表, 使用 RIGHT 时右侧的表是主表. 代码清单 7-11 中使用了 RIGHT, 因此, 右侧的表, 也就是 product 表是主表. 我们还可以像代码清单 7-12 这样进行改写, 意思完全相同. 这样你可能会困惑, 到底应该使用 LEFT 还是 RIGHT? 其实它们的功能没有任何区别, 使用哪一个都可以. 通常使用 LEFT 的情况会多一些, 但也并没有非使用这个不可的理由, 使用 RIGHT 也没有问题.

通过交换两个表的顺序, 同时将 LEFT 更换为 RIGHT(如果原先是 RIGHT, 则更换为 LEFT), 两种方式会到完全相同的结果.

- 结合 WHERE 子句使用左连结

上一小节我们学到了外连结的基础用法, 并且在上一节也学习了结合 WHERE 子句使用内连结的方法, 但在结合 WHERE 子句使用外连结时, 由于外连结的结果很可能与内连结的结果不一样, 会包含那些主表中无法匹配到的行, 并用缺失值填写另一表中的列, 由于这些行的存在, 因此在外连结时使用 WHERE 子句, 情况会有些不一样. 我们来看一个例子:

练习题

使用外连结从 shopproduct 表和 product 表中找出那些在某个商店库存少于 50 的商品及对应的商店。希望得到如下结果。

product_id	product_name	sale_price	shop_id	shop_name	quantity
0001	T恤	1000	000A	东京	30
0003	运动T恤	4000	000A	东京	15
0002	打孔器	500	000B	名古屋	30
0004	菜刀	3000	000B	名古屋	20
0006	叉子	500	000B	名古屋	10
0007	擦菜板	880	000B	名古屋	40
0003	运动T恤	4000	000C	大阪	20
0005	高压锅	6800	(Null)	(Null)	(Null)
0008	圆珠笔	100	(Null)	(Null)	(Null)

图 5.26: 查询结果

注意高压锅和圆珠笔两种商品在所有商店都无货，所以也应该包括在内。

按照“结合 WHERE 子句使用内连结”的思路，我们很自然会写出如下代码

源码地址 [sql](#)

```
Line 1  SELECT P.product_id
      ,P.product_name
      ,P.sale_price
      ,SP.shop_id
      ,SP.shop_name
      ,SP.quantity
  - FROM product AS P
  - LEFT OUTER JOIN shopproduct AS SP
  -   ON SP.product_id = P.product_id
10 WHERE quantity < 50
```

然而不幸的是，得到的却是如下的结果：

product_id	product_name	sale_price	shop_id	shop_name	quantity
0001	T恤	1000	000A	东京	30
0003	运动T恤	4000	000A	东京	15
0002	打孔器	500	000B	名古屋	30
0004	菜刀	3000	000B	名古屋	20
0006	叉子	500	000B	名古屋	10
0007	擦菜板	880	000B	名古屋	40
0003	运动T恤	4000	000C	大阪	20

图 5.27: 查询结果

观察发现，返回结果缺少了在所有商店都无货的高压锅和圆珠笔。聪明的你可能很容易想到，在 WHERE 过滤条件中增加 **OR, quantity IS NULL** 的判断条件，便可以得到预期结果。然而在实际环境中，由于数据量大且数据质量并非像我们设想的那样“干净”，我们并不能容易地意识到缺失值等问题数据的存在，因此，还是让我们想一下如何改写我们的查询以使得它能够适应更复杂的真实数据的情形吧。

联系到我们已经掌握了的 SQL 查询的执行顺序 (FROM->WHERE->SELECT), 我们发现, 问题可能出在筛选条件上, 因为在进行完外连结后才会执行 WHERE 子句, 因此那些主表中无法被匹配到的行就被 WHERE 条件筛选掉了。

明白了这一点, 我们就可以试着把 WHERE 子句挪到外连结之前进行: 先写个子查询, 用来从 shop-product 表中筛选 quantity<50 的商品, 然后再把这个子查询和主表连结起来。

我们把上述思路写成 SQL 查询语句:

源码地址 [sql](#)

```
Line 1  SELECT P.product_id
      ,P.product_name
      ,P.sale_price
      ,SP.shop_id
      ,SP.shop_name
      ,SP.quantity
  FROM product AS P
  LEFT OUTER JOIN-- 先筛选quantity<50的商品
  (SELECT *
   FROM shopproduct
   WHERE quantity < 50 ) AS SP
  ON SP.product_id = P.product_id
```

得到的结果如下:

product_id	product_name	sale_price	shop_id	shop_name	quantity
0001	T恤	1000	000A	东京	30
0003	运动T恤	4000	000A	东京	15
0002	打孔器	500	000B	名古屋	30
0004	菜刀	3000	000B	名古屋	20
0006	叉子	500	000B	名古屋	10
0007	擦菜板	880	000B	名古屋	40
0003	运动T恤	4000	000C	大阪	20
0005	高压锅	6800	(Null)	(Null)	(Null)
0008	圆珠笔	100	(Null)	(Null)	(Null)

图 5.28: 查询结果

- 在 MySQL 中实现全外连结

有了对左连结和右连结的了解, 就不难理解全外连结的含义了. 全外连结本质上就是对左表和右表的所有行都予以保留, 能用 ON 关联到的就把左表和右表的内容在一行内显示, 不能被关联到的就分别显示, 然后把多余的列用缺失值填充。

遗憾的是, MySQL8.0 目前还不支持全外连结, 不过我们可以对左连结和右连结的结果进行 UNION 来实现全外连结。

5.2.3 多表连结

通常连结只涉及 2 张表, 但有时也会出现必须同时连结 3 张以上的表的情况, 原则上连结表的数量并没有限制。

- 多表进行内连结

首先创建一个用于三表连结的表 Inventoryproduct. 首先我们创建一张用来管理库存商品的表, 假设商品都保存在 P001 和 P002 这 2 个仓库之中。

inventory_id	product_id	inventory_quantity
P001	0001	0
P001	0002	120
P001	0003	200
P001	0004	3
P001	0005	0
P001	0006	99
P001	0007	999
P001	0008	200
P002	0001	10
P002	0002	25
P002	0003	34
P002	0004	19
P002	0005	99
P002	0006	0
P002	0007	0
P002	0008	18

图 5.29: 查询结果

建表语句如下:

源码地址 [sql](#)

```
Line 1 CREATE TABLE Inventoryproduct
- ( inventory_id      CHAR(4) NOT NULL,
- product_id          CHAR(4) NOT NULL,
- inventory_quantity INTEGER NOT NULL,
5 PRIMARY KEY (inventory_id, product_id));
```

然后插入一些数据:

源码地址 [sql](#)

```
Line 1 --- DML: 插入数据
- START TRANSACTION;
- INSERT INTO Inventoryproduct (inventory_id, product_id, inventory_quantity)
- VALUES ('P001', '0001', 0);
5 INSERT INTO Inventoryproduct (inventory_id, product_id, inventory_quantity)
- VALUES ('P001', '0002', 120);
- INSERT INTO Inventoryproduct (inventory_id, product_id, inventory_quantity)
- VALUES ('P001', '0003', 200);
- INSERT INTO Inventoryproduct (inventory_id, product_id, inventory_quantity)
10 VALUES ('P001', '0004', 3);
- INSERT INTO Inventoryproduct (inventory_id, product_id, inventory_quantity)
```

```

- VALUES ('P001', '0005', 0);
- INSERT INTO Inventoryproduct (inventory_id, product_id, inventory_quantity)
- VALUES ('P001', '0006', 99);
15 INSERT INTO Inventoryproduct (inventory_id, product_id, inventory_quantity)
- VALUES ('P001', '0007', 999);
- INSERT INTO Inventoryproduct (inventory_id, product_id, inventory_quantity)
- VALUES ('P001', '0008', 200);
- INSERT INTO Inventoryproduct (inventory_id, product_id, inventory_quantity)
20 VALUES ('P002', '0001', 10);
- INSERT INTO Inventoryproduct (inventory_id, product_id, inventory_quantity)
- VALUES ('P002', '0002', 25);
- INSERT INTO Inventoryproduct (inventory_id, product_id, inventory_quantity)
- VALUES ('P002', '0003', 34);
25 INSERT INTO Inventoryproduct (inventory_id, product_id, inventory_quantity)
- VALUES ('P002', '0004', 19);
- INSERT INTO Inventoryproduct (inventory_id, product_id, inventory_quantity)
- VALUES ('P002', '0005', 99);
- INSERT INTO Inventoryproduct (inventory_id, product_id, inventory_quantity)
30 VALUES ('P002', '0006', 0);
- INSERT INTO Inventoryproduct (inventory_id, product_id, inventory_quantity)
- VALUES ('P002', '0007', 0 );
- INSERT INTO Inventoryproduct (inventory_id, product_id, inventory_quantity)
- VALUES ('P002', '0008', 18);
35 COMMIT;

```

接下来, 我们根据上表及 shopproduct 表和 product 表, 使用内连接找出每个商店都有那些商品, 每种商品的库存总量分别是多少。

源码地址 [sql](#)

```

Line 1 SELECT SP.shop_id
- ,SP.shop_name
- ,SP.product_id
- ,P.product_name
5 ,P.sale_price
- ,IP.inventory_quantity
- FROMshopproduct AS SP
- INNER JOINproduct AS P
- ON SP.product_id = P.product_id
10 INNER JOIN Inventoryproduct AS IP
- ON SP.product_id = IP.product_id
- WHERE IP.inventory_id = 'P001';

```

得到如下结果

shop_id	shop_name	product_id	product_name	sale_price	inventory_quantity
000A	东京	0001	T恤	1000	0
000A	东京	0002	打孔器	500	120
000A	东京	0003	运动T恤	4000	200
000B	名古屋	0002	打孔器	500	120
000B	名古屋	0003	运动T恤	4000	200
000B	名古屋	0004	菜刀	3000	3
000B	名古屋	0006	叉子	500	99
000B	名古屋	0007	擦菜板	880	999
000C	大阪	0003	运动T恤	4000	200
000C	大阪	0004	菜刀	3000	3
000C	大阪	0006	叉子	500	99
000C	大阪	0007	擦菜板	880	999
000D	福冈	0001	T恤	1000	0

图 5.30: 查询结果

我们可以看到，连结第三张表的时候，也是通过 ON 子句指定连结条件（这里使用最基础的等号将作为连结条件的 product 表和 shopproduct 表中的商品编号 product_id 连结了起来），由于 product 表和 shopproduct 表已经进行了连结，因此就无需再对 product 表和 Inventoryproduct 表进行连结了（虽然也可以进行连结，但结果并不会发生改变，因为本质上并没有增加新的限制条件）。

即使想要把连结的表增加到 4 张、5 张……使用 INNER JOIN 进行添加的方式也是完全相同的。

- 多表进行外连结

正如之前所学发现的，外连结一般能比内连结有更多的行，从而能够比内连结给出更多关于主表的信息，多表连结的时候使用外连结也有同样的作用。

例如，

源码地址 [sql](#)

```

Line 1  SELECT P.product_id
      ,P.product_name
      ,P.sale_price
      ,SP.shop_id
      ,SP.shop_name
      ,IP.inventory_quantity
5
-   FROM product AS P
-   LEFT OUTER JOIN shopproduct AS SP
-   ON SP.product_id = P.product_id
10  LEFT OUTER JOIN Inventoryproduct AS IP
-   ON SP.product_id = IP.product_id

```

查询结果：

product_id	product_name	sale_price	shop_id	shop_name	inventory_quantity
0001	T恤	1000	000A	东京	0
0001	T恤	1000	000D	福冈	0
0002	打孔器	500	000A	东京	120
0002	打孔器	500	000B	名古屋	120
0003	运动T恤	4000	000A	东京	200
0003	运动T恤	4000	000B	名古屋	200
0003	运动T恤	4000	000C	大阪	200
0004	菜刀	3000	000B	名古屋	3
0004	菜刀	3000	000C	大阪	3
0006	叉子	500	000B	名古屋	99
0006	叉子	500	000C	大阪	99
0007	擦菜板	880	000B	名古屋	999
0007	擦菜板	880	000C	大阪	999
0001	T恤	1000	000A	东京	10
0001	T恤	1000	000D	福冈	10
0002	打孔器	500	000A	东京	25
0002	打孔器	500	000B	名古屋	25
0003	运动T恤	4000	000A	东京	34
0003	运动T恤	4000	000B	名古屋	34
0003	运动T恤	4000	000C	大阪	34
0004	菜刀	3000	000B	名古屋	19
0004	菜刀	3000	000C	大阪	19
0006	叉子	500	000B	名古屋	0
0006	叉子	500	000C	大阪	0
0007	擦菜板	880	000B	名古屋	0
0007	擦菜板	880	000C	大阪	0
0005	高压锅	6800	(Null)	(Null)	(Null)
0008	圆珠笔	100	(Null)	(Null)	(Null)

图 5.31: 查询结果

5.2.4 ON 子句进阶--非等值连结

在刚开始介绍连结的时候, 书上提到过, 除了使用相等判断的等值连结, 也可以使用比较运算符来进行连接. 实际上, 包括比较运算符 (<, <=, >, >=, BETWEEN) 和谓词运算 (LIKE, IN, NOT 等等) 在内的所有的逻辑运算都可以放在 ON 子句内作为连结条件.

- 非等值自左连结 (SELF JOIN)

使用非等值自左连结实现排名。

练习题:

希望对 product 表中的商品按照售价赋予排名. 一个从集合论出发, 使用自左连结的思路是, 对每一种商品, 找出售价不低于它的所有商品, 然后对售价不低于它的商品使用 COUNT 函数计数. 例如, 对于价格最高的商品,

源码地址 [sql](#)

```
Line 1  SELECT  product_id
        ,product_name
```

```

        ,sale_price
        ,COUNT(p2_id) AS my_rank
5   FROM (--使用自左连结对每种商品找出价格不低于它的商品
      -     SELECT P1.product_id
      -           ,P1.product_name
      -           ,P1.sale_price
      -           ,P2.product_id AS P2_id
10          ,P2.product_name AS P2_name
      -           ,P2.sale_price AS P2_price
      -     FROM product AS P1
      -     LEFT OUTER JOIN product AS P2
      -       ON P1.sale_price <= P2.sale_price
15    ) AS X
      - GROUP BY product_id, product_name, sale_price
      - ORDER BY my_rank;

```

注 1: COUNT 函数的参数是列名时, 会忽略该列中的缺失值, 参数为 * 时则不忽略缺失值。

注 2: 上述排名方案存在一些问题--如果两个商品的价格相等, 则会导致两个商品的排名错误, 例如, 叉子和打孔器的排名应该都是第六, 但上述查询导致二者排名都是第七. 试修改上述查询使得二者的排名均为第六.

product_id	product_name	sale_price	rank
0005	高压锅	6800	1
0003	运动T恤	4000	2
0004	菜刀	3000	3
0001	T恤	1000	4
0007	擦菜板	880	5
0006	叉子	500	7
0002	打孔器	500	7
0008	圆珠笔	100	8

图 5.32: 查询结果

注 3: 实际上, 进行排名有专门的函数, 这是 MySQL 8.0 新增加的窗口函数中的一种 (窗口函数将在下一章学习), 但在较低版本的 MySQL 中只能使用上述自左连结的思路.

使用非等值自左连结进行累计求和:

练习题

请按照商品的售价从低到高, 对售价进行累计求和 [注: 这个案例缺少实际意义, 并且由于有两种商品价格相同导致了不必要的复杂度, 但示例数据库的表结构比较简单, 暂时未想出有实际意义的例题]

首先, 按照题意, 对每种商品使用自左连结, 找出比该商品售价价格更低或相等的商品

源码地址 [sql](#)

```

Line 1  SELECT  P1.product_id
      -     ,P1.product_name
      -     ,P1.sale_price
      -     ,P2.product_id AS P2_id
      -     ,P2.product_name AS P2_name

```

```

    ,P2.sale_price AS P2_price
  FROM product AS P1
  LEFT OUTER JOIN product AS P2
    ON P1.sale_price >= P2.sale_price
10 ORDER BY P1.sale_price,P1.product_id

```

查看查询结果：

product_id	product_name	sale_price	P2_id	P2_name	P2_price
0008	圆珠笔	100	0008	圆珠笔	100
0002	打孔器	500	0006	叉子	500
0002	打孔器	500	0008	圆珠笔	100
0002	打孔器	500	0002	打孔器	500
0006	叉子	500	0008	圆珠笔	100
0006	叉子	500	0002	打孔器	500
0006	叉子	500	0006	叉子	500
0007	擦菜板	880	0008	圆珠笔	100
0007	擦菜板	880	0006	叉子	500
0007	擦菜板	880	0002	打孔器	500
0007	擦菜板	880	0007	擦菜板	880
0001	T恤	1000	0002	打孔器	500
0001	T恤	1000	0007	擦菜板	880
0001	T恤	1000	0001	T恤	1000
0001	T恤	1000	0006	叉子	500
0001	T恤	1000	0008	圆珠笔	100
0004	菜刀	3000	0004	菜刀	3000
0004	菜刀	3000	0001	T恤	1000
0004	菜刀	3000	0007	擦菜板	880
0004	菜刀	3000	0006	叉子	500
0004	菜刀	3000	0008	圆珠笔	100
0004	菜刀	3000	0002	打孔器	500
0003	运动T恤	4000	0008	圆珠笔	100
0003	运动T恤	4000	0007	擦菜板	880
0003	运动T恤	4000	0001	T恤	1000
0003	运动T恤	4000	0006	叉子	500
0003	运动T恤	4000	0004	菜刀	3000
0003	运动T恤	4000	0003	运动T恤	4000
0003	运动T恤	4000	0002	打孔器	500
0005	高压锅	6800	0006	叉子	500
0005	高压锅	6800	0007	擦菜板	880
0005	高压锅	6800	0005	高压锅	6800
0005	高压锅	6800	0004	菜刀	3000
0005	高压锅	6800	0003	运动T恤	4000
0005	高压锅	6800	0008	圆珠笔	100
0005	高压锅	6800	0002	打孔器	500
0005	高压锅	6800	0001	T恤	1000

图 5.33: 查询结果

看起来似乎没什么问题.

下一步, 按照 P1.product_Id 分组, 对 P2_price 求和:

[源码地址 sql](#)

```
Line 1  SELECT  product_id
      ,product_name
      ,sale_price
      ,SUM(P2_price) AS cum_price
5   FROM (SELECT  P1.product_id
      ,P1.product_name
      ,P1.sale_price
      ,P2.product_id AS P2_id
      ,P2.product_name AS P2_name
      ,P2.sale_price AS P2_price
     FROM product AS P1
    LEFT OUTER JOIN product AS P2
      ON P1.sale_price >= P2.sale_price
    ORDER BY P1.sale_price,P1.product_id ) AS X
15  GROUP BY product_id, product_name, sale_price
   ORDER BY sale_price,product_id;
```

得到的查询结果为:

product_id	product_name	sale_price	cum_price
0008	圆珠笔	100	100
0002	打孔器	500	1100
0006	叉子	500	1100
0007	擦菜板	880	1980
0001	T恤	1000	2980
0004	菜刀	3000	5980
0003	运动T恤	4000	9980
0005	高压锅	6800	16780

图 5.34: 查询结果

观察上述查询结果发现, 由于有两种商品的售价相同, 在使用 \geq 进行连结时, 导致了累计求和错误, 这是由于这两种商品售价相同导致的. 因此实际上之前是不应该单独只用 \geq 作为连结条件的. 考察我们建立自左连结的本意, 是要找出满足:

1. 比该商品售价更低的, 或者是
2. 该种商品自身, 以及
3. 如果 A 和 B 两种商品售价相等, 则建立连结时, 如果 P1.A 和 P2.A,P2.B 建立了连接, 则 P1.B 不再和 P2.A 建立连结, 因此根据上述约束条件, 利用 ID 的有序性, 进一步将上述查询改写为:

[源码地址 sql](#)

```
Line 1  SELECT  product_id, product_name, sale_price
      ,SUM(P2_price) AS cum_price
     FROM
      (SELECT  P1.product_id, P1.product_name, P1.sale_price
```

```

5          ,P2.product_id AS P2_id
6          ,P2.product_name AS P2_name
7          ,P2.sale_price AS P2_price
8      FROM product AS P1
9      LEFT OUTER JOIN product AS P2
10     ON ((P1.sale_price > P2.sale_price)
11     OR (P1.sale_price = P2.sale_price
12     AND P1.product_id<=P2.product_id))
13     ORDER BY P1.sale_price,P1.product_id) AS X
14 GROUP BY product_id, product_name, sale_price
15 ORDER BY sale_price,cum_price;

```

这次结果就正确了。

product_id	product_name	sale_price	cum_price
0008	圆珠笔	100	100
0006	叉子	500	600
0002	打孔器	500	1100
0007	擦菜板	880	1980
0001	T恤	1000	2980
0004	菜刀	3000	5980
0003	运动T恤	4000	9980
0005	高压锅	6800	16780

图 5.35: 查询结果

5.2.5 交叉连结 - CROSS JOIN(笛卡尔积)

之前的无论是外连结内连结, 一个共同的必备条件就是连结条件-ON 子句, 用来指定连结的条件. 如果你试过不使用这个连结条件的连结查询, 你可能已经发现, 结果会有很多行. 在连结去掉 ON 子句, 就是所谓的交叉连结 (CROSS JOIN), 交叉连结又叫笛卡尔积, 后者是一个数学术语. 两个集合做笛卡尔积, 就是使用集合 A 中的每一个元素与集合 B 中的每一个元素组成一个有序的组合. 数据库表 (或者子查询) 的并, 交和差都是在纵向上对表进行扩张或筛选限制等运算的, 这要求表的列数及对应位置的列的数据类型"相容", 因此这些运算并不会增加新的列, 而交叉连接 (笛卡尔积) 则是在横向上对表进行扩张, 即增加新的列, 这一点和连结的功能是一致的. 但因为没有了 ON 子句的限制, 会对左表和右表的每一行进行组合, 这经常会导致很多无意义的行出现在检索结果中. 当然, 在某些查询需求中, 交叉连结也有一些用处.

交叉连结的语法有如下几种形式:

源码地址 [sql](#)

```

Line 1 -- 1. 使用关键字 CROSS JOIN 显式地进行交叉连结
- SELECT SP.shop_id
-           ,SP.shop_name
-           ,SP.product_id
-           ,P.product_name
-           ,P.sale_price
-      FROM shopproduct AS SP
-      CROSS JOIN product AS P;

```

```
--2. 使用逗号分隔两个表，并省略 ON 子句
10 SELECT SP.shop_id
      ,SP.shop_name
      ,SP.product_id
      ,P.product_name
      ,P.sale_price
15 FROM shopproduct AS SP ,product AS P;
```

请大家试着执行一下以上语句。可能大家会惊讶于结果的行数，但我们还是先来介绍一下语法结构吧。对满足相同规则的表进行交叉连结的集合运算符是 CROSS JOIN (笛卡儿积)。进行交叉连结时无法使用内连结和外连结中所使用的 ON 子句，这是因为交叉连结是对两张表中的全部记录进行交叉组合，因此结果中的记录数通常是两张表中行数的乘积。本例中，因为 shopproduct 表存在 13 条记录，product 表存在 8 条记录，所以结果中就包含了 $13 \times 8 = 104$ 条记录。

可能这时会有读者想起前面我们提到过集合运算中的乘法会在本节中进行详细学习，这就是上面介绍的交叉连结。内连结是交叉连结的一部分，“内”也可以理解为“包含在交叉连结结果中的部分”。相反，外连结的“外”可以理解为“交叉连结结果之外的部分”。

交叉连结没有应用到实际业务之中的原因有两个。一是其结果没有实用价值，二是由于其结果行数太多，需要花费大量的运算时间和高性能设备的支持。

• 【扩展阅读】连结与笛卡儿积的关系

考察笛卡儿积和连结，不难发现，笛卡儿积可以视作一种特殊的连结（事实上笛卡儿积的语法也可以写作 CROSS JOIN），这种连结的 ON 子句是一个恒为真的谓词。

反过来思考，在对笛卡儿积进行适当的限制之后，也就得到了内连结和外连结。

例如，对于 shopproduct 表和 product 表，首先建立笛卡尔乘积：

源码地址 [sql](#)

```
Line 1  SELECT SP.* , P.*
        -   FROM shopproduct AS SP
        -   CROSS JOIN product AS P;
```

查询结果的一部分如下：

shop_id	shop_name	product_id	quantity	product_id(1)	product_name	product_type	sale_price	purchase_price	regist_date
000A	东京	0001	30	0008	圆珠笔	办公用品	100	(Null)	2009-11-11
000A	东京	0001	30	0007	擦菜板	厨房用具	880	790	2008-04-28
000A	东京	0001	30	0006	叉子	厨房用具	500	(Null)	2009-09-20
000A	东京	0001	30	0005	高压锅	厨房用具	6800	5000	2009-01-15
000A	东京	0001	30	0004	菜刀	厨房用具	3000	2800	2009-09-20
000A	东京	0001	30	0003	运动T恤	衣服	4000	2800	(Null)
000A	东京	0001	30	0002	打孔器	办公用品	500	320	2009-09-11
000A	东京	0001	30	0001	T恤	衣服	1000	500	2009-09-20
000A	东京	0002	50	0008	圆珠笔	办公用品	100	(Null)	2009-11-11
000A	东京	0002	50	0007	擦菜板	厨房用具	880	790	2008-04-28
000A	东京	0002	50	0006	叉子	厨房用具	500	(Null)	2009-09-20
000A	东京	0002	50	0005	高压锅	厨房用具	6800	5000	2009-01-15
000A	东京	0002	50	0004	菜刀	厨房用具	3000	2800	2009-09-20
000A	东京	0002	50	0003	运动T恤	衣服	4000	2800	(Null)
000A	东京	0002	50	0002	打孔器	办公用品	500	320	2009-09-11
000A	东京	0002	50	0001	T恤	衣服	1000	500	2009-09-20
000A	东京	0003	15	0008	圆珠笔	办公用品	100	(Null)	2009-11-11
000A	东京	0003	15	0007	擦菜板	厨房用具	880	790	2008-04-28
000A	东京	0003	15	0006	叉子	厨房用具	500	(Null)	2009-09-20
000A	东京	0003	15	0005	高压锅	厨房用具	6800	5000	2009-01-15
000A	东京	0003	15	0004	菜刀	厨房用具	3000	2800	2009-09-20
000A	东京	0003	15	0003	运动T恤	衣服	4000	2800	(Null)
000A	东京	0003	15	0002	打孔器	办公用品	500	320	2009-09-11
000A	东京	0003	15	0001	T恤	衣服	1000	500	2009-09-20
000B	名古屋	0002	30	0008	圆珠笔	办公用品	100	(Null)	2009-11-11
000B	名古屋	0002	30	0007	擦菜板	厨房用具	880	790	2008-04-28
000B	名古屋	0002	30	0006	叉子	厨房用具	500	(Null)	2009-09-20
000B	名古屋	0002	30	0005	高压锅	厨房用具	6800	5000	2009-01-15
000B	名古屋	0002	30	0004	菜刀	厨房用具	3000	2800	2009-09-20
000B	名古屋	0002	30	0003	运动T恤	衣服	4000	2800	(Null)
000B	名古屋	0002	30	0002	打孔器	办公用品	500	320	2009-09-11
000B	名古屋	0002	30	0001	T恤	衣服	1000	500	2009-09-20
000B	名古屋	0003	120	0008	圆珠笔	办公用品	100	(Null)	2009-11-11
000B	名古屋	0003	120	0007	擦菜板	厨房用具	880	790	2008-04-28
000B	名古屋	0003	120	0006	叉子	厨房用具	500	(Null)	2009-09-20
000B	名古屋	0003	120	0005	高压锅	厨房用具	6800	5000	2009-01-15
000B	名古屋	0003	120	0004	菜刀	厨房用具	3000	2800	2009-09-20
000B	名古屋	0003	120	0003	运动T恤	衣服	4000	2800	(Null)
000B	名古屋	0003	120	0002	打孔器	办公用品	500	320	2009-09-11
000B	名古屋	0003	120	0001	T恤	衣服	1000	500	2009-09-20
000B	名古屋	0004	20	0008	圆珠笔	办公用品	100	(Null)	2009-11-11
000B	名古屋	0004	20	0007	擦菜板	厨房用具	880	790	2008-04-28
000B	名古屋	0004	20	0006	叉子	厨房用具	500	(Null)	2009-09-20
000B	名古屋	0004	20	0005	高压锅	厨房用具	6800	5000	2009-01-15
000B	名古屋	0004	20	0004	菜刀	厨房用具	3000	2800	2009-09-20
000B	名古屋	0004	20	0003	运动T恤	衣服	4000	2800	(Null)
000B	名古屋	0004	20	0002	打孔器	办公用品	500	320	2009-09-11
000B	名古屋	0004	20	0001	T恤	衣服	1000	500	2009-09-20
000B	名古屋	0006	10	0008	圆珠笔	办公用品	100	(Null)	2009-11-11
000B	名古屋	0006	10	0007	擦菜板	厨房用具	880	790	2008-04-28
000B	名古屋	0006	10	0006	叉子	厨房用具	500	(Null)	2009-09-20
000B	名古屋	0006	10	0005	高压锅	厨房用具	6800	5000	2009-01-15
000B	名古屋	0006	10	0004	菜刀	厨房用具	3000	2800	2009-09-20
000B	名古屋	0006	10	0003	运动T恤	衣服	4000	2800	(Null)
000B	名古屋	0006	10	0002	打孔器	办公用品	500	320	2009-09-11
000B	名古屋	0006	10	0001	T恤	衣服	1000	500	2009-09-20
000B	名古屋	0007	40	0008	圆珠笔	办公用品	100	(Null)	2009-11-11

图 5.36: 查询结果

然后对上述笛卡尔乘积增加筛选条件 SP.product_id=P.product_id, 就得到了和内连结一致的结果:

源码地址 [sql](#)

```
Line 1  SELECT SP.* , P.*  
        FROM shopproduct AS SP  
    CROSS JOIN product AS P  
   WHERE SP.product_id = P.product_id;
```

查询结果如下:

shop_id	shop_name	product_id	quantity	product_id(1)	product_name	product_type	sale_price	purchase_price	regist_date
000A	东京	0001	30	0001	T恤	衣服	1000	500	2009-09-20
000A	东京	0002	50	0002	打孔器	办公用品	500	320	2009-09-11
000A	东京	0003	15	0003	运动T恤	衣服	4000	2800	(Null)
000B	名古屋	0002	30	0002	打孔器	办公用品	500	320	2009-09-11
000B	名古屋	0003	120	0003	运动T恤	衣服	4000	2800	(Null)
000B	名古屋	0004	20	0004	菜刀	厨房用具	3000	2800	2009-09-20
000B	名古屋	0006	10	0006	叉子	厨房用具	500	(Null)	2009-09-20
000B	名古屋	0007	40	0007	擦菜板	厨房用具	880	790	2008-04-28
000C	大阪	0003	20	0003	运动T恤	衣服	4000	2800	(Null)
000C	大阪	0004	50	0004	菜刀	厨房用具	3000	2800	2009-09-20
000C	大阪	0006	90	0006	叉子	厨房用具	500	(Null)	2009-09-20
000C	大阪	0007	70	0007	擦菜板	厨房用具	880	790	2008-04-28
000D	福冈	0001	100	0001	T恤	衣服	1000	500	2009-09-20

图 5.37: 查询结果

实际上,正如书中所说,上述写法中,将 CROSS JOIN 改为逗号后,正是内连结的旧式写法,但在 ANSI 和 ISO 的 SQL-92 标准中,已经将使用 INNER JOIN ..ON.. 的写法规定为标准写法,因此极力推荐大家在平时写 SQL 查询时,使用规范写法.

5.2.6 连结的特定语法和过时语法

在笛卡尔积的基础上,我们增加一个 WHERE 子句,将之前的连结条件作为筛选条件加进去,我们会发现,得到的结果恰好是直接使用内连接的结果.

试执行以下查询,并将查询结果与内连结一节第一个例子的结果做对比.

源码地址 [sql](#)

```
Line 1 SELECT SP.shop_id
      ,SP.shop_name
      ,SP.product_id
      ,P.product_name
      ,P.sale_price
  FROM shopproduct AS SP
  CROSS JOIN product AS P
 WHERE SP.product_id = P.product_id;
```

我们发现,这两个语句得到的结果是相同的.之前我们学习的内连结和外连结的语法都符合标准 SQL 的规定,可以在所有 DBMS 中执行,因此大家可以放心使用.但是如果大家之后从事系统开发工作,或者阅读遗留 SQL 查询语句的话,一定会碰到需要阅读他人写的代码并进行维护的情况,而那些使用特定和过时语法的程序就会成为我们的麻烦.

SQL 是一门特定语法及过时语法非常多的语言,虽然之前本书中也多次提及,但连结是其中特定语法的部分,现在还有不少年长的程序员和系统工程师仍在使用这些特定的语法.例如,将本节最初介绍的内连结的 SELECT 语句替换为过时语法的结果如下所示.

使用过时语法的内连结 (结果与代码清单 7-9 相同)

源码地址 [sql](#)

```
Line 1 SELECT SP.shop_id
      ,SP.shop_name
      ,SP.product_id
      ,P.product_name
      ,P.sale_price
```

```
- FROM shopproduct SP,product P  
- WHERE SP.product_id = P.product_id  
- AND SP.shop_id = '000A';
```

这样的书写方式所得到的结果与标准语法完全相同，并且这样的语法可以在所有的 DBMS 中执行，并不能算是特定的语法，只是过时了而已。但是，由于这样的语法不仅过时，而且还存在很多其他的问题，因此不推荐大家使用，理由主要有以下三点：

第一，使用这样的语法无法马上判断出到底是内连结还是外连结（又或者是其他种类的连结）。

第二，由于连结条件都写在 WHERE 子句之中，因此无法在短时间内分辨出哪部分是连结条件，哪部分是用来选取记录的限制条件。

第三，我们不知道这样的语法到底还能使用多久。每个 DBMS 的开发者都会考虑放弃过时的语法，转而支持新的语法。虽然并不是马上就不能使用了，但那一天总会到来的。

虽然这么说，但是现在使用这些过时语法编写的程序还有很多，到目前为止还都能正常执行。我想大家很可能会碰到这样的代码，因此还是希望大家能够了解这些知识。

练习题

5.1

找出 product 和 product2 中售价高于 500 的商品的基本信息。

5.2

借助对称差的实现方式，求 product 和 product2 的交集。

5.3

每类商品中售价最高的商品都在哪些商店有售？

5.4

分别使用内连结和关联子查询每一类商品中售价最高的商品。

5.5

用关联子查询实现：在 product 表中，取出 product_id, product_name, sale_price，并按照商品的售价从低到高进行排序、对售价进行累计求和。

第 6 章 SQL 高级处理

6.1 窗口函数

6.1.1 窗口函数概念及基本的使用方法

窗口函数也称为 OLAP 函数。OLAP 是 OnLine AnalyticalProcessing 的简称，意思是对数据库数据进行实时分析处理。

为了便于理解，称之为窗口函数。常规的 SELECT 语句都是对整张表进行查询，而窗口函数可以让我们有选择的去某一部分数据进行汇总、计算和排序。

窗口函数的通用形式：

[源码地址 sql](#)

```
Line 1 <窗口函数> OVER ([PARTITION BY <列名>
    ORDER BY <排序用列名>)
```

[] 中的内容可以省略。

窗口函数最关键的是搞明白关键字 PARTITION BY 和 ORDER BY 的作用。

PARTITION BY 是用来分组，即选择要看哪个窗口，类似于 GROUP BY 子句的分组功能，但是 PARTITION BY 子句并不具备 GROUP BY 子句的汇总功能，并不会改变原始表中记录的行数。

ORDER BY 是用来排序，即决定窗口内，是按那种规则（字段）来排序的。

举个栗子：

[源码地址 sql](#)

```
Line 1 SELECT product_name
    ,product_type
    ,sale_price
    ,RANK() OVER (PARTITION BY product_type
        ORDER BY sale_price) AS ranking
5
    FROM product;
```

得到的结果是：

执行结果

product_name	product_type	sale_price	ranking
叉子	厨房用具	500	1
擦菜板	厨房用具	880	2
菜刀	厨房用具	3000	3
高压锅	厨房用具	6800	4
T恤衫	衣服	1000	1
运动T恤	衣服	4000	2
圆珠笔	办公用品	100	1
打孔器	办公用品	500	2

图 6.1：执行结果

我们先忽略生成的新列 - [ranking]，看下原始数据在 PARTITION BY 和 ORDER BY 关键字的作用下发生了什么变化。

PARTITION BY 能够设定窗口对象范围。本例中，为了按照商品种类进行排序，我们指定了 product_type。即一个商品种类就是一个小的"窗口"。

ORDER BY 能够指定按照哪一列、何种顺序进行排序。为了按照销售单价的升序进行排列，我们指定了 sale_price。此外，窗口函数中的 ORDER BY 与 SELECT 语句末尾的 ORDER BY 一样，可以通过关键字 ASC/DESC 来指定升序/降序。省略该关键字时会默认按照 ASC，也就是

升序进行排序。本例中就省略了上述关键字。

图 8-1 PARTITION BY 和 ORDER BY 的作用

The diagram shows a table with 8 rows, partitioned into three groups by product type: Kitchenware (rows 1-4), Clothing (rows 5-6), and Office Supplies (rows 7-8). A dashed bracket on the left indicates the partitions. A blue arrow labeled 'PARTITION BY 的分组 (根据商品种类)' points from the bottom to the vertical boundary between the partitions. Another blue arrow labeled 'ORDER BY 的顺序 (销售单价由低到高的顺序)' points from the top to the 'sale_price' column, indicating the sorting order.

product_id (商品编号)	product_name (商品名称)	product_type (商品种类)	sale_price (销售单价)	purchase_price (进货单价)	regist_date (登记日期)
0006	叉子	厨房用具	500		2009-09-20
0007	擦菜板	厨房用具	880	790	2008-04-28
0004	菜刀	厨房用具	3000	2800	2009-09-20
0005	高压锅	厨房用具	6800	5000	2009-01-15
0001	T恤衫	衣服	1000	500	2009-09-20
0003	运动T恤	衣服	4000	2800	
0008	圆珠笔	办公用品	100		2009-11-11
0002	打孔器	办公用品	500	320	2009-09-11

图 6.2: PARTITION BY 和 ORDER BY 的作用

6.2 窗口函数种类

大致来说，窗口函数可以分为两类。

1. 将 SUM、MAX、MIN 等聚合函数用在窗口函数中
2. RANK、DENSE_RANK 等排序用的专用窗口函数

6.2.1 专用窗口函数

- RANK 函数（英式排序）

计算排序时，如果存在相同位次的记录，则会跳过之后的位次。

例) 有 3 条记录排在第 1 位时：1 位、1 位、1 位、4 位……

- DENSE_RANK 函数（中式排序）

同样是计算排序，即使存在相同位次的记录，也不会跳过之后的位次。

例) 有 3 条记录排在第 1 位时：1 位、1 位、1 位、2 位……

- ROW_NUMBER 函数

赋予唯一的连续位次。

例) 有 3 条记录排在第 1 位时：1 位、2 位、3 位、4 位

运行以下代码：

[源码地址 sql](#)

```
Line 1  SELECT product_name
      ,product_type
      ,sale_price
      ,RANK() OVER (ORDER BY sale_price) AS ranking
      ,DENSE_RANK() OVER (ORDER BY sale_price) AS dense_ranking
      ,ROW_NUMBER() OVER (ORDER BY sale_price) AS row_num
FROM product;
```

执行结果			RANK	DENSE_RANK	ROW_NUMBER
product_name	product_type	sale_price	ranking	dense_ranking	row_num
圆珠笔	办公用品	100	1	1	1
叉子	厨房用具	500	2	2	2
打孔器	办公用品	500	2	2	3
擦菜板	厨房用具	880	4	3	4
T恤衫	衣服	1000	5	4	5
菜刀	厨房用具	3000	6	5	6
运动T恤	衣服	4000	7	6	7
高压锅	厨房用具	6800	8	7	8

图 6.3: 执行结果

6.2.2 聚合函数在窗口函数上的使用

聚合函数在开窗函数中的使用方法和之前的专用窗口函数一样，只是出来的结果是一个累计的聚合函数值。

运行以下代码：

[源码地址 sql](#)

```
Line 1  SELECT product_id
      ,product_name
      ,sale_price
      ,SUM(sale_price) OVER (ORDER BY product_id) AS current_sum
FROM product;
```

执行结果

product_id	product_name	sale_price	current_sum	
0001	T恤衫	1000	1000	←1000
0002	打孔器	500	1500	←1000+500
0003	运动T恤	4000	5500	←1000+500+4000
0004	菜刀	3000	8500	←1000+500+4000+3000
0005	高压锅	6800	15300	.
0006	叉子	500	15800	.
0007	擦菜板	880	16680	.
0008	圆珠笔	100	16780	.

图 6.4: current_sum 执行结果

源码地址 [sql](#)

```

Line 1  SELECT product_id
      ,product_name
      ,sale_price
      ,AVG(sale_price) OVER (ORDER BY product_id) AS current_avg
5   FROM product;

```

执行结果

product_id	product_name	sale_price	current_avg	
0001	T恤衫	1000	1000.0000000000000000	←(1000)/1
0002	打孔器	500	750.0000000000000000	←(1000+500)/2
0003	运动T恤	4000	1833.3333333333333333	←(1000+500+4000)/3
0004	菜刀	3000	2125.0000000000000000	←(1000+500+4000+3000)/4
0005	高压锅	6800	3060.0000000000000000	←(1000+500+4000+3000+6800)/5
0006	叉子	500	2633.3333333333333333	.
0007	擦菜板	880	2382.8571428571428571	.
0008	圆珠笔	100	2097.5000000000000000	.

图 6.5: current_avg 执行结果

可以看出，聚合函数结果是，按我们指定的排序，这里是 product_id，当前所在行及之前所有的行的合计或均值。即累计到当前行的聚合。

6.3 窗口函数的应用 - 计算移动平均

在上面提到，聚合函数在窗口函数使用时，计算的是累积到当前行的所有的数据的聚合。实际上，还可以指定更加详细的汇总范围。该汇总范围成为框架 (frame)。

语法

[源码地址 sql](#)

```

Line 1 <窗口函数> OVER (ORDER BY <排序用列名>
-           ROWS n PRECEDING )
-
- <窗口函数> OVER (ORDER BY <排序用列名>
5          ROWS BETWEEN n PRECEDING AND n FOLLOWING)

```

PRECEDING (“之前”), 将框架指定为“截止到之前 n 行”, 加上自身行
 FOLLOWING (“之后”), 将框架指定为“截止到之后 n 行”, 加上自身行
 BETWEEN 1 PRECEDING AND 1 FOLLOWING, 将框架指定为“之前 1 行” + “之后 1 行” +
 “自身”

注意观察框架的范围。

ROWS 2 PRECEDING:

[源码地址 sql](#)

```

Line 1 SELECT product_id
-         ,product_name
-         ,sale_price
-         ,AVG(sale_price) OVER (ORDER BY product_id
5             ROWS 2 PRECEDING) AS moving_avg
-     FROM product;

```

product_id	product_name	sale_price	moving_avg
0001	T恤衫	1000	1000 ←(1000)/1
0002	打孔器	500	750 ←(1000+500)/2
0003	运动T恤	4000	1833 ←(1000+500+4000)/3
0004	菜刀	3000	2500 ←(500+4000+3000)/3
0005	高压锅	6800	4600 ←(4000+3000+6800)/3
0006	叉子	500	3433 .
0007	擦菜板	880	2726 .
0008	圆珠笔	100	493 .

图 6.6: ROWS 2 PRECEDING 执行结果

ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING:

[源码地址 sql](#)

```

Line 1 SELECT product_id
-         ,product_name
-         ,sale_price
-         ,AVG(sale_price) OVER (ORDER BY product_id
5             ROWS BETWEEN 1 PRECEDING
-                           AND 1 FOLLOWING) AS moving_avg
-     FROM product;

```

product_id	product_name	sale_price	moving_avg	
0001	T恤衫	1000	750	$\leftarrow (1000+500)/2$
0002	打孔器	500	1833	$\leftarrow (1000+500+4000)/3$
0003	运动T恤	4000	2500	$\leftarrow (500+4000+3000)/3$
0004	菜刀	3000	4600	$\leftarrow (4000+3000+6800)/3$
0005	高压锅	6800	3433	.
0006	叉子	500	2726	.
0007	擦菜板	880	493	.
0008	圆珠笔	100	490	.

图 6.7: ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING 执行结果

6.3.1 窗口函数适用范围和注意事项

原则上，窗口函数只能在 SELECT 子句中使用。窗口函数 OVER 中的 ORDER BY 子句并不会影响最终结果的排序。其只是用来决定窗口函数按何种顺序计算。

6.4 GROUPING 运算符

6.4.1 ROLLUP - 计算合计及小计

常规的 GROUP BY 只能得到每个分类的小计，有时候还需要计算分类的合计，可以用 ROLLUP 关键字。

源码地址 [sql](#)

```
Line 1  SELECT product_type
      , regist_date
      , SUM(sale_price) AS sum_price
      FROM product
      GROUP BY product_type, regist_date WITH ROLLUP
```

得到的结果为：

product_type	regist_date	sum_price
办公用品	2009-09-11	500
办公用品	2009-11-11	100
办公用品	NULL	600
厨房用具	2008-04-28	880
厨房用具	2009-01-15	6800
厨房用具	2009-09-20	3500
厨房用具	NULL	11180
衣服	NULL	4000
衣服	2009-09-20	1000
衣服	NULL	5000
NULL	NULL	16780

← 小计(办公用品)
← 小计(厨房用品)
← 小计(衣服)
← 合计

图 6.8: ROLLUP 执行结果

product_type	regist_date	sum_price
厨房用具		16780 ←合计
厨房用具	2008-04-28	11180 ←小计 (厨房用具)
厨房用具	2009-01-15	880
厨房用具	2009-09-20	6800
办公用品		3500
办公用品	2009-09-11	600 ←小计 (办公用品)
办公用品	2009-11-11	500
衣服		100
衣服	2009-09-20	5000 ←小计 (衣服)
衣服		1000
衣服		4000

图 6.9: ROLLUP 执行结果

这里 ROLLUP 对 product_type, regist_date 两列进行合计汇总。结果实际上有三层聚合，如下图模块 3 是常规的 GROUP BY 的结果，需要注意的是衣服有个注册日期为空的，这是本来数据就存在日期为空的，不是对衣服类别的合计；模块 2 和 1 是 ROLLUP 带来的合计，模块 2 是对产品种类的合计，模块 1 是对全部数据的总计。

ROLLUP 可以对多列进行汇总求小计和合计。

product_type	regist_date	sum_price
		16780 模块①
厨房用具		11180
办公用品		600 模块②
衣服		5000
办公用品	2009-09-11	500
办公用品	2009-11-11	100
厨房用具	2008-04-28	880
厨房用具	2009-01-15	6800 模块③
厨房用具	2009-09-20	3500
衣服	2009-09-20	1000
衣服		4000

图 6.10: ROLLUP 多列汇总

练习题

6.1

请说出针对本章中使用的 product (商品) 表执行如下 SELECT 语句所能得到的结果。

源码地址 [sql](#)

Line 1 `SELECT product_id`

```
-     ,product_name  
-     ,sale_price  
-     ,MAX(sale_price) OVER (ORDER BY product_id) AS Current_max_price  
5  FROM product
```

6.2

继续使用 product 表,计算出按照登记日期(regist_date)升序进行排列的各日期的销售单价(sale_price)的总额。排序是需要将登记日期为 NULL 的“运动 T 恤”记录排在第 1 位(也就是将其看作比其他日期都早)

6.3

思考题

1. 窗口函数不指定 PARTITION BY 的效果是什么?
2. 为什么说窗口函数只能在 SELECT 子句中使用? 实际上,在 ORDER BY 子句使用系统并不会报错。

第 7 章 综合练习

练习一：各部门工资最高的员工（难度：中等）

创建 Employee 表，包含所有员工信息，每个员工有其对应的 Id, salary 和 department Id。

[源码地址 sql](#)

```
Line 1
- +-----+-----+-----+
- | Id | Name | Salary | DepartmentId |
- +-----+-----+-----+
5 | 1 | Joe | 70000 | 1
- | 2 | Henry | 80000 | 2
- | 3 | Sam | 60000 | 2
- | 4 | Max | 90000 | 1
- +-----+-----+-----+
10
- +-----+
- | Id | Name |
- +-----+
- | 1 | IT |
- 15 | 2 | Sales |
- +-----+
```

编写一个 SQL 查询，找出每个部门工资最高的员工。例如，根据上述给定的表格，Max 在 IT 部门有最高工资，Henry 在 Sales 部门有最高工资。

[源码地址 sql](#)

```
Line 1
- +-----+-----+-----+
- | Department | Employee | Salary |
- +-----+-----+-----+
- | IT | Max | 90000 |
5 | Sales | Henry | 80000 |
- +-----+-----+-----+
```

练习二：换座位（难度：中等）

小美是一所中学的信息科技老师，她有一张 seat 座位表，平时用来储存学生名字和与他们相对应的座位 id。

其中纵列的 **id** 是连续递增的

小美想改变相邻俩学生的座位。

你能不能帮她写一个 SQL query 来输出小美想要的结果呢？

请创建如下所示 seat 表：

示例：

[源码地址 sql](#)

```
Line 1 +-----+-----+
- | id   | student |
- +-----+-----+
- | 1    | Abbot   |
5 | 2    | Doris   |
- | 3    | Emerson |
- | 4    | Green   |
- | 5    | Jeames  |
- +-----+-----+
```

假如数据输入的是上表，则输出结果如下：

[源码地址 sql](#)

```
Line 1 +-----+-----+
- | id   | student |
- +-----+-----+
- | 1    | Doris   |
5 | 2    | Abbot   |
- | 3    | Green   |
- | 4    | Emerson |
- | 5    | Jeames  |
- +-----+-----+
```

注意：如果学生人数是奇数，则不需要改变最后一个同学的座位。

练习三：分数排名（难度：中等）

编写一个 SQL 查询来实现分数排名。如果两个分数相同，则两个分数排名（Rank）相同。请注意，平分后的下一个名次应该是下一个连续的整数值。换句话说，名次之间不应该有“间隔”。

创建以下 score 表：

[源码地址 sql](#)

```
Line 1 +-----+
- | Id  | Score  |
- +-----+
- | 1   | 3.50  |
5 | 2   | 3.65  |
- | 3   | 4.00  |
- | 4   | 3.85  |
- | 5   | 4.00  |
- | 6   | 3.65  |
10 +-----+
```

例如，根据上述给定的 Scores 表，你的查询应该返回（按分数从高到低排列）：

[源码地址 sql](#)

```
Line 1 +-----+-----+
```

```

- | Score | Rank |
- +-----+-----+
- | 4.00 | 1    |
5 | 4.00 | 1    |
- | 3.85 | 2    |
- | 3.65 | 3    |
- | 3.65 | 3    |
- | 3.50 | 4    |
10 +-----+-----+

```

练习四：连续出现的数字（难度：中等）

编写一个 SQL 查询，查找所有至少连续出现三次的数字。

[源码地址](#) [sql](#)

```

Line 1 +-----+
- | Id | Num |
- +-----+
- | 1  | 1  |
5 | 2  | 1  |
- | 3  | 1  |
- | 4  | 2  |
- | 5  | 1  |
- | 6  | 2  |
10 | 7  | 2  |
- +-----+

```

例如，给定上面的 Logs 表，1 是唯一连续出现至少三次的数字。

[源码地址](#) [sql](#)

```

Line 1 +-----+
- | ConsecutiveNums |
- +-----+
- | 1                 |
5 | +-----+

```

练习五：树节点（难度：中等）

对于 tree 表，**id** 是树节点的标识，**p_id** 是其父节点的 **id**。

[源码地址](#) [sql](#)

```

Line 1 +-----+
- | id | p_id |
- +-----+
- | 1  | null  |
5 | 2  | 1     |

```

```

- | 3 | 1 |
- | 4 | 2 |
- | 5 | 2 |
- +---+---+

```

每个节点都是以下三种类型中的一种：

Root: 如果节点是根节点。

Leaf: 如果节点是叶子节点。

Inner: 如果节点既不是根节点也不是叶子节点。

写一条查询语句打印节点 id 及对应的节点类型。按照节点 id 排序。上面例子的对应结果为：

[源码地址](#) [sql](#)

```

Line 1 +---+---+
- | id | Type |
- +---+---+
- | 1 | Root |
5 | 2 | Inner |
- | 3 | Leaf |
- | 4 | Leaf |
- | 5 | Leaf |
- +---+---+

```

说明

节点'1' 是根节点，因为它的父节点为 NULL，有'2' 和'3' 两个子节点。

节点'2' 是内部节点，因为它的父节点是'1'，有子节点'4' 和'5'。

节点'3', '4', '5' 是叶子节点，因为它们有父节点但没有子节点。

下面是树的图形：

[源码地址](#) [sql](#)

```

Line 1
      1
     /   \
    2     3
   /   \
  4     5

```

注意

如果一个树只有一个节点，只需要输出根节点属性。

练习六：至少有五名直接下属的经理（难度：中等）

Employee 表包含所有员工及其上级的信息。每位员工都有一个 Id，并且还有一个对应主管的 Id (ManagerId)。

[源码地址](#) [sql](#)

```

Line 1 +---+---+---+---+
- | Id | Name | Department | ManagerId |

```

-	101 John	A	null	
5	102 Dan	A	101	
-	103 James	A	101	
-	104 Amy	A	101	
-	105 Anne	A	101	
-	106 Ron	B	101	
10	+-----+	+-----+	+-----+	+-----+

针对 Employee 表，写一条 SQL 语句找出有 5 个下属的主管。对于上面的表，结果应输出：

源码地址 [sql](#)

Line	1	+-----+
-	Name	
-	+-----+	
-	John	
5	+-----+	

注意：

没有人向自己汇报。

练习七：分数排名（难度：中等）

练习三的分数表，实现排名功能，但是排名需要是非连续的，如下：

源码地址 [sql](#)

Line	1	+-----+
-	Score	Rank
-	+-----+	+-----+
-	4.00	1
5	4.00	1
-	3.85	3
-	3.65	4
-	3.65	4
-	3.50	6
10	+-----+	+-----+

练习八：查询回答率最高的问题（难度：中等）

求出 survey_log 表中回答率最高的问题，表格的字段有：uid, action, question_id, answer_id, q_num, timestamp。

uid 是用户 id; action 的值为：“show”，“answer”，“skip”；当 action 是"answer" 时，answer_id 不为空，相反，当 action 是"show" 和"skip" 时为空 (null); q_num 是问题的数字序号。

写一条 sql 语句找出回答率最高的问题。

举例：

输入

[源码地址](#) [sql](#)

```

Line 1 | uid   | action  | question_id | answer_id | q_num|timestamp |
- |-----|:-----|:-----|:-----|:-----|:-----|
- | 5    | show   | 285        | null      | 1     | 123   |
- | 5    | answer  | 285        | 124124    | 1     | 124   |
5 | 5    | show   | 369        | null      | 2     | 125   |
- | 5    | skip   | 369        | null      | 2     | 126   |

-
- 输出

10 | survey_log |
- |-----|
- | 285   |

```

说明

问题 285 的回答率为 1/1，然而问题 369 的回答率是 0/1，所以输出是 285。

注意：最高回答率的意思是：同一个问题出现的次数中回答的比例。

练习九：各部门前 3 高工资的员工（难度：中等）

将项目一中的 employee 表清空，重新插入以下数据（其实是多插入 5,6 两行）：

[源码地址](#) [sql](#)

```

Line 1 +-----+-----+-----+
- | Id  | Name   | Salary | DepartmentId |
- +-----+-----+-----+
- | 1   | Joe    | 70000  | 1           |
5 | 2   | Henry  | 80000  | 2           |
- | 3   | Sam    | 60000  | 2           |
- | 4   | Max    | 90000  | 1           |
- | 5   | Janet  | 69000  | 1           |
- | 6   | Randy  | 85000  | 1           |
10 +-----+-----+-----+

```

编写一个 SQL 查询，找出每个部门工资前三高的员工。例如，根据上述给定的表格，查询结果应返回：

[源码地址](#) [sql](#)

```

Line 1 +-----+-----+-----+
- | Department | Employee | Salary |
- +-----+-----+-----+
- | IT         | Max      | 90000  |
5 | IT         | Randy    | 85000  |
- | IT         | Joe      | 70000  |
- | Sales      | Henry    | 80000  |
- | Sales      | Sam      | 60000  |

```

```
+-----+-----+-----+
|
```

此外，请考虑实现各部门前 N 高工资的员工功能。

练习十：平面上最近距离（难度：困难）

point_2d 表包含一个平面内一些点（超过两个）的坐标值 (x, y)。

写一条查询语句求出这些点中的最短距离并保留 2 位小数。

[源码地址](#) [sql](#)

```
Line 1 | x    | y    |
- | ---- | ---- |
- | -1   | -1   |
- | 0    | 0    |
5 | -1   | -2   |
```

最短距离是 1，从点 (-1, -1) 到点 (-1, -2)。所以输出结果为：

```
| shortest |
1.00
```

[源码地址](#) [sql](#)

```
+-----+
| shortest |
+-----+
| 1.00    |
5 +-----+
```

注意：所有点的最大距离小于 10000。

练习十一：行程和用户（难度：困难）

Trips 表中存所有出租车的行程信息。每段行程有唯一键 Id，Client_Id 和 Driver_Id 是 **Users** 表中 Users_Id 的外键。Status 是枚举类型，枚举成员为 (‘completed’，‘cancelled_by_driver’，‘cancelled_by_client’)。

[源码地址](#) [sql](#)

```
Line 1 | Id    | Client_Id | Driver_Id | City_Id | Status           | Request_at |
- | :--- | :----- | :----- | :----- | :----- | :----- |
- | 1    | 1        | 10       | 1        | completed      | 2013-10-1 |
- | 2    | 2        | 11       | 1        | cancelled_by_driver | 2013-10-1 |
5 | 3    | 3        | 12       | 6        | completed      | 2013-10-1 |
- | 4    | 4        | 13       | 6        | cancelled_by_client | 2013-10-1 |
- | 5    | 1        | 10       | 1        | completed      | 2013-10-2 |
- | 6    | 2        | 11       | 6        | completed      | 2013-10-2 |
- | 7    | 3        | 12       | 6        | completed      | 2013-10-2 |
10 | 8    | 2        | 12       | 12       | completed     | 2013-10-3 |
- | 9    | 3        | 10       | 12       | completed     | 2013-10-3 |
- | 10   | 4        | 13       | 12       | cancelled_by_driver | 2013-10-3 |
```

Users 表存所有用户。每个用户有唯一键 Users_Id。Banned 表示这个用户是否被禁止，Role 则是一个表示（‘client’，‘driver’，‘partner’）的枚举类型。

源码地址 [sql](#)

```
Line 1 +-----+-----+-----+
- | Users_Id | Banned | Role   |
- +-----+-----+-----+
- |     1    | No     | client |
5 |     2    | Yes    | client |
- |     3    | No     | client |
- |     4    | No     | client |
- |    10    | No     | driver |
- |    11    | No     | driver |
10 |    12    | No     | driver |
- |    13    | No     | driver |
- +-----+-----+-----+
```

写一段 SQL 语句查出 2013 年 10 月 1 日至 2013 年 10 月 3 日期间非禁止用户的取消率。基于上表，你的 SQL 语句应返回如下结果，取消率（Cancellation Rate）保留两位小数。

源码地址 [sql](#)

```
Line 1 +-----+-----+
- | Day      | Cancellation Rate |
- +-----+-----+
- | 2013-10-01 | 0.33        |
5 | 2013-10-02 | 0.00        |
- | 2013-10-03 | 0.50        |
- +-----+-----+
```

第 8 章 附录 1 - SQL 语法规规范

语法规范

源码地址 [sql](#)

```

Line 1 -- 样例一
- (SELECT flora.species_name
-     ,AVG(flora.height) AS average_height
-     ,AVG(flora.diameter) AS average_diameter
5   FROM flora
- WHERE flora.species_name = 'Banksia'
-     OR flora.species_name = 'Sheoak'
-     OR flora.species_name = 'Wattle'
- GROUP BY flora.species_name, flora.observation_date)
10 UNION ALL
- (SELECT botanic.species_name
-     ,AVG(b.height) AS average_height
-     ,AVG(b.diameter) AS average_diameter
-   FROM botanic_garden_flora AS botanic
15 WHERE botanic.species_name = 'Banksia'
-     OR botanic.species_name = 'Sheoak'
-     OR botanic.species_name = 'Wattle'
- GROUP BY botanic.species_name, botanic.observation_date);
- -- 样例二
20 SELECT botanic.species_name
-     ,AVG(b.height) AS average_height
-     ,AVG(b.diameter) AS average_diameter
-   FROM botanic_garden_flora AS botanic
- WHERE botanic.species_name IN (SELECT species_name
25                               FROM flora
-                                     WHERE height >= 3)
- GROUP BY botanic.species_name, botanic.observation_da;
-
- -- 样例三
30 SELECT SP.shop_id, SP.shop_name, SP.product_id, P.product_name, P.product_type
-     , P.purchase_price
-   FROM shopproduct AS SP
- INNER JOIN -- product
-     (SELECT product_name, product_type, purchase_price
-       FROM Product ) AS P
-     ON SP.product_id=P.product_id
- WHERE P.product_type='衣服';

```

SQL 语法规规范总得的原则是，清楚、易读并且层次清晰。实际场景中常常动辄几百上千行的 SQL 语句，如果不写清楚，事后 review 或者别人接手的时候，会让人怀疑人生。

当然这些规范都是笔者根据实际经验总结归纳的，并不是金科铁律，但是强烈建议新手按以下规范入门编写 SQL 语句。

常见注意事项如下：

1. MySQL 本身不区分大小写，但强烈要求关键字大写，表名、列名用小写；
2. 创建表时，使用统一的、描述性强的字段命名规则保证字段名是独一无二且不是保留字的，不要使用连续的下划线，不用下划线结尾；最好以字母开头；
3. 关键字右对齐，且不同层级的用空格或缩进控制，使其区分开，见样例二；
4. 列名少的时候写在一行里无伤大雅；多的时候以及涉及到 CASE WHEN 或者聚合计算的时候，建议分行写；个人习惯是逗号在列名前面，方便之后删除某些列，放列名后亦可；
5. 表别名和列别名尽量用有具体含义的词组，不要用 a b c，不然以后 review 的时候会非常痛苦；
6. 运算符前后都加一个空格；
7. 当用到多个表时，请在所有列名前写上引用的表别名，不要嫌麻烦；
8. 每条命令用分号结尾；
9. 养成随手写注释的习惯，注释方法：
 - 单行注释 # 注释文字
 - 单行注释 – 注释文字
 - 多行注释：/* 注释文字 */

P.S. 养成良好的书写习惯，可以避免以后发生流血事件。

参考资料：

1. SQL 编程风格: <https://zhuanlan.zhihu.com/p/27466166>
2. SQL Style Guide: <https://www.sqlstyle.guide/>

第 9 章 附录 2 - [选学] 使用 Python 连接 MySQL

9.1 使用 Python 连接 MySQL

Python 是最近几年最热门的编程语言, 特别是在数据科学和机器学习领域, Python 已经成为了主流的编程语言. 和其他如 JAVA,C++ 等通用编程语言一样, Python 也能够连接 MySQL 执行查询, 并将结果从数据库中取回, 以供 Python 程序使用。

以下我们假设大家已经安装了 anaconda 套装 (推荐, 如尚未安装 anaconda 可参考 <https://zhuanlan.zhihu.com/p/75717350> 或者自行安装配置好了 Python 环境加 Jupyter notebook , 接下来介绍在 Jupyter notebook 中使用 Python3 的 PyMySQL 模块连接 MySQL 并执行 SQL 查询和取回查询结果的方法。

9.1.1 安装 PyMySQL 模块

首先安装 PyMySQL 模块, 这里选择直接在 jupyter notebook 中进行安装.

源码地址 [python](#)

```
Line 1 # 安装 pymysql 模块
- pip install pymysql
```

安装完成后, 会有如下的输出, 表示已成功安装该模块.

```
Collecting pymysql
  Downloading PyMySQL-0.10.0-py2.py3-none-any.whl (47 kB)
Installing collected packages: pymysql
Successfully installed pymysql-0.10.0
```

图 9.1: install_pymysql

9.1.2 导入模块

现在我们就可以导入 PyMySQL 模块, 然后建立到 MySQL 的连接了. 如果下述代码执行后没有任何输出, 则是正常导入了该模块。

源码地址 [python](#)

```
Line 1 # 导入 pymysql 模块
- import pymysql
```

9.1.3 使用 PyMySQL 的 connect 函数建立到 MySQL 的连接

导入 PyMySQL 模块后, 我们使用该模块中的 connect 函数建立到本机 MySQL 服务的连接:

源码地址 [python](#)

```
Line 1 # 连接MySQL. 密码请使用你自己设置的密码, shop为上一节导入的示例数据库
- conn = pymysql.connect(host="127.0.0.1", user="root", password="123456",
                        database="shop", charset="utf8")
```

> 注:

>pymysql.connect() 有很多的参数, 但是上述最基本的参数就能保证连接正常, 并且能使得中文字符正确显示。

> 关于该函数更多的参数介绍

源码地址 [python](#)

```

Line 1 host=None,# 要连接的主机地址, 本机上的 MySQL 使用 127.0.0.1
- user=None,# 用于登录的数据库用户名, 例如 root.
- password=' ',# 上述账号的相应密码
- database=None,# 要连接的数据库, 本教程使用的是来源于<SQL 基础教程>的 shop 数据
    库
5 port=0,# 端口, 一般为 3306
- unix_socket=None,# 选择是否要用 unix_socket 而不是 TCP/IP
- charset='',# 字符编码, 需要支持中文请使用"utf8"
- sql_mode=None,# Default SQL_MODE to use.
- read_default_file=None,# 从默认配置文件(my.ini 或 my.cnf)中读取参数
10 conv=None,# 转换字典
- use_unicode=None,# 是否使用 unicode 编码
- client_flag=0,# Custom flags to send to MySQL. Find potential values in
    constants.CLIENT.
- cursorclass,# 选择 Cursor 类型
- init_command=None,# 连接建立时运行的初始语句
15 connect_timeout=10,# 连接超时时间, (default: 10, min: 1, max: 31536000)
- ssl=None,# A dict of arguments similar to mysql_ssl_set()'s parameters. For now
    _the_capath_and_cipher_arguments_are_not_supported
- read_default_group=None,# Group to read from in the configuration file
- compress=None,# 不支持
- named_pipe=None,# 不支持
20 no_delay=None,
- autocommit=False,# 是否自动提交事务
- db=None,# 同 database, 为了兼容 MySQLdb
- passwd=None,# 同 password, 为了兼容 MySQLdb
- local_infile=False,# 是否允许载入本地文件
25 max_allowed_packet=16777216,# 限制 LOCAL DATA INFILE 大小
Line 26 defer_connect=False,# Don't explicitly connect on construction - wait for
    connect call.

Line 27 auth_plugin_map={},#
- read_timeout=None,
- write_timeout=None,
30 bind_address=None # 当客户有多个网络接口, 指定一个连接到主机

```

9.1.4 创建用于执行 SQL 语句的游标对象

对上述建立的连接, 使用 cursor 方法得到一个游标对象:

源码地址 [python](#)

```

Line 1 # 得到一个可以执行 SQL 语句的游标对象
- cur = conn.cursor()

```

本教程主要是讲解 SQL 查询语句, 因此这里以一个读取 MySQL 数据库的表中数据的 SQL 查询为例. 首先定义一个 SQL 查询语句, 以取回 product 表所有数据为例:

[源码地址 python](#)

```
Line 1 # 定义要执行的 SQL 语句
- sql = """
- SELECT_*_FROM_product;
- """
```

9.1.5 执行 SQL 语句

然后使用游标对象的 execute 方法在 MySQL 数据库里执行上述 SQL 查询:

[源码地址 python](#)

```
Line 1 # 执行 SQL 语句
- cur.execute(sql)
```

9.1.6 取回 SQL 语句的执行结果

完成 SQL 语句后, 由于我们上述定义的 SQL 语句是一个 SQL 查询, 因此可以接着使用游标对象的 fetchall 方法取回查询结果:

[源码地址 python](#)

```
Line 1 # 取回查询结果
- # 如果需要将取回的结果赋值给一个变量, 可以使用 data = cur.fetchall()
- cur.fetchall()
```

取回的结果是一个嵌套的元组, 且没有数据表中的列名. 上述代码正确执行后得到如下结果:

```
[13]: (('0001', 'T恤', '衣服', 1000, 500, datetime.date(2009, 9, 20)),
      ('0002', '打孔器', '办公用品', 500, 320, datetime.date(2009, 9, 11)),
      ('0003', '运动T恤', '衣服', 4000, 2800, None),
      ('0004', '菜刀', '厨房用具', 3000, 2800, datetime.date(2009, 9, 20)),
      ('0005', '高压锅', '厨房用具', 6800, 5000, datetime.date(2009, 1, 15)),
      ('0006', '叉子', '厨房用具', 500, None, datetime.date(2009, 9, 20)),
      ('0007', '擦菜板', '厨房用具', 880, 790, datetime.date(2008, 4, 28)),
      ('0008', '圆珠笔', '办公用品', 100, None, datetime.date(2009, 11, 11)))
```

图 9.2: 返回结果

9.1.7 使用 pandas 的 read_sql 函数执行 SQL 查询并取回结果为 DataFrame

如果我们想要同时取回列名, 并且想让取回的数据具有更好的结构化, 则可以使用 pandas 库的 read_sql 函数来读取检索结果:

[源码地址 python](#)

```
Line 1 # 另一种执行SQL查询并取回检索结果的方法是使用 pandas 库读取检索结果
- # 如未安装pandas可在notebook里使用"pip_install_pandas"进行安装
- import pandas as pd
```

```
- pd.read_sql(sql,conn)
5 # 如果需要将取回的结果赋值给一个变量, 可以使用 df = pd.read_sql(sql,conn)
```

此时取回的结果就是一个 pandas 中的 DataFrame 对象了, 它在 jupyter notebook 中看起来就是一张表格。

	product_id	product_name	product_type	sale_price	purchase_price	regist_date
0	0001	T恤	衣服	1000	500.0	2009-09-20
1	0002	打孔器	办公用品	500	320.0	2009-09-11
2	0003	运动T恤	衣服	4000	2800.0	None
3	0004	菜刀	厨房用具	3000	2800.0	2009-09-20
4	0005	高压锅	厨房用具	6800	5000.0	2009-01-15
5	0006	叉子	厨房用具	500	NaN	2009-09-20
6	0007	擦菜板	厨房用具	880	790.0	2008-04-28
7	0008	圆珠笔	办公用品	100	NaN	2009-11-11

图 9.3: jupyter 返回结果 1

> 注:

> 在使用 pandas 的 read_sql 函数时, 只需要建立 Python 到 MySQL 的连接就可以了, 不需要建立游标。

> 关于该函数的更多参数介绍:

源码地址 [python](#)

```
Line 1 # read_sql函数各参数的意义:
- sql # 必备参数, SQL命令字符串
- con # 连接sql数据库的engine, 我们这里使用pymysql的connect函数建立
- index_col=None # 选择某一列作为pandas对象的index
5 coerce_float=True # 将数字形式的字符串直接以float型读入
- parse_dates=None # 将数据表中datetime类型的列读取为datetime型数据, 与pd.
    to_datetime 功能类似. 可直接提供需要转换的列名然后以默认的日期形式转换, 也
    可以用字典的格式提供列名和转换的日期格式, 比如{列名A: 时间日期格式1, 列名B
    : 时间日期格式2}, 其中的时间日期格式需要是合法的格式, 例如:"%Y:%m:%H:%M:%S
    ".

- columns # 要读取的列, 基本不会用到, 因为我们在sql命令里面就可以指定需要取回的列
    .

- chunksize # 对于取回大批量数据时有用. 如果提供了一个整数值, 那么就会返回一个
    generator, 每次输出的行数就等于你指定的该参数的值.
```

9.1.8 关闭游标和数据库连接

最后, 在完成数据检索和取回数据后, 需要手动关闭数据库连接。

首先关闭游标对象:

[源码地址 python](#)

```
Line 1 # 关闭游标对象
- cur.close()
```

然后关闭数据库连接

[源码地址 python](#)

```
Line 1 # 关闭数据库连接
- conn.close()
```

9.1.9 使用函数封装数据库连接

上述流程就是使用 PyMySQL 连接到本机的 MySQL 数据库的完整过程了。

为了便于复用上述代码, 我们可以把上述流程封装为一个函数, 函数的参数为 SQL 查询语句:

[源码地址 python](#)

```
Line 1 # 封装为函数
- def conn2mysql(sql):
-     """
-     函数的参数为一个字符串类型的 SQL 语句
5    返回值为一个 DataFrame 对象
-     """
-     from pandas import read_sql
-     from pymysql import connect
-     # 连接本机上的 MySQL 服务器中的 shop 数据库
10    conn = connect(host="127.0.0.1", user="root", password="123456", database="shop",
-                     charset="utf8")
-     # 使用 pandas 的 read_sql 函数执行 SQL 语句并取回检索结果
-     df=read_sql(sql,conn)
-     # 关闭数据库连接
-     conn.close()
-     return df
15
```

使用上述函数执行 sql

[源码地址 python](#)

```
Line 1 # 定义要执行的 SQL 查询
- sql = """
- SELECT *
- FROM product;
5 """
- # 执行 sql 查询并取回查询结果
- df = conn2mysql(sql)
- # 查看取回的结果
- df
```

	product_id	product_name	product_type	sale_price	purchase_price	regist_date
0	0001	T恤	衣服	1000	500.0	2009-09-20
1	0002	打孔器	办公用品	500	320.0	2009-09-11
2	0003	运动T恤	衣服	4000	2800.0	None
3	0004	菜刀	厨房用具	3000	2800.0	2009-09-20
4	0005	高压锅	厨房用具	6800	5000.0	2009-01-15
5	0006	叉子	厨房用具	500	NaN	2009-09-20
6	0007	擦菜板	厨房用具	880	790.0	2008-04-28
7	0008	圆珠笔	办公用品	100	NaN	2009-11-11

图 9.4: jupyter 返回结果 2

将数据库中的查询取回为 pandas 对象后, 就可以在 Python 中使用各种工具进行进一步的处理了。