课程：Mosh_完全掌握SQL【笔记】

上一级目录：数据科学旅行地图

关于 '课程：Mosh_完全掌握SQL【笔记】'

http://codewithmosh.com 网站的 Complete SQL Mastery（完全掌握SQL）课程可能是最清晰易懂又完整全面的SQL视频教程了。相关资源如下：

- 课程官网连接（请尽量支持正版！）
- B站搬运-版本1（按小节分p，中文字幕更新到第十一章，置顶评论有课程资料地址）
- B站搬运-版本2（按整章分p，有完整中文字幕，置顶评论有课程资料地址）

此系列是自己学习该课程的分章节笔记，主要是每节课代码的复现，也包含一些自己的总结感悟。

主要目的是总结、记录和方便以后查阅，当然也欢迎任何讨论、交流。

数据概要

查看数据概要，大致了解一下课程所用到数据的含义及其相互关系，这对理解课程有极大帮助

目录

第一部分：基础——增删查改

【第一章】做好准备 Getting Started (时长25分钟)
【第二章】在单一表格中检索数据 Retrieving Data From a Single Table (时长53分钟)
【第三章】在多张表格中检索数据 Retrieving Data From Multiple Tables (时长1小时2分)
【第四章】插入、更新和删除数据 Inserting, Updating, and Deleting Data (时长42分钟)

第二部分：基础进阶——汇总、复杂查询、内置函数

【第五章】汇总数据 Summarizing Data (时长33分钟)
【第六章】编写复杂查询 Writing Complex Query (时长45分钟)
【第七章】MySQL的基本函数 Essential MySQL Functions (时长33分钟)

第三部分：提高效率——视图、存储过程、函数

【第八章】视图 Views (时长18分钟)
【第九章】存储过程 Stored Procedures (时长48分钟)

第四部分：高阶主题——触发器、事件、事务、并发

【第十章】触发器和事件 Triggers and Events (时长22分钟)
【十一章】事务和并发 Transactions and Concurrency (时长49分钟)

第五部分：脱颖而出——数据类型、设计数据库、索引、保护

【十二章】数据类型 Data Types (时长35分钟)
【十三章】设计数据库 Designing Databases (时长1时30分)
【十四章】高效的索引 Indexing for High Performance (时长58分钟)
【十五章】保护数据库 Securing Databases (时长20分钟)

学习总结

我是纯小白，这算是考虑转行数据分析后第一个认真学完了的课程，自己的感觉是，课程质量很高而且非常新手友好，言简意赅、清晰易懂，也比较全面。不过，感觉后面的高级主题只是蜻蜓点水让你了解一下，而且也有些重要的内容如窗口函数（听别人说很重要）没有涉及到。而且，很多内容明显感觉虽然做到了“浅出”但还不够“深入”，大概知道怎么用但不清楚原理，以后很难举一反三灵活使用。所以，很有必要通过更多课程书籍进一步深入学习以及更多的练习和实际使用来熟悉巩固。

所谓 "Complete SQL Mastery"（完全掌握SQL）还是有些夸大了，但这课就帮助新手入门SQL而言真的很棒，课程设计得很好，学起来循序渐进很舒服

视频总时长只有11h，但自己学的很慢，再加上有些笔记做的过于冗长（其实没必要），实际竟花了差不多20倍的时间，从 2020-09-03 断断续续学到 2020-10-06，共计约 220 小时的学习时间，效率有点太低了

不过，我的数据科学之旅总算是踏踏实实地踏出第一步了，哈哈~

https://zhuanlan.zhihu.com/p/222865842

发布于 2020-09-07

专栏介绍和总目录
数据概要

数据概要

课程总共用到四个数据库，分别是：

1. sql_store （商店数据库）
2. sql_invoicing （发票记录数据库）
3. sql_hr （人力资源数据库）
4. sql_inventory （存货数据库）

其中，主要是前两个数据库用的比较多，结构也复杂一些，后两个数据库只是在讲特定主题时用到过一两次，结构也比较简单

下面对各数据库进行依次描述：

1. sql_store

sql_store（商店数据库）是课程前半段用的最多的一个数据库，其结构如图所示，可以看作以orders表（订单表）为核心，然后……

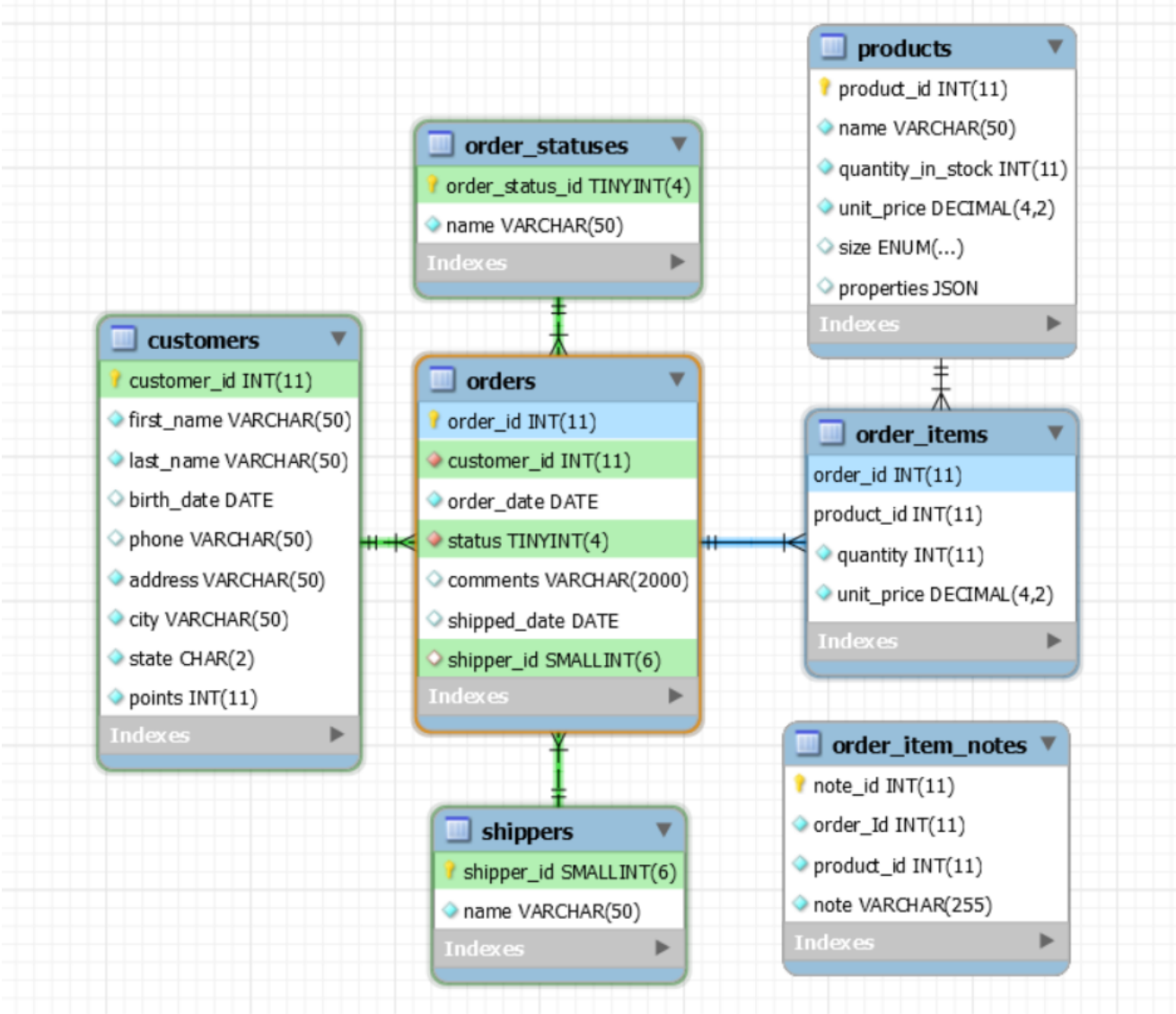
1. 通过customer_id（顾客编号）与customers表（顾客表）相联
2. 通过status_id（状态编号）与order_statuses表（订单状态表）相联
3. 通过shipper_id（运货商编号）与shippers表（运货商表）相联
4. 通过order_id（订单编号）与order_items表（订单项目表）相联

其中，customers表提供了每个顾客的详细信息，order_statuses表提供了每种订单状态编号的含义（包括：processed 已处理、shipped 已寄出、delivered 已送达），shippers表提供了每个运货商的详细信息，order_items表详细列明了每个订单包含的具体产品项目。

这几个表（顾客表、订单状态表、运货商表、订单项目表）相当于充当了订单表的"查询表"（lookup table），为订单提供了各个角度的更详细信息，与在每个订单都记录顾客和运货商等详细信息相比，这样把顾客运货商等详细信息单独分离出来作为查询表可以减少数据冗余和重复性，这在第十三章设计数据库里会讲到

orders_items（订单项目表）里的商品都是以product_id（商品编号）的形式存在的，其通过product_id与products表（商品表）相联可查询商品的详细信息，products表是orders_items表的查询表

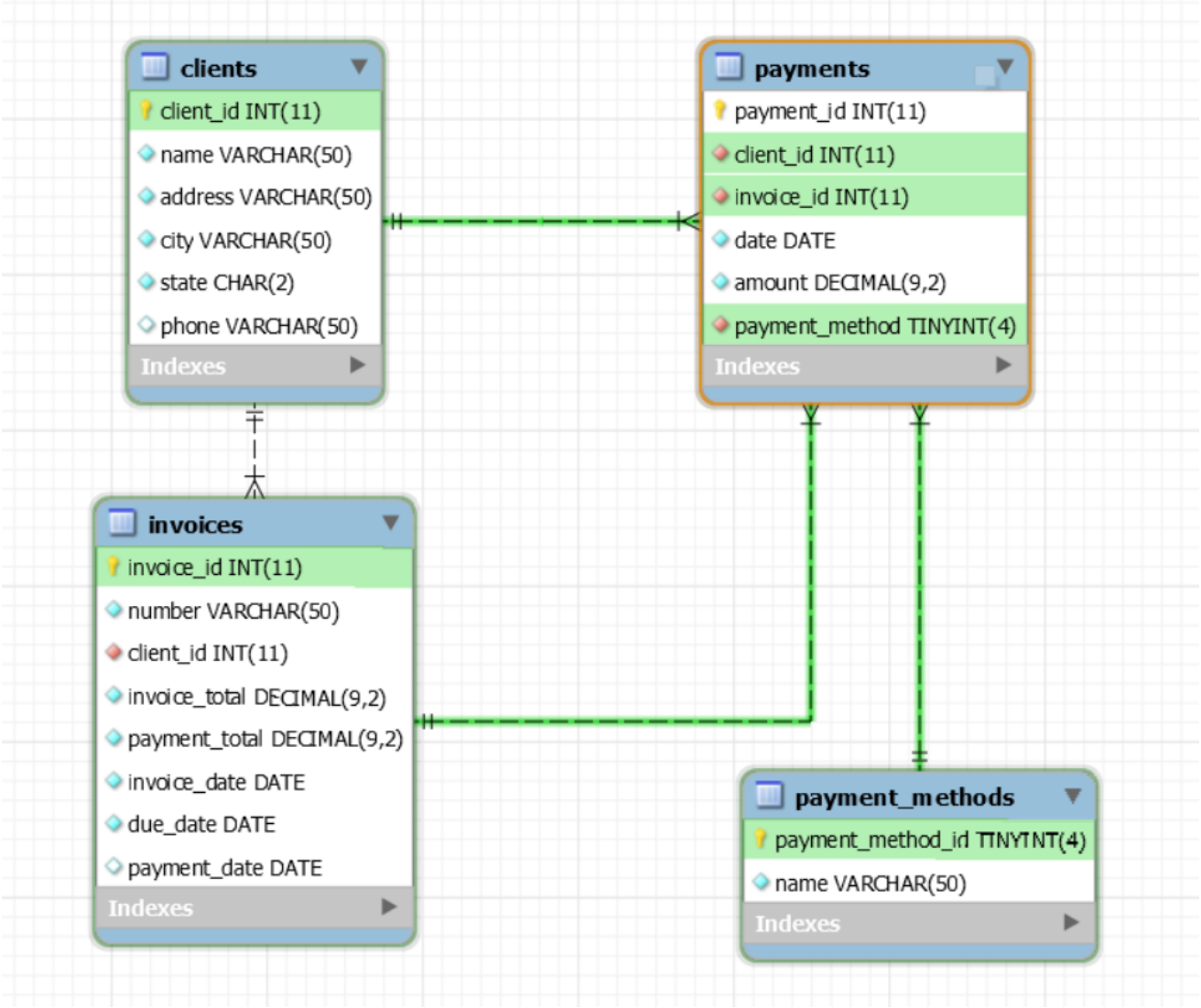
还有一个order_item_notes表（订单项目备注表）本来也该是orders_items表的查询表的，但实际上，如图可见，并没有和orders_items表联系起来，这是Mosh为了课程讲解的需要故意保留的一个设计错误



2. sql_invoicing

sql_invoicing（发票记录数据库）是课程后半段用的最多的一个数据库，其中最重要的三张表clients表（客户表）invoices表（发票记录表）payments表（支付记录表）是通过client_id（客户编号）和invoice_id（发票记录编号）来相互联系的。我个人的理解是：客户表记录客户的详细信息，发票记录表记录的是某次交易的应付款总额（一次交易对应一次发票记录），而支付记录表记录的是客户为特定发票进行付款的记录，注意之后课程中会常常要将特定发票的应付款总额与已付款总额相减来得到该发票的balance（剩余欠款）

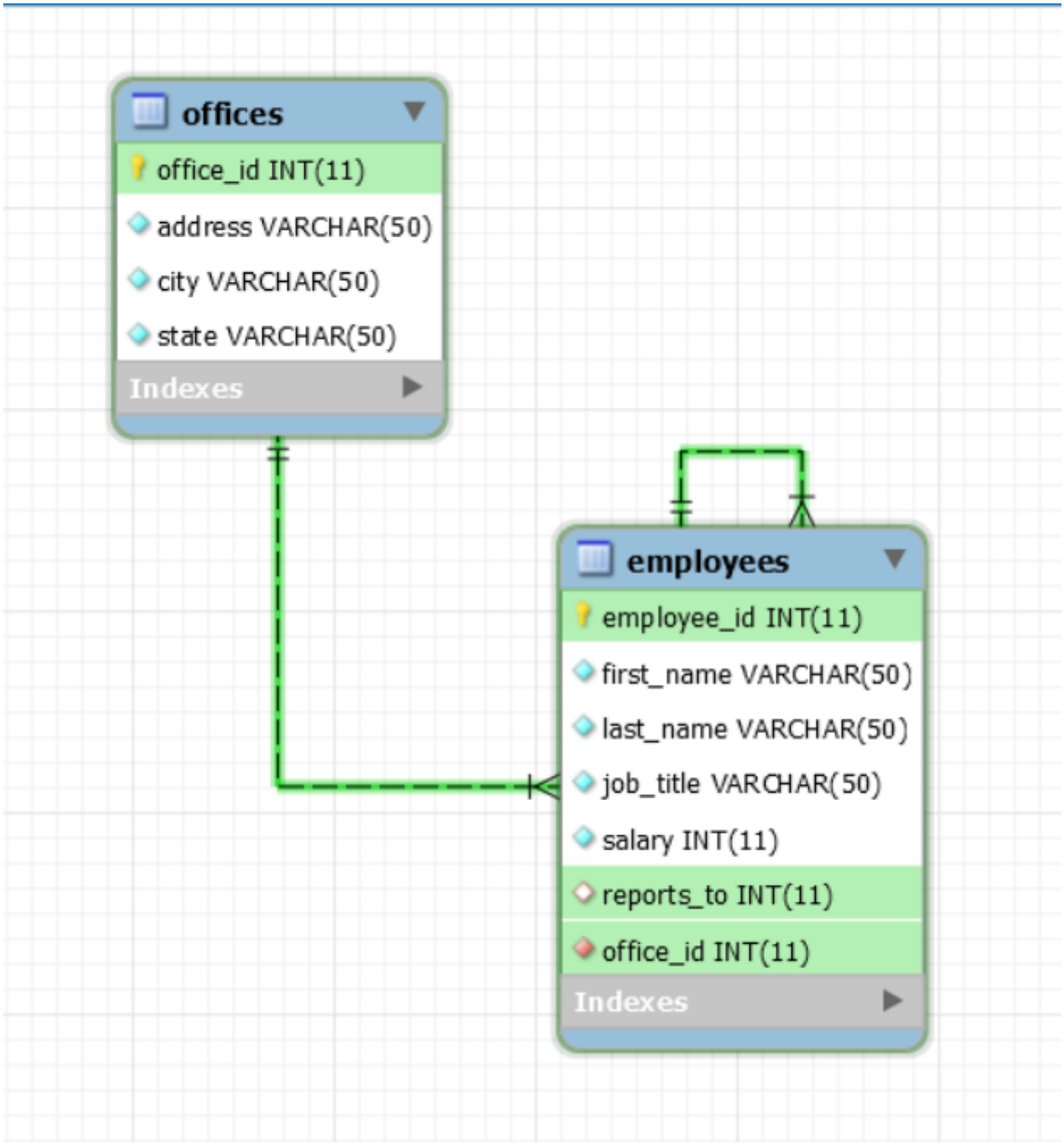
另外还有一个payment_methods表（付款方式表），通过payment_method_id与payments表相联，作为查询表提供各种付款方式的详细信息



3. sql_hr

sql_hr（人力资源数据库）结构很简单，就两个表，offices表（办公室表）和employees表（雇员表），这两张表通过office_id（办公室编号）相联，offices表充当查询表为employees表提供雇员所在办公室的详细信息

值得注意的是employees表本身的一个特性：它的reports_to（向谁汇报）字段引用了该表本身的emploee_id（雇员编号字段），毕竟，雇员的上级也是该公司的雇员。正因为这个特性，这个表在之后讲解 “self join 自连接”时被当作素材。



4. sql_inventory

sql_inventory （存货数据库）只有一张products表（商品表），其实和sql_store里商品表是一样的，这个数据库在整个课程中只在讲解 “跨数据库连接” 时用到了一次

专栏介绍和总目录
数据概要

第一部分：基础——增删查改

【第一章】做好准备

Getting Started (时长25分钟)

1. 介绍

Introduction (0:18)

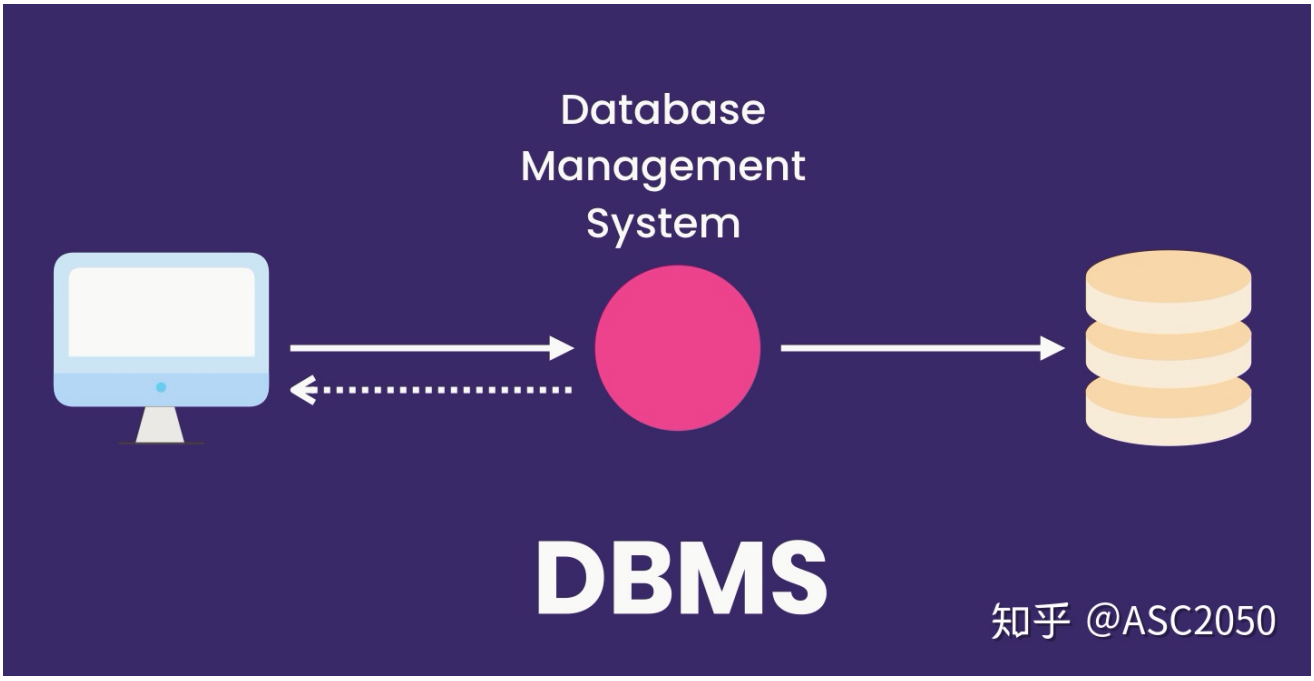
.....

2. 什么是SQL

What is SQL (3:24)

- A DATABASE is a collection of data stored in a format that can easily be accessed
  数据库是一个以易访问格式存储的数据集合

- 为了管理我们的数据库 我们使用一种叫做数据库管理系统（DBMS, Database Management System）的软件。我们连接到一个DBMS然后下达查询或者修改数据的指令，DBMS就会执行我们的指令并返回结果



- 有关系型和非关系型两类数据库，在更流行的关系型数据库中，我们把数据存储在通过某些关系相互关联的数据表中，每张表储存特定的一类数据，这正是关系型数据库名称的由来。（如：顾客表通过顾客id与订单表联系，订单表又通过商品id与商品表联系）
- SQL（Structured Query Language，结构化查询语言）是专门用来处理（查询、修改）关系型数据库的标准语言
- 不同关系型数据库管理系统语法略有不同，但都是基于标准SQL，本课使用最流行的开源关系型数据库管理系统，MySQL

3. MySQL Mac版本安装

Installing MySQL on Mac (4:58)

.....

4. MySQL Windows版本安装

Installing MySQL on Windows (5:20)

注意

- Mosh是用Installer一次性安装了MySQL和workbench，若已安装MySQL而没有workbench的可 单独安装workbench
- 注意**以下四项**的区别：

  1. connection（连接，如：Local instance）
  2. host（主机，如：127.0.0.1 或 localhost，应该指代的是用于连接的电脑）
  3. port（端口，默认：3306）
  4. user（用户，如：root，指代帐户，有相应密码）

5. 创建数据库

Creating the Databases (8:32)

注意

如果MySQL版本较低导致导入 create-databases.sql 时出现collate规则问题而报错，可以用记事本打开 create-databases.sql 将 utf8mb4_0990_ai_ci 全部替换为 utf8mb4_general_ci 并保存，再次导入就能顺利运行了。（注：修改的是排序规则（collation））

背景

查看 数据概要，大致了解一下课程所用数据的含义和相互联系，这对理解课程有极大帮助

6. 你会学到什么

What You'll Learn (2:31)

见 专栏介绍和总目录

【第二章】在单一表格中检索数据

Retrieving Data From a Single Table (时长53分钟)

*补充：在Jupyter Notebook中运行SQL

如何在Jupyter Notebook中运行SQL?

Python操作MySQL之SQLAlchemy

```
# ! pip install ipython-sql
# ! pip install pymysql
```

```
import pymysql
import sqlalchemy
```

```
sqlalchemy.create_engine(
    'mysql+pymysql://root:101994410@localhost/sql_store')
# 'mysql+pymysql://
# <username>:<password>@<host>/<dbname>[?<options>]'
```

```
Engine(mysql+pymysql://root:***@localhost/sql_store)
```

```
%load_ext sql
```

```
%sql mysql+pymysql://root:101994410@localhost/sql_store
```

```
%%sql
SELECT *  # 1, 2
FROM customers
WHERE customer_id < 4
ORDER BY first_name
```

```
 * mysql+pymysql://root:***@localhost/sql_store
3 rows affected.
```

customer_id	first_name	last_name	birth_date	phone	address	city	state	points
1	Babara	MacCaffrey	1986-03-28	781-932-9754	0 Sage Terrace	Waltham	MA	2273
3	Freddi	Boagey	1985-02-07	719-724-7869	251 Springs Junction	Colorado Springs	CO	2967
2	Ines	Brushfield	1986-04-13	804-427-9456	14187 Commercial Trail	Hampton	VA	947

1. 选择语句

The SELECT Statement (5:50)

导航

第1节先看一下选择语句整体是什么样子，本章后面的小节会分别讲解其中各子句的具体写法

实例

```sql
USE sql_store;

SELECT * / 1, 2  -- 纵向筛选列
FROM customers  -- 选择表
WHERE customer_id < 4  -- 横向筛选行
ORDER BY first_name  -- 排序

-- 单行注释

/*
多行注释
*/
```

2. 选择子句

The SELECT Clause (8:48)

小结

SELECT 是列/字段选择语句，可选择列，列间数学表达式，特定值或文本，可用AS关键字设置列别名（AS可省略），注意DISTINCT关键字的使用。

注意

SQL会完全无视大小写（任何情况下的大小写）、多余的空格（超过一个的空格）、缩进和换行，SQL语句间完全由分号 ; 分割，用缩进、换行等只是为了代码看着更美观结构更清晰，这些与Python很不同，要注意。

实例

```sql
USE sql_store;

SELECT
    DISTINCT last_name,
    first_name,
    points,
    (points + 70) % 100 AS discount_factor/'discount factor'
FROM customers
```

练习

单价涨价10%作为新单价

```sql
SELECT
    name,
    unit_price,
    unit_price * 1.1 'new price'
    -- AS 可省，空格后跟别名就行，可看作是将列变量及其数学运算之后的第一个空格识别为AS
FROM products
```

3. WHERE子句

The WHERE Clause (5:17)

小结

WHERE 是行筛选条件，实际是一行一行/一条条记录依次验证是否符合条件，进行筛选

导航

3~9 节讲的都是写WHERE子句中条件的不同方法，这一节（第3节）主要讲比较运算，第4节讲逻辑运算 AND、OR、NOT，5~9可看作都是在讲特殊的比较运算（是否符合某种条件）：IN、BETWEEN、LIKE、REGEXP、IS NULL。

所以总的来说WHERE条件就是数学→比较→逻辑运算，逻辑层次和执行优先级也是按照这三个的顺序来的。

实例

```
USE sql_store;

SELECT *
FROM customers
WHERE points > 3000
/WHERE state != 'va'  -- 'VA'/'va'一样
```

比较运算符 `> < = >= <= !=/<>`，注意等于是一个等号而不是两个等号

也可对日期或文本进行比较运算，注意SQL里日期的标准写法以及其需要用引号包裹这一点

```
WHERE birth_date > '1990-01-01'
```

练习

今年（2019）的订单

```
USE sql_store;

select *
from orders
where order_date > '2019-01-01'
-- 有更一般的方法，不用每年改代码，之后教
```

4. AND, OR, NOT运算符

The AND, OR and NOT Operators (6:52)

小结

用逻辑运算符AND、OR、NOT对（数学和）比较运算进行组合实现多重条件筛选
执行优先级：数学→比较→逻辑

实例

```
USE sql_store;

SELECT *
FROM customers
WHERE birth_date > '1990-01-01' AND points > 1000
/WHERE birth_date > '1990-01-01' OR
      points > 1000 AND state = 'VA'
```

AND优先级高于OR，但最好加括号，更清晰

```
WHERE birth_date > '1990-01-01' OR
      (points > 1000 AND state = 'VA')
```

NOT的用法

```
WHERE NOT (birth_date > '1990-01-01' OR points > 1000)
```

去括号等效转化为

```
WHERE birth_date <= '1990-01-01' AND points <= 1000
```

练习

订单6中总价大于30的商品

```
USE sql_store;

SELECT *
FROM order_items
WHERE order_id = 6 AND quantity * unit_price > 30
```

注意优先级：数学→比较→逻辑
select子句，where子句以及后面的ORDER BY子句等都能用列间数学表达式

5. IN运算符

The IN Operator (3:03)

小结

用IN运算符将某一属性与多个值（一列值）进行比较
实质是多重相等比较运算条件的简化

案例

选出'va'、'fl'、'ga'三个州的顾客

```
USE sql_store;

SELECT *
FROM customers
WHERE state = 'va' OR state = 'fl' OR state = 'ga'
```

不能 `state = 'va' OR 'fl' OR 'ga'` 因为数学和比较运算优先于逻辑运算，加括号 `state = ('va' OR 'fl' OR 'ga')` 也不行，**逻辑运算符只能连接布林值。**

用 IN 操作符简化该条件

```
WHERE state IN ('va', 'fl', 'ga')
```

可加NOT

```sql
WHERE state NOT IN ('va', 'fl', 'ga')
```

这里可用NOT的原因：可以这么看，IN语句 `IN ('va', 'fl', 'ga')` 是在进行一种是否符合条件的判断，可看作是一种特殊的比较运算，得到的是一串逻辑值，故可用NOT进行取反

练习

库存量刚好为49、38或72的产品

```sql
USE sql_store;

select * from products
where quantity_in_stock in (49, 38, 72)
```

6. BETWEEN运算符

The BETWEEN Operator (2:12)

小结

范围型条件

注意

- **用AND而非括号**
- **闭区间，包含两端点**
- 也可用于日期，**毕竟日期本质也是数值**，日期也有大小（早晚），可比较运算
- 同 IN 一样，BETWEEN 本质也是一种特定的 多重比较运算条件 的简化

案例

选出积分在1k到3k的顾客

```sql
USE sql_store;

select * from customers
where points >= 1000 and points <= 3000
```

等效简化为：

```sql
WHERE points BETWEEN 1000 AND 3000
```

注意两端都是包含的
不能写作BETWEEN (1000, 3000)！别和IN的写法搞混

练习

选出90后的顾客

```sql
SELECT * FROM customers
WHERE birth_date BETWEEN '1990-01-01' AND '2000-01-01'
```

7. LIKE运算符

The LIKE Operator (5:37)

小结

模糊查找，查找具有某种模式的字符串的记录/行

注意

- 过时用法（但有时还是比较好用），下节课的正则表达式更灵活更强大
- 注意和正则表达式一样都是用引号包裹表示字符串

```
USE sql_store;

select * from customers
where last_name like 'brush%' / 'b____y'
```

引号内描述想要的字符串模式，注意SQL任何情况都是不区分大小写的

- % 任何个数（包括0个）的字符（类似通配符里的 *）
- _ 单个字符（类似通配符里的 ?）

练习

分别选择满足如下条件的顾客：

1. 地址包含 'TRAIL' 或 'AVENUE'
2. 电话号码以 9 结束

```
USE sql_store;

select *
from customers
where address like '%Trail%' or
      address like '%avenue%'
```

执行优先级在逻辑运算符之前，**毕竟IN BETWEEN LIKE 本质可看作是比较运算符的简化，应该和比较运算同级**，数学→比较→逻辑，始终记住这个顺序，上面这个如果用正则表达式会简单得多

LIKE的判断结果也是个TRUE/FASLE的问题，任何逻辑值/布林值都可前置NOT来取反：

```
where phone like '%9'
where phone not like '%9'
```

之前的 BETWEEN 也可以取反：NOT BETWEEN

8. REGEXP运算符

The REGEXP Operator (9:21)

小结

正则表达式，在搜索字符串方面更为强大，可搜索更复杂的模板

实例

```sql
USE sql_store;

select * from customers
where last_name like '%field%'
```

等效于：

```sql
where last_name 【regexp】 'field'
```

正则表达式可以组合来表达更复杂的字符串模式

```sql
where last_name regexp '^mac|field$|rose'
where last_name regexp '[gi]e|e[fmq]' -- 查找含ge/ie或ef/em/eq的
where last_name regexp '[a-h]e|e[c-j]'
```

正则表达式总结：

符号	意义
^	beginning
$	end
[abc]	含列表中字幕，即abc的
[^abc]	不含列表中字母的，即不含abc的
[a-f]	含a到f的
|	logical or

（正则表达式用法还有更多，自己去查）

练习

分别选择满足如下条件的顾客：

1. first names 是 ELKA 或 AMBUR
2. last names 以 EY 或 ON 结束
3. last names 以 MY 开头 或包含 SE
4. last names 包含 BR 或 BU

```sql
select *
from customers
where first_name regexp 'elka|ambur'
/where last_name regexp 'ey$|on$'
/where last_name regexp '^my|se'
/where last_name regexp 'b[ru]'/'br|bu'
```

注意

- like 和 regexp 的模糊搜索里对文本模式的表达依旧不区分大小写
- like 和 regexp 本质上也是条件判断，结果也是布林值，自然也可以加 NOT 取反：NOT LIKE、NOT REGEXP

9. IS NULL运算符

The IS NULL Operator (2:26)

小结

找出空值，找出有某些属性缺失的记录

案例

找出电话号码缺失的顾客，也许发个邮件提醒他们之类

```
USE sql_store;

select * from customers
where phone is null/is not null
```

注意是 IS NULL 和 **IS NOT NULL**（而非 NOT IS NULL），这里 NOT 更**符合英语语法**的放在了be动词后

练习

找出还没发货的订单（在线商城管理员的常见查询需求）

```
USE sql_store;

select * from orders
where shipper_id is null
```

回顾

3~9 节全在讲WHERE子句中**条件**的具体写法：

- 第3节：比较运算
- 第4节：逻辑运算 `AND`、`OR`、`NOT`
- 5~9节：特殊的比较运算（是否符合某种条件）：`IN` 和 `BETWEEN`、`LIKE` 和 `REGEXP`、`IS NULL`

所以总的来说WHERE条件就是

> 数学运算 → 比较运算（包括特殊的比较运算）→ 逻辑运算（**NOT** → **OR** → **AND**）

逻辑层次和执行优先级也是按照这三个的顺序来的。

10. ORDER BY子句

The ORDER BY Clause (7:06)

小结

排序语句，和 `SELECT ……` 很像：
可多列，**可包括没选择的列（MySQL特性）**，不仅可以是列，**也可是列间的数学表达式以及之前定义好的别名列（MySQL特性）**，任何一个排序依据列后面都**可选加 DESC**

总之，**MySQL 里 ORDER BY 子句里可选排序依据的灵活性极大**

注意

最好别用 ORDER BY 1，2（表示以 SELECT …… 选中列中的第1、2列为排序依据） 这种隐性依据，因为SELECT选择的列一变就容易出错，还是**显性地写出列名作为排序依据**比较好

> 注：workbench 中扳手图标打开表格的设计模式，查看或修改表中各列（属性）。**省略排序语句的话会默认按主键排序**

实例

```
USE sql_store;

select name, unit_price * 1.1 + 10 as new_price
from products
order by new_price desc, product_id
/order by unit_price
/order by unit_price * 0.9
```

练习

订单2的商品按总价降序排列:

1. 可以以总价的数学表达式为排序依据

```
select * from order_items
where order_id = 2
order by quantity * unit_price desc
```

2. 或先定义总价别名，在以别名为排序依据

```
select *, quantity * unit_price as total_price
from order_items
where order_id = 2
order by total_price desc
```

11. LIMIT子句

The LIMIT Clause (3:26)

小结

限制返回结果的记录数量，**“前N个”或“跳过M个后的前N个”**

实例

```
USE sql_store;

select * from customers
limit 3 / 300 / 6, 3
```

6, 3 表示跳过前6个，取第7~9个，**6是偏移量**，
如：网页分页 每3条记录显示一页 第3页应该显示的记录就是 limit 6, 3

练习

找出积分排名前三的死忠粉

```sql
USE sql_store;

select *
from customers
(where)
order by points desc
limit 3
```

回顾

SELECT 语句完结了，里面的子句顺序固定要记牢，顺序乱会报错
`select from where + order by limit`
纵选列，确定表，**横选行（各种条件写法和组合要清楚熟悉）**，**最后再进行排序和限制**

【第三章】在多张表格中检索数据

Retrieving Data From Multiple Tables (时长1小时2分)

常常需要在多张表中检索数据，这一章讲的就是这个

1. 内连接

Inner Joins (8:26)

小结

各表分开存放是为了减少重复信息和方便修改，需要时可以根据相互之间的关系连接成相应的合并详情表以满足相应的查询需求。FROM JOIN ON 语句就是告诉sql：
将哪几张表以什么基础连接/合并起来。

这种多表合并的语句可分两部分看，从后往前看：

1. 后面的 `from` 表A `join` 表B `on` AB的关系，就是以某些相关联的列为依据（关系型数据库就是这么来的）进行多表合并得到所需的详情表
2. 前面的 select 就是在合并详情表中找到所需的列

关于表别名

之前在SELECT中给选定的列加别名主要是为了得到**更有意义的列名**，这里在 FROM JOIN 中给表加别名主要是为了**简化**

```sql
USE sql_store;

select
    order_id,
    o.customer_id,
    ……
```

o.是别名，是之后from语句里定义的，试过，不用确实会报错，因为两个表都有这个customer_id列，只写customer_id的话会报错：ambiguous，必须指定一个表的customer_id，这里指定任意一个表的都行，因为正是按相等的customer_id来连接两个表的。**总之选择多张表里都有的同名列时必须要加上表名前缀来明确所选。Mosh 个人喜欢在多表查询时将SELECT里所有的列名都加上表名前缀，因为这样更清晰，也能保证不会出错**

```
    ……
    first_name,
    last_name
from orders as o
```

用了别名后其他地方（包括前面select语句，从实际执行顺序可以理解这一点）只能用别名，用全名会报错。另外就像在select里一样，这个as也是可省略的。

```
……
(inner) join customers c
    on o.customer_id = c.customer_id
```

因为表别名需要在FROM JOIN语句中确定，所以最好先SELECT * FROM 选择全部，等写好了FROM JOIN ON等后面的语句，即取好了别名并确定了选哪些表的以及怎么连接它们后，**再回头去SELECT里细化明确需要的列。（即按执行顺序写）**

练习

通过product_id结合orders_items和products:

```
USE sql_store;

select *
/select oi.*, p.name
/select
    order_id,
    oi.product_id,
    name,
    quantity,
    oi.unit_price
    ……
```

两个表都有unit_price，故要指明要的是哪一个，有**两个单价**是因为单价会变，订单项目表里的是下订单时的实际单价，产品表里的单价是目前的价格，若计算总价该用前者，别搞错了

```
……
from order_items oi
join products p
    on oi.product_id = p.product_id  -- 连接的基础
```

2. ★跨数据库连接（合并）

Joining Across Databases (2:47)

小结

有时需要选取不同库的表的列，其他都一样，就只是FROM JOIN里对于非现在正在用的库的表要加上库名前缀而已。依然可用别名来简化

实例

```sql
use sql_store;

select * from order_items oi
join 【sql_inventory.】products p
    on oi.product_id = p.product_id
```

或

```sql
use sql_inventory;

select * from sql_store.order_items oi
join products p
    on oi.product_id = p.product_id
```

3. 自连接

Self Joins (4:13)

小结

一个表和它自己合并。如下面的例子，员工的上级也是员工，所以也在员工表里，所以要想得到的有员工和他的上级信息的合并表，**就要员工表自己和自己合并，用两个不同的表别名即可实现。**这个例子中只有两级，但也可用类似的方法构建多层级的组织结构。

案例

```sql
USE sql_hr;

select
    e.employee_id,
    e.first_name,
    m.first_name as manager
……
```

自合并必然每列都要加表前缀，因为每列都同时在两张表中出现。另外，两个first_name列有歧义，注意将最后一列改名为manager使得结果表更易于理解

```sql
……
from employees e
join employees m
    on e.reports_to = m.employee_id
```

4. 多表连接

Joining Multiple Tables (6:46)

小结

`FROM` 一个核心表A，用多个 `JOIN …… ON ……` 分别通过不同的连接关系连接不同的表B、C、D……，通常是让表B、C、D……为表A提供更详细的信息从而合并为一张详情合并版A表，即：

```
FROM  A
    JOIN B ON AB的关系
    JOIN C ON AC的关系
    JOIN D ON AD的关系
    ……
```

将得到一个合并了BCD……等表详细信息的详情合并版A表

真实工作场景中有时甚至要合并十多张表

案例1

订单表同时连接顾客表和订单状态表，合并为有顾客和状态详情信息的详细订单表

```sql
USE sql_store;

SELECT
    o.order_id,
    o.order_date,
    c.first_name,
    c.last_name,
    os.name AS status
FROM orders o
JOIN customers c
    ON o.customer_id = c.customer_id
JOIN order_statuses os
    ON o.status = os.order_status_id
```

案例2

同理，支付记录表连接顾客表和支付方式表形成顾客支付记录详情表

```sql
USE sql_invoicing;

SELECT
    p.invoice_id,
    p.date,
    p.amount,
    c.name,
    pm.name AS payment_method
FROM payments p
JOIN clients c
    ON p.client_id = c.client_id
JOIN payment_methods pm
    ON p.payment_method = pm.payment_method_id
```

5. 复合连接条件

Compound Join Conditions (3:41)

小结

像订单项目（order_items）这种表，订单id和产品id合在一起才能唯一表示一条记录，这叫**复合主键**，设计模式下也可以看到两个字段都有PK标识，订单项目备注表（order_item_notes）也是这两个复合主键，因此他们两合并时要用复合条件：`FROM 表1 JOIN 表2 ON 条件1 【AND】 条件2`

案例

将订单项目表和订单项目备注表合并

```
USE sql_store;

SELECT *
FROM order_items oi
JOIN order_item_notes oin
    ON oi.order_Id = oin.order_Id
    AND oi.product_id = oin.product_id
```

6. 隐含连接语法

Implicit Join Syntax (2:20)

小结

就是用 FROM WHERE 取代 FROM JOIN ON

注意

尽量别用，因为若忘记WHERE条件筛选语句，不会报错但会得到**交叉合并（cross join）**结果：即10条order会分别与10个customer结合，得到100条记录。最好使用显性合并语法，因为会强制要求你写合并条件ON语句，不至于漏掉。

案例

合并顾客表和订单表

```
USE sql_store;

SELECT *
FROM orders o
JOIN customers c
    ON o.customer_id = c.customer_id
```

隐式合并语法

```
SELECT *
【FROM orders o, customers c
WHERE o.customer_id = c.customer_id】
```

注意from子句里的逗号，就像select多条列用逗号隔开一样，**from多个表也用逗号隔开，此时若忘记WHERE条件筛选语句则得到这几张表的交叉合结果**

7. 外连接

Outer Joins (6:27)

小结

- `(INNER) JOIN` 结果只包含两表的交集，注意""广播（broadcast）"效应
- `LEFT/RIGHT (OUTER) JOIN` 结果里除了交集，还包含只出现在左/右表中的记录

案例

合并顾客表和订单表

```sql
USE sql_store;

SELECT
    c.customer_id,
    c.first_name,
    o.order_id
FROM customers c
JOIN orders o
    ON o.customer_id = c.customer_id
ORDER BY customer_id
```

这样是INNER JOIN，只展示有订单的顾客（及其订单），也就是两张表的交集，但注意这里因为一个顾客可能有多个订单，所以**INNER JOIN以后顾客信息其实是是广播了的，即一条顾客信息被多条订单记录共用**，当然 这是广播（broadcast）是另一个问题，这里关注的重点是INNER JOIN的结果确实是两表的交集，是那些同时有顾客信息和订单信息的记录。

若要展示全部顾客（及其订单，如果有的话），要**改用LEFT (OUTER) JOIN，结果相较于 (INNER) JOIN多了没有订单的那些顾客**，即只有顾客信息没有订单信息的记录

当然，也可以调换左右表的顺序(即调换FROM和JOIN的对象)再RIGHT JOIN，即：

```sql
FROM orders o
    RIGHT JOIN customers c
    ON o.customer_id = c.customer_id
```

若要展示全部订单（及其顾客），要改用 RIGHT JOIN，结果相较于 `INNER JOIN` 多了没有顾客的那些订单，即只有订单信息没有顾客信息的记录。（注：因为这里所有订单都有顾客，所以这里 RIGHT JOIN 结果和 INNER JOIN 一样）

练习

展示各产品在订单项目中出现的记录和销量，也要包括没有订单的产品

```sql
SELECT
    p.product_id,
    p.name, -- 或者直接name
    oi.quantity -- 或者直接quantity
FROM products p
LEFT JOIN order_items oi
    ON p.product_id = oi.product_id
```

8. 多表外连接

Outer Join Between Multiple Tables (6:18)

小结

与内连接类似，我们可以对多个表（3个及以上）进行外连接，**最好只用JOIN和LEFT JOIN**

案例

查询顾客、订单和发货商记录，要包括所有顾客（包括无订单的顾客），也要包括所有订单（包括未发出的）

```sql
USE sql_store;

SELECT
    c.customer_id,
    c.first_name,
    o.order_id,
    sh.name AS shipper
FROM customers c
LEFT JOIN orders o
    ON c.customer_id = o.customer_id
LEFT JOIN shippers sh
    ON o.shipper_id = sh.shipper_id
ORDER BY customer_id
```

最佳实践

虽然可以调换顺序并用RIGHT JOIN，但作为最佳实践，最好调整顺序并统一只用 (INNER)JOIN 和 LEFT JOIN（总是左表全包含），这样，当要合并的表比较多时才方便书写和理解而不易混乱

练习

查询 订单 + 顾客 + 发货商 + 订单状态，包括所有的订单（包括未发货的），其实就只是前两个优先级变了一下，是要看全部订单而非全部顾客了

```sql
USE sql_store;

SELECT
    o.order_id,
    o.order_date,
    c.first_name AS customer,
    sh.name AS shipper,
    os.name AS status
FROM orders o
JOIN customers c
    ON o.customer_id = c.customer_id
LEFT JOIN shippers sh
    ON o.shipper_id = sh.shipper_id
JOIN order_statuses os
    ON o.status = os.order_status_id
```

订单必有顾客和状态，所以这两个加不加LEFT效果一样
但订单不一定发货了，即不一定有发货商，所以这个必须LEFT JOIN，否者会筛掉没发货的订单

9. 自我外部连接

Self Outer Joins (2:12)

小结

就用前面那个员工表的例子来说，就是用LEFT JOIN让得到的 员工-上级 合并表**也包括老板本人（上级为空）**

```sql
USE sql_hr;

SELECT
    e.employee_id,
    e.first_name,
    m.first_name AS manager
FROM employees e
【LEFT JOIN】 employees m  -- 包含所有雇员（包括没有report_to的老板本人）
    ON e.reports_to = m.employee_id
```

10. USING子句

The USING Clause (5:22)

小结

当作为合并条件 join condition 的列在两个表中有**相同列名**时，可用 `USING (……, ……)` 取代 `ON …… AND ……` 予以简化，内/外连接均可如此简化。

注意

一定注意**USING后接的是括号**，特容易搞忘

实例

```sql
SELECT
    o.order_id,
    c.first_name,
    sh.name AS shipper
FROM orders o
JOIN customers c
    USING (customer_id)
LEFT JOIN shippers sh
    USING (shipper_id)
ORDER BY order_id
```

复合主键表间复合连接条件的合并也可用USING:

```sql
SELECT *
FROM order_items oi
JOIN order_item_notes oin

ON oi.order_id = oin.order_Id AND
    oi.product_id = oin.product_id
/USING (order_id【,】 product_id)
```

USING对复合主键的简化效果更加明显

练习

sql_invoicing库里，将payments、clients、payment_methods三张表合并起来，以知道什么日期谁用什么方式付了多少钱

```sql
USE sql_invoicing;

SELECT
    p.date,
    c.name AS client,
    pm.name AS payment_method,
    p.amount
FROM payments p
JOIN clients c USING (client_id)
JOIN payment_methods pm
    ON p.payment_method = pm.payment_method_id
```

注意

列名不同就必须用 ON ..... 了

实际中同一个字段在不同表列名不同的情况也很常见，不能想当然的用 USING，要先确认一下

11. 自然连接

Natural Joins (1:21)

小结

`NATURAL JOIN` **就是让MySQL自动检索同名列作为合并条件。**

注意

最好别用，因为不确定合并条件是否找对了，有时会造成无法预料的问题，**编程时保持对结果的掌控力很重要。**

但也要知道用这个东西，**混个脸熟**，别人用了也看的懂。

实例

```sql
USE sql_store;

SELECT
    o.order_id,
    c.first_name
FROM orders o
NATURAL JOIN customers c
```

12. 交叉连接

Cross Joins (3:14)

小结

得到名字和产品的**所有组合，因此不需要合并条件。**（其实之前的内连接和左右外连接**本质上**都可以看做是在交叉连接的基础上做条件筛选）

实际运用如：要得到尺寸和颜色表的全部组合

实例

```sql
USE sql_store;

SELECT
    c.first_name AS customer,
    p.name AS product
FROM customers c
CROSS JOIN products p
ORDER BY c.first_name
```

上面是显性语法，还有**隐式语法**，之前讲过，其实就是隐式内合并忽略 WHERE 子句（即合并条件）的情况，也就是把 `CROSS JOIN` 改为逗号，**即 `FROM A CROSS JOIN B` 等效于 `FROM A, B`，Mosh 更推荐显式语法，因为更清晰**

```sql
USE sql_store;

SELECT
    c.first_name,
    p.name
FROM customers c, products p
ORDER BY c.first_name
```

练习

交叉合并shippers和products，分别用显式和隐式语法

```sql
USE sql_store;

SELECT
    sh.name AS shippers,
    p.name AS product
FROM shippers sh
CROSS JOIN products p
ORDER BY sh.name
```

或

```sql
SELECT
    sh.name AS shippers,
    p.name AS product
FROM shippers sh, products p
ORDER BY sh.name
```

13. 联合

Unions (8:29)

小结

`FROM …… JOIN ……` 可对多张表进行横向列合并，而 `…… UNION ……` 可用来按行纵向合并多个查询结果，这些查询结果可能来自相同或不同的表

- 同一张表可通过UNION添加新的分类字段，即先通过分类查询并添加新的分类字段再UNION合并为带分类字段的新表。
- 不同表通过UNION合并的情况如：将一张18年的订单表和19年的订单表纵向合并起来在一张表里展示

注意

- **合并的查询结果必须列数相等，否则会报错**
- 合并表里的列名由排在UNION前面的决定

案例1

给订单表增加一个新字段——status，用以区分今年的和以前的订单

```
USE sql_store;

    SELECT
        order_id,
        order_date,
        【'Active' AS status】
    FROM orders
    WHERE order_date >= '2019-01-01'

UNION

    SELECT
        order_id,
        order_date,
        'Archived' AS status  -- Archived 归档
    FROM orders
    WHERE order_date < '2019-01-01';
```

案例2

合并不同表的例子：同一列表里显示所有顾客以及商品名

```
USE sql_store;

    SELECT first_name AS name_of_all
    -- 新列名由排UNION前面的决定
    FROM customers

UNION

    SELECT name
    FROM products
```

练习

给顾客按积分大小分类，添加新字段type，并按顾客id排序，分类标准如下

points	type
<2000	Bronze
2000~3000	Silver
>3000	Gold

```
    SELECT
        customer_id,
        first_name,
        points,
        【'Bronze' AS type】
    FROM customers
```

```
    WHERE points < 2000

UNION

    SELECT
        customer_id,
        first_name,
        points,
        'Silver' AS type
    FROM customers
    WHERE points BETWEEN 2000 and 3000

UNION

    SELECT
        customer_id,
        first_name,
        points,
        'Gold' AS type
    FROM customers
    WHERE points > 3000

ORDER BY customer_id
```

可以看出ORDER BY的优先级在UNION之后，应该是排序和限制语句的执行优先级比较靠后，不知能否用括号调整执行顺序让这个ORDER BY只作用于最后一个子查询？。另外，这里如果没有ORDER BY的话就会按3个query的先后来排序。

总结

感觉本质上可以将查询语句的任何一步和任何一个层次，包括（按实际执行顺序排列）：

1. 选取表 `FROM ……`
2. 横向连接合并 …… `JOIN ……`
3. 纵向筛选 `WHERE ……`
4. 横向筛选 `SELECT ……`
5. 纵向连接合并 …… `UNION ……`
6. 排序、限制，`ORDER BY …… LIMIT ……`

本质上都可以看作暂时生成了一张新表（储存在内存中的虚拟表，中间过程表，桥梁表），将后续步骤都看作是在对这些新表进行进一步的操作，这样，层次步骤就能理清，就好理解了，也才真的能从本质上掌握并灵活运用

【第四章】插入、更新和删除数据

Inserting, Updating, and Deleting Data (时长42分钟)

导航

第一章是简要介绍，第二、三章讲如何 “查”，这一章讲如何 “增、改、删”

前四章构成了SQL的基础 “增删改查”。

1. 列属性

Column Attributes (3:24)

小结

点击表的扳手按钮：打开设计模式，介绍了一些表中字段/列的属性。

一个疑问：为什么 points 的默认值是带引号的0 `'0'` ？

2. 插入单行

Inserting a Row (5:46)

小结

```
INSERT INTO 目标表 （目标列，逗号隔开，也可以省略这一部分不指定列）
VALUES（目标值，逗号隔开）
```

案例

在顾客表里插入一个新顾客的信息

法1. 不指明列名（注意连括号也省了），但插入的值必须按所有字段的顺序完整插入

```
USE sql_store

INSERT INTO customers -- 目标表，不指定列名，连括号都没有了
VALUES (  -- 目标值
    DEFAULT,
    'Michael',
    'Jackson',
    '1958-08-29',  -- DEFAULT/NULL/'1958-08-29'
    DEFAULT,
    '5225 Figueroa Mountain Rd',
    'Los Olivos',
    'CA',
    DEFAULT
    );
```

法2. **指明列名，可跳过取默认值的列且可更改顺序，我感觉还是这种更好，更清晰**

```
INSERT INTO customers (  -- 目标表 + 目标列
    address,
    city,
    state,
    last_name,
    first_name,
    birth_date,
    )
VALUES (  -- 目标值
    '5225 Figueroa Mountain Rd',
    'Los Olivos',
    'CA',
    'Jackson',
    'Michael',
```

```
    '1958-08-29',
    )
```

3. 插入多行

Inserting Multiple Rows (3:18)

小结

VALUES …… 里一行内数据用**括号内逗号**隔开，而多行数据用**括号间逗号**隔开

案例

插入多条运货商信息

```sql
USE sql_store

INSERT INTO shippers (name)
VALUES ('shipper1'),
       ('shipper2'),
       ('shipper3');
```

练习

插入多条产品信息

```sql
USE sql_store;

INSERT INTO products
VALUES (DEFAULT, 'product1', 1, 10),
       (DEFAULT, 'product2', 2, 20),
       (DEFAULT, 'product3', 3, 30)
```

或

```sql
INSERT INTO products (name, quantity_in_stock, unit_price)
VALUES ('product1', 1, 10),
       ('product2', 2, 20),
       ('product3', 3, 30)
```

还是感觉后面这种指明列名的要清晰一点

注意

对于AI (Auto Incremental 自动递增) 的id字段，MySQL会记住删除的/用过的id，并在此基础上递增

4. 插入分级行

Inserting Hierarchical Rows (5:53)

小结

订单表（orders表）里的一条记录对应订单项目表（order_items表）里的多条记录，**一对多，是相互关联的父子表。**通过添加一条订单记录和**对应的**多条订单项目记录，学习如何**向父子表插入分级（层）/耦合数据（insert hierarchical data)**

相关知识点：

- **内建函数**：MySQL里有很多可用的内置函数，也就是**可复用的代码块，各有不同的功能**，注意函数名的单词之间用**下划线**连接
- `LAST_INSERT_ID()` 函数：**获取最新的成功的 `INSERT`语句 中的自增id**，在这个例子中就是父表里新增的order_id.
- 关键：在插入子表记录时，用内建函数 `LAST_INSERT_ID()` **来获取相关父表记录的自增ID**（这个例子中就是orders表中的order_id)

案例

新增一个订单（order），里面包含两个订单项目/两种商品（order_items），请**同时更新订单表和订单项目表**

```sql
USE sql_store;

INSERT INTO orders (customer_id, order_date, status)
VALUES (1, '2019-01-01', 1);

-- 可以先试一下用 SELECT last_insert_id() 看能否成功获取到的最新的order_id

INSERT INTO order_items  -- 全是必须字段，就不用指定了
VALUES
    (【last_insert_id()】, 1, 2, 2.5),
    (last_insert_id(), 2, 5, 1.5)
```

5. 创建表的副本

Creating a Copy of a Table (8:47)

小结

法1. 删除重建：`DROP TABLE 要删的表名`、`CREATE TABLE 新表名 AS 子查询`

法2. 清空重填：`TRUCATE '要清空的表名'`、`INSERT INTO 表名 子查询`

子查询里当然也可以用WHERE语句进行筛选

案例 1

运用 `CREAT TABLE 新表名 AS 子查询` 快速创建表 orders 的副本表 orders_archived

```sql
USE sql_store;

CREATE TABLE orders_archived AS
    SELECT * FROM orders  -- 子查询
```

`SELECT * FROM orders` 选择了 oders 中所有数据，作为AS的内容，是一个子查询

- **子查询： 任何一个充当另一个SQL语句的一部分的 `SELECT……` 查询语句都是子查询，子查询是一个很有用的技巧。**

注意

创建已有的表或删除不存在的表的话都会报错，所以建表和删表语句都最好加上条件语句（后面会讲）

案例 2

不再用全部数据，而选用原表中部分数据创建副本表，如，**用今年以前的 orders 创建一个副本表 orders_archived**，其实就是在子查询里增加了一个WHERE语句进行筛选。**注意要先 drop 删掉 或 truncate 清空掉之前建的 orders_archived 表再重建或重填。**

法1. `DROP TABLE 要删的表名、CREATE TABLE 新表名 AS 子查询`

```
USE sql_store;

DROP TABLE orders_archived;  -- 也可右键该表点击 drop
CREATE TABLE orders_archived AS
    SELECT * FROM orders
    WHERE order_date < '2019-01-01'
```

法2. `TRUCATE '要清空的表名'、INSERT INTO 表名 子查询`

`INSERT INTO 表名 子查询` 很常用，子查询替代原先插入语句中 `VALUES(……,……),(……,……),……` 的部分

```
TRUNCATE 'orders_archived';
-- 也可右键该表点击 truncate
-- 新的 8.0版 MySQL 的语法好像变为了 TRUNCATE TABLE orders_archived? 那样就与 DROP TABLE orders_archived 一致了
INSERT INTO orders_archived  -- 不用指明列名，会直接用子查询表里的列名
    SELECT * FROM orders  -- 子查询，替代原先插入语句中VALUES(……,……),(……,……),…… 的部分
    WHERE order_date < '2019-01-01'
```

练习

创建一个存档发票表，只包含有过支付记录的发票并将顾客id换成顾客名字

构建的思路顺序：

1. 先创建子查询，确定新表内容：
    1. 合并发票表和顾客表
    2. **筛选支付记录不为空的行/记录**
    3. 筛选（并重命名）需要的列
2. 第1步得到的查询内容，可以先运行看一下，确保准确无误后，再作为子查询内容存入新创建的副本订单存档表 `CREATE TABLE 新表名 AS 子查询`

```
USE sql_invoicing;

DROP TABLE invoices_archived;

CREATE TABLE invoices_archived AS
    SELECT i.invoice_id, c.name AS client, i.payment_date
    -- 为了简化，就选这三列
    FROM invoices i
    JOIN clients c
        USING (client_id)
    【WHERE i.payment_date IS NOT NULL】
    -- 或者 i.payment_total > 0
```

6. 更新单行

Updating a Single Row (3:55)

小结

用 UPDATE …… 语句 来修改表中的一条或多条记录，具体语法结构：

```
UPDATE 表
【SET 要修改的字段 = 具体值/NULL/DEFAULT/列间数学表达式】 （【修改多个字段用逗号分隔】）
WHERE 行筛选
```

实际执行顺序应该是 UPDATE → WHERE → SET

实例

```
USE sql_invoicing;

UPDATE invoices
SET
    payment_total = 100 / 0 / DEFAULT / NULL / 0.5 * invoice_total,
    -- 【注意 0.5 * invoice_total 的结果小数被舍弃，之后讲数据类型会讲到这个问题】
    payment_date = '2019-01-01' / DEFAULT / NULL / due_date
WHERE invoice_id = 3
```

7. 更新多行

Updating Multiple Rows (3:14)

小结

语法一样的，就是让 WHERE…… 的条件包含更多记录，就会同时更改多条记录了

注意

Workbench默认开启了**Safe Updates功能**，不允许同时更改多条记录，要先关闭该功能（在 Preference——SQL Editor 里）

```
USE sql_invoicing;

UPDATE invoices
SET payment_total = 233, payment_date = due_date
WHERE client_id = 3  -- 该客户的发票记录不止一条，将同时更改
/WHERE client_id IN (3, 4) -- 第二章 4~9 讲的那些写 WHERE 条件的方式当然都可以用
-- 甚至可以直接省略 WHERE 语句，会直接更改整个表的全部记录
```

练习

给所有非90后顾客增加50点积分

```
USE sql_store;

UPDATE customers
SET points = points + 50
WHERE birth_date < '1990-01-01';
```

8. 在Updates中使用子查询

Using Subqueries in Updates (5:36)

小结

非常有用，其实本质上是**将子查询用在 `WHERE……` 行筛选条件中**

注意

1. 括号的使用
2. **`IN` …… 后除了可接 （……，……） 也可接由子查询得到的多个数据（一列多条数据）**

案例

更改发票记录表中名字叫 Yadel 的记录，但该表只有 client_id，故**先要从另一个顾客表中查询叫 Yadel 人的 client_id**

实际中这是很可能的情形，比如一个App是通过搜索名字来更改发票记录的

```sql
USE sql_invoicing;

UPDATE invoices
SET payment_total = 567, payment_date = due_date

★WHERE client_id =
    (SELECT client_id
    FROM clients
    WHERE name = 'Yadel');
    -- 放入括号，确保先执行

-- 若子查询返回多个数据（一列多条数据）时就不能用等号而要用 IN 了：
WHERE client_id 【IN】
    (SELECT client_id
    FROM clients
    WHERE state IN ('CA', 'NY'))
```

最佳实践

Update 前，最好先验证看一看子查询以及WHERE行筛选条件是不是准确的，筛选出的是不是我们的修改目标，确保不会改错记录，再套入UPDATE SET语句更新，如上面那个就可以先验证子查询：

```sql
SELECT client_id
FROM clients
WHERE state IN ('CA', 'NY')
```

以及验证WHERE行筛选条件（即先不UPDATE，先SELECT，改之前，先看一看要改的目标选对了没）

```sql
SELECT *
FROM invoices
WHERE client_id IN (
    SELECT client_id
    FROM clients
    WHERE state IN ('CA', 'NY')
)
```

确保WHERE行筛选条件准确准确无误后，再放到修改语句后执行修改：

```sql
UPDATE invoices
SET payment_total = 567, payment_date = due_date
WHERE client_id IN (
    SELECT client_id
    FROM clients
    WHERE state IN ('CA', 'NY')
```

有子查询的 Update 主要验证 Where 条件中的 子查询部分正不正确，而没有子查询的 Update 则应该**将 update 换成 select 先验证一下整个 Where 筛选条件正不正确。**

练习

将 orders 表里那些 分数>3k 的用户的订单 comments 改为 'gold customer'，

思考步骤：

1. WHERE 行筛选出要求的顾客
2. SELECT 列筛选他们的id
3. 将前两步 作为子查询 用在修改语句中的 WHERE 条件中，执行修改

```sql
USE sql_store;

UPDATE orders
SET comments = 'gold customer'
WHERE customer_id IN
    (SELECT customer_id
    FROM customers
    WHERE points > 3000)
```

9. 删除行

Deleting Rows (1:24)

小结

语法结构：

```sql
【DELETE FROM】 表
WHERE 行筛选条件（当然也可用子查询）(若省略WHERE条件语句会删除表中所有记录（和TRUNCATE啥区别？）)
```

案例

选出顾客id为3/顾客名字叫'Myworks'的发票记录

```sql
USE sql_invoicing;

DELETE FROM invoices
WHERE client_id = 3
-- WHERE可选，省略就是会删除整个表的所有行/记录
/WHERE client_id =
    (SELECT client_id
     -- Mosh 错写成了 SELECT *，将报错：Error Code: 1241. Operand n. [计] 操作数；[计] 运算对象；运算元 should contain 1 column(s)
    FROM clients
    WHERE name = 'Myworks')
```

10. 恢复数据库

Restoring the Databases (1:06)

就是重新运行那个SQL文件以重置数据库

专栏介绍和总目录
数据概要

专栏介绍和总目录
数据概要

第二部分：基础进阶——汇总、复杂查询、内置函数

【第五章】汇总数据

Summarizing Data (时长33分钟)

汇总统计型查询非常有用，甚至可能常常是你的主要工作内容

1. 聚合函数

Aggregate Functions (9:19)

小结

聚合函数：输入一系列值并聚合为一个结果的函数

实例

```sql
USE sql_invoicing;

SELECT
    MAX(invoice_date) AS latest_date,
    -- SELECT选择的不仅可以是列，也可以是数字、列间表达式、列的聚合函数
    MIN(invoice_total) lowest,
    AVG(invoice_total) average,
    SUM(invoice_total * 1.1) total,
    COUNT(*) total_records,
    COUNT(invoice_total) number_of_invoices,
    -- 和上一个相等
    COUNT(payment_date) number_of_payments,
```

```
    -- 聚合函数会忽略空值，支付数少于发票数
    【COUNT(DISTINCT client_id) number_of_distinct_clients】
    -- DISTINCT client_id筛掉了该列的重复值，再COUNT计数，不同顾客数
FROM invoices
WHERE invoice_date > '2019-07-01'  -- 想只统计下半年的结果

```

练习

目标：

date_range	total_sales	total_payments	what_we_expect(the difference)
1st_half_of_2019			
2nd_half_of_2019			
Total			

思路：很明显要 分类子查询+聚合函数+UNION

```
USE sql_invoicing;

    SELECT
        【'1st_half_of_2019' AS date_range】,
        SUM(invoice_total) AS total_sales,
        SUM(payment_total) AS total_payments,
        SUM(invoice_total - payment_total) AS what_we_expect
    FROM invoices
    WHERE invoice_date BETWEEN '2019-01-01' AND '2019-06-30'

UNION

    SELECT
        '2st_half_of_2019' AS date_range,
        SUM(invoice_total) AS total_sales,
        SUM(payment_total) AS total_payments,
        SUM(invoice_total - payment_total) AS what_we_expect
    FROM invoices
    WHERE invoice_date BETWEEN '2019-07-01' AND '2019-12-31'

UNION

    SELECT
        'Total' AS date_range,
        SUM(invoice_total) AS total_sales,
        SUM(payment_total) AS total_payments,
        SUM(invoice_total - payment_total) AS what_we_expect
    FROM invoices
    WHERE invoice_date BETWEEN '2019-01-01' AND '2019-12-31'
```

2. GROUP BY子句

The GROUP BY Clause (7:21)

小结

按一列或多列分组，注意语句的位置。

案例1：按一个字段分组

在发票记录表中**按不同顾客分组**统计各个顾客下半年总销售额并降序排列

```
USE sql_invoicing;

SELECT
    client_id,
    SUM(invoice_total) AS total_sales
……
```

只有聚合函数是按client_id分组时，这里选择client_id列才有意义（分组统计语句里SELECT通常都是选择分组依据列+目标统计列的聚合函数，选别的列没意义）。若未分类，结果会是一条总total_sales和一条client_i（该client_id无意义），即client_id会被压缩为只显示一条而非SUM广播为多条，可以理解为聚合函数比较强势吧。（要SUM扩散为多条得用OVER，即窗口函数）

```
……
FROM invoices
WHERE invoice_date >= '2019-07-01'  -- 筛选，过滤器
GROUP BY client_id  -- 分组
ORDER BY invoice_total DESC
```

若省略排序语句就会默认按分组依据排序（后面一个例子发现好像也不一定，所以最好别省略）

记住语句顺序很重要 WHERE GROUP BY ORDER BY，分组语句在排序语句之前，调换顺序会报错

案例2：按多个字段分组

算各州各城市的总销售额

如前所述，一般分组依据字段也正是 `SELECT ……` 里的选择字段，如下面例子里的 state 和 city

```
USE sql_invoicing;

SELECT
    state,
    city,
    SUM(invoice_total) AS total_sales
FROM invoices
JOIN clients USING (client_id)
-- 【别忘了USING之后是括号，太容易忘了】
GROUP BY state, city
-- 逗号分隔就行
-- 这个例子里 GROUP BY里去掉state结果一样
ORDER BY state
```

其实上面的例子里一个城市只能属于一个州中，所有归根结底还是算的各城市的销售额，`GROUP BY ……` 里去掉 state 只写 city （但select和order by里保留state）结果是完全一样的（包括结果里的state列），下面这个例子更能说明以多个字段为分组依据进行分组统计的意义

练习

在 payments 表中，按日期和支付方式分组统计总付款额

每个分组显示一个日期和支付方式的独立组合，可以看到某特定日期并有某特定支付方式的总付款额。这个例子里每一种支付方式可以在不同日子里出现，每一天也可以出现多种支付方式，这种情况，才叫真·多字段分组。【不过上一个例子里那种假·多字段分组，把state加在分组依据里也没坏处还能落个心安，也还是加上别省比较好，而且像PostgreSQL之类的甚至强制要求只能SELECT在分组依据中出现过的字段（以及聚合函数），所以有时仅仅为了在 SELECT 里选择的目的也该在 GROUP BY 里包含某些字段，即便它们不会起到实质的分组依据的作用】

```sql
USE sql_invoicing;

SELECT
    date,
    pm.name AS payment_method,
    SUM(amount) AS total_payments
FROM payments p
JOIN payment_methods pm
    ON p.payment_method = pm.payment_method_id
GROUP BY date, payment_method
ORDER BY date
```

思想

解答复杂问题时，学会先分解拆分为简单的小问题或小步骤逐个击破。合理运用分解组合和IPO思想。

3. HAVING子句

The HAVING Clause (8:50)

小结

HAVING 和 WHERE 都是是条件筛选语句，条件的写法相通，数学比较（包括特殊比较）逻辑运算都可以用（如AND、REGEXP等等）

两者本质区别:

- **WHERE 是对 FROM JOIN 里原表中的列进行 事前筛选，所以WHERE可以对没选择的列进行筛选，但必须用原表列名而不能用SELECT中确定的列别名**
- **相反 HAVING …… 对 `SELECT` …… 查询后（通常是分组并聚合查询后）的结果列进行 事后筛选，若SELECT里起了别名的字段则必须用别名进行筛选，且不能对SELECT里未选择的字段进行筛选。唯一特殊情况是，当HAVING筛选的是聚合函数时，该聚合函数可以不在SELECT里显性出现，见最后补充**

实例

```sql
USE sql_invoicing;

SELECT
    client_id,
    SUM(invoice_total) AS total_sales,
    COUNT(*/invoice_total/invoice_date) AS number_of_invoices
FROM invoices
GROUP BY client_id
HAVING total_sales > 500 AND number_of_invoices > 5
```

若写：`WHERE total_sales > 500 AND number_of_invoices > 5`，会报错：Error Code: 1054. **Unknown column 'total_sales' in 'where clause'**

练习

在sql_store数据库（有顾客表、订单表、订单项目表等）中，找出在'VA'州且消费总额超过100美元的顾客（这是一个面试级的问题，还很常见）

思路：

1. 需要的信息在顾客表、订单表、订单项目表三张表中，先将三张表合并
2. 事前筛选 'VA'州的
3. 按顾客分组，并选取所需的列并聚合得到每位顾客的付款总额
4. 事后筛选超过 100美元 的

```sql
USE sql_store;

SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    SUM(oi.quantity * oi.unit_price) AS total_sales
FROM customers c
JOIN orders o USING (customer_id)  -- 别忘了括号，特容易忘
JOIN order_items oi USING (order_id)
WHERE state = 'VA'
GROUP BY
    c.customer_id,
    【c.first_name,
    c.last_name】-- 因为 SELECT 中需要选，所以在 GROUP BY 里最好加上，虽然按 id 分组本已足够
HAVING total_sales > 100
```

补充

学第六章第6节时发现，当HAVING筛选的是聚合函数时，该聚合函数可以不在SELECT里显性出现。（作为一种需要记住的特殊情况） 如：下面这两种写法都能筛选出总积分大于3k的州，如果不要求显示总点数，应该用后一种

```sql
SELECT state, SUM(points)
FROM customers
GROUP BY state
HAVING SUM(points) > 3000
```

```sql
SELECT state
FROM customers
GROUP BY state
【HAVING SUM(points) > 3000】
```

4. ROLLUP运算符

The ROLLUP Operator (5:05)

小结

`GROUP BY …… WITH ROLL UP` **自动汇总型分组（对 SUM 之类的聚合值进行分组汇总），若是多字段分组的话汇总也会是多层次的**，注意这是**MySQL扩展语法**，不是SQL标准语法

案例

分组查询各客户的发票总额**以及所有人的总发票额**

```sql
USE sql_invoicing;

SELECT
    client_id,
    SUM(invoice_total)
FROM invoices
GROUP BY client_id WITH ROLLUP
-- 当然，总发票额那一行 client_id 为空
```

多字段分组 例1：分组查询各州、市的总销售额（发票总额）以及州层次和全国层次的**两个层次的汇总额**

```sql
SELECT
    state,
    city,
    SUM(invoice_total) AS total_sales
FROM invoices
JOIN clients USING (client_id)
GROUP BY state, city WITH ROLLUP
-- 先按 city 汇总，再按 state 汇总，与分组顺序相反（当然，分组和汇总本来就是相反的两个过程）
```

多字段分组 例2：分组查询特定日期特定付款方式的总支付额**以及单日汇总和整体汇总**

```sql
USE sql_invoicing;

SELECT
    date,
    pm.name AS payment_method,
    SUM(amount) AS total_payments
FROM payments p
JOIN payment_methods pm
    ON p.payment_method = pm.payment_method_id
GROUP BY date, pm.name WITH ROLLUP
-- 注意这儿 GROUP BY 里若使用列别名 payment_method 则结果没有每日层次的汇总
-- GROUP BY 分组依据和 SELECT 选择字段（除聚合函数外）最好是能一一对应，这是最保险的
```

练习

分组计算各个付款方式的总付款 并汇总

```sql
SELECT
    pm.name AS payment_method,
    SUM(amount) AS total
FROM payments p
JOIN payment_methods pm
    ON p.payment_method = pm.payment_method_id
GROUP BY pm.name WITH ROLLUP
```

★总结

根据之后三篇参考文章，据说标准的 SQL 查询语句的执行顺序应该是下面这样的：

1. FROM JOIN 选择和连接本次查询所需的表
2. ON/USING WHERE 按条件筛选行
3. GROUP BY 分组
4. SELECT 筛选列（注意若进行了分组，这一步常常要聚合）
   注意：SELECT 和 HAVING 在 MySQL 里的执行顺序我还有点疑问，见后面的叙述
5. HAVING （分组聚合后）按条件筛选行

6. DISTINCT 去重
7. UNION 纵向合并
8. ORDER BY 排序
9. LIMIT 限制

"SELECT 是在大部分语句执行了之后才执行的，严格的说是在 FROM、WHERE 和 GROUP BY （以及 HAVING*？）之后执行的。理解这一点是非常重要的，**这就是你不能在 WHERE 中使用在 SELECT 中设定别名的字段作为判断条件的原因。**"

这个顺序可以由下面这个例子的缩进表现出来（出右往左）（注意 DISTINCT 放不进去了只有以注释的形式展示出来，另外SELECT 还是选择放在了 HAVING 之前）

```
USE sql_invoicing;

                    SELECT name, SUM(invoice_total) AS total_sales
         -- DISTINCT
                                FROM invoices JOIN clients USING (client_id)
                            WHERE due_date < '2019-07-01'
                        GROUP BY name
                HAVING total_sales > 150

        UNION

                    SELECT name, SUM(invoice_total) AS total_sales
         -- DISTINCT
                                FROM invoices JOIN clients USING (client_id)
                            WHERE due_date > '2019-07-01'
                        GROUP BY name
                HAVING total_sales > 150

    ORDER BY total_sales
LIMIT 2
```

关于 SELECT 的位置

1. 如后面几篇参考文章所说，**按标准 SQL 的执行顺序， SELECT 是在 HAVING 之后**
2. 但根据前面的内容，**似乎在 MySQL 里，SELECT 的执行顺序是在 WHERE GROUP BY 之后，而在 HAVING 之前 —— 因而 WHERE GROUP BY 要用原列名（后来发现只是WHERE必须用原列名，GROUP BY 是原列名或列别名都可用（甚至可以用1，2来指代 SELECT 中的列，不过 Mosh 不建议这样做））而 HAVING 必须用 SELECT 里的列别名（聚合函数除外）**

按实践经验来看，就按 2 来记忆和理解是可行的，但之后最好还是要去看书看资料把这个执行顺序的疑惑彻底搞清楚，这个还挺重要的。

参考文章1："十步完全理解 SQL"
主要看文中的第2点：SQL 的语法并不按照语法顺序执行，但注意文中只说了标准的SQL执行顺序，MySQL的执行顺序还是有所不同的

参考文章2：SQL语句优化--执行顺序

参考文章3：查询执行顺序

【第六章】编写复杂查询

Writing Complex Query (时长45分钟)

1. 介绍

Introduction (1:28)

主要是子查询，有些前面已经讲过

2. 子查询

Subqueries (2:29)

回顾

子查询： **任何一个充当另一个SQL语句的一部分的 SELECT 查询语句都是子查询**，子查询是一个很有用的技巧。**子查询的层级用括号实现。**

注意

另外发现各种语言，各种语句，各种逻辑结构，各种情形下**一般好像多加括号都不会有问题，只有少加括号才会出问题，所以不确定执行顺序是否正确时最好加上括号确保万无一失。**

案例

在products中，找到所有比生菜（id = 3）价格高的

关键：要找比生菜价格高的，得先用子查询找到生菜的价格

```
USE sql_store;

SELECT *
FROM products
WHERE unit_price > (
    SELECT unit_price
    FROM products
    WHERE product_id = 3
)
```

MySQL执行时会先执行括号内的子查询（内查询），将获得的生菜价格作为结果返回给外查询

子查询不仅可用在 WHERE …… 中，也可用在 SELECT …… 或 FROM …… 等子句中，本章后面会讲

练习

在sql_hr.employees表里，选择所有工资超过平均工资的雇员

关键：先由子查询得到平均工资

```sql
USE sql_hr;

SELECT *
FROM employees
WHERE salary > (
    SELECT AVG(salary)
    FROM employees
)
```

3. IN运算符

The IN Operator (3:39)

案例

在sql_store.products找出那些从未被订购过的产品

思路：

1. orders.items表里有所有产品被订购的记录，从中可得到 所有被订购过的产品 的列表（注意用 DISTINCT 关键字进行去重）
2. 不在这列表里（NOT IN 的使用）的产品即为从未被订购过的产品

```sql
USE sql_store;

SELECT *
FROM products
WHERE product_id NOT IN (
    SELECT DISTINCT product_id
    FROM order_items
)
```

上一节是子查询返回一个值（平均工资），这一节是返回一列数据（被订购过的产品id列表），之后还会用子查询返回一个多列表

练习

在 sql_invoicing.clients 中找到那些没有过发票记录的客户

思路：和上一个例子完全一致，在invoices里用DISTINCT找到所有有过发票记录的客户的列表，再用NOT IN来筛选

```sql
USE sql_invoicing;

SELECT *
FROM clients
WHERE client_id NOT IN (
    SELECT DISTINCT client_id
    FROM invoices
)
```

4. 子查询vs连接

Subqueries vs Joins (5:07)

小结

子查询（Subquery）是**将一张表的查询结果作为另一张表的查询依据并层层嵌套**，其实也可以先将这些表连接（Join）**合并成一个包含所需全部信息的详情表再直接在详情表里筛选查询。**两种方法**一般是可互换的，具体用哪一种取决于 性能（Performance） 和 可读性（readability）**，之后会学习 执行计划，到时候就知道怎样编写并更快速地执行查询，现在主要考虑可读性

案例

上节课的案例，找出从未订购（没有invoices）的顾客：

法1. 子查询

先用子查询查出有过发票记录的顾客名单，作为筛选依据

```sql
USE sql_invoicing;

SELECT *
FROM clients
WHERE client_id NOT IN (
    SELECT DISTINCT client_id
    /*
    其实这里加不加DISTINCT对子查询返回的结果有影响
    但对最后的结果没有影响
    */
    FROM invoices
)
```

法2. 连接表

用顾客表 LEFT JOIN 发票记录表，再直接在这个合并详情表中**筛选出发票记录为空的顾客**

```sql
USE sql_invoicing;

SELECT DISTINCT client_id, name ……
-- 不能SELECT DISTINCT *
FROM clients
LEFT JOIN invoices USING (client_id)
-- 【注意不能用内连接，否则没有发票记录的顾客（我们的目标）直接就被筛掉了】
WHERE invoice_id IS NULL
```

就上面这个案例而言，子查询可读性更好，但有时子查询会过于复杂（嵌套层数过多），用连接表更好（下面的练习就是）。总之在选择方法时，**可读性**是很重要的考虑因素

练习

在sql_store中，选出买过生菜（id = 3）的顾客的id、姓和名

分别用子查询法和连接表法实现并比较可读性

法1. 完全子查询

```sql
USE sql_store;

SELECT customer_id, first_name, last_name
FROM customers
WHERE customer_id IN (
    -- 子查询2—找顾客：从订单表中找出哪些顾客买过生菜
```

```
    SELECT customer_id
    FROM orders
    WHERE order_id IN (
        -- 子查询1—找订单：从订单项目表中找出哪些订单包含生菜
        SELECT DISTINCT order_id
        FROM order_items
        WHERE product_id = 3
    )
)
```

另外注意上面的括号和缩进

法2. 子查询 + 表连接

```
USE sql_store;

SELECT customer_id, first_name, last_name
FROM customers
WHERE customer_id IN (
    -- 子查询：哪些顾客买过生菜
    SELECT customer_id
    FROM orders
    JOIN order_items USING (order_id)
    -- 表连接：合并订单和订单项目表得到 订单详情表
    WHERE product_id = 3
)
```

法3. 完全表连接

直接连接合并3张表（顾客表、订单表和订单项目表）得到 带顾客信息的订单详情表，该合并表包含我们所需的所有信息，可直接在合并表中用WHERE筛选买过生菜的顾客（注意 DISTINCT 关键字的运用）。

```
USE sql_store;

SELECT DISTINCT customer_id, first_name, last_name
FROM customers
LEFT JOIN orders USING (customer_id)
LEFT JOIN order_items USING (order_id)
-- 另外注意上面两次使用 USING 连接三张表的方式，依次连接
WHERE product_id = 3
```

这个案例中，先将所需信息所在的几张表全部连接**合并成一张大表再来查询筛选明显比层层嵌套的多重子查询更加清晰明了**

5. ALL关键字

The ALL Keyword (4:52)

小结

`> (MAX (……))` 和 `> ALL(……)` **等效可互换**

“比这里面最大的还大” = “比这里面的所有的都大”

案例

sql_invoicing库中，选出金额大于 3号顾客所有发票金额（或最大发票金额） 的发票

法1. 用MAX关键字

```
USE sql_invoicing;

SELECT *
FROM invoices
WHERE invoice_total > (
    SELECT MAX(invoice_total)
    FROM invoices
    WHERE client_id = 3
)
```

法2. 用ALL关键字

```
USE sql_invoicing;

SELECT *
FROM invoices
WHERE invoice_total > ALL (
    SELECT invoice_total
    FROM invoices
    WHERE client_id = 3
)
```

其实就是把内层括号的MAX拿到了外层括号变成ALL：
MAX法是用MAX()**返回一个顾客3的最大订单金额，再判断哪些发票的金额比这个值大；**
ALL法是先**返回顾客3的所有订单金额，是一列值，再用ALL()判断比所有这些金额都大的发票有哪些。**
两种方法是完全等效的

6. ANY关键字

The ANY Keyword (2:36)

小结

`> ANY/SOME (……)` 与 `> (MIN (……))` **等效**

`= ANY/SOME (……)` 与 `IN (……)` **等效**

案例1

`> ANY (……)` 与 `> (MIN (……))` 等效的例子：
sql_invoicing库中，选出金额大于 3号顾客任何发票金额（或最小发票金额） 的发票

```
USE sql_invoicing;

SELECT *
FROM invoices

WHERE invoice_total > ANY (
    SELECT invoice_total
    FROM invoices
    WHERE client_id = 3
)

```

```
或

WHERE invoice_total > (
    SELECT MIN(invoice_total)
    FROM invoices
    WHERE client_id = 3
)
```

案例2

= ANY (……) 与 IN (……) 等效的例子:
选出至少有两次发票记录的顾客

```
USE sql_invoicing;

SELECT *
FROM clients
WHERE client_id IN (  -- 或 = ANY (
    -- 子查询：有2次以上发票记录的顾客
    SELECT client_id
    FROM invoices
    GROUP BY client_id
    ★【HAVING COUNT(*) >= 2】
)
```

7. 相关子查询

Correlated Subqueries (5:36)

小结

之前都是非关联主/子（外/内）查询，比如子查询先查出整体的某平均值或满足某些条件的一列id，作为主查询的筛选依据，这种子查询与主查询无关，会先一次性得出查询结果再返回给主查询供其使用。

而下面这种相关联子查询例子里，**子查询要查询的是某员工【所在/对应】办公室的平均值，子查询是依赖主查询的，注意这种关联查询是在主查询的每一行/每一条记录层面上依次进行的，这一点可以为我们写关联子查询提供线索（注意表别名的使用），另外也正因为这一点，相关子查询会比非关联查询执行起来慢一些。**

案例

选出 sql_hr.employees 里那些工资超过**他所在办公室平均工资（而不是整体平均工资）**的员工
关键：如何查询目前主查询员工的所在办公室的平均工资而不是整体的平均工资?
思路：给主查询 employees表 设置别名 e，这样在子查询查询平均工资时加上 `WHERE office_id = e.office_id` 筛选条件即可相关联地查询到目前员工所在地办公室的平均工资

```
USE sql_hr;

SELECT *
FROM employees e  -- 【关键】
WHERE salary > (
    SELECT AVG(salary)
    FROM employees
    WHERE office_id = e.office_id  -- 【这个关联筛选条件是关键】
    -- 【子查询表字段不用加前缀，主查询表的字段要加前缀，以此区分】
)
```

相关子查询很慢，但**很强大，也有很多实际运用**

练习

在sql_invoicing.invoices中，找出高于每位顾客平均发票金额的发票

```
USE sql_invoicing;

SELECT *
FROM invoices i
WHERE  invoice_total > (
    -- 子查询：目前客户的平均发票额
    SELECT AVG(invoice_total)
    FROM invoices
    【WHERE client_id = i.client_id】
)
```

8. EXISTS运算符

The EXISTS Operator (5:39)

小结

- **IN + 子查询 等效于 EXIST + 相关子查询，如果前者子查询的结果集过大占用内存，用后者逐条验证更有效率。**
- **EXIST()本质上是根据是否为空返回TRUE和FALSE**
- 与往常一样，自然也可以**加NOT取反**

案例

找出有过发票记录的客户，第4节学过用子查询或表连接来实现

法1. 子查询

```
USE sql_invoicing;

SELECT *
FROM clients
WHERE client_id IN (
    SELECT DISTINCT client_id
    FROM invoices
)
```

法2. 连接表

```
USE sql_invoicing;

SELECT DISTINCT client_id, name ……
FROM clients
JOIN invoices USING (client_id)
-- 【内连接】，只留下有过发票记录的客户
```

法3. 用EXISTS运算符实现

```sql
USE sql_invoicing;

SELECT *
FROM clients c
WHERE EXISTS (
    SELECT */client_id
    /*
    就这个子查询的目的来说，SELECT的选择不影响结果，
    因为【EXISTS()函数只根据是否为空返回TRUE/FALSE】
    */
    FROM invoices
    WHERE client_id = c.client_id
)
```

从这个案例可以看得出来：

- **EXISTS(…) 函数相当于是前置的 … IS NULL（共同点：都是根据是否为空返回布林值）**
- **WHERE 确实是逐条验证筛选行/记录的**

这还是个相关子查询，因为在其中引用了主查询的clients表。这同样是按照主查询的记录一条条验证执行的。具体说来，对于每一个client，子查询查找invoices表里是否有这个人的发票记录，有就返回相关记录否者返回空，然后EXISTS()根据是否为空返回TRUE和FALSE，然后主查询凭此确定是否保留此条记录。

对比一下，法1是用子查询返回一个有发票记录的顾客id列表，如（1，3，8 ……），然后用IN运算符来判断，如果子查询表太大，可能返回一个上百万千万甚至上亿的id列表，这个id列表就会很占内存非常影响性能，**对于这种子查询会返回一个很大的结果集（常常是作为筛选条件用IN判断，必须全部放入内存）的情况，用这里的EXIST+相关子查询逐条筛选（里外两层都是逐条判断筛选，不会出现要在内存中放入很大的列表的情况）会更有效率**

另外，因为SELECT()返回的是TRUE/FALSE，所以自然也可以加上NOT取反，见下面的练习

练习

在sql_store中，找出**从来没有被订购过的产品。**

```sql
USE sql_store;

SELECT *
FROM products
WHERE product_id NOT IN (
    SELECT product_id
    -- 加不加DISTINCT对最终结果无影响
    FROM order_items
)
```

或

```sql
SELECT *
FROM products p
WHERE NOT EXISTS (
    SELECT *
    FROM order_items
    WHERE product_id = p.product_id
)
```

对于亚马逊这样的大电商来说，如果用IN+子查询法，子查询可能会返回一个百万量级的产品列表，这种情况还是用EXIST+相关子查询逐条验证法更有效率

9. SELECT子句的子查询

Subqueries in the SELECT Clause (4:29)

小结

不仅WHERE筛选条件里可以用子查询，SELECT选择子句和FROM来源表子句也能用子查询，这节课讲SELECT子句里的子查询

简单讲就是，SELECT选择语句是用来确定查询结果选择包含哪些字段，每个字段都可以是一个表达式，而每个字段表达式里的元素除了可以是原始的列，具体的数值，也同样可以是其它各种花里胡哨的子查询的结果

任何子查询都是都是简单查询的嵌套，没什么新东西，只是多了一个层级而已，【由外向内】地一层层梳理就很清楚

另外，要特别注意记住以子查询方式实现在SELECT中使用同级列别名的方法（作为一种特殊的不需要FROM的子查询来记忆）

案例

得到一个有如下字段的表格：invoice_id, invoice_total, avarege（总平均发票额）, difference

```sql
USE sql_invoicing;

SELECT
    invoice_id,
    invoice_total,
    (SELECT AVG(invoice_total) FROM invoices) AS invoice_average,
    /*
    不能直接用聚合函数，因为会压缩聚合结果为一条
    用括号+子查询改变顺序，【子查询 (SELECT AVG(invoice_total) FROM invoices)
    是作为一个数值结果 152.388235 加入主查询语句的】
    【也可以用窗口函数，更简洁：AVG(invoice_total) OVER() AS invoice_average,
    但注意用了窗口函数，后面就不能引用这个用窗口函数产生的列别名 invoice_average，报错：
    “You cannot use the alias 'invoice_average' of an expression containing a window function in this context.”
    可能是因为窗口函数的执行顺序是在SELECT子句其它字段选择完成之后吧】
    */
    invoice_total - 【(SELECT invoice_average)】 AS difference
    /*
    SELECT表达式里要用原列名，不能直接用别名invoice_average
    要用列别名的话用子查询（SELECT 同级的列别名）即可
    说真的，感觉这个子查询有点难以理解，这场能作为【一种特殊的不需要FROM的子查询来记忆】
    */
FROM invoices
```

练习

得到一个有如下字段的表格：client_id, name, total_sales（各个客户的发票总额）, average（总平均发票额）, difference

```sql
USE sql_invoicing;

SELECT
    client_id,
    name,
    (SELECT SUM(invoice_total) FROM invoices WHERE client_id = c.client_id) AS total_sales,
    -- 要得到【相关】客户的发票总额，要用【相关子查询】
    (SELECT AVG(invoice_total) FROM invoices) AS average,
    (SELECT 【total_sales - average】) AS difference
FROM clients c
```

注意第四个客户的total_sales和difference都是空值null

10. FROM子句的子查询

Subqueries in the FROM Clause (2:58)

小结

子查询的结果同样可以充当一个“虚拟表”作为FROM语句中的来源表，即将筛选查询结果作为来源再进行进一步的筛选查询。但注意只有在子查询不太复杂时进行这样的嵌套，否则最好用后面讲的视图先把子查询结果储存起来再使用。

案例

将上一节练习里的查询结果当作来源表，查询其中total_sales非空的记录

```sql
USE sql_invoicing;

SELECT *
FROM (
    SELECT
        client_id,
        name,
        (SELECT SUM(invoice_total) FROM invoices WHERE client_id = c.client_id) AS
total_sales,
        (SELECT AVG(invoice_total) FROM invoices) AS average,
        (SELECT total_sales - average) AS difference
    FROM clients c
) AS sales_summury
/*
在FROM中使用子查询，即使用“派生表”时，
【必须给派生表取个别名（不管用不用）】，这是硬性要求，不写会报错：
Error Code: 1248. Every derived table (派生表、导出表) must have its own alias
*/
WHERE total_sales IS NOT NULL
```

复杂的子查询再嵌套进FROM里会让整个查询**看起来过于复杂**，上面这个最好是将子查询结果储存为叫sales_summury的**视图**，然后再直接使用该视图作为来源表，之后会讲。

【第七章】MySQL的基本函数

Essential MySQL Functions (时长33分钟)

内置的用来处理数值、文本、日期等的函数

1. 数值函数

Numeric Functions (2:54)

小结

主要介绍最常用的几个数值函数：ROUND、TRUNCATE、CEILING、FLOOR、ABS、RAND

查看MySQL全部数值函数可谷歌 **'mysql numeric function'（或 'mysql数值函数'）**，第一个就是官方文档。

```
SELECT ROUND(5.7365, 2)  -- 四舍五入
SELECT TRUNCATE(5.7365, 2)  -- 截断
SELECT CEILING(5.2)  -- 天花板函数，大于等于此数的最小整数
SELECT FLOOR(5.6)  -- 地板函数，小于等于此数的最大整数
SELECT ABS(-5.2)  -- 绝对值
SELECT RAND()  -- 随机函数，0到1的随机值
```

2. 字符串函数

String Functions (5:47)

小结

依然介绍**最常用的字符串函数**：

1. LENGTH, UPPER, LOWER
2. TRIM, LTRIM, RTRIM
3. LEFT, RIGHT, SUBSTRING
4. LOCATE, REPLACE, 【CONCAT】

查看全部搜索关键词 **'mysql string functions'**

长度、转大小写：

```
SELECT LENGTH('sky')  -- 字符串字符个数/长度（LENGTH）
SELECT UPPER('sky')  -- 转大写
SELECT LOWER('Sky')  -- 转小写
```

修剪：**用户输入时时常多打空格**，下面三个函数用于处理/修剪（trim）字符串前后的空格，L、R 表示 LEFT、RIGHT：

```
SELECT LTRIM('  Sky')
SELECT RTRIM('Sky  ')
SELECT TRIM(' Sky ')
```

切片（提取）：

```
SELECT LEFT('Kindergarden', 4)  -- 取左边（LEFT）4个字符
SELECT RIGHT('Kindergarden', 6)  -- 取右边（RIGHT）6个字符
SELECT SUBSTRING('Kindergarden', 7, 6)
-- 取从第7个开始的长度为6的子串（SUBSTRING）
-- 【注意SQL是从第1个（而非第0个）开始计数的】
-- 【省略第3参数（子串长度）则一直截取到最后】
```

定位：

```
SELECT LOCATE('gar', 'Kindergarden')  -- 定位（LOCATE）首次出现的位置
-- 【没有的话返回0（其他编程语言大多返回-1，可能因为索引是从0开始的）】
-- 【这个定位/查找函数依然是不区分大小写的】
```

替换：

```
SELECT REPLACE('Kindergarten', 'garten', 'garden')
```

连接：

con`catenate v. 连接，连结

```
USE sql_store;

SELECT CONCAT(first_name, ' ', last_name) AS full_name
FROM customers
```

3. MySQL中的日期函数

Date Functions in MySQL (4:08)

小结

本节学基本的处理时间日期的函数，下节课学日期时间的格式化

1. 当前时间函数：NOW, **CURDAT, CURTIME**
2. 提取函数，同时也是时间单位：YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, **DAYNAME, MONTHNAME**
3. SQL标准提取函数：**EXTRACT(单位 FROM 日期时间对象) 如 EXTRACT(YEAR FROM NOW())**

实例
当前时间

```
SELECT NOW()  -- 2020-09-12 08:50:46
SELECT CURDATE()  -- current date, 2020-09-12
SELECT CURTIME()  --  current time, 08:50:46
```

以上函数将返回**时间日期对象**

提取时间日期对象中的元素：

```
SELECT YEAR(NOW())  -- 2020
```

还有MONTH, DAY, HOUR, MINUTE, SECOND。

以上函数均**返回整数**，还有**另外两个返回字符串的**：

```
SELECT DAYNAME(NOW())  -- Saturday
SELECT MONTHNAME(NOW())  -- September
```

标准SQL语句有一个类似的函数EXTRACT()，若需要在不同DBMS中录入代码，最好用EXTRACT()：

```
SELECT EXTRACT(YEAR FROM NOW())
-- 对比：MySQL里是 YEAR(NOW())
```

当然也可以是MONTH, DAY, HOUR ……
总之就是：EXTRACT(单位 FROM 日期时间对象)

练习

返回【今年】的订单

用时间日期函数而非手动输入年份，代码更可靠，不会随着时间的改变而失效

```
USE sql_store;

SELECT *
FROM orders
【WHERE YEAR(order_date) = YEAR(now())】
-- 两次提取“年”元素来比较
```

4. 格式化日期和时间

Formatting Dates and Times (2:14)

小结

`DATE_FORMAT(date, 【format】)` 将date**根据format字符串进行格式化**）。

`TIME_FORMAT(time, format)` 类似于DATE_FORMAT函数，但这里format字符串只能包含用于小时，分钟，秒和微秒的格式说明符。其他说明符产生一个NULL值或0。

日期事件格式化函数应该只是转换日期时间对象的显示格式（另外始终铭记日期时间本质是数值）

方法

很多像这种完全不需要记也不可能记得完，重要的是知道有这么个可以实现这个功能的函数，具体的**格式说明符（Specifiers）**可以需要的时候去查，至少有两种方法：

1. 直接谷歌关键词 如 mysql date format functions, 其实是在官方文档的 12.7 Date and Time Functions 小结里，有两个函数的说明和specifiers表
2. 用软件里的帮助功能，如workbench里的HELP INDEX打开官方文档查询或者右侧栏的 automatic comtext help (其是也是查官方文档，不过是自动的)

实例

```
SELECT DATE_FORMAT(NOW(), '%M %d, %Y')  -- September 12, 2020
--  【格式说明符里，大小写是不同的，这是目前SQL里第一次出现大小写不同的情况】
SELECT TIME_FORMAT(NOW(), '%H:%i %p')  -- 11:07 AM
```

注意

格式说明符里，大小写代表不同的格式，这是目前SQL里第一次出现大小写不同的情况

5. 计算日期和时间

Calculating Dates and Times (3:08)

小结

有时需要对日期事件对象进行运算，如增加一天或算两个时间的差值之类，介绍一些最有用的日期时间计算函数：

1. DATE_ADD, DATE_SUB
2. DATEDIFF
3. TIME_TO_SEC

增加或减少一定的天数、月数、年数、小时数等等

```
SELECT DATE_ADD(NOW(), 【INTERVAL -1 DAY】)
SELECT DATE_SUB(NOW(), INTERVAL 1 YEAR)
```

计算日期差异

```
SELECT DATEDIFF('2019-01-01 09:00', '2019-01-05')
-- -4
-- 会忽略时间部分，【只算日期差异】
-- 再次注意手写日期要加引号
```

借助 TIME_TO_SEC 函数计算时间差异，**TIME_TO_SEC 会计算从 00:00 到某时间经历的秒数**

```
SELECT TIME_TO_SEC('09:00')  -- 32400
SELECT TIME_TO_SEC('09:00') - TIME_TO_SEC('09:02')  -- -120
```

6. IFNULL和COALESE函数

The IFNULL and COALESCE Functions (3:29)

导航

之前讲了基本的处理数值、文本、日期时间的函数，再介绍几个其它的有用的MySQL函数

小结

两个用来替换空值的函数：IFNULL, COALESCE.
前者用来返回**两个值中的首个非空值**，用来替换空值
后者用来返回**一系列值中的首个非空值**，用法更灵活

案例

将orders里shipper.id中的空值替换为'Not Assigned'（未分配）

```sql
USE sql_store;

SELECT
    order_id,
     IFNULL(shipper_id, 'Not Assigned') AS shipper
     -- If expr1 is not NULL, IFNULL() returns expr1; otherwise it returns expr2.
FROM orders
```

将orders里shipper.id中的空值**先替换comments，若comments也为空再替换为'Not Assigned'**（未分配）

```sql
USE sql_store;

SELECT
    order_id,
    COALESCE(shipper_id, comments, 'Not Assigned') AS shipper
    -- Returns the first non-NULL value in the list, or NULL if there are no non-NULLvalues.
FROM orders
```

COALESCE 函数是返回一系列值中的首个非空值，更灵活

coalesce vi. 合并；结合；联合

练习

返回一个有如下两列的查询结果：

1. customer(顾客的全名)
2. phone(**没有的话，显示'Unknown'**)

```sql
USE sql_store;

SELECT
    CONCAT(first_name, ' ', last_name) AS customer,
    IFNULL/COALESCE(phone, 'Unknown') AS phone
FROM customers
```

7. IF函数

The IF Function (4:54)

小结

根据是否满足条件返回不同的值:

`IF(条件表达式，返回值1，返回值2)` 返回值可以是任何东西，数值、文本、日期时间、空值null均可

案例

将订单表中订单按是否是今年的订单分类为 active（活跃）和 archived（存档），之前讲过用UNION法，即用两次查询分别得到今年的和今年以前的订单，添加上分类列再用UNION合并，**这里直接在SELECT里运用IF函数可以更容易**地得到相同的结果

```sql
USE sql_store;

SELECT
    *,
    IF(
    YEAR(order_date) = YEAR(NOW()),
    -- 两次提取‘年’元素并比较
    'Active',
    'Archived') AS category
FROM orders
```

练习

得到包含如下字段的表：

1. product_id
2. name(产品名称)
3. **orders(该产品出现在订单中的次数)**
4. **frequency(根据是否多于一次而分类为'Once'或'Many times')**

```sql
USE sql_store;

SELECT
    product_id,
    name,
    【COUNT(*) AS orders,
    IF(COUNT(*) = 1, 'Once', 'Many times') AS frequency】
FROM products
JOIN order_items USING(product_id)
【GROUP BY product_id】
```

注意因为 JOIN 内连接筛掉了没有订单的商品，所以至少是一次，所以可以用 `IF(COUNT(*) = 1, 'Once', 'Many times') AS frequency`

另外，发现如果想用同级列别名orders怎么都不行，
若写成 `IF(orders = 1, 'Once', 'Many times') AS frequency`
会报错：Error Code: 1054. Unknown column 'orders' in 'field list'
若写成 `IF((SELECT orders) = 1, 'Once', 'Many times') AS frequency`
会报错：Error Code: 1247. Reference 'orders' not supported (reference to group function)

似乎 “SELECT 引用同级列别名” 这种特殊的子查询不能用于分组聚合函数

所以这个IF函数里若要引用同级列表名orders，应该怎么写？

8. CASE运算符

The CASE Operator (5:23)

小结

当分类多于两种时，可以用IF嵌套，也可以用CASE语句，后者可读性更好

CASE语句结构：

```
【CASE 】
   WHEN …… THEN ……
   WHEN …… THEN ……
   WHEN …… THEN ……
   ……
   [ELSE ……]  (ELSE子句是可选的)
【END】
```

案例

不是将订单分两类，而是**分为三类**：今年的是'Active', 去年的是'Last Year', 比去年更早的是'Achived'：

```
USE sql_store;

SELECT
    order_id,
    CASE
        WHEN YEAR(order_date) = YEAR(NOW()) THEN 'Active'
        WHEN YEAR(order_date) = YEAR(NOW()) - 1 THEN 'Last Year'
        WHEN YEAR(order_date) < YEAR(NOW()) - 1 THEN 'Achived'
        [ELSE 'Future']
    END AS 'category'
FROM orders
```

ELSE 'Future' 是可选的，实验发现**若分类不完整**，比如只写了今年和去年的两个分类条件，则不在这两个分类的记录的category字段会**得到是null（当然）**.

练习

得到包含如下字段的表：customer, points, category（根据积分<2k，2k~3k（包含两端），>3k分为青铜白银和黄金用户）

之前也是用过**UNION法，分别查询加分类字段再合并，很麻烦。**

```
USE sql_store;

SELECT
    【CONCAT(first_name, ' ', last_name) AS customer】,
    points,
    【CASE
        WHEN points < 2000 THEN 'Bronze'
        WHEN points 【BETWEEN 2000 AND 3000】 THEN 'Silver'
        -- 注意书写 “判断条件” 时各种写法的运用
        WHEN points > 3000 THEN 'Gold'
        -- ELSE null
    END AS category】
FROM customers
ORDER BY points DESC
```

其实也可以用IF嵌套，但感觉没有CASE语句结构清晰、可读性好

```sql
SELECT
    CONCAT(first_name, ' ', last_name) AS customer,
    points,
    IF(points < 2000, 'Bronze',
        IF(points BETWEEN 2000 AND 3000, 'Silver',
            IF(points > 3000, 'Gold', null))) AS category
FROM customers
ORDER BY points DESC
```

其实分类条件可以进一步**简化**如下：

```sql
    CASE
        WHEN points < 2000 THEN 'Bronze'
        WHEN points <= 3000 THEN 'Silver'
        ELSE 'Gold'
    END AS category
```

或

```sql
    IF(points < 2000, 'Bronze',
        IF(points <= 3000, 'Silver', 'Gold')) AS category
```

结果是一样的，更简洁。

但有时候像前面那样写的虽然冗余但详细一点，可以提高可读性。

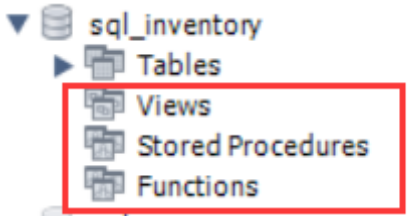
专栏介绍和总目录
数据概要

专栏介绍和总目录
数据概要

第三部分：提高效率——视图、存储过程、函数

导航

八、九、十 三个章节讲的是如何通过 视图、储存过程和函数 来自动化可复用的查询和功能模块以提高效率，与下图红框里的三个条目一一对应。



【第八章】视图

Views (时长18分钟)

1. 创建视图

Creating Views (5:36)

小结

就是创建虚拟表，自动化一些重复性的查询模块，简化各种复杂操作（包括复杂的子查询和连接等）

注意视图虽然可以像一张表一样进行各种操作，但**并没有真正储存数据**，数据仍然储存在原始表中，视图**只是储存起来的模块化的查询结果（会随着原表数据的改变而改变），是为了方便和简化后续进一步操作而储存起来的虚拟表。**

案例

创建sales_by_client视图

```
USE sql_invoicing;

【CREATE VIEW sales_by_client AS】
    SELECT
        client_id,
        name,
        SUM(invoice_total) AS total_sales
    FROM clients c
    JOIN invoices i USING(client_id)
    GROUP BY client_id, name;
    -- 【虽然实际上这里GROUP BY加不加上name都一样，但一般把选择子句中出现的所有非聚合列都加入到分类依据中比较好，在有些DBMS里不这样做会报错】
```

若要**删掉该视图用 【DROP VIEW sales_by_client】 或右键**

创建视图后可就**当作sql_invoicing库下一张表一样**进行各种操作

```
USE sql_invoicing;

SELECT
    s.name,
    s.total_sales,
    phone
FROM sales_by_client s
JOIN clients c USING(client_id)
WHERE s.total_sales > 500
```

练习

创建一个客户差额表视图，可以看到客户的id，名字以及差额（发票总额-支付总额）

```sql
USE sql_invoicing;

CREATE VIEW clients_balance AS
    SELECT
        client_id,
        c.name,
        SUM(invoice_total - payment_total) AS balance
    FROM clients c
    JOIN invoices USING(client_id)
    GROUP BY client_id
```

2. 更新或删除视图

Altering or Dropping Views (2:52)

小结

修改视图可以先 DROP 再 CREATE，但最好是用 **CREATE OR REPLACE**

视图的查询语句可以在该视图的设计模式（点击扳手图标）下查看和修改，但最好是保存为sql文件并放在源码控制妥善管理

案例

想在上一节的顾客差额视图的查询语句最后**加上按差额降序排列**

法1. 先删除再重建

```sql
USE sql_invoicing;

DROP VIEW clients_balance;
-- 若不存在这个视图，用DROP会报错，最好加上 IF EXISTS，后面会讲

CREATE VIEW clients_balance AS
    ……
    ORDER BY balance DESC
```

法2. 用REPLACE关键字，即用 【CREATE OR REPLACE】 VIEW clients_balance AS, 这个比较通用，不管现在这个视图现在是否已经存在都不会出问题，推荐使用种这种方式。（法1 若能加上 IF EXISTS 其实也一样好，而且因为将删除和重建分成了两步，有时情形下甚至是更好的选择）

```sql
USE sql_invoicing;

【CREATE OR REPLACE VIEW clients_balance AS】
    ……
    ORDER BY balance DESC
```

方法

如何保存视图的原始查询语句?

法1.

（推荐方法） 将原始查询语句保存为与视图名同名的 clients_balance.sql 文件并放在 views 文件夹下，然后将这个文件夹放在源码控制下（put these files under source control），通常放在git repository（仓库）里与其它人共享，团队其他人因此能在自己的电脑上重建这个数据库

法2.

若丢失了原始查询语句，要修改的话可点击视图的扳手按钮打开设计模式，可看到如下被MySQL处理了的查询语句

MySQL在前面加了些用户主机名之类的东西并且在所有库名表名字段名外套上反引号防止名称冲突（当对象名和MySQL里的关键字相同时确保被当作对象名而不是关键字），但这都不影响

直接做我们需要的修改，如加上`ORDER BY balance DESC` 然后点apply就行了

```
CREATE
    ALGORITHM = UNDEFINED
    DEFINER = `root`@`localhost`
    SQL SECURITY DEFINER
VIEW `clients_balance` AS
    SELECT
        `c`.`client_id` AS `client_id`,
        `c`.`name` AS `name`,
        SUM((`invoices`.`invoice_total` - `invoices`.`payment_total`)) AS `balance`
    FROM
        (`clients` `c`
        JOIN `invoices` ON ((`c`.`client_id` = `invoices`.`client_id`)))
    GROUP BY `c`.`client_id`
    【ORDER BY balance DESC】
```

法2是没有办法的办法，当然最好还是将创建views的原始sql代码保存为sql文件并放入源码控制

3. 可更新视图

Updatable Views (5:12)

小结

如果一个视图的原始查询语句中**没有如下元素：**

1. **GROUP BY 分组、聚合函数、HAVING 分组聚合后筛选 (即分组聚合筛选三兄弟)**
2. **DISTINCT 去重**
3. **UNION 纵向合并**

则该视图是可更新视图（Updatable Views），可以用在 INSERT DELETE UPDATE 语句中进行增删改，否则只能用在 SELECT 语句中进行查询。（1好理解，2和3需要记一下）

另外，增（INSERT）还要满足附加条件：视图必须包含**底层原表的所有必须字段**（也很好理解）

总之，一般通过原表修改数据，但当出于安全考虑或其他原因没有某表的直接权限时，可以通过视图来修改数据，前提是视图是可更新的。

之后会讲关于安全和权限的内容

案例

创建视图（新虚拟表）invoices_with_balance（带差额的发票记录表）

```
USE sql_invoicing;

CREATE OR REPLACE VIEW invoices_with_balance AS
SELECT
    /*
```

```
    这里有个小技巧，要插入表中的多列列名时，
    可从左侧栏中【连选并拖入】相关列
    */
    invoice_id,
    number,
    client_id,
    invoice_total,
    payment_total,
    invoice_date,
    invoice_total - payment_total AS balance,
    -- 新增列
    due_date,
    payment_date
FROM invoices
WHERE (invoice_total - payment_total) > 0
/*
这里不能用列别名balance，会报错说不存在，
必须用原列名的表达式
之前从执行顺序解释过
*/
```

该视图满足条件，是可更新视图，故可以增删改：

1. 删：

删掉id为1的发票记录

```
DELETE FROM invoices_with_balance
WHERE invoice_id = 1
```

注意：这其实是通过视图把原表中的1号订单记录删除了，而由于视图只是储存起来的自动化的查询，所以视图里的1号订单记录也自然随之消失了

可通过如下实验验证这一点，把原表的2号订单改为1号，发现视图里的2号订单也随之改为了1号

```
UPDATE invoices
SET invoice_id = 1
WHERE invoice_id = 2;
```

2. 改：

将2号发票记录的期限延后两天

```
UPDATE invoices_with_balance
SET due_date = DATE_ADD(due_date, INTERVAL 2 DAY)
WHERE invoice_id = 2
```

3. 增：

在视图中用 INSERT 新增记录的话还有另一个前提，即视图必须包含其底层所有原始表的所有必须字段（这很好理解）
例如，若这个invoices_with_balance视图里没有invoice_date字段（invoices中的必须字段），那就无法通过该视图向invoices表新增记录，因为**invoices表不会接受必须字段 invoice_date 为空的记录**

4. WITH CHECK OPTION 子句

THE WITH CHECK OPTION Clause (2:18)

小结

在视图的原始创建语句的最后加上 `WITH CHECK OPTION` 可以防止执行那些会让视图中某些行（记录）消失的修改语句。

案例

接前面的invoices_with_balance视图的例子，该视图与原始的orders表相比增加了balance (invouce_total - payment_total)列，且只显示balance大于0的行（记录），若将某记录（如2号订单）的payment_total改为和invouce_total相等，则balance为0，该记录会从视图中消失：

```
UPDATE invoices_with_balance
SET payment_total = invoice_total
WHERE invoice_id = 2
```

更新后会发现 invoices_with_balance 视图里2号订单消失。

但在视图原始创建语句的最后加入 `WITH CHECK OPTION` 后，对3号订单执行类似上面的语句后会报错：

```
USE sql_invoicing;

CREATE OR REPLACE VIEW invoices_with_balance AS
……
WHERE (invoice_total - payment_total) > 0
【WITH CHECK OPTION】;

UPDATE invoices_with_balance
SET payment_total = invoice_total
WHERE invoice_id = 3;
-- Error Code: 1369. CHECK OPTION failed 'sql_invoicing.invoices_with_balance'
```

5. 视图的其他优点

Other Benefits of Views (2:37)

小结

三大优点：

简化查询、增加抽象层和减少变化的影响、数据安全性

具体来讲：

1. **（首要优点）简化查询** simplify queries
2. **增加抽象层，减少变化的影响** Reduce the impact of changes：视图给表增加了一个抽象层（模块化），这样如果数据库设计改变了（如一个字段从一个表转移到了另一个表），只需修改视图的查询语句使其能保持原有查询结果即可，不需要修改使用这个视图的那几十个查询。相反，如果没有视图这一层的话，所有查询将直接使用指向原表的原始查询语句，这样一旦更改原表设计，就要相应地更改所有的这些查询。
3. **限制对原数据的访问权限** Restrict access to the data：在视图中可以对原表的行和列进行筛选，这样如果你禁止了对原始表的访问权限，用户只能通过视图来修改数据，他们就无法修改视图中未返回的那些字段和记录。但需注意，这并不像听上去这么简单，需要良好的规划，否则最后可能搞得一团乱。

了解这些优点，但不要盲目将他们运用在所有的情形中。

导航

下一章将讲解储存过程和函数

【第九章】存储过程

Stored Procedures (时长48分钟)

1. 什么是存储过程

What are Stored Procedures (2:18)

小结

存储过程三大作用：（其实视图、储存过程、函数作用都很**类似**）

1. **储存和管理SQL代码 Store and organize SQL （置于数据库中，与应用层分离，同视图和函数一样，都是增加抽象层，作为模块化抽象工具）**
2. **数据安全 Data security（其实是1的结果）**
3. **性能优化 Faster execution**
4.

导航

之前学了增删改查，复杂查询以及如何运用视图来简化查询。

假设你要开发一个使用数据库的应用程序，你应该将SQL语句写在哪里呢?

如果将SQL语句内嵌在应用程序的代码里，将使其混乱且难以维护，所以应该将SQL代码和应用程序代码分开，将SQL代码储存在所属的数据库中，具体来说，是放在储存过程（stored procedure）和函数中。

储存过程是一个包含SQL代码模块的数据库对象，在应用程序代码中，我们调用储存过程来获取和保存数据（get and save the data）。也就是说，我们使用储存过程来储存和管理SQL代码。

使用储存程序还有**另外两个好处。**首先，大部分DBMS会对储存过程中的代码进行一些**优化**，因此有时储存过中的SQL代码执行起来会更快。

此外，就像视图一样，储存过程能加强**数据安全。比如，我们可以移除对所有原始表的访问权限，让各种增删改的操作都通过储存过程来完成，然后就可以决定谁可以执行何种储存过程，用以限制用户对我们数据的操作范围，例如，防止特定的用户删除数据。**

所以，储存过程很有用，本章将学习如何创建和使用它。

2. 创建一个存储过程

Creating a Stored Procedure (5:34)

小结

注意以下结构中共有3条语句：改分隔符，定义储存过程，再把分隔符改回来

```
【DELIMITER $$】
-- delimiter 定界符；分隔符

CREATE PROCEDURE 储存过程名()
    BEGIN
        ……;
        ……;
        ……;
    END【$$】

【DELIMITER ;】
```

改分隔符原因参考官方文档：

> The example uses the mysql client delimiter command to change the statement delimiter from ; to // （or $$）while the procedure is being defined. This enables the ; delimiter used in the procedure body to be passed through to the server rather than being interpreted by mysql itself.

实例

创造一个get_clients()储存过程

```
CREATE PROCEDURE get_clients()
-- 括号内可传入参数，之后会讲
-- 储存过程名用小写单词和下划线表示，这是约定熟成的做法
    BEGIN
        SELECT * FROM clients;
    END
```

BEGIN 和 END 之间包裹的是此储存过程（PROCEDURE）的主体（body），主体里可以有多个语句，但每个语句都要以；结束，包括最后一个。

为了将储存过程主体内部的语句分隔符与SQL本身执行层面的语句分隔符；区别开，要先用 DELIMITER(分隔符)关键字暂时将 SQL 语句的默认分隔符改为双美元符号 $$ 或双斜杠 // 等，创建储存过程结束后再改回来。注意创建储存过程本身也是一个完整 SQL 语句，所以别忘了在 END 后要加一个暂时语句分隔符 $$

注意

储存过程主体中所有语句都要以；结尾并且因此要暂时修改SQL本身的默认分隔符，**这些都是MySQL的特性，在SQL Server等就不需要这样**

```
USE sql_invoicing;

DELIMITER $$

CREATE PROCEDURE get_clients()
    BEGIN
        SELECT * FROM clients;
    END$$

DELIMITER ;
```

调用此程序：

法1. 点击闪电按钮

法2. 用CALL关键字

```sql
USE sql_invoicing;
CALL get_clients()

【或】

CALL sql_invoicing.get_clients()
```

注意

上面讲的是如何在 SQL 中调用储存过程，但更多的时候其实是要在应用程序代码（可能是C#、JAVA 或 Python 编写的）中调用。

练习

创造一个储存过程 get_invoices_with_balance（取得有差额（差额不为0）的发票记录）

```sql
DROP PROCEDURE get_invoices_with_balance;
-- 注意DROP语句删除储存过程时直接指明要删除的储存过程名就行了，不涉及参数的问题所以不需要带括号
-- 但创建CREATE和调用CALL储存过程时需要加括号指明储存过程有没有参数或有什么参数

DELIMITER $$

CREATE PROCEDURE get_invoices_with_balance()
    BEGIN
        SELECT *
        FROM invoices_with_balance
        -- 这是之前创造的视图
        -- 用视图好些，因为有现成的balance列
        WHERE balance > 0;
    END$$

DELIMITER ;

CALL get_invoices_with_balance();
```

3. 使用MySQL工作台创建存储过程

Creating Procedures Using MySQLWorkbench (1:21)

也可以用点击的方式创造储存过程，右键选择 Create Stored Procedure，填空，Apply。这种方式Workbench会帮你处理暂时修改分隔符的问题

这种方式一样可以储存SQL文件

事实证明，mosh很喜欢用这种方式，后面基本都是用这种方式创建储存过程

4. 删除存储过程

Dropping Stored Procedures (2:09)

这一节学习删除储存过程，这在发现储存过程里有错误需要重建时很有用。

实例

一个创建储存过程（get_clients）的标准模板

```sql
USE sql_invoicing;

【DROP PROCEDURE IF EXISTS get_clients;
-- 注意加上IF EXISTS，以免因为此储存过程不存在而报错】

DELIMITER $$

    CREATE PROCEDURE get_clients()
        BEGIN
            SELECT * FROM clients;
        END$$

DELIMITER ;

CALL get_clients()

```

最佳实践

同视图一样，最好把删除和创建每一个储存过程的代码也储存在不同的SQL文件中，并把这样的文件放在Git这样的源码控制下，这样就能与其它团队成员共享Git储存库。他们就能**在自己的机器上重建（复现）**数据库以及该数据库下的所有的视图和储存过程

如上面那个实例，可储存在stored_procedure文件夹（之前已有views文件夹）下的get_clients.sql文件。当你把所有这些脚本放进源码控制，你能随时回来查看你对数据库对象所做的改动。

5. 参数

Parameters (5:26)

小结

```sql
CREATE PROCEDURE 储存过程名
(
    参数1 数据类型,
    参数2 数据类型,
    ……
)
BEGIN
……
END
```

导航

学完了如何创建和删除储存过程，这一节学习如何给储存过程添加参数

通常我们使用参数来给储存过程传值，但我们也可以用参数（或者说"变量"）来获取调用储存程序的结果值，第二个我们之后再讲

案例

创建储存过程get_clients_by_state，可返回特定州的顾客

```sql
USE sql_invoicing;

DROP PROCEDURE IF EXISTS get_clients_by_state;

```

```
DELIMITER $$

CREATE PROCEDURE get_clients_by_state
(
    state CHAR(2)  -- 参数的数据类型
)
BEGIN
    SELECT * FROM clients c
    WHERE c.state = state;
END$$

DELIMITER ;
```

参数类型一般设定为VARCHAR, 除非能确定参数的字符数

多个参数可用逗号隔开

`WHERE state = state` 是没有意义的，**有两种方法可以区分参数和列名**：一种是取不一样的参数名如p_state或state_param，第二种是像上面一样给表起别名，然后使用带表别名前缀的列名来与参数名区分。

```
CALL get_clients_by_state('CA')
```

注意：不传入'CA'会报错，因为MySQL （里储存程序和函数）的所有参数都是必须参数，报错信息如下：

> Error Code: 1318. Incorrect number of arguments for PROCEDURE sql_invoicing.get_clients_by_state; expected 1, got 0

练习

创建储存过程get_invoices_by_client，通过client_id来获得发票记录

client_id的数据类型设置可以参考原表中该字段的数据类型

```
-- 创建储存过程get_invoices_by_client，通过client_id来获得发票记录
USE sql_invoicing;

DROP PROCEDURE IF EXISTS get_invoices_by_client ;

DELIMITER $$

CREATE PROCEDURE get_invoices_by_client
(
    client_id INT  -- 为何不写INT(11)?
)
BEGIN
    SELECT *
    FROM invoices i
    WHERE i.client_id = client_id;
END$$

DELIMITER ;

CALL get_invoices_by_client(1)
```

Mosh创建和调用都直接用的点击法

6. 带默认值的参数

Parameters with Default Value (8:18)

小结

给参数设置默认值，主要是运用条件语句块和替换空值函数

回顾

SQL中的条件类语句：

1. 替换空值 `IFNULL(值1，值2)`
2. IF函数 `IF(条件表达式，返回值1，返回值2)`
3. IF语句

```
IF 条件表达式 THEN
    语句1;
    语句2;
    ……;
[ELSE] (可选)
    语句1;
    语句2;
    ……;
END IF;
```

4. CASE语句：

```
CASE
    WHEN …… THEN ……
    WHEN …… THEN ……
    WHEN …… THEN ……
    ……
    [ELSE ……]  (ELSE子句是可选的)
END
```

案例1

把 get_clients_by_state 储存过程的默认参数设为'CA'，即**默认查询加州的客户**

```
USE sql_invoicing;

DROP PROCEDURE IF EXISTS get_clients_by_state;

DELIMITER $$

CREATE PROCEDURE get_clients_by_state
(
    state CHAR(2)
)
BEGIN
    【IF state IS NULL THEN
        SET state = 'CA';
        --【注意别忽略SET，SQL里单个等号'='是比较操作符而非赋值操作符】
    END IF;】
    SELECT * FROM clients c
    WHERE c.state = state;
END$$

```

```
DELIMITER ;
```

调用

```
CALL get_clients_by_state(【NULL】)
```

注意要调用储存过程并使用其默认值时时要传入参数 NULL ， MySQL不允许不传参数。

案例2

将 get_clients_by_state 储存过程设置为**默认选取所有顾客**

法1. 用IF条件语句块实现

```
……
BEGIN
    IF state IS NULL THEN
        SELECT * FROM clients c;
    ELSE
        SELECT * FROM clients c
        WHERE c.state = state;
    END IF;
END$$
……
```

法2. 用IFNULL替换空值函数实现

```
……
BEGIN
    SELECT * FROM clients c
    【WHERE c.state = IFNULL(state, c.state)】
END$$
……
```

若参数为NULL，则返回c.state，利用 c.state = c.state 永远成立来返回所有顾客，思路很巧妙。

练习

创建一个叫 get_payments 的储存过程，包含 client_id 和 payment_method_id 两个参数，数据类型分别为 INT(4) 和 TINYINT(1)（1字节整数，能存0~255，之后会讲数据类型，好奇可以谷歌 'mysql int size'），默认参数设置为返回所有记录

一个为你的工作预热的练习

```
USE sql_invoicing;

DROP PROCEDURE IF EXISTS get_payments;

DELIMITER $$

CREATE PROCEDURE get_payments
(
    client_id INT,  -- 不用写成INT(4)
    payment_method_id TINYINT
)
BEGIN
    SELECT * FROM payments p
    【WHERE
```

```
        p.client_id = IFNULL(client_id, p.client_id) AND
        p.payment_method = IFNULL(payment_method_id, p.payment_method);】
        -- 另外，再次小心这种实际工作中各表相同字段名称不同的情况
END$$

DELIMITER ;
```

调用：

1. 所有支付记录

```
CALL get_payments(NULL, NULL)
```

2. **1号顾客的所有记录**

```
CALL get_payments(1, NULL)
```

3. 3号支付方式的所有记录

```
CALL get_payments(NULL, 3)
```

4. 5号顾客用2号支付方式的所有记录

```
CALL get_payments(5, 2)
```

注意

注意一个区别：

1. **Parameter 形参**（形式参数）：创建储存过程中用的占位符，如 client_id、payment_method_id
2. **Argument 实参**（实际参数）：调用时实际传入的值，如 1、3、5、NULL

7. 参数验证

Parameter Validation (6:40)

小结

储存过程除了可以查，也可以增删改，但修改数据前最好先进行参数验证以防止不合理的修改

主要利用 IF 条件语句和 `SIGNAL SQLSTATE MESSAGE_TEXT` 关键字

具体来说是在储存过程的主体开头加上这样的语句：

```
IF 错误参数条件表达式 THEN
    SIGNAL SQLSTATE '错误类型'
        [SET MESSAGE_TEXT = '关于错误的补充信息'] (可选)
```

案例

创建一个 make_payment 储存过程，含 invoice_id, payment_amount, payment_date 三个参数

（Mosh还是喜欢通过右键Create Stored Procedure地方式创建，不用考虑暂时改分隔符的问题，更简便）

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `make_payment`(
    invoice_id INT,
    payment_amount DECIMAL(9, 2),
    /*
    9是精度（存储的有效位数），  2是小数位数。
    见：https://dev.mysql.com/doc/refman/8.0/en/fixed-point-types.html
    */
    payment_date DATE
)
BEGIN
    UPDATE invoices i
    SET
        i.payment_total = payment_amount,
        i.payment_date = payment_date
    WHERE i.invoice_id = invoice_id;
END
```

为了防止传入像 -100 的 payment_total 这样不合理的参数，要再**增加一段参数验证语句**，利用的是 IF 条件语句加 SIGNAL关键字，和其他编程语言中的**抛出异常**等类似

具体的错误类型可通过谷歌 "sqlstate error" 查阅（推荐使用IBM的那个表），这里是 '22 Data Exception' 大类中的 '22003 A numeric value is out of range.' 类型

注意还添加了 MESSAGE_TEXT 以提供给用户参数错误的更具体信息。

```
……
BEGIN
    【IF payment_amount <= 0 THEN
        SIGNAL SQLSTATE '22003'
            SET MESSAGE_TEXT = 'Invalid payment amount';
    END IF;】

    UPDATE invoices i
    ……
END
```

现在传入 负数的 payment_amount 就会报错 'Error Code: 1644. Invalid payment amount'

注意

过犹不及（"Too much of a good thing is a bad thing"），加入过多的参数验证会让代码过于复杂难以维护，像payment_amount 非空这样的验证就不需要添加因为 payment_amount 字段本身就不允许空值因此 MySQL 会自动报错。

参数验证工作更多的应该在**应用程序端**接受用户输入数据时就检测和报告，那样更快也更有效。储存过程里的参数验证只是在有人越过应用程序直接访问储存过程时作为**最后的防线**。这里只应该写那些**最关键和必要的参数验证**。

8. 输出参数

Output Parameters (3:55)

小结

输入参数是用来给储存过程传入值的，我们也可以用输出参数（变量）来获取储存程序的值

具体是在参数的前面加上 **OUT 关键字**，然后再 SELECT 后加上 **INTO**.....

调用麻烦，**如无需要，不要多此一举**

案例

创造 get_unpaid_invoices_for_client 储存过程，获取某一顾客所有未支付过的发票记录（即满足 payment_total = 0 的发票记录）的数量和总额

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `get_unpaid_invoices_for_client`(
        client_id INT
)
BEGIN
    SELECT COUNT(*), SUM(invoice_total)
    FROM invoices i
    WHERE
        i.client_id = client_id AND
        payment_total = 0;
END
```

调用

```
call sql_invoicing.get_unpaid_invoices_for_client(3);
```

得到3号顾客的 COUNT(*), SUM(invoice_total) 分别为2和286

我们也可以通过输出参数（变量）来获取这些值，修改储存过程，添加两个输出参数 invoice_count 和 invoice_total：

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `get_unpaid_invoices_for_client`(
        client_id INT,
        【OUT】 invoice_count INT,
        OUT invoice_total DECIMAL(9, 2)
        -- 默认是输入参数，输出参数要加OUT前缀
)
BEGIN
    SELECT COUNT(*), SUM(invoice_total)
    【INTO】 invoice_count, invoice_total
    -- SELECT后跟上INTO语句将SELECT选出的值传入输出参数（输出变量）中
    FROM invoices i
    WHERE
        i.client_id = client_id AND
        payment_total = 0;
END
```

调用：**单击闪电按钮调用**，只用输入client_id，得到如下语句结果

```
set @invoice_count = 0;
set @invoice_total = 0;
call sql_invoicing.get_unpaid_invoices_for_client(3, @invoice_count, @invoice_total);
select @invoice_count, @invoice_total;
```

先定义以@前缀表示用户变量，将初始值设为0。（变量（variable）简单讲就是储存单一值的对象）再调用储存过程，将储存过程结果赋值给这两个输出参数，最后再用SELECT查看。

很明显，通过输出参数获取并读取数据有些麻烦，若无充足的原因，不要多此一举。

9. 变量

Variables (4:33)

小结

两种变量:

1. 用户或会话变量 `SET @变量名 = ……`
2. 本地变量 **`DECLARE 变量名 数据类型 [DEFAULT 默认值]`**

用户或会话变量（User or session variable）：

上节课讲过，用 SET 语句并在变量名前加 @ 前缀来定义，将在整个用户会话期间存续，在会话结束断开MySQL连接时才被清空，这种变量主要在调用带输出变量的储存过程时使用，用来传入储存过程作为输出参数来获取结果值。

实例

```
set @invoice_count = 0;
set @invoice_total = 0;
call sql_invoicing.get_unpaid_invoices_for_client(3, @invoice_count, @invoice_total);
select @invoice_count, @invoice_total;
```

本地变量（Local variable）

在储存过程或函数中通过 DECLARE 声明并使用，在函数或储存过程执行结束时就被清空，**常用来执行储存过程（或函数）中的计算**

案例

创造一个 get_risk_factor 储存过程，使用公式 risk_factor = invoices_total / invoices_count * 5

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `get_risk_factor`()
BEGIN
    -- 声明三个本地变量，可设默认值
    【DECLARE risk_factor DECIMAL(9, 2) [DEFAULT 0];
    DECLARE invoices_total DECIMAL(9, 2);
    DECLARE invoices_count INT;】

    -- 用SELECT得到需要的值并用INTO传入invoices_total和invoices_count
    SELECT SUM(invoice_total), COUNT(*)
    【INTO】 invoices_total, invoices_count
    FROM invoices;

    -- 【用SET语句给risk_factor计算赋值】
    【SET】 risk_factor = invoices_total / invoices_count * 5;

    -- 【SELECT展示】最终结果risk_factor
    SELECT risk_factor;
END
```

10. 函数

Functions (6:28)

小结

创建函数和创建储存过程的两点不同

```
1.
RETURNS INTEGER
DETERMINISTIC
READS SQL DATA
MODIFIES SQL DATA
……
2.
RETURN IFNULL(risk_factor, 0);
```

删除

```
DROP FUNCTION [IF EXISTS] 函数名
```

导航

现在已经学了很多内置函数，包括聚合函数和处理数值、文本、日期时间的函数，这一届学习如何创建函数

函数和储存过程的作用非常相似，唯一区别是函数只能返回单一值而不能返回多行多列的结果集，当你只需要返回一个值时就可以创建函数。

案例

再上一节的储存过程 get_risk_factor 的基础上，创建函数 get_risk_factor_for_client，计算特定顾客的 risk_factor

还是用右键 Create Function 来简化创建

创建函数的语法和创建储存过程的语法极其相似，**区别只在两点：**

1. **参数设置和 body 主体之间，有一段确定返回值类型以及函数属性的语句段**
2. **最后是返回（RETURN）值而不是查询（SELECT）值**

另外，**关于函数属性**的说明：

1. DETERMINISTIC 决定性的，唯一输入决定唯一输出，和数据的改动更新无关，比如税收是订单总额的10%，则以订单总额为输入、税收为输出的函数就是决定性的，但这里每个顾客的 risk_factor 会随着其发票记录的增加更新而改变，所以不是 DETERMINISTIC 的（？）
2. READS SQL DATA 需要用到 SELECT 语句进行数据读取的函数，几乎所有函数都满足
3. MODIFIES SQL DATA 函数中有 增删改 或者说有 INSERT DELETE UPDATE 语句，这个例子不需要

```
CREATE DEFINER=`root`@`localhost` 【FUNCTION】 `get_risk_factor_for_client`
(
    client_id INT
)
【RETURNS INTEGER
-- DETERMINISTIC
READS SQL DATA
-- MODIFIES SQL DATA】
BEGIN
    DECLARE risk_factor DECIMAL(9, 2) DEFAULT 0;
    DECLARE invoices_total DECIMAL(9, 2);
    DECLARE invoices_count INT;

    SELECT SUM(invoice_total), COUNT(*)
    INTO invoices_total, invoices_count
```

```
    FROM invoices i
    WHERE i.client_id = client_id;
    -- 注意不再是整体risk_factor而是特定顾客的risk_factor

    SET risk_factor = invoices_total / invoices_count * 5;
    【RETURN IFNULL(risk_factor, 0);】
END
```

注意考虑周全严谨一些：有些顾客没有发票记录，NULL乘除结果还是NULL，所以最后用 IFNULL 函数将这些人的 risk_factor 替换为 0

调用案例：

```
SELECT
    client_id,
    name,
    【get_risk_factor_for_client(client_id) AS risk_factor】
    -- 其实是逐行调用
FROM clients
```

删除，还是用DROP

```
DROP FUNCTION [IF EXISTS] get_risk_factor_for_client
```

注意

和视图和储存过程一样，也**最好**存入SQL文件并加入源码控制，老生常谈了。

11. 其他约定

Other Conventions (1:51)

有各种各样的命名习惯（包括对函数储存过程的命名习惯以及对更改分隔符的习惯），没有明显的好坏之分，重要的是在一个项目或团队中保持恒定不变，学会**入乡随俗**

A Quick Note

专栏介绍和总目录
数据概要

专栏介绍和总目录
数据概要

第四部分：高阶主题——触发器、事件、事务、并发

【第十章】触发器和事件

Triggers and Events (时长22分钟)

1. 触发器

Triggers (7:31)

小结

触发器是在增删改语句前后自动执行的一段SQL代码（A block of SQL code that automatically gets executed before or after an insert,update or delete statement）通常我们使用触发器来保持数据的一致性

创建触发器的语法要点：命名三要素，触发条件语句和触发频率语句，主体中 OLD/NEW 的使用

案例

在 sql_invoicing 库中，发票表中同一个发票记录可以对应付款表中的多次付款记录，发票表中的付款总额应该等于这张发票所有付款记录之和，为了保持数据一致性，可以通过触发器让每一次付款表中新增付款记录时，发票表中相应发票的付款总额（payement_total）自动增加相应数额

语法上，和创建储存过程等类似，要暂时更改分隔符，用CREATE关键字，用BEGIN和END包裹的主体

```
DELIMITER $$

CREATE 【TRIGGER】 【payments_after_insert】 -- 命名习惯
    【AFTER INSERT ON payments  -- 触发条件语句
    FOR EACH ROW】  -- 触发频率语句
BEGIN
    UPDATE invoices
    【SET payment_total = payment_total + NEW.amount
    WHERE invoice_id = NEW.invoice_id;】
    -- 注意 NEW/OLD 的使用
END$$

DELIMITER ;
```

几个关键点：

1. **命名习惯**（三要素）：触发表_before/after(表示SQL语句执行之前或之后触发)_触发的SQL语句类型
2. **触发条件语句**：BEFORE/AFTER INSERT/UPDATE/DELETE ON 触发表
3. **触发频率语句**：这里 FOR EACH ROW 表明每一个受影响的行都会启动一次触发器。其它有的DBMS还支持表级别的触发器，即不管插入一行还是五行都只启动一次触发器，**到Mosh录制为止MySQL还不支持这样的功能**

4. 主体：主体里可以对各种表的数据进行修改以保持数据一致性，但注意唯一不能修改的表是触发表，否则会引发无限循环（**“触发器自燃”**），**主体中最关键的是使用 NEW/OLD 关键字来指代受影响的新/旧行**（若INSERT用NEW，若DELETE用OLD，若UPDATE似乎理论上两个都可以用，但应该业主要用NEW）**并可跟 '点+字段' 地方式来引用这些行的相应属性**

测试：往payments里新增付款记录，发现invoices表对应发票的付款总额确实相应更新

```
INSERT INTO payments
VALUES (DEFAULT, 5, 3, '2019-01-01', 10, 1)
```

练习

创建一个和刚刚的触发器作用刚好相反的触发器，每当有付款记录被删除时，自动减少发票表中对应发票的付款总额

```
DELIMITER $$

CREATE TRIGGER payments_after_delete
    AFTER DELETE ON payments
    FOR EACH ROW
BEGIN
    UPDATE invoices
    SET payment_total = payment_total - 【OLD.amount】
    WHERE invoice_id = 【OLD.invoice_id;】
END$$

DELIMITER ;
```

测试：删掉付款表里刚刚的那个给3号发票支付10美元的付款记录，则果然发票表里3号发票的付款总额相应减少10美元.

```
DELETE FROM payments
WHERE payment_id = 9
```

2. 查看触发器

Viewing Triggers (1:20)

用以下命令来查看已存在的触发器及其各要素

```
SHOW TRIGGERS
```

如果之前创建时遵行了三要素命名习惯，这里也可以用 LIKE 关键字来筛选特定表的触发器

```
SHOW TRIGGERS LIKE 'payments%'
```

3. 删除触发器

Dropping Triggers (0:52)

和删除储存过程的语句一样

```sql
DROP TRIGGER [IF EXISTS] payments_after_insert
-- IF EXISTS 是可选的，但一般最好加上
```

最佳实践

最好将删除和创建数据库/视图/储存过程/触发器的语句放在同一个脚本中（即将删除语句放在创建语句前，DROP IF EXISTS + CREATE，用于创建或更新数据库/视图/储存过程/触发器，等效于 CREATE OR REPLACE，但分成了两个语句）并将脚本录入源码库中，**这样不仅团队都可以创建相同的数据库，还都能查看数据库的所有修改历史（查看每个版本）**

```sql
DELIMITER $$

DROP TRIGGER IF EXISTS payments_after_insert;
/*
实验了一下好像这里用$$也可以，
但为什么可以用;啊?
*/

CREATE TRIGGER payments_after_insert
    AFTER INSERT ON payments
    FOR EACH ROW
BEGIN
    UPDATE invoices
    SET payment_total = payment_total + NEW.amount
    WHERE invoice_id = NEW.invoice_id;
END$$

DELIMITER ;
```

4. 使用触发器进行审核

Using Triggers for Auditing (4:52)

导航

之前已经学习了如何用触发器来保持数据一致性，触发器的另一个常见用途是为了审核的目的将修改数据的操作记录在日志里。

小结

建立一个审核表（日志表）以记录谁在什么时间做了什么修改，实现方法就是在触发器里加上创建日志记录的语句，日志记录应包含修改内容信息和操作信息两部分。

案例

用 create-payments-table.sql 创建 payments_audit 表，记录所有对 payments 表的增删操作，注意该表包含 client_id, date, amount 字段来记录修改的内容信息（方便之后恢复操作，如果需要的话）和 action_type, action_date 字段来记录操作信息。注意这是个简化了的 audit 表以方便理解。

具体实现方法是，重建在 payments 表里的的增删触发器 payments_after_insert 和 payments_after_delete，在触发器里加上往 payments_audit 表里添加日志记录的语句

具体而言：

往 payments_after_insert 的主体里加上这样的语句：

```
INSERT INTO payments_audit
VALUES (NEW.client_id, NEW.date, NEW.amount, 'insert', NOW());
```

往 payments_after_delete 的主体里加上这样的语句：

```
INSERT INTO payments_audit
VALUES (OLD.client_id, OLD.date, OLD.amount, 'delete', NOW());
```

测试：

```
-- 增:
INSERT INTO payments
VALUES (DEFAULT, 5, 3, '2019-01-01', 10, 1);

-- 删:
DELETE FROM payments
WHERE payment_id = 10
```

发现 payments_audit 表里果然多了两条记录以记录这两次增和删的操作

注意

实际运用中不会为数据库中的每张表建立一个审核表，相反，会有一个整体架构，通过一个总审核表来记录，这在之后设计数据库中会讲到。

导航

下节课学习事件

5. 事件

Events (4:33)

事件是一段根据计划执行的代码，可以执行一次，或者按某种规律执行，比如每天早上10点或每月一次

通过事件我们可以自动化数据库维护任务，比如删除过期数据、将数据从一张表复制到存档表 或者 汇总数据生成报告，所以事件十分有用。

首先，需要打开MySQL**事件调度器（event_scheduler），这是一个时刻寻找需要执行的事件的后台程序**

查看MySQL所有**系统变量**：

```
SHOW VARIABLES;
SHOW VARIABLES LIKE 'event%';
-- 使用 LIKE 操作符查找以event开头的系统变量
-- 【通常为了节约系统资源而默认关闭】
```

用SET语句开启或关闭,不想用事件时可关闭以节省资源，这样就不会有一个不停寻找需要执行的事件的后台程序

```
SET GLOBAL event_scheduler = ON/OFF
```

案例

创建这样一个 yearly_delete_stale_audit_row 事件，每年删除过期的（超过一年的）日志记录

```
DELIMITER $$

CREATE 【EVENT】 yearly_delete_stale_audit_row
-- stale adj. 陈腐的；不新鲜的

-- 设定事件的执行计划：
ON SCHEDULE
    EVERY 1 YEAR [STARTS '2019-01-01'] [ENDS '2029-01-01']

-- 主体部分：（注意 DO 关键字）
DO BEGIN
    DELETE FROM payments_audit
    WHERE action_date < NOW() - INTERVAL 1 YEAR;
END$$

DELIMITER ;
```

关键点：

1. 命名：**用时间间隔（频率）开头，可以方便之后分类检索**，时间间隔（频率）包括 once/hourly/daily/monthly/yearly 等等
2. 执行计划：

- 规律性周期性执行用 EVERY 关键字，可以是 EVERY 1 HOUR / EVERY 2 DAY 等等
- 若只执行一次就用 AT 关键字，如：`AT '2019-05-01'`
- 开始(STARTS)和结束(ENDS)时间都是可选的

补充：

3. `NOW() - INTERVAL 1 YEAR` **等效于** `DATE_ADD(NOW(), INTERVAL -1 YEAR)` 或 `DATE_SUB(NOW(), INTERVAL 1 YEAR)`，但感觉不用DATEADD/DATESUB函数，**直接相加减（但INTERVAL关键字还是要用）还简单直白点**

小结

查看和开启/关闭事件调度器（event_scheduler）：

```
SHOW VARIABLES LIKE 'event%';
SET GLOBAL event_scheduler = ON/OFF
```

创建事件：

```
……
CREATE EVENT 以频率打头的命名
ON SCHEDULE
    EVERY 时间间隔 / AT 特定时间 [STARTS 开始时间][ENDS 结束时间]
DO BEGIN
……
END$$
……
```

6. 查看、删除和更改事件

Viewing, Dropping and Altering Events (2:04)

小结

SHOW、DROP、ALTER、ENABLE、DISABLE

导航

上节课讲的是创建事件，即“增”，这节课讲如何“查、删、改”，**说来说去其实任何对象都是这四种操作**

查（SHOW）和删（DROP）和之前的类似：

```
SHOW EVENTS
-- 可看到各个数据库的事件
SHOW EVENTS [LIKE 'yearly%'];
-- 【之前命名以时间间隔开头的好处：方便筛选】
DROP EVENT IF EXISTS yearly_delete_stale_audit_row;
```

“改”要特殊一些，这里首次用到 ALTER 关键字，而且有两种用法：

1. 如果要修改事件内容（包括执行计划和主体内容），直接**把 ALTER 当 CREATE 用（或者说更像是 REPLACE）**直接重建语句
2. **暂时地启用或停用事件（用 DISABLE 和 ENABLE 关键字）**

```
ALTER EVENT yearly_delete_stale_audit_row DISABLE/ENABLE
```

回顾

我们主要通过事件来自动化数据库维护任务

导航

下节课讲解事务

【十一章】事务和并发

Transactions and Concurrency (时长49分钟)

1. 事务

Transactions (2:44)

事务

事务（trasaction）是完成一个完整事件的一系列SQL语句。这一组SQL语句是一条船上的蚂蚱，要不然都成功，要不然都失败，如果一部分执行成功一部分执行失败那成功的那一部分就会复原（revert）以保持数据的一致性。

例子1

银行交易：你给朋友转账包含从你账户转出和往他账户转入两个步骤，两步必须同时成功，如果转出成功但转入不成功则转出的金额会返还

例子2

订单记录：之前学过向父子表插入分级（层）/耦合数据，一个订单(order)记录对应多个订单项目(order_items)记录，如果在记录一个新订单时，订单记录录入成功但对应的订单项目记录录一半系统就崩了，**那这个订单的信息就是不完整的，我们的数据库将失去数据一致性**

ACID 特性

事务有四大特性，总结为 ACID（**刚好是英文单词"酸的"**）：

1. Atomicity 原子性，即整体性，不可拆分性（unbreakable），所有语句必须都执行成功事务才算完成，否则只要有语句执行失败，已执行的语句也会被复原
2. Consistency 一致性，指的是通过事务我们的数据库将永远保持一致性状态，比如不会出现没有完整订单项目的订单
3. Isolation 隔离性，指事务间是相互隔离互不影响的，尤其是需要访问相同数据时。具体而言，如果多个事务要修改相同数据，**该数据会被锁定**，每次只能被一个事务有权修改，其它事务必须等这个事务执行结束后才能进行
4. Durability 持久性，指的是一旦事务执行完毕，这种修改就是永久的，任何停电或系统死机都不会影响这种数据修改（？）

导航

下节课将学习如何创建事务

2. 创建事务

Creating Transactions (5:11)

准备

先用 create-databases.sql 恢复一下数据库

案例

创建一个事务来储存订单及其订单项目（为了简化，这个订单只有一个项目）

用 START TRANSACTION 来开始创建事务，用 COMMIT 来关闭事务（这是两个单独的语句）

```sql
USE sql_store;

START TRANSACTION;

INSERT INTO orders (customer_id, order_date, status)
VALUES (1, '2019-01-01', 1);
-- 只需明确声明并插入这三个【非自增不可空字段】

INSERT INTO order_items
-- 所有字段都是必须的（是不可空的意思吗？那有默认值和自增呢？），就不必申明了
VALUES (last_insert_id(), 1, 2, 3);

COMMIT;
```

执行，会看到最新的订单和订单项目记录

当 MySQL 看到上面这样的事务语句组，会把所有这些更改写入数据库，如果有任何一个更改失败，会自动撤销之前的修改，这种情况被称为事务被退回(回滚)（is rolled back）

为了**模拟退回的情况**，可以用 Ctrl + Enter 逐条执行语句，执行一半，即录入了订单还没录入订单项目时断开连接（模拟客户端或服务器崩溃或断网之类的情况），重连后会发现订单和订单项目都没有录入

手动退回

多数时候是用上面的 `START TRANSACTION; + COMMIT;` 来创建事务，但当我们想先对事务里语句**进行测试/错误检查**并因此想在执行结束后手动退回时，可以**将最后的 `COMMIT;` 换成 `ROLLBACK;`**，这会退回事务并撤销所有的更改

autocommit
我们执行的每一个语句（可以是增删查改 SELECT、INSERT、UPDATE 或 DELETE 语句），就算没有 START TRANSACTION + COMMIT，也都会被 MySQL 包装（wrap）成事务并在没有错误的前提下自动提交，**这个过程由一个叫做 autocommit 的系统变量控制，默认开启**

因为有 autocommit 的存在，当事务只有**一个语句时**，用不用 START TRANSACTION + COMMIT 都一样，但要将**多个语句**作为一个事务时就必须要加 START TRANSACTION + COMMIT 来手动包装了

```
SHOW VARIABLES LIKE 'autocommit';
```

小结

1.

```
START TRANSACTION;

……;

COMMIT / ROLLBACK;
```

2.

```
SHOW VARIABLES LIKE 'autocommit';
```

3. 并发和锁定

Concurrency（并发性；同时发生） and Locking (4:07)

并发

之前都只有一个用户访问数据，现实中常出现**多个用户访问相同数据的情况，这被称为“并发”（concurrency）**，当一个用户企图修改另一个用户正在检索或修改的数据时，并发会成为一个问题

导航

本节介绍默认情况下MySQL是如何处理并发问题的，接下来几节课将介绍如何最小化并发问题

案例

假设要通过如下事务语句给1号顾客的积分增加10分

```
USE sql_store;
START TRANSACTION;
UPDATE customers
SET points = points + 10
WHERE customer_id = 1;
COMMIT;
```

现在有两个会话（注意是两个连接（connection），而不是同一个会话下的两个SQL标签，这两个连接相当于是在模拟两个用户）都要执行这段语句，**用 Ctrl+Enter 逐句执行**， 当第一个执行到 UPDATE 而还没有 COMMIT 提交时，转到第二个会话，执行到UPDATE语句时会出现旋转指针表示在等待执行（若等的时间太久会超时而放弃执行），这时跳回第一个对话 COMMIT 提交，第二个会话的 UDDATE 才不再转圈而得以执行，最后将第二段对

话的事务也COMMIT提交，此时刷新顾客表会发现1号顾客的积分多了20分

上锁

所以，可以看到，当一个事务修改一行或多行时，会给这些行上锁，这些锁会阻止其他事务修改这些行，直到前一个事务完成（不管是提交还是退回）为止，**由于上述MySQL默认状态下的锁定行为，多数时候不需要担心并发问题**，但在一些特殊情况下，默认行为不足以满足你应用里的特定场景，这时你可以**修改**默认行为，这是我们接下来会学习的

导航

我们接下来会学习常见并发问题以及如何解决他们

4. 并发问题

Concurrency Problems (7:25)

现在已经知道什么是并发了，我们来看看它带来的常见问题：

1. Lost Updates 丢失更新

例如，当事务A要更新john的所在州而事务B要更新john的积分时，若两个事务都读取了john的记录，在A跟新了州且尚未提交时，B更新了积分，那后执行的B的更新会覆盖先执行的A的更新，州的更新将会丢失。

解决方法就是前面说的锁定机制，锁定会防止多个事务同时更新同一条数据，必须一个完成的再执行另一个

2. Dirty Reads 脏读

例如，事务A将某顾客的积分从10分增加为20分，但在提交前就被事务B读取了，事务B按照这个尚未提交的顾客积分确定了折扣数额，可之后事务A被退回了，所以该顾客的积分其实仍然是10分，因此**事务B等于是读取了一个数据库中从未提交的数据并以此做决定，这被称作为脏读**

解决办法是设定事务的隔离等级，例如让一个事务无法看见其它事务尚未提交的更新数据，这个下节课会学习。标准SQL有四个隔离等级，比如，我们可以**把事务B设为 READ COMMMITED 等级，它将只能读取提交后的数据**

积分提交完之后，B事务依次做决定，如果之后积分再修改，这就不是我们考虑的问题了，我们只需要保证B事务读取的是提交后的数据就行了

3. Non-repeating Reads 不可重复读取 （或 Inconsistent Read 不一致读取）

上面的隔离能保证只读取提交过的数据，但有时会发生一个事务读取同一个数据两次但两次结果不一致的情况

例如，事务A的语句里需要读取两次某顾客的积分数据，读取第一次时是10分，此时事务B把该积分更新为0分并提交，然后事务A第二次读取积分为0分，这就发生了不可重复读取 或 不一致读取

一种说法是，我们应该总是依照最新的数据做决定，所以这不是个问题。在商务场景中，我们一般不用担心这个问题

另一种说法是，我们应该保持数据一致性，以事务A在开始执行时的数据初始状态为依据来做决定，如果这是我们想要的话，就要增加事务A的隔离等级，让它在执行过程中看不见其它事务的数据更改（即便是提交过的），**SQL有个标准隔离等级叫 Repeatable Read 可重复读取，可以保证读取的数据是可重复和一致的，无论过程中其它事务对数据做了何种更改，读取到的都是数据的初始状态**

4. Phantom Reads 幻读 （n. 幽灵；幻影，幻觉）

最后一个并发问题是幻读

例如，事务A要查询所有积分超过10的顾客并向他们发送带折扣码的E-mail，查询后执行结束前，事务B更新了（可能时增删改）数据，然后多了一个满足条件的顾客，事务A执行结束后就会有这么一个满足条件的顾客没有收到折扣码，这就是幻读，Phantom是幽灵的意思，这种突然出现的数据就像幽灵一样，**我们在查询中错过了它因为它是在我们查询语句后才更新的**

解决办法取决于想解决的商业问题具体是什么样的以及把这个顾客包括进事务中有多重要

我们总可以再次执行事务A来让这顾客包含进去

但**如果确保我们总是包含了最新的所有满足条件的顾客是至关重要的**，我们就要保证查询过程中没有任何其他可能影响查询结果的事务在进行，为此，我们建立另一个隔离等级叫 **Serializable 序列化**，它让事务能够知晓是否有其它事务正在进行可能影响查询结果的数据更改，并会等待这些事务执行完毕后再执行，这是最高的隔离等级，为我们提供了最高的操作确定性。但 Serializable 序列化 等级是**有代价的**，当用户和并发增加时，等待的时间会变长，系统会变慢，所以这个隔离等级会影响性能和可扩展性，出于这个原因，我们**只有在避免幻读确实必要的情形下才使用这个隔离等级**

导航

这里只是先总体介绍，之后的课程会详细讲解每个并发问题以及如何用相应的隔离等级来解决它们

5. 事务隔离级别

Transaction Isolation Levels (5:42)

总结：并发问题与隔离等级

我觉得这个**表里后面三个问题都是读取问题，与四个事务的隔离等级正是一一对应**，所以很好理解和记忆。

而第一个问题——丢失更新，要特别一些，是修改覆盖问题，前面讲了是用锁定来解决，从这四个事务的隔离等级的名字上，只会觉得最高一级的序列化像是锁定了的，**但按这表格的意思，MySQL 默认的可重复读取等级也是锁定了的因而最后这两个级别都能防止丢失更新，这一点是需要特别记忆的**

	Lost Updates	Dirty Reads	Non-repeating Reads	Phantom Reads
READ UNCOMMITTED				
READ COMMITTED		✓		
REPEATABLE READ	✓	✓	✓	
SERIALIZABLE	✓	✓	✓	✓

四个并发问题：

1. Lost Updates 丢失更新：两个事务更新同一行，最后提交的事务将覆盖先前所做的更改
2. Dirty Reads 脏读：读取了未提交的数据
3. Non-repeating Reads 不可重复读取 （或 Inconsistent Read 不一致读取）：在事务中读取了相同的数据两次，但得到了不同的结果

4. Phantom Reads 幻读：在查询中缺失了一行或多行，因为另一个事务正在修改数据而我们没有意识到事务的修改，我们就像遇见了鬼或者幽灵

为了解决这些问题，我们有四个标准的事务隔离等级：

1. Read Uncommitted 读取未提交：无法解决任何一个问题，因为事务间并**没有任何隔离，他们甚至可以读取彼此未提交的更改**
2. Read Committed 读取已提交：给予事务一定的隔离，这样我们只能读取已提交的数据，这防止了Dirty Reads 脏读，但在这个级别下，事务仍可能读取同个内容两次而得到不同的结果，因为另一个事务可能在两次读取之间更新并提交了数据，也就是它不能防止Non-repeating Reads 不可重复读取 （或 Inconsistent Read 不一致读取）
3. Repeatable Read 可重复读取：在这一级别下，我们可以确信不同的读取会返回相同的结果，即便数据在这期间被更改和提交
4. Serializable 序列化：可以防止以上所有问题，这一级别还能防止幻读，**如果数据在我们执行过程中改变了，我们的事务会等待以获取最新的数据，但这很明显会给服务器增加负担，因为管理等待的事务需要消耗额外的储存和CPU资源**

并发问题 VS 性能和可扩展性：

1. 更低的隔离级别更容易并发，会有更多用户能在相同时间接触到相同数据，但也因此会有更多的并发问题，另一方面因为用以隔离的锁定更少，性能会更高；
   相反，更高的隔离等级限制了并发并减少了并发问题，但代价是性能和可扩展性的降低，因为我们需要更多的锁定和资源。
2. MySQL的默认等级是 Repeatable Read 可重复读取，它可以防止除幻读外的所有并发问题并且比序列化更快，多数情况下应该保持这个默认等级。
3. 如果对于某个特定的事务，防止幻读至关重要，可以改为 Serializable 序列化
4. 对于某些对数据一致性要求不高的批量报告或者对于数据很少更新的情况，同时又想获得更好性能时，可考虑前两种等级

总的来说，一般保持默认隔离等级，只在特别需要时才做改变

设定隔离等级的方法

读取隔离等级

```
SHOW VARIABLES LIKE 'transaction_isolation';
-- transaction isolation 事务隔离等级
```

显示默认的事务隔离等级为 'REPEATABLE READ'

改变隔离等级：

```
SET [SESSION]/[GLOBAL] TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

- 不加 SESSION/GLOBAL 则默认设定的是本次会话的下一次事务的隔离等级
- 加上 SESSION 就是设置本次会话（连接）之后所有事务的隔离等级
- 加上 GLOBAL 就是设置之后所有对话的所有事务的隔离等级

如果你是个应用开发人员，你的应用内有一个功能或函数可以连接数据库来执行某一事务（可能是利用对象关系映射或是直接连接MySQL）,你就可以连接数据库，**用 SESSION 关键词设置本次连接的事务的隔离等级**，然后执行事务，最后断开连接，这样数据库的其它事务就不会受影响

导航

接下来讲逐一讲解各个隔离级别

6. 读取未提交隔离级别

READ UNCOMMITTED Isolation Level (3:26)

小结

主要通过模拟脏读来表明 Read Uncommitted（读取未提交） 是最低的隔离等级并会遇到所有并发问题

案例

建立连接1和连接2，模拟用户1和用户2，分别执行如下语句：

连接1：

查询顾客1的积分，用于之后的商业决策（如确定折扣等级）

注意里面的 SELECT 查询语句虽然没被 START TRANSACTION + COMMIT 包裹，但由于 **autucommit**，MySQL会把执行的每一条所没错误的语句包装在事务中并自动提交，所以这个查询语句也是一个事务，隔离等级为上一句设定的 READ UNCOMMITTED（读取未提交）

```
USE sql_store;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT points
FROM customers
WHERE customer_id = 1;
```

连接2：

建立事务，将顾客1的积分（由原本的2293）改为20

```
USE sql_store;
START TRANSACTION;
UPDATE customers
SET points = 20
WHERE customer_id = 1;
ROLLBACK;
```

模拟过程:

→ 连接1将本次连接的下一次事务的隔离等级设定为 READ UNCOMMITTED 读取未提交
→ 连接2执行了更新但尚未提交
→ 连接1执行了查询，得到结果为尚未提交的数据，即查询结果为20分而非原本的2293分
→ 连接2的更新事务被中断退回（可能是手动退回也可能是因故障中断）

这样我们的对话1就使用了一个数据库中从未存在过的值，这就是脏读问题，总之，READ UNCOMMITTED 读取未提交 是最低的隔离等级，在这一级别我们会遇到所有的并发问题

7. 读取已提交隔离级别

READ COMMITTED Isolation Level (3:01)

小结

Read Committed 读取已提交 等级只会读取别人已提交的数据，所以不会发生脏读，但因为能够读取到执行过程中别人已提交的更改，所以还是会发生不可重复读取（不一致读取）的问题

案例1：不会发生脏读

就是把上一节连接1的设置隔离级别语句改为 READ COMMITTED 读取已提交 等级，就会发现连接1不会读取到连接2未提交的更改，只有当改为20分的事务提交以后才能被连接1的查询语句读取到

案例2：可能会发生不可重复读取（不一致读取）

虽然不会存在脏读，但会出现其他的并发问题，如 Non-repeating Reads 不可重复读取，即在一个事务中你会两次读取相同的内容，但每次都得到不同的值

为模拟该问题，将顾客1的分数还原为2293，将上面的连接1里的语句变为两次相同的查询（查询1号顾客的积分），连接2里的UPDATE语句不变，还是将1号顾客的积分（由原本的2293）更改为20

```
USE sql_store;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
START TRANSACTION;
SELECT points FROM customers WHERE customer_id = 1;
SELECT points FROM customers WHERE customer_id = 1;
COMMIT;
```

注意虽然案例1里已经执行过一次 `SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED`; 但这里还是要再执行一次，因为该语句是设定（本对话内）【下一次（next）】事务的隔离等级，**如果这里不执行，事务就会恢复为MySQL默认隔离等级**，即 Repeatable Read 可重复读取

还有因为这里事务里有两个语句，所以必须手动添加 START TRANSACTION + COMMIT 包装成一个事务，否则autocommit会把它们分别包装形成两个事务

模拟过程：

再次设定隔离等级为 READ UNCOMMITTED，启动事务，执行第一次查询，得到分数为2293 → 执行连接2的 UPDATE 语句并提交 → 再执行连接1的第二次查询，得到分数为20，同一个事务里的两次查询得到不同的结果，发生了 Non-repeating Reads 不可重复读取 （或 Inconsistent Read 不一致读取）

导航

为了解决 Non-repeating Reads 不可重复读取 的问题，我们需要提高隔离等级，这正是接下来要学习的

8. 重复读取隔离级别

REPEATABLE READ Isolation Level (3:29)

小结

在这一默认级别上，不仅只会读取已提交的更改，而且**同一个事务内读取会始终保持一致性，但因为可能会忽视正在进行但未提交的可能影响查询结果的更改而漏掉一些结果，即发生幻读**

Mosh只讲了这个级别在读取方面的问题，但从第3节以及第5节的表格可以看得出来，**这个默认级别还会在执行事务内的增删改语句时锁定相关行以避免更新丢失问题**

案例1：不会发生不可重复读取（不一致读取）

注意，先要将上一节最后的事务COMMIT提交了，才能执行新的，设定下一次事务隔离等级的语句

此案例和上一个案例完全一样，只是把隔离等级的设定语句改为了 REPEATABLE READ 可重复读取，然后发现两次查询中途别人把积分从2293改为20不会影响两次查询的结果，都是初始状态的20分，不会发生不可重复读取（不一致读取）

案例2：可能发生幻读

但这一级别还是会发生幻读的问题，一个模拟情形如下：

用户1：查询在'VA'州的顾客

```
USE sql_store;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION;
SELECT * FROM customers WHERE state = 'VA';
SELECT points FROM customers WHERE customer_id = 1;
COMMIT;
```

用户2：将1号顾客所在州更改为VA

```
USE sql_store;
START TRANSACTION;
UPDATE customers
SET state = 'VA'
WHERE customer_id = 1;
COMMIT;
```

假设customer表中原原本只有2号顾客在维州（'VA'）

→ 用户2现在正要将1号顾客也改为VA州，已执行UPDATE语句但还没有提交，所以这个更改技术上讲还在内存里
→ 此时用户1查询身处VA州的顾客，只会查到2号顾客
→ 用户2提交更改
→ 若1号用户未提交，再执行一次事务中的查询语句会还是只有2号顾客，因为在 REPEATABLE READ 可重复读取隔离级别，我们的读取会保持一致性
→ 若1号用户提交后再执行一次查询，会得到1号和2号两个顾客的结果，我们之前的查询遗漏了2号顾客，这被称作为幻读

简单讲就是在这一等级下1号用户的事务只顾读取当前已提交的数据，不能察觉现在正在进行但还未提交的可能对查询结果造成影响的更改，导致遗漏这些新的"幽灵"结果（但一般遗漏这些幽灵结果问题）

下节课讲如何用序列化隔离级别来解决幻读问题

9. 序列化隔离级别

SERIALIZABLE Isolation Level (2:18)

案例

和上面那个案例一摸一样，只是把用户1事务的隔离等级设置为 SERIALIZABLE 序列化，模拟场景如下：

→ 用户2现在正要将1号顾客也改为VA州，已执行UPDATE语句但还没有提交，所以这个更改技术上讲还在内存里
→ 此时用户1查询身处VA州的顾客，**会察觉到用户2的事务正在进行，因而会出现旋转指针等待用户2的完成**
→ 用户2提交更改
→ 用户1的查询语句执行并返回最新结果：顾客1和顾客2

小结

SERIALIZABLE 序列化 是最高隔离等级，它等于是把系统变成了一个单用户系统，事务只能一个接一个依次进行，所以所有并发问题（更新丢失、脏读、不一致读取、幻读）都从从根本上解决了，但用户和事务越多等待时间就会越漫长，所以，**只有**对那些避免幻读至关重要的事务使用这个隔离等级。**默认的可重复读取等级对大多数场景都有效，最好保持这个默认等级，除非你知道你在干什么（Stick to that, unless you know what you are doing）**

10. 死锁

Deadlocks (6:11)

小结

不管什么隔离等级，事务里的增删改语句都会锁定相关行（按表格看前两级应该不会啊，不然怎么会有跟新丢失的问题，此处有疑问？），如果两个同时在进行的事务分别锁定了对方下一步要使用的行，就会发生死锁，死锁不能完全避免但有一些方法能减少其发生的可能性

案例

用户1：将1号顾客的州改为'VA'，再将1号订单的状态改为1

```
USE sql_store;
START TRANSACTION;
UPDATE customers SET state = 'VA' WHERE customer_id = 1;
UPDATE orders SET status = 1 WHERE order_id = 1;
COMMIT;
```

用户2：和用户1**完全相同的两次更改，只是顺序颠倒**

```
USE sql_store;
START TRANSACTION;
UPDATE orders SET status = 1 WHERE order_id = 1;
UPDATE customers SET state = 'VA' WHERE customer_id = 1;
COMMIT;
```

模拟场景：

用户1和2均执行完各自的第一个更改
→ 用户2执行第二个更改，出现旋转指针
→ 用户1执行第二个更改，出现死锁，**报错：Error Code: 1213. Deadlock found** ……

缓解方法

死锁如果只是偶尔发生一般不是什么问题，重新尝试或提醒用户重新尝试即可，死锁不可能完全避免，但有一些方法可以最小化其发生的概率：

1. **注意语句顺序：如果检测到两个事务总是发生死锁，检查它们的代码，这些事务可能是储存过程的一部分，看一下事务里的语句顺序，如果这些事务以相反的顺序更新记录，就很可能出现死锁，为了减少死锁，我们在更新多条记录时可以遵循相同的顺序**
2. **尽量让你的事务小一些，持续时间短一些，这样就不太容易和其他事务相冲突**
3. **如果你的事务要操作非常大的表，运行时间可能会很长，冲突的风险就会很高，看看能不能让这样的事务避开高峰期运行，以避开大量活跃用户**

导航

如果觉得这一章很难是正常的，**并发算是很高级的内容了**，Mosh当年为了真正理解并发在各处查阅了大量资料，可能没有任何资料能像这一章的视屏这样能把并发的概念掰开揉碎并讲的这样简单清晰了，难理解的话认真多看几遍就一定能理解的。

专栏介绍和总目录
数据概要

Mosh完全掌握SQL课程学习笔记
数据概要

第五部分：脱颖而出——数据类型、设计数据库、索引、保护

【十二章】数据类型

Data Types (时长35分钟)

1. 介绍

Introduction (0:43)

知道MySQL支持的数据类型并且知道什么时候该用什么是很重要的

MySQL的数据分为以下**几个大类**:

1. String Types 字符串类型
2. Numeric Types 数字类型
3. Date and Time Types 日期和时间类型
4. Blog Types 存放二进制的数据类型
5. Spatial Types 存放地理数据的类型

借下来将学习每一大类中的具体数据类型

2. 字符串类型

String Types (2:25)

最常用的两个字符串类型

1. `CHAR()` 固定长度的字符串，如州（'CA', 'NY', ……）就是 CHAR(2)
2. `VARCHAR()` 可变字符串
   - Mosh习惯用 VARCHAR(50) 来记录用户名和密码这样的短文本 以及 用 VARCHAR(255) 来记录像地址这样较长一些的文本，保持这样的习惯能够简化数据库设计，不必每次都想每一列该用多长的 VARCHAR
   - **VARCHAR 最多能储存 64KB, 也就是最多约 65k 个字符（如果都是英文即每个字母只占一字节的话），超出部分会被截断**

字符串类型也可以用来储存邮编，电话号码这样的**特殊的数字型文本数据**，因为邮编电话号码等不会用来做数学运算而且常常包含'-'或括号等

储存较大文本的两个类型

1. `MEDIUMTEXT` 最大储存16MB（约16百万的英文字符），适合储存JSON对象，CS视图字符串，中短长度的书
2. `LONGTEXT` 最大储存4GB，适合储存书籍和以年记的日志

还有两个用的少一些的

1. `TINYTEXT` 最大储存 255 Bytes
2. `TEXT` 最大储存 64KB，最大储存长度和 VARCHAR 一样，但最好用 VARCHAR，因为 **VARCHAR 可以使用索引**（之后会讲，索引可以提高查询速度）

国际字符

所有这些字符串类型都支持国际字符，其中：

- **英文字符占1个字节**
- **欧洲和中东语言字符占2个字节**
- **像中日这样的亚洲语言的字符占3个字节**

所以，如果一列数据的类型为 CHAR(10)，MySQL会预留30字节给那一列的值

导航

下节课讲整数

3. 整数类型

Integer Types (2:52)

我们用整数类型来保存没有小数的数字，MySQL里共有5种常用的整数类型，它们的**区别在于占用的空间和能记录的数字范围**

背景知识：关于储存单位

- 一个晶体管可开和关，表示0或1两个值，代表最小储存单位，叫一位（bit）
- 一字节（Byte）有8位，可表示2^8个值，即256个值
- 字节（B）、千字节（KB）、兆字节（GB）、太字节（TB）的换算单位是2^10，即1024，约1000.

整数类型	占用储存	记录的数字范围
TINYINT	1B	[-128,127]
SMALLINT	2B	[-32K,32K]
MEDIUMINT	3B	[-8M,8M]
INT	**4B**	**[-2B,2B]**
BIGINT	8B	[-9Z,9Z]

INT 占 4 字节，最多表示 356^4 即约 4B 种数值，正负各一半，所以范围约为 [-2B,2B]

属性1.不带符号 UNSIGNED
这些整数可以选择不带符号，加上 `UNSIGNED` 则只储存非负数
如最常用的 `UNSIGNED TINYINT`，占用空间和 `TINYINT` 一样也是一字节，**但表示的数字范围不是 [-128-127] 而是 [0-255]，适合储存像年龄这样的数据，可以表示更大的正数范围也可以防止意外输入负数**

属性2.填零 ZEROFILL
整数类型的另一个属性是填零（Zerofill），主要用于当你需要给数字前面添加零让它们位数保持一致时
我们**用括号表示显示位数**，如 `INT(4)` 则显示为 0001，注意这只影响MySQL如何**显示**数字而不影响如何保存数字

（注意区分 INT(4) 和 CHAR(2)、VACHAR(50) 括号里数字的作用）

方法

不用强行去记，**谷歌 mysql integer types** 即可查阅

注意

如果试图存入超出范围的数字，MySQL会抛出异常 'The value is **out of range**'

最佳实践

总是使用能满足你需求的最小整数类型，如储存人的年龄用 UNSIGNED TINYINT（1B，[0-255]） 就足够了，至少可见的未来内没人能活过255岁
数据需要在磁盘和内存间传输，虽然不同类型间只有几个字节的差异，但数据量大了以后对空间和查询效率的影响就很大了，所以在数据量较大时，有意识地分配每一字节，保持数据最小化是很有必要的。

4. 定点数类型和浮点数类型

Fixedpoint and Floatingpoint Types (1:42)
这节主要讲储存小数的数据类型，有定点数和浮点数两种类型

Fixedpoint Types 定点数类型

`DECIMAL(p, s)` 两个参数分别指定最大的有效数字位数和小数点后小数位数（小数位数固定）
如：**DECIMAL(9, 2)** => 1234567.89 总共最多9位，小数点后两位，整数部分最多7位

`DECIMAL` 还有几个别名：`DEC` / `NUMERIC` / `FIXED`，**最好就使用 `DECIMAL` 以保持一致性，但其它几个也要眼熟**，别人用了要认识

Floatingpoint Types 浮点数类型

进行科学计算，要计算特别大或特别小的数时，就会用到浮点数类型，浮点数不是精确值而是近似值，这也正是它能表示更大范围数值的原因
具体有两个类型：

- `FLOAT` 浮点数类型，占用4B
- `DOUBLE` **双精度**浮点数，占用8B，显然能比前者储存更大（小？）的数值

小结

如果需要记录精确数字，比如【货币金额】，就是用 `DECIMAL` 类型
如果要进行【科学计算】，要处理很大或很小的数据，而且精确值不重要的话，就用 `FLOAT` 或 `DOUBLE`

5. 布尔类型

Boolean Types (0:46)

有时我们需要储存 是/否 型数据，如 “这个博客是否发布了？”，这里我们就要用到布林值，来表示真或假

MySQL里有个数据类型叫 `BOOL` / `BOOLEAN`

案例

```sql
UPDATE posts
SET is_published = TRUE / FALSE
【或】
SET is_published = 1 / 0
```

注意

布林值其实本质上就是 微整数 `TINYINT` 的另一种表现形式，TRUE / FALSE 实质上就是 1 / 0，但 Mosh 个人觉得写成 TRUE / FALSE 表意更清楚

6. 枚举和集合类型

Enum and Set Types (3:36)

enumeration n. 枚举

有时我们希望某个字段从固定的一系列值中取值，我们就可以用到 `ENUM()` 和 `SET()` 类型，前者是取一个值，后者是取多个值

ENUM()

从固定一系列值中取一个值

案例

例如，我们希望 sql_store.products（产品表）里多一个**size（尺码）字段**，取值为 small/medium/large 中的一个，可以打开产品表的设计模式，添加size列，数据类型设置为 `ENUM('small','medium','large')`，然后apply，对应SQL语句为：(修改表结构的语句下一章会讲)

```sql
ALTER TABLE `sql_store`.`products`
ADD COLUMN `size` ENUM('small', 'medium', 'large') NULL AFTER `unit_price`;
```

则产品表会增加一个尺码列，可将其中的值设为 small/medium/large(**大小写无所谓**)，但若设为**其他值会报错**

SET()

SET 和 ENUM 类似，区别是，SET是从固定一系列值中**取多个值**而非一个值

注意

讲解 ENUM 和 SET **只是为了眼熟，最好不要用**这两个数据类型，问题很多：

1. 修改可选的值（如想增加一个'extra large'）会重建整个表，耗费资源
2. 想查询可选值的列表或者想用可选值当作一个下拉列表都会比较麻烦
3. 难以在其它表里复用，其它地方要用只有重建相同的列，之后想修改就要多处修改，又会很麻烦

最佳实践

像这种某个字段从固定的一系列值中取值的情况，不应该使用 ENUM 和 SET 而应该**用这一系列的值另外建一个“查询表” (lookup table)**
如，上面案例中，应该另外建一个size尺码表，就像 sql_invoicing 里为支付方式专门建了一个 payment_methods 表一样。这样就解决了上面的所有问题，**既方便查询可选值的列表和作为下拉选项，也方便复用和更改**

导航

下一章设计数据库讲里讲 **normalization（标准化/归一化）**时会更详细地讲解这个问题

7. 日期和时间类型

Date and Time Types (0:44)

MySQL 有**4种**储存日期时间的类型：

1. `DATE` 有日期没时间
2. `TIME` 有时间没日期
3. `DATETIME` 包含日期和时间
4. `TIMESTAMP` 时间戳，常用来记录一行数据的的插入或最后更新时间

最后两个的区别是：

- TIMESTAMP 占4B，最晚记录2038年，被称为“2038年问题”
- DATETIME 占8B
  所以，如果要储存超过2038年的日期时间，就要用 DATETIME

另外，还有一个 `YEAR` 类型专门储存四位的年份

8. 二进制大对象类型

Blob Types (1:17)

我们用 `Blob` 类型来储存大的二进制数据，包括PDF，图像，视频等等**几乎所有的二进制的文件**
具体来说，MySQL里共有4种 Blob 类型，它们的**区别在于可储存的最大文件大小**：

类型	最大可储存
TINYBOLB	255B
BLOB	65KB
MEDIUM BLOB	16MB
LONG BLOB	4GB

注意

通常应该将二进制文件**存放在数据库之外**，关系型数据库是设计来专门处理结构化关系型数据而非二进制文件的。

如果将文件储存在数据库内，会有如下问题：

1. 数据库的大小将迅速增长
2. 备份会很慢
3. 性能问题
4. 需要额外的读写图像的代码

所以，尽量别用数据库来存文件，除非这样做确实有必要而且上面这些问题已经被考虑到了

9. JSON类型

JSON Type (10:24)

背景：关于JSON

- MySQL还可以储存 JSON 文件，JSON 是 JavaScript Object Notation（JavaScript 对象标记法）的简称
- **简单讲，JSON 是一种在互联网上储存和传播数据的简便格式**（Lightweight format for storing and transferring data over the Internet）
- JSON 在网络和移动应用中被大量使用，多数时候你的手机应用会以 JSON 形式向后端传输数据

语法结构：

```
{
    "key": value
}
```

- **JSON 用大括号 {} 表示一个对象，里面有多对键值对**
- **键 key 必须用引号包裹（而且必须是双引号，不能用单引号）**
- **值 value 可以是数值，布林值，数组，文本， 甚至另一个对象（形成嵌套 JSON 对象）**

案例

用 sql_store 数据库，在 products 商品表里，在设计模式下新增一列 properties，设定为 JSON 类型，注意在 Workbench里，要将 Eidt-Preferences-Modeling-MySQL-Default Target MySQL Version 设定为 8.0 以上，不然设定 JSON 类型会报错

这里的 properties 记录每件产品附加的独特属性，**注意这里每件产品的独特属性的种类是不一样的，如衣服是颜色和尺码，而电视机是的重量和尺寸，把所有可能的属性都作为不同的列添加进表是很糟糕的设计，因为每个商品都只能用到所有这些属性的一部分，相反，通过增加一列 JSON 类型的 properties 列，我们可以利用 JSON 里的键值对很方便的储存每个商品独特的属性**

现在我们已经有了一个 JSON 类型的列，接下来从 **增 删 改 查** 各角度来看看如何操作使用 JSON 类型的列，注意这里的 增删查改 主要针对的不是整个 JSON 对象而是里面的特定键值，即如何 增删查改 属性列里的某些特定的具体属性

增

给1号商品增加一系列具体属性，有两种方法

法1：

用单引号包裹（注意不能是双引号），里面是 JSON 的标准格式：

- 双引号包裹键 key
- 值 value 可以是数、数组、甚至另一个用 {} 包裹的JSON对象
- 键值对间用逗号隔开

```
USE sql_store;
UPDATE products
SET properties = '
{
    "dimensions": [1, 2, 3],
    "weight": 10,
    "manufacturer": {"name": "sony"}
}
'
WHERE product_id = 1;
```

法2：

也可以用 MySQL 里的**一些针对 JSON 的内置函数**来创建商品属性：

```sql
UPDATE products
SET properties = JSON_OBJECT(
    'weight', 10,
    'dimensions', JSON_ARRAY(1, 2, 3),
    'manufacturer', JSON_OBJECT('name', 'sony')
)
WHERE product_id = 1;
```

两个方法是**等效**的

查

现在来讲如何查询 JSON 对象里的特定键值对，这是将一列设为 JSON 对象的优势所在，如果 properties 列是字符串类型如 VARCHAR 等，是很难获取特定的键值对的

有两种方法：

法1

使用 `JSON_EXTRACT()` 函数，其中：

- 第1参数指明 JSON 对象
- **第2参数是用单引号包裹的路径，路径中 `$` 表示当前对象，点操作符 `.` 表示对象的属性**

```sql
SELECT product_id, JSON_EXTRACT(properties, '$.weight') AS weight
FROM products
WHERE product_id = 1;
```

法2

更简便的方法，使用**列路径操作符 -> 和 ->>，后者可以去掉结果外层的引号**

```sql
SELECT properties -> '$.weight' AS weight
FROM products
WHERE product_id = 1;
-- 结果为：10

SELECT properties -> '$.dimensions'
……
-- 结果为：[1, 2, 3]

【SELECT properties -> '$.dimensions[0]' 】
……
-- 结果为：1
/*
看来 JSON 对象里的数组和其它大多数语言一样，索引是从0开始的，
不像MYSQL的字符串那样索引从1开始（从之前的字符串切片和定位函数可以看得出来）
*/

SELECT properties -> '$.manufacturer'
……
-- 结果为：{"name": "sony"}

SELECT properties -> '$.manufacturer.name'
……
-- 结果为："sony"

【SELECT properties ->> '$.manufacturer.name'】
……
```

```
28 -- 结果为：sony
```

通过路径操作符来获取 JSON 对象的特定属性不仅可以用在 SELECT 选择语句中，也可以用在 WHERE 筛选语句中，如：
筛选出制造商名称为 sony 的产品：

```
SELECT product_id, properties ->> '$.manufacturer.name' AS manufacturer_name
FROM products
【WHERE properties ->/->> '$.manufacturer.name' = 'sony'】
-- 最好还是像 Mosh 一样用 ->> 吧
```

结果为：

product_id	manufacturer_name
1	sony

Mosh说最后这个查询的 WHERE 条件语句里用路径获取制作商名字时必须用双箭头 ->> 才能去掉结果的双引号，才能是的比较运算成立并最终查找出符合条件的一号产品，但实验发现用单箭头 -> 也可以，另一方面在 SELECT 选择语句中用单双箭头确实会使得显示的结果带或不带双引号，所以综合来看，单双箭头应该是只影响路径结果 "sony" 是否【显示】外层的引号，但不会改变其实质，所以不会影响其比较运算结果，即单双箭头得出的 sony都是 = 'sony' 的

改

如果我们是要重新设置整个 JSON 对象就用前面 **增** 里讲到的 `JSON_OBJECT()` 函数，但如果是想修改已有 JSON 对象里的**部分属性**，就要用 **`JSON_SET()` 函数**

```
USE sql_store;
UPDATE products
SET properties = JSON_SET(
    properties,
    '$.weight', 20,  -- 【修改weight属性】
    '$.age', 10  -- 【增加age属性】
)
WHERE product_id = 1;
```

注意 `JSON_SET()` 是获取已有 JSON 对象并修改部分属性然后返回修改后的 JSON 对象，所以其第1参数是要修改的 JSON 对象，并且可以用 `SET porperties = JSON_SET(properties, ……)` 的语法结构来实现对 properties 的修改

删

可以用 `JSON_REMOVE()` 函数实现对已有 JSON 对象特性属性的删除，原理和 `JSON_SET()` 一样

```
USE sql_store;
UPDATE products
SET properties = JSON_REMOVE(
    properties,
    '$.weight',
    '$.age'
)
WHERE product_id = 1;
```

小结

感觉JSON对象就是个储存键值对的字典，可以嵌套

标准格式为： `{"key":value,……}`

1. **增：利用标准格式或利用** `JSON_OBJECT`**,** `JSON_ARRAY` **等函数**
2. **查：** `JSON_EXTRACT` **或** `->/-->`**，注意表达路径时** `$` **和** `.` **的使用**
3. **改：** `JSON_SET`**，注意其原理**
4. **删：** `JSON_REMOVE`**，原理同上**

【十三章】设计数据库

Designing Databases (时长1时30分)

1. 介绍

Introduction (1:25)

之前都是对已有数据库进行增删查改（主要是查询），这一章学习如何设计和创建数据库（以及表格）。

设计一个结构良好的数据库是需要耗费不少时间和心力的，但这是十分必要的，**设计良好的数据库可以快速地查询到想要的数据并且有很好的扩展性（很容易满足新的业务需求），相反，一个糟糕的数据库可能需要大量维护且查询又慢又麻烦**，Mosh之前一家公司的数据库就做得很糟糕，有些储存过程有上千行代码而且有些查询执行时间长达数分钟，所以，拥有设计良好的数据库是非常重要的。

这一章将系统性地逐步讲解如何设计一个结构良好的数据库

2. 数据建模

Data Modelling (2:26)

这一节讲数据建模，即为想要储存进数据库的数据建立模型的过程，其中包含4步：

1. Understand the requirements 理解需求

第1步是理解和分析商业/业务需求，遗憾是很多程序员跳过了这一步就急着去设计数据库里的表和列了，**实际上，这一步是最关键的一步，你对问题理解的越透彻，你才越容易找到最合适的解决方案，设计数据库也一样。**所以，在动手创建表和列之前，要先完整了解你的业务需求，包括和产品经理、行业专家、从业人员甚至终端用户深入交流以及收集查阅已有的领域相关的表、文件、应用程序、数据库，以及其他与问题领域相关的任何信息或资料

2. Build a conceptional model 概念建模

当收集并理解了所有相关信息后，下一步就是**为业务创建一个概念性的模型。**这一步包括**找出/识别/确认（identify）**业务中的 实体/事务/概念（entities/things/concepts）以及它们之间的**关系。**概念模型只是**这些概念的一个图形化表达，用来与利益相关方交流和达成共识**

3. Build a logical model 逻辑建模

创建好概念模型后，转而创建数据模型（data model）或数据结构（data structure for storing data），即逻辑建模。这一步创建的是**不依赖于具体数据库技术的抽象的数据模型，主要是确认所需要的表和列以及大体的数据类型**

4. Build a physical model 实体建模（实际建模）

实体建模指的是将逻辑模型**在具体某种DBMS上加以实现**的过程，相比于逻辑模型，实体模型会确定**更多细节**，包括各表**主键**的设定，各列在某一DBMS下特定的**具体的数据类型，默认值，是否可为空**，还包括**储存过程和触发器等对象的创建。**总之，实体模型是在某一特定DBMS下对数据模型非常具体的实现

以上就是数据建模的流程

导航

如果以上流程听上去有点懵也没关系，之后会用一个实例来具体走一遍这些流程

下一节将讲解概念模型

3. 概念模型

Conceptual Models (4:34)

案例

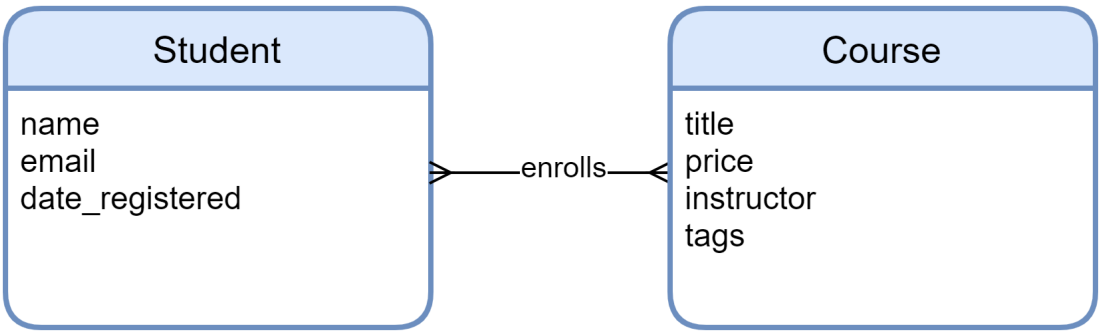
想要建一个销售在线课程的网站，用户可以注册一项或多项课程，课程可以有诸如 "前端" "后端" 这样的标签

对于一个线上课程网站来说，重要的概念/实体有哪些？很容易想到有学生（student）和课程（course）

我们需要一种**将实体及其关系可视化的方法**，一种是**实体关系图（Entity Relationship, ER）**，一种是**统一建模语言（Unified Modeling Language，UML）**，这里我们用实体关系图（ER），使用的工具是 **draw.io**

步骤如下：

1. 建立学生实体并确定相关属性，如姓名、电子邮件、注册时间
2. 建立课程实体并确定相关属性，如课程名、价格、老师、标签
3. 建立两个实体间的关系，暂时先用多对多连线（概念模型里只是画好连线，逻辑建模时再考虑连线的类型），**加上 enrolls 标签表示两者间的关系是“学生→注册 →课程”**



注意

建模是个迭代过程，不可能第一次就建立完美模型，需要在理解需求和模型设计之间不断反复，多次调整。比如这里的学生属性，可以先确定个大概，之后可以根据需要再进行增删修改

小结

概念模型主要是从很高的视角来总览业务需求，识别业务中的实体/事务/概念以及他们彼此间的关系，通常这些实体包括人、事件、地点等

这一步暂不考虑数据类型和具体的DBMS这样的技术细节，**只是从概念上总揽全局，目的是和业务人员交流，保持理解一致，避免鸡同鸭讲**

导航

下一节我们将用这个概念模型来建立逻辑模型

4. 逻辑模型

Logical Models (7:24)

案例

接前面线上课程网站的例子，对概念模型逻辑化的过程如下：

1. 细化实体间关系：

   考虑学生和课程的关系，首先这是一种**多对多关系（通常意味着需要进一步细化）**，其次了解到**业务上有如下需求**：

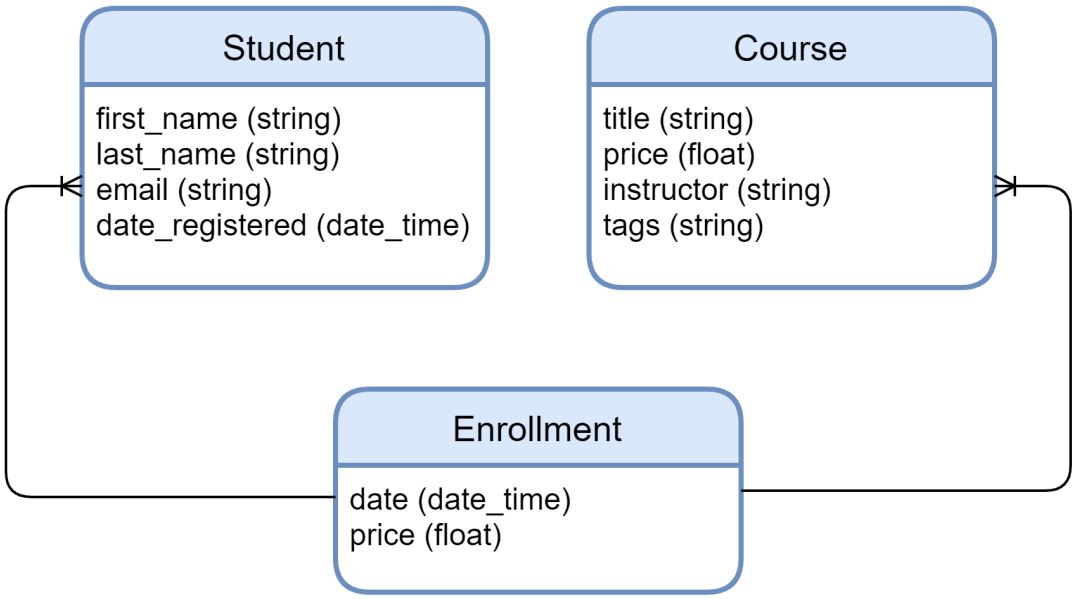
   - 需要记录学生注册特定课程的日期
   - 课程价格是变化的，需要记录学生注册某门课程时的特定价格

   这些属性相对于学生和课程而言都是一对多关系，不管放在学生还是课程身上都不合适，所以，应该为学生和课程之间的关系，即 注册课程的事件 另外设立一个实体 enrollmemt，上面的注册日期和注册价格都应该是这个 enrollment 注册事件 的属性

2. 调整字段并大体确定字段的数据类型：

   姓名（name）最好**拆分**为姓和名 (first_name 和 last_name)，同理，地址应该拆分为省、市、街道等等小的部分，这样方便查询。注意课程里的 **tags 标签字段不是一个好的设计，之后讲归一化时再来处理**

   这里的**数据类型只需确定个大概即可**，如：是 string，float 而非 VARCHAR, DECIMAL。等到下一步实体模型里再来确定某个DBMS下的具体数据类型



小结

逻辑模型是在概念模型的基础上，在不依赖特定数据库系统的前提下确定数据结构，包括细化实体间的关系（常常要为关系创造新的实体），调整字段设置，确定大体的数据类型。总之，逻辑模型会基本确立数据库中的表、列以及表间关系。

5. 实体模型

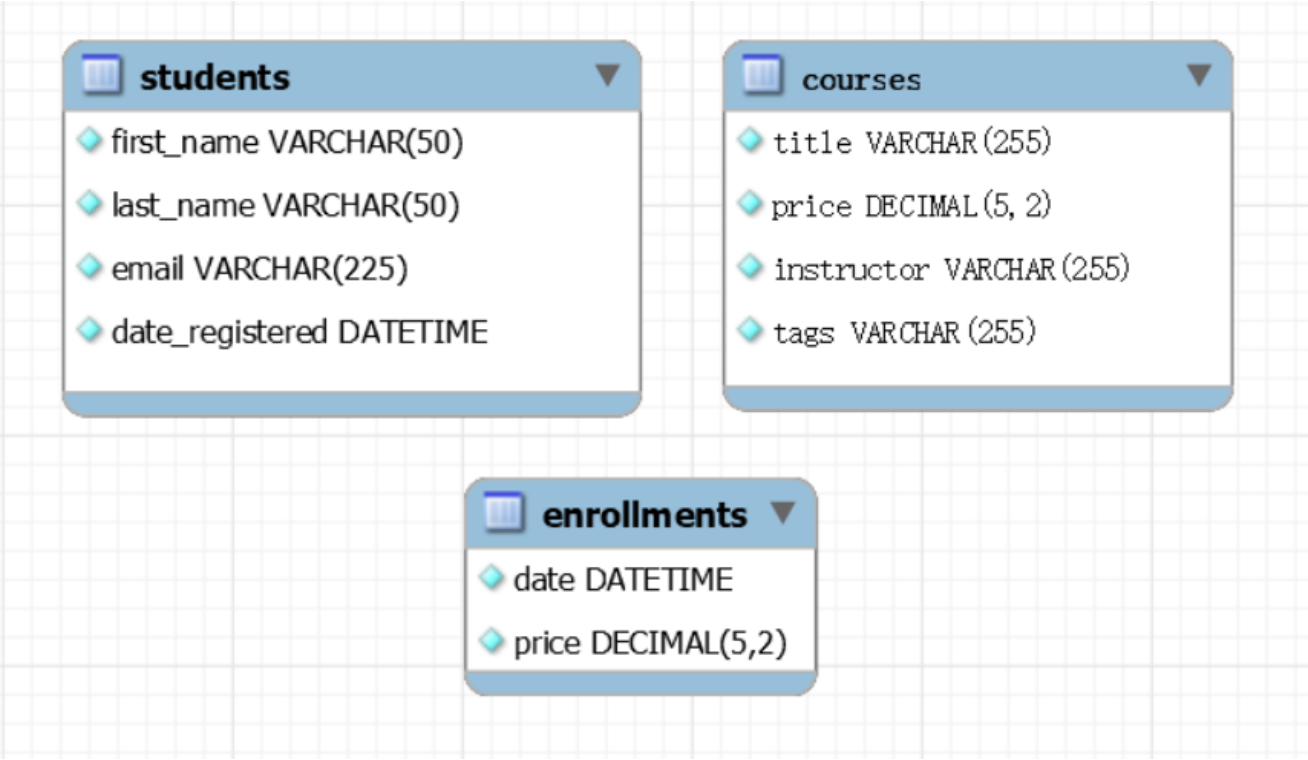
Physical Models (6:27)

实体模型（实际模型）就是逻辑模型在具体DBMS的实现，这里我们用MySQL实现前面线上课程网站的逻辑模型

在 Workbench-file-【new model】 新建数据库，右键 edit 修改数据库名字为 school

上方用 add diagram 作 EER 图，这里 EER 表示 Enhanced Entity Relationship 增强型实体关系图。为三个实体创建三张表，设定表名、字段、具体的数据类型、**是否可为空（即是否为必须字段），也可以选择设定默认值（主键设定之后再讲）**。有几个注意点：

- 表名：
  之前逻辑模型里表名用单数，但这里表名用复数。这只是一种惯例，**单复数都行，关键是要保持一致。**
  如果团队有相关惯例就去遵守它，即便那不够理想，也别去破坏惯例，否则沟通和维护成本会大大增加，你需要不断去想该用单数还是复数
- 字段名：
  以enrollments表为例，注册事件的属性应该是 date日期 和 price价格 而非 enrollment_date注册日期 和 enrollement_price注册价格，**不要将表名前缀加上字段上造成不必要的麻烦，保持精简（keep things simple）**
- 数据类型：
  数据类型要根据业务需要来，例如，和业务人员确认后发现课程价格最高是999美元，所以 price价格 就可以设定为 DECIMAL(5,2)，之后如果需求变了了也可以随时更改，不要一上来就设定DECIMAL(9,2)，浪费磁盘，注意保持精简（keep things small）



小结

实体模型是逻辑模型**在特定DBMS上的实现，主要是一些技术上的细化，包括确定字段具体数据类型和性质（能否为空等），设置主键等**

导航

接下来我们要给每一个表设置一个主键并定义表之间的关系

6. 主键

Primary Keys (3:23)

主键就是能唯一标识表中每条记录的字段

设定 students 表的主键：

不管是 first_name 还是 last_name 都不能唯一标识每条记录，它们两个合起来作为联合主键也不行，因为两个人全名相同也是可能的（都叫 Tom Smith）。**Email 也不适合作主键，首先太长了，之后需要作为外键复制到其他表浪费资源，而且 Email 也可能改变。**

总之主键要短，可唯一标识记录，且永不改变。我们增加一个 student_id 作为主键，**类型设为 INT（最大可表示2亿，一般足够了，但记得总是根据具体的需求决定），设为主键后自动变为不可为空，另外还要设定 AI（Auto Incremental）自动递增，这样会方便许多，不用担心主键唯一性的问题，最后我们把主键拖到表的第一列让表的结构看起来更清晰**

设定 courses 表的主键：

增加一个 course_id 作为主键，其它和 student_id 一样

导航

下节课讲 enrollemnts 表的主键问题

7. 外键

Foreign Keys (5:48)

注意 enrollments 表的特殊性，它可以说是 students 和 courses 的**衍生表**，先要有学生和课程，才能有 学生注册课程 这一事件，后者表述的是前两者的关系，**学生和课程是因，注册课程这一事件是果**

MySQL里可以通过一对一或一对多两种连线表达这种先后关系/因果关系并自动建立外键，其中学生和课程被称作父表或主键表，注册事件被称作子表或外键表，外键是子表里对父表主键的引用

几个细节：

- **连线时记不得先连主表还是子表可以看状态栏的提示**
- MySQL自动添加的外键会带父表前缀，没必要，建议去掉

可以看到，相对于逻辑模型，实体模型有更多实现细节，包括设置字段具体类型和性质以及根据表间关系确定主键和外键

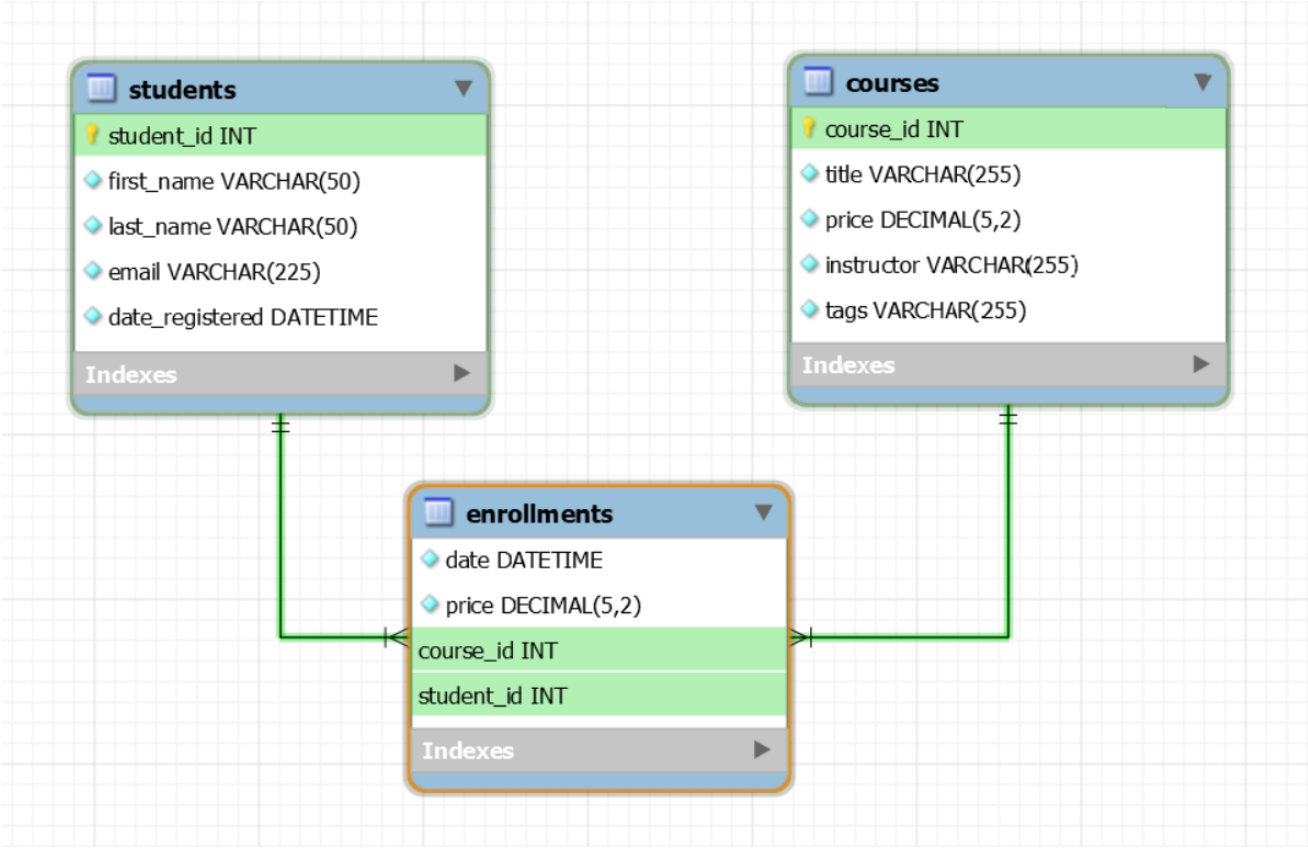
现在，根据表间关系给 enrollments 表添加了 student_id 和 course_id 两个外键，**enrollments 的主键设置有两个选择：**

1. 将这两个外键作为联合主键
2. 另外设置一个单独的主键 enrollment_id

两种选择**各有优缺点**，以联合主键为例：

- 好处是可以避免重复的注册记录，即可以防止同一个学生重复注册同一门课程，因为主键（这里是联合主键）是唯一不可重复的，这可以防止一些不合理的数据输入
- 坏处是如果 enrollments 未来有新的子表，就需要复制两个字段（而不是 enrollment_id 这样的一个字段）作为外键，这也不一定是很大的麻烦，要根据数据量以及子表是否还有子表等情况来考虑，在一定情况下可能会造成不必要冗余和麻烦（相对于将 enrollment_id 一个字段作为主键来说）

但目前来说，没有为 enrollments 建立子表的需求，永远不要为未来不知道会不会出现的需求进行设计开发，如果之后需要的话也可以通过脚本修改表结构，也不会很麻烦，所以目前的情况，用联合主键就好了。在 enrollments 表里把两个外键的黄钥匙都点亮，即成为联合主键



8. 外键约束

Foreign Key Constraints (5:22)

有外键时，需要设置约束以防止数据损坏/污染（不一致）

在 enrollements 表里，打开 **Foreign Keys 标签页**，可以看到两个外键，以 `fk_子表_父表` 的方式命名，名称后可能有数字，是MySQL为了防止外键与其他外键重名自动添加的，这里没必要，可去掉。**右边 Foreign Key Options 可分别选择当父表里的主键被修改或删除（Update / Delete）时，子表里的外键如何反应**，有三种选项：

1. CASCADE:
   瀑布/串联/级联，表示随着主键改变而改变，如主键某学生的 student_id 从1变成2，则改学生的所有注册课程记录的 student_id 也会全部变为2 （**注意主键一般也最好是永远不要变的，这里讨论的是特殊情况**）
2. RESTRICT / NO ACTION:
   等效，作用都是禁止更改或删除主键。如：对于有过注册记录的课程，**除非**先删除该课程的注册购买记录，不然不能在 courses表 里删除该课程的信息。（另外注意：MySQL 里外键**默认**的 On Update 和 On Delete 的反应都是 **NO ACTION)**
3. **SET NUL**L:
   就是当主键更改或删除时，使得相应的外键变为空，这样的子表记录就没有对应的主键和对应的父表记录了（no parent），被称为**孤儿记录（orphan record），这是垃圾数据**，让我们不知道是谁注册的课程或不知道注册的是什么课程，一般不用，只在极其特殊的情况可能有用。

经验法则

通常对于 UPDATE, 设置为 CASCADE 级联，随之改变

对于 DELETE，看情况而定，可能设置为 CASCADE 随之删除 也可能设置为 RESTRICT / NO ACTION 禁止删除。不要死板，永远按照业务/商业需求来选择，这也正是为什么之前强调"理解业务需求"是最重要的一步。比如我们课程注册记录里包含购买价格信息，则应该禁止删除，否则之后想查询某课或某时间段的收入情况就不能实现，相反如果只是个用户登录并设定一系列提醒的软件，可能用户允许用户注销并删除所有提醒就没什么大不了

的，但万一，我们需要这些提醒记录来进行统计，则又该设置为禁止删除，总之一定要根据具体业务需求来（always check with the business）

9. 数据库规范化/正规化/归一化

Normalization (1:24)

正式建立数据库前我们先要检查并确定现在的设计是最优化的（optimal），关键是没有任何冗余或重复，简洁且便于修改和保持一致性。重复数据会占用更多空间并且使得增删查改的操作复杂化，比如，如果用户名在多处出现的话，一旦更改用户名就要到多处更改否则就会使得数据不一致，出现无效数据。

为了防止重复冗余，需要遵循**数据库设计的7大规则或者说7大范式，每一条都是建立在你已经遵循了前一条的基础上。实际上，99%的数据库之需要遵循前3大范式就够了**，其他几个并没有那么重要。接下来将依次讲解前三大范式并给出可操作的建议，让你能够在不死记硬背这些规则的情况下轻松设计出归一化的数据库

补充：维基百科——数据库规范化（Normalization）

> 数据库规范化，又称正规化、标准化，是数据库设计的一系列原理和技术，以**减少数据库中数据冗余，增进数据的一致性。**关系模型的发明者埃德加·科德最早提出这一概念，并于1970年代初定义了第一范式、第二范式和第三范式的概念，还与Raymond F. Boyce于1974年共同定义了第三范式的改进范式——BC范式。
>
> 除外还包括针对多值依赖的第四范式，连接依赖的第五范式、DK范式和第六范式。
>
> **现在数据库设计最多满足3NF，普遍认为范式过高，虽然具有对数据关系更好的约束性，但也导致数据关系表增加而令数据库IO更易繁忙，原来交由数据库处理的关系约束现更多在数据库使用程序中完成。**

10. 第一范式

First Normal Form, 1NF (2:42)

第一范式：

> Each cell should have a single value and we cannot have repeated columns.
>
> 每个单元格都应该是**单一值并且不能有重复的列**

courses 里的 tags **标签列**就不符合第一范式。tags 列用逗号隔开多个标签，不是单一值。若将 tags 分割成多列，每个标签一列，问题是我们不知道到底有多少标签，每次出现新标签就要改动表结构，这样的设计很糟糕。这也正是范式1要求没有重复列的原因（没有重复列是这个意思？我还以为重复列是指在多表出现相同列（如姓名列）的情况）

所以我们另外单独**创建一个 tags 表**，设置两个字段：

1. tag_id TINYINT 如果标签是终端用户设定的，那数量就可能会迅速增长，但这里假定标签是管理员设定的，最多可能五六十个，那 TINYINT 足够了
2. name VARCHAR(50)

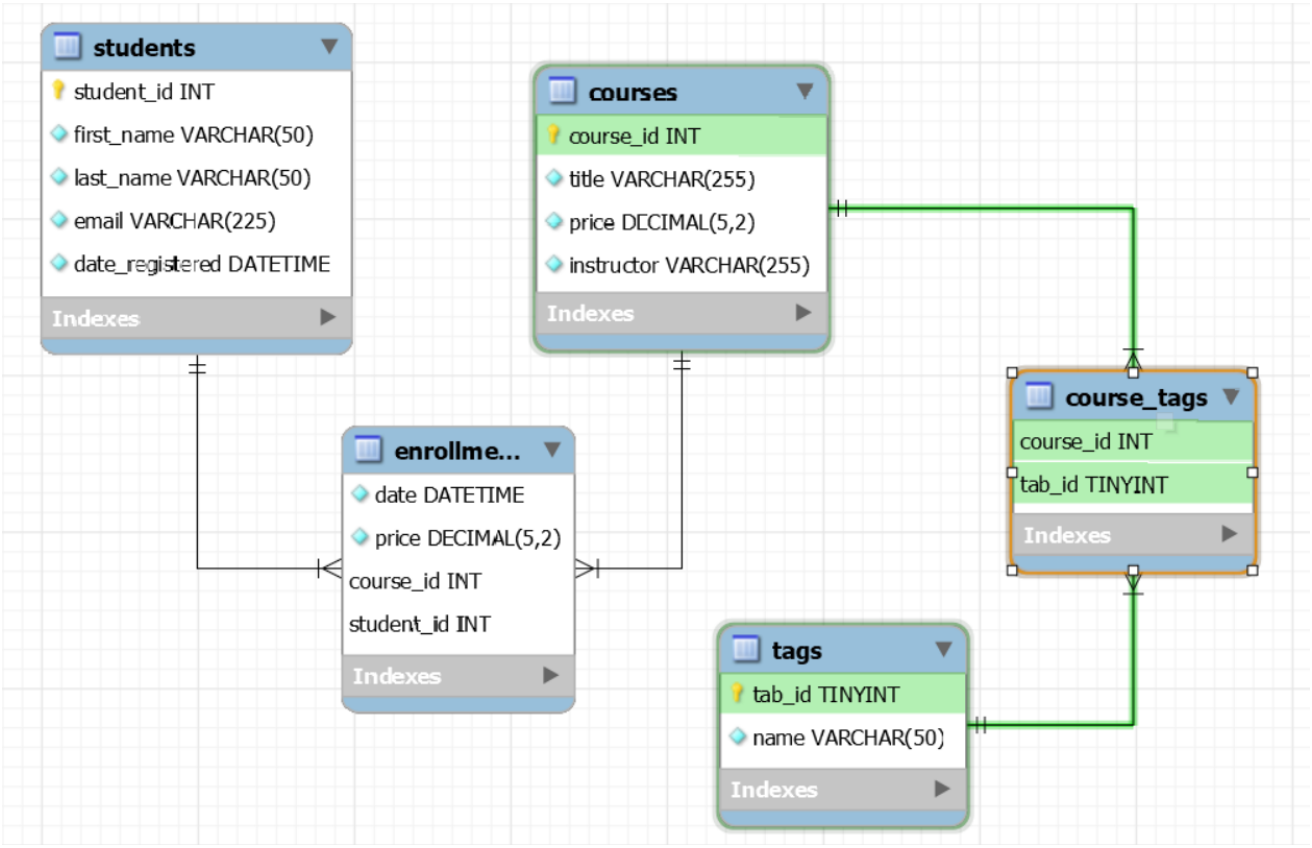
导航

下节课我们将在 tags 与 courses 间建立**多对多关系**（准确讲是将多对多关系通过链表的方式细化成两个多对一关系，就像之前对学生和课程关系的处理一样）

11. 链表

Link Tables (4:01)

尝试建立 courses 和 tags 之间的联系，发现两者是多对多关系（MySQL里只有一对一和一对多，没有多对多），这说明两者的关系需要进一步细化，**我们添加一个 course_tags 表来专门描述两者间的关系，记录每一对课程和标签的组合，这个中间表或者说链表（link table）同时是 courses 和 tags 的子表，与这两个父表均为一对多的关系，建立两条一对多连线后 MySql 自动给 course_tags 表增加了两个外键 course_id 和 tage_id（注意去掉自动添加的表前缀），两者构成了 course_tags 表 的联合主键**



通过 course_tags 细化 courses 和 tags 的关系 与 之前通过 enrollments 表细化 students 和 courses 的关系一样，都是通过**建立链表细化多对多关系，这是很常用的一种方法，有时链表只包含引用的两个外键，如 course_tags 表，有时链表还包含其它信息，如 enrollments 表**

至此，删除掉 courses 里的 tags 列，我们的数据库就符合第一范式了，所有列都是单一值也没有诸如tag1，tag2这样的重复列，所有标签都保存在独立的 tags 表里拥有唯一记录。如果像之前那样标签以逗号分隔保存在 courses 表中，同一个标签如 "frontend" 会多次出现，如果要将这个标签改名为 "front-end" 就会多出很多不必要的锁定操作，修改标签却要锁定 courses 表里的记录，这本身就很不合理，tags 表才该是唯一储存标签的地方，修改标签时 tags 表里的标签条目才是唯一应该被锁定的条目

导航

接下来讲第二范式

12. 第二范式

Second Normal Form, 2NF (6:32)

第二范式的人话解释：

> Every table should describe one entity, and every column in that table should describe that entity.
>
> **每个表都应该是单一功能的/应该表示一个实体类型，这个表的所有字段都是用来描述这个实体的**

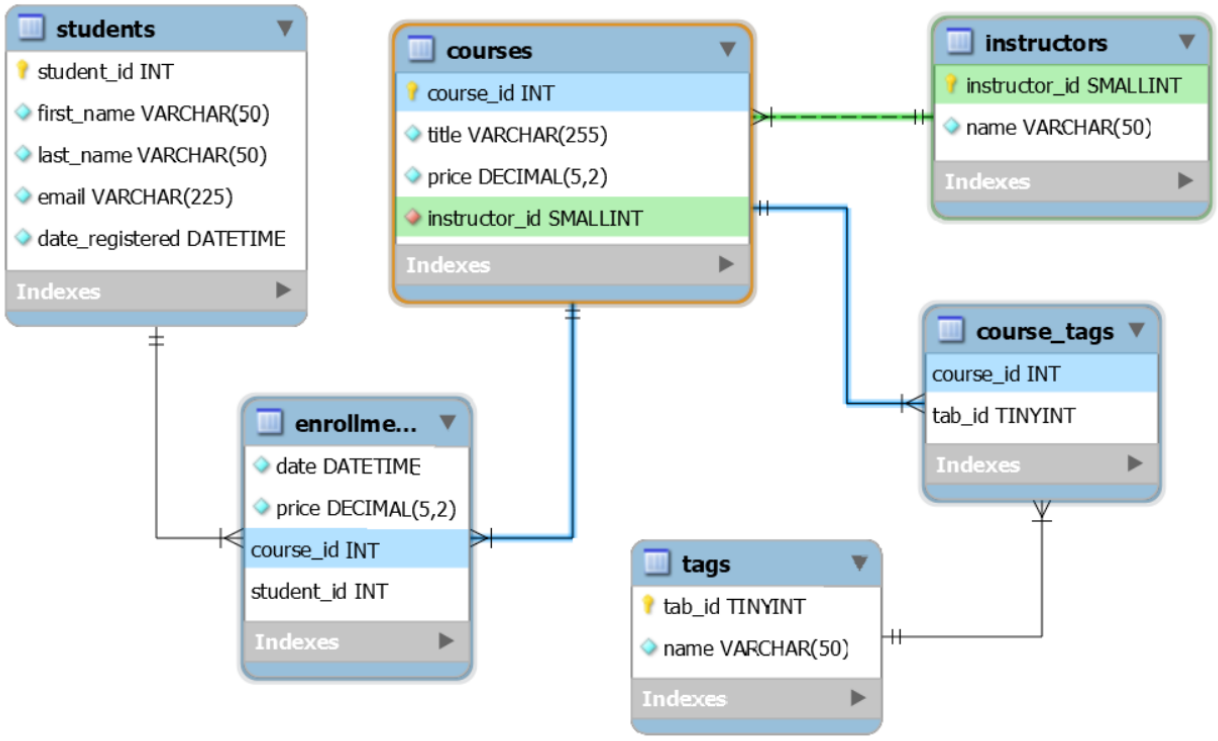
以 courses 表为例，course_id、title、price 都完全是属于课程的属性，放在 courses 表里没问题，但注册时间 enrollment_date 放在 courses 表里就不合适，因为一个课程可以被多个学生注册所以有多个注册时间，同样的注册时间也不应该是 students 表的属性，因为一个学生可以注册多门课所以可以有多个注册时间，注册时间应该是属于"注册事件"的属性，所以应该另外建个 enrollments 表，放在该表里。

同理，对于订单表 orders 来说，order_id 和 date 应该是其中的属性，但 customer 就不是，虽然每个订单确实有对应的顾客，但顾客可能在不同订单里重复，这会占用多余的储存空间并使得修改变得困难，应该单独建一个顾客表来储存顾客，订单表里用顾客id而非顾客名来引用顾客表，**当然，顾客id还是会重复，但4字节的数字比字符串占用的空间小多了，这已经是让重复最小化了**

总之，第一范式是要求单一值和无重复列，这里第二范式是要求表中所有列都只能是完全描述该表所代表的实体的属性，不属于该实体（如订单表）的、在记录中可重复的属性，应该另外放在描述相应实体的表里（如顾客表）

以我们这个模型为例，courses 里的 instructors 虽然是单一值符合第一范式却不符合第二范式，因为老师不是完全属于课程的属性，老师在不同课程中可能重复。所以，另外建立 instructrors 表作为父表，包含 instructor_id 和 name 字段，其中 instructor_id 为主键，一对多连接 courses 表后自动引进 courses 表作为外键，删除原先的 instructor 列。还有注意设置外键约束，UPDATE 设置为 CASCADE，DELETE 设置为 NO ACTION，也就是 instructor_id 会随着 instructors 表更改，但不允许在某教师有课程的情况下删除该教师的信息

至此，我们的数据库已符合第二范式。



补充：第二范式的维基百科

> 第二范式（2NF）是数据库正规化所使用的正规形式。规则是要求资料表里的所有资料都要和该资料表的键（主键与候选键）有**完全依赖关系**：每个非键属性必须独立于任意一个候选键的任意一部分属性。如果有哪些资料只和一个键的一**部分有关的话，就得把它们独立出来变成另一个资料表。**

回顾

简单讲，我觉得前两条范式讲的是：

1. 每一列都得是单一值（否则独立为查询表/资料表，并通过链表与原表相连）
2. 每个表都得是该实体的专属属性（否则独立为查询表/资料表）

13. 第三范式

Third Normal Form, 3NF (1:43)

第三范式的人话解释：

> A column in a table should not be derived from other columns.
>
> 一个表中的字段不应该是由表中其他字段推导而来

例如，invoices 发票表里假设有三个字段：**发票额、支付额 和 余额**，第三个可以由前两个相减得到所以不符合3NF，每次前两者更新第三个就要随之更新，假设没有这样做，让三者出现了 100，40，80 这样不一致的数据，就不知道到底该相信哪个了，余额到底是 80 还是 100-40=60?

同理，如果表里已经有 **first_name 和 last_name 就不该有 full_name**，因为第三者总是可以由前两者合并得到

不管是上面的 余额balance 还是 全名fullname，都是一种冗余，应该删除

补充：第三范式的维基百科

> 第三范式（3NF）是数据库正规化所使用的正规形式，要求所有非主键属性都只和候选键有相关性，也就是说非主键属性**之间应该是独立无关的。**
>
> 如果再对第三范式做进一步加强就成了BC正规化，强调的重点在于"资料间的关系是奠基在主键上、以整个主键为考量、而且除了主键之外不考虑其他因素"。

小结

第三范式和前两范式一样，都是为了**减少数据重复和冗余，增强数据的一致性和完整性**（data integrity）

14. 我的实用建议

My Pragmatic Advice (2:55)

除非需要考试，不然没必要记忆和死板套用三大范式，实际工作中**只需要专注于减少数据的重复性即可**，比如发现一个name字段下**出现的是一些重复的名字而不是重复的外键（如某种id），那就说明设计还不够归一化，具体违反哪条范式并不重要，关键是专注于避免重复性**

例子

假设一个顾客表里每条都是一个顾客信息，有名字年龄生日性别还有收货地址，假设一个**顾客可以有多条收货地址**怎么办呢?

如果仍然把收货地址放在这个顾客表，就要为了保存一个顾客的多条地址而将这个**顾客的所有信息复制多条**，这是一种没必要的重复和冗余

我们先思考从概念和逻辑模型上思考，这里有两个关键实体，顾客 和 地址，它们是一对多关系，然后再细化为实体模型，应该建立两张表，顾客表保存顾客其他信息，**地址表（实际上是顾客地址关系表）只保存顾客id和地址两个字段，这样就将重复性降到了最低**

小结

总之，一定要先从概念和逻辑模型去考虑实体和关系，再逐步细化，过程中专注于避免数据的重复冗余以及保证数据的一致性和完整性，一定不要一上来就建表，这样几乎总是得出糟糕由混乱的数据库设计

注意

上面的例子以一个顾客有多个收货地址为前提，**但如果一个顾客只有一个收货地址，那用一张表就足够了**，用两张表是没必要的，所以**关键是理解业务需求，总是按照业务需求来设计**，这也引入了下一节内容：不要对全宇宙建模！

15. 不要对什么都建模

Don't Model the Universe (4:24)

设计数据库时总是考虑当前的业务需求，不要试图包罗万象，总有开发人员会考虑各种未来可能出现的需求，实际上大部分那些需求都从未发生，反而使得数据库增加了很多没必要的复杂性，复杂化了查询并拖慢了执行效率

之前的公司曾有个人设计了一个过于一般化但也过于复杂难懂的数据库，企图满足所有未来可能的需求，但结果是没人能懂他的模型，而且执行增删改查异常麻烦且速度极低，最后成了一个没人敢碰的烂摊子

建立复杂模型不是本事，让模型尽可能优美简单易懂又能满足目前的需求这才是本事，如果还能有不错的拓展性以满足未来可能的新特性就更好了

总之，尽可能保持简洁，简洁才是终极哲学，**无论你对未来的预测有多好，总会有意料之外的需求出现，总有一天你会写脚本改数据库甚至进行数据迁移，这是避免不了的，当前只需考虑如何最好地满足目前的需求就好了，不要企图对全宇宙建模**

16. 模型的正向工程

Forward Engineering a Model (2:35)

通过模型正向搭建数据库：workbench 菜单的 **Database 选项 → Forward Engineer 正向搭建数据库**

依据向导保持默认不断点下一步就好了，不要更改，除非你知道你在做什么

有一步**可以选择** 除了创建数据库中的表 是否还要创建 储存程序、触发器、事务和用户对象，而且表格可以筛选到底要创建哪一些

最后一步会展示对应的SQL代码，里面有创建 school 数据库（schema 架构；模式；纲目；结构方案）以及各表的SQL代码，之后会详细讲。**可以选择保存代码为文件（以保存到仓库中）或者复制到剪贴板然后到 workbench 查询窗口里以脚本方式运行，这里我们直接运行**，返回 local instance 连接刷新界面就可以看到新的 school 数据库和里面的6张表了

17. 同步数据库模型

Synchronizing a Model with a Database (4:48)

之后可能会修改数据库结构，比如更改某些表中字段的数据类型或增加字段之类，如果只是自己一个人用的一个本地数据库，可以直接打开对应表的设计模式并点击更改即可，但如果是在团队中工作通常不是这样。

在中大型团队中，我们通常有多个服务器来模拟各种环境，其中有：

1. 生产环境（production environment）：用户真正访问应用和数据库的地方
2. 模拟环境（staging 演出，展示 environment）：与生产环境十分接近
3. 测试环境（testing environment）：存粹用来做测试的
4. 开发环境（development environment）

每次需要对数据库做修改时我们需要复制相同的修改到不同的环境以保持数据的一致性

所以不能是在设计模式中直接点击修改，相反，是在之前模型标签（注意模型可以保存为一个 MySQL 模型文件，下次可以直接打开使用）里的实体模型图（EER Diagram）中修改表或字段并使用菜单中的 Database → Synchronize Model（用模型创建数据库时用 Forward Engineer，对已有数据库进行同步修改时用 Synchronize Model）。注意点开 Synchronize Model 后可以选择连接，这里我们选择本地连接 local_instance，但如果是在团队中可能需要选择与测试环境、模拟环境甚至开发环境的连接以对相应环境中的数据库执行同步更改以保持一致，MySQL会自动检测到需要修改的是 school 数据库并提示要修改的表，例如我们想在 enrollments 中加上一个 coupons 折扣券 字段（注意不是所有注册都有折扣券所以该字段为非必须可空字段），会提示将影响的表除了 enrollments 还有 courses 等表，因为这些表与要修改的表是相互关联的，从之后的 SQL 的语句可以看出来，会先暂时删除相关外键以消除这些联系，对目标表做出相应更改（增加 coupons 字段）后再重建这些联系，同样的，我们可以把这些修改数据库的代码保存起来并上传到git仓库，**就可以在不同环境执行相同修改以保持数据库的一致性**

18. 数据库逆向工程

Reverse Engineering a Database (3:11)

如果要修改没有实体模型的数据库，第一次可以先逆向工程（Reverse Engineering）建立模型，之后每次就可以在该模型上修改了

例如，我们要修改 sql_store ，应如下操作：

1. **关闭当前 Model，不然之后的逆向工程结果会添加到当前模型上，最好是每个数据库都有一个单独的模型，除非数据库间相互关联否者不要在一个模型中处理多个数据库**
2. Database → Reverse Engineer，可以选择目标数据库，如上说所，除非数据库相互关联，否者最好**一次只逆向工程一个数据库，让每个数据库都有一个单独的模型。**
3. 同样，可以筛选要哪些表

在反向搭建出的模型中，可以更好的看到和理解数据库的结构设计，可以修改表结构（并将相关修改脚本保存并用于其它环境的数据库），还可以发现问题，如在 sql_store 数据库中，有一个 order_items_notes 表，并未与任何表相联，这样里面的order_id就可能输入无效值，相反如果是建立了连接的表，MySQL会自动验证数据的一致性/完整性/有效性（integrity），只允许子表中添加父表中存在的id值

小结

第一次修改无模型的数据库可以使用MySQL自带的逆向工程，之后就可以用这个模型查看表结构，做修改和检查问题

19. 项目：航班订票系统

Project Flight Booking System (0:23)

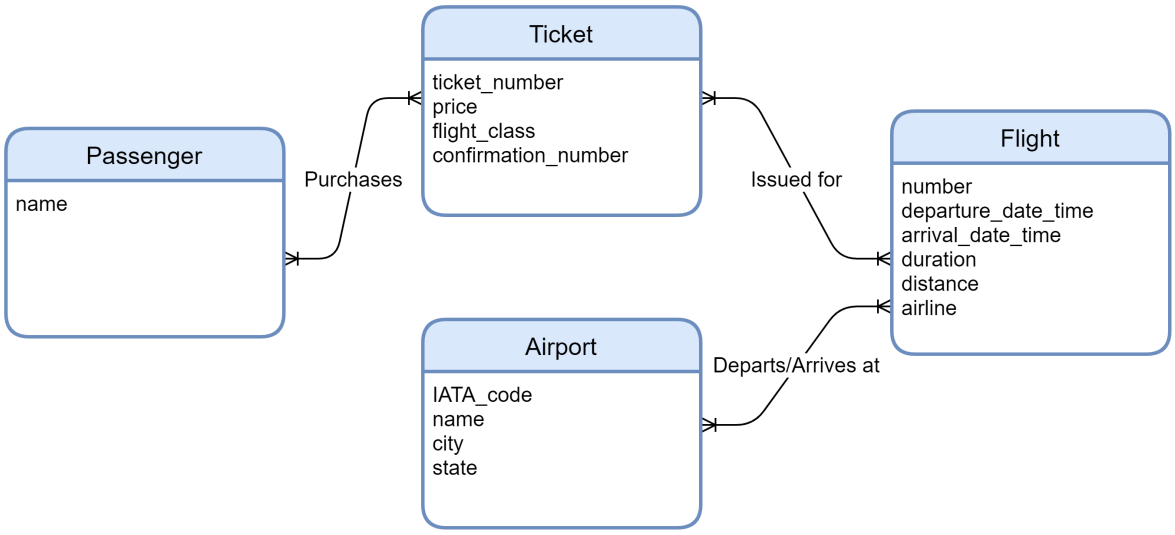
通过一张机票上的信息理解航班订票业务需求并建立数据库模型

20. 解答：概念模型

Solution Conceptual Model (7:59)

主要建立**实体，字段，和关系**，不用确定具体关系类型和字段类型等细节，主要用于和业务方交流

注意只根据机票信息决定需要的字段，满足当下需求就好，未来有新需求时再修改增加新的字段



21. 解答：逻辑模型

Solution Logical Model (9:03)

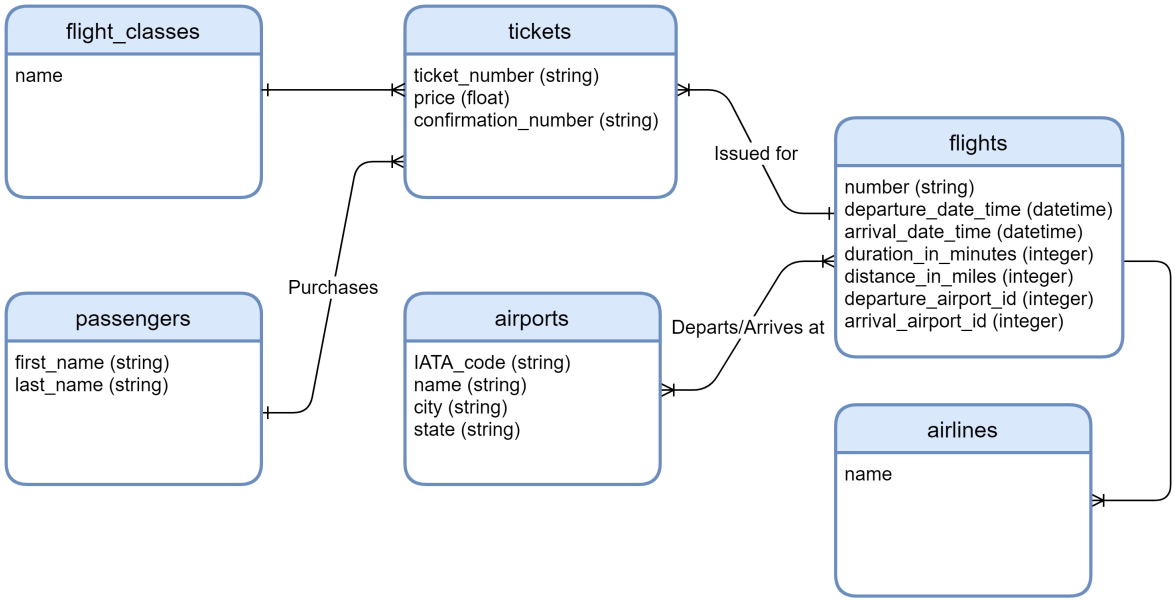
与概念模型相比，逻辑模型主要做了如下细化调整：

1. **细化关系，尤其是多对多关系，通常要另外添加链表做桥梁变成两个多对一关系**，但是注意flights和airports的关系很特殊，一个机场可对应多个航班但一个航班只能对应起飞和降落两个机场，是多对二的关系，或者说是两个多对一关系，如果还是通过一个链表来做桥梁替换这个多对多关系，则不能防止一个航班出现多于两个机场，**最好的办法**是将fights中的机场区分为起飞机场id和降落机场id两个字段，分别建立外键引用airports
2. **调整字段，name 这样的字段要拆分成 first_name 和 last_name 这样的更小组成成分，而重复性的字段常常要另外单独建表（查询表 lookup table）再以外键形式在原表中引用**，但是airports里的city和state比较特殊，因为考虑到city和state与airport**经常一起查询，合并在一张表上能提高查询速度**，而且机场数量并不多，重复性的问题并不严重，反而如果另外单独再建cities表和states表会使得数据过于**碎片化**，所以这里进行**“反归一化”（denormalize）**，在airports表中**保留city和state的原始字段，用一定的重复性来换取查询便利和效率（以及没必要的碎片化）**
3. **确认数据类型**，注意有的**所谓的 number** 其实不需做计算且包含符号，所以应该用字符串类型

另外注意用词和表达要向业务方咨询，确保用词准确表达方式与业务规范相一致，这很重要

还有注意调整字段时，可以将flights里的duration和distance改为duration_in_minutes和distance_in_miles，这样**更明确，不用去猜单位是什么**

实体模型就不展示了，从逻辑模型到实体模型只是具体DBMS技术上的调整和实现，没必要反复讲（只是**注意 flights 和airports 的二对多关系实际上应该是两个一对多关系，应该是在 MySQL 的 model-EER 关系图里用两条一对多连线来实现**）



22. 项目：视频租赁应用

Project Video Rental Application (1:05)

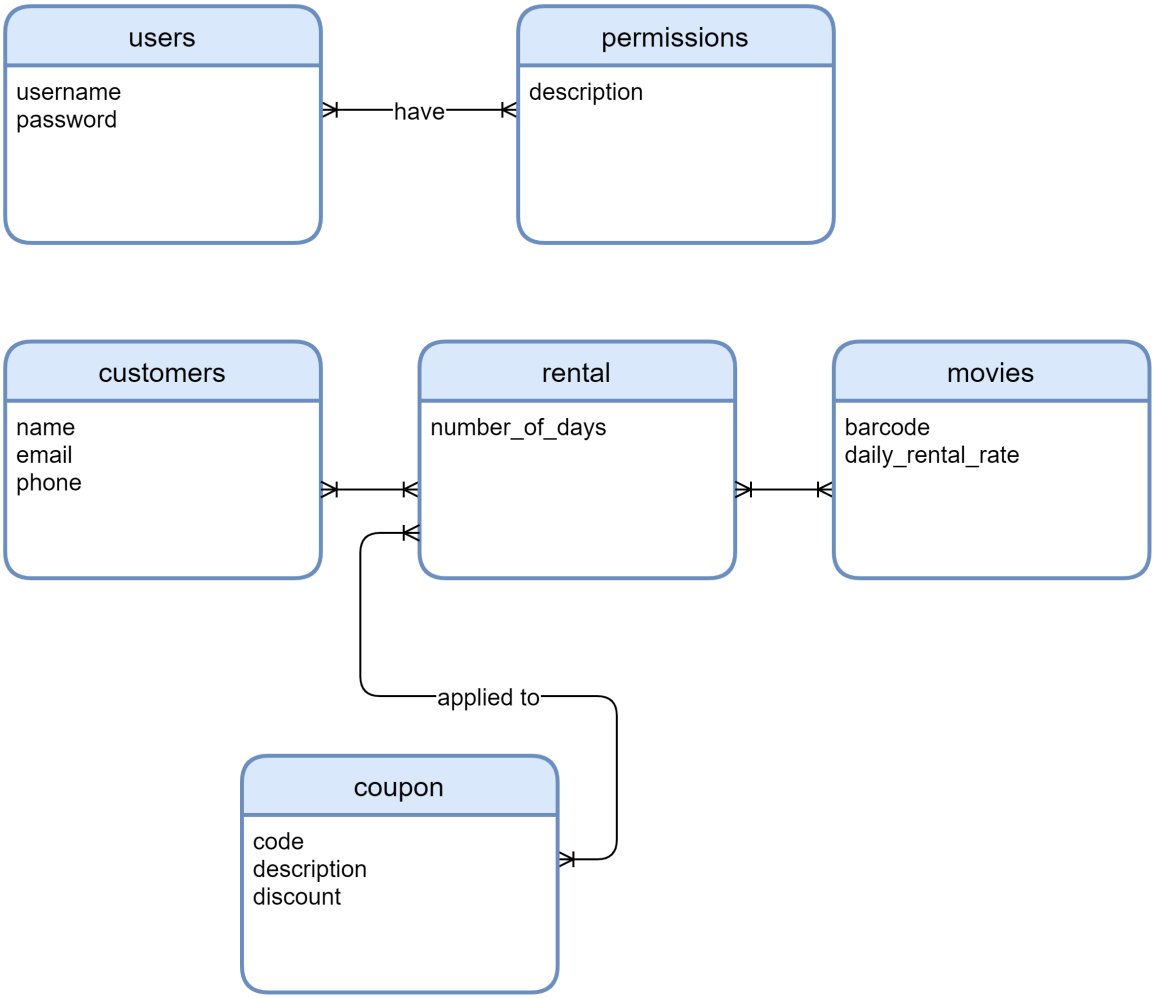
为视屏租赁应用 Vidly 建立数据库

23. 解答：概念模型

Solution Conceptual Model (6:59)

这一步还是一样，为了建立概念模型，**根据业务需求文档确定大概的实体、实体属性、实体间关系**

注意这里将顾客和电影的多对多关系细化为一个 rentals 链表（link table），变成可操作的两个一对多关系，这个方法之前也反复用到，如将学生和课程之间的关系细化为 enrollments 表 以及 乘客和航班的关系 细化为 tickets 表



24. 解答：逻辑模型

Solution Logical Model (8:29)

如之前一样，在逻辑模型里，我们要确定数据储存方式，所以要**进一步细化具体的实体间关系类型和字段的数据类型，也会为了减少数据重复性和提高数据一致性对表结构数据库结构进行一些调整修改**

关系类型具体化和字段数据类型确定：

和之前差不多，只是要注意coupon和rental的关系比较特殊，是**多对零或一**（注意箭头的不同），因为一个rental可能有也可能没有coupon

字段调整：

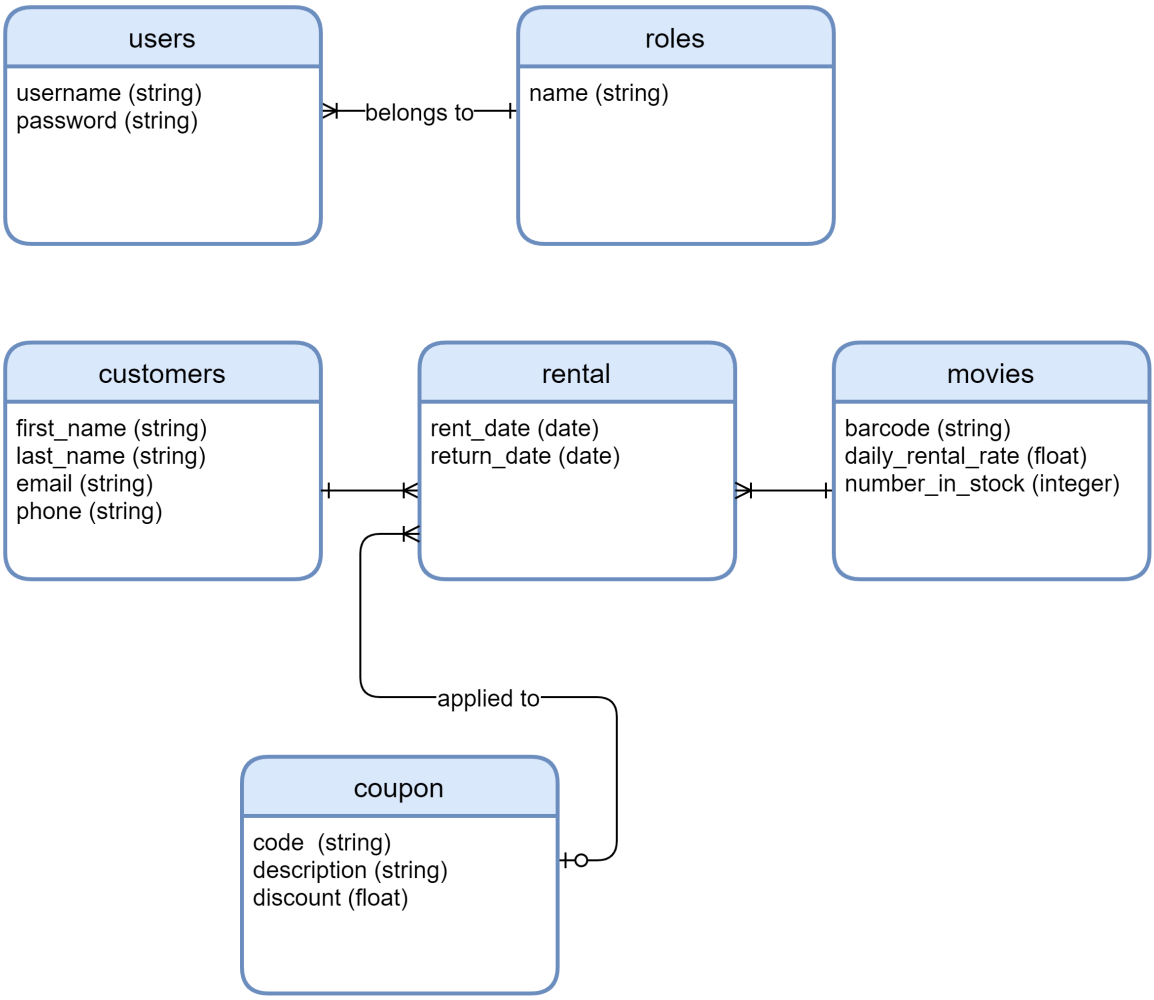
将名字拆分为姓和名，**将租赁天数拆分为借电影日期和还电影日期（后两个才能提供足够信息计算各月收入等）**

设计调整：

之前的 users-permissions 用户-权限设计并不好，虽然业务文档确实提到了这两个实体，但**仔细分析发现实际上**用户只有两类 管理员和店员 而对应的权限唯一的区别也只是是否能修改电影列表，在这一业务情形下，没必要有一个完整的权限表将所有权限列出来，只需要有一个roles岗位表将用户分为两类即可，实际的权限可以通过if条件语句来根据用户是管理员还是店员来决定是否禁止其修改电影列表，这样的设计更精简，减少了每次增加一个用户就要按个分配10个权限的重复性也防止了给相同职位不同权限这样的错误的发生，增强了一致性

思想

没必要列出10个权限然后依次分配，以用户表-权限表的方式设计模型**过于一般化，提供了业务并不需要的过高的控制等级，这种多余的复杂化和冗余会一直跟随系统一直造成不必要的麻烦。**如果你有100元预算只想找个能歇脚能睡觉的地方，那一个500元的豪华宾馆多出的功能如高质量的网络漂亮的海景奢华的床铺等等对你来说都是没必要的，**你不会多花400元买你不需要的功能，开发软件也一样， 所有功能和复杂性都是有成本的，都会有人买单，不要把公司的钱浪费在不需要的地方，要尽可能用最精简的方式满足当前的业务需求。**



25. 创建和删除数据库

Creating and Dropping Databases (1:41)

用workbench的向导来创建和修改数据库能够提高效率，但作为 DBA (Database Administrator 数据库管理员)，你必须要能理解并审核相关代码，确保其不会对数据库有不利影响，而且也有能力手动写代码完成创建和修改数据库的操作，可以不依赖（图形化和向导）工具。

这节课讲创建和删除数据库，依然是用 CREATE 和 DROP 这两个关键词

```
1  CREATE DATABASE IF NOT EXISTS sql_store2;
2  DROP DATABASE IF EXISTS sql_store2;
```

之后讲创建和修改表和表间关系等

26. 创建表

Creating Tables (3:13)

这节学习如何创建表

案例

以在 sql_store2 中建表 customers 为例，注意创建表之前还是**要先用 USE 选择数据库**，不然不知道你是要在那个数据库中创建表

```
USE sql_store2;

DROP TABLE IF EXISTS customers;
CREATE TABLE customers
-- 没有就创建，有的话就推倒重建

或

CREATE TABLE IF NOT EXISTS customers
-- 没有就创建，有的话就算了

(
    -- 只挑选几个字段来建立
    customer_id 【INT PRIMARY KEY AUTO_INCREMENT】,
    first_name VARCHAR(50) 【NOT NULL】,
    points INT NOT NULL 【DEFAULT 0】,
    email VARCHAR(255) NOT NULL 【UNIQUE】
    -- 【UNIQUE 确保 email 值唯一，即每个用户的 email 必须不一样】
)
```

注意

左侧栏导航窗口选择某表中的列时，**下面的 Object Info** 可以查看列的数据类型

小结

- 如上，**创建对象（不管是数据库还是表）有两种方式**，`DROP …… IF EXIXTS ……;` `CREAT ……` 和 `CREAT …… IF NOT EXISTS ……`，**注意两种方式的区别在于，当原对象存在时，前者是推倒重建，后者是保持原状放弃创建（所以按通常的需求来看还是前者更符合）**
- **括号中设置列的方式为 列名 数据类型 各种列性质，列间逗号分隔，常用的列性质有 `PRIMARY KEY`、`AUTO_INCREMENT`、 `NOT NULL`、`DEFAULT 0`、 `UNIQUE`**

27. 更改表

Altering Tables (2:56)

这节学习如何**更改已存在的表，包括增删列和修改列类型和属性**

```
USE sql_store2;
ALTER TABLE customers
    【ADD [COLUMN] last_name VARCHAR(50) NOT NULL [AFTER first_name]】,
    ADD city VARCHAR(50) NOT NULL,
    【MODIFY】 [COLUMN] first_name VARCHAR(60) DEFAULT '',
    DROP [COLUMN] points;
```

COLUMN 是可选的，有的人喜欢加上以增加可读性

AFTER first_name 是可选的，不加的话默认将新列添加到最后一列

MODIFY 修改已有列经实验发现其实应该是重置该列（= DROP + ADD），**所以注意要列出全部类型和属性信息**，如上例中将 first_name 修改为 VARCHAR(60) 类型并将默认值修改为空字符串''，但忘了加 NOT NULL，刷新后发现 first_name 不再有 NOT NULL 属性

列名最好不要有空格，但如果有的话可用反引号包裹，如 `last name`

注意

修改表永远不要直接在生产环境中进行，要首先在测试环境进行，确保没有错误和不良影响后再到生产环境进行修改

导航

下节课学习如何创建关系

28. 创建关系

Creating Relationships (4:47)

这节学习创建表间关系

案例

第26节在新的 store2 数据库中创建了 customers 表，这里我们**接着创建 orders 表，并在 orders 表中添加 customer_id 外键来建立表间关系**

```sql
CREATE DATABASE IF NOT EXISTS sql_store2;
USE sql_store2;
DROP TABLE IF EXISTS customers;
CREATE TABLE customers
(……);  -- 可点击加减号来展开或收起代码块

DROP TABLE IF EXISTS orders;
CREATE TABLE orders
(
    order_id    INT PRIMARY KEY,
    customer_id INT NOT NULL,
    order_date  DATE NOT NULL,
    -- 【在添加完所有列之后添加外键】
    FOREIGN KEY fk_orders_customers (customer_id)
        【REFERENCES】 customers (customer_id)
        【ON UPDATE CASCADE】
        -- 【也有人主张用 NO ACTION / RESTRICT】
        ON DELETE NO ACTION
        -- 【禁止删除有订单的顾客】
)
```

外键名的命名习惯：

fk_子表_父表

【设置外键的语法结构】：

```
    FOREIGN KEY 外键名（外键字段）
        REFERENCES 父表（主键字段）
        -- 【设置外键约束】
        ON UPDATE CASCADE
        ON DELETE NO ACTION
```

关于外键约束

ON DELETE 设置为 NO ACTION / RESTRICT 可以防止删除有的订单的顾客，这没什么问题；**而对于 ON UPDATE**，也有人主张设为 NO ACTION / RESTRICT，因为**主键是永远不应该被更改的，理论上Mosh支持这个观点**，但实际世界并不完美，由于意外或系统错误等原因，主键是有可能改变的，所以Mosh一般设置为CASCADE，随着主键的更改而更改，但你要设置为 NO ACTION / RESTRICT 也同样有道理。另外，想查看外键约束的可选项以及想通过菜单选择来更改外键约束的话，**可以打开某列的设计模式，在 Foreign Keys 标签页里进行选择**

【表间依赖】

还有注意一点，运行以上SQL文件从头创建 sql_store2数据库以及customers和orders两张表时，第一次运行没问题，但要再次运行的化会报以下错误：

```
-- Error Code: 1217. Cannot delete or update a parent row: a foreign key constraint fails
```

这是因为建立主外键关系后， customers 现在和 orders 是父子表，orders表【依赖】于customers表，所以必须先删除 orders 表才能删除 customers 表，所以**应该把orders表（子表）的DROP语句放到最前面**：

```
CREATE DATABASE IF NOT EXISTS sql_store2;
USE sql_store2;
【DROP TABLE IF EXISTS orders;
DROP TABLE IF EXISTS customers;】

CREATE TABLE customers
(……);

CREATE TABLE orders
(……);
```

这样运行再多次也没问题了，总是可以从头建立sql_store2数据库和customers、orders两张表 （另外我觉得在数据库层及也应该用 `DROP DATABASE IF EXISTS sql_store2; + CREATE DATABASE sql_store2` 而非 Mosh 这里用的 `CREATE DATABASE IF NOT EXISTS`）

29. 更改主键和外键约束

Altering Primary and Foreign Key Constraints (2:10)

这一节学习如何在已经存在的表间创建和删除关系，**还是用 `ALTER TABLE` 语句 + `ADD`、`DROP` 关键词**，和27节修改表里的字段一样，**只不过这里增删的不是列而是外键**：

```
USE sql_store2;
ALTER TABLE orders
    DROP FOREIGN KEY fk_orders_customers,  -- orders_ibfk_1
    ADD FOREIGN KEY fk_orders_customers (customer_id)
        REFERENCES customers (customer_id)
        ON UPDATE CASCADE
        ON DELETE NO ACTION;
```

所以斌没有用像修改列那样用 MODIFY 关键词，而是直接 DROP + ADD 重置外键

注意

不知道为什么，我这里不管是之前第28节创建orders表时设置外键还是这里通过修改ADD增加外键，**外键名明明写的是 `fk_orders_customers`，实际上都会变成 `orders_ibfk_1`**，要去设计模式手动修改才行，可能是bug

另外也**可以通过类似的 ALTER TABLE 语句增删主键**：

```
USE sql_store2;
ALTER TABLE orders
    ADD PRIMARY KEY (order_id,……,……),
    -- 可增加多个主键，在括号内用逗号隔开，注意说的是 ADD 其实是重置，所以一次要把该表所有需要的主键声明完整
    DROP PRIMARY KEY;
    -- 删除主键不用声明，会直接删除所有主键
```

另外，**像增删主键这种既可以用菜单点击也可以用代码运行实现的操作（Workbench里这种操作相当多了）**，当忘记相关SQL代码写法时，可以通过菜单点击方式操作然后在 Review the SQL script 那一步看一看，就知道代码怎么写的了

30. 字符集和排序规则

Character Sets and Collations (6:29)

字符是以数字序列的形式储存于电脑中的，**字符集是数字序列与字符相互转换的字典，不同的字符集支持不同的字符范围**，有些支持拉美语言字符，有些也支持亚洲语言字符，有些支持全世界所有字符，**查看MySQL支持的所有字符集：**

```
SHOW CHARSET;
```

其中armscii8支持亚美尼亚语，big5支持繁体中文，**gb2312和gbk支持简体中文（gk是国标的拼音简称，k是扩展的拼音简称），而utf-8支持全世界的语言，utf-8也是MySQL自版本5之后的默认字符集。**

还可以看到字符集描述，默认排序规则，最大长度

排序规则（collation）指的是某语言内字符的排序方式，utf-8的默认排序规则是utf8_general_ci，其中ci表示case insensitive大小写不敏感，即MySQL在排序时不会区分大小写，这在大部分时候都是适用的，比如用户输入名字的时候大小写不固定，我们希望只按照字符顺序而不管大小写来对名字进行排序。总之，99.9%的情况下都不需要更改默认排序规则。

最大长度指的是对该字符集来说，给每个字符预留的最大字节数，如latin1是1字节，utf-8就是3 Byte，前面说过，在utf-8里，拉丁字符使用1字节，欧洲和中东字符使用2字节，亚洲语言的字符使用3字节，所以utf-8给每个字符预留3字节。

对于字符集来说，大部分时候用默认的utf-8就行了。但有时，我们可以通过更改字符集来减少空间占用，例如，我们某个特定的应用（对应的数据库）/特定表/特定列是只能输入英文字符的，那如果将该列的字符集从utf-8改为latin1，占用空间就会缩小到原来的1/3，以字段类型为CHAR(10)(固定预留10个字符)且有1百万条记录为例，占用空间就会从约30MB减到10MB。接下来将如何用菜单和代码方式更改库/表/列的字符集。

菜单方式更改字符集

右键sql_store2 数据库，点击 Schema Inspector（或者点击扳手图标旁边的那个i图标也一样），可以**查看**整个数据库以及各表各列的字符集和排序规则，Schema Inspector**也能查看**该数据库的主键外键、视图、触发器、储存程序、事务、函数等各种情况

要修改库或者表和列的字符集，直接点开库或者表的设计模式（扳手按钮）在里面选择更改即可（其中，打开表的设计模式后，上方可以改表的字符集和排序，下方可以改具体某列的字符集和排序），一般我们会让表和列的字符集和整个库保持一致，毕竟一个应用要不然是国际化的要不然就不是。

代码方式更改字符集

如果用 SQL 代码来进行更改的话，总的来说就是将设置字符集的语句 `CHARACTER SET latin1` 加上之前哪些创建/更改数据库/表/列的合适位置即可

1. 在创建数据库时设置字符集或更改已有**数据库字符集**

```
CREATE/ALTER DATABASE db_name
    【CHARACTER SET latin1】
```

2. 在创建表时设置字符集或更改已有**表的字符集**

```
CREATE/ALTER TABLE table1
(……)
【CHARACTER SET latin1】
```

3. 在创建表时设置**列的字符集**

就是在设置列时，将设置字符集的语句 `CHARACTER SET latin1` **加在字段类型和字段性质之间**

```
CREATE TABLE IF NOT EXISTS customers
(
    customer_id INT PRIMARY KEY AUTO_INCREMENT,
    first_name VARCHAR(50) 【CHARACTER SET latin1】 NOT NULL,
    points INT NOT NULL DEFAULT 0,
    email VARCHAR(255) NOT NULL UNIQUE
)

或

USE sql_store2;
ALTER TABLE customers
    MODIFY first_name VARCHAR(50) 【CHARACTER SET latin1】 NOT NULL,
    ADD    last_name  VARCHAR(50) CHARACTER SET latin1 NOT NULL AFTER first_name;
```

31. 存储引擎

Storage Engines (2:27)

在MySQL中我们有若干种储存引擎，**储存引擎决定了我们数据的储存方式以及可用的功能**

展示可用的储存引擎：

```
SHOW ENGINES;
```

储存引擎有很多，**我们真正需要知道只有两个：MyISAM（读作 My-I-SAM） 和 InnoDB**

MyISAM是曾经很流行的引擎，但**自MySQL5.5之后，默认引擎就改为InnoDB了，InnoDB支持更多的功能特性，包括事务、外键等等，所以最好使用InnoDB**

引擎是表层级的设置，每个表都可以设置不同的引擎（虽然这没必要）

外键是十分重要的，它可以增加引用一致性/完整性（referential integrity），如果我们有一个老数据库的引擎是 MyISAM，我们想要给它设置外键，就必须要将其引擎升级为 InnoDB，**可以在表的设计模式里选择更改，也可以用修改表的代码：**

```
ALTER TABLE customers
ENGINE = InnoDB;
```

注意

改变引擎是一个代价极高（expensive）的操作，它会重建整个表，在此期间无法方法访问数据。所以，不要轻易在生产环境中改变储存引擎，除非有特殊的理由。

【十四章】高效的索引

Indexing for High Performance (时长58分钟)

1. 介绍

Introduction (0:41)

这一章我们来看**提高性能的索引，索引对大型和高并发数据库非常有用，因为它可以显著提升查询的速度**

这一章我们将学习关于索引的一切，它们是**如何工作的，以及我们如何创造索引来提升查询速度，学习和理解这一章对于程序员和数据库管理员十分重要**

准备

打开 load_1000_customers.sql 并运行，该文件会向 sql_store 库的 customers 表插入上千条记录，这样我们就能看出索引对查询效率的影响

2. 索引

Indexes (2:49)

原理和作用

以寻找所在州（state）为 'CA' 的顾客为例，**如果没索引，MySQL就必须逐条扫描筛选所有记录。**索引，就好比书籍最后的那些关键词索引一样，按字母排序，这样就能迅速找到需要的关键词，所以**如果对state字段建立索引，也就是把state列单独拿出来分类排序并建立与原表顾客记录的对应关系，就可以通过该索引迅速找到在'CA'的顾客。**

另一方面，索引会比原表小得多，通常能够储存在内存中，而从内存读取数据总是比从硬盘读取快多了，这也会提升查询速度

如果**数据量**比较小，几百几千这种，没必要用索引，但如果是上百万的数据量，有无索引对查询时间的影响就很大了

注意

但建立索引也是有代价的，首先索引会占用内存，其次它会降低写入速度，因为每次修改数据时都会自动重建索引。所以不要对整个表建立索引，而只是针对关键的查询建立索引。

简化

严格来讲，应该用**二叉树**来描述索引，但只是为了学习如何操作索引的话没必要理解二叉树，所以这节课简化为用**表格**来展示索引以降低理解难度

导航

下节课我们学习如何创建索引

3. 创建索引

Creating Indexes (5:00)

案例

接着上面的例子，假设查询 'CA' 的顾客，**为了查看查询操作的详细信息，前面加上 `EXPLAIN` 关键字**

注意这里**只选择 customer_id 是有原因的，之后会解释**

```
EXPLAIN SELECT customer_id FROM customers WHERE state = 'CA';
```

有很多信息，目前我们只关注 type 和 rows

type 是 ALL 而 rows 是 1010 行，说明在没有索引的情况下，MySQL扫描了所有的记录。可用下面的语句确认customers表总共就是1010条记录

```
SELECT COUNT(*) FROM customers;
-- 1010
```

现在创建索引，**索引名习惯以idx或ix做前缀，后面的名字最好有意义，不要别取 idx_1、idx_2 这样没人知道是什么意思的名字**

```
【CREATE INDEX idx_state ON customers (state);】
```

再次运行加上EXPLAIN的解释查询语句

```
EXPLAIN SELECT customer_id FROM customers WHERE state = 'CA';
```

这次显示 **type 是 ref** 而 rows 只有 112，**扫面的行数显著减少，查询效率极大提升。**

另外，注意 possible keys 和 key 代表了 MySQL 找到的执行此查询可用的索引（之后会看到，可能不止一个）以及最终实际选择的最优的索引

练习

解释查询积分过千的顾客 id，建立索引后再来一次并对比两次结果

```
EXPLAIN SELECT customer_id FROM customers WHERE points > 1000;

CREATE INDEX idx_points ON customers (points);

EXPLAIN SELECT customer_id FROM customers WHERE points > 1000;
```

建立索引后的查询 **type 为 range**，表明我们查询的是一个取值范围的记录，扫描的行数 rows 从 1010 降为了 529，减了一半左右

导航

下节课学习如何查看索引

小结

解释性查询是在查询语句前加上 `EXPLAIN`

创建索引的语法：

```
CREATE INDEX 索引名 ON 表名 (列名);
-- 索引名通常是 idx_列名
```

4. 查看索引

Viewing Indexes (3:19)

实例1

查看 customers 表的索引：

```
SHOW INDEXES IN customers;
```

可以看到有三个索引，第一个是 MySQL 为主键 customer_id 创建的索引 PRIMARY，被称作**clustered index 聚合索引，每当我们为表创建主键时，MySQL就会自动为其创建索引，这样就能快速通过主键（通常是id）找到记录。**后两个是我们之前手动为state和points字段建立的索引 idx_state 和 idx_points，它们是 **secondary index 从属索引，MySQL在创建从属索引时会自动为其添加主键列，如每个idx_points索引的记录有两个值：客户的积分points和对应的客户编号customer_id，这样就可以通过客户积分快速找到对应的客户记录**

索引查询表中还列示了索引的一些性质，其中：

- Non_unique 是否是非唯一的，即是否是可重复的、可相同的，一般主键索引是0，其它是1
- Column_name 表明索引建立在什么字段上
- Collation 是索引内数据的排序方式，其中**A是升序，B是降序**
- **Cardinality（基数）表明索引中独特值/不同值的数量**，如PRIMARY的基数就是1010，毕竟每条记录都都有独特的主键，而另两个索引的基数都要少一些，从之前 Non_unique 为 1 也可以看得出来 state 和 points 有重复值，这里的基数可以更明确看到 state 和 points 具体有多少种不同的值
- **Index_type 都是BTREE（二叉树）**，之前说过MySQL里大部分的索引都是以二叉树的形式储存的，但Mosh把它们表格化了以使其更清晰易懂

注意

Cardinality 这里只是近似值而非精确值，要**先用以下语句重建顾客表的统计数据：**

```
ANALYZE TABLE customers;
```

然后再用 `SHOW INDEXES IN customers;` **得到的才是精确的 Cardinality 基数**

实例2

查看orders表的索引

```
SHOW INDEXES IN orders;
```

总共有四个：PRIMARY、fk_orders_customers_idx、fk_orders_shippers_idx、fk_orders_order_statuses_idx，第一个是建立在主键order_id上的聚合索引，**后三个是建立在三个外键customer_id、shipper_id、status上的从属索引。**

当我们建立表间连接时，MySQL会自动为外键添加索引，这样就能快速就行表连接（join tables）了

另外

还可以通过菜单方式查看某表中的索引，在左侧导航栏里 customers 表的子文件里就有一个 indexes 文件夹，点击里面的索引可以看到该索引的若干属性，**其中 visible（可见性） 表示其是否可用（enabeled）**

导航

下节课看如何索引字符串列

5. 前缀索引

Prefix Indexes (3:40)

当索引的列是字符串时（包括 CHAR、VARCHAR、TEXT、BLOG），尤其是当字符串较长时，我们通常不会使用整个字符串而是只是用字符串的前面几个字符来建立索引，这被称作 **Prefix Indexes 前缀索引**，这样可以减少索引的大小使其更容易在内存中操作，毕竟在内存中操作数据比在硬盘中快很多

案例

为 customers 表的 last_name 建立索引并且只使用其前20个字符：

```
CREATE INDEX idx_lastname ON customers (last_name【(20)】);
```

这个字符数的设定对于 CHAR 和 VARCHAR 是可选的，但对于 TEXT 和 BLOG 是必须的

最佳字符数

可最佳字符数如何确定呢？**太多了会使得索引太大难以在内存中运行，太少又达不到筛选的效果**，比如，只用第一个字符建立索引，那如果查找A开头的名字，索引可能会返回10万个结果，然后就必须对这10万个结果逐条筛选。

可以**利用 DISTINCT、LEFT 关键词和 COUNT 函数**来测试不同数目的前缀字符得到的独特值个数，**目标是用尽可能少的前缀字符得到尽可能多的独特值个数：**

```
SELECT
    COUNT(DISTINCT LEFT(last_name, 1)),
    COUNT(DISTINCT LEFT(last_name, 5)),
    COUNT(DISTINCT LEFT(last_name, 10))
FROM customers
```

结果是 '25', '966', '996'

可见从前1个到前5个字符，效果提升是很显著的，但从前5个到前10个字符，**所用的字符数增加了一倍但识别效应只增加了一点点**，再加上5个字符已经能识别出966个独特值，与1010的记录总数相去不远了，所以可以认为用前5个字符来创建前缀索引是最优的

导航

下节课学习一个很特殊又很强大的索引，全文索引

6. 全文索引

Fulltext Indexes (7:50)

这节课学习如何建立全文索引以支持全文检索的需求

案例

运行 create-db-blog.sql 得到 sql_blog 数据库，里面只包含一个 posts 表（文章表），每条记录就是一篇文章的编号 post_id、标题 title、内容 body 和 发布日期 data_published

假设我们创建了一个博客网站，里面有一些文章，并存放在上面这个 sql_blog 数据库里，**如何让用户可以对博客文章进行搜索呢**?

假设，用户想搜索包含 react 及 redux（两个前端的重要的 javascript 库）的文章，如果用LIKE操作符进行筛选：

```
USE sql_blog;
SELECT *
FROM posts
WHERE title LIKE '%react redux%'
    OR body LIKE '%react redux%';
```

有两个问题：

1. 在没有索引的情况下，会对所有文本进行全面扫描，效率低下。如果用上节课讲的前缀索引也不行，因为前缀索引只包含标题或内容开头的若干字符，若搜索的内容不在开头，以依然需要全面扫描
2. 这种搜索方式只会返回完全符合'%react redux%'的结果，但我们一般想搜索的是**包括这两个单词的任意一个或两个，任意顺序，中间有任意间隔的所有相关结果，即google式的模糊搜索**

我们通过**建立 Fulltext Index 全文索引**来实现这样的搜索

全文索引对相应字符串列的所有字符串建立索引，它就像一个字典，它会剔除掉in、the这样无意义的词汇并记录其他所有出现过的词汇以及每一个词汇出现过的一系列位置

建立全文索引：

```
CREATE FULLTEXT INDEX idx_title_body ON 【posts (title, body)】;
```

利用全文索引，结合 **MATCH 和 AGAINST 关键字** 进行google式的模糊搜索:

```
SELECT *
FROM posts
WHERE MATCH(title, body) AGAINST('react redux');
```

注意 MATCH 后的括号里必须包含全文索引 idx_title_body 建立时相关的所有列，不然会报错

还可以把 `MATCH(title, body) AGAINST('react redux')` 包含在选择语句里， 这样还能看到各结果的 **relevance score 相关性得分（一个0到1的浮点数）**，可以看出结果是按相关行**降序排列**的

```
SELECT *, 【MATCH(title, body) AGAINST('react redux')】
FROM posts
WHERE MATCH(title, body) AGAINST('react redux');
```

全文检索有两个模式：自然语言模式和布林模式，自然语言模式是默认模式，也是上面用到的模式。布林模式可以更明确地选择包含或排除一些词汇（google也有类似功能），如：

1. 尽量有 react，不要有 redux，必须有 form

```
……
WHERE MATCH(title, body) AGAINST('react 【-redux +form】' 【IN BOOLEAN MODE】);
```

2. 布林模式也可以实现**精确搜索**，就是将需要精确搜索的内容再用双引号包起来

```
……
WHERE MATCH(title, body) AGAINST('【"handling a form"】' IN BOOLEAN MODE);
```

小结

全文索引十分强大，如果你要建一个搜索引擎可以使用它，特别是要搜索的是长文本时，如文章、博客、说明和描述，否则，如果搜索**比较短的字符串**，比如名字或地址，就使用前置字符串

导航

接下来我们来看组合索引

7. 组合索引

Composite Indexes (5:12)

案例

查看customers表中的索引：

```
USE sql_store;
SHOW INDEXES IN customers;
```

目前有 PRIMARY、idx_state、idx_points 三个索引

之前只是对 state 或 points 单独进行筛选查询，现在我们要用 AND 同时**对两个字段进行筛选查询**，例如，查询所在州为 'CA' 而且积分大于 1000 的顾客id：

```
EXPLAIN SELECT customer_id
FROM customers
WHERE state = 'CA' AND points > 1000;
```

会发现 MySQL 在 idx_state、idx_points 两个候选索引最终选择了 idx_state，总共扫描了 112 行记录

相对于无索引时要扫描所有的1010条记录，这要快很多，但问题是，idx_state 这种单字段的索引**只做了一半的工作：它能帮助快速找到在 'CA' 的顾客，但要寻找其中积分大于1000的人时，却不得不回到磁盘里进行原表扫描（因为idx_state索引里没有积分信息）**，如果加州有一百万人的话这就会变得很慢。

所以我们要**建立 state 和 points 的组合索引：（两者的顺序其实很重要，下节课讲）**

```
CREATE INDEX idx_state_points ON customers (state, points);
```

再次运行之前的查询，发现在 idx_state、idx_points、idx_state_points 三个候选索引中 MySQL 发现组合索引对我们要做的查询而言是最优的因而选择了它，最终扫描的行数由112降到了58，速度确实提高了

之后会看到**组合索引也能提高排序的效率**

我们可以用 DROP 关键字删除掉那两个单列的索引

```sql
DROP INDEX idx_state ON customers;
DROP INDEX idx_points ON customers;
```

注意

新手爱犯的错误是给表里每一列都建立一个单独的索引，再加上MySQL会给每个索引自动加上主键，这些过多的索引会占用大量储存空间并且拖慢更新速度 （因为每次数据更新都会重建索引）

但实际中更多的是用到组合索引，所以不应该无脑地为每一列建立单独的索引而应该依据查询需求来建立合适的组合索引，一个组合索引最多可组合16列上，但一般4到6列的组合索引是比较合适的，但别把这个数字当作金科玉律，总是根据实际的查询需求和数据量来考虑

导航

下节课讲组合索引的顺序问题

8. 组合索引的列顺序

Order of Columns in Composite Indexes (9:16)

确定组合索引的列顺序时有**两个指导原则：**

1. **将最常使用的列放在前面**

关于组合索引有一个**重要原理**需要理解，比如 idx_state_lastname， 它是**先**对state建立分类排序的索引，然后**再**在同一州（如'CA'）内建立lastname的分类排序索引，所以这个索引**对两类查询有效**： 1. 单独针对state的查询（快速找到州） 2. 同时针对state和lastname的查询（快速找到州再在该州内快速找到该姓氏），但它对第 3 类查询——单独针对lastname的查询无效，因为它必须在每个州里去找该姓氏，相当于全面扫描了。**所以如果单独查找某州的需求存在的话**，就还需要**另外**为其单独建一个索引 idx_state

（总结一下：组合索引只对针对索引的组合的**前 n 个字段**的筛选有效）

基于对以上原理的理解，在建立组合索引时应该将最常用的列放在最前面，这样的索引会对更多的查询有效

2. **将基数（Cardinality）最大/独特性最高的列放在前面**

因为基数越大/独特性越高，起到的**筛选作用越明显，能够迅速缩小查询范围。**比如如果首先以性别来筛选，那只能将选择范围缩小到一半左右，但如果先以所在州来筛选，以总共20个州且每个州人数相当为例，那就会迅速将选择范围缩小到1/20

但最终仍然要根据实际的查询需求来决定，因为实际查询的筛选条件不一定能完全利用相应列的全部独特性，举例说明如下：

首先，为了比较的目的，针对 state 和 last_name 两列，同时建立两种顺序的索引 idx_state_lastname 和 idx_lastname_state

last_name 的独特性肯定是比 state 的独特性高的，可以用以下语句验证：

```sql
SELECT
    COUNT(DISTINCT state),
    COUNT(DISTINCT last_name)
FROM customers;
-- 48, 996
```

所以如果查询语句的筛选条件为 `WHERE state = 'CA' AND last_name = 'Smith'`，这种目标是特定州和特定姓氏的的查询能够充分利用各列独特性，肯定用 idx_lastname_state 先筛选姓氏能更快缩小范围提高效率

但如果进行姓氏的模糊查询，如，要查询 在加州 且 姓氏以A开头 的顾客，我们可以**用 `USE INDEX （索引名）` 子句来强制选择使用的索引**，对两种索引的查询结果进行对比：

```
EXPLAIN SELECT customer_id
FROM customers
USE INDEX (idx_state_lastname)
-- 注意括号
-- 注意位置：FROM之后WHERE之前
WHERE state = 'CA' AND last_name LIKE 'A%';
-- 7 rows

EXPLAIN SELECT customer_id
FROM customers
USE INDEX (idx_lastname_state)
WHERE state = 'CA' AND last_name LIKE 'A%';
-- 40 rows
```

会发现 idx_state_lastname 反而扫描的行数更少，效率更高，把查找的state换为'NY'也是一样。这是因为last_name的筛选条件是 'LIKE' 而不是 '='，约束性更小（less restrictive），更开放（more open），并没有充分利用姓氏列的高独特性，对于这种针对姓氏的模糊查找，先筛选州反而能更快缩小范围提高效率，所以 idx_state_lastname 更有效

当然，如果对两列都进行模糊查询，如查询语句的筛选条件变为 `WHERE state LIKE 'A%' AND last_name LIKE 'A%'`，可以想得到，验证也能证实，idx_lastname_state 再次胜出

总之，不仅要考虑各列的独特性高低，**也要考虑**常用的查询是否能充分利用各列的独特性，两者结合来决定组合索引里的排序，**不确定就测试对比验证**，所以，第二条原则也许应该改为**将常用查询实际利用到的独特性程度最高的列放在前面**

以上面的例子来说，如果业务中常用查询是特定州和特定姓（很可能）或者模糊州和模糊姓（不太可能），就用 idx_lastname_state 而 舍弃 idx_state_lastname **（不十分必要的索引不要保留，浪费空间和拖慢更新）**，相反，如果常用查询是特定州和模糊姓，就用 idx_state_lastname 而舍 idx_lastname_state

假设后一种情况成立，即只保留 idx_state_lastname，还要注意一点是，如前所述， idx_state_lastname 对 单独针对 last_name 的查询无效，如果有这样的查询需要就还要另外为该列建一个可用的索引 idx_lastname

思想

总之，任何一个索引都只对一类查询有效而且对特定的查询内容最高效，我们要现实一些，要去最优化那些性能关键查询，而不是所有可能的查询（optimize performance critical queries, not all queries in the world）

能加速所有查询的索引是不存在的，随着数据库以及查询需求的增长和扩展，我们可能需要建立不同列的不同顺序的组合索引

9. 索引无效时

When Indexes are Ignored (5:03)

有时你有一个可用的索引，但你的查询却未能充分利用它，这里我们看两种常见的情形：

案例1

查找在加州**或**积分大于1000的顾客id

注意之前查询的筛选条件都是**与（AND）**，这里是**或（OR）**

```sql
USE sql_store;
EXPLAIN SELECT customer_id
FROM customers
WHERE state = 'CA' OR points > 1000;
```

发现虽然显示 type 是 index，用的索引是 idx_state_points，但扫描的行数却是 1010 rows

因为这里是 或（OR） 查询，在找到加州的顾客后，仍然需要在每个州里去找积分大于 1000 的顾客，所以要扫描所有的 1010 条索引记录，即进行了 全索引扫描（full index scan）。当然**全索引扫描比全表扫描要快一点，因为前者只有三列而后者有很多列，前者在内存里进行而后者在硬盘里进行，但 全索引扫描 依然说明索引未被有效利用**，如果是百万条记录还是会很慢

我们需要**以尽可能充分利用索引地方式来编写查询，或者说以最迎合索引的方式编写查询**，就这个例子而言，可另建一个 idx_points 并将这个 OR 查询**改写为两部分，分别用各自最合适的索引，再用 UNION 融合结果（注意 UNION 是自动去重的，所以起到了和 OR 相同的作用，如果要保留重复记录就要用 UNION ALL，这里显然不是**）

```sql
CREATE INDEX idx_points ON customers (points);

EXPLAIN

        SELECT customer_id FROM customers
        WHERE state = 'CA'

    UNION

        SELECT customer_id FROM customers
        WHERE points > 1000;
```

结果显示，两部分查询中，MySQL分别自动选用了对该查询最有效的索引 idx_state_points 和 idx_points，扫描的行数分别为112和529，总共641行，相比于1010行有很大的提升

案例2

查询目前积分增加10分后超过2000分的顾客id:

```sql
EXPLAIN SELECT customer_id
FROM customers
WHERE points + 10 > 2010;
-- key: idx_points
-- rows: 1010
```

又变成了 1010 行全索引扫描，**因为 column expression 列表达式（列运算） 不能最有效地使用索引，要重写运算表达式，独立/分离此列（isolate the column）**

```sql
EXPLAIN SELECT customer_id FROM customers
WHERE points > 2000;
-- key: idx_points
-- rows: 4
```

直接从1010行降为4行，效率提升显著。所以想要MySQL有效利用索引，就总是**在表达式中将列独立出来**

导航

下节课讲用索引排序

10. 使用索引排序

Using Indexes for Sorting (7:02)

之前创建的索引杂七杂八的太多了，只保留 idx_lastname, idx_state_points 两个索引，把其他的 drop 了

```
USE sql_store;
SHOW INDEXES IN customers;
DROP INDEX idx_points ON customers;
DROP INDEX idx_state_lastname ON customers;
DROP INDEX idx_lastname_state ON customers;
SHOW INDEXES IN customers;
```

可以用 `SHOW STATUS`; 来查看Mysql服务器使用的众多变量，其中有个叫 'last_query_cost' 是上次查询的消耗值，我们可以用 `LIKE` 关键字来筛选该变量，即： `SHOW STATUS LIKE 'last_query_cost';`

按 state 给 customer_id 排序（**下节课讲为什么是customer_id**），再按 first_name 给 customer_id 排序，对比：

```
EXPLAIN SELECT customer_id
FROM customers
ORDER BY state;
-- type: index, rows: 1010, Extra: Using index
【SHOW STATUS LIKE 'last_query_cost';  】
-- cost: 102.749

EXPLAIN SELECT customer_id
FROM customers
ORDER BY first_name;
-- type: ALL, rows: 1010, Extra: Using filesort
SHOW STATUS LIKE 'last_query_cost';
-- cost: 1112.749
```

注意查看 Extra 信息，非索引列排序常常用的是 filesort 算法，从cost可以看到 filesort 消耗的资源几乎是用索引排序的10倍，这很好理解，因为索引就是对字段进行分类和排序，等于是已经提前排好序了

所以，不到万不得已不要给非索引数据排序，有可能的话尽量设计好索引用于查询和排序

但如之前所说，特定的索引只对特定的查询（取决于WHERE子句筛选条件）和排序（取决于ORDER BY排序条件）有效，这还是要从原理上理解：

> 以 idx_state_points 为例，它等于是先对state分类排序，再在同一个state内对points进行分类排序，再加上customer_id映射到相应的原表记录

所以，索引 idx_state_points 对于以下排序有效，**对最后那个“对加州范围内的顾客按积分排序” 为何有效，从刚刚的索引原理上也是很好理解的：**

```
ORDER BY state
ORDER BY state, points
【ORDER BY points WHERE state = 'CA' 】
```

相反，idx_state_points 对以下索引无效或只是部分有效，这些都是会部分或全部用到 filesort 算法的：

```
ORDER BY points
【ORDER BY points, state】
【ORDER BY state, first_name, points】
```

总的来说一个组合索引对于按它的组合列 **“从头按顺序”** 进行的 WHERE 筛选和 ORDER BY 排序最有效

对于 ORDER BY 子句还有一个问题是**升降序**，索引本身是升序的，但可以 Backward index scan 倒序索引扫描，所以它对所有同向的（同升序或同降序）的 ORDER BY 子句都有效，但对于升降序**混合方向**的 ORDER BY 语句则不够有效，还是以 idx_state_points 为例，对以下 ORDER BY 子句有效，即完全是 Using index 且 cost 在一两百以内：

```
ORDER BY state
ORDER BY state DESC
ORDER BY state, points
ORDER BY state DESC, points DESC
```

但下面这两种就不能充分利用 idx_state_points，会部分使用 filesort 算法且 cost > 1000

```
ORDER BY state, points DESC
ORDER BY state DESC, points
```

总结

特定索引只对特定查询和排序最有效，而且这些从索引的原理上都很好理解

建立什么索引取决于查询和排序需求，而查询和排序也要尽量去迎合索引以尽可能提高效率

11. 覆盖索引

Covering Indexes (1:58)

这节课讲为什么之前 SELECT 选择子句里只选 customer_id 这一个字段

实例

以 state 排序查询 customers 表，每次 SELECT 不同的列并对比结果：

```
USE sql_store;

-- 1. 只选择 customer_id:
EXPLAIN SELECT customer_id
FROM customers
ORDER BY state;
SHOW STATUS LIKE 'last_query_cost';

-- 2. 选择 customer_id 和 state:
EXPLAIN SELECT
    customer_id,
    state
FROM customers
ORDER BY state;
SHOW STATUS LIKE 'last_query_cost';

-- 3. 选择所有字段:
EXPLAIN SELECT *
FROM customers
ORDER BY state;
SHOW STATUS LIKE 'last_query_cost';
```

会验证发现前两次是完全 Using index 而且 cost 均只有两百左右，而第3种是 Using filesort 而且 cost 超过一千，这**从 idx_state_points 的原理上也很好理解：**

前面提到过，从属索引除了包含相关列还会自动包含主键列（通常是某种id列）来和原表中的记录建立对应关系，所以 组合索引 idx_state_points 中包含三列：state、points 以及 customer_id，所以如果 SELECT 子句里选择的列是这三列中的一列或几列的话，整个查询就可以在只使用索引不碰原表的情况下完成，这叫作覆盖索引（covering index），即索引满足了查询的所有需求，这是最快的

总结

设计索引时，先看 WHERE 子句，看看最常用的筛选字段是什么，把它们包含在索引中，这样就能迅速缩小查找范围，**其次查看 ORDER BY 子句**，看看能不能将这些列包含在索引中，**最后，看看 SELECT 子句中的列，如果你连这些也包含了，就得到了覆盖索引，MySQL 就能只用索引就完成你的查询，实现最快的查询速度**

12. 维护索引

Index Maintenance (1:25)

索引维护注意三点：

1. **重复**索引（duplicate index）：

MySQL 不会阻止你建立重复的索引，所以记得在建立新索引前前检查一下已有索引。验证后发现，具体而言：

同名索引是不被允许的：

```
CREATE INDEX idx_state_points ON customers (state, points);
-- Error Code: 1061. Duplicate key name 'idx_state_points'
```

对相同列的相同顺序建立不同命的索引，5.7 版本暂时允许，但 8.0 版本不被允许：

```
CREATE INDEX idx_state_points2 ON customers (state, points);
-- warning(s): 1831 Duplicate index 'idx_state_points2' defined on the table
'sql_store.customers'. This is deprecated and will be disallowed in a future release.
Records: 0  Duplicates: 0  Warnings: 1
```

2. **冗余**索引(redundant index)：

比如已有 idx_state_points，那 idx_state 就是冗余的了，因为所有 idx_state 能**满足的筛选和排序需求** idx_state_points 都能满足

但 idx_points 和 idx_points_state 不是冗余的，因为它们**可以满足不同的筛选和排序需求**

3. 无用索引（unused index）：

这个很好理解，就是那些**常用查询、排序用不到的**索引没必要建立，毕竟索引是会占用空间和拖慢数据更新速度的

小结

要做好索引管理：

1. **在新建索引时，总是先查看一下现有索引，避免重复、冗余、无用的索引，这是最基本的要求。**
2. 其次，索引本身要是要占用空间和拖慢更新速度的所以也是有代价的，而且不同索引对不同的筛选、排序、查询内容的有效性不同，因此，理想状态下，索引管理也应该是个**根据业务查询需求需要不断去权衡成本效益，抓大放小，迭代优化的过程**

13. 性能最佳实践 (文档)

Performance Best Practices

课程资料中有一个 Performance Best Practices.pdf 文件，总结了课程中提到过的性能最佳实践，翻译如下：

1. 较小的表性能更好。不要存储不需要的数据。**解决今天的问题**，而不是明天可能永远不会发生的问题。
2. **使用尽可能小的数据类型。**如果你需要存储人们的年龄，一个TINYINT就足够了，无需使用INT。对于一个小的表来说，洁身几个字节没什么大不了的，但在包含数百万条记录的表中却具有显著的影响。
3. 每个表都必须有一个**主键。**
4. 主键应短。**如果您只需要存储一百条记录**，最好选择 TINYINT 而不是 INT。
5. 首选**数字**类型而不是字符串作为**主键。**这使得通过主键查找记录更快。
6. 避免 BLOB。它们会增加数据库的大小，并会对性能产生负面影响。如果可以，请将文件存储在磁盘上。
7. 如果表的**列太多**，请考虑使用一对一关系将其拆分为两个相关表。这称为垂直分区（vertical partitioning）。例如，您可能有一个包含地址列的客户表。**如果这些地址不经常被读取，请将表拆分为两个表**（users 和 user_addresses）。
8. **相反，如果由于数据过于碎片化而总是需要在查询中多次使用表联接，则可能需要考虑对数据反归一化。反归一化与归一化相反。它涉及把一个表中的列合并到另一个表（以减少联接数）（如之前 airports 里的 city 和 state）。**
9. 请考虑为昂贵的查询创建**摘要/缓存表。**例如，如果获取论坛列表和每个论坛中的帖子数量的查询非常昂贵，请创建一个名为 forums_summary 的表，其中包含论坛列表及其中的帖子数量。您**可以使用事件定期刷新此表中的数据。您还可以使用触发器在每次有新帖子时更新计数。**
10. **全表扫描**是查询速度慢的一个主要原因。使用 EXPLAIN 语句并查找类型为 "ALL" 的查询。这些是全表扫描。使用索引优化这些查询。
11. 在**设计索引时，**请先查看WHERE子句中的列。这些是第一批候选人，因为它们有助于缩小搜索范围。接下来，查看 ORDER BY 子句中使用的列。如果它们存在于索引中，MySQL 可以扫描索引以返回有序的数据，而无需执行排序操作（filesort）。最后，考虑将 SELECT 子句中的列添加到索引中。这为您提供了覆盖索引，能覆盖你查询的完整需求。MySQL 不再需要从原表中检索任何内容。
12. 选择**组合索引**，而不是多个单列索引。
13. 索引中的列**顺序**很重要。将最常用的列和基数较高的列放在第一位，但始终考虑您的查询。
14. **删除重复、冗余和未使用的索引。**重复索引是同一组具有相同顺序的列上的索引。冗余索引是不必要的索引，可以替换为现有索引。例如，如果在列（A、 B）上有索引，并在列 （A）上创建另一个索引，则后者是冗余的，因为前一个索引可以满足相同的需求。
15. 在**分析现有索引**之前，不要创建新索引。
16. **在查询中隔离你的列**，以便 MySQL 可以使用你的索引。
17. **避免 SELECT *。**大多数时候，选择所有列会忽略索引并返回您可能不需要的不必要的列。这会给数据库服务器带来额外负载。
18. 只返回你需要的行。使用 **LIMIT 子句**限制返回的行数。
19. **避免使用前导通配符的LIKE 表达式（eg."%name"）。**
20. 如果您有一个使用 **OR 运算符**的速度较慢的查询，请考虑将查询分解为两个使用单独索引的查询，并使用 UNION 运算符组合它们。

延伸阅读：

- 施瓦茨 (Baron Schwartz) 《高性能MySQL》
- Tapio Lahdenmaki 《数据库索引设计与优化》

【十五章】保护数据库

Securing Databases (时长20分钟)

1. 介绍

Introduction (0:33)

导航

之前都是介绍本地数据库而你自己就是数据库的唯一用户，所以不必考虑安全问题。

但实际业务中数据库大多放在服务器里，你必须妥善处理好用户账户和权限的问题，**合理决定谁拥有什么程度的权限以防止对数据的破坏和误用**

这一章，我们学习如何**增强数据库的安全性**

2. 创建一个用户

Creating a User (3:12)

到目前为止我们一直使用的是 **root 用户帐户，这是在安装 MySQL 时就设置的根账户**

在生产环境中我们需要创建新用户并合理分配权限

例如，你有一个应用程序及其相关联的数据库，你要让使用你应用程序的用户拥有读写数据的权限，但他们不应该有改变数据库结构的权限，如创建和删除一张表，否则会出大问题

又如，你新招了一个DBA（数据库管理员），你需要给他新建一个账户，让他可以访问一个或多个数据库乃至整个MySQL服务器

导航

这节课我们学习如何创建帐户，之后学习如何设置权限

实例

设置一个新用户，用户名为 john，可以选择**用 @ 来限制他可以从哪些地方（可以是 主机名、ip 地址、域名）访问数据库**

（其实还是困惑 【主机、端口、ip、域名】 这些东西的区别）

```
-- @ 从哪里访问
CREATE USER john
-- 无限制，可从任何位置访问
CREATE USER john@127.0.0.1;
-- 限制ip地址，可以是特定电脑，也可以是特定网络服务器 web server
CREATE USER john@localhost;
-- 限制主机名，特定电脑
CREATE USER john@'codewithmosh.com';
-- 限制域名（【后面实验发现其实加不加引号都一样】），可以是该域名内的任意电脑，但子域名则不行
CREATE USER john@'%.codewithmosh.com';
--  【加上了通配符，可以是该域名及其子域名下的任意电脑】
```

可以用 `IDENTIFIED BY` 来设置密码

```
CREATE USER john IDENTIFIED BY '1234'
-- 可从任何地方访问，但密码为 '1234'，注意引号
-- 该密码只是为了简化，请总是用很长的强密码
```

导航

下节课讲如何查看服务器上的用户

3. 查看用户

Viewing Users (1:29)

假设上节课我们最后以 `CREATE USER john` 方式创建了一个新账户 john，无限制，无密码

用两种方式可以查看MySQL服务器上的所有用户：

法1

在一个自动创建的名为 mysql 的数据库（导航里似乎是隐藏了看不到）里，有个 user 表记录了帐户信息，查询即可：

```
SELECT * FROM mysql.user;
```

可以看到罗列出的所有用户，**除了 john 和 root 帐户，还有几个 MySQL 内部自动建立和使用的帐户（用户名均为 mysql.*）**

Host 字段表示用户可以从哪里访问数据库，john 是一个通配符 %，表示他可以从任意位置访问数据库，其它几个用户都是 localhost，表示都只能从本电脑访问数据库，不能从远程连接访问

后面的一系列字段都是各种权限的配置，后面会细讲

法2

也可以直接点击左侧导航栏的 Administration 标签页里的 Users and Privileges，同样可以查看服务器上的用户列表和信息

导航

下节课讲如何删除用户

4. 删除用户

Dropping Users (0:48)

案例

假设之前创建了bob的帐户，允许在 codewithmosh.com 域名内访问数据库，密码是'1234'：

```
CREATE USER bob@codewithmosh.com IDENTIFIED BY '1234';
```

之后bob离开了组织，就应该删除它的账户，注意依然要在用户名后跟上 `@主机名（host）`

```
DROP USER bob@codewithmosh.com;
-- 注意：实验发现不管是创建时还是删除时，域名/主机名加不加引号都可以，甚至创建时加引号删除时不加引号或者反过来都没问题
```

最佳实践

记得总是及时删除掉组织中那些不用的账户

导航

下节课讲如何修改密码

5. 修改密码

Changing Passwords (1:06)

人们时常忘记自己的密码，作为管理员，你时常被要求修改别人的或自己的密码，这很简单，有两种方法：

法1

用 SET 语句

```
SET PASSWORD [FOR john] = '1234';
-- 修改john的密码
-- for 都来了，这条语句也太大白话了，如果把'='改成as的话简直就是一句完整的英语句子了
-- 省略[for john]就是修改当前登录账户的密码
```

法2

用导航面板：**还是在 Administration 标签页 Users and Privileges 里**，点击用户 john，可修改其密码，最后记得点 Apply 应用。另外**还可以点击 Expire Password 强制其密码过期，下次用户登录必须修改密码。**

导航

以上是关于用户帐户的内容，接下来我们讲权限

6. 权限许可

Granting Privileges (4:53)

创建用户后需要分配权限，最常见的是两种情形：

常见情形1. 对于网页或桌面应用程序的使用用户，给予其读写数据的权限，但禁止其增删表或修改表结构

例如，我们有个叫作 moon 的应用程序，我们给这个应用程序建个用户帐户名为 moon_app (app指明这代表的是**整个应用程序而非一个人**)

```
CREATE USER moon_app IDENTIFIED BY '1234';
```

给予其对 sql_store 数据库增删查改**以及执行储存过程（EXECUTE）的权限，这是给终端用户常用的权限配置**

```
GRANT SELECT, INSERT, UPDATE, DELETE, EXECUTE
-- GRANT子句表明授予哪些权限
ON sql_store.*
-- ON子句表明可访问哪些数据库和表
-- ON sql_store.*代表可访问某数据库所有表，常见设置
-- 只允许访问特定表则是 ON sql_store.customers，不常见
TO moon_app;
-- 表明授权给哪个用户
-- 如果该用户有【访问地址限制】，如： @ip地址/域名/主机名，也要加上
```

这样就完成了权限配置

我们来测试一下，**用这个新账户 moon_app 建立新连接**（点击 workbench 主页 MySQL connections处的加号按钮）：

将连接名（Connection Name）设置为：moon_app_connection；
主机名（Hostname）和端口（Post）是自动设置的，分别为：127.0.0.1 和 3306；
用户名（Username）和密码（Password）输入建立时的设置的用户名和密码：moon_app 和 1234

在新连接里测试，发现果然**只能访问**sql_store数据库而不能访问其他数据库（实际上导航面板**只会显示**sql_store数据库）

```
USE sql_store;
SELECT * FROM customers;

USE sql_invoicing;
-- Error Code: 1044. 【Access denied】 for user 'moon_app'@'%' to database 'sql_invoicing'
```

常见情形2. 对于管理员，给予其一个或多个数据库乃至整个服务器的管理权限，这不仅包括表中数据的读写，还包括增删表、修改表结构以及创建事务和触发器等

可以**谷歌 MySQL privileges**，第一个结果就是官方文档里罗列的所有可用的权限及含义，其中的 `ALL` 是最高权限，通常我们给予管理员 **`ALL` 权限**

```
GRANT ALL
ON sql_store.*
-- 【*.*】 代表所有数据库的所有表或整个服务器
TO john;
```

导航

以上就是权限配置的两种最常见情形，接下来讲如何查看已经给出的权限

7. 查看权限

Viewing Privileges (1:34)

查看以给出的权限**仍然有 SQL语句 和 导航菜单 两种方法：**

法1

查看john的权限

```
SHOW 【GRANTS】 [FOR john];
-- 注意之前授予权限时是 GRANT，没有S
-- 省略 FOR john，即查看当前登录帐户的权限
```

可以看到，当前 root 帐户拥有最高权限，**除了 ALL 的所有权限，还有另外一个叫 PROXY 的权限。**感觉 root 帐户和 john 这样的 DBA 帐户的区别就跟群主和群管理员的区别一样

法2

依然可以通过导航栏 Administration 标签页里的 Users and Privileges 来查看各用户的权限，其中 Administrative Roles 展示了该用户的角色（Roles, 如 DBA，有很多可选项，感觉像是预设权限组合）和全局权限（Global Privileges），而 Schema Privileges 则显示该用户在特定数据库的权限，因为 root 和 john 的权限是针对所有数据库的，所以没有特定数据库权限而 moon_app 就显示有针对sql_store数据库的权限，所有这些都是可以选择和更改的，记得最后要点Apply应用

导航

下节课讲如何撤销权限

8. 撤销权限

Revoking/Dropping Privileges (1:20)

有时你可能发现给某人的权限给错了，或者给某人的权限过多，结果他开始滥用权限，这节课学习如何**收回权限，很简单，和给予权限很类似**

案例

之前说过，应该只给予 moon_app 读写 sql_store 数据库的表内数据以及执行储存过程的权限，假设我们错误的给予了其**创建视图的权限：**

```
GRANT CREATE VIEW
ON sql_store.*
TO moon_app;
```

要收回此权限，只用把语句中的 **GRANT 变 REVOKE，把 TO 变 FROM 就行了**，就这么简单：

```
REVOKE CREATE VIEW
ON sql_store.*
FROM moon_app;
```

思想

不要给予一个账户过多权限，总是给予他所必须的最小权限，不然就在系统中创造了太多的安全漏洞

9. 总结

Wrap Up (0:44)

全面掌握SQL课程 到此结束，欢迎大家到 codewithmosh.com 继续学习 Mosh 的其它编程课程~

Mosh完全掌握SQL课程学习笔记
数据概要