

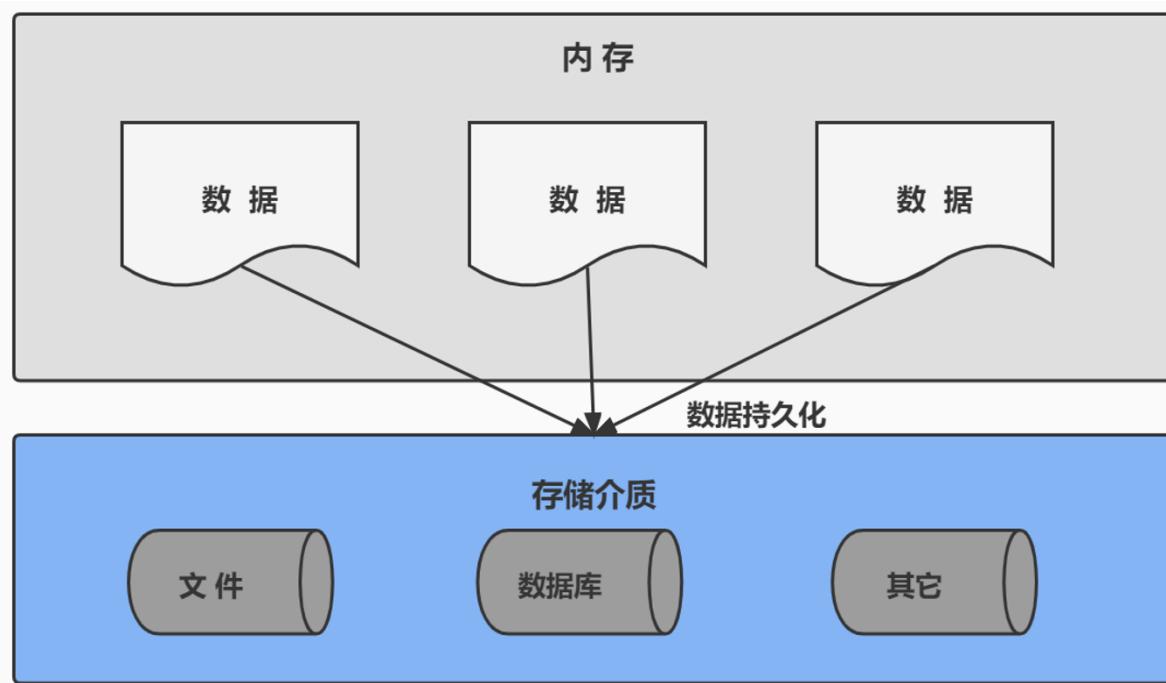
第01章_数据库概述

讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

1. 为什么要使用数据库

- 持久化(persistence)：**把数据保存到可掉电式存储设备中以供之后使用**。大多数情况下，特别是企业级应用，**数据持久化意味着将内存中的数据保存到硬盘上加以“固化”**，而持久化的实现过程大多通过各种关系数据库来完成。
- 持久化的主要作用是**将内存中的数据存储在关系型数据库中**，当然也可以存储在磁盘文件、XML数据文件中。



生活中的例子：



2. 数据库与数据库管理系统

2.1 数据库的相关概念

DB: 数据库 (Database)

即存储数据的“仓库”，其本质是一个文件系统。它保存了一系列有组织的数据。

DBMS: 数据库管理系统 (Database Management System)

是一种操纵和管理数据库的大型软件，用于建立、使用和维护数据库，对数据库进行统一管理和控制。用户通过数据库管理系统访问数据库中表内的数据。

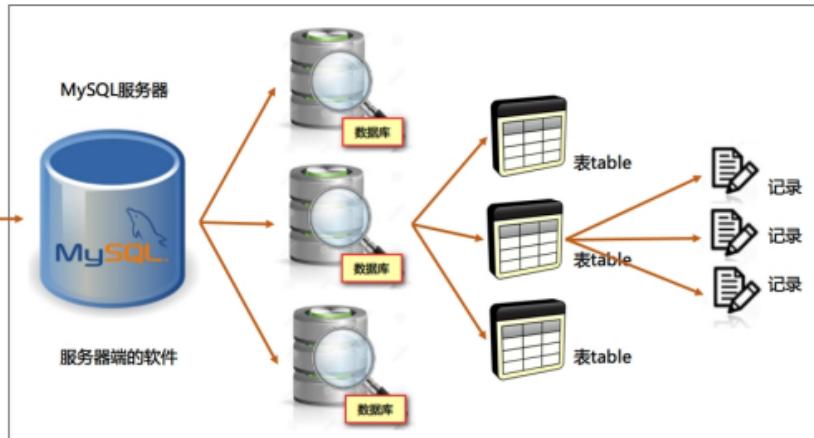
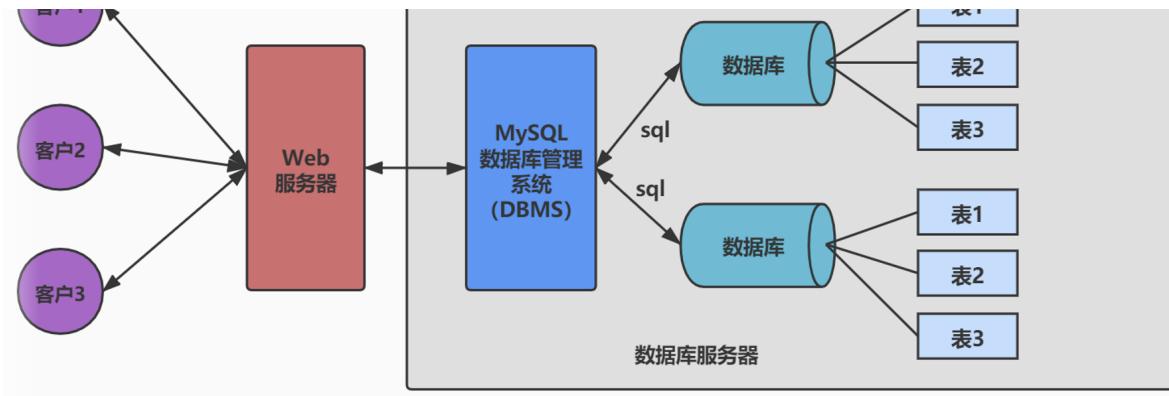
SQL: 结构化查询语言 (Structured Query Language)

专门用来与数据库通信的语言。

2.2 数据库与数据库管理系统的关系

数据库管理系统(DBMS)可以管理多个数据库，一般开发人员会针对每一个应用创建一个数据库。为保存应用中实体的数据，一般会在数据库创建多个表，以保存程序中实体用户的数据。

数据库管理系统、数据库和表的关系如图所示：



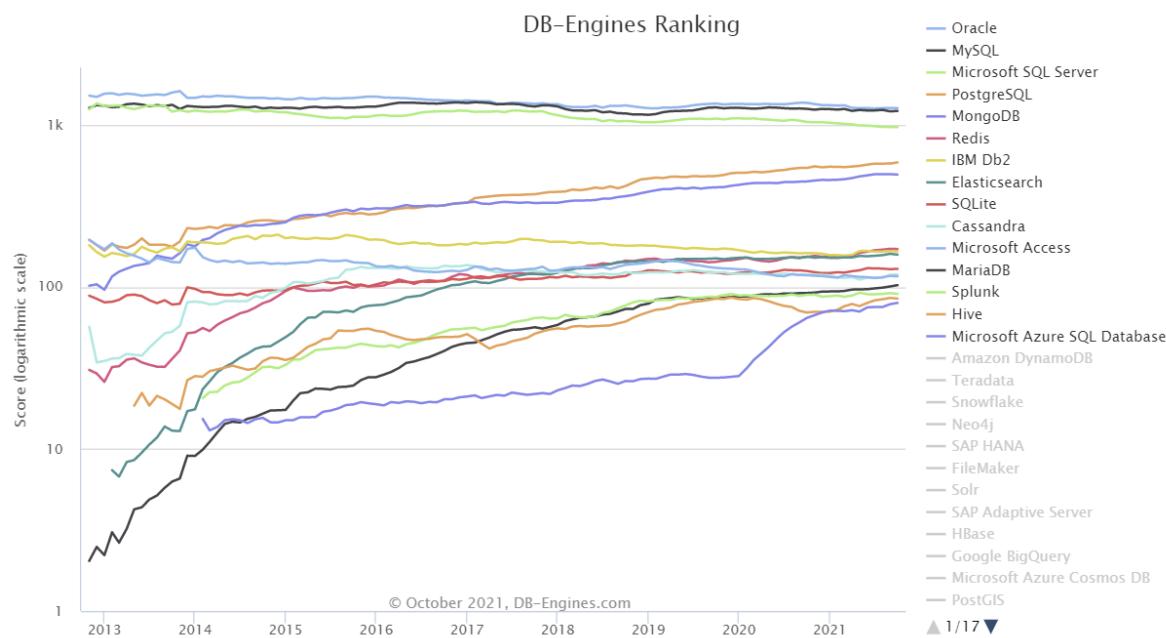
2.3 常见的数据库管理系统排名(DBMS)

目前互联网上常见的数据库管理软件有Oracle、MySQL、MS SQL Server、DB2、PostgreSQL、Access、Sybase、Informix这几种。以下是2021年**DB-Engines Ranking** 对各数据库受欢迎程度进行调查后的统计结果：（查看数据库最新排名：<https://db-engines.com/en/ranking>）

				DBMS	Database Model	Score		
Oct 2021	Sep 2021	Oct 2020	Oct 2021			Oct 2021	Sep 2021	Oct 2020
1.	1.	1.	1270.35	Oracle	Relational, Multi-model	-1.19	-98.42	
2.	2.	2.	1219.77	MySQL	Relational, Multi-model	+7.24	+36.61	
3.	3.	3.	970.61	Microsoft SQL Server	Relational, Multi-model	-0.24	-72.51	
4.	4.	4.	586.97	PostgreSQL	Relational, Multi-model	+9.47	+44.57	
5.	5.	5.	493.55	MongoDB	Document, Multi-model	-2.95	+45.53	
6.	6.	8.	171.35	Redis	Key-value, Multi-model	-0.59	+18.07	
7.	7.	6.	165.96	IBM Db2	Relational, Multi-model	-0.60	+4.06	
8.	8.	7.	158.25	Elasticsearch	Search engine, Multi-model	-1.98	+4.41	
9.	9.	9.	129.37	SQLite	Relational	+0.72	+3.95	
10.	10.	10.	119.28	Cassandra	Wide column	+0.29	+0.18	
11.	11.	11.	116.38	Microsoft Access	Relational	-0.56	-1.87	
12.	12.	12.	102.59	MariaDB	Relational, Multi-model	+1.90	+10.82	
13.	13.	13.	90.61	Splunk	Search engine	-0.99	+1.21	
14.	14.	15.	84.74	Hive	Relational	-0.83	+15.19	
15.	15.	17.	79.72	Microsoft Azure SQL Database	Relational, Multi-model	+1.46	+15.32	
16.	16.	16.	76.55	Amazon DynamoDB	Multi-model	-0.38	+8.14	
17.	17.	14.	69.83	Teradata	Relational, Multi-model	+0.15	-5.96	
18.	21.	64.	58.26	Snowflake	Relational	+6.19	+52.32	
19.	18.	21.	57.87	Neo4j	Graph	+0.24	+6.53	
20.	19.	19.	55.28	SAP HANA	Relational, Multi-model	-0.96	+1.04	

351. ▲ 352. ▼ 310. Sadas Engine	Relational	0.00	+0.00	-0.10
353. ▲ 354. ▼ 325. ActorDB	Relational	0.00	±0.00	-0.05
353. ▲ 354. ▼ 341. BergDB	Key-value	0.00	±0.00	±0.00
353. ▲ 354. ▼ 341. Cachelot.io	Key-value	0.00	±0.00	±0.00
353. ▼ 347. ▼ 341. CovenantSQL	Relational	0.00	-0.01	±0.00
353. ▲ 354. ▼ 341. DaggerDB	Relational	0.00	±0.00	±0.00
353. ▲ 354. ▼ 341. Edge Intelligence	Relational	0.00	±0.00	±0.00
353. ▲ 354. FiredrakeDB	Relational	0.00	±0.00	±0.00

对应的走势图： (https://db-engines.com/en/ranking_trend)



2.4 常见的数据库介绍

Oracle

1979年，Oracle 2诞生，它是第一个商用的RDBMS（关系型数据库管理系统）。随着Oracle软件的名气越来越大，公司也改名叫Oracle公司。

2007年，总计85亿美金收购BEA Systems。

2009年，总计74亿美金收购SUN。此前的2008年，SUN以10亿美金收购MySQL。意味着Oracle同时拥有了MySQL的管理权，至此Oracle在数据库领域中成为绝对的领导者。

2013年，甲骨文超越IBM，成为继Microsoft后全球第二大软件公司。

如今Oracle的年收入达到了400亿美金，足以证明商用（收费）数据库软件的价值。

SQL Server

SQL Server是微软开发的大型商业数据库，诞生于1989年。C#、.net等语言常使用，与WinNT完全集成，也可以很好地与Microsoft BackOffice产品集成。

DB2

IBM公司的数据库产品，收费的。常应用在银行系统中。

PostgreSQL

PostgreSQL的稳定性极强，最符合SQL标准，开放源码，具备商业级DBMS质量。PG对数据量大的文本以及SQL处理较快。

SQLite

嵌入式的小型数据库，应用在手机端。零配置，SQLite3不用安装，不用配置，不用启动，关闭或者配置数据库实例。当系统崩溃后不用做任何恢复操作，再下次使用数据库的时候自动恢复。

informix

IBM公司出品，取自Information 和Unix的结合，它是第一个被移植到Linux上的商业数据库产品。仅运行于unix/linux平台，命令行操作。性能较高，支持集群，适应于安全性要求极高的系统，尤其是银行，证券系统的应用。

3. MySQL介绍



3.1 概述

- MySQL是一个 **开放源代码的关系型数据库管理系统**，由瑞典MySQL AB（创始人Michael Widenius）公司1995年开发，迅速成为开源数据库的 No.1。
- 2008被 **Sun** 收购（10亿美金），2009年Sun被 **Oracle** 收购。**MariaDB** 应运而生。（MySQL的创造者担心 MySQL 有闭源的风险，因此创建了 MySQL 的分支项目 MariaDB）
- MySQL6.x 版本之后分为 **社区版** 和 **商业版**。
- MySQL是一种关联数据库管理系统，将数据保存在不同的表中，而不是将所有数据放在一个大仓库内，这样就增加了速度并提高了灵活性。
- MySQL是开源的，所以你不需要支付额外的费用。
- MySQL是可以定制的，采用了 **GPL (GNU General Public License)** 协议，你可以修改源码来开发自己的MySQL系统。
- MySQL支持大型的数据库。可以处理拥有上千万条记录的大型数据库。
- MySQL支持大型数据库，支持5000万条记录的数据仓库，32位系统表文件最大可支持 **4GB**，64位系统支持最大的表文件为 **8TB**。
- MySQL使用 **标准的SQL数据语言** 形式。
- MySQL可以允许运行于多个系统上，并且支持多种语言。这些编程语言包括C、C++、Python、Java、Perl、PHP和Ruby等。

3.2 MySQL发展史重大事件

MySQL的历史就是整个互联网的发展史。互联网业务从社交领域、电商领域到金融领域的发展，推动着应用对数据库的需求提升，对传统的数据库服务能力提出了挑战。高并发、高性能、高可用、轻资源、易维护、易扩展的需求，促进了MySQL的长足发展。



1.4 关于MySQL 8.0

MySQL从5.7版本直接跳跃发布了8.0版本，可见这是一个令人兴奋的里程碑版本。MySQL 8版本在功能上做了显著的改进与增强，开发者对MySQL的源代码进行了重构，最突出的一点是多MySQL Optimizer优化器进行了改进。不仅在速度上得到了改善，还为用户带来了更好的性能和更棒的体验。



为什么如此多的厂商要选用MySQL？大概总结的原因主要有以下几点：

1. 开放源代码，使用成本低。
2. 性能卓越，服务稳定。
3. 软件体积小，使用简单，并且易于维护。
4. 历史悠久，社区用户非常活跃，遇到问题可以寻求帮助。
5. 许多互联网公司在用，经过了时间的验证。

1.6 Oracle vs MySQL

Oracle 更适合大型跨国企业的使用，因为他们对费用不敏感，但是对性能要求以及安全性有更高的要求。

MySQL 由于其**体积小、速度快、总体拥有成本低，可处理上千万条记录的大型数据库，尤其是开放源码这一特点，使得很多互联网公司、中小型网站选择了MySQL作为网站数据库**（Facebook, Twitter, YouTube, 阿里巴巴/蚂蚁金服, 去哪儿, 美团外卖, 腾讯）。

4. RDBMS 与非RDBMS

从排名中我们能看出来，关系型数据库绝对是 DBMS 的主流，其中使用最多的 DBMS 分别是 Oracle、MySQL 和 SQL Server。这些都是关系型数据库（RDBMS）。

4.1 关系型数据库(RDBMS)

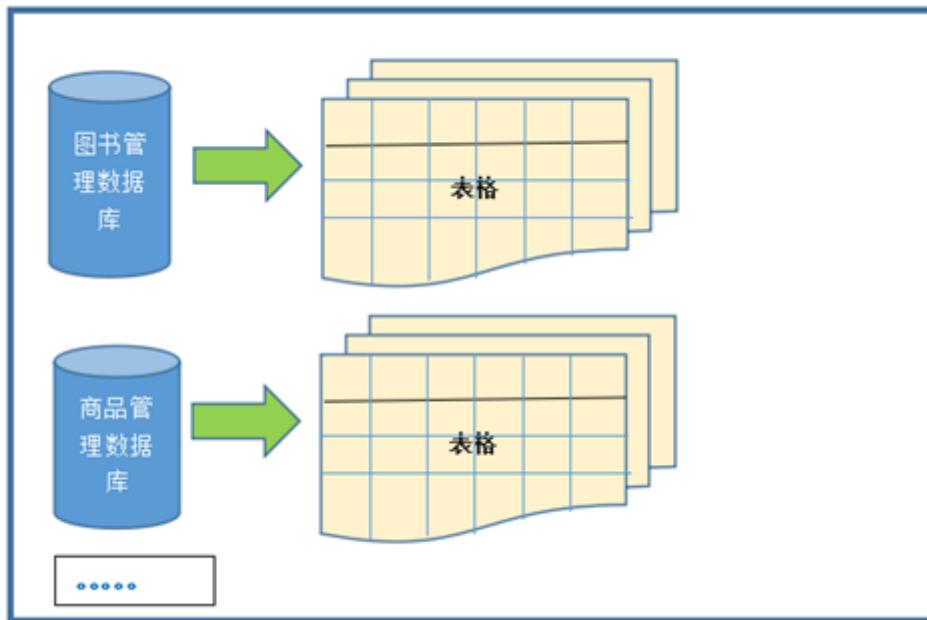
4.1.1 实质

- 这种类型的数据库是 **最古老** 的数据库类型，关系型数据库模型是把复杂的数据结构归结为简单的 **二元关系**（即二维表格形式）。

Tno	Tname	Tsex	Department	
T001	石云丹	女	计算机系
T002	罗莉	女	计算机系
T003	王国强	男	计算机系
T004	吴栋	男	计算机系
T005	高鸿轩	男	数学系
T006	张怀良	男	数学系
T007	刘晓伟	男	数学系
T108	马莉莲	女	物电学院
.....

- 关系型数据库以 **行(row)** 和 **列(column)** 的形式存储数据，以便于用户理解。这一系列的行和列被

- SQL 就是关系型数据库的查询语言。



4.1.2 优势

- **复杂查询** 可以用SQL语句方便的在一个表以及多个表之间做非常复杂的数据查询。
- **事务支持** 使得对于安全性能很高的数据访问要求得以实现。

4.2 非关系型数据库(非RDBMS)

4.2.1 介绍

非关系型数据库，可看成传统关系型数据库的功能 **阉割版本**，基于键值对存储数据，不需要经过SQL层的解析，**性能非常高**。同时，通过减少不常用的功能，进一步提高性能。

目前基本上大部分主流的非关系型数据库都是免费的。

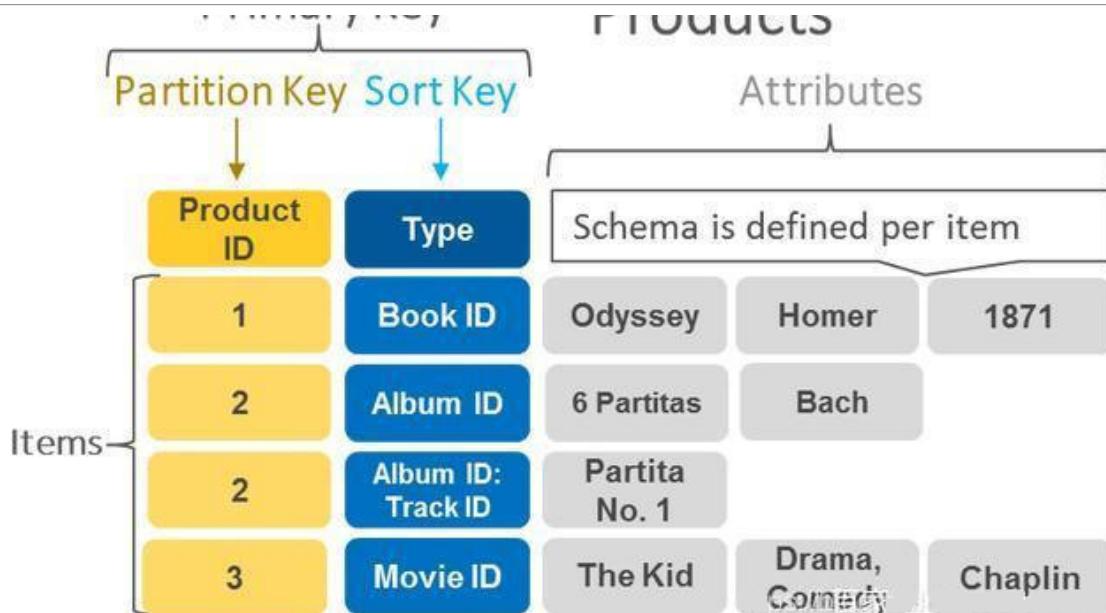
4.2.2 有哪些非关系型数据库

相比于 SQL，NoSQL 泛指非关系型数据库，包括了榜单上的键值型数据库、文档型数据库、搜索引擎和列存储等，除此以外还包括图形数据库。也只有用 NoSQL 一词才能将这些技术囊括进来。

键值型数据库

键值型数据库通过 Key-Value 键值的方式来存储数据，其中 Key 和 Value 可以是简单的对象，也可以是复杂的对象。Key 作为唯一的标识符，优点是查找速度快，在这方面明显优于关系型数据库，缺点是无法像关系型数据库一样使用条件过滤（比如 WHERE），如果你不知道去哪里找数据，就要遍历所有的键，这就会消耗大量的计算。

键值型数据库典型的使用场景是作为 **内存缓存**。**Redis** 是最流行的键值型数据库。



文档型数据库

此类数据库可存放并获取文档，可以是XML、JSON等格式。在数据库中文档作为处理信息的基本单位，一个文档就相当于一条记录。文档数据库所存放的文档，就相当于键值数据库所存放的“值”。MongoDB是最流行的文档型数据库。此外，还有CouchDB等。

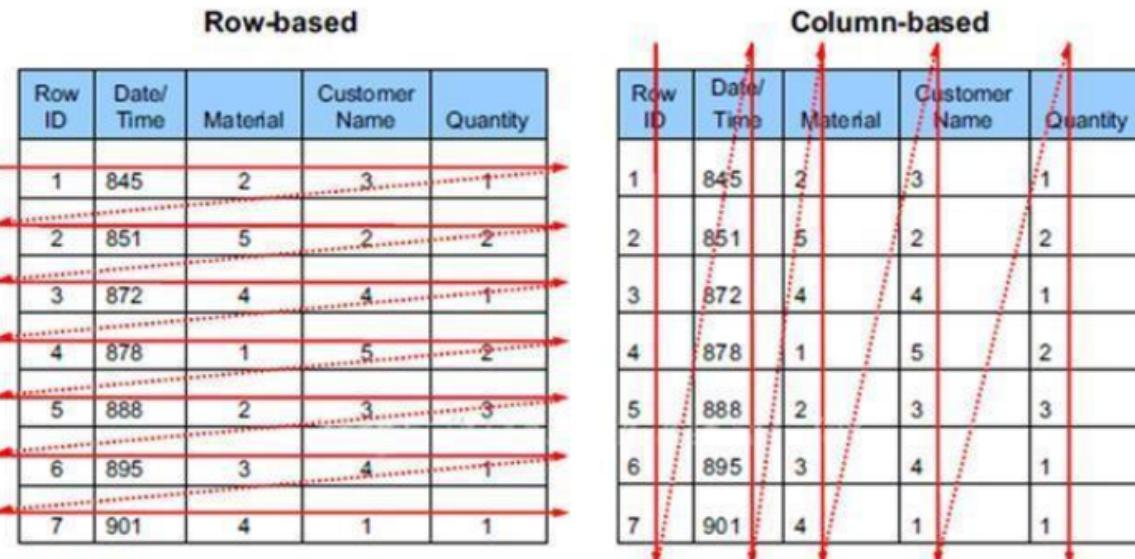
搜索引擎数据库

虽然关系型数据库采用了索引提升检索效率，但是针对全文索引效率却较低。搜索引擎数据库是应用在搜索引擎领域的数据存储形式，由于搜索引擎会爬取大量的数据，并以特定的格式进行存储，这样在检索的时候才能保证性能最优。核心原理是“倒排索引”。

典型产品：Solr、Elasticsearch、Splunk等。

列式数据库

列式数据库是相对于行式存储的数据库，Oracle、MySQL、SQL Server等数据库都是采用的行式存储（Row-based），而列式数据库是将数据按照列存储到数据库中，这样做的好处是可以大量降低系统的I/O，适合于分布式文件系统，不足在于功能相对有限。典型产品：HBase等。



Row-based

Row ID	Date/Time	Material	Customer Name	Quantity
1	845	2	3	1
2	851	5	2	2
3	872	4	4	1
4	878	1	5	2
5	888	2	3	3
6	895	3	4	1
7	901	4	1	1

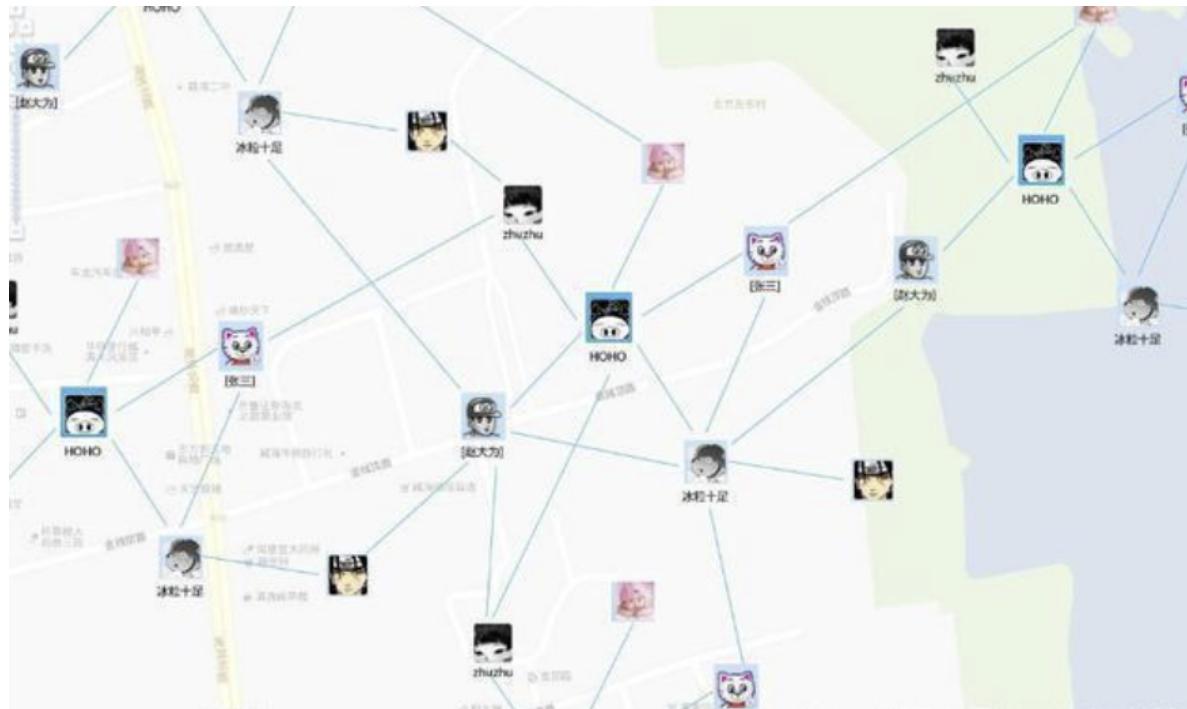
Column-based

Row ID	Date/Time	Material	Customer Name	Quantity
1	845	2	3	1
2	851	5	2	2
3	872	4	4	1
4	878	1	5	2
5	888	2	3	3
6	895	3	4	1
7	901	4	1	1

图形数据库

系问题。

图形数据库顾名思义，就是一种存储图形关系的数据库。它利用了图这种数据结构存储了实体（对象）之间的关系。关系型数据用于存储明确关系的数据，但对于复杂关系的数据存储却有些力不从心。如社交网络中人物之间的关系，如果用关系型数据库则非常复杂，用图形数据库将非常简单。典型产品：Neo4J、InfoGrid等。



4.2.3 NoSQL的演变

由于 SQL 一直称霸 DBMS，因此许多人在思考是否有一种数据库技术能远离 SQL，于是 NoSQL 诞生了，但是随着发展却发现越来越离不开 SQL。到目前为止 NoSQL 阵营中的 DBMS 都会有实现类似 SQL 的功能。下面是“NoSQL”这个名词在不同时期的诠释，从这些释义的变化中可以看出 NoSQL 功能的演变：

1970: NoSQL = We have no SQL

1980: NoSQL = Know SQL

2000: NoSQL = No SQL!

2005: NoSQL = Not only SQL

2013: NoSQL = No, SQL!

NoSQL 对 SQL 做出了很多

库功能，非关系型数据库的功能就足够使用了。这种情况下，使用 性能更高、成本更低 的非关系型数据库当然是更明智的选择。比如：日志收集、排行榜、定时器等。

4.5 丁组

4 个是关系型数据库，而排名前 20 的 DBMS 中也有 12 个是关系型数据库。所以说，掌握 SQL 是非常有必要的。整套课程将围绕 SQL 展开。

5. 关系型数据库设计规则

- 一个数据库中可以有多个表，每个表都有一个名字，用来标识自己。表名具有唯一性。
- 表具有一些特性，这些特性定义了数据在表中如何存储，类似Java和Python中“类”的设计。

5.1 表、记录、字段

- E-R (entity-relationship, 实体-联系) 模型中有三个主要概念是：实体集、属性、联系集。
- 一个实体集 (class) 对应于数据库中的一个表 (table)，一个实体 (instance) 则对应于数据库表中的一行 (row)，也称为一条记录 (record)。一个属性 (attribute) 对应于数据库表中的一列 (column)，也称为一个字段 (field)。

列

字段	学号	姓名	年龄	性别	专业	属性
记录	161228001	张三	20	男	JavaEE	
行	161228002	李四	19	女	H5	
161228003	王五	21	男	Android		
161228004	赵六	20	女	PHP		
161228005	钱七	23	男	JavaEE		
161228006	孙八	22	男	Android		

实体、对象

ORM思想 (Object Relational Mapping)体现:

数据库中的一个表 <----> Java或Python中的一个类

表中的一条数据 <----> 类中的一个对象（或实体）

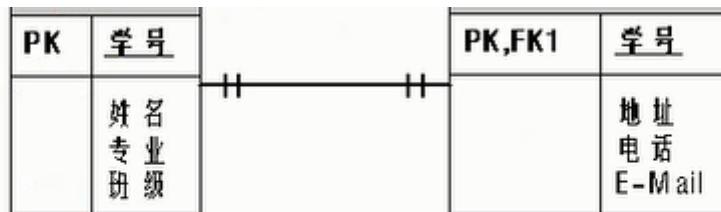
表中的一个列 <----> 类中的一个字段、属性 (field)

5.2 表的关联关系

- 表与表之间的数据记录有关系 (relationship)。现实世界中的各种实体以及实体之间的各种联系均用关系模型来表示。
- 四种：一对关联、一对多关联、多对多关联、自我引用

5.2.1 一对关联 (one-to-one)

- 在实际的开发中应用不多，因为一对一可以创建成一张表。
- 举例：设计 学生表：学号、姓名、手机号码、班级、系别、身份证号码、家庭住址、籍贯、紧急联系人、...
 - 拆为两个表：两个表的记录是一一对应关系。
 - 基础信息表（常用信息）：学号、姓名、手机号码、班级、系别
 - 档案信息表（不常用信息）：学号、身份证号码、家庭住址、籍贯、紧急联系人、...
- 两种建表原则：
 - 外键唯一：主表的主键和从表的外键（唯一），形成主外键关系，外键唯一。
 - 外键是主键：主表的主键和从表的主键，形成主外键关系。

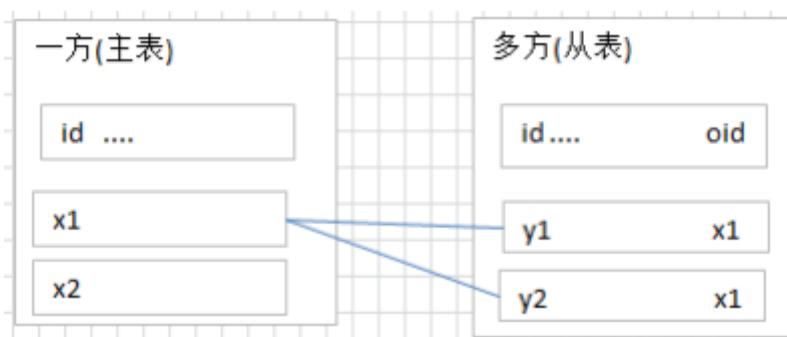


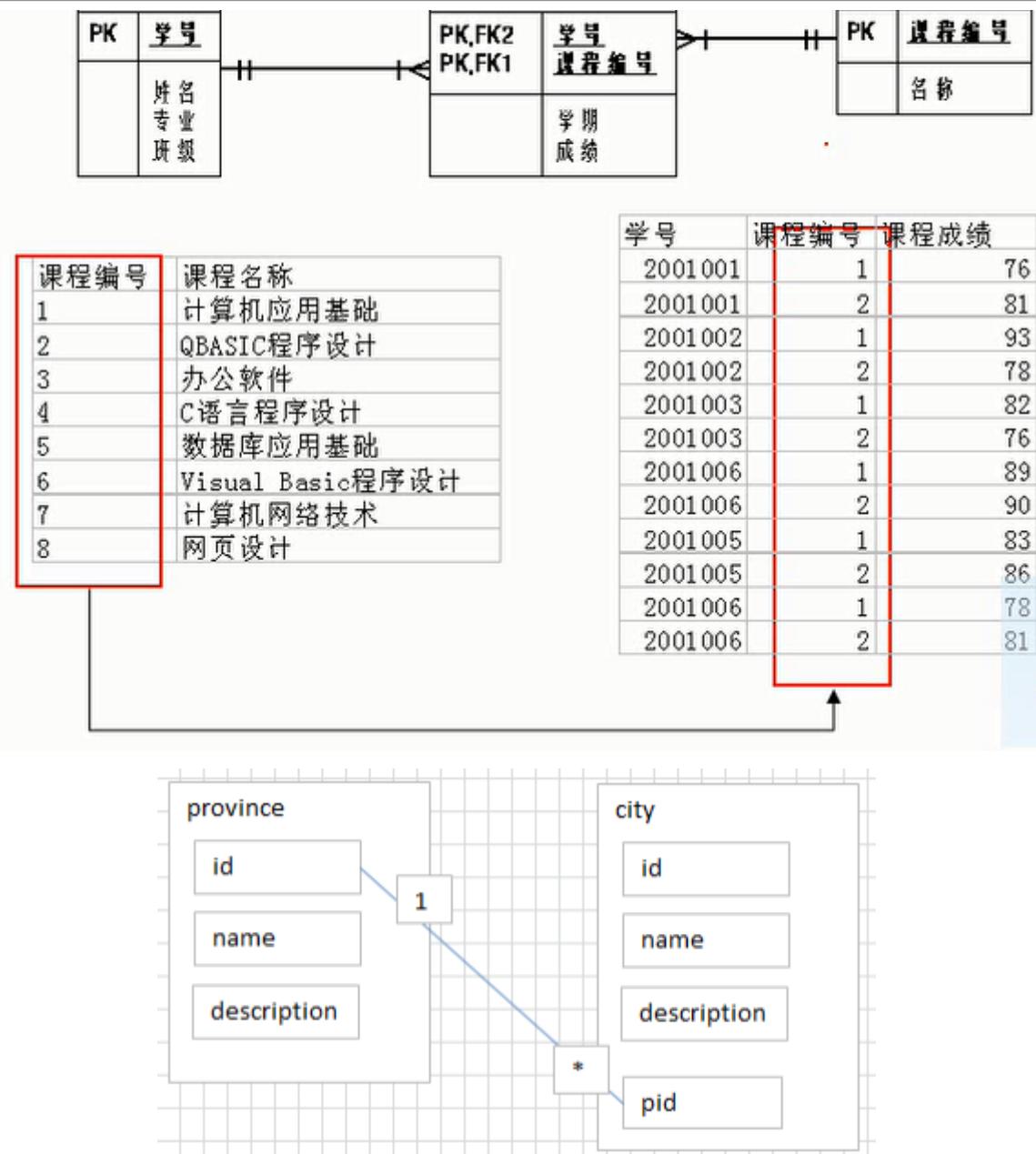
学生	姓名	专业	班级
2001001	吴小亮	计算机应用基础	101
2001002	刘京生	计算机应用基础	101
2001003	李向明	计算机应用基础	101
2001006	张哲夫	计算机应用基础	102
2001005	黄威	计算机应用基础	102
2001006	高大山	计算机应用基础	102

学号	地址	电话	E-Mail
2001001	郑州市黄河路	450008	Tom@163.com
2001002	北京市复兴门外	100859	Vinv@163.com
2001003	西安市太自路	710069	Toy@163.com
2001006	北京市复兴门外	100859	Sina@163.com
2001005	西安市太白路	710069	SS@163.com
2001006	郑州市黄河路	450008	dddSToy@163.com

5.2.2 一对多关系 (one-to-many)

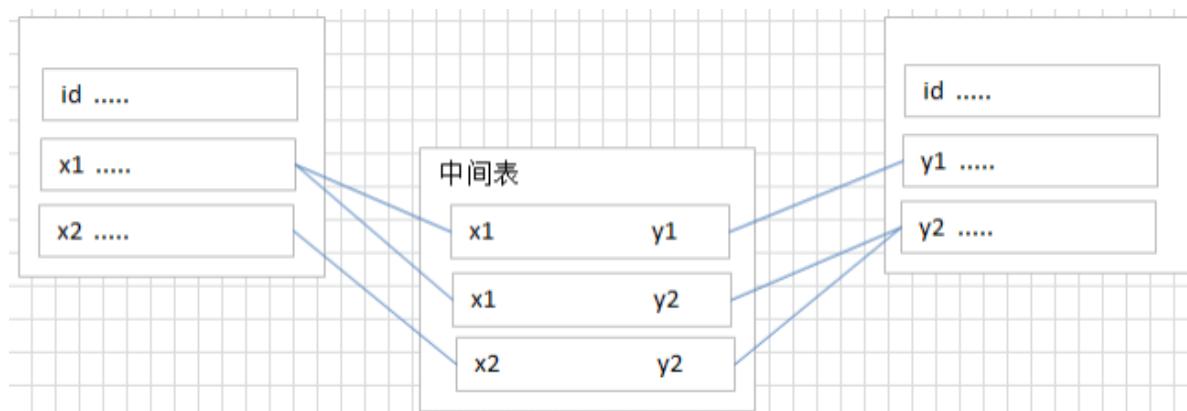
- 常见实例场景：客户表和订单表，分类表和商品表，部门表和员工表。
- 举例：
 - 员工表：编号、姓名、...、所属部门
 - 部门表：编号、名称、简介
- 一对多建表原则：在从表(多方)创建一个字段，字段作为外键指向主表(一方)的主键





5.2.3 多对多 (many-to-many)

要表示多对多关系，必须创建第三个表，该表通常称为 **联接表**，它将多对多关系划分为两个一对多关系。将这两个表的主键都插入到第三个表中。



• 举例1：学生-课程

- 学生信息表：一行代表一个学生的信息（学号、姓名、手机号码、班级、系别...）

- 远程信息表：一个子表可以远多于1，1个表可以很多个子表连接

学号	课程编号
1	1001
2	1001
1	1002

• 举例2：产品-订单

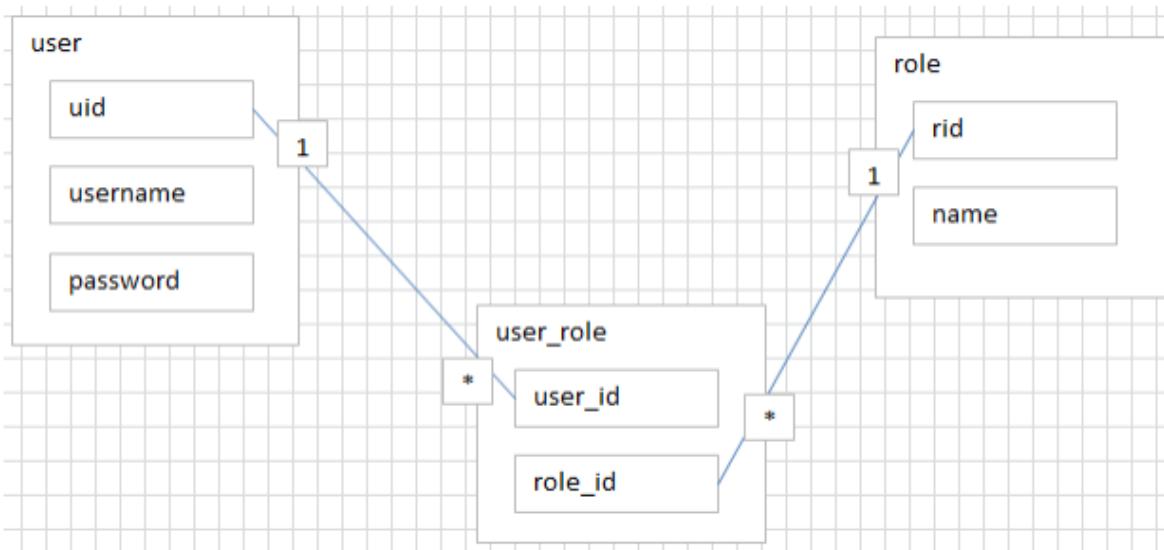
“订单”表和“产品”表有一种多对多的关系，这种关系是通过与“订单明细”表建立两个一对多关系来定义的。一个订单可以有多个产品，每个产品可以出现在多个订单中。

- **产品表**：“产品”表中的每条记录表示一个产品。
- **订单表**：“订单”表中的每条记录表示一个订单。
- **订单明细表**：每个产品可以与“订单”表中的多条记录对应，即出现在多个订单中。一个订单可以与“产品”表中的多条记录对应，即包含多个产品。



• 举例3：用户-角色

- 多对多关系建表原则：需要创建第三张表，中间表中至少两个字段，这两个字段分别作为外键指向各自一方的主键。



5.3.4 自我引用(Self reference)



员工编号	姓名	部门编号	主管编号
101	吴小亮	30	NULL
103	刘京生	30	101
104	李向明	30	103
105	张哲夫	30	103
210	黄威	45	101
231	高大山	45	210

第02章_MySQL环境搭建

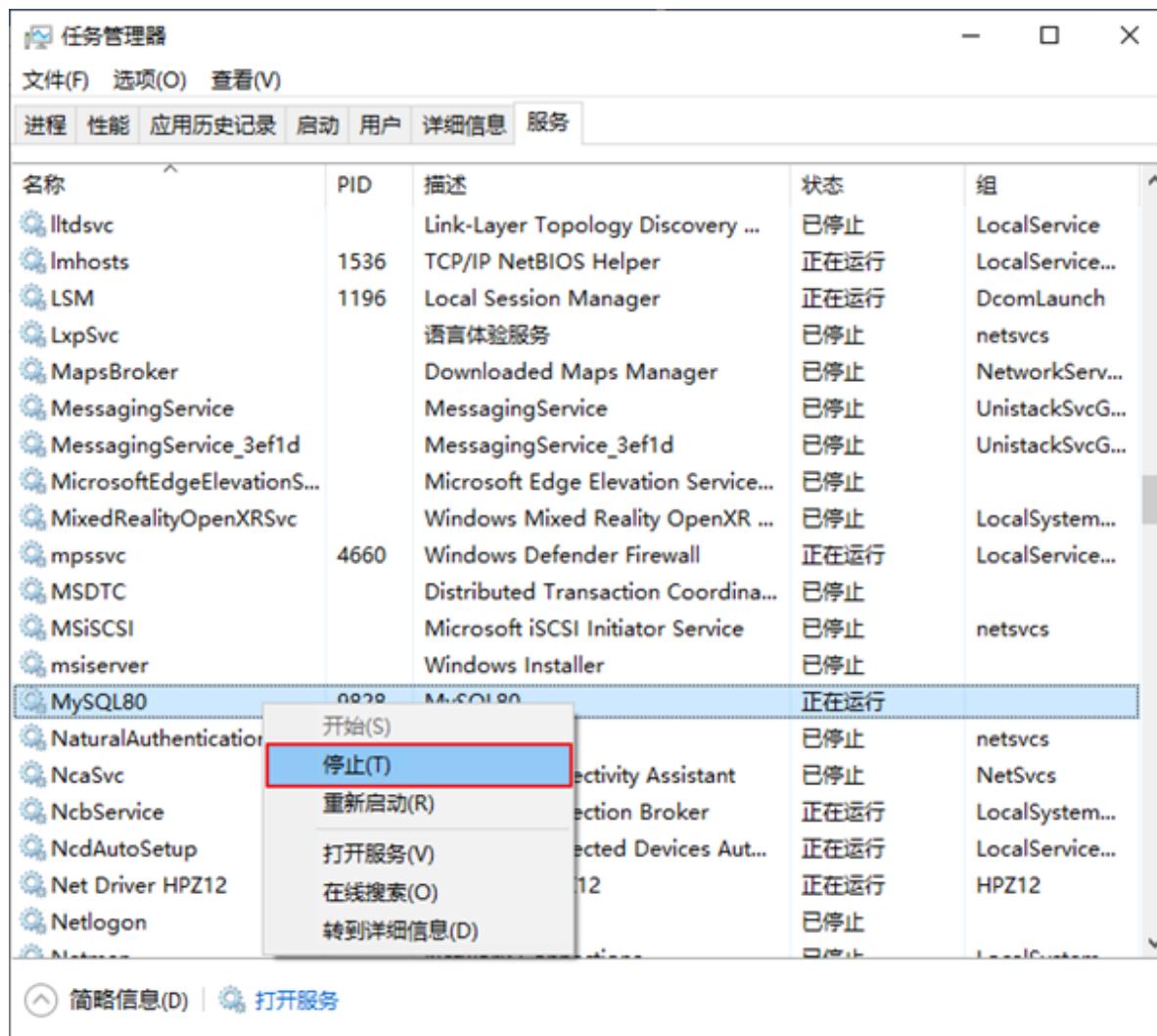
讲师：尚硅谷 宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

1. MySQL的卸载

步骤1：停止MySQL服务

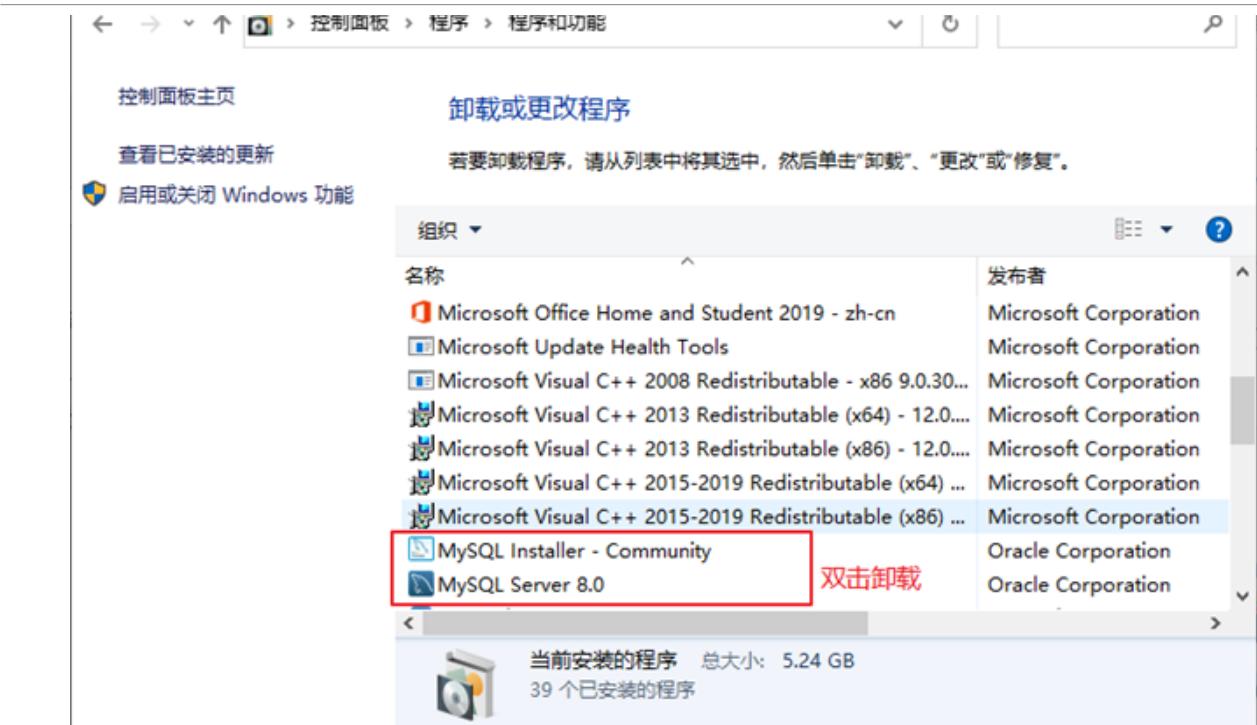
在卸载之前，先停止MySQL8.0的服务。按键盘上的“Ctrl + Alt + Delete”组合键，打开“任务管理器”对话框，可以在“服务”列表找到“MySQL8.0”的服务，如果现在“正在运行”状态，可以右键单击服务，选择“停止”选项停止MySQL8.0的服务，如图所示。



步骤2：软件的卸载

方式1：通过控制面板方式

卸载MySQL8.0的程序可以和其他桌面应用程序一样直接在“控制面板”选择“卸载程序”，并在程序列表中找到MySQL8.0服务器程序，直接双击卸载即可，如图所示。这种方式删除，数据目录下的数据不会跟着删除。



方式2：通过360或电脑管家等软件卸载

略

方式3：通过安装包提供的卸载功能卸载

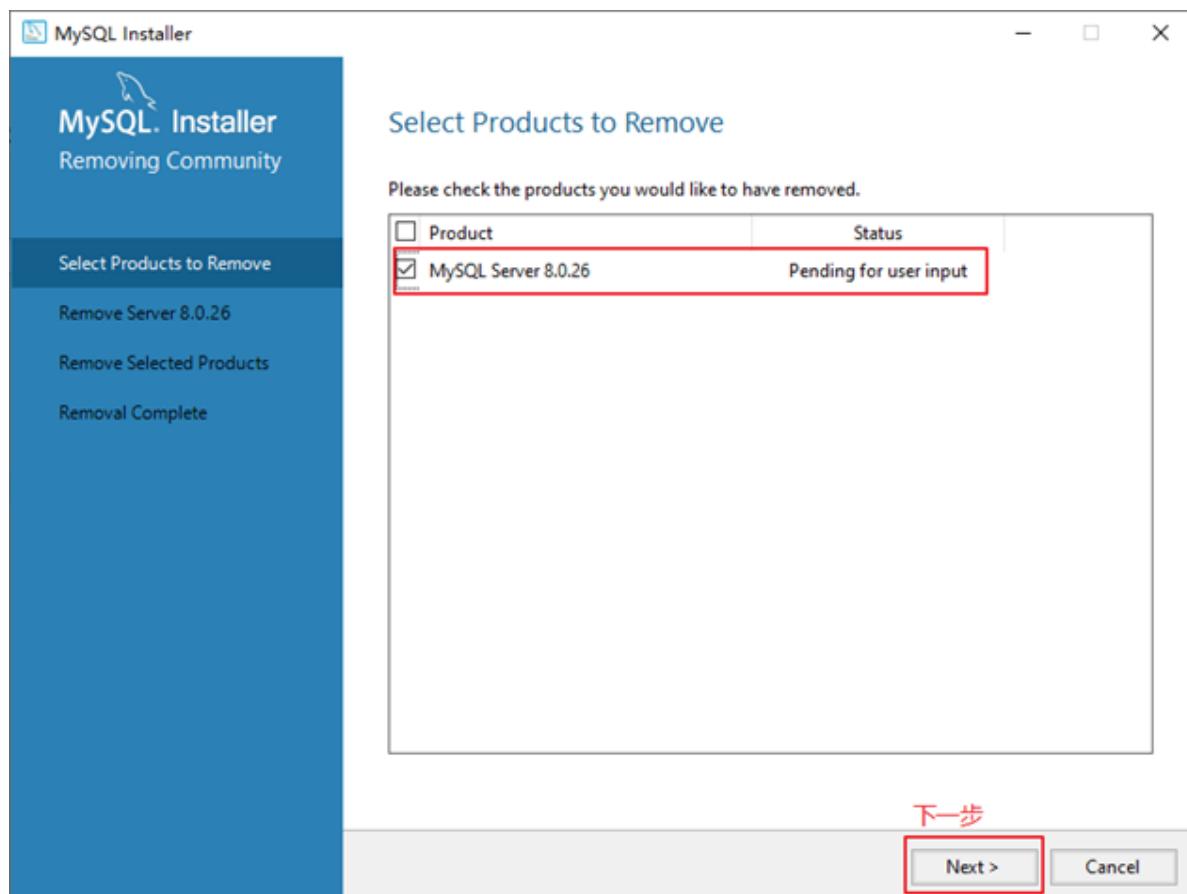
你也可以通过安装向导程序进行MySQL8.0服务器程序的卸载。

① 再次双击下载的mysql-installer-community-8.0.26.0.msi文件，打开安装向导。安装向导会自动检测已安装的MySQL服务器程序。

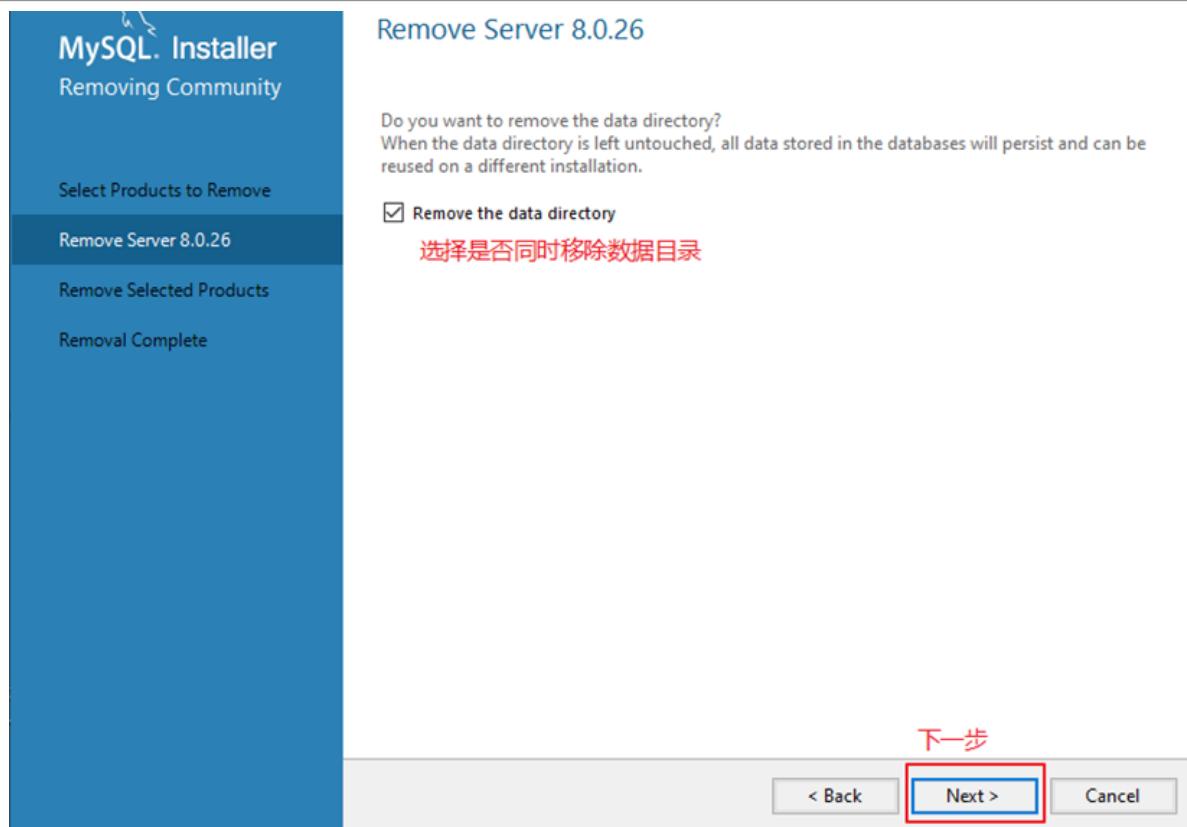
② 选择要卸载的MySQL服务器程序，单击“Remove”（移除），即可进行卸载。



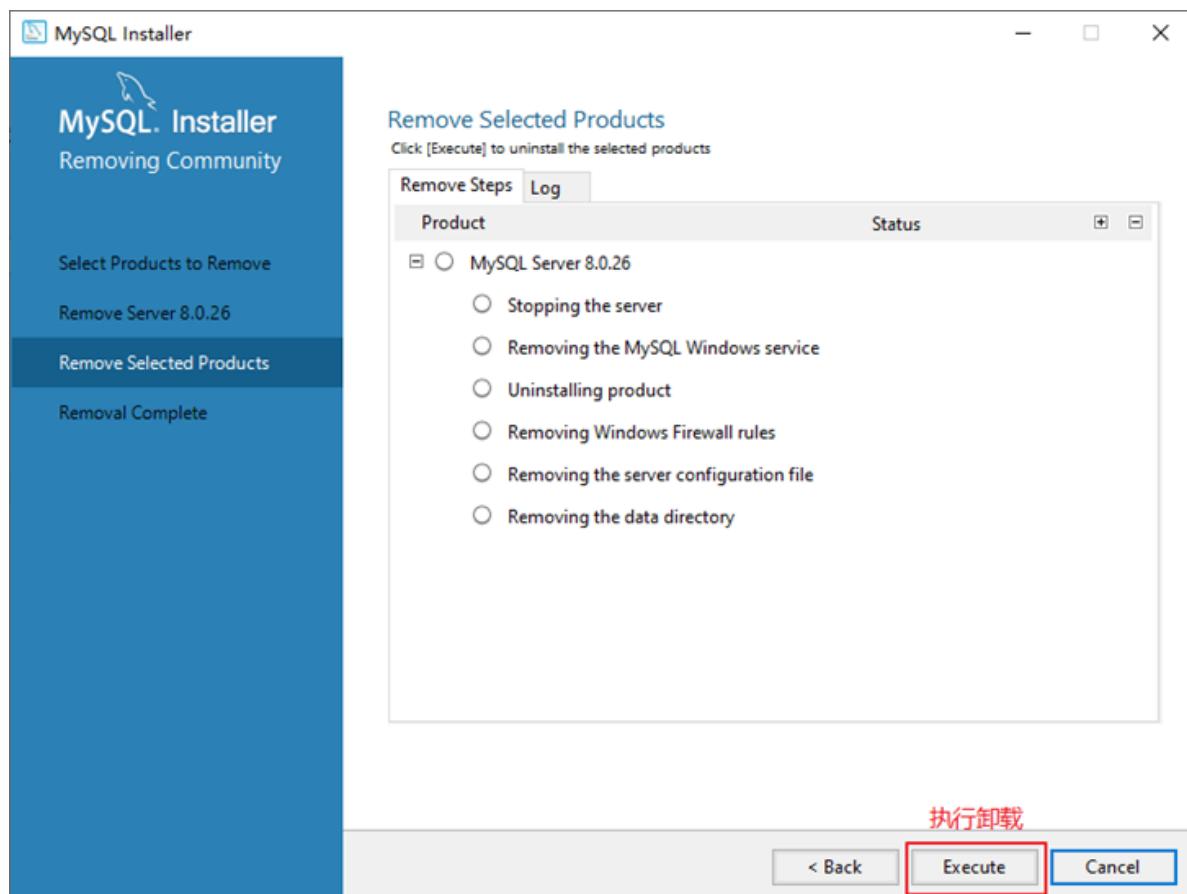
③ 单击“Next”（下一步）按钮，确认卸载。



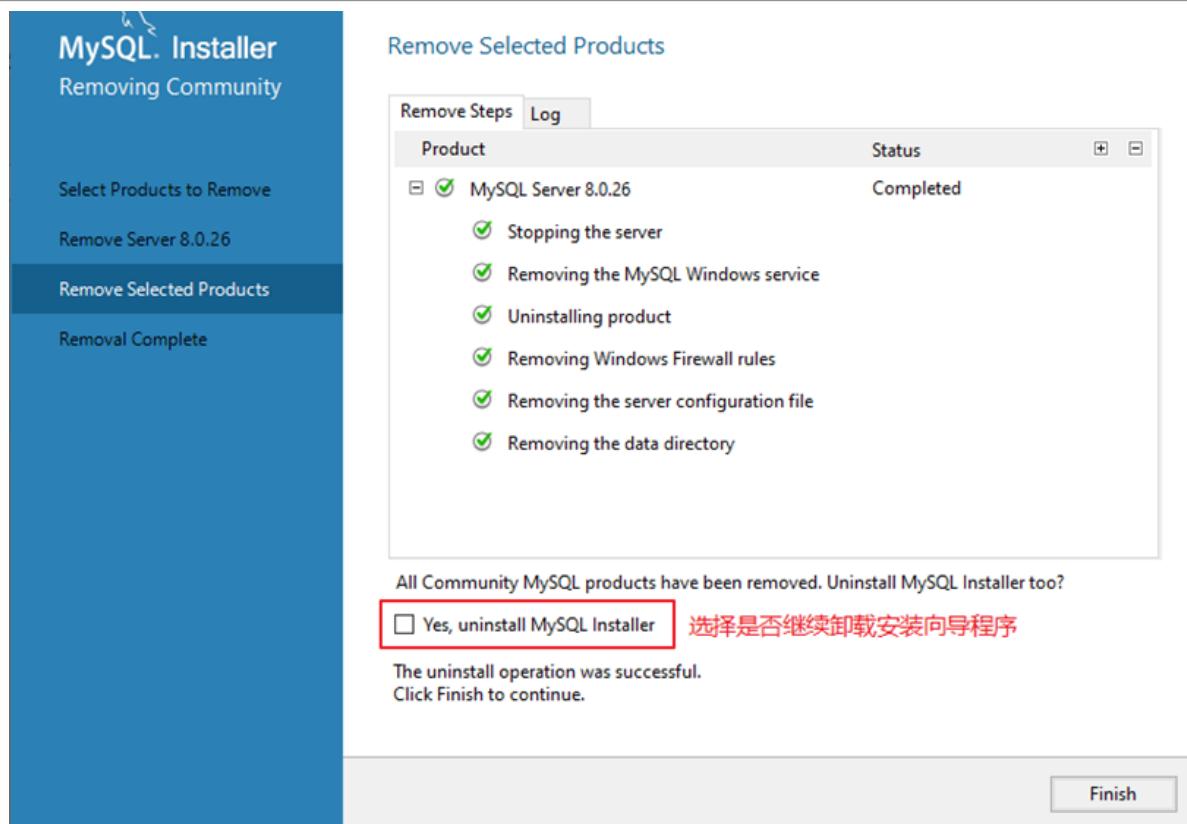
④ 弹出是否同时移除数据目录选择窗口。如果想要同时删除MySQL服务器中的数据，则勾选“Remove the data directory”，如图所示。



⑤ 执行卸载。单击“Execute”（执行）按钮进行卸载。



⑥ 完成卸载。单击“Finish”（完成）按钮即可。如果想要同时卸载MySQL8.0的安装向导程序，勾选“Yes, Uninstall MySQL Installer”即可，如图所示。



步骤3：残余文件的清理

如果再次安装不成功，可以卸载后对残余文件进行清理后再安装。

- (1) 服务目录：mysql服务的安装目录
- (2) 数据目录：默认在C:\ProgramData\MySQL

如果自己单独指定过数据目录，就找到自己的数据目录进行删除即可。

注意：请在卸载前做好数据备份

在操作完以后，需要重启计算机，然后进行安装即可。**如果仍然安装失败，需要继续操作如下步骤4。**

步骤4：清理注册表（选做）

如果前几步做了，再次安装还是失败，那么可以清理注册表。

如何打开注册表编辑器：在系统的搜索框中输入 **regedit**

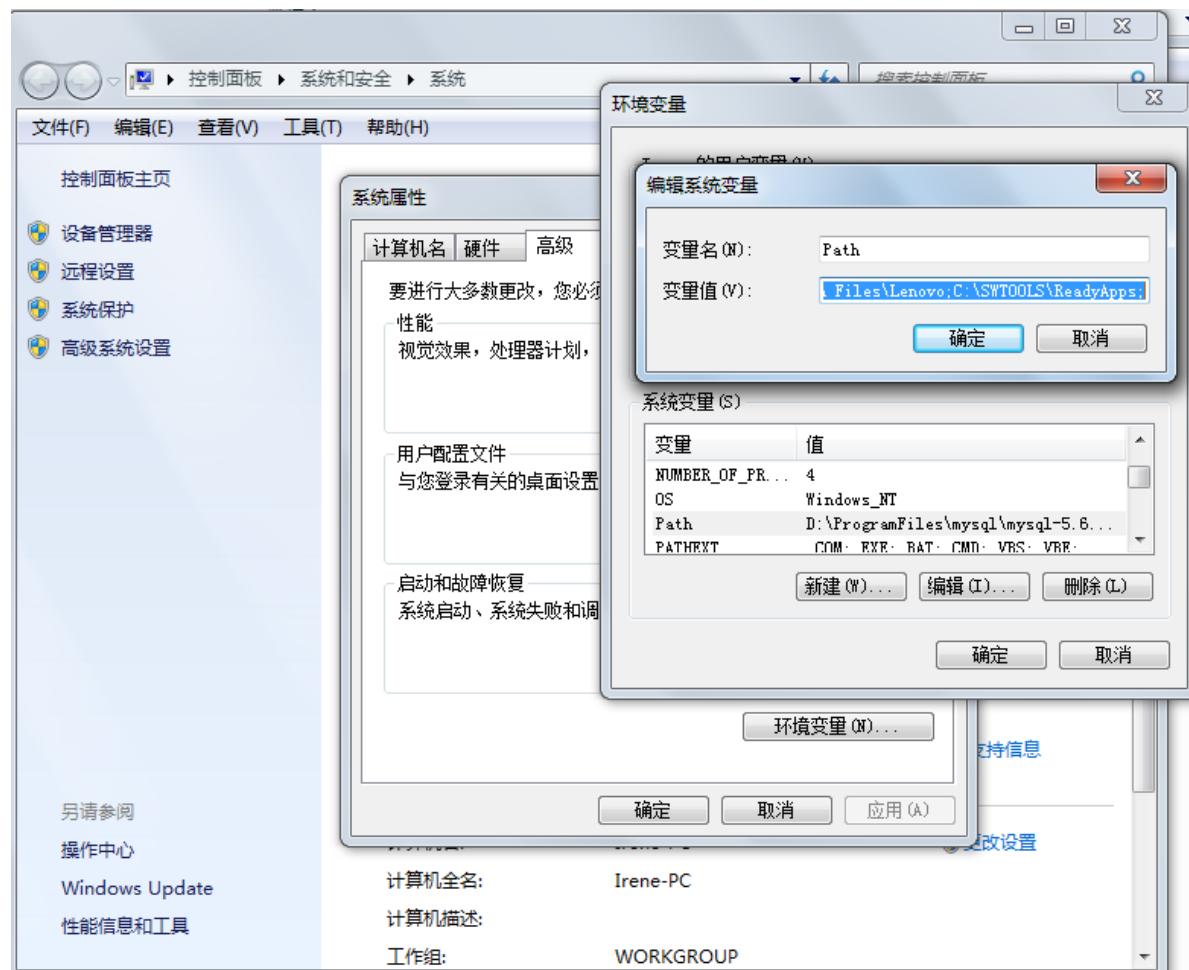
HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\MySQL服务 目录删除
HKEY_LOCAL_MACHINE\SYSTEM\ControlSet002\Services\Eventlog\Application\MySQL服务 目录删除
HKEY_LOCAL_MACHINE\SYSTEM\ControlSet002\Services\MySQL服务 目录删除
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Eventlog\Application\MySQL服务目录删除
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\MySQL服务删除

注册表中的ControlSet001,ControlSet002,不一定是001和002,可能是ControlSet005、006之类

步骤5：删除环境变量配置

找到path环境变量，将其中关于mysql的环境变量删除，**切记不要全部删除**。

例如：删除 D:\develop_tools\mysql\MySQLServer8.0.26\bin; 这个部分



2. MySQL的下载、安装、配置

- **MySQL Community Server 社区版本**，开源免费，自由下载，但不提供官方技术支持，适用于大多数普通用户。
- **MySQL Enterprise Edition 企业版本**，需付费，不能在线下载，可以试用30天。提供了更多的功能和更完备的技术支持，更适合于对数据库的功能和可靠性要求较高的企业客户。
- **MySQL Cluster 集群版**，开源免费。用于架设集群服务器，可将几个MySQL Server封装成一个Server。需要在社区版或企业版的基础上使用。
- **MySQL Cluster CGE 高级集群版**，需付费。

- 目前最新版本为 **8.0.27**，发布时间 **2021年10月**。此前，8.0.0 在 2016.9.12日就发布了。
- 本课程中使用 **8.0.26** 版本。

此外，官方还提供了 **MySQL Workbench** (GUITOOL) 一款专为MySQL设计的 图形界面管理工具 。MySQLWorkbench又分为两个版本，分别是 **社区版** (MySQL Workbench OSS) 、 **商用版** (MySQL WorkbenchSE) 。

2.2 软件的下载

1. 下载地址

官网：<https://www.mysql.com>

2. 打开官网，点击 DOWNLOADS

然后，点击 **MySQL Community (GPL) Downloads**

The screenshot shows the MySQL Downloads page. At the top, there's a navigation bar with tabs like 'MySQL Downloads' and 'Downloads'. Below the navigation, there's a large banner for 'MySQL Database Service with HeatWave for Real-time Analytics'. The banner highlights 'Faster Performance' (with points: 400x MySQL query acceleration, 1100x faster than Amazon Aurora, 2.7x faster than Amazon Redshift) and 'Lower Total Cost of Ownership' (with points: 1/3 the cost of Amazon RDS, 1/3 the cost of Amazon Redshift, easy migration from Amazon RDS). Below the banner, there are sections for 'Free Webinars', 'MySQL Enterprise Edition', 'MySQL Cluster CGE', and 'Contact Sales'. The 'MySQL Community (GPL) Downloads' link is highlighted with a red box.

3. 点击 MySQL Community Server

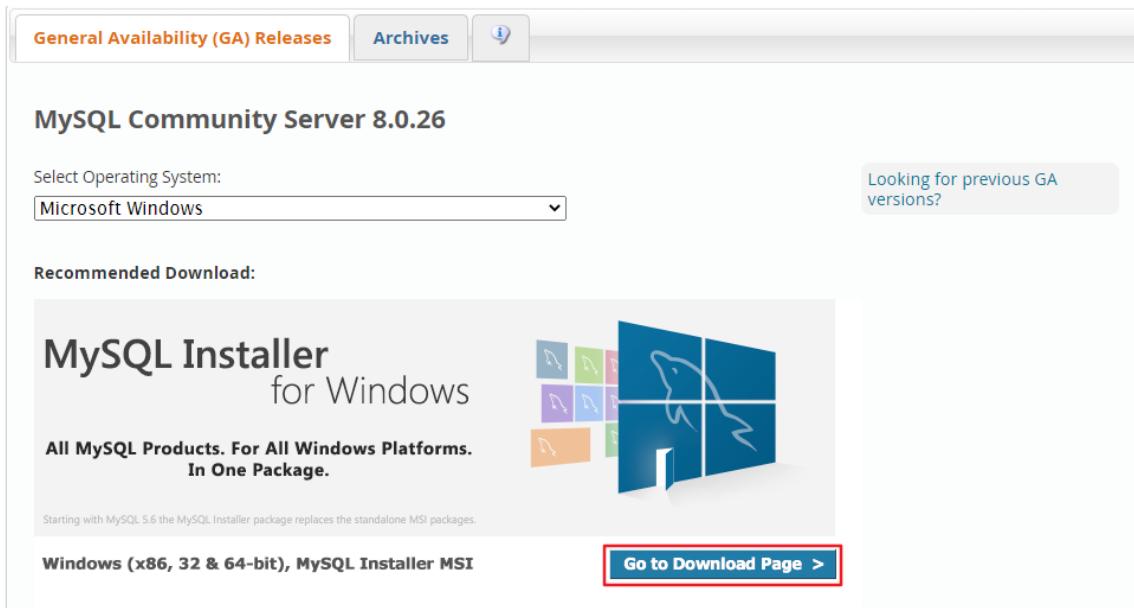
④ MySQL Community Downloads

- MySQL Yum Repository
- MySQL APT Repository
- MySQL SUSE Repository
- MySQL Community Server
- MySQL Cluster
- MySQL Router
- MySQL Shell
- MySQL Workbench
- MySQL Installer for Windows
- MySQL for Visual Studio
- C API (libmysqlclient)
- Connector/C++
- Connector/J
- Connector/.NET
- Connector/Node.js
- Connector/ODBC
- Connector/Python
- MySQL Native Driver for PHP
- MySQL Benchmark Tool
- Time zone description tables
- Download Archives

4. 在General Availability(GA) Releases中选择适合的版本

Windows平台下提供两种安装文件：MySQL二进制分发版（.msi安装文件）和免安装版（.zip压缩文件）。一般来讲，应当使用二进制分发版，因为该版本提供了图形化的安装向导过程，比其他的分发版使用起来要简单，不再需要其他工具启动就可以运行MySQL。

- 这里在Windows系统下推荐下载 **MSI安装程序**；点击 **Go to Download Page** 进行下载即可



The screenshot shows the MySQL Community Server 8.0.26 download page. At the top, there are tabs for "General Availability (GA) Releases" (which is selected and highlighted in orange), "Archives", and a help icon. Below the tabs, it says "Select Operating System:" with a dropdown menu set to "Microsoft Windows". To the right, there's a link "Looking for previous GA versions?". The main section is titled "MySQL Community Server 8.0.26" and features a large image of the MySQL logo. Below the image, it says "Recommended Download:" and shows the "MySQL Installer for Windows" package. The package image includes the text "All MySQL Products. For All Windows Platforms. In One Package.". At the bottom of the package image, it says "Starting with MySQL 5.6 the MySQL Installer package replaces the standalone MSI packages." and "Windows (x86, 32 & 64-bit), MySQL Installer MSI". A blue button at the bottom right is labeled "Go to Download Page >" with a red border around it.

MySQL Installer 8.0.26

Select Operating System: Microsoft Windows

Looking for previous GA versions?

Windows (x86, 32-bit), MSI Installer (mysql-installer-web-community-8.0.26.0.msi)	8.0.26	2.4M	Download
Windows (x86, 32-bit), MSI Installer (mysql-installer-community-8.0.26.0.msi)	8.0.26	450.7M	Download

! We suggest that you use the MD5 checksums and GnuPG signatures to verify the integrity of the packages you download.

- Windows下的MySQL8.0安装有两种安装程序
 - `mysql-installer-web-community-8.0.26.0.msi` 下载程序大小: 2.4M; 安装时需要联网安装组件。
 - `mysql-installer-community-8.0.26.0.msi` 下载程序大小: 450.7M; 安装时离线安装即可。推荐。
- 如果安装MySQL5.7版本的话, 选择 **Archives**, 接着选择MySQL5.7的相应版本即可。这里下载最近期的MySQL5.7.34版本。

General Availability (GA) Releases **Archives** 

MySQL Installer 8.0.26

Select Operating System: Microsoft Windows

Looking for previous GA versions?

Windows (x86, 32-bit), MSI Installer (mysql-installer-web-community-8.0.26.0.msi)	8.0.26	2.4M	Download
Windows (x86, 32-bit), MSI Installer (mysql-installer-community-8.0.26.0.msi)	8.0.26	450.7M	Download

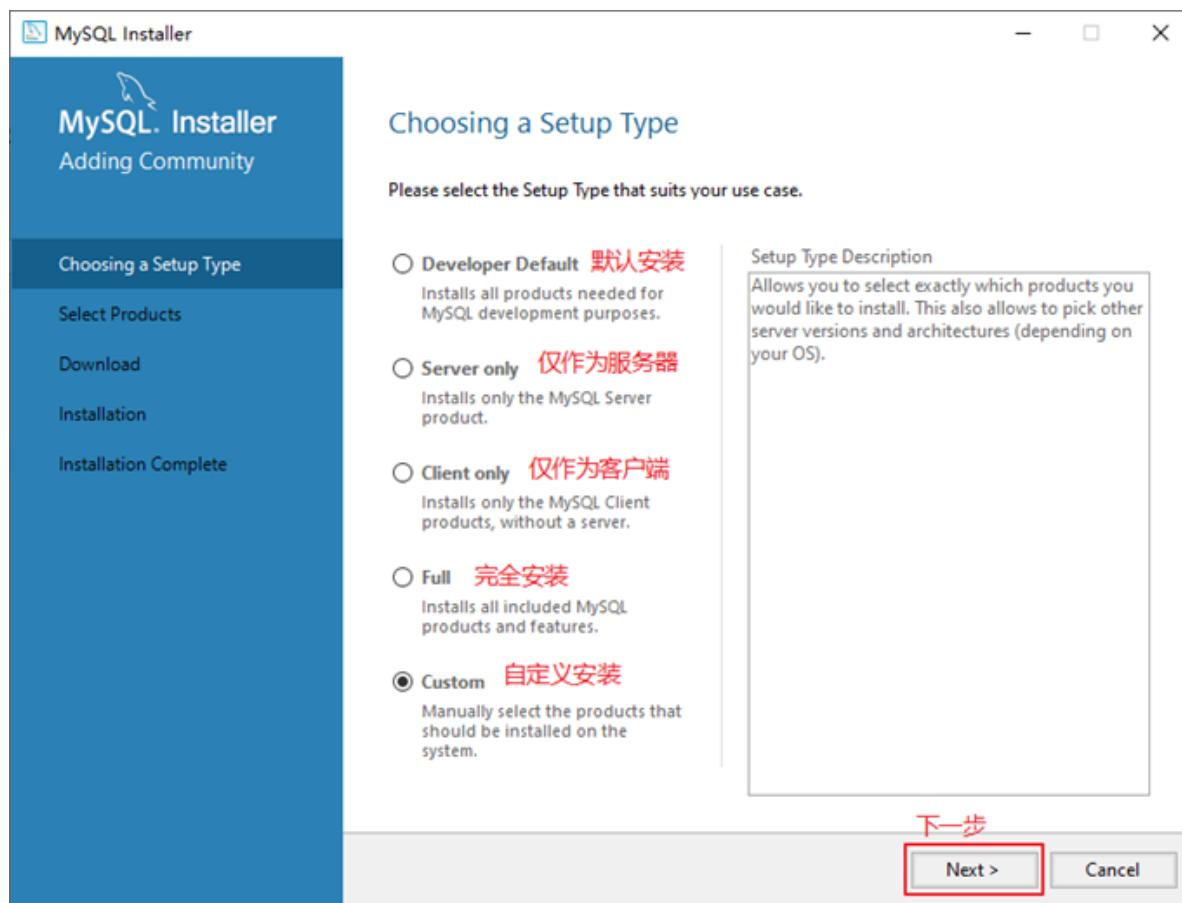
Windows (x86, 32-bit), MSI Installer (mysql-installer-web-community-5.7.34.0.msi)	Apr 6, 2021	2.4M	Download
Windows (x86, 32-bit), MSI Installer (mysql-installer-community-5.7.34.0.msi)	Apr 6, 2021	513.2M	Download

2.3 MySQL8.0 版本的安装

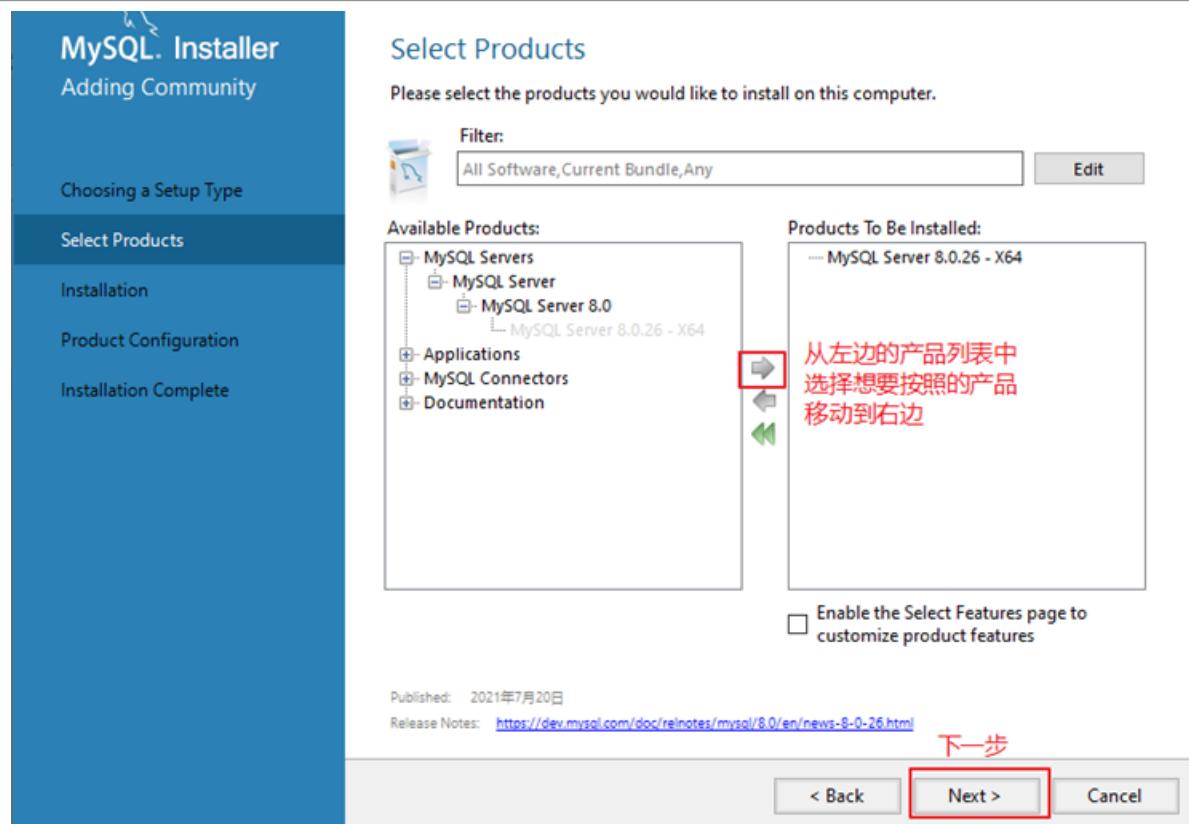
MySQL下载完成后，找到下载文件，双击进行安装，具体操作步骤如下。

步骤1：双击下载的mysql-installer-community-8.0.26.0.msi文件，打开安装向导。

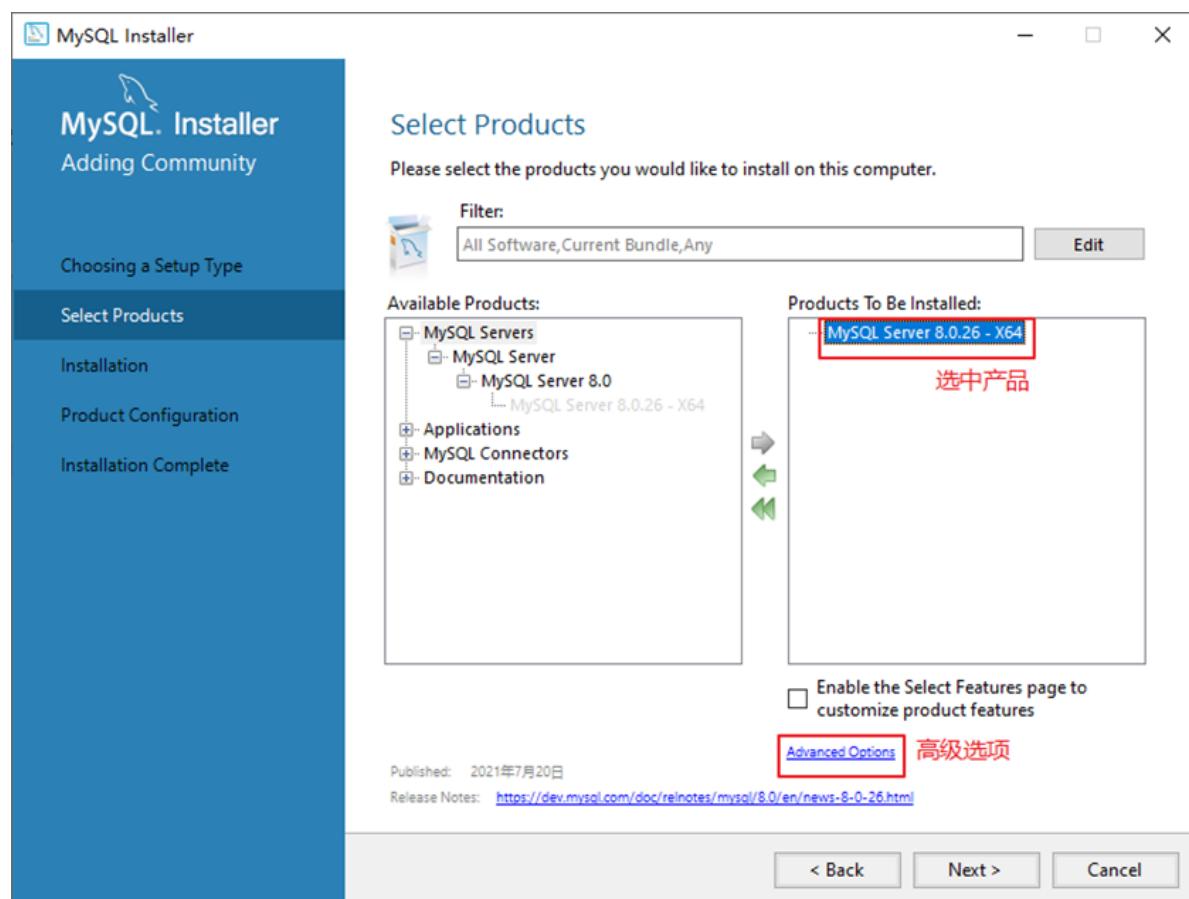
步骤2：打开“Choosing a Setup Type”（选择安装类型）窗口，在其中列出了5种安装类型，分别是Developer Default（默认安装类型）、Server only（仅作为服务器）、Client only（仅作为客户端）、Full（完全安装）、Custom（自定义安装）。这里选择“Custom（自定义安装）”类型按钮，单击“Next(下一步)”按钮。



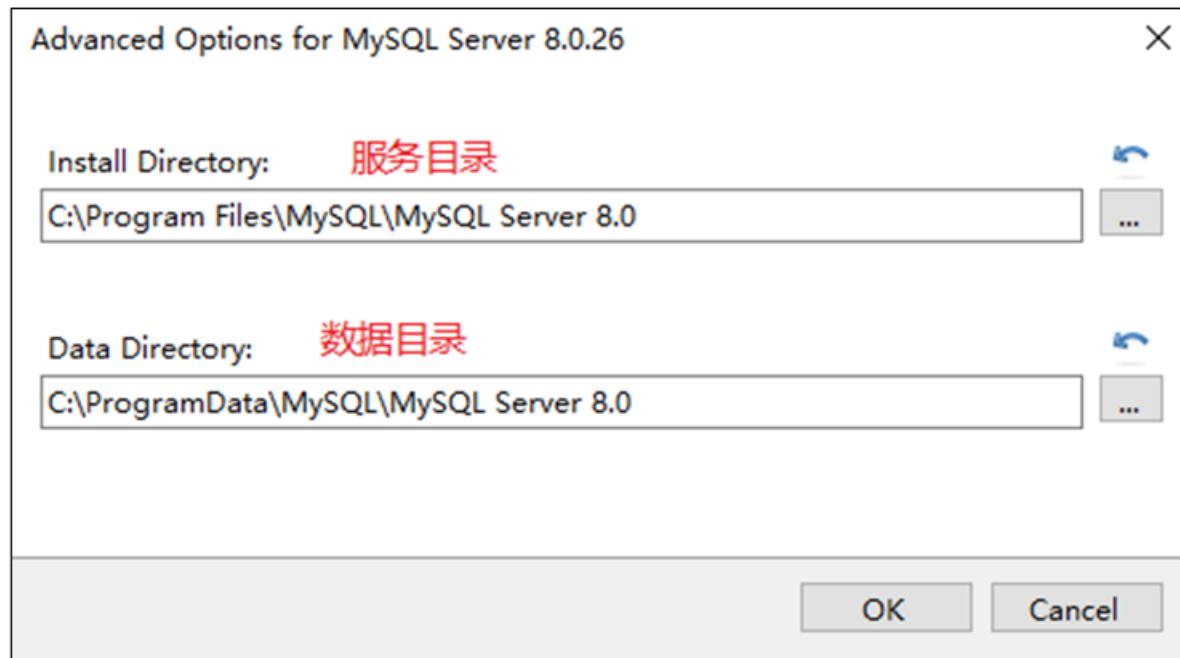
步骤3：打开“Select Products”（选择产品）窗口，可以定制需要安装的产品清单。例如，选择“MySQL Server 8.0.26-X64”后，单击“→”添加按钮，即可选择安装MySQL服务器，如图所示。采用通用的方法，可以添加其他你需要安装的产品。



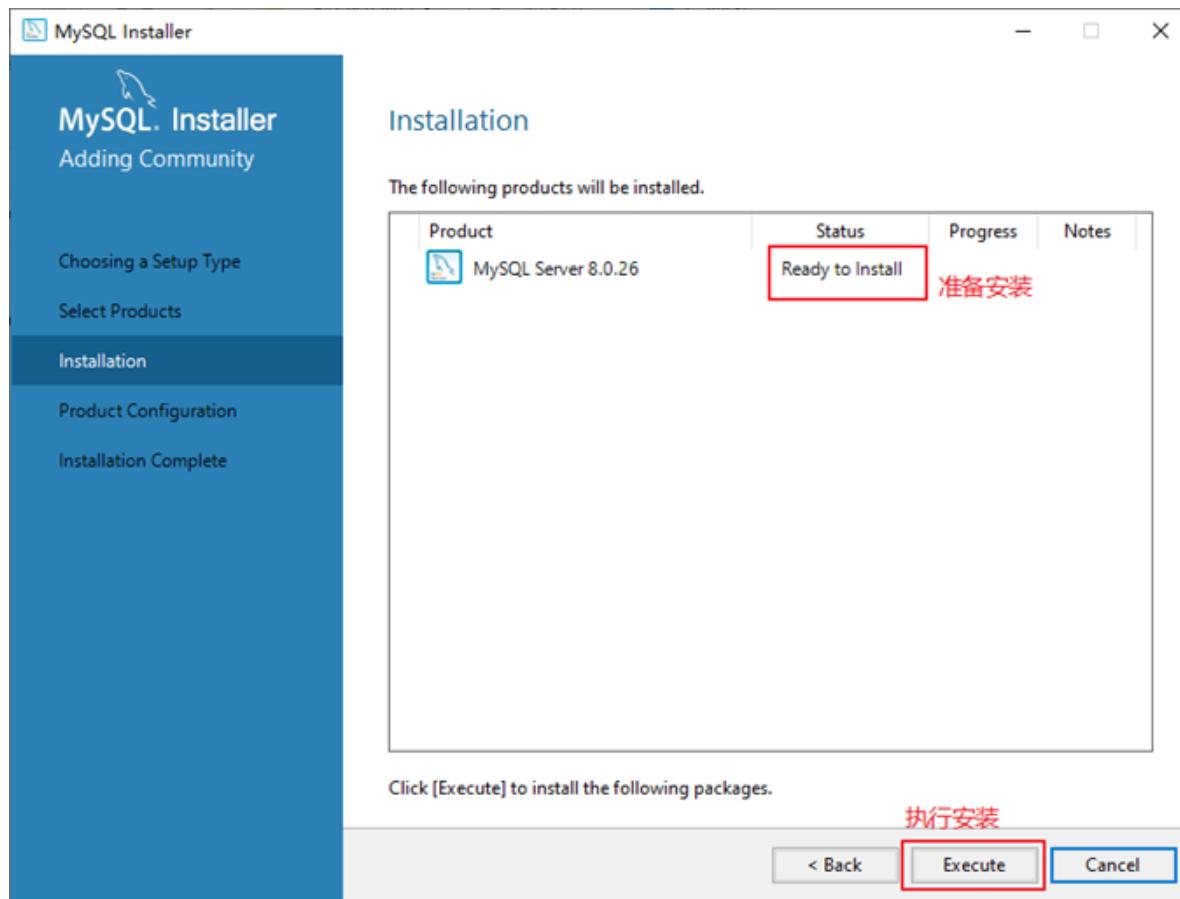
此时如果直接“Next”（下一步），则产品的安装路径是默认的。如果想要自定义安装目录，则可以选中对应的产品，然后在下面会出现“Advanced Options”（高级选项）的超链接。



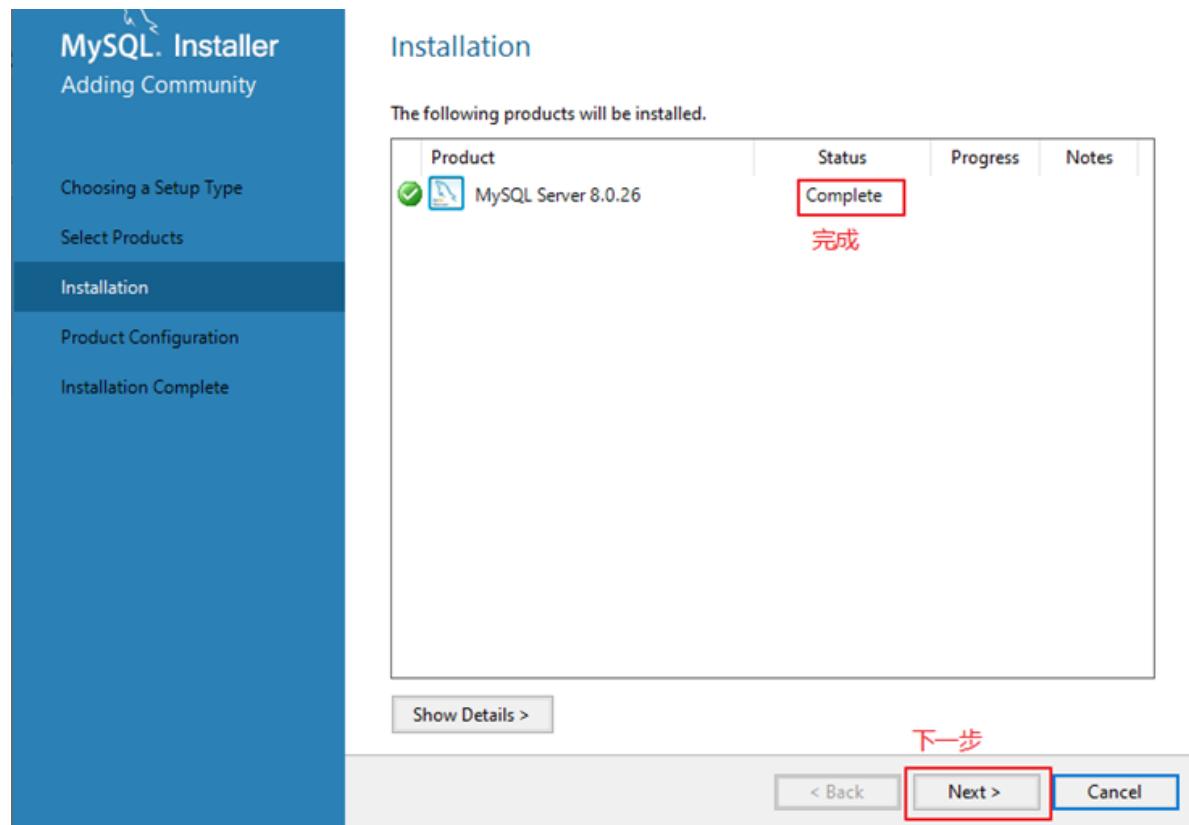
ProgramData目录（这是一个隐藏目录）。如果自定义安装目录，请避免“中文”目录。另外，建议服务目录和数据目录分开存放。



步骤4：在上一步选择好要安装的产品之后，单击“Next”（下一步）进入确认窗口，如图所示。单击“Execute”（执行）按钮开始安装。



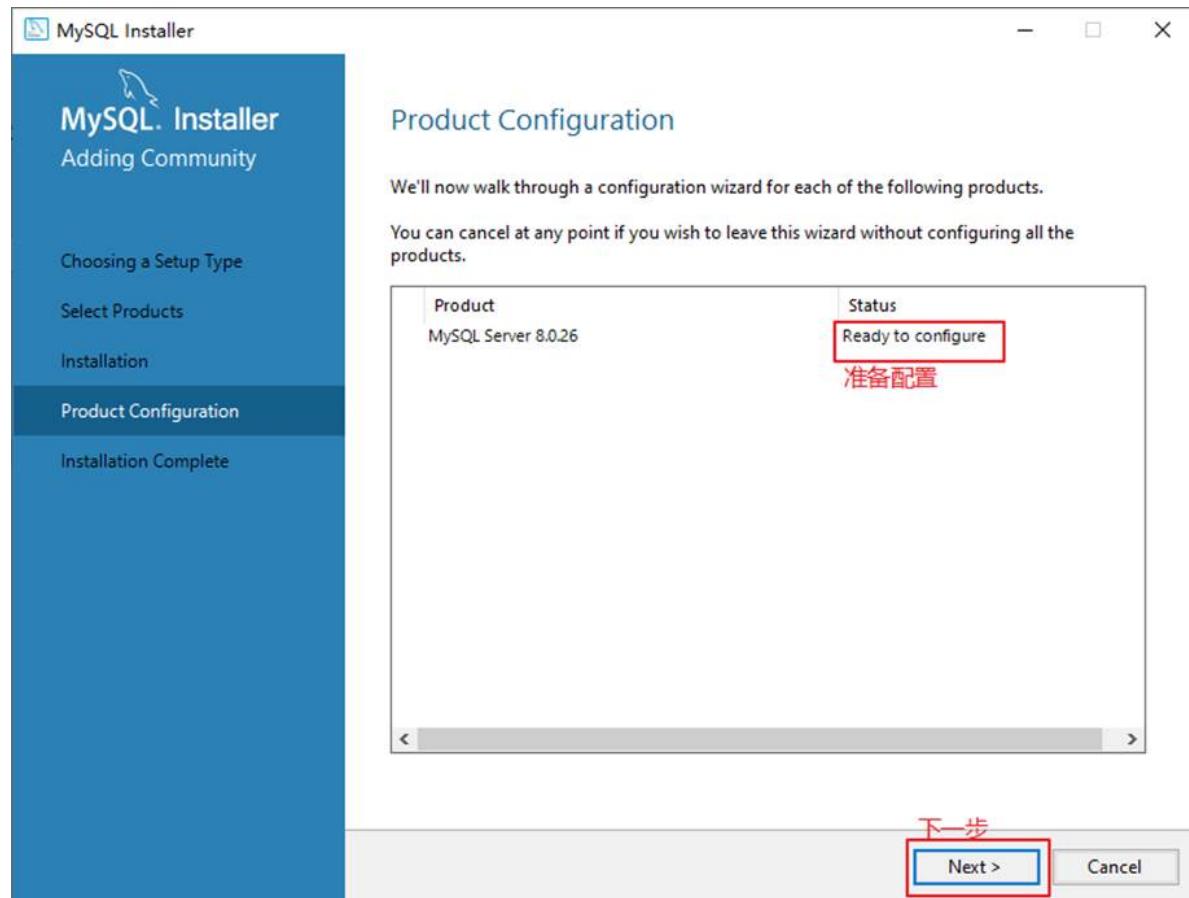
步骤5：安装完成后在“Status”（状态）列表下将显示“Complete”（安装完成），如图所示。

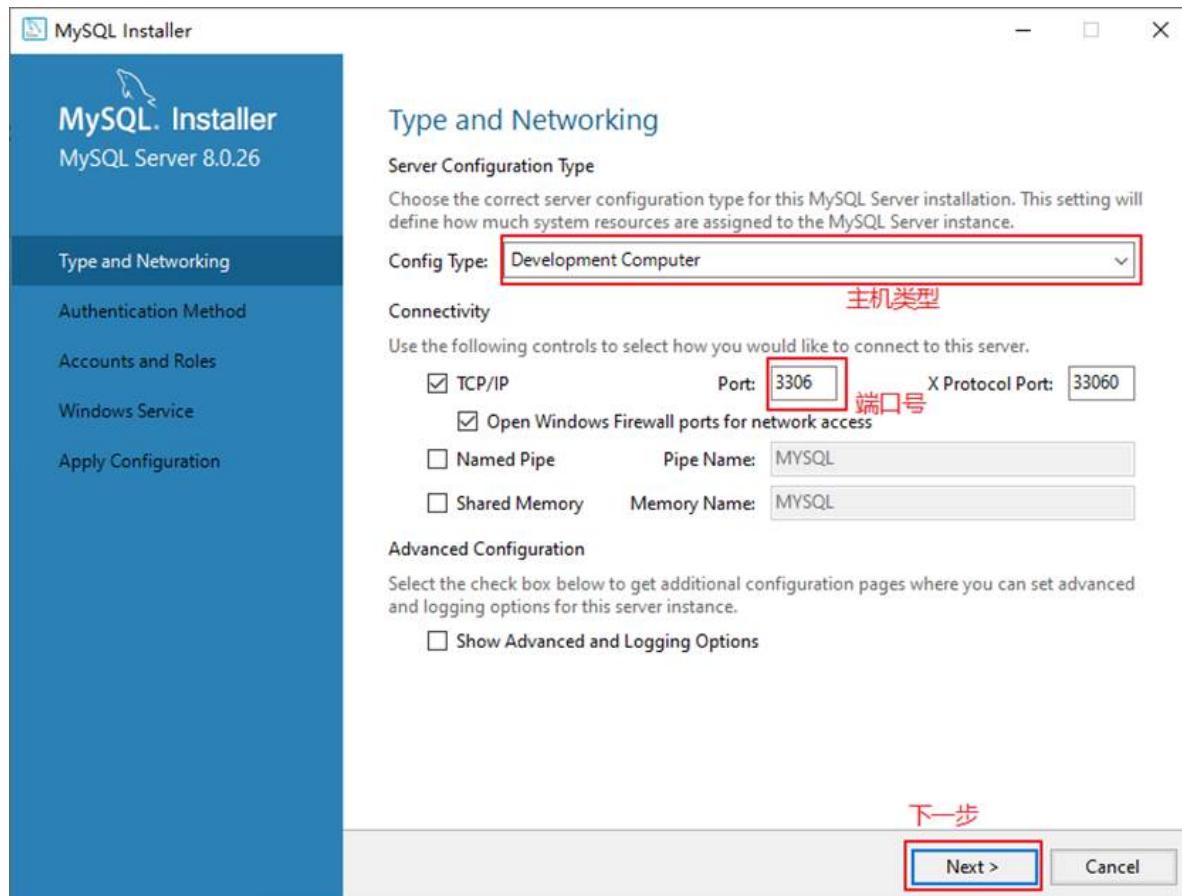


2.4 配置MySQL8.0

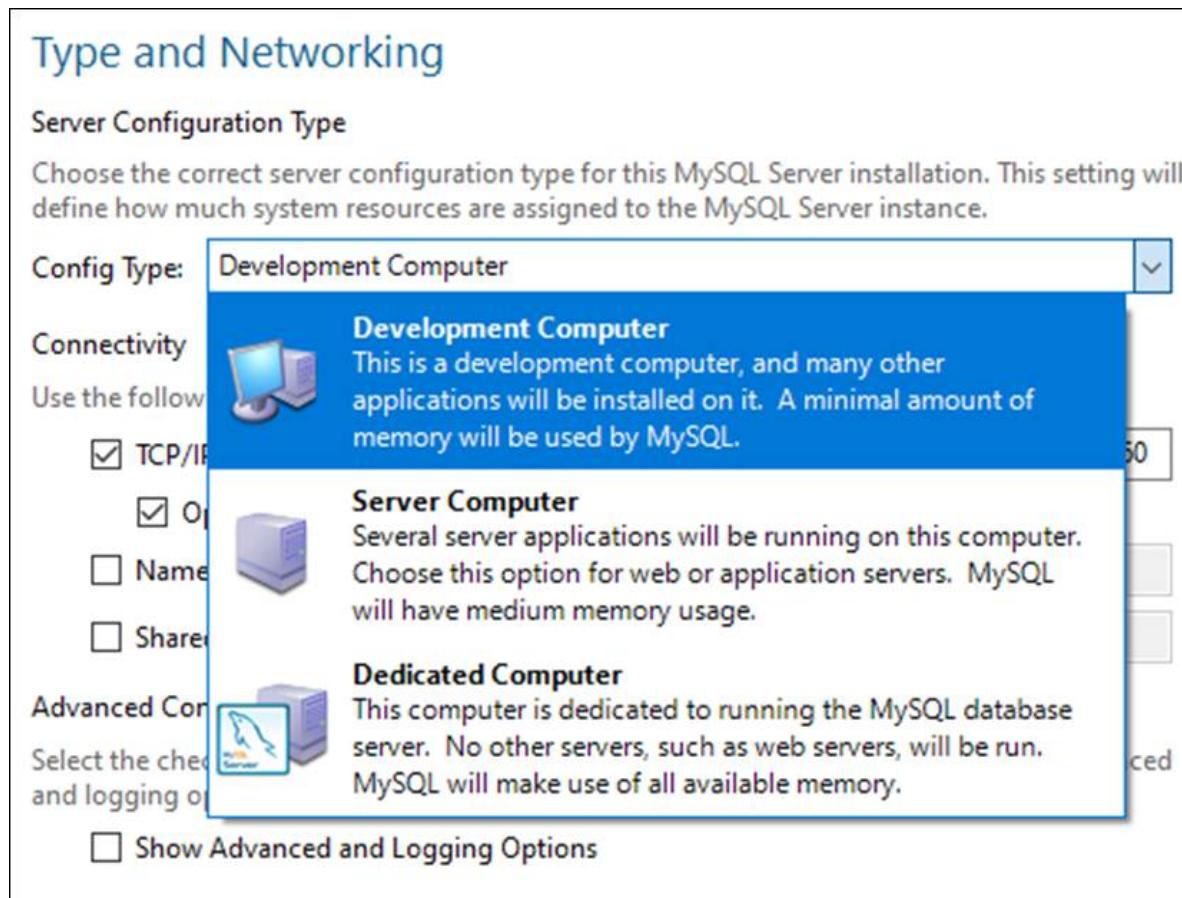
MySQL安装之后，需要对服务器进行配置。具体的配置步骤如下。

步骤1：在上一个小节的最后一步，单击“Next”（下一步）按钮，就可以进入产品配置窗口。



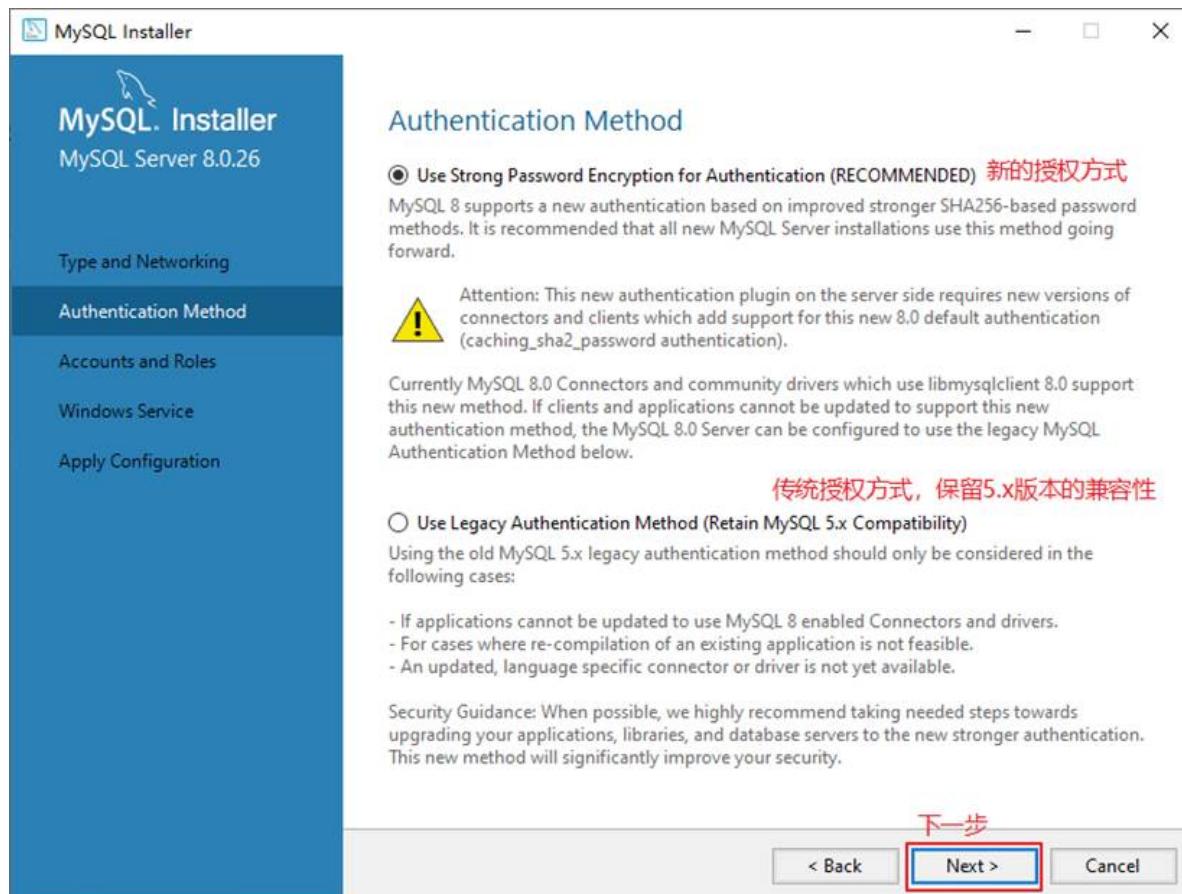


其中，“Config Type”选项用于设置服务器的类型。单击该选项右侧的下三角按钮，即可查看3个选项，如图所示。

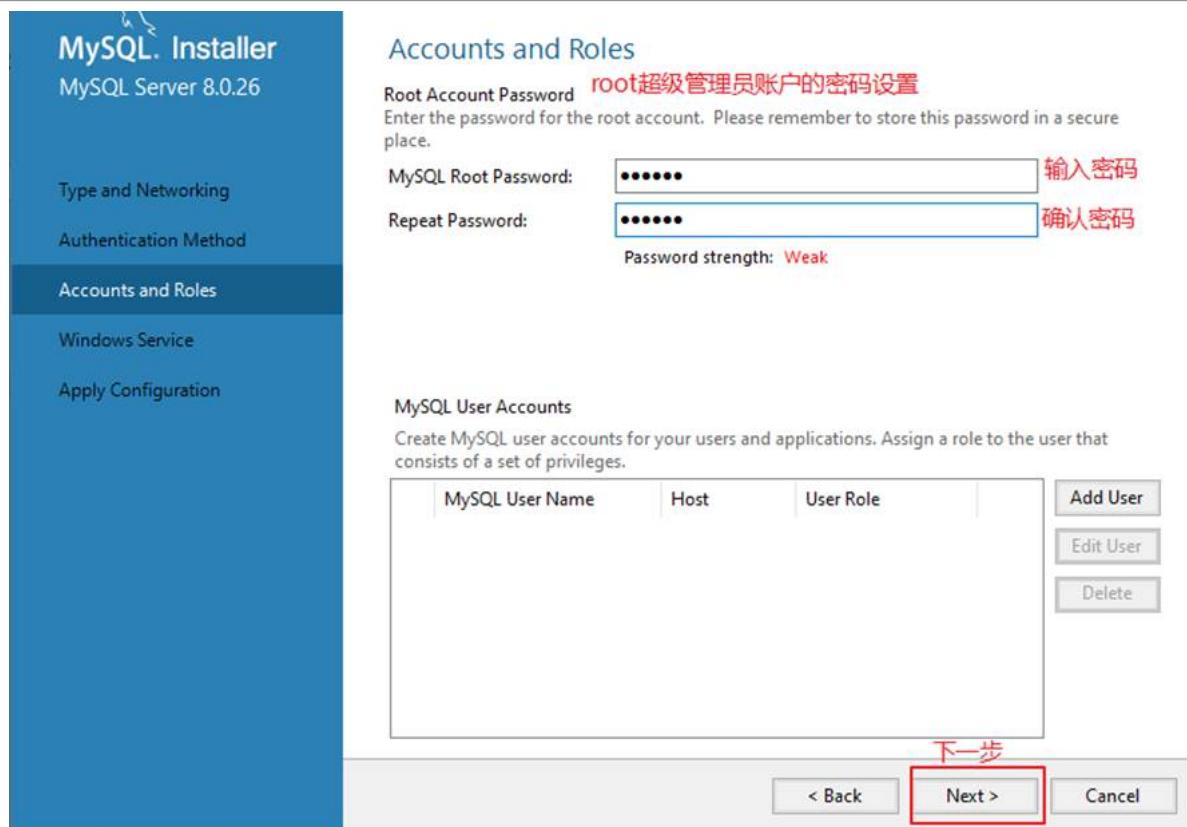


- **Server Machine (服务器)**：该选项代表服务器，MySQL服务器可以同其他服务器应用程序一起运行，例如Web服务器等。MySQL服务器配置成适当比例的系统资源。
- **Dedicated Machine (专用服务器)**：该选项代表只运行MySQL服务的服务器。MySQL服务器配置成使用所有可用系统资源。

步骤3：单击“Next”（下一步）按钮，打开设置授权方式窗口。其中，上面的选项是MySQL8.0提供的新的授权方式，采用SHA256基础的密码加密方法；下面的选项是传统授权方法（保留5.x版本兼容性）。

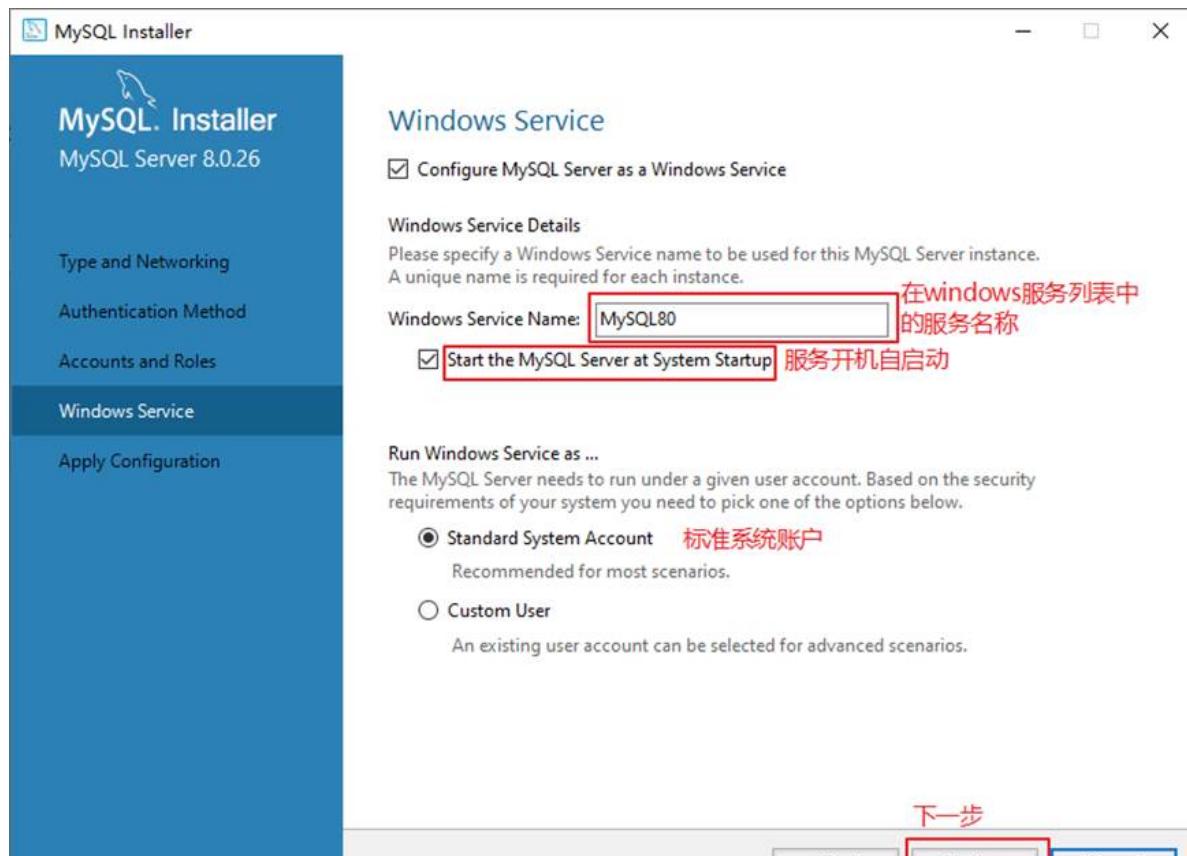


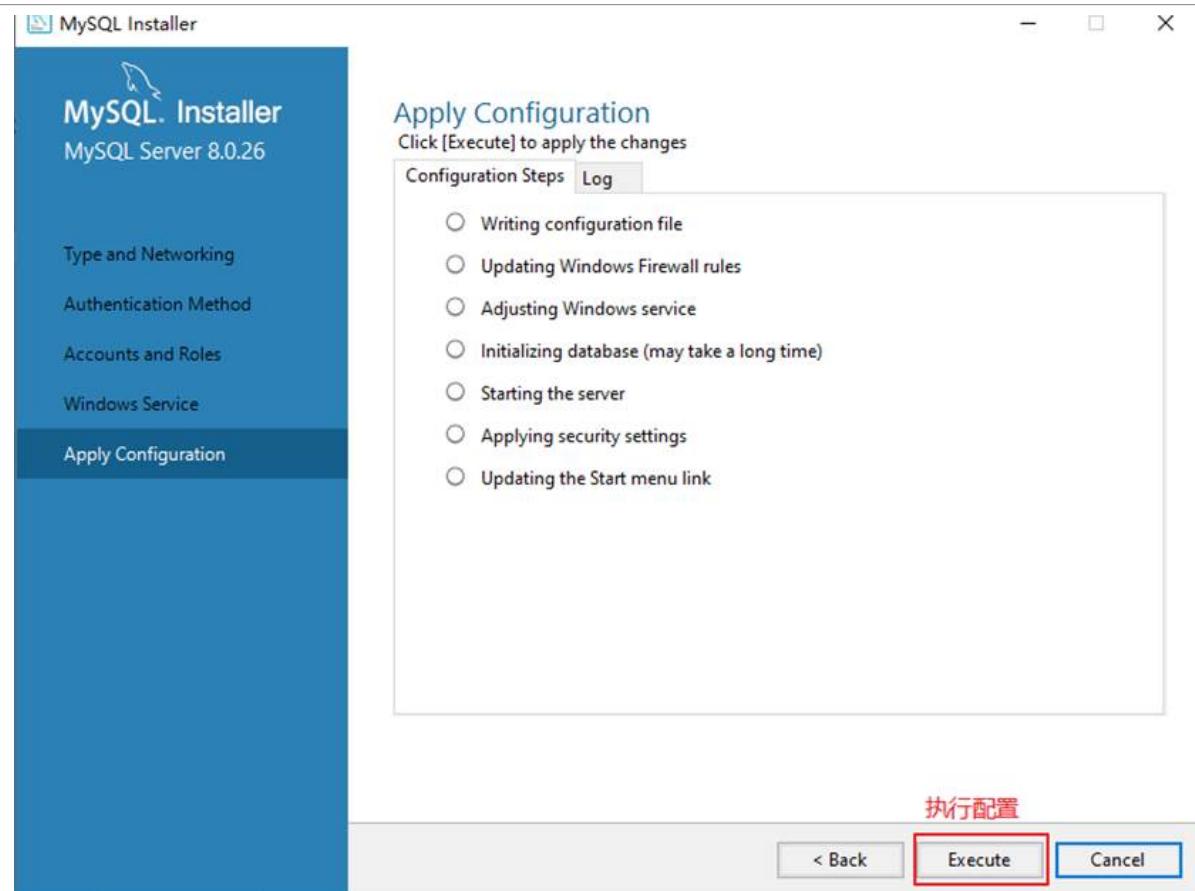
步骤4：单击“Next”（下一步）按钮，打开设置服务器root超级管理员的密码窗口，如图所示，需要输入两次同样的登录密码。也可以通过“Add User”添加其他用户，添加其他用户时，需要指定用户名、允许该用户名在哪台/哪些主机上登录，还可以指定用户角色等。此处暂不添加用户，用户管理在MySQL高级特性篇中讲解。



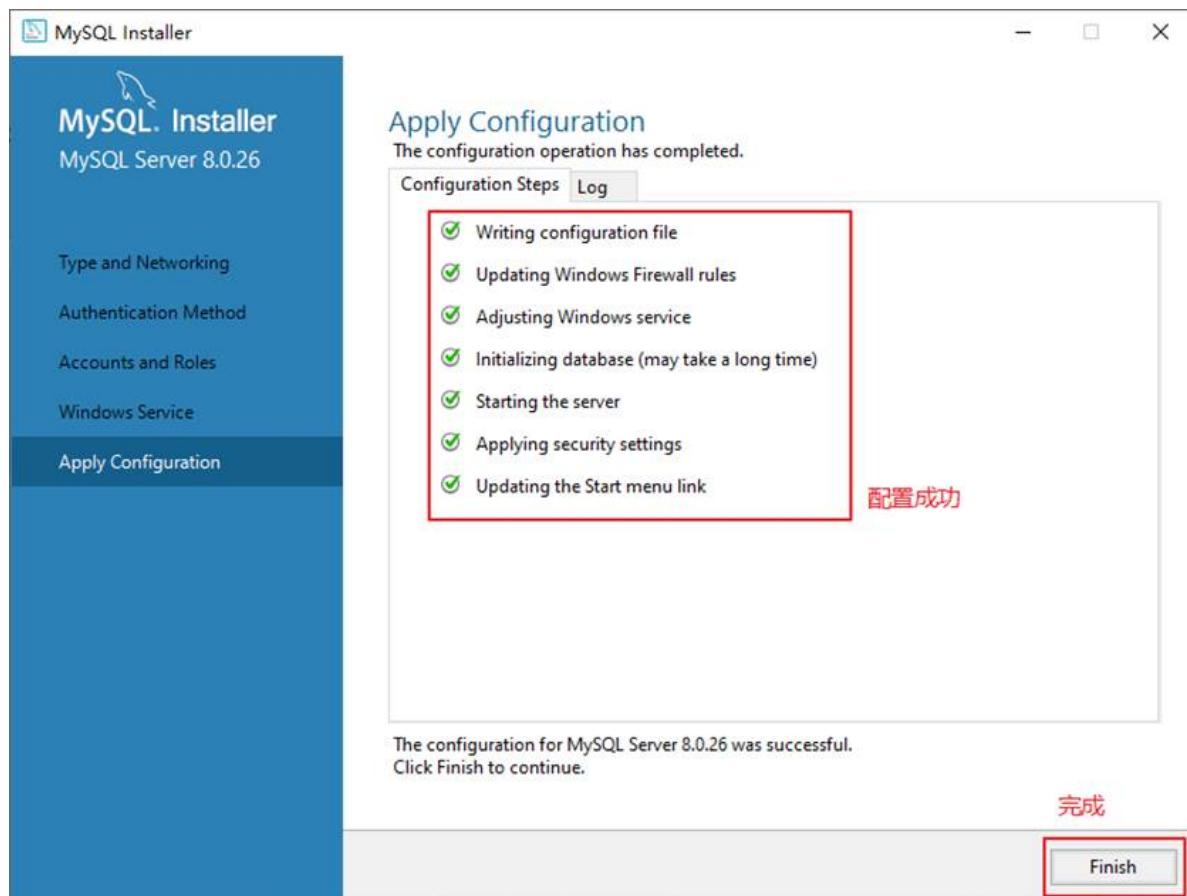
步骤5：单击“Next”（下一步）按钮，打开设置服务器名称窗口，如图所示。该服务名会出现在Windows服务列表中，也可以在命令行窗口中使用该服务名进行启动和停止服务。本书将服务名设置为“MySQL80”。如果希望开机自启动服务，也可以勾选“Start the MySQL Server at System Startup”选项（推荐）。

下面是选择以什么方式运行服务？可以选择“Standard System Account”(标准系统用户)或者“Custom User”(自定义用户)中的一个。这里推荐前者。

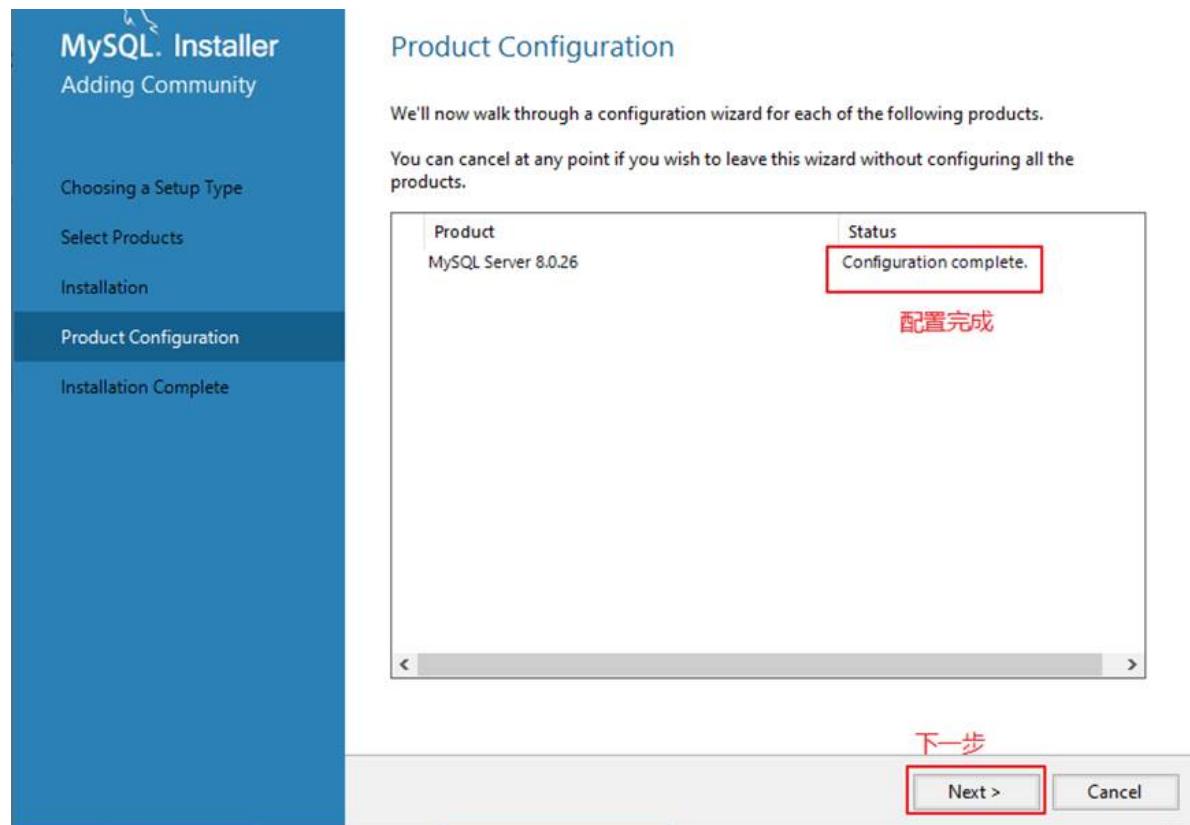




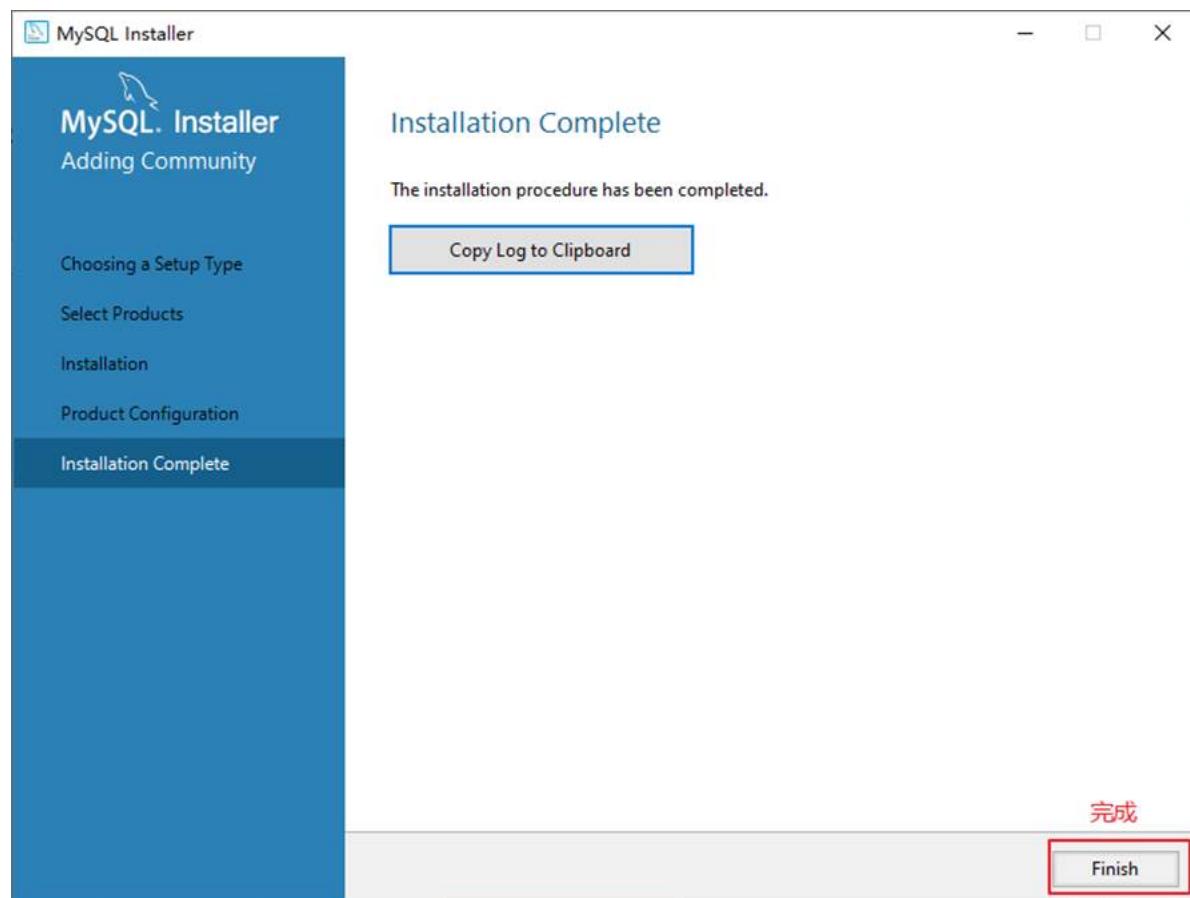
步骤7：完成配置，如图所示。单击“Finish”（完成）按钮，即可完成服务器的配置。



步骤8：如果还有其他产品需要配置，可以选择其他产品，然后继续配置。如果没有，直接选择“Next”（下一步），直接完成整个安装和配置过程。



步骤9：结束安装和配置。



如果不配置MySQL环境变量，就不能在命令行直接输入MySQL登录命令。下面说如何配置MySQL的环境变量：

步骤1：在桌面上右击【此电脑】图标，在弹出的快捷菜单中选择【属性】菜单命令。步骤2：打开【系统】窗口，单击【高级系统设置】链接。步骤3：打开【系统属性】对话框，选择【高级】选项卡，然后单击【环境变量】按钮。步骤4：打开【环境变量】对话框，在系统变量列表中选择path变量。步骤5：单击【编辑】按钮，在【编辑环境变量】对话框中，将MySQL应用程序的bin目录（C:\Program Files\MySQL\MySQL Server 8.0\bin）添加到变量值中，用分号将其与其他路径分隔开。步骤6：添加完成之后，单击【确定】按钮，这样就完成了配置path变量的操作，然后就可以直接输入MySQL命令来登录数据库了。

2.6 MySQL5.7 版本的安装、配置

• 安装

此版本的安装过程与上述过程除了版本号不同之外，其它环节都是相同的。所以这里省略了MySQL5.7.34版本的安装截图。

• 配置

配置环节与MySQL8.0版本确有细微不同。大部分情况下直接选择“Next”即可，不影响整理使用。

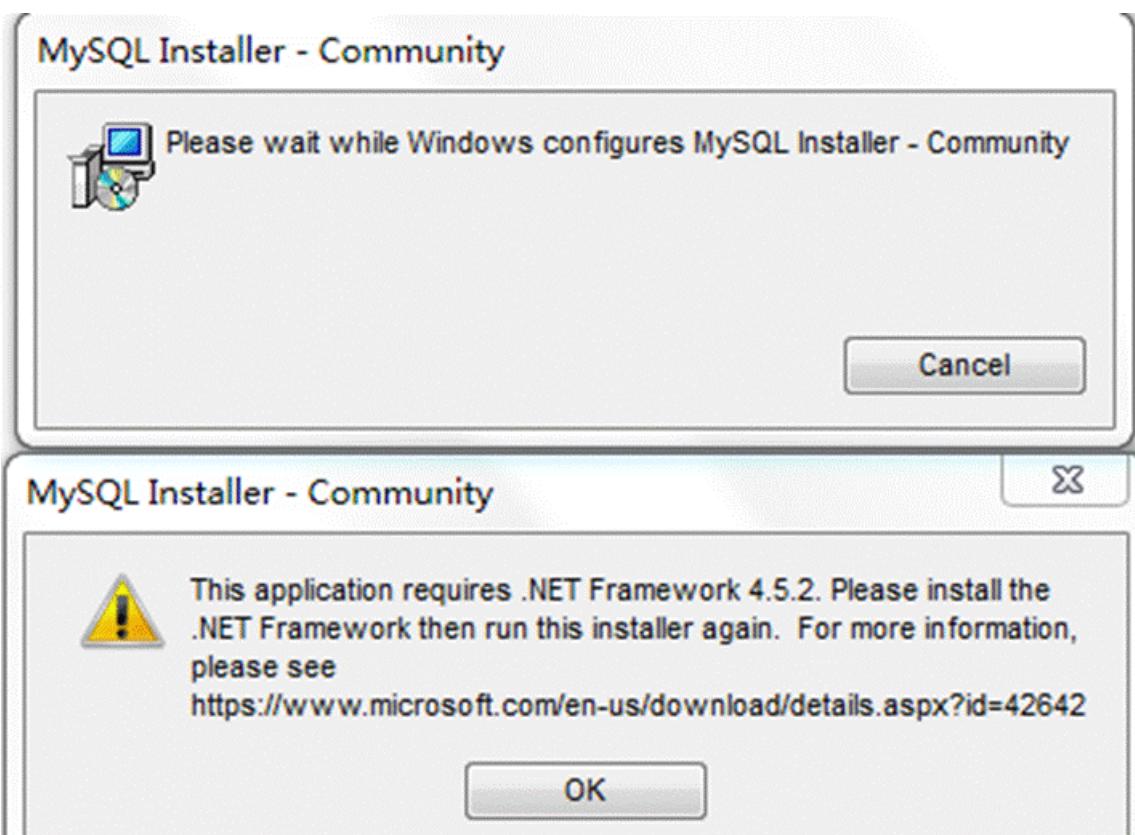
这里配置MySQL5.7时，重点强调：**与前面安装好的MySQL8.0不能使用相同的端口号。**

2.7 安装失败问题

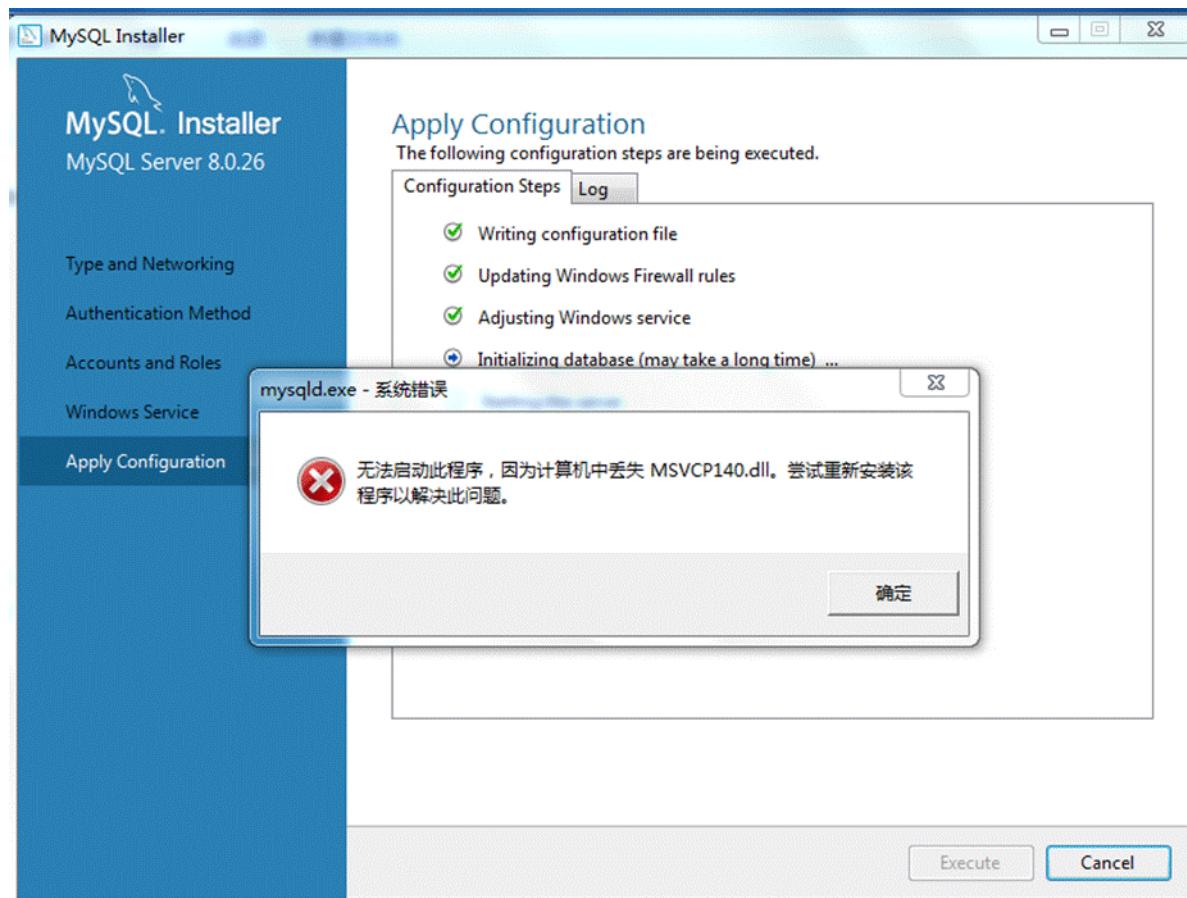
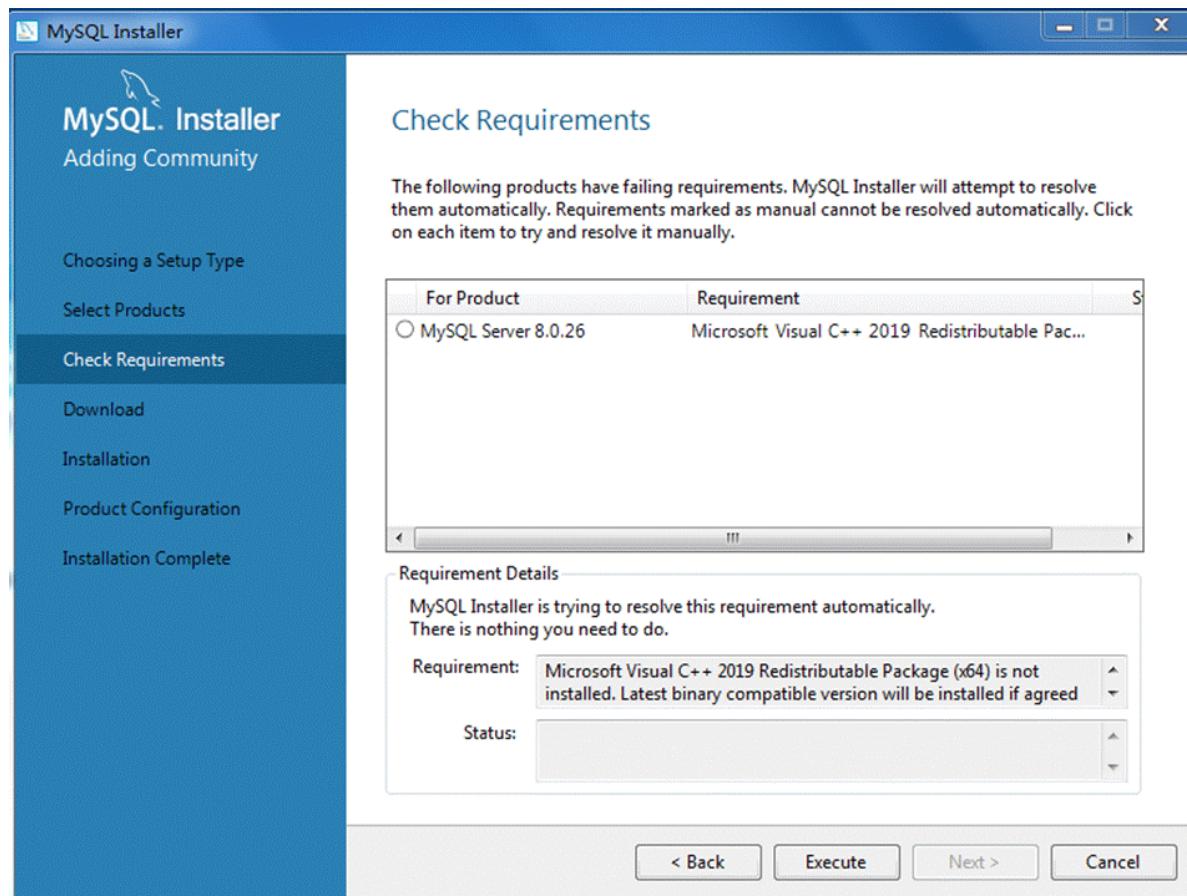
MySQL的安装和配置是一件非常简单的事，但是在操作过程中也可能出现问题，特别是初学者。

问题1：无法打开MySQL8.0软件安装包或者安装过程中失败，如何解决？

在运行MySQL8.0软件安装包之前，用户需要确保系统中已经安装了.NET Framework相关软件，如果缺少此软件，将不能正常地安装MySQL8.0软件。



另外，还要确保Windows Installer正常安装。windows上安装mysql8.0需要操作系统提前已安装好Microsoft Visual C++ 2015-2019。



问题2：卸载重装MySQL失败？

该问题通常是因为MySQL卸载时，没有完全清除相关信息导致的。

解决办法是，把以前的安装目录删除。如果之前安装并未单独指定过服务安装目录，则默认安装目录是“C:\Program Files\MySQL”，彻底删除该目录。同时删除MySQL的Data目录，如果之前安装并未单独指定过数据目录，则默认安装目录是“C:\ProgramData\MySQL”，该目录一般为隐藏目录。删除后，重新安装即可。

问题3：如何在Windows系统删除之前的未卸载干净的MySQL服务列表？

操作方法如下，在系统“搜索框”中输入“cmd”，按“Enter”（回车）键确认，弹出命令提示符界面。然后输入“sc delete MySQL服务名”，按“Enter”（回车）键，就能彻底删除残余的MySQL服务了。

3. MySQL的登录

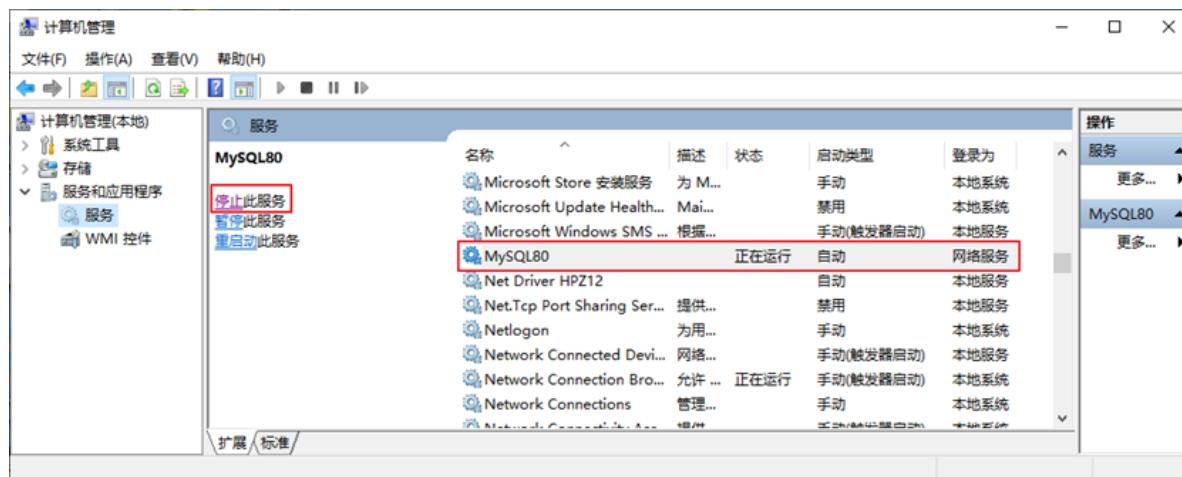
3.1 服务的启动与停止

MySQL安装完毕之后，需要启动服务器进程，不然客户端无法连接数据库。

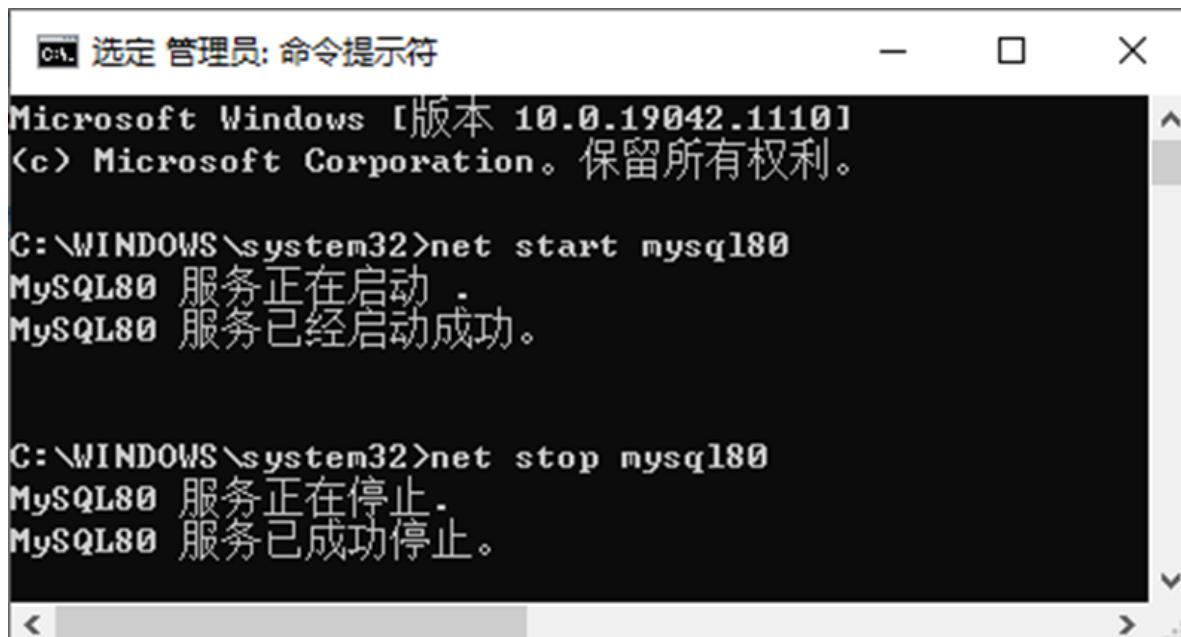
在前面的配置过程中，已经将MySQL安装为Windows服务，并且勾选当Windows启动、停止时，MySQL也自动启动、停止。

方式1：使用图形界面工具

- 步骤1：打开windows服务
 - 方式1：计算机（点击鼠标右键）→管理（点击）→服务和应用程序（点击）→服务（点击）
 - 方式2：控制面板（点击）→系统和安全（点击）→管理工具（点击）→服务（点击）
 - 方式3：任务栏（点击鼠标右键）→启动任务管理器（点击）→服务（点击）
 - 方式4：单击【开始】菜单，在搜索框中输入“services.msc”，按Enter键确认
- 步骤2：找到MySQL80（点击鼠标右键）→启动或停止（点击）



```
# 启动 MySQL 服务命令:  
net start MySQL服务名  
  
# 停止 MySQL 服务命令:  
net stop MySQL服务名
```



```
Microsoft Windows [版本 10.0.19042.1110]  
<c> Microsoft Corporation。保留所有权利。  
  
C:\WINDOWS\system32>net start mysql80  
MySQL80 服务正在启动。  
MySQL80 服务已经启动成功。  
  
C:\WINDOWS\system32>net stop mysql80  
MySQL80 服务正在停止。  
MySQL80 服务已成功停止。
```

说明：

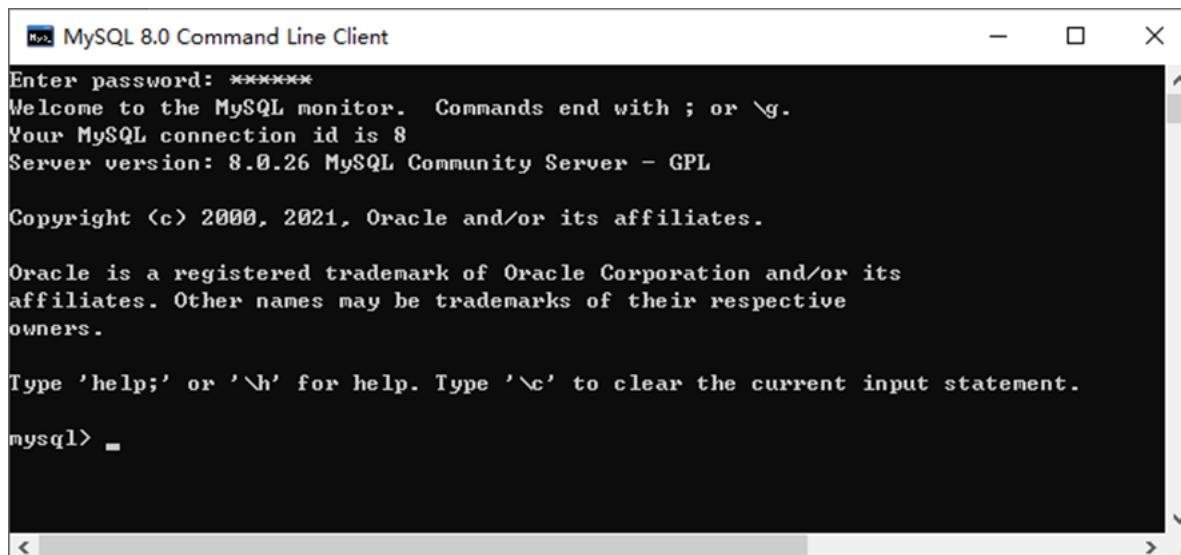
1. start和stop后面的服务名应与之前配置时指定的服务名一致。
2. 如果当你输入命令后，提示“拒绝服务”，请以 [系统管理员身份](#) 打开命令提示符界面重新尝试。

3.2 自带客户端的登录与退出

当MySQL服务启动完成后，便可以通过客户端来登录MySQL数据库。注意：确认服务是开启的。

登录方式1：MySQL自带客户端

开始菜单 → 所有程序 → MySQL → MySQL 8.0 Command Line Client



```
MySQL 8.0 Command Line Client  
Enter password: *****  
Welcome to the MySQL monitor. Commands end with ; or \g.  
Your MySQL connection id is 8  
Server version: 8.0.26 MySQL Community Server - GPL  
  
Copyright <c> 2000, 2021, Oracle and/or its affiliates.  
  
Oracle is a registered trademark of Oracle Corporation and/or its  
affiliates. Other names may be trademarks of their respective  
owners.  
  
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.  
mysql>
```

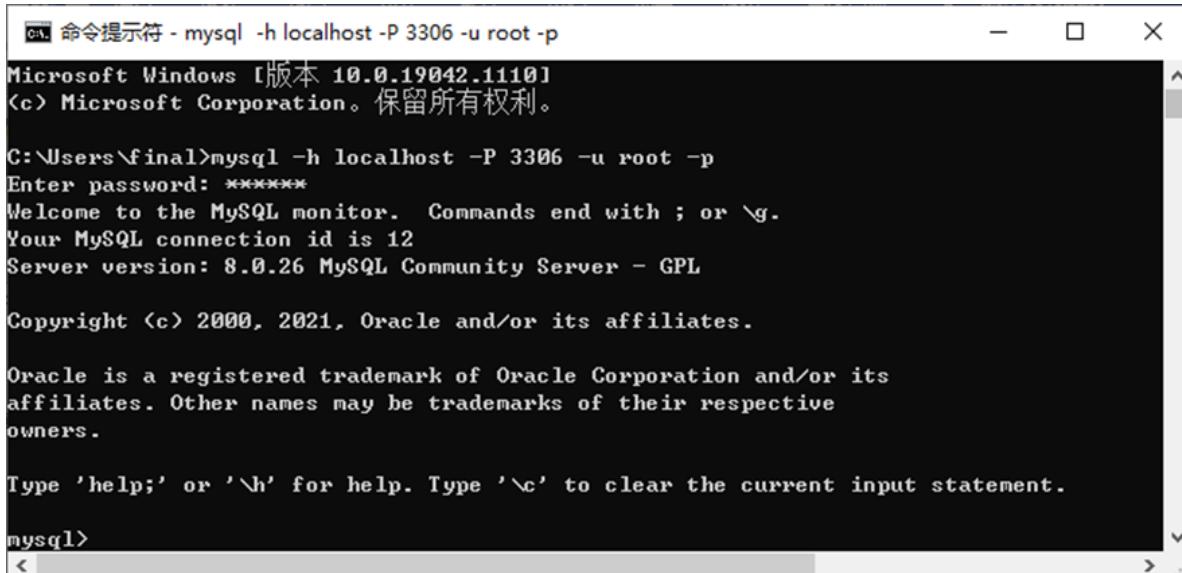
说明：仅限于root用户

- 格式：

```
mysql -h 主机名 -P 端口号 -u 用户名 -p密码
```

- 举例：

```
mysql -h localhost -P 3306 -u root -pabc123 # 这里我设置的root用户的密码是abc123
```



The screenshot shows a Windows command prompt window titled '命令提示符 - mysql -h localhost -P 3306 -u root -p'. It displays the MySQL monitor welcome message, including the connection ID (12), server version (8.0.26), and various copyright and trademark information. At the bottom, it shows the MySQL prompt 'mysql>'.

注意：

- (1) -p与密码之间不能有空格，其他参数名与参数值之间可以有空格也可以没有空格。如：

```
mysql -hlocalhost -P3306 -uroot -pabc123
```

- (2) 密码建议在下一行输入，保证安全

```
mysql -h localhost -P 3306 -u root -p  
Enter password:****
```

- (3) 客户端和服务器在同一台机器上，所以输入localhost或者IP地址127.0.0.1。同时，因为是连接本机： -hlocalhost就可以省略，如果端口号没有修改： -P3306也可以省略

简写成：

```
mysql -u root -p  
Enter password:****
```

连接成功后，有关于MySQL Server服务版本的信息，还有第几次连接的id标识。

也可以在命令行通过以下方式获取MySQL Server服务版本的信息：

```
c:\> mysql -V
```

```
c:\> mysql --version
```

或**登录**后，通过以下方式查看当前版本信息：

```
mysql> select version();
```

```
exit  
或  
quit
```

4. MySQL演示使用

4.1 MySQL的使用演示

1、查看所有的数据库

```
show databases;
```

“information_schema”是 MySQL 系统自带的数据库，主要保存 MySQL 数据库服务器的系统信息，比如数据库的名称、数据表的名称、字段名称、存取权限、数据文件所在的文件夹和系统使用的文件夹，等等

“performance_schema”是 MySQL 系统自带的数据库，可以用来监控 MySQL 的各类性能指标。

“sys”数据库是 MySQL 系统自带的数据库，主要作用是以一种更容易被理解的方式展示 MySQL 数据库服务器的各类性能指标，帮助系统管理员和开发人员监控 MySQL 的技术性能。

“mysql”数据库保存了 MySQL 数据库服务器运行时需要的系统信息，比如数据文件夹、当前使用的字符集、约束检查信息，等等

为什么 Workbench 里面我们只能看到“demo”和“sys”这 2 个数据库呢？

这是因为，Workbench 是图形化的管理工具，主要面向开发人员，“demo”和“sys”这 2 个数据库已经够用了。如果有特殊需求，比如，需要监控 MySQL 数据库各项性能指标、直接操作 MySQL 数据库系统文件等，可以由 DBA 通过 SQL 语句，查看其它的系统数据库。

2、创建自己的数据库

```
create database 数据库名;  
  
#创建atguigudb数据库，该名称不能与已经存在的数据库重名。  
create database atguigudb;
```

3、使用自己的数据库

```
use 数据库名;  
  
#使用atguigudb数据库  
use atguigudb;
```

说明：如果没有使用use语句，后面针对数据库的操作也没有加“数据名”的限定，那么会报“ERROR 1046 (3D000): No database selected”（没有选择数据库）

使用完use语句之后，如果接下来的SQL都是针对一个数据库操作的，那就不用重复use了，如果要针对另一个数据库操作，那么要重新use。

4、查看某个库的所有表格

```
show tables from 数据库名;
```

5、创建新的表格

```
create table 表名称(  
    字段名 数据类型,  
    字段名 数据类型  
) ;
```

说明：如果是最后一个字段，后面就用加逗号，因为逗号的作用是分割每个字段。

```
#创建学生表  
create table student(  
    id int,  
    name varchar(20) #说名字最长不超过20个字符  
) ;
```

6、查看一个表的数据

```
select * from 数据库表名称;
```

```
#查看学生表的数据  
select * from student;
```

7、添加一条记录

```
insert into 表名称 values(值列表);  
  
#添加两条记录到student表中  
insert into student values(1,'张三');  
insert into student values(2,'李四');
```

报错：

```
mysql> insert into student values(1,'张三');  
ERROR 1366 (HY000): Incorrect string value: '\xD5\xC5\xC8\xFD' for column 'name' at  
row 1  
mysql> insert into student values(2,'李四');  
ERROR 1366 (HY000): Incorrect string value: '\xC0\xEE\xCB\xC4' for column 'name' at  
row 1  
mysql> show create table student;
```

字符集的问题。

8、查看表的创建信息

```
show create table 表名称\G
```

```
#查看student表的详细创建信息  
show create table student\G
```

```
Table: student
Create Table: CREATE TABLE `student` (
  `id` int(11) DEFAULT NULL,
  `name` varchar(20) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1
1 row in set (0.00 sec)
```

上面的结果显示student的表格的默认字符集是“latin1”不支持中文。

9、查看数据库的创建信息

```
show create database 数据库名\G

#查看atguigudb数据库的详细创建信息
show create database atguigudb\G
```

```
#结果如下
***** 1. row *****
Database: atguigudb
Create Database: CREATE DATABASE `atguigudb` /*!40100 DEFAULT CHARACTER SET latin1 */
1 row in set (0.00 sec)
```

上面的结果显示atguigudb数据库也不支持中文，字符集默认是latin1。

10、删除表格

```
drop table 表名称;

#删除学生表
drop table student;
```

11、删除数据库

```
drop database 数据库名;

#删除atguigudb数据库
drop database atguigudb;
```

4.2 MySQL的编码设置

MySQL5.7中

问题再现：命令行操作sql乱码问题

```
mysql> INSERT INTO t_stu VALUES(1,'张三','男');
ERROR 1366 (HY000): Incorrect string value: '\xD5\xC5\xC8\xFD' for column 'sname' at
row 1
```

问题解决

步骤1：查看编码命令

```
show variables like 'character_%';
show variables like 'collation_%';
```

步骤2：修改mysql的数据目录下的my.ini配置文件

北京宏福校区：010-56253825 深圳西部硅谷校区：0755-23060254 上海大江商厦校区：021-57652717

```
default-character-set=utf8 #默认字符集

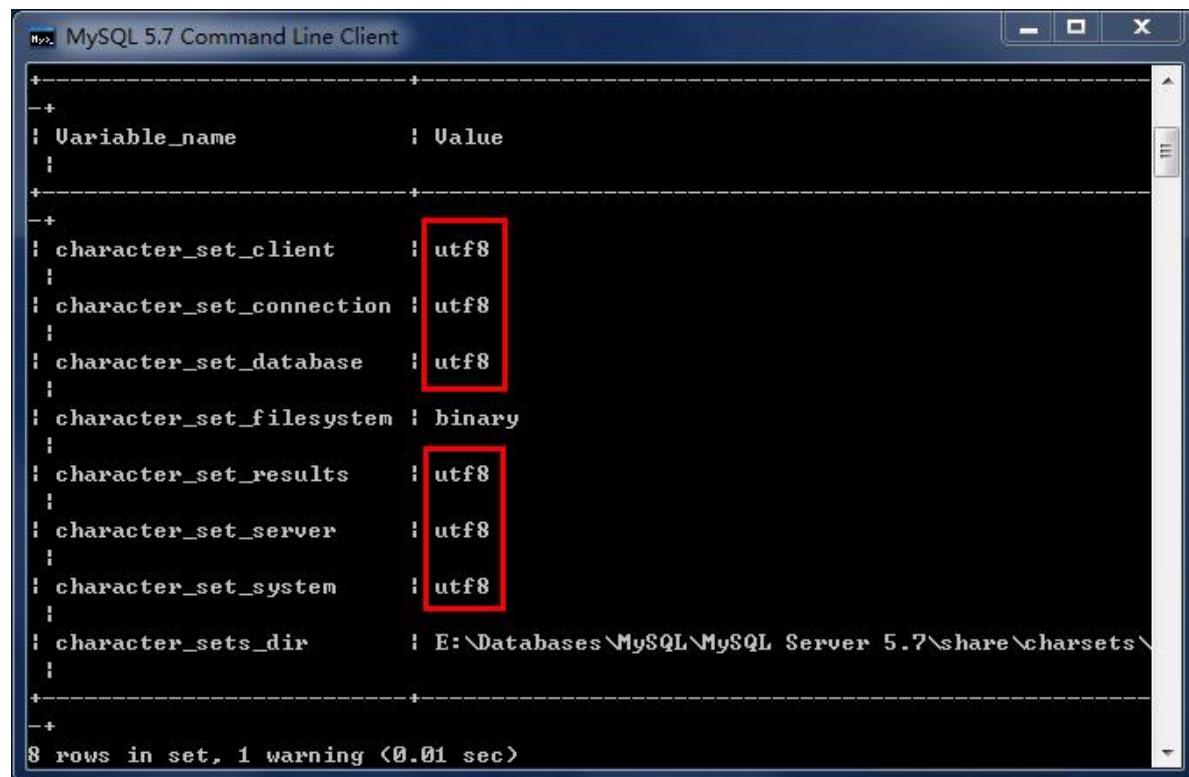
[mysqld] # 大概在76行左右，在其下添加
...
character-set-server=utf8
collation-server=utf8_general_ci
```

注意：建议修改配置文件使用notepad++等高级文本编辑器，使用记事本等软件打开修改后可能会导致文件编码修改为“含BOM头”的编码，从而服务重启失败。

步骤3：重启服务

步骤4：查看编码命令

```
show variables like 'character_%';
show variables like 'collation_%';
```



The screenshot shows the MySQL 5.7 Command Line Client window. The command `show variables like 'character_%';` was run, displaying the following table:

Variable_name	Value
character_set_client	utf8
character_set_connection	utf8
character_set_database	utf8
character_set_filesystem	binary
character_set_results	utf8
character_set_server	utf8
character_set_system	utf8
character_sets_dir	E:\Databases\MySQL\MySQL Server 5.7\share\charsets\

Rows: 8 rows in set, 1 warning (0.01 sec)

```
8 rows in set, 1 warning (0.01 sec)

mysql> show variables like 'collation_%';
+-----+-----+
| Variable_name      | Value          |
+-----+-----+
| collation_connection | utf8_general_ci |
| collation_database   | utf8_general_ci |
| collation_server     | utf8_general_ci |
+-----+-----+
3 rows in set, 1 warning (0.01 sec)

mysql>
```

- 如果是以上配置就说明对了。接着我们就可以新创建数据库、新创建数据表，接着添加包含中文的数据了。

MySQL8.0中

在MySQL 8.0版本之前，默认字符集为latin1，utf8字符集指向的是utf8mb3。网站开发人员在数据库设计的时候往往会将编码修改为utf8字符集。如果遗忘修改默认的编码，就会出现乱码的问题。从MySQL 8.0开始，数据库的默认编码改为 **utf8mb4**，从而避免了上述的乱码问题。

5. MySQL图形化管理工具

MySQL图形化管理工具极大地方便了数据库的操作与管理，常用的图形化管理工具有：MySQL Workbench、phpMyAdmin、Navicat Premium、MySQLDumper、SQLyog、dbeaver、MySQL ODBC Connector。

工具1. MySQL Workbench

MySQL官方提供的图形化管理工具MySQL Workbench完全支持MySQL 5.0以上的版本。MySQL Workbench分为社区版和商业版，社区版完全免费，而商业版则是按年收费。

MySQL Workbench 为数据库管理员、程序开发者和系统规划师提供可视化设计、模型建立、以及数据库管理功能。它包含了用于创建复杂的数据建模ER模型，正向和逆向数据库工程，也可以用于执行通常需要花费大量时间的、难以变更和管理的文档任务。

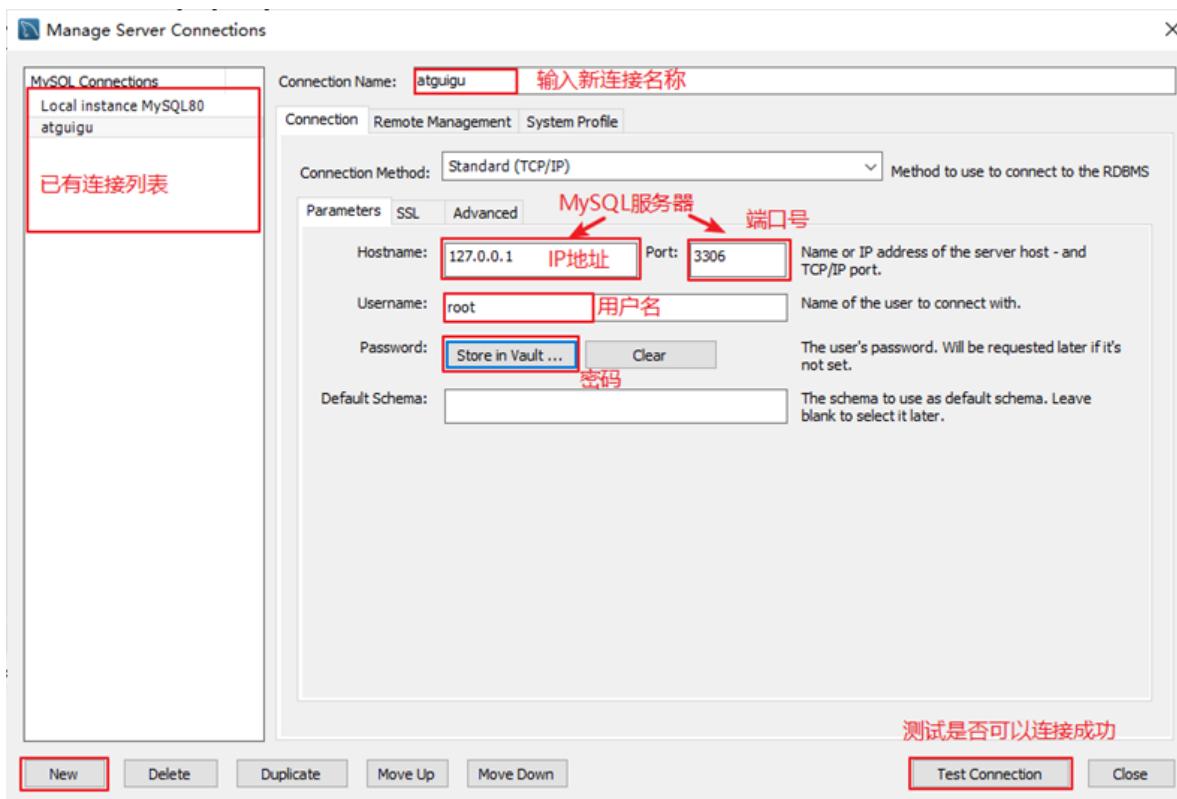
下载地址：<http://dev.mysql.com/downloads/workbench/>。

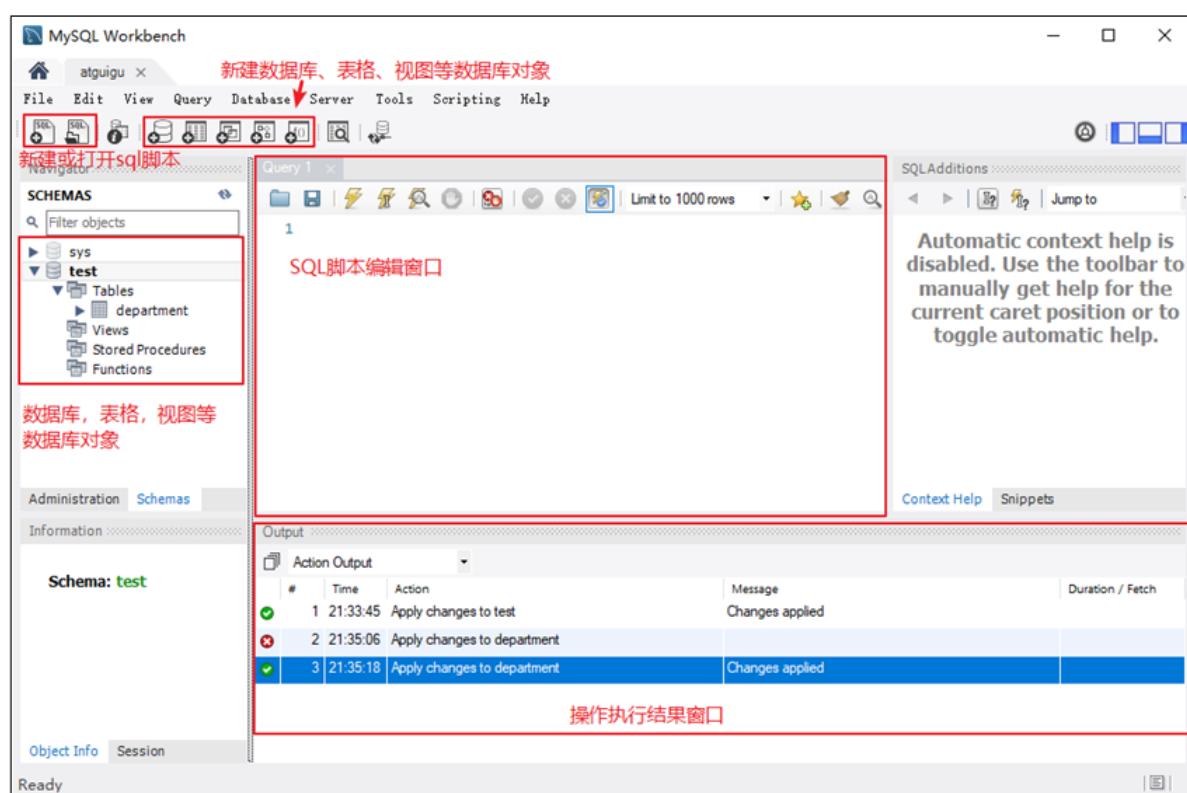
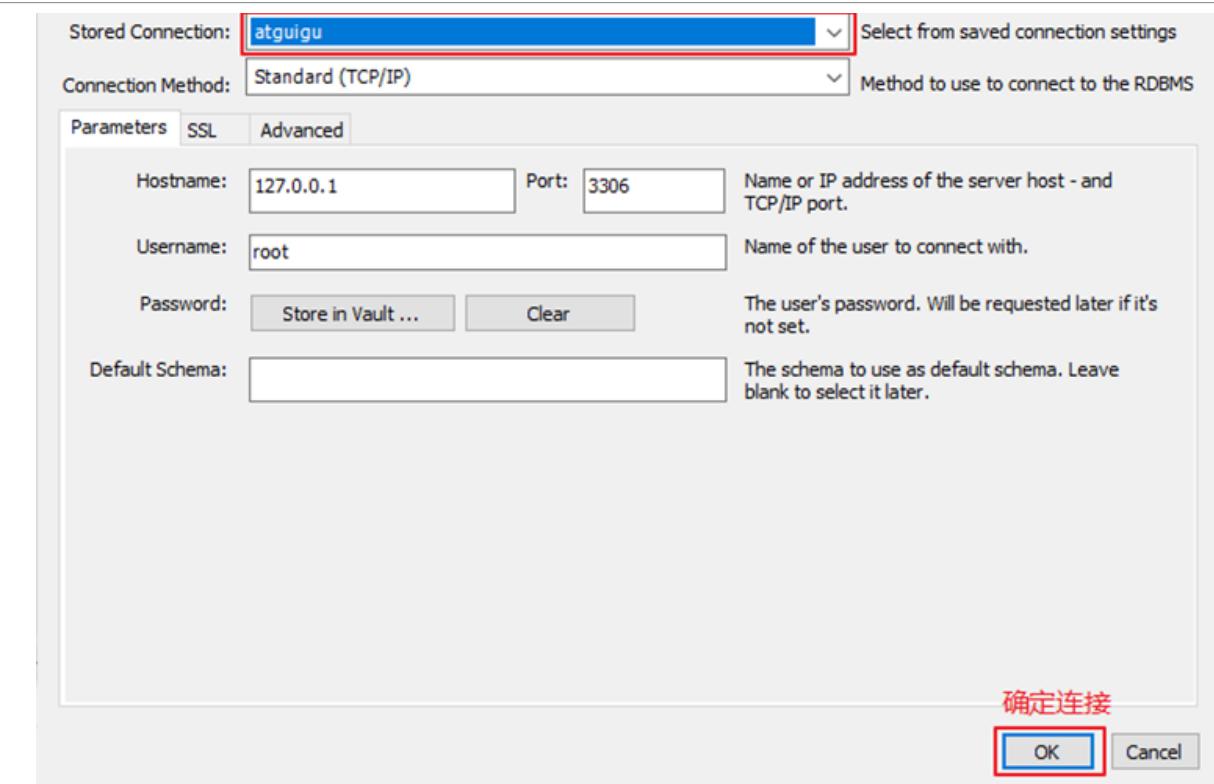
使用：

首先，我们点击 Windows 左下角的“开始”按钮，如果你是 Win10 系统，可以直接看到所有程序。接着，找到“MySQL”，点开，找到“MySQL Workbench 8.0 CE”。点击打开 Workbench，如下图所示：



左下角有个本地连接，点击，录入 Root 的密码，登录本地 MySQL 数据库服务器，如下图所示：



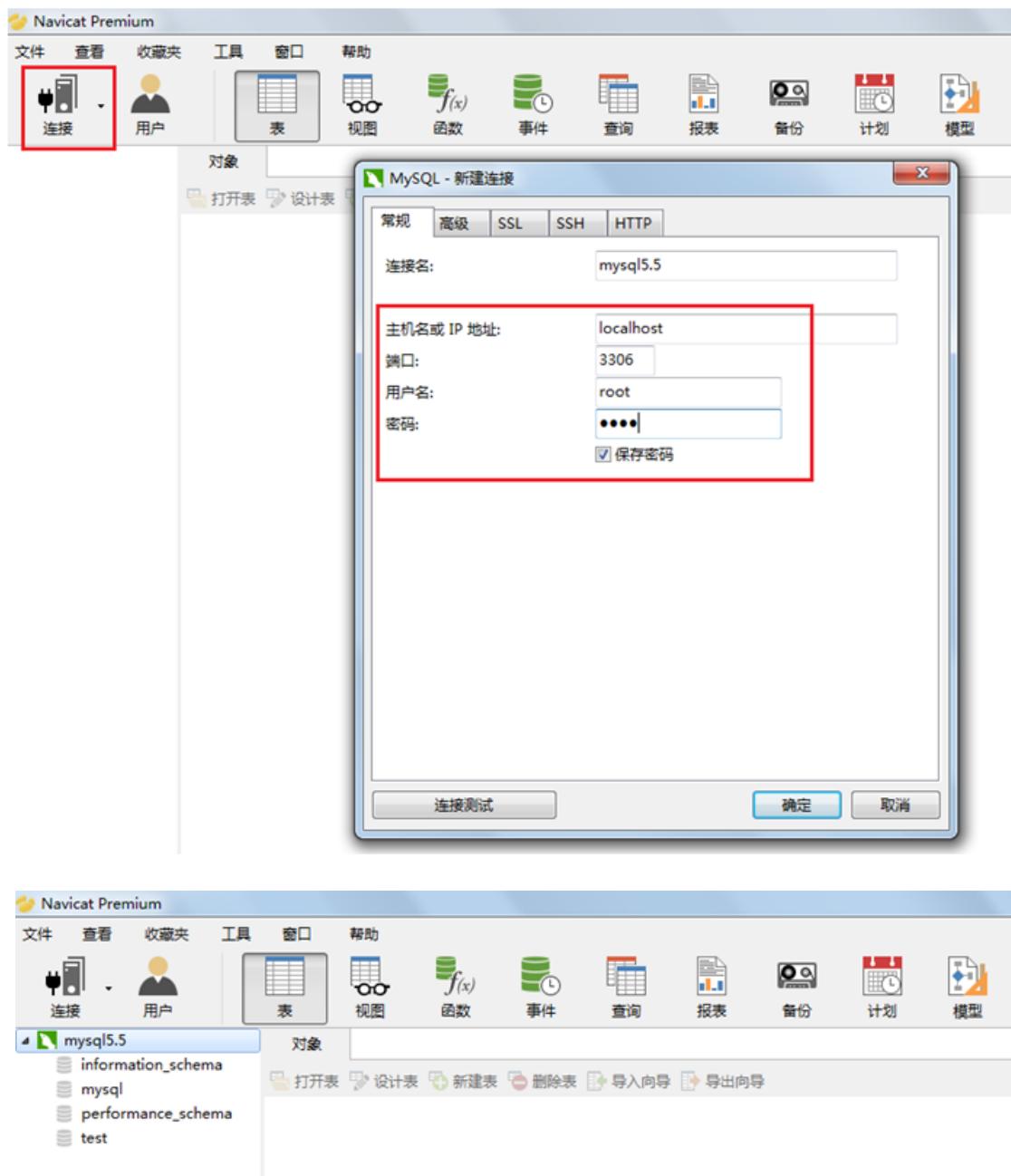


这是一个图形化的界面，我来给你介绍下这个界面。

- 上方是菜单。左上方是导航栏，这里我们可以看到 MySQL 数据库服务器里面的数据库，包括数据表、视图、存储过程和函数；左下方是信息栏，可以显示上方选中的数据库、数据表等对象的信息。
- 中间上方是工作区，你可以在这里写 SQL 语句，点击上方菜单栏左边的第三个运行按钮，就可以执行工作区的 SQL 语句了。
- 中间下方是输出区，用来显示 SQL 语句的运行情况，包括什么时间开始运行的、运行的内容、运行的输出，以及所花费的时长等信息。

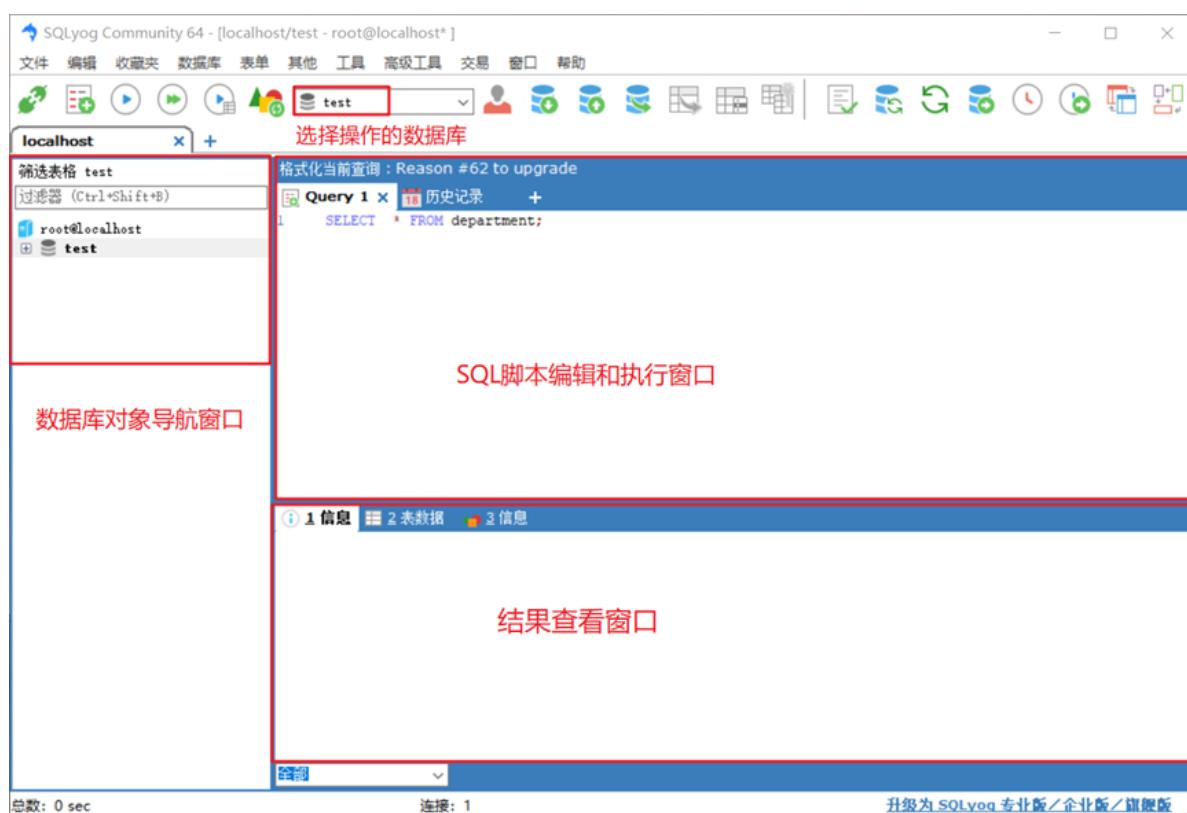
工具2. Navicat

Navicat MySQL是一个强大的MySQL数据库服务器管理和开发工具。它可以与任何3.21或以上版本的MySQL一起工作，支持触发器、存储过程、函数、事件、视图、管理用户等，对于新手来说易学易用。其精心设计的图形用户界面（GUI）可以让用户用一种安全简便的方式来快速方便地创建、组织、访问和共享信息。Navicat支持中文，有免费版本提供。下载地址：<http://www.navicat.com/>。



工具3. SQLyog

SQLyog 是业界著名的 Webyog 公司出品的一款简洁高效、功能强大的图形化 MySQL 数据库管理工具。这款工具是使用C++语言开发的。该工具可以方便地创建数据库、表、视图和索引等，还可以方便地进行插入、更新和删除等操作，同时可以方便地进行数据库、数据表的备份和还原。该工具不仅可以通过SQL文件进行大量文件的导入和导出，还可以导入和导出XML、HTML和CSV等多种格式的数据。下载地址：<http://www.webyog.com/>，读者也可以搜索中文版的下载地址。

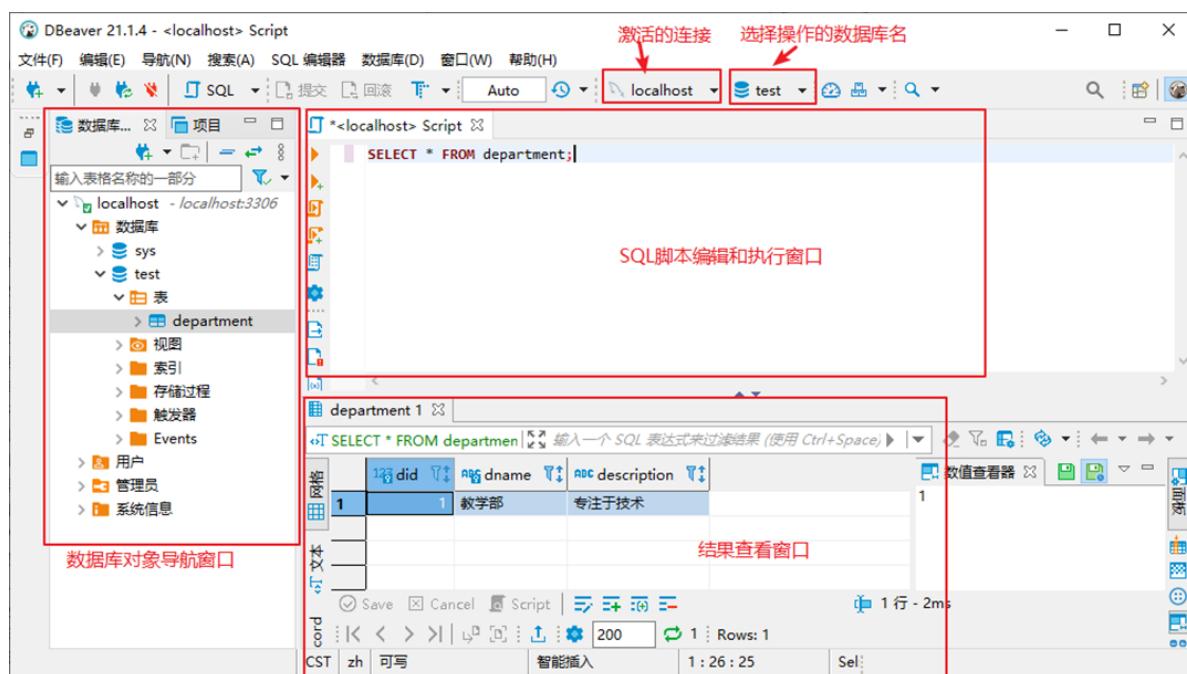
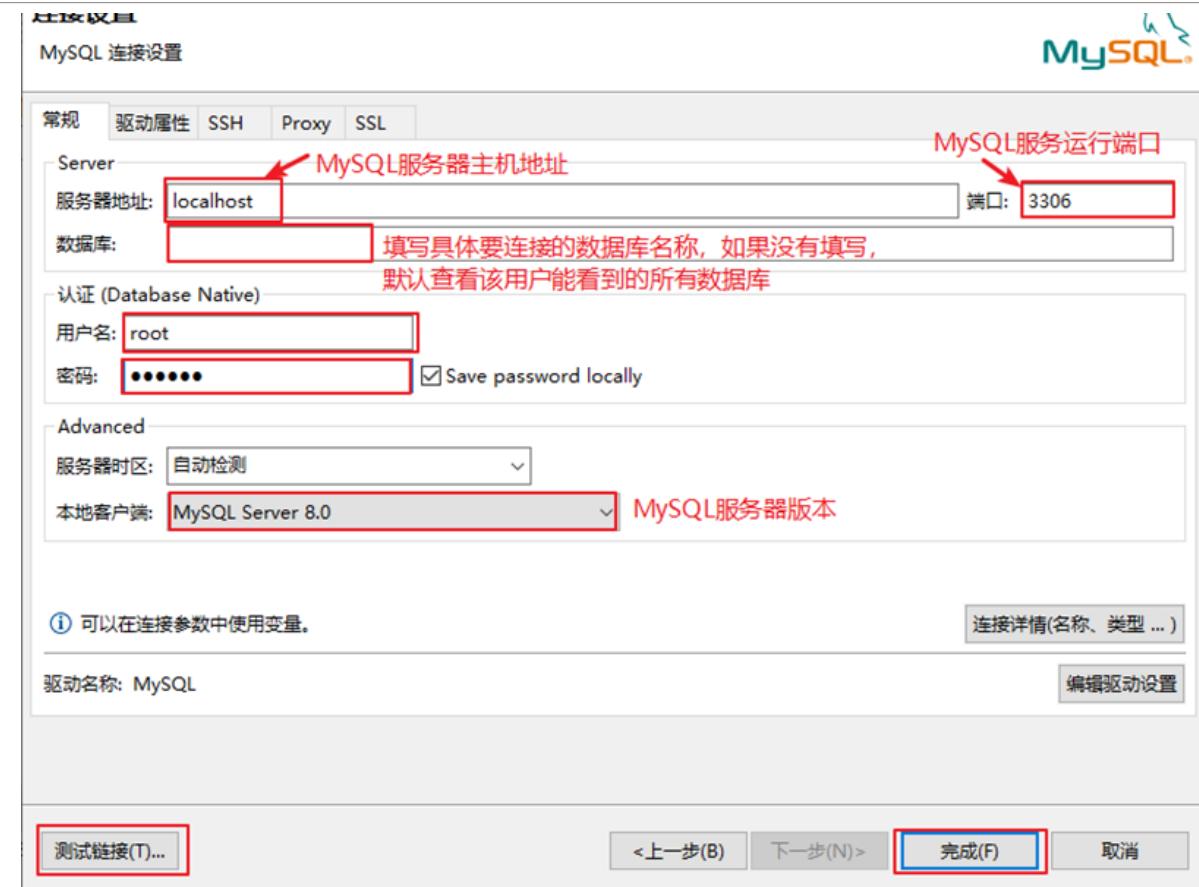


工具4：dbeaver

DBeaver是一个通用的数据库管理工具和 SQL 客户端，支持所有流行的数据库：MySQL、PostgreSQL、SQLite、Oracle、DB2、SQL Server、Sybase、MS Access、Teradata、Firebird、Apache Hive、Phoenix、Presto等。DBeaver比大多数的SQL管理工具要轻量，而且支持中文界面。DBeaver社区版作为一个免费开源的产品，和其他类似的软件相比，在功能和易用性上都毫不逊色。

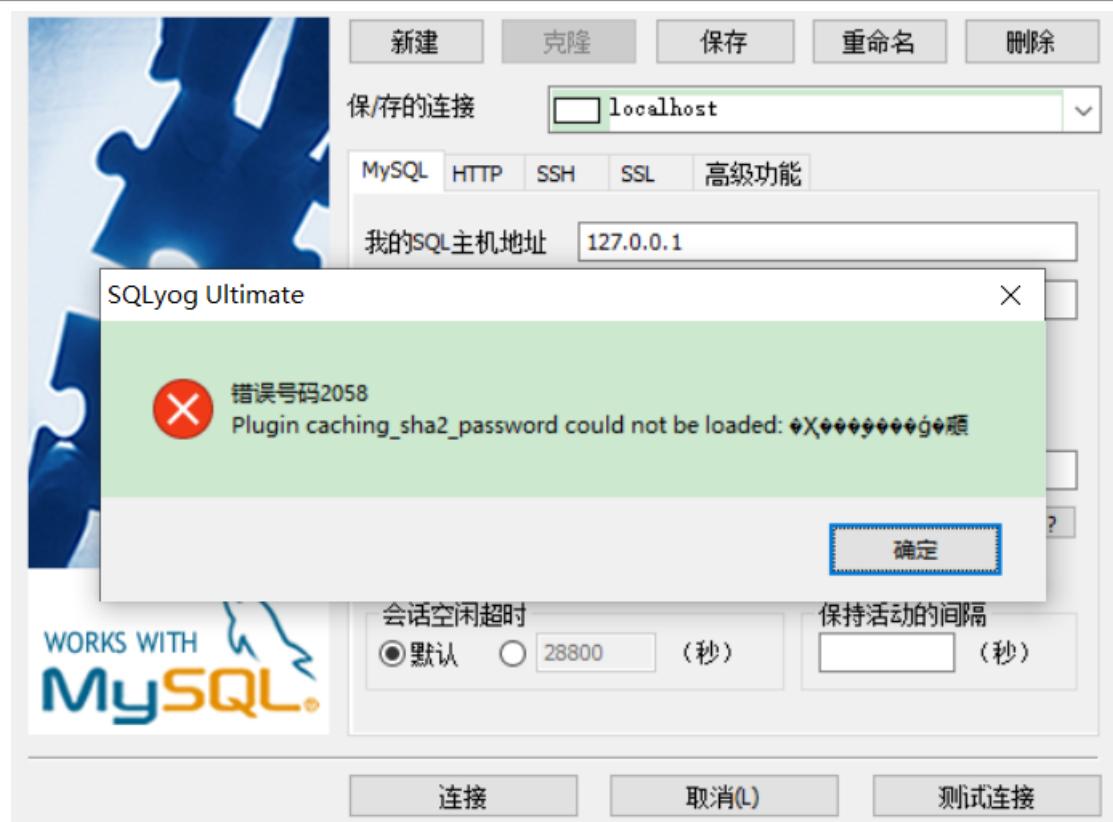
下载地址: <https://dbeaver.io/download/>





可能出现连接问题：

有些图形界面工具，特别是旧版本的图形界面工具，在连接MySQL8时出现“Authentication plugin 'caching_sha2_password' cannot be loaded”错误。



出现这个原因是MySQL8之前的版本中加密规则是mysql_native_password，而在MySQL8之后，加密规则是caching_sha2_password。解决问题方法有两种，第一种是升级图形界面工具版本，第二种是把MySQL8用户登录密码加密规则还原成mysql_native_password。

第二种解决方案如下，用命令行登录MySQL数据库之后，执行如下命令修改用户密码加密规则并更新用户名，这里修改用户名为“root@localhost”的用户密码规则为“mysql_native_password”，密码值为“123456”，如图所示。

```
#使用mysql数据库
USE mysql;

#修改'root'@'localhost'用户的密码规则和密码
ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password BY 'abc123';

#刷新权限
FLUSH PRIVILEGES;
```

```
(c) 2019 Microsoft Corporation。保留所有权利。  
C:\Users\songhk>mysql -uroot -p  
Enter password: *****  
Welcome to the MySQL monitor. Commands end with ; or \g.  
Your MySQL connection id is 19  
Server version: 8.0.26 MySQL Community Server - GPL  
  
Copyright (c) 2000, 2021, Oracle and/or its affiliates.  
  
Oracle is a registered trademark of Oracle Corporation and/or its  
affiliates. Other names may be trademarks of their respective  
owners.  
  
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.  
  
mysql> use mysql;  
Database changed  
mysql> ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password BY 'abc123';  
Query OK, 0 rows affected (0.01 sec)  
  
mysql> FLUSH PRIVILEGES;  
Query OK, 0 rows affected (0.01 sec)  
  
mysql>
```

6. MySQL目录结构与源码

6.1 主要目录结构

MySQL的目录结构	说明
bin目录	所有MySQL的可执行文件。如：mysql.exe
MySQLInstanceConfig.exe	数据库的配置向导，在安装时出现的内容
data目录	系统数据库所在的目录
my.ini文件	MySQL的主要配置文件
c:\ProgramData\MySQL\MySQL Server 8.0\data\	用户创建的数据库所在的目录

6.2 MySQL 源代码获取

首先，你要进入 MySQL下载界面。这里你不要选择用默认的“Microsoft Windows”，而是要通过下拉栏，找到“Source Code”，在下面的操作系统版本里面，选择 Windows (Architecture Independent)，然后点击下载。

接下来，把下载下来的压缩文件解压，我们就得到了 MySQL 的源代码。

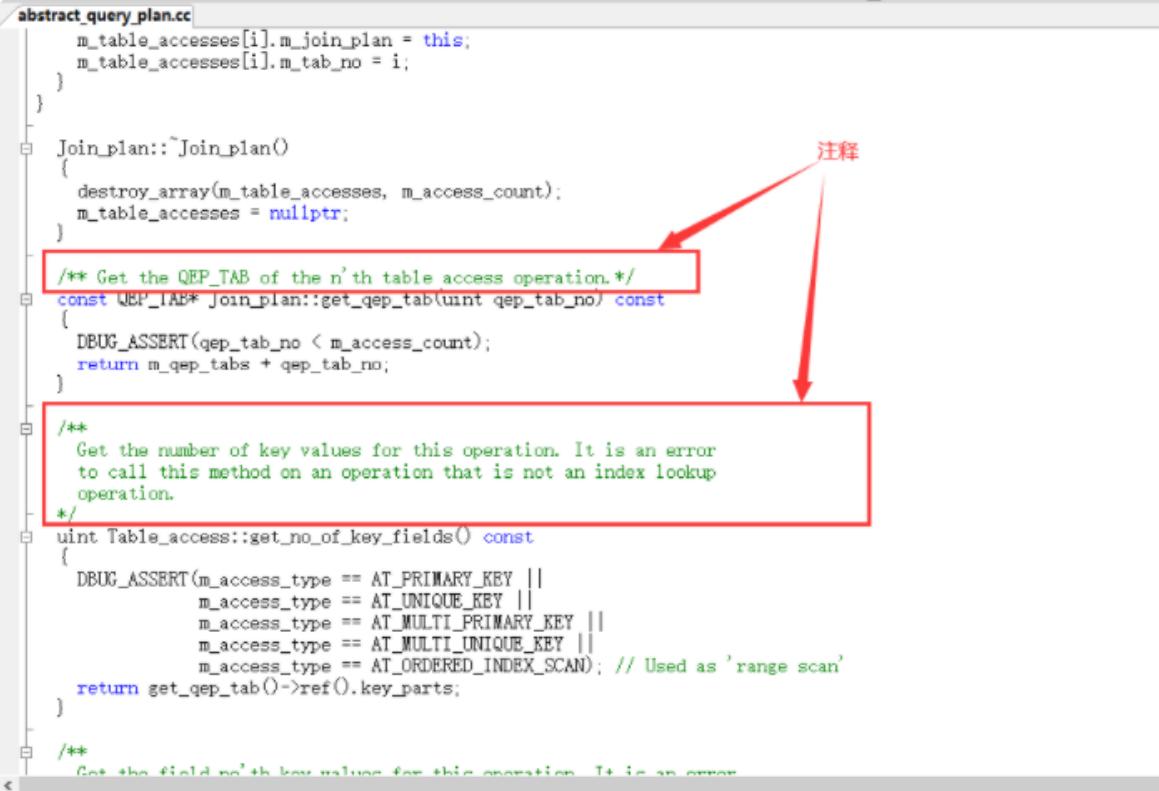
MySQL 是用 C++ 开发而成的，我简单介绍一下源代码的组成。

mysql-8.0.22 目录下的各个子目录，包含了 MySQL 各部分组件的源代码：

client	2020-09-23 15:04	文件夹
cmake	2020-09-23 15:04	文件夹
components	2020-09-23 15:04	文件夹
Docs	2020-09-23 15:04	文件夹
doxygen_resources	2020-09-23 15:04	文件夹
extra	2020-09-23 15:04	文件夹
include	2020-09-23 15:04	文件夹
libbinlogevents	2020-09-23 15:04	文件夹
libbinlogstandalone	2020-09-23 15:04	文件夹
libmysql	2020-09-23 15:04	文件夹
libservices	2020-09-23 15:04	文件夹
man	2020-09-23 15:04	文件夹
mysql-test	2020-09-23 15:04	文件夹
mysys	2020-09-23 15:04	文件夹
packaging	2020-09-23 15:04	文件夹
plugin	2020-09-23 15:04	文件夹
router	2020-09-23 15:04	文件夹
scripts	2020-09-23 15:04	文件夹
share	2020-09-23 15:04	文件夹
source_downloads	2020-09-23 15:04	文件夹
sql	2020-09-23 15:05	文件夹

- sql 子目录是 MySQL 核心代码；
- libmysql 子目录是客户端程序 API；
- mysql-test 子目录是测试工具；
- mysys 子目录是操作系统相关函数和辅助函数；

源代码可以用记事本打开查看，如果你有 C++ 的开发环境，也可以在开发环境中打开查看。



```

abstract_query_plan.cc
{
    m_table_accesses[i].m_join_plan = this;
    m_table_accesses[i].m_tab_no = i;
}

Join_plan::~Join_plan()
{
    destroy_array(m_table_accesses, m_access_count);
    m_table_accesses = nullptr;
}

/** Get the QEP_TAB of the n'th table access operation */
const QEP_TAB* Join_plan::get_qep_tab(uint qep_tab_no) const
{
    DBUG_ASSERT(qep_tab_no < m_access_count);
    return m_qep_tabs + qep_tab_no;
}

/**
 * Get the number of key values for this operation. It is an error
 * to call this method on an operation that is not an index lookup
 * operation.
 */
uint Table_access::get_no_of_key_fields() const
{
    DBUG_ASSERT(m_access_type == AT_PRIMARY_KEY ||
                m_access_type == AT_UNIQUE_KEY ||
                m_access_type == AT_MULTI_PRIMARY_KEY ||
                m_access_type == AT_MULTI_UNIQUE_KEY ||
                m_access_type == AT_ORDERED_INDEX_SCAN); // Used as 'range scan'
    return get_qep_tab()->ref().key_parts;
}

/**
 * Get the field no'th key value for this operation. It is an error
 */

```

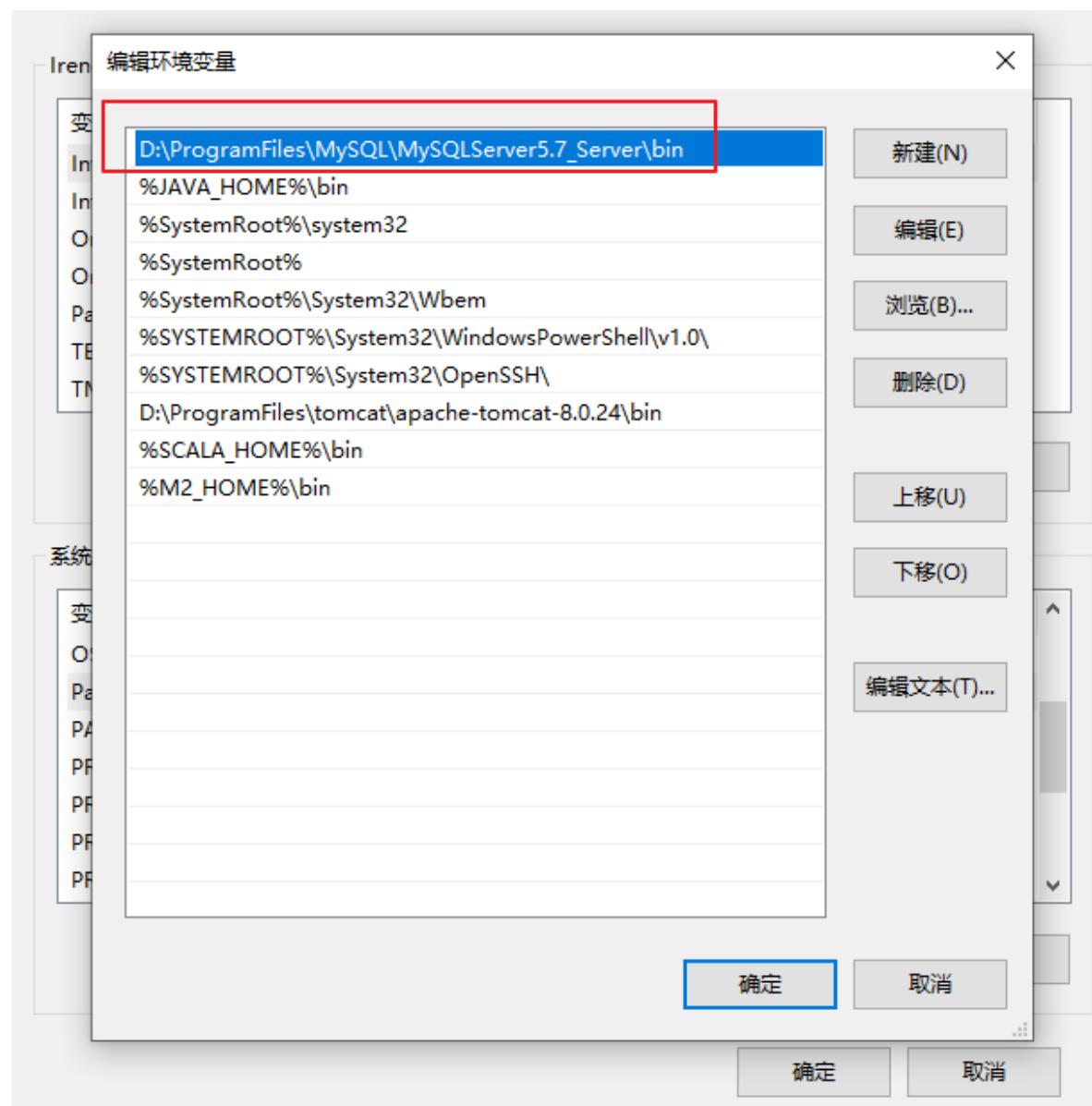
如上图所示，源代码并不神秘，就是普通的 C++ 代码，跟你熟悉的一样，而且有很多注释，可以帮助你理解。阅读源代码就像在跟 MySQL 的开发人员对话一样，十分有趣。

问题1：root用户密码忘记，重置的操作

- 1: 通过任务管理器或者服务管理，关掉mysqld(服务进程)
- 2: 通过命令行+特殊参数开启mysqld mysql --defaults-file="D:\ProgramFiles\mysql\MySQLServer5.7Data\my.ini" --skip-grant-tables
- 3: 此时，mysqld服务进程已经打开。并且不需要权限检查
- 4: mysql -uroot 无密码登陆服务器。另启动一个客户端进行
- 5: 修改权限表 (1) use mysql; (2) update user set authentication_string=password('新密码') where user='root' and Host='localhost'; (3) flush privileges;
- 6: 通过任务管理器，关掉mysqld服务进程。
- 7: 再次通过服务管理，打开mysql服务。
- 8: 即可用修改后的密码登陆。

问题2：mysql命令报“不是内部或外部命令”

如果输入mysql命令报“不是内部或外部命令”，把mysql安装目录的bin目录配置到环境变量path中。如下：



ERROR 1046 (3D000): No database selected

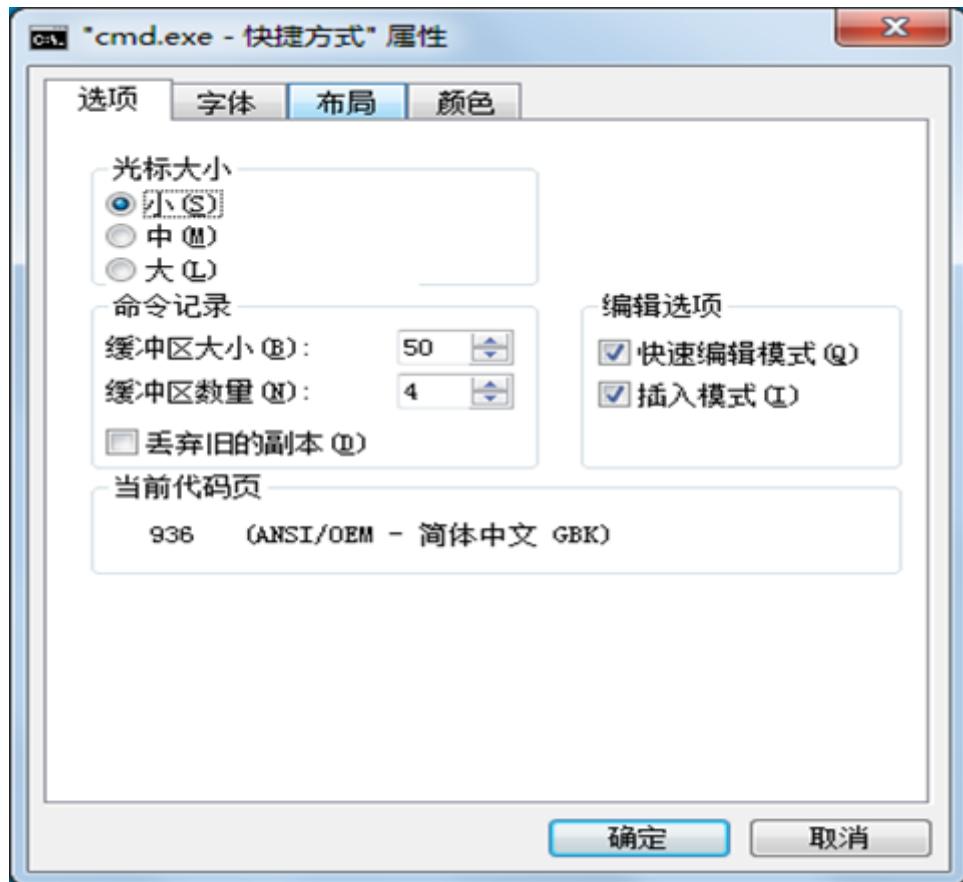
解决方案一：就是使用“USE 数据库名;”语句，这样接下来的语句就默认针对这个数据库进行操作

解决方案二：就是所有的表对象前面都加上“数据库.”

问题4：命令行客户端的字符集问题

```
mysql> INSERT INTO t_stu VALUES(1, '张三', '男');
ERROR 1366 (HY000): Incorrect string value: '\xD5\xC5\xC8\xFD' for column 'sname' at
row 1
```

原因：服务器端认为你的客户端的字符集是utf-8，而实际上你的客户端的字符集是GBK。



查看所有字符集：**SHOW VARIABLES LIKE 'character_set_%';**

Variable_name	Value
character_set_client	utf8
character_set_connection	utf8
character_set_database	utf8
character_set_filesystem	binary
character_set_results	utf8
character_set_server	utf8
character_set_system	utf8
character_sets_dir	D:\ProgramFiles\MySQL Server 5.5\share\charsets\

8 rows in set (0.00 sec)

解决方案，设置当前连接的客户端字符集 “SET NAMES GBK;”

```
管理员: cmd.exe - 快捷方式 - mysql -uroot -p
```

```
mysql> set names gbk;
Query OK, 0 rows affected (0.00 sec)

mysql> SHOW VARIABLES LIKE 'character_set_%';
+-----+-----+
| Variable_name      | Value   |
+-----+-----+
| character_set_client | gbk     |
| character_set_connection | gbk    |
| character_set_database | utf8   |
| character_set_filesystem | binary |
| character_set_results | gbk    |
| character_set_server | utf8   |
| character_set_system | utf8   |
| character_sets_dir | D:\ProgramFiles\MySQL Server 5.5\share\charsets\ |
+-----+-----+
8 rows in set (0.00 sec)
```

问题5：修改数据库和表的字符编码

修改编码：

(1)先停止服务， (2) 修改my.ini文件 (3) 重新启动服务

说明：

如果是在修改my.ini之前建的库和表，那么库和表的编码还是原来的Latin1，要么删了重建，要么使用alter语句修改编码。

```
mysql> create database 0728db charset Latin1;
Query OK, 1 row affected (0.00 sec)
```

```
mysql> use 0728db;
Database changed
```

```
mysql> show create table student\G
***** 1. row *****
      Table: student
Create Table: CREATE TABLE `student` (
  `id` int(11) NOT NULL,
  `name` varchar(20) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
1 row in set (0.00 sec)
```

```
mysql> alter table student charset utf8; #修改表字符编码为UTF8
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
mysql> show create table student\G
***** 1. row *****
      Table: student
Create Table: CREATE TABLE `student` (
  `id` int(11) NOT NULL,
  `name` varchar(20) CHARACTER SET latin1 DEFAULT NULL,  #字段仍然是latin1编码
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.00 sec)
```

```
mysql> alter table student modify name varchar(20) charset utf8; #修改字段字符编码为UTF8
Query OK, 0 rows affected (0.05 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
mysql> show create table student\G
***** 1. row *****
      Table: student
Create Table: CREATE TABLE `student` (
  `id` int(11) NOT NULL,
  `name` varchar(20) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.00 sec)
```

```
mysql> show create database 0728db;;
+-----+-----+
|Database| Create Database |
+-----+-----+
|0728db| CREATE DATABASE `0728db` /*!40100 DEFAULT CHARACTER SET latin1 */ |
+-----+-----+
1 row in set (0.00 sec)
```

```
mysql> alter database 0728db charset utf8; #修改数据库的字符编码为utf8
Query OK, 1 row affected (0.00 sec)
```



```
+-----+-----+
|Database| Create Database
+-----+-----+
| 0728db | CREATE DATABASE `0728db` /*!40100 DEFAULT CHARACTER SET utf8 */ |
+-----+-----+
1 row in set (0.00 sec)
```

第03章_基本的SELECT语句

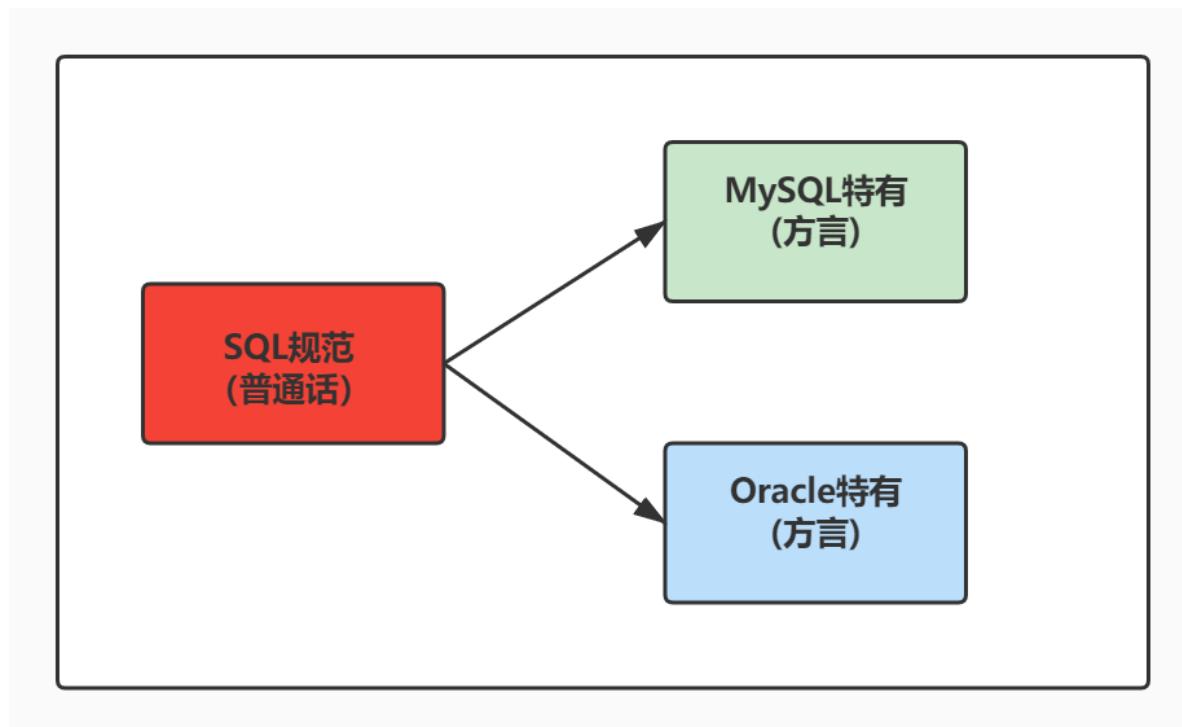
讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

1. SQL概述

1.1 SQL背景知识

- 1946年，世界上第一台电脑诞生，如今，借由这台电脑发展起来的互联网已经自成江湖。在这几十年里，无数的技术、产业在这片江湖里沉浮，有的方兴未艾，有的已经几幕兴衰。但在这片浩荡的波动里，有一门技术从未消失，甚至“老当益壮”，那就是SQL。
 - 45年前，也就是1974年，IBM研究员发布了一篇揭开数据库技术的论文《SEQUEL：一门结构化的英语查询语言》，直到今天这门结构化的查询语言并没有太大的变化，相比于其他语言，SQL的半衰期可以说是非常长了。
- 不论是前端工程师，还是后端算法工程师，都一定会和数据打交道，都需要了解如何又快又准确地提取自己想要的数据。更别提数据分析师了，他们的工作就是和数据打交道，整理不同的报告，以便指导业务决策。
- SQL（Structured Query Language，结构化查询语言）是使用关系模型的数据库应用语言，与数据直接打交道，由IBM上世纪70年代开发出来。后由美国国家标准局（ANSI）开始着手制定SQL标准，先后有SQL-86，SQL-89，SQL-92，SQL-99等标准。
 - SQL有两个重要的标准，分别是SQL92和SQL99，它们分别代表了92年和99年颁布的SQL标准，我们今天使用的SQL语言依然遵循这些标准。
- 不同的数据库生产厂商都支持SQL语句，但都有特有内容。



自从 SQL 加入了 TIOBE 编程语言排行榜，就一直保持在 Top 10。

Oct 2021	Oct 2020	Change	Programming Language	Ratings	Change
1	3	▲	 Python	11.27%	-0.00%
2	1	▼	 C	11.16%	-5.79%
3	2	▼	 Java	10.46%	-2.11%
4	4		 C++	7.50%	+0.57%
5	5		 C#	5.26%	+1.10%
6	6		 Visual Basic	5.24%	+1.27%
7	7		 JavaScript	2.19%	+0.05%
8	10	▲	 SQL	2.17%	+0.61%
9	8	▼	 PHP	2.10%	+0.01%
10	17	▲	 Assembly language	2.06%	+0.99%
11	19	▲	 Classic Visual Basic	1.83%	+1.06%
12	14	▲	 Go	1.28%	+0.13%
13	15	▲	 MATLAB	1.20%	+0.08%
14	9	▼	 R	1.20%	-0.79%
15	12	▼	 Groovy	1.18%	-0.05%

1.3 SQL 分类

SQL语言在功能上主要分为如下3大类：

- **DDL (Data Definition Languages、数据定义语言)**，这些语句定义了不同的数据库、表、视图、索引等数据库对象，还可以用来创建、删除、修改数据库和数据表的结构。
 - 主要的语句关键字包括 `CREATE`、`DROP`、`ALTER` 等。
- **DML (Data Manipulation Language、数据操作语言)**，用于添加、删除、更新和查询数据库记录，并检查数据完整性。
 - 主要的语句关键字包括 `INSERT`、`DELETE`、`UPDATE`、`SELECT` 等。
 - **SELECT是SQL语言的基础，最为重要。**
- **DCL (Data Control Language、数据控制语言)**，用于定义数据库、表、字段、用户的访问权限和安全级别。
 - 主要的语句关键字包括 `GRANT`、`REVOKE`、`COMMIT`、`ROLLBACK`、`SAVEPOINT` 等。

因为查询语句使用的非常的频繁，所以很多人把查询语句单拎出来一类：DQL (数据查询语言)。

还有单独将 `COMMIT`、`ROLLBACK` 取出来称为TCL (Transaction Control Language, 事务控制语言)。

2. SQL语言的规则与规范

- SQL 可以写在一行或者多行。为了提高可读性，各子句分行写，必要时使用缩进
- 每条命令以 ; 或 \g 或 \G 结束
- 关键字不能被缩写也不能分行
- 关于标点符号
 - 必须保证所有的()、单引号、双引号是成对结束的
 - 必须使用英文状态下的半角输入方式
 - 字符串型和日期时间类型的数据可以使用单引号 ('') 表示
 - 列的别名，尽量使用双引号 (" ")，而且不建议省略as

2.2 SQL大小写规范（建议遵守）

- MySQL 在 Windows 环境下是大小写不敏感的
- MySQL 在 Linux 环境下是大小写敏感的
 - 数据库名、表名、表的别名、变量名是严格区分大小写的
 - 关键字、函数名、列名(或字段名)、列的别名(字段的别名)是忽略大小写的。
- 推荐采用统一的书写规范：
 - 数据库名、表名、表别名、字段名、字段别名等都小写
 - SQL 关键字、函数名、绑定变量等都大写

2.3 注释

可以使用如下格式的注释结构

```
单行注释: #注释文字 (MySQL特有的方式)
单行注释: -- 注释文字 (--后面必须包含一个空格。)
多行注释: /* 注释文字 */
```

2.4 命名规则（暂时了解）

- 数据库、表名不得超过30个字符，变量名限制为29个
- 必须只能包含 A-Z, a-z, 0-9, _ 共63个字符
- 数据库名、表名、字段名等对象名中间不要包含空格
- 同一个MySQL软件中，数据库不能同名；同一个库中，表不能重名；同一个表中，字段不能重名
- 必须保证你的字段没有和保留字、数据库系统或常用方法冲突。如果坚持使用，请在SQL语句中使用`（着重号）引起来
- 保持字段名和类型的一致性，在命名字段并为其指定数据类型的时候一定要保证一致性。假如数据类型在一个表里是整数，那在另一个表里可就别变成字符串了

举例：

```
#以下两句是一样的，不区分大小写
show databases;
SHOW DATABASES;

#创建表格
#create table student info(...); #表名错误，因为表名有空格
create table student_info(...);

#其中order使用``飘号，因为order和系统关键字或系统函数名等预定义标识符重名了
CREATE TABLE `order`(`
```

```
);

select id as "编号", `name` as "姓名" from t_stu; #起别名时, as都可以省略
select id as 编号, `name` as 姓名 from t_stu; #如果字段别名中没有空格, 那么可以省略 ""
select id as 编号, `name` as 姓名 from t_stu; #错误, 如果字段别名中有空格, 那么不能省略 ""
```

2.5 数据导入指令

在命令行客户端登录mysql, 使用source指令导入

```
mysql> source d:\mysqldb.sql
```

```
mysql> desc employees;
+-----+-----+-----+-----+-----+
| Field      | Type       | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| employee_id | int(6)     | NO   | PRI | 0        |          |
| first_name  | varchar(20) | YES  |      | NULL    |          |
| last_name   | varchar(25) | NO   |      | NULL    |          |
| email       | varchar(25) | NO   | UNI | NULL    |          |
| phone_number | varchar(20) | YES  |      | NULL    |          |
| hire_date   | date        | NO   |      | NULL    |          |
| job_id      | varchar(10) | NO   | MUL | NULL    |          |
| salary       | double(8,2)  | YES  |      | NULL    |          |
| commission_pct | double(2,2) | YES  |      | NULL    |          |
| manager_id  | int(6)     | YES  | MUL | NULL    |          |
| department_id | int(4)     | YES  | MUL | NULL    |          |
+-----+-----+-----+-----+-----+
11 rows in set (0.00 sec)
```

3. 基本的SELECT语句

3.0 SELECT...

```
SELECT 1; #没有任何子句
SELECT 9/2; #没有任何子句
```

3.1 SELECT ... FROM

- 语法:

```
SELECT 标识选择哪些列
FROM 标识从哪个表中选择
```

- 选择全部列:

```
SELECT *
FROM departments;
```

20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500
90	Executive	100	1700
110	Accounting	205	1700
190	Contracting		1700

一般情况下，除非需要使用表中所有的字段数据，最好不要使用通配符‘*’。使用通配符虽然可以节省输入查询语句的时间，但是获取不需要的列数据通常会降低查询和所使用的应用程序的效率。通配符的优势是，当不知道所需要的列的名称时，可以通过它获取它们。

在生产环境下，不推荐你直接使用 `SELECT *` 进行查询。

- 选择特定的列：

```
SELECT department_id, location_id
FROM departments;
```

DEPARTMENT_ID	LOCATION_ID
10	1700
20	1800
50	1500
60	1400
80	2500
90	1700
110	1700
190	1700

MySQL中的SQL语句是不区分大小写的，因此SELECT和select的作用是相同的，但是，许多开发人员习惯将关键字大写、数据列和表名小写，读者也应该养成一个良好的编程习惯，这样写出来的代码更容易阅读和维护。

3.2 列的别名

- 重命名一个列
- 便于计算
- 紧跟列名，也可以**在列名和别名之间加入关键字AS，别名使用双引号**，以便在别名中包含空格或特殊的字符并区分大小写。
- AS 可以省略
- 建议别名简短，见名知意
- 举例

```
SELECT last_name AS name, commission_pct comm
FROM employees;
```

NAME	COMM
King	
Kochhar	
De Haan	

```
SELECT last_name "Name", salary*12 "Annual Salary"  
FROM employees;
```

Name	Annual Salary
King	288000
Kochhar	204000
De Haan	204000

20 rows selected.

3.3 去除重复行

默认情况下，查询会返回全部行，包括重复行。

```
SELECT department_id  
FROM employees;
```

DEPARTMENT_ID
90
90
90
60
60
60
50
50
50

20 rows selected.

在SELECT语句中使用关键字DISTINCT去除重复行

```
SELECT DISTINCT department_id  
FROM employees;
```

DEPARTMENT_ID
10
20
50
60
80
90
110

8 rows selected.

针对于：

```
SELECT DISTINCT department_id,salary  
FROM employees;
```

这里有两点需要注意：

2. DISTINCT 其实是对后面所有列名的组合进行去重，你能看到最后的结果是 74 条，因为这 74 个部门id不同，都有 salary 这个属性值。如果你想要看都有哪些不同的部门 (department_id)，只需要写 DISTINCT department_id 即可，后面不需要再加其他的列名了。

3.4 空值参与运算

- 所有运算符或列值遇到null值，运算的结果都为null

```
SELECT employee_id,salary,commission_pct,
12 * salary * (1 + commission_pct) "annual_sal"
FROM employees;
```

这里你一定要注意，在 MySQL 里面，空值不等于空字符串。一个空字符串的长度是 0，而一个空值的长度是空。而且，在 MySQL 里面，空值是占用空间的。

3.5 着重号

- 错误的

```
mysql> SELECT * FROM ORDER;
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax to use near 'ORDER' at
line 1
```

- 正确的

```
mysql> SELECT * FROM `ORDER`;
+-----+-----+
| order_id | order_name |
+-----+-----+
|      1 | shkstart   |
|      2 | tomcat     |
|      3 | dubbo      |
+-----+-----+
3 rows in set (0.00 sec)
```

```
mysql> SELECT * FROM `order`;
+-----+-----+
| order_id | order_name |
+-----+-----+
|      1 | shkstart   |
|      2 | tomcat     |
|      3 | dubbo      |
+-----+-----+
3 rows in set (0.00 sec)
```

- 结论

我们需要保证表中的字段、表名等没有和保留字、数据库系统或常用方法冲突。如果真的相同，请在 SQL语句中使用一对``（着重号）引起来。

SELECT 查询还可以对常数进行查询。对的，就是在 SELECT 查询结果中增加一列固定的常数列。这列的取值是我们指定的，而不是从数据表中动态取出的。

你可能会问为什么我们还要对常数进行查询呢？

SQL 中的 SELECT 语法的确提供了这个功能，一般来说我们只从一个表中查询数据，通常不需要增加一个固定的常数列，但如果我们要整合不同的数据源，用常数列作为这个表的标记，就需要查询常数。

比如说，我们想对 employees 数据表中的员工姓名进行查询，同时增加一列字段 corporation，这个字段固定值为“尚硅谷”，可以这样写：

```
SELECT '尚硅谷' as corporation, last_name FROM employees;
```

4. 显示表结构

使用DESCRIBE 或 DESC 命令，表示表结构。

```
DESCRIBE employees;  
或  
DESC employees;
```

```
mysql> desc employees;  
+-----+-----+-----+-----+-----+  
| Field | Type | Null | Key | Default | Extra |  
+-----+-----+-----+-----+-----+  
| employee_id | int(6) | NO | PRI | 0 |  
| first_name | varchar(20) | YES | | NULL |  
| last_name | varchar(25) | NO | | NULL |  
| email | varchar(25) | NO | UNI | NULL |  
| phone_number | varchar(20) | YES | | NULL |  
| hire_date | date | NO | | NULL |  
| job_id | varchar(10) | NO | MUL | NULL |  
| salary | double(8,2) | YES | | NULL |  
| commission_pct | double(2,2) | YES | | NULL |  
| manager_id | int(6) | YES | MUL | NULL |  
| department_id | int(4) | YES | MUL | NULL |  
+-----+-----+-----+-----+-----+  
11 rows in set (0.00 sec)
```

其中，各个字段的含义分别解释如下：

- Field：表示字段名称。
- Type：表示字段类型，这里 barcode、goodsname 是文本型的， price 是整数类型的。
- Null：表示该列是否可以存储NULL值。
- Key：表示该列是否已编制索引。PRI表示该列是表主键的一部分；UNI表示该列是UNIQUE索引的一部分；MUL表示在列中某个给定值允许出现多次。
- Default：表示该列是否有默认值，如果有，那么值是多少。
- Extra：表示可以获取的与给定列有关的附加信息，例如AUTO_INCREMENT等。

- 背景：

EMPLOYEES

EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
100	King	AD_PRES	90
101	Kochhar	AD_VP	90
102	De Haan	AD_VP	90
103	Hunold	IT_PROG	60
104	Ernst	IT_PROG	60
107	Lorentz	IT_PROG	60
124	Mourgos	ST_MAN	50
...			

20 rows selected.

返回在 90 号部门工作的
所有员工的信息



EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
100	King	AD_PRES	90
101	Kochhar	AD_VP	90
102	De Haan	AD_VP	90

- 语法：

```

SELECT 字段1, 字段2
FROM 表名
WHERE 过滤条件
    
```

- 使用 WHERE 子句，将不满足条件的行过滤掉
- **WHERE子句紧随 FROM子句**
- 举例

```

SELECT employee_id, last_name, job_id, department_id
FROM employees
WHERE department_id = 90 ;
    
```

EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
100	King	AD_PRES	90
101	Kochhar	AD_VP	90
102	De Haan	AD_VP	90

第04章_运算符

讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

1. 算术运算符

算术运算符主要用于数学运算，其可以连接运算符前后的两个数值或表达式，对数值或表达式进行加 (+)、减 (-)、乘 (*)、除 (/) 和取模 (%) 运算。

运 算 符	名 称	作 用	示 例
+	加法运算符	计算两个值或表达式的和	SELECT A + B
-	减法运算符	计算两个值或表达式的差	SELECT A - B
*	乘法运算符	计算两个值或表达式的乘积	SELECT A * B
/或DIV	除法运算符	计算两个值或表达式的商	SELECT A / B 或者 SELECT A DIV B
%或MOD	求模（求余）运算符	计算两个值或表达式的余数	SELECT A % B 或者 SELECT A MOD B

1. 加法与减法运算符

```
mysql> SELECT 100, 100 + 0, 100 - 0, 100 + 50, 100 + 50 -30, 100 + 35.5, 100 - 35.5
FROM dual;
+-----+-----+-----+-----+-----+-----+-----+
| 100 | 100 + 0 | 100 - 0 | 100 + 50 | 100 + 50 -30 | 100 + 35.5 | 100 - 35.5 |
+-----+-----+-----+-----+-----+-----+-----+
| 100 |      100 |      100 |      150 |          120 |      135.5 |      64.5 |
+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

由运算结果可以得出如下结论：

- 一个整数类型的值对整数进行加法和减法操作，结果还是一个整数；
- 一个整数类型的值对浮点数进行加法和减法操作，结果是一个浮点数；
- 加法和减法的优先级相同，进行先加后减操作与进行先减后加操作的结果是一样的；
- 在Java中，+的左右两边如果有字符串，那么表示字符串的拼接。但是在MySQL中+只表示数值相加。如果遇到非数值类型，先尝试转成数值，如果转失败，就按0计算。（补充：MySQL中字符串拼接要使用字符串函数CONCAT()实现）

2. 乘法与除法运算符

```
+-----+-----+-----+-----+-----+-----+
| 100 | 100 * 1 | 100 * 1.0 | 100 / 1.0 | 100 / 2 | 100 + 2 * 5 / 2 | 100 / 3 | 100
DIV 0 |
+-----+-----+-----+-----+-----+-----+
| 100 |      100 |     100.0 |   100.000 |   50.0000 |      105.0000 |    33.3333 |
NULL |
+-----+-----+-----+-----+-----+-----+
-----+
1 row in set (0.00 sec)
```

```
#计算出员工的年基本工资
SELECT employee_id,salary,salary * 12 annual_sal
FROM employees;
```

由运算结果可以得出如下结论：

- 一个数乘以整数1和除以整数1后仍得原数；
- 一个数乘以浮点数1和除以浮点数1后变成浮点数，数值与原数相等；
- 一个数除以整数后，不管是否能除尽，结果都为一个浮点数；
- 一个数除以另一个数，除不尽时，结果为一个浮点数，并保留到小数点后4位；
- 乘法和除法的优先级相同，进行先乘后除操作与先除后乘操作，得出的结果相同。
- 在数学运算中，0不能用作除数，在MySQL中，一个数除以0为NULL。

3. 求模（求余）运算符 将t22表中的字段对3和5进行求模（求余）运算。

```
mysql> SELECT 12 % 3, 12 MOD 5 FROM dual;
+-----+-----+
| 12 % 3 | 12 MOD 5 |
+-----+-----+
|      0 |        2 |
+-----+-----+
1 row in set (0.00 sec)
```

```
#筛选出employee_id是偶数的员工
SELECT * FROM employees
WHERE employee_id MOD 2 = 0;
```

可以看到，100对3求模后的结果为3，对5求模后的结果为0。

2. 比较运算符

比较运算符用来对表达式左边的操作数和右边的操作数进行比较，比较的结果为真则返回1，比较的结果为假则返回0，其他情况则返回NULL。

比较运算符经常被用来作为SELECT查询语句的条件来使用，返回符合条件的结果记录。

=	等于运算符	判断两个值、字符串或表达式是否相等	<code>SELECT C FROM TABLE WHERE A = B</code>
<code><=</code>	安全等于运算符	安全地判断两个值、字符串或表达式是否相等	<code>SELECT C FROM TABLE WHERE A <= B</code>
<code><>(!=)</code>	不等于运算符	判断两个值、字符串或表达式是否不相等	<code>SELECT C FROM TABLE WHERE A <> B</code> <code>SELECT C FROM TABLE WHERE A != B</code>
<	小于运算符	判断前面的值、字符串或表达式是否小于后面的值、字符串或表达式	<code>SELECT C FROM TABLE WHERE A < B</code>
<code><=</code>	小于等于运算符	判断前面的值、字符串或表达式是否小于等于后面的值、字符串或表达式	<code>SELECT C FROM TABLE WHERE A <= B</code>
>	大于运算符	判断前面的值、字符串或表达式是否大于后面的值、字符串或表达式	<code>SELECT C FROM TABLE WHERE A > B</code>
<code>>=</code>	大于等于运算符	判断前面的值、字符串或表达式是否大于等于后面的值、字符串或表达式	<code>SELECT C FROM TABLE WHERE A >= B</code>

1. 等号运算符

- 等号运算符 (=) 判断等号两边的值、字符串或表达式是否相等，如果相等则返回1，不相等则返回0。
- 在使用等号运算符时，遵循如下规则：
 - 如果等号两边的值、字符串或表达式都为字符串，则MySQL会按照字符串进行比较，其比较的是每个字符串中字符的ANSI编码是否相等。
 - 如果等号两边的值都是整数，则MySQL会按照整数来比较两个值的大小。
 - 如果等号两边的值一个是整数，另一个是字符串，则MySQL会将字符串转化为数字进行比较。
 - 如果等号两边的值、字符串或表达式中有一个为NULL，则比较结果为NULL。
- 对比：SQL中赋值符号使用 :=

```
mysql> SELECT 1 = 1, 1 = '1', 1 = 0, 'a' = 'a', (5 + 3) = (2 + 6), '' = NULL, NULL = NULL;
+-----+-----+-----+-----+-----+-----+
| 1 = 1 | 1 = '1' | 1 = 0 | 'a' = 'a' | (5 + 3) = (2 + 6) | '' = NULL | NULL = NULL |
+-----+-----+-----+-----+-----+-----+
|     1 |        1 |     0 |       1 |          1 |      NULL |      NULL |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT 1 = 2, 0 = 'abc', 1 = 'abc' FROM dual;
+-----+-----+
| 1 = 2 | 0 = 'abc' | 1 = 'abc' |
+-----+-----+
|     0 |        1 |        0 |
+-----+-----+
1 row in set, 2 warnings (0.00 sec)
```

```
#查询salary=10000，注意在Java中是比较是==
SELECT employee_id,salary FROM employees WHERE salary = 10000;
```

时，其返回值为0，而不为NULL。

```
mysql> SELECT 1 <=> '1', 1 <=> 0, 'a' <=> 'a', (5 + 3) <=> (2 + 6), '' <=> NULL, NULL <=> NULL FROM dual;
+-----+-----+-----+-----+
| 1 <=> '1' | 1 <=> 0 | 'a' <=> 'a' | (5 + 3) <=> (2 + 6) | '' <=> NULL | NULL <=> NULL |
+-----+-----+-----+-----+
|       1 |      0 |       1 |      1 |      0 |
1 |
+-----+-----+-----+-----+
--+
1 row in set (0.00 sec)
```

```
#查询commission_pct等于0.40
SELECT employee_id,commission_pct FROM employees WHERE commission_pct = 0.40;

SELECT employee_id,commission_pct FROM employees WHERE commission_pct <=> 0.40;

#如果把0.40改成 NULL 呢?
```

可以看到，使用安全等于运算符时，两边的操作数的值都为NULL时，返回的结果为1而不是NULL，其他返回结果与等于运算符相同。

3. 不等于运算符 不等于运算符 (<>和!=) 用于判断两边的数字、字符串或者表达式的值是否不相等，如果不相等则返回1，相等则返回0。不等于运算符不能判断NULL值。如果两边的值有任意一个为NULL，或两边都为NULL，则结果为NULL。SQL语句示例如下：

```
mysql> SELECT 1 <> 1, 1 != 2, 'a' != 'b', (3+4) <> (2+6), 'a' != NULL, NULL <> NULL;
+-----+-----+-----+-----+
| 1 <> 1 | 1 != 2 | 'a' != 'b' | (3+4) <> (2+6) | 'a' != NULL | NULL <> NULL |
+-----+-----+-----+-----+
|      0 |      1 |      1 |      1 |      NULL |      NULL |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

此外，还有非符号类型的运算符：

运 算 符	名 称	作 用	示 例
-------	-----	-----	-----

IS NOT NULL	不为空运算符	判断值、字符串或表达式是否不为空	SELECT B FROM TABLE WHERE A IS NOT NULL
LEAST	最小值运算符	在多个值中返回最小值	SELECT D FROM TABLE WHERE C LEAST(A, B)
GREATEST	最大值运算符	在多个值中返回最大值	SELECT D FROM TABLE WHERE C GREATEST(A, B)
BETWEEN AND	两值之间的运算符	判断一个值是否在两个值之间	SELECT D FROM TABLE WHERE C BETWEEN A AND B
ISNULL	为空运算符	判断一个值、字符串或表达式是否为空	SELECT B FROM TABLE WHERE A ISNULL
IN	属于运算符	判断一个值是否为列表中的任意一个值	SELECT D FROM TABLE WHERE C IN (A, B)
NOT IN	不属于运算符	判断一个值是否不是一个列表中的任意一个值	SELECT D FROM TABLE WHERE C NOT IN (A, B)
LIKE	模糊匹配运算符	判断一个值是否符合模糊匹配规则	SELECT C FROM TABLE WHERE A LIKE B
REGEXP	正则表达式运算符	判断一个值是否符合正则表达式的规则	SELECT C FROM TABLE WHERE A REGEXP B
RLIKE	正则表达式运算符	判断一个值是否符合正则表达式的规则	SELECT C FROM TABLE WHERE A RLIKE B

4. 空运算符 空运算符 (IS NULL或者ISNULL) 判断一个值是否为NULL, 如果为NULL则返回1, 否则返回0。 SQL语句示例如下:

```
mysql> SELECT NULL IS NULL, ISNULL(NULL), ISNULL('a'), 1 IS NULL;
+-----+-----+-----+-----+
| NULL IS NULL | ISNULL(NULL) | ISNULL('a') | 1 IS NULL |
+-----+-----+-----+-----+
|          1 |           1 |          0 |          0 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

```
#查询commission_pct等于NULL。比较如下的四种写法
SELECT employee_id,commission_pct FROM employees WHERE commission_pct IS NULL;
SELECT employee_id,commission_pct FROM employees WHERE commission_pct <=> NULL;
SELECT employee_id,commission_pct FROM employees WHERE ISNULL(commission_pct);
SELECT employee_id,commission_pct FROM employees WHERE commission_pct = NULL;
```

```
SELECT last_name, manager_id
FROM employees
WHERE manager_id IS NULL;
```

5. 非空运算符 非空运算符 (IS NOT NULL) 判断一个值是否不为NULL, 如果不为NULL则返回1, 否则返回0。 SQL语句示例如下:

```
mysql> SELECT NULL IS NOT NULL, 'a' IS NOT NULL, 1 IS NOT NULL;
+-----+-----+-----+
| NULL IS NOT NULL | 'a' IS NOT NULL | 1 IS NOT NULL |
+-----+-----+-----+
|          0 |           1 |          1 |
+-----+-----+-----+
1 row in set (0.01 sec)
```

```
SELECT employee_id,commission_pct FROM employees WHERE NOT commission_pct <=> NULL;
SELECT employee_id,commission_pct FROM employees WHERE NOT ISNULL(commission_pct);
```

6. 最小值运算符 语法规则为：LEAST(值1, 值2, ..., 值n)。其中，“值n”表示参数列表中有n个值。在有两个或多个参数的情况下，返回最小值。

```
mysql> SELECT LEAST (1,0,2), LEAST('b','a','c'), LEAST(1,NULL,2);
+-----+-----+-----+
| LEAST (1,0,2) | LEAST('b','a','c') | LEAST(1,NULL,2) |
+-----+-----+-----+
|      0       |      a       |      NULL      |
+-----+-----+-----+
1 row in set (0.00 sec)
```

由结果可以看到，当参数是整数或者浮点数时，LEAST将返回其中最小的值；当参数为字符串时，返回字母表中顺序最靠前的字符；当比较值列表中有NULL时，不能判断大小，返回值为NULL。

7. 最大值运算符 语法规则为：GREATEST(值1, 值2, ..., 值n)。其中，n表示参数列表中有n个值。当有两个或多个参数时，返回值为最大值。假如任意一个自变量为NULL，则GREATEST()的返回值为NULL。

```
mysql> SELECT GREATEST(1,0,2), GREATEST('b','a','c'), GREATEST(1,NULL,2);
+-----+-----+-----+
| GREATEST(1,0,2) | GREATEST('b','a','c') | GREATEST(1,NULL,2) |
+-----+-----+-----+
|          2      |      c       |      NULL      |
+-----+-----+-----+
1 row in set (0.00 sec)
```

由结果可以看到，当参数中是整数或者浮点数时，GREATEST将返回其中最大的值；当参数为字符串时，返回字母表中顺序最靠后的字符；当比较值列表中有NULL时，不能判断大小，返回值为NULL。

8. BETWEEN AND运算符 BETWEEN运算符使用的格式通常为SELECT D FROM TABLE WHERE C BETWEEN A AND B，此时，当C大于或等于A，并且C小于或等于B时，结果为1，否则结果为0。

```
mysql> SELECT 1 BETWEEN 0 AND 1, 10 BETWEEN 11 AND 12, 'b' BETWEEN 'a' AND 'c';
+-----+-----+-----+
| 1 BETWEEN 0 AND 1 | 10 BETWEEN 11 AND 12 | 'b' BETWEEN 'a' AND 'c' |
+-----+-----+-----+
|           1       |          0       |           1        |
+-----+-----+-----+
1 row in set (0.00 sec)
```

```
SELECT last_name, salary
FROM   employees
WHERE  salary BETWEEN 2500 AND 3500;
```

9. IN运算符 IN运算符用于判断给定的值是否是IN列表中的一个值，如果是则返回1，否则返回0。如果给定的值为NULL，或者IN列表中存在NULL，则结果为NULL。

```
mysql> SELECT 'a' IN ('a','b','c'), 1 IN (2,3), NULL IN ('a','b'), 'a' IN ('a', NULL);
+-----+-----+-----+
| 'a' IN ('a','b','c') | 1 IN (2,3) | NULL IN ('a','b') | 'a' IN ('a', NULL) |
+-----+-----+-----+
|           1       |          0       |        NULL      |           1        |
+-----+-----+-----+
1 row in set (0.00 sec)
```

```
WHERE manager_id IN (100, 101, 201);
```

10. NOT IN运算符 NOT IN运算符用于判断给定的值是否不是IN列表中的一个值，如果不是IN列表中的一个值，则返回1，否则返回0。

```
mysql> SELECT 'a' NOT IN ('a','b','c'), 1 NOT IN (2,3);
+-----+-----+
| 'a' NOT IN ('a','b','c') | 1 NOT IN (2,3) |
+-----+-----+
| 0 | 1 |
+-----+-----+
1 row in set (0.00 sec)
```

11. LIKE运算符 LIKE运算符主要用来匹配字符串，通常用于模糊匹配，如果满足条件则返回1，否则返回0。如果给定的值或者匹配条件为NULL，则返回结果为NULL。

LKE运算符通常使用如下通配符：

```
“%”：匹配0个或多个字符。  
“_”：只能匹配一个字符。
```

SQL语句示例如下：

```
mysql> SELECT NULL LIKE 'abc', 'abc' LIKE NULL;
+-----+-----+
| NULL LIKE 'abc' | 'abc' LIKE NULL |
+-----+-----+
| NULL | NULL |
+-----+-----+
1 row in set (0.00 sec)
```

```
SELECT first_name
FROM employees
WHERE first_name LIKE 'S%';
```

```
SELECT last_name
FROM employees
WHERE last_name LIKE '_o%';
```

ESCAPE

- 回避特殊符号的：**使用转义符**。例如：将[%]转为[\$%]、[]转为[\$]，然后再加上[ESCAPE'\$']即可。

```
SELECT job_id
FROM jobs
WHERE job_id LIKE 'IT\_%';
```

- 如果使用\表示转义，要省略ESCAPE。如果不是\，则要加上ESCAPE。

```
SELECT job_id
FROM jobs
WHERE job_id LIKE 'IT$_%' escape '$';
```

12. REGEXP运算符

REGEXP运算符用来匹配字符串，语法格式为：expr REGEXP 匹配条件。如果expr满足匹配条件，返回

- (1) '^' 匹配以该字符后面的字符开头的字符串。
- (2) '\$' 匹配以该字符前面的字符结尾的字符串。
- (3) '.' 匹配任何一个单字符。
- (4) "[...]" 匹配在方括号内的任何字符。例如，"[abc]" 匹配 "a" 或 "b" 或 "c"。为了命名字符的范围，使用一个 '-'。"[a-z]" 匹配任何字母，而 "[0-9]" 匹配任何数字。
- (5) '*' 匹配零个或多个在它前面的字符。例如，"x*" 匹配任何数量的 'x' 字符，"[0-9]*" 匹配任何数量的数字，而 "*" 匹配任何数量的任何字符。

SQL语句示例如下：

```
mysql> SELECT 'shkstart' REGEXP '^s', 'shkstart' REGEXP 't$', 'shkstart' REGEXP 'hk';
+-----+-----+-----+
| 'shkstart' REGEXP '^s' | 'shkstart' REGEXP 't$' | 'shkstart' REGEXP 'hk' |
+-----+-----+-----+
|           1 |           1 |           1 |
+-----+-----+-----+
1 row in set (0.01 sec)
```

```
mysql> SELECT 'atguigu' REGEXP 'gu.gu', 'atguigu' REGEXP '[ab]';
+-----+-----+
| 'atguigu' REGEXP 'gu.gu' | 'atguigu' REGEXP '[ab]' |
+-----+-----+
|           1 |           1 |
+-----+-----+
1 row in set (0.00 sec)
```

3. 逻辑运算符

逻辑运算符主要用来判断表达式的真假，在MySQL中，逻辑运算符的返回结果为1、0或者NULL。

MySQL中支持4种逻辑运算符如下：

运 算 符	作 用	示 例
NOT 或 !	逻辑非	SELECT NOT A
AND 或 &&	逻辑与	SELECT A AND B SELECT A && B
OR 或	逻辑或	SELECT A OR B SELECT A B
XOR	逻辑异或	SELECT A XOR B

1. 逻辑非运算符 逻辑非 (NOT或!) 运算符表示当给定的值为0时返回1；当给定的值为非0值时返回0；当给定的值为NULL时，返回NULL。

```
mysql> SELECT NOT 1, NOT 0, NOT(1+1), NOT !1, NOT NULL;
+-----+-----+-----+-----+
| NOT 1 | NOT 0 | NOT(1+1) | NOT !1 | NOT NULL |
+-----+-----+-----+-----+
|     0 |     1 |      0 |      1 |      NULL |
+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

```
WHERE job_id NOT IN ('IT_PROG', 'ST_CLERK', 'SA_REP');
```

- 2. 逻辑与运算符** 逻辑与 (AND或&&) 运算符是当给定的所有值均为非0值，并且都不为NULL时，返回1；当给定的一个值或者多个值为0时则返回0；否则返回NULL。

```
mysql> SELECT 1 AND -1, 0 AND 1, 0 AND NULL, 1 AND NULL;
+-----+-----+-----+
| 1 AND -1 | 0 AND 1 | 0 AND NULL | 1 AND NULL |
+-----+-----+-----+
|      1 |      0 |          0 |      NULL |
+-----+-----+-----+
1 row in set (0.00 sec)
```

```
SELECT employee_id, last_name, job_id, salary
FROM employees
WHERE salary >=10000
AND job_id LIKE '%MAN%';
```

- 3. 逻辑或运算符** 逻辑或 (OR或||) 运算符是当给定的值都不为NULL，并且任何一个值为非0值时，则返回1，否则返回0；当一个值为NULL，并且另一个值为非0值时，返回1，否则返回NULL；当两个值都为NULL时，返回NULL。

```
mysql> SELECT 1 OR -1, 1 OR 0, 1 OR NULL, 0 || NULL, NULL || NULL;
+-----+-----+-----+-----+
| 1 OR -1 | 1 OR 0 | 1 OR NULL | 0 || NULL | NULL || NULL |
+-----+-----+-----+-----+
|      1 |      1 |          1 |      NULL |      NULL |
+-----+-----+-----+-----+
1 row in set, 2 warnings (0.00 sec)
```

```
#查询基本薪资不在9000-12000之间的员工编号和基本薪资
SELECT employee_id,salary FROM employees
WHERE NOT (salary >= 9000 AND salary <= 12000);

SELECT employee_id,salary FROM employees
WHERE salary <9000 OR salary > 12000;

SELECT employee_id,salary FROM employees
WHERE salary NOT BETWEEN 9000 AND 12000;
```

```
SELECT employee_id, last_name, job_id, salary
FROM employees
WHERE salary >= 10000
OR job_id LIKE '%MAN%';
```

注意：

OR可以和AND一起使用，但是在使用时要注意两者的优先级，由于AND的优先级高于OR，因此先对AND两边的操作数进行操作，再与OR中的操作数结合。

- 4. 逻辑异或运算符** 逻辑异或 (XOR) 运算符是当给定的值中任意一个值为NULL时，则返回NULL；如果两个非NULL的值都是0或者都不等于0时，则返回0；如果一个值为0，另一个值不为0时，则返回1。

```
| 1 XOR -1 | 1 XOR 0 | 0 XOR 0 | 1 XOR NULL | 1 XOR 1 XOR 1 | 0 XOR 0 XOR 0 |
+-----+-----+-----+-----+-----+-----+
|      0 |      1 |      0 |    NULL |      1 |      0 |
+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

```
select last_name,department_id,salary
from employees
where department_id in (10,20) XOR salary > 8000;
```

4. 位运算符

位运算符是在二进制数上进行计算的运算符。位运算符会先将操作数变成二进制数，然后进行位运算，最后将计算结果从二进制变回十进制数。

MySQL支持的位运算符如下：

运 算 符	作 用	示 例
&	按位与（位AND）	SELECT A & B
	按位或（位OR）	SELECT A B
^	按位异或（位XOR）	SELECT A ^ B
~	按位取反	SELECT ~ A
>>	按位右移	SELECT A >> 2
<<	按位左移	SELECT B << 2

1. 按位与运算符 按位与（&）运算符将给定值对应的二进制数逐位进行逻辑与运算。当给定值对应的二进制位的数值都为1时，则该位返回1，否则返回0。

```
mysql> SELECT 1 & 10, 20 & 30;
+-----+
| 1 & 10 | 20 & 30 |
+-----+
|      0 |     20 |
+-----+
1 row in set (0.00 sec)
```

1的二进制数为0001，10的二进制数为1010，所以1 & 10的结果为0000，对应的十进制数为0。20的二进制数为10100，30的二进制数为11110，所以20 & 30的结果为10100，对应的十进制数为20。

2. 按位或运算符 按位或（|）运算符将给定的值对应的二进制数逐位进行逻辑或运算。当给定值对应的二进制位的数值有一个或两个为1时，则该位返回1，否则返回0。

```
mysql> SELECT 1 | 10, 20 | 30;
+-----+
| 1 | 10 | 20 | 30 |
+-----+
|      11 |      30 |
+-----+
1 row in set (0.00 sec)
```

1的二进制数为0001，10的二进制数为1010，所以1 | 10的结果为1011，对应的十进制数为11。20的二进制数为10100，30的二进制数为11110，所以20 | 30的结果为11110，对应的十进制数为30。

```
mysql> SELECT 1 ^ 10, 20 ^ 30;
+-----+
| 1 ^ 10 | 20 ^ 30 |
+-----+
|      11 |       10 |
+-----+
1 row in set (0.00 sec)
```

1的二进制数为0001，10的二进制数为1010，所以 $1 \wedge 10$ 的结果为1011，对应的十进制数为11。20的二进制数为10100，30的二进制数为11110，所以 $20 \wedge 30$ 的结果为01010，对应的十进制数为10。

再举例：

```
mysql> SELECT 12 & 5, 12 | 5, 12 ^ 5 FROM DUAL;
+-----+-----+-----+
| 12 & 5 | 12 | 5 | 12 ^ 5 |
+-----+-----+-----+
|      4 |     13 |      9 |
+-----+-----+-----+
1 row in set (0.00 sec)
```

	0	0	0	0	1	1	0	0	12
&	0	0	0	0	0	1	0	1	5
	0	0	0	0	0	1	0	0	4
	0	0	0	0	1	1	0	0	12
	0	0	0	0	0	1	0	1	5
	0	0	0	0	1	1	0	1	13
	0	0	0	0	1	1	0	0	12
^	0	0	0	0	0	1	0	1	5
	0	0	0	0	1	0	0	1	9

4. 按位取反运算符 按位取反 (\sim) 运算符将给定的值的二进制数逐位进行取反操作，即将1变为0，将0变为1。

```
mysql> SELECT 10 & ~1;
+-----+
| 10 & ~1 |
+-----+
|      10 |
+-----+
1 row in set (0.00 sec)
```

5. 按位右移运算符 按位右移 (`>>`) 运算符将给定的值的二进制数的所有位右移指定的位数。右移指定的位数后，右边低位的数值被移出并丢弃，左边高位空出的位置用0补齐。

```
mysql> SELECT 1 >> 2, 4 >> 2;
+-----+-----+
| 1 >> 2 | 4 >> 2 |
+-----+-----+
|      0 |      1 |
+-----+-----+
1 row in set (0.00 sec)
```

1的二进制数为0000 0001，右移2位为0000 0000，对应的十进制数为0。4的二进制数为0000 0100，右移2位为0000 0001，对应的十进制数为1。

6. 按位左移运算符 按位左移 (`<<`) 运算符将给定的值的二进制数的所有位左移指定的位数。左移指定的位数后，左边高位的数值被移出并丢弃，右边低位空出的位置用0补齐。

```
mysql> SELECT 1 << 2, 4 << 2;
+-----+-----+
| 1 << 2 | 4 << 2 |
+-----+-----+
|      4 |     16 |
+-----+-----+
1 row in set (0.00 sec)
```

1的二进制数为0000 0001，左移两位为0000 0100，对应的十进制数为4。4的二进制数为0000 0100，左移两位为0001 0000，对应的十进制数为16。

5. 运算符的优先级

优 先 级	运 算 符
1	<code>:=, =</code> (赋值)
2	<code> , OR, XOR</code>
3	<code>&&, AND</code>
4	<code>NOT</code>
5	<code>BETWEEN, CASE, WHEN, THEN 和 ELSE</code>
6	<code>= (比较运算符) , <=>, >=, >, <=, <, >, !=, IS, LIKE, REGEXP和IN</code>
7	<code> </code>
8	<code>&</code>
9	<code><<与>></code>
10	<code>-和+</code>
11	<code>*, /, DIV, %和MOD</code>
12	<code>^</code>
13	<code>- (负号) 和~ (按位取反)</code>
14	<code>!</code>
15	<code>0</code>

数字编号越大，优先级越高，优先级高的运算符先进行计算。可以看到，赋值运算符的优先级最低，使

拓展：使用正则表达式查询

正则表达式通常被用来检索或替换那些符合某个模式的文本内容，根据指定的匹配模式匹配文本中符合要求的特殊字符串。例如，从一个文本文件中提取电话号码，查找一篇文章中重复的单词或者替换用户输入的某些敏感词语等，这些地方都可以使用正则表达式。正则表达式强大而且灵活，可以应用于非常复杂的查询。

MySQL中使用REGEXP关键字指定正则表达式的字符匹配模式。下表列出了REGEXP操作符中常用字符匹配列表。

选项	说明	例子	匹配值示例
^	匹配文本的开始字符	'^b'匹配以字母 b 开头的字符串	book, big, banana, bike
\$	匹配文本的结束字符	'st\$'匹配以 st 结尾的字符串	test, resist, persist
.	匹配任何单个字符	'b.t'匹配任何 b 和 t 之间有一个字符的字符串	bit, bat, but, bite
*	匹配零个或多个在它前面的字符	'f*n'匹配字符 n 前面有任意个字符 f 的字符串	fn, fan, faan, fabcn
+	匹配前面的字符 1 次或多次	'ba+'匹配以 b 开头后面紧跟至少有一个 a 的字符串	ba, bay, bare, battle
<字符串>	匹配包含指定的字符串的文本	'fa'匹配包含 fa 的字符串	fan, afa, faad
[字符集合]	匹配字符集合中的任何一个字符	'[xz]' 匹配包含 x 或者 z 的字符串	dizzy, zebra, x-ray, extra
[^]	匹配不在括号中的任何字符	'[^abc]'匹配任何不包含 a、b 或 c 的字符串	desk, fox, f8ke
字符串{n,}	匹配前面的字符串至少 n 次	'b{2}'匹配 2 个或更多的 b	bbb, bbbb, bbbbbbb
字符串{n,m}	匹配前面的字符串至少 n 次,至多 m 次。如果 n 为 0, 此参数为可选参数	'b{2,4}'匹配含最少 2 个、最多 4 个 b 的字符串	bb, bbb, bbbb

1. 查询以特定字符或字符串开头的记录

字符'^'匹配以特定字符或者字符串开头的文本。

在fruits表中，查询f_name字段以字母'b'开头的记录，SQL语句如下：

```
mysql> SELECT * FROM fruits WHERE f_name REGEXP '^b';
```

2. 查询以特定字符或字符串结尾的记录

字符'\$'匹配以特定字符或者字符串结尾的文本。

在fruits表中，查询f_name字段以字母'y'结尾的记录，SQL语句如下：

```
mysql> SELECT * FROM fruits WHERE f_name REGEXP 'y$';
```

3. 用符号"."来替代字符串中的任意一个字符

字符'.'匹配任意一个字符。在fruits表中，查询f_name字段值包含字母'a'与'g'且两个字母之间只有一个字母的记录，SQL语句如下：

```
mysql> SELECT * FROM fruits WHERE f_name REGEXP 'a.g';
```

4. 使用"*"和"+"来匹配多个字符

星号'*'匹配前面的字符任意多次，包括0次。加号'+'匹配前面的字符至少一次。

在fruits表中，查询f_name字段值以字母‘b’开头且‘b’后面出现字母‘a’至少一次的记录，SQL语句如下：

```
mysql> SELECT * FROM fruits WHERE f_name REGEXP '^ba+';
```

5. 匹配指定字符串 正则表达式可以匹配指定字符串，只要这个字符串在查询文本中即可，如要匹配多个字符串，多个字符串之间使用分隔符‘|’隔开。

在fruits表中，查询f_name字段值包含字符串“on”的记录，SQL语句如下：

```
mysql> SELECT * FROM fruits WHERE f_name REGEXP 'on';
```

在fruits表中，查询f_name字段值包含字符串“on”或者“ap”的记录，SQL语句如下：

```
mysql> SELECT * FROM fruits WHERE f_name REGEXP 'on|ap';
```

之前介绍过，LIKE运算符也可以匹配指定的字符串，但与REGEXP不同，LIKE匹配的字符串如果在文本中间出现，则找不到它，相应的行也不会返回。REGEXP在文本内进行匹配，如果被匹配的字符串在文本中出现，REGEXP将会找到它，相应的行也会被返回。对比结果如下所示。

在fruits表中，使用LIKE运算符查询f_name字段值为“on”的记录，SQL语句如下：

```
mysql> SELECT * FROM fruits WHERE f_name like 'on';
Empty set(0.00 sec)
```

6. 匹配指定字符中的任意一个 方括号“[]”指定一个字符集合，只匹配其中任何一个字符，即为所查找的文本。

在fruits表中，查找f_name字段中包含字母‘o’或者‘t’的记录，SQL语句如下：

```
mysql> SELECT * FROM fruits WHERE f_name REGEXP '[ot]';
```

在fruits表中，查询s_id字段中包含4、5或者6的记录，SQL语句如下：

```
mysql> SELECT * FROM fruits WHERE s_id REGEXP '[456]';
```

7. 匹配指定字符以外的字符 “[^字符集合]” 匹配不在指定集合中的任何字符。

在fruits表中，查询f_id字段中包含字母a~e和数字1~2以外字符的记录，SQL语句如下：

```
mysql> SELECT * FROM fruits WHERE f_id REGEXP '[^a-e1-2]';
```

8. 使用{n}或者{n,m}来指定字符串连续出现的次数 “字符串{n}”表示至少匹配n次前面的字符；“字符串{n,m}”表示匹配前面的字符串不少于n次，不多于m次。例如，a{2,}表示字母a连续出现至少2次，也可以大于2次；a{2,4}表示字母a连续出现最少2次，最多不能超过4次。

在fruits表中，查询f_name字段值出现字母‘x’至少2次的记录，SQL语句如下：

```
mysql> SELECT * FROM fruits WHERE f_name REGEXP 'x{2,}';
```

在fruits表中，查询f_name字段值出现字符串“ba”最少1次、最多3次的记录，SQL语句如下：

```
mysql> SELECT * FROM fruits WHERE f_name REGEXP 'ba{1,3}';
```

第05章_排序与分页

讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

1. 排序数据

1.1 排序规则

- 使用 ORDER BY 子句排序
 - ASC (ascend) :升序
 - DESC (descend) :降序
- ORDER BY 子句在SELECT语句的结尾。

1.2 单列排序

```
SELECT last_name, job_id, department_id, hire_date
FROM employees
ORDER BY hire_date ;
```

LAST_NAME	JOB_ID	DEPARTMENT_ID	HIRE_DATE
King	AD_PRES	90	17-JUN-87
Whalen	AD_ASST	10	17-SEP-87
Kochhar	AD_VP	90	21-SEP-89
Hunold	IT_PROG	60	03-JAN-90
Ernst	IT_PROG	60	21-MAY-91

20 rows selected.

```
SELECT last_name, job_id, department_id, hire_date
FROM employees
ORDER BY hire_date DESC ;
```

LAST_NAME	JOB_ID	DEPARTMENT_ID	HIRE_DATE
Zlotkey	SA_MAN	80	29-JAN-00
Mourgos	ST_MAN	50	16-NOV-99
Grant	SA_REP		24-MAY-99
Lorentz	IT_PROG	60	07-FEB-99
Vargas	ST_CLERK	50	09-JUL-98
Taylor	SA_REP	80	24-MAR-98
Matos	ST_CLERK	50	15-MAR-98
Fay	MK_REP	20	17-AUG-97
Davies	ST_CLERK	50	29-JAN-97

20 rows selected.

```
SELECT employee_id, last_name, salary*12 annsal
FROM employees
ORDER BY annsal;
```

	143	Matos	31200
	142	Davies	37200
	141	Rajs	42000
	107	Lorentz	50400
	200	Whalen	52800
	124	Mourgos	69600
	104	Ernst	72000
	202	Fay	72000
	178	Grant	84000

20 rows selected.

1.3 多列排序

```
SELECT last_name, department_id, salary
FROM employees
ORDER BY department_id, salary DESC;
```

LAST_NAME	DEPARTMENT_ID	SALARY
Whalen	10	4400
Hartstein	20	13000
Fay	20	6000
Mourgos	50	5800
Rajs	50	3500
Davies	50	3100
Matos	50	2600
Vargas	50	2500

20 rows selected.

- 可以使用不在SELECT列表中的列排序。
- 在对多列进行排序的时候，首先排序的第一列必须有相同的列值，才会对第二列进行排序。如果第一列数据中所有值都是唯一的，将不再对第二列进行排序。

2. 分页

2.1 背景

背景1：查询返回的记录太多了，查看起来很不方便，怎么样能够实现分页查询呢？

背景2：表里有 4 条数据，我们只想要显示第 2、3 条数据怎么办呢？

2.2 实现规则

- 分页原理

所谓分页显示，就是将数据库中的结果集，一段一段显示出来需要的条件。

• MySQL 中使用 LIMIT 实现分页

- 格式：

```
LIMIT [位置偏移量, ] 行数
```

1, 以此类推) ; 第二个参数“行数”指示返回的记录条数。

- 举例

```
--前10条记录:  
SELECT * FROM 表名 LIMIT 0, 10;  
或者  
SELECT * FROM 表名 LIMIT 10;  
  
--第11至20条记录:  
SELECT * FROM 表名 LIMIT 10, 10;  
  
--第21至30条记录:  
SELECT * FROM 表名 LIMIT 20, 10;
```

MySQL 8.0中可以使用“LIMIT 3 OFFSET 4”，意思是获取从第5条记录开始后面的3条记录，和“LIMIT 4,3;”返回的结果相同。

- 分页显式公式： **(当前页数-1) *每页条数, 每页条数**

```
SELECT * FROM table  
LIMIT (PageNo - 1)*PageSize, PageSize;
```

- **注意：LIMIT 子句必须放在整个SELECT语句的最后！**
- 使用 LIMIT 的好处

约束返回结果的数量可以 **减少数据表的网络传输量**，也可以 **提升查询效率**。如果我们知道返回结果只有1条，就可以使用 **LIMIT 1**，告诉 SELECT 语句只需要返回一条记录即可。这样的好处就是 SELECT 不需要扫描完整的表，只需要检索到一条符合条件的记录即可返回。

2.3 拓展

在不同的 DBMS 中使用的关键字可能不同。在 MySQL、PostgreSQL、MariaDB 和 SQLite 中使用 LIMIT 关键字，而且需要放到 SELECT 语句的最后面。

- 如果是 SQL Server 和 Access，需要使用 **TOP** 关键字，比如：

```
SELECT TOP 5 name, hp_max FROM heros ORDER BY hp_max DESC
```

- 如果是 DB2，使用 **FETCH FIRST 5 ROWS ONLY** 这样的关键字：

```
SELECT name, hp_max FROM heros ORDER BY hp_max DESC FETCH FIRST 5 ROWS ONLY
```

- 如果是 Oracle，你需要基于 **ROWNUM** 来统计行数：

```
SELECT rownum, last_name, salary FROM employees WHERE rownum < 5 ORDER BY salary DESC;
```

需要说明的是，这条语句是先取出来前 5 条数据行，然后再按照 hp_max 从高到低的顺序进行排序。但这样产生的结果和上述方法的并不一样。我会在后面讲到子查询，你可以使用



让天下没有难学的技术

```
SELECT last_name,salary  
FROM employees  
ORDER BY salary DESC)  
WHERE rownum < 10;
```

得到与上述方法一致的结果。

第06章_多表查询

讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

多表查询，也称为关联查询，指两个或更多个表一起完成查询操作。

前提条件：这些一起查询的表之间是有关系的（一对一、一对多），它们之间一定是有关联字段，这个关联字段可能建立了外键，也可能没有建立外键。比如：员工表和部门表，这两个表依靠“部门编号”进行关联。

1. 一个案例引发的多表连接

1.1 案例说明

EMPLOYEES表

employee_id
first_name
last_name
email
phone_number
job_id
salary
commission_pct
manager_id
department_id

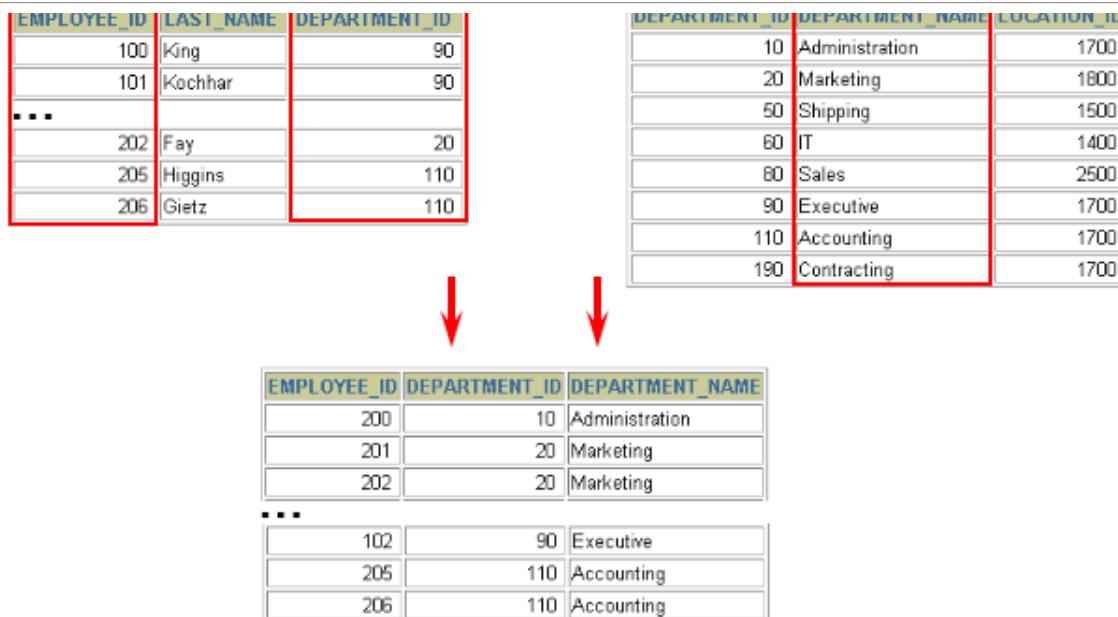
DEPARTMENTS表

department_id
department_name
manager_id
location_id

LOCATIONS表

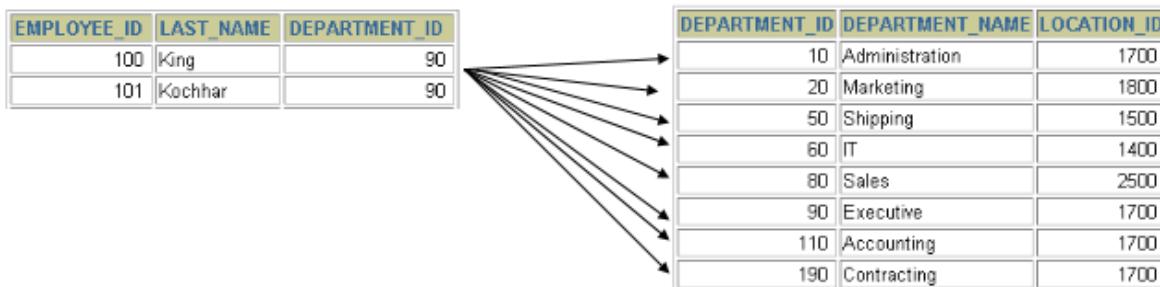
location_id
street_address
postal_code
city
state_province
country_id

从多个表中获取数据：



#案例：查询员工的姓名及其部门名称

```
SELECT last_name, department_name
FROM employees, departments;
```



查询结果：

```
+-----+-----+
| last_name | department_name |
+-----+-----+
| King      | Administration   |
| King      | Marketing       |
| King      | Purchasing     |
| King      | Human Resources |
| King      | Shipping        |
| King      | IT              |
| King      | Public Relations |
| King      | Sales            |
| King      | Executive       |
| King      | Finance          |
| King      | Accounting      |
| King      | Treasury         |
...
| Gietz    | IT Support      |
| Gietz    | NOC             |
| Gietz    | IT Helpdesk     |
| Gietz    | Government Sales|

```

```
+-----+-----+
2889 rows in set (0.01 sec)
```

分析错误情况：

```
SELECT COUNT(employee_id) FROM employees;
#输出107行

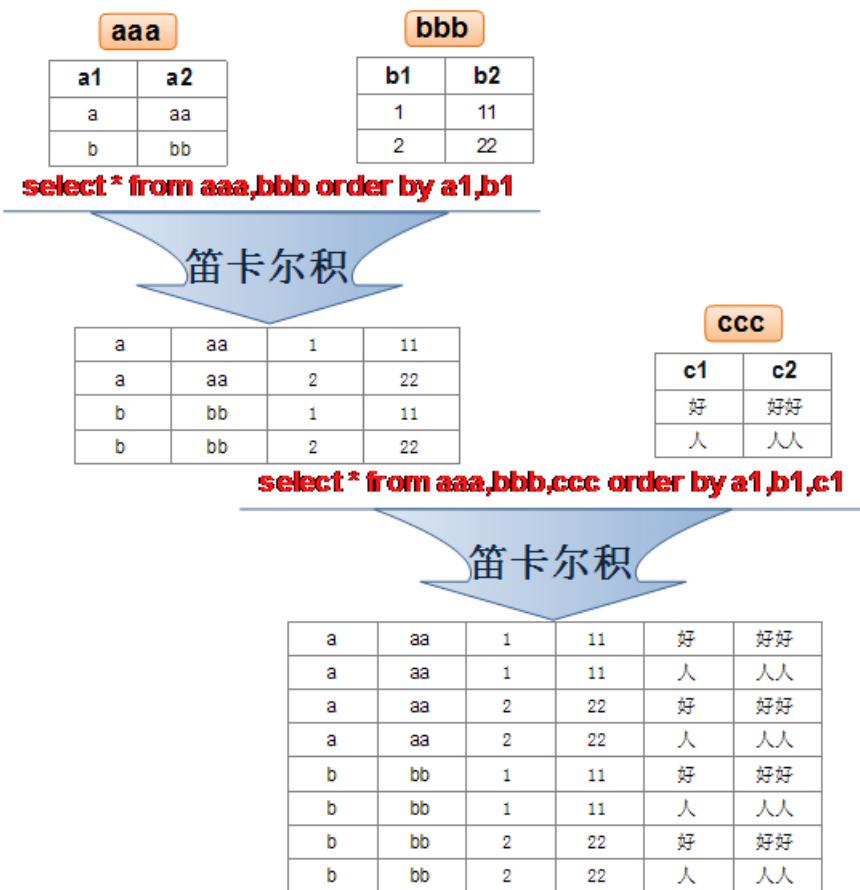
SELECT COUNT(department_id) FROM departments;
#输出27行

SELECT 107*27 FROM dual;
```

我们把上述多表查询中出现的问题称为：笛卡尔积的错误。

1.2 笛卡尔积（或交叉连接）的理解

笛卡尔乘积是一个数学运算。假设我有两个集合 X 和 Y，那么 X 和 Y 的笛卡尔积就是 X 和 Y 的所有可能组合，也就是第一个对象来自于 X，第二个对象来自于 Y 的所有可能。组合的个数即为两个集合中元素个数的乘积数。



SQL92中，笛卡尔积也称为 **交叉连接**，英文是 **CROSS JOIN**。在 SQL99 中也是使用 **CROSS JOIN** 表示交叉连接。它的作用就是可以把任意表进行连接，即使这两张表不相关。在MySQL中如下情况会出现笛卡尔积：

```
#查询员工姓名和所在部门名称
SELECT last_name,department_name FROM employees,departments;
SELECT last_name,department_name FROM employees CROSS JOIN departments;
SELECT last_name,department_name FROM employees INNER JOIN departments;
SELECT last_name,department_name FROM employees LEFT JOIN departments;
```

- 笛卡尔积的错误会在下面条件下产生：
 - 省略多个表的连接条件（或关联条件）
 - 连接条件（或关联条件）无效
 - 所有表中的所有行互相连接
- 为了避免笛卡尔积，可以在 WHERE 加入有效的连接条件。
- 加入连接条件后，查询语法：

```
SELECT table1.column, table2.column  
FROM table1, table2  
WHERE table1.column1 = table2.column2; #连接条件
```

- 在 WHERE 子句中写入连接条件。
- 正确写法：

```
#案例：查询员工的姓名及其部门名称  
SELECT last_name, department_name  
FROM employees, departments  
WHERE employees.department_id = departments.department_id;
```

- 在表中有相同列时，在列名之前加上表名前缀。

2. 多表查询分类讲解

分类1：等值连接 vs 非等值连接

等值连接

EMPLOYEES

EMPLOYEE_ID	DEPARTMENT_ID
200	10
201	20
202	20
124	50
141	50
142	50
143	50
144	50
103	60
104	60
107	60
149	80
174	80
176	80

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME
10	Administration
20	Marketing
20	Marketing
50	Shipping
60	IT
60	IT
60	IT
80	Sales
80	Sales
80	Sales

...

外键

...

主键：唯一、非空

```

    departments.location_id
FROM employees, departments
WHERE employees.department_id = departments.department_id;

```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID	LOCATION_ID
200	Whalen	10	10	1700
201	Hartstein	20	20	1800
202	Fay	20	20	1800
124	Mourgos	50	50	1500
141	Rajs	50	50	1500
142	Davies	50	50	1500
143	Matos	50	50	1500
144	Vargas	50	50	1500

19 rows selected.

拓展1：多个连接条件与 AND 操作符

EMPLOYEES

LAST_NAME	DEPARTMENT_ID
Whalen	10
Hartstein	20
Fay	20
Mourgos	50
Rajs	50
Davies	50
Matos	50
Vargas	50
Hunold	60
Ernst	60

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME
10	Administration
20	Marketing
20	Marketing
50	Shipping
60	IT
60	IT

...

...

拓展2：区分重复的列名

- 多个表中有相同列时，必须在列名之前加上表名前缀。
- 在不同表中具有相同列名的列可以用 表名 加以区分。

```

SELECT employees.last_name, departments.department_name, employees.department_id
FROM employees, departments
WHERE employees.department_id = departments.department_id;

```

拓展3：表的别名

- 使用别名可以简化查询。
- 列名前使用表名前缀可以提高查询效率。

```

SELECT e.employee_id, e.last_name, e.department_id,
       d.department_id, d.location_id
FROM   employees e , departments d
WHERE  e.department_id = d.department_id;

```

需要注意的是，如果我们使用了表的别名，在查询字段中、过滤条件中就只能使用别名进行代替，不能直接使用表名。

【强制】对于数据库中表记录的查询和变更，只要涉及多个表，都需要在列名前加表的别名（或表名）进行限定。

说明：对多表进行查询记录、更新记录、删除记录时，如果对操作列没有限定表的别名（或表名），并且操作列在多个表中存在时，就会抛异常。

正例：select t1.name from table_first as t1 , table_second as t2 where t1.id=t2.id;

反例：在某业务中，由于多表关联查询语句没有加表的别名（或表名）的限制，正常运行两年后，最近在某个表中增加一个同名字段，在预发布环境做数据库变更后，线上查询语句出现出1052 异常：Column 'name' in field list is ambiguous。

拓展4：连接多个表

EMPLOYEES		DEPARTMENTS		LOCATIONS	
LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID	LOCATION_ID	LOCATION_ID	CITY
King	90	10	1700	1400	Southlake
Kochhar	90	20	1800	1500	South San Francisco
De Haan	90	50	1500	1700	Seattle
Hunold	60	60	1400	1800	Toronto
Ernst	60	80	2500	2500	Oxford
Lorentz	60	90	1700		
Mourgos	50	110	1700		
Rajs	50	190	1700		
Davies	50				
Matos	50				
Vargas	50				
Zlotkey	80				
Abel	80				
Taylor	80				

20 rows selected.

8 rows selected.

总结：连接 n个表,至少需要n-1个连接条件。 比如，连接三个表，至少需要两个连接条件。

练习：查询出公司员工的 last_name,department_name, city

非等值连接

EMPLOYEES		JOB_GRADES		
LAST_NAME	SALARY	GRA	LOWEST_SAL	HIGHEST_SAL
King	24000	A	1000	2999
Kochhar	17000	B	3000	5999
De Haan	17000	C	6000	9999
Hunold	9000	D	10000	14999
Ernst	6000	E	15000	24999
Lorentz	4200	F	25000	40000
Mourgos	5800			
Rajs	3500			
Davies	3100			
Matos	2600			
Vargas	2500			
Zlotkey	10500			
Abel	11000			
Taylor	8600			

...
20 rows selected.

 EMPLOYEES表中的列工资
 应在JOB_GRADES表中的最高
 工资与最低工资之间

```
WHERE e.salary BETWEEN j.lowest_sal AND j.highest_sal;
```

LAST_NAME	SALARY	GRA
Matos	2600	A
Vargas	2500	A
Lorentz	4200	B
Mourgos	5800	B
Rajs	3500	B
Davies	3100	B
Whalen	4400	B
Hunold	9000	C
Ernst	6000	C

20 rows selected.

分类2：自连接 vs 非自连接

EMPLOYEES (WORKER)

EMPLOYEE_ID	LAST_NAME	MANAGER_ID
100	King	
101	Kochhar	100
102	De Haan	100
103	Hunold	102
104	Ernst	103
107	Lorentz	103
124	Mourgos	100

EMPLOYEES (MANAGER)

EMPLOYEE_ID	LAST_NAME
100	King
101	Kochhar
102	De Haan
103	Hunold
104	Ernst
107	Lorentz
124	Mourgos

...



WORKER 表中的MANAGER_ID 和 MANAGER 表中的EMPLOYEE_ID相等

- 当table1和table2本质上是同一张表，只是用取别名的方式虚拟成两张表以代表不同的意义。然后两个表再进行内连接，外连接等查询。

题目：查询employees表，返回“Xxx works for Xxx”

```
SELECT CONCAT(worker.last_name , ' works for '
            , manager.last_name)
  FROM employees worker, employees manager
 WHERE worker.manager_id = manager.employee_id ;
```

WORKER.LAST_NAME 'WORKSFOR' MANAGER.LAST_NAME
Kochhar works for King
De Haan works for King
Mourgos works for King
Zlotkey works for King
Hartstein works for King
Whalen works for Kochhar
Higgins works for Kochhar
Hunold works for De Haan
Ernst works for Hunold

19 rows selected.

练习：查询出last_name为 ‘Chen’ 的员工的 manager 的信息。

除了查询满足条件的记录以外，外连接还可以查询某一方不满足条件的记录。

DEPARTMENTS

DEPARTMENT_NAME	DEPARTMENT_ID
Administration	10
Marketing	20
Shipping	50
IT	60
Sales	80
Executive	90
Accounting	110
Contracting	190

8 rows selected.

EMPLOYEES

DEPARTMENT_ID	LAST_NAME
90	King
90	Kochhar
90	De Haan
60	Hunold
60	Ernst
60	Lorentz
50	Moungos
50	Rajs
50	Davies
50	Matos
50	Vargas
80	Zlotkey

20 rows selected.

190号部门没有员工

- 内连接: 合并具有同一列的两个以上的表的行, **结果集中不包含一个表与另一个表不匹配的行**
- 外连接: 两个表在连接过程中除了返回满足连接条件的行以外**还返回左 (或右) 表中不满足条件的行, 这种连接称为左 (或右) 外连接**。没有匹配的行时, 结果表中相应的列为空(NULL)。
- 如果是左外连接, 则连接条件中左边的表也称为**主表**, 右边的表称为**从表**。
如果是右外连接, 则连接条件中右边的表也称为**主表**, 左边的表称为**从表**。

SQL92: 使用(+)创建连接

- 在 SQL92 中采用 (+) 代表从表所在的位置。即左或右外连接中, (+) 表示哪个是从表。
- Oracle 对 SQL92 支持较好, 而 MySQL 则不支持 SQL92 的外连接。

```
#左外连接
SELECT last_name, department_name
FROM employees ,departments
WHERE employees.department_id = departments.department_id(+);
```

```
#右外连接
SELECT last_name, department_name
FROM employees ,departments
WHERE employees.department_id(+) = departments.department_id;
```

- 而且在 SQL92 中, 只有左外连接和右外连接, 没有满 (或全) 外连接。

3. SQL99语法实现多表查询

- 使用JOIN...ON子句创建连接的语法结构：

```
SELECT table1.column, table2.column, table3.column  
FROM table1  
    JOIN table2 ON table1 和 table2 的连接条件  
    JOIN table3 ON table2 和 table3 的连接条件
```

它的嵌套逻辑类似我们使用的FOR循环：

```
for t1 in table1:  
    for t2 in table2:  
        if condition1:  
            for t3 in table3:  
                if condition2:  
                    output t1 + t2 + t3
```

SQL99采用的这种嵌套结构非常清爽、层次性更强、可读性更强，即使再多的表进行连接也都清晰可见。如果你采用SQL92，可读性就会大打折扣。

- 语法说明：

- 可以使用ON子句指定额外的连接条件。
- 这个连接条件是与其它条件分开的。
- ON子句使语句具有更高的易读性。
- 关键字JOIN、INNER JOIN、CROSS JOIN的含义是一样的，都表示内连接

3.2 内连接(INNER JOIN)的实现

- 语法：

```
SELECT 字段列表  
FROM A表 INNER JOIN B表  
ON 关联条件  
WHERE 等其他子句；
```

题目1：

```
SELECT e.employee_id, e.last_name, e.department_id,  
       d.department_id, d.location_id  
FROM   employees e JOIN departments d  
ON     (e.department_id = d.department_id);
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID	LOCATION_ID
200	Whalen	10	10	1700
201	Hartstein	20	20	1800
202	Fay	20	20	1800
124	Mourgos	50	50	1500
141	Rajs	50	50	1500
142	Davies	50	50	1500
143	Matos	50	50	1500

19 rows selected.

题目2：

```

JOIN departments d
ON d.department_id = e.department_id
JOIN locations l
ON d.location_id = l.location_id;

```

EMPLOYEE_ID	CITY	DEPARTMENT_NAME
103	Southlake	IT
104	Southlake	IT
107	Southlake	IT
124	South San Francisco	Shipping
141	South San Francisco	Shipping
142	South San Francisco	Shipping
143	South San Francisco	Shipping
144	South San Francisco	Shipping

19 rows selected.

3.3 外连接(OUTER JOIN)的实现

3.3.1 左外连接(LEFT OUTER JOIN)

- 语法：

```

#实现查询结果是A
SELECT 字段列表
FROM A表 LEFT JOIN B表
ON 关联条件
WHERE 等其他子句;

```

- 举例：

```

SELECT e.last_name, e.department_id, d.department_name
FROM employees e
LEFT OUTER JOIN departments d
ON (e.department_id = d.department_id) ;

```

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Fay	20	Marketing
Hartstein	20	Marketing
De Haan	90	Executive
Kochhar	90	Executive
King	90	Executive
Gietz	110	Accounting
Higgins	110	Accounting
Grant		

20 rows selected.

3.3.2 右外连接(RIGHT OUTER JOIN)

- 语法：

```
FROM A表 RIGHT JOIN B表  
ON 关联条件  
WHERE 等其他子句；
```

- 举例：

```
SELECT e.last_name, e.department_id, d.department_name  
FROM employees e  
RIGHT OUTER JOIN departments d  
ON (e.department_id = d.department_id) ;
```

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
King	90	Executive
Kochhar	90	Executive
...		
Whalen	10	Administration
Hartstein	20	Marketing
Fay	20	Marketing
Higgins	110	Accounting
Gietz	110	Accounting
		Contracting

20 rows selected.

需要注意的是，LEFT JOIN 和 RIGHT JOIN 只存在于 SQL99 及以后的标准中，在 SQL92 中不存在，只能用 (+) 表示。

3.3.3 满外连接(FULL OUTER JOIN)

- 满外连接的结果 = 左右表匹配的数据 + 左表没有匹配到的数据 + 右表没有匹配到的数据。
- SQL99是支持满外连接的。使用FULL JOIN 或 FULL OUTER JOIN来实现。
- 需要注意的是，MySQL不支持FULL JOIN，但是可以用 LEFT JOIN UNION RIGHT join代替。

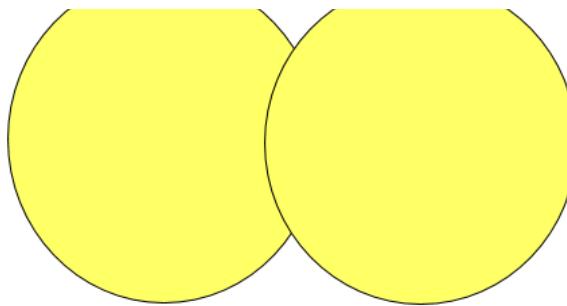
4. UNION的使用

合并查询结果 利用UNION关键字，可以给出多条SELECT语句，并将它们的结果组合成单个结果集。合并时，两个表对应的列数和数据类型必须相同，并且相互对应。各个SELECT语句之间使用UNION或UNION ALL关键字分隔。

语法格式：

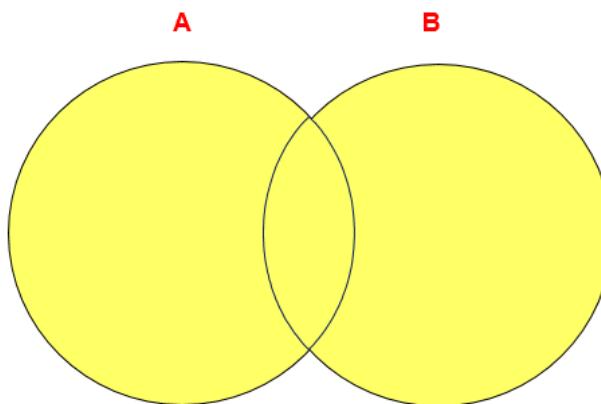
```
SELECT column,... FROM table1  
UNION [ALL]  
SELECT column,... FROM table2
```

UNION操作符



UNION 操作符返回两个查询的结果集的并集，去除重复记录。

UNION ALL操作符



UNION ALL操作符返回两个查询的结果集的并集。对于两个结果集的重复部分，不去重。

注意：执行UNION ALL语句时所需要的资源比UNION语句少。如果明确知道合并数据后的结果数据不存在重复数据，或者不需要去除重复的数据，则尽量使用UNION ALL语句，以提高数据查询的效率。

举例：查询部门编号>90或邮箱包含a的员工信息

#方式1

```
SELECT * FROM employees WHERE email LIKE '%a%' OR department_id>90;
```

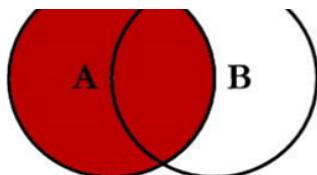
#方式2

```
SELECT * FROM employees WHERE email LIKE '%a%'  
UNION  
SELECT * FROM employees WHERE department_id>90;
```

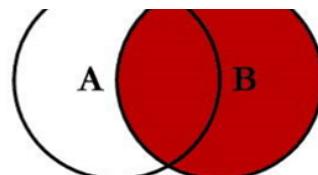
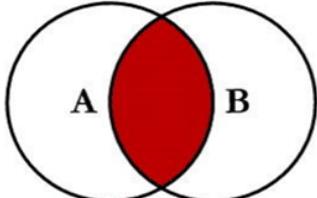
举例：查询中国用户中男性的信息以及美国用户中年男性的用户信息

```
SELECT id,cname FROM t_chinamale WHERE csex='男'  
UNION ALL  
SELECT id,tname FROM t_usmale WHERE tGender='male';
```

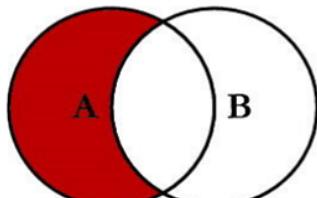
5. 7种SQL JOINS的实现



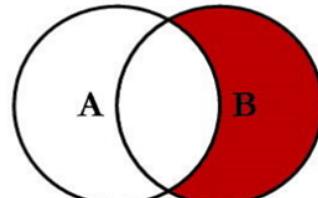
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



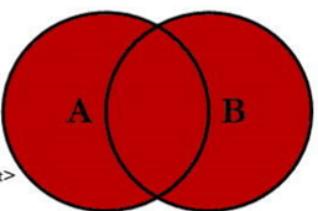
```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```

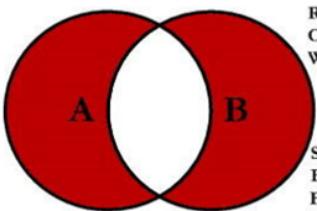


```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```



```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```

```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```



```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

5.7.1 代码实现

#中图: 内连接 $A \cap B$

```
SELECT employee_id, last_name, department_name
FROM employees e JOIN departments d
ON e.`department_id` = d.`department_id`;
```

#左上图: 左外连接

```
SELECT employee_id, last_name, department_name
FROM employees e LEFT JOIN departments d
ON e.`department_id` = d.`department_id`;
```

#右上图: 右外连接

```
SELECT employee_id, last_name, department_name
FROM employees e RIGHT JOIN departments d
ON e.`department_id` = d.`department_id`;
```

#左中图: $A - A \cap B$

```
SELECT employee_id, last_name, department_name
FROM employees e LEFT JOIN departments d
ON e.`department_id` = d.`department_id`
WHERE d.`department_id` IS NULL
```

#右中图: $B - A \cap B$

```
SELECT employee_id, last_name, department_name
FROM employees e RIGHT JOIN departments d
ON e.`department_id` = d.`department_id`
WHERE e.`department_id` IS NULL
```

```
SELECT employee_id, last_name, department_name
FROM employees e LEFT JOIN departments d
ON e.`department_id` = d.`department_id`
WHERE d.`department_id` IS NULL
UNION ALL #没有去重操作，效率高
SELECT employee_id, last_name, department_name
FROM employees e RIGHT JOIN departments d
ON e.`department_id` = d.`department_id`;
```

```
#右下图
#左中图 + 右中图 A ∪ B - A ∩ B 或者 (A - A ∩ B) ∪ (B - A ∩ B)
SELECT employee_id, last_name, department_name
FROM employees e LEFT JOIN departments d
ON e.`department_id` = d.`department_id`
WHERE d.`department_id` IS NULL
UNION ALL
SELECT employee_id, last_name, department_name
FROM employees e RIGHT JOIN departments d
ON e.`department_id` = d.`department_id`
WHERE e.`department_id` IS NULL
```

5.7.2 语法规则小结

- 左中图

```
#实现A - A ∩ B
select 字段列表
from A表 left join B表
on 关联条件
where 从表关联字段 is null and 等其他子句;
```

- 右中图

```
#实现B - A ∩ B
select 字段列表
from A表 right join B表
on 关联条件
where 从表关联字段 is null and 等其他子句;
```

- 左下图

```
#实现查询结果是A ∪ B
#用左外的A, union 右外的B
select 字段列表
from A表 left join B表
on 关联条件
where 等其他子句

union

select 字段列表
from A表 right join B表
on 关联条件
where 等其他子句;
```

```
#实现AUB - A∩B 或 (A - A∩B) ∪ (B - A∩B)
#使用左外的 (A - A∩B) union 右外的 (B - A∩B)
select 字段列表
from A表 left join B表
on 关联条件
where 从表关联字段 is null and 等其他子句

union

select 字段列表
from A表 right join B表
on 关联条件
where 从表关联字段 is null and 等其他子句
```

6. SQL99语法新特性

6.1 自然连接

SQL99 在 SQL92 的基础上提供了一些特殊语法，比如 `NATURAL JOIN` 用来表示自然连接。我们可以把自然连接理解为 SQL92 中的等值连接。它会帮你自动查询两张连接表中 `所有相同的字段`，然后进行 `等值连接`。

在SQL92标准中：

```
SELECT employee_id, last_name, department_name
FROM employees e JOIN departments d
ON e.`department_id` = d.`department_id`
AND e.`manager_id` = d.`manager_id`;
```

在 SQL99 中你可以写成：

```
SELECT employee_id, last_name, department_name
FROM employees e NATURAL JOIN departments d;
```

6.2 USING连接

当我们进行连接的时候，SQL99还支持使用 `USING` 指定数据表里的 `同名字段` 进行等值连接。但是只能配合 `JOIN`一起使用。比如：

```
SELECT employee_id, last_name, department_name
FROM employees e JOIN departments d
USING (department_id);
```

你能看出与自然连接 `NATURAL JOIN` 不同的是，`USING` 指定了具体的相同的字段名称，你需要在 `USING` 的括号 () 中填入要指定的同名字段。同时使用 `JOIN...USING` 可以简化 `JOIN ON` 的等值连接。它与下面的 SQL 查询结果是相同的：

```
SELECT employee_id, last_name, department_name
FROM employees e ,departments d
WHERE e.department_id = d.department_id;
```

表连接的约束条件可以有三种方式：WHERE, ON, USING

- WHERE: 适用于所有关联查询
- ON : 只能和JOIN一起使用，只能写关联条件。虽然关联条件可以并到WHERE中和其他条件一起写，但分开写可读性更好。
- USING: 只能和JOIN一起使用，而且要求**两个**关联字段在关联表中名称一致，而且只能表示关联字段值相等

```
#关联条件
#把关联条件写在where后面
SELECT last_name,department_name
FROM employees,departments
WHERE employees.department_id = departments.department_id;

#把关联条件写在on后面，只能和JOIN一起使用
SELECT last_name,department_name
FROM employees INNER JOIN departments
ON employees.department_id = departments.department_id;

SELECT last_name,department_name
FROM employees CROSS JOIN departments
ON employees.department_id = departments.department_id;

SELECT last_name,department_name
FROM employees JOIN departments
ON employees.department_id = departments.department_id;

#把关联字段写在using()中，只能和JOIN一起使用
#而且两个表中的关联字段必须名称相同，而且只能表示=
#查询员工姓名与基本工资
SELECT last_name,job_title
FROM employees INNER JOIN jobs USING(job_id);

#n张表关联，需要n-1个关联条件
#查询员工姓名，基本工资，部门名称
SELECT last_name,job_title,department_name FROM employees,departments,jobs
WHERE employees.department_id = departments.department_id
AND employees.job_id = jobs.job_id;

SELECT last_name,job_title,department_name
FROM employees INNER JOIN departments INNER JOIN jobs
ON employees.department_id = departments.department_id
AND employees.job_id = jobs.job_id;
```

注意：

我们要 **控制连接表的数量**。多表连接就相当于嵌套 for 循环一样，非常消耗资源，会让 SQL 查询性能下降得很严重，因此不要连接不必要的表。在许多 DBMS 中，也都会有最大连接表的限制。

【强制】超过三个表禁止 join。需要 join 的字段，数据类型保持绝对一致；多表关联查询时，保证被关联的字段需要有索引。

说明：即使双表 join 也要注意表索引、SQL 性能。

附录：常用的 SQL 标准有哪些

在正式开始讲连接表的种类时，我们首先需要知道 SQL 存在不同版本的标准规范，因为不同规范下的表连接操作是有区别的。

SQL 有两个主要的标准，分别是 **SQL92** 和 **SQL99**。92 和 99 代表了标准提出的时间，SQL92 就是 92 年提出的标准规范。当然除了 SQL92 和 SQL99 以外，还存在 SQL-86、SQL-89、SQL:2003、SQL:2008、SQL:2011 和 SQL:2016 等其他的标准。

这么多标准，到底该学习哪个呢？**实际上最重要的 SQL 标准就是 SQL92 和 SQL99**。一般来说 SQL92 的形式更简单，但是写的 SQL 语句会比较长，可读性较差。而 SQL99 相比于 SQL92 来说，语法更加复杂，但可读性更强。我们从这两个标准发布的页数也能看出，SQL92 的标准有 500 页，而 SQL99 标准超过了 1000 页。实际上从 SQL99 之后，很少有人能掌握所有内容，因为确实太多了。就好比我们使用 Windows、Linux 和 Office 的时候，很少有人能掌握全部内容一样。我们只需要掌握一些核心的功能，满足日常工作需求即可。

SQL92 和 SQL99 是经典的 SQL 标准，也分别叫做 SQL-2 和 SQL-3 标准。也正是在这两个标准发布之后，SQL 影响力越来越大，甚至超越了数据库领域。现如今 SQL 已经不仅仅是数据库领域的主流语言，还是信息领域中信息处理的主流语言。在图形检索、图像检索以及语音检索中都能看到 SQL 语言的使用。

第07章_单行函数

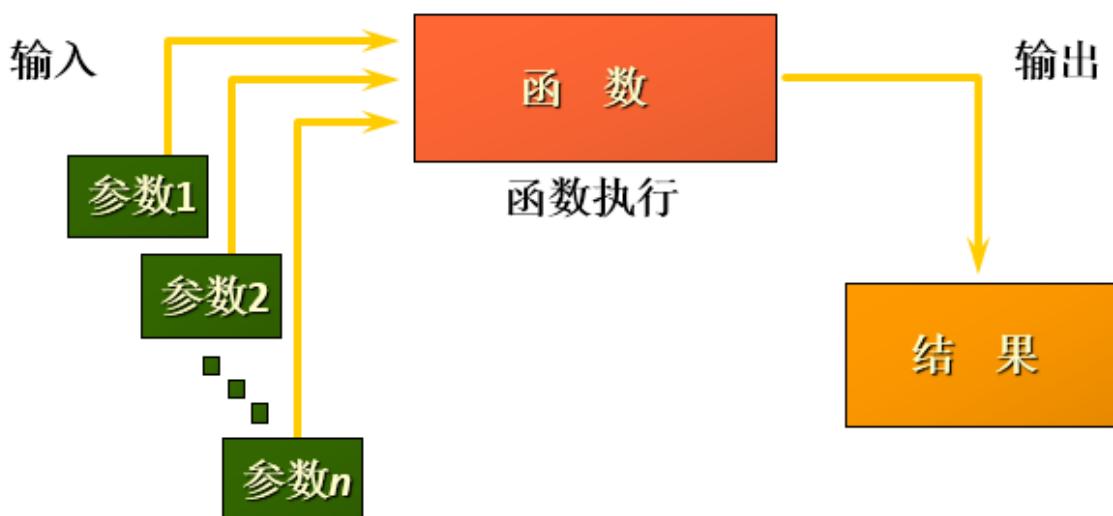
讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

1. 函数的理解

1.1 什么是函数

函数在计算机语言的使用中贯穿始终，函数的作用是什么呢？它可以把我们经常使用的代码封装起来，需要的时候直接调用即可。这样既 提高了代码效率，又 提高了可维护性。在 SQL 中我们也可以使用函数对检索出来的数据进行函数操作。使用这些函数，可以极大地 提高用户对数据库的管理效率。



$$y = f(x_1, \dots, x_n)$$

从函数定义的角度出发，我们可以将函数分成 内置函数 和 自定义函数。在 SQL 语言中，同样也包括了内置函数和自定义函数。内置函数是系统内置的通用函数，而自定义函数是我们根据自己的需要编写的，本章及下一章讲解的是 SQL 的内置函数。

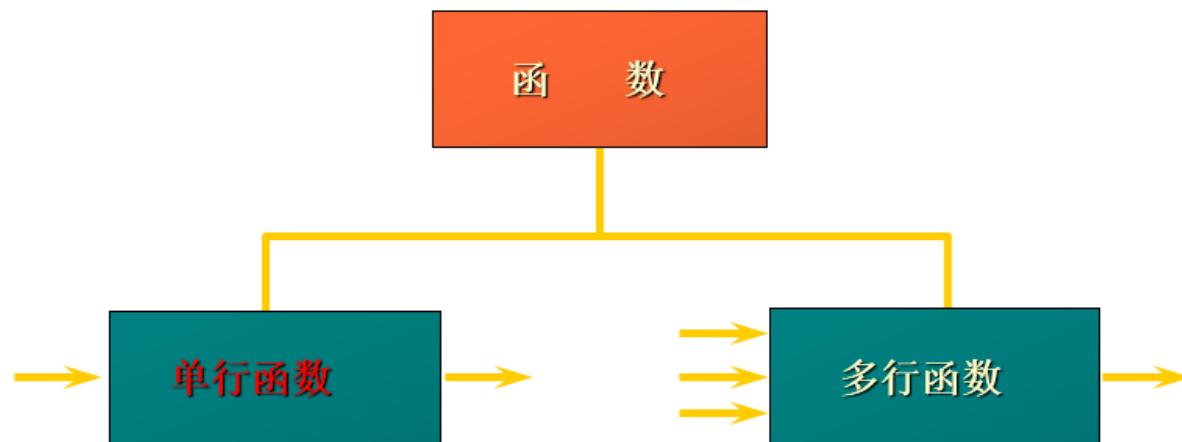
1.2 不同DBMS函数的差异

我们在使用 SQL 语言的时候，不是直接和这门语言打交道，而是通过它使用不同的数据库软件，即 DBMS。**DBMS 之间的差异性很大，远大于同一个语言不同版本之间的差异。**实际上，只有很少的函数是被 DBMS 同时支持的。比如，大多数 DBMS 使用 (||) 或者 (+) 来做拼接符，而在 MySQL 中的字符串拼接函数为concat()。大部分 DBMS 会有自己特定的函数，这就意味着**采用 SQL 函数的代码可移植性是很差的**，因此在使用函数的时候需要特别注意。

MySQL提供了丰富的内置函数，这些函数使得数据的维护与管理更加方便，能够更好地提供数据的分析与统计功能，在一定程度上提高了开发人员进行数据分析与统计的效率。

MySQL提供的内置函数从 实现的功能角度 可以分为数值函数、字符串函数、日期和时间函数、流程控制函数、加密与解密函数、获取MySQL信息函数、聚合函数等。这里，我将这些丰富的内置函数再分为两类： 单行函数 、 聚合函数（或分组函数）。

两种SQL函数



单行函数

- 操作数据对象
- 接受参数返回一个结果
- **只对一行进行变换**
- **每行返回一个结果**
- 可以嵌套
- 参数可以是一列或一个值

2. 数值函数

2.1 基本函数

ABS(x)	返回x的绝对值
SIGN(X)	返回X的符号。正数返回1，负数返回-1，0返回0
PI()	返回圆周率的值
CEIL(x), CEILING(x)	返回大于或等于某个值的最小整数
FLOOR(x)	返回小于或等于某个值的最大整数
LEAST(e1,e2,e3...)	返回列表中的最小值
GREATEST(e1,e2,e3...)	返回列表中的最大值
MOD(x,y)	返回X除以Y后的余数
RAND()	返回0~1的随机值
RAND(x)	返回0~1的随机值，其中x的值用作种子值，相同的x值会产生相同的随机数
ROUND(x)	返回一个对x的值进行四舍五入后，最接近于x的整数
ROUND(x,y)	返回一个对x的值进行四舍五入后最接近x的值，并保留到小数点后面y位
TRUNCATE(x,y)	返回数字x截断为y位小数的结果
SQRT(x)	返回x的平方根。当x的值为负数时，返回NULL

举例：

```
SELECT
    ABS(-123), ABS(32), SIGN(-23), SIGN(43), PI(), CEIL(32.32), CEILING(-43.23), FLOOR(32.32),
    FLOOR(-43.23), MOD(12, 5)
FROM DUAL;
```

```
+-----+-----+-----+-----+-----+-----+-----+
| ABS(-123) | ABS(32) | SIGN(-23) | SIGN(43) | PI()      | CEIL(32.32) | CEILING(-43.23) | FLOOR(32.32) | FLOOR(-43.23) | MOD(12, 5) |
+-----+-----+-----+-----+-----+-----+-----+
|   123 |     32 |       -1 |       1 | 3.141593 |        33 |      -43 |        32 |      -44 |        2 |
+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

```
SELECT RAND(), RAND(), RAND(10), RAND(10), RAND(-1), RAND(-1)
FROM DUAL;
```

```
+-----+-----+-----+-----+-----+-----+
| RAND() | RAND() | RAND(10) | RAND(10) | RAND(-1) | RAND(-1) |
+-----+-----+-----+-----+-----+-----+
| 0.27774592701135753 | 0.04671648335337311 | 0.6570515219653505 | 0.6570515219653505 | 0.9050373219931845 | 0.9050373219931845 |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

```
SELECT
    ROUND(12.33), ROUND(12.343, 2), ROUND(12.324, -1), TRUNCATE(12.66, 1), TRUNCATE(12.66, -1)
FROM DUAL;
```

```
+-----+-----+-----+-----+-----+
| ROUND(12.33) | ROUND(12.343, 2) | ROUND(12.324, -1) | TRUNCATE(12.66, 1) | TRUNCATE(12.66, -1) |
+-----+-----+-----+-----+-----+
|      12 |      12.34 |      10 |      12.6 |      10 |
+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

函数	用法
RADIANS(x)	将角度转化为弧度，其中，参数x为角度值
DEGREES(x)	将弧度转化为角度，其中，参数x为弧度值

```
SELECT RADIANS(30), RADIANS(60), RADIANS(90), DEGREES(2*PI()), DEGREES(RADIANS(90))
FROM DUAL;
```

2.3 三角函数

函数	用法
SIN(x)	返回x的正弦值，其中，参数x为弧度值
ASIN(x)	返回x的反正弦值，即获取正弦为x的值。如果x的值不在-1到1之间，则返回NULL
COS(x)	返回x的余弦值，其中，参数x为弧度值
ACOS(x)	返回x的反余弦值，即获取余弦为x的值。如果x的值不在-1到1之间，则返回NULL
TAN(x)	返回x的正切值，其中，参数x为弧度值
ATAN(x)	返回x的反正切值，即返回正切值为x的值
ATAN2(m,n)	返回两个参数的反正切值
COT(x)	返回x的余切值，其中，x为弧度值

举例：

ATAN2(M,N)函数返回两个参数的反正切值。与ATAN(X)函数相比，ATAN2(M,N)需要两个参数，例如有两个点point(x1,y1)和point(x2,y2)，使用ATAN(X)函数计算反正切值为ATAN((y2-y1)/(x2-x1))，使用ATAN2(M,N)计算反正切值则为ATAN2(y2-y1,x2-x1)。由使用方式可以看出，当x2-x1等于0时，ATAN(X)函数会报错，而ATAN2(M,N)函数则仍然可以计算。

ATAN2(M,N)函数的使用示例如下：

```
SELECT
SIN(RADIANS(30)),DEGREES(ASIN(1)),TAN(RADIANS(45)),DEGREES(ATAN(1)),DEGREES(ATAN2(1,1))
)
FROM DUAL;
```

SIN(RADIANS(30))	DEGREES(ASIN(1))	TAN(RADIANS(45))	DEGREES(ATAN(1))	DEGREES(ATAN2(1,1))
0.4999999999999994	90	0.9999999999999999	45	45

1 row in set (0.00 sec)

函数	用法
POW(x,y), POWER(X,Y)	返回x的y次方
EXP(X)	返回e的X次方, 其中e是一个常数, 2.718281828459045
LN(X), LOG(X)	返回以e为底的X的对数, 当X <= 0 时, 返回的结果为NULL
LOG10(X)	返回以10为底的X的对数, 当X <= 0 时, 返回的结果为NULL
LOG2(X)	返回以2为底的X的对数, 当X <= 0 时, 返回NULL

```
mysql> SELECT POW(2, 5),POWER(2, 4),EXP(2),LN(10),LOG10(10),LOG2(4)
-> FROM DUAL;
+-----+-----+-----+-----+-----+
| POW(2, 5) | POWER(2, 4) | EXP(2)          | LN(10)           | LOG10(10)        | LOG2(4)         |
+-----+-----+-----+-----+-----+
|      32   |        16   | 7.38905609893065 | 2.302585092994046 | 1               | 2               |
+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

2.5 进制间的转换

函数	用法
BIN(x)	返回x的二进制编码
HEX(x)	返回x的十六进制编码
OCT(x)	返回x的八进制编码
CONV(x,f1,f2)	返回f1进制数变成f2进制数

```
mysql> SELECT BIN(10),HEX(10),OCT(10),CONV(10,2,8)
-> FROM DUAL;
+-----+-----+-----+
| BIN(10) | HEX(10) | OCT(10) | CONV(10,2,8) |
+-----+-----+-----+
| 1010    | A       | 12      | 2           |
+-----+-----+-----+
1 row in set (0.00 sec)
```

3. 字符串函数

ASCII(s)	返回字符串s中的第一个字符的ASCII码值
CHAR_LENGTH(s)	返回字符串s的字符数。作用与CHARACTER_LENGTH(s)相同
LENGTH(s)	返回字符串s的字节数，和字符集有关
CONCAT(s1,s2,...,sn)	连接s1,s2,...,sn为一个字符串
CONCAT_WS(x, s1,s2,...,sn)	同CONCAT(s1,s2,...)函数，但是每个字符串之间要加上x
INSERT(str, idx, len, replacestr)	将字符串str从第idx位置开始，len个字符长的子串替换为字符串replacestr
REPLACE(str, a, b)	用字符串b替换字符串str中所有出现的字符串a
UPPER(s) 或 UCASE(s)	将字符串s的所有字母转成大写字母
LOWER(s) 或 LCASE(s)	将字符串s的所有字母转成小写字母
LEFT(str,n)	返回字符串str最左边的n个字符
RIGHT(str,n)	返回字符串str最右边的n个字符
LPAD(str, len, pad)	用字符串pad对str最左边进行填充，直到str的长度为len个字符
RPAD(str ,len, pad)	用字符串pad对str最右边进行填充，直到str的长度为len个字符
LTRIM(s)	去掉字符串s左侧的空格
RTRIM(s)	去掉字符串s右侧的空格
TRIM(s)	去掉字符串s开始与结尾的空格
TRIM(s1 FROM s)	去掉字符串s开始与结尾的s1
TRIM(LEADING s1 FROM s)	去掉字符串s开始处的s1
TRIM(TRAILING s1 FROM s)	去掉字符串s结尾处的s1
REPEAT(str, n)	返回str重复n次的结果
SPACE(n)	返回n个空格
STRCMP(s1,s2)	比较字符串s1,s2的ASCII码值的大小
SUBSTR(s,index,len)	返回从字符串s的index位置其len个字符，作用与SUBSTRING(s,n,len)、MID(s,n,len)相同
LOCATE(substr,str)	返回字符串substr在字符串str中首次出现的位置，作用于POSITION(substr IN str)、INSTR(str,substr)相同。未找到，返回0
ELT(m,s1,s2,...,sn)	返回指定位置的字符串，如果m=1，则返回s1，如果m=2，则返回s2，如果m=n，则返回sn
FIELD(s,s1,s2,...,sn)	返回字符串s在字符串列表中第一次出现的位置

FIND_IN_SET(s1,s2)	返回字符串s1在字符串s2中出现的位置。其中，字符串s2是一个以逗号分隔的字符串
REVERSE(s)	返回s反转后的字符串
NULLIF(value1,value2)	比较两个字符串，如果value1与value2相等，则返回NULL，否则返回value1

注意：MySQL中，字符串的位置是从1开始的。

举例：

```
mysql> SELECT FIELD('mm','hello','msm','amma'),FIND_IN_SET('mm','hello,mm,amma')
-> FROM DUAL;
+-----+-----+
| FIELD('mm','hello','msm','amma') | FIND_IN_SET('mm','hello,mm,amma') |
+-----+-----+
| 0 | 2 |
+-----+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT NULLIF('mysql','mysql'),NULLIF('mysql','');
+-----+-----+
| NULLIF('mysql','mysql') | NULLIF('mysql','') |
+-----+-----+
| mysql |          |
+-----+-----+
1 row in set (0.00 sec)
```

4. 日期和时间函数

4.1 获取日期、时间

函数	用法
CURDATE() , CURRENT_DATE()	返回当前日期，只包含年、月、日
CURTIME() , CURRENT_TIME()	返回当前时间，只包含时、分、秒
NOW() / SYSDATE() / CURRENT_TIMESTAMP() / LOCALTIME() / LOCALTIMESTAMP()	返回当前系统日期和时间
UTC_DATE()	返回UTC (世界标准时间) 日期
UTC_TIME()	返回UTC (世界标准时间) 时间

```
FROM DUAL;
```

```
+-----+-----+-----+-----+-----+-----+
| CURDATE() | CURTIME() | NOW()          | SYSDATE() +0   | UTC_DATE() | UTC_DATE() + 0 | UTC_TIME() | UTC_TIME() + 0 |
+-----+-----+-----+-----+-----+-----+
| 2021-10-25 | 19:36:55 | 2021-10-25 19:36:55 | 20211025193655 | 2021-10-25 | 20211025 | 11:36:55 | 113655 |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

4.2 日期与时间戳的转换

函数	用法
UNIX_TIMESTAMP()	以UNIX时间戳的形式返回当前时间。SELECT UNIX_TIMESTAMP() ->1634348884
UNIX_TIMESTAMP(date)	将时间date以UNIX时间戳的形式返回。
FROM_UNIXTIME(timestamp)	将UNIX时间戳的时间转换为普通格式的时间

举例：

```
mysql> SELECT UNIX_TIMESTAMP(now());
+-----+
| UNIX_TIMESTAMP(now()) |
+-----+
|           1576380910 |
+-----+
1 row in set (0.01 sec)

mysql> SELECT UNIX_TIMESTAMP(CURDATE());
+-----+
| UNIX_TIMESTAMP(CURDATE()) |
+-----+
|           1576339200 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT UNIX_TIMESTAMP(CURTIME());
+-----+
| UNIX_TIMESTAMP(CURTIME()) |
+-----+
|           1576380969 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT UNIX_TIMESTAMP('2011-11-11 11:11:11')
+-----+
| UNIX_TIMESTAMP('2011-11-11 11:11:11') |
+-----+
|           1320981071 |
+-----+
1 row in set (0.00 sec)
```

```
| FROM_UNIXTIME(1576380910) |
+-----+
| 2019-12-15 11:35:10 |
+-----+
1 row in set (0.00 sec)
```

4.3 获取月份、星期、星期数、天数等函数

函数	用法
YEAR(date) / MONTH(date) / DAY(date)	返回具体的日期值
HOUR(time) / MINUTE(time) / SECOND(time)	返回具体的时间值
MONTHNAME(date)	返回月份: January, ...
DAYNAME(date)	返回星期几: MONDAY, TUESDAY....SUNDAY
WEEKDAY(date)	返回周几, 注意, 周1是0, 周2是1, ... 周日是6
QUARTER(date)	返回日期对应的季度, 范围为1~4
WEEK(date), WEEKOFYEAR(date)	返回一年中的第几周
DAYOFYEAR(date)	返回日期是一年中的第几天
DAYOFMONTH(date)	返回日期位于所在月份的第几天
DAYOFWEEK(date)	返回周几, 注意: 周日是1, 周一是2, ... 周六是7

举例:

```
SELECT YEAR(CURDATE()), MONTH(CURDATE()), DAY(CURDATE()),
       HOUR(CURTIME()), MINUTE(NOW()), SECOND(SYSDATE())
  FROM DUAL;
```

```
+-----+
| YEAR(CURDATE()) | MONTH(CURDATE()) | DAY(CURDATE()) | HOUR(CURTIME()) | MINUTE(NOW()) | SECOND(SYSDATE()) |
+-----+
|          2021 |            10 |             25 |            21 |           34 |            50 |
+-----+
1 row in set (0.00 sec)
```

```
SELECT MONTHNAME('2021-10-26'), DAYNAME('2021-10-26'), WEEKDAY('2021-10-26'),
       QUARTER(CURDATE()), WEEK(CURDATE()), DAYOFYEAR(NOW()),
       DAYOFMONTH(NOW()), DAYOFWEEK(NOW())
  FROM DUAL;
```

```
+-----+
| MONTHNAME('2021-10-26') | DAYNAME('2021-10-26') | WEEKDAY('2021-10-26') | QUARTER(CURDATE()) | WEEK(CURDATE()) | DAYOFYEAR(NOW()) | DAYOFMONTH(NOW()) | DAYOFWEEK(NOW()) |
+-----+
| October                | Tuesday                 |          1 |           4 |        43 |        298 |         25 |          2 |
+-----+
1 row in set (0.00 sec)
```

函数	用法
EXTRACT(type FROM date)	返回指定日期中特定的部分, type指定返回的值

EXTRACT(type FROM date)函数中type的取值与含义:

type取值	含 义
MICROSECOND	返回毫秒数
SECOND	返回秒数
MINUTE	返回分钟数
HOUR	返回小时数
DAY	返回天数
WEEK	返回日期在一年中的第几个星期
MONTH	返回日期在一年中的第几个月
QUARTER	返回日期在一年中的第几个季度
YEAR	返回日期的年份
SECOND_MICROSECOND	返回秒和毫秒值
MINUTE_MICROSECOND	返回分钟和毫秒值
MINUTE_SECOND	返回分钟和秒值
HOUR_MICROSECOND	返回小时和毫秒值
HOUR_SECOND	返回小时和秒值
HOUR_MINUTE	返回小时和分钟值
DAY_MICROSECOND	返回天和毫秒值
DAY_SECOND	返回天和秒值
DAY_MINUTE	返回天和分钟值
DAY_HOUR	返回天和小时
YEAR_MONTH	返回年和月

```
SELECT EXTRACT(MINUTE FROM NOW()),EXTRACT( WEEK FROM NOW()),
EXTRACT( QUARTER FROM NOW()),EXTRACT( MINUTE_SECOND FROM NOW())
FROM DUAL;
```

4.5 时间和秒钟转换的函数

函数	用法
TIME_TO_SEC(time)	将 time 转化为秒并返回结果值。转化的公式为: 小时*3600+分钟*60+秒
SEC_TO_TIME(seconds)	将 seconds 描述转化为包含小时、分钟和秒的时间

举例:

```
| TIME_TO_SEC(NOW()) |
+-----+
|          78774 |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT SEC_TO_TIME(78774);
+-----+
| SEC_TO_TIME(78774) |
+-----+
| 21:52:54           |
+-----+
1 row in set (0.12 sec)
```

4.6 计算日期和时间的函数

第1组：

函数	用法
DATE_ADD(datetime, INTERVAL expr type), ADDDATE(date,INTERVAL expr type)	返回与给定日期时间相差INTERVAL时间段的日期时间
DATE_SUB(date,INTERVAL expr type), SUBDATE(date,INTERVAL expr type)	返回与date相差INTERVAL时间间隔的日期

上述函数中type的取值：

间 隔 类 型	含 义
HOUR	小时
MINUTE	分钟
SECOND	秒
YEAR	年
MONTH	月
DAY	日
YEAR_MONTH	年和月
DAY_HOUR	日和小时
DAY_MINUTE	日和分钟
DAY_SECOND	日和秒
HOUR_MINUTE	小时和分钟
HOUR_SECOND	小时和秒
MINUTE_SECOND	分钟和秒

举例：

```

ADDDATE('2021-10-21 23:32:12', INTERVAL 1 SECOND) AS col3,
DATE_ADD('2021-10-21 23:32:12', INTERVAL '1_1' MINUTE_SECOND) AS col4,
DATE_ADD(NOW(), INTERVAL -1 YEAR) AS col5, #可以是负数
DATE_ADD(NOW(), INTERVAL '1_1' YEAR_MONTH) AS col6 #需要单引号
FROM DUAL;
    
```

```

SELECT DATE_SUB('2021-01-21', INTERVAL 31 DAY) AS col1,
SUBDATE('2021-01-21', INTERVAL 31 DAY) AS col2,
DATE_SUB('2021-01-21 02:01:01', INTERVAL '1_1' DAY_HOUR) AS col3
FROM DUAL;
    
```

第2组：

函数	用法
ADDTIME(time1,time2)	返回time1加上time2的时间。当time2为一个数字时，代表的是秒，可以为负数
SUBTIME(time1,time2)	返回time1减去time2后的时间。当time2为一个数字时，代表的是秒，可以为负数
DATEDIFF(date1,date2)	返回date1 - date2的日期间隔天数
TIMEDIFF(time1, time2)	返回time1 - time2的时间间隔
FROM_DAYS(N)	返回从0000年1月1日起，N天以后的日期
TO_DAYS(date)	返回日期date距离0000年1月1日的天数
LAST_DAY(date)	返回date所在月份的最后一天的日期
MAKEDATE(year,n)	针对给定年份与所在年份中的天数返回一个日期
MAKETIME(hour,minute,second)	将给定的小时、分钟和秒组合成时间并返回
PERIOD_ADD(time,n)	返回time加上n后的时间

举例：

```

SELECT
ADDTIME(NOW(),20),SUBTIME(NOW(),30),SUBTIME(NOW(),'1:1:3'),DATEDIFF(NOW(),'2021-10-01'),
TIMEDIFF(NOW(),'2021-10-25 22:10:10'),FROM_DAYS(366),TO_DAYS('0000-12-25'),
LAST_DAY(NOW()),MAKEDATE(YEAR(NOW()),12),MAKETIME(10,21,23),PERIOD_ADD(20200101010101,10)
FROM DUAL;
    
```

```

mysql> SELECT ADDTIME(NOW(), 50);
+-----+
| ADDTIME(NOW(), 50) |
+-----+
| 2019-12-15 22:17:47 |
+-----+
1 row in set (0.00 sec)
    
```

```
+-----+
| 2019-12-15 23:18:46      |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT SUBTIME(NOW(), '1:1:1');
+-----+
| SUBTIME(NOW(), '1:1:1') |
+-----+
| 2019-12-15 21:23:50      |
+-----+
1 row in set (0.00 sec)

mysql> SELECT SUBTIME(NOW(), '-1:-1:-1');
+-----+
| SUBTIME(NOW(), '-1:-1:-1') |
+-----+
| 2019-12-15 22:25:11      |
+-----+
1 row in set, 1 warning (0.00 sec)
```

```
mysql> SELECT FROM_DAYS(366);
+-----+
| FROM_DAYS(366) |
+-----+
| 0001-01-01      |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT MAKEDATE(2020,1);
+-----+
| MAKEDATE(2020,1) |
+-----+
| 2020-01-01      |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT MAKEDATE(2020,32);
+-----+
| MAKEDATE(2020,32) |
+-----+
| 2020-02-01      |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT MAKETIME(1,1,1);
+-----+
| MAKETIME(1,1,1) |
+-----+
| 01:01:01      |
+-----+
1 row in set (0.00 sec)
```

```
| PERIOD_ADD(20200101010101,1) |
+-----+
|          20200101010102 |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT TO_DAYS(NOW());
+-----+
| TO_DAYS(NOW()) |
+-----+
|      737773 |
+-----+
1 row in set (0.00 sec)
```

举例：查询 7 天内的新增用户数有多少？

```
SELECT COUNT(*) as num FROM new_user WHERE TO_DAYS(NOW())-TO_DAYS(regist_time)<=7
```

4.7 日期的格式化与解析

函数	用法
DATE_FORMAT(date,fmt)	按照字符串fmt格式化日期date值
TIME_FORMAT(time,fmt)	按照字符串fmt格式化时间time值
GET_FORMAT(date_type,format_type)	返回日期字符串的显示格式
STR_TO_DATE(str,fmt)	按照字符串fmt对str进行解析，解析为一个日期

上述 **非**GET_FORMAT 函数中fmt参数常用的格式符：

符	符	符	符
%Y	4位数字表示年份	%y	表示两位数字表示年份
%M	月名表示月份 (January,...)	%m	两位数字表示月份 (01,02,03。。.)
%b	缩写的月名 (Jan., Feb., ...)	%c	数字表示月份 (1,2,3,...)
%D	英文后缀表示月中的天数 (1st,2nd,3rd,...)	%d	两位数字表示月中的天数(01,02...)
%e	数字形式表示月中的天数 (1,2,3,4,5.....)		
%H	两位数字表示小数, 24小时制 (01,02..)	%h 和%l	两位数字表示小时, 12小时制 (01,02..)
%k	数字形式的小时, 24小时制(1,2,3)	%l	数字形式表示小时, 12小时制 (1,2,3,4....)
%i	两位数字表示分钟 (00,01,02)	%S 和%ss	两位数字表示秒(00,01,02...)
%W	一周中的星期名称 (Sunday...)	%a	一周中的星期缩写 (Sun., Mon.,Tues., ..)
%w	以数字表示周中的天数 (0=Sunday,1=Monday....)		
%j	以3位数字表示年中的天数(001,002...)	%U	以数字表示年中的第几周, (1,2,3。。) 其中Sunday为周中第一天
%u	以数字表示年中的第几周, (1,2,3。。) 其中Monday为周中第一天		
%T	24小时制	%r	12小时制
%p	AM或PM	%%	表示%

GET_FORMAT函数中date_type和format_type参数取值如下：

DAT	USA	/0111. /001. /0 1
DATE	JIS	%Y-%m-%d
DATE	ISO	%Y-%m-%d
DATE	EUR	%d.%m.%Y
DATE	INTERNAL	%Y%m%d
TIME	USA	%h:%i:%s %p
TIME	JIS	%H:%i:%s
TIME	ISO	%H:%i:%s
TIME	EUR	%H.%i.%s
TIME	INTERNAL	%H%i%s
DATETIME	USA	%Y-%m-%d %H.%i.%s
DATETIME	JIS	%Y-%m-%d %H:%i:%s
DATETIME	ISO	%Y-%m-%d %H:%i:%s
DATETIME	EUR	%Y-%m-%d %H.%i.%s
DATETIME	INTERNAL	%Y%m%d%H%i%s

举例：

```
mysql> SELECT DATE_FORMAT(NOW(), '%H:%i:%s');
+-----+
| DATE_FORMAT(NOW(), '%H:%i:%s') |
+-----+
| 22:57:34 |
+-----+
1 row in set (0.00 sec)
```

```
SELECT STR_TO_DATE('09/01/2009', '%m/%d/%Y')
FROM DUAL;

SELECT STR_TO_DATE('20140422154706', '%Y%m%d%H%i%s')
FROM DUAL;

SELECT STR_TO_DATE('2014-04-22 15:47:06', '%Y-%m-%d %H:%i:%s')
FROM DUAL;
```

```
mysql> SELECT GET_FORMAT(DATE, 'USA');
+-----+
| GET_FORMAT(DATE, 'USA') |
+-----+
| %m.%d.%Y |
+-----+
1 row in set (0.00 sec)
```

```
SELECT DATE_FORMAT(NOW(), GET_FORMAT(DATE, 'USA')),
FROM DUAL;
```

```
mysql> SELECT STR_TO_DATE('2020-01-01 00:00:00', '%Y-%m-%d');
+-----+
| STR_TO_DATE('2020-01-01 00:00:00', '%Y-%m-%d') |
+-----+
| 2020-01-01 |
+-----+
```

流程处理函数可以根据不同的条件，执行不同的处理流程，可以在SQL语句中实现不同的条件选择。MySQL中的流程处理函数主要包括IF()、IFNULL()和CASE()函数。

函数	用法
IF(value,value1,value2)	如果value的值为TRUE，返回value1，否则返回value2
IFNULL(value1, value2)	如果value1不为NULL，返回value1，否则返回value2
CASE WHEN 条件1 THEN 结果1 WHEN 条件2 THEN 结果2 [ELSE resultn] END	相当于Java的if...else if...else...
CASE expr WHEN 常量值1 THEN 值1 WHEN 常量值1 THEN 值1 [ELSE 值n] END	相当于Java的switch...case...

```
SELECT IF(1 > 0, '正确', '错误')
```

->正确

```
SELECT IFNULL(null, 'Hello Word')
```

->Hello Word

```
SELECT CASE
    WHEN 1 > 0
    THEN '1 > 0'
    WHEN 2 > 0
    THEN '2 > 0'
    ELSE '3 > 0'
    END
->1 > 0
```

```
SELECT CASE 1
    WHEN 1 THEN '我是1'
    WHEN 2 THEN '我是2'
    ELSE '你是谁'
```

```
SELECT employee_id,salary, CASE WHEN salary>=15000 THEN '高薪'
                                WHEN salary>=10000 THEN '潜力股'
                                WHEN salary>=8000 THEN '屌丝'
                                ELSE '草根' END "描述"
FROM employees;
```

```
SELECT oid,`status`, CASE `status` WHEN 1 THEN '未付款'
                            WHEN 2 THEN '已付款'
                            WHEN 3 THEN '已发货'
                            WHEN 4 THEN '确认收货'
                            ELSE '无效订单' END
FROM t_order;
```

```
mysql> SELECT CASE WHEN 1 > 0 THEN 'yes' WHEN 1 <= 0 THEN 'no' ELSE 'unknown' END;
+-----+
| CASE WHEN 1 > 0 THEN 'yes' WHEN 1 <= 0 THEN 'no' ELSE 'unknown' END |
+-----+
```

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

```
mysql> SELECT CASE WHEN 1 < 0 THEN 'yes' WHEN 1 = 0 THEN 'no' ELSE 'unknown' END;
+-----+
| CASE WHEN 1 < 0 THEN 'yes' WHEN 1 = 0 THEN 'no' ELSE 'unknown' END |
+-----+
| unknown |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT CASE 1 WHEN 0 THEN 0 WHEN 1 THEN 1 ELSE -1 END;
+-----+
| CASE 1 WHEN 0 THEN 0 WHEN 1 THEN 1 ELSE -1 END |
+-----+
| 1 |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT CASE -1 WHEN 0 THEN 0 WHEN 1 THEN 1 ELSE -1 END;
+-----+
| CASE -1 WHEN 0 THEN 0 WHEN 1 THEN 1 ELSE -1 END |
+-----+
| -1 |
+-----+
1 row in set (0.00 sec)
```

```
SELECT employee_id, 12 * salary * (1 + IFNULL(commission_pct, 0))
FROM employees;
```

```
SELECT last_name, job_id, salary,
CASE job_id WHEN 'IT_PROG' THEN 1.10*salary
WHEN 'ST_CLERK' THEN 1.15*salary
WHEN 'SA_REP' THEN 1.20*salary
ELSE salary END "REVISED_SALARY"
FROM employees;
```

LAST_NAME	JOB_ID	SALARY	REVISED_SALARY
Lorentz	IT_PROG	4200	4620
Mourgos	ST_MAN	5800	5800
Rajs	ST_CLERK	3600	4025
Gietz	AC_ACCOUNT	8300	8300

20 rows selected.

练习：查询部门号为 10,20, 30 的员工信息, 若部门号为 10, 则打印其工资的 1.1 倍, 20 号部门, 则打印其工资的 1.2 倍, 30 号部门打印其工资的 1.3 倍数。

6. 加密与解密函数

加密与解密函数主要用于对数据库中的数据进行加密和解密处理，以防止数据被他人窃取。这些函数在

PASSWORD(str)	返回字符串str的加密版本，41位长的字符串。加密结果不可逆，常用于用户的密码加密
MD5(str)	返回字符串str的md5加密后的值，也是一种加密方式。若参数为NULL，则会返回NULL
SHA(str)	从原明文密码str计算并返回加密后的密码字符串，当参数为NULL时，返回NULL。 SHA加密算法比MD5更加安全。
ENCODE(value,password_seed)	返回使用password_seed作为加密密码加密value
DECODE(value,password_seed)	返回使用password_seed作为加密密码解密value

可以看到，ENCODE(value,password_seed)函数与DECODE(value,password_seed)函数互为反函数。

举例：

```
mysql> SELECT PASSWORD('mysql'), PASSWORD(NULL);
+-----+-----+
| PASSWORD('mysql') | PASSWORD(NULL) |
+-----+-----+
| *E74858DB86EBA20BC33D0AECAE8A8108C56B17FA |           |
+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

```
SELECT md5('123')
->202cb962ac59075b964b07152d234b70
```

```
SELECT SHA('Tom123')
->c7c506980abc31cc390a2438c90861d0f1216d50
```

```
mysql> SELECT ENCODE('mysql', 'mysql');
+-----+
| ENCODE('mysql', 'mysql') |
+-----+
| ig ¼ iÉ |
+-----+
1 row in set, 1 warning (0.01 sec)
```

```
mysql> SELECT DECODE(ENCODE('mysql', 'mysql'), 'mysql');
+-----+
| DECODE(ENCODE('mysql', 'mysql'), 'mysql') |
+-----+
| mysql |
+-----+
1 row in set, 2 warnings (0.00 sec)
```

7. MySQL信息函数

MySQL中内置了一些可以查询MySQL信息的函数，这些函数主要用于帮助数据库开发或运维人员更好地对数据库进行维护工作。

VERSION()	返回当前MySQL的版本号
CONNECTION_ID()	返回当前MySQL服务器的连接数
DATABASE(), SCHEMA()	返回MySQL命令行当前所在的数据库
USER(), CURRENT_USER()、SYSTEM_USER(), SESSION_USER()	返回当前连接MySQL的用户名，返回结果格式为“主机名@用户名”
CHARSET(value)	返回字符串value自变量的字符集
COLLATION(value)	返回字符串value的比较规则

举例：

```
mysql> SELECT DATABASE();
+-----+
| DATABASE() |
+-----+
| test      |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT DATABASE();
+-----+
| DATABASE() |
+-----+
| test      |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT USER(), CURRENT_USER(), SYSTEM_USER(), SESSION_USER();
+-----+-----+-----+-----+
| USER() | CURRENT_USER() | SYSTEM_USER() | SESSION_USER() |
+-----+-----+-----+-----+
| root@localhost | root@localhost | root@localhost | root@localhost |
+-----+-----+-----+-----+
```

```
mysql> SELECT CHARSET('ABC');
+-----+
| CHARSET('ABC') |
+-----+
| utf8mb4      |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT COLLATION('ABC');
+-----+
| COLLATION('ABC') |
+-----+
| utf8mb4_general_ci |
+-----+
1 row in set (0.00 sec)
```

MySQL中有些函数无法对其进行具体的分类，但是这些函数在MySQL的开发和运维过程中也是不容忽视的。

函数	用法
FORMAT(value,n)	返回对数字value进行格式化后的结果数据。n表示 四舍五入 后保留到小数点后n位
CONV(value,from,to)	将value的值进行不同进制之间的转换
INET_ATON(ipvalue)	将以点分隔的IP地址转化为一个数字
INET_NTOA(value)	将数字形式的IP地址转化为以点分隔的IP地址
BENCHMARK(n,expr)	将表达式expr重复执行n次。用于测试MySQL处理expr表达式所耗费的时间
CONVERT(value USING char_code)	将value所使用的字符编码修改为char_code

举例：

```
# 如果n的值小于或者等于0，则只保留整数部分
mysql> SELECT FORMAT(123.123, 2), FORMAT(123.523, 0), FORMAT(123.123, -2);
+-----+-----+-----+
| FORMAT(123.123, 2) | FORMAT(123.523, 0) | FORMAT(123.123, -2) |
+-----+-----+-----+
| 123.12           | 124            | 123             |
+-----+-----+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT CONV(16, 10, 2), CONV(8888,10,16), CONV(NULL, 10, 2);
+-----+-----+-----+
| CONV(16, 10, 2) | CONV(8888,10,16) | CONV(NULL, 10, 2) |
+-----+-----+-----+
| 10000          | 22B8           | NULL            |
+-----+-----+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT INET_ATON('192.168.1.100');
+-----+
| INET_ATON('192.168.1.100') |
+-----+
| 3232235876 |
+-----+
1 row in set (0.00 sec)
```

以“192.168.1.100”为例，计算方式为192乘以256的3次方，加上168乘以256的2次方，加上1乘以256，再加上100。

```
| INET_NTOA(3232235876) |
+-----+
| 192.168.1.100          |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT BENCHMARK(1, MD5('mysql'));
+-----+
| BENCHMARK(1, MD5('mysql')) |
+-----+
|                      0 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT BENCHMARK(1000000, MD5('mysql'));
+-----+
| BENCHMARK(1000000, MD5('mysql')) |
+-----+
|                      0 |
+-----+
1 row in set (0.20 sec)
```

```
mysql> SELECT CHARSET('mysql'), CHARSET(CONVERT('mysql' USING 'utf8'));
+-----+-----+
| CHARSET('mysql') | CHARSET(CONVERT('mysql' USING 'utf8')) |
+-----+-----+
| utf8mb4        | utf8                         |
+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

第08章_聚合函数

讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

我们上一章讲到了 SQL 单行函数。实际上 SQL 函数还有一类，叫做聚合（或聚集、分组）函数，它是对一组数据进行汇总的函数，输入的是一组数据的集合，输出的是单个值。

1. 聚合函数介绍

- **什么是聚合函数**

聚合函数作用于一组数据，并对一组数据返回一个值。

EMPLOYEES

DEPARTMENT_ID	SALARY
90	24000
90	17000
90	17000
60	9000
60	6000
60	4200
50	5800
50	3500
50	3100
50	2600
50	2500
80	10500
80	11000
80	8600
	7000
10	4400
...	

20 rows selected.

表 EMPLOYEES
中的工资最大值

MAX(SALARY)
24000

- **聚合函数类型**

- AVG()
- SUM()
- MAX()
- MIN()
- COUNT()

- **聚合函数语法**

```
SELECT      [column,] group function(column), ...
FROM        table
[WHERE      condition]
[GROUP BY   column]
[ORDER BY   column];
```

1.1 AVG和SUM函数

可以对**数值型数据**使用AVG 和 SUM 函数。

```
SELECT AVG(salary), MAX(salary), MIN(salary), SUM(salary)
FROM employees
WHERE job_id LIKE '%REP%';
```

AVG(SALARY)	MAX(SALARY)	MIN(SALARY)	SUM(SALARY)
8150	11000	6000	32600

1.2 MIN和MAX函数

可以对**任意数据类型**的数据使用 MIN 和 MAX 函数。

```
SELECT MIN(hire_date), MAX(hire_date)
FROM employees;
```

MIN(HIRE_	MAX(HIRE_
17-JUN-87	29-JAN-00

1.3 COUNT函数

- COUNT(*)返回表中记录总数，适用于**任意数据类型**。

```
SELECT COUNT(*)
FROM employees
WHERE department_id = 50;
```

COUNT()
5

- COUNT(expr) 返回**expr不为空**的记录总数。

```
SELECT COUNT(commission_pct)
FROM employees
WHERE department_id = 50;
```

COUNT(COMMISSION_PCT)
3

- 问题：用count(*), count(1), count(列名)谁好呢？**

其实，对于MyISAM引擎的表是没有区别的。这种引擎内部有一计数器在维护着行数。

Innodb引擎的表用count(*),count(1)直接读行数，复杂度是O(n)，因为innodb真的要去数一遍。但好于具体的count(列名)。

- 问题：能不能使用count(列名)替换count(*)？**

不要使用 count(列名)来替代 **count(*)**， **count(*)** 是 SQL92 定义的标准统计行数的语法，跟数据库无关，跟 NULL 和非 NULL 无关。

说明：count(*)会统计值为 NULL 的行，而 count(列名)不会统计此列为 NULL 值的行。

2.1 基本使用

EMPLOYEES

DEPARTMENT_ID	SALARY
10	4400
20	9500
20	6000
50	3500
50	3100
50	2500
50	2600
60	6400
60	9000
60	6000
60	4200
80	10033
80	10500
80	8600
80	11000
90	19333
90	24000
90	17000

...
20 rows selected.

DEPARTMENT_ID	AVG(SALARY)
10	4400
20	9500
50	3500
60	6400
80	10033.3333
90	19333.3333
110	10150
	7000

可以使用**GROUP BY**子句将表中的数据分成若干组

```
SELECT column, group_function(column)
FROM table
[WHERE condition]
[GROUP BY group_by_expression]
[ORDER BY column];
```

明确： WHERE一定放在FROM后面

在SELECT列表中所有未包含在组函数中的列都应该包含在 GROUP BY子句中

```
SELECT department_id, AVG(salary)
FROM employees
GROUP BY department_id ;
```

DEPARTMENT_ID	AVG(SALARY)
10	4400
20	9500
50	3500
60	6400
80	10033.3333
90	19333.3333
110	10150
	7000

8 rows selected.

包含在 GROUP BY 子句中的列不必包含在SELECT 列表中

```
GROUP BY department_id ;
```

AVG(SALARY)
4400
9500
3500
6400
10033.3333
19333.3333
10150
7000

2.2 使用多个列分组

EMPLOYEES

DEPARTMENT_ID	JOB_ID	SALARY
90	AD_PRES	24000
90	AD_VP	17000
90	AD_VP	17000
60	IT_PROG	9000
60	IT_PROG	6000
60	IT_PROG	4200
50	ST_MAN	5800
50	ST_CLERK	3500
50	ST_CLERK	3100
50	ST_CLERK	2600
50	ST_CLERK	2500
80	SA_MAN	10500
80	SA_REP	11000
80	SA_REP	8600
...		
20	MK_REP	6000
110	AC_MGR	12000
110	AC_ACCOUNT	8300

20 rows selected.

使用多个列
进行分组

DEPARTMENT_ID	JOB_ID	SUM(SALARY)
10	AD_ASST	4400
20	MK_MAN	13000
20	MK_REP	6000
50	ST_CLERK	11700
50	ST_MAN	5800
60	IT_PROG	19200
80	SA_MAN	10500
80	SA_REP	19600
90	AD_PRES	24000
90	AD_VP	34000
110	AC_ACCOUNT	8300
110	AC_MGR	12000
	SA_REP	7000

13 rows selected.

```
SELECT department_id dept_id, job_id, SUM(salary)
FROM employees
GROUP BY department_id, job_id ;
```

DEPT_ID	JOB_ID	SUM(SALARY)
10	AD_ASST	4400
20	MK_MAN	13000
20	MK_REP	6000
50	ST_CLERK	11700
50	ST_MAN	5800
60	IT_PROG	19200
80	SA_MAN	10500
80	SA_REP	19600
90	AD_PRES	24000
90	AD_VP	34000
110	AC_ACCOUNT	8300
110	AC_MGR	12000
	SA_REP	7000

13 rows selected.

使用 `WITH ROLLUP` 关键字之后，在所有查询出的分组记录之后增加一条记录，该记录计算查询出的所有记录的总和，即统计记录数量。

```
SELECT department_id, AVG(salary)
FROM employees
WHERE department_id > 80
GROUP BY department_id WITH ROLLUP;
```

注意：

当使用ROLLUP时，不能同时使用ORDER BY子句进行结果排序，即ROLLUP和ORDER BY是互相排斥的。

3. HAVING

3.1 基本使用

EMPLOYEES

DEPARTMENT_ID	SALARY
90	24000
90	17000
90	17000
60	9000
60	6000
60	4200
50	5800
50	3500
50	3100
50	2600
50	2500
80	10500
80	11000
80	8600
...	
20	6000
110	12000
110	8300

20 rows selected.

部门最高工资
比 10000 高的
部门

DEPARTMENT_ID	MAX(SALARY)
20	13000
80	11000
90	24000
110	12000

过滤分组：HAVING子句

1. 行已经被分组。
2. 使用了聚合函数。
3. 满足 HAVING 子句中条件的分组将被显示。
4. HAVING 不能单独使用，必须要跟 GROUP BY 一起使用。

```
FROM      table  
[WHERE    condition]  
[GROUP BY group_by_expression]  
[HAVING  group_condition]  
[ORDER BY column];
```

```
SELECT  department_id, MAX(salary)  
FROM    employees  
GROUP BY department_id  
HAVING  MAX(salary)>10000 ;
```

DEPARTMENT_ID	MAX(SALARY)
20	13000
80	11000
90	24000
110	12000

- 非法使用聚合函数：不能在 WHERE 子句中使用聚合函数。如下：

```
SELECT  department_id, AVG(salary)  
FROM    employees  
WHERE   AVG(salary) > 8000  
GROUP BY department_id;
```

```
WHERE  AVG(salary) > 8000  
*  
ERROR at line 3:  
ORA-00934: group function is not allowed here
```

3.2 WHERE和HAVING的对比

区别1： WHERE 可以直接使用表中的字段作为筛选条件，但不能使用分组中的计算函数作为筛选条件； HAVING 必须要与 GROUP BY 配合使用，可以把分组计算的函数和分组字段作为筛选条件。

这决定了，在需要对数据进行分组统计的时候， HAVING 可以完成 WHERE 不能完成的任务。这是因为，在查询语法结构中， WHERE 在 GROUP BY 之前，所以无法对分组结果进行筛选。 HAVING 在 GROUP BY 之后，可以使用分组字段和分组中的计算函数，对分组的结果集进行筛选，这个功能是 WHERE 无法完成的。另外， WHERE 排除的记录不再包括在分组中。

区别2：如果需要通过连接从关联表中获取需要的数据， WHERE 是先筛选后连接，而 HAVING 是先连接后筛选。这一点，就决定了在关联查询中， WHERE 比 HAVING 更高效。因为 WHERE 可以先筛选，用一个筛选后的较小数据集和关联表进行连接，这样占用的资源比较少，执行效率也比较高。 HAVING 则需要先把结果集准备好，也就是用未被筛选的数据集进行关联，然后对这个大的数据集进行筛选，这样占用的资源就比较多，执行效率也较低。

小结如下：

WHERE	先筛选数据再关联，执行效率高	不能使用分组中的计算函数进行筛选
HAVING	可以使用分组中的计算函数	在最后的结果集中进行筛选，执行效率较低

开发中的选择：

WHERE 和 HAVING 也不是互相排斥的，我们可以在一个查询里面同时使用 WHERE 和 HAVING。包含分组统计函数的条件用 HAVING，普通条件用 WHERE。这样，我们就既利用了 WHERE 条件的高效快速，又发挥了 HAVING 可以使用包含分组统计函数的查询条件的优点。当数据量特别大的时候，运行效率会有很大的差别。

4. SELECT的执行过程

4.1 查询的结构

```
#方式1:  
SELECT ...  
FROM ...  
WHERE 多表的连接条件  
AND 不包含组函数的过滤条件  
GROUP BY ...  
HAVING 包含组函数的过滤条件  
ORDER BY ... ASC/DESC  
LIMIT ...  
  
#方式2:  
SELECT ...  
FROM ... JOIN ...  
ON 多表的连接条件  
JOIN ...  
ON ...  
WHERE 不包含组函数的过滤条件  
AND/OR 不包含组函数的过滤条件  
GROUP BY ...  
HAVING 包含组函数的过滤条件  
ORDER BY ... ASC/DESC  
LIMIT ...  
  
#其中：  
# (1) from: 从哪些表中筛选  
# (2) on: 关联多表查询时，去除笛卡尔积  
# (3) where: 从表中筛选的条件  
# (4) group by: 分组依据  
# (5) having: 在统计结果中再次筛选  
# (6) order by: 排序  
# (7) limit: 分页
```

你需要记住 SELECT 查询时的两个顺序：

1. 关键字的顺序是不能颠倒的：

```
SELECT ... FROM ... WHERE ... GROUP BY ... HAVING ... ORDER BY ... LIMIT...
```

2. SELECT 语句的执行顺序 (在 MySQL 和 Oracle 中, SELECT 执行顺序基本相同) :

```
FROM -> WHERE -> GROUP BY -> HAVING -> SELECT 的字段 -> DISTINCT -> ORDER BY -> LIMIT
```

```
1 FROM <left_table>
2 ON <join_condition>
3 <join_type> JOIN <right_table>
4 WHERE <where_condition>
5 GROUP BY <group_by_list>
6 HAVING <having_condition>
7 SELECT
8 DISTINCT <select_list>
9 ORDER BY <order_by_condition>
10 LIMIT <limit_number>
```

比如你写了一个 SQL 语句, 那么它的关键字顺序和执行顺序是下面这样的:

```
SELECT DISTINCT player_id, player_name, count(*) as num # 顺序 5
FROM player JOIN team ON player.team_id = team.team_id # 顺序 1
WHERE height > 1.80 # 顺序 2
GROUP BY player.team_id # 顺序 3
HAVING num > 2 # 顺序 4
ORDER BY num DESC # 顺序 6
LIMIT 2 # 顺序 7
```

在 SELECT 语句执行这些步骤的时候, 每个步骤都会产生一个 [虚拟表](#), 然后将这个虚拟表传入下一个步骤中作为输入。需要注意的是, 这些步骤隐含在 SQL 的执行过程中, 对于我们来说是不可见的。

4.3 SQL 的执行原理

SELECT 是先执行 FROM 这一步的。在这个阶段, 如果是多张表联查, 还会经历下面的几个步骤:

1. 首先先通过 CROSS JOIN 求笛卡尔积, 相当于得到虚拟表 vt (virtual table) 1-1;
2. 通过 ON 进行筛选, 在虚拟表 vt1-1 的基础上进行筛选, 得到虚拟表 vt1-2;
3. 添加外部行。如果我们使用的是左连接、右链接或者全连接, 就会涉及到外部行, 也就是在虚拟表 vt1-2 的基础上增加外部行, 得到虚拟表 vt1-3。

当然如果我们操作的是两张以上的表, 还会重复上面的步骤, 直到所有表都被处理完为止。这个过程得到的是我们的原始数据。

然后进入第三步和第四步，也就是 `GROUP` 和 `HAVING` 阶段。在这个阶段中，实际上是在虚拟表 `vt2` 的基础上进行分组和分组过滤，得到中间的虚拟表 `vt3` 和 `vt4`。

当我们完成了条件筛选部分之后，就可以筛选表中提取的字段，也就是进入到 `SELECT` 和 `DISTINCT` 阶段。

首先在 `SELECT` 阶段会提取想要的字段，然后在 `DISTINCT` 阶段过滤掉重复的行，分别得到中间的虚拟表 `vt5-1` 和 `vt5-2`。

当我们提取了想要的字段数据之后，就可以按照指定的字段进行排序，也就是 `ORDER BY` 阶段，得到虚拟表 `vt6`。

最后在 `vt6` 的基础上，取出指定行的记录，也就是 `LIMIT` 阶段，得到最终的结果，对应的是虚拟表 `vt7`。

当然我们在写 `SELECT` 语句的时候，不一定存在所有的关键字，相应的阶段就会省略。

同时因为 SQL 是一门类似英语的结构化查询语言，所以我们在写 `SELECT` 语句的时候，还要注意相应的关键字顺序，**所谓底层运行的原理，就是我们刚才讲到的执行顺序。**

第09章_子查询

讲师：尚硅谷·宋红康（江湖人称：康师傅）

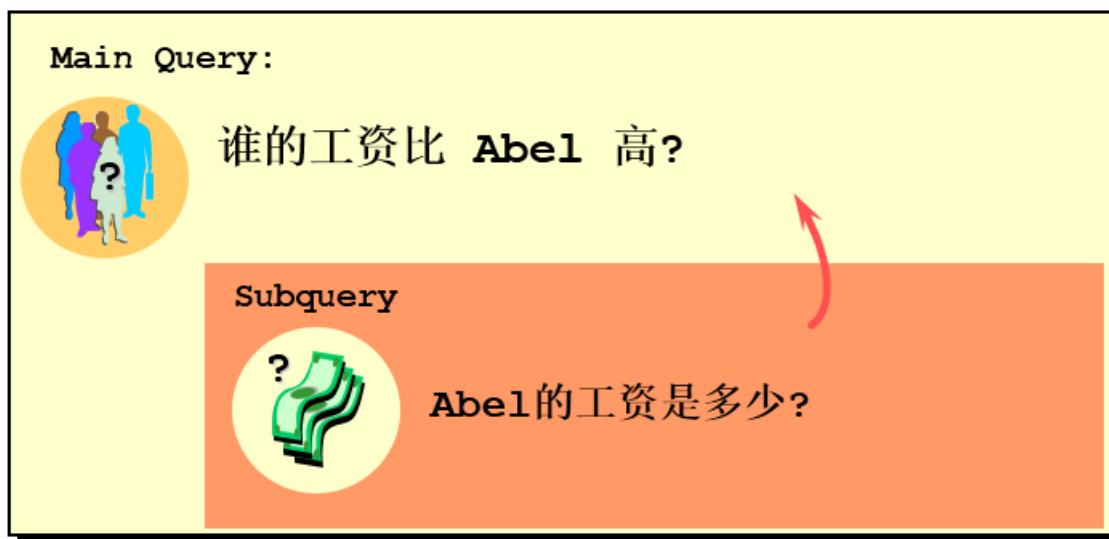
官网：<http://www.atguigu.com>

子查询指一个查询语句嵌套在另一个查询语句内部的查询，这个特性从MySQL 4.1开始引入。

SQL 中子查询的使用大大增强了 SELECT 查询的能力，因为很多时候查询需要从结果集中获取数据，或者需要从同一个表中先计算得出一个数据结果，然后与这个数据结果（可能是某个标量，也可能是某个集合）进行比较。

1. 需求分析与问题解决

1.1 实际问题



现有解决方式：

```
#方式一:  
SELECT salary  
FROM employees  
WHERE last_name = 'Abel';
```

```
SELECT last_name,salary  
FROM employees  
WHERE salary > 11000;
```

```
#方式二：自连接  
SELECT e2.last_name,e2.salary  
FROM employees e1,employees e2  
WHERE e1.last_name = 'Abel'  
AND e1.`salary` < e2.`salary`
```

```
FROM employees
WHERE salary > (
    SELECT salary
    FROM employees
    WHERE last_name = 'Abel'
);
```

LAST_NAME
King
Kochhar
De Haan
Hartstein
Higgins

1.2 子查询的基本使用

- 子查询的基本语法结构：

```
SELECT      select_list
FROM        table
WHERE       expr operator
            (SELECT      select_list
             FROM       table);
```

- 子查询（内查询）在主查询之前一次执行完成。
- 子查询的结果被主查询（外查询）使用。
- 注意事项**
 - 子查询要包含在括号内
 - 将子查询放在比较条件的右侧
 - 单行操作符对应单行子查询，多行操作符对应多行子查询

1.3 子查询的分类

分类方式1：

我们按内查询的结果返回一条还是多条记录，将子查询分为 **单行子查询**、**多行子查询**。

- 单行子查询



- 多行子查询



我们按内查询是否被执行多次，将子查询划分到 相关(或关联)子查询 和 不相关(或非关联)子查询。

子查询从数据表中查询了数据结果，如果这个数据结果只执行一次，然后这个数据结果作为主查询的条件进行执行，那么这样的子查询叫做不相关子查询。

同样，如果子查询需要执行多次，即采用循环的方式，先从外部查询开始，每次都传入子查询进行查询，然后再将结果反馈给外部，这种嵌套的执行方式就称为相关子查询。

2. 单行子查询

2.1 单行比较操作符

操作符	含义
=	equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
<>	not equal to

2.2 代码示例

题目：查询工资大于149号员工工资的员工的信息

```
SELECT last_name
FROM employees
WHERE salary > 10500
      ↑
      (SELECT salary
       FROM employees
       WHERE employee_id = 149) ;
```

LAST_NAME
King
Kochhar
De Haan
Abel
Hartstein
Higgins

6 rows selected.

题目：返回job_id与141号员工相同，salary比143号员工多的员工姓名，job_id和工资

```

WHERE job_id =
      (SELECT job_id
       FROM employees
       WHERE employee_id = 141)
AND salary >
      (SELECT salary
       FROM employees
       WHERE employee_id = 143);
    
```

LAST_NAME	JOB_ID	SALARY
Rajs	ST_CLERK	3500
Davies	ST_CLERK	3100

题目：返回公司工资最少的员工的last_name,job_id和salary

```

SELECT last_name, job_id, salary
FROM employees
WHERE salary =
      (SELECT MIN(salary)
       FROM employees);
    
```

LAST_NAME	JOB_ID	SALARY
Vargas	ST_CLERK	2500

题目：查询与141号或174号员工的manager_id和department_id相同的其他员工的employee_id, manager_id, department_id

实现方式1：不成对比较

```

SELECT employee_id, manager_id, department_id
FROM employees
WHERE manager_id IN
      (SELECT manager_id
       FROM employees
       WHERE employee_id IN (174, 141))
AND department_id IN
      (SELECT department_id
       FROM employees
       WHERE employee_id IN (174, 141))
AND employee_id NOT IN(174, 141);
    
```

实现方式2：成对比较

```

SELECT employee_id, manager_id, department_id
FROM employees
WHERE (manager_id, department_id) IN
      (SELECT manager_id, department_id
       FROM employees
       WHERE employee_id IN (141, 174))
AND employee_id NOT IN (141, 174);
    
```

- 首先执行子查询。
- 向主查询中的HAVING 子句返回结果。

题目：查询最低工资大于50号部门最低工资的部门id和其最低工资

```
SELECT department_id, MIN(salary)
FROM employees
GROUP BY department_id
HAVING MIN(salary) >
       (SELECT MIN(salary)
        FROM employees
        WHERE department_id = 50);
```

2.4 CASE中的子查询

在CASE表达式中使用单列子查询：

题目：显式员工的employee_id,last_name和location。其中，若员工department_id与location_id为1800的department_id相同，则location为'Canada'，其余则为'USA'。

```
SELECT employee_id, last_name,
(CASE department_id
WHEN
    (SELECT department_id FROM departments
     WHERE location_id = 1800)
    THEN 'Canada' ELSE 'USA' END) location
FROM employees;
```

2.5 子查询中的空值问题

```
SELECT last_name, job_id
FROM employees
WHERE job_id =
      (SELECT job_id
       FROM employees
       WHERE last_name = 'Haas');
```

no rows selected

子查询不返回任何行

2.5 非法使用子查询

```
SELECT employee_id, last_name
FROM employees
WHERE salary =
      (SELECT MIN(salary)
       FROM employees
       GROUP BY department_id);
```

Subquery returns more than 1 row

多行子查询使用单行比较符

3. 多行子查询

- 也称为集合比较子查询
- 内查询返回多行
- 使用多行比较操作符

3.1 多行比较操作符

操作符	含义
IN	等于列表中的任意一个
ANY	需要和单行比较操作符一起使用，和子查询返回的某一个值比较
ALL	需要和单行比较操作符一起使用，和子查询返回的所有值比较
SOME	实际上是ANY的别名，作用相同，一般常使用ANY

体会 ANY 和 ALL 的区别

3.2 代码示例

题目：返回其它job_id中比job_id为‘IT_PROG’部门任一工资低的员工的员工号、姓名、job_id 以及salary

```

SELECT employee_id, last_name, job_id, salary
FROM   employees      9000, 6000, 4800, 4200
WHERE  salary < ANY
       (SELECT salary
        FROM   employees
        WHERE  job_id = 'IT_PROG')
AND    job_id <> 'IT_PROG';

```

EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
124	Mourgos	ST_MAN	5800
141	Rajs	ST_CLERK	3500
142	Davies	ST_CLERK	3100
143	Matos	ST_CLERK	2600
144	Vargas	ST_CLERK	2500

10 rows selected.

题目：返回其它job_id中比job_id为‘IT_PROG’部门所有工资都低的员工的员工号、姓名、job_id 以及 salary

```

FROM employees
WHERE salary < ALL (
    SELECT salary
    FROM employees
    WHERE job_id = 'IT_PROG')
AND job_id <> 'IT_PROG';

```

EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
141	Rajs	ST_CLERK	3500
142	Davies	ST_CLERK	3100
143	Matos	ST_CLERK	2600
144	Vargas	ST_CLERK	2500

题目：查询平均工资最低的部门id

```

#方式1:
SELECT department_id
FROM employees
GROUP BY department_id
HAVING AVG(salary) = (
    SELECT MIN(avg_sal)
    FROM (
        SELECT AVG(salary) avg_sal
        FROM employees
        GROUP BY department_id
    ) dept_avg_sal
)

```

```

#方式2:
SELECT department_id
FROM employees
GROUP BY department_id
HAVING AVG(salary) <= ALL (
    SELECT AVG(salary) avg_sal
    FROM employees
    GROUP BY department_id
)

```

3.3 空值问题

```

SELECT last_name
FROM employees
WHERE employee_id NOT IN (
    SELECT manager_id
    FROM employees
);

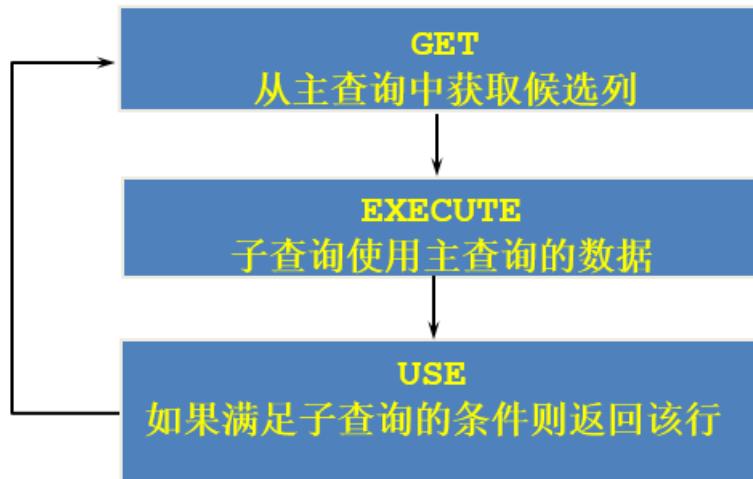
```

| no rows selected |

4. 相关子查询

如果子查询的执行依赖于外部查询，通常情况下都是因为子查询中的表用到了外部的表，并进行了条件关联，因此每执行一次外部查询，子查询都要重新计算一次，这样的子查询就称之为 **关联子查询**。

相关子查询按照一行接一行的顺序执行，主查询的每一行都执行一次子查询。



```

SELECT column1, column2, ...
FROM   table1 [outer]
WHERE  column1 operator
       (SELECT column1, column2
        FROM   table2
        WHERE  expr1 =
               [outer].expr2);

```

说明：子查询中使用主查询中的列

4.2 代码示例

题目：查询员工中工资大于本部门平均工资的员工的last_name,salary和其department_id

方式一：相关子查询

```

SELECT last_name, salary, department_id
FROM   employees outer
WHERE  salary >
       (SELECT AVG(salary)
        FROM   employees
        WHERE  department_id =
               [outer].department_id);

```

方式二：在 FROM 中使用子查询

```

SELECT last_name, salary, e1.department_id
FROM   employees e1, (SELECT department_id, AVG(salary) dept_avg_sal FROM employees GROUP
BY department_id) e2
WHERE  e1.`department_id` = e2.department_id
AND   e2.dept_avg_sal < e1.`salary`;

```

名，把它当成一张“临时的虚拟的表”来使用。

在ORDER BY 中使用子查询：

题目：查询员工的id,salary,按照department_name 排序

```
SELECT employee_id,salary
FROM employees e
ORDER BY (
    SELECT department_name
    FROM departments d
    WHERE e.department_id = d.department_id
);
```

题目：若employees表中employee_id与job_history表中employee_id相同的数目不小于2，输出这些相同id的员工的employee_id,last_name和其job_id

```
SELECT e.employee_id, last_name,e.job_id
FROM   employees e
WHERE  2 <= (SELECT COUNT(*)
               FROM   job_history
               WHERE  employee_id = e.employee_id);
```

4.3 EXISTS 与 NOT EXISTS关键字

- 关联子查询通常也会和 EXISTS操作符一起来使用，用来检查在子查询中是否存在满足条件的行。
- 如果在子查询中不存在满足条件的行：**
 - 条件返回 FALSE
 - 继续在子查询中查找
- 如果在子查询中存在满足条件的行：**
 - 不在子查询中继续查找
 - 条件返回 TRUE
- NOT EXISTS关键字表示如果不存在某种条件，则返回TRUE，否则返回FALSE。

题目：查询公司管理者的employee_id, last_name, job_id, department_id信息

方式一：

```
SELECT employee_id, last_name, job_id, department_id
FROM   employees e1
WHERE  EXISTS ( SELECT *
                 FROM   employees e2
                 WHERE  e2.manager_id =
                        e1.employee_id);
```

方式二：自连接

```
SELECT DISTINCT e1.employee_id, e1.last_name, e1.job_id, e1.department_id
FROM   employees e1 JOIN employees e2
WHERE  e1.employee_id = e2.manager_id;
```

方式三：

```
WHERE employee_id IN (
    SELECT DISTINCT manager_id
    FROM employees

);
```

题目：查询departments表中，不存在于employees表中的部门的department_id和department_name

```
SELECT department_id, department_name
FROM departments d
WHERE NOT EXISTS (SELECT 'X'
                   FROM employees
                   WHERE department_id = d.department_id);
```

DEPARTMENT_ID	DEPARTMENT_NAME
190	Contracting

4.4 相关更新

```
UPDATE table1 alias1
SET column = (SELECT expression
               FROM table2 alias2
               WHERE alias1.column = alias2.column);
```

使用相关子查询依据一个表中的数据更新另一个表的数据。

题目：在employees中增加一个department_name字段，数据为员工对应的部门名称

```
# 1)
ALTER TABLE employees
ADD(department_name VARCHAR2(14));

# 2)
UPDATE employees e
SET department_name = (SELECT department_name
                       FROM departments d
                       WHERE e.department_id = d.department_id);
```

4.4 相关删除

```
DELETE FROM table1 alias1
WHERE column operator (SELECT expression
                        FROM table2 alias2
                        WHERE alias1.column = alias2.column);
```

使用相关子查询依据一个表中的数据删除另一个表的数据。

题目：删除表employees中，其与emp_history表皆有的数据

```
DELETE FROM employees e
WHERE employee_id in
    (SELECT employee_id
     FROM emp_history
     WHERE employee_id = e.employee_id);
```

问题：谁的工资比Abel的高？

解答：

#方式1：自连接

```
SELECT e2.last_name,e2.salary  
FROM employees e1,employees e2  
WHERE e1.last_name = 'Abel'  
AND e1.`salary` < e2.`salary`
```

#方式2：子查询

```
SELECT last_name,salary  
FROM employees  
WHERE salary > (  
    SELECT salary  
    FROM employees  
    WHERE last_name = 'Abel'  
) ;
```

问题：以上两种方式有好坏之分吗？

解答：自连接方式好！

题目中可以使用子查询，也可以使用自连接。一般情况建议你使用自连接，因为在许多 DBMS 的处理过程中，对于自连接的处理速度要比子查询快得多。

可以这样理解：子查询实际上是通过未知表进行查询后的条件判断，而自连接是通过已知的自身数据表进行条件判断，因此在大部分 DBMS 中都对自连接处理进行了优化。

第10章_创建和管理表

讲师：尚硅谷·宋红康（江湖人称：康师傅）

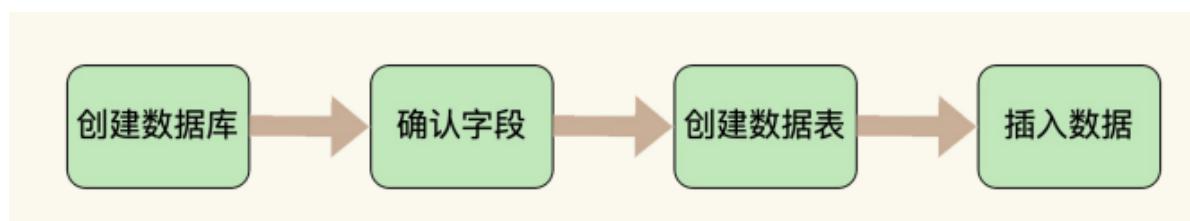
官网：<http://www.atguigu.com>

1. 基础知识

1.1 一条数据存储的过程

存储数据是处理数据的第一步。只有正确地把数据存储起来，我们才能进行有效的处理和分析。否则，只能是一团乱麻，无从下手。

那么，怎样才能把用户各种经营相关的、纷繁复杂的数据，有序、高效地存储起来呢？在 MySQL 中，一个完整的数据存储过程总共有 4 步，分别是创建数据库、确认字段、创建数据表、插入数据。



我们要先创建一个数据库，而不是直接创建数据表呢？

因为从系统架构的层次上看，MySQL 数据库系统从大到小依次是 数据库服务器、数据库、数据表、数据表的 行与列。

MySQL 数据库服务器之前已经安装。所以，我们就从创建数据库开始。

1.2 标识符命名规则

- 数据库名、表名不得超过30个字符，变量名限制为29个
- 必须只能包含 A-Z, a-z, 0-9, _ 共63个字符
- 数据库名、表名、字段名等对象名中间不要包含空格
- 同一个MySQL软件中，数据库不能同名；同一个库中，表不能重名；同一个表中，字段不能重名
- 必须保证你的字段没有和保留字、数据库系统或常用方法冲突。如果坚持使用，请在SQL语句中使用`（着重号）引起来
- 保持字段名和类型的一致性：在命名字段并为其指定数据类型的时候一定要保证一致性，假如数据类型在一个表里是整数，那在另一个表里可就别变成字符串了

1.3 MySQL中的数据类型

整数类型	TINYINT、SMALLINT、MEDIUMINT、 INT(或INTEGER) 、BIGINT
浮点类型	FLOAT、DOUBLE
定点数类型	DECIMAL
位类型	BIT
日期时间类型	YEAR、TIME、 DATE 、DATETIME、TIMESTAMP
文本字符串类型	CHAR、 VARCHAR 、TINYTEXT、TEXT、MEDIUMTEXT、LONGTEXT
枚举类型	ENUM
集合类型	SET
二进制字符串类型	BINARY、VARBINARY、TINYBLOB、BLOB、MEDIUMBLOB、LONGBLOB
JSON类型	JSON对象、JSON数组
空间数据类型	单值：GEOMETRY、POINT、LINESTRING、POLYGON； 集合：MULTIPOINT、MULTILINESTRING、MULTIPOLYGON、GEOMETRYCOLLECTION

其中，常用的几类类型介绍如下：

数据类型	描述
INT	从-2^31到2^31-1的整型数据。存储大小为4个字节
CHAR(size)	定长字符数据。若未指定，默认为1个字符，最大长度255
VARCHAR(size)	可变长字符数据，根据字符串实际长度保存， 必须指定长度
FLOAT(M,D)	单精度，占用4个字节，M=整数位+小数位，D=小数位。 D<=M<=255,0<=D<=30， 默认M+D<=6
DOUBLE(M,D)	双精度，占用8个字节，D<=M<=255,0<=D<=30， 默认M+D<=15
DECIMAL(M,D)	高精度小数，占用M+2个字节，D<=M<=65, 0<=D<=30，最大取值范围与DOUBLE相同。
DATE	日期型数据，格式'YYYY-MM-DD'
BLOB	二进制形式的长文本数据，最大可达4G
TEXT	长文本数据，最大可达4G

2. 创建和管理数据库

- 方式1：创建数据库

```
CREATE DATABASE 数据库名;
```

- 方式2：创建数据库并指定字符集

```
CREATE DATABASE 数据库名 CHARACTER SET 字符集;
```

- 方式3：判断数据库是否存在，不存在则创建数据库（推荐）

```
CREATE DATABASE IF NOT EXISTS 数据库名;
```

如果MySQL中已经存在相关的数据库，则忽略创建语句，不再创建数据库。

注意：DATABASE 不能改名。一些可视化工具可以改名，它是建新库，把所有表复制到新库，再删旧库完成的。

2.2 使用数据库

- 查看当前所有的数据库

```
SHOW DATABASES; #有一个S，代表多个数据库
```

- 查看当前正在使用的数据库

```
SELECT DATABASE(); #使用的一个 mysql 中的全局函数
```

- 查看指定库下所有的表

```
SHOW TABLES FROM 数据库名;
```

- 查看数据库的创建信息

```
SHOW CREATE DATABASE 数据库名;
```

或者：

```
SHOW CREATE DATABASE 数据库名\G
```

- 使用/切换数据库

```
USE 数据库名;
```

注意：要操作表格和数据之前必须先说明是对哪个数据库进行操作，否则就要对所有对象加上“数据库名.”。

2.3 修改数据库

- 更改数据库字符集

```
ALTER DATABASE 数据库名 CHARACTER SET 字符集; #比如：gbk、utf8等
```

- 方式1：删除指定的数据库

```
DROP DATABASE 数据库名;
```

- 方式2：删除指定的数据库（推荐）

```
DROP DATABASE IF EXISTS 数据库名;
```

3. 创建表

3.1 创建方式1

- 必须具备：

- CREATE TABLE权限
- 存储空间

- 语法格式：

```
CREATE TABLE [IF NOT EXISTS] 表名(  
    字段1, 数据类型 [约束条件] [默认值],  
    字段2, 数据类型 [约束条件] [默认值],  
    字段3, 数据类型 [约束条件] [默认值],  
    ....  
    [表约束条件]  
) ;
```

加上了IF NOT EXISTS关键字，则表示：如果当前数据库中不存在要创建的数据表，则创建数据表；如果当前数据库中已经存在要创建的数据表，则忽略建表语句，不再创建数据表。

- 必须指定：

- 表名
- 列名(或字段名)，数据类型，**长度**

- 可选指定：

- 约束条件
- 默认值

- 创建表举例1：

```
-- 创建表  
CREATE TABLE emp (  
    -- int类型  
    emp_id INT,  
    -- 最多保存20个中英文字符  
    emp_name VARCHAR(20),  
    -- 总位数不超过15位  
    salary DOUBLE,  
    -- 日期类型  
    birthday DATE  
) ;
```

```
DESC emp;
```

emp_name	varchar(20)	11B	YES	(NULL)	OK	
salary	double	6B	YES	(NULL)	OK	
birthday	date	4B	YES	(NULL)	OK	

MySQL在执行建表语句时，将id字段的类型设置为int(11)，这里的11实际上是int类型指定的显示宽度，默认的显示宽度为11。也可以在创建数据表的时候指定数据的显示宽度。

- 创建表举例2：

```
CREATE TABLE dept(
    -- int类型，自增
    deptno INT(2) AUTO_INCREMENT,
    dname VARCHAR(14),
    loc VARCHAR(13),
    -- 主键
    PRIMARY KEY (deptno)
);
```

```
DESCRIBE dept;
```

Field	Type	Null	Key	Default	Extra
deptno	int(2)	6B	NO	PRI	(NULL) OK auto_increment
dname	varchar(14)	11B	YES		(NULL) OK
loc	varchar(13)	11B	YES		(NULL) OK

在MySQL 8.x版本中，不再推荐为INT类型指定显示长度，并在未来的版本中可能去掉这样的语法。

3.2 创建方式2

- 使用 AS subquery 选项，**将创建表和插入数据结合起来**

```
CREATE TABLE table
    [(column, column...)]
AS subquery;
```

- 指定的列和子查询中的列要一一对应
- 通过列名和默认值定义列

```
CREATE TABLE emp1 AS SELECT * FROM employees;
```

```
CREATE TABLE emp2 AS SELECT * FROM employees WHERE 1=2; -- 创建的emp2是空表
```

```
CREATE TABLE dept80
AS
SELECT employee_id, last_name, salary*12 ANNSAL, hire_date
FROM employees
WHERE department_id = 80;
```

```
DESCRIBE dept80;
```

LAST_NAME	NOT NULL	VARCHAR2(25)
ANNSAL		NUMBER
HIRE_DATE	NOT NULL	DATE

34 rows selected

3.3 查看数据表结构

在MySQL中创建好数据表之后，可以查看数据表的结构。MySQL支持使用 **DESCRIBE/DESC** 语句查看数据表结构，也支持使用 **SHOW CREATE TABLE** 语句查看数据表结构。

语法格式如下：

```
SHOW CREATE TABLE 表名\G
```

使用**SHOW CREATE TABLE**语句不仅可以查看表创建时的详细语句，还可以查看存储引擎和字符编码。

4. 修改表

修改表指的是修改数据库中已经存在的数据表的结构。

使用 `ALTER TABLE` 语句可以实现：

- 向已有的表中添加列
- 修改现有表中的列
- 删除现有表中的列
- 重命名现有表中的列

4.1 追加一个列

语法格式如下：

```
ALTER TABLE 表名 ADD 【COLUMN】 字段名 字段类型 【FIRST|AFTER 字段名】;
```

举例：

```
ALTER TABLE dept80
ADD job_id varchar(15);
```

DEPT100

EMPLOYEE_ID	LAST_NAME	ANNSAL	HIRE_DATE
149	Zlotkey	126000	29-JAN-00
174	Abel	132000	11-MAY-96
176	Taylor	103200	24-MAR-98

JOB_ID

追加一个新列

DEPT80

EMPLOYEE_ID	LAST_NAME	ANNSAL	HIRE_DATE
149	Zlotkey	126000	29-JAN-00
174	Abel	132000	11-MAY-96
176	Taylor	103200	24-MAR-98

JOB_ID

4.2 修改一个列

- 可以修改列的数据类型，长度、默认值和位置
- 修改字段数据类型、长度、默认值、位置的语法格式如下：

```
ALTER TABLE 表名 MODIFY 【COLUMN】 字段名1 字段类型 【DEFAULT 默认值】 【FIRST|AFTER 字段名2】 ;
```

- 举例：

```
ALTER TABLE dept80
MODIFY last_name VARCHAR(30);
```

```
ALTER TABLE dept80
MODIFY salary double(9,2) default 1000;
```

- 对默认值的修改只影响今后对表的修改
- 此外，还可以通过此种方式修改列的约束。这里暂先不讲。

4.3 重命名一个列

使用 CHANGE old_column new_column dataType 子句重命名列。语法格式如下：

```
ALTER TABLE 表名 CHANGE 【column】 列名 新列名 新数据类型;
```

举例：

```
ALTER TABLE dept80
CHANGE department_name dept_name varchar(15);
```

4.4 删除一个列

删除表中某个字段的语法格式如下：

```
ALTER TABLE 表名 DROP 【COLUMN】 字段名
```

举例：

```
ALTER TABLE dept80
DROP COLUMN ...;
```

- 方式一：使用RENAME

```
RENAME TABLE emp  
TO myemp;
```

- 方式二：

```
ALTER table dept  
RENAME [TO] detail_dept; -- [TO]可以省略
```

- 必须是对象的拥有者

6. 删除表

- 在MySQL中，当一张数据表 没有与其他任何数据表形成关联关系 时，可以将当前数据表直接删除。
- 数据和结构都被删除
- 所有正在运行的相关事务被提交
- 所有相关索引被删除
- 语法格式：

```
DROP TABLE [IF EXISTS] 数据表1 [, 数据表2, ..., 数据表n];
```

IF EXISTS 的含义为：如果当前数据库中存在相应的数据表，则删除数据表；如果当前数据库中不存在相应的数据表，则忽略删除语句，不再执行删除数据表的操作。

- 举例：

```
DROP TABLE dept80;
```

- DROP TABLE 语句不能回滚

7. 清空表

- TRUNCATE TABLE语句：
 - 删除表中所有的数据
 - 释放表的存储空间
- 举例：

```
TRUNCATE TABLE detail_dept;
```

- TRUNCATE语句**不能回滚**，而使用 DELETE 语句删除数据，可以回滚
- 对比：

```
DELETE FROM emp2;
#TRUNCATE TABLE emp2;

SELECT * FROM emp2;

ROLLBACK;

SELECT * FROM emp2;
```

阿里开发规范：

【参考】TRUNCATE TABLE 比 DELETE 速度快，且使用的系统和事务日志资源少，但 TRUNCATE 无事务且不触发 TRIGGER，有可能造成事故，故不建议在开发代码中使用此语句。

说明：TRUNCATE TABLE 在功能上与不带 WHERE 子句的 DELETE 语句相同。

8. 内容拓展

拓展1：阿里巴巴《Java开发手册》之MySQL字段命名

- 【强制】表名、字段名必须使用小写字母或数字，禁止出现数字开头，禁止两个下划线中间只出现数字。数据库字段名的修改代价很大，因为无法进行预发布，所以字段名称需要慎重考虑。
 - 正例：aliyun_admin, rdc_config, level3_name
 - 反例：AliyunAdmin, rdcConfig, level_3_name
- 【强制】禁用保留字，如 desc、range、match、delayed 等，请参考 MySQL 官方保留字。
- 【强制】表必备三字段：id, gmt_create, gmt_modified。
 - 说明：其中 id 必为主键，类型为BIGINT UNSIGNED、单表时自增、步长为 1。gmt_create, gmt_modified 的类型均为 DATETIME 类型，前者现在时表示主动式创建，后者过去分词表示被动式更新
- 【推荐】表的命名最好是遵循“业务名称_表的作用”。
 - 正例：alipay_task、force_project、trade_config
- 【推荐】库名与应用名称尽量一致。
- 【参考】合适的字符存储长度，不但节约数据库表空间、节约索引存储，更重要的是提升检索速度。
 - 正例：无符号值可以避免误存负数，且扩大了表示范围。

对象	年龄区间	类型	字节	表示范围
人	150 岁之内	tinyint unsigned	1	无符号值：0 到 255
龟	数百岁	smallint unsigned	2	无符号值：0 到 65535
恐龙化石	数千万年	int unsigned	4	无符号值：0 到约 43 亿
太阳	约 50 亿年	bigint unsigned	8	无符号值：0 到约 10 的 19 次方

表删除 操作将把表的定义和表中的数据一起删除，并且MySQL在执行删除操作时，不会有任何的确认信息提示，因此执行删除操时应当慎重。在删除表前，最好对表中的数据进行 **备份**，这样当操作失误时可以对数据进行恢复，以免造成无法挽回的后果。

同样的，在使用 **ALTER TABLE** 进行表的基本修改操作时，在执行操作过程之前，也应该确保对数据进行完整的 **备份**，因为数据库的改变是 **无法撤销** 的，如果添加了一个不需要的字段，可以将其删除；相同的，如果删除了一个需要的列，该列下面的所有数据都将会丢失。

拓展3：MySQL8新特性—DDL的原子化

在MySQL 8.0版本中，InnoDB表的DDL支持事务完整性，即 **DDL操作要么成功要么回滚**。DDL操作回滚日志写入到data dictionary数据字典表mysql.innodb_ddl_log（该表是隐藏的表，通过show tables无法看到）中，用于回滚操作。通过设置参数，可将DDL操作日志打印输出到MySQL错误日志中。

分别在MySQL 5.7版本和MySQL 8.0版本中创建数据库和数据表，结果如下：

```
CREATE DATABASE mytest;

USE mytest;

CREATE TABLE book1(
book_id INT ,
book_name VARCHAR(255)
);

SHOW TABLES;
```

(1) 在MySQL 5.7版本中，测试步骤如下：删除数据表book1和数据表book2，结果如下：

```
mysql> DROP TABLE book1,book2;
ERROR 1051 (42S02): Unknown table 'mytest.book2'
```

再次查询数据库中的数据表名称，结果如下：

```
mysql> SHOW TABLES;
Empty set (0.00 sec)
```

从结果可以看出，虽然删除操作时报错了，但是仍然删除了数据表book1。

(2) 在MySQL 8.0版本中，测试步骤如下：删除数据表book1和数据表book2，结果如下：

```
mysql> DROP TABLE book1,book2;
ERROR 1051 (42S02): Unknown table 'mytest.book2'
```

再次查询数据库中的数据表名称，结果如下：

```
mysql> show tables;
+-----+
| Tables_in_mytest |
+-----+
| book1           |
+-----+
1 row in set (0.00 sec)
```

从结果可以看出，数据表book1并没有被删除。



让天下没有难学的技术

第11章_数据处理之增删改

讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

1. 插入数据

1.1 实际问题

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500
90	Executive	100	1700
110	Accounting	205	1700
190	Contracting		1700

新行

向 DEPARTMENTS 表中插入新的记录

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500
90	Executive	100	1700
110	Accounting	205	1700
190	Contracting		1700

70 | Public Relations | 100 | 1700

解决方式：使用 INSERT 语句向表中插入数据。

1.2 方式1：VALUES的方式添加

使用这种语法一次只能向表中插入一条数据。

情况1：为表的所有字段按默认顺序插入数据

```
INSERT INTO 表名  
VALUES (value1,value2,...);
```

值列表中需要为表的每一个字段指定值，并且值的顺序必须和数据表中字段定义时的顺序相同。

举例：

```
INSERT INTO departments  
VALUES (70, 'Pub', 100, 1700);
```

```
INSERT INTO departments  
VALUES (100, 'Finance', NULL, NULL);
```

```
INSERT INTO 表名(column1 [, column2, ..., columnn])
VALUES (value1 [,value2, ..., valuen]);
```

为表的指定字段插入数据，就是在INSERT语句中只向部分字段中插入值，而其他字段的值为表定义时的默认值。

在 INSERT 子句中随意列出列名，但是一旦列出，VALUES中要插入的value1,...valuen需要与column1,...columnn列一一对应。如果类型不同，将无法插入，并且MySQL会产生错误。

举例：

```
INSERT INTO departments(department_id, department_name)
VALUES (80, 'IT');
```

情况3：同时插入多条记录

INSERT语句可以同时向数据表中插入多条记录，插入时指定多个值列表，每个值列表之间用逗号分隔开，基本语法格式如下：

```
INSERT INTO table_name
VALUES
(value1 [,value2, ..., valuen]),
(value1 [,value2, ..., valuen]),
.....
(value1 [,value2, ..., valuen]);
```

或者

```
INSERT INTO table_name(column1 [, column2, ..., columnn])
VALUES
(value1 [,value2, ..., valuen]),
(value1 [,value2, ..., valuen]),
.....
(value1 [,value2, ..., valuen]);
```

举例：

```
mysql> INSERT INTO emp(emp_id,emp_name)
-> VALUES (1001,'shkstart'),
-> (1002,'atguigu'),
-> (1003,'Tom');
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

使用INSERT同时插入多条记录时，MySQL会返回一些在执行单行插入时没有的额外信息，这些信息的含义如下：

- Records：表明插入的记录条数。
- Duplicates：表明插入时被忽略的记录，原因可能是这些记录包含了重复的主键值。
- Warnings：表明有问题的数据值，例如发生数据类型转换。

一个同时插入多行记录的INSERT语句等同于多个单行插入的INSERT语句，但是多行的INSERT语句在处理过程中**效率更高**。因为MySQL执行单条INSERT语句插入多行数据比使用多条INSERT语句快，所以在插入多条记录时最好选择使用单条INSERT语句的方式插入。

小结：

- **VALUES** 也可以写成 **VALUE**，但是VALUES是标准写法。

INSERT还可以将SELECT语句查询的结果插入到表中，此时不需要把每一条记录的值一个一个输入，只需要使用一条INSERT语句和一条SELECT语句组成的组合语句即可快速地从一个或多个表中向一个表中插入多行。

基本语法格式如下：

```
INSERT INTO 目标表名
(tar_column1 [ , tar_column2, ..., tar_columnn])
SELECT
(src_column1 [ , src_column2, ..., src_columnn])
FROM 源表名
[WHERE condition]
```

- 在 INSERT 语句中加入子查询。
- **不必书写 VALUES 子句。**
- 子查询中的值列表应与 INSERT 子句中的列名对应。

举例：

```
INSERT INTO emp2
SELECT *
FROM employees
WHERE department_id = 90;
```

```
INSERT INTO sales_reps(id, name, salary, commission_pct)
SELECT employee_id, last_name, salary, commission_pct
FROM employees
WHERE job_id LIKE '%REP%';
```

2. 更新数据

EMPLOYEES

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	HIRE_DATE	JOB_ID	SALARY	DEPARTMENT_ID	COMMISSION_PCT
100	Steven	King	SKING	17-JUN-87	AD_PRES	24000	90	
101	Neena	Kochhar	NKOCHHAR	21-SEP-89	AD_VP	17000	90	
102	Lex	De Haan	LDEHAAN	13-JAN-93	AD_VP	17000	90	
103	Alexander	Hunold	AHUNOLD	03-JAN-90	IT_PROG	9000	60	
104	Bruce	Ernst	BERNST	21-MAY-91	IT_PROG	6000	60	
107	Diana	Lorentz	DLORENTZ	07-FEB-99	IT_PROG	4200	60	
124	Kevin	Mourgos	KMOURGOS	16-NOV-99	ST_MAN	5800	50	

更新 EMPLOYEES 表



EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	HIRE_DATE	JOB_ID	SALARY	DEPARTMENT_ID	COMMISSION_PCT
100	Steven	King	SKING	17-JUN-87	AD_PRES	24000	90	
101	Neena	Kochhar	NKOCHHAR	21-SEP-89	AD_VP	17000	90	
102	Lex	De Haan	LDEHAAN	13-JAN-93	AD_VP	17000	90	
103	Alexander	Hunold	AHUNOLD	03-JAN-90	IT_PROG	9000	30	
104	Bruce	Ernst	BERNST	21-MAY-91	IT_PROG	6000	30	
107	Diana	Lorentz	DLORENTZ	07-FEB-99	IT_PROG	4200	30	
124	Kevin	Mourgos	KMOURGOS	16-NOV-99	ST_MAN	5800	50	

- 使用 UPDATE 语句更新数据。语法如下：

[WHERE condition]

- 可以一次更新**多条**数据。
- 如果需要回滚数据，需要保证在DML前，进行设置：**SET AUTOCOMMIT = FALSE;**

- 使用 **WHERE** 子句指定需要更新的数据。

```
UPDATE employees  
SET department_id = 70  
WHERE employee_id = 113;
```

- 如果省略 WHERE 子句，则表中的所有数据都将被更新。

```
UPDATE copy_emp  
SET department_id = 110;
```

- 更新中的数据完整性错误**

```
UPDATE employees  
SET department_id = 55  
WHERE department_id = 110;
```

错误代码： 1452

```
Cannot add or update a child row: a foreign key  
constraint fails (`myemployees`.`employees`,  
CONSTRAINT `dept_id_fk` FOREIGN KEY (`department_id`)  
REFERENCES `departments` (`department_id`))
```

说明：不存在 55 号部门

3. 删除数据

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing		
100	Finance		
50	Shipping	124	1500
60	IT	103	1400

从表**DEPARTMENTS** 中删除一条记录。

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing		
50	Shipping	124	1500
60	IT	103	1400

```
DELETE FROM      table  
[WHERE           condition];
```

```
DELETE FROM table_name [WHERE <condition>];
```

table_name指定要执行删除操作的表；“[WHERE]”为可选参数，指定删除条件，如果没有WHERE子句，DELETE语句将删除表中的所有记录。

- 使用 WHERE 子句删除指定的记录。

```
DELETE FROM departments  
WHERE department_name = 'Finance';
```

- 如果省略 WHERE 子句，则表中的全部数据将被删除

```
DELETE FROM copy_emp;
```

• **删除中的数据完整性错误**

```
DELETE FROM departments  
WHERE department_id = 60;
```

错误代码： 1451

```
Cannot delete or update a parent row: a foreign key  
constraint fails (`myemployees`.`employees`,  
CONSTRAINT `dept_id_fk` FOREIGN KEY (`department_id`)  
REFERENCES `departments` (`department_id`))
```

说明： You cannot delete a row that contains a primary key that is used as a foreign key in another table.

4. MySQL8新特性：计算列

什么叫计算列呢？简单来说就是某一列的值是通过别的列计算得来的。例如，a列值为1、b列值为2，c列不需要手动插入，定义a+b的结果为c的值，那么c就是计算列，是通过别的列计算得来的。

在MySQL 8.0中，CREATE TABLE 和 ALTER TABLE 中都支持增加计算列。下面以CREATE TABLE为例进行讲解。

举例：定义数据表tb1，然后定义字段id、字段a、字段b和字段c，其中字段c为计算列，用于计算a+b的值。首先创建测试表tb1，语句如下：

```
CREATE TABLE tb1(  
    id INT,  
    a INT,  
    b INT,  
    c INT GENERATED ALWAYS AS (a + b) VIRTUAL  
);
```

插入演示数据语句如下：

北京宏福校区：010-56253825 深圳西部硅谷校区：0755-23060254 上海大江商厦校区：021-57652717

查询数据表tb1中的数据，结果如下：

```
mysql> SELECT * FROM tb1;
+----+----+----+----+
| id | a   | b   | c   |
+----+----+----+----+
| NULL | 100 | 200 | 300 |
+----+----+----+----+
1 row in set (0.00 sec)
```

更新数据中的数据，语句如下：

```
mysql> UPDATE tb1 SET a = 500;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 1  Changed: 0  Warnings: 0
```

5. 综合案例

1、创建数据库test01_library

2、创建表 books，表结构如下：

字段名	字段说明	数据类型
id	书编号	INT
name	书名	VARCHAR(50)
authors	作者	VARCHAR(100)
price	价格	FLOAT
pubdate	出版日期	YEAR
note	说明	VARCHAR(100)
num	库存	INT

3、向books表中插入记录

1) 不指定字段名称，插入第一条记录
2) 指定所有字段名称，插入第二记录
3) 同时插入多条记录（剩下的所有记录）

1	Tal of AAA	Dickes	23	1995	novel	11
2	EmmaT	Jane lura	35	1993	joke	22
3	Story of Jane	Jane Tim	40	2001	novel	0
4	Lovey Day	George Byron	20	2005	novel	30
5	Old land	Honore Blade	30	2010	law	0
6	The Battle	Upton Sara	30	1999	medicine	40
7	Rose Hood	Richard haggard	28	2008	cartoon	28

4、将小说类型(novel)的书的价格都增加5。

5、将名称为EmmaT的书的价格改为40，并将说明改为drama。

6、删除库存为0的记录。

7、统计书名中包含a字母的书

8、统计书名中包含a字母的书的数量和库存总量

9、找出“novel”类型的书，按照价格降序排列

10、查询图书信息，按照库存量降序排列，如果库存量相同的按照note升序排列

11、按照note分类统计书的数量

12、按照note分类统计书的库存量，显示库存量超过30本的

13、查询所有图书，每页显示5本，显示第二页

14、按照note分类统计书的库存量，显示库存量最多的

15、查询书名达到10个字符的书，不包括里面的空格

16、查询书名和类型，其中note值为novel显示小说，law显示法律，medicine显示医药，cartoon显示卡通，joke显示笑话

17、查询书名、库存，其中num值超过30本的，显示滞销，大于0并低于10的，显示畅销，为0的显示需要无货

18、统计每一种note的库存量，并合计总量

19、统计每一种note的数量，并合计总量

20、统计库存量前三名的图书

21、找出最早出版的一本书

22、找出novel中价格最高的一本书

23、找出书名中字数最多的一本书，不含空格

```
#指定使用哪个数据库
USE test01_library;

#2、创建表 books
CREATE TABLE books(
    id INT,
    name VARCHAR(50),
    `authors` VARCHAR(100) ,
    price FLOAT,
    pubdate YEAR ,
    note VARCHAR(100),
    num INT
);

#3、向books表中插入记录
# 1) 不指定字段名称，插入第一条记录
INSERT INTO books
VALUES(1,'Tal of AAA','Dickes',23,1995,'novel',11);
# 2) 指定所有字段名称，插入第二记录
INSERT INTO books (id,name,`authors`,price,pubdate,note,num)
VALUES(2,'EmmaT','Jane lura',35,1993,'Joke',22);
# 3) 同时插入多条记录（剩下的所有记录）
INSERT INTO books (id,name,`authors`,price,pubdate,note,num) VALUES
(3,'Story of Jane','Jane Tim',40,2001,'novel',0),
(4,'Lovey Day','George Byron',20,2005,'novel',30),
(5,'Old land','Honore Blade',30,2010,'Law',0),
(6,'The Battle','Upton Sara',30,1999,'medicine',40),
(7,'Rose Hood','Richard haggard',28,2008,'cartoon',28);

# 4、将小说类型(novel)的书的价格都增加5。
UPDATE books SET price=price+5 WHERE note = 'novel';

# 5、将名称为EmmaT的书的价格改为40，并将说明改为drama。
UPDATE books SET price=40,note='drama' WHERE name='EmmaT';

# 6、删除库存为0的记录。
DELETE FROM books WHERE num=0;
```

```
# 7、统计书名中包含a字母的书
SELECT * FROM books WHERE name LIKE '%a%';

# 8、统计书名中包含a字母的书的数量和库存总量
SELECT COUNT(*),SUM(num) FROM books WHERE name LIKE '%a%';

# 9、找出“novel”类型的书，按照价格降序排列
SELECT * FROM books WHERE note = 'novel' ORDER BY price DESC;

# 10、查询图书信息，按照库存量降序排列，如果库存量相同的按照note升序排列
SELECT * FROM books ORDER BY num DESC,note ASC;

# 11、按照note分类统计书的数量
SELECT note,COUNT(*) FROM books GROUP BY note;

# 12、按照行note分类统计书的库存量，显示库存量超过30本的
```

13、查询所有图书，每页显示5本，显示第二页

```
SELECT * FROM books LIMIT 5,5;
```

14、按照note分类统计书的库存量，显示库存量最多的

```
SELECT note,SUM(num) sum_num FROM books GROUP BY note ORDER BY sum_num DESC LIMIT 0,1;
```

15、查询书名达到10个字符的书，不包括里面的空格

```
SELECT * FROM books WHERE CHAR_LENGTH(REPLACE(name, ' ', ''))>=10;
```

/*

16、查询书名和类型，

其中note值为 novel显示小说， law显示法律， medicine显示医药， cartoon显示卡通， joke显示笑话

*/

```
SELECT name AS "书名" ,note, CASE note
```

```
WHEN 'novel' THEN '小说'
```

```
WHEN 'law' THEN '法律'
```

```
WHEN 'medicine' THEN '医药'
```

```
WHEN 'cartoon' THEN '卡通'
```

```
WHEN 'joke' THEN '笑话'
```

```
END AS "类型"
```

```
FROM books;
```

17、查询书名、库存，其中num值超过30本的，显示滞销，大于0并低于10的，显示畅销，为0的显示需要无货

```
SELECT name,num,CASE
```

```
WHEN num>30 THEN '滞销'
```

```
WHEN num>0 AND num<10 THEN '畅销'
```

```
WHEN num=0 THEN '无货'
```

```
ELSE '正常'
```

```
END AS "库存状态"
```

```
FROM books;
```

18、统计每一种note的库存量，并合计总量

```
SELECT IFNULL(note,'合计总库存量') AS note,SUM(num) FROM books GROUP BY note WITH ROLLUP;
```

19、统计每一种note的数量，并合计总量

```
SELECT IFNULL(note,'合计总数') AS note,COUNT(*) FROM books GROUP BY note WITH ROLLUP;
```

20、统计库存量前三名的图书

```
SELECT * FROM books ORDER BY num DESC LIMIT 0,3;
```

21、找出最早出版的一本书

```
SELECT * FROM books ORDER BY pubdate ASC LIMIT 0,1;
```

22、找出novel中价格最高的一本书

```
SELECT * FROM books WHERE note = 'novel' ORDER BY price DESC LIMIT 0,1;
```

23、找出书名中字数最多的一本书，不含空格

```
SELECT * FROM books ORDER BY CHAR_LENGTH(REPLACE(name, ' ', '')) DESC LIMIT 0,1;
```



让天下没有难学的技术

第12章_MySQL数据类型精讲

讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

1. MySQL中的数据类型

类型	类型举例
整数类型	TINYINT、SMALLINT、MEDIUMINT、INT(或INTEGER)、BIGINT
浮点类型	FLOAT、DOUBLE
定点数类型	DECIMAL
位类型	BIT
日期时间类型	YEAR、TIME、DATE、DATETIME、TIMESTAMP
文本字符串类型	CHAR、VARCHAR、TINYTEXT、TEXT、MEDIUMTEXT、LONGTEXT
枚举类型	ENUM
集合类型	SET
二进制字符串类型	BINARY、VARBINARY、TINYBLOB、BLOB、MEDIUMBLOB、LONGBLOB
JSON类型	JSON对象、JSON数组
空间数据类型	单值类型：GEOMETRY、POINT、LINESTRING、POLYGON； 集合类型：MULTIPOINT、MULTILINESTRING、MULTIPOLYGON、GEOMETRYCOLLECTION

常见数据类型的属性，如下：

MySQL关键字	含义
NULL	数据列可包含NULL值
NOT NULL	数据列不允许包含NULL值
DEFAULT	默认值
PRIMARY KEY	主键
AUTO_INCREMENT	自动递增，适用于整数类型
UNSIGNED	无符号
CHARACTER SET name	指定一个字符集

2.1 类型介绍

整数类型一共有 5 种，包括 TINYINT、SMALLINT、MEDIUMINT、INT (INTEGER) 和 BIGINT。

它们的区别如下表所示：

整数类型	字节	有符号数取值范围	无符号数取值范围
TINYINT	1	-128~127	0~255
SMALLINT	2	-32768~32767	0~65535
MEDIUMINT	3	-8388608~8388607	0~16777215
INT、INTEGER	4	-2147483648~2147483647	0~4294967295
BIGINT	8	-9223372036854775808~9223372036854775807	0~18446744073709551615

2.2 可选属性

整数类型的可选属性有三个：

2.2.1 M

M : 表示显示宽度，M 的取值范围是(0, 255)。例如，int(5)：当数据宽度小于5位的时候在数字前面需要用字符填满宽度。该项功能需要配合“**ZEROFILL**”使用，表示用“0”填满宽度，否则指定显示宽度无效。

如果设置了显示宽度，那么插入的数据宽度超过显示宽度限制，会不会截断或插入失败？

答案：不会对插入的数据有任何影响，还是按照类型的实际宽度进行保存，即 **显示宽度与类型可以存储的值范围无关**。从MySQL 8.0.17开始，整数数据类型不推荐使用显示宽度属性。

整型数据类型可以在定义表结构时指定所需要的显示宽度，如果不指定，则系统为每一种类型指定默认的宽度值。

举例：

```
CREATE TABLE test_int1 ( x TINYINT, y SMALLINT, z MEDIUMINT, m INT, n BIGINT );
```

查看表结构（MySQL5.7中显式如下，MySQL8中不再显式范围）

```
mysql> desc test_int1;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| x    | tinyint(4) | YES  |     | NULL    |       |
| y    | smallint(6) | YES  |     | NULL    |       |
| z    | mediumint(9) | YES  |     | NULL    |       |
| m    | int(11)    | YES  |     | NULL    |       |
| n    | bigint(20) | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)
```

TINYINT有符号数和无符号数的取值范围分别为-128~127和0~255，由于负号占了一个数位，因此TINYINT默认的显示宽度为4。同理，其他整数类型的默认显示宽度与其有符号数的最小值的宽度相同。

举例：

北京宏福校区：010-56253825 深圳西部硅谷校区：0755-23060254 上海大江商厦校区：021-57652717

```
f2 INT(5),
f3 INT(5) ZEROFILL
)

DESC test_int2;

INSERT INTO test_int2(f1,f2,f3)
VALUES(1,123,123);

INSERT INTO test_int2(f1,f2)
VALUES(123456,123456);

INSERT INTO test_int2(f1,f2,f3)
VALUES(123456,123456,123456);
```

```
mysql> SELECT * FROM test_int2;
+-----+-----+-----+
| f1   | f2   | f3   |
+-----+-----+-----+
|     1 | 123  | 00123 |
| 123456 | 123456 |    NULL |
| 123456 | 123456 | 123456 |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

2.2.2 UNSIGNED

UNSIGNED : 无符号类型（非负），所有的整数类型都有一个可选的属性UNSIGNED（无符号属性），无符号整数类型的最小取值为0。所以，如果需要在MySQL数据库中保存非负整数值时，可以将整数类型设置为无符号类型。

int类型默认显示宽度为int(11)，无符号int类型默认显示宽度为int(10)。

```
CREATE TABLE test_int3(
f1 INT UNSIGNED
);

mysql> desc test_int3;
+-----+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| f1    | int(10) unsigned | YES |     | NULL    |       |
+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

2.2.3 ZEROFILL

ZEROFILL : 0填充, (如果某列是ZEROFILL, 那么MySQL会自动为当前列添加UNSIGNED属性), 如果指定了ZEROFILL只是表示不够M位时, 用0在左边填充, 如果超过M位, 只要不超过数据存储范围即可。

原来, 在 int(M) 中, M 的值跟 int(M) 所占多少存储空间并无任何关系。int(3)、int(4)、int(8) 在磁盘上都是占用 4 bytes 的存储空间。也就是说, **int(M), 必须和UNSIGNED ZEROFILL一起使用才有意义。** 如果整数值超过M位, 就按照实际位数存储。只是无须再用字符 0 进行填充。

TINYINT：一般用于枚举数据，比如系统设定取值范围很小且固定的场景。

SMALLINT：可以用于较小范围的统计数据，比如统计工厂的固定资产库存数量等。

MEDIUMINT：用于较大整数的计算，比如车站每日的客流量等。

INT、INTEGER：取值范围足够大，一般情况下不用考虑超限问题，用得最多。比如商品编号。

BIGINT：只有当你处理特别巨大的整数时才会用到。比如双十一的交易量、大型门户网站点击量、证券公司衍生产品持仓等。

2.4 如何选择？

在评估用哪种整数类型的时候，你需要考虑 **存储空间** 和 **可靠性** 的平衡问题：一方面，用占用字节数少的整数类型可以节省存储空间；另一方面，要是为了节省存储空间，使用的整数类型取值范围太小，一旦遇到超出取值范围的情况，就可能引起 **系统错误**，影响可靠性。

举个例子，商品编号采用的数据类型是 INT。原因就在于，客户门店中流通的商品种类较多，而且，每天都有旧商品下架，新商品上架，这样不断迭代，日积月累。

如果使用 SMALLINT 类型，虽然占用字节数比 INT 类型的整数少，但是却不能保证数据不会超出范围 65535。相反，使用 INT，就能确保有足够大的取值范围，不用担心数据超出范围影响可靠性的的问题。

你要注意的是，在实际工作中，**系统故障产生的成本远远超过增加几个字段存储空间所产生的成本**。因此，我建议你首先确保数据不会超过取值范围，在这个前提之下，再去考虑如何节省存储空间。

3. 浮点类型

3.1 类型介绍

浮点数和定点数类型的特点是可以 **处理小数**，你可以把整数看成小数的一个特例。因此，浮点数和定点数的使用场景，比整数大多了。MySQL 支持的浮点数类型，分别是 FLOAT、DOUBLE、REAL。

- FLOAT 表示单精度浮点数；
- DOUBLE 表示双精度浮点数；

类型	有符号数取值范围	无符号数取值范围	占用字节数
FLOAT	(-3.402823466E+38, -1.175494351E-38), 0, (1.175494351 E-38, 3.402823466351 E+38)	0, (1.175494351 E-38, 3.402823466 E+38)	4
DOUBLE	(-1.7976931348623157E+308, -2.2250738585072014E-308), 0, (2.2250738585072014E-308, 1.7976931348623157E+308)	0, (2.2250738585072014E-308, 1.7976931348623157E+308)	8

- REAL 默认就是 DOUBLE。如果你把 SQL 模式设定为启用“**REAL_AS_FLOAT**”，那么，MySQL 就认为 REAL 是 FLOAT。如果要启用“**REAL_AS_FLOAT**”，可以通过以下 SQL 语句实现：

```
SET sql_mode = "REAL_AS_FLOAT";
```

问题1： FLOAT 和 DOUBLE 这两种数据类型的区别是啥呢？

问题2：为什么浮点数类型的无符号数取值范围，只包含于有符号数取值范围的一半，也就是只包含于有符号数取值范围大于等于零的部分呢？

MySQL 存储浮点数的格式为： 符号(S)、尾数(M) 和 阶码(E)。因此，无论有没有符号，MySQL 的浮点数都会存储表示符号的部分。因此，所谓的无符号数取值范围，其实就是有符号数取值范围大于等于零的部分。

3.2 数据精度说明

对于浮点类型，在MySQL中单精度值使用 4 个字节，双精度值使用 8 个字节。

- MySQL允许使用 非标准语法（其他数据库未必支持，因此如果涉及到数据迁移，则最好不要这么用）：`FLOAT(M,D)` 或 `DOUBLE(M,D)`。这里，M称为 精度，D称为 标度。（M,D）中 M=整数位+小数位，D=小数位。 D<=M<=255，0<=D<=30。

例如，定义为`FLOAT(5,2)`的一个列可以显示为-999.99-999.99。如果超过这个范围会报错。

- `FLOAT`和`DOUBLE`类型在不指定(M,D)时，默认会按照实际的精度（由实际的硬件和操作系统决定）来显示。
- 说明：浮点类型，也可以加 `UNSIGNED`，但是不会改变数据范围，例如：`FLOAT(3,2) UNSIGNED`仍然只能表示0-9.99的范围。
- 不管是否显式设置了精度(M,D)，这里MySQL的处理方案如下：
 - 如果存储时，整数部分超出了范围，MySQL就会报错，不允许存这样的值
 - 如果存储时，小数点部分若超出范围，就分以下情况：
 - 若四舍五入后，整数部分没有超出范围，则只警告，但能成功操作并四舍五入删除多余的小数位后保存。例如在`FLOAT(5,2)`列内插入999.009，近似结果是999.01。
 - 若四舍五入后，整数部分超出范围，则MySQL报错，并拒绝处理。如`FLOAT(5,2)`列内插入999.995和-999.995都会报错。
- 从MySQL 8.0.17开始，`FLOAT(M,D)` 和 `DOUBLE(M,D)`用法在官方文档中已经明确不推荐使用**，将来可能被移除。另外，关于浮点型`FLOAT`和`DOUBLE`的`UNSIGNED`也不推荐使用了，将来也可能被移除。
- 举例

```
CREATE TABLE test_double1(
    f1 FLOAT,
    f2 FLOAT(5,2),
    f3 DOUBLE,
    f4 DOUBLE(5,2)
);

DESC test_double1;

INSERT INTO test_double1
VALUES(123.456, 123.456, 123.4567, 123.45);

#Out of range value for column 'f2' at row 1
INSERT INTO test_double1
VALUES(123.456, 1234.456, 123.4567, 123.45);

SELECT * FROM test_double1;
```

浮点数类型有个缺陷，就是不精准。下面我来重点解释一下为什么 MySQL 的浮点数不够精准。比如，我们设计一个表，有f1这个字段，插入值分别为0.47,0.44,0.19，我们期待的运行结果是： $0.47 + 0.44 + 0.19 = 1.1$ 。而使用sum之后查询：

```
CREATE TABLE test_double2(
    f1 DOUBLE
);

INSERT INTO test_double2
VALUES(0.47),(0.44),(0.19);
```

```
mysql> SELECT SUM(f1)
    -> FROM test_double2;
+-----+
| SUM(f1) |
+-----+
| 1.0999999999999999 |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT SUM(f1) = 1.1, 1.1 = 1.1
    -> FROM test_double2;
+-----+-----+
| SUM(f1) = 1.1 | 1.1 = 1.1 |
+-----+-----+
|          0 |         1 |
+-----+-----+
1 row in set (0.00 sec)
```

查询结果是 1.0999999999999999。看到了吗？虽然误差很小，但确实有误差。你也可以尝试把数据类型改成 FLOAT，然后运行求和查询，得到的是， 1.099999940395355。显然，误差更大了。

那么，为什么会存在这样的误差呢？问题还是出在 MySQL 对浮点类型数据的存储方式上。

MySQL 用 4 个字节存储 FLOAT 类型数据，用 8 个字节来存储 DOUBLE 类型数据。无论哪个，都是采用二进制的方式来进行存储的。比如 9.625，用二进制来表达，就是 1001.101，或者表达成 1.001101×2^3 。如果尾数不是 0 或 5（比如 9.624），你就无法用一个二进制数来精确表达。进而，就只好在取值允许的范围内进行四舍五入。

在编程中，如果用到浮点数，要特别注意误差问题，**因为浮点数是不准确的，所以我们要避免使用“=”来判断两个数是否相等**。同时，在一些对精确度要求较高的项目中，千万不要使用浮点数，不然会导致结果错误，甚至是造成不可挽回的损失。那么，MySQL 有没有精准的数据类型呢？当然有，这就是定点数类型：**DECIMAL**。

4. 定点数类型

4.1 类型介绍

- MySQL中的定点数类型只有 DECIMAL 一种类型。

DECIMAL(M,D),DEC,NUMERIC

M+2字节

有效范围由M和D决定

使用 DECIMAL(M,D) 的方式表示高精度小数。其中，M被称为精度，D被称为标度。0<=M<=65，0<=D<=30，D<M。例如，定义DECIMAL (5,2) 的类型，表示该列取值范围是-999.99~999.99。

- **DECIMAL(M,D)的最大取值范围与DOUBLE类型一样**，但是有效的数据范围是由M和D决定的。
DECIMAL 的存储空间并不是固定的，由精度值M决定，总共占用的存储空间为M+2个字节。也就是说，在一些对精度要求不高的场景下，比起占用同样字节长度的定点数，浮点数表达的数值范围可以更大一些。
- 定点数在MySQL内部是以 **字符串** 的形式进行存储，这就决定了它一定是精准的。
- 当DECIMAL类型不指定精度和标度时，其默认为DECIMAL(10,0)。当数据的精度超出了定点数类型的精度范围时，则MySQL同样会进行四舍五入处理。
- **浮点数 vs 定点数**
 - 浮点数相对于定点数的优点是在长度一定的情况下，浮点类型取值范围大，但是不精准，适用于需要取值范围大，又可以容忍微小误差的科学计算场景（比如计算化学、分子建模、流体动力学等）
 - 定点数类型取值范围相对小，但是精准，没有误差，适合于对精度要求极高的场景（比如涉及金额计算的场景）
- 举例

```
CREATE TABLE test_decimal1(
    f1 DECIMAL,
    f2 DECIMAL(5,2)
);

DESC test_decimal1;

INSERT INTO test_decimal1(f1,f2)
VALUES(123.123,123.456);

#Out of range value for column 'f2' at row 1
INSERT INTO test_decimal1(f2)
VALUES(1234.34);
```

```
mysql> SELECT * FROM test_decimal1;
+----+----+
| f1 | f2   |
+----+----+
| 123 | 123.46 |
+----+----+
1 row in set (0.00 sec)
```

- 举例

我们运行下面的语句，把test_double2表中字段“f1”的数据类型修改为 DECIMAL(5,2)：

```
ALTER TABLE test_double2
MODIFY f1 DECIMAL(5,2);
```

然后，我们再一次运行求和语句：

```
+-----+
| SUM(f1) |
+-----+
|      1.10 |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT SUM(f1) = 1.1
      -> FROM test_double2;
+-----+
| SUM(f1) = 1.1 |
+-----+
|          1 |
+-----+
1 row in set (0.00 sec)
```

4.2 开发中经验

“由于 DECIMAL 数据类型的精准性，在我们的项目中，除了极少数（比如商品编号）用到整数类型外，其他的数值都用的是 DECIMAL，原因就是这个项目所处的零售行业，要求精准，一分钱也不能差。”——来自某项目经理

5. 位类型：BIT

BIT类型中存储的是二进制值，类似010110。

二进制字符串类型	长度	长度范围	占用空间
BIT(M)	M	1 <= M <= 64	约为(M + 7)/8个字节

BIT类型，如果没有指定(M)，默认是1位。这个1位，表示只能存1位的二进制值。这里(M)是表示二进制的位数，位数最小值为1，最大值为64。

```
CREATE TABLE test_bit1(
f1 BIT,
f2 BIT(5),
f3 BIT(64)
);

INSERT INTO test_bit1(f1)
VALUES(1);

#Data too long for column 'f1' at row 1
INSERT INTO test_bit1(f1)
VALUES(2);

INSERT INTO test_bit1(f2)
VALUES(23);
```

注意：在向BIT类型的字段中插入数据时，一定要确保插入的数据在BIT类型支持的范围内。

```
mysql> SELECT * FROM test_bit1;
+-----+-----+-----+
| f1      | f2      | f3      |
+-----+-----+-----+
| 0x01    | NULL    | NULL    |
| NULL    | 0x17    | NULL    |
+-----+-----+-----+
2 rows in set (0.00 sec)
```

```
mysql> SELECT BIN(f2),HEX(f2)
-> FROM test_bit1;
+-----+-----+
| BIN(f2) | HEX(f2) |
+-----+-----+
| NULL    | NULL    |
| 10111   | 17      |
+-----+-----+
2 rows in set (0.00 sec)
```

```
mysql> SELECT f2 + 0
-> FROM test_bit1;
+-----+
| f2 + 0 |
+-----+
| NULL   |
| 23     |
+-----+
2 rows in set (0.00 sec)
```

可以看到，使用`b+0`查询数据时，可以直接查询出存储的十进制数据的值。

6. 日期与时间类型

日期与时间是重要的信息，在我们的系统中，几乎所有的数据表都用得到。原因是客户需要知道数据的时间标签，从而进行数据查询、统计和处理。

MySQL有多种表示日期和时间的数据类型，不同的版本可能有所差异，MySQL8.0版本支持的日期和时间类型主要有：`YEAR`类型、`TIME`类型、`DATE`类型、`DATETIME`类型和`TIMESTAMP`类型。

- `YEAR` 类型通常用来表示年
- `DATE` 类型通常用来表示年、月、日
- `TIME` 类型通常用来表示时、分、秒
- `DATETIME` 类型通常用来表示年、月、日、时、分、秒
- `TIMESTAMP` 类型通常用来表示带时区的年、月、日、时、分、秒

YEAR	年	1	YYYY或YY	1901	2155
TIME	时间	3	HH:MM:SS	-838:59:59	838:59:59
DATE	日期	3	YYYY-MM-DD	1000-01-01	9999-12-03
DATETIME	日期 时间	8	YYYY-MM-DD HH:MM:SS	1000-01-01 00:00:00	9999-12-31 23:59:59
TIMESTAMP	日期 时间	4	YYYY-MM-DD HH:MM:SS	1970-01-01 00:00:00 UTC	2038-01-19 03:14:07UTC

可以看到，不同数据类型表示的时间内容不同、取值范围不同，而且占用的字节数也不一样，你要根据实际需要灵活选取。

为什么时间类型 TIME 的取值范围不是 -23:59:59 ~ 23:59:59 呢？原因是 MySQL 设计的 TIME 类型，不光表示一天之内的时间，而且可以用来表示一个时间间隔，这个时间间隔可以超过 24 小时。

6.1 YEAR类型

YEAR类型用来表示年份，在所有的日期时间类型中所占用的存储空间最小，只需要 1个字节 的存储空间。

在MySQL中，YEAR有以下几种存储格式：

- 以4位字符串或数字格式表示YEAR类型，其格式为YYYY，最小值为1901，最大值为2155。
- 以2位字符串格式表示YEAR类型，最小值为00，最大值为99。
 - 当取值为01到69时，表示2001到2069；
 - 当取值为70到99时，表示1970到1999；
 - 当取值整数的0或00添加的话，那么是0000年；
 - 当取值是日期/字符串的'0'添加的话，是2000年。

从MySQL5.5.27开始，2位格式的YEAR已经不推荐使用。YEAR默认格式就是“YYYY”，没必要写成YEAR(4)，从MySQL 8.0.19开始，不推荐使用指定显示宽度的YEAR(4)数据类型。

```
CREATE TABLE test_year(
f1 YEAR,
f2 YEAR(4)
);
```

```
mysql> DESC test_year;
+-----+-----+-----+-----+-----+
| Field | Type  | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| f1   | year(4) | YES |   | NULL    |       |
| f2   | year(4) | YES |   | NULL    |       |
+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

```
mysql> SELECT * FROM test_year;
+----+----+
| f1 | f2 |
+----+----+
| 2020 | 2021 |
+----+----+
1 rows in set (0.00 sec)
```

```
INSERT INTO test_year
VALUES('45','71');

INSERT INTO test_year
VALUES(0,'0');

mysql> SELECT * FROM test_year;
+----+----+
| f1 | f2 |
+----+----+
| 2020 | 2021 |
| 2045 | 1971 |
| 0000 | 2000 |
+----+----+
3 rows in set (0.00 sec)
```

6.2 DATE类型

DATE类型表示日期，没有时间部分，格式为 `YYYY-MM-DD`，其中，`YYYY`表示年份，`MM`表示月份，`DD`表示日期。需要 **3个字节** 的存储空间。在向DATE类型的字段插入数据时，同样需要满足一定的格式条件。

- 以 `YYYY-MM-DD` 格式或者 `YYYYMMDD` 格式表示的字符串日期，其最小取值为 `1000-01-01`，最大取值为 `9999-12-03`。`YYYYMMDD` 格式会被转化为 `YYYY-MM-DD` 格式。
- 以 `YY-MM-DD` 格式或者 `YYMMDD` 格式表示的字符串日期，此格式中，年份为两位数值或字符串满足 `YEAR` 类型的格式条件为：当年份取值为 00 到 69 时，会被转化为 2000 到 2069；当年份取值为 70 到 99 时，会被转化为 1970 到 1999。
- 使用 `CURRENT_DATE()` 或者 `NOW()` 函数，会插入当前系统的日期。

举例：

创建数据表，表中只包含一个DATE类型的字段f1。

```
CREATE TABLE test_date1(
f1 DATE
);
Query OK, 0 rows affected (0.13 sec)
```

插入数据：

```
INSERT INTO test_date1
VALUES ('2020-10-01'), ('20201001'), (20201001);

INSERT INTO test_date1
VALUES ('00-01-01'), ('000101'), ('69-10-01'), ('691001'), ('70-01-01'), ('700101'),
('99-01-01'), ('990101');
```

```
INSERT INTO test_date1  
VALUES (CURRENT_DATE()), (NOW());  
  
SELECT *  
FROM test_date1;
```

6.3 TIME类型

TIME类型用来表示时间，不包含日期部分。在MySQL中，需要 3个字节 的存储空间来存储TIME类型的数据，可以使用“HH:MM:SS”格式来表示TIME类型，其中，HH表示小时，MM表示分钟，SS表示秒。

在MySQL中，向TIME类型的字段插入数据时，也可以使用几种不同的格式。（1）可以使用带有冒号的字符串，比如' D HH:MM:SS'、' HH:MM:SS '、' HH:MM '、' D HH:MM '、' D HH '或' SS '格式，都能被正确地插入TIME类型的字段中。其中D表示天，其最小值为0，最大值为34。如果使用带有D格式的字符串插入TIME类型的字段时，D会被转化为小时，计算格式为D*24+HH。当使用带有冒号并且不带D的字符串表示时间时，表示当天的时间，比如12:10表示12:10:00，而不是00:12:10。（2）可以使用不带有冒号的字符串或者数字，格式为' HHMMSS '或者' HHMMSS '。如果插入一个不合法的字符串或者数字，MySQL在存储数据时，会将其自动转化为00:00:00进行存储。比如1210，MySQL会将最右边的两位解析成秒，表示00:12:10，而不是12:10:00。（3）使用 CURRENT_TIME() 或者 NOW()，会插入当前系统的时间。

举例：

创建数据表，表中包含一个TIME类型的字段f1。

```
CREATE TABLE test_time1(  
f1 TIME  
);  
Query OK, 0 rows affected (0.02 sec)
```

```
INSERT INTO test_time1  
VALUES('2 12:30:29'), ('12:35:29'), ('12:40'), ('2 12:40'), ('1 05'), ('45');  
  
INSERT INTO test_time1  
VALUES ('123520'), (124011), (1210);  
  
INSERT INTO test_time1  
VALUES (NOW()), (CURRENT_TIME());  
  
SELECT * FROM test_time1;
```

6.4 DATETIME类型

DATETIME类型在所有的日期时间类型中占用的存储空间最大，总共需要 8 个字节的存储空间。在格式上为DATE类型和TIME类型的组合，可以表示为 YYYY-MM-DD HH:MM:SS，其中YYYY表示年份，MM表示月份，DD表示日期，HH表示小时，MM表示分钟，SS表示秒。

在向DATETIME类型的字段插入数据时，同样需要满足一定的格式条件。

- 以 YYYY-MM-DD HH:MM:SS 格式或者 YYYYMMDDHHMMSS 格式的字符串插入DATETIME类型的字段时，最小值为1000-01-01 00:00:00，最大值为9999-12-31 23:59:59。
 - 以YYYYMMDDHHMMSS格式的数字插入DATETIME类型的字段时，会被转化为YYYY-MM-DD HH:MM:SS格式。

- 使用函数 CURRENT_TIMESTAMP() 和 NOW()，可以向DATETIME类型的字段插入系统的当前日期和时间。

举例：

创建数据表，表中包含一个DATETIME类型的字段dt。

```
CREATE TABLE test_datetime1(
    dt DATETIME
);
Query OK, 0 rows affected (0.02 sec)
```

插入数据：

```
INSERT INTO test_datetime1
VALUES ('2021-01-01 06:50:30'), ('20210101065030');

INSERT INTO test_datetime1
VALUES ('99-01-01 00:00:00'), ('990101000000'), ('20-01-01 00:00:00'),
('200101000000');

INSERT INTO test_datetime1
VALUES (20200101000000), (200101000000), (19990101000000), (990101000000);

INSERT INTO test_datetime1
VALUES (CURRENT_TIMESTAMP()), (NOW());
```

6.5 TIMESTAMP类型

TIMESTAMP类型也可以表示日期时间，其显示格式与DATETIME类型相同，都是 YYYY-MM-DD HH:MM:SS，需要4个字节的存储空间。但是TIMESTAMP存储的时间范围比DATETIME要小很多，只能存储“1970-01-01 00:00:01 UTC”到“2038-01-19 03:14:07 UTC”之间的时间。其中，UTC表示世界统一时间，也叫作世界标准时间。

- 存储数据的时候需要对当前时间所在的时区进行转换，查询数据的时候再将时间转换回当前的时区。因此，使用TIMESTAMP存储的同一个时间值，在不同的时区查询时会显示不同的时间。**

向TIMESTAMP类型的字段插入数据时，当插入的数据格式满足YY-MM-DD HH:MM:SS和YYMMDDHHMMSS时，两位数值的年份同样符合YEAR类型的规则条件，只不过表示的时间范围要小很多。

如果向TIMESTAMP类型的字段插入的时间超出了TIMESTAMP类型的范围，则MySQL会抛出错误信息。

举例：

创建数据表，表中包含一个TIMESTAMP类型的字段ts。

```
CREATE TABLE test_timestamp1(
    ts TIMESTAMP
);
```

插入数据：

```
( '990101030405');

INSERT INTO test_timestamp1
VALUES ('2020@01@01@00@00@00'), ('20@01@01@00@00@00');

INSERT INTO test_timestamp1
VALUES (CURRENT_TIMESTAMP()), (NOW());

#Incorrect datetime value
INSERT INTO test_timestamp1
VALUES ('2038-01-20 03:14:07');
```

TIMESTAMP和DATETIME的区别：

- TIMESTAMP存储空间比较小，表示的日期时间范围也比较小
- 底层存储方式不同，TIMESTAMP底层存储的是毫秒值，距离1970-1-1 0:0:0 0毫秒的毫秒值。
- 两个日期比较大小或日期计算时，TIMESTAMP更方便、更快。
- TIMESTAMP和时区有关。TIMESTAMP会根据用户的时区不同，显示不同的结果。而DATETIME则只能反映出插入时当地的时区，其他时区的人查看数据必然会有误差的。

```
CREATE TABLE temp_time(
d1 DATETIME,
d2 TIMESTAMP
);
```

```
INSERT INTO temp_time VALUES('2021-9-2 14:45:52', '2021-9-2 14:45:52');

INSERT INTO temp_time VALUES(NOW(), NOW());
```

```
mysql> SELECT * FROM temp_time;
+-----+-----+
| d1      | d2      |
+-----+-----+
| 2021-09-02 14:45:52 | 2021-09-02 14:45:52 |
| 2021-11-03 17:38:17 | 2021-11-03 17:38:17 |
+-----+-----+
2 rows in set (0.00 sec)
```

```
#修改当前的时区
SET time_zone = '+9:00';
```

```
mysql> SELECT * FROM temp_time;
+-----+-----+
| d1      | d2      |
+-----+-----+
| 2021-09-02 14:45:52 | 2021-09-02 15:45:52 |
| 2021-11-03 17:38:17 | 2021-11-03 18:38:17 |
+-----+-----+
2 rows in set (0.00 sec)
```

用得最多的日期时间类型，就是 **DATETIME**。虽然 MySQL 也支持 YEAR (年)、TIME (时间)、DATE (日期)，以及 TIMESTAMP 类型，但是在实际项目中，尽量用 DATETIME 类型。因为这个数据类型包括了完整的日期和时间信息，取值范围也最大，使用起来比较方便。毕竟，如果日期时间信息分散在好几个字段，很不容易记，而且查询的时候，SQL 语句也会更加复杂。

此外，一般存注册时间、商品发布时间等，不建议使用DATETIME存储，而是使用 **时间戳**，因为 DATETIME 虽然直观，但不便于计算。

```
mysql> SELECT UNIX_TIMESTAMP();
+-----+
| UNIX_TIMESTAMP() |
+-----+
|      1635932762 |
+-----+
1 row in set (0.00 sec)
```

7. 文本字符串类型

在实际的项目中，我们还经常遇到一种数据，就是字符串数据。

MySQL中，文本字符串总体上分为 **CHAR**、**VARCHAR**、**TINYTEXT**、**TEXT**、**MEDIUMTEXT**、**LONGTEXT**、**ENUM**、**SET** 等类型。

文本字符串类型	值的长度	长度范围	占用的存储空间
CHAR(M)	M	0 <= M <= 255	M个字节
VARCHAR(M)	M	0 <= M <= 65535	M+1个字节
TINYTEXT	L	0 <= L <= 255	L+2个字节
TEXT	L	0 <= L <= 65535	L+2个字节
MEDIUMTEXT	L	0 <= L <= 16777215	L+3个字节
LONGTEXT	L	0 <= L <= 4294967295	L+4个字节
ENUM	L	1 <= L <= 65535	1或2个字节
SET	L	0 <= L <= 64	1,2,3,4或8个字节

7.1 CHAR与VARCHAR类型

CHAR和VARCHAR类型都可以存储比较短的字符串。

字符串(文本)类型	特点	长度	长度范围	占用的存储空间
CHAR(M)	固定长度	M	0 <= M <= 255	M个字节
VARCHAR(M)	可变长度	M	0 <= M <= 65535	(实际长度 + 1) 个字节

CHAR类型：

- CHAR(M) 类型一般需要预先定义字符串长度。如果不指定(M)，则表示长度默认是1个字符。
- 如果保存时，数据的实际长度比CHAR类型声明的长度小，则会在 **右侧填充** 空格以达到指定的长度。当MySQL检索CHAR类型的数据时，CHAR类型的字段会去除尾部的空格。
- 定义CHAR类型字段时，声明的字段长度即为CHAR类型字段所占的存储空间的字节数。

```
c2 CHAR(5)
);

DESC test_char1;

INSERT INTO test_char1
VALUES('a', 'Tom');

SELECT c1, CONCAT(c2, '***') FROM test_char1;

INSERT INTO test_char1(c2)
VALUES('a ');

SELECT CHAR_LENGTH(c2)
FROM test_char1;
```

VARCHAR类型:

- VARCHAR(M) 定义时， 必须指定 长度M， 否则报错。
- MySQL4.0版本以下， varchar(20)：指的是20字节，如果存放UTF8汉字时，只能存6个（每个汉字3字节）；MySQL5.0版本以上， varchar(20)：指的是20字符。
- 检索VARCHAR类型的字段数据时，会保留数据尾部的空格。VARCHAR类型的字段所占用的存储空间为字符串实际长度加1个字节。

```
CREATE TABLE test_varchar1(
NAME VARCHAR #错误
);

#Column length too big for column 'NAME' (max = 21845);
CREATE TABLE test_varchar2(
NAME VARCHAR(65535) #错误
);

CREATE TABLE test_varchar3(
NAME VARCHAR(5)
);

INSERT INTO test_varchar3
VALUES('尚硅谷'), ('尚硅谷教育');

#Data too long for column 'NAME' at row 1
INSERT INTO test_varchar3
VALUES('尚硅谷IT教育');
```

哪些情况使用 CHAR 或 VARCHAR 更好

类型	特点	空间上	时间上	适用场景
CHAR(M)	固定长度	浪费存储空间	效率高	存储不大，速度要求高
VARCHAR(M)	可变长度	节省存储空间	效率低	非CHAR的情况

情况1：存储很短的信息。比如门牌号码101, 201.....这样很短的信息应该用char，因为varchar还要占一个byte用于存储信息长度，本来打算节约存储的，结果得不偿失。

情况3：十分频繁改变的column。因为varchar每次存储都要有额外的计算，得到长度等工作，如果一个非常频繁改变的，那就要有很多的精力用于计算，而这些对于char来说是不需要的。

情况4：具体存储引擎中的情况：

- **MyISAM** 数据存储引擎和数据列：MyISAM数据表，最好使用固定长度(CHAR)的数据列代替可变长度(VARCHAR)的数据列。这样使得整个表静态化，从而使**数据检索更快**，用空间换时间。
- **MEMORY** 存储引擎和数据列：MEMORY数据表目前都使用固定长度的数据行存储，因此无论使用CHAR或VARCHAR列都没有关系，两者都是作为CHAR类型处理的。
- **InnoDB** 存储引擎，建议使用VARCHAR类型。因为对于InnoDB数据表，内部的行存储格式并没有区分固定长度和可变长度列（所有数据行都使用指向数据列值的头指针），而且**主要影响性能的因素是数据行使用的存储总量**，由于char平均占用的空间多于varchar，所以除了简短并且固定长度的，其他考虑varchar。这样节省空间，对磁盘I/O和数据存储总量比较好。

7.2 TEXT类型

在MySQL中，TEXT用来保存文本类型的字符串，总共包含4种类型，分别为TINYTEXT、TEXT、MEDIUMTEXT 和 LONGTEXT 类型。

在向TEXT类型的字段保存和查询数据时，系统自动按照实际长度存储，不需要预先定义长度。这一点和VARCHAR类型相同。

每种TEXT类型保存的数据长度和所占用的存储空间不同，如下：

文本字符串类型	特点	长度	长度范围	占用的存储空间
TINYTEXT	小文本、可变长度	L	0 <= L <= 255	L + 2 个字节
TEXT	文本、可变长度	L	0 <= L <= 65535	L + 2 个字节
MEDIUMTEXT	中等文本、可变长度	L	0 <= L <= 16777215	L + 3 个字节
LONGTEXT	大文本、可变长度	L	0 <= L <= 4294967295 (相当于4GB)	L + 4 个字节

由于实际存储的长度不确定，MySQL 不允许 TEXT 类型的字段做主键。遇到这种情况，你只能采用CHAR(M)，或者 VARCHAR(M)。

举例：

创建数据表：

```
CREATE TABLE test_text(
tx TEXT
);
```

```
INSERT INTO test_text
VALUES('atguigu');

SELECT CHAR_LENGTH(tx)
FROM test_text; #10
```

尚硅谷经验：

TEXT文本类型，可以存比较大的文本段，搜索速度稍慢，因此如果不是特别大的内容，建议使用CHAR, VARCHAR来代替。还有TEXT类型不用加默认值，加了也没用。而且text和blob类型的数据删除后容易导致“空洞”，使得文件碎片比较多，所以频繁使用的表不建议包含TEXT类型字段，建议单独分出去，单独用一个表。

8. ENUM类型

ENUM类型也叫作枚举类型， ENUM类型的取值范围需要在定义字段时进行指定。设置字段值时， ENUM类型只允许从成员中选取单个值，不能一次选取多个值。

其所需要的存储空间由定义 ENUM类型时指定的成员个数决定。

文本字符串类型	长度	长度范围	占用的存储空间
ENUM	L	1 <= L <= 65535	1或2个字节

- 当ENUM类型包含1~255个成员时，需要1个字节的存储空间；
- 当ENUM类型包含256~65535个成员时，需要2个字节的存储空间。
- ENUM类型的成员个数的上限为65535个。

举例：

创建表如下：

```
CREATE TABLE test_enum(
    season ENUM('春', '夏', '秋', '冬', 'unknow')
);
```

添加数据：

```
INSERT INTO test_enum
VALUES('春'), ('秋');

# 忽略大小写
INSERT INTO test_enum
VALUES('UNKNOW');

# 允许按照角标的方式获取指定索引位置的枚举值
INSERT INTO test_enum
VALUES('1'), (3);

# Data truncated for column 'season' at row 1
INSERT INTO test_enum
VALUES('ab');

# 当ENUM类型的字段没有声明为NOT NULL时，插入NULL也是有效的
INSERT INTO test_enum
VALUES(NULL);
```

9. SET类型

当SET类型包含的成员个数不同时，其所占用的存储空间也是不同的，具体如下：

成员个数范围 (L表示实际成员个数)	占用的存储空间
1 <= L <= 8	1个字节
9 <= L <= 16	2个字节
17 <= L <= 24	3个字节
25 <= L <= 32	4个字节
33 <= L <= 64	8个字节

SET类型在存储数据时成员个数越多，其占用的存储空间越大。注意：SET类型在选取成员时，可以一次选择多个成员，这一点与ENUM类型不同。

举例：

创建表：

```
CREATE TABLE test_set(
    s SET ('A', 'B', 'C')
);
```

向表中插入数据：

```
INSERT INTO test_set (s) VALUES ('A'), ('A,B');

#插入重复的SET类型成员时，MySQL会自动删除重复的成员
INSERT INTO test_set (s) VALUES ('A,B,C,A');

#向SET类型的字段插入SET成员中不存在的值时，MySQL会抛出错误。
INSERT INTO test_set (s) VALUES ('A,B,C,D');

SELECT *
FROM test_set;
```

举例：

```
CREATE TABLE temp_mul(
    gender ENUM('男', '女'),
    hobby SET('吃饭', '睡觉', '打豆豆', '写代码')
);

INSERT INTO temp_mul VALUES('男', '睡觉,打豆豆'); #成功

# Data truncated for column 'gender' at row 1
INSERT INTO temp_mul VALUES('男,女', '睡觉,写代码'); #失败

# Data truncated for column 'gender' at row 1
INSERT INTO temp_mul VALUES('妖', '睡觉,写代码'); #失败

INSERT INTO temp_mul VALUES('男', '睡觉,写代码,吃饭'); #成功
```

MySQL中的二进制字符串类型主要存储一些二进制数据，比如可以存储图片、音频和视频等二进制数据。

MySQL中支持的二进制字符串类型主要包括BINARY、VARBINARY、TINYBLOB、BLOB、MEDIUMBLOB 和 LONGBLOB类型。

BINARY与VARBINARY类型

BINARY和VARBINARY类似于CHAR和VARCHAR，只是它们存储的是二进制字符串。

BINARY (M)为固定长度的二进制字符串，M表示最多能存储的字节数，取值范围是0~255个字符。如果未指定(M)，表示只能存储 1个字节。例如BINARY (8)，表示最多能存储8个字节，如果字段值不足(M)个字节，将在右边填充'\0'以补齐指定长度。

VARBINARY (M)为可变长度的二进制字符串，M表示最多能存储的字节数，总字节数不能超过行的字节长度限制65535，另外还要考虑额外字节开销，VARBINARY类型的数据除了存储数据本身外，还需要1或2个字节来存储数据的字节数。VARBINARY类型 必须指定 (M) ，否则报错。

二进制字符串类型	特点	值的长度	占用空间
BINARY(M)	固定长度	M (0 <= M <= 255)	M个字节
VARBINARY(M)	可变长度	M (0 <= M <= 65535)	M+1个字节

举例：

创建表：

```
CREATE TABLE test_binary1(
f1 BINARY,
f2 BINARY(3),
# f3 VARBINARY,
f4 VARBINARY(10)
);
```

添加数据：

```
INSERT INTO test_binary1(f1,f2)
VALUES ('a','a');

INSERT INTO test_binary1(f1,f2)
VALUES ('尚','尚');#失败
```

```
INSERT INTO test_binary1(f2,f4)
VALUES ('ab','ab');

mysql> SELECT LENGTH(f2),LENGTH(f4)
-> FROM test_binary1;
+-----+-----+
| LENGTH(f2) | LENGTH(f4) |
+-----+-----+
| 3 | NULL |
| 3 | 2 |
+-----+-----+
```

BLOB是一个 **二进制大对象**，可以容纳可变数量的数据。

MySQL中的BLOB类型包括TINYBLOB、BLOB、MEDIUMBLOB和LONGBLOB 4种类型，它们可容纳值的最大长度不同。可以存储一个二进制的大对象，比如 **图片**、**音频** 和 **视频** 等。

需要注意的是，在实际工作中，往往不会在MySQL数据库中使用BLOB类型存储大对象数据，通常会将图片、音频和视频文件存储到 **服务器的磁盘上**，并将图片、音频和视频的访问路径存储到MySQL中。

二进制字符串类型	值的长度	长度范围	占用空间
TINYBLOB	L	0 <= L <= 255	L + 1 个字节
BLOB	L	0 <= L <= 65535 (相当于64KB)	L + 2 个字节
MEDIUMBLOB	L	0 <= L <= 16777215 (相当于16MB)	L + 3 个字节
LONGBLOB	L	0 <= L <= 4294967295 (相当于4GB)	L + 4 个字节

举例：

```
CREATE TABLE test_blob1(
    id INT,
    img MEDIUMBLOB
);
```

TEXT和BLOB的使用注意事项：

在使用text和blob字段类型时要注意以下几点，以便更好的发挥数据库的性能。

① BLOB和TEXT值也会引起自己的一些问题，特别是执行了大量的删除或更新操作的时候。删除这种值会在数据表中留下很大的“**空洞**”，以后填入这些“空洞”的记录可能长度不同。为了提高性能，建议定期使用 **OPTIMIZE TABLE** 功能对这类表进行 **碎片整理**。

② 如果需要对大文本字段进行模糊查询，MySQL 提供了 **前缀索引**。但是仍然要在不必要的时候避免检索大型的BLOB或TEXT值。例如，**SELECT *** 查询就不是很好的想法，除非你能够确定作为约束条件的 **WHERE**子句只会找到所需要的数据行。否则，你可能毫无目的地在网络上传输大量的值。

③ 把BLOB或TEXT列 **分离到单独的表** 中。在某些环境中，如果把这些数据列移动到第二张数据表中，可以让你把原数据表中的数据列转换为固定长度的数据行格式，那么它就是有意义的。这会 **减少主表中的碎片**，使你得到固定长度数据行的性能优势。它还使你在主数据表上运行 **SELECT *** 查询的时候不会通过网络传输大量的BLOB或TEXT值。

11. JSON 类型

JSON (JavaScript Object Notation) 是一种轻量级的 **数据交换格式**。简洁和清晰的层次结构使得 JSON 成为理想的数据交换语言。它易于人阅读和编写，同时也易于机器解析和生成，并有效地提升网络传输效率。**JSON 可以将 JavaScript 对象中表示的一组数据转换为字符串，然后就可以在网络或者程序之间轻松地传递这个字符串，并在需要的时候将它还原为各编程语言所支持的数据格式。**

在MySQL 5.7中，就已经支持JSON数据类型。在MySQL 8.x版本中，JSON类型提供了可以进行自动验证的JSON文档和优化的存储结构，使得在MySQL中存储和读取JSON类型的数据更加方便和高效。创建数据表，表中包含一个JSON类型的字段 js。

```
);
```

向表中插入JSON数据。

```
INSERT INTO test_json (js)
VALUES ('{"name":"songhk", "age":18, "address":{"province":"beijing",
"city":"beijing"}');
```

查询t19表中的数据。

```
mysql> SELECT *
-> FROM test_json;
```

```
+-----+
| js |
+-----+
| {"age": 18, "name": "songhk", "address": {"city": "beijing", "province": "beijing"}) |
+-----+
1 row in set (0.00 sec)
```

当需要检索JSON类型的字段中数据的某个具体值时，可以使用“->”和“->>”符号。

```
mysql> SELECT js -> '$.name' AS NAME,js -> '$.age' AS age ,js -> '$.address.province'
AS province, js -> '$.address.city' AS city
-> FROM test_json;
+-----+-----+-----+
| NAME | age | province | city |
+-----+-----+-----+
| "songhk" | 18 | "beijing" | "beijing" |
+-----+-----+-----+
1 row in set (0.00 sec)
```

通过“->”和“->>”符号，从JSON字段中正确查询出了指定的JSON数据的值。

12. 空间类型

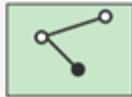
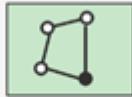
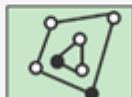
MySQL 空间类型扩展支持地理特征的生成、存储和分析。这里的地理特征表示世界上具有位置的任何东西，可以是一个实体，例如一座山；可以是空间，例如一座办公楼；也可以是一个可定义的位置，例如一个十字路口等等。MySQL中使用 **Geometry**（几何）来表示所有地理特征。Geometry指一个点或点的集合，代表世界上任何具有位置的事物。

MySQL的空间数据类型（Spatial Data Type）对应于OpenGIS类，包括单值类型：GEOMETRY、POINT、LINESTRING、POLYGON以及集合类型：MULTIPOINT、MULTILINESTRING、MULTIPOLYGON、GEOMETRYCOLLECTION。

- Geometry是所有空间集合类型的基类，其他类型如POINT、LINESTRING、POLYGON都是Geometry的子类。
 - Point，顾名思义就是点，有一个坐标值。例如POINT(121.213342 31.234532)，POINT(30 10)，坐标值支持DECIMAL类型，经度（longitude）在前，维度（latitude）在后，用空格分隔。
 - LineString，线，由一系列点连接而成。如果线从头至尾没有交叉，那就是简单的（simple）；如果起点和终点重叠，那就是封闭的（closed）。例如LINESTRING(30 10,10 30,40 40)，点与点之间用逗号分隔，一个点中的经纬度用空格分隔，与POINT格式一致。

下面展示几种常见的几何图形元素：

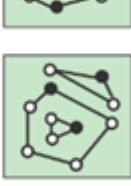
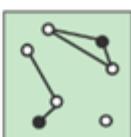
Geometry primitives(2D)

Type	Examples	
Point		POINT(30 10)
LineString		LINESTRING(30 10,40 40,20 40,10 20,30 10)
Polygon	 	POLYGON((35 10,45 45, 15 40, 10 20, 35 10), (20 30,35 35,30 20,20 30))

- MultiPoint、MultiLineString、MultiPolygon、GeometryCollection 这4种类型都是集合类，是多个 Point、LineString或Polygon组合而成。

下面展示的是多个同类或异类几何图形元素的组合：

Multipart geometries (2D)

Type	Examples	
MultiPoint		MULTIPOINT(10 40),(40 30),(20 20),(30 10)
		MULTIPOINT(10 40, 40 30, 20 20, 30 10)
MultiLineString		MULTILINESTRING((10 10, 20 20, 10 40), (40 40, 30 30, 40 20, 30 10))
MultiPolygon		MULTIPOLYGON(((30 20, 45 40, 10 40, 30 20), ((15 5, 40 10, 10 20, 5 10, 15 5)))
		MULTIPOLYGON(((40 40, 20 45, 45 30, 40 40), ((20 35, 10 30, 10 10, 30 5, 45 20, 20 35), (30 20, 20 15, 20 25, 30 20)))
GeometryCollection		GEOMETRYCOLLECTION(POINT(40 10), LINESTRING(10 10, 20 20, 10 40), POLYGON((40 40, 20 45, 45 30, 40 40)))

13. 小结及选择建议

这样做的好处是，首先确保你的系统不会因为数据类型定义出错。不过，凡事都是有两面的，可靠性好，并不意味着高效。比如，TEXT 虽然使用方便，但是效率不如 CHAR(M) 和 VARCHAR(M)。

关于字符串的选择，建议参考如下阿里巴巴的《Java开发手册》规范：

阿里巴巴《Java开发手册》之MySQL数据库：

- 任何字段如果为非负数，必须是 UNSIGNED
- 【强制】小数类型为 DECIMAL，禁止使用 FLOAT 和 DOUBLE。
 - 说明：在存储的时候，FLOAT 和 DOUBLE 都存在精度损失的问题，很可能在比较值的时候，得到不正确的结果。如果存储的数据范围超过 DECIMAL 的范围，建议将数据拆成整数和小数并分开存储。
- 【强制】如果存储的字符串长度几乎相等，使用 CHAR 定长字符串类型。
- 【强制】VARCHAR 是可变长字符串，不预先分配存储空间，长度不要超过 5000。如果存储长度大于此值，定义字段类型为 TEXT，独立出来一张表，用主键来对应，避免影响其它字段索引效率。

第13章_约束

讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

1. 约束(constraint)概述

1.1 为什么需要约束

数据完整性（Data Integrity）是指数据的精确性（Accuracy）和可靠性（Reliability）。它是防止数据库中存在不符合语义规定的数据和防止因错误信息的输入输出造成无效操作或错误信息而提出的。

为了保证数据的完整性，SQL规范以约束的方式对表数据进行额外的条件限制。从以下四个方面考虑：

- 实体完整性（Entity Integrity）：例如，同一个表中，不能存在两条完全相同无法区分的记录
- 域完整性（Domain Integrity）：例如：年龄范围0-120，性别范围“男/女”
- 引用完整性（Referential Integrity）：例如：员工所在部门，在部门表中要能找到这个部门
- 用户自定义完整性（User-defined Integrity）：例如：用户名唯一、密码不能为空等，本部门经理的工资不得高于本部门职工的平均工资的5倍。

1.2 什么是约束

约束是表级的强制规定。

可以在**创建表时规定约束（通过 CREATE TABLE 语句）**，或者在**表创建之后通过 ALTER TABLE 语句规定约束**。

1.3 约束的分类

- **根据约束数据列的限制**，约束可分为：
 - **单列约束**：每个约束只约束一列
 - **多列约束**：每个约束可约束多列数据
- **根据约束的作用范围**，约束可分为：
 - **列级约束**：只能作用在一个列上，跟在列的定义后面
 - **表级约束**：可以作用在多个列上，不与列一起，而是单独定义

	位置	支持的约束类型	是否可以起约束名
列级约束：	列的后面	语法都支持，但外键没有效果	不可以
表级约束：	所有列的下面	默认和非空不支持，其他支持	可以（主键没有效果）

- **根据约束起的作用**，约束可分为：
 - **NOT NULL 非空约束**，规定某个字段不能为空
 - **UNIQUE 唯一约束**，规定某个字段在整个表中是唯一的
 - **PRIMARY KEY 主键(非空且唯一)约束**
 - **FOREIGN KEY 外键约束**
 - **CHECK 检查约束**
 - **DEFAULT 默认值约束**

- 查看某个表已有的约束

```
#information_schema数据库名（系统库）
#table_constraints表名称（专门存储各个表的约束）
SELECT * FROM information_schema.table_constraints
WHERE table_name = '表名称' ;
```

2. 非空约束

2.1 作用

限定某个字段/某列的值不允许为空

EMPLOYEE_ID	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	DEPARTMENT_ID
100	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000	90
101	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	17000	90
102	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	17000	90
103	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	9000	60
104	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	6000	60
178	Grant	KGRANT	011.44.1644.429263	24-MAY-99	SA_REP	7000	
200	Whalen	JWHALEN	515.123.4444	17-SEP-87	AD_ASST	4400	10
...							

NOT NULL 约束 NOT NULL 约束 无NOT NULL 约束

2.2 关键字

NOT NULL

2.3 特点

- 默认，所有的类型的值都可以是NULL，包括INT、FLOAT等数据类型
- 非空约束只能出现在表对象的列上，只能某个列单独限定非空，不能组合非空
- 一个表可以有很多列都分别限定了非空
- 空字符串"不等于NULL，0也不等于NULL

2.4 添加非空约束

(1) 建表时

```
CREATE TABLE 表名称(
    字段名 数据类型,
    字段名 数据类型 NOT NULL,
    字段名 数据类型 NOT NULL
);
```

举例：

```
NAME VARCHAR(20) NOT NULL,  
sex CHAR NULL  
) ;
```

```
CREATE TABLE student(  
    sid int,  
    sname varchar(20) not null,  
    tel char(11),  
    cardid char(18) not null  
) ;
```

```
insert into student values(1, '张三', '13710011002', '110222198912032545') ; #成功  
  
insert into student values(2, '李四', '13710011002', null); #身份证号为空  
ERROR 1048 (23000): Column 'cardid' cannot be null  
  
insert into student values(2, '李四', null, '110222198912032546') ;#成功, tel允许为空  
  
insert into student values(3, null, null, '110222198912032547') ;#失败  
ERROR 1048 (23000): Column 'sname' cannot be null
```

(2) 建表后

```
alter table 表名称 modify 字段名 数据类型 not null;
```

举例:

```
ALTER TABLE emp  
MODIFY sex VARCHAR(30) NOT NULL;
```

```
alter table student modify sname varchar(20) not null;
```

2.5 删除非空约束

```
alter table 表名称 modify 字段名 数据类型 NULL; #去掉not null, 相当于修改某个非注解字段, 该字段允许为空
```

或

```
alter table 表名称 modify 字段名 数据类型; #去掉not null, 相当于修改某个非注解字段, 该字段允许为空
```

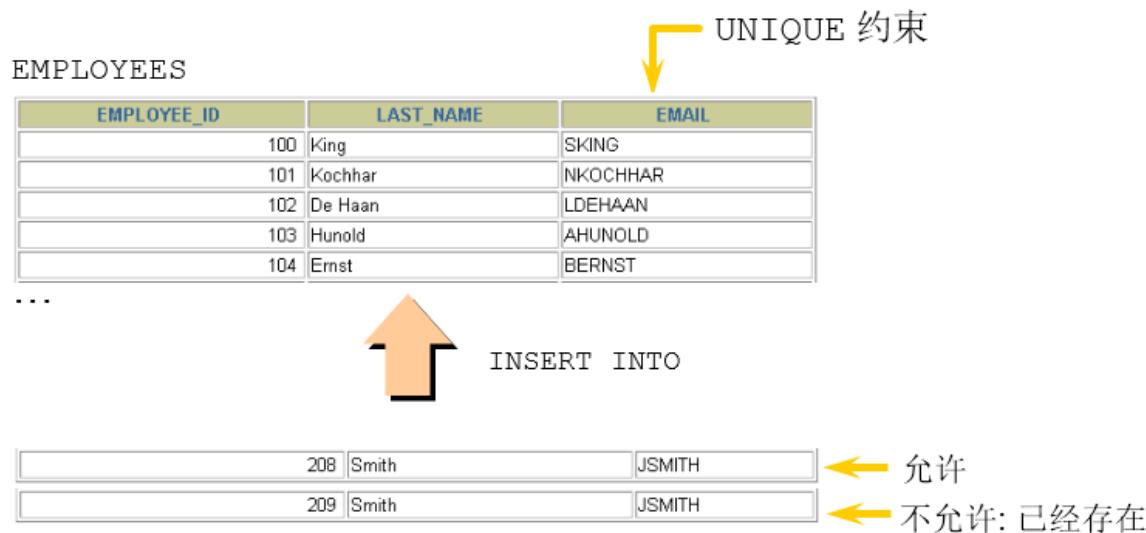
举例:

```
ALTER TABLE emp  
MODIFY sex VARCHAR(30) NULL;
```

```
ALTER TABLE emp  
MODIFY NAME VARCHAR(15) DEFAULT 'abc' NULL;
```

3. 唯一性约束

用来限制某个字段/某列的值不能重复。



唯一约束，允许出现多个空值：NULL。

3.2 关键字

UNIQUE

3.3 特点

- 同一个表可以有多个唯一约束。
- 唯一约束可以是某一个列的值唯一，也可以多个列组合的值唯一。
- 唯一性约束允许列值为空。
- 在创建唯一约束的时候，如果不给唯一约束命名，就默认和列名相同。
- MySQL会给唯一约束的列上默认创建一个唯一索引。

3.4 添加唯一约束

(1) 建表时

```
create table 表名称(
    字段名 数据类型,
    字段名 数据类型 unique,
    字段名 数据类型 unique key,
    字段名 数据类型
);
create table 表名称(
    字段名 数据类型,
    字段名 数据类型,
    字段名 数据类型,
    [constraint 约束名] unique key(字段名)
);
```

举例：

```
sname varchar(20),
tel char(11) unique,
cardid char(18) unique key
);
```

```
CREATE TABLE t_course(
cid INT UNIQUE,
cname VARCHAR(100) UNIQUE,
description VARCHAR(200)
);
```

```
CREATE TABLE USER(
id INT NOT NULL,
NAME VARCHAR(25),
PASSWORD VARCHAR(16),
-- 使用表级约束语法
CONSTRAINT uk_name_pwd UNIQUE(NAME,PASSWORD)
);
```

表示用户名和密码组合不能重复

```
insert into student values(1, '张三', '13710011002', '101223199012015623');
insert into student values(2, '李四', '13710011003', '101223199012015624');
```

```
mysql> select * from student;
+----+-----+-----+-----+
| sid | sname | tel      | cardid      |
+----+-----+-----+-----+
| 1  | 张三  | 13710011002 | 101223199012015623 |
| 2  | 李四  | 13710011003 | 101223199012015624 |
+----+-----+-----+-----+
2 rows in set (0.00 sec)
```

```
insert into student values(3, '王五', '13710011004', '101223199012015624'); #身份证号重复
ERROR 1062 (23000): Duplicate entry '101223199012015624' for key 'cardid'
```

```
insert into student values(3, '王五', '13710011003', '101223199012015625');
ERROR 1062 (23000): Duplicate entry '13710011003' for key 'tel'
```

(2) 建表后指定唯一键约束

```
#字段列表中如果是一个字段，表示该列的值唯一。如果是两个或更多个字段，那么复合唯一，即多个字段的组合是唯一的
#方式1:
alter table 表名称 add unique key(字段列表);
```

```
#方式2:
alter table 表名称 modify 字段名 字段类型 unique;
```

举例：

```
ALTER TABLE USER
ADD UNIUIF(NAME PASSWORD);
```

```
ALTER TABLE USER  
MODIFY NAME VARCHAR(20) UNIQUE;
```

举例：

```
create table student(  
    sid int primary key,  
    sname varchar(20),  
    tel char(11) ,  
    cardid char(18)  
) ;
```

```
alter table student add unique key(tel);  
alter table student add unique key(cardid);
```

3.5 关于复合唯一约束

```
create table 表名称(  
    字段名 数据类型,  
    字段名 数据类型,  
    字段名 数据类型,  
    unique key(字段列表) #字段列表中写的是多个字段名，多个字段名用逗号分隔，表示那么是复合唯一，即多个字段的组合是唯一的  
) ;
```

```
#学生表  
create table student(  
    sid int,      #学号  
    sname varchar(20),        #姓名  
    tel char(11) unique key, #电话  
    cardid char(18) unique key #身份证号  
) ;
```

```
#课程表  
create table course(  
    cid int,    #课程编号  
    cname varchar(20)      #课程名称  
) ;
```

```
#选课表  
create table student_course(  
    id int,  
    sid int,  
    cid int,  
    score int,  
    unique key(sid,cid)  #复合唯一  
) ;
```

```
insert into student values(1,'张三','13710011002','101223199012015623')#成功  
insert into student values(2,'李四','13710011003','101223199012015624')#成功  
insert into course values(1001,'Java'),(1002,'MySQL')#成功
```

```
mvsal> select * from student:
```

北京宏福校区：010-56253825 深圳西部硅谷校区：0755-23060254 上海大江商厦校区：021-57652717

```
+----+----+----+----+
| 1 | 张三 | 13710011002 | 101223199012015623 |
| 2 | 李四 | 13710011003 | 101223199012015624 |
+----+----+----+----+
2 rows in set (0.00 sec)
```

```
mysql> select * from course;
+----+----+
| cid | cname |
+----+----+
| 1001 | Java |
| 1002 | MySQL |
+----+----+
2 rows in set (0.00 sec)
```

```
insert into student_course values
(1, 1, 1001, 89),
(2, 1, 1002, 90),
(3, 2, 1001, 88),
(4, 2, 1002, 56);#成功
```

```
mysql> select * from student_course;
+----+----+----+----+
| id | sid | cid | score |
+----+----+----+----+
| 1 | 1 | 1001 | 89 |
| 2 | 1 | 1002 | 90 |
| 3 | 2 | 1001 | 88 |
| 4 | 2 | 1002 | 56 |
+----+----+----+----+
4 rows in set (0.00 sec)
```

```
insert into student_course values (5, 1, 1001, 88);#失败
#ERROR 1062 (23000): Duplicate entry '1-1001' for key 'sid'    违反sid-cid的复合唯一
```

3.5 删除唯一约束

- 添加唯一性约束的列上也会自动创建唯一索引。
- 删除唯一约束只能通过删除唯一索引的方式删除。
- 删除时需要指定唯一索引名，唯一索引名就和唯一约束名一样。
- 如果创建唯一约束时未指定名称，如果是单列，就默认和列名相同；如果是组合列，那么默认和()中排在第一个的列名相同。也可以自定义唯一性约束名。

```
SELECT * FROM information_schema.table_constraints WHERE table_name = '表名'; #查看都有哪些约束
```

```
ALTER TABLE USER
DROP INDEX uk_name_pwd;
```

注意：可以通过 `show index from 表名称;` 查看表的索引

4.1 作用

用来唯一标识表中的一行记录。

4.2 关键字

primary key

4.3 特点

- 主键约束相当于**唯一约束+非空约束的组合**，主键约束列不允许重复，也不允许出现空值。

DEPARTMENTS

PRIMARY KEY

The diagram shows the DEPARTMENTS table with columns: DEPARTMENT_ID, DEPARTMENT_NAME, MANAGER_ID, and LOCATION_ID. A yellow arrow points down to the DEPARTMENT_ID column header, labeled "PRIMARY KEY". Below the table, three dots indicate more rows.

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500

不允许
(空值)

INSERT INTO

The diagram shows an attempt to insert a new row into the DEPARTMENTS table. An orange arrow points up to the "INSERT INTO" text. A yellow arrow points down to the first empty cell in the DEPARTMENT_ID column of a new row. Red text to the left says "不允许 (空值)" (Not allowed (null)).

	Public Accounting		1400
50	Finance	124	1500

不允许
(50 已经存在)

- 一个表最多只能有一个主键约束，建立主键约束可以在列级别创建，也可以在表级别上创建。
- 主键约束对应着表中的一列或者多列（复合主键）
- 如果是多列组合的复合主键约束，那么这些列都不允许为空值，并且组合的值不允许重复。
- MySQL的**主键名总是PRIMARY**，就算自己命名了主键约束名也没用。
- 当创建主键约束时，系统默认会在所在的列或列组合上建立对应的**主键索引**（能够根据主键查询的，就根据主键查询，效率更高）。如果删除主键约束了，主键约束对应的索引就自动删除了。
- 需要注意的一点是，不要修改主键字段的值。因为主键是数据记录的唯一标识，如果修改了主键的值，就有可能会破坏数据的完整性。

4.4 添加主键约束

(1) 建表时指定主键约束

```
字段名 数据类型,
字段名 数据类型
);
create table 表名称(
    字段名 数据类型,
    字段名 数据类型,
    字段名 数据类型,
    [constraint 约束名] primary key(字段名) #表级模式
);
```

举例：

```
create table temp(
    id int primary key,
    name varchar(20)
);
```

```
mysql> desc temp;
+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+
| id    | int(11)   | NO   | PRI | NULL    |       |
| name  | varchar(20) | YES  |     | NULL    |       |
+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

```
insert into temp values(1, '张三');#成功
insert into temp values(2, '李四');#成功
```

```
mysql> select * from temp;
+---+---+
| id | name |
+---+---+
| 1 | 张三 |
| 2 | 李四 |
+---+---+
2 rows in set (0.00 sec)
```

```
insert into temp values(1, '张三');#失败
ERROR 1062 (23000): Duplicate entry '1' for key 'PRIMARY'
```

```
insert into temp values(1, '王五');#失败
ERROR 1062 (23000): Duplicate entry '1' for key 'PRIMARY'
```

```
insert into temp values(3, '张三');#成功
```

```
| id | name |
+---+---+
| 1 | 张三 |
| 2 | 李四 |
| 3 | 张三 |
+---+---+
3 rows in set (0.00 sec)
```

```
insert into temp values(4,null);#成功

insert into temp values(null,'李琦');#失败
ERROR 1048 (23000): Column 'id' cannot be null
```

```
mysql> select * from temp;
+---+---+
| id | name |
+---+---+
| 1 | 张三 |
| 2 | 李四 |
| 3 | 张三 |
| 4 | NULL |
+---+---+
4 rows in set (0.00 sec)
```

```
#演示一个表建立两个主键约束
create table temp(
    id int primary key,
    name varchar(20) primary key
);
ERROR 1068 (42000): Multiple (多重的) primary key defined (定义)
```

再举例：

- 列级约束

```
CREATE TABLE emp4(
    id INT PRIMARY KEY AUTO_INCREMENT ,
    NAME VARCHAR(20)
);
```

- 表级约束

```
CREATE TABLE emp5(
    id INT NOT NULL AUTO_INCREMENT,
    NAME VARCHAR(20),
    pwd VARCHAR(15),
    CONSTRAINT emp5_id_pk PRIMARY KEY(id)
);
```

(2) 建表后增加主键约束

```
ALTER TABLE 表名称 ADD PRIMARY KEY(字段列表); #字段列表可以是一个字段，也可以是多个字段，如果是多个字段的话，是复合主键
```

```
ALTER TABLE emp5 ADD PRIMARY KEY(NAME, pwd);
```

4.5 关于复合主键

```
create table 表名称(  
    字段名 数据类型,  
    字段名 数据类型,  
    字段名 数据类型,  
    primary key(字段名1, 字段名2) #表示字段1和字段2的组合是唯一的, 也可以有更多个字段  
) ;
```

```
#学生表  
create table student(  
    sid int primary key, #学号  
    sname varchar(20) #学生姓名  
) ;
```

```
#课程表  
create table course(  
    cid int primary key, #课程编号  
    cname varchar(20) #课程名称  
) ;
```

```
#选课表  
create table student_course(  
    sid int,  
    cid int,  
    score int,  
    primary key(sid,cid) #复合主键  
) ;
```

```
insert into student values(1, '张三'), (2, '李四');  
insert into course values(1001, 'Java'), (1002, 'MySQL');
```

```
mysql> select * from student;  
+----+----+  
| sid | sname |  
+----+----+  
| 1 | 张三 |  
| 2 | 李四 |  
+----+----+  
2 rows in set (0.00 sec)
```

```
mysql> select * from course;  
+----+----+  
| cid | cname |  
+----+----+  
| 1001 | Java |  
| 1002 | MySQL |  
+----+----+  
2 rows in set (0.00 sec)
```

```
insert into student_course values(1, 1001, 89), (1, 1002, 90), (2, 1001, 88), (2, 1002, 56);
```

```
| sid | cid   | score |
+----+----+-----+
| 1  | 1001 |     89 |
| 1  | 1002 |      90 |
| 2  | 1001 |     88 |
| 2  | 1002 |      56 |
+----+----+-----+
4 rows in set (0.00 sec)
```

```
insert into student_course values(1, 1001, 100);
ERROR 1062 (23000): Duplicate entry '1-1001' for key 'PRIMARY'
```

```
mysql> desc student_course;
+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| sid   | int(11) | NO   | PRI | NULL    |       |
| cid   | int(11) | NO   | PRI | NULL    |       |
| score | int(11) | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

- 再举例

```
CREATE TABLE emp6(
id INT NOT NULL,
NAME VARCHAR(20),
pwd VARCHAR(15),
CONSTRAINT emp7_pk PRIMARY KEY(NAME,pwd)
);
```

4.6 删除主键约束

```
alter table 表名称 drop primary key;
```

举例：

```
ALTER TABLE student DROP PRIMARY KEY;
```

```
ALTER TABLE emp5 DROP PRIMARY KEY;
```

说明：删除主键约束，不需要指定主键名，因为一个表只有一个主键，删除主键约束后，非空还存在。

5. 自增列：AUTO_INCREMENT

某个字段的值自增

5.2 关键字

auto_increment

5.3 特点和要求

- (1) 一个表最多只能有一个自增长列
- (2) 当需要产生唯一标识符或顺序值时，可设置自增长
- (3) 自增长列约束的列必须是键列（主键列，唯一键列）
- (4) 自增约束的列的数据类型必须是整数类型
- (5) 如果自增列指定了 0 和 null，会在当前最大值的基础上自增；如果自增列手动指定了具体值，直接赋值为具体值。

错误演示：

```
create table employee(  
    eid int auto_increment,  
    ename varchar(20)  
);  
# ERROR 1075 (42000): Incorrect table definition; there can be only one auto column  
and it must be defined as a key
```

```
create table employee(  
    eid int primary key,  
    ename varchar(20) unique key auto_increment  
);  
# ERROR 1063 (42000): Incorrect column specifier for column 'ename' 因为ename不是整数类  
型
```

5.4 如何指定自增约束

(1) 建表时

```
create table 表名称(  
    字段名 数据类型 primary key auto_increment,  
    字段名 数据类型 unique key not null,  
    字段名 数据类型 unique key,  
    字段名 数据类型 not null default 默认值,  
);  
create table 表名称(  
    字段名 数据类型 default 默认值 ,  
    字段名 数据类型 unique key auto_increment,  
    字段名 数据类型 not null default 默认值,,  
    primary key(字段名)  
);
```

```
create table employee(  
    eid int primary key auto_increment,  
    ename varchar(20)  
,
```

```
| Field | Type      | Null | Key | Default | Extra      |
+-----+-----+-----+-----+-----+
| eid  | int(11)  | NO   | PRI | NULL    | auto_increment |
| ename | varchar(20) | YES  |     | NULL    |             |
+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

(2) 建表后

```
alter table 表名称 modify 字段名 数据类型 auto_increment;
```

例如：

```
create table employee(
    eid int primary key ,
    ename varchar(20)
);
```

```
alter table employee modify eid int auto_increment;
```

```
mysql> desc employee;
+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra      |
+-----+-----+-----+-----+
| eid  | int(11)  | NO   | PRI | NULL    | auto_increment |
| ename | varchar(20) | YES  |     | NULL    |             |
+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

5.5 如何删除自增约束

```
#alter table 表名称 modify 字段名 数据类型 auto_increment;#给这个字段增加自增约束
```

```
alter table 表名称 modify 字段名 数据类型; #去掉auto_increment相当于删除
```

```
alter table employee modify eid int;
```

```
mysql> desc employee;
+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra      |
+-----+-----+-----+-----+
| eid  | int(11)  | NO   | PRI | NULL    |             |
| ename | varchar(20) | YES  |     | NULL    |             |
+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

5.6 MySQL 8.0新特性—自增变量的持久化

在MySQL 8.0之前，自增主键AUTO_INCREMENT的值如果大于max(primary key)+1，在MySQL重启后，会重置AUTO_INCREMENT=max(primary key)+1，这种现象在某些情况下会导致业务主键冲突或者其他难以发现的问题。下面通过案例来对比不同的版本中自增变量是否持久化。在MySQL 5.7版本中，测试步骤如下：创建的数据表中包含自增主键的id字段，语句如下：

```
);
```

插入4个空值，执行如下：

```
INSERT INTO test1  
VALUES(0),(0),(0),(0);
```

查询数据表test1中的数据，结果如下：

```
mysql> SELECT * FROM test1;  
+---+  
| id |  
+---+  
| 1 |  
| 2 |  
| 3 |  
| 4 |  
+---+  
4 rows in set (0.00 sec)
```

删除id为4的记录，语句如下：

```
DELETE FROM test1 WHERE id = 4;
```

再次插入一个空值，语句如下：

```
INSERT INTO test1 VALUES(0);
```

查询此时数据表test1中的数据，结果如下：

```
mysql> SELECT * FROM test1;  
+---+  
| id |  
+---+  
| 1 |  
| 2 |  
| 3 |  
| 5 |  
+---+  
4 rows in set (0.00 sec)
```

从结果可以看出，虽然删除了id为4的记录，但是再次插入空值时，并没有重用被删除的4，而是分配了5。删除id为5的记录，结果如下：

```
DELETE FROM test1 where id=5;
```

重启数据库，重新插入一个空值。

```
INSERT INTO test1 values(0);
```

再次查询数据表test1中的数据，结果如下：

```
| id |
+---+
| 1 |
| 2 |
| 3 |
| 4 |
+---+
4 rows in set (0.00 sec)
```

从结果可以看出，新插入的0值分配的是4，按照重启前的操作逻辑，此处应该分配6。出现上述结果的主要原因是自增主键没有持久化。在MySQL 5.7系统中，对于自增主键的分配规则，是由InnoDB数据字典内部一个计数器来决定的，而该计数器只在内存中维护，并不会持久化到磁盘中。当数据库重启时，该计数器会被初始化。

在MySQL 8.0版本中，上述测试步骤最后一步的结果如下：

```
mysql> SELECT * FROM test1;
+---+
| id |
+---+
| 1 |
| 2 |
| 3 |
| 6 |
+---+
4 rows in set (0.00 sec)
```

从结果可以看出，自增变量已经持久化了。

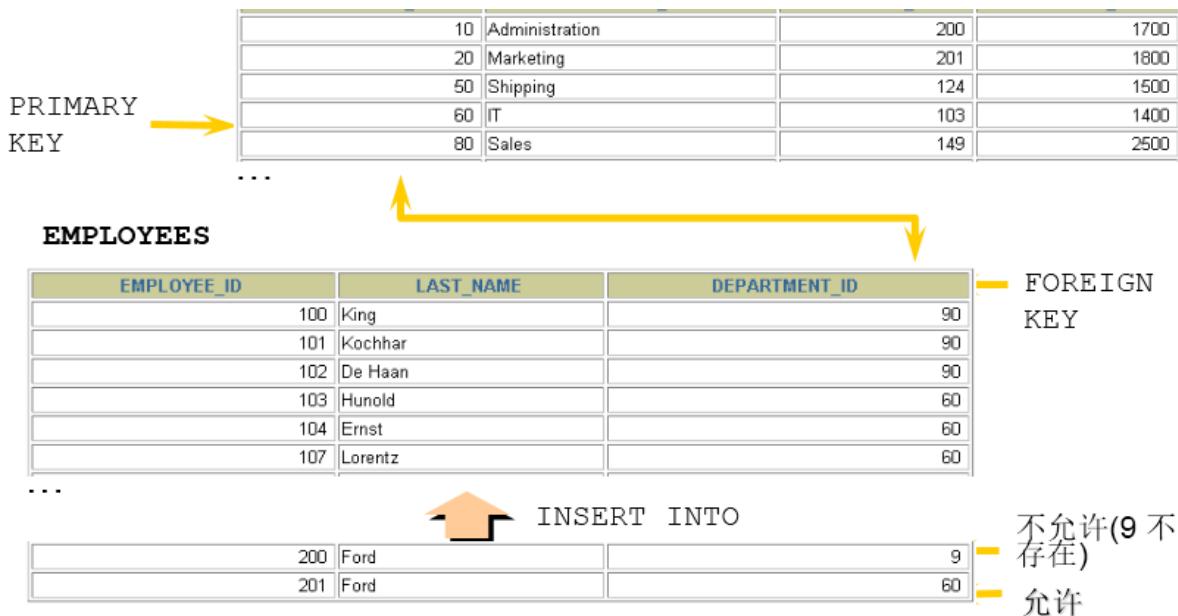
MySQL 8.0将自增主键的计数器持久化到重做日志中。每次计数器发生改变，都会将其写入重做日志中。如果数据库重启，InnoDB会根据重做日志中的信息来初始化计数器的内存值。

6. FOREIGN KEY 约束

6.1 作用

限定某个表的某个字段的引用完整性。

比如：员工表的员工所在部门的选择，必须在部门表能找到对应的部分。



6.2 关键字

FOREIGN KEY

6.3 主表和从表/父表和子表

主表（父表）：被引用的表，被参考的表

从表（子表）：引用别人的表，参考别人的表

例如：员工表的员工所在部门这个字段的值要参考部门表：部门表是主表，员工表是从表。

例如：学生表、课程表、选课表：选课表的学生和课程要分别参考学生表和课程表，学生表和课程表是主表，选课表是从表。

6.4 特点

(1) 从表的外键列，必须引用/参考主表的主键或唯一约束的列

为什么？因为被依赖/被参考的值必须是唯一的

(2) 在创建外键约束时，如果不给外键约束命名，**默认名不是列名，而是自动产生一个外键名**（例如 `student_ibfk_1;`），也可以指定外键约束名。

(3) 创建(CREATE)表时就指定外键约束的话，先创建主表，再创建从表

(4) 删表时，先删从表（或先删除外键约束），再删除主表

(5) 当主表的记录被从表参照时，主表的记录将不允许删除，如果要删除数据，需要先删除从表中依赖该记录的数据，然后才可以删除主表的数据

(6) 在“从表”中指定外键约束，并且一个表可以建立多个外键约束

(7) 从表的外键列与主表被参照的列名字可以不相同，但是数据类型必须一样，逻辑意义一致。如果类型不一样，创建子表时，就会出现错误“`ERROR 1005 (HY000): Can't create table'database.tablename'(errno: 150)`”。

例如：都是表示部门编号，都是int类型。

(9) 删除外键约束后，必须 手动 删除对应的索引

6.5 添加外键约束

(1) 建表时

```
create table 主表名称(
    字段1 数据类型 primary key,
    字段2 数据类型
);

create table 从表名称(
    字段1 数据类型 primary key,
    字段2 数据类型,
    [CONSTRAINT <外键约束名称>] FOREIGN KEY (从表的某个字段) references 主表名(被参考字段)
);
#(从表的某个字段)的数据类型必须与主表名(被参考字段)的数据类型一致，逻辑意义也一样
#(从表的某个字段)的字段名可以与主表名(被参考字段)的字段名一样，也可以不一样

-- FOREIGN KEY: 在表级指定子表中的列
-- REFERENCES: 标示在父表中的列
```

```
create table dept( #主表
    did int primary key,          #部门编号
    dname varchar(50)            #部门名称
);

create table emp(#从表
    eid int primary key,      #员工编号
    ename varchar(5),          #员工姓名
    deptid int,                #员工所在的部门
    foreign key (deptid) references dept(did)  #在从表中指定外键约束
    #emp表的deptid和dept表的did的数据类型一致，意义都是表示部门的编号
);
```

说明：

- (1) 主表dept必须先创建成功，然后才能创建emp表，指定外键成功。
- (2) 删除表时，先删除从表emp，再删除主表dept

(2) 建表后

一般情况下，表与表的关联都是提前设计好了的，因此，会在创建表的时候就把外键约束定义好。不过，如果需要修改表的设计（比如添加新的字段，增加新的关联关系），但没有预先定义外键约束，那么，就要用修改表的方式来补充定义。

格式：

```
ALTER TABLE 从表名 ADD [CONSTRAINT 约束名] FOREIGN KEY (从表的字段) REFERENCES 主表名(被引用
字段) [on update xx][on delete xx];
```

举例：

```
ALTER TABLE emp1
ADD [CONSTRAINT emp_dept_id_fk] FOREIGN KEY(dept_id) REFERENCES dept(dept_id);
```

```
create table dept(
    did int primary key,          #部门编号
    dname varchar(50)             #部门名称
);

create table emp(
    eid int primary key,          #员工编号
    ename varchar(5),             #员工姓名
    deptid int                    #员工所在的部门
);
#这两个表创建时，没有指定外键的话，那么创建顺序是随意
```

```
alter table emp add foreign key (deptid) references dept(did);
```

6.6 演示问题

(1) 失败：不是键列

```
create table dept(
    did int ,                   #部门编号
    dname varchar(50)           #部门名称
);

create table emp(
    eid int primary key,        #员工编号
    ename varchar(5),            #员工姓名
    deptid int,                  #员工所在的部门
    foreign key (deptid) references dept(did)
);
#ERROR 1215 (HY000): Cannot add foreign key constraint 原因是dept的did不是键列
```

(2) 失败：数据类型不一致

```
create table dept(
    did int primary key,          #部门编号
    dname varchar(50)             #部门名称
);

create table emp(
    eid int primary key,          #员工编号
    ename varchar(5),              #员工姓名
    deptid char,                  #员工所在的部门
    foreign key (deptid) references dept(did)
);
#ERROR 1215 (HY000): Cannot add foreign key constraint 原因是从表的deptid字段和主表的did字段的数据类型不一致，并且要它俩的逻辑意义一致
```

(3) 成功，两个表字段名一样

```
dname varchar(50)          #部门名称
);

create table emp(
    eid int primary key,   #员工编号
    ename varchar(5),       #员工姓名
    did int,                #员工所在的部门
    foreign key (did) references dept(did)
    #emp表的deptid和dept表的did的数据类型一致，意义都是表示部门的编号
    #是否重名没问题，因为两个did在不同的表中
);

```

(4) 添加、删除、修改问题

```
create table dept(
    did int primary key,      #部门编号
    dname varchar(50)         #部门名称
);

create table emp(
    eid int primary key,     #员工编号
    ename varchar(5),        #员工姓名
    deptid int,              #员工所在的部门
    foreign key (deptid) references dept(did)
    #emp表的deptid和dept表的did的数据类型一致，意义都是表示部门的编号
);

```



```
insert into dept values(1001, '教学部');
insert into dept values(1003, '财务部');

insert into emp values(1, '张三', 1001); #添加从表记录成功，在添加这条记录时，要求部门表有1001部门

insert into emp values(2, '李四', 1005);#添加从表记录失败
ERROR 1452 (23000): Cannot add or update a child row: a foreign key
constraint fails ('atguigudb`.`emp`, CONSTRAINT `emp_ibfk_1` FOREIGN KEY (`deptid`)
REFERENCES `dept` (`did`)) 从表emp添加记录失败，因为主表dept没有1005部门

```

```
mysql> select * from dept;
+-----+-----+
| did | dname |
+-----+-----+
| 1001 | 教学部 |
| 1003 | 财务部 |
+-----+-----+
2 rows in set (0.00 sec)

mysql> select * from emp;
+-----+-----+-----+
| eid | ename | deptid |
+-----+-----+-----+
| 1   | 张三   | 1001 |
+-----+-----+-----+
1 row in set (0.00 sec)
```

```
foreign key constraint fails (外键约束失败) (`atguigudb`.`emp`, CONSTRAINT `emp_ibfk_1` FOREIGN KEY (`deptid`) REFERENCES `dept` (`did`)) #部门表did字段现在没有1002的值，所以员工表中不能修改员工所在部门deptid为1002
```

```
update dept set did = 1002 where did = 1001;#修改主表失败  
ERROR 1451 (23000): Cannot delete (删除) or update (修改) a parent row (父表的记录) : a foreign key constraint fails (`atguigudb`.`emp`, CONSTRAINT `emp_ibfk_1` FOREIGN KEY (`deptid`) REFERENCES `dept` (`did`)) #部门表did的1001字段已经被emp引用了，所以部门表的1001字段就不能修改了。
```

```
update dept set did = 1002 where did = 1003;#修改主表成功 因为部门表的1003部门没有被emp表引用，所以可以修改
```

```
delete from dept where did=1001; #删除主表失败  
ERROR 1451 (23000): Cannot delete (删除) or update (修改) a parent row (父表记录) : a foreign key constraint fails (`atguigudb`.`emp`, CONSTRAINT `emp_ibfk_1` FOREIGN KEY (`deptid`) REFERENCES `dept` (`did`)) #因为部门表did的1001字段已经被emp引用了，所以部门表的1001字段对应的记录就不能被删除
```

总结：约束关系是针对双方的

- 添加了外键约束后，主表的修改和删除数据受约束
- 添加了外键约束后，从表的添加和修改数据受约束
- 在从表上建立外键，要求主表必须存在
- 删除主表时，要求从表从表先删除，或将从表中外键引用该主表的关系先删除

6.7 约束等级

- Cascade方式**：在父表上update/delete记录时，同步update/delete掉子表的匹配记录
- Set null方式**：在父表上update/delete记录时，将子表上匹配记录的列设为null，但是要注意子表的外键列不能为not null
- No action方式**：如果子表中有匹配的记录，则不允许对父表对应候选键进行update/delete操作
- Restrict方式**：同no action，都是立即检查外键约束
- Set default方式**（在可视化工具SQLyog中可能显示空白）：父表有变更时，子表将外键列设置成一个默认的值，但Innodb不能识别

如果没有指定等级，就相当于Restrict方式。

对于外键约束，最好是采用: **ON UPDATE CASCADE ON DELETE RESTRICT** 的方式。

(1) 演示1: on update cascade on delete set null

```
create table dept(  
    did int primary key,          #部门编号  
    dname varchar(50)            #部门名称  
);  
  
create table emp(  
    eid int primary key,         #员工编号  
    ename varchar(5),           #员工姓名  
    deptid int,                 #员工所在的部门  
    foreign key (deptid) references dept(did) on update cascade on delete set null  
    #把修改操作设置为级联修改等级，把删除操作设置为set null等级  
);
```

```
insert into dept values(1003, '咨询部');

insert into emp values(1, '张三', 1001); #在添加这条记录时，要求部门表有1001部门
insert into emp values(2, '李四', 1001);
insert into emp values(3, '王五', 1002);
```

```
mysql> select * from dept;

mysql> select * from emp;
```

```
#修改主表成功，从表也跟着修改，修改了主表被引用的字段1002为1004，从表的引用字段就跟着修改为1004了
mysql> update dept set did = 1004 where did = 1002;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
mysql> select * from dept;
+-----+
| did | dname |
+-----+
| 1001 | 教学部 |
| 1003 | 咨询部 |
| 1004 | 财务部 | #原来是1002，修改为1004
+-----+
3 rows in set (0.00 sec)
```

```
mysql> select * from emp;
+-----+
| eid | ename | deptid |
+-----+
| 1 | 张三 | 1001 |
| 2 | 李四 | 1001 |
| 3 | 王五 | 1004 | #原来是1002，跟着修改为1004
+-----+
3 rows in set (0.00 sec)
```

```
#删除主表的记录成功，从表对应的字段的值被修改为null
mysql> delete from dept where did = 1001;
Query OK, 1 row affected (0.01 sec)
```

```
mysql> select * from dept;
+-----+
| did | dname | #记录1001部门被删除了
+-----+
| 1003 | 咨询部 |
| 1004 | 财务部 |
+-----+
2 rows in set (0.00 sec)
```

```
mysql> select * from emp;
```

```
| 3 | 王五 | 1004 |
+----+----+----+
3 rows in set (0.00 sec)
```

(2) 演示2: on update set null on delete cascade

```
create table dept(
    did int primary key,          #部门编号
    dname varchar(50)            #部门名称
);

create table emp(
    eid int primary key,        #员工编号
    ename varchar(50),          #员工姓名
    deptid int,                 #员工所在的部门
    foreign key (deptid) references dept(did)  on update set null on delete cascade
    #把修改操作设置为set null等级，把删除操作设置为级联删除等级
);
```

```
insert into dept values(1001, '教学部');
insert into dept values(1002, '财务部');
insert into dept values(1003, '咨询部');

insert into emp values(1, '张三', 1001); #在添加这条记录时，要求部门表有1001部门
insert into emp values(2, '李四', 1001);
insert into emp values(3, '王五', 1002);
```

```
mysql> select * from dept;
+----+----+
| did | dname |
+----+----+
| 1001 | 教学部 |
| 1002 | 财务部 |
| 1003 | 咨询部 |
+----+----+
3 rows in set (0.00 sec)
```

```
mysql> select * from emp;
+----+----+----+
| eid | ename | deptid |
+----+----+----+
| 1 | 张三 | 1001 |
| 2 | 李四 | 1001 |
| 3 | 王五 | 1002 |
+----+----+----+
3 rows in set (0.00 sec)
```

```
#修改主表，从表对应的字段设置为null
mysql> update dept set did = 1004 where did = 1002;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
mysql> select * from dept;
+----+----+
| did | dname |
+----+----+
```

```
| 1003 | 咨询部 |
| 1004 | 财务部 | #原来did是1002
+-----+
3 rows in set (0.00 sec)

mysql> select * from emp;
+-----+-----+-----+
| eid | ename | deptid |
+-----+-----+-----+
| 1 | 张三 | 1001 |
| 2 | 李四 | 1001 |
| 3 | 王五 | NULL | #原来deptid是1002, 因为部门表1002被修改了, 1002没有对应的了, 就设置为
null
+-----+-----+
3 rows in set (0.00 sec)
```

#删除主表的记录成功, 主表的1001行被删除了, 从表相应的记录也被删除了

```
mysql> delete from dept where did=1001;
Query OK, 1 row affected (0.00 sec)
```

```
mysql> select * from dept;
+-----+-----+
| did | dname | #部门表中1001部门被删除
+-----+-----+
| 1003 | 咨询部 |
| 1004 | 财务部 |
+-----+
2 rows in set (0.00 sec)
```

```
mysql> select * from emp;
+-----+-----+-----+
| eid | ename | deptid | #原来1001部门的员工也被删除了
+-----+-----+-----+
| 3 | 王五 | NULL |
+-----+-----+
1 row in set (0.00 sec)
```

(3) 演示: on update cascade on delete cascade

```
create table dept(
    did int primary key,          #部门编号
    dname varchar(50)            #部门名称
);

create table emp(
    eid int primary key,        #员工编号
    ename varchar(5),           #员工姓名
    deptid int,                 #员工所在的部门
    foreign key (deptid) references dept(did) on update cascade on delete cascade
    #把修改操作设置为级联修改等级, 把删除操作也设置为级联删除等级
);
```

```
insert into dept values(1003, '咨询部');

insert into emp values(1, '张三', 1001); #在添加这条记录时，要求部门表有1001部门
insert into emp values(2, '李四', 1001);
insert into emp values(3, '王五', 1002);
```

```
mysql> select * from dept;
+----+-----+
| did | dname |
+----+-----+
| 1001 | 教学部 |
| 1002 | 财务部 |
| 1003 | 咨询部 |
+----+-----+
3 rows in set (0.00 sec)
```

```
mysql> select * from emp;
+----+-----+-----+
| eid | ename | deptid |
+----+-----+-----+
| 1 | 张三 | 1001 |
| 2 | 李四 | 1001 |
| 3 | 王五 | 1002 |
+----+-----+-----+
3 rows in set (0.00 sec)
```

```
#修改主表，从表对应的字段自动修改
mysql> update dept set did = 1004 where did = 1002;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
mysql> select * from dept;
+----+-----+
| did | dname |
+----+-----+
| 1001 | 教学部 |
| 1003 | 咨询部 |
| 1004 | 财务部 | #部门1002修改为1004
+----+-----+
3 rows in set (0.00 sec)
```

```
mysql> select * from emp;
+----+-----+-----+
| eid | ename | deptid |
+----+-----+-----+
| 1 | 张三 | 1001 |
| 2 | 李四 | 1001 |
| 3 | 王五 | 1004 | #级联修改
+----+-----+
3 rows in set (0.00 sec)
```

```
#删除主表的记录成功，主表的1001行被删除了，从表相应的记录也被删除了
mysql> delete from dept where did=1001;
Query OK, 1 row affected (0.00 sec)
```

```
| did | dname | #1001部门被删除了
+-----+-----+
| 1003 | 咨询部 |
| 1004 | 财务部 |
+-----+-----+
2 rows in set (0.00 sec)

mysql> select * from emp;
+-----+-----+-----+
| eid | ename | deptid | #1001部门的员工也被删除了
+-----+-----+-----+
| 3 | 王五 | 1004 |
+-----+-----+
1 row in set (0.00 sec)
```

6.8 删除外键约束

流程如下：

(1) 第一步先查看约束名和删除外键约束

```
SELECT * FROM information_schema.table_constraints WHERE table_name = '表名称'; #查看某个表的约束名
```

```
ALTER TABLE 从表名 DROP FOREIGN KEY 外键约束名;
```

(2) 第二步查看索引名和删除索引。（注意，只能手动删除）

```
SHOW INDEX FROM 表名称; #查看某个表的索引名
```

```
ALTER TABLE 从表名 DROP INDEX 索引名;
```

举例：

```
mysql> SELECT * FROM information_schema.table_constraints WHERE table_name = 'emp';
```

```
mysql> alter table emp drop foreign key emp_ibfk_1;
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
mysql> show index from emp;
```

```
mysql> alter table emp drop index deptid;
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
mysql> show index from emp;
```

问题1：如果两个表之间有关系（一对一、一对多），比如：员工表和部门表（一对多），它们之间是否一定要建外键约束？

答：不是的

问题2：建和不建外键约束有什么区别？

答：建外键约束，你的操作（创建表、删除表、添加、修改、删除）会受到限制，从语法层面受到限制。例如：在员工表中不可能添加一个员工信息，它的部门的值在部门表中找不到。

不建外键约束，你的操作（创建表、删除表、添加、修改、删除）不受限制，要保证数据的引用完整性，只能依靠程序员的自觉，或者是在Java程序中进行限定。例如：在员工表中，可以添加一个员工的信息，它的部门指定为一个完全不存在的部门。

问题3：那么建和不建外键约束和查询有没有关系？

答：没有

在 MySQL 里，外键约束是有成本的，需要消耗系统资源。对于大并发的 SQL 操作，有可能会不适合。比如大型网站的中央数据库，可能会因为外键约束的系统开销而变得非常慢。所以，MySQL 允许你不使用系统自带的外键约束，在应用层面完成检查数据一致性的逻辑。也就是说，即使你不用外键约束，也要想办法通过应用层面的附加逻辑，来实现外键约束的功能，确保数据的一致性。

6.10 阿里开发规范

【强制】不得使用外键与级联，一切外键概念必须在应用层解决。

说明：（概念解释）学生表中的 student_id 是主键，那么成绩表中的 student_id 则为外键。如果更新学生表中的 student_id，同时触发成绩表中的 student_id 更新，即为级联更新。外键与级联更新适用于单机低并发，不适合分布式、高并发集群；级联更新是强阻塞，存在数据库更新风暴的风险；外键影响数据库的插入速度。

7. CHECK 约束

7.1 作用

检查某个字段的值是否符合xx要求，一般指的是值的范围

2、关键字

CHECK

3、说明：MySQL 5.7 不支持

MySQL5.7 可以使用check约束，但check约束对数据验证没有任何作用。添加数据时，没有任何错误或警告

但是MySQL 8.0中可以使用check约束了。

```
ename varchar(5),  
gender char check ('男' or '女')  
);  
  
insert into employee values(1, '张三', '女');
```

```
mysql> select * from employee;  
+----+----+----+  
| eid | ename | gender |  
+----+----+----+  
| 1 | 张三 | 女 |  
+----+----+----+  
1 row in set (0.00 sec)
```

- 再举例

```
CREATE TABLE temp(  
id INT AUTO_INCREMENT,  
NAME VARCHAR(20),  
age INT CHECK(age > 20),  
PRIMARY KEY(id)  
);
```

- 再举例

```
age tinyint check(age > 20) 或 sex char(2) check(sex in('男','女'))
```

- 再举例

```
CHECK(height>=0 AND height<3)
```

8. DEFAULT约束

8.1 作用

给某个字段/某列指定默认值，一旦设置默认值，在插入数据时，如果此字段没有显式赋值，则赋值为默认值。

8.2 关键字

DEFAULT

8.3 如何给字段加默认值

(1) 建表时

```
create table 表名称(  
    字段名 数据类型 primary key,  
    字段名 数据类型 unique key not null,  
    字段名 数据类型 unique key,  
    字段名 数据类型 not null default 默认值,  
);
```

```
字段名 数据类型 not null default 默认值,  
primary key(字段名),  
unique key(字段名)  
);
```

说明：默认值约束一般不在唯一键和主键列上加

```
create table employee(  
    eid int primary key,  
    ename varchar(20) not null,  
    gender char default '男',  
    tel char(11) not null default '' #默认是空字符串  
)
```

```
mysql> desc employee;  
+-----+-----+-----+-----+-----+  
| Field | Type      | Null | Key | Default | Extra |  
+-----+-----+-----+-----+-----+  
| eid   | int(11)   | NO   | PRI | NULL    |       |  
| ename | varchar(20)| NO   |     | NULL    |       |  
| gender | char(1)   | YES  |     | 男      |       |  
| tel   | char(11)  | NO   |     |         |       |  
+-----+-----+-----+-----+-----+  
4 rows in set (0.00 sec)
```

```
insert into employee values(1, '汪飞', '男', '13700102535'); #成功
```

```
mysql> select * from employee;  
+-----+-----+-----+  
| eid | ename | gender | tel      |  
+-----+-----+-----+  
| 1   | 汪飞  | 男     | 13700102535 |  
+-----+-----+-----+  
1 row in set (0.00 sec)
```

```
insert into employee(eid,ename) values(2, '天琪'); #成功
```

```
mysql> select * from employee;  
+-----+-----+-----+  
| eid | ename | gender | tel      |  
+-----+-----+-----+  
| 1   | 汪飞  | 男     | 13700102535 |  
| 2   | 天琪  | 男     |           |  
+-----+-----+-----+  
2 rows in set (0.00 sec)
```

```
insert into employee(eid,ename) values(3, '二虎');  
#ERROR 1062 (23000): Duplicate entry '' for key 'tel'  
#如果tel有唯一性约束的话会报错，如果tel没有唯一性约束，可以添加成功
```

再举例：

```
NAME VARCHAR(15),  
salary DOUBLE(10,2) DEFAULT 2000  
);
```

(2) 建表后

```
alter table 表名称 modify 字段名 数据类型 default 默认值;
```

#如果这个字段原来有非空约束，你还保留非空约束，那么在加默认值约束时，还得保留非空约束，否则非空约束就被删除了

#同理，在给某个字段加非空约束也一样，如果这个字段原来有默认值约束，你想保留，也要在modify语句中保留默认值约束，否则就删除了

```
alter table 表名称 modify 字段名 数据类型 default 默认值 not null;
```

```
create table employee(  
    eid int primary key,  
    ename varchar(20),  
    gender char,  
    tel char(11) not null  
) ;
```

```
mysql> desc employee;  
+-----+-----+-----+-----+-----+  
| Field | Type      | Null | Key | Default | Extra |  
+-----+-----+-----+-----+-----+  
| eid   | int(11)   | NO  | PRI | NULL   |       |  
| ename | varchar(20)| YES |     | NULL   |       |  
| gender | char(1)   | YES |     | NULL   |       |  
| tel   | char(11)  | NO  |     | NULL   |       |  
+-----+-----+-----+-----+-----+  
4 rows in set (0.00 sec)
```

```
alter table employee modify gender char default '男'; #给gender字段增加默认值约束  
alter table employee modify tel char(11) default ''; #给tel字段增加默认值约束
```

```
mysql> desc employee;  
+-----+-----+-----+-----+-----+  
| Field | Type      | Null | Key | Default | Extra |  
+-----+-----+-----+-----+-----+  
| eid   | int(11)   | NO  | PRI | NULL   |       |  
| ename | varchar(20)| YES |     | NULL   |       |  
| gender | char(1)   | YES |     | 男     |       |  
| tel   | char(11)  | YES |     |       |       |  
+-----+-----+-----+-----+-----+  
4 rows in set (0.00 sec)
```

```
alter table employee modify tel char(11) default '' not null;#给tel字段增加默认值约束，并保留非空约束
```

```

| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| eid   | int(11)  | NO   | PRI | NULL    |       |
| ename | varchar(20) | YES  |     | NULL    |       |
| gender | char(1)  | YES  |     | 男      |       |
| tel   | char(11) | NO   |     |         |       |
+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

```

8.4 如何删除默认值约束

```
alter table 表名称 modify 字段名 数据类型 ; #删除默认值约束，也不保留非空约束
```

```
alter table 表名称 modify 字段名 数据类型 not null; #删除默认值约束，保留非空约束
```

```
alter table employee modify gender char; #删除gender字段默认值约束，如果有非空约束，也一并删除
alter table employee modify tel char(11) not null; #删除tel字段默认值约束，保留非空约束
```

```

mysql> desc employee;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| eid   | int(11)  | NO   | PRI | NULL    |       |
| ename | varchar(20) | YES  |     | NULL    |       |
| gender | char(1)  | YES  |     | NULL    |       |
| tel   | char(11) | NO   |     | NULL    |       |
+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

```

9. 面试

面试1、为什么建表时，加 not null default '' 或 default 0

答：不想让表中出现null值。

面试2、为什么不想要 null 的值

答：(1) 不好比较。null是一种特殊值，比较时只能用专门的is null 和 is not null来比较。碰到运算符，通常返回null。

(2) 效率不高。影响提高索引效果。因此，我们往往在建表时 not null default '' 或 default 0

面试3、带AUTO_INCREMENT约束的字段值是从1开始的吗？ 在MySQL中，默认AUTO_INCREMENT的初始值是1，每新增一条记录，字段值自动加1。设置自增属性（AUTO_INCREMENT）的时候，还可以指定第一条插入记录的自增字段的值，这样新插入的记录的自增字段值从初始值开始递增，如在表中插入第一条记录，同时指定id值为5，则以后插入的记录的id值就会从6开始往上增加。添加主键约束时，往往需要设置字段自动增加属性。

面试4、并不是每个表都可以任意选择存储引擎？ 外键约束（FOREIGN KEY）不能跨引擎使用。

MySQL支持多种存储引擎，每一个表都可以指定一个不同的存储引擎，需要注意的是：外键约束是用来保证数据的参照完整性的，如果表之间需要关联外键，却指定了不同的存储引擎，那么这些表之间是不能创建外键约束的。所以说，存储引擎的选择也不完全是随意的。



让天下没有难学的技术

第14章_视图

讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

1. 常见的数据库对象

对象	描述
表(TABLE)	表是存储数据的逻辑单元，以行和列的形式存在，列就是字段，行就是记录
数据字典	就是系统表，存放数据库相关信息的表。系统表的数据通常由数据库系统维护，程序员通常不应该修改，只可查看
约束 (CONSTRAINT)	执行数据校验的规则，用于保证数据完整性的规则
视图(VIEW)	一个或者多个数据表里的数据的逻辑显示，视图并不存储数据
索引(INDEX)	用于提高查询性能，相当于书的目录
存储过程 (PROCEDURE)	用于完成一次完整的业务处理，没有返回值，但可通过传出参数将多个值传给调用环境
存储函数 (FUNCTION)	用于完成一次特定的计算，具有一个返回值
触发器 (TRIGGER)	相当于一个事件监听器，当数据库发生特定事件后，触发器被触发，完成相应的处理

2. 视图概述

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY
100	Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	17000
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	17000
103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	9000
104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	6000
107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-99	IT_PROG	4200
124	Kevin	Mourgos	KMOURGOS	650.123.5234	16-NOV-99	ST_MAN	5800
141	Trenna	Rajs	TRAJS	650.121.8009	17-OCT-95	ST_CLERK	3500
142	Curtis	Davies	CDAVIES	650.121.2994	29-JAN-97	ST_CLERK	3100
143	Randall	Matos	RMATOS	650.121.2874	15-MAR-98	ST_CLERK	2600
EMPLOYEE_ID		LAST_NAME		SALARY			
	149	Zlotkey		10500	I-JUL-98	ST_CLERK	2500
	174	Abel		11000	I-JAN-00	SA_MAN	10500
	176	Taylor		8600	I-MAY-96	SA_REP	11000
178	Kimberely	Grant	KGRANT	U11.44.1b44.4292b3	I-24-MAY-99	SA_REP	7000
200	Jennifer	Whalen	JWHALEN	515.123.4444	I-17-SEP-87	AD_ASST	4400
201	Michael	Hartstein	MHARTSTE	515.123.5555	I-17-FEB-96	MK_MAN	13000
202	Fat	Fay	PFAY	603.123.8666	I-17-AUG-97	MK_REP	6000
205	Shelley	Higgins	SHIGGINS	515.123.8080	I-07-JUN-94	AC_MGR	12000
206	William	Gietz	WGIETZ	515.123.8181	I-07-JUN-94	AC_ACCOUNT	8300

20 rows selected.

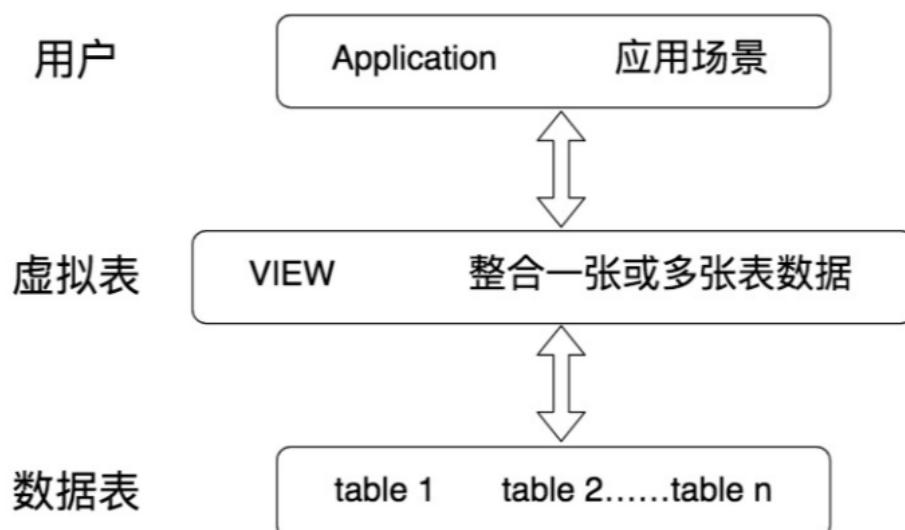
2.1 为什么使用视图？

视图一方面可以帮我们使用表的一部分而不是所有的表，另一方面也可以针对不同的用户制定不同的查询视图。比如，针对一个公司的销售人员，我们只想给他看部分数据，而某些特殊的数据，比如采购的价格，则不会提供给他。再比如，人员薪酬是个敏感的字段，那么只给某个级别以上的人员开放，其他人的查询视图中则不提供这个字段。

刚才讲的只是视图的一个使用场景，实际上视图还有很多作用。最后，我们总结视图的优点。

2.2 视图的理解

- 视图是一种 **虚拟表**，本身是 **不具有数据** 的，占用很少的内存空间，它是 SQL 中的一个重要概念。
- **视图建立在已有表的基础上**，视图赖以建立的这些表称为 **基表**。



- 视图的创建和删除只影响视图本身，不影响对应的基表。但是当对视图中的数据进行增加、删除和修改操作时，数据表中的数据会相应地发生变化，反之亦然。

- 当插入、更新或删除操作时，数据表中的数据会相应地发生变化；反之亦然。
- 视图，是向用户提供基表数据的另一种表现形式。通常情况下，小型项目的数据库可以不使用视图，但是在大型项目中，以及数据表比较复杂的情况下，视图的价值就凸显出来了，它可以帮助我们把经常查询的结果集放到虚拟表中，提升使用效率。理解和使用起来都非常方便。

3. 创建视图

- 在 CREATE VIEW 语句中嵌入子查询

```
CREATE [OR REPLACE]
[ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
VIEW 视图名称 [(字段列表)]
AS 查询语句
[WITH [CASCADED|LOCAL] CHECK OPTION]
```

- 精简版

```
CREATE VIEW 视图名称
AS 查询语句
```

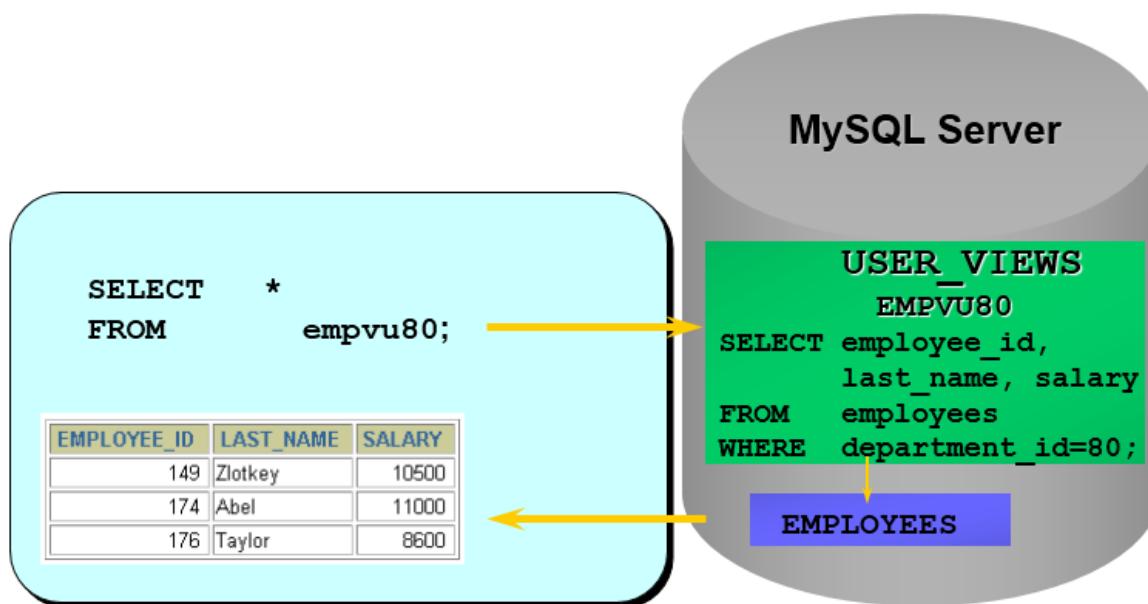
3.1 创建单表视图

举例：

```
CREATE VIEW empvu80
AS
SELECT employee_id, last_name, salary
FROM employees
WHERE department_id = 80;
```

查询视图：

```
SELECT *
FROM salvu80;
```



```
CREATE VIEW emp_year_salary (ename,year_salary)
AS
SELECT ename,salary*12*(1+IFNULL(commission_pct,0))
FROM t_employee;
```

举例：

```
CREATE VIEW salvu50
AS
SELECT employee_id ID_NUMBER, last_name NAME,salary*12 ANN_SALARY
FROM employees
WHERE department_id = 50;
```

说明1：实际上就是我们在 SQL 查询语句的基础上封装了视图 VIEW，这样就会基于 SQL 语句的结果集形成一张虚拟表。

说明2：在创建视图时，没有在视图名后面指定字段列表，则视图中字段列表默认和SELECT语句中的字段列表一致。如果SELECT语句中给字段取了别名，那么视图中的字段名和别名相同。

3.2 创建多表联合视图

举例：

```
CREATE VIEW empview
AS
SELECT employee_id emp_id,last_name NAME,department_name
FROM employees e,departments d
WHERE e.department_id = d.department_id;
```

```
CREATE VIEW emp_dept
AS
SELECT ename,dname
FROM t_employee LEFT JOIN t_department
ON t_employee.did = t_department.did;
```

```
CREATE VIEW dept_sum_vu
(name, minsal, maxsal, avgsal)
AS
SELECT d.department_name, MIN(e.salary), MAX(e.salary), AVG(e.salary)
FROM employees e, departments d
WHERE e.department_id = d.department_id
GROUP BY d.department_name;
```

• 利用视图对数据进行格式化

我们经常需要输出某个格式的内容，比如我们想输出员工姓名和对应的部门名，对应格式为 emp_name(department_name)，就可以使用视图来完成数据格式化的操作：

```
CREATE VIEW emp_depart
AS
SELECT CONCAT(last_name, '(' ,department_name, ')' ) AS emp_dept
FROM employees e JOIN departments d
WHERE e.department_id = d.department_id
```

当我们创建好一张视图之后，还可以在它的基础上继续创建视图。

举例：联合“emp_dept”视图和“emp_year_salary”视图查询员工姓名、部门名称、年薪信息创建“emp_dept_ysalary”视图。

```
CREATE VIEW emp_dept_ysalary
AS
SELECT emp_dept.ename, dname, year_salary
FROM emp_dept INNER JOIN emp_year_salary
ON emp_dept.ename = emp_year_salary.ename;
```

4. 查看视图

语法1：查看数据库的表对象、视图对象

```
SHOW TABLES;
```

语法2：查看视图的结构

```
DESC / DESCRIBE 视图名称;
```

语法3：查看视图的属性信息

```
# 查看视图信息（显示数据表的存储引擎、版本、数据行数和数据大小等）
SHOW TABLE STATUS LIKE '视图名称'\G
```

执行结果显示，注释Comment为VIEW，说明该表为视图，其他的信息为NULL，说明这是一个虚表。

语法4：查看视图的详细定义信息

```
SHOW CREATE VIEW 视图名称;
```

5. 更新视图的数据

5.1 一般情况

MySQL支持使用INSERT、UPDATE和DELETE语句对视图中的数据进行插入、更新和删除操作。当视图中的数据发生变化时，数据表中的数据也会发生变化，反之亦然。

举例：UPDATE操作

```
mysql> SELECT ename, tel FROM emp_tel WHERE ename = '孙洪亮';
+-----+-----+
| ename | tel      |
+-----+-----+
| 孙洪亮 | 13789098765 |
+-----+-----+
1 row in set (0.01 sec)

mysql> UPDATE emp_tel SET tel = '13789091234' WHERE ename = '孙洪亮';
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
+-----+-----+
| ename | tel      |
+-----+-----+
| 孙洪亮 | 13789091234 |
+-----+-----+
1 row in set (0.00 sec)

mysql> SELECT ename,tel FROM t_employee WHERE ename = '孙洪亮';
+-----+-----+
| ename | tel      |
+-----+-----+
| 孙洪亮 | 13789091234 |
+-----+-----+
1 row in set (0.00 sec)
```

举例：DELETE操作

```
mysql> SELECT ename,tel FROM emp_tel WHERE ename = '孙洪亮';
+-----+-----+
| ename | tel      |
+-----+-----+
| 孙洪亮 | 13789091234 |
+-----+-----+
1 row in set (0.00 sec)

mysql> DELETE FROM emp_tel WHERE ename = '孙洪亮';
Query OK, 1 row affected (0.01 sec)

mysql> SELECT ename,tel FROM emp_tel WHERE ename = '孙洪亮';
Empty set (0.00 sec)

mysql> SELECT ename,tel FROM t_employee WHERE ename = '孙洪亮';
Empty set (0.00 sec)
```

5.2 不可更新的视图

要使视图可更新，视图中的行和底层基本表中的行之间必须存在 **一对一** 的关系。另外当视图定义出现如下情况时，视图不支持更新操作：

- 在定义视图的时候指定了“ALGORITHM = TEMPTABLE”，视图将不支持INSERT和DELETE操作；
- 视图中不包含基表中所有被定义为非空又未指定默认值的列，视图将不支持INSERT操作；
- 在定义视图的SELECT语句中使用了 **JOIN联合查询**，视图将不支持INSERT和DELETE操作；
- 在定义视图的SELECT语句后的字段列表中使用了 **数学表达式** 或 **子查询**，视图将不支持INSERT，也不支持UPDATE使用了数学表达式、子查询的字段值；
- 在定义视图的SELECT语句后的字段列表中使用 **DISTINCT**、**聚合函数**、**GROUP BY**、**HAVING**、**UNION** 等，视图将不支持INSERT、UPDATE、DELETE；
- 在定义视图的SELECT语句中包含了子查询，而子查询中引用了FROM后面的表，视图将不支持INSERT、UPDATE、DELETE；
- 视图定义基于一个 **不可更新视图**；
- 常量视图。

举例：

```
--> AS SELECT ename,salary,birthday,tel,email,hiredate,dname  
-> FROM t_employee INNER JOIN t_department  
-> ON t_employee.did = t_department.did ;  
Query OK, 0 rows affected (0.01 sec)
```

```
mysql> INSERT INTO emp_dept(ename,salary,birthday,tel,email,hiredate,dname)  
-> VALUES('张三',15000,'1995-01-08','18201587896',  
-> 'zs@atguigu.com','2022-02-14','新部门');
```

```
#ERROR 1393 (HY000): Can not modify more than one base table through a join view  
'atguigu_chapter9.emp_dept'
```

从上面的SQL执行结果可以看出，在定义视图的SELECT语句中使用了JOIN联合查询，视图将不支持更新操作。

虽然可以更新视图数据，但总的来说，视图作为 **虚拟表**，主要用于 **方便查询**，不建议更新视图的数据。**对视图数据的更改，都是通过对实际数据表里数据的操作来完成的。**

6. 修改、删除视图

6.1 修改视图

方式1：使用CREATE OR REPLACE VIEW 子句 **修改视图**

```
CREATE OR REPLACE VIEW empvu80  
(id_number, name, sal, department_id)  
AS  
SELECT employee_id, first_name || ' ' || last_name, salary, department_id  
FROM employees  
WHERE department_id = 80;
```

说明：CREATE VIEW 子句中各列的别名应和子查询中各列相对应。

方式2：ALTER VIEW

修改视图的语法是：

```
ALTER VIEW 视图名称  
AS  
查询语句
```

6.2 删除视图

- 删除视图只是删除视图的定义，并不会删除基表的数据。
- 删除视图的语法是：

```
DROP VIEW IF EXISTS 视图名称；
```

- 举例：

```
DROP VIEW empvu80;
```

- 说明：基于视图a、b创建了新的视图c，如果将视图a或者视图b删除，会导致视图c的查询失败。这样的视图c需要手动删除或修改，否则影响使用。

7. 总结

7.1 视图优点

1. 操作简单

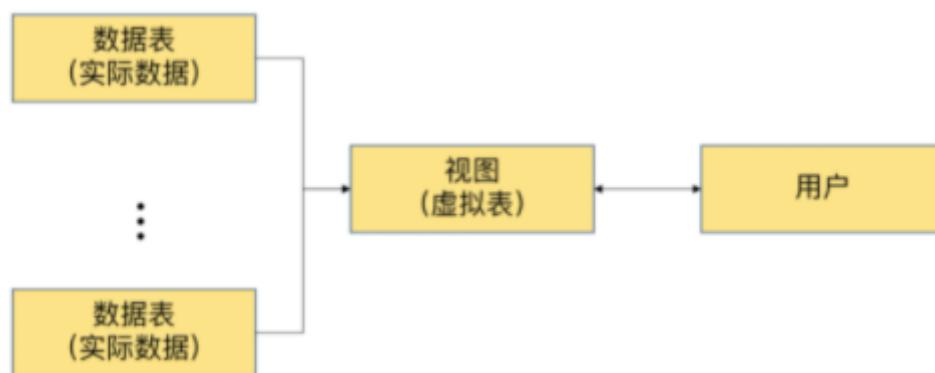
将经常使用的查询操作定义为视图，可以使开发人员不需要关心视图对应的数据表的结构、表与表之间的关联关系，也不需要关心数据表之间的业务逻辑和查询条件，而只需要简单地操作视图即可，极大简化了开发人员对数据库的操作。

2. 减少数据冗余

视图跟实际数据表不一样，它存储的是查询语句。所以，在使用的时候，我们要通过定义视图的查询语句来获取结果集。而视图本身不存储数据，不占用数据存储的资源，减少了数据冗余。

3. 数据安全

MySQL将用户对数据的 **访问限制** 在某些数据的结果集上，而这些数据的结果集可以使用视图来实现。用户不必直接查询或操作数据表。这也可以说为视图具有 **隔离性**。视图相当于在用户和实际的数据表之间加了一层虚拟表。



同时，MySQL可以根据权限将用户对数据的访问限制在某些视图上，**用户不需要查询数据表，可以直接通过视图获取数据表中的信息**。这在一定程度上保障了数据表中数据的安全性。

4. 适应灵活多变的需求 当业务系统的需求发生变化后，如果需要改动数据表的结构，则工作量相对较大，可以使用视图来减少改动的工作量。这种方式在实际工作中使用得比较多。

5. 能够分解复杂的查询逻辑 数据库中如果存在复杂的查询逻辑，则可以将问题进行分解，创建多个视图获取数据，再将创建的多个视图结合起来，完成复杂的查询逻辑。

如果我们在实际数据表的基础上创建了视图，那么，**如果实际数据表的结构变更了，我们就需要及时对相关的视图进行相应的维护**。特别是嵌套的视图（就是在视图的基础上创建视图），维护会变得比较复杂，**可读性不好**，容易变成系统的潜在隐患。因为创建视图的 SQL 查询可能会对字段重命名，也可能包含复杂的逻辑，这些都会增加维护的成本。

实际项目中，如果视图过多，会导致数据库维护成本的问题。

所以，在创建视图的时候，你要结合实际项目需求，综合考虑视图的优点和不足，这样才能正确使用视图，使系统整体达到最优。

第15章_存储过程与函数

讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

MySQL从5.0版本开始支持存储过程和函数。存储过程和函数能够将复杂的SQL逻辑封装在一起，应用程序无须关注存储过程和函数内部复杂的SQL逻辑，而只需要简单地调用存储过程和函数即可。

1. 存储过程概述

1.1 理解

含义：存储过程的英文是 **Stored Procedure**。它的思想很简单，就是一组经过 **预先编译** 的 SQL 语句的封装。

执行过程：存储过程预先存储在 MySQL 服务器上，需要执行的时候，客户端只需要向服务器端发出调用存储过程的命令，服务器端就可以把预先存储好的这一系列 SQL 语句全部执行。

好处：

- 1、简化操作，提高了sql语句的重用性，减少了开发程序员的压力
- 2、减少操作过程中的失误，提高效率
- 3、减少网络传输量（客户端不需要把所有的 SQL 语句通过网络发给服务器）
- 4、减少了 SQL 语句暴露在网上的风险，也提高了数据查询的安全性

和视图、函数的对比：

它和视图有着同样的优点，清晰、安全，还可以减少网络传输量。不过它和视图不同，视图是 **虚拟表**，通常不对底层数据表直接操作，而存储过程是程序化的 SQL，可以 **直接操作底层数据表**，相比于面向集合的操作方式，能够实现一些更复杂的数据处理。

一旦存储过程被创建出来，使用它就像使用函数一样简单，我们直接通过调用存储过程名即可。相较于函数，存储过程是 **没有返回值** 的。

1.2 分类

存储过程的参数类型可以是IN、OUT和INOUT。根据这点分类如下：

- 1、没有参数（无参数无返回）
- 2、仅仅带 IN 类型（有参数无返回）
- 3、仅仅带 OUT 类型（无参数有返回）
- 4、既带 IN 又带 OUT（有参数有返回）
- 5、带 INOUT（有参数有返回）

注意：IN、OUT、INOUT 都可以在一个存储过程中带多个。

2. 创建存储过程

语法：

```
CREATE PROCEDURE 存储过程名(IN|OUT|INOUT 参数名 参数类型, ...)  
[characteristics ...]  
BEGIN  
    存储过程体  
  
END
```

类似于Java中的方法：

```
修饰符 返回类型 方法名(参数类型 参数名, ...){  
  
    方法体;  
}
```

说明：

1、参数前面的符号的意思

- **IN**：当前参数为输入参数，也就是表示入参；
存储过程只是读取这个参数的值。如果没有定义参数种类，**默认就是 IN**，表示输入参数。
- **OUT**：当前参数为输出参数，也就是表示出参；
执行完成之后，调用这个存储过程的客户端或者应用程序就可以读取这个参数返回的值了。
- **INOUT**：当前参数既可以为输入参数，也可以为输出参数。

2、形参类型可以是 MySQL数据库中的任意类型。

3、**characteristics** 表示创建存储过程时指定的对存储过程的约束条件，其取值信息如下：

```
LANGUAGE SQL  
| [NOT] DETERMINISTIC  
| { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }  
| SQL SECURITY { DEFINER | INVOKER }  
| COMMENT 'string'
```

- **LANGUAGE SQL**：说明存储过程执行体是由SQL语句组成的，当前系统支持的语言为SQL。
- **[NOT] DETERMINISTIC**：指明存储过程执行的结果是否确定。DETERMINISTIC表示结果是确定的。每次执行存储过程时，相同的输入会得到相同的输出。NOT DETERMINISTIC表示结果是不确定的，相同的输入可能得到不同的输出。如果没有指定任意一个值，默認為NOT DETERMINISTIC。
- **{ CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }**：指明子程序使用SQL语句的限制。
 - CONTAINS SQL表示当前存储过程的子程序包含SQL语句，但是并不包含读写数据的SQL语句；
 - NO SQL表示当前存储过程的子程序中不包含任何SQL语句；
 - READS SQL DATA表示当前存储过程的子程序中包含读数据的SQL语句；
 - MODIFIES SQL DATA表示当前存储过程的子程序中包含写数据的SQL语句。
 - 默认情况下，系统会指定为CONTAINS SQL。
- **SQL SECURITY { DEFINER | INVOKER }**：执行当前存储过程的权限，即指明哪些用户能够执行当前存储过程。
 - **DEFINER** 表示只有当前存储过程的创建者或者定义者才能执行当前存储过程；
 - **INVOKER** 表示拥有当前存储过程的访问权限的用户能够执行当前存储过程。

4、存储过程体中可以有多条 SQL 语句，如果仅仅一条SQL 语句，则可以省略 BEGIN 和 END

编写存储过程并不是一件简单的事情，可能存储过程中需要复杂的 SQL 语句。

1. **BEGIN...END**: BEGIN...END 中间包含了多个语句，每个语句都以 (;) 号为结束符。
2. **DECLARE**: DECLARE 用来声明变量，使用的位置在于 BEGIN...END 语句中间，而且需要在其他语句使用之前进行变量的声明。
3. **SET**: 赋值语句，用于对变量进行赋值。
4. **SELECT ... INTO**: 把从数据表中查询的结果存放到变量中，也就是为变量赋值。

5、需要设置新的结束标记

DELIMITER 新的结束标记

因为MySQL默认的语句结束符号为分号';'。为了避免与存储过程中SQL语句结束符相冲突，需要使用 DELIMITER 改变存储过程的结束符。

比如：“DELIMITER //”语句的作用是将MySQL的结束符设置为//，并以“END //”结束存储过程。存储过程定义完毕之后再使用“DELIMITER ;”恢复默认结束符。DELIMITER也可以指定其他符号作为结束符。

当使用DELIMITER命令时，应该避免使用反斜杠（'\'）字符，因为反斜线是MySQL的转义字符。

示例：

```
DELIMITER $  
  
CREATE PROCEDURE 存储过程名(IN|OUT|INOUT 参数名 参数类型, ...)  
[characteristics ...]  
BEGIN  
    sql语句1;  
    sql语句2;  
  
END $
```

2.2 代码举例

举例1：创建存储过程select_all_data()，查看 emps 表的所有数据

```
DELIMITER $  
  
CREATE PROCEDURE select_all_data()  
BEGIN  
    SELECT * FROM emps;  
  
END $  
  
DELIMITER ;
```

举例2：创建存储过程avg_employee_salary()，返回所有员工的平均工资

```
CREATE PROCEDURE avg_employee_salary ()
BEGIN
    SELECT AVG(salary) AS avg_salary FROM emps;
END //  
  
DELIMITER ;
```

举例3：创建存储过程show_max_salary()，用来查看“emps”表的最高薪资值。

```
CREATE PROCEDURE show_max_salary()
LANGUAGE SQL
NOT DETERMINISTIC
CONTAINS SQL
SQL SECURITY DEFINER
COMMENT '查看最高薪资'
BEGIN
    SELECT MAX(salary) FROM emps;
END //  
  
DELIMITER ;
```

举例4：创建存储过程show_min_salary()，查看“emps”表的最低薪资值。并将最低薪资通过OUT参数“ms”输出

```
DELIMITER //  
  
CREATE PROCEDURE show_min_salary(OUT ms DOUBLE)
BEGIN
    SELECT MIN(salary) INTO ms FROM emps;
END //  
  
DELIMITER ;
```

举例5：创建存储过程show_someone_salary()，查看“emps”表的某个员工的薪资，并用IN参数empname输入员工姓名。

```
DELIMITER //  
  
CREATE PROCEDURE show_someone_salary(IN empname VARCHAR(20))
BEGIN
    SELECT salary FROM emps WHERE ename = empname;
END //  
  
DELIMITER ;
```

举例6：创建存储过程show_someone_salary2()，查看“emps”表的某个员工的薪资，并用IN参数empname输入员工姓名，用OUT参数empsalary输出员工薪资。

```
CREATE PROCEDURE show_someone_salary2(IN empname VARCHAR(20),OUT empsalary DOUBLE)
BEGIN
    SELECT salary INTO empsalary FROM emps WHERE ename = empname;
END //

DELIMITER ;
```

举例7：创建存储过程show_mgr_name()，查询某个员工领导的姓名，并用INOUT参数“empname”输入员工姓名，输出领导的姓名。

```
DELIMITER //

CREATE PROCEDURE show_mgr_name(INOUT empname VARCHAR(20))
BEGIN
    SELECT ename INTO empname FROM emps
    WHERE eid = (SELECT MID FROM emps WHERE ename=empname);
END //

DELIMITER ;
```

3. 调用存储过程

3.1 调用格式

存储过程有多种调用方法。存储过程必须使用CALL语句调用，并且存储过程和数据库相关，如果要执行其他数据库中的存储过程，需要指定数据库名称，例如CALL dbname.procname。

```
CALL 存储过程名(实参列表)
```

格式：

1、调用in模式的参数：

```
CALL sp1('值');
```

2、调用out模式的参数：

```
SET @name;
CALL sp1(@name);
SELECT @name;
```

3、调用inout模式的参数：

```
SET @name=值;
CALL sp1(@name);
SELECT @name;
```

举例1：

```
DELIMITER //

CREATE PROCEDURE CountProc(IN sid INT, OUT num INT)
BEGIN
    SELECT COUNT(*) INTO num FROM fruits
    WHERE s_id = sid;
END //

DELIMITER ;
```

调用存储过程：

```
mysql> CALL CountProc (101, @num);
Query OK, 1 row affected (0.00 sec)
```

查看返回结果：

```
mysql> SELECT @num;
```

该存储过程返回了指定 s_id=101 的水果商提供的水果种类，返回值存储在 num 变量中，使用 SELECT 查看，返回结果为 3。

举例2： 创建存储过程，实现累加运算，计算 $1+2+\dots+n$ 等于多少。具体的代码如下：

```
DELIMITER //
CREATE PROCEDURE `add_num`(IN n INT)
BEGIN
    DECLARE i INT;
    DECLARE sum INT;

    SET i = 1;
    SET sum = 0;
    WHILE i <= n DO
        SET sum = sum + i;
        SET i = i + 1;
    END WHILE;
    SELECT sum;
END //
DELIMITER ;
```

如果你用的是 Navicat 工具，那么在编写存储过程的时候，Navicat 会自动设置 DELIMITER 为其他符号，我们不需要再进行 DELIMITER 的操作。

直接使用 `CALL add_num(50);` 即可。这里我传入的参数为 50，也就是统计 $1+2+\dots+50$ 的积累之和。

3.3 如何调试

在 MySQL 中，存储过程不像普通的编程语言（比如 VC++、Java 等）那样有专门的集成开发环境。因此，你可以通过 SELECT 语句，把程序执行的中间结果查询出来，来调试一个 SQL 语句的正确性。调试成功之后，把 SELECT 语句后移到下一个 SQL 语句之后，再调试下一个 SQL 语句。这样 **逐步推进**，就可以完成对存储过程中所有操作的调试了。当然，你也可以把存储过程中的 SQL 语句复制出来，逐段单独调试。

前面学习了很多函数，使用这些函数可以对数据进行的各种处理操作，极大地提高用户对数据库的管理效率。MySQL支持自定义函数，定义好之后，调用方式与调用MySQL预定义的系统函数一样。

4.1 语法分析

学过的函数：LENGTH、SUBSTR、CONCAT等

语法格式：

```
CREATE FUNCTION 函数名(参数名 参数类型, ...)  
RETURNS 返回值类型  
[characteristics ...]  
BEGIN  
    函数体      #函数体中肯定有 RETURN 语句  
  
END
```

说明：

1、参数列表：指定参数为IN、OUT或INOUT只对PROCEDURE是合法的，FUNCTION中总是默认为IN参数。

2、RETURNS type语句表示函数返回数据的类型；

RETURNS子句只能对FUNCTION做指定，对函数而言这是 强制 的。它用来指定函数的返回类型，而且函数体必须包含一个 RETURN value 语句。

3、characteristic 创建函数时指定的对函数的约束。取值与创建存储过程时相同，这里不再赘述。

4、函数体也可以用BEGIN...END来表示SQL代码的开始和结束。如果函数体只有一条语句，也可以省略BEGIN...END。

4.2 调用存储函数

在MySQL中，存储函数的使用方法与MySQL内部函数的使用方法是一样的。换言之，用户自己定义的存储函数与MySQL内部函数是一个性质的。区别在于，存储函数是 用户自己定义 的，而内部函数是MySQL的 开发者定义 的。

```
SELECT 函数名(实参列表)
```

4.3 代码举例

举例1：

创建存储函数，名称为email_by_name()，参数定义为空，该函数查询Abel的email，并返回，数据类型为字符串型。

```
CREATE FUNCTION email_by_name()
RETURNS VARCHAR(25)
DETERMINISTIC
CONTAINS SQL
BEGIN
    RETURN (SELECT email FROM employees WHERE last_name = 'Abel');
END //
```

DELIMITER ;

调用：

```
SELECT email_by_name();
```

举例2：

创建存储函数，名称为email_by_id()，参数传入emp_id，该函数查询emp_id的email，并返回，数据类型为字符串型。

```
DELIMITER //

CREATE FUNCTION email_by_id(emp_id INT)
RETURNS VARCHAR(25)
DETERMINISTIC
CONTAINS SQL
BEGIN
    RETURN (SELECT email FROM employees WHERE employee_id = emp_id);
END //
```

DELIMITER ;

调用：

```
SET @emp_id = 102;
SELECT email_by_id(102);
```

举例3：

创建存储函数count_by_id()，参数传入dept_id，该函数查询dept_id部门的员工人数，并返回，数据类型为整型。

```
DELIMITER //

CREATE FUNCTION count_by_id(dept_id INT)
RETURNS INT
LANGUAGE SQL
NOT DETERMINISTIC
READS SQL DATA
SQL SECURITY DEFINER
COMMENT '查询部门平均工资'
BEGIN
    RETURN (SELECT COUNT(*) FROM employees WHERE department_id = dept_id);
END //
```

DELIMITER ;

```
SET @dept_id = 50;
SELECT count_by_id(@dept_id);
```

注意：

若在创建存储函数中报错“`you might want to use the less safe log_bin_trust_function_creators variable`”，有两种处理方法：

- 方式1：加上必要的函数特性“[NOT] DETERMINISTIC”和“{CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA}”
- 方式2：

```
mysql> SET GLOBAL log_bin_trust_function_creators = 1;
```

4.4 对比存储函数和存储过程

	关键字	调用语法	返回值	应用场景
存储过程	PROCEDURE	CALL 存储过程()	理解为有0个或多个	一般用于更新
存储函数	FUNCTION	SELECT 函数()	只能是一个	一般用于查询结果为一个值并返回时

此外，**存储函数可以放在查询语句中使用，存储过程不行**。反之，存储过程的功能更加强大，包括能够执行对表的操作（比如创建表，删除表等）和事务操作，这些功能是存储函数不具备的。

5. 存储过程和函数的查看、修改、删除

5.1 查看

创建完之后，怎么知道我们创建的存储过程、存储函数是否成功了呢？

MySQL存储了存储过程和函数的状态信息，用户可以使用SHOW STATUS语句或SHOW CREATE语句来查看，也可直接从系统的information_schema数据库中查询。这里介绍3种方法。

1. 使用SHOW CREATE语句查看存储过程和函数的创建信息

基本语法结构如下：

```
SHOW CREATE {PROCEDURE | FUNCTION} 存储过程名或函数名
```

举例：

```
SHOW CREATE FUNCTION test_db.CountProc \G
```

2. 使用SHOW STATUS语句查看存储过程和函数的状态信息

基本语法结构如下：

```
SHOW {PROCEDURE | FUNCTION} STATUS [LIKE 'pattern']
```

这个语句返回子程序的特征，如数据库、名字、类型、创建者及创建和修改日期。

```
mysql> SHOW PROCEDURE STATUS LIKE 'SELECT%' \G
***** 1. row ****
Db: test_db
Name: SelectAllData
Type: PROCEDURE
Definer: root@localhost
Modified: 2021-10-16 15:55:07
Created: 2021-10-16 15:55:07
Security_type: DEFINER
Comment:
character_set_client: utf8mb4
collation_connection: utf8mb4_general_ci
Database Collation: utf8mb4_general_ci
1 row in set (0.00 sec)
```

3. 从information_schema.Routines表中查看存储过程和函数的信息

MySQL中存储过程和函数的信息存储在information_schema数据库下的Routines表中。可以通过查询该表的记录来查询存储过程和函数的信息。其基本语法形式如下：

```
SELECT * FROM information_schema.Routines
WHERE ROUTINE_NAME='存储过程或函数的名' [AND ROUTINE_TYPE = {'PROCEDURE|FUNCTION'}];
```

说明：如果在MySQL数据库中存在存储过程和函数名称相同的情况，最好指定ROUTINE_TYPE查询条件来指明查询的是存储过程还是函数。

举例：从Routines表中查询名称为CountProc的存储函数的信息，代码如下：

```
SELECT * FROM information_schema.Routines
WHERE ROUTINE_NAME='count_by_id' AND ROUTINE_TYPE = 'FUNCTION' \G
```

5.2 修改

修改存储过程或函数，不影响存储过程或函数功能，只是修改相关特性。使用ALTER语句实现。

```
ALTER {PROCEDURE | FUNCTION} 存储过程或函数的名 [characteristic ...]
```

其中，characteristic指定存储过程或函数的特性，其取值信息与创建存储过程、函数时的取值信息略有不同。

```
{ CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
| SQL SECURITY { DEFINER | INVOKER }
| COMMENT 'string'
```

- **CONTAINS SQL**，表示子程序包含SQL语句，但不包含读或写数据的语句。
- **NO SQL**，表示子程序中不包含SQL语句。
- **READS SQL DATA**，表示子程序中包含读数据的语句。
- **MODIFIES SQL DATA**，表示子程序中包含写数据的语句。
- **SQL SECURITY { DEFINER | INVOKER }**，指明谁有权限来执行。
 - **DEFINER**，表示只有定义者自己才能够执行。
 - **INVOKER**，表示调用者可以执行。
- **COMMENT 'string'**，表示注释/注自

一个语句的结构是一样的，语句中的所有参数也是一样的。

举例1：

修改存储过程CountProc的定义。将读写权限改为MODIFIES SQL DATA，并指明调用者可以执行，代码如下：

```
ALTER PROCEDURE CountProc  
MODIFIES SQL DATA  
SQL SECURITY INVOKER ;
```

查询修改后的信息：

```
SELECT specific_name,sql_data_access,security_type  
FROM information_schema.`ROUTINES`  
WHERE routine_name = 'CountProc' AND routine_type = 'PROCEDURE' ;
```

结果显示，存储过程修改成功。从查询的结果可以看出，访问数据的权限（SQL_DATA_ACCESS）已经变成MODIFIES SQL DATA，安全类型（SECURITY_TYPE）已经变成INVOKER。

举例2：

修改存储函数CountProc的定义。将读写权限改为READS SQL DATA，并加上注释信息“FIND NAME”，代码如下：

```
ALTER FUNCTION CountProc  
READS SQL DATA  
COMMENT 'FIND NAME' ;
```

存储函数修改成功。从查询的结果可以看出，访问数据的权限（SQL_DATA_ACCESS）已经变成READS SQL DATA，函数注释（ROUTINE_COMMENT）已经变成FIND NAME。

5.3 删除

删除存储过程和函数，可以使用DROP语句，其语法结构如下：

```
DROP {PROCEDURE | FUNCTION} [IF EXISTS] 存储过程或函数的名
```

IF EXISTS：如果程序或函数不存在，它可以防止发生错误，产生一个用SHOW WARNINGS查看的警告。

举例：

```
DROP PROCEDURE CountProc;
```

```
DROP FUNCTION CountProc;
```

6. 关于存储过程使用的争议

尽管存储过程有诸多优点，但是对于存储过程的使用，**一直都存在着很多争议**，比如有些公司对于大型项目要求使用存储过程，而有些公司在手册中明确禁止使用存储过程，为什么这些公司对存储过程的使用需求差别这么大呢？

- 1、存储过程可以一次编译多次使用。** 存储过程只在创建时进行编译，之后的使用都不需要重新编译，这就提升了 SQL 的执行效率。
- 2、可以减少开发工作量。** 将代码 **封装** 成模块，实际上是编程的核心思想之一，这样可以把复杂的问题拆解成不同的模块，然后模块之间可以 **重复使用**，在减少开发工作量的同时，还能保证代码的结构清晰。
- 3、存储过程的安全性强。** 我们在设定存储过程的时候可以 **设置对用户的使用权限**，这样就和视图一样具有较强的安全性。
- 4、可以减少网络传输量。** 因为代码封装到存储过程中，每次使用只需要调用存储过程即可，这样就减少了网络传输量。
- 5、良好的封装性。** 在进行相对复杂的数据库操作时，原本需要使用一条一条的 SQL 语句，可能要连接多次数据库才能完成的操作，现在变成了一次存储过程，只需要 **连接一次即可**。

6.2 缺点

基于上面这些优点，不少大公司都要求大型项目使用存储过程，比如微软、IBM 等公司。但是国内的阿里并不推荐开发人员使用存储过程，这是为什么呢？

阿里开发规范

【强制】禁止使用存储过程，存储过程难以调试和扩展，更没有移植性。

存储过程虽然有诸如上面的好处，但缺点也是很明显的。

- 1、可移植性差。** 存储过程不能跨数据库移植，比如在 MySQL、Oracle 和 SQL Server 里编写的存储过程，在换成其他数据库时都需要重新编写。
- 2、调试困难。** 只有少数 DBMS 支持存储过程的调试。对于复杂的存储过程来说，开发和维护都不容易。虽然也有一些第三方工具可以对存储过程进行调试，但要收费。
- 3、存储过程的版本管理很困难。** 比如数据表索引发生变化了，可能会导致存储过程失效。我们在开发软件的时候往往需要进行版本管理，但是存储过程本身没有版本控制，版本迭代更新的时候很麻烦。
- 4、它不适合高并发的场景。** 高并发的场景需要减少数据库的压力，有时数据库会采用分库分表的方式，而且对可扩展性要求很高，在这种情况下，存储过程会变得难以维护，**增加数据库的压力**，显然就不适用了。

小结：

存储过程既方便，又有局限性。尽管不同的公司对存储过程的态度不一，但是对于我们开发人员来说，不论怎样，掌握存储过程都是必备的技能之一。

第16章 变量、流程控制与游标

讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

1. 变量

在MySQL数据库的存储过程和函数中，可以使用变量来存储查询或计算的中间结果数据，或者输出最终的结果数据。

在 MySQL 数据库中，变量分为 系统变量 以及 用户自定义变量 。

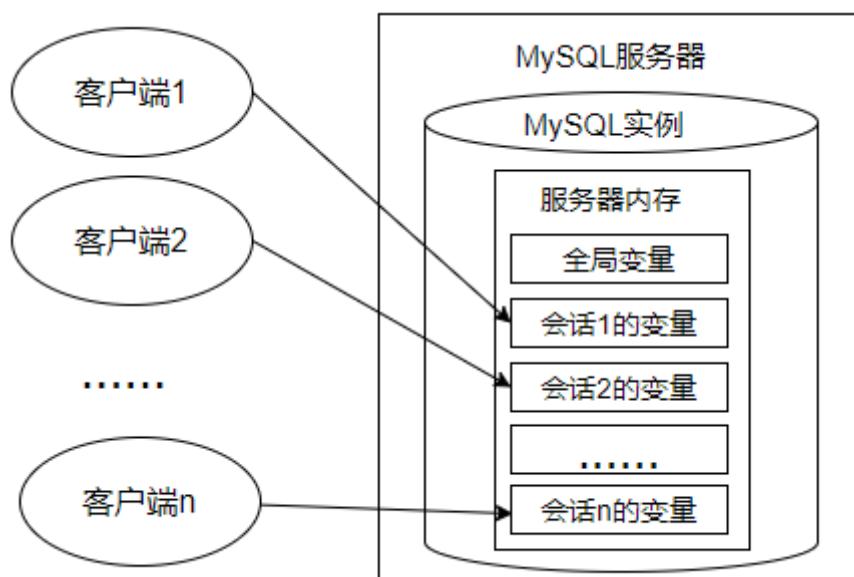
1.1 系统变量

1.1.1 系统变量分类

变量由系统定义，不是用户定义，属于 服务器 层面。启动MySQL服务，生成MySQL服务实例期间，MySQL将为MySQL服务器内存中的系统变量赋值，这些系统变量定义了当前MySQL服务实例的属性、特征。这些系统变量的值要么是 编译MySQL时参数 的默认值，要么是 配置文件 （例如my.ini等）中的参数值。大家可以通过网址 <https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html> 查看MySQL文档的系统变量。

系统变量分为全局系统变量（需要添加 `global` 关键字）以及会话系统变量（需要添加 `session` 关键字），有时也把全局系统变量简称为全局变量，有时也把会话系统变量称为local变量。**如果不写，默认会话级别**。静态变量（在 MySQL 服务实例运行期间它们的值不能使用 `set` 动态修改）属于特殊的全局系统变量。

每一个MySQL客户机成功连接MySQL服务器后，都会产生与之对应的会话。会话期间，MySQL服务实例会在MySQL服务器内存中生成与该会话对应的会话系统变量，这些会话系统变量的初始值是全局系统变量值的复制。如下图：



- 全局系统变量针对于所有会话（连接）有效，但 不能跨重启

- 会话1对某个全局系统变量值的修改会导致会话2中同一个全局系统变量值的修改。

在MySQL中有些系统变量只能是全局的，例如 `max_connections` 用于限制服务器的最大连接数；有些系统变量作用域既可以是全局又可以是会话，例如 `character_set_client` 用于设置客户端的字符集；有些系统变量的作用域只能是当前会话，例如 `pseudo_thread_id` 用于标记当前会话的 MySQL 连接 ID。

1.1.2 查看系统变量

- 查看所有或部分系统变量

```
#查看所有全局变量  
SHOW GLOBAL VARIABLES;  
  
#查看所有会话变量  
SHOW SESSION VARIABLES;  
或  
SHOW VARIABLES;  
  
#查看满足条件的部分系统变量。  
SHOW GLOBAL VARIABLES LIKE '%标识符' ;  
  
#查看满足条件的部分会话变量  
SHOW SESSION VARIABLES LIKE '%标识符' ;
```

举例：

```
SHOW GLOBAL VARIABLES LIKE 'admin_%' ;
```

- 查看指定系统变量

作为 MySQL 编码规范，MySQL 中的系统变量以 **两个“@”** 开头，其中“@@global”仅用于标记全局系统变量，“@@session”仅用于标记会话系统变量。“@@”首先标记会话系统变量，如果会话系统变量不存在，则标记全局系统变量。

```
#查看指定的系统变量的值  
SELECT @@global.变量名;  
  
#查看指定的会话变量的值  
SELECT @@session.变量名;  
#或者  
SELECT @@变量名;
```

- 修改系统变量的值

有些时候，数据库管理员需要修改系统变量的默认值，以便修改当前会话或者MySQL服务实例的属性、特征。具体方法：

方式1：修改MySQL 配置文件，继而修改MySQL系统变量的值（该方法需要重启MySQL服务）

方式2：在MySQL服务运行期间，使用“set”命令重新设置系统变量的值

```
SET @@global.变量名=变量值;
```

#方式2:

```
SET GLOBAL 变量名=变量值;
```

#为某个会话变量赋值

#方式1:

```
SET @@session.变量名=变量值;
```

#方式2:

```
SET SESSION 变量名=变量值;
```

举例:

```
SELECT @@global.autocommit;
```

```
SET GLOBAL autocommit=0;
```

```
SELECT @@session.tx_isolation;
```

```
SET @@session.tx_isolation='read-uncommitted';
```

```
SET GLOBAL max_connections = 1000;
```

```
SELECT @@global.max_connections;
```

1.2 用户变量

1.2.1 用户变量分类

用户变量是用户自己定义的，作为 MySQL 编码规范，MySQL 中的用户变量以一个“@”开头。根据作用范围不同，又分为 **会话用户变量** 和 **局部变量**。

- 会话用户变量：作用域和会话变量一样，只对 **当前连接** 会话有效。
- 局部变量：只在 BEGIN 和 END 语句块中有效。局部变量只能在 **存储过程和函数** 中使用。

1.2.2 会话用户变量

- 变量的定义

#方式1：“=”或“:=”

```
SET @用户变量 = 值;
```

```
SET @用户变量 := 值;
```

#方式2：“:=”或 INTO关键字

```
SELECT @用户变量 := 表达式 [FROM 等子句];
```

```
SELECT 表达式 INTO @用户变量 [FROM 等子句];
```

- 查看用户变量的值（查看、比较、运算等）

```
SELECT @用户变量
```

- 举例

```
SELECT @a;

SELECT @num := COUNT(*) FROM employees;

SELECT @num;

SELECT AVG(salary) INTO @avgsalary FROM employees;

SELECT @avgsalary;

SELECT @big; #查看某个未声明的变量时，将得到NULL值
```

1.2.3 局部变量

定义：可以使用 **DECLARE** 语句定义一个局部变量

作用域：仅仅在定义它的 BEGIN ... END 中有效

位置：只能放在 BEGIN ... END 中，而且只能放在第一句

```
BEGIN
    #声明局部变量
    DECLARE 变量名1 变量数据类型 [DEFAULT 变量默认值];
    DECLARE 变量名2, 变量名3, ... 变量数据类型 [DEFAULT 变量默认值];

    #为局部变量赋值
    SET 变量名1 = 值;
    SELECT 值 INTO 变量名2 [FROM 子句];

    #查看局部变量的值
    SELECT 变量1, 变量2, 变量3;
END
```

1. 定义变量

```
DECLARE 变量名 类型 [default 值]; # 如果没有DEFAULT子句，初始值为NULL
```

举例：

```
DECLARE myparam INT DEFAULT 100;
```

2. 变量赋值

方式1：一般用于赋简单的值

```
SET 变量名=值;
SET 变量名:=值;
```

方式2：一般用于赋表中的字段值

```
SELECT 字段名或表达式 INTO 变量名 FROM 表;
```

3. 使用变量（查看、比较、运算等）

```
SELECT 局部变量名;
```



```
DELIMITER //

CREATE PROCEDURE set_value()
BEGIN
    DECLARE emp_name VARCHAR(25);
    DECLARE sal DOUBLE(10,2);

    SELECT last_name,salary INTO emp_name,sal
    FROM employees
    WHERE employee_id = 102;

    SELECT emp_name,sal;
END //

DELIMITER ;
```

举例2：声明两个变量，求和并打印（分别使用会话用户变量、局部变量的方式实现）

```
#方式1：使用用户变量
SET @m=1;
SET @n=1;
SET @sum=@m+@n;

SELECT @sum;
```

```
#方式2：使用局部变量
DELIMITER //
```

```
CREATE PROCEDURE add_value()
BEGIN
    #局部变量
    DECLARE m INT DEFAULT 1;
    DECLARE n INT DEFAULT 3;
    DECLARE sum INT;

    SET sum = m+n;
    SELECT sum;
END //

DELIMITER ;
```

举例3：创建存储过程“different_salary”查询某员工和他领导的薪资差距，并用IN参数emp_id接收员工id，用OUT参数dif_salary输出薪资差距结果。

```
#声明
DELIMITER //

CREATE PROCEDURE different_salary(IN emp_id INT,OUT dif_salary DOUBLE)
BEGIN
    #声明局部变量
    DECLARE emp_sal,mgr_sal DOUBLE DEFAULT 0.0;
    DECLARE mgr_id INT;

    SELECT salary INTO emp_sal FROM employees WHERE employee_id = emp_id;
    SELECT manager_id INTO mgr_id FROM employees WHERE employee_id = emp_id;
```

```
END //  
  
DELIMITER ;  
  
#调用  
SET @emp_id = 102;  
CALL different_salary(@emp_id,@diff_sal);  
  
#查看  
SELECT @diff_sal;
```

1.2.4 对比会话用户变量与局部变量

	作用域	定义位置	语法
会话用户变量	当前会话	会话的任何地方	加@符号, 不用指定类型
局部变量	定义它的BEGIN END中	BEGIN END的第一句话	一般不用加@, 需要指定类型

2. 定义条件与处理程序

定义条件 是事先定义程序执行过程中可能遇到的问题， 处理程序 定义了在遇到问题时应当采取的处理方式，并且保证存储过程或函数在遇到警告或错误时能继续执行。这样可以增强存储程序处理问题的能力，避免程序异常停止运行。

说明：定义条件和处理程序在存储过程、存储函数中都是支持的。

2.1 案例分析

案例分析： 创建一个名称为“UpdateDataNoCondition”的存储过程。代码如下：

```
DELIMITER //  
  
CREATE PROCEDURE UpdateDataNoCondition()  
BEGIN  
    SET @x = 1;  
    UPDATE employees SET email = NULL WHERE last_name = 'Abel';  
    SET @x = 2;  
    UPDATE employees SET email = 'aabbel' WHERE last_name = 'Abel';  
    SET @x = 3;  
END //  
  
DELIMITER ;
```

调用存储过程：

```
mysql> SELECT @x;
+-----+
| @x   |
+-----+
|    1  |
+-----+
1 row in set (0.00 sec)
```

可以看到，此时@x变量的值为1。结合创建存储过程的SQL语句代码可以得出：在存储过程中未定义条件和处理程序，且当存储过程中执行的SQL语句报错时，MySQL数据库会抛出错误，并退出当前SQL逻辑，不再向下继续执行。

2.2 定义条件

定义条件就是给MySQL中的错误码命名，这有助于存储的程序代码更清晰。它将一个 错误名字 和 指定的 错误条件 关联起来。这个名字可以随后被用在定义处理程序的 `DECLARE HANDLER` 语句中。

定义条件使用`DECLARE`语句，语法格式如下：

```
DECLARE 错误名称 CONDITION FOR 错误码（或错误条件）
```

错误码的说明：

- `MySQL_error_code` 和 `sqlstate_value` 都可以表示MySQL的错误。
 - `MySQL_error_code` 是数值类型错误代码。
 - `sqlstate_value` 是长度为5的字符串类型错误代码。
- 例如，在`ERROR 1418 (HY000)`中，`1418`是`MySQL_error_code`，'`HY000`'是`sqlstate_value`。
- 例如，在`ERROR 1142 (42000)`中，`1142`是`MySQL_error_code`，'`42000`'是`sqlstate_value`。

举例1： 定义“`Field_Not_Be_NULL`”错误名与MySQL中违反非空约束的错误类型是“`ERROR 1048 (23000)`”对应。

```
#使用MySQL_error_code
DECLARE Field_Not_Be_NULL CONDITION FOR 1048;

#使用sqlstate_value
DECLARE Field_Not_Be_NULL CONDITION FOR SQLSTATE '23000';
```

举例2： 定义“`ERROR 1148(42000)`”错误，名称为`command_not_allowed`。

```
#使用MySQL_error_code
DECLARE command_not_allowed CONDITION FOR 1148;

#使用sqlstate_value
DECLARE command_not_allowed CONDITION FOR SQLSTATE '42000';
```

可以为SQL执行过程中发生的某种类型的错误定义特殊的处理程序。定义处理程序时，使用DECLARE语句的语法如下：

```
DECLARE 处理方式 HANDLER FOR 错误类型 处理语句
```

- **处理方式**：处理方式有3个取值：CONTINUE、EXIT、UNDO。
 - CONTINUE：表示遇到错误不处理，继续执行。
 - EXIT：表示遇到错误马上退出。
 - UNDO：表示遇到错误后撤回之前的操作。MySQL中暂时不支持这样的操作。
- **错误类型**（即条件）可以有如下取值：
 - SQLSTATE '字符串错误码'：表示长度为5的sqlstate_value类型的错误代码；
 - MySQL_error_code：匹配数值类型错误代码；
 - 错误名称：表示DECLARE ... CONDITION定义的错误条件名称。
 - SQLWARNING：匹配所有以01开头的SQLSTATE错误代码；
 - NOT_FOUND：匹配所有以02开头的SQLSTATE错误代码；
 - SQLEXCEPTION：匹配所有没有被SQLWARNING或NOT FOUND捕获的SQLSTATE错误代码；
- **处理语句**：如果出现上述条件之一，则采用对应的处理方式，并执行指定的处理语句。语句可以是像“SET 变量 = 值”这样的简单语句，也可以是使用 BEGIN ... END 编写的复合语句。

定义处理程序的几种方式，代码如下：

```
#方法1：捕获sqlstate_value
DECLARE CONTINUE HANDLER FOR SQLSTATE '42S02' SET @info = 'NO SUCH_TABLE';

#方法2：捕获mysql_error_value
DECLARE CONTINUE HANDLER FOR 1146 SET @info = 'NO SUCH_TABLE';

#方法3：先定义条件，再调用
DECLARE no_such_table CONDITION FOR 1146;
DECLARE CONTINUE HANDLER FOR NO SUCH_TABLE SET @info = 'NO SUCH_TABLE';

#方法4：使用SQLWARNING
DECLARE EXIT HANDLER FOR SQLWARNING SET @info = 'ERROR';

#方法5：使用NOT FOUND
DECLARE EXIT HANDLER FOR NOT FOUND SET @info = 'NO SUCH_TABLE';

#方法6：使用SQLEXCEPTION
DECLARE EXIT HANDLER FOR SQLEXCEPTION SET @info = 'ERROR';
```

2.4 案例解决

在存储过程中，定义处理程序，捕获sqlstate_value值，当遇到MySQL_error_code值为1048时，执行CONTINUE操作，并且将@proc_value的值设置为-1。

```
DELIMITER //

CREATE PROCEDURE UpdateDataNoCondition()
BEGIN
    #定义处理程序
    DECLARE CONTINUE HANDLER FOR 1048 SET @proc_value = -1;

    ...

```



```
UPDATE employees SET email = 'aabbel' WHERE last_name = 'Abel';
SET @x = 3;
END //

DELIMITER ;
```

调用过程：

```
mysql> CALL UpdateDataWithCondition();
Query OK, 0 rows affected (0.01 sec)

mysql> SELECT @x,@proc_value;
+-----+-----+
| @x   | @proc_value |
+-----+-----+
|    3 |      -1   |
+-----+-----+
1 row in set (0.00 sec)
```

举例：

创建一个名称为“InsertDataWithCondition”的存储过程，代码如下。

在存储过程中，定义处理程序，捕获sqlstate_value值，当遇到sqlstate_value值为23000时，执行EXIT操作，并且将@proc_value的值设置为-1。

```
#准备工作
CREATE TABLE departments
AS
SELECT * FROM atguigudb.`departments`;

ALTER TABLE departments
ADD CONSTRAINT uk_dept_name UNIQUE(department_id);

DELIMITER //

CREATE PROCEDURE InsertDataWithCondition()
BEGIN
    DECLARE duplicate_entry CONDITION FOR SQLSTATE '23000' ;
    DECLARE EXIT HANDLER FOR duplicate_entry SET @proc_value = -1;

    SET @x = 1;
    INSERT INTO departments(department_name) VALUES('测试');
    SET @x = 2;
    INSERT INTO departments(department_name) VALUES('测试');
    SET @x = 3;
END //

DELIMITER ;
```

调用存储过程：

```
mysql> SELECT @x,@proc_value;
+-----+-----+
| @x   | @proc_value |
+-----+-----+
|    2 |         -1 |
+-----+
1 row in set (0.00 sec)
```

3. 流程控制

解决复杂问题不可能通过一个 SQL 语句完成，我们需要执行多个 SQL 操作。流程控制语句的作用就是控制存储过程中 SQL 语句的执行顺序，是我们完成复杂操作必不可少的一部分。只要是执行的程序，流程就分为三大类：

- **顺序结构**：程序从上往下依次执行
- **分支结构**：程序按条件进行选择执行，从两条或多条路径中选择一条执行
- **循环结构**：程序满足一定条件下，重复执行一组语句

对于MySQL 的流程控制语句主要有 3 类。注意：只能用于存储程序。

- **条件判断语句**：IF 语句和 CASE 语句
- **循环语句**：LOOP、WHILE 和 REPEAT 语句
- **跳转语句**：ITERATE 和 LEAVE 语句

3.1 分支结构之 IF

- IF 语句的语法结构是：

```
IF 表达式1 THEN 操作1
[ELSEIF 表达式2 THEN 操作2] .....
[ELSE 操作N]
END IF
```

根据表达式的结果为TRUE或FALSE执行相应的语句。这里“[]”中的内容是可选的。

- 特点：① 不同的表达式对应不同的操作 ② 使用在begin end中
- **举例1：**

```
IF val IS NULL
    THEN SELECT 'val is null';
ELSE SELECT 'val is not null';

END IF;
```

- **举例2：** 声明存储过程“update_salary_by_eid1”，定义IN参数emp_id，输入员工编号。判断该员工薪资如果低于8000元并且入职时间超过5年，就涨薪500元；否则就不变。

```
DELIMITER //
CREATE PROCEDURE update_salary_by_eid1(IN emp_id INT)
-----
```

```
SELECT salary INTO emp_salary FROM employees WHERE employee_id = emp_id;

SELECT DATEDIFF(CURDATE(),hire_date)/365 INTO hire_year
FROM employees WHERE employee_id = emp_id;

IF emp_salary < 8000 AND hire_year > 5
THEN UPDATE employees SET salary = salary + 500 WHERE employee_id = emp_id;
END IF;
END //

DELIMITER ;
```

- **举例3：** 声明存储过程“update_salary_by_eid2”，定义IN参数emp_id，输入员工编号。判断该员工薪资如果低于9000元并且入职时间超过5年，就涨薪500元；否则就涨薪100元。

```
DELIMITER //

CREATE PROCEDURE update_salary_by_eid2(IN emp_id INT)
BEGIN
DECLARE emp_salary DOUBLE;
DECLARE hire_year DOUBLE;

SELECT salary INTO emp_salary FROM employees WHERE employee_id = emp_id;

SELECT DATEDIFF(CURDATE(),hire_date)/365 INTO hire_year
FROM employees WHERE employee_id = emp_id;

IF emp_salary < 8000 AND hire_year > 5
THEN UPDATE employees SET salary = salary + 500 WHERE employee_id =
emp_id;
ELSE
    UPDATE employees SET salary = salary + 100 WHERE employee_id = emp_id;
END IF;
END //

DELIMITER ;
```

- **举例4：** 声明存储过程“update_salary_by_eid3”，定义IN参数emp_id，输入员工编号。判断该员工薪资如果低于9000元，就更新薪资为9000元；薪资如果大于等于9000元且低于10000的，但是奖金比例为NULL的，就更新奖金比例为0.01；其他的涨薪100元。

```
DELIMITER //

CREATE PROCEDURE update_salary_by_eid3(IN emp_id INT)
BEGIN
DECLARE emp_salary DOUBLE;
DECLARE bonus DECIMAL(3,2);

SELECT salary INTO emp_salary FROM employees WHERE employee_id = emp_id;
SELECT commission_pct INTO bonus FROM employees WHERE employee_id = emp_id;

IF emp_salary < 9000
```

```
emp_id;
ELSE
    UPDATE employees SET salary = salary + 100 WHERE employee_id = emp_id;
END IF;
END //

DELIMITER ;
```

3.2 分支结构之 CASE

CASE 语句的语法结构1：

```
#情况一：类似于switch
CASE 表达式
WHEN 值1 THEN 结果1或语句1(如果是语句，需要加分号)
WHEN 值2 THEN 结果2或语句2(如果是语句，需要加分号)
...
ELSE 结果n或语句n(如果是语句，需要加分号)
END [case] (如果是放在begin end中需要加上case，如果放在select后面不需要)
```

CASE 语句的语法结构2：

```
#情况二：类似于多重if
CASE
WHEN 条件1 THEN 结果1或语句1(如果是语句，需要加分号)
WHEN 条件2 THEN 结果2或语句2(如果是语句，需要加分号)
...
ELSE 结果n或语句n(如果是语句，需要加分号)
END [case] (如果是放在begin end中需要加上case，如果放在select后面不需要)
```

- **举例1：**

使用CASE流程控制语句的第1种格式，判断val值等于1、等于2，或者两者都不等。

```
CASE val
WHEN 1 THEN SELECT 'val is 1';
WHEN 2 THEN SELECT 'val is 2';
ELSE SELECT 'val is not 1 or 2';
END CASE;
```

- **举例2：**

使用CASE流程控制语句的第2种格式，判断val是否为空、小于0、大于0或者等于0。

```
CASE
WHEN val IS NULL THEN SELECT 'val is null';
WHEN val < 0 THEN SELECT 'val is less than 0';
WHEN val > 0 THEN SELECT 'val is greater than 0';
ELSE SELECT 'val is 0';
END CASE;
```

- **举例3：** 声明存储过程“update_salary_by_eid4”，定义IN参数emp_id，输入员工编号。判断该员工薪资如果低于9000元，就更新薪资为9000元；薪资大于等于9000元且低于10000的，但是奖金比例为NULL的，就更新奖金比例为0.01；其他的涨薪100元。

```
DELIMITER //
```

```
BEGIN  
    DECLARE emp_sal DOUBLE;  
    DECLARE bonus DECIMAL(3,2);  
  
    SELECT salary INTO emp_sal FROM employees WHERE employee_id = emp_id;  
    SELECT commission_pct INTO bonus FROM employees WHERE employee_id = emp_id;  
  
    CASE  
        WHEN emp_sal<9000  
            THEN UPDATE employees SET salary=9000 WHERE employee_id = emp_id;  
        WHEN emp_sal<10000 AND bonus IS NULL  
            THEN UPDATE employees SET commission_pct=0.01 WHERE employee_id = emp_id;  
        ELSE  
            UPDATE employees SET salary=salary+100 WHERE employee_id = emp_id;  
    END CASE;  
END //  
  
DELIMITER ;
```

- 举例4：声明存储过程update_salary_by_eid5，定义IN参数emp_id，输入员工编号。判断该员工的入职年限，如果是0年，薪资涨50；如果是1年，薪资涨100；如果是2年，薪资涨200；如果是3年，薪资涨300；如果是4年，薪资涨400；其他的涨薪500。

```
DELIMITER //  
  
CREATE PROCEDURE update_salary_by_eid5(IN emp_id INT)  
BEGIN  
    DECLARE emp_sal DOUBLE;  
    DECLARE hire_year DOUBLE;  
  
    SELECT salary INTO emp_sal FROM employees WHERE employee_id = emp_id;  
  
    SELECT ROUND(DATEDIFF(CURDATE(),hire_date)/365) INTO hire_year FROM employees  
    WHERE employee_id = emp_id;  
  
    CASE hire_year  
        WHEN 0 THEN UPDATE employees SET salary=salary+50 WHERE employee_id = emp_id;  
        WHEN 1 THEN UPDATE employees SET salary=salary+100 WHERE employee_id = emp_id;  
        WHEN 2 THEN UPDATE employees SET salary=salary+200 WHERE employee_id = emp_id;  
        WHEN 3 THEN UPDATE employees SET salary=salary+300 WHERE employee_id = emp_id;  
        WHEN 4 THEN UPDATE employees SET salary=salary+400 WHERE employee_id = emp_id;  
        ELSE UPDATE employees SET salary=salary+500 WHERE employee_id = emp_id;  
    END CASE;  
END //  
  
DELIMITER ;
```

3.3 循环结构之LOOP

LOOP循环语句用来重复执行某些语句。LOOP内的语句一直重复执行直到循环被退出（使用LEAVE子句），跳出循环过程。

LOOP语句的基本格式如下：

```
END LOOP [loop_label]
```

其中，loop_label表示LOOP语句的标注名称，该参数可以省略。

举例1：

使用LOOP语句进行循环操作，id值小于10时将重复执行循环过程。

```
DECLARE id INT DEFAULT 0;
add_loop:LOOP
    SET id = id +1;
    IF id >= 10 THEN LEAVE add_loop;
    END IF;

END LOOP add_loop;
```

举例2：当市场环境变好时，公司为了奖励大家，决定给大家涨工资。声明存储过程

“update_salary_loop()”，声明OUT参数num，输出循环次数。存储过程中实现循环给大家涨薪，薪资涨为原来的1.1倍。直到全公司的平均薪资达到12000结束。并统计循环次数。

```
DELIMITER //

CREATE PROCEDURE update_salary_loop(OUT num INT)
BEGIN
    DECLARE avg_salary DOUBLE;
    DECLARE loop_count INT DEFAULT 0;

    SELECT AVG(salary) INTO avg_salary FROM employees;

    label_loop:LOOP
        IF avg_salary >= 12000 THEN LEAVE label_loop;
        END IF;

        UPDATE employees SET salary = salary * 1.1;
        SET loop_count = loop_count + 1;
        SELECT AVG(salary) INTO avg_salary FROM employees;
    END LOOP label_loop;

    SET num = loop_count;

END //
DELIMITER ;
```

3.4 循环结构之WHILE

WHILE语句创建一个带条件判断的循环过程。WHILE在执行语句执行时，先对指定的表达式进行判断，如果为真，就执行循环内的语句，否则退出循环。WHILE语句的基本格式如下：

```
[while_label:] WHILE 循环条件 DO
    循环体
END WHILE [while_label];
```

while_label为WHILE语句的标注名称；如果循环条件结果为真，WHILE语句内的语句或语句群被执行，直至循环条件为假，退出循环。



WHILE语句示例，值小于10时，将重复执行循环过程，代码如下：

```
DELIMITER //

CREATE PROCEDURE test_while()
BEGIN
    DECLARE i INT DEFAULT 0;

    WHILE i < 10 DO
        SET i = i + 1;
    END WHILE;

    SELECT i;
END //

DELIMITER ;
#调用
CALL test_while();
```

举例2：市场环境不好时，公司为了渡过难关，决定暂时降低大家的薪资。声明存储过程“update_salary_while()”，声明OUT参数num，输出循环次数。存储过程中实现循环给大家降薪，薪资降为原来的90%。直到全公司的平均薪资达到5000结束。并统计循环次数。

```
DELIMITER //

CREATE PROCEDURE update_salary_while(OUT num INT)
BEGIN
    DECLARE avg_sal DOUBLE ;
    DECLARE while_count INT DEFAULT 0;

    SELECT AVG(salary) INTO avg_sal FROM employees;

    WHILE avg_sal > 5000 DO
        UPDATE employees SET salary = salary * 0.9;

        SET while_count = while_count + 1;

        SELECT AVG(salary) INTO avg_sal FROM employees;
    END WHILE;

    SET num = while_count;

END //

DELIMITER ;
```

3.5 循环结构之REPEAT

REPEAT语句创建一个带条件判断的循环过程。与WHILE循环不同的是，REPEAT 循环首先会执行一次循环，然后在 UNTIL 中进行表达式的判断，如果满足条件就退出，即 END REPEAT；如果条件不满足，则会继续执行循环，直到满足退出条件为止。

REPEAT语句的基本格式如下：

```
UNTIL 结束循环的条件表达式  
END REPEAT [repeat_label]
```

repeat_label为REPEAT语句的标注名称，该参数可以省略；REPEAT语句内的语句或语句群被重复，直至expr_condition为真。

举例1：

```
DELIMITER //  
  
CREATE PROCEDURE test_repeat()  
BEGIN  
    DECLARE i INT DEFAULT 0;  
  
    REPEAT  
        SET i = i + 1;  
    UNTIL i >= 10  
    END REPEAT;  
  
    SELECT i;  
END //  
  
DELIMITER ;
```

举例2：当市场环境变好时，公司为了奖励大家，决定给大家涨工资。声明存储过程“update_salary_repeat()”，声明OUT参数num，输出循环次数。存储过程中实现循环给大家涨薪，薪资涨为原来的1.15倍。直到全公司的平均薪资达到13000结束。并统计循环次数。

```
DELIMITER //  
  
CREATE PROCEDURE update_salary_repeat(OUT num INT)  
BEGIN  
    DECLARE avg_sal DOUBLE ;  
    DECLARE repeat_count INT DEFAULT 0;  
  
    SELECT AVG(salary) INTO avg_sal FROM employees;  
  
    REPEAT  
        UPDATE employees SET salary = salary * 1.15;  
  
        SET repeat_count = repeat_count + 1;  
  
        SELECT AVG(salary) INTO avg_sal FROM employees;  
    UNTIL avg_sal >= 13000  
    END REPEAT;  
  
    SET num = repeat_count;  
  
END //  
  
DELIMITER ;
```

对比三种循环结构：

至少执行一次

3.6 跳转语句之LEAVE语句

LEAVE语句：可以用在循环语句内，或者以 BEGIN 和 END 包裹起来的程序体内，表示跳出循环或者跳出程序体的操作。如果你有面向过程的编程语言的使用经验，你可以把 LEAVE 理解为 break。

基本格式如下：

```
LEAVE 标记名
```

其中，label参数表示循环的标志。LEAVE和BEGIN ... END或循环一起被使用。

举例1： 创建存储过程 “leave_begin()”， 声明INT类型的IN参数num。给BEGIN...END加标记名，并在 BEGIN...END中使用IF语句判断num参数的值。

- 如果num<=0，则使用LEAVE语句退出BEGIN...END；
- 如果num=1，则查询“employees”表的平均薪资；
- 如果num=2，则查询“employees”表的最低薪资；
- 如果num>2，则查询“employees”表的最高薪资。

IF语句结束后查询“employees”表的总人数。

```
DELIMITER //

CREATE PROCEDURE leave_begin(IN num INT)

begin_label: BEGIN
    IF num<=0
        THEN LEAVE begin_label;
    ELSEIF num=1
        THEN SELECT AVG(salary) FROM employees;
    ELSEIF num=2
        THEN SELECT MIN(salary) FROM employees;
    ELSE
        SELECT MAX(salary) FROM employees;
    END IF;

    SELECT COUNT(*) FROM employees;
END //


DELIMITER ;
```

举例2：

当市场环境不好时，公司为了渡过难关，决定暂时降低大家的薪资。声明存储过程“leave_while()”，声明 OUT参数num，输出循环次数，存储过程中使用WHILE循环给大家降低薪资为原来薪资的90%，直到全公司的平均薪资小于等于10000，并统计循环次数。

```
DELIMITER //
CREATE PROCEDURE leave_while(OUT num INT)

BEGIN
#
DECLARE avg_sal DOUBLE:#记录平均工资
```

```
SELECT AVG(salary) INTO avg_sal FROM employees; #① 初始化条件

while_label:WHILE TRUE DO #② 循环条件

    #③ 循环体
    IF avg_sal <= 10000 THEN
        LEAVE while_label;
    END IF;

    UPDATE employees SET salary = salary * 0.9;
    SET while_count = while_count + 1;

    #④ 迭代条件
    SELECT AVG(salary) INTO avg_sal FROM employees;

END WHILE;

#赋值
SET num = while_count;

END // 

DELIMITER ;
```

3.7 跳转语句之ITERATE语句

ITERATE语句：只能用在循环语句（LOOP、REPEAT和WHILE语句）内，表示重新开始循环，将执行顺序转到语句段开头处。如果你有面向过程的编程语言的使用经验，你可以把ITERATE理解为continue，意思为“再次循环”。

语句基本格式如下：

```
ITERATE label
```

label参数表示循环的标志。ITERATE语句必须跟在循环标志前面。

举例： 定义局部变量num，初始值为0。循环结构中执行num + 1操作。

- 如果num < 10，则继续执行循环；
- 如果num > 15，则退出循环结构；

```
DELIMITER //
```

```
CREATE PROCEDURE test_iterate()
```

```
BEGIN
    DECLARE num INT DEFAULT 0;
```

```
my_loop:LOOP
    SET num = num + 1;

    IF num < 10
        THEN ITERATE my_loop;
    ELSEIF num > 15
```

```
SELECT '尚硅谷：让天下没有难学的技术';

END LOOP my_loop;

END //

DELIMITER ;
```

4. 游标

4.1 什么是游标（或光标）

虽然我们也可以通过筛选条件 WHERE 和 HAVING，或者是限定返回记录的关键字 LIMIT 返回一条记录，但是，却无法在结果集中像指针一样，向前定位一条记录、向后定位一条记录，或者是 **随意定位到某一条记录**，并对记录的数据进行处理。

这个时候，就可以用到游标。游标，提供了一种灵活的操作方式，让我们能够对结果集中的每一条记录进行定位，并对指向的记录中的数据进行操作的数据结构。**游标让 SQL 这种面向集合的语言有了面向过程开发的能力。**

在 SQL 中，游标是一种临时的数据库对象，可以指向存储在数据库表中的数据行指针。这里游标 **充当了指针的作用**，我们可以通过操作游标来对数据行进行操作。

MySQL中游标可以在存储过程和函数中使用。

比如，我们查询了 employees 数据表中工资高于15000的员工都有哪些：

```
SELECT employee_id, last_name, salary FROM employees
WHERE salary > 15000;
```

	employee_id	last_name	salary
	100	King	31436.95
	101	Kochhar	23096.53
	102	De Haan	22583.30
→	108	Greenberg	16039.27
	145	Russell	18605.54
	146	Partners	17963.98
	147	Errazuriz	16039.27
	168	Ozer	15397.68
	201	Hartstein	17322.40
	205	Higgins	16039.27

这里我们就可以通过游标来操作数据行，如图所示此时游标所在的行是“108”的记录，我们也可以在结果集上滚动游标，指向结果集中的任意一行。

游标必须在声明处理程序之前被声明，并且变量和条件还必须在声明游标或处理程序之前被声明。

如果我们想要使用游标，一般需要经历四个步骤。不同的 DBMS 中，使用游标的语法可能略有不同。

第一步，声明游标

在MySQL中，使用DECLARE关键字来声明游标，其语法的基本形式如下：

```
DECLARE cursor_name CURSOR FOR select_statement;
```

这个语法适用于 MySQL, SQL Server, DB2 和 MariaDB。如果是用 Oracle 或者 PostgreSQL，需要写成：

```
DECLARE cursor_name CURSOR IS select_statement;
```

要使用 SELECT 语句来获取数据结果集，而此时还没有开始遍历数据，这里 select_statement 代表的是 SELECT 语句，返回一个用于创建游标的结果集。

比如：

```
DECLARE cur_emp CURSOR FOR  
SELECT employee_id,salary FROM employees;
```

```
DECLARE cursor_fruit CURSOR FOR  
SELECT f_name, f_price FROM fruits ;
```

第二步，打开游标

打开游标的语法如下：

```
OPEN cursor_name
```

当我们定义好游标之后，如果想要使用游标，必须先打开游标。打开游标的时候 SELECT 语句的查询结果集就会送到游标工作区，为后面游标的 逐条读取 结果集中的记录做准备。

```
OPEN cur_emp ;
```

第三步，使用游标（从游标中取得数据）

语法如下：

```
FETCH cursor_name INTO var_name [ , var_name] ...
```

这句的作用是使用 cursor_name 这个游标来读取当前行，并且将数据保存到 var_name 这个变量中，游标指针指到下一行。如果游标读取的数据行有多个列名，则在 INTO 关键字后面赋值给多个变量名即可。

注意：var_name 必须在声明游标之前就定义好。

```
FETCH cur_emp INTO emp_id, emp_sal ;
```

注意：**游标的查询结果集中的字段数，必须跟 INTO 后面的变量数一致**，否则，在存储过程执行的时候，MySQL 会提示错误。

第四步，关闭游标

```
CLOSE cursor_name
```

有 OPEN 就会有 CLOSE，也就是打开和关闭游标。当我们使用完游标后需要关闭掉该游标。因为游标会占用系统资源，如果不及时关闭，**游标会一直保持到存储过程结束**，影响系统运行的效率。而关闭游标

```
CLOSE cur_emp;
```

4.3 举例

创建存储过程“get_count_by_limit_total_salary()”，声明IN参数 limit_total_salary, DOUBLE类型；声明OUT参数total_count, INT类型。函数的功能可以实现累加薪资最高的几个员工的薪资值，直到薪资总和达到limit_total_salary参数的值，返回累加的人数给total_count。

```
DELIMITER //

CREATE PROCEDURE get_count_by_limit_total_salary(IN limit_total_salary DOUBLE, OUT
total_count INT)

BEGIN
    DECLARE sum_salary DOUBLE DEFAULT 0; #记录累加的总工资
    DECLARE cursor_salary DOUBLE DEFAULT 0; #记录某一个工资值
    DECLARE emp_count INT DEFAULT 0; #记录循环个数
    #定义游标
    DECLARE emp_cursor CURSOR FOR SELECT salary FROM employees ORDER BY salary DESC;
    #打开游标
    OPEN emp_cursor;

    REPEAT
        #使用游标（从游标中获取数据）
        FETCH emp_cursor INTO cursor_salary;

        SET sum_salary = sum_salary + cursor_salary;
        SET emp_count = emp_count + 1;

        UNTIL sum_salary >= limit_total_salary
    END REPEAT;

    SET total_count = emp_count;
    #关闭游标
    CLOSE emp_cursor;

END //
DELIMITER ;
```

4.5 小结

游标是 MySQL 的一个重要的功能，为 **逐条读取** 结果集中的数据，提供了完美的解决方案。跟在应用层面实现相同的功能相比，游标可以在存储程序中使用，效率高，程序也更加简洁。

但同时也会带来一些性能问题，比如在使用游标的過程中，会对数据行进行 **加锁**，这样在业务并发量大的时候，不仅会影响业务之间的效率，还会 **消耗系统资源**，造成内存不足，这是因为游标是在内存中进行的处理。

建议：养成用完之后就关闭的习惯，这样才能提高系统的整体效率。

在MySQL数据库中，全局变量可以通过SET GLOBAL语句来设置。例如，设置服务器语句超时的限制，可以通过设置系统变量max_execution_time来实现：

```
SET GLOBAL MAX_EXECUTION_TIME=2000;
```

使用SET GLOBAL语句设置的变量值只会临时生效。数据库重启后，服务器又会从MySQL配置文件中读取变量的默认值。MySQL 8.0版本新增了SET PERSIST命令。例如，设置服务器的最大连接数为1000：

```
SET PERSIST global max_connections = 1000;
```

MySQL会将该命令的配置保存到数据目录下的`mysqld-auto.cnf`文件中，下次启动时会读取该文件，用其中的配置来覆盖默认的配置文件。

举例：

查看全局变量max_connections的值，结果如下：

```
mysql> show variables like '%max_connections%';
+-----+-----+
| Variable_name      | Value |
+-----+-----+
| max_connections    | 151   |
| mysqlx_max_connections | 100   |
+-----+-----+
2 rows in set, 1 warning (0.00 sec)
```

设置全局变量max_connections的值：

```
mysql> set persist max_connections=1000;
Query OK, 0 rows affected (0.00 sec)
```

重启MySQL服务器，再次查询max_connections的值：

```
mysql> show variables like '%max_connections%';
+-----+-----+
| Variable_name      | Value |
+-----+-----+
| max_connections    | 1000  |
| mysqlx_max_connections | 100   |
+-----+-----+
2 rows in set, 1 warning (0.00 sec)
```

第17章_触发器

讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

在实际开发中，我们经常会遇到这样的情况：有2个或者多个相互关联的表，如商品信息和库存信息分别存放在2个不同的数据表中，我们在添加一条新商品记录的时候，为了保证数据的完整性，必须同时在库存表中添加一条库存记录。

这样一来，我们就必须把这两个关联的操作步骤写到程序里面，而且要用**事务**包裹起来，确保这两个操作成为一个**原子操作**，要么全部执行，要么全部不执行。要是遇到特殊情况，可能还需要对数据进行手动维护，这样就很**容易忘记其中的一步**，导致数据缺失。

这个时候，咱们可以使用触发器。**你可以创建一个触发器，让商品信息数据的插入操作自动触发库存数据的插入操作。**这样一来，就不用担心因为忘记添加库存数据而导致的数据缺失了。

1. 触发器概述

MySQL从**5.0.2**版本开始支持触发器。MySQL的触发器和存储过程一样，都是嵌入到MySQL服务器的一段程序。

触发器是由**事件来触发**某个操作，这些事件包括**INSERT**、**UPDATE**、**DELETE**事件。所谓事件就是指用户的动作或者触发某项行为。如果定义了触发程序，当数据库执行这些语句时候，就相当于事件发生了，就会**自动**激发触发器执行相应的操作。

当对数据表中的数据执行插入、更新和删除操作，需要自动执行一些数据库逻辑时，可以使用触发器来实现。

2. 触发器的创建

2.1 创建触发器语法

创建触发器的语法结构是：

```
CREATE TRIGGER 触发器名称
{BEFORE|AFTER} {INSERT|UPDATE|DELETE} ON 表名
FOR EACH ROW
触发器执行的语句块；
```

说明：

- **表名**：表示触发器监控的对象。
- **BEFORE | AFTER**：表示触发的时间。BEFORE 表示在事件之前触发；AFTER 表示在事件之后触发。
- **INSERT | UPDATE | DELETE**：表示触发的事件。
 - INSERT 表示插入记录时触发；
 - UPDATE 表示更新记录时触发；
 - DELETE 表示删除记录时触发；

2.2 代码举例

举例1：

1、创建数据表：

```
CREATE TABLE test_trigger (
    id INT PRIMARY KEY AUTO_INCREMENT,
    t_note VARCHAR(30)
);

CREATE TABLE test_trigger_log (
    id INT PRIMARY KEY AUTO_INCREMENT,
    t_log VARCHAR(30)
);
```

2、创建触发器：创建名称为before_insert的触发器，向test_trigger数据表插入数据之前，向test_trigger_log数据表中插入before_insert的日志信息。

```
DELIMITER //

CREATE TRIGGER before_insert
BEFORE INSERT ON test_trigger
FOR EACH ROW
BEGIN
    INSERT INTO test_trigger_log (t_log)
    VALUES('before_insert');

END //

DELIMITER ;
```

3、向test_trigger数据表中插入数据

```
INSERT INTO test_trigger (t_note) VALUES ('测试 BEFORE INSERT 触发器');
```

4、查看test_trigger_log数据表中的数据

```
mysql> SELECT * FROM test_trigger_log;
+----+-----+
| id | t_log      |
+----+-----+
| 1  | before_insert |
+----+-----+
1 row in set (0.00 sec)
```

举例2：

1、创建名称为after_insert的触发器，向test_trigger数据表插入数据之后，向test_trigger_log数据表中插入after_insert的日志信息。

```
CREATE TRIGGER after_insert
AFTER INSERT ON test_trigger
FOR EACH ROW
BEGIN
    INSERT INTO test_trigger_log (t_log)
    VALUES('after_insert');
END //;

DELIMITER ;
```

2、向test_trigger数据表中插入数据。

```
INSERT INTO test_trigger (t_note) VALUES ('测试 AFTER INSERT 触发器');
```

3、查看test_trigger_log数据表中的数据

```
mysql> SELECT * FROM test_trigger_log;
+----+-----+
| id | t_log      |
+----+-----+
| 1  | before_insert |
| 2  | before_insert |
| 3  | after_insert  |
+----+-----+
3 rows in set (0.00 sec)
```

举例3： 定义触发器“salary_check_trigger”，基于员工表“employees”的INSERT事件，在INSERT之前检查将要添加的新员工薪资是否大于他领导的薪资，如果大于领导薪资，则报sqlstate_value为'HY000'的错误，从而使得添加失败。

```
DELIMITER //

CREATE TRIGGER salary_check_trigger
BEFORE INSERT ON employees FOR EACH ROW
BEGIN
    DECLARE mgrsalary DOUBLE;
    SELECT salary INTO mgrsalary FROM employees WHERE employee_id = NEW.manager_id;

    IF NEW.salary > mgrsalary THEN
        SIGNAL SQLSTATE 'HY000' SET MESSAGE_TEXT = '薪资高于领导薪资错误';
    END IF;
END //

DELIMITER ;
```

上面触发器声明过程中的NEW关键字代表INSERT添加语句的新记录。

3. 查看、删除触发器

查看触发器是查看数据库中已经存在的触发器的定义、状态和语法信息等。

方式1：查看当前数据库的所有触发器的定义

```
SHOW TRIGGERS\G
```

方式2：查看当前数据库中某个触发器的定义

```
SHOW CREATE TRIGGER 触发器名
```

方式3：从系统库information_schema的TRIGGERS表中查询“salary_check_trigger”触发器的信息。

```
SELECT * FROM information_schema.TRIGGERS;
```

3.2 删除触发器

触发器也是数据库对象，删除触发器也用DROP语句，语法格式如下：

```
DROP TRIGGER IF EXISTS 触发器名称;
```

4. 触发器的优缺点

4.1 优点

1、触发器可以确保数据的完整性。

假设我们用 **进货单头表** (demo.importhead) 来保存进货单的总体信息，包括进货单编号、供货商编号、仓库编号、总计进货数量、总计进货金额和验收日期。

listnumber (进货单编号)	supplierid (供货商编号)	stockid (参库编号)	quantity (总计数量)	importvalue (总计金额)	confirmationdate (验收日期)

用 **进货单明细表** (demo.importdetails) 来保存进货商品的明细，包括进货单编号、商品编号、进货数量、进货价格和进货金额。

listnumber (进货单编号)	itemnumber (商品编号)	quantity (进货数量)	importprice (进货价格)	importvalue (进货金额)

额就不等于进货单明细表中数量合计和金额合计了，这就是数据不一致。

为了解决这个问题，我们就可以使用触发器，**规定每当进货单明细表有数据插入、修改和删除的操作时，自动触发 2 步操作：**

- 1) 重新计算进货单明细表中的数量合计和金额合计；
- 2) 用第一步中计算出来的值更新进货单头表中的合计数量与合计金额。

这样一来，进货单头表中的合计数量与合计金额的值，就始终与进货单明细表中计算出来的合计数量与合计金额的值相同，数据就是一致的，不会互相矛盾。

2、触发器可以帮助我们记录操作日志。

利用触发器，可以具体记录什么时间发生了什么。比如，记录修改会员储值金额的触发器，就是一个很好的例子。这对我们还原操作执行时的具体场景，更好地定位问题原因很有帮助。

3、触发器还可以用在操作数据前，对数据进行合法性检查。

比如，超市进货的时候，需要库管录入进货价格。但是，人为操作很容易犯错误，比如说在录入数量的时候，把条形码扫进去了；录入金额的时候，看串了行，录入的价格远超售价，导致账面上的巨亏……这些都可以通过触发器，在实际插入或者更新操作之前，对相应的数据进行检查，及时提示错误，防止错误数据进入系统。

4.2 缺点

1、触发器最大的一个问题就是可读性差。

因为触发器存储在数据库中，并且由事件驱动，这就意味着触发器有可能**不受应用层的控制**。这对系统维护是非常有挑战的。

比如，创建触发器用于修改会员储值操作。如果触发器中的操作出了问题，会导致会员储值金额更新失败。我用下面的代码演示一下：

```
mysql> update demo.membermaster set memberdeposit=20 where memberid = 2;
ERROR 1054 (42S22): Unknown column 'aa' in 'field list'
```

结果显示，系统提示错误，字段“aa”不存在。

这是因为，触发器中的数据插入操作多了一个字段，系统提示错误。可是，如果你不了解这个触发器，很可能会认为是更新语句本身的问题，或者是会员信息表的结构出了问题。说不定你还会给会员信息表添加一个叫“aa”的字段，试图解决这个问题，结果只能是白费力。

2、相关数据的变更，可能会导致触发器出错。

特别是数据表结构的变更，都可能会导致触发器出错，进而影响数据操作的正常运行。这些都会由于触发器本身的隐蔽性，影响到应用中错误原因排查的效率。

4.3 注意点

注意，如果在子表中定义了外键约束，并且外键指定了ON UPDATE/DELETE CASCADE/SET NULL子句，此时修改父表被引用的键值或删除父表被引用的记录行时，也会引起子表的修改和删除操作，此时基于子表的UPDATE和DELETE语句定义的触发器并不会被激活。

例如：基于子表员工表（t_employee）的DELETE语句定义了触发器t1，而子表的部门编号（did）字段定义了外键约束引用了父表部门表（t_department）的主键列部门编号（did），并且该外键加了“ON DELETE SET NULL”子句，那么如果此时删除父表部门表（t_department）在子表员工表（t_employee）



让天下没有难学的技术

第18章_MySQL8其它新特性

讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

1. MySQL8新特性概述

MySQL从5.7版本直接跳跃发布了8.0版本，可见这是一个令人兴奋的里程碑版本。MySQL 8版本在功能上做了显著的改进与增强，开发者对MySQL的源代码进行了重构，最突出的一点是多MySQL Optimizer优化器进行了改进。不仅在速度上得到了改善，还为用户带来了更好的性能和更棒的体验。

1.1 MySQL8.0 新增特性

1. 更简便的NoSQL支持 NoSQL泛指非关系型数据库和数据存储。随着互联网平台的规模飞速发展，传统的关系型数据库已经越来越不能满足需求。从5.6版本开始，MySQL就开始支持简单的NoSQL存储功能。MySQL 8对这一功能做了优化，以更灵活的方式实现NoSQL功能，不再依赖模式（schema）。

2. 更好的索引 在查询中，正确地使用索引可以提高查询的效率。MySQL 8中新增了`隐藏索引`和`降序索引`。隐藏索引可以用来测试去掉索引对查询性能的影响。在查询中混合存在多列索引时，使用降序索引可以提高查询的性能。

3. 更完善的JSON支持 MySQL从5.7开始支持原生JSON数据的存储，MySQL 8对这一功能做了优化，增加了聚合函数`JSON_ARRAYAGG()`和`JSON_OBJECTAGG()`，将参数聚合为JSON数组或对象，新增了行内操作符`->`，是列路径运算符`->`的增强，对JSON排序做了提升，并优化了JSON的更新操作。

4. 安全和账户管理 MySQL 8中新增了`caching_sha2_password`授权插件、角色、密码历史记录和FIPS模式支持，这些特性提高了数据库的安全性和性能，使数据库管理员能够更灵活地进行账户管理工作。

5. InnoDB的变化 InnoDB是MySQL默认的存储引擎，是事务型数据库的首选引擎，支持事务安全表（ACID），支持行锁定和外键。在MySQL 8版本中，InnoDB在自增、索引、加密、死锁、共享锁等方面做了大量的`改进和优化`，并且支持原子数据定义语言（DDL），提高了数据安全性，对事务提供更好的支持。

6. 数据字典 在之前的MySQL版本中，字典数据都存储在元数据文件和非事务表中。从MySQL 8开始新增了事务数据字典，在这个字典里存储着数据库对象信息，这些数据字典存储在内部事务表中。

7. 原子数据定义语句 MySQL 8开始支持原子数据定义语句（Automic DDL），即`原子DDL`。目前，只有InnoDB存储引擎支持原子DDL。原子数据定义语句（DDL）将与DDL操作相关的数据字典更新、存储引擎操作、二进制日志写入结合到一个单独的原子事务中，这使得即使服务器崩溃，事务也会提交或回滚。使用支持原子操作的存储引擎所创建的表，在执行`DROP TABLE`、`CREATE TABLE`、`ALTER TABLE`、`RENAME TABLE`、`TRUNCATE TABLE`、`CREATE TABLESPACE`、`DROP TABLESPACE`等操作时，都支持原子操作，即事务要么完全操作成功，要么失败后回滚，不再进行部分提交。对于从MySQL 5.7复制到MySQL 8版本中的语句，可以添加`IF EXISTS`或`IF NOT EXISTS`语句来避免发生错误。

8. 资源管理 MySQL 8开始支持创建和管理资源组，允许将服务器内运行的线程分配给特定的分组，以便线程根据组内可用资源执行。组属性能够控制组内资源，启用或限制组内资源消耗。数据库管理员能够根据不同的工作负载适当地更改这些属性。目前，CPU时间是可控资源，由“虚拟CPU”这个概念来表示，此术语包含CPU的核心数，超线程，硬件线程等等。服务器在启动时确定可用的虚拟CPU数量。拥有对应权限的数据库管理员可以将这些CPU与资源组关联，并为资源组分配线程。资源组组件为MySQL中

属性，除去名字和类型，其他属性都可在创建之后进行更改。在一些平台下，或进行了某些MySQL的配置时，资源管理的功能将受到限制，甚至不可用。例如，如果安装了线程池插件，或者使用的是macOS系统，资源管理将处于不可用状态。在FreeBSD和Solaris系统中，资源线程优先级将失效。在Linux系统中，只有配置了CAP_SYS_NICE属性，资源管理优先级才能发挥作用。

9.字符集支持 MySQL 8中默认的字符集由 `latin1` 更改为 `utf8mb4`，并首次增加了日语所特定使用的集合，`utf8mb4_ja_0900_as_cs`。

10.优化器增强 MySQL优化器开始支持隐藏索引和降序索引。隐藏索引不会被优化器使用，验证索引的必要性时不需要删除索引，先将索引隐藏，如果优化器性能无影响就可以真正地删除索引。降序索引允许优化器对多个列进行排序，并且允许排序顺序不一致。

11.公用表表达式 公用表表达式（Common Table Expressions）简称为CTE，MySQL现在支持递归和非递归两种形式的CTE。CTE通过在SELECT语句或其他特定语句前 使用WITH语句对临时结果集 进行命名。

基础语法如下：

```
WITH cte_name (col_name1,col_name2 ...) AS (Subquery)
SELECT * FROM cte_name;
```

Subquery代表子查询，子查询前使用WITH语句将结果集命名为cte_name，在后续的查询中即可使用cte_name进行查询。

12.窗口函数 MySQL 8开始支持窗口函数。在之前的版本中已存在的大部分 聚合函数 在MySQL 8中也可以作为窗口函数来使用。

函数名称	描述
<code>CUME_DIST()</code>	累计的分布值
<code>DENSE_RANK()</code>	对当前记录不间断排序
<code>FIRST_VALUE()</code>	返回窗口首行记录的对应字段值
<code>LAG()</code>	返回对应字段的前 N 行记录
<code>LAST_VALUE()</code>	返回窗口尾行记录的对应字段值
<code>LEAD()</code>	返回对应字段的后 N 行记录
<code>NTH_VALUE()</code>	返回第 N 条记录对应的字段值
<code>NTILE()</code>	将区划分为 N 组，并返回组的数量
<code>PERCENT_RANK()</code>	返回 0 到 1 之间的小数，表示某个字段值在数据分区中的排名
<code>RANK()</code>	返回分区内每条记录对应的排名
<code>ROW_NUMBER()</code>	返回每一条记录对应的序号，且不重复

13.正则表达式支持 MySQL在8.0.4以后的版本中采用支持Unicode的国际化组件库实现正则表达式操作，这种方式不仅能提供完全的Unicode支持，而且是多字节安全编码。MySQL增加了`REGEXP_LIKE()`、`EGEXP_INSTR()`、`REGEXP_REPLACE()`和`REGEXP_SUBSTR()`等函数来提升性能。另外，`regexp_stack_limit`和`regexp_time_limit`系统变量能够通过匹配引擎来控制资源消耗。

14.内部临时表 `TempTable`存储引擎取代`MEMORY`存储引擎成为内部临时表的默认存储引擎。`TempTable`存储引擎为`VARCHAR`和`VARBINARY`列提供高效存储。`internal_tmp_mem_storage_engine`会话变量定义了内部临时表的存储引擎，可选的值有两个，`TempTable`和`MEMORY`，其中`TempTable`为默认的存储引擎。`temptable_max_ram`系统配置项定义了`TempTable`存储引擎可使用的最大内存数量。

15.日志记录 在MySQL 8中错误日志子系统由一系列MySQL组件构成。这些组件的构成由系统变量`log_error_services`来配置，能够实现日志事件的过滤和写入。

员特权。

17.增强的MySQL复制 MySQL 8复制支持对 JSON文档 进行部分更新的 二进制日志记录，该记录 使用紧凑的二进制格式，从而节省记录完整JSON文档的空间。当使用基于语句的日志记录时，这种紧凑的日志记录会自动完成，并且可以通过将新的binlog_row_value_options系统变量值设置为PARTIAL_JSON来启用。

1.2 MySQL8.0移除的旧特性

在MySQL 5.7版本上开发的应用程序如果使用了MySQL8.0 移除的特性，语句可能会失败，或者产生不同的执行结果。为了避免这些问题，对于使用了移除特性的应用，应当尽力修正避免使用这些特性，并尽可能使用替代方法。

1. 查询缓存 查询缓存已被移除，删除的项有： **(1) 语句：** FLUSH QUERY CACHE和RESET QUERY CACHE。
(2) 系统变量： query_cache_limit、query_cache_min_res_unit、query_cache_size、query_cache_type、query_cache_wlock_invalidate。
(3) 状态变量： Qcache_free_blocks、Qcache_free_memory、Qcache_hits、Qcache_inserts、Qcache_lowmem_prunes、Qcache_not_cached、Qcache_queries_in_cache、Qcache_total_blocks。
(4) 线程状态： checking privileges on cached query、checking query cache for query、invalidating query cache entries、sending cached result to client、storing result in query cache、waiting for query cache lock。

2. 加密相关 删除的加密相关的内容有：ENCODE()、DECODE()、ENCRYPT()、DES_ENCRYPT()和DES_DECRYPT()函数，配置项des-key-file，系统变量have_crypt，FLUSH语句的DES_KEY_FILE选项，HAVE_CRYPT CMake选项。对于移除的ENCRYPT()函数，考虑使用SHA2()替代，对于其他移除的函数，使用AES_ENCRYPT()和AES_DECRYPT()替代。

3. 空间函数相关 在MySQL 5.7版本中，多个空间函数已被标记为过时。这些过时函数在MySQL 8中都已被移除，只保留了对应的ST_和MBR函数。

4.\N和NULL 在SQL语句中，解析器不再将\N视为NULL，所以在SQL语句中应使用NULL代替\N。这项变化不会影响使用LOAD DATA INFILE或者SELECT...INTO OUTFILE操作文件的导入和导出。在这类操作中，NULL仍等同于\N。

5. mysql_install_db 在MySQL分布中，已移除了mysql_install_db程序，数据字典初始化需要调用带著--initialize或者--initialize-insecure选项的mysqld来代替实现。另外，--bootstrap和INSTALL_SCRIPTDIR CMake也被删除。

6.通用分区处理程序 通用分区处理程序已从MySQL服务中被移除。为了实现给定表分区，表所使用的存储引擎需要自有的分区处理程序。提供本地分区支持的MySQL存储引擎有两个，即InnoDB和NDB，而在MySQL 8中只支持InnoDB。

7.系统和状态变量信息 在INFORMATION_SCHEMA数据库中，对系统和状态变量信息不再进行维护。GLOBAL_VARIABLES、SESSION_VARIABLES、GLOBAL_STATUS、SESSION_STATUS表都已被删除。另外，系统变量show_compatibility_56也被删除。被删除的状态变量有Slave_heartbeat_period、Slave_last_heartbeat、Slave_received_heartbeats、Slave_retried_transactions、Slave_running。以上被删除的内容都可使用性能模式中对应的内容进行替代。

8.mysql_plugin工具 mysql_plugin工具用来配置MySQL服务器插件，现已被删除，可使用--plugin-load或--plugin-load-add选项在服务器启动时加载插件或者在运行时使用INSTALL PLUGIN语句加载插件来替代该工具。

2. 新特性1：窗口函数

假设我现在有这样一个数据表，它显示了某购物网站在每个城市每个区的销售额：

```
CREATE TABLE sales(
    id INT PRIMARY KEY AUTO_INCREMENT,
    city VARCHAR(15),
    county VARCHAR(15),
    sales_value DECIMAL

);

INSERT INTO sales(city, county, sales_value)
VALUES
    ('北京', '海淀', 10.00),
    ('北京', '朝阳', 20.00),
    ('上海', '黄埔', 30.00),
    ('上海', '长宁', 10.00);
```

查询：

```
mysql> SELECT * FROM sales;
+----+----+----+-----+
| id | city | county | sales_value |
+----+----+----+-----+
| 1 | 北京 | 海淀 | 10 |
| 2 | 北京 | 朝阳 | 20 |
| 3 | 上海 | 黄浦 | 30 |
| 4 | 上海 | 长宁 | 10 |
+----+----+----+-----+
4 rows in set (0.00 sec)
```

需求：现在计算这个网站在每个城市的销售总额、在全国的销售总额、每个区的销售额占所在城市销售中的比率，以及占总销售额中的比率。

如果用分组和聚合函数，就需要分好几步来计算。

第一步，计算总销售金额，并存入临时表 a：

```
CREATE TEMPORARY TABLE a          -- 创建临时表
SELECT SUM(sales_value) AS sales_value -- 计算总计金额
FROM sales;
```

查看一下临时表 a：

```
mysql> SELECT * FROM a;
+-----+
| sales_value |
+-----+
|      70 |
+-----+
1 row in set (0.00 sec)
```

第二步，计算每个城市的销售总额并存入临时表 b：

```
CREATE TEMPORARY TABLE b          -- 创建临时表
SELECT city, SUM(sales_value) AS sales_value -- 计算城市销售合计
FROM sales
```

```
mysql> SELECT * FROM b;
+-----+-----+
| city | sales_value |
+-----+-----+
| 北京 |      30 |
| 上海 |      40 |
+-----+-----+
2 rows in set (0.00 sec)
```

第三步，计算各区的销售占所在城市的总计金额的比例，和占全部销售总计金额的比例。我们可以通过下面的连接查询获得需要的结果：

```
mysql> SELECT s.city AS 城市,s.county AS 区,s.sales_value AS 区销售额,
-> b.sales_value AS 市销售额,s.sales_value/b.sales_value AS 市比率,
-> a.sales_value AS 总销售额,s.sales_value/a.sales_value AS 总比率
-> FROM sales s
-> JOIN b ON (s.city=b.city) -- 连接市统计结果临时表
-> JOIN a           -- 连接总计金额临时表
-> ORDER BY s.city,s.county;
+-----+-----+-----+-----+-----+-----+-----+
| 城市 | 区   | 区销售额 | 市销售额 | 市比率 | 总销售额 | 总比率 |
+-----+-----+-----+-----+-----+-----+-----+
| 上海 | 长宁 |      10 |      40 | 0.2500 |      70 | 0.1429 |
| 上海 | 黄浦 |      30 |      40 | 0.7500 |      70 | 0.4286 |
| 北京 | 朝阳 |      20 |      30 | 0.6667 |      70 | 0.2857 |
| 北京 | 海淀 |      10 |      30 | 0.3333 |      70 | 0.1429 |
+-----+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

结果显示：市销售金额、市销售占比、总销售金额、总销售占比都计算出来了。

同样的查询，如果用窗口函数，就简单多了。我们可以用下面的代码来实现：

```
mysql> SELECT city AS 城市,county AS 区,sales_value AS 区销售额,
-> SUM(sales_value) OVER(PARTITION BY city) AS 市销售额, -- 计算市销售额
-> sales_value/SUM(sales_value) OVER(PARTITION BY city) AS 市比率,
-> SUM(sales_value) OVER() AS 总销售额,    -- 计算总销售额
-> sales_value/SUM(sales_value) OVER() AS 总比率
-> FROM sales
-> ORDER BY city,county;
+-----+-----+-----+-----+-----+-----+
| 城市 | 区   | 区销售额 | 市销售额 | 市比率 | 总销售额 | 总比率 |
+-----+-----+-----+-----+-----+-----+
| 上海 | 长宁 |      10 |      40 | 0.2500 |      70 | 0.1429 |
| 上海 | 黄浦 |      30 |      40 | 0.7500 |      70 | 0.4286 |
| 北京 | 朝阳 |      20 |      30 | 0.6667 |      70 | 0.2857 |
| 北京 | 海淀 |      10 |      30 | 0.3333 |      70 | 0.1429 |
+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

结果显示，我们得到了与上面那种查询同样的结果。

使用窗口函数，只用了一步就完成了查询。而且，由于没有用到临时表，执行的效率也更高了。很显然，**在这种需要用到分组统计的结果对每一条记录进行计算的场景下，使用窗口函数更好。**

MySQL从8.0版本开始支持窗口函数。窗口函数的作用类似于在查询中对数据进行分组，不同的是，分组操作会把分组的结果聚合成一条记录，而窗口函数是将结果置于每一条数据记录中。

窗口函数可以分为 **静态窗口函数** 和 **动态窗口函数**。

- 静态窗口函数的窗口大小是固定的，不会因为记录的不同而不同；
- 动态窗口函数的窗口大小会随着记录的不同而变化。

MySQL官方网站窗口函数的网址为 https://dev.mysql.com/doc/refman/8.0/en/window-function-descriptions.html#function_row-number。

窗口函数总体上可以分为序号函数、分布函数、前后函数、首尾函数和其他函数，如下表：

函数分类	函数	函数说明
序号函数	ROW_NUMBER()	顺序排序
	RANK()	并列排序，会跳过重复的序号，比如序号为1、1、3
	DENSE_RANK()	并列排序，不会跳过重复的序号，比如序号为1、1、2
分布函数	PERCENT_RANK()	等级值百分比
	CUME_DIST()	累积分布值
前后函数	LAG(expr, n)	返回当前行的前n行的expr的值
	LEAD(expr, n)	返回当前行的后n行的expr的值
首尾函数	FIRST_VALUE(expr)	返回第一个expr的值
	LAST_VALUE(expr)	返回最后一个expr的值
其他函数	NTH_VALUE(expr, n)	返回第n个expr的值
	NTILE(n)	将分区中的有序数据分为n个桶，记录桶编号

2.3 语法结构

窗口函数的语法结构是：

```
函数 OVER ([PARTITION BY 字段名 ORDER BY 字段名 ASC|DESC])
```

或者是：

```
函数 OVER 窗口名 ... WINDOW 窗口名 AS ([PARTITION BY 字段名 ORDER BY 字段名 ASC|DESC])
```

- OVER关键字指定函数窗口的范围。
 - 如果省略后面括号中的内容，则窗口会包含满足WHERE条件的所有记录，窗口函数会基于所有满足WHERE条件的记录进行计算。
 - 如果OVER关键字后面的括号不为空，则可以使用如下语法设置窗口。
- 窗口名：为窗口设置一个别名，用来标识窗口。
- PARTITION BY子句：指定窗口函数按照哪些字段进行分组。分组后，窗口函数可以在每个分组中分别执行。
- ORDER BY子句：指定窗口函数按照哪些字段进行排序。执行排序操作使窗口函数按照排序后的数据记录的顺序进行编号。
- FRAME子句：为分区中的某个子集定义规则，可以用来作为滑动窗口使用。

创建表：

```
CREATE TABLE goods(
    id INT PRIMARY KEY AUTO_INCREMENT,
    category_id INT,
    category VARCHAR(15),
    NAME VARCHAR(30),
    price DECIMAL(10,2),
    stock INT,
    upper_time DATETIME

);
```

添加数据：

```
INSERT INTO goods(category_id,category,NAME,price,stock,upper_time)
VALUES
(1, '女装/女士精品', 'T恤', 39.90, 1000, '2020-11-10 00:00:00'),
(1, '女装/女士精品', '连衣裙', 79.90, 2500, '2020-11-10 00:00:00'),
(1, '女装/女士精品', '卫衣', 89.90, 1500, '2020-11-10 00:00:00'),
(1, '女装/女士精品', '牛仔裤', 89.90, 3500, '2020-11-10 00:00:00'),
(1, '女装/女士精品', '百褶裙', 29.90, 500, '2020-11-10 00:00:00'),
(1, '女装/女士精品', '呢绒外套', 399.90, 1200, '2020-11-10 00:00:00'),
(2, '户外运动', '自行车', 399.90, 1000, '2020-11-10 00:00:00'),
(2, '户外运动', '山地自行车', 1399.90, 2500, '2020-11-10 00:00:00'),
(2, '户外运动', '登山杖', 59.90, 1500, '2020-11-10 00:00:00'),
(2, '户外运动', '骑行装备', 399.90, 3500, '2020-11-10 00:00:00'),
(2, '户外运动', '运动外套', 799.90, 500, '2020-11-10 00:00:00'),
(2, '户外运动', '滑板', 499.90, 1200, '2020-11-10 00:00:00');
```

下面针对goods表中的数据来验证每个窗口函数的功能。

1. 序号函数

1. ROW_NUMBER()函数

ROW_NUMBER()函数能够对数据中的序号进行顺序显示。

举例：查询 goods 数据表中每个商品分类下价格降序排列的各个商品信息。

```
mysql> SELECT ROW_NUMBER() OVER(PARTITION BY category_id ORDER BY price DESC) AS
row_num,
-> id, category_id, category, NAME, price, stock
-> FROM goods;
+-----+-----+-----+-----+-----+-----+
| row_num | id | category_id | category      | NAME       | price     | stock   |
+-----+-----+-----+-----+-----+-----+
|      1 |  6 |         1 | 女装/女士精品 | 呢绒外套 | 399.90  | 1200   |
|      2 |  3 |         1 | 女装/女士精品 | 卫衣      | 89.90   | 1500   |
|      3 |  4 |         1 | 女装/女士精品 | 牛仔裤   | 89.90   | 3500   |
|      4 |  2 |         1 | 女装/女士精品 | 连衣裙   | 79.90   | 2500   |
|      5 |  1 |         1 | 女装/女士精品 | T恤      | 39.90   | 1000   |
|      6 |  5 |         1 | 女装/女士精品 | 百褶裙   | 29.90   | 500    |
|      1 |  8 |         2 | 户外运动     | 山地自行车 | 1399.90 | 2500   |
|      2 | 11 |         2 | 户外运动     | 运动外套   | 799.90  | 500    |
|      3 | 12 |         2 | 户外运动     | 滑板      | 499.90  | 1200   |
```

```
+-----+-----+-----+-----+-----+
12 rows in set (0.00 sec)
```

举例：查询 goods 数据表中每个商品分类下价格最高的3种商品信息。

```
mysql> SELECT *
-> FROM (
->   SELECT ROW_NUMBER() OVER(PARTITION BY category_id ORDER BY price DESC) AS
row_num,
->     id, category_id, category, NAME, price, stock
->   FROM goods) t
-> WHERE row_num <= 3;
+-----+-----+-----+-----+-----+
| row_num | id | category_id | category      | NAME        | price      | stock |
+-----+-----+-----+-----+-----+
|      1 |  6 |         1 | 女装/女士精品 | 呢绒外套    | 399.90    | 1200 |
|      2 |  3 |         1 | 女装/女士精品 | 卫衣        | 89.90     | 1500 |
|      3 |  4 |         1 | 女装/女士精品 | 牛仔裤     | 89.90     | 3500 |
|      1 |  8 |         2 | 户外运动      | 山地自行车 | 1399.90   | 2500 |
|      2 | 11 |         2 | 户外运动      | 运动外套    | 799.90    | 500  |
|      3 | 12 |         2 | 户外运动      | 滑板        | 499.90    | 1200 |
+-----+-----+-----+-----+-----+
6 rows in set (0.00 sec)
```

在名称为“女装/女士精品”的商品类别中，有两款商品的价格为89.90元，分别是卫衣和牛仔裤。两款商品的序号都应该为2，而不是一个为2，另一个为3。此时，可以使用RANK()函数和DENSE_RANK()函数解决。

2. RANK()函数

使用RANK()函数能够对序号进行并列排序，并且会跳过重复的序号，比如序号为1、1、3。

举例：使用RANK()函数获取 goods 数据表中各类别的价格从高到低排序的各商品信息。

```
mysql> SELECT RANK() OVER(PARTITION BY category_id ORDER BY price DESC) AS row_num,
->   id, category_id, category, NAME, price, stock
->   FROM goods;
+-----+-----+-----+-----+-----+
| row_num | id | category_id | category      | NAME        | price      | stock |
+-----+-----+-----+-----+-----+
|      1 |  6 |         1 | 女装/女士精品 | 呢绒外套    | 399.90    | 1200 |
|      2 |  3 |         1 | 女装/女士精品 | 卫衣        | 89.90     | 1500 |
|      2 |  4 |         1 | 女装/女士精品 | 牛仔裤     | 89.90     | 3500 |
|      4 |  2 |         1 | 女装/女士精品 | 连衣裙     | 79.90     | 2500 |
|      5 |  1 |         1 | 女装/女士精品 | T恤        | 39.90     | 1000 |
|      6 |  5 |         1 | 女装/女士精品 | 百褶裙     | 29.90     | 500  |
|      1 |  8 |         2 | 户外运动      | 山地自行车 | 1399.90   | 2500 |
|      2 | 11 |         2 | 户外运动      | 运动外套    | 799.90    | 500  |
|      3 | 12 |         2 | 户外运动      | 滑板        | 499.90    | 1200 |
|      4 |  7 |         2 | 户外运动      | 自行车      | 399.90    | 1000 |
|      4 | 10 |         2 | 户外运动      | 骑行装备    | 399.90    | 3500 |
|      6 |  9 |         2 | 户外运动      | 登山杖      | 59.90     | 1500 |
+-----+-----+-----+-----+-----+
12 rows in set (0.00 sec)
```

举例：使用RANK()函数获取 goods 数据表中类别为“女装/女士精品”的价格最高的4款商品信息。

```

-> SELECT RANK() OVER(PARTITION BY category_id ORDER BY price DESC) AS row_num,
-> id, category_id, category, NAME, price, stock
-> FROM goods) t
-> WHERE category_id = 1 AND row_num <= 4;
+-----+-----+-----+-----+-----+-----+
| row_num | id | category_id | category      | NAME        | price     | stock |
+-----+-----+-----+-----+-----+-----+
|      1 | 6  |          1 | 女装/女士精品 | 呢绒外套    | 399.90   | 1200  |
|      2 | 3  |          1 | 女装/女士精品 | 卫衣        | 89.90    | 1500  |
|      2 | 4  |          1 | 女装/女士精品 | 牛仔裤     | 89.90    | 3500  |
|      4 | 2  |          1 | 女装/女士精品 | 连衣裙     | 79.90    | 2500  |
+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

```

可以看到，使用RANK()函数得出的序号为1、2、2、4，相同价格的商品序号相同，后面的商品序号是不连续的，跳过了重复的序号。

3. DENSE_RANK()函数

DENSE_RANK()函数对序号进行并列排序，并且不会跳过重复的序号，比如序号为1、1、2。

举例：使用DENSE_RANK()函数获取 goods 数据表中各类别的价格从高到低排序的各商品信息。

```

mysql> SELECT DENSE_RANK() OVER(PARTITION BY category_id ORDER BY price DESC) AS
row_num,
-> id, category_id, category, NAME, price, stock
-> FROM goods;
+-----+-----+-----+-----+-----+-----+
| row_num | id | category_id | category      | NAME        | price     | stock |
+-----+-----+-----+-----+-----+-----+
|      1 | 6  |          1 | 女装/女士精品 | 呢绒外套    | 399.90   | 1200  |
|      2 | 3  |          1 | 女装/女士精品 | 卫衣        | 89.90    | 1500  |
|      2 | 4  |          1 | 女装/女士精品 | 牛仔裤     | 89.90    | 3500  |
|      3 | 2  |          1 | 女装/女士精品 | 连衣裙     | 79.90    | 2500  |
|      4 | 1  |          1 | 女装/女士精品 | T恤        | 39.90    | 1000  |
|      5 | 5  |          1 | 女装/女士精品 | 百褶裙     | 29.90    | 500   |
|      1 | 8  |          2 | 户外运动      | 山地自行车 | 1399.90  | 2500  |
|      2 | 11 |          2 | 户外运动      | 运动外套    | 799.90   | 500   |
|      3 | 12 |          2 | 户外运动      | 滑板        | 499.90   | 1200  |
|      4 | 7  |          2 | 户外运动      | 自行车      | 399.90   | 1000  |
|      4 | 10 |          2 | 户外运动      | 骑行装备    | 399.90   | 3500  |
|      5 | 9  |          2 | 户外运动      | 登山杖      | 59.90    | 1500  |
+-----+-----+-----+-----+-----+-----+
12 rows in set (0.00 sec)

```

举例：使用DENSE_RANK()函数获取 goods 数据表中类别为“女装/女士精品”的价格最高的4款商品信息。

```

mysql> SELECT *
-> FROM(
->   SELECT DENSE_RANK() OVER(PARTITION BY category_id ORDER BY price DESC) AS
row_num,
->     id, category_id, category, NAME, price, stock
->   FROM goods) t
-> WHERE category_id = 1 AND row_num <= 3;
+-----+-----+-----+-----+-----+-----+
| row_num | id | category_id | category      | NAME        | price     | stock |
+-----+-----+-----+-----+-----+-----+

```

```

|      2 |   4 |           1 | 女装/女士精品 | 牛仔裤 | 89.90 | 3500 |
|      3 |   2 |           1 | 女装/女士精品 | 连衣裙 | 79.90 | 2500 |
+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

```

可以看到，使用DENSE_RANK()函数得出的行号为1、2、2、3，相同价格的商品序号相同，后面的商品序号是连续的，并且没有跳过重复的序号。

2. 分布函数

1. PERCENT_RANK()函数

PERCENT_RANK()函数是等级值百分比函数。按照如下方式进行计算。

```
(rank - 1) / (rows - 1)
```

其中，rank的值为使用RANK()函数产生的序号，rows的值为当前窗口的总记录数。

举例：计算 goods 数据表中名称为“女装/女士精品”的类别下的商品的PERCENT_RANK值。

```
#写法一:
SELECT RANK() OVER (PARTITION BY category_id ORDER BY price DESC) AS r,
PERCENT_RANK() OVER (PARTITION BY category_id ORDER BY price DESC) AS pr,
id, category_id, category, NAME, price, stock
FROM goods
WHERE category_id = 1;

#写法二:
mysql> SELECT RANK() OVER w AS r,
-> PERCENT_RANK() OVER w AS pr,
-> id, category_id, category, NAME, price, stock
-> FROM goods
-> WHERE category_id = 1 WINDOW w AS (PARTITION BY category_id ORDER BY price
DESC);
+---+---+---+-----+-----+-----+-----+-----+
| r | pr | id | category_id | category | NAME | price | stock |
+---+---+---+-----+-----+-----+-----+-----+
| 1 | 0 | 6 |           1 | 女装/女士精品 | 呢绒外套 | 399.90 | 1200 |
| 2 | 0.2 | 3 |           1 | 女装/女士精品 | 卫衣 | 89.90 | 1500 |
| 2 | 0.2 | 4 |           1 | 女装/女士精品 | 牛仔裤 | 89.90 | 3500 |
| 4 | 0.6 | 2 |           1 | 女装/女士精品 | 连衣裙 | 79.90 | 2500 |
| 5 | 0.8 | 1 |           1 | 女装/女士精品 | T恤 | 39.90 | 1000 |
| 6 | 1 | 5 |           1 | 女装/女士精品 | 百褶裙 | 29.90 | 500 |
+---+---+---+-----+-----+-----+-----+-----+
6 rows in set (0.00 sec)
```

2. CUME_DIST()函数

CUME_DIST()函数主要用于查询小于或等于某个值的比例。

举例：查询goods数据表中小于或等于当前价格的比例。

```
mysql> SELECT CUME_DIST() OVER(PARTITION BY category_id ORDER BY price ASC) AS cd,
-> id, category, NAME, price
-> FROM goods;
+-----+-----+-----+-----+-----+
| cd | id | category | NAME | price |
+-----+-----+-----+-----+-----+
```

```

|      0.5 | 2 | 女装/女士精品 | 连衣裙 | 79.90 |
| 0.8333333333333334 | 3 | 女装/女士精品 | 卫衣 | 89.90 |
| 0.8333333333333334 | 4 | 女装/女士精品 | 牛仔裤 | 89.90 |
|      1 | 6 | 女装/女士精品 | 呢绒外套 | 399.90 |
| 0.1666666666666666 | 9 | 户外运动 | 登山杖 | 59.90 |
|      0.5 | 7 | 户外运动 | 自行车 | 399.90 |
|      0.5 | 10 | 户外运动 | 骑行装备 | 399.90 |
| 0.6666666666666666 | 12 | 户外运动 | 滑板 | 499.90 |
| 0.8333333333333334 | 11 | 户外运动 | 运动外套 | 799.90 |
|      1 | 8 | 户外运动 | 山地自行车 | 1399.90 |
+-----+-----+-----+-----+
12 rows in set (0.00 sec)

```

3. 前后函数

1. LAG(expr,n)函数

LAG(expr,n)函数返回当前行的前n行的expr的值。

举例：查询goods数据表中前一个商品价格与当前商品价格的差值。

```

mysql> SELECT id, category, NAME, price, pre_price, price - pre_price AS diff_price
-> FROM (
->   SELECT id, category, NAME, price,LAG(price,1) OVER w AS pre_price
->   FROM goods
->   WINDOW w AS (PARTITION BY category_id ORDER BY price)) t;
+-----+-----+-----+-----+-----+
| id | category | NAME | price | pre_price | diff_price |
+-----+-----+-----+-----+-----+
| 5 | 女装/女士精品 | 百褶裙 | 29.90 | NULL | NULL |
| 1 | 女装/女士精品 | T恤 | 39.90 | 29.90 | 10.00 |
| 2 | 女装/女士精品 | 连衣裙 | 79.90 | 39.90 | 40.00 |
| 3 | 女装/女士精品 | 卫衣 | 89.90 | 79.90 | 10.00 |
| 4 | 女装/女士精品 | 牛仔裤 | 89.90 | 89.90 | 0.00 |
| 6 | 女装/女士精品 | 呢绒外套 | 399.90 | 89.90 | 310.00 |
| 9 | 户外运动 | 登山杖 | 59.90 | NULL | NULL |
| 7 | 户外运动 | 自行车 | 399.90 | 59.90 | 340.00 |
| 10 | 户外运动 | 骑行装备 | 399.90 | 399.90 | 0.00 |
| 12 | 户外运动 | 滑板 | 499.90 | 399.90 | 100.00 |
| 11 | 户外运动 | 运动外套 | 799.90 | 499.90 | 300.00 |
| 8 | 户外运动 | 山地自行车 | 1399.90 | 799.90 | 600.00 |
+-----+-----+-----+-----+
12 rows in set (0.00 sec)

```

2. LEAD(expr,n)函数

LEAD(expr,n)函数返回当前行的后n行的expr的值。

举例：查询goods数据表中后一个商品价格与当前商品价格的差值。

```

mysql> SELECT id, category, NAME, behind_price, price,behind_price - price AS
diff_price
-> FROM(
->   SELECT id, category, NAME, price,LEAD(price, 1) OVER w AS behind_price
->   FROM goods WINDOW w AS (PARTITION BY category_id ORDER BY price)) t;
+-----+-----+-----+-----+-----+
| id | category | NAME | behind_price | price | diff_price |
+-----+-----+-----+-----+-----+

```

```

| 1 | 女装/女士精品 | T恤 | 79.90 | 39.90 | 40.00 |
| 2 | 女装/女士精品 | 连衣裙 | 89.90 | 79.90 | 10.00 |
| 3 | 女装/女士精品 | 卫衣 | 89.90 | 89.90 | 0.00 |
| 4 | 女装/女士精品 | 牛仔裤 | 399.90 | 89.90 | 310.00 |
| 6 | 女装/女士精品 | 呢绒外套 | NULL | 399.90 | NULL |
| 9 | 户外运动 | 登山杖 | 399.90 | 59.90 | 340.00 |
| 7 | 户外运动 | 自行车 | 399.90 | 399.90 | 0.00 |
| 10 | 户外运动 | 骑行装备 | 499.90 | 399.90 | 100.00 |
| 12 | 户外运动 | 滑板 | 799.90 | 499.90 | 300.00 |
| 11 | 户外运动 | 运动外套 | 1399.90 | 799.90 | 600.00 |
| 8 | 户外运动 | 山地自行车 | NULL | 1399.90 | NULL |
+-----+-----+-----+-----+-----+
12 rows in set (0.00 sec)

```

4. 首尾函数

1. FIRST_VALUE(expr)函数

FIRST_VALUE(expr)函数返回第一个expr的值。

举例：按照价格排序，查询第1个商品的价格信息。

```

mysql> SELECT id, category, NAME, price, stock,FIRST_VALUE(price) OVER w AS
first_price
    -> FROM goods WINDOW w AS (PARTITION BY category_id ORDER BY price);
+-----+-----+-----+-----+-----+
| id | category | NAME | price | stock | first_price |
+-----+-----+-----+-----+-----+
| 5 | 女装/女士精品 | 百褶裙 | 29.90 | 500 | 29.90 |
| 1 | 女装/女士精品 | T恤 | 39.90 | 1000 | 29.90 |
| 2 | 女装/女士精品 | 连衣裙 | 79.90 | 2500 | 29.90 |
| 3 | 女装/女士精品 | 卫衣 | 89.90 | 1500 | 29.90 |
| 4 | 女装/女士精品 | 牛仔裤 | 89.90 | 3500 | 29.90 |
| 6 | 女装/女士精品 | 呢绒外套 | 399.90 | 1200 | 29.90 |
| 9 | 户外运动 | 登山杖 | 59.90 | 1500 | 59.90 |
| 7 | 户外运动 | 自行车 | 399.90 | 1000 | 59.90 |
| 10 | 户外运动 | 骑行装备 | 399.90 | 3500 | 59.90 |
| 12 | 户外运动 | 滑板 | 499.90 | 1200 | 59.90 |
| 11 | 户外运动 | 运动外套 | 799.90 | 500 | 59.90 |
| 8 | 户外运动 | 山地自行车 | 1399.90 | 2500 | 59.90 |
+-----+-----+-----+-----+
12 rows in set (0.00 sec)

```

2. LAST_VALUE(expr)函数

LAST_VALUE(expr)函数返回最后一个expr的值。

举例：按照价格排序，查询最后一个商品的价格信息。

```

mysql> SELECT id, category, NAME, price, stock,LAST_VALUE(price) OVER w AS last_price
    -> FROM goods WINDOW w AS (PARTITION BY category_id ORDER BY price);
+-----+-----+-----+-----+-----+
| id | category | NAME | price | stock | last_price |
+-----+-----+-----+-----+-----+
| 5 | 女装/女士精品 | 百褶裙 | 29.90 | 500 | 29.90 |
| 1 | 女装/女士精品 | T恤 | 39.90 | 1000 | 39.90 |
| 2 | 女装/女士精品 | 连衣裙 | 79.90 | 2500 | 79.90 |

```

```

| 6 | 女装/女士精品 | 呢绒外套 | 399.90 | 1200 | 399.90 |
| 9 | 户外运动 | 登山杖 | 59.90 | 1500 | 59.90 |
| 7 | 户外运动 | 自行车 | 399.90 | 1000 | 399.90 |
| 10 | 户外运动 | 骑行装备 | 399.90 | 3500 | 399.90 |
| 12 | 户外运动 | 滑板 | 499.90 | 1200 | 499.90 |
| 11 | 户外运动 | 运动外套 | 799.90 | 500 | 799.90 |
| 8 | 户外运动 | 山地自行车 | 1399.90 | 2500 | 1399.90 |
+-----+-----+-----+-----+
12 rows in set (0.00 sec)

```

5. 其他函数

1. NTH_VALUE(expr,n)函数

NTH_VALUE(expr,n)函数返回第n个expr的值。

举例：查询goods数据表中排名第2和第3的价格信息。

```

mysql> SELECT id, category, NAME, price,NTH_VALUE(price,2) OVER w AS second_price,
-> NTH_VALUE(price,3) OVER w AS third_price
-> FROM goods WINDOW w AS (PARTITION BY category_id ORDER BY price);
+-----+-----+-----+-----+
| id | category | NAME | price | second_price | third_price |
+-----+-----+-----+-----+
| 5 | 女装/女士精品 | 百褶裙 | 29.90 | NULL | NULL |
| 1 | 女装/女士精品 | T恤 | 39.90 | 39.90 | NULL |
| 2 | 女装/女士精品 | 连衣裙 | 79.90 | 39.90 | 79.90 |
| 3 | 女装/女士精品 | 卫衣 | 89.90 | 39.90 | 79.90 |
| 4 | 女装/女士精品 | 牛仔裤 | 89.90 | 39.90 | 79.90 |
| 6 | 女装/女士精品 | 呢绒外套 | 399.90 | 39.90 | 79.90 |
| 9 | 户外运动 | 登山杖 | 59.90 | NULL | NULL |
| 7 | 户外运动 | 自行车 | 399.90 | 399.90 | 399.90 |
| 10 | 户外运动 | 骑行装备 | 399.90 | 399.90 | 399.90 |
| 12 | 户外运动 | 滑板 | 499.90 | 399.90 | 399.90 |
| 11 | 户外运动 | 运动外套 | 799.90 | 399.90 | 399.90 |
| 8 | 户外运动 | 山地自行车 | 1399.90 | 399.90 | 399.90 |
+-----+-----+-----+-----+
12 rows in set (0.00 sec)

```

2. NTILE(n)函数

NTILE(n)函数将分区中的有序数据分为n个桶，记录桶编号。

举例：将goods表中的商品按照价格分为3组。

```

mysql> SELECT NTILE(3) OVER w AS nt,id, category, NAME, price
-> FROM goods WINDOW w AS (PARTITION BY category_id ORDER BY price);
+-----+-----+-----+-----+
| nt | id | category | NAME | price |
+-----+-----+-----+-----+
| 1 | 5 | 女装/女士精品 | 百褶裙 | 29.90 |
| 1 | 1 | 女装/女士精品 | T恤 | 39.90 |
| 2 | 2 | 女装/女士精品 | 连衣裙 | 79.90 |
| 2 | 3 | 女装/女士精品 | 卫衣 | 89.90 |
| 3 | 4 | 女装/女士精品 | 牛仔裤 | 89.90 |
| 3 | 6 | 女装/女士精品 | 呢绒外套 | 399.90 |
| 1 | 9 | 户外运动 | 登山杖 | 59.90 |

```

```
| 2 | 12 | 户外运动 | 滑板 | 499.90 |
| 3 | 11 | 户外运动 | 运动外套 | 799.90 |
| 3 | 8 | 户外运动 | 山地自行车 | 1399.90 |
+---+---+-----+-----+-----+
12 rows in set (0.00 sec)
```

2.5 小结

窗口函数的特点是可以分组，而且可以在分组内排序。另外，窗口函数不会因为分组而减少原表中的行数，这对我们在原表数据的基础上进行统计和排序非常有用。

3. 新特性2：公用表表达式

公用表表达式（或通用表表达式）简称为CTE（Common Table Expressions）。CTE是一个命名的临时结果集，作用范围是当前语句。CTE可以理解成一个可以复用的子查询，当然跟子查询还是有点区别的，CTE可以引用其他CTE，但子查询不能引用其他子查询。所以，可以考虑代替子查询。

依据语法结构和执行方式的不同，公用表表达式分为 **普通公用表表达式** 和 **递归公用表表达式** 2种。

3.1 普通公用表表达式

普通公用表表达式的语法结构是：

```
WITH CTE名称
AS (子查询)
SELECT|DELETE|UPDATE 语句；
```

普通公用表表达式类似于子查询，不过，跟子查询不同的是，它可以被多次引用，而且可以被其他的普通公用表表达式所引用。

举例：查询员工所在的部门的详细信息。

```
mysql> SELECT * FROM departments
-> WHERE department_id IN (
->           SELECT DISTINCT department_id
->             FROM employees
->           );
+-----+-----+-----+-----+
| department_id | department_name | manager_id | location_id |
+-----+-----+-----+-----+
|      10 | Administration |      200 |      1700 |
|      20 | Marketing |      201 |      1800 |
|      30 | Purchasing |      114 |      1700 |
|      40 | Human Resources |      203 |      2400 |
|      50 | Shipping |      121 |      1500 |
|      60 | IT |      103 |      1400 |
|      70 | Public Relations |      204 |      2700 |
|      80 | Sales |      145 |      2500 |
|      90 | Executive |      100 |      1700 |
|     100 | Finance |      108 |      1700 |
|     110 | Accounting |      205 |      1700 |
+-----+-----+-----+-----+
11 rows in set (0.00 sec)
```

```
mysql> WITH emp_dept_id
-> AS (SELECT DISTINCT department_id FROM employees)
-> SELECT *
-> FROM departments d JOIN emp_dept_id e
-> ON d.department_id = e.department_id;
+-----+-----+-----+-----+
| department_id | department_name | manager_id | location_id | department_id |
+-----+-----+-----+-----+
|      90 | Executive     |      100 |      1700 |      90 |
|      60 | IT             |      103 |      1400 |      60 |
|     100 | Finance        |      108 |      1700 |     100 |
|      30 | Purchasing    |      114 |      1700 |      30 |
|      50 | Shipping       |      121 |      1500 |      50 |
|      80 | Sales           |      145 |      2500 |      80 |
|      10 | Administration |      200 |      1700 |      10 |
|      20 | Marketing      |      201 |      1800 |      20 |
|      40 | Human Resources|      203 |      2400 |      40 |
|      70 | Public Relations|      204 |      2700 |      70 |
|     110 | Accounting    |      205 |      1700 |     110 |
+-----+-----+-----+-----+
11 rows in set (0.00 sec)
```

例子说明，公用表表达式可以起到子查询的作用。以后如果遇到需要使用子查询的场景，你可以在查询之前，先定义公用表表达式，然后在查询中用它来代替子查询。而且，跟子查询相比，公用表表达式有一个优点，就是定义过公用表表达式之后的查询，可以像一个表一样多次引用公用表表达式，而子查询则不能。

3.2 递归公用表表达式

递归公用表表达式也是一种公用表表达式，只不过，除了普通公用表表达式的特点以外，它还有自己的特点，就是**可以调用自己**。它的语法结构是：

```
WITH RECURSIVE
CTE名称 AS (子查询)
SELECT|DELETE|UPDATE 语句；
```

递归公用表表达式由 2 部分组成，分别是种子查询和递归查询，中间通过关键字 UNION [ALL]进行连接。这里的**种子查询，意思就是获得递归的初始值**。这个查询只会运行一次，以创建初始数据集，之后递归查询会一直执行，直到没有任何新的查询数据产生，递归返回。

案例：针对我们常用的employees表，包含employee_id, last_name和manager_id三个字段。如果a是b的管理者，那么，我们可以把b叫做a的下属，如果同时b又是c的管理者，那么c就是b的下属，是a的下下属。

下面我们尝试用查询语句列出所有具有下下属身份的人员信息。

如果用我们之前学过的知识来解决，会比较复杂，至少要进行 4 次查询才能搞定：

- 第一步，先找出初代管理者，就是不以任何别人为管理者的人，把结果存入临时表；
- 第二步，找出所有以初代管理者为管理者的人，得到一个下属集，把结果存入临时表；
- 第三步，找出所有以下属为管理者的人，得到一个下下属集，把结果存入临时表。
- 第四步，找出所有以下下属为管理者的人，得到一个结果集。

如果第四步的结果集为空，则计算结束，第三步的结果集就是我们需要的下下属集了，否则就必须继续进行第四步，一直到结果集为空为止。比如上面的这个数据表，就需要到第五步，才能得到空结果集。

- 用递归公用表表达式中的种子查询，找出第一代管理者。子枝 n 表示代次，初始值为 1，表示是第一代管理者。
- 用递归公用表表达式中的递归查询，查出以这个递归公用表表达式中的人为管理者的人，并且代次的值加 1。直到没有人以这个递归公用表表达式中的人为管理者了，递归返回。
- 在最后的查询中，选出所有代次大于等于 3 的人，他们肯定是第三代及以上代次的下属了，也就是下属了。这样就得到了我们需要的结果集。

这里看似也是 3 步，实际上是一个查询的 3 个部分，只需要执行一次就可以了。而且也不需要用临时表保存中间结果，比刚刚的方法简单多了。

代码实现：

```
WITH RECURSIVE cte
AS
(
SELECT employee_id, last_name, manager_id, 1 AS n FROM employees WHERE employee_id = 100
-- 种子查询，找到第一代领导
UNION ALL
SELECT a.employee_id, a.last_name, a.manager_id, n+1 FROM employees AS a JOIN cte
ON (a.manager_id = cte.employee_id) -- 递归查询，找出以递归公用表表达式的人为领导的人
)
SELECT employee_id, last_name FROM cte WHERE n >= 3;
```

总之，递归公用表表达式对于查询一个有共同的根节点的树形结构数据，非常有用。它可以不受层级的限制，轻松查出所有节点的数据。如果用其他的查询方式，就比较复杂了。

3.3 小结

公用表表达式的作用是可以替代子查询，而且可以被多次引用。递归公用表表达式对查询有一个共同根节点的树形结构数据非常高效，可以轻松搞定其他查询方式难以处理的查询。

第01章_Linux下MySQL的安装与使用

讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>



1. 安装前说明

1.1 Linux系统及工具的准备

- 安装并启动好两台虚拟机：[CentOS 7](#)
 - 掌握克隆虚拟机的操作
 - mac地址
 - 主机名
 - ip地址
 - UUID
- 安装有[Xshell](#) 和 [Xftp](#) 等访问CentOS系统的工具
- CentOS6和CentOS7在MySQL的使用中的区别

1. 防火墙：6是iptables，7是firewalld
2. 启动服务的命令：6是service，7是systemctl

1.2 查看是否安装过MySQL

- 如果你是用rpm安装，检查一下RPM PACKAGE：

```
rpm -qa | grep -i mysql # -i 忽略大小写
```

- 检查mysql service：

```
systemctl status mysqld.service
```

- 如果存在mysql-libs的旧版本包，显示如下：

```
[root@atguigu01 opt]# rpm -qa | grep -i mysql
mysql-community-server-8.0.25-1.el7.x86_64
mysql-community-client-plugins-8.0.25-1.el7.x86_64
mysql-community-libs-8.0.25-1.el7.x86_64
mysql-community-client-8.0.25-1.el7.x86_64
mysql-community-common-8.0.25-1.el7.x86_64
[root@atguigu01 opt]#
```

- 如果不存在mysql-lib的版本，显示如下：

```
[root@atguigu02 opt]# rpm -qa | grep -i mysql
[root@atguigu02 opt]#
```

1.3 MySQL的卸载

1. 关闭 mysql 服务

```
systemctl stop mysqld.service
```

2. 查看当前 mysql 安装状况

```
rpm -qa | grep -i mysql
```

或

```
yum list installed | grep mysql
```

3. 卸载上述命令查询出的已安装程序

```
yum remove mysql-xxx mysql-xxx mysql-xxx mysqk-xxxx
```

务必卸载干净，反复执行 `rpm -qa | grep -i mysql` 确认是否有卸载残留

4. 删除 mysql 相关文件

- 查找相关文件

```
find / -name mysql
```

- 删除上述命令查找出的相关文件

```
rm -rf xxx
```

5.删除 my.cnf

```
rm -rf /etc/my.cnf
```

2. MySQL的Linux版安装

2.1 MySQL的4大版本

- **MySQL Community Server 社区版**，开源免费，自由下载，但不提供官方技术支持，适用于大多数普通用户。
- **MySQL Enterprise Edition 企业版**，需付费，不能在线下载，可以试用30天。提供了更多的功能和更完备的技术支持，更适合于对数据库的功能和可靠性要求较高的企业客户。
- **MySQL Cluster 集群版**，开源免费。用于架设集群服务器，可将几个MySQL Server封装成一个Server。需要在社区版或企业版的基础上使用。
- **MySQL Cluster CGE 高级集群版**，需付费。

- 截止目前，官方最新版本为 **8.0.27**。此前，8.0.0 在 2016.9.12日就发布了。
- 本课程中主要使用 **8.0.25版本**。同时为了更好的说明MySQL8.0新特性，还会安装 **MySQL5.7** 版本，作为对比。

此外，官方还提供了 **MySQL Workbench** (GUITOOL) 一款专为MySQL设计的 **ER/数据库建模工具**。它是著名的数据库设计工具DBDesigner4的继任者。MySQLWorkbench又分为两个版本，分别是 **社区版** (MySQL Workbench OSS) 、 **商用版** (MySQL WorkbenchSE) 。

2.2 下载MySQL指定版本

1. 下载地址

官网：<https://www.mysql.com>

2. 打开官网，点击DOWNLOADS

然后，点击 **MySQL Community (GPL) Downloads**

The screenshot shows the MySQL Downloads page. At the top, there's a banner for "MySQL Database Service with HeatWave for Real-time Analytics". Below the banner, there are two main sections: "Faster Performance" and "Lower Total Cost of Ownership". The "Faster Performance" section lists: "400x MySQL query acceleration", "1100x Faster than Amazon Aurora", and "2.7x Faster than Amazon Redshift". The "Lower Total Cost of Ownership" section lists: "1/3 the Cost of Amazon RDS", "1/3 the Cost of Amazon Redshift", and "Easy migration from Amazon RDS". Below these sections, there are two columns of links. The left column includes "Free Webinars" (HeatWave in Depth: Machine Learning-based Automation - MySQL AutoPilot, Wednesday, August 18, 2021; Introduction to MySQL Operator for Kubernetes, Thursday, August 19, 2021; MySQL Virtual Hands-on Lab Sessions | ASEAN - Lab 1: Run Your Analytics with Astonishing Performances Leveraging HeatWave, Thursday, August 19, 2021; More), "Contact Sales" (USA: +1-866-221-0634; Canada: +1-866-221-0634), and "MySQL Enterprise Edition" (Learn More, Customer Download, Trial Download). The right column includes "MySQL Cluster CGE" (Learn More, Customer Download, Trial Download), and a prominent red-bordered button labeled "MySQL Community (GPL) Downloads".

3. 点击 MySQL Community Server

MySQL :: MySQL Community x +
← → C dev.mysql.com/downloads/

① MySQL Community Downloads

- MySQL Yum Repository
- MySQL APT Repository
- MySQL SUSE Repository
- MySQL Community Server MySQL Community Server
- MySQL Cluster
- MySQL Router
- MySQL Shell
- MySQL Workbench
- MySQL Installer for Windows
- MySQL for Visual Studio
- C API (libmysqlclient)
- Connector/C++
- Connector/J
- Connector/.NET
- Connector/Node.js
- Connector/ODBC
- Connector/Python
- MySQL Native Driver for PHP
- MySQL Benchmark Tool
- Time zone description tables
- Download Archives

4. 在General Availability(GA) Releases中选择适合的版本

- 如果安装Windows系统下MySQL，推荐下载 MSI安装程序；点击 [Go to Download Page](#) 进行下载即可

General Availability (GA) Releases Archives [i](#)

MySQL Community Server 8.0.26

Select Operating System: Microsoft Windows ▾

Looking for previous GA versions?

Recommended Download:

MySQL Installer for Windows
All MySQL Products. For All Windows Platforms. In One Package.

Starting with MySQL 5.6 the MySQL Installer package replaces the standalone MSI packages.

Windows (x86, 32 & 64-bit), MySQL Installer MSI [Go to Download Page >](#)

- Windows下的MySQL安装有两种安装程序
 - [mysql-installer-web-community-8.0.25.0.msi](#) 下载程序大小：2.4M；安装时需要联网安装组件。
 - [mysql-installer-community-8.0.25.0.msi](#) 下载程序大小：435.7M；安装时离线安装即可。**推荐**。

5. Linux系统下安装MySQL的几种方式

5.1 Linux系统下安装软件的常用三种方式：

方式1：rpm命令

使用rpm命令安装扩展名为".rpm"的软件包。

.rpm包的一般格式：



方式2：yum命令

需联网，从 [互联网获取](#) 的yum源，直接使用yum命令安装。

方式3：编译安装源码包

针对 [tar.gz](#) 这样的压缩格式，要用tar命令来解压；如果是其它压缩格式，就使用其它命令。

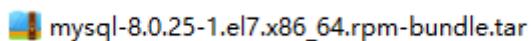
5.2 Linux系统下安装MySQL，官方给出多种安装方式

安装方式	特点
rpm	安装简单，灵活性差，无法灵活选择版本、升级
rpm repository	安装包极小，版本安装简单灵活，升级方便，需要联网安装
通用二进制包	安装比较复杂，灵活性高，平台通用性好
源码包	安装最复杂，时间长，参数设置灵活，性能好

- 这里不能直接选择CentOS 7系统的版本，所以选择与之对应的 [Red Hat Enterprise Linux](#)
- <https://downloads.mysql.com/archives/community/> 直接点Download下载RPM Bundle全量包。包括了所有下面的组件。不需要一个一个下载了。

Product Version:	Operating System:	Date	Size	Action
8.0.25	Red Hat Enterprise Linux / Oracle Linux	Apr 26, 2021	730.2M	Download
	All			
Red Hat Enterprise Linux 8 / Oracle Linux 8 (x86, 64-bit), RPM Bundle (mysql-8.0.25-1.el8.x86_64.rpm-bundle.tar)				
Red Hat Enterprise Linux 8 / Oracle Linux 8 (ARM, 64-bit), RPM Bundle (mysql-8.0.25-1.el8.aarch64.rpm-bundle.tar)			741.1M	Download
Red Hat Enterprise Linux 8 / Oracle Linux 8 (x86, 64-bit), RPM Package MySQL Server (mysql-community-server-8.0.25-1.el8.x86_64.rpm)			52.9M	Download
Red Hat Enterprise Linux 8 / Oracle Linux 8 (ARM, 64-bit), RPM Package MySQL Server (mysql-community-server-8.0.25-1.el8.aarch64.rpm)			60.2M	Download
Red Hat Enterprise Linux 8 / Oracle Linux 8 (x86, 64-bit), RPM Package Client Utilities (mysql-community-client-8.0.25-1.el8.x86_64.rpm)			13.4M	Download
Red Hat Enterprise Linux 8 / Oracle Linux 8 (ARM, 64-bit), RPM Package Client Utilities (mysql-community-client-8.0.25-1.el8.aarch64.rpm)			14.9M	Download

6. 下载的tar包，用压缩工具打开



- 解压后rpm安装包（红框为抽取出来的安装包）

名称	修改日期
mysql-community-client-8.0.25-1.el7.x86_64.rpm	2021/4/26 15:36
mysql-community-client-plugins-8.0.25-1.el7.x86_64.rpm	2021/4/26 15:36
mysql-community-common-8.0.25-1.el7.x86_64.rpm	2021/4/26 15:36
mysql-community-devel-8.0.25-1.el7.x86_64.rpm	2021/4/26 15:36
mysql-community-embedded-compat-8.0.25-1.el7.x86_64.rpm	2021/4/26 15:37
mysql-community-libs-8.0.25-1.el7.x86_64.rpm	2021/4/26 15:37
mysql-community-libs-compat-8.0.25-1.el7.x86_64.rpm	2021/4/26 15:37
mysql-community-server-8.0.25-1.el7.x86_64.rpm	2021/4/26 15:38
mysql-community-test-8.0.25-1.el7.x86_64.rpm	2021/4/26 15:40

2.3 CentOS7下检查MySQL依赖

1. 检查/tmp临时目录权限 (必不可少)

由于mysql安装过程中，会通过mysql用户在/tmp目录下新建tmp_db文件，所以请给/tmp较大的权限。执行：

```
chmod -R 777 /tmp
```

```
dr-xr-xr-x. 13 root root 0 12月 31 16:57 sys
drwxrwxrwt. 35 root root 4096 12月 31 17:26 tmp
drwxr-xr-x. 13 root root 155 12月 6 15:39 usr
drwxr-xr-x. 21 root root 4096 12月 6 15:52 var
```

2. 安装前，检查依赖

```
rpm -qa|grep libaio
```

- 如果存在libaio包如下：

```
[root@atguigu01 ~]# rpm -qa|grep libaio
libaio-0.3.109-13.el7.x86_64
```

```
rpm -qa|grep net-tools
```

- 如果存在net-tools包如下：

```
[root@atguigu01 ~]# rpm -qa|grep net-tools
net-tools-2.0-0.22.20131004git.el7.x86_64
```

```
rpm -qa|grep net-tools
```

- 如果不存在需要到centos安装盘里进行rpm安装。安装linux如果带图形化界面，这些都是安装好的。

2.4 CentOS7下MySQL安装过程

1. 将安装程序拷贝到/opt目录下

在mysql的安装文件目录下执行：（必须按照顺序执行）

```
rpm -ivh mysql-community-common-8.0.25-1.el7.x86_64.rpm  
rpm -ivh mysql-community-client-plugins-8.0.25-1.el7.x86_64.rpm  
rpm -ivh mysql-community-libs-8.0.25-1.el7.x86_64.rpm  
rpm -ivh mysql-community-client-8.0.25-1.el7.x86_64.rpm  
rpm -ivh mysql-community-server-8.0.25-1.el7.x86_64.rpm
```

- 注意：如在检查工作时，没有检查mysql依赖环境在安装mysql-community-server会报错
- rpm 是Redhat Package Manager缩写，通过RPM的管理，用户可以把源代码包装成以rpm为扩展名的文件形式，易于安装。
- i ,--install 安装软件包
- v ,--verbose 提供更多的详细信息输出
- h ,--hash 软件包安装的时候列出哈希标记(和-v一起使用效果更好)，展示进度条

```
[root@atguigu01 opt]# ll  
总用量 1237756  
-rw-r--r--. 1 root root 8924465 7月 26 23:36 apache-tomcat-7.0.70.tar.gz  
-rw-r--r--. 1 root root 567238341 7月 26 23:36 ideaIC-2019.2.4-no-jbr.tar.gz  
-rw-r--r--. 1 root root 189784266 7月 26 23:36 jdk-8u152-linux-x64.tar.gz  
-rw-r--r--. 1 root root 47810444 7月 27 22:59 mysql-community-client-8.0.25-1.el7.x86_64.rpm  
-rw-r--r--. 1 root root 193616 7月 27 23:02 mysql-community-client-plugins-8.0.25-1.el7.x86_64.rpm  
-rw-r--r--. 1 root root 628904 7月 27 22:59 mysql-community-common-8.0.25-1.el7.x86_64.rpm  
-rw-r--r--. 1 root root 4240320 7月 27 22:59 mysql-community-libs-8.0.25-1.el7.x86_64.rpm  
-rw-r--r--. 1 root root 448614076 7月 27 22:59 mysql-community-server-8.0.25-1.el7.x86_64.rpm  
drwxr-xr-x. 2 root root 4096 9月 7 2017 lib
```

2. 安装过程截图

```
[root@atguigu01 opt]# rpm -ivh mysql-community-client-plugins-8.0.25-1.el7.x86_64.rpm  
警告: mysql-community-client-plugins-8.0.25-1.el7.x86_64.rpm: 头V3 DSA/SHA1 Signature, 密钥 ID 5072e1f5: NOKEY  
准备中... ##### [100%]  
正在升级/安装...  
1:mysql-community-client-plugins-8.##### [100%]  
[root@atguigu01 opt]# rpm -ivh mysql-community-libs-8.0.25-1.el7.x86_64.rpm  
警告: mysql-community-libs-8.0.25-1.el7.x86_64.rpm: 头V3 DSA/SHA1 Signature, 密钥 ID 5072e1f5: NOKEY  
准备中... ##### [100%]  
正在升级/安装...  
1:mysql-community-libs-8.0.25-1.el7##### [100%]  
[root@atguigu01 opt]# rpm -ivh mysql-community-client-8.0.25-1.el7.x86_64.rpm  
警告: mysql-community-client-8.0.25-1.el7.x86_64.rpm: 头V3 DSA/SHA1 Signature, 密钥 ID 5072e1f5: NOKEY  
准备中... ##### [100%]  
正在升级/安装...  
1:mysql-community-client-8.0.25-1.e##### [100%]  
[root@atguigu01 opt]# rpm -ivh mysql-community-server-8.0.25-1.el7.x86_64.rpm  
警告: mysql-community-server-8.0.25-1.el7.x86_64.rpm: 头V3 DSA/SHA1 Signature, 密钥 ID 5072e1f5: NOKEY  
准备中... ##### [100%]  
正在升级/安装...  
1:mysql-community-server-8.0.25-1.e^C##### [100%]
```

安装过程中可能的报错信息：

```
[root@localhost opt]# rpm -ivh mysql-community-libs-8.0.25-1.el7.x86_64.rpm  
警告: mysql-community-libs-8.0.25-1.el7.x86_64.rpm: 头V3 DSA/SHA1 Signature, 密钥 ID 5072e1f5: NOKEY  
错误: 依赖检测失败:  
    mariadb-libs 被 mysql-community-libs-8.0.25-1.el7.x86_64 取代  
[root@localhost opt]# rpm -ivh mysql-community-libs-8.0.25-1.el7.x86_64.rpm  
警告: mysql-community-libs-8.0.25-1.el7.x86_64.rpm: 头V3 DSA/SHA1 Signature, 密钥 ID 5072e1f5: NOKEY
```

一个命令：**yum remove mysql-libs** 解决，清除之前安装过的依赖即可

3. 查看MySQL版本

执行如下命令，如果成功表示安装mysql成功。类似java -version如果打出版本等信息

```
mysql --version  
#或  
mysqladmin --version
```

```
[root@atguigu01 opt]# mysqladmin --version  
mysqladmin Ver 8.0.25 for Linux on x86_64 (MySQL Community Server - GPL)
```

执行如下命令，查看是否安装成功。需要增加-i 不用去区分大小写，否则搜索不到。

```
rpm -qa|grep -i mysql
```

```
[root@atguigu01 opt]# rpm -qa|grep -i mysql  
mysql-community-server-8.0.25-1.el7.x86_64  
mysql-community-client-plugins-8.0.25-1.el7.x86_64  
mysql-community-libs-8.0.25-1.el7.x86_64  
mysql-community-client-8.0.25-1.el7.x86_64  
mysql-community-common-8.0.25-1.el7.x86_64
```

4. 服务的初始化

为了保证数据库目录与文件的所有者为 mysql 登录用户，如果你是以 root 身份运行 mysql 服务，需要执行下面的命令初始化：

```
mysqld --initialize --user=mysql
```

说明：--initialize 选项默认以“安全”模式来初始化，则会为 root 用户生成一个密码并将 该密码标记为过期，登录后你需要设置一个新的密码。生成的 临时密码 会往日志中记录一份。

查看密码：

```
cat /var/log/mysqld.log
```

```
[root@atguigu01 log]# cat /var/log/mysqld.log  
2021-07-27T16:27:30.018766Z 0 [System] [MY-013169] [Server] /usr/sbin/mysqld (mysqld 8.0.25) initializing of server in progress as process 4532  
2021-07-27T16:27:30.027795Z 1 [System] [MY-013576] [InnoDB] InnoDB initialization has started.  
2021-07-27T16:27:30.245036Z 1 [System] [MY-013577] [InnoDB] InnoDB initialization has ended.  
2021-07-27T16:27:31.054680Z 6 [Note] [MY-010454] [Server] A temporary password is generated for root@localhost: !E/uZ_o.2sa  
2021-07-27T16:27:33.490420Z 0 [System] [MY-010116] [Server] /usr/sbin/mysqld (mysqld 8.0.25) starting as process 4577  
2021-07-27T16:27:33.500790Z 1 [System] [MY-013576] [InnoDB] InnoDB initialization has started.  
2021-07-27T16:27:33.656675Z 1 [System] [MY-013577] [InnoDB] InnoDB initialization has ended.  
2021-07-27T16:27:33.750754Z 0 [System] [MY-011323] [Server] X Plugin ready for connections. Bind-address: '::' port: 33060, socket: /var/run/mysqld/mys  
2021-07-27T16:27:33.902560Z 0 [Warning] [MY-010068] [Server] CA certificate ca.pem is self signed.  
2021-07-27T16:27:33.902731Z 0 [System] [MY-013602] [Server] Channel mysql_main configured to support TLS. Encrypted connections are now supported for t  
2021-07-27T16:27:33.921842Z 0 [System] [MY-010931] [Server] /usr/sbin/mysqld: ready for connections. Version: '8.0.25' socket: '/var/lib/mysql/mysql.s  
2021-07-27T16:27:56.073500Z 0 [System] [MY-012171] [General] Received CHUTDOWN from master: 0 (Version: 0.0.25)
```

root@localhost: 后面就是初始化的密码

5. 启动MySQL，查看状态

```
#加不加.service后缀都可以
```

```
启动: systemctl start mysqld.service
```

```
关闭: systemctl stop mysqld.service
```

```
重启: systemctl restart mysqld.service
```

```
查看状态: systemctl status mysqld.service
```

`mysqld` 这个可执行文件就代表着 MySQL 服务器程序，运行这个可执行文件就可以直接启动一个服务器进程。

```
[root@atguigu01 log]# systemctl start mysqld.service
[root@atguigu01 log]# systemctl status mysqld.service
● mysqld.service - MySQL Server
   Loaded: loaded (/usr/lib/systemd/system/mysqld.service; disabled; vendor preset: disabled)
   Active: active (running) since 三 2021-07-28 00:27:33 CST; 12s ago
     Docs: man:mysqld(8)
           http://dev.mysql.com/doc/refman/en/using-systemd.html
  Process: 4495 ExecStartPre=/usr/bin/mysql_pre_systemd (code=exited, status=0/SUCCESS)
 Main PID: 4577 (mysqld)
   Status: "Server is operational"
    Tasks: 38
   CGroup: /system.slice/mysqld.service
           └─4577 /usr/sbin/mysqld

7月 28 00:27:22 atguigu01 systemd[1]: Starting MySQL Server...
7月 28 00:27:33 atguigu01 systemd[1]: Started MySQL Server.
```

查看进程：

```
ps -ef | grep -i mysql
```

```
[root@atguigu01 log]# ps -ef | grep -i mysql
mysql      4674      1  0 00:28 ?        00:00:00 /usr/sbin/mysqld
root       4729    3193  0 00:29 pts/1    00:00:00 grep --color=auto -i mysql
```

6. 查看MySQL服务是否自启动

```
systemctl list-unit-files|grep mysqld.service
```

```
[root@atguigu01 opt]# systemctl list-unit-files|grep mysqld.service
mysqld.service                                enabled
```

默认是enabled。

- 如不是enabled可以运行如下命令设置自启动

```
systemctl enable mysqld.service
```

```
[root@atguigu01 log]# systemctl enable mysqld.service
Created symlink from /etc/systemd/system/multi-user.target.wants/mysqld.service to /usr/lib/systemd/system/mysqld.service.
[root@atguigu01 log]# systemctl list-unit-files|grep mysqld.service
mysqld.service                                [enabled]
```

- 如果希望不进行自启动，运行如下命令设置

```
systemctl disable mysqld.service
```

```
[root@atguigu01 log]# systemctl disable mysqld.service
Removed symlink /etc/systemd/system/multi-user.target.wants/mysqld.service.
[root@atguigu01 log]# systemctl list-unit-files|grep mysqld.service
mysqld.service                                [disabled]
```

3. MySQL登录

3.1 首次登录

通过 `mysql -hlocalhost -P3306 -uroot -p` 进行登录，在Enter password: 录入初始化密码

```
[root@atguigu01 init.d]# mysql -hlocalhost -P3306 -uroot -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 8
Server version: 8.0.25 MySQL Community Server - GPL

Copyright (c) 2000, 2021, Oracle and/or its affiliates.

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

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

3.2 修改密码

- 因为初始化密码默认是过期的，所以查看数据库会报错
- 修改密码：

```
ALTER USER 'root'@'localhost' IDENTIFIED BY 'new_password';
```

- 5.7版本之后（不含5.7），mysql加入了全新的密码安全机制。设置新密码太简单会报错。

```
mysql> ALTER USER 'root'@'localhost' IDENTIFIED BY 'HelloWorld';
ERROR 1819 (HY000): Your password does not satisfy the current policy requirements
mysql> ALTER USER 'root'@'localhost' IDENTIFIED BY 'HelloWorld123';
ERROR 1819 (HY000): Your password does not satisfy the current policy requirements
mysql> ALTER USER 'root'@'localhost' IDENTIFIED BY 'Hello_World';
ERROR 1819 (HY000): Your password does not satisfy the current policy requirements
```

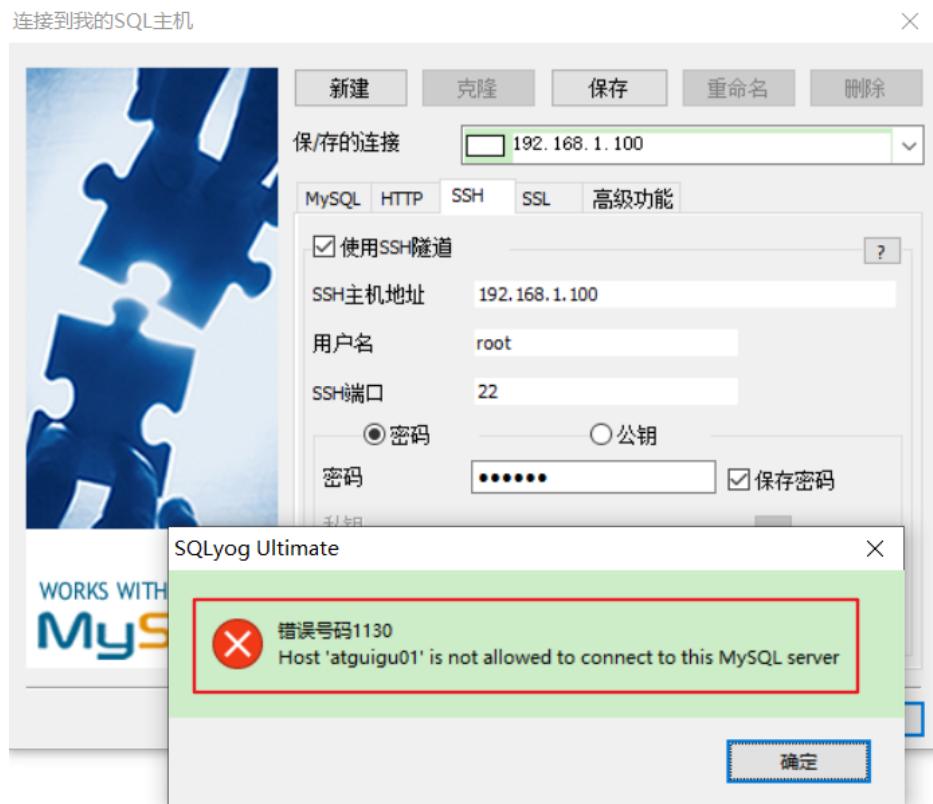
- 改为更复杂的密码规则之后，设置成功，可以正常使用数据库了

```
mysql> ALTER USER 'root'@'localhost' IDENTIFIED BY 'Hello_World123';
Query OK, 0 rows affected (0.01 sec)
```

3.3 设置远程登录

1. 当前问题

在用SQLyog或Navicat中配置远程连接Mysql数据库时遇到如下报错信息，这是由于Mysql配置了不支持远程连接引起的。



2. 确认网络

1. 在远程机器上使用 ping ip 地址 保证网络畅通

2. 在远程机器上使用 telnet 命令 保证端口号开放 访问

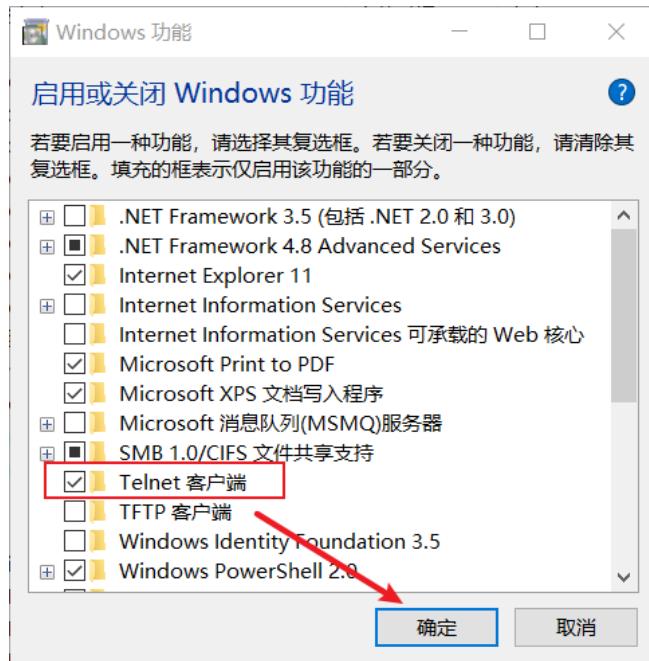
telnet ip 地址 端口号

拓展： telnet 命令开启：

The screenshot shows the Windows Control Panel under the "所有控制面板项" (All Control Panel Items) view. The "查看方式" (View mode) is set to "小图标" (Small icons). The "程序和功能" (Programs and Features) link is highlighted with a red box.

Below the Control Panel, the "控制面板主页" (Control Panel Home) and "卸载或更改程序" (Uninstall or Change Program) sections are visible. A red arrow points from the "Programs and Features" link in the Control Panel to the "卸载或更改程序" section below.

Text in the "卸载或更改程序" section reads: "若要卸载程序, 请从列表中将其选中, 然后单击“卸载”、“更改”或“修复”" (To uninstall a program, select it from the list and click "Uninstall", "Change", or "Repair").



3. 关闭防火墙或开放端口

方式一：关闭防火墙

- CentOS6：

```
service iptables stop
```

- CentOS7：

```
systemctl start firewalld.service  
  
systemctl status firewalld.service  
  
systemctl stop firewalld.service  
  
#设置开机启用防火墙  
systemctl enable firewalld.service  
  
#设置开机禁用防火墙  
systemctl disable firewalld.service
```

方式二：开放端口

- 查看开放的端口号

```
firewall-cmd --list-all
```

- 设置开放的端口号

```
firewall-cmd --add-service=http --permanent  
  
firewall-cmd --add-port=3306/tcp --permanent
```

- 重启防火墙

```
firewall-cmd --reload
```

4. Linux下修改配置

在Linux系统MySQL下测试：

```
use mysql;  
  
select Host,User from user;
```

```
mysql> use mysql;  
Database changed  
mysql> select Host,User from user;  
+-----+-----+  
| Host | User |  
+-----+-----+  
| localhost | mysql.session |  
| localhost | mysql.sys |  
| localhost | root |  
+-----+-----+  
3 rows in set (0.00 sec)
```

可以看到root用户的当前主机配置信息为localhost。

- **修改Host为通配符%**

Host列指定了允许用户登录所使用的IP，比如user=root Host=192.168.1.1。这里的意思就是说root用户只能通过192.168.1.1的客户端去访问。user=root Host=localhost，表示只能通过本机客户端去访问。而%是个**通配符**，如果Host=192.168.1.%，那么就表示只要是IP地址前缀为“192.168.1.”的客户端都可以连接。如果Host=%，表示所有IP都有连接权限。

注意：在生产环境下不能为了省事将host设置为%，这样做会存在安全问题，具体的设置可以根据生产环境的IP进行设置。

```
update user set host = '%' where user = 'root';
```

Host设置了“%”后便可以允许远程访问。

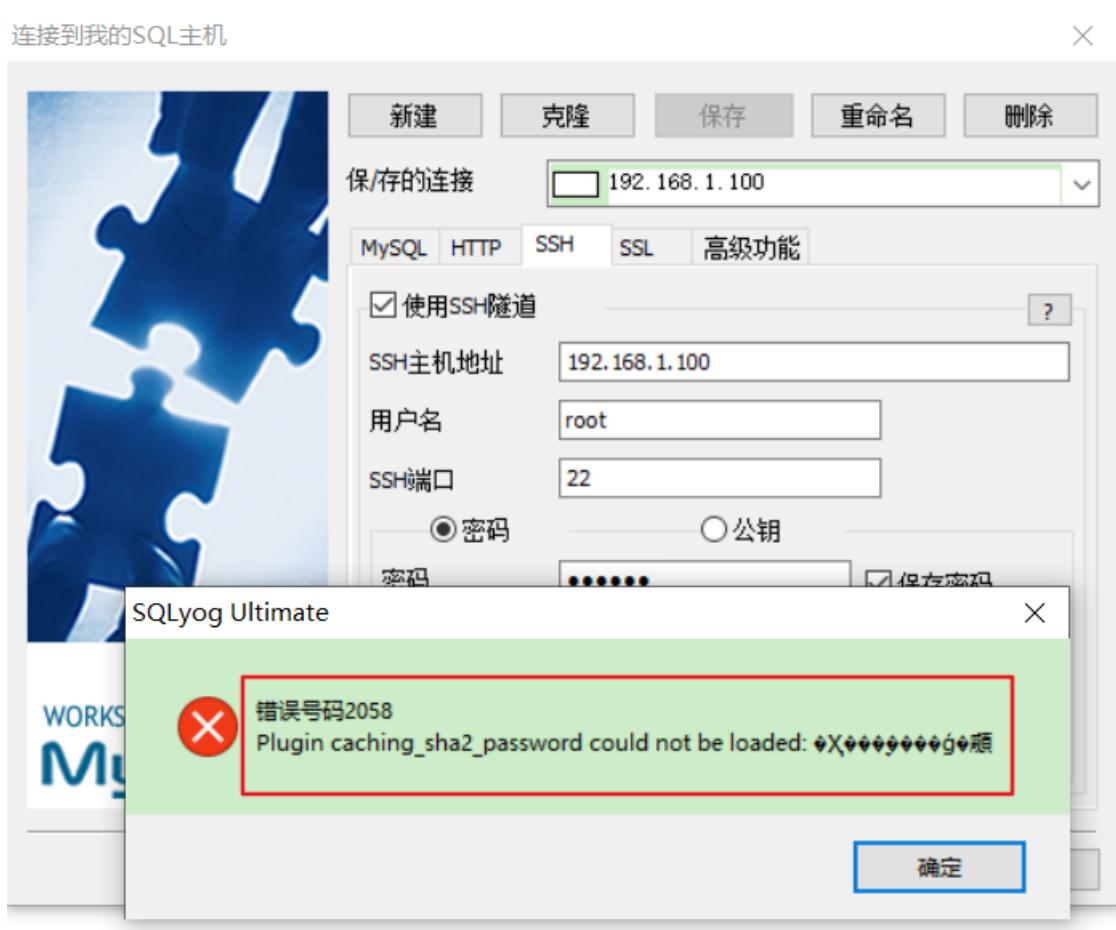
```
mysql> update user set Host='%' where User='root';  
Query OK, 1 row affected (0.00 sec)  
Rows matched: 1  Changed: 1  Warnings: 0  
  
mysql> select Host,User from user;  
+-----+-----+  
| Host | User |  
+-----+-----+  
| % | root |  
| localhost | mysql.session |  
| localhost | mysql.sys |  
+-----+-----+  
3 rows in set (0.00 sec)
```

Host修改完成后记得执行flush privileges使配置立即生效：

```
flush privileges;
```

5. 测试

- 如果是 MySQL5.7 版本，接下来就可以使用SQLyog或者Navicat成功连接至MySQL了。
- 如果是 MySQL8 版本，连接时还会出现如下问题：



配置新连接报错：错误号码 2058，分析是 mysql 密码加密方法变了。

解决方法：Linux下 mysql -u root -p 登录你的 mysql 数据库，然后 执行这条SQL：

```
ALTER USER 'root'@'%' IDENTIFIED WITH mysql_native_password BY 'abc123';
```

然后在重新配置SQLyog的连接，则可连接成功了，OK。

4. MySQL8的密码强度评估（了解）

4.1 MySQL不同版本设置密码(可能出现)

- MySQL5.7中：成功

```
mysql> alter user 'root' identified by 'abcd1234';
Query OK, 0 rows affected (0.00 sec)
```

- MySQL8.0中：失败

```
mysql> alter user 'root' identified by 'abcd1234';    # HelloWorld_123
ERROR 1819 (HY000): Your password does not satisfy the current policy requirements
```

4.2 MySQL8之前的安全策略

在MySQL 8.0之前，MySQL使用的是validate_password插件检测、验证账号密码强度，保障账号的安全性。

安装/启用插件方式1：在参数文件my.cnf中添加参数

```
[mysqld]

plugin-load-add=validate_password.so

\#ON/OFF/FORCE/FORCE_PLUS_PERMANENT: 是否使用该插件(及强制/永久强制使用)

validate-password=FORCE_PLUS_PERMANENT
```

说明1： plugin library中的validate_password文件名的后缀名根据平台不同有所差异。对于Unix和 Unix-like系统而言，它的文件后缀名是.so，对于Windows系统而言，它的文件后缀名是.dll。

说明2： 修改参数后必须重启MySQL服务才能生效。

说明3： 参数FORCE_PLUS_PERMANENT是为了防止插件在MySQL运行时的时候被卸载。当你卸载插件时就会报错。如下所示。

```
mysql> SELECT PLUGIN_NAME, PLUGIN_LIBRARY, PLUGIN_STATUS, LOAD_OPTION
-> FROM INFORMATION_SCHEMA.PLUGINS
-> WHERE PLUGIN_NAME = 'validate_password';
+-----+-----+-----+-----+
| PLUGIN_NAME      | PLUGIN_LIBRARY      | PLUGIN_STATUS | LOAD_OPTION |
+-----+-----+-----+-----+
| validate_password | validate_password.so | ACTIVE        | FORCE_PLUS_PERMANENT |
+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> UNINSTALL PLUGIN validate_password;
ERROR 1702 (HY000): Plugin 'validate_password' is force_plus_permanent and can not be
unloaded
mysql>
```

安装/启用插件方式2：运行时命令安装（推荐）

```
mysql> INSTALL PLUGIN validate_password SONAME 'validate_password.so';
Query OK, 0 rows affected, 1 warning (0.11 sec)
```

此方法也会注册到元数据，也就是mysql.plugin表中，所以不用担心MySQL重启后插件会失效。

4.3 MySQL8的安全策略

1. validate_password说明

MySQL 8.0，引入了服务器组件（Components）这个特性，validate_password插件已用服务器组件重新实现。8.0.25版本的数据库中，默认自动安装validate_password组件。

未安装插件前，执行如下两个指令，执行效果：

```
mysql> show variables like 'validate_password%';
Empty set (0.04 sec)

mysql> SELECT * FROM mysql.component;
ERROR 1146 (42S02): Table 'mysql.component' doesn't exist
```

安装插件后，执行如下两个指令，执行效果：

```
mysql> SELECT * FROM mysql.component;
+-----+-----+-----+
| component_id | component_group_id | component_urn           |
+-----+-----+-----+
|          1   |                 1 | file:///component_validate_password |
+-----+-----+-----+
1 row in set (0.00 sec)
```

```
mysql> show variables like 'validate_password%';
+-----+-----+
| Variable_name          | Value   |
+-----+-----+
| validate_password.check_user_name | ON      |
| validate_password.dictionary_file |        |
| validate_password.length       | 8       |
| validate_password.mixed_case_count | 1       |
| validate_password.number_count | 1       |
| validate_password.policy      | MEDIUM |
| validate_password.special_char_count | 1      |
+-----+-----+
7 rows in set (0.01 sec)
```

关于 `validate_password` 组件对应的系统变量说明：

选项	默认值	参数描述
validate_password_check_user_name	ON	设置为ON的时候表示能将密码设置成当前用户名。
validate_password_dictionary_file		用于检查密码的字典文件的路径名， 默认为空
validate_password_length	8	密码的最小长度，也就是说密码长度必须大于或等于8
validate_password_mixed_case_count	1	如果密码策略是中等或更强的， validate_password要求密码具有的小写和大写字符的最小数量。对于给定的这个值密码必须有那么多小写字符和那么多大写字符。
validate_password_number_count	1	密码必须包含的数字个数
validate_password_policy	MEDIUM	密码强度检验等级，可以使用数值0、1、2或相应的符号值LOW、 MEDIUM、 STRONG来指定。 0/LOW：只检查长度。 1/MEDIUM：检查长度、数字、大小写、特殊字符。 2/STRONG：检查长度、数字、大小写、特殊字符、字典文件。
validate_password_special_char_count	1	密码必须包含的特殊字符个数

提示：

组件和插件的默认值可能有所不同。例如， MySQL 5.7. validate_password_check_user_name的默认值为OFF。

2. 修改安全策略

修改密码验证安全强度

```
SET GLOBAL validate_password_policy=LOW;

SET GLOBAL validate_password_policy=MEDIUM;

SET GLOBAL validate_password_policy=STRONG;

SET GLOBAL validate_password_policy=0; # For LOW

SET GLOBAL validate_password_policy=1; # For MEDIUM

SET GLOBAL validate_password_policy=2; # For HIGH

#注意，如果是插件的话，SQL为set global validate_password_policy=LOW
```

此外，还可以修改密码中字符的长度

```
set global validate_password_length=1;
```

3. 密码强度测试

如果你创建密码是遇到“Your password does not satisfy the current policy requirements”，可以通过函数数组去检测密码是否满足条件：0-100。当评估在100时就是说明使用上了最基本的规则：大写+小写+特殊字符+数字组成的8位以上密码

```
mysql> SELECT VALIDATE_PASSWORD_STRENGTH('medium');
+-----+
| VALIDATE_PASSWORD_STRENGTH('medium') |
+-----+
|                               25 |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT VALIDATE_PASSWORD_STRENGTH('K354*45jKd5');
+-----+
| VALIDATE_PASSWORD_STRENGTH('K354*45jKd5') |
+-----+
|                               100 |
+-----+
1 row in set (0.00 sec)
```

注意：如果没有安装validate_password组件或插件的话，那么这个函数永远都返回0。关于密码复杂度对应的密码复杂度策略。如下表格所示：

Password Test	Return Value
Length < 4	0
Length ≥ 4 and < validate_password.length	25
Satisfies policy 1 (LOW)	50
Satisfies policy 2 (MEDIUM)	75
Satisfies policy 3 (STRONG)	100

4.4 卸载插件、组件(了解)

卸载插件

```
mysql> UNINSTALL PLUGIN validate_password;
Query OK, 0 rows affected, 1 warning (0.01 sec)
```

卸载组件

```
mysql> UNINSTALL COMPONENT 'file:///component_validate_password';
Query OK, 0 rows affected (0.02 sec)
```

5. 字符集的相关操作

5.1 修改MySQL5.7字符集

1. 修改步骤

在MySQL 8.0版本之前，默认字符集为 `latin1`，`utf8`字符集指向的是 `utf8mb3`。网站开发人员在数据库设计的时候往往会将编码修改为`utf8`字符集。如果遗忘修改默认的编码，就会出现乱码的问题。从MySQL 8.0开始，数据库的默认编码将改为 `utf8mb4`，从而避免上述乱码的问题。

操作1：查看默认使用的字符集

```
show variables like 'character%';  
  
# 或者  
  
show variables like '%char%';
```

- MySQL8.0中执行：

```
mysql> show variables like 'character%';
+-----+-----+
| Variable_name      | Value
+-----+-----+
| character_set_client | utf8mb4
| character_set_connection | utf8mb4
| character_set_database | utf8mb4
| character_set_filesystem | binary
| character_set_results | utf8mb4
| character_set_server | utf8mb4
| character_set_system | utf8mb3
| character_sets_dir   | /usr/share/mysql-8.0/charsets/
+-----+-----+
8 rows in set (0.01 sec)

mysql> show variables like '%char%';
+-----+-----+
| Variable_name      | Value
+-----+-----+
| character_set_client | utf8mb4
| character_set_connection | utf8mb4
| character_set_database | utf8mb4
| character_set_filesystem | binary
| character_set_results | utf8mb4
| character_set_server | utf8mb4
| character_set_system | utf8mb3
| character_sets_dir   | /usr/share/mysql-8.0/charsets/
| validate_password.special_char_count | 1
+-----+-----+
9 rows in set (0.00 sec)
```

- MySQL5.7中执行：

MySQL 5.7 默认的客户端和服务器都用了 `latin1`，不支持中文，保存中文会报错。MySQL5.7截图如下：

```

mysql> show variables like 'character%';
+-----+-----+
| Variable_name      | Value   |
+-----+-----+
| character_set_client | utf8
| character_set_connection | utf8
| character_set_database | latin1
| character_set_filesystem | binary
| character_set_results | utf8
| character_set_server | latin1
| character_set_system | utf8
| character_sets_dir | /usr/share/mysql/charsets/
+-----+-----+
8 rows in set (0.00 sec)

```

在MySQL5.7中添加中文数据时，报错：

```

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

mysql> create database dbtest1;
Query OK, 1 row affected (0.00 sec)

mysql> use dbtest1;
Database changed
mysql> create table t_emp(id int,name varchar(15));
Query OK, 0 rows affected (0.01 sec)

mysql> insert into t_emp(id,name)values(1001,'Tom');
Query OK, 1 row affected (0.01 sec)

mysql> insert into t_emp(id,name)values(1002,'张三');
ERROR 1366 (HY000): Incorrect string value: '\xE5\xBC\xA0\xE4\xB8\x89' for column 'name' at row 1
mysql>

```

因为默认情况下，创建表使用的是 `latin1`。如下：

```

Database changed
mysql> show create table t_emp;
+-----+-----+
| Table | Create Table
+-----+
| t_emp | CREATE TABLE `t_emp` (
  `id` int(11) DEFAULT NULL,
  `name` varchar(15) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1 |
+-----+
1 row in set (0.00 sec)

```

操作2：修改字符集

```
vim /etc/my.cnf
```

在MySQL5.7或之前的版本中，在文件最后加上中文字符集配置：

```
character_set_server=utf8
```

```
# For advice on how to change settings please see
# http://dev.mysql.com/doc/refman/5.7/en/server-configuration-defaults.html

[mysqld]
#
# Remove leading # and set to the amount of RAM for the most important data
# cache in MySQL. Start at 70% of total RAM for dedicated server, else 10%.
# innodb_buffer_pool_size = 128M
#
# Remove leading # to turn on a very important data integrity option: logging
# changes to the binary log between backups.
# log_bin
#
# Remove leading # to set options mainly useful for reporting servers.
# The server defaults are faster for transactions and fast SELECTs.
# Adjust sizes as needed, experiment to find the optimal values.
# join_buffer_size = 128M
# sort_buffer_size = 2M
# read_rnd_buffer_size = 2M
datadir=/var/lib/mysql
socket=/var/lib/mysql/mysql.sock

# Disabling symbolic-links is recommended to prevent assorted security risks
symbolic-links=0

log-error=/var/log/mysqld.log
pid-file=/var/run/mysqld/mysqld.pid
character_set_server=utf8
~  
~  
~  
~  
~  
:wq!
```

操作3：重新启动MySQL服务

```
systemctl restart mysqld
```

但是原库、原表的设定不会发生变化，参数修改只对新建的数据库生效。

2. 已有库&表字符集的变更

MySQL5.7版本中，以前创建的库，创建的表字符集还是latin1。

```
mysql> show create table t_emp;
+-----+
| Table | Create Table
+-----+
| t_emp | CREATE TABLE `t_emp` (
  `id` int(11) DEFAULT NULL,
  `name` varchar(15) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1 |
+-----+
1 row in set (0.00 sec)
```

修改已创建数据库的字符集

```
alter database dbtest1 character set 'utf8';
```

修改已创建数据表的字符集

```
alter table t_emp convert to character set 'utf8';
```

```
mysql> alter database dbtest1 character set 'utf8';
Query OK, 1 row affected (0.00 sec)

mysql> alter table t_emp convert to character set 'utf8';
Query OK, 1 row affected (0.04 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> insert into t_emp(id,name)values(1002,'张三');
Query OK, 1 row affected (0.00 sec)

mysql> select * from t_emp;
+----+----+
| id | name |
+----+----+
| 1001 | Tom  |
| 1002 | 张三  |
+----+----+
2 rows in set (0.00 sec)

mysql> show create table t_emp;
+-----+
| Table | Create Table
+-----+
| t_emp | CREATE TABLE `t_emp` (
  `id` int(11) DEFAULT NULL,
  `name` varchar(15) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8 |
+-----+
1 row in set (0.00 sec)
```

注意：但是原有的数据如果是用非'utf8'编码的话，数据本身编码不会发生改变。已有数据需要导出或删除，然后重新插入。

5.2 各级别的字符集

MySQL有4个级别的字符集和比较规则，分别是：

- 服务器级别
- 数据库级别
- 表级别
- 列级别

执行如下SQL语句：

```
show variables like 'character%';
```

```
mysql> show variables like 'character%';
+-----+-----+
| Variable_name      | Value   |
+-----+-----+
| character_set_client | utf8mb4 |
| character_set_connection | utf8mb4 |
| character_set_database | utf8mb4 |
| character_set_filesystem | binary |
| character_set_results | utf8mb4 |
| character_set_server | utf8mb4 |
| character_set_system | utf8mb3 |
| character_sets_dir | /usr/share/mysql-8.0/charsets/ |
+-----+-----+
8 rows in set (0.02 sec)
```

- `character_set_server`: 服务器级别的字符集
- `character_set_database`: 当前数据库的字符集
- `character_set_client`: 服务器解码请求时使用的字符集
- `character_set_connection`: 服务器处理请求时会把请求字符串从`character_set_client`转为`character_set_connection`
- `character_set_results`: 服务器向客户端返回数据时使用的字符集

1. 服务器级别

- `character_set_server` : 服务器级别的字符集。

我们可以在启动服务器程序时通过启动选项或者在服务器程序运行过程中使用 `SET` 语句修改这两个变量的值。比如我们可以在配置文件中这样写：

```
[server]
character_set_server=gbk # 默认字符集
collation_server=gbk_chinese_ci #对应的默认的比较规则
```

当服务器启动的时候读取这个配置文件后这两个系统变量的值便修改了。

2. 数据库级别

- `character_set_database` : 当前数据库的字符集

我们在创建和修改数据库的时候可以指定该数据库的字符集和比较规则，具体语法如下：

```
CREATE DATABASE 数据库名
[[DEFAULT] CHARACTER SET 字符集名称]
[[DEFAULT] COLLATE 比较规则名称];

ALTER DATABASE 数据库名
[[DEFAULT] CHARACTER SET 字符集名称]
[[DEFAULT] COLLATE 比较规则名称];
```

3. 表级别

我们也可以在创建和修改表的时候指定表的字符集和比较规则，语法如下：

```
CREATE TABLE 表名 (列的信息)
  [[DEFAULT] CHARACTER SET 字符集名称]
  [COLLATE 比较规则名称]]
```

```
ALTER TABLE 表名
  [[DEFAULT] CHARACTER SET 字符集名称]
  [COLLATE 比较规则名称]]
```

如果创建和修改表的语句中没有指明字符集和比较规则，将使用该表所在数据库的字符集和比较规则作为该表的字符集和比较规则。

4. 列级别

对于存储字符串的列，同一个表中的不同的列也可以有不同的字符集和比较规则。我们在创建和修改列定义的时候可以指定该列的字符集和比较规则，语法如下：

```
CREATE TABLE 表名(
  列名 字符串类型 [CHARACTER SET 字符集名称] [COLLATE 比较规则名称],
  其他列...
);
```

```
ALTER TABLE 表名 MODIFY 列名 字符串类型 [CHARACTER SET 字符集名称] [COLLATE 比较规则名称];
```

对于某个列来说，如果在创建和修改的语句中没有指明字符集和比较规则，将使用该列所在表的字符集和比较规则作为该列的字符集和比较规则。

提示

在转换列的字符集时需要注意，如果转换前列中存储的数据不能用转换后的字符集进行表示会发生错误。比方说原先列使用的字符集是utf8，列中存储了一些汉字，现在把列的字符集转换为ascii的话就会出错，因为ascii字符集并不能表示汉字字符。

5. 小结

我们介绍的这4个级别字符集和比较规则的联系如下：

- 如果 **创建或修改列** 时没有显式的指定字符集和比较规则，则该列 **默认用表的字符集和比较规则**
- 如果 **创建表时** 没有显式的指定字符集和比较规则，则该表 **默认用数据库的字符集和比较规则**
- 如果 **创建数据库时** 没有显式的指定字符集和比较规则，则该数据库 **默认用服务器的字符集和比较规则**

知道了这些规则之后，对于给定的表，我们应该知道它的各个列的字符集和比较规则是什么，从而根据这个列的类型来确定存储数据时每个列的实际数据占用的存储空间大小了。比方说我们向表 `t` 中插入一条记录：

```
mysql> INSERT INTO t(col) VALUES('我们');
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM t;
+-----+
| s    |
+-----+
| 我们 |
+-----+
1 row in set (0.00 sec)
```

首先列 `col` 使用的字符集是 `gbk`，一个字符‘我’在 `gbk` 中的编码为 `0xCED2`，占用两个字节，两个字符的实际数据就占用4个字节。如果把该列的字符集修改为 `utf8` 的话，这两个字符就实际占用6个字节。

5.3 字符集与比较规则(了解)

1. utf8 与 utf8mb4

`utf8` 字符集表示一个字符需要使用1~4个字节，但是我们常用的一些字符使用1~3个字节就可以表示了。而字符集表示一个字符所用的最大字节长度，在某些方面会影响系统的存储和性能，所以设计MySQL的设计者偷偷的定义了两个概念：

- `utf8mb3`：阉割过的 `utf8` 字符集，只使用1~3个字节表示字符。
- `utf8mb4`：正宗的 `utf8` 字符集，使用1~4个字节表示字符。

2. 比较规则

上表中，MySQL版本一共支持41种字符集，其中的 `Default collation` 列表示这种字符集中一种默认的比较规则，里面包含着该比较规则主要作用于哪种语言，比如 `utf8_polish_ci` 表示以波兰语的规则比较，`utf8_spanish_ci` 是以西班牙语的规则比较，`utf8_general_ci` 是一种通用的比较规则。

后缀表示该比较规则是否区分语言中的重音、大小写。具体如下：

后缀	英文释义	描述
<code>_ai</code>	<code>accent insensitive</code>	不区分重音
<code>_as</code>	<code>accent sensitive</code>	区分重音
<code>_ci</code>	<code>case insensitive</code>	不区分大小写
<code>_cs</code>	<code>case sensitive</code>	区分大小写
<code>_bin</code>	<code>binary</code>	以二进制方式比较

最后一列 `Maxlen`，它代表该种字符集表示一个字符最多需要几个字节。

常用操作1：

```
#查看GBK字符集的比较规则
SHOW COLLATION LIKE 'gbk%';

#查看UTF-8字符集的比较规则
SHOW COLLATION LIKE 'utf8%';
```

常用操作2：

```
#查看服务器的字符集和比较规则
SHOW VARIABLES LIKE '%_server';

#查看数据库的字符集和比较规则
SHOW VARIABLES LIKE '%_database';

#查看具体数据库的字符集
SHOW CREATE DATABASE dbtest1;

#修改具体数据库的字符集
ALTER DATABASE dbtest1 DEFAULT CHARACTER SET 'utf8' COLLATE 'utf8_general_ci';
```

常用操作3：

```
#查看表的字符集  
show create table employees;  
  
#查看表的比较规则  
show table status from atguigudb like 'employees';  
  
#修改表的字符集和比较规则  
ALTER TABLE emp1 DEFAULT CHARACTER SET 'utf8' COLLATE 'utf8_general_ci';
```

5.4 请求到响应过程中字符集的变化

系统变量	描述
character_set_client	服务器解码请求时使用的字符集
character_set_connection	服务器处理请求时会把请求字符串从 character_set_client 转为 character_set_connection
character_set_results	服务器向客户端返回数据时使用的字符集

这几个系统变量在我的计算机上的默认值如下（不同操作系统的默认值可能不同）：

```
mysql> show variables like 'character%';
+-----+-----+
| Variable_name      | Value |
+-----+-----+
| character_set_client | utf8mb4 |
| character_set_connection | utf8mb4 |
| character_set_database | utf8mb4 |
| character_set_filesystem | binary |
| character_set_results | utf8mb4 |
| character_set_server | utf8mb4 |
| character_set_system | utf8mb3 |
| character_sets_dir   | /usr/share/mysql-8.0/charsets/ |
+-----+-----+
8 rows in set (0.02 sec)
```

为了体现出字符集在请求处理过程中的变化，我们这里特意修改一个系统变量的值：

```
mysql> set character_set_connection = gbk;
Query OK, 0 rows affected (0.00 sec)
```

现在假设我们客户端发送的请求是下边这个字符串：

```
SELECT * FROM t WHERE s = '我';
```

为了方便大家理解这个过程，我们只分析字符 '我' 在这个过程中字符集的转换。

现在看一下在请求从发送到结果返回过程中字符集的变化：

1. 客户端发送请求所使用的字符集

一般情况下客户端所使用的字符集和当前操作系统一致，不同操作系统使用的字符集可能不一样，如下：

- 类 Unix 系统使用的是 utf8
- Windows 使用的是 gbk

当客户端使用的是 utf8 字符集，字符 '我' 在发送给服务器的请求中的字节形式就是：

0xE68891

提示

如果你使用的是可视化工具，比如navicat之类的，这些工具可能会使用自定义的字符集来编码发送到服务器的字符串，而不采用操作系统默认的字符集（所以在学习的时候还是尽量用命令行窗口）。

- 服务器接收到客户端发送来的请求其实是一串二进制的字节，它会认为这串字节采用的字符集是 character_set_client，然后把这串字节转换为 character_set_connection 字符集编码的字符。

由于我的计算机上 character_set_client 的值是 utf8，首先会按照 utf8 字符集对字节串 0xE68891 进行解码，得到的字符串就是 '我'，然后按照 character_set_connection 代表的字符集，也就是 gbk 进行编码，得到的结果就是字节串 0xCED2。

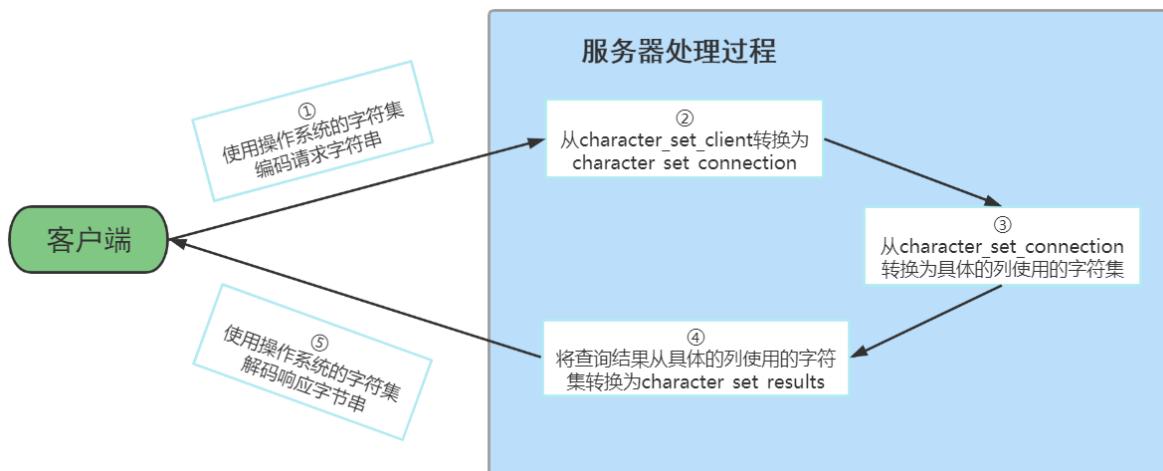
- 因为表 t 的列 col 采用的是 gbk 字符集，与 character_set_connection 一致，所以直接到列中找字节值为 0xCED2 的记录，最后找到了一条记录。

提示

如果某个列使用的字符集和character_set_connection代表的字符集不一致的话，还需要进行一次字符集转换。

- 上一步骤找到的记录中的 col 列其实是一个字节串 0xCED2， col 列是采用 gbk 进行编码的，所以首先会将这个字节串使用 gbk 进行解码，得到字符串 '我'，然后再把这个字符串使用 character_set_results 代表的字符集，也就是 utf8 进行编码，得到了新的字节串：0xE68891，然后发送给客户端。
- 由于客户端是用的字符集是 utf8，所以可以顺利的将 0xE68891 解释成字符 我，从而显示到我们的显示器上，所以我们人类也读懂了返回的结果。

总结图示如下：



6. SQL大小写规范

6.1 Windows和Linux平台区别

在 SQL 中，关键字和函数名是不用区分字母大小写的，比如 SELECT、WHERE、ORDER、GROUP BY 等关键字，以及 ABS、MOD、ROUND、MAX 等函数名。

不过在 SQL 中，你还是要确定大小写的规范，因为在 Linux 和 Windows 环境下，你可能会遇到不同的大小写问题。[windows系统默认大小写不敏感](#)，但是[linux系统是大小写敏感的](#)。

通过如下命令查看：

```
SHOW VARIABLES LIKE '%lower_case_table_names'
```

- Windows系统下：

```
mysql> SHOW VARIABLES LIKE '%lower_case_table_names';
+-----+-----+
| Variable_name      | Value |
+-----+-----+
| lower_case_table_names | 1      |
+-----+-----+
1 row in set, 1 warning (0.01 sec)
```

- Linux系统下：

```
mysql> SHOW VARIABLES LIKE '%lower_case_table_names';
+-----+-----+
| Variable_name      | Value |
+-----+-----+
| lower_case_table_names | 0      |
+-----+-----+
1 row in set (0.01 sec)
```

- lower_case_table_names参数值的设置：

- 默认为0，大小写敏感。
- 设置1，大小写不敏感。创建的表，数据库都是以小写形式存放在磁盘上，对于sql语句都是转换为小写对表和数据库进行查找。
- 设置2，创建的表和数据库依据语句上格式存放，凡是查找都是转换为小写进行。

- 两个平台上SQL大小写的区别具体来说：

MySQL在Linux下数据库名、表名、列名、别名大小写规则是这样的：

- 1、数据库名、表名、表的别名、变量名是严格区分大小写的；
- 2、关键字、函数名称在 SQL 中不区分大小写；
- 3、列名（或字段名）与列的别名（或字段别名）在所有的情况下均是忽略大小写的；

MySQL在Windows的环境下全部不区分大小写

6.2 Linux下大小写规则设置

当想设置为大小写不敏感时，要在 [my.cnf](#) 这个配置文件 [mysqld] 中加入
`lower_case_table_names=1`，然后重启服务器。

- 但是在重启数据库实例之前就需要将原来的数据库和表转换为小写，否则将找不到数据库名。

- 此参数适用于MySQL5.7。在MySQL 8下禁止在重新启动 MySQL 服务时将 `lower_case_table_names` 设置成不同于初始化 MySQL 服务时设置的 `lower_case_table_names` 值。如果非要将MySQL8设置为大小写不敏感，具体步骤为：

- 1、停止MySQL服务
- 2、删除数据目录，即删除 `/var/lib/mysql` 目录
- 3、在MySQL配置文件（`/etc/my.cnf`）中添加 `lower_case_table_names=1`
- 4、启动MySQL服务

6.3 SQL编写建议

如果你的变量名命名规范没有统一，就可能产生错误。这里有一个有关命名规范的建议：

1. 关键字和函数名称全部大写；
2. 数据库名、表名、表别名、字段名、字段别名等全部小写；
3. SQL语句必须以分号结尾。

数据库名、表名和字段名在 Linux MySQL 环境下是区分大小写的，因此建议你统一这些字段的命名规则，比如全部采用小写的方式。

虽然关键字和函数名称在 SQL 中不区分大小写，也就是如果小写的话同样可以执行。但是同时将关键词和函数名称全部大写，以便于区分数据库名、表名、字段名。

7. sql_mode的合理设置

7.1 宽松模式 vs 严格模式

宽松模式：

如果设置的是宽松模式，那么我们在插入数据的时候，即便是给了一个错误的数据，也可能被接受，并且不报错。

举例：我在创建一个表时，该表中有一个字段为name，给name设置的字段类型时 `char(10)`，如果我在插入数据的时候，其中name这个字段对应的有一条数据的 **长度超过了10**，例如'1234567890abc'，超过了设定的字段长度10，那么不会报错，并且取前10个字符存上，也就是说你这个数据被存为了'1234567890'，而'abc'就没有了。但是，我们给的这条数据是错误的，因为超过了字段长度，但是并没有报错，并且mysql自行处理并接受了，这就是宽松模式的效果。

应用场景：通过设置sql mode为宽松模式，来保证大多数sql符合标准的sql语法，这样应用在不同数据库之间进行 **迁移** 时，则不需要对业务sql进行较大的修改。

严格模式：

出现上面宽松模式的错误，应该报错才对，所以MySQL5.7版本就将sql_mode默认值改为了严格模式。所以在 **生产等环境** 中，我们必须采用的是严格模式，进而 **开发、测试环境** 的数据库也必须要设置，这样在开发测试阶段就可以发现问题。并且我们即便是用的MySQL5.6，也应该自行将其改为严格模式。

开发经验：MySQL等数据库总想把关于数据的所有操作都自己包揽下来，包括数据的校验，其实开发中，我们应该在自己 **开发的项目程序级别** 将这些校验给做了，虽然写项目的时候麻烦了一些步骤，但是这样做之后，我们在进行数据库迁移或者在项目的迁移时，就会方便很多。

改为严格模式后可能会存在的问题：

若设置模式中包含了 `NO_ZERO_DATE`，那么MySQL数据库不允许插入零日期，插入零日期会抛出错误而不是警告。例如，表中含字段TIMESTAMP列（如果未声明为NULL或显示DEFAULT子句）将自动分配 `DEFAULT '0000-00-00 00:00:00'`（零时间戳），这显然是不满足sql_mode中的`NO_ZERO_DATE`而报错。

7.2 宽松模式再举例

宽松模式举例1：

```
select * from employees group by department_id limit 10;

set sql_mode = ONLY_FULL_GROUP_BY;

select * from employees group by department_id limit 10;

mysql> select * from emp group by ename limit 10;
+-----+-----+-----+-----+-----+-----+-----+-----+
| id   | empno | ename  | job    | mgr  | hiredate | sal   | comm  | deptno |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 262691 | 362692 | aAAaAA | SALESMAN | 1 | 2017-03-07 | 2000.00 | 400.00 | 100   |
| 105416 | 205417 | aaaaab | SALESMAN | 1 | 2017-03-07 | 2000.00 | 400.00 | 100   |
| 387366 | 487367 | AaAAbE | SALESMAN | 1 | 2017-03-07 | 2000.00 | 400.00 | 108   |
| 211538 | 311539 | aaaabf | SALESMAN | 1 | 2017-03-07 | 2000.00 | 400.00 | 104   |
| 255851 | 355852 | AAaABg | SALESMAN | 1 | 2017-03-07 | 2000.00 | 400.00 | 100   |
| 55039  | 155040 | aAAaBJ | SALESMAN | 1 | 2017-03-07 | 2000.00 | 400.00 | 105   |
| 163126 | 263127 | AaAACI | SALESMAN | 1 | 2017-03-07 | 2000.00 | 400.00 | 102   |
| 160333 | 260334 | aAAaCJ | SALESMAN | 1 | 2017-03-07 | 2000.00 | 400.00 | 107   |
| 294502 | 394503 | aAAaCK | SALESMAN | 1 | 2017-03-07 | 2000.00 | 400.00 | 108   |
| 2696   | 102697 | aaaacl | SALESMAN | 1 | 2017-03-07 | 2000.00 | 400.00 | 109   |
+-----+-----+-----+-----+-----+-----+-----+-----+
10 rows in set (0.00 sec)

mysql> set sql_mode =ONLY_FULL_GROUP_BY;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from emp group by ename limit 10;
ERROR 1055 (42000): 'mytest.emp.id' isn't in GROUP BY
mysql>
```

宽松模式举例2：

```
mysql> desc t1;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key  | Default | Extra |
+-----+-----+-----+-----+-----+
| id    | int(11)   | NO   | PRI  | NULL    |       |
| name  | varchar(2) | YES  |       | NULL    |       |
| age   | smallint(6)| YES  |       | NULL    |       |
+-----+-----+-----+-----+-----+
3 rows in set (0.01 sec)

mysql> set sql_mode = '';
Query OK, 0 rows affected (0.00 sec)
```

```

mysql> insert into t1 values(1,'aa','aaa');
Query OK, 1 row affected, 1 warning (0.01 sec)

mysql> show warnings;
+-----+-----+-----+
| Level | Code | Message          |
+-----+-----+-----+
| Warning | 1366 | Incorrect integer value: 'aaa' for column 'age' at row 1 |
+-----+-----+
1 row in set (0.00 sec)

mysql> select * from t1;
+----+----+----+
| id | name | age  |
+----+----+----+
| 1  | aa   |    0 |
+----+----+----+
1 row in set (0.00 sec)

```

设置 sql_mode 模式为 STRICT_TRANS_TABLES , 然后插入数据:

```

mysql> set sql_mode = 'STRICT_TRANS_TABLES';
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> insert into t1 values(2,'bb','bbb');
ERROR 1366 (HY000): Incorrect integer value: 'bbb' for column 'age' at row 1
mysql>

```

7.3 模式查看和设置

- **查看当前的sql_mode**

```

select @@session.sql_mode

select @@global.sql_mode

#或者

show variables like 'sql_mode';

```

```

mysql> select @@session.sql_mode;
+-----+
| @@session.sql_mode           |
+-----+
| ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION |
+-----+
1 row in set (0.00 sec)

```

```

mysql> select @@global.sql_mode;
+-----+
| @@global.sql_mode           |
+-----+
| ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION |
+-----+
1 row in set (0.00 sec)

```

- **临时设置方式: 设置当前窗口中设置sql_mode**

```

SET GLOBAL sql_mode = 'modes...'; #全局

SET SESSION sql_mode = 'modes...'; #当前会话

```

举例:

```
#改为严格模式。此方法只在当前会话中生效，关闭当前会话就不生效了。
```

```
set SESSION sql_mode='STRICT_TRANS_TABLES';
```

```
#改为严格模式。此方法在当前服务中生效，重启MySQL服务后失效。
```

```
set GLOBAL sql_mode='STRICT_TRANS_TABLES';
```

- **永久设置方式：在/etc/my.cnf中配置sql_mode**

在my.cnf文件(windows系统是my.ini文件)，新增：

```
[mysqld]
sql_mode=ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION
```

然后 **重启MySQL**。

当然生产环境上是禁止重启MySQL服务的，所以采用 **临时设置方式 + 永久设置方式** 来解决线上的问题，那么即便是有一天真的重启了MySQL服务，也会永久生效了。

第02章_MySQL的数据目录

讲师：尚硅谷-宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

1. MySQL的主要目录结构

```
[root@atguigu01 ~]# find / -name mysql
```

安装好MySQL 8之后，我们查看如下的目录结构：

1.1 数据库文件的存放路径

MySQL数据库文件的存放路径：`/var/lib/mysql/`

```
mysql> show variables like 'datadir';
+-----+-----+
| Variable_name | Value      |
+-----+-----+
| datadir       | /var/lib/mysql/ |
+-----+-----+
1 row in set (0.04 sec)
```

从结果中可以看出，在我的计算机上MySQL的数据目录就是`/var/lib/mysql/`。

1.2 相关命令目录

相关命令目录：`/usr/bin` (`mysqladmin`、`mysqlbinlog`、`mysqldump`等命令) 和`/usr/sbin`。

```
[root@atguigu01 sbin]# cd /usr/bin
[root@atguigu01 bin]# find . -name "mysqladmin*"
./mysqladmin
[root@atguigu01 bin]# find . -name "mysqldump*"
./mysqldump
./mysqldumpslow
```

1.3 配置文件目录

配置文件目录：`/usr/share/mysql-8.0` (命令及配置文件)，`/etc/mysql` (如`my.cnf`)

```
[root@atguigu01 mysql]# cd /usr/share/mysql-8.0/
[root@atguigu01 mysql-8.0]# ll
总用量 1072
drwxr-xr-x. 2 root root 4096 7月 27 23:00 bulgarian
drwxr-xr-x. 2 root root 4096 7月 27 23:00 charsets
drwxr-xr-x. 2 root root 4096 7月 27 23:00 czech
drwxr-xr-x. 2 root root 4096 7月 27 23:00 danish
-rw-r--r--. 1 root root 25575 4月 23 23:06 dictionary.txt
drwxr-xr-x. 2 root root 4096 7月 27 23:00 dutch
drwxr-xr-x. 2 root root 4096 7月 27 23:00 english
drwxr-xr-x. 2 root root 4096 7月 27 23:00 estonian
drwxr-xr-x. 2 root root 4096 7月 27 23:00 french
drwxr-xr-x. 2 root root 4096 7月 27 23:00 german
drwxr-xr-x. 2 root root 4096 7月 27 23:00 greek
drwxr-xr-x. 2 root root 4096 7月 27 23:00 hungarian
-rw-r--r--. 1 root root 3999 4月 23 23:06 innodb_memcached_config.sql
-rw-r--r--. 1 root root 2216 4月 24 00:28 install_rewriter.sql
drwxr-xr-x. 2 root root 4096 7月 27 23:00 italian
drwxr-xr-x. 2 root root 4096 7月 27 23:00 japanese
drwxr-xr-x. 2 root root 4096 7月 27 23:00 korean
-rw-r--r--. 1 root root 608148 4月 23 23:06 messages_to_clients.txt
-rw-r--r--. 1 root root 339567 4月 23 23:06 messages_to_error_log.txt
-rw-r--r--. 1 root root 1977 4月 24 00:28 mysql-log-rotate
drwxr-xr-x. 2 root root 4096 7月 27 23:00 norwegian
drwxr-xr-x. 2 root root 4096 7月 27 23:00 norwegian-ny
drwxr-xr-x. 2 root root 4096 7月 27 23:00 polish
drwxr-xr-x. 2 root root 4096 7月 27 23:00 portuguese
drwxr-xr-x. 2 root root 4096 7月 27 23:00 romanian
drwxr-xr-x. 2 root root 4096 7月 27 23:00 russian
```

2. 数据库和文件系统的关系

2.1 查看默认数据库

查看一下在我的计算机上当前有哪些数据库：

```
mysql> SHOW DATABASES;
```

可以看到有4个数据库是属于MySQL自带的系统数据库。

- `mysql`

MySQL 系统自带的核心数据库，它存储了MySQL的用户账户和权限信息，一些存储过程、事件的定义信息，一些运行过程中产生的日志信息，一些帮助信息以及时区信息等。

- `information_schema`

MySQL 系统自带的数据库，这个数据库保存着MySQL服务器 维护的所有其他数据库的信息，比如有哪些表、哪些视图、哪些触发器、哪些列、哪些索引。这些信息并不是真实的用户数据，而是一些描述性信息，有时候也称之为 元数据。在系统数据库 `information_schema` 中提供了一些以 `innodb_sys` 开头的表，用于表示内部系统表。

```
mysql> USE information_schema;
Database changed

mysql> SHOW TABLES LIKE 'innodb_sys%';
+-----+
| Tables_in_information_schema (innodb_sys%) |
```

```

+-----+
| INNODB_SYS_DATAFILES          |
| INNODB_SYS_VIRTUAL           |
| INNODB_SYS_INDEXES           |
| INNODB_SYS_TABLES            |
| INNODB_SYS_FIELDS             |
| INNODB_SYS_TABLESPACES        |
| INNODB_SYS_FOREIGN_COLS       |
| INNODB_SYS_COLUMNS            |
| INNODB_SYS_FOREIGN            |
| INNODB_SYS_TABLESTATS         |
+-----+
10 rows in set (0.00 sec)

```

- **performance_schema**

MySQL 系统自带的数据库，这个数据库里主要保存 MySQL 服务器运行过程中的一些状态信息，可以用来 监控 MySQL 服务的各类性能指标。包括统计最近执行了哪些语句，在执行过程的每个阶段都花费了多长时间，内存的使用情况等信息。

- **sys**

MySQL 系统自带的数据库，这个数据库主要是通过 视图 的形式把 **information_schema** 和 **performance_schema** 结合起来，帮助系统管理员和开发人员监控 MySQL 的技术性能。

2.2 数据库在文件系统中的表示

看一下我的计算机上的数据目录下的内容：

```

[root@atguigu01 mysql]# cd /var/lib/mysql
[root@atguigu01 mysql]# ll
总用量 189980
-rw-r-----. 1 mysql mysql      56 7月 28 00:27 auto.cnf
-rw-r-----. 1 mysql mysql     179 7月 28 00:27 binlog.000001
-rw-r-----. 1 mysql mysql     820 7月 28 01:00 binlog.000002
-rw-r-----. 1 mysql mysql     179 7月 29 14:08 binlog.000003
-rw-r-----. 1 mysql mysql     582 7月 29 16:47 binlog.000004
-rw-r-----. 1 mysql mysql     179 7月 29 16:51 binlog.000005
-rw-r-----. 1 mysql mysql     179 7月 29 16:56 binlog.000006
-rw-r-----. 1 mysql mysql     179 7月 29 17:37 binlog.000007
-rw-r-----. 1 mysql mysql   24555 7月 30 00:28 binlog.000008
-rw-r-----. 1 mysql mysql     179 8月 1 11:57 binlog.000009
-rw-r-----. 1 mysql mysql     156 8月 1 23:21 binlog.000010
-rw-r-----. 1 mysql mysql     156 8月 2 09:25 binlog.000011
-rw-r-----. 1 mysql mysql    1469 8月 4 01:40 binlog.000012
-rw-r-----. 1 mysql mysql     156 8月 6 00:24 binlog.000013
-rw-r-----. 1 mysql mysql     179 8月 6 08:43 binlog.000014
-rw-r-----. 1 mysql mysql     156 8月 6 10:56 binlog.000015
-rw-r-----. 1 mysql mysql     240 8月 6 10:56 binlog.index
-rw-----. 1 mysql mysql    1676 7月 28 00:27 ca-key.pem
-rw-r--r--. 1 mysql mysql    1112 7月 28 00:27 ca.pem
-rw-r--r--. 1 mysql mysql    1112 7月 28 00:27 client-cert.pem
-rw-----. 1 mysql mysql    1676 7月 28 00:27 client-key.pem
drwxr-x---. 2 mysql mysql    4096 7月 29 16:34 dbtest
-rw-r-----. 1 mysql mysql  196608 8月 6 10:58 #ib_16384_0 dblwr
-rw-r-----. 1 mysql mysql  8585216 7月 28 00:27 #ib_16384_1 dblwr
-rw-r-----. 1 mysql mysql    3486 8月 6 08:43 ib_buffer_pool
-rw-r-----. 1 mysql mysql 12582912 8月 6 10:56 ibdata1
-rw-r-----. 1 mysql mysql 50331648 8月 6 10:58 ib_logfile0

```

```
-rw-r-----. 1 mysql mysql 50331648 7月 28 00:27 ib_logfile1
-rw-r-----. 1 mysql mysql 12582912 8月 6 10:56 ibtmp1
drwxr-x---. 2 mysql mysql 4096 8月 6 10:56 #innodb_temp
drwxr-x---. 2 mysql mysql 4096 7月 28 00:27 mysql
-rw-r-----. 1 mysql mysql 26214400 8月 6 10:56 mysql.ibd
srwxrwxrwx. 1 mysql mysql 0 8月 6 10:56 mysql.sock
-rw-----. 1 mysql mysql 5 8月 6 10:56 mysql.sock.lock
drwxr-x---. 2 mysql mysql 4096 7月 28 00:27 performance_schema
-rw-----. 1 mysql mysql 1680 7月 28 00:27 private_key.pem
-rw-r--r--. 1 mysql mysql 452 7月 28 00:27 public_key.pem
-rw-r--r--. 1 mysql mysql 1112 7月 28 00:27 server-cert.pem
-rw-----. 1 mysql mysql 1680 7月 28 00:27 server-key.pem
drwxr-x---. 2 mysql mysql 4096 7月 28 00:27 sys
drwxr-x---. 2 mysql mysql 4096 7月 29 23:10 temp
-rw-r-----. 1 mysql mysql 16777216 8月 6 10:58 undo_001
-rw-r-----. 1 mysql mysql 16777216 8月 6 10:58 undo_002
```

这个数据目录下的文件和子目录比较多，除了 `information_schema` 这个系统数据库外，其他的数据库在 `数据目录` 下都有对应的子目录。

以我的 `temp` 数据库为例，在MySQL5.7 中打开：

```
[root@atguigu02 mysql]# cd ./temp
[root@atguigu02 temp]# ll
总用量 1144
-rw-r-----. 1 mysql mysql 8658 8月 18 11:32 countries.frm
-rw-r-----. 1 mysql mysql 114688 8月 18 11:32 countries.ibd
-rw-r-----. 1 mysql mysql 61 8月 18 11:32 db.opt
-rw-r-----. 1 mysql mysql 8716 8月 18 11:32 departments.frm
-rw-r-----. 1 mysql mysql 147456 8月 18 11:32 departments.ibd
-rw-r-----. 1 mysql mysql 3017 8月 18 11:32 emp_details_view.frm
-rw-r-----. 1 mysql mysql 8982 8月 18 11:32 employees.frm
-rw-r-----. 1 mysql mysql 180224 8月 18 11:32 employees.ibd
-rw-r-----. 1 mysql mysql 8660 8月 18 11:32 job_grades.frm
-rw-r-----. 1 mysql mysql 98304 8月 18 11:32 job_grades.ibd
-rw-r-----. 1 mysql mysql 8736 8月 18 11:32 job_history.frm
-rw-r-----. 1 mysql mysql 147456 8月 18 11:32 job_history.ibd
-rw-r-----. 1 mysql mysql 8688 8月 18 11:32 jobs.frm
-rw-r-----. 1 mysql mysql 114688 8月 18 11:32 jobs.ibd
-rw-r-----. 1 mysql mysql 8790 8月 18 11:32 locations.frm
-rw-r-----. 1 mysql mysql 131072 8月 18 11:32 locations.ibd
-rw-r-----. 1 mysql mysql 8614 8月 18 11:32 regions.frm
-rw-r-----. 1 mysql mysql 114688 8月 18 11:32 regions.ibd
```

在MySQL8.0中打开：

```
[root@atguigu01 mysql]# cd ./temp
[root@atguigu01 temp]# ll
总用量 1080
-rw-r-----. 1 mysql mysql 131072 7月 29 23:10 countries.ibd
-rw-r-----. 1 mysql mysql 163840 7月 29 23:10 departments.ibd
-rw-r-----. 1 mysql mysql 196608 7月 29 23:10 employees.ibd
-rw-r-----. 1 mysql mysql 114688 7月 29 23:10 job_grades.ibd
-rw-r-----. 1 mysql mysql 163840 7月 29 23:10 job_history.ibd
-rw-r-----. 1 mysql mysql 131072 7月 29 23:10 jobs.ibd
-rw-r-----. 1 mysql mysql 147456 7月 29 23:10 locations.ibd
-rw-r-----. 1 mysql mysql 131072 7月 29 23:10 regions.ibd
```

2.3 表在文件系统中的表示

2.3.1 InnoDB存储引擎模式

1. 表结构

为了保存表结构， InnoDB 在 数据目录 下对应的数据库子目录下创建了一个专门用于 描述表结构的文件， 文件名是这样：

表名.frm

比方说我们在 atguigu 数据库下创建一个名为 test 的表：

```
mysql> USE atguigu;
Database changed

mysql> CREATE TABLE test (
    ->     c1 INT
    -> );
Query OK, 0 rows affected (0.03 sec)
```

那在数据库 atguigu 对应的子目录下就会创建一个名为 test.frm 的用于描述表结构的文件。.frm文件的格式在不同的平台上都是相同的。这个后缀名为.frm是以 二进制格式 存储的，我们直接打开是乱码的。

2. 表中数据和索引

① 系统表空间 (system tablespace)

默认情况下，InnoDB会在数据目录下创建一个名为 ibdata1、大小为 12M 的文件，这个文件就是对应的 系统表空间 在文件系统上的表示。怎么才12M？注意这个文件是 自扩展文件，当不够用的时候它会自己增加文件大小。

当然，如果你想让系统表空间对应文件系统上多个实际文件，或者仅仅觉得原来的 ibdata1 这个文件名难听，那可以在MySQL启动时配置对应的文件路径以及它们的大小，比如我们这样修改一下my.cnf 配置文件：

```
[server]
innodb_data_file_path=data1:512M;data2:512M:autoextend
```

② 独立表空间(file-per-table tablespace)

在MySQL5.6以及之后的版本中，InnoDB并不会默认的把各个表的数据存储到系统表空间中，而是为每一个表建立一个独立表空间，也就是说我们创建了多少个表，就有多少个独立表空间。使用 独立表空间 来存储表数据的话，会在该表所属数据库对应的子目录下创建一个表示该独立表空间的文件，文件名和表名相同，只不过添加了一个 .ibd 的扩展名而已，所以完整的文件名称长这样：

表名.ibd

比如：我们使用了 独立表空间 去存储 atguigu 数据库下的 test 表的话，那么在该表所在数据库对应的 atguigu 目录下会为 test 表创建这两个文件：

test.frm
test.ibd

其中 test.ibd 文件就用来存储 test 表中的数据和索引。

③ 系统表空间与独立表空间的设置

我们可以自己指定使用 系统表空间 还是 独立表空间 来存储数据，这个功能由启动参数 innodb_file_per_table 控制，比如说我们想刻意将表数据都存储到 系统表空间 时，可以在启动 MySQL服务器的时候这样配置：

```
[server]  
innodb_file_per_table=0 # 0: 代表使用系统表空间; 1: 代表使用独立表空间
```

默认情况：

```
mysql> show variables like 'innodb_file_per_table';  
+-----+-----+  
| Variable_name      | Value |  
+-----+-----+  
| innodb_file_per_table | ON    |  
+-----+-----+  
1 row in set (0.01 sec)
```

④ 其他类型的表空间

随着MySQL的发展，除了上述两种老牌表空间之外，现在还新提出了一些不同类型的表空间，比如通用表空间（general tablespace）、临时表空间（temporary tablespace）等。

2.3.2 MyISAM存储引擎模式

1. 表结构

在存储表结构方面， MyISAM 和 InnoDB 一样，也是在 数据目录 下对应的数据库子目录下创建了一个专门用于描述表结构的文件：

表名.frm

2. 表中数据和索引

在MyISAM中的索引全部都是 二级索引，该存储引擎的 数据和索引是分开存放 的。所以在文件系统中也是使用不同的文件来存储数据文件和索引文件，同时表数据都存放在对应的数据库子目录下。假如 test 表使用MyISAM存储引擎的话，那么在它所在数据库对应的 atguigu 目录下会为 test 表创建这三个文件：

```
test.frm 存储表结构  
test.MYD 存储数据 (MYData)  
test.MYI 存储索引 (MYIndex)
```

举例：创建一个 MyISAM 表，使用 ENGINE 选项显式指定引擎。因为 InnoDB 是默认引擎。

```
CREATE TABLE `student_myisam` (  
    `id` bigint NOT NULL AUTO_INCREMENT,  
    `name` varchar(64) DEFAULT NULL,  
    `age` int DEFAULT NULL,  
    `sex` varchar(2) DEFAULT NULL,  
    PRIMARY KEY (`id`)  
)ENGINE=MYISAM AUTO_INCREMENT=0 DEFAULT CHARSET=utf8mb3;
```

2.4 小结

举例： 数据库a， 表b。

1、如果表b采用 InnoDB， data\ a中会产生1个或者2个文件：

- b.frm：描述表结构文件，字段长度等
- 如果采用 系统表空间 模式的，数据信息和索引信息都存储在 ibdata1 中
- 如果采用 独立表空间 存储模式， data\ a中还会产生 b.ibd 文件（存储数据信息和索引信息）

此外：

① MySQL5.7 中会在data/a的目录下生成 db.opt 文件用于保存数据库的相关配置。比如：字符集、比较规则。而MySQL8.0不再提供db.opt文件。

② MySQL8.0中不再单独提供b.frm，而是合并在b.ibd文件中。

2、如果表b采用 MyISAM， data\ a中会产生3个文件：

- MySQL5.7 中： b.frm：描述表结构文件，字段长度等。
MySQL8.0 中 b.xxx.sdi：描述表结构文件，字段长度等
- b.MYD (MYData)：数据信息文件，存储数据信息(如果采用独立表存储模式)
- b.MYI (MYIndex)：存放索引信息文件

第03章_用户与权限管理

讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

1. 用户管理

1.1 登录MySQL服务器

启动MySQL服务后，可以通过mysql命令来登录MySQL服务器，命令如下：

```
mysql -h hostname|hostIP -P port -u username -p DatabaseName -e "SQL语句"
```

下面详细介绍命令中的参数：

- **-h参数** 后面接主机名或者主机IP，hostname为主机，hostIP为主机IP。
- **-P参数** 后面接MySQL服务的端口，通过该参数连接到指定的端口。MySQL服务的默认端口是3306，不使用该参数时自动连接到3306端口，port为连接的端口号。
- **-u参数** 后面接用户名，username为用户名。
- **-p参数** 会提示输入密码。
- **DatabaseName参数** 指明登录到哪一个数据库中。如果没有该参数，就会直接登录到MySQL数据库中，然后可以使用USE命令来选择数据库。
- **-e参数** 后面可以直接加SQL语句。登录MySQL服务器以后即可执行这个SQL语句，然后退出MySQL服务器。

举例：

```
mysql -uroot -p -hlocalhost -P3306 mysql -e "select host,user from user"
```

1.2 创建用户

CREATE USER语句的基本语法形式如下：

```
CREATE USER 用户名 [ IDENTIFIED BY '密码' ][,用户名 [ IDENTIFIED BY '密码' ]];
```

- 用户名参数表示新建用户的账户，由 **用户（User）** 和 **主机名（Host）** 构成；
- “[]”表示可选，也就是说，可以指定用户登录时需要密码验证，也可以不指定密码验证，这样用户可以直接登录。不过，不指定密码的方式不安全，不推荐使用。如果指定密码值，这里需要使用 IDENTIFIED BY 指定明文密码值。
- CREATE USER语句可以同时创建多个用户。

举例：

```
CREATE USER zhang3 IDENTIFIED BY '123123'; # 默认host是 %
```

```
CREATE USER 'kangshifu'@'localhost' IDENTIFIED BY '123456';
```

1.3 修改用户

修改用户名：

```
UPDATE mysql.user SET USER='li4' WHERE USER='wang5';  
  
FLUSH PRIVILEGES;
```

```
mysql> update mysql.user set user='li4' where user='wang5';  
Query OK, 1 row affected (0.27 sec)  
Rows matched: 1  Changed: 1  Warnings: 0
```

1.4 删除用户

方式1：使用DROP方式删除（推荐）

使用DROP USER语句来删除用户时，必须用于DROP USER权限。DROP USER语句的基本语法形式如下：

```
DROP USER user[,user]...;
```

举例：

```
DROP USER li4 ; # 默认删除host为%的用户  
  
DROP USER 'kangshifu'@'localhost' ;
```

```
mysql> drop user li4;  
Query OK, 0 rows affected (0.00 sec)  
  
mysql> select host,user,password,select_priv,insert_priv,drop_priv from mysql.user;  
+-----+-----+-----+-----+-----+-----+  
| host | user | password | select_priv | insert_priv | drop_priv |  
+-----+-----+-----+-----+-----+-----+  
| %   | root | *E56A114692FE0DE073F9A1DD68A00EEB9703F3F1 | Y       | Y       | Y       |  
| jack.atguigu | root | *E56A114692FE0DE073F9A1DD68A00EEB9703F3F1 | Y       | Y       | Y       |  
| 127.0.0.1 | root | *E56A114692FE0DE073F9A1DD68A00EEB9703F3F1 | Y       | Y       | Y       |  
| ::1   | root | *E56A114692FE0DE073F9A1DD68A00EEB9703F3F1 | Y       | Y       | Y       |  
| localhost |      | *E56A114692FE0DE073F9A1DD68A00EEB9703F3F1 | N       | N       | N       |  
| jack.atguigu |      | *E56A114692FE0DE073F9A1DD68A00EEB9703F3F1 | N       | N       | N       |  
| localhost | root | *E56A114692FE0DE073F9A1DD68A00EEB9703F3F1 | Y       | Y       | Y       |  
+-----+-----+-----+-----+-----+-----+  
7 rows in set (0.00 sec)
```

方式2：使用DELETE方式删除

```
DELETE FROM mysql.user WHERE Host='hostname' AND User='username';
```

执行完DELETE命令后要使用FLUSH命令来使用户生效，命令如下：

```
FLUSH PRIVILEGES;
```

举例：

```
DELETE FROM mysql.user WHERE Host='localhost' AND User='Emily';  
  
FLUSH PRIVILEGES;
```

注意：不推荐通过 `DELETE FROM USER u WHERE USER='li4'` 进行删除，系统会有残留信息保留。而drop user命令会删除用户以及对应的权限，执行命令后你会发现mysql.user表和mysql.db表的相应记录都消失了。

1.5 设置当前用户密码

旧的写法如下：

```
# 修改当前用户的密码：（MySQL5.7测试有效）
SET PASSWORD = PASSWORD('123456');
```

这里介绍 推荐的写法：

1. 使用ALTER USER命令来修改当前用户密码 用户可以使用ALTER命令来修改自身密码，如下语句代表修改当前登录用户的密码。基本语法如下：

```
ALTER USER USER() IDENTIFIED BY 'new_password';
```

2. 使用SET语句来修改当前用户密码 使用root用户登录MySQL后，可以使用SET语句来修改密码，具体SQL语句如下：

```
SET PASSWORD='new_password';
```

该语句会自动将密码加密后再赋给当前用户。

1.6 修改其它用户密码

1. 使用ALTER语句来修改普通用户的密码 可以使用ALTER USER语句来修改普通用户的密码。基本语法形式如下：

```
ALTER USER user [IDENTIFIED BY '新密码']
[,user[IDENTIFIED BY '新密码']]...;
```

2. 使用SET命令来修改普通用户的密码 使用root用户登录到MySQL服务器后，可以使用SET语句来修改普通用户的密码。SET语句的代码如下：

```
SET PASSWORD FOR 'username'@'hostname' = 'new_password';
```

3. 使用UPDATE语句修改普通用户的密码（不推荐）

```
UPDATE MySQL.user SET authentication_string=PASSWORD("123456")
WHERE User = "username" AND Host = "hostname";
```

1.7 MySQL8密码管理(了解)

1. 密码过期策略

- 在MySQL中，数据库管理员可以 手动设置 账号密码过期，也可以建立一个 自动 密码过期策略。
- 过期策略可以是 全局的 ，也可以为 每个账号 设置单独的过期策略。

```
ALTER USER user PASSWORD EXPIRE;
```

练习：

```
ALTER USER 'kangshifu'@'localhost' PASSWORD EXPIRE;
```

- 方式①：使用SQL语句更改该变量的值并持久化**

```
SET PERSIST default_password_lifetime = 180; # 建立全局策略，设置密码每隔180天过期
```

- 方式②：配置文件my.cnf中进行维护**

```
[mysqld]
default_password_lifetime=180 #建立全局策略，设置密码每隔180天过期
```

手动设置指定时间过期方式2：单独设置

每个账号既可延用全局密码过期策略，也可单独设置策略。在 `CREATE USER` 和 `ALTER USER` 语句上加入 `PASSWORD EXPIRE` 选项可实现单独设置策略。下面是一些语句示例。

```
#设置kangshifu账号密码每90天过期:
CREATE USER 'kangshifu'@'localhost' PASSWORD EXPIRE INTERVAL 90 DAY;
ALTER USER 'kangshifu'@'localhost' PASSWORD EXPIRE INTERVAL 90 DAY;

#设置密码永不过期:
CREATE USER 'kangshifu'@'localhost' PASSWORD EXPIRE NEVER;
ALTER USER 'kangshifu'@'localhost' PASSWORD EXPIRE NEVER;

#延用全局密码过期策略:
CREATE USER 'kangshifu'@'localhost' PASSWORD EXPIRE DEFAULT;
ALTER USER 'kangshifu'@'localhost' PASSWORD EXPIRE DEFAULT;
```

2. 密码重用策略

- 手动设置密码重用方式1：全局
 - 方式①：使用SQL

```
SET PERSIST password_history = 6; #设置不能选择最近使用过的6个密码

SET PERSIST password_reuse_interval = 365; #设置不能选择最近一年内的密码
```

```
mysql> SET PERSIST password_history = 6;
Query OK, 0 rows affected (0.00 sec)

mysql> SET PERSIST password_reuse_interval = 365;
Query OK, 0 rows affected (0.00 sec)
```

- 方式②：my.cnf配置文件

```
[mysqld]
password_history=6
password_reuse_interval=365
```

- 手动设置密码重用方式2：单独设置

```
#不能使用最近5个密码:
CREATE USER 'kangshifu'@'localhost' PASSWORD HISTORY 5;
ALTER USER 'kangshifu'@'localhost' PASSWORD HISTORY 5;

#不能使用最近365天内的密码:
CREATE USER 'kangshifu'@'localhost' PASSWORD REUSE INTERVAL 365 DAY;
ALTER USER 'kangshifu'@'localhost' PASSWORD REUSE INTERVAL 365 DAY;

#既不能使用最近5个密码，也不能使用365天内的密码
CREATE USER 'kangshifu'@'localhost'
PASSWORD HISTORY 5
PASSWORD REUSE INTERVAL 365 DAY;

ALTER USER 'kangshifu'@'localhost'
```

```
PASSWORD HISTORY 5  
PASSWORD REUSE INTERVAL 365 DAY;
```

2. 权限管理

2.1 权限列表

MySQL到底都有哪些权限呢？

```
mysql> show privileges;
```

(1) **CREATE**和**DROP权限**，可以创建新的数据库和表，或删除（移掉）已有的数据库和表。如果将MySQL数据库中的**DROP权限**授予某用户，用户就可以删除MySQL访问权限保存的数据库。 (2) **SELECT、INSERT、UPDATE和DELETE权限** 允许在一个数据库现有的表上实施操作。 (3) **SELECT权限** 只有在它们真正从一个表中检索行时才被用到。 (4) **INDEX权限** 允许创建或删除索引，**INDEX**适用于已有的表。如果具有某个表的**CREATE权限**，就可以在**CREATE TABLE**语句中包括索引定义。 (5) **ALTER权限** 可以使用**ALTER TABLE**来更改表的结构和重新命名表。 (6) **CREATE ROUTINE权限** 用来创建保存的程序（函数和程序），**ALTER ROUTINE权限**用来更改和删除保存的程序，**EXECUTE权限** 用来执行保存的程序。 (7) **GRANT权限** 允许授权给其他用户，可用于数据库、表和保存的程序。 (8) **FILE权限** 使用户可以使用**LOAD DATA INFILE**和**SELECT ... INTO OUTFILE**语句读或写服务器上的文件，任何被授予**FILE权限**的用户都能读或写MySQL服务器上的任何文件（说明用户可以读任何数据库目录下的文件，因为服务器可以访问这些文件）。

2.2 授予权限的原则

权限控制主要是出于安全因素，因此需要遵循以下几个 **经验原则**：

- 1、只授予能 **满足需要的最小权限**，防止用户干坏事。比如用户只是需要查询，那就只给**select权限**就可以了，不要给用户赋予**update**、**insert**或者**delete权限**。
- 2、创建用户的时候 **限制用户的登录主机**，一般是限制成指定IP或者内网IP段。
- 3、为每个用户 **设置满足密码复杂度的密码**。
- 4、**定期清理不需要的用户**，回收权限或者删除用户。

2.3 授予权限

给用户授权的方式有 2 种，分别是通过把 **角色赋予用户**给**用户授权** 和 **直接给用户授权**。用户是数据库的使用者，我们可以通过给用户授予访问数据库中资源的权限，来控制使用者对数据库的访问，消除安全隐患。

授权命令：

```
GRANT 权限1, 权限2, ...权限n ON 数据库名称.表名称 TO 用户名@用户地址 [IDENTIFIED BY '密码口令'];
```

- 该权限如果发现没有该用户，则会直接新建一个用户。

比如：

- 给li4用户用本地命令行方式，授予atguigudb这个库下的所有表的插删改查的权限。

```
GRANT SELECT, INSERT, DELETE, UPDATE ON atguigudb.* TO li4@localhost ;
```

- 授予通过网络方式登录的joe用户，对所有库所有表的全部权限，密码设为123。注意这里唯独不包括grant的权限

```
GRANT ALL PRIVILEGES ON *.* TO joe@'%' IDENTIFIED BY '123';
```

我们在开发应用的时候，经常会遇到一种需求，就是要根据用户的不同，对数据进行横向和纵向的分组。

- 所谓横向的分组，就是指用户可以接触到的数据的范围，比如可以看到哪些表的数据；
- 所谓纵向的分组，就是指用户对接触到的数据能访问到什么程度，比如能看、能改，甚至是删除。

2.4 查看权限

- 查看当前用户权限

```
SHOW GRANTS;  
# 或  
SHOW GRANTS FOR CURRENT_USER;  
# 或  
SHOW GRANTS FOR CURRENT_USER();
```

- 查看某用户的全局权限

```
SHOW GRANTS FOR 'user'@'主机地址' ;
```

2.5 收回权限

收回权限就是取消已经赋予用户的某些权限。**收回用户不必要的权限可以在一定程度上保证系统的安全性。** MySQL中使用 REVOKE语句 取消用户的某些权限。使用REVOKE收回权限之后，用户账户的记录将从db、host、tables_priv和columns_priv表中删除，但是用户账户记录仍然在user表中保存（删除user表中的账户记录使用DROP USER语句）。

注意：在将用户账户从user表删除之前，应该收回相应用户的所有权限。

- 收回权限命令

```
REVOKE 权限1, 权限2, ...权限n ON 数据库名称.表名称 FROM 用户名@用户地址;
```

- 举例

```
#收回全库全表的所有权限  
REVOKE ALL PRIVILEGES ON *.* FROM joe@'%';  
  
#收回mysql库下的所有表的插删改查权限  
REVOKE SELECT, INSERT, UPDATE, DELETE ON mysql.* FROM joe@localhost;
```

- 注意：须用户重新登录后才能生效

3. 权限表

3.1 user表

user表是MySQL中最重要的一个权限表，记录用户账号和权限信息，有49个字段。如下图：

字段名	数据类型	默认值
Host	char(60)	
User	char(16)	
authentication_string	text	
Select_priv	enum('N','Y')	N
Insert_priv	enum('N','Y')	N
Update_priv	enum('N','Y')	N
Delete_priv	enum('N','Y')	N
Create_priv	enum('N','Y')	N
Drop_priv	enum('N','Y')	N
Reload_priv	enum('N','Y')	N
Shutdown_priv	enum('N','Y')	N
Process_priv	enum('N','Y')	N
File_priv	enum('N','Y')	N
Grant_priv	enum('N','Y')	N
References_priv	enum('N','Y')	N
Index_priv	enum('N','Y')	N
Alter_priv	enum('N','Y')	N
Show_db_priv	enum('N','Y')	N
Super_priv	enum('N','Y')	N
Create_tmp_table_priv	enum('N','Y')	N
Lock_tables_priv	enum('N','Y')	N
Execute_priv	enum('N','Y')	N
Repl_slave_priv	enum('N','Y')	N
Repl_client_priv	enum('N','Y')	N
Create_view_priv	enum('N','Y')	N
Show_view_priv	enum('N','Y')	N
Create_routine_priv	enum('N','Y')	N
Alter_routine_priv	enum('N','Y')	N
Create_user_priv	enum('N','Y')	N
Event_priv	enum('N','Y')	N
Trigger_priv	enum('N','Y')	N
Create_tablespace_priv	enum('N','Y')	N
ssl_type	enum("ANY",'X509','SPECIFIED')	
ssl_cipher	blob	NULL
x509_issuer	blob	NULL
x509_subject	blob	NULL
max_questions	int(11) unsigned	0
max_updates	int(11) unsigned	0
max_connections	int(11) unsigned	0
max_user_connections	int(11) unsigned	0
plugin	char(64)	
authentication_string	text	NULL

这些字段可以分成4类，分别是范围列（或用户列）、权限列、安全列和资源控制列。

1. 范围列（或用户列）

- host : 表示连接类型
 - % 表示所有远程通过 TCP 方式的连接
 - IP 地址 如 (192.168.1.2、127.0.0.1) 通过制定 ip 地址进行的 TCP 方式的连接
 - 机器名 通过制定网络中的机器名进行的 TCP 方式的连接
 - ::1 IPv6 的本地 ip 地址，等同于 IPv4 的 127.0.0.1
 - localhost 本地方式通过命令行方式的连接，比如 mysql -u xxx -p xxx 方式的连接。
- user : 表示用户名，同一用户通过不同方式链接的权限是不一样的。
- password : 密码
 - 所有密码串通过 password(明文字符串) 生成的密文字符串。MySQL 8.0 在用户管理方面增加了角色管理，默认的密码加密方式也做了调整，由之前的 SHA1 改为了 SHA2，不可逆。同时加上 MySQL 5.7 的禁用用户和用户过期的功能，MySQL 在用户管理方面的功能和安全性都较之前版本大大的增强了。
 - mysql 5.7 及之后版本的密码保存到 authentication_string 字段中不再使用 password 字段。

2. 权限列

- Grant_priv 字段
 - 表示是否拥有 GRANT 权限
- Shutdown_priv 字段
 - 表示是否拥有停止 MySQL 服务的权限
- Super_priv 字段
 - 表示是否拥有超级权限
- Execute_priv 字段
 - 表示是否拥有 EXECUTE 权限。拥有 EXECUTE 权限，可以执行存储过程和函数。
- Select_priv, Insert_priv 等
 - 为该用户所拥有的权限。

3. 安全列 安全列只有 6 个字段，其中两个是 ssl 相关的 (ssl_type、ssl_cipher)，用于 加密；两个是 x509 相关的 (x509_issuer、x509_subject)，用于 标识用户；另外两个 Plugin 字段用于 验证用户身份 的插件，该字段不能为空。如果该字段为空，服务器就使用内建授权验证机制验证用户身份。

4. 资源控制列 资源控制列的字段用来 限制用户使用的资源，包含 4 个字段，分别为：

① max_questions，用户每小时允许执行的查询操作次数；② max_updates，用户每小时允许执行的更新操作次数；③ max_connections，用户每小时允许执行的连接操作次数；④ max_user_connections，用户允许同时建立的连接次数。

查看字段：

```
DESC mysql.user;
```

查看用户，以列的方式显示数据：

```
SELECT * FROM mysql.user \G;
```

查询特定字段：

```
SELECT host,user,authentication_string,select_priv,insert_priv,drop_priv
FROM mysql.user;
```

```

mysql> select host,user,authentication_string,Select_priv from user;
+-----+-----+-----+-----+
| host | user | authentication_string | Select_priv |
+-----+-----+-----+-----+
| localhost | root | *E56A114692FE0DE073F9A1DD68A00EEB9703F3F1 | Y
| localhost | mysql.sys | *THISISNOTAVALIDPASSWORDTHATCANBEUSEDHERE | N
| % | zhang3 | *E56A114692FE0DE073F9A1DD68A00EEB9703F3F1 | N
| % | root | *E56A114692FE0DE073F9A1DD68A00EEB9703F3F1 | Y
+-----+-----+-----+-----+
4 rows in set (0.01 sec)

```

3.2 db表

使用DESCRIBE查看db表的基本结构：

```
DESCRIBE mysql.db;
```

1. 用户列 db表用户列有3个字段，分别是Host、User、Db。这3个字段分别表示主机名、用户名和数据库名。表示从某个主机连接某个用户对某个数据库的操作权限，这3个字段的组合构成了db表的主键。

2. 权限列

Create_routine_priv和Alter_routine_priv这两个字段决定用户是否具有创建和修改存储过程的权限。

3.3 tables_priv表和columns_priv表

tables_priv表用来 对表设置操作权限，columns_priv表用来对表的 某一列设置权限。tables_priv表和columns_priv表的结构分别如图：

```
desc mysql.tables_priv;
```

tables_priv表有8个字段，分别是Host、Db、User、Table_name、Grantor、Timestamp、Table_priv和Column_priv，各个字段说明如下：

- Host、Db、User 和 Table_name 四个字段分别表示主机名、数据库名、用户名和表名。
- Grantor表示修改该记录的用户。
- Timestamp表示修改该记录的时间。
- Table_priv 表示对象的操作权限。包括Select、Insert、Update、Delete、Create、Drop、Grant、References、Index和Alter。
- Column_priv字段表示对表中的列的操作权限，包括Select、Insert、Update和References。

```
desc mysql.columns_priv;
```

3.4 procs_priv表

procs_priv表可以对 存储过程和存储函数设置操作权限，表结构如图：

```
desc mysql.procs_priv;
```

字段名	数据类型	默认值
Host	char(60)	
Db	char(64)	
User	char(16)	
Routine_name	char(64)	
Routine_type	enum('FUNCTION','PROCEDURE')	NULL
Grantor	char(77)	
Proc_priv	set('Execute','Alter Routine','Grant')	
Timestamp	timestamp	CURRENT_TIMESTAMP

4. 访问控制(了解)

4.1 连接核实阶段

当用户试图连接MySQL服务器时，服务器基于用户的身份以及用户是否能提供正确的密码验证身份来确定接受或者拒绝连接。即客户端用户会在连接请求中提供用户名、主机地址、用户密码，MySQL服务器接收到用户请求后，会**使用user表中的host、user和authentication_string这3个字段匹配客户端提供的信息**。

服务器只有在user表记录的Host和User字段匹配客户端主机名和用户名，并且提供正确的密码时才接受连接。**如果连接核实没有通过，服务器就完全拒绝访问；否则，服务器接受连接，然后进入阶段2等待用户请求。**

4.2 请求核实阶段

一旦建立了连接，服务器就进入了访问控制的阶段2，也就是请求核实阶段。对此连接上进来的每个请求，服务器检查该请求要执行什么操作、是否有足够的权限来执行它，这正是需要授权表中的权限列发挥作用的地方。这些权限可以来自user、db、table_priv和column_priv表。

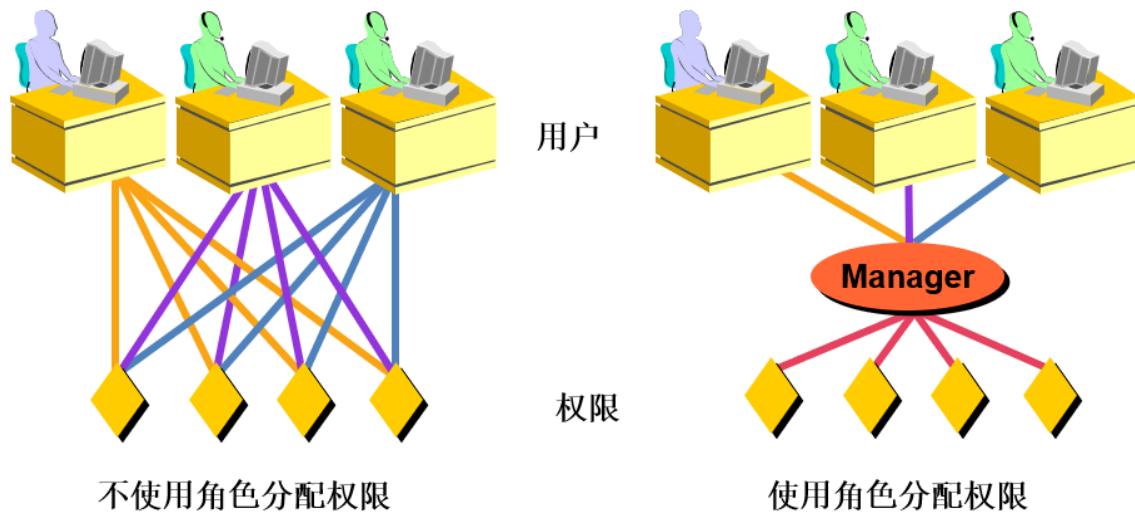
确认权限时，MySQL首先**检查user表**，如果指定的权限没有在user表中被授予，那么MySQL就会继续**检查db表**，db表是下一安全层级，其中的权限限定于数据库层级，在该层级的SELECT权限允许用户查看指定数据库的所有表中的数据；如果在该层级没有找到限定的权限，则MySQL继续**检查tables_priv表**以及**columns_priv表**，如果所有权限表都检查完毕，但还是没有找到允许的权限操作，MySQL将**返回错误信息**，用户请求的操作不能执行，操作失败。

提示：MySQL通过向下层级的顺序（从user表到columns_priv表）检查权限表，但并不是所有的权限都要执行该过程。例如，一个用户登录到MySQL服务器之后只执行对MySQL的管理操作，此时只涉及管理权限，因此MySQL只检查user表。另外，如果请求的权限操作不被允许，MySQL也不会继续检查下一层级的表。

5. 角色管理

5.1 角色的理解

引入角色的目的是**方便管理拥有相同权限的用户**。**恰当的权限设定，可以确保数据的安全性，这是至关重要的。**



5.2 创建角色

创建角色使用 `CREATE ROLE` 语句，语法如下：

```
CREATE ROLE 'role_name'[@'host_name'] [, 'role_name'[@'host_name']]...
```

角色名称的命名规则和用户名类似。如果 `host_name`省略，默认为%，`role_name`不可省略，不可为空。

练习：我们现在需要创建一个经理的角色，就可以用下面的代码：

```
CREATE ROLE 'manager'@'localhost';
```

5.3 给角色赋予权限

创建角色之后，默认这个角色是没有任何权限的，我们需要给角色授权。给角色授权的语法结构是：

```
GRANT privileges ON table_name TO 'role_name'[@'host_name'];
```

上述语句中privileges代表权限的名称，多个权限以逗号隔开。可使用SHOW语句查询权限名称，图11-43列出了部分权限列表。

```
SHOW PRIVILEGES\G;

mysql> show privileges\G
***** 1. row *****
Privilege: Alter
Context: Tables
Comment: To alter the table
***** 2. row *****
Privilege: Alter routine
Context: Functions,Procedures
Comment: To alter or drop stored functions/procedures
***** 3. row *****
Privilege: Create
Context: Databases,Tables,Indexes
Comment: To create new databases and tables
***** 4. row *****
***** 60. row *****
Privilege: INNODB_REDO_LOG_ENABLE
Context: Server Admin
Comment:
***** 61. row *****
Privilege: INNODB_REDO_LOG_ARCHIVE
Context: Server Admin
Comment:
***** 62. row *****
Privilege: REPLICATION_APPLIER
Context: Server Admin
Comment:
62 rows in set (0.00 sec)
```

练习1：我们现在想给经理角色授予商品信息表、盘点表和应付账款表的只读权限，就可以用下面的代码来实现：

```
GRANT SELECT ON demo.settlement TO 'manager';

GRANT SELECT ON demo.goodsmaster TO 'manager';

GRANT SELECT ON demo.invcount TO 'manager';
```

5.4 查看角色的权限

赋予角色权限之后，我们可以通过 SHOW GRANTS 语句，来查看权限是否创建成功了：

```
mysql> SHOW GRANTS FOR 'manager';
+-----+
| Grants for manager@% |
+-----+
| GRANT USAGE ON *.* TO `manager`@`%` |
| GRANT SELECT ON `demo`.`goodsmaster` TO `manager`@`%` |
| GRANT SELECT ON `demo`.`invcount` TO `manager`@`%` |
| GRANT SELECT ON `demo`.`settlement` TO `manager`@`%` |
+-----+
```

只要你创建了一个角色，系统就会自动给你一个“**USAGE**”权限，意思是 **连接登录数据库的权限**。代码的最后三行代表了我们给角色“manager”赋予的权限，也就是对商品信息表、盘点表和应付账款表的只读权限。

结果显示，库管角色拥有商品信息表的只读权限和盘点表的增删改查权限。

5.5 回收角色的权限

角色授权后，可以对角色的权限进行维护，对权限进行添加或撤销。添加权限使用GRANT语句，与角色授权相同。撤销角色或角色权限使用REVOKE语句。

修改了角色的权限，会影响拥有该角色的账户的权限。

撤销角色权限的SQL语法如下：

```
REVOKE privileges ON tablename FROM 'rolename';
```

练习1：撤销school_write角色的权限。（1）使用如下语句撤销school_write角色的权限。

```
REVOKE INSERT, UPDATE, DELETE ON school.* FROM 'school_write';
```

（2）撤销后使用SHOW语句查看school_write对应的权限，语句如下。

```
SHOW GRANTS FOR 'school_write';
```

5.6 删除角色

当我们需要对业务重新整合的时候，可能就需要对之前创建的角色进行清理，删除一些不会再使用的角色。删除角色的操作很简单，你只要掌握语法结构就行了。

```
DROP ROLE role [,role2]...
```

注意，**如果你删除了角色，那么用户也就失去了通过这个角色所获得的所有权限**。

练习：执行如下SQL删除角色school_read。

```
DROP ROLE 'school_read';
```

5.7 给用户赋予角色

角色创建并授权后，要赋给用户并处于 **激活状态** 才能发挥作用。给用户添加角色可使用GRANT语句，语法形式如下：

```
GRANT role [,role2,...] TO user [,user2,...];
```

在上述语句中，role代表角色，user代表用户。可将多个角色同时赋予多个用户，用逗号隔开即可。

练习：给kangshifu用户添加角色school_read权限。（1）使用GRANT语句给kangshifu添加school_read权限，SQL语句如下。

```
GRANT 'school_read' TO 'kangshifu'@'localhost';
```

（2）添加完成后使用SHOW语句查看是否添加成功，SQL语句如下。

```
SHOW GRANTS FOR 'kangshifu'@'localhost';
```

（3）使用kangshifu用户登录，然后查询当前角色，如果角色未激活，结果将显示NONE。SQL语句如下。

```
SELECT CURRENT_ROLE();
```

```
mysql> select current_role();
+-----+
| current_role() |
+-----+
| NONE           |
+-----+
1 row in set (0.00 sec)
```

5.8 激活角色

方式1：使用set default role 命令激活角色

举例：

```
SET DEFAULT ROLE ALL TO 'kangshifu'@'localhost';
```

举例：使用 **SET DEFAULT ROLE** 为下面4个用户默认激活所有已拥有的角色如下：

```
SET DEFAULT ROLE ALL TO
'dev1'@'localhost',
'read_user1'@'localhost',
'read_user2'@'localhost',
'rw_user1'@'localhost';
```

方式2：将activate_all_roles_on_login设置为ON

- 默认情况：

```
mysql> show variables like 'activate_all_roles_on_login';
+-----+-----+
| Variable_name          | Value   |
+-----+-----+
| activate_all_roles_on_login | OFF    |
+-----+-----+
1 row in set (0.00 sec)
```

- 设置：

```
SET GLOBAL activate_all_roles_on_login=ON;
```

这条 SQL 语句的意思是，对 **所有角色永久激活**。运行这条语句之后，用户才真正拥有了赋予角色的所有权限。

5.9 撤销用户的角色

撤销用户角色的SQL语法如下：

```
REVOKE role FROM user;
```

练习：撤销kangshifu用户的school_read角色。 (1) 撤销的SQL语句如下

```
REVOKE 'school_read' FROM 'kangshifu'@'localhost';
```

(2) 撤销后，执行如下查询语句，查看kangshifu用户的角色信息

```
SHOW GRANTS FOR 'kangshifu'@'localhost';
```

执行发现，用户kangshifu之前的school_read角色已被撤销。

5.10 设置强制角色(mandatory role)

方式1：服务启动前设置

```
[mysqld]
mandatory_roles='role1,role2@localhost,r3@%.atguigu.com'
```

方式2：运行时设置

```
SET PERSIST mandatory_roles = 'role1,role2@localhost,r3@%.example.com'; #系统重启后仍然有效
SET GLOBAL mandatory_roles = 'role1,role2@localhost,r3@%.example.com'; #系统重启后失效
```

第04章_逻辑架构

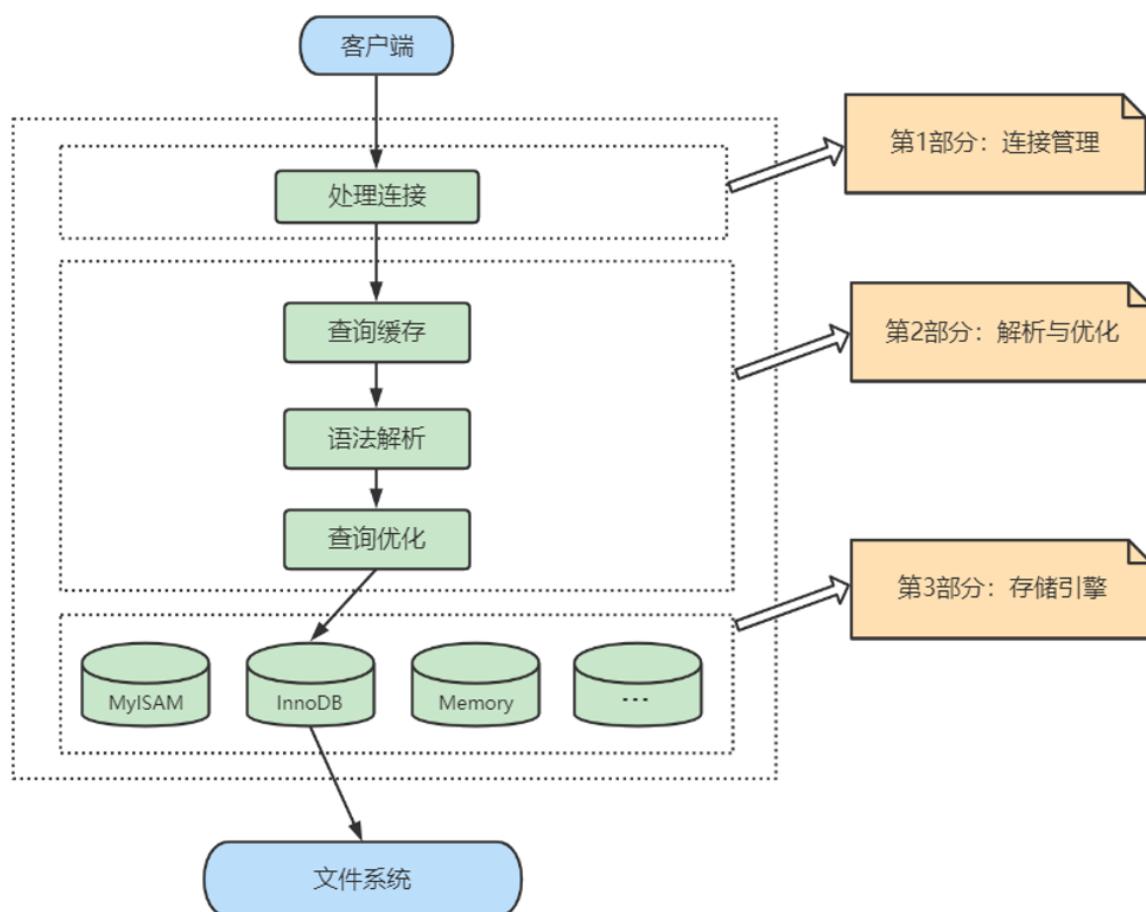
讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

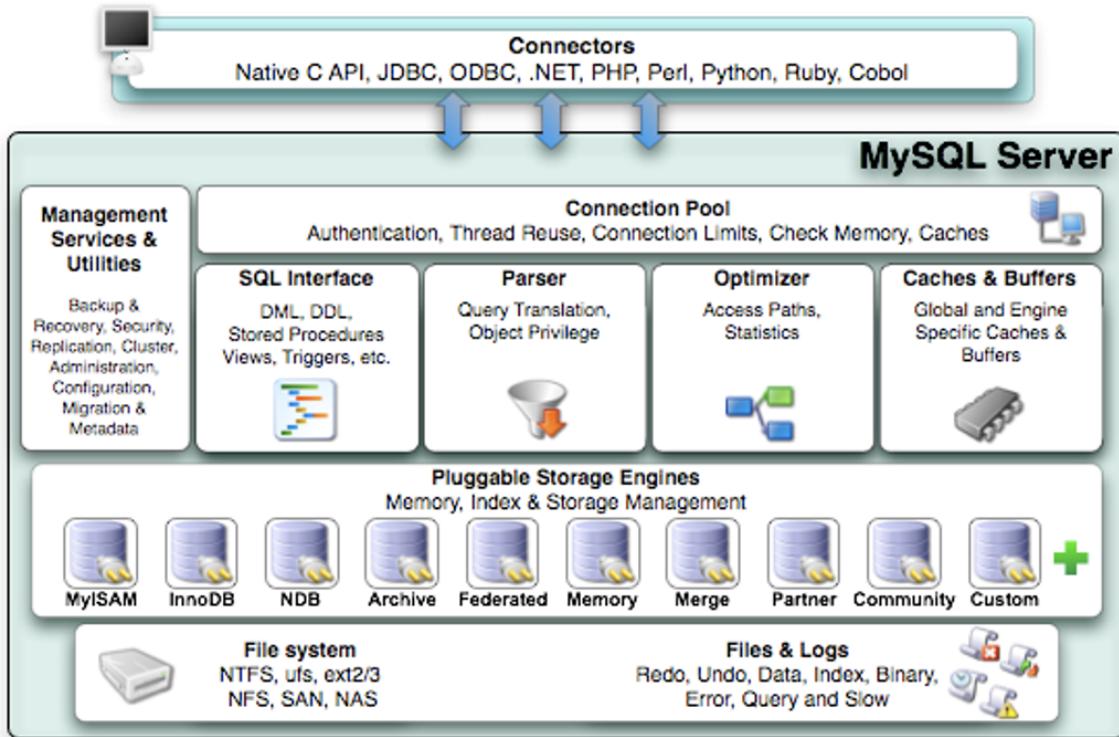
1. 逻辑架构剖析

1.1 服务器处理客户端请求

那服务器进程对客户端进程发送的请求做了什么处理，才能产生最后的处理结果呢？这里以查询请求为例展示：



下面具体展开看一下：



1.2 Connectors

1.3 第1层：连接层

系统（客户端）访问 MySQL 服务器前，做的第一件事就是建立 TCP 连接。

经过三次握手建立连接成功后，MySQL 服务器对 TCP 传输过来的账号密码做身份认证、权限获取。

- 用户名或密码不对，会收到一个 Access denied for user 错误，客户端程序结束执行
- 用户名密码认证通过，会从权限表查出账号拥有的权限与连接关联，之后的权限判断逻辑，都将依赖于此时读到的权限

TCP 连接收到请求后，必须要分配给一个线程专门与这个客户端的交互。所以还会有个线程池，去走后面的流程。每一个连接从线程池中获取线程，省去了创建和销毁线程的开销。

1.4 第2层：服务层

- SQL Interface: SQL接口**
 - 接收用户的SQL命令，并且返回用户需要查询的结果。比如SELECT ... FROM 就是调用SQL Interface
 - MySQL支持DML（数据操作语言）、DDL（数据定义语言）、存储过程、视图、触发器、自定义函数等多种SQL语言接口
- Parser: 解析器**
 - 在解析器中对 SQL 语句进行语法分析、语义分析。将SQL语句分解成数据结构，并将这个结构传递到后续步骤，以后SQL语句的传递和处理就是基于这个结构的。如果在分解构成中遇到错误，那么就说明这个SQL语句是不合理的。
 - 在SQL命令传递到解析器的时候会被解析器验证和解析，并为其创建 语法树，并根据数据字典丰富查询语法树，会 验证该客户端是否具有执行该查询的权限。创建好语法树后，MySQL还会对SQL查询进行语法上的优化，进行查询重写。
- Optimizer: 查询优化器**

- SQL语句在语法解析之后、查询之前会使用查询优化器确定SQL语句的执行路径，生成一个执行计划。
- 这个执行计划表明应该 使用哪些索引 进行查询（全表检索还是使用索引检索），表之间的连接顺序如何，最后会按照执行计划中的步骤调用存储引擎提供的方法来真正的执行查询，并将查询结果返回给用户。
- 它使用“选取-投影-连接”策略进行查询。例如：

```
SELECT id, name FROM student WHERE gender = '女';
```

这个SELECT查询先根据WHERE语句进行 选取，而不是将表全部查询出来以后再进行gender过滤。这个SELECT查询先根据id和name进行属性 投影，而不是将属性全部取出以后再进行过滤，将这两个查询条件 连接 起来生成最终查询结果。

- Caches & Buffers: 查询缓存组件**

- MySQL内部维持着一些Cache和Buffer，比如Query Cache用来缓存一条SELECT语句的执行结果，如果能够在其中找到对应的查询结果，那么就不必再进行查询解析、优化和执行的整个过程了，直接将结果反馈给客户端。
- 这个缓存机制是由一系列小缓存组成的。比如表缓存，记录缓存，key缓存，权限缓存等。
- 这个查询缓存可以在 不同客户端之间共享。
- 从MySQL 5.7.20开始，不推荐使用查询缓存，并在 MySQL 8.0中删除。

小故事：

如果我问你 $9+8\times16-3\times2\times17$ 的值是多少，你可能会用计算器去算一下，最终结果35。如果再问你一遍 $9+8\times16-3\times2\times17$ 的值是多少，你还用再傻呵呵的再算一遍吗？我们刚刚已经算过了，直接说答案就好了。

1.5 第3层：引擎层

插件式存储引擎层（Storage Engines），**真正的负责了MySQL中数据的存储和提取，对物理服务器级别的维护的底层数据执行操作**，服务器通过API与存储引擎进行通信。不同的存储引擎具有的功能不同，这样我们可以根据自己的实际需要进行选取。

MySQL 8.0.25默认支持的存储引擎如下：

Engine	Support	Comment	Transactions	XA	Savepoints
FEDERATED	NO	Federated MySQL storage engine	NULL	NULL	NULL
MEMORY	YES	Hash based, stored in memory, useful for temporary tables	NO	NO	NO
InnoDB	DEFAULT	Supports transactions, row-level locking, and foreign keys	YES	YES	YES
PERFORMANCE_SCHEMA	YES	Performance Schema	NO	NO	NO
MyISAM	YES	MyISAM storage engine	NO	NO	NO
MRG_MYISAM	YES	Collection of identical MyISAM tables	NO	NO	NO
BLACKHOLE	YES	/dev/null storage engine (anything you write to it disappears)	NO	NO	NO
CSV	YES	CSV storage engine	NO	NO	NO
ARCHIVE	YES	Archive storage engine	NO	NO	NO

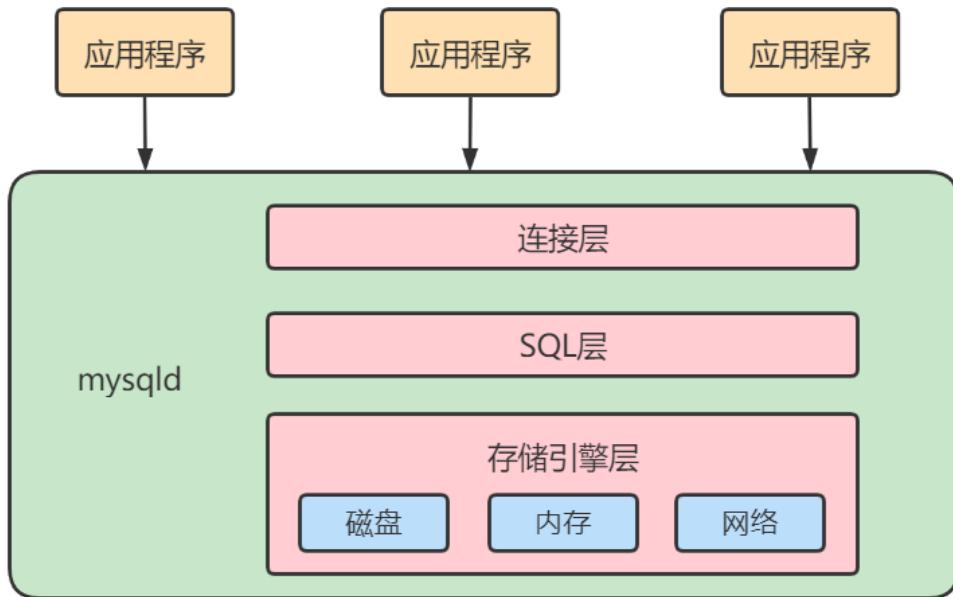
9 rows in set (0.00 sec)

1.6 存储层

所有的数据，数据库、表的定义，表的每一行的内容，索引，都是存在 文件系统 上，以 文件 的方式存在的，并完成与存储引擎的交互。当然有些存储引擎比如InnoDB，也支持不使用文件系统直接管理裸设备，但现代文件系统的实现使得这样做没有必要了。在文件系统之下，可以使用本地磁盘，可以使用DAS、NAS、SAN等各种存储系统。

1.7 小结

MySQL架构图本节开篇所示。下面为了熟悉SQL执行流程方便，我们可以简化如下：

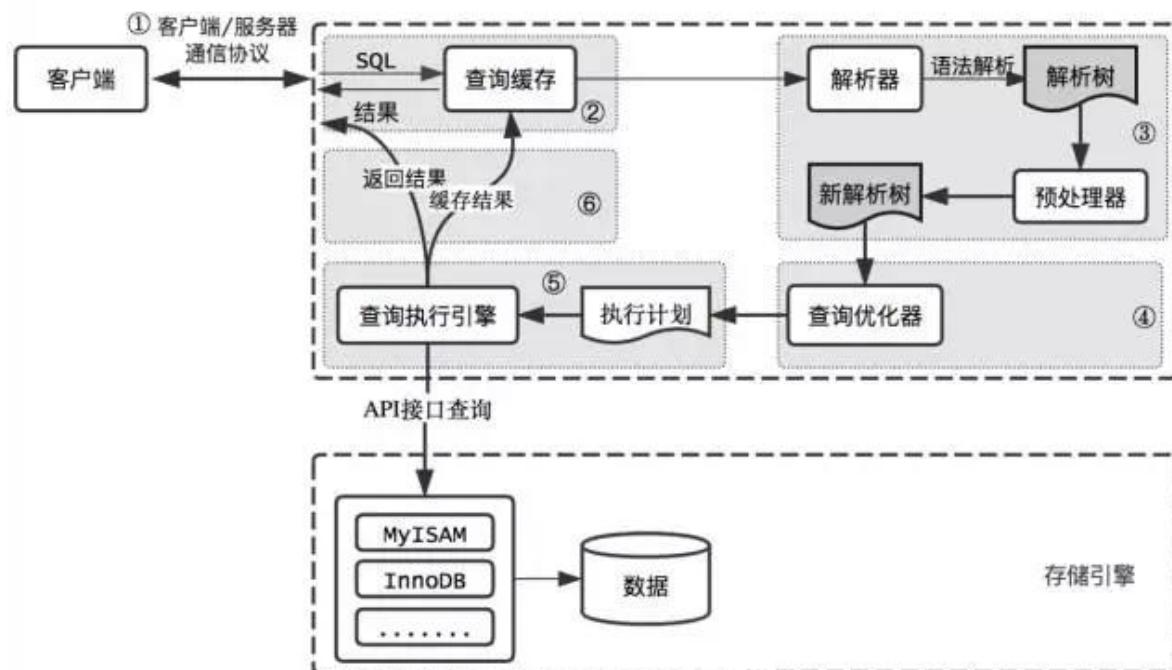


简化为三层结构：

1. 连接层：客户端和服务器端建立连接，客户端发送 SQL 至服务器端；
2. SQL 层（服务层）：对 SQL 语句进行查询处理；与数据库文件的存储方式无关；
3. 存储引擎层：与数据库文件打交道，负责数据的存储和读取。

2. SQL执行流程

2.1 MySQL 中的 SQL 执行流程



MySQL的查询流程：

1. 查询缓存: Server 如果在查询缓存中发现了这条 SQL 语句，就会直接将结果返回给客户端；如果没有，就进入到解析器阶段。需要说明的是，因为查询缓存往往效率不高，所以在 MySQL8.0 之后就抛弃了这个功能。

大多数情况查询缓存就是个鸡肋，为什么呢？

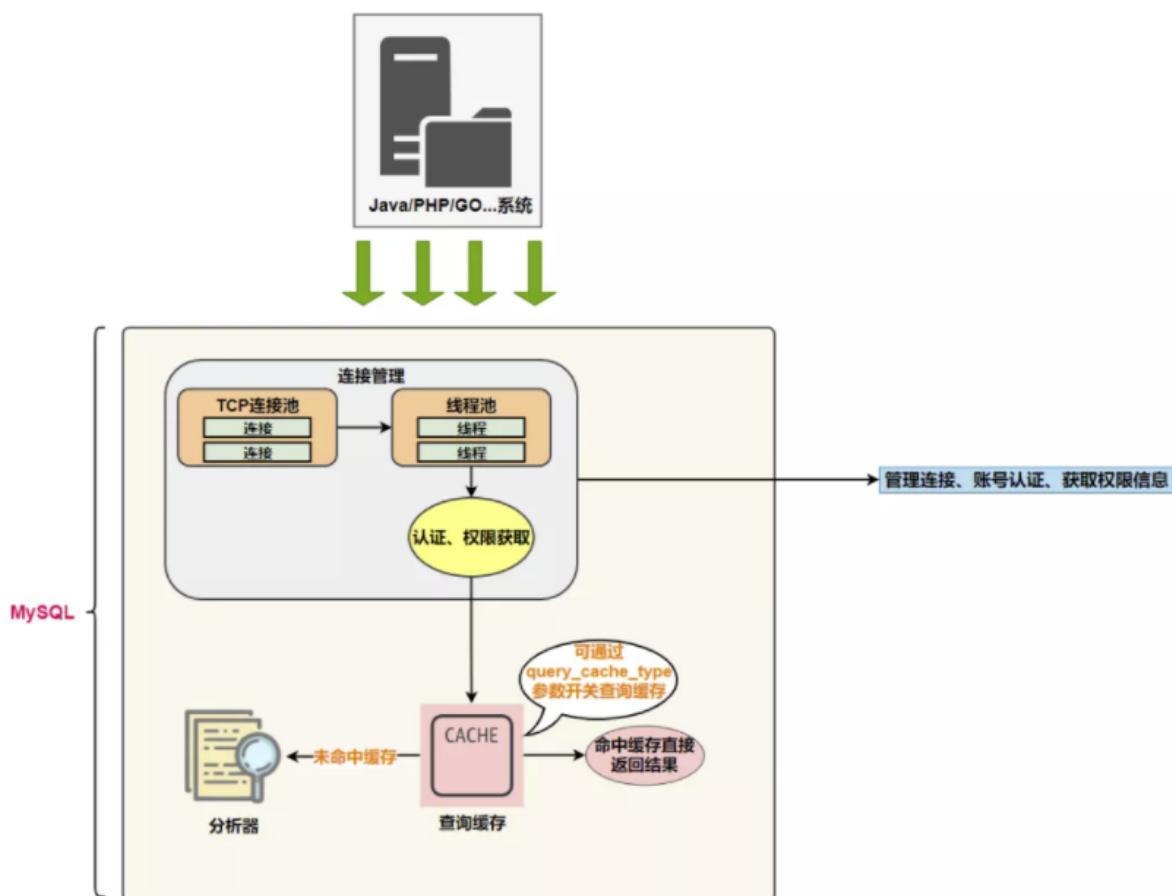
```
SELECT employee_id, last_name FROM employees WHERE employee_id = 101;
```

查询缓存是提前把查询结果缓存起来，这样下次不需要执行就可以直接拿到结果。需要说明的是，在 MySQL 中的查询缓存，不是缓存查询计划，而是查询对应的结果。这就意味着查询匹配的 鲁棒性大大降低，只有 相同的查询操作才会命中查询缓存。两个查询请求在任何字符上的不同（例如：空格、注释、大小写），都会导致缓存不会命中。因此 MySQL 的 查询缓存命中率不高。

同时，如果查询请求中包含某些系统函数、用户自定义变量和函数、一些系统表，如 mysql、information_schema、performance_schema 数据库中的表，那这个请求就不会被缓存。以某些系统函数举例，可能同样的函数的两次调用会产生不一样的结果，比如函数 NOW，每次调用都会产生最新的当前时间，如果在一个查询请求中调用了这个函数，那即使查询请求的文本信息都一样，那不同时间的两次查询也应该得到不同的结果，如果在第一次查询时就缓存了，那第二次查询的时候直接使用第一次查询的结果就是错误的！

此外，既然是缓存，那就有它 缓存失效的时候。MySQL 的缓存系统会监测涉及到的每张表，只要该表的结构或者数据被修改，如对该表使用了 INSERT、UPDATE、DELETE、TRUNCATE TABLE、ALTER TABLE、DROP TABLE 或 DROP DATABASE 语句，那使用该表的所有高速缓存查询都将变为无效并从高速缓存中删除！对于 更新压力大的数据库 来说，查询缓存的命中率会非常低。

2. 解析器: 在解析器中对 SQL 语句进行语法分析、语义分析。

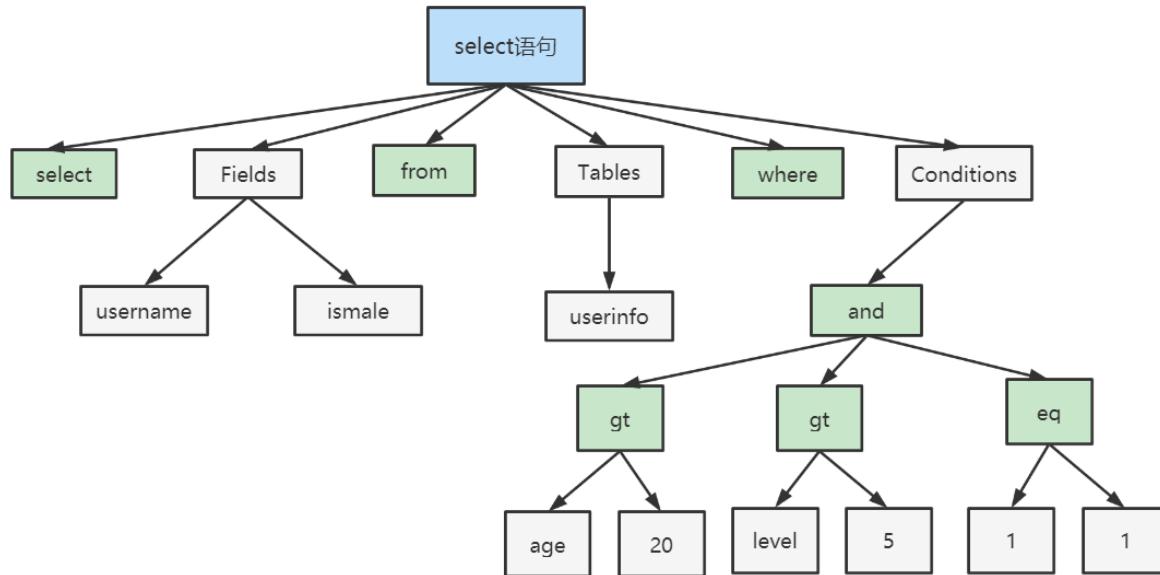


分析器先做“词法分析”。你输入的是由多个字符串和空格组成的一条 SQL 语句，MySQL 需要识别出里面的字符串分别是什么，代表什么。MySQL 从你输入的“select”这个关键字识别出来，这是一个查询语句。它也要把字符串“T”识别成“表名 T”，把字符串“ID”识别成“列 ID”。

接着，要做“[语法分析](#)”。根据词法分析的结果，语法分析器（比如：Bison）会根据语法规则，判断你输入的这个 SQL 语句是否 [满足 MySQL 语法](#)。

```
select department_id,job_id,avg(salary) from employees group by department_id;
```

如果SQL语句正确，则会生成一个这样的语法树：



3. 优化器：在优化器中会确定 SQL 语句的执行路径，比如是根据 [全表检索](#)，还是根据 [索引检索](#) 等。

举例：如下语句是执行两个表的 join：

```
select * from test1 join test2 using(ID)
where test1.name='zhangwei' and test2.name='mysql高级课程';
```

方案1：可以先从表 `test1` 里面取出 `name='zhangwei'` 的记录的 ID 值，再根据 ID 值关联到表 `test2`，再判断 `test2` 里面 `name` 的值是否等于 `'mysql高级课程'`。

方案2：可以先从表 `test2` 里面取出 `name='mysql高级课程'` 的记录的 ID 值，再根据 ID 值关联到 `test1`，再判断 `test1` 里面 `name` 的值是否等于 `zhangwei`。

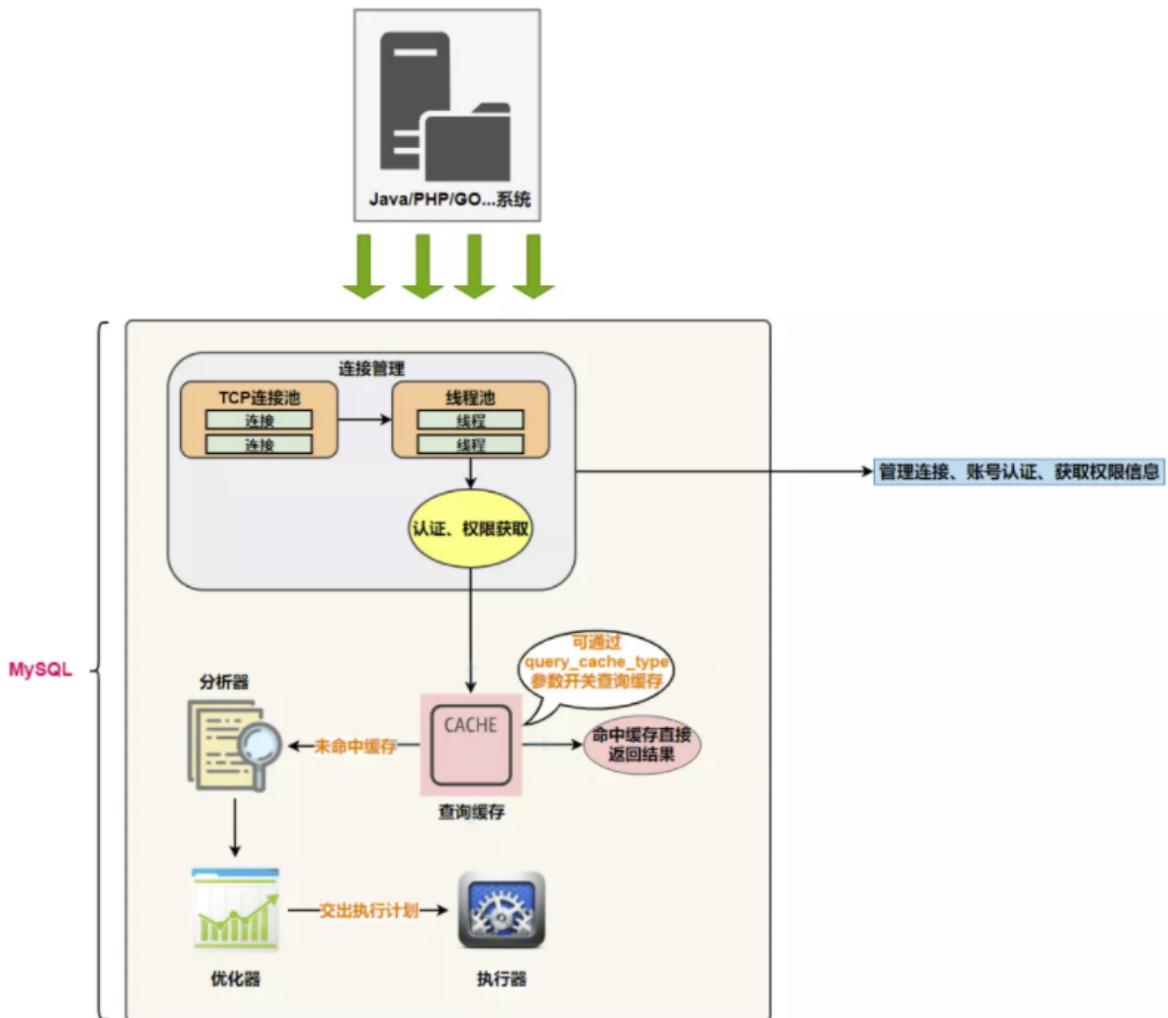
这两种执行方法的逻辑结果是一样的，但是执行的效率会有不同，而优化器的作用就是决定选择使用哪一个方案。优化器阶段完成后，这个语句的执行方案就确定下来了，然后进入执行器阶段。

如果你还有一些疑问，比如优化器是怎么选择索引的，有没有可能选择错等。后面讲到索引我们再谈。

在查询优化器中，可以分为 [逻辑查询](#) 优化阶段和 [物理查询](#) 优化阶段。

4. 执行器：

截止到现在，还没有真正去读写真实的表，仅仅只是产出了一个执行计划。于是就进入了 [执行器阶段](#)。



在执行之前需要判断该用户是否 **具备权限**。如果没有，就会返回权限错误。如果具备权限，就执行 SQL 查询并返回结果。在 MySQL8.0 以下的版本，如果设置了查询缓存，这时会将查询结果进行缓存。

```
select * from test where id=1;
```

比如：表 test 中，ID 字段没有索引，那么执行器的执行流程是这样的：

调用 InnoDB 引擎接口取这个表的第一行，判断 ID 值是不是1，如果不是则跳过，如果是则将这行存在结果集中；调用引擎接口取“下一行”，重复相同的判断逻辑，直到取到这个表的最后一行。

执行器将上述遍历过程中所有满足条件的行组成的记录集作为结果集返回给客户端。

至此，这个语句就执行完成了。对于有索引的表，执行的逻辑也差不多。

SQL 语句在 MySQL 中的流程是： **SQL语句→查询缓存→解析器→优化器→执行器**。



2.2 MySQL8中SQL执行原理

1. 确认profiling是否开启

```
mysql> select @@profiling;  
  
mysql> show variables like 'profiling';
```

```
mysql> select @@profiling;  
+-----+  
| @@profiling |  
+-----+  
|          0 |  
+-----+  
1 row in set, 1 warning (0.00 sec)
```

```
mysql> show variables like 'profiling';  
+-----+-----+  
| Variable_name | Value |  
+-----+-----+  
| profiling     | OFF   |  
+-----+-----+  
1 row in set (0.01 sec)
```

profiling=0 代表关闭，我们需要把 profiling 打开，即设置为 1：

```
mysql> set profiling=1;
```

2. 多次执行相同SQL查询

然后我们执行一个 SQL 查询（你可以执行任何一个 SQL 查询）：

```
mysql> select * from employees;
```

3. 查看profiles

查看当前会话所产生的所有 profiles：

```
mysql> show profiles; # 显示最近的几次查询
```

```
mysql> show profiles;  
+-----+-----+-----+  
| Query_ID | Duration | Query      |  
+-----+-----+-----+  
|    1 | 0.00019625 | select @@profiling  
|    2 | 0.00741450 | show databases  
|    3 | 0.00056150 | SELECT DATABASE()  
|    4 | 0.00056400 | show databases  
|    5 | 0.00317325 | show tables  
|    6 | 0.00163500 | select * from employees  
|    7 | 0.00044000 | select * from employees  
+-----+-----+-----+  
7 rows in set, 1 warning (0.00 sec)
```

4. 查看profile

显示执行计划，查看程序的执行步骤：

```
mysql> show profile;
```

```
mysql> show profile;
+-----+-----+
| Status          | Duration |
+-----+-----+
| starting        | 0.000045 |
| Executing hook on transaction | 0.000004 |
| starting        | 0.000007 |
| checking permissions | 0.000004 |
| Opening tables   | 0.000026 |
| init             | 0.000004 |
| System lock      | 0.000025 |
| optimizing       | 0.000003 |
| statistics       | 0.000012 |
| preparing         | 0.000031 |
| executing        | 0.000178 |
| end              | 0.000003 |
| query end        | 0.000003 |
| waiting for handler commit | 0.000005 |
| closing tables   | 0.000005 |
| freeing items    | 0.000079 |
| cleaning up       | 0.000008 |
+-----+-----+
17 rows in set, 1 warning (0.00 sec)
```

权限检查
打开表
初始化
锁系统
优化查询
准备
执行

当然你也可以查询指定的 Query ID，比如：

```
mysql> show profile for query 7;
```

查询 SQL 的执行时间结果和上面是一样的。

此外，还可以查询更丰富的内容：

```
mysql> show profile cpu,block io for query 6;
```

```
mysql> show profile cpu,block io for query 6;
+-----+-----+-----+-----+-----+-----+
| Status          | Duration | CPU_user | CPU_system | Block_ops_in | Block_ops_out |
+-----+-----+-----+-----+-----+-----+
| starting        | 0.000095 | 0.000004 | 0.000087 | 0           | 0           |
| Executing hook on transaction | 0.000004 | 0.000000 | 0.000003 | 0           | 0           |
| starting        | 0.000027 | 0.000001 | 0.000026 | 0           | 0           |
| checking permissions | 0.000006 | 0.000001 | 0.000006 | 0           | 0           |
| Opening tables   | 0.000056 | 0.000002 | 0.000053 | 0           | 0           |
| init             | 0.000024 | 0.000001 | 0.000023 | 0           | 0           |
| System lock      | 0.000006 | 0.000001 | 0.000006 | 0           | 0           |
| optimizing       | 0.000003 | 0.000000 | 0.000003 | 0           | 0           |
| statistics       | 0.000026 | 0.000001 | 0.000024 | 0           | 0           |
| preparing         | 0.000012 | 0.000001 | 0.000011 | 0           | 0           |
| executing        | 0.001266 | 0.000000 | 0.001267 | 0           | 0           |
| end              | 0.000005 | 0.000000 | 0.000004 | 0           | 0           |
| query end        | 0.000003 | 0.000000 | 0.000002 | 0           | 0           |
| waiting for handler commit | 0.000005 | 0.000000 | 0.000005 | 0           | 0           |
| closing tables   | 0.000006 | 0.000000 | 0.000006 | 0           | 0           |
| freeing items    | 0.000083 | 0.000000 | 0.000084 | 0           | 0           |
| cleaning up       | 0.000010 | 0.000000 | 0.000009 | 0           | 0           |
+-----+-----+-----+-----+-----+-----+
17 rows in set, 1 warning (0.00 sec)
```

继续：

```
mysql> show profile cpu,block io for query 7;
```

```

mysql> show profile cpu,block io for query 7;
+-----+-----+-----+-----+-----+-----+
| Status          | Duration | CPU_user | CPU_system | Block_ops_in | Block_ops_out |
+-----+-----+-----+-----+-----+-----+
| starting        | 0.000045 | 0.000001 | 0.000039 | 0           | 0           |
| Executing hook on transaction | 0.000004 | 0.000001 | 0.000004 | 0           | 0           |
| starting        | 0.000007 | 0.000000 | 0.000006 | 0           | 0           |
| checking permissions | 0.000004 | 0.000000 | 0.000004 | 0           | 0           |
| Opening tables   | 0.000026 | 0.000001 | 0.000025 | 0           | 0           |
| init            | 0.000004 | 0.000001 | 0.000022 | 0           | 0           |
| System lock      | 0.000025 | 0.000001 | 0.000005 | 0           | 0           |
| optimizing       | 0.000003 | 0.000000 | 0.000004 | 0           | 0           |
| statistics       | 0.000012 | 0.000000 | 0.000010 | 0           | 0           |
| preparing         | 0.000031 | 0.000002 | 0.000030 | 0           | 0           |
| executing        | 0.000178 | 0.000008 | 0.000170 | 0           | 0           |
| end              | 0.000003 | 0.000000 | 0.000002 | 0           | 0           |
| query end        | 0.000003 | 0.000000 | 0.000003 | 0           | 0           |
| waiting for handler commit | 0.000005 | 0.000001 | 0.000004 | 0           | 0           |
| closing tables    | 0.000005 | 0.000000 | 0.000005 | 0           | 0           |
| freeing items     | 0.000079 | 0.000004 | 0.000076 | 0           | 0           |
| cleaning up       | 0.000008 | 0.000000 | 0.000007 | 0           | 0           |
+-----+-----+-----+-----+-----+-----+
17 rows in set, 1 warning (0.00 sec)

```

2.3 MySQL5.7中SQL执行原理

上述操作在MySQL5.7中测试，发现前后两次相同的sql语句，执行的查询过程仍然是相同的。不是会使用缓存吗？这里我们需要 **显式开启查询缓存模式**。在MySQL5.7中如下设置：

1. 配置文件中开启查询缓存

在 /etc/my.cnf 中新增一行：

```
query_cache_type=1
```

2. 重启mysql服务

```
systemctl restart mysqld
```

3. 开启查询执行计划

由于重启过服务，需要重新执行如下指令，开启profiling。

```
mysql> set profiling=1;
```

4. 执行语句两次：

```

mysql> select * from locations;

mysql> select * from locations;

```

5. 查看profiles

```

mysql> show profiles;
+-----+-----+-----+
| Query_ID | Duration | Query          |
+-----+-----+-----+
| 1 | 0.00021550 | select * from locations |
| 2 | 0.00005975 | select * from locations |
+-----+-----+-----+
2 rows in set, 1 warning (0.00 sec)

```

6. 查看profile

显示执行计划，查看程序的执行步骤：

```
mysql> show profile for query 1;
```

```
mysql> show profile for query 1;
+-----+-----+
| Status          | Duration |
+-----+-----+
| starting        | 0.000023 |
| Waiting for query cache lock | 0.000002 |
| starting        | 0.000002 |
| checking query cache for query | 0.000026 |
| checking permissions | 0.000005 |
| Opening tables  | 0.000012 |
| init            | 0.000012 |
| System lock     | 0.000006 |
| Waiting for query cache lock | 0.000002 |
| System lock     | 0.000012 |
| optimizing       | 0.000003 |
| statistics       | 0.000007 |
| preparing        | 0.000006 |
| executing        | 0.000002 |
| Sending data    | 0.000060 |
| end              | 0.000003 |
| query end        | 0.000005 |
| closing tables   | 0.000004 |
| freeing items    | 0.000004 |
| Waiting for query cache lock | 0.000002 |
| freeing items    | 0.000006 |
| Waiting for query cache lock | 0.000002 |
| freeing items    | 0.000002 |
| storing result in query cache | 0.000002 |
| cleaning up      | 0.000008 |
+-----+-----+
25 rows in set, 1 warning (0.00 sec)
```

```
mysql> show profile for query 2;
```

```
mysql> show profile for query 2;
+-----+-----+
| Status          | Duration |
+-----+-----+
| starting        | 0.000030 |
| Waiting for query cache lock | 0.000002 |
| starting        | 0.000002 |
| checking query cache for query | 0.000004 |
| checking privileges on cached | 0.000004 |
| checking permissions | 0.000007 |
| sending cached result to client | 0.000008 |
| cleaning up      | 0.000003 |
+-----+-----+
8 rows in set, 1 warning (0.00 sec)
```

结论不言而喻。执行编号2时，比执行编号1时少了很多信息，从截图中可以看出查询语句直接从缓存中获取数据。

2.4 SQL语法顺序

随着Mysql版本的更新换代，其优化器也在不断的升级，优化器会分析不同执行顺序产生的性能消耗不同而动态调整执行顺序。

需求：查询每个部门年龄高于20岁的人数且高于20岁人数不能少于2人，显示人数最多的第一名部门信息

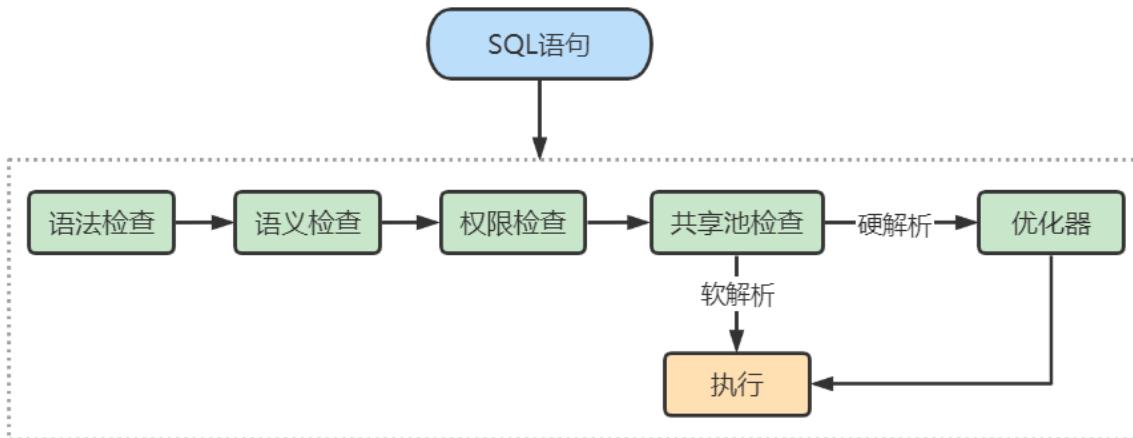
下面是经常出现的查询顺序：

手写	机读
< select_list >	1 FROM <left_table>
FROM	2 ON <join_condition>
< left_table > < join_type >	3 <join_type> JOIN <right_table>
JOIN < right_table > ON < join_condition >	4 WHERE <where_condition>
WHERE	5 GROUP BY <group_by_list>
< where_condition >	6 HAVING <having_condition>
GROUP BY	7 SELECT
< group_by_list >	8 DISTINCT <select_list>
HAVING	9 ORDER BY <order_by_condition>
< having_condition >	10 LIMIT <limit_number>
ORDER BY	
< order_by_condition >	
LIMIT < limit_number >	

2.5 Oracle中的SQL执行流程(了解)

Oracle 中采用了 **共享池** 来判断 SQL 语句是否存在缓存和执行计划，通过这一步骤我们可以知道应该采用硬解析还是软解析。

我们先来看下 SQL 在 Oracle 中的执行过程：



从上面这张图中可以看出，SQL 语句在 Oracle 中经历了以下几个步骤。

1.语法检查： 检查 SQL 拼写是否正确，如果不正确，Oracle 会报语法错误。

2.语义检查： 检查 SQL 中的访问对象是否存在。比如我们在写 SELECT 语句的时候，列名写错了，系统就会提示错误。语法检查和语义检查的作用是保证 SQL 语句没有错误。

3.权限检查： 看用户是否具备访问该数据的权限。

4.共享池检查：共享池（Shared Pool）是一块内存池，**最主要的作用是缓存 SQL 语句和该语句的执行计划。** Oracle 通过检查共享池是否存在 SQL 语句的执行计划，来判断进行软解析，还是硬解析。那软解析和硬解析又该怎么理解呢？

在共享池中，Oracle 首先对 SQL 语句进行 **Hash 运算**，然后根据 Hash 值在库缓存（Library Cache）中查找，如果 **存在 SQL 语句的执行计划**，就直接拿来执行，直接进入“执行器”的环节，这就是 **软解析**。

如果没有找到 SQL 语句和执行计划，Oracle 就需要创建解析树进行解析，生成执行计划，进入“优化器”这个步骤，这就是 **硬解析**。

5. 优化器：优化器中就是要进行硬解析，也就是决定怎么做，比如创建解析树，生成执行计划。
6. 执行器：当有了解析树和执行计划之后，就知道了 SQL 该怎么被执行，这样就可以在执行器中执行语句了。

共享池是 Oracle 中的术语，包括了库缓存，数据字典缓冲区等。我们上面已经讲到了库缓存区，它主要缓存 SQL 语句和执行计划。而 **数据字典缓冲区** 存储的是 Oracle 中的对象定义，比如表、视图、索引等对象。当对 SQL 语句进行解析的时候，如果需要相关的数据，会从数据字典缓冲区中提取。

库缓存 这一个步骤，决定了 SQL 语句是否需要进行硬解析。为了提升 SQL 的执行效率，我们应该尽量避免硬解析，因为在 SQL 的执行过程中，创建解析树，生成执行计划是很消耗资源的。

你可能会问，如何避免硬解析，尽量使用软解析呢？在 Oracle 中，**绑定变量** 是它的一大特色。绑定变量就是在 SQL 语句中使用变量，通过不同的变量取值来改变 SQL 的执行结果。这样做的好处是能 **提升软解析的可能性**，不足之处在于可能会导致生成的执行计划不够优化，因此是否需要绑定变量还需要视情况而定。

举个例子，我们可以使用下面的查询语句：

```
SQL> select * from player where player_id = 10001;
```

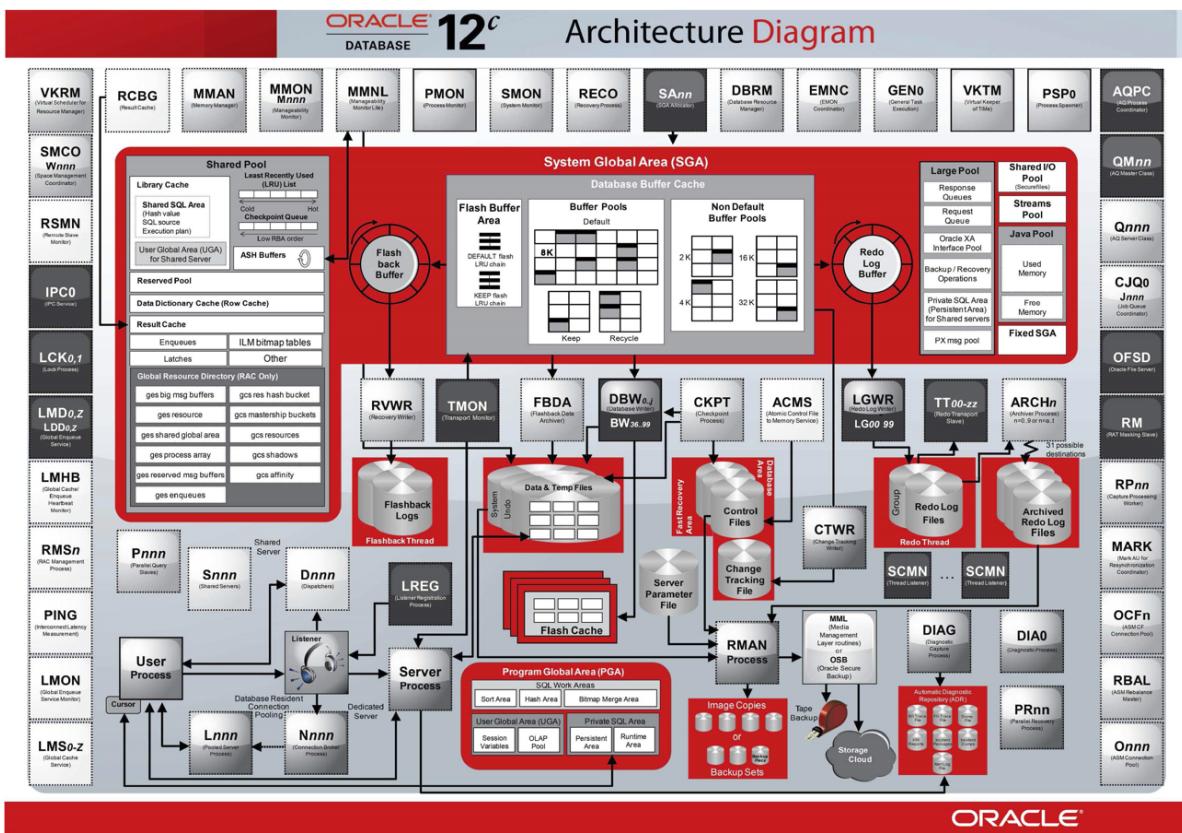
你也可以使用绑定变量，如：

```
SQL> select * from player where player_id = :player_id;
```

这两个查询语句的效率在 Oracle 中是完全不同的。如果你在查询 `player_id = 10001` 之后，还会查询 `10002`、`10003` 之类的数据，那么每一次查询都会创建一个新的查询解析。而第二种方式使用了绑定变量，那么在第一次查询之后，在共享池中就会存在这类查询的执行计划，也就是软解析。

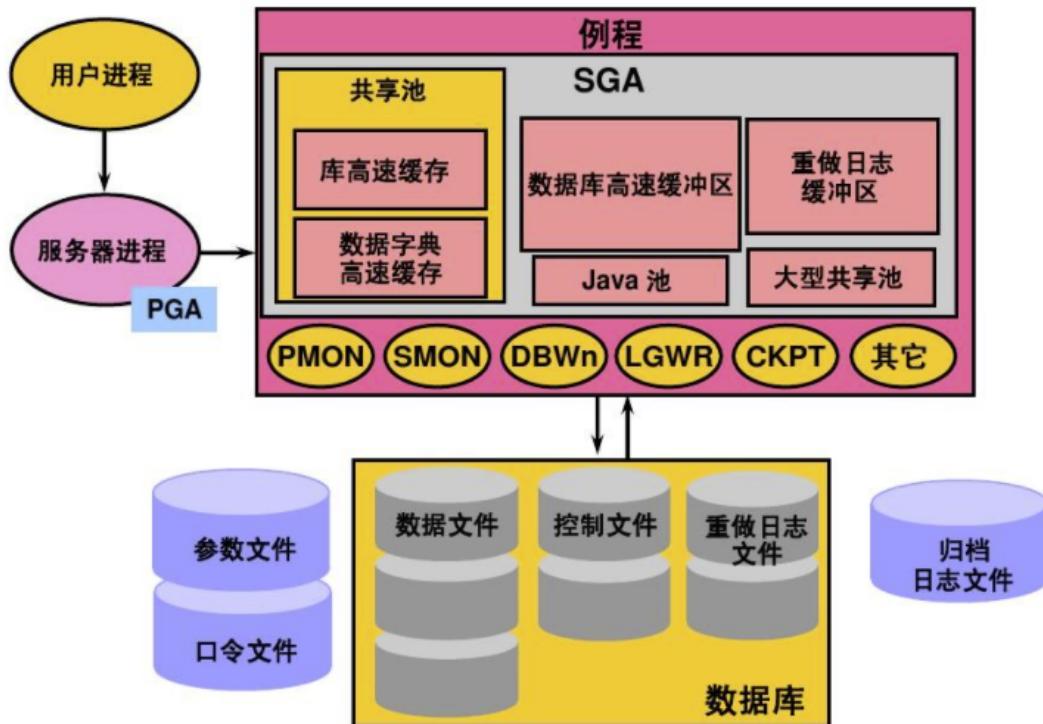
因此，**我们可以通过使用绑定变量来减少硬解析，减少 Oracle 的解析工作量**。但是这种方式也有缺点，使用动态 SQL 的方式，因为参数不同，会导致 SQL 的执行效率不同，同时 SQL 优化也会比较困难。

Oracle的架构图：



简图：

基本组件概览



小结：

Oracle 和 MySQL 在进行 SQL 的查询上面有软件实现层面的差异。Oracle 提出了共享池的概念，通过共享池来判断是进行软解析，还是硬解析。

3. 数据库缓冲池(buffer pool)

InnoDB 存储引擎是以页为单位来管理存储空间的，我们进行的增删改查操作其实本质上都是在访问页面（包括读页面、写页面、创建新页面等操作）。而磁盘 I/O 需要消耗的时间很多，而在内存中进行操作，效率则会高很多，为了能让数据表或者索引中的数据随时被我们所用，DBMS 会申请 占用内存来作为 数据缓冲池，在真正访问页面之前，需要把在磁盘上的页缓存到内存中的 Buffer Pool 之后才可以访问。

这样做的好处是可以让磁盘活动最小化，从而 减少与磁盘直接进行 I/O 的时间。要知道，这种策略对提升 SQL 语句的查询性能来说至关重要。如果索引的数据在缓冲池里，那么访问的成本就会降低很多。

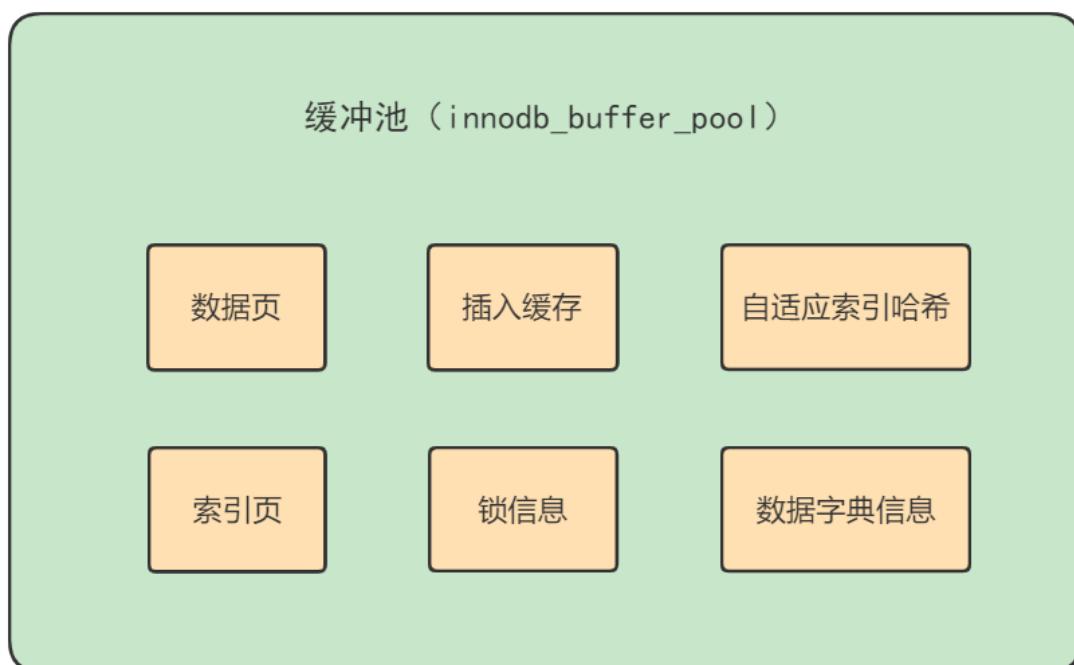
3.1 缓冲池 vs 查询缓存

缓冲池和查询缓存是一个东西吗？不是。

1. 缓冲池 (Buffer Pool)

首先我们需要了解在 InnoDB 存储引擎中，缓冲池都包括了哪些。

在 InnoDB 存储引擎中有一部分数据会放到内存中，缓冲池则占了这部分内存的大部分，它用来存储各种数据的缓存，如下图所示：



从图中，你能看到 InnoDB 缓冲池包括了数据页、索引页、插入缓冲、锁信息、自适应 Hash 和数据字典信息等。

缓冲池的重要性：

缓存原则：

“位置 * 频次”这个原则，可以帮我们对 I/O 访问效率进行优化。

首先，位置决定效率，提供缓冲池就是为了在内存中可以直接访问数据。

其次，频次决定优先级顺序。因为缓冲池的大小是有限的，比如磁盘有 200G，但是内存只有 16G，缓冲池大小只有 1G，就无法将所有数据都加载到缓冲池里，这时就涉及到优先级顺序，会 优先对使用频次高的热数据进行加载。

缓冲池的预读特性：

2. 查询缓存

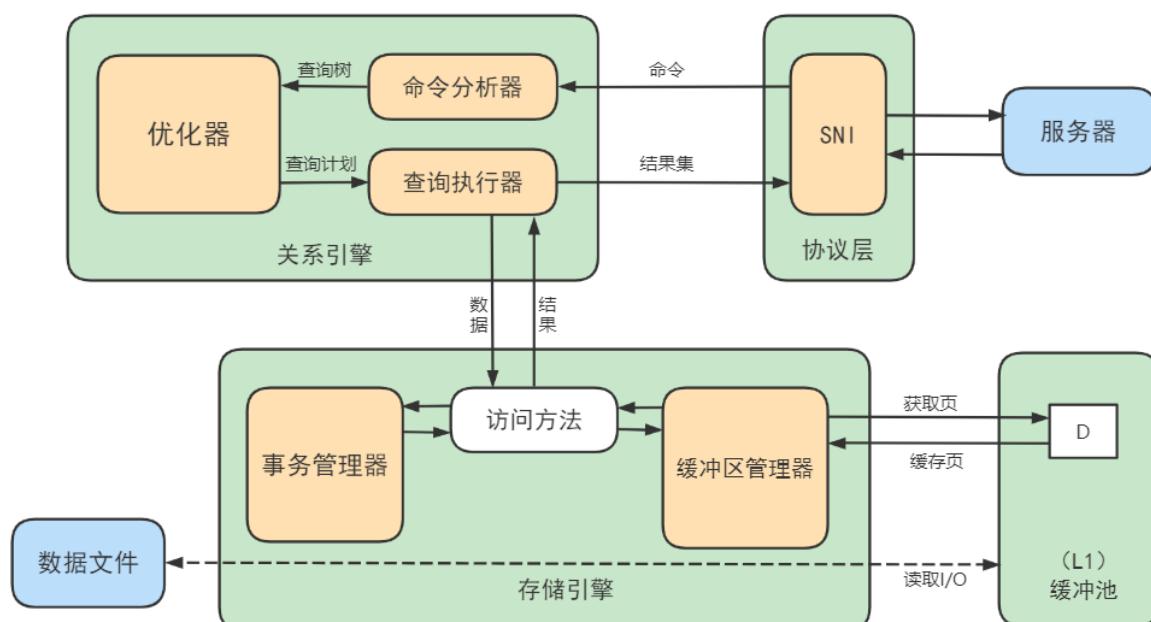
那么什么是查询缓存呢？

查询缓存是提前把 **查询结果缓存** 起来，这样下次不需要执行就可以直接拿到结果。需要说明的是，在 MySQL 中的查询缓存，不是缓存查询计划，而是查询对应的结果。因为命中条件苛刻，而且只要数据表发生变化，查询缓存就会失效，因此命中率低。

3.2 缓冲池如何读取数据

缓冲池管理器会尽量将经常使用的数据保存起来，在数据库进行页面读操作的时候，首先会判断该页面是否在缓冲池中，如果存在就直接读取，如果不存在，就会通过内存或磁盘将页面存放到缓冲池中再进行读取。

缓存在数据库中的结构和作用如下图所示：



如果我们执行 SQL 语句的时候更新了缓存池中的数据，那么这些数据会马上同步到磁盘上吗？

3.3 查看/设置缓冲池的大小

如果你使用的是 InnoDB 存储引擎，可以通过查看 `innodb_buffer_pool_size` 变量来查看缓冲池的大小。命令如下：

```
show variables like 'innodb_buffer_pool_size';
```

```
mysql> show variables like 'innodb_buffer_pool_size';
+-----+-----+
| Variable_name      | Value   |
+-----+-----+
| innodb_buffer_pool_size | 134217728 |
+-----+-----+
1 row in set (0.01 sec)
```

你能看到此时 InnoDB 的缓冲池大小只有 $134217728/1024/1024=128\text{MB}$ 。我们可以修改缓冲池大小，比如改为256MB，方法如下：

```
set global innodb_buffer_pool_size = 268435456;
```

```
mysql> set global innodb_buffer_pool_size = 268435456;
Query OK, 0 rows affected (0.00 sec)
```

或者：

```
[server]
innodb_buffer_pool_size = 268435456
```

然后再来看下修改后的缓冲池大小，此时已成功修改成了 256 MB：

```
mysql> show variables like 'innodb_buffer_pool_size';
+-----+-----+
| Variable_name      | Value   |
+-----+-----+
| innodb_buffer_pool_size | 268435456 |
+-----+-----+
1 row in set (0.01 sec)
```

3.4 多个Buffer Pool实例

```
[server]
innodb_buffer_pool_instances = 2
```

这样就表明我们要创建2个 Buffer Pool 实例。

我们看下如何查看缓冲池的个数，使用命令：

```
show variables like 'innodb_buffer_pool_instances';
```

```
mysql> show variables like 'innodb_buffer_pool_instances';
+-----+-----+
| Variable_name      | Value   |
+-----+-----+
| innodb_buffer_pool_instances | 1       |
+-----+-----+
1 row in set (0.00 sec)
```

那每个 Buffer Pool 实例实际占多少内存空间呢？其实使用这个公式算出来的：

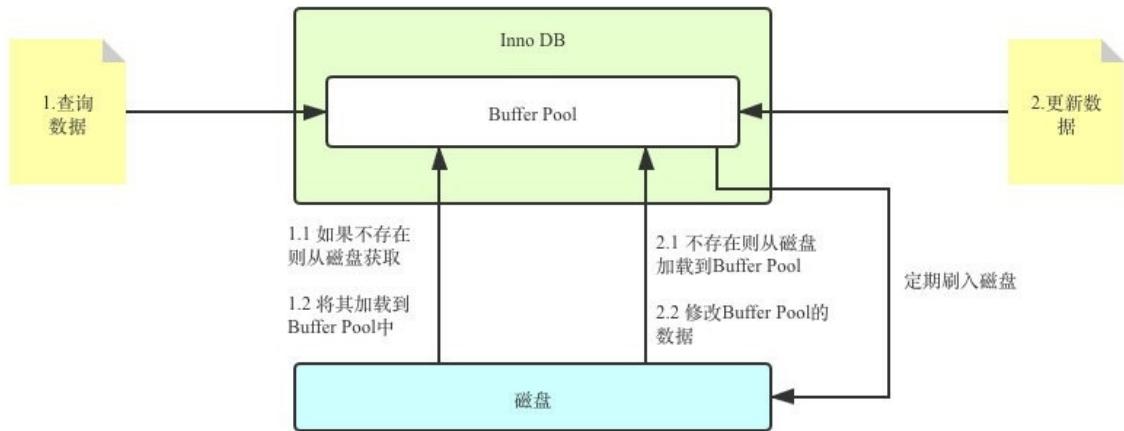
```
innodb_buffer_pool_size/innodb_buffer_pool_instances
```

也就是总共的大小除以实例的个数，结果就是每个 Buffer Pool 实例占用的大小。

3.5 引申问题

Buffer Pool是MySQL内存结构中十分核心的一个组成，你可以先把它想象成一个黑盒子。

黑盒下的更新数据流程



我更新到一半突然发生错误了，想要回滚到更新之前的版本，该怎么办？连数据持久化的保证、事务回滚都做不到还谈什么崩溃恢复？

答案：**Redo Log & Undo Log**

第05章_存储引擎

讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

1. 查看存储引擎

- 查看mysql提供什么存储引擎：

```
show engines;
```

```
mysql> show engines;
+-----+-----+-----+-----+-----+-----+
| Engine | Support | Comment | Transactions | XA | Savepoints |
+-----+-----+-----+-----+-----+-----+
| PERFORMANCE_SCHEMA | YES | Performance Schema | NO | NO | NO |
| MRG_MYISAM | YES | Collection of identical MyISAM tables | NO | NO | NO |
| CSV | YES | CSV storage engine | NO | NO | NO |
| BLACKHOLE | YES | /dev/null storage engine (anything you write to it disappears) | NO | NO | NO |
| MEMORY | YES | Hash based, stored in memory, useful for temporary tables | NO | NO | NO |
| InnoDB | DEFAULT | Supports transactions, row-level locking, and foreign keys | YES | YES | YES |
| ARCHIVE | YES | Archive storage engine | NO | NO | NO |
| MyISAM | YES | MyISAM storage engine | NO | NO | NO |
| FEDERATED | NO | Federated MySQL storage engine | NULL | NULL | NULL |
+-----+-----+-----+-----+-----+-----+
9 rows in set (0.00 sec)
```

```
show engines \G;
```

显式如下：

```
***** 1. row *****
Engine: InnoDB
Support: DEFAULT
Comment: Supports transactions, row-level locking, and foreign keys
Transactions: YES
XA: YES
Savepoints: YES
***** 2. row *****
Engine: MRG_MYISAM
Support: YES
Comment: Collection of identical MyISAM tables
Transactions: NO
XA: NO
Savepoints: NO
***** 3. row *****
Engine: MEMORY
Support: YES
Comment: Hash based, stored in memory, useful for temporary tables
Transactions: NO
XA: NO
Savepoints: NO
***** 4. row *****
Engine: BLACKHOLE
Support: YES
Comment: /dev/null storage engine (anything you write to it disappears)
Transactions: NO
XA: NO
```

```
Savepoints: NO
***** 5. row *****
Engine: MyISAM
Support: YES
Comment: MyISAM storage engine
Transactions: NO
XA: NO
Savepoints: NO
***** 6. row *****
Engine: CSV
Support: YES
Comment: CSV storage engine
Transactions: NO
XA: NO
Savepoints: NO
***** 7. row *****
Engine: ARCHIVE
Support: YES
Comment: Archive storage engine
Transactions: NO
XA: NO
Savepoints: NO
***** 8. row *****
Engine: PERFORMANCE_SCHEMA
Support: YES
Comment: Performance Schema
Transactions: NO
XA: NO
Savepoints: NO
***** 9. row *****
Engine: FEDERATED
Support: NO
Comment: Federated MySQL storage engine
Transactions: NULL
XA: NULL
Savepoints: NULL
```

2. 设置系统默认的存储引擎

- 查看默认的存储引擎:

```
show variables like '%storage_engine%';
#或
SELECT @@default_storage_engine;
```

```
mysql> show variables like '%storage_engine%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| default_storage_engine | InnoDB |
| default_tmp_storage_engine | InnoDB |
| disabled_storage_engines |       |
| internal_tmp_disk_storage_engine | InnoDB |
+-----+
4 rows in set (0.01 sec)
```

- 修改默认的存储引擎

如果在创建表的语句中没有显式指定表的存储引擎的话，那就会默认使用 **InnoDB** 作为表的存储引擎。如果我们想改变表的默认存储引擎的话，可以这样写启动服务器的命令行：

```
SET DEFAULT_STORAGE_ENGINE=MyISAM;
```

或者修改 **my.cnf** 文件：

```
default-storage-engine=MyISAM

# 重启服务
systemctl restart mysqld.service
```

3. 设置表的存储引擎

存储引擎是负责对表中的数据进行提取和写入工作的，我们可以为 **不同的表设置不同的存储引擎**，也就是说不同的表可以有不同的物理存储结构，不同的提取和写入方式。

3.1 创建表时指定存储引擎

我们之前创建表的语句都没有指定表的存储引擎，那就会使用默认的存储引擎 **InnoDB**。如果我们想显式的指定一下表的存储引擎，那可以这么写：

```
CREATE TABLE 表名(
    建表语句;
) ENGINE = 存储引擎名称;
```

3.2 修改表的存储引擎

如果表已经建好了，我们也可以使用下边这个语句来修改表的存储引擎：

```
ALTER TABLE 表名 ENGINE = 存储引擎名称;
```

比如我们修改一下 **engine_demo_table** 表的存储引擎：

```
mysql> ALTER TABLE engine_demo_table ENGINE = InnoDB;
Query OK, 0 rows affected (0.05 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

这时我们再查看一下 **engine_demo_table** 的表结构：

```
mysql> SHOW CREATE TABLE engine_demo_table\G
***** 1. row ****
      Table: engine_demo_table
Create Table: CREATE TABLE `engine_demo_table` (
  `i` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.01 sec)
```

4. 引擎介绍

4.1 InnoDB 引擎：具备外键支持功能的事务存储引擎

- MySQL从3.23.34a开始就包含InnoDB存储引擎。**大于等于5.5之后，默认采用InnoDB引擎。**
- InnoDB是MySQL的**默认事务型引擎**，它被设计用来处理大量的短期(short-lived)事务。可以确保事务的完整提交(Commit)和回滚(Rollback)。
- 除了增加和查询外，还需要更新、删除操作，那么，应优先选择InnoDB存储引擎。
- 除非有非常特别的原因需要使用其他的存储引擎，否则应该优先考虑InnoDB引擎。**
- 数据文件结构：(在《第02章_MySQL数据目录》章节已讲)
 - 表名.frm 存储表结构 (MySQL8.0时，合并在表名.ibd中)
 - 表名.ibd 存储数据和索引
- InnoDB是**为处理巨大数据量的最大性能设计**。
 - 在以前的版本中，字典数据以元数据文件、非事务表等来存储。现在这些元数据文件被删除了。比如：`.frm`, `.par`, `.trn`, `.isl`, `.db.opt` 等都在MySQL8.0中不存在了。
 - 对比MyISAM的存储引擎，**InnoDB写的处理效率差一些**，并且会占用更多的磁盘空间以保存数据和索引。
- MyISAM只缓存索引，不缓存真实数据；InnoDB不仅缓存索引还要缓存真实数据，**对内存要求较高**，而且内存大小对性能有决定性的影响。

4.2 MyISAM 引擎：主要的非事务处理存储引擎

- MyISAM提供了大量的特性，包括全文索引、压缩、空间函数(GIS)等，但MyISAM**不支持事务、行级锁、外键**，有一个毫无疑问的缺陷就是**崩溃后无法安全恢复**。
- 5.5之前默认的存储引擎**
- 优势是访问的**速度快**，对事务完整性没有要求或者以SELECT、INSERT为主的应用
- 针对数据统计有额外的常数存储。故而 count(*) 的查询效率很高
- 数据文件结构：(在《第02章_MySQL数据目录》章节已讲)
 - 表名.frm 存储表结构
 - 表名.MYD 存储数据 (MYData)
 - 表名.MYI 存储索引 (MYIndex)
- 应用场景：只读应用或者以读为主的业务

4.3 Archive 引擎：用于数据存档

- 下表展示了ARCHIVE 存储引擎功能

特征	支持
B树索引	不支持
备份/时间点恢复 (在服务器中实现，而不是在存储引擎中)	支持
集群数据库支持	不支持
聚集索引	不支持
压缩数据	支持
数据缓存	不支持
加密数据 (加密功能在服务器中实现)	支持
外键支持	不支持
全文检索索引	不支持
地理空间数据类型支持	支持
地理空间索引支持	不支持
哈希索引	不支持
索引缓存	不支持
锁粒度	行锁
MVCC	不支持
存储限制	没有任何限制
交易	不支持
更新数据字典的统计信息	支持

4.4 Blackhole 引擎：丢弃写操作，读操作会返回空内容

4.5 CSV 引擎：存储数据时，以逗号分隔各个数据项

使用案例如下

```
mysql> CREATE TABLE test (i INT NOT NULL, c CHAR(10) NOT NULL) ENGINE = CSV;
Query OK, 0 rows affected (0.06 sec)

mysql> INSERT INTO test VALUES(1,'record one'),(2,'record two');
Query OK, 2 rows affected (0.05 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM test;
+---+-----+
| i | c      |
+---+-----+
```

```
| 1 | record one |
| 2 | record two |
+---+-----+
2 rows in set (0.00 sec)
```

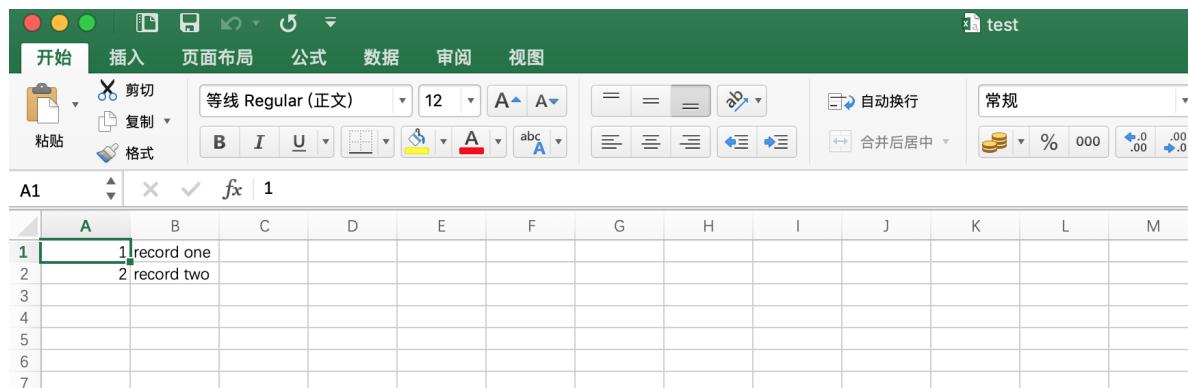
创建CSV表还会创建相应的 元文件，用于 存储表的状态 和 表中存在的行数。此文件的名称与表的名称相同，后缀为 CSM。如图所示

```
-rw-r----- 1 mysql mysql 4342 8月 19 20:29 student_myisam_390.sdi
-rw-r----- 1 mysql mysql 0 8月 19 20:29 student_myisam.MYD
-rw-r----- 1 mysql mysql 1024 8月 19 20:29 student_myisam.MYI
-rw-r----- 1 mysql mysql 2441 8月 19 21:08 test_391.sdi
-rw-r----- 1 mysql mysql 35 8月 19 21:08 test.CSM
-rw-r----- 1 mysql mysql 30 8月 19 21:08 test.CSV
```

如果检查 test.CSV 通过执行上述语句创建的数据库目录中的文件，其内容使用Notepad++打开如下：

```
"1", "record one"
"2", "record two"
```

这种格式可以被 Microsoft Excel 等电子表格应用程序读取，甚至写入。使用Microsoft Excel打开如图所示



4.6 Memory 引擎：置于内存的表

概述：

Memory采用的逻辑介质是 内存， 响应速度很快，但是当mysqld守护进程崩溃的时候 数据会丢失。另外，要求存储的数据是数据长度不变的格式，比如， Blob和Text类型的数据不可用(长度不固定的)。

主要特征：

- Memory同时 支持哈希（HASH）索引 和 B+树索引。
- Memory表至少比MyISAM表要 快一个数量级。
- MEMORY 表的大小是受到限制 的。表的大小主要取决于两个参数，分别是 max_rows 和 max_heap_table_size。其中，max_rows可以在创建表时指定；max_heap_table_size的大小默认认为16MB，可以按需要进行扩大。
- 数据文件与索引文件分开存储。
- 缺点：其数据易丢失，生命周期短。基于这个缺陷，选择MEMORY存储引擎时需要特别小心。

使用Memory存储引擎的场景：

1. 目标数据比较小，而且非常 频繁的进行访问，在内存中存放数据，如果太大的数据会造成 内存溢出。可以通过参数 max_heap_table_size 控制Memory表的大小，限制Memory表的最大的大小。
2. 如果 数据是临时的，而且 必须立即可用 得到，那么就可以放在内存中。

3. 存储在Memory表中的数据如果突然间丢失的话也没有太大的关系。

4.7 Federated 引擎：访问远程表

- Federated引擎是访问其他MySQL服务器的一个代理，尽管该引擎看起来提供了一种很好的跨服务器的灵活性，但也经常带来问题，因此默认是禁用的。

4.8 Merge引擎：管理多个MyISAM表构成的表集合

4.9 NDB引擎：MySQL集群专用存储引擎

也叫做 NDB Cluster 存储引擎，主要用于 MySQL Cluster 分布式集群 环境，类似于 Oracle 的 RAC 集群。

4.10 引擎对比

MySQL中同一个数据库，不同的表可以选择不同的存储引擎。如下表对常用存储引擎做出了对比。

特点	MyISAM	InnoDB	MEMORY	MERGE	NDB
存储限制	有	64TB	有	没有	有
事务安全		支持			
锁机制	表锁，即使操作一条记录也会锁住整个表，不适合高并发的操作	行锁，操作时只锁某一行，不对其它行有影响，适合高并发的操作	表锁	表锁	行锁
B树索引	支持	支持	支持	支持	支持
哈希索引			支持		支持
全文索引	支持				
集群索引		支持			
数据缓存		支持	支持		支持
索引缓存	只缓存索引，不缓存真实数据	不仅缓存索引还要缓存真实数据，对内存要求较高，而且内存大小对性能有决定性的影响	支持	支持	支持
数据可压缩	支持				
空间使用	低	高	N/A	低	低

特点	MyISAM	InnoDB	MEMORY	MERGE	NDB
内存使用	低	高	中等	低	高
批量插入的速度	高	低	高	高	高
支持外键		支持			

其实这些东西大家没必要立即就给记住，列出来的目的就是想让大家明白不同的存储引擎支持不同的功能。

其实我们最常用的就是 InnoDB 和 MyISAM，有时会提一下 Memory。其中 InnoDB 是 MySQL 默认的存储引擎。

5. MyISAM和InnoDB

很多人对 InnoDB 和 MyISAM 的取舍存在疑问，到底选择哪个比较好呢？

MySQL5.5之前的默认存储引擎是MyISAM，5.5之后改为了InnoDB。

对比项	MyISAM	InnoDB
外键	不支持	支持
事务	不支持	支持
行表锁	表锁，即使操作一条记录也会锁住整个表，不适合高并发的操作	行锁，操作时只锁某一行，不对其它行有影响，适合高并发的操作
缓存	只缓存索引，不缓存真实数据	不仅缓存索引还要缓存真实数据，对内存要求较高，而且内存大小对性能有决定性的影响
自带系统表使用	Y	N
关注点	性能：节省资源、消耗少、简单业务	事务：并发写、事务、更大资源
默认安装	Y	Y
默认使用	N	Y

6. 阿里巴巴、淘宝用哪个

产品	价格	目标	主要功能	是否可投入生产？
Percona Server	免费	提供 XtraDB 存储引擎的包装器和其他分析工具	XtraDB	是
MariaDB	免费	扩展 MySQL 以包含 XtraDB 和其他性能改进	XtraDB	是
Drizzle	免费	提供比 MySQL 更强大的可扩展性和性能改进	高可用性	是

- **Percona** 为 MySQL 数据库服务器进行了改进，在功能和性能上较 MySQL 有很显著的提升。
- 该版本提升了在高负载情况下的 InnoDB 的性能、为 DBA 提供一些非常有用的性能诊断工具；另外有更多的参数和命令来控制服务器行为。
- 该公司新建了一款存储引擎叫 **Xtradb** 完全可以替代 **Innodb**，并且在性能和并发上做得更好
- 阿里巴巴大部分mysql数据库其实使用的percona的原型加以修改。

课外补充：

1、InnoDB表的优势

InnoDB存储引擎在实际应用中拥有诸多优势，比如操作便利、提高了数据库的性能、维护成本低等。如果由于硬件或软件的原因导致服务器崩溃，那么在重启服务器之后不需要进行额外的操作。InnoDB崩溃恢复功能自动将之前提交的内容定型，然后撤销没有提交的进程，重启之后继续从崩溃点开始执行。

InnoDB存储引擎在主内存中维护缓冲池，高频率使用的数据将在内存中直接被处理。这种缓存方式应用于多种信息，加速了处理进程。

在专用服务器上，物理内存中高达80%的部分被应用于缓冲池。如果需要将数据插入不同的表中，可以设置外键加强数据的完整性。更新或者删除数据，关联数据将会被自动更新或删除。如果试图将数据插入从表，但在主表中没有对应的数据，插入的数据将被自动移除。如果磁盘或内存中的数据出现崩溃，在使用脏数据之前，校验和机制会发出警告。当每个表的主键都设置合理时，与这些列有关的操作会被自动优化。插入、更新和删除操作通过做改变缓冲自动机制进行优化。**InnoDB不仅支持当前读写，也会缓冲改变的数据到数据流磁盘。**

InnoDB的性能优势不只存在于长时运行查询的大型表。在同一列多次被查询时，自适应哈希索引会提高查询的速度。使用InnoDB可以压缩表和相关的索引，可以在不影响性能和可用性的情况下创建或删除索引。对于大型文本和BLOB数据，使用动态行形式，这种存储布局更高效。通过查询INFORMATION_SCHEMA库中的表可以监控存储引擎的内部工作。在同一个语句中，InnoDB表可以与其他存储引擎表混用。即使有些操作系统限制文件大小为2GB，InnoDB仍然可以处理。**当处理大数据量时，InnoDB兼顾CPU，以达到最大性能。**

2. InnoDB和ACID模型

ACID模型是一系列数据库设计规则，这些规则着重强调可靠性，而可靠性对于商业数据和任务关键型应用非常重要。MySQL包含类似InnoDB存储引擎的组件，与ACID模型紧密相连，这样出现意外时，数据不会崩溃，结果不会失真。如果依赖ACID模型，可以不使用一致性检查和崩溃恢复机制。如果拥有额外的软件保护，极可靠的硬件或者应用可以容忍一小部分的数据丢失和不一致，可以将MySQL设置调整为只依赖部分ACID特性，以达到更高的性能。下面讲解InnoDB存储引擎与ACID模型相同作用的四个方面。

1. 原子方面 ACID的原子方面主要涉及InnoDB事务，与MySQL相关的特性主要包括：

- 自动提交设置。
- COMMIT语句。
- ROLLBACK语句。
- 操作INFORMATION_SCHEMA库中的表数据。

2. 一致性方面 ACID模型的一致性主要涉及保护数据不崩溃的内部InnoDB处理过程，与MySQL相关的特性主要包括：

- InnoDB双写缓存。
- InnoDB崩溃恢复。

3. 隔离方面 隔离是应用于事务的级别，与MySQL相关的特性主要包括：

- 自动提交设置。
- SET ISOLATION LEVEL语句。
- InnoDB锁的低级别信息。

4. 耐久性方面 ACID模型的耐久性主要涉及与硬件配置相互影响的MySQL软件特性。由于硬件复杂多样化，耐久性方面没有具体的规则可循。与MySQL相关的特性有：

- InnoDB双写缓存，通过innodb_doublewrite配置项配置。
- 配置项innodb_flush_log_at_trx_commit。
- 配置项sync_binlog。
- 配置项innodb_file_per_table。
- 存储设备的写入缓存。
- 存储设备的备用电池缓存。
- 运行MySQL的操作系统。
- 持续的电力供应。
- 备份策略。
- 对分布式或托管的应用，最主要的是在于硬件设备的地点以及网络情况。

3、InnoDB架构

1. 缓冲池 缓冲池是主内存中的一部分空间，用来缓存已使用的表和索引数据。缓冲池使得经常被使用的数据能够直接在内存中获得，从而提高速度。

2. 更改缓存 更改缓存是一个特殊的数据结构，当受影响的索引页不在缓存中时，更改缓存会缓存辅助索引页的更改。索引页被其他读取操作时会加载到缓存池，缓存的更改内容就会被合并。不同于集群索引，辅助索引并非独一无二的。当系统大部分闲置时，清除操作会定期运行，将更新的索引页刷入磁盘。更新缓存合并期间，可能会大大降低查询的性能。在内存中，更新缓存占用一部分InnoDB缓冲池。在磁盘中，更新缓存是系统表空间的一部分。更新缓存的数据类型由innodb_change_buffering配置项管理。

3. 自适应哈希索引 自适应哈希索引将负载和足够的内存结合起来，使得InnoDB像内存数据库一样运行，不需要降低事务上的性能或可靠性。这个特性通过innodb_adaptive_hash_index选项配置，或者通过--skip-innodb_adaptive_hash_index命令行在服务启动时关闭。

4. 重做日志缓存 重做日志缓存存放要放入重做日志的数据。重做日志缓存大小通过innodb_log_buffer_size配置项配置。重做日志缓存会定期地将日志文件刷入磁盘。大型的重做日志缓存使得大型事务能够正常运行而不需要写入磁盘。

5. 系统表空间 系统表空间包括InnoDB数据字典、双写缓存、更新缓存和撤销日志，同时也包括表和索引数据。多表共享，系统表空间被视为共享表空间。

6. 双写缓存 双写缓存位于系统表空间中，用于写入从缓存池刷新的数据页。只有在刷新并写入双写缓存后，InnoDB才会将数据页写入合适的位置。

7. 撤销日志 撤销日志是一系列与事务相关的撤销记录的集合，包含如何撤销事务最近的更改。如果其他事务要查询原始数据，可以从撤销日志记录中追溯未更改的数据。撤销日志存在于撤销日志片段中，这些片段包含于回滚片段中。

8. 每个表一个文件的表空间 每个表一个文件的表空间是指每个单独的表空间创建在自身的数据文件中，而不是系统表空间中。这个功能通过innodb_file_per_table配置项开启。每个表空间由一个单独的.ibd数据文件代表，该文件默认被创建在数据库目录中。

9. 通用表空间 使用CREATE TABLESPACE语法创建共享的InnoDB表空间。通用表空间可以创建在MySQL数据目录之外能够管理多个表并支持所有行格式的表。

10. 撤销表空间 撤销表空间由一个或多个包含撤销日志的文件组成。撤销表空间的数量由innodb_undo_tablespaces配置项配置。

11. 临时表空间 用户创建的临时表空间和基于磁盘的内部临时表都创建于临时表空间。innodb_temp_data_file_path配置项定义了相关的路径、名称、大小和属性。如果该值为空，默认会在innodb_data_home_dir变量指定的目录下创建一个自动扩展的数据文件。

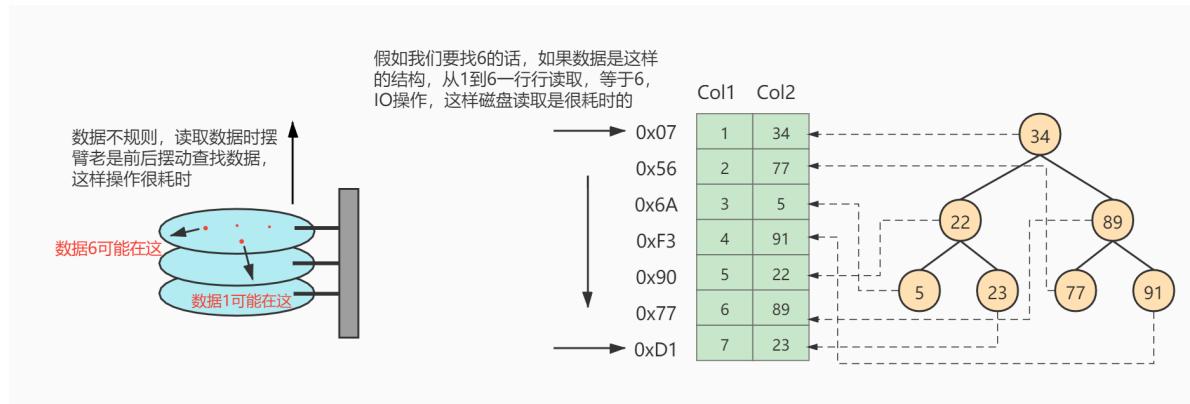
12. 重做日志 重做日志是基于磁盘的数据结构，在崩溃恢复期间使用，用来纠正数据。正常操作期间，重做日志会将请求数据进行编码，这些请求会改变InnoDB表数据。遇到意外崩溃后，未完成的更改会自动在初始化期间重新进行。

第06章_索引的数据结构

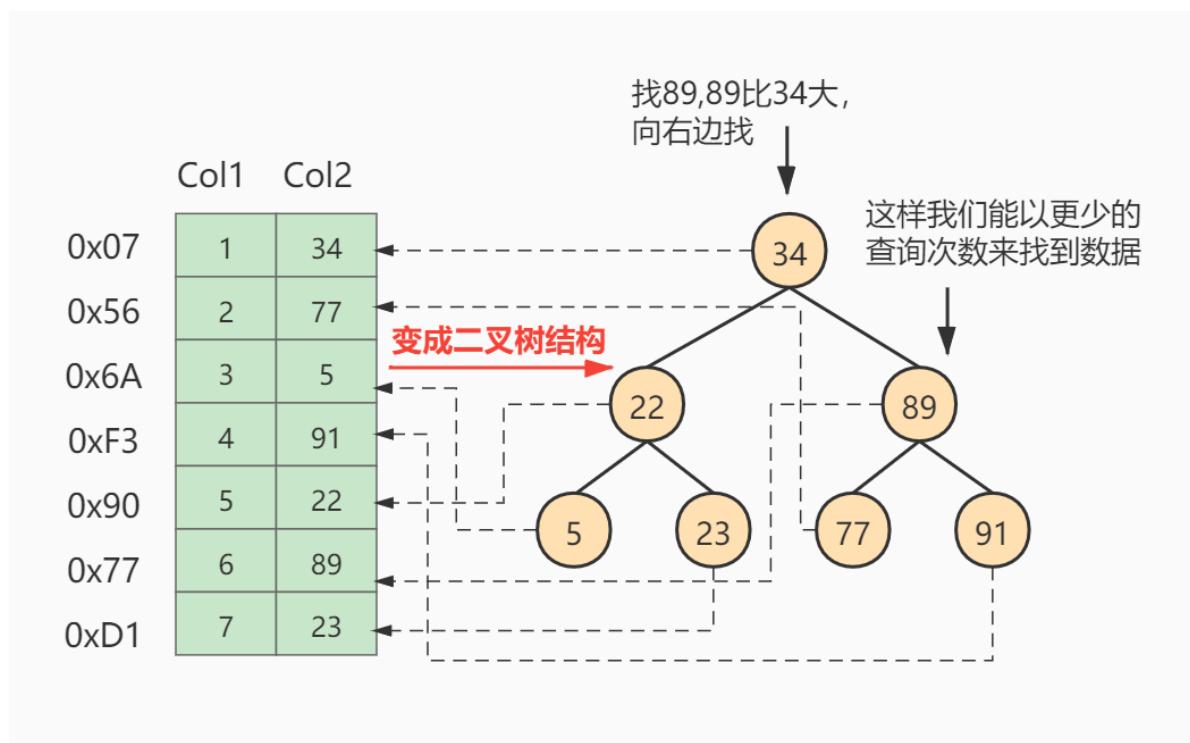
讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

1. 为什么使用索引



假如给数据使用 **二叉树** 这样的数据结构进行存储，如下图所示



2. 索引及其优缺点

2.1 索引概述

MySQL官方对索引的定义为： **索引（Index）是帮助MySQL高效获取数据的数据结构。**

索引的本质：索引是数据结构。你可以简单理解为“排好序的快速查找数据结构”，满足特定查找算法。这些数据结构以某种方式指向数据，这样就可以在这些数据结构的基础上实现 **高级查找算法**。

2.2 优点

(1) 类似大学图书馆建书目索引，提高数据检索的效率，降低 **数据库的IO成本**，这也是创建索引最主要的原因。 (2) 通过创建唯一索引，可以保证数据库表中每一行 **数据的唯一性**。 (3) 在实现数据的参考完整性方面，可以 **加速表和表之间的连接**。换句话说，对于有依赖关系的子表和父表联合查询时，可以提高查询速度。 (4) 在使用分组和排序子句进行数据查询时，可以显著 **减少查询中分组和排序的时间**，降低了CPU的消耗。

2.3 缺点

增加索引也有许多不利的方面，主要表现在如下几个方面： (1) 创建索引和维护索引要 **耗费时间**，并且随着数据量的增加，所耗费的时间也会增加。 (2) 索引需要占 **磁盘空间**，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，**存储在磁盘上**，如果有大量的索引，索引文件就可能比数据文件更快达到最大文件尺寸。 (3) 虽然索引大大提高了查询速度，同时却会 **降低更新表的速度**。当对表中的数据进行增加、删除和修改的时候，索引也要动态地维护，这样就降低了数据的维护速度。

因此，选择使用索引时，需要综合考虑索引的优点和缺点。

3. InnoDB中索引的推演

3.1 索引之前的查找

先来看一个精确匹配的例子：

```
SELECT [列名列表] FROM 表名 WHERE 列名 = xxx;
```

1. 在一个页中的查找

2. 在很多页中查找

在没有索引的情况下，不论是根据主键列或者其他列的值进行查找，由于我们并不能快速的定位到记录所在的页，所以只能 **从第一个页** 沿着 **双向链表** 一直往下找，在每一个页中根据我们上面的查找方式去查找指定的记录。因为要遍历所有的数据页，所以这种方式显然是 **超级耗时** 的。如果一个表有一亿条记录呢？此时 **索引** 应运而生。

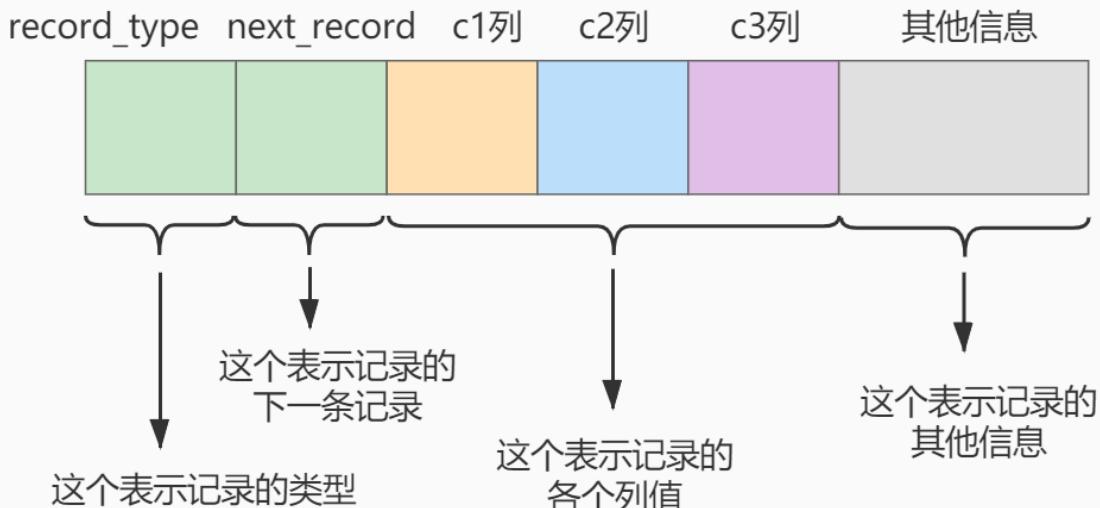


3.2 设计索引

建一个表:

```
mysql> CREATE TABLE index_demo(
->     c1 INT,
->     c2 INT,
->     c3 CHAR(1),
->     PRIMARY KEY(c1)
-> ) ROW_FORMAT = Compact;
```

这个新建的 `index_demo` 表中有2个INT类型的列，1个CHAR(1)类型的列，而且我们规定了c1列为主键，这个表使用 `Compact` 行格式来实际存储记录的。这里我们简化了`index_demo`表的行格式示意图：

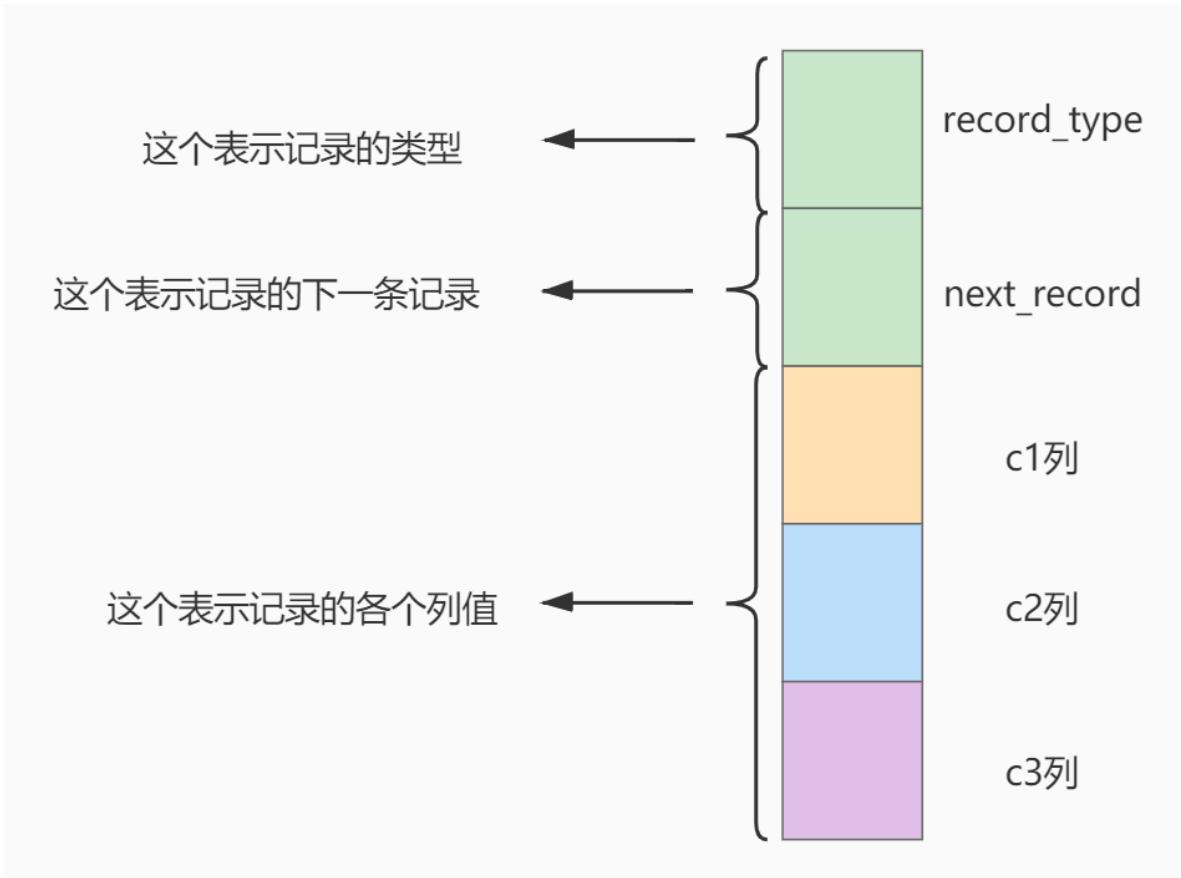


我们只在示意图里展示记录的这几个部分：

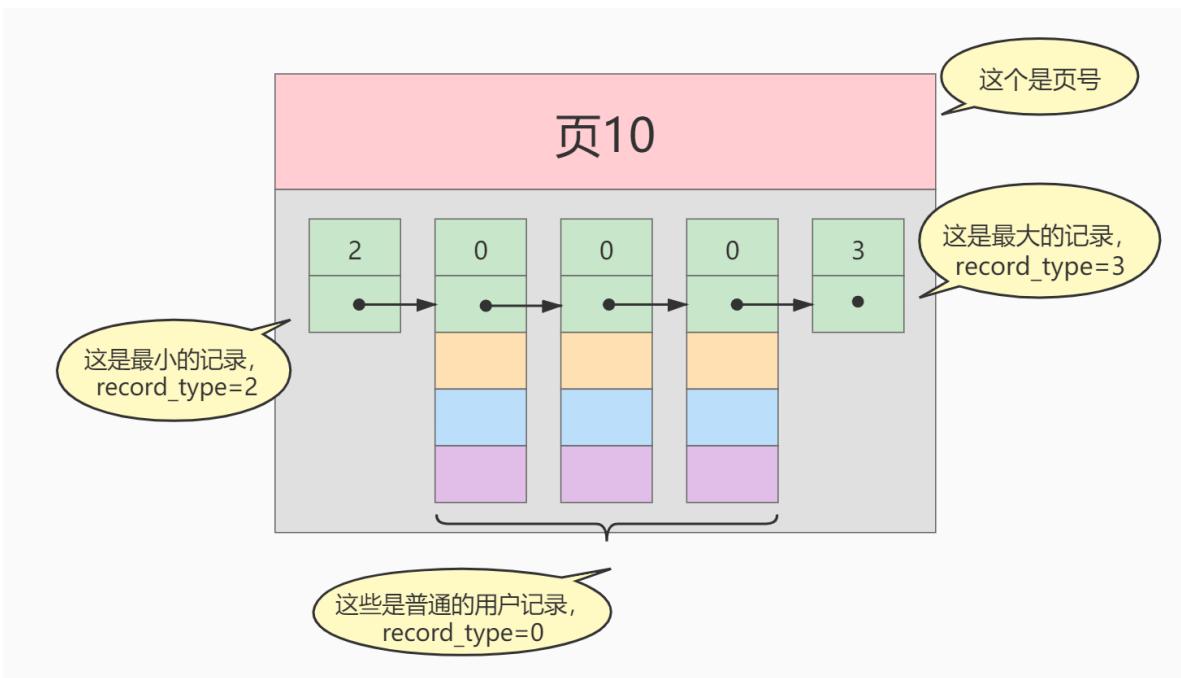
- `record_type`：记录头信息的一项属性，表示记录的类型，0 表示普通记录、2 表示最小记录、3 表示最大记录、1 暂时还没用过，下面讲。

- `next_record`：记录头信息的一项属性，表示下一条地址相对于本条记录的地址偏移量，我们用箭头来表明下一条记录是谁。
- `各个列的值`：这里只记录在 `index_demo` 表中的三个列，分别是 `c1`、`c2` 和 `c3`。
- `其他信息`：除了上述3种信息以外的所有信息，包括其他隐藏列的值以及记录的额外信息。

将记录格式示意图的其他信息项暂时去掉并把它竖起来的效果就是这样：



把一些记录放到页里的示意图就是：



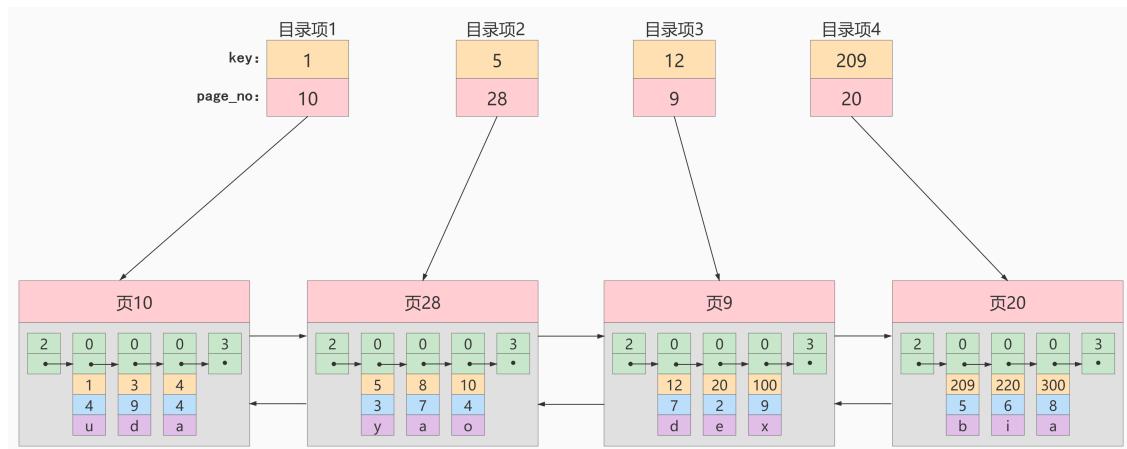
1. 一个简单的索引设计方案

我们在根据某个搜索条件查找一些记录时为什么要遍历所有的数据页呢？因为各个页中的记录并没有规律，我们并不知道我们的搜索条件匹配哪些页中的记录，所以不得不依次遍历所有的数据页。所以如果我们要想快速的定位到需要查找的记录在哪些数据页中该咋办？我们可以为快速定位记录所在的数据页而建立一个目录，建这个目录必须完成下边这些事：

- 下一个数据页中用户记录的主键值必须大于上一个页中用户记录的主键值。

- 给所有的页建立一个目录项。

所以我们为上边几个页做好的目录就像这样子：



以 **页28** 为例，它对应 **目录项2**，这个目录项中包含着该页的页号 **28** 以及该页中用户记录的最小主键值 **5**。我们只需要把几个目录项在物理存储器上连续存储（比如：数组），就可以实现根据主键值快速查找某条记录的功能了。比如：查找主键值为 **20** 的记录，具体查找过程分两步：

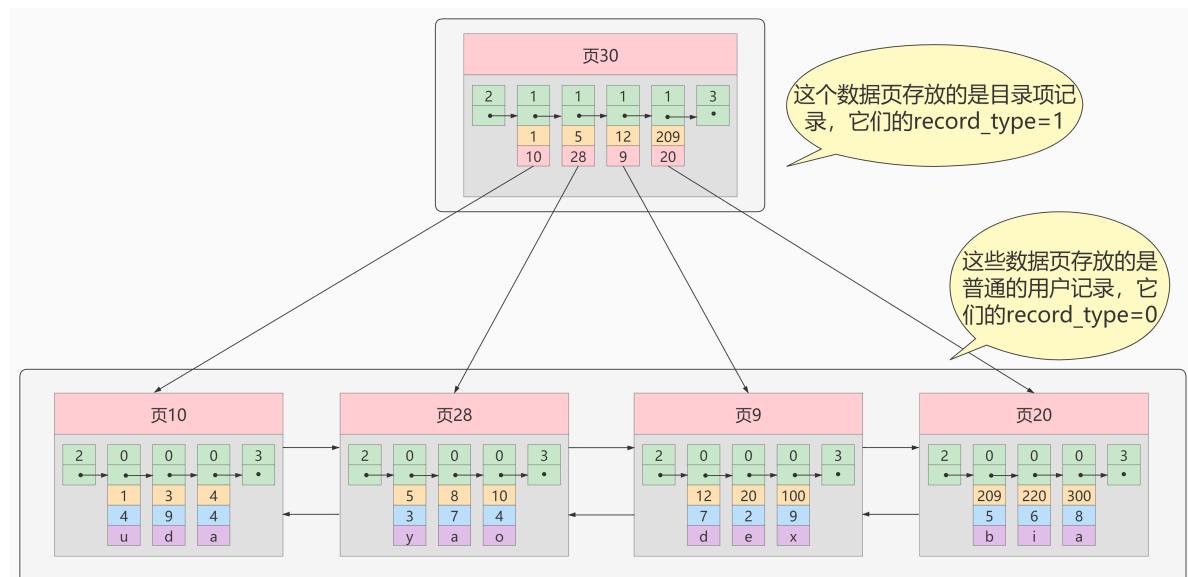
- 先从目录项中根据 **二分法** 快速确定出主键值为 **20** 的记录在 **目录项3** 中（因为 **12 < 20 < 209**），它对应的页是 **页9**。
- 再根据前边说的在页中查找记录的方式去 **页9** 中定位具体的记录。

至此，针对数据页做的简易目录就搞定了。这个目录有一个别名，称为 **索引**。

2. InnoDB中的索引方案

① 迭代1次：目录项纪录的页

我们把前边使用到的目录项放到数据页中的样子就是这样：



从图中可以看出来，我们新分配了一个编号为30的页来专门存储目录项记录。这里再次强调 目录项记录 和普通的 用户记录 的不同点：

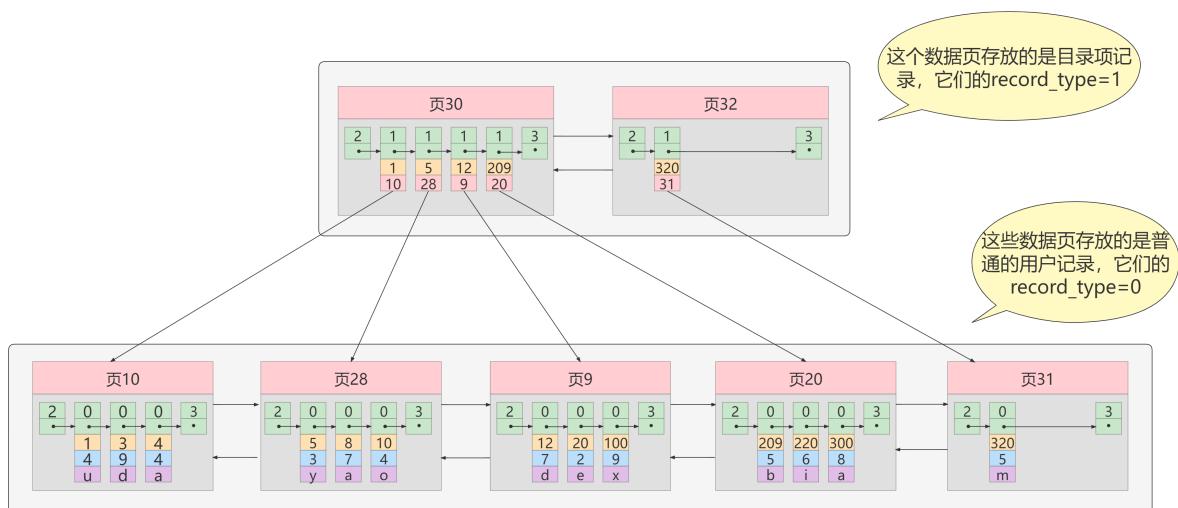
- 目录项记录 的 record_type 值是1，而 普通用户记录 的 record_type 值是0。
- 目录项记录只有 主键值和页的编号 两个列，而普通的用户记录的列是用户自己定义的，可能包含 很多列，另外还有InnoDB自己添加的隐藏列。
- 了解：记录头信息里还有一个叫 min_rec_mask 的属性，只有在存储 目录项记录 的页中的主键值最小的 目录项记录 的 min_rec_mask 值为 1，其他别的记录的 min_rec_mask 值都是 0。

相同点：两者用的是一样的数据页，都会为主键值生成 Page Directory (页目录)，从而在按照主键值进行查找时可以使用 二分法 来加快查询速度。

现在以查找主键为 20 的记录为例，根据某个主键值去查找记录的步骤就可以大致拆分成下边两步：

1. 先到存储 目录项记录 的页，也就是页30中通过 二分法 快速定位到对应目录项，因为 $12 < 20 < 209$ ，所以定位到对应的记录所在的页就是页9。
2. 再到存储用户记录的页9中根据 二分法 快速定位到主键值为 20 的用户记录。

② 迭代2次：多个目录项纪录的页



从图中可以看出，我们插入了一条主键值为320的用户记录之后需要两个新的数据页：

- 为存储该用户记录而新生成了 页31。
- 因为原先存储目录项记录的 页30的容量已满 (我们前边假设只能存储4条目录项记录)，所以不得不需要一个新的 页32 来存放 页31 对应的目录项。

现在因为存储目录项记录的页不止一个，所以如果我们想根据主键值查找一条用户记录大致需要3个步骤，以查找主键值为 20 的记录为例：

1. 确定 目录项记录页

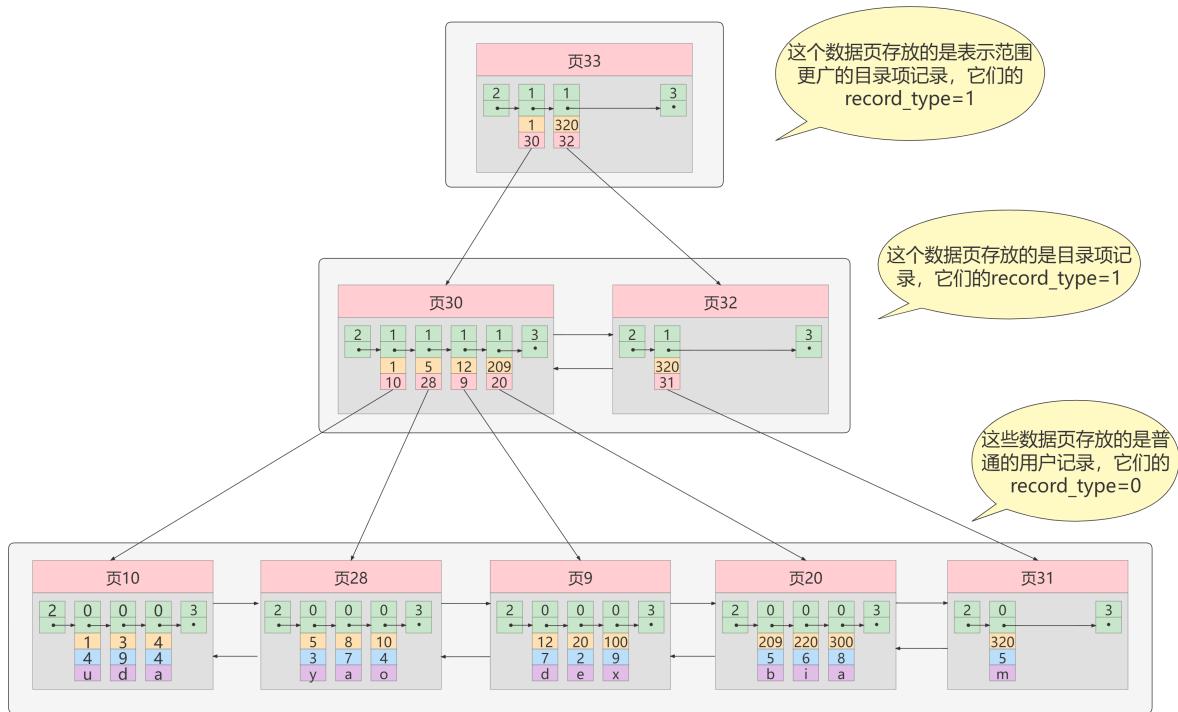
我们现在的存储目录项记录的页有两个，即 页30 和 页32，又因为页30表示的目录项的主键值的范围是 [1, 320)，页32表示的目录项的主键值不小于 320，所以主键值为 20 的记录对应的目录项记录在 页30 中。

2. 通过目录项记录页 确定用户记录真实所在的页。

在一个存储 目录项记录 的页中通过主键值定位一条目录项记录的方式说过了。

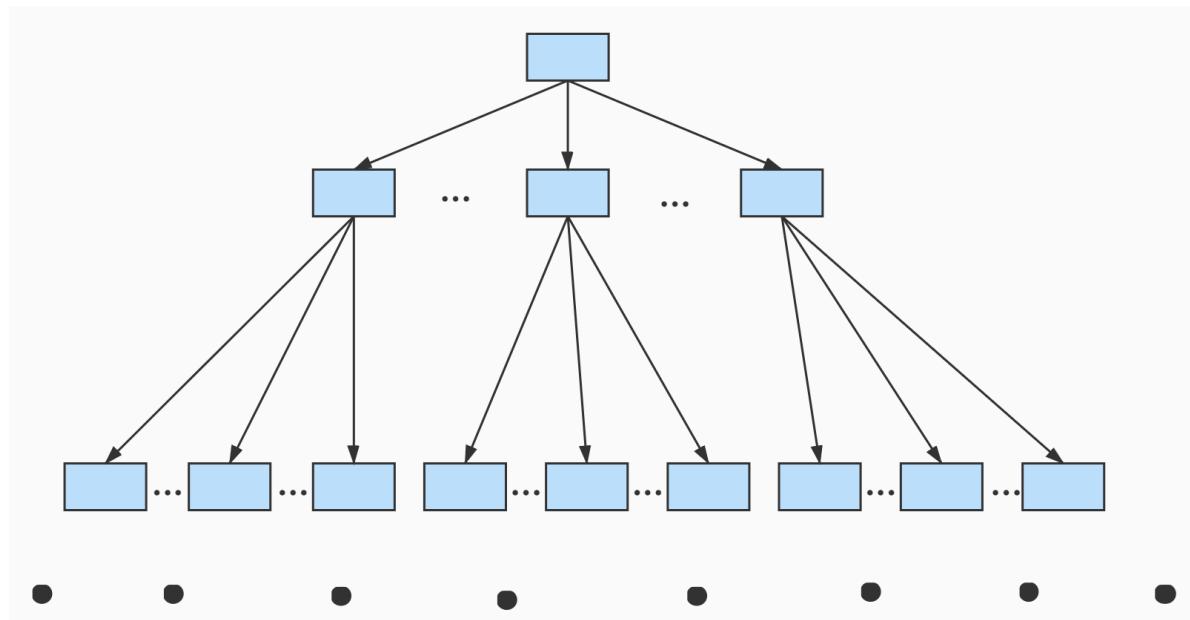
3. 在真实存储用户记录的页中定位到具体的记录。

③ 迭代3次：目录项记录页的目录页



如图，我们生成了一个存储更高级目录项的 [页33](#)，这个页中的两条记录分别代表页30和页32，如果用户记录的主键值在 [1, 320] 之间，则到页30中查找更详细的目录项记录，如果主键值 不小于320 的话，就到页32中查找更详细的目录项记录。

我们可以用下边这个图来描述它：



这个数据结构，它的名称是 [B+树](#)。

④ B+Tree

一个B+树的节点其实可以分成好多层，规定最下边的那层，也就是存放我们用户记录的那层为第 0 层，之后依次往上加。之前我们做了一个非常极端的假设：存放用户记录的页 [最多存放3条记录](#)，存放目录项记录的页 [最多存放4条记录](#)。其实真实环境中一个页存放的记录数量是非常大的，假设所有存放用户记录的叶子节点代表的数据页可以存放 [100条用户记录](#)，所有存放目录项记录的内节点代表的数据页可以存放 [1000条目录项记录](#)，那么：

- 如果B+树只有1层，也就是只有1个用于存放用户记录的节点，最多能存放 [100 条记录](#)。

- 如果B+树有2层，最多能存放 $1000 \times 100 = 10,000$ 条记录。
- 如果B+树有3层，最多能存放 $1000 \times 1000 \times 100 = 1,000,000$ 条记录。
- 如果B+树有4层，最多能存放 $1000 \times 1000 \times 1000 \times 100 = 1000,000,000$ 条记录。相当多的记录！！！

你的表里能存放 100000000000 条记录吗？所以一般情况下，我们用到的B+树都不会超过4层，那我们通过主键值去查找某条记录最多只需要做4个页面内的查找（查找3个目录项页和一个用户记录页），又因为在每个页面内有所谓的 **Page Directory**（页目录），所以在页面内也可以通过 **二分法** 实现快速定位记录。

3.3 常见索引概念

索引按照物理实现方式，索引可以分为2种：聚簇（聚集）和非聚簇（非聚集）索引。我们也把非聚集索引称为二级索引或者辅助索引。

1. 聚簇索引

特点：

1. 使用记录主键值的大小进行记录和页的排序，这包括三个方面的含义：
 - 页内 的记录是按照主键的大小顺序排成一个 **单向链表**。
 - 各个存放 **用户记录的页** 也是根据页中用户记录的主键大小顺序排成一个 **双向链表**。
 - 存放 **目录项记录的页** 分为不同的层次，在同一层次中的页也是根据页中目录项记录的主键大小顺序排成一个 **双向链表**。
2. B+树的 **叶子节点** 存储的是完整的用户记录。

所谓完整的用户记录，就是指这个记录中存储了所有列的值（包括隐藏列）。

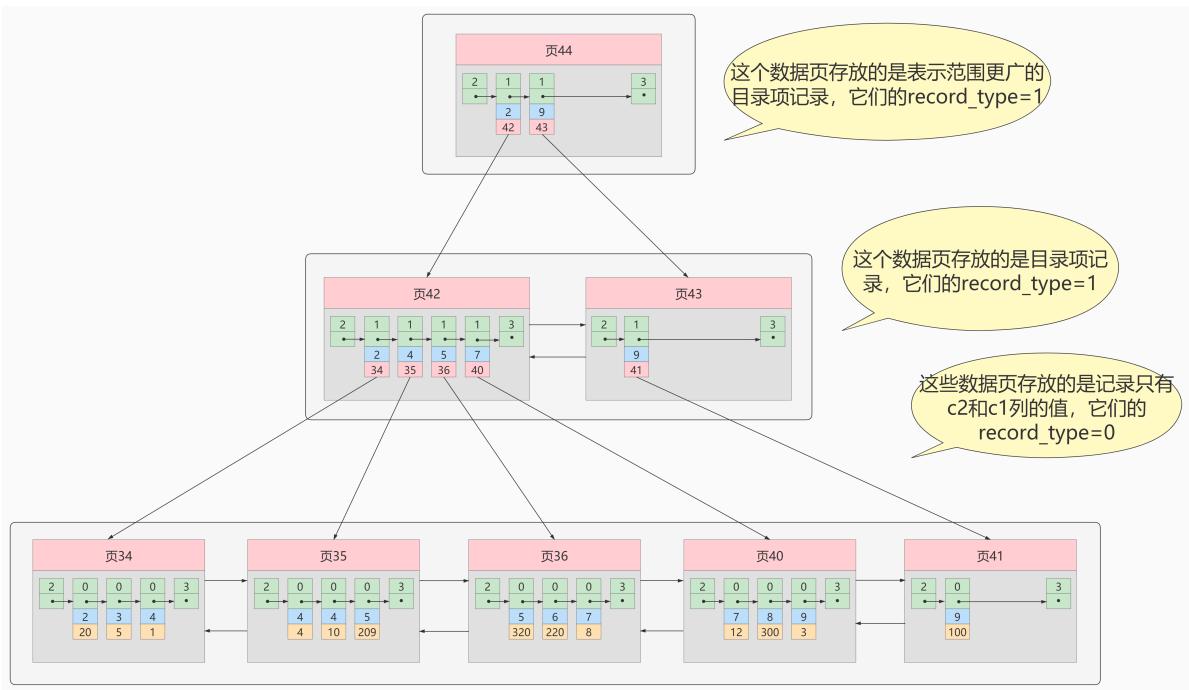
优点：

- **数据访问更快**，因为聚簇索引将索引和数据保存在同一个B+树中，因此从聚簇索引中获取数据比非聚簇索引更快
- 聚簇索引对于主键的 **排序查找** 和 **范围查找** 速度非常快
- 按照聚簇索引排列顺序，查询显示一定范围数据的时候，由于数据都是紧密相连，数据库不用从多个数据块中提取数据，所以 **节省了大量的io操作**。

缺点：

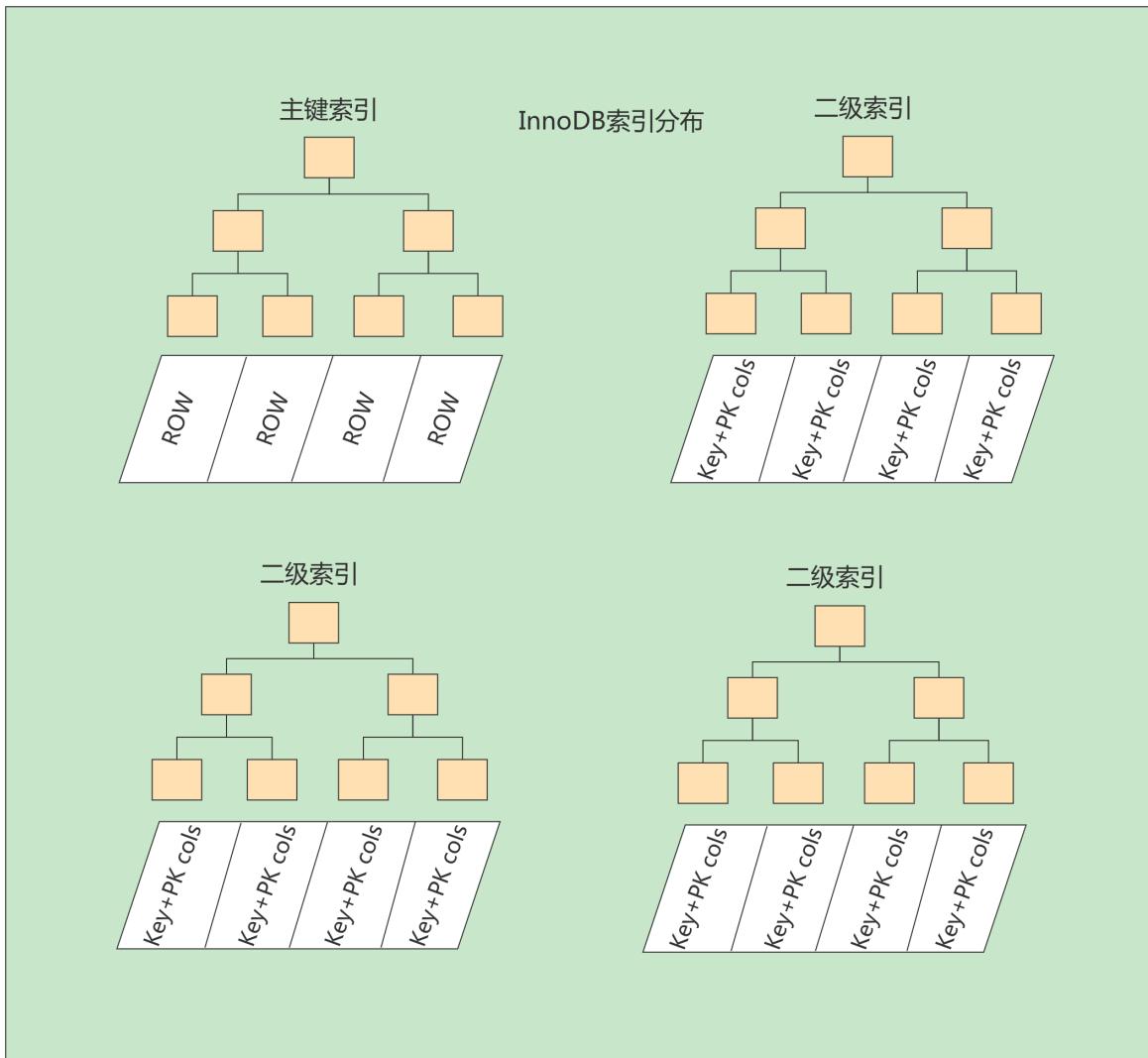
- **插入速度严重依赖于插入顺序**，按照主键的顺序插入是最快的方式，否则将会出现页分裂，严重影响性能。因此，对于InnoDB表，我们一般都会定义一个**自增的ID列为主键**
- **更新主键的代价很高**，因为将会导致被更新的行移动。因此，对于InnoDB表，我们一般定义**主键为不可更新**
- **二级索引访问需要两次索引查找**，第一次找到主键值，第二次根据主键值找到行数据

2. 二级索引（辅助索引、非聚簇索引）



概念：回表 我们根据这个以c2列大小排序的B+树只能确定我们要查找记录的主键值，所以如果我们想根据c2列的值查找到完整的用户记录的话，仍然需要到聚簇索引中再查一遍，这个过程称为**回表**。也就是根据c2列的值查询一条完整的用户记录需要使用到**2**棵B+树！

问题：为什么我们还需要一次**回表**操作呢？直接把完整的用户记录放到叶子节点不OK吗？



3. 联合索引

我们也可以同时以多个列的大小作为排序规则，也就是同时为多个列建立索引，比方说我们想让B+树按照 c2和c3列 的大小进行排序，这个包含两层含义：

- 先把各个记录和页按照c2列进行排序。
- 在记录的c2列相同的情况下，采用c3列进行排序

注意一点，以c2和c3列的大小为排序规则建立的B+树称为 [联合索引](#)，本质上也是一个二级索引。它的意思与分别为c2和c3列分别建立索引的表述是不同的，不同点如下：

- 建立 [联合索引](#) 只会建立如上图一样的1棵B+树。
- 为c2和c3列分别建立索引会分别以c2和c3列的大小为排序规则建立2棵B+树。

3.4 InnoDB的B+树索引的注意事项

1. 根页面位置万年不动
2. 内节点中目录项记录的唯一性
3. 一个页面最少存储2条记录

4. MyISAM中的索引方案

B树索引适用存储引擎如表所示：

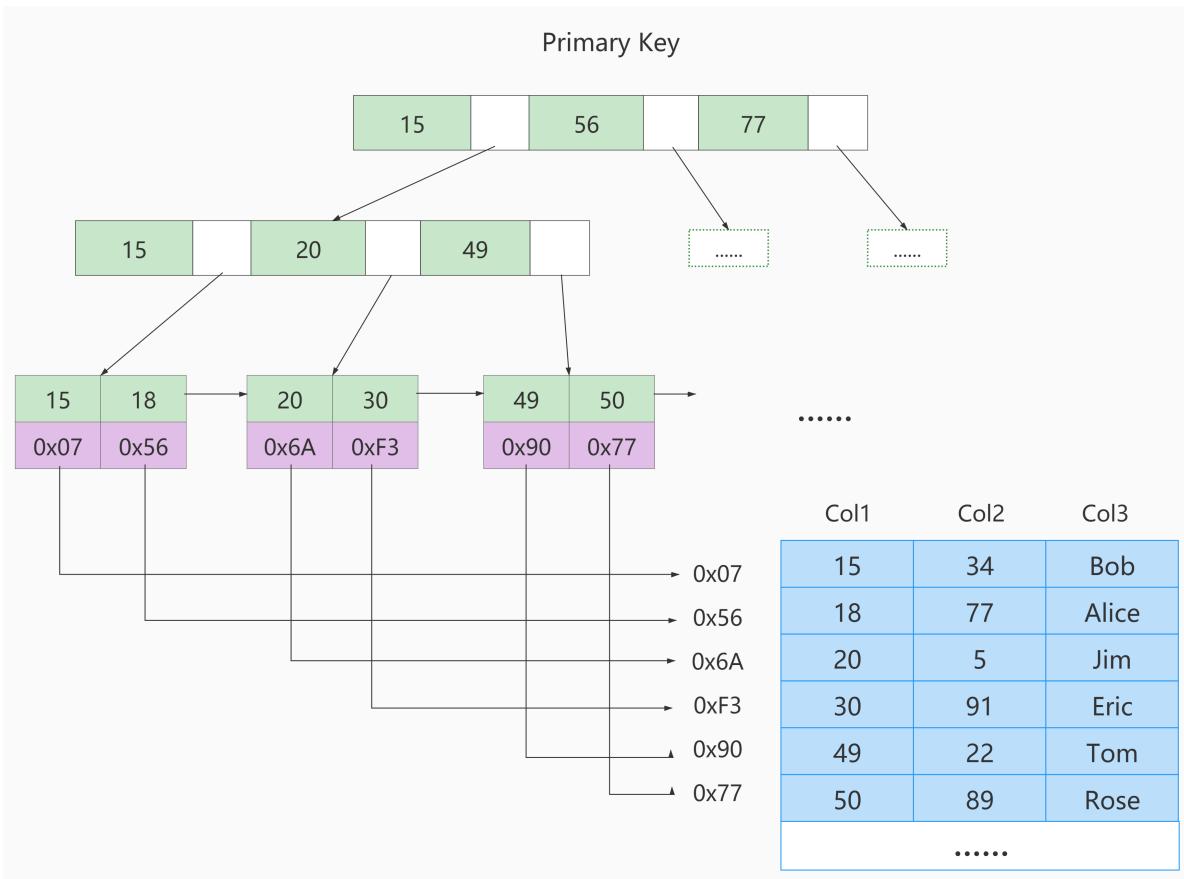
索引 / 存储引擎	MyISAM	InnoDB	Memory
B-Tree索引	支持	支持	支持

即使多个存储引擎支持同一种类型的索引，但是他们的实现原理也是不同的。Innodb和MyISAM默认的索引是Btree索引；而Memory默认的索引是Hash索引。

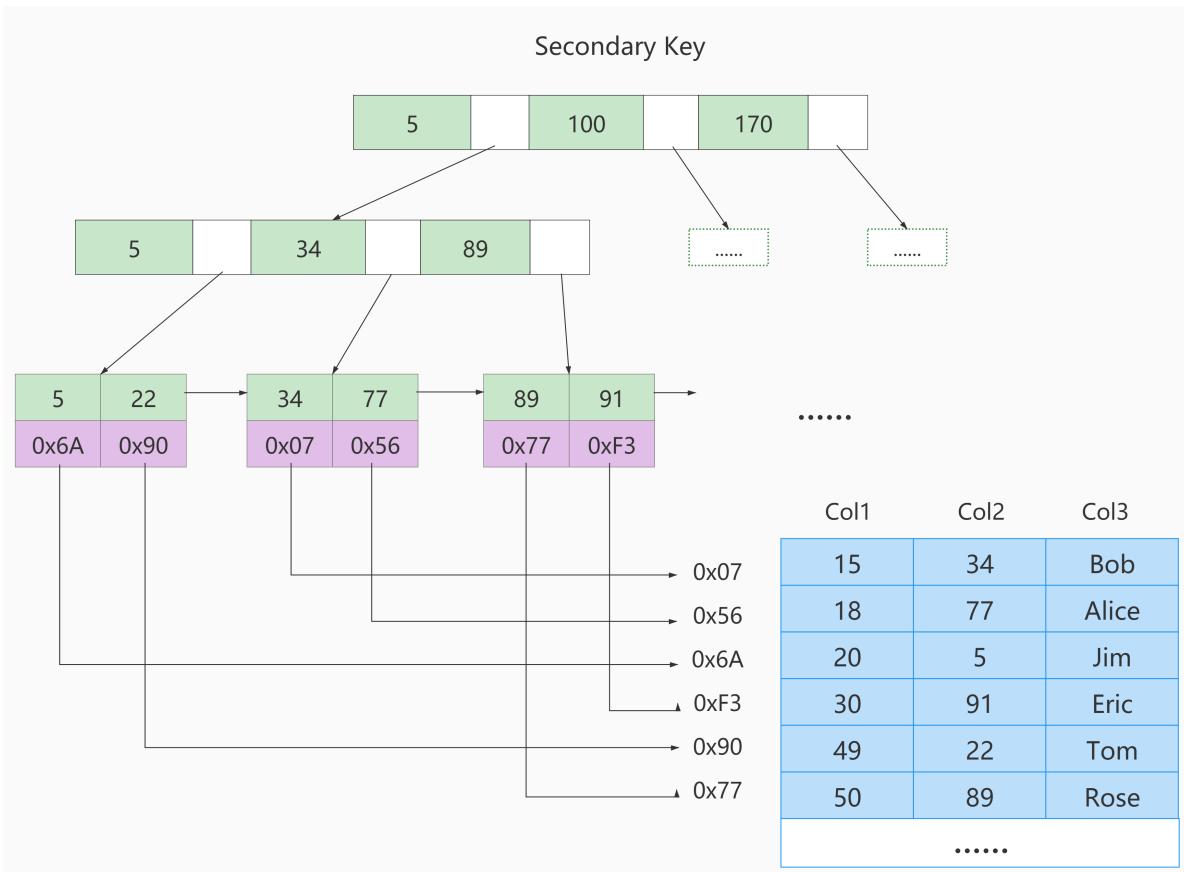
MyISAM引擎使用 [B+Tree](#) 作为索引结构，叶子节点的数据域存放的是 [数据记录的地址](#)。

4.2 MyISAM索引的原理

下图是MyISAM索引的原理图。



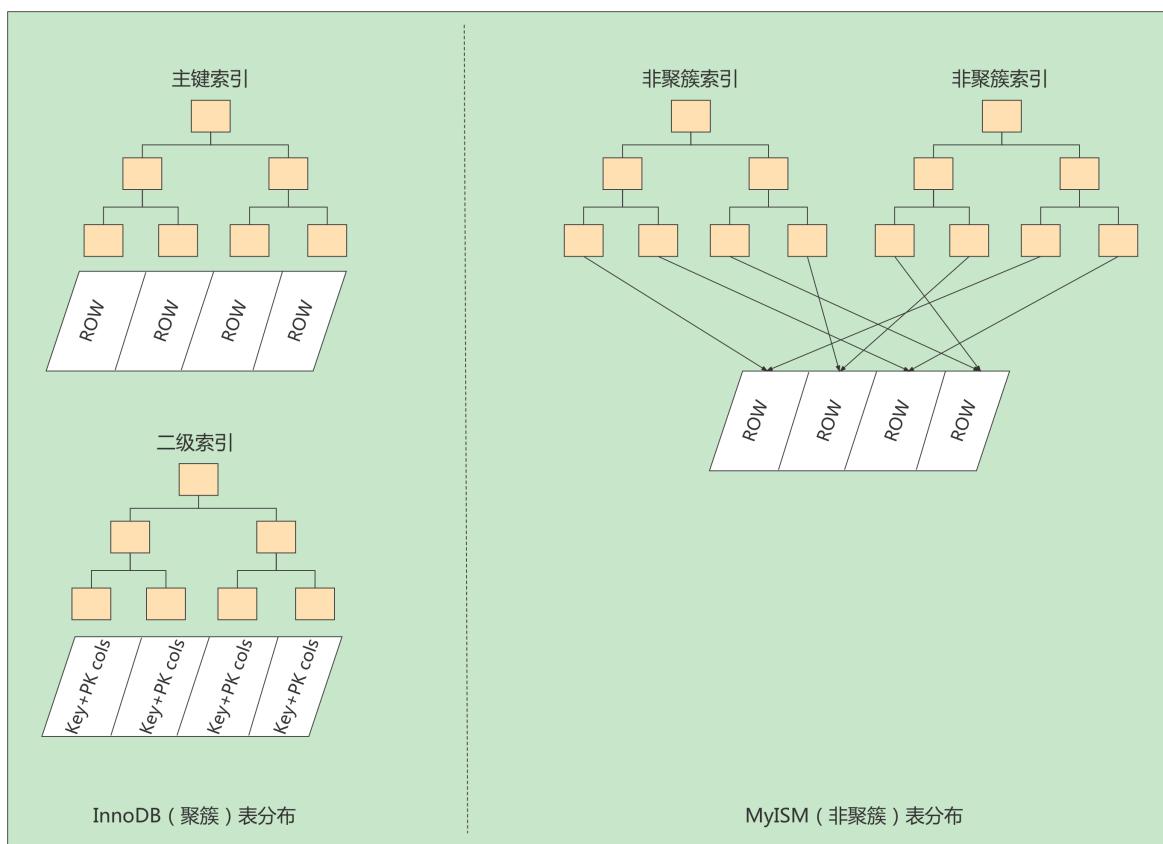
如果我们在Col2上建立一个二级索引，则此索引的结构如下图所示：



4.3 MyISAM 与 InnoDB 对比

MyISAM 的索引方式都是“非聚簇”的，与 InnoDB 包含 1 个聚簇索引是不同的。小结两种引擎中索引的区别：

- ① 在 InnoDB 存储引擎中，我们只需要根据主键值对 聚簇索引 进行一次查找就能找到对应的记录，而在 MyISAM 中却需要进行一次 回表 操作，意味着 MyISAM 中建立的索引相当于全部都是 二级索引 。
- ② InnoDB 的数据文件本身就是索引文件，而 MyISAM 索引文件和数据文件是 分离的 ，索引文件仅保存数据记录的地址。
- ③ InnoDB 的非聚簇索引 data 域存储相应记录 主键的值 ，而 MyISAM 索引记录的是 地址 。换句话说，InnoDB 的所有非聚簇索引都引用主键作为 data 域。
- ④ MyISAM 的回表操作是十分 快速 的，因为是拿着地址偏移量直接到文件中取数据的，反观 InnoDB 是通过获取主键之后再去聚簇索引里找记录，虽然说也不慢，但还是比不上直接用地址去访问。
- ⑤ InnoDB 要求表 必须有主键 （ MyISAM 可以没有 ）。如果没有显式指定，则 MySQL 系统会自动选择一个可以非空且唯一标识数据记录的列作为主键。如果不存在这种列，则 MySQL 自动为 InnoDB 表生成一个隐含字段作为主键，这个字段长度为 6 个字节，类型为长整型。



5. 索引的代价

索引是个好东西，可不能乱建，它在空间和时间上都会有消耗：

- **空间上的代价**

每建立一个索引都要为它建立一棵 B+ 树，每一棵 B+ 树的每一个节点都是一个数据页，一个页默认会占用 16KB 的存储空间，一棵很大的 B+ 树由许多数据页组成，那就是很大的一片存储空间。

- **时间上的代价**

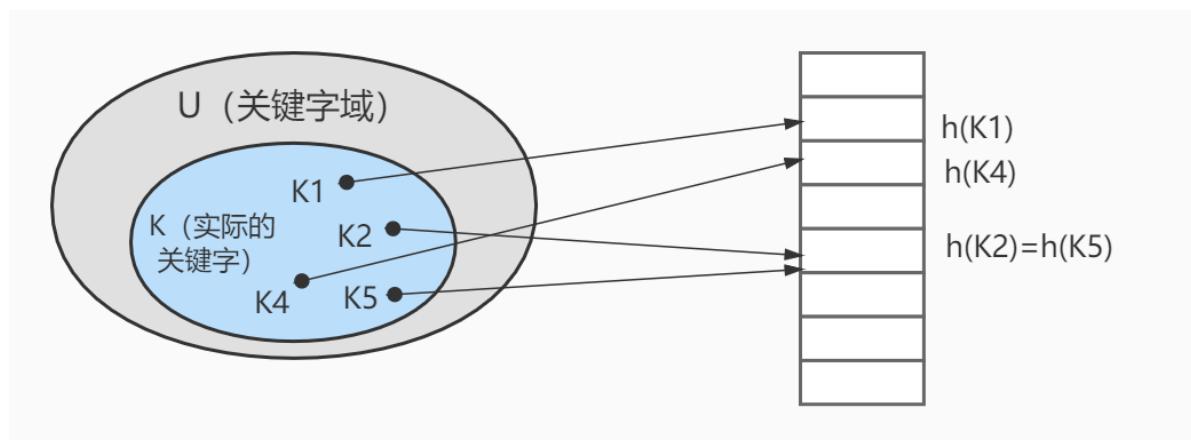
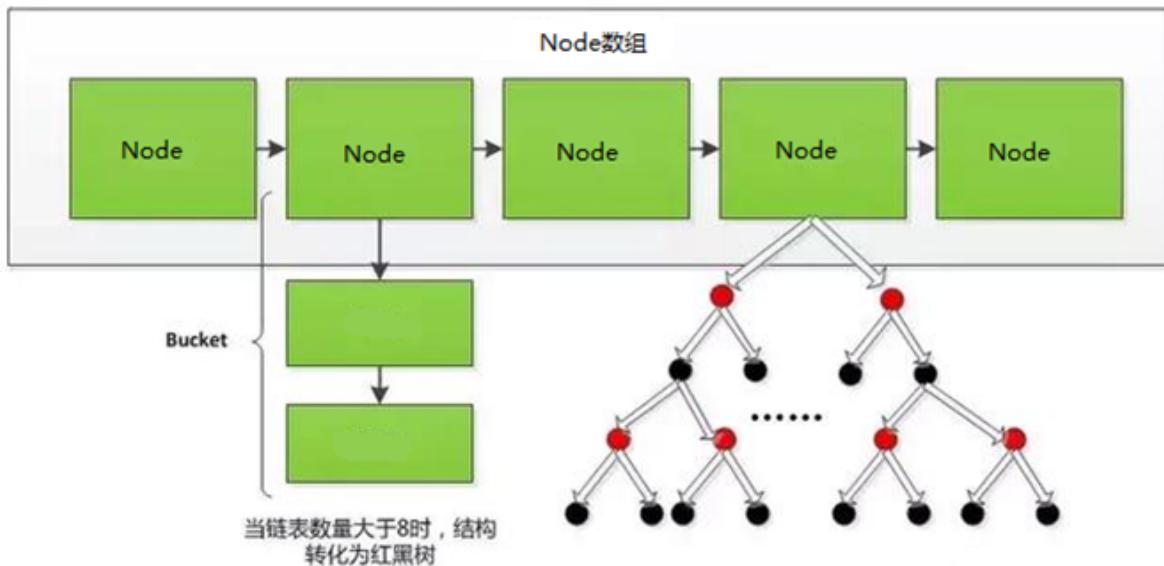
每次对表中的数据进行 增、删、改 操作时，都需要去修改各个B+树索引。而且我们讲过，B+树每层节点都是按照索引列的值 从小到大的顺序排序 而组成了 双向链表 。不论是叶子节点中的记录，还是内节点中的记录（也就是不论是用户记录还是目录项记录）都是按照索引列的值从小到大的顺序而形成了一个单向链表。而增、删、改操作可能会对节点和记录的排序造成破坏，所以存储引擎需要额外的时间进行一些 记录移位， 页面分裂、 页面回收 等操作来维护好节点和记录的排序。如果我们建了许多索引，每个索引对应的B+树都要进行相关的维护操作，会给性能拖后腿。

6. MySQL数据结构选择的合理性

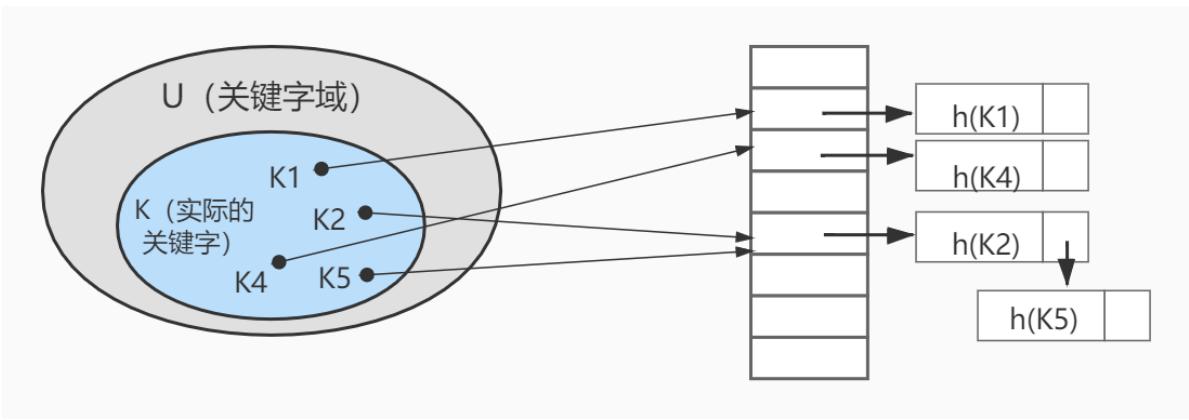
6.1 全表遍历

这里都懒得说了。

6.2 Hash结构



上图中哈希函数 h 有可能将两个不同的关键字映射到相同的位置，这叫做 碰撞，在数据库中一般采用 链接法 来解决。在链接法中，将散列到同一槽位的元素放在一个链表中，如下图所示：



实验：体会数组和hash表的查找方面的效率区别

```
// 算法复杂度为 O(n)
@Test
public void test1(){
    int[] arr = new int[100000];
    for(int i = 0;i < arr.length;i++){
        arr[i] = i + 1;
    }

    long start = System.currentTimeMillis();
    for(int j = 1; j<=100000;j++){
        int temp = j;

        for(int i = 0;i < arr.length;i++){
            if(temp == arr[i]){
                break;
            }
        }
    }
    long end = System.currentTimeMillis();
    System.out.println("time: " + (end - start)); //time: 823
}
```

```
//算法复杂度为 O(1)
@Test
public void test2(){
    HashSet<Integer> set = new HashSet<>(100000);
    for(int i = 0;i < 100000;i++){
        set.add(i + 1);
    }

    long start = System.currentTimeMillis();

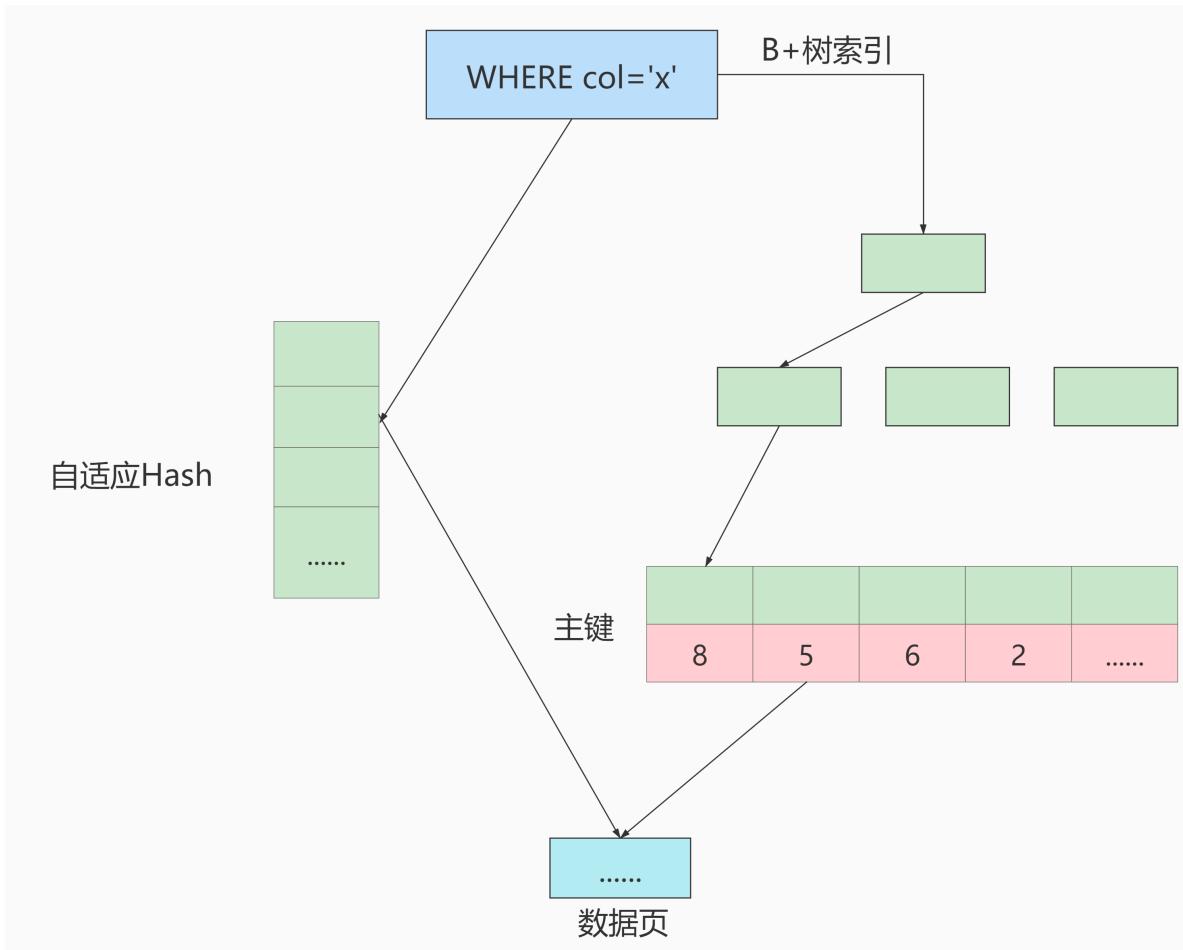
    for(int j = 1; j<=100000;j++) {
        int temp = j;
        boolean contains = set.contains(temp);
    }
    long end = System.currentTimeMillis();
    System.out.println("time: " + (end - start)); //time: 5
}
```

Hash结构效率高，那为什么索引结构要设计成树型呢？

Hash索引适用存储引擎如表所示：

索引 / 存储引擎	MyISAM	InnoDB	Memory
HASH索引	不支持	不支持	支持

Hash索引的适用性：



采用自适应 Hash 索引目的是方便根据 SQL 的查询条件加速定位到叶子节点，特别是当 B+ 树比较深的时候，通过自适应 Hash 索引可以明显提高数据的检索效率。

我们可以通过 `innodb_adaptive_hash_index` 变量来查看是否开启了自适应 Hash，比如：

```
mysql> show variables like '%adaptive_hash_index';
```

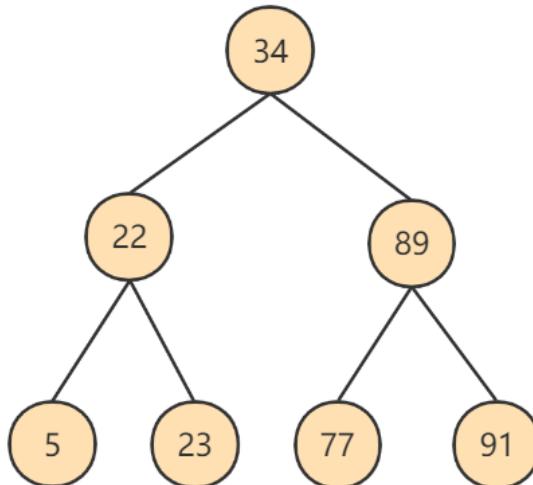
```
mysql> show variables like '%adaptive_hash_index';
+-----+-----+
| Variable_name          | Value   |
+-----+-----+
| innodb_adaptive_hash_index | ON      |
+-----+-----+
1 row in set (0.01 sec)
```

6.3 二叉搜索树

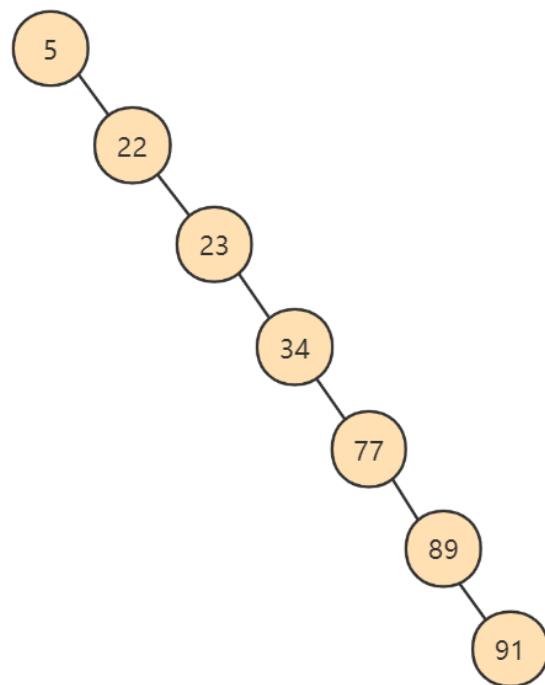
如果我们利用二叉树作为索引结构，那么磁盘的IO次数和索引树的高度是相关的。

1. 二叉搜索树的特点

2. 查找规则

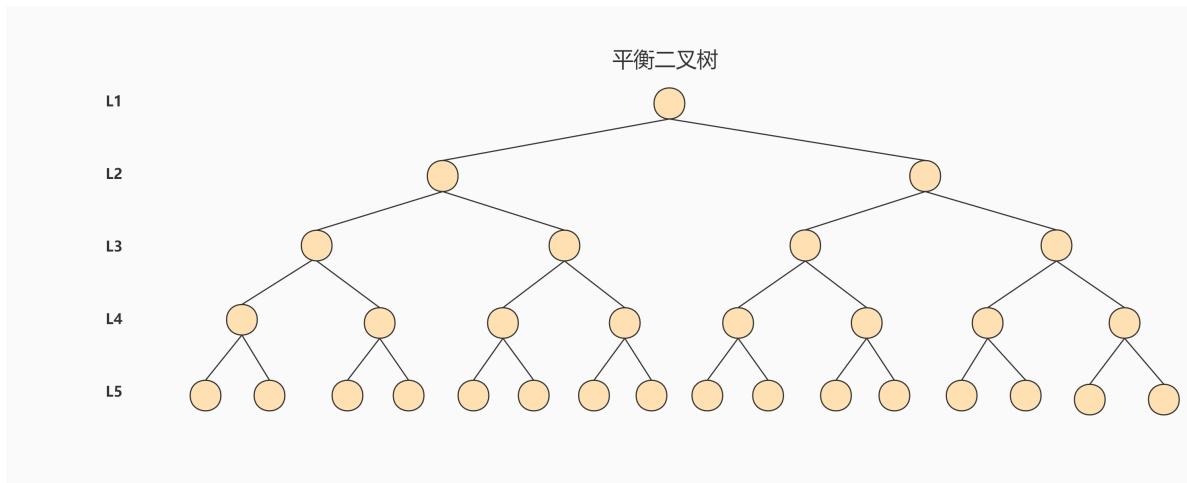


创造出来的二分搜索树如下图所示：

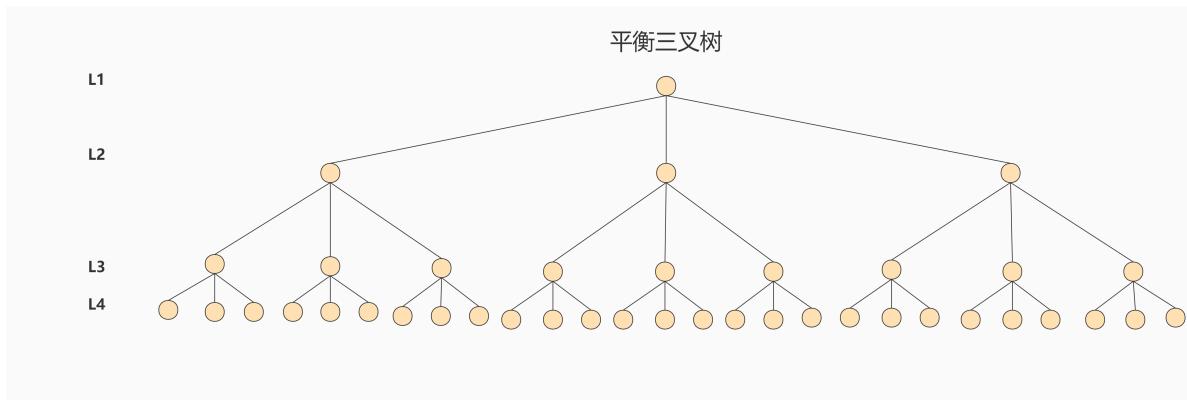


为了提高查询效率，就需要 **减少磁盘IO数**。为了减少磁盘IO的次数，就需要尽量 **降低树的高度**，需要把原来“瘦高”的树结构变的“矮胖”，树的每层的分叉越多越好。

6.4 AVL树

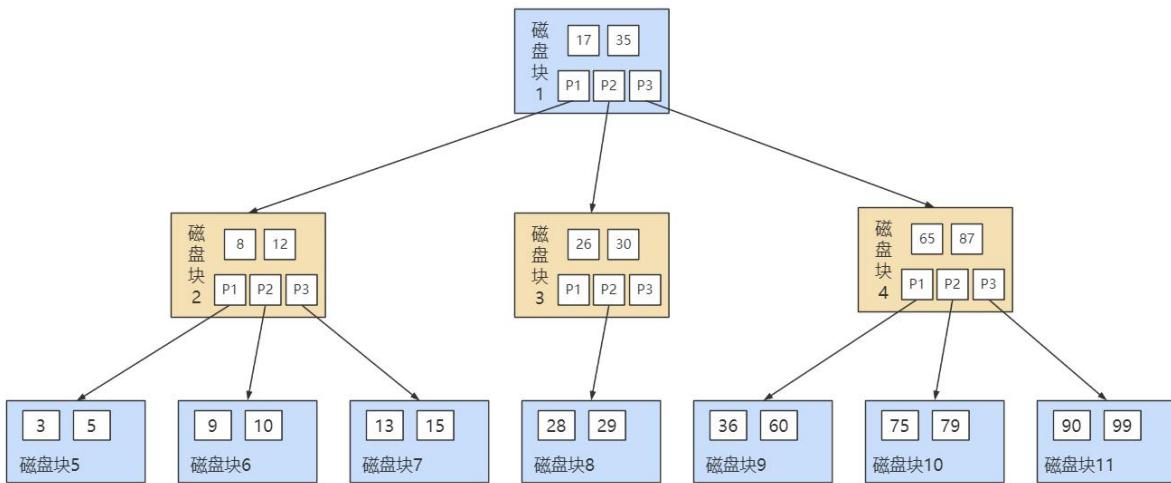


针对同样的数据，如果我们把二叉树改成 **M 叉树** ($M > 2$) 呢？当 $M=3$ 时，同样的 31 个节点可以由下面的三叉树来进行存储：



6.5 B-Tree

B 树的结构如下图所示：



一个 M 阶的 B 树 ($M > 2$) 有以下的特性：

1. 根节点的儿子数的范围是 $[2, M]$ 。
2. 每个中间节点包含 $k-1$ 个关键字和 k 个孩子，孩子的数量 = 关键字的数量 +1， k 的取值范围为 $[ceil(M/2), M]$ 。
3. 叶子节点包括 $k-1$ 个关键字（叶子节点没有孩子）， k 的取值范围为 $[ceil(M/2), M]$ 。
4. 假设中间节点节点的关键字为：Key[1], Key[2], ..., Key[k-1]，且关键字按照升序排序，即 $Key[i] < Key[i+1]$ 。此时 $k-1$ 个关键字相当于划分了 k 个范围，也就是对应着 k 个指针，即为：P[1], P[2], ..., P[k]

$P[k]$, 其中 $P[1]$ 指向关键字小于 $Key[1]$ 的子树, $P[i]$ 指向关键字属于 $(Key[i-1], Key[i])$ 的子树, $P[k]$ 指向关键字大于 $Key[k-1]$ 的子树。

- 所有叶子节点位于同一层。

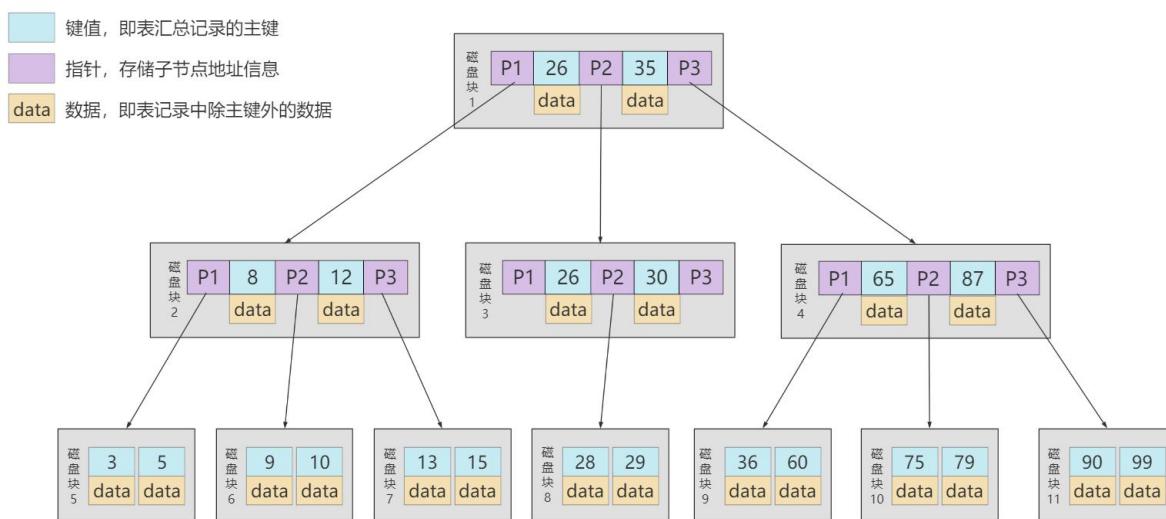
上面那张图所表示的 B 树就是一棵 3 阶的 B 树。我们可以看下磁盘块 2, 里面的关键字为 (8, 12), 它有 3 个孩子 (3, 5), (9, 10) 和 (13, 15), 你能看到 (3, 5) 小于 8, (9, 10) 在 8 和 12 之间, 而 (13, 15) 大于 12, 刚好符合刚才我们给出的特征。

然后我们来看下如何用 B 树进行查找。假设我们想要 **查找的关键字是 9**, 那么步骤可以分为以下几步:

- 我们与根节点的关键字 (17, 35) 进行比较, 9 小于 17 那么得到指针 P1;
- 按照指针 P1 找到磁盘块 2, 关键字为 (8, 12), 因为 9 在 8 和 12 之间, 所以我们得到指针 P2;
- 按照指针 P2 找到磁盘块 6, 关键字为 (9, 10), 然后我们找到了关键字 9。

你能看出来在 B 树的搜索过程中, 我们比较的次数并不少, 但如果把数据读取出来然后在内存中进行比较, 这个时间就是可以忽略不计的。而读取磁盘块本身需要进行 I/O 操作, 消耗的时间比在内存中进行比较所需要的时间要多, 是数据查找用时的重要因素。**B 树相比于平衡二叉树来说磁盘 I/O 操作要少**, 在数据查询中比平衡二叉树效率要高。所以 **只要树的高度足够低, IO 次数足够少, 就可以提高查询性能**。

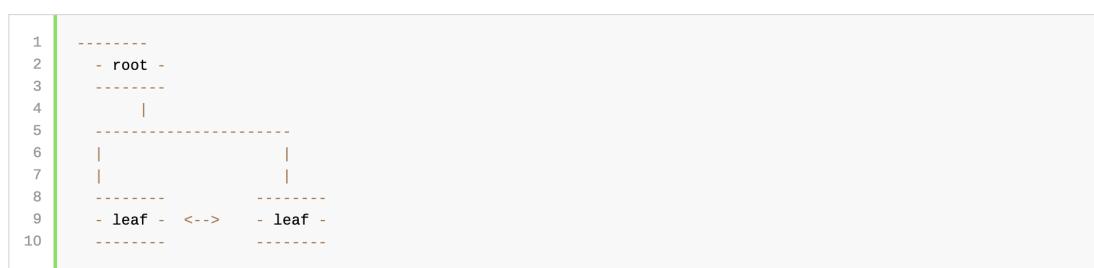
再举例1:



6.6 B+Tree

- MySQL官网说明:

• FIL_PAGE_PREV and FIL_PAGE_NEXT are the page's "backward" and "forward" pointers. To show what they're about, I'll draw a two-level B-tree.



Everyone has seen a B-tree and knows that the entries in the root page point to the leaf pages. (I indicate those pointers with vertical '|' bars in the drawing.) But sometimes people miss the detail that leaf pages can also point to each other (I indicate those pointers with a horizontal two-way pointer '<->' in the drawing). This feature allows InnoDB to navigate from leaf to leaf without having to back up to the root level. This is a sophistication which you won't find in the classic B-tree, which is why InnoDB should perhaps be called a B+-tree instead.

B+ 树和 B 树的差异:

- 有 k 个孩子的节点就有 k 个关键字。也就是孩子数量 = 关键字数, 而 B 树中, 孩子数量 = 关键字数 +1。

- 非叶子节点的关键字也会同时存在在子节点中，并且是在子节点中所有关键字的最大（或最小）。
- 非叶子节点仅用于索引，不保存数据记录，跟记录有关的信息都放在叶子节点中。而 B 树中，**非**
叶子节点既保存索引，也保存数据记录。
- 所有关键字都在叶子节点出现，叶子节点构成一个有序链表，而且叶子节点本身按照关键字的大小从小到大顺序链接。

B 树和 B+ 树都可以作为索引的数据结构，在 MySQL 中采用的是 B+ 树。

但 B 树和 B+ 树各有自己的应用场景，不能说 B+ 树完全比 B 树好，反之亦然。

思考题：为了减少 IO，索引树会一次性加载吗？

思考题：B+ 树的存储能力如何？为何说一般查找行记录，最多只需 1~3 次磁盘 IO？

思考题：为什么说 B+ 树比 B 树更适合实际应用中操作系统的文件索引和数据库索引？

思考题：Hash 索引与 B+ 树索引的区别

思考题：Hash 索引与 B+ 树索引是在建索引的时候手动指定的吗？

6.7 R树

R-Tree 在 MySQL 很少使用，仅支持 **geometry** 数据类型，支持该类型的存储引擎只有 myisam、bdb、innodb、ndb、archive 几种。举个 R 树在现实领域中能够解决的例子：查找 20 英里以内所有的餐厅。如果没有 R 树你会怎么解决？一般情况下我们会把餐厅的坐标(x,y)分为两个字段存放在数据库中，一个字段记录经度，另一个字段记录纬度。这样的话我们就需要遍历所有的餐厅获取其位置信息，然后计算是否满足要求。如果一个地区有 100 家餐厅的话，我们就要进行 100 次位置计算操作了，如果应用到谷歌、百度地图这种超大数据库中，这种方法便必定不可行了。R 树就很好的 **解决了这种高维空间搜索问题**。它把 B 树的思想很好的扩展到了多维空间，采用了 B 树分割空间的思想，并在添加、删除操作时采用合并、分解结点的方法，保证树的平衡性。因此，R 树就是一棵用来 **存储高维数据的平衡树**。相对于 B-Tree，R-Tree 的优势在于范围查找。

索引 / 存储引擎	MyISAM	InnoDB	Memory
R-Tree 索引	支持	支持	不支持

附录：算法的时间复杂度

同一问题可用不同算法解决，而一个算法的质量优劣将影响到算法乃至程序的效率。算法分析的目的在于选择合适算法和改进算法。

数据结构	查找	插入	删除	遍历
数组	O(N)	O(1)	O(N)	—
有序数组	O(logN)	O(N)	O(N)	O(N)
链表	O(N)	O(1)	O(N)	—
有序链表	O(N)	O(N)	O(N)	O(N)
二叉树（一般情况）	O(logN)	O(logN)	O(logN)	O(N)
二叉树（最坏情况）	O(N)	O(N)	O(N)	O(N)
平衡树（一般情况和最坏情况）	O(logN)	O(logN)	O(logN)	O(N)
哈希表	O(1)	O(1)	O(1)	—

第07章_InnoDB数据存储结构

尚硅谷-宋红康

▼ 01-数据页内部结构

大小：默认16KB

构成：

▼ 第1部分

▼ File Header (文件头部) (38字节)

作用：

描述各种页的通用信息。（比如页的编号、其上一页、下一页是谁等）

大小：38字节

构成：

- FIL_PAGE_OFFSET (4字节)

每一个页都有一个单独的页号，就跟你的身份证号码一样，InnoDB通过页号可以唯一定位一个页。

- FIL_PAGE_TYPE (2字节)

这个代表当前页的类型。

- FIL_PAGE_PREV (4字节) 和FIL_PAGE_NEXT (4字节)

InnoDB都是以页为单位存放数据的，如果数据分散到多个不连续的页中存储的话需要把这些页关联起来，FIL_PAGE_PREV和FIL_PAGE_NEXT就分别代表本页的上一个和下一个页的页号。这样通过建立一个双向链表把许许多多的页就都串联起来了，保证这些页之间不需要是物理上的连续，而是逻辑上的连续。

- **FIL_PAGE_SPACE_OR_CHKSUM** (4字节)

代表当前页面的校验和 (checksum) 。

什么是校验和?

就是对于一个很长的字节串来说，我们会通过某种算法来计算一个比较短的值来代表这个很长的字节串，这个比较短的值就称为校验和。

在比较两个很长的字节串之前，先比较这两个长字节串的校验和，如果校验和都不一样，则两个长字节串肯定是不同的，所以省去了直接比较两个比较长的字节串的时间损耗。

文件头部和文件尾部都有属性：FIL_PAGE_SPACE_OR_CHKSUM

作用：...

- **FIL_PAGE_LSN** (8字节)

页面被最后修改时对应的日志序列位置 (英文名是：Log Sequence Number)

- **File Trailer** (文件尾部) (8字节)

前4个字节代表页的校验和：

这个部分是和File Header中的校验和相对应的。

后4个字节代表页面被最后修改时对应的日志序列位置 (LSN) :

这个部分也是为了校验页的完整性的，如果首部和尾部的LSN值校验不成功的话，就说明同步过程出现了问题。

▼ 第2部分

第二个部分是记录部分，页的主要作用是存储记录，所以“最大和最小记录”和“用户记录”部分占了页结构的主要空间。

- **Free Space (空闲空间)**

我们自己存储的记录会按照指定的行格式存储到User Records部分。但是在一开始生成页的时候，其实并没有User Records这个部分，每当我们插入一条记录，都会从Free Space部分，也就是尚未使用的存储空间中申请一个记录大小的空间划分到User Records部分，当Free Space部分的空间全部被User Records部分替代掉之后，也就意味着这个页使用完了，如果还有新的记录插入的话，就需要去申请新的页了。

- **User Records (用户记录)**

User Records中的这些记录按照指定的行格式一条一条摆在User Records部分，相互之间形成单链表。

用户记录里的一条条数据如何记录？

这里需要讲讲记录行格式的记录头信息。

- Infimum + Supremum (最小最大记录)

记录可以比较大小吗？

是的，记录可以比大小，对于一条完整的记录来说，比较记录的大小就是比较主键的大小。比方说我们插入的4行记录的主键值分别是：1、2、3、4，这也就意味着这4条记录是从小到大依次递增。

InnoDB规定的最小记录与最大记录这两条记录的构造十分简单，都是由5字节大小的记录头信息和8字节大小的一个固定的部分组成的，如图所示：

这两条记录不是我们自己定义的记录，所以它们并不存放在页的User Records部分，他们被单独放在一个称为Infimum + Supremum的部分，如图所示：

▼ 第3部分

- Page Directory (页目录)

为什么需要页目录？

在页中，记录是以单向链表的形式进行存储的。单向链表的特点就是插入、删除非常方便，但是检索效率不高，最差的情况下需要遍历链表上的所有节点才能完成检索。因此在页结构中专门设计了页目录这个模块，专门给记录做一个目录，通过二分查找法的方式进行检索，提升效率。

需求：根据主键值查找页中的某条记录，如何实现快速查找呢？

```
SELECT * FROM page_demo WHERE c1 = 3;
```

方式1：顺序查找

从Infimum记录（最小记录）开始，沿着链表一直往后找，总有一天会找到（或者找不到），在找的时候还能投机取巧，因为链表中各个记录的值是按照从小到大顺序排列的，所以当链表的某个节点代表的记录的主键值大于你想要查找的主键值时，你就可以停止查找了，因为该节点后边的节点的主键值依次递增。

...

- 页目录分组的个数如何确定？

为什么最小记录的n_owned值为1，而最大记录的n_owned值为5呢？

InnoDB规定：对于最小记录所在的分组只能有1条记录，最大记录所在的分组拥有的记录条数只能在1~8条之间，剩下的分组中记录的条数范围只能在是 4~8条之间。

分组是按照下边的步骤进行的：

初始情况下一个数据页里只有最小记录和最大记录两条记录，它们分属于两个分组。

之后每插入一条记录，都会从页目录中找到主键值比本记录的主键值大并且差值最小的槽，然后把该槽对应的记录的n_owned值加1，表示本组内又添加了一条记录，直到该组中的记录数等于8个。

在一个组中的记录数等于8个后再插入一条记录时，会将组中的记录拆分成两个组，一个组中4条记录，另一个5条记录。这个过程会在页目录中新增一个槽来记录这个新增分组...中

- 页目录结构下如何快速查找记录？

现在向page_demo表中添加更多的数据。如下：

```
INSERT INTO page_demo  
VALUES  
(5, 500, 'zhou'),  
(6, 600, 'chen'),  
(7, 700, 'deng'),  
(8, 800, 'yang'),  
(9, 900, 'wang'),  
(10, 1000, 'zhao'),  
(11, 1100, 'qian'), ...
```

▼ Page Header (页面头部)

为了能得到一个数据页中存储的记录的状态信息，比如本页中已经存储了多少条记录，第一条记录的地址是什么，页目录中存储了多少个槽等等，特意在页中定义了一个叫Page Header的部分，这个部分占用固定的56个字节，专门存储各种状态信息。

- PAGE_DIRECTION

假如新插入的一条记录的主键值比上一条记录的主键值大，我们说这条记录的插入方向是右边，反之则是左边。用来表示最后一条记录插入方向的状态就是PAGE_DIRECTION。

- PAGE_N_DIRECTION

假设连续几次插入新记录的方向都是一致的，InnoDB会把沿着同一个方向插入记录的条数记下来，这个条数就用PAGE_N_DIRECTION这个状态表示。当然，如果最后一条记录的插入方向改变了的话，这个状态的值会被清零重新统计。

▼ 02-InnoDB行格式（或记录格式）

我们平时的数据以行为单位来向表中插入数据，这些记录在磁盘上的存放方式也被称为‘行格式’或者‘记录格式’。InnoDB存储引擎设计了4种不同类型的‘行格式’，分别是‘Compact’、‘Redundant’、‘Dynamic’和‘Compressed’行格式。

查看MySQL8的默认行格式：

```
mysql> SELECT @@innodb_default_row_format;
+-----+
| @@innodb_default_row_format |
+-----+
| dynamic                         |
+-----+...
```

- 指定行格式的语法

在创建或修改表的语句中指定行格式：

CREATE TABLE 表名 (列的信息) ROW_FORMAT=行格式名称

ALTER TABLE 表名 ROW_FORMAT=行格式名称

举例：

```
mysql> CREATE TABLE record_test_table (
    ->   col1 VARCHAR(8),
    ->   col2 VARCHAR(8) NOT NULL,
    ->   col3 CHAR(8),
    ->   col4 VARCHAR(8)...)
```

▼ COMPACT行格式

在MySQL 5.1版本中，默认设置为Compact行格式。一条完整的记录其实可以被分为记录的额外信息和记录的真实数据两大部分。

- 变长字段长度列表

MySQL支持一些变长的数据类型，比如VARCHAR(M)、VARBINARY(M)、TEXT类型，BLOB类型，这些数据类型修饰列称为变长字段，变长字段中存储多少字节的数据不是固定的，所以我们在存储真实数据的时候需要顺便把这些数据占用的字节数也存起来。在Compact行格式中，把所有变长字段的真实数据占用的字节长度都存放在记录的开头部位，从而形成一个变长字段长度列表。

注意：这里面存储的变长长度和字段顺序是反过来的。比如两个varchar字段在表结构的顺序是a(10), b(15)。那么在变长字段长度列表中存储的长度顺序就是15, 10，是反过来的。

以record_test_table表中的第一条记录举例：因为record_test_table表的col1、col2、col4列都是VARCHAR(8)类型的，所以这三个列的值的长度都需要保存在记录开头处，注意record_test_table表中的各个列都使用的是ascii字符集（每个字符只需要1个字节来进行编码）。…

▪ NULL值列表

Compact行格式会把可以为NULL的列统一管理起来，存在一个标记为NULL值列表中。如果表中没有允许存储NULL的列，则NULL值列表也不存在了。

为什么定义NULL值列表？

之所以要存储NULL是因为数据都是需要对齐的，如果没有标注出来NULL值的位置，就有可能在查询数据的时候出现混乱。如果使用一个特定的符号放到相应的数据位表示空置的话，虽然能达到效果，但是这样很浪费空间，所以直接就在行数据得头部开辟出一块空间专门用来记录该行数据哪些是非空数据，哪些是空数据，格式如下：

1. 二进制位的值为1时，代表该列的值为NULL。
2. 二进制位的值为0时，代表该列的值不为NULL。

例如：字段a、b、c，其中a是主键，在某一行中存储的数据依次是a=1、b=null、c=2。那么Compact行格式中的NULL值列表中存储：01。第一个0表示c不为null，第二个1表示b是null。这里之所以没有a是因为a是主键，不需要存储在NULL值列表中。

▼ 记录头信息（5字节）

```
mysql> CREATE TABLE page_demo(  
->     c1 INT,  
->     c2 INT,  
->     c3 VARCHAR(10000),  
->     PRIMARY KEY (c1)  
-> ) CHARSET=ascii ROW_FORMAT=Compact;  
Query OK, 0 rows affected (0.03 sec)
```

这个表中记录的行格式示意图：

...

▪ delete_mask

这个属性标记着当前记录是否被删除，占用1个二进制位。

值为0：代表记录并没有被删除

值为1：代表记录被删除掉了

被删除的记录为什么还在页中存储呢？

你以为它删除了，可它还在真实的磁盘上。这些被删除的记录之所以不立即从磁盘上移除，是因为移除它们之后其他的记录在磁盘上需要重新排列，导致性能消耗。所以只是打一个删除标记而已，所有被删除掉的记录都会组成一个所谓的垃圾链表，在这个链表中的记录占用的空间称之为可重用空间，之后如果有新记录插入到表中的话，可能把这些被删除的记录占用的存储空间覆盖掉。

▪ min_rec_mask

B+树的每层非叶子节点中的最小记录都会添加该标记，min_rec_mask值为1。

我们自己插入的四条记录的min_rec_mask值都是0，意味着它们都不是B+树的非叶子节点中的最小记录。

- **record_type**

这个属性表示当前记录的类型，一共有4种类型的记录：

- 0: 表示普通记录
- 1: 表示B+树非叶节点记录
- 2: 表示最小记录
- 3: 表示最大记录

从图中我们也可以看出来，我们自己插入的记录就是普通记录，它们的 record_type 值都是0，而最小记录和最大记录的 record_type 值分别为2和3。至于 record_type 为1的情况，我们在索引的数据结构章节讲过。

- **heap_no**

这个属性表示当前记录在本页中的位置。

从图中可以看出来，我们插入的4条记录在本页中的位置分别是：2、3、4、5。

怎么不见 heap_no 值为0和1的记录呢？

MySQL会自动给每个页里加了两个记录，由于这两个记录并不是我们自己插入的，所以有时候也称为伪记录或者虚拟记录。这两个伪记录一个代表最小记录，一个代表最大记录。最小记录和最大记录的 heap_no 值分别是0和1，也就是说它们的位置最靠前。

- **n_owned**

页目录中每个组中最后一条记录的头信息中会存储该组一共有多少条记录，作为 n_owned 字段。

详情见 page directory。

- ▼ **next_record**

记录头信息里该属性非常重要，它表示从当前记录的真实数据到下一条记录的真实数据的地址偏移量。

比如：第一条记录的 next_record 值为32，意味着从第一条记录的真实数据的地址处向后找32个字节便是下一条记录的真实数据。

注意，下一条记录指得并不是按照我们插入顺序的下一条记录，而是按照主键值由小到大的顺序的下一条记录。而且规定 Infimum 记录（也就是最小记录）的下一条记录就是本页中主键值最小的用户记录，而本页中主键值最大的用户记录的下一条记录就是 Supremum 记录（也就是最大记录）。下图用箭头代替偏移量表示 next_record。

- 演示：删除操作

删除操作：

从表中删除掉一条记录，这个链表也是会跟着变化：

```
mysql> DELETE FROM page_demo WHERE c1 = 2;
```

```
Query OK, 1 row affected (0.02 sec)
```

删掉第2条记录后的示意图就是：

从图中可以看出来，删除第2条记录前后主要发生了这些变化：

- 第2条记录并没有从存储空间中移除，而是把该条记录的delete_mask值设置为1。
- 第2条记录的next_record值变为了0，意味着该记录没有下一条记录了。
- 第1条记录的next_record指向了第3条记录。...

- 演示：添加操作

添加操作：

主键值为2的记录被我们删掉了，但是存储空间却没有回收，如果我们再次把这条记录插入到表中，会发生什么事呢？

```
mysql> INSERT INTO page_demo VALUES(2, 200, 'tong');
```

```
Query OK, 1 row affected (0.00 sec)
```

我们看一下记录的存储情况：

直接复用了原来被删除记录的存储空间。

...

- 记录的真实数据

记录的真实数据除了我们自己定义的列的数据以外，还会有三个隐藏列：

实际上这几个列的真正名称其实是：DB_ROW_ID、DB_TRX_ID、DB_ROLL_PTR。

一个表没有手动定义主键，则会选取一个Unique键作为主键，如果连Unique键都没有定义的话，则会为表默认添加一个名为row_id的隐藏列作为主键。所以row_id是在没有自定义主键以及Unique键的情况下才会存在的。

事务ID和回滚指针在后面的《第14章_MySQL事务日志》章节中讲解。

举例：分析Compact行记录的内部结构：

```
CREATE TABLE mytest(...
```

- ▼ Dynamic和Compressed行格式

- 行溢出

InnoDB存储引擎可以将一条记录中的某些数据存储在真正的数据页面之外。

很多DBA喜欢MySQL数据库提供的VARCHAR(M)类型，认为可以存放65535字节。这是真的吗？如果我们使用 ascii字符集的话，一个字符就代表一个字节，我们看看VARCHAR(65535)是否可用。

```
CREATE TABLE varchar_size_demo(  
    c VARCHAR(65535)  
) CHARSET=ascii ROW_FORMAT=Compact;
```

结果如下：

ERROR 1118 (42000): Row size too large. The maximum row size for the used table type, not counting BLOBs, is 65535. This includes storage overhead, check...

- Dynamic和Compressed行格式

在MySQL 8.0中，默认行格式就是Dynamic、Dynamic、Compressed行格式和Compact行格式挺像，只不过在处理行溢出数据时有分歧：

Compressed和Dynamic两种记录格式对于存放在BLOB中的数据采用了完全的行溢出的方式。如图，在数据页中只存放20个字节的指针（溢出页的地址），实际的数据都存放在Off Page（溢出页）中。

Compact和Redundant两种格式会在记录的真实数据处存储一部分数据（存放768个前缀字节）。

Compressed行记录格式的另一个功能就是，存储在其中的行数据会以zlib的算法进行压缩，因此对于BLOB、TEXT、VARCHAR这类大长度类型的数据能够进行非常有效的存储。

- ▼ Redundant行格式

Redundant是MySQL 5.0版本之前InnoDB的行记录存储方式，MySQL 5.0支持Redundant是为了兼容之前版本的页格式。

现在我们把表record_test_table的行格式修改为Redundant：

```
ALTER TABLE record_test_table ROW_FORMAT=Redundant;  
Query OK, 0 rows affected (0.05 sec)  
Records: 0 Duplicates: 0 Warnings: 0
```

从上图可以看到，不同于Compact行记录格式，Redundant行格式的首部是一个字段长度偏移列表，同样按照列的顺序逆序放置的。

下边我们从各个方面看一下Redundant行格式有什么不同的地方。

- 字段长度偏移列表

注意Compact行格式的开头是变长字段长度列表，而Redundant行格式的开头是字段长度偏移列表，与变长字段长度列表有两处不同：

少了“变长”两个字：Redundant行格式会把该条记录中所有列（包括隐藏列）的长度信息都按照逆序存储到字段长度偏移列表。

多了“偏移”两个字：这意味着计算列值长度的方式不像Compact行格式那么直观，它是采用两个相邻数值的差值来计算各个列值的长度。

举例：比如第一条记录的字段长度偏移列表就是：

2B 25 1F 1B 13 0C 06

因为它是逆序排放的，所以按照列的顺序排列就是：

06 0C 13 17 1A 24 25...

- 记录头信息 (record header)

不同于Compact行格式，Redundant行格式中的记录头信息固定占用6个字节（48位），每位的含义见下表。

与Compact行格式的记录头信息对比来看，有两处不同：

Redundant行格式多了n_field和1byte_offs_flag这两个属性。

Redundant行格式没有record_type这个属性。

其中，n_fields：代表一行中列的数量，占用10位，这也很好地解释了为什么MySQL一个行支持最多的列为1023。另一个值为1byte_offs_flags，该值定义了偏移列表占用1个字节还是2个字节。当它的值为1时，表明使用1个字节存储。当它的值为0时，表明使用2个字节存储。

1byte_offs_flag的值是怎么选择的...

第08章_索引的创建与设计原则

讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

1. 索引的声明与使用

1.1 索引的分类

MySQL的索引包括普通索引、唯一性索引、全文索引、单列索引、多列索引和空间索引等。

- 从 **功能逻辑** 上说，索引主要有 4 种，分别是普通索引、唯一索引、主键索引、全文索引。
- 按照 **物理实现方式**，索引可以分为 2 种：聚簇索引和非聚簇索引。
- 按照 **作用字段个数** 进行划分，分成单列索引和联合索引。

1. 普通索引

2. 唯一性索引

3. 主键索引

4. 单列索引

5. 多列(组合、联合)索引

6. 全文索引

7. 补充：空间索引

小结：不同的存储引擎支持的索引类型也不一样 **InnoDB**：支持 B-tree、Full-text 等索引，不支持 Hash 索引； **MyISAM**：支持 B-tree、Full-text 等索引，不支持 Hash 索引； **Memory**：支持 B-tree、Hash 等索引，不支持 Full-text 索引； **NDB**：支持 Hash 索引，不支持 B-tree、Full-text 等索引； **Archive**：不支持 B-tree、Hash、Full-text 等索引；

1.2 创建索引

1. 创建表的时候创建索引

举例：

```
CREATE TABLE dept(
dept_id INT PRIMARY KEY AUTO_INCREMENT,
dept_name VARCHAR(20)
);

CREATE TABLE emp(
emp_id INT PRIMARY KEY AUTO_INCREMENT,
emp_name VARCHAR(20) UNIQUE,
dept_id INT,
CONSTRAINT emp_dept_id_fk FOREIGN KEY(dept_id) REFERENCES dept(dept_id)
);
```

但是，如果显式创建表时创建索引的话，基本语法格式如下：

```
CREATE TABLE table_name [col_name data_type]
[UNIQUE | FULLTEXT | SPATIAL] [INDEX | KEY] [index_name] (col_name [length]) [ASC | DESC]
```

- **UNIQUE**、**FULLTEXT** 和 **SPATIAL** 为可选参数，分别表示唯一索引、全文索引和空间索引；
- **INDEX** 与 **KEY** 为同义词，两者的作用相同，用来指定创建索引；
- **index_name** 指定索引的名称，为可选参数，如果不指定，那么MySQL默认**col_name**为索引名；
- **col_name** 为需要创建索引的字段列，该列必须从数据表中定义的多个列中选择；
- **length** 为可选参数，表示索引的长度，只有字符串类型的字段才能指定索引长度；
- **ASC** 或 **DESC** 指定升序或者降序的索引值存储。

1. 创建普通索引

在book表中的year_publication字段上建立普通索引，SQL语句如下：

```
CREATE TABLE book(
book_id INT ,
book_name VARCHAR(100),
authors VARCHAR(100),
info VARCHAR(100) ,
comment VARCHAR(100),
year_publication YEAR,
INDEX(year_publication)
);
```

2. 创建唯一索引

举例：

```
CREATE TABLE test1(
id INT NOT NULL,
name varchar(30) NOT NULL,
UNIQUE INDEX uk_idx_id(id)
);
```

该语句执行完毕之后，使用SHOW CREATE TABLE查看表结构：

```
SHOW INDEX FROM test1 \G
```

3. 主键索引

设定为主键后数据库会自动建立索引，innodb为聚簇索引，语法：

- 随表一起建索引：

```
CREATE TABLE student (
id INT(10) UNSIGNED AUTO_INCREMENT ,
student_no VARCHAR(200),
student_name VARCHAR(200),
PRIMARY KEY(id)
);
```

- 删除主键索引：

```
ALTER TABLE student
drop PRIMARY KEY ;
```

- 修改主键索引：必须先删除掉(drop)原索引，再新建(add)索引

4. 创建单列索引

举例：

```
CREATE TABLE test2(
    id INT NOT NULL,
    name CHAR(50) NULL,
    INDEX single_idx_name(name(20))
);
```

该语句执行完毕之后，使用SHOW CREATE TABLE查看表结构：

```
SHOW INDEX FROM test2 \G
```

5. 创建组合索引

举例：创建表test3，在表中的id、name和age字段上建立组合索引，SQL语句如下：

```
CREATE TABLE test3(
    id INT(11) NOT NULL,
    name CHAR(30) NOT NULL,
    age INT(11) NOT NULL,
    info VARCHAR(255),
    INDEX multi_idx(id, name, age)
);
```

该语句执行完毕之后，使用SHOW INDEX 查看：

```
SHOW INDEX FROM test3 \G
```

6. 创建全文索引

举例1：创建表test4，在表中的info字段上建立全文索引，SQL语句如下：

```
CREATE TABLE test4(
    id INT NOT NULL,
    name CHAR(30) NOT NULL,
    age INT NOT NULL,
    info VARCHAR(255),
    FULLTEXT INDEX futxt_idx_info(info)
) ENGINE=MyISAM;
```

在MySQL5.7及之后版本中可以不指定最后的ENGINE了，因为在此版本中InnoDB支持全文索引。

举例2：

```
CREATE TABLE articles (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR (200),
    body TEXT,
    FULLTEXT index (title, body)
) ENGINE = INNODB ;
```

创建了一个给title和body字段添加全文索引的表。

举例3：

```
CREATE TABLE `papers` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `title` varchar(200) DEFAULT NULL,
  `content` text,
  PRIMARY KEY (`id`),
  FULLTEXT KEY `title` (`title`, `content`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

不同于like方式的的查询：

```
SELECT * FROM papers WHERE content LIKE '%查询字符串%';
```

全文索引用match+against方式查询：

```
SELECT * FROM papers WHERE MATCH(title,content) AGAINST ('查询字符串');
```

注意点

1. 使用全文索引前，搞清楚版本支持情况；
2. 全文索引比 like + % 快 N 倍，但是可能存在精度问题；
3. 如果需要全文索引的是大量数据，建议先添加数据，再创建索引。

7. 创建空间索引

空间索引创建中，要求空间类型的字段必须为 **非空**。

举例：创建表test5，在空间类型为GEOMETRY的字段上创建空间索引，SQL语句如下：

```
CREATE TABLE test5(
geo GEOMETRY NOT NULL,
SPATIAL INDEX spa_idx_geo(geo)
) ENGINE=MyISAM;
```

2. 在已经存在的表上创建索引

在已经存在的表中创建索引可以使用ALTER TABLE语句或者CREATE INDEX语句。

1. 使用ALTER TABLE语句创建索引

ALTER TABLE语句创建索引的基本语法如下：

```
ALTER TABLE table_name ADD [UNIQUE | FULLTEXT | SPATIAL] [INDEX | KEY]
[index_name] (col_name[length],...) [ASC | DESC]
```

2. 使用CREATE INDEX创建索引 CREATE INDEX语句可以在已经存在的表上添加索引，在MySQL中，CREATE INDEX被映射到一个ALTER TABLE语句上，基本语法结构为：

```
CREATE [UNIQUE | FULLTEXT | SPATIAL] INDEX index_name
ON table_name (col_name[length],...) [ASC | DESC]
```

1.3 删除索引

1. 使用ALTER TABLE删除索引

ALTER TABLE删除索引的基本语法格式如下：

```
ALTER TABLE table_name DROP INDEX index_name;
```

2. 使用DROP INDEX语句删除索引

DROP INDEX删除索引的基本语法格式如下：

```
DROP INDEX index_name ON table_name;
```

提示删除表中的列时，如果要删除的列为索引的组成部分，则该列也会从索引中删除。如果组成索引的所有列都被删除，则整个索引将被删除。

2. MySQL8.0索引新特性

2.1 支持降序索引

举例：分别在MySQL 5.7版本和MySQL 8.0版本中创建数据表ts1，结果如下：

```
CREATE TABLE ts1(a int,b int,index idx_a_b(a,b desc));
```

在MySQL 5.7版本中查看数据表ts1的结构，结果如下：

```
mysql> show create table ts1\G
***** 1. row *****
      Table: ts1
Create Table: CREATE TABLE `ts1` (
  `a` int(11) DEFAULT NULL,
  `b` int(11) DEFAULT NULL,
  KEY `idx_a_b` (`a`,`b`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.01 sec)
```

从结果可以看出，索引仍然是默认的升序。

在MySQL 8.0版本中查看数据表ts1的结构，结果如下：

```
mysql> show create table ts1\G
***** 1. row *****
      Table: ts1
Create Table: CREATE TABLE `ts1` (
  `a` int DEFAULT NULL,
  `b` int DEFAULT NULL,
  KEY `idx_a_b` (`a`,`b` DESC)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
1 row in set (0.00 sec)
```

从结果可以看出，索引已经是降序了。下面继续测试降序索引在执行计划中的表现。

分别在MySQL 5.7版本和MySQL 8.0版本的数据表ts1中插入800条随机数据，执行语句如下：

```
DELIMITER //
CREATE PROCEDURE ts_insert()
BEGIN
    DECLARE i INT DEFAULT 1;
    WHILE i < 800
    DO
        insert into ts1 select rand()*80000,rand()*80000;
        SET i = i + 1;
    END WHILE;
    commit;
END //
DELIMITER ;
```

#调用

```
CALL ts_insert();
```

在MySQL 5.7版本中查看数据表ts1的执行计划，结果如下：

```
EXPLAIN SELECT * FROM ts1 ORDER BY a,b DESC LIMIT 5;
```

从结果可以看出，执行计划中扫描数为799，而且使用了Using filesort。

提示 Using filesort是MySQL中一种速度比较慢的外部排序，能避免是最好的。多数情况下，管理员可以通过优化索引来尽量避免出现Using filesort，从而提高数据库执行速度。

在MySQL 8.0版本中查看数据表ts1的执行计划。从结果可以看出，执行计划中扫描数为5，而且没有使用Using filesort。

注意 降序索引只对查询中特定的排序顺序有效，如果使用不当，反而查询效率更低。例如，上述查询排序条件改为order by a desc, b desc， MySQL 5.7的执行计划要明显好于MySQL 8.0。

将排序条件修改为order by a desc, b desc后，下面来对比不同版本中执行计划的效果。在MySQL 5.7版本中查看数据表ts1的执行计划，结果如下：

```
EXPLAIN SELECT * FROM ts1 ORDER BY a DESC,b DESC LIMIT 5;
```

在MySQL 8.0版本中查看数据表ts1的执行计划。

从结果可以看出，修改后MySQL 5.7的执行计划要明显好于MySQL 8.0。

2.2 隐藏索引

在MySQL 5.7版本及之前，只能通过显式的方式删除索引。此时，如果发现删除索引后出现错误，又只能通过显式创建索引的方式将删除的索引创建回来。如果数据表中的数据量非常大，或者数据表本身比较大，这种操作就会消耗系统过多的资源，操作成本非常高。

从MySQL 8.x开始支持 **隐藏索引 (invisible indexes)**，只需要将待删除的索引设置为隐藏索引，使查询优化器不再使用这个索引（即使使用force index（强制使用索引），优化器也不会使用该索引），确认将索引设置为隐藏索引后系统不受任何响应，就可以彻底删除索引。**这种通过先将索引设置为隐藏索引，再删除索引的方式就是软删除。**

1. 创建表时直接创建 在MySQL中创建隐藏索引通过SQL语句INVISIBLE来实现，其语法形式如下：

```
CREATE TABLE tablename(
    propname1 type1[CONSTRAINT1],
    propname2 type2[CONSTRAINT2],
    ....
    propnamen typen,
    INDEX [indexname](propname1 [(length)]) INVISIBLE
);
```

上述语句比普通索引多了一个关键字INVISIBLE，用来标记索引为不可见索引。

2. 在已经存在的表上创建

可以为已经存在的表设置隐藏索引，其语法形式如下：

```
CREATE INDEX indexname
ON tablename(propname[(length)]) INVISIBLE;
```

3. 通过ALTER TABLE语句创建

语法形式如下：

```
ALTER TABLE tablename  
ADD INDEX indexname (propname [(length)]) INVISIBLE;
```

4. 切换索引可见状态

已存在的索引可通过如下语句切换可见状态：

```
ALTER TABLE tablename ALTER INDEX index_name INVISIBLE; #切换成隐藏索引  
ALTER TABLE tablename ALTER INDEX index_name VISIBLE; #切换成非隐藏索引
```

如果将index_cname索引切换成可见状态，通过explain查看执行计划，发现优化器选择了index_cname索引。

注意当索引被隐藏时，它的内容仍然是和正常索引一样实时更新的。如果一个索引需要长期被隐藏，那么可以将其删除，因为索引的存在会影响插入、更新和删除的性能。

通过设置隐藏索引的可见性可以查看索引对调优的帮助。

5. 使隐藏索引对查询优化器可见

在MySQL 8.x版本中，为索引提供了一种新的测试方式，可以通过查询优化器的一个开关（use_invisible_indexes）来打开某个设置，使隐藏索引对查询优化器可见。如果use_invisible_indexes设置为off（默认），优化器会忽略隐藏索引。如果设置为on，即使隐藏索引不可见，优化器在生成执行计划时仍会考虑使用隐藏索引。

(1) 在MySQL命令行执行如下命令查看查询优化器的开关设置。

```
mysql> select @@optimizer_switch \G
```

在输出的结果信息中找到如下属性配置。

```
use_invisible_indexes=off
```

此属性配置值为off，说明隐藏索引默认对查询优化器不可见。

(2) 使隐藏索引对查询优化器可见，需要在MySQL命令行执行如下命令：

```
mysql> set session optimizer_switch="use_invisible_indexes=on";  
Query OK, 0 rows affected (0.00 sec)
```

SQL语句执行成功，再次查看查询优化器的开关设置。

```
mysql> select @@optimizer_switch \G  
*****  
@@optimizer_switch:  
index_merge=on, index_merge_union=on, index_merge_sort_union=on, index_merge_ intersection=on, engine_condition_pushdown=on, index_condition_pushdown=on, mrr=on, mrr_cost_based=on, block_nested_loop=on, batched_key_access=off, materialization=on, semijoin=on, loosescan=on, firstmatch=on, duplicateweedout=on, subquery_materialization_cost_based=on, use_index_extensions=on, condition_fanout_filter=on, derived_merge=on, use_invisible_indexes=on, skip_scan=on, hash_join=on  
1 row in set (0.00 sec)
```

此时，在输出结果中可以看到如下属性配置。

```
use_invisible_indexes=on
```

use_invisible_indexes属性的值为on，说明此时隐藏索引对查询优化器可见。

(3) 使用EXPLAIN查看以字段invisible_column作为查询条件时的索引使用情况。

```
explain select * from classes where cname = '高一2班';
```

查询优化器会使用隐藏索引来查询数据。

(4) 如果需要使隐藏索引对查询优化器不可见，则只需要执行如下命令即可。

```
mysql> set session optimizer_switch="use_invisible_indexes=off";
Query OK, 0 rows affected (0.00 sec)
```

再次查看查询优化器的开关设置。

```
mysql> select @@optimizer_switch \G
```

此时，use_invisible_indexes属性的值已经被设置为“off”。

3. 索引的设计原则

3.1 数据准备

第1步：创建数据库、创建表

```
CREATE DATABASE atguigudb1;

USE atguigudb1;

#1. 创建学生表和课程表
CREATE TABLE `student_info` (
  `id` INT(11) NOT NULL AUTO_INCREMENT,
  `student_id` INT NOT NULL ,
  `name` VARCHAR(20) DEFAULT NULL,
  `course_id` INT NOT NULL ,
  `class_id` INT(11) DEFAULT NULL,
  `create_time` DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`)
) ENGINE=INNODB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;

CREATE TABLE `course` (
  `id` INT(11) NOT NULL AUTO_INCREMENT,
  `course_id` INT NOT NULL ,
  `course_name` VARCHAR(40) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=INNODB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

第2步：创建模拟数据必需的存储函数

```
#函数1：创建随机产生字符串函数

DELIMITER //
CREATE FUNCTION rand_string(n INT)
    RETURNS VARCHAR(255) #该函数会返回一个字符串
BEGIN
    DECLARE chars_str VARCHAR(100) DEFAULT
        'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ';
    DECLARE return_str VARCHAR(255) DEFAULT '';
    -- 生成 n 个字符的字符串
    SET return_str = '';
    WHILE n > 0 DO
        SET n = n - 1;
        SET return_str = concat(return_str, substr(chars_str, rand() % 100 + 1, 1));
    END WHILE;
    RETURN return_str;
END//
```

```
DECLARE i INT DEFAULT 0;
WHILE i < n DO
    SET return_str =CONCAT(return_str,SUBSTRING(chars_str,FLOOR(1+RAND()*52),1));
    SET i = i + 1;
END WHILE;
RETURN return_str;
END //
DELIMITER ;
```

```
#函数2：创建随机数函数
DELIMITER //
CREATE FUNCTION rand_num (from_num INT ,to_num INT) RETURNS INT(11)
BEGIN
DECLARE i INT DEFAULT 0;
SET i = FLOOR(from_num +RAND()*(to_num - from_num+1)) ;
RETURN i;
END //
DELIMITER ;
```

创建函数，假如报错：

```
This function has none of DETERMINISTIC.....
```

由于开启过慢查询日志bin-log,我们就必须为我们的function指定一个参数。

主从复制，主机会将写操作记录在bin-log日志中。从机读取bin-log日志，执行语句来同步数据。如果使用函数来操作数据，会导致从机和主键操作时间不一致。所以，默认情况下，mysql不开启创建函数设置。

- 查看mysql是否允许创建函数：

```
show variables like 'log_bin_trust_function_creators';
```

- 命令开启：允许创建函数设置：

```
set global log_bin_trust_function_creators=1;      # 不加global只是当前窗口有效。
```

- mysqld重启，上述参数又会消失。永久方法：

- windows下：my.ini[mysqld]加上：

```
log_bin_trust_function_creators=1
```

- linux下：/etc/my.cnf下my.cnf[mysqld]加上：

```
log_bin_trust_function_creators=1
```

第3步：创建插入模拟数据的存储过程

```
# 存储过程1：创建插入课程表存储过程
DELIMITER //
CREATE PROCEDURE insert_course( max_num INT )
BEGIN
DECLARE i INT DEFAULT 0;
SET autocommit = 0;      #设置手动提交事务
REPEAT #循环
SET i = i + 1; #赋值
INSERT INTO course (course_id, course_name ) VALUES
(rand_num(10000,10100),rand_string(6));
UNTIL i = max_num
```

```

END REPEAT;
COMMIT; #提交事务
END //
DELIMITER ;

# 存储过程2：创建插入学生信息表存储过程
DELIMITER //
CREATE PROCEDURE insert_stu( max_num INT )
BEGIN
DECLARE i INT DEFAULT 0;
SET autocommit = 0;      #设置手动提交事务
REPEAT #循环
SET i = i + 1; #赋值
INSERT INTO student_info (course_id, class_id ,student_id ,NAME ) VALUES
(rand_num(10000,10100),rand_num(10000,10200),rand_num(1,200000),rand_string(6));
UNTIL i = max_num
END REPEAT;
COMMIT; #提交事务
END //
DELIMITER ;

```

第4步：调用存储过程

```

CALL insert_course(100);

CALL insert_stu(1000000);

```

3.2 哪些情况适合创建索引

1. 字段的数值有唯一性的限制

业务上具有唯一特性的字段，即使是组合字段，也必须建成唯一索引。（来源：Alibaba）

说明：不要以为唯一索引影响了 insert 速度，这个速度损耗可以忽略，但提高查找速度是明显的。

2. 频繁作为 WHERE 查询条件的字段

某个字段在SELECT语句的 WHERE 条件中经常被使用到，那么就需要给这个字段创建索引了。尤其是在数据量大的情况下，创建普通索引就可以大幅提升数据查询的效率。

比如student_info数据表（含100万条数据），假设我们想要查询 student_id=123110 的用户信息。

3. 经常 GROUP BY 和 ORDER BY 的列

索引就是让数据按照某种顺序进行存储或检索，因此当我们使用 GROUP BY 对数据进行分组查询，或者使用 ORDER BY 对数据进行排序的时候，就需要 **对分组或者排序的字段进行索引**。如果待排序的列有多个，那么可以在这些列上建立 **组合索引**。

4. UPDATE、DELETE 的 WHERE 条件列

对数据按照某个条件进行查询后再进行 UPDATE 或 DELETE 的操作，如果对 WHERE 字段创建了索引，就能大幅提升效率。原理是因为我们需要先根据 WHERE 条件列检索出来这条记录，然后再对它进行更新或删除。**如果进行更新的时候，更新的字段是非索引字段，提升的效率会更明显，这是因为非索引字段更新不需要对索引进行维护。**

5.DISTINCT 字段需要创建索引

有时候我们需要对某个字段进行去重，使用 DISTINCT，那么对这个字段创建索引，也会提升查询效率。

比如，我们想要查询课程表中不同的 student_id 都有哪些，如果我们没有对 student_id 创建索引，执行 SQL 语句：

```
SELECT DISTINCT(student_id) FROM `student_info`;
```

运行结果 (600637 条记录，运行时间 0.683s) :

如果我们对 student_id 创建索引，再执行 SQL 语句：

```
SELECT DISTINCT(student_id) FROM `student_info`;
```

运行结果 (600637 条记录，运行时间 0.010s) :

你能看到 SQL 查询效率有了提升，同时显示出来的 student_id 还是按照 递增的顺序 进行展示的。这是因为索引会对数据按照某种顺序进行排序，所以在去重的时候也会快很多。

6.多表 JOIN 连接操作时，创建索引注意事项

首先，**连接表的数量尽量不要超过 3 张**，因为每增加一张表就相当于增加了一次嵌套的循环，数量级增长会非常快，严重影响查询的效率。

其次，**对 WHERE 条件创建索引**，因为 WHERE 才是对数据条件的过滤。如果在数据量非常大的情况下，没有 WHERE 条件过滤是非常可怕的。

最后，**对用于连接的字段创建索引**，并且该字段在多张表中的**类型必须一致**。比如 course_id 在 student_info 表和 course 表中都为 int(11) 类型，而不能一个为 int 另一个为 varchar 类型。

举个例子，如果我们只对 student_id 创建索引，执行 SQL 语句：

```
SELECT course_id, name, student_info.student_id, course_name
FROM student_info JOIN course
ON student_info.course_id = course.course_id
WHERE name = '462eed7ac6e791292a79';
```

运行结果 (1 条数据，运行时间 0.189s) :

这里我们对 name 创建索引，再执行上面的 SQL 语句，运行时间为 0.002s。

7. 使用列的类型小的创建索引

8. 使用字符串前缀创建索引

创建一张商户表，因为地址字段比较长，在地址字段上建立前缀索引

```
create table shop(address varchar(120) not null);

alter table shop add index(address(12));
```

问题是，截取多少呢？截取得多了，达不到节省索引存储空间的目的；截取得少了，重复内容太多，字段的散列度(选择性)会降低。**怎么计算不同的长度的选择性呢？**

先看一下字段在全部数据中的选择度：

```
select count(distinct address) / count(*) from shop;
```

通过不同长度去计算，与全表的选择性对比：

公式：

```
count(distinct left(列名, 索引长度))/count(*)
```

例如：

```
select count(distinct left(address,10)) / count(*) as sub10, -- 截取前10个字符的选择度
count(distinct left(address,15)) / count(*) as sub11, -- 截取前15个字符的选择度
count(distinct left(address,20)) / count(*) as sub12, -- 截取前20个字符的选择度
count(distinct left(address,25)) / count(*) as sub13 -- 截取前25个字符的选择度
from shop;
```

引申另一个问题：索引列前缀对排序的影响

拓展：Alibaba 《Java开发手册》

【强制】在 varchar 字段上建立索引时，必须指定索引长度，没必要对全字段建立索引，根据实际文本区分度决定索引长度。

说明：索引的长度与区分度是一对矛盾体，一般对字符串类型数据，长度为 20 的索引，区分度会 **高达 90% 以上**，可以使用 `count(distinct left(列名, 索引长度))/count(*)` 的区分度来确定。

9. 区分度高(散列性高)的列适合作为索引

10. 使用最频繁的列放到联合索引的左侧

这样也可以较少的建立一些索引。同时，由于“最左前缀原则”，可以增加联合索引的使用率。

11. 在多个字段都要创建索引的情况下，联合索引优于单值索引

3.3 限制索引的数目

3.4 哪些情况不适合创建索引

1. 在where中使用不到的字段，不要设置索引

2. 数据量小的表最好不要使用索引

举例：创建表1：

```
CREATE TABLE t_without_index(
    a INT PRIMARY KEY AUTO_INCREMENT,
    b INT
);
```

提供存储过程1：

```
#创建存储过程
DELIMITER //
CREATE PROCEDURE t_wout_insert()
BEGIN
    DECLARE i INT DEFAULT 1;
    WHILE i <= 900
    DO
        INSERT INTO t_without_index(b) SELECT RAND()*10000;
        SET i = i + 1;
    END WHILE;
```

```
    COMMIT;
END //
DELIMITER ;

#调用
CALL t_wout_insert();
```

创建表2：

```
CREATE TABLE t_with_index(
a INT PRIMARY KEY AUTO_INCREMENT,
b INT,
INDEX idx_b(b)
);
```

创建存储过程2：

```
#创建存储过程
DELIMITER //
CREATE PROCEDURE t_with_insert()
BEGIN
DECLARE i INT DEFAULT 1;
WHILE i <= 900
DO
    INSERT INTO t_with_index(b) SELECT RAND()*10000;
    SET i = i + 1;
END WHILE;
COMMIT;
END //
DELIMITER ;

#调用
CALL t_with_insert();
```

查询对比：

```
mysql> select * from t_without_index where b = 9879;
+----+----+
| a | b |
+----+----+
| 1242 | 9879 |
+----+----+
1 row in set (0.00 sec)

mysql> select * from t_with_index where b = 9879;
+----+----+
| a | b |
+----+----+
| 112 | 9879 |
+----+----+
1 row in set (0.00 sec)
```

你能看到运行结果相同，但是在数据量不大的情况下，索引就发挥不出作用了。

结论：在数据表中的数据行数比较少的情况下，比如不到 1000 行，是不需要创建索引的。

3. 有大量重复数据的列上不要建立索引

举例1：要在 100 万行数据中查找其中的 50 万行（比如性别为男的数据），一旦创建了索引，你需要先访问 50 万次索引，然后再访问 50 万次数据表，这样加起来的开销比不使用索引可能还要大。

举例2：假设有一个学生表，学生总数为 100 万人，男性只有 10 个人，也就是占总人口的 10 万分之 1。

学生表 student_gender 结构如下。其中数据表中的 student_gender 字段取值为 0 或 1，0 代表女性，1 代表男性。

```
CREATE TABLE student_gender(
    student_id INT(11) NOT NULL,
    student_name VARCHAR(50) NOT NULL,
    student_gender TINYINT(1) NOT NULL,
    PRIMARY KEY(student_id)
)ENGINE = INNODB;
```

如果我们要筛选出这个学生表中的男性，可以使用：

```
SELECT * FROM student_gender WHERE student_gender = 1
```

运行结果（10 条数据，运行时间 **0.696s**）：

student_id	student_name	student_gender
110000	student_100000	1
210000	student_200000	1
.....
1010000	student_1000000	1

结论：当数据重复度大，比如 **高于 10%** 的时候，也不需要对这个字段使用索引。

4. 避免对经常更新的表创建过多的索引

5. 不建议用无序的值作为索引

例如身份证、UUID(在索引比较时需要转为ASCII，并且插入时可能造成页分裂)、MD5、HASH、无序长字符串等。

6. 删除不再使用或者很少使用的索引

7. 不要定义冗余或重复的索引

① 冗余索引

举例：建表语句如下

```
CREATE TABLE person_info(
    id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    birthday DATE NOT NULL,
    phone_number CHAR(11) NOT NULL,
    country varchar(100) NOT NULL,
    PRIMARY KEY (id),
    KEY idx_name_birthday_phone_number (name(10), birthday, phone_number),
    KEY idx_name (name(10))
);
```

我们知道，通过 `idx_name_birthday_phone_number` 索引就可以对 `name` 列进行快速搜索，再创建一个专门针对 `name` 列的索引就算是一个 **冗余索引**，维护这个索引只会增加维护的成本，并不会对搜索有什么好处。

② 重复索引

另一种情况，我们可能会对某个列 **重复建立索引**，比方说这样：

```
CREATE TABLE repeat_index_demo (
    col1 INT PRIMARY KEY,
    col2 INT,
    UNIQUE uk_idx_c1 (col1),
    INDEX idx_c1 (col1)
);
```

我们看到，`col1` 既是主键、又给它定义为一个唯一索引，还给它定义了一个普通索引，可是主键本身就会生成聚簇索引，所以定义的唯一索引和普通索引是重复的，这种情况要避免。

第09章_性能分析工具的使用

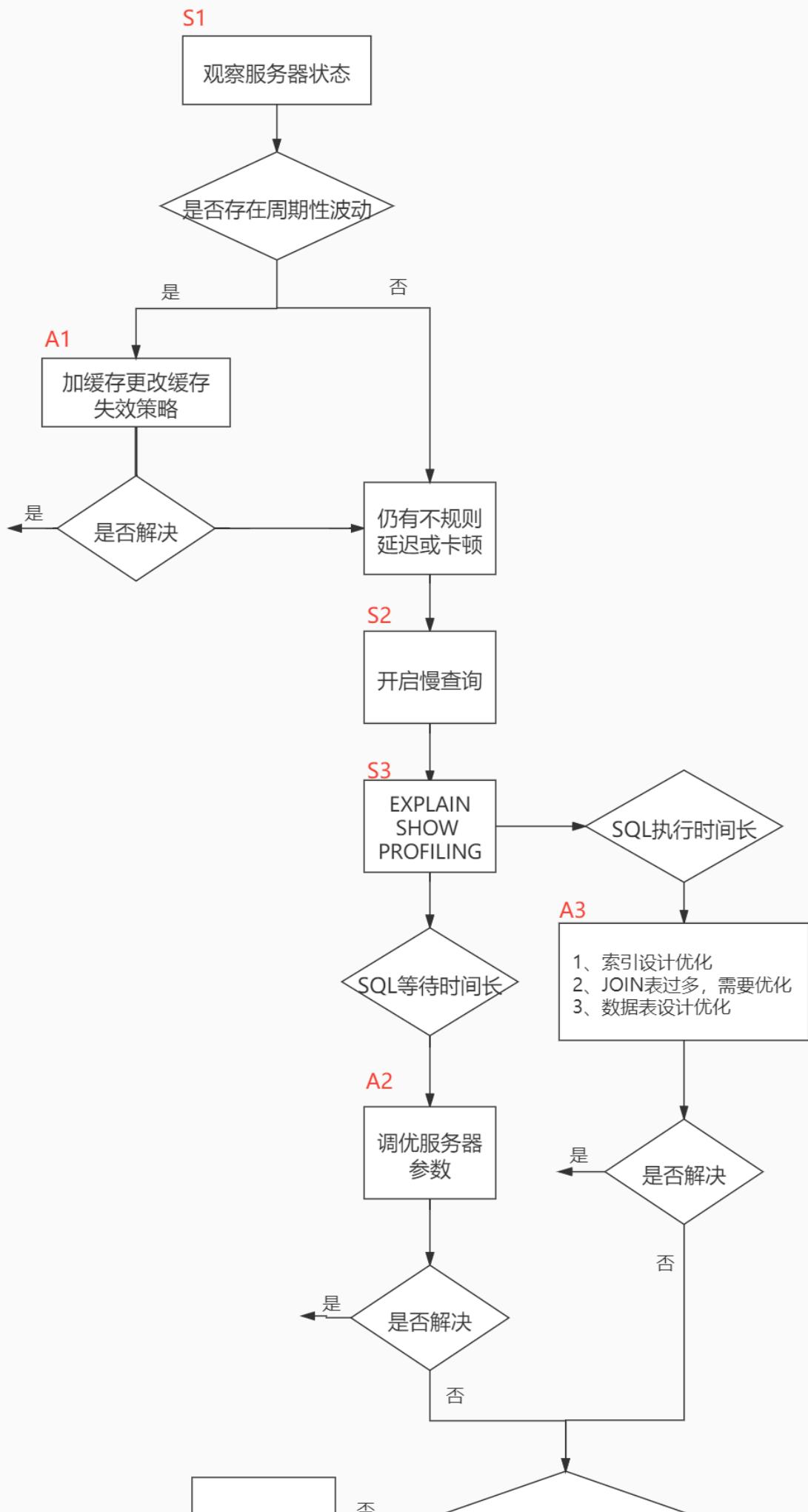
讲师：尚硅谷·宋红康（江湖人称：康师傅）

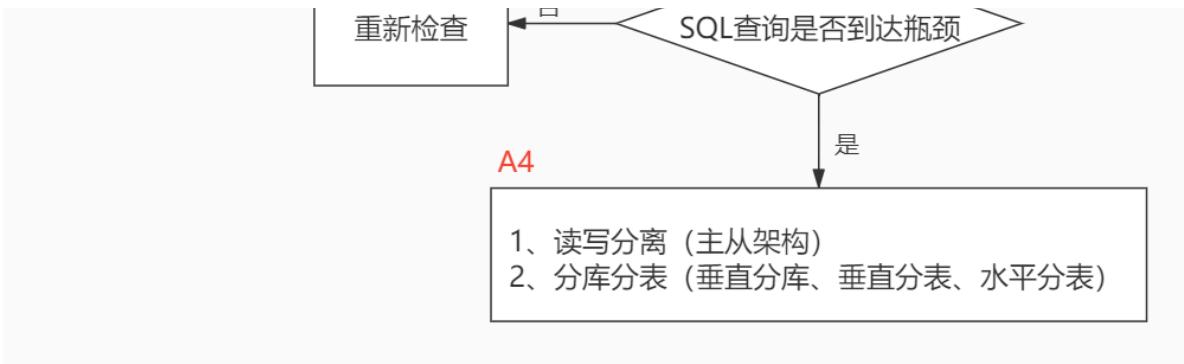
官网：<http://www.atguigu.com>

1. 数据库服务器的优化步骤

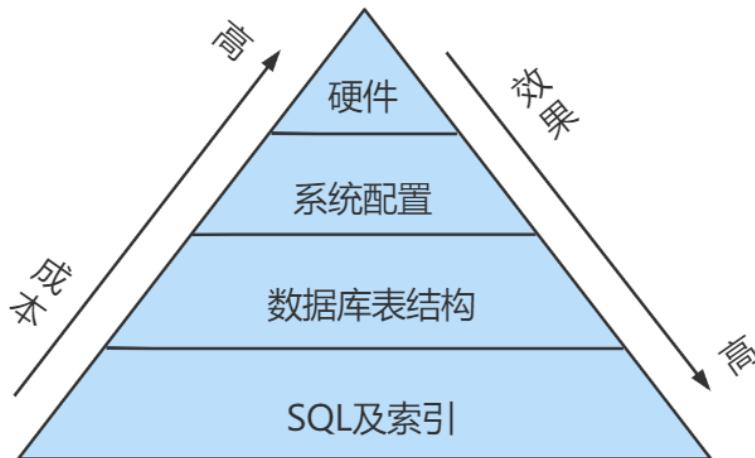
当我们遇到数据库调优问题的时候，该如何思考呢？这里把思考的流程整理成下面这张图。

整个流程划分成了 观察（Show status） 和 行动（Action） 两个部分。字母 S 的部分代表观察（会使用相应的分析工具），字母 A 代表的部分是行动（对应分析可以采取的行动）。





小结:



2. 查看系统性能参数

在MySQL中，可以使用 `SHOW STATUS` 语句查询一些MySQL数据库服务器的 性能参数 、 执行频率 。

`SHOW STATUS`语句语法如下：

```
SHOW [GLOBAL|SESSION] STATUS LIKE '参数';
```

一些常用的性能参数如下：

- `Connections`: 连接MySQL服务器的次数。
- `Uptime`: MySQL服务器的上线时间。
- `Slow_queries`: 慢查询的次数。
- `Innodb_rows_read`: Select查询返回的行数。
- `Innodb_rows_inserted`: 执行INSERT操作插入的行数。
- `Innodb_rows_updated`: 执行UPDATE操作更新的行数。
- `Innodb_rows_deleted`: 执行DELETE操作删除的行数。
- `Com_select`: 查询操作的次数。
- `Com_insert`: 插入操作的次数。对于批量插入的 `INSERT` 操作，只累加一次。
- `Com_update`: 更新操作的次数。
- `Com_delete`: 删除操作的次数。

3. 统计SQL的查询成本：last_query_cost

我们依然使用第8章的 `student_info` 表为例：

```
CREATE TABLE `student_info` (
  `id` INT(11) NOT NULL AUTO_INCREMENT,
  `student_id` INT NOT NULL ,
  `name` VARCHAR(20) DEFAULT NULL,
  `course_id` INT NOT NULL ,
  `class_id` INT(11) DEFAULT NULL,
  `create_time` DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`)
) ENGINE=INNODB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

如果我们想要查询 id=900001 的记录，然后看下查询成本，我们可以直接在聚簇索引上进行查找：

```
SELECT student_id, class_id, NAME, create_time FROM student_info
WHERE id = 900001;
```

运行结果 (1 条记录，运行时间为 0.042s)

然后再看下查询优化器的成本，实际上我们只需要检索一个页即可：

```
mysql> SHOW STATUS LIKE 'last_query_cost';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Last_query_cost | 1.000000 |
+-----+-----+
```

如果我们想要查询 id 在 900001 到 9000100 之间的学生记录呢？

```
SELECT student_id, class_id, NAME, create_time FROM student_info
WHERE id BETWEEN 900001 AND 900100;
```

运行结果 (100 条记录，运行时间为 0.046s)：

然后再看下查询优化器的成本，这时我们大概需要进行 20 个页的查询。

```
mysql> SHOW STATUS LIKE 'last_query_cost';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Last_query_cost | 21.134453 |
+-----+-----+
```

你能看到页的数量是刚才的 20 倍，但是查询的效率并没有明显的变化，实际上这两个 SQL 查询的时间基本上一样，就是因为采用了顺序读取的方式将页面一次性加载到缓冲池中，然后再进行查找。虽然页数量 (last_query_cost) 增加了不少，但是通过缓冲池的机制，并没有增加多少查询时间。

使用场景：它对于比较开销是非常有用的，特别是我们有好几种查询方式可选的时候。

4. 定位执行慢的 SQL：慢查询日志

4.1 开启慢查询日志参数

1. 开启slow_query_log

```
mysql > set global slow_query_log='ON' ;
```

然后我们再来查看下慢查询日志是否开启，以及慢查询日志文件的位置：

```
mysql> show variables like '%slow_query_log%';
+-----+-----+
| Variable_name      | Value
+-----+-----+
| slow_query_log     | ON
| slow_query_log_file | /var/lib/mysql/atguigu02-slow.log
+-----+-----+
2 rows in set (0.00 sec)
```

你能看到这时慢查询分析已经开启，同时文件保存在 `/var/lib/mysql/atguigu02-slow.log` 文件中。

2. 修改long_query_time阈值

接下来我们来看下慢查询的时间阈值设置，使用如下命令：

```
mysql > show variables like '%long_query_time%' ;
```

```
mysql> show variables like '%long_query_time%';
+-----+-----+
| Variable_name    | Value
+-----+-----+
| long_query_time  | 10.000000
+-----+-----+
1 row in set (0.01 sec)
```

这里如果我们想把时间缩短，比如设置为 1 秒，可以这样设置：

```
#测试发现：设置global的方式对当前session的long_query_time失效。对新连接的客户端有效。所以可以一并
#执行下述语句
mysql > set global long_query_time = 1;
mysql> show global variables like '%long_query_time%' ;

mysql> set long_query_time=1;
mysql> show variables like '%long_query_time%' ;
```

```
mysql> set global long_query_time = 1;
Query OK, 0 rows affected (0.00 sec)

mysql> show global variables like '%long_query_time%' ;
+-----+-----+
| Variable_name    | Value
+-----+-----+
| long_query_time  | 1.000000
+-----+-----+
1 row in set (0.00 sec)
```

4.2 查看慢查询数目

查询当前系统中有多少条慢查询记录

```
SHOW GLOBAL STATUS LIKE '%Slow_queries';
```

4.3 案例演示

步骤1. 建表

```
CREATE TABLE `student` (
  `id` INT(11) NOT NULL AUTO_INCREMENT,
  `stuno` INT NOT NULL ,
  `name` VARCHAR(20) DEFAULT NULL,
  `age` INT(3) DEFAULT NULL,
  `classId` INT(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=INNODB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

步骤2：设置参数 log_bin_trust_function_creators

创建函数，假如报错：

```
This function has none of DETERMINISTIC.....
```

- 命令开启：允许创建函数设置：

```
set global log_bin_trust_function_creators=1;      # 不加global只是当前窗口有效。
```

步骤3：创建函数

随机产生字符串：（同上一章）

```
DELIMITER //
CREATE FUNCTION rand_string(n INT)
    RETURNS VARCHAR(255) #该函数会返回一个字符串
BEGIN
    DECLARE chars_str VARCHAR(100) DEFAULT
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ';
    DECLARE return_str VARCHAR(255) DEFAULT '';
    DECLARE i INT DEFAULT 0;
    WHILE i < n DO
        SET return_str =CONCAT(return_str,SUBSTRING(chars_str,FL00R(1+RAND()*52),1));
        SET i = i + 1;
    END WHILE;
    RETURN return_str;
END //
DELIMITER ;
```



```
#测试
SELECT rand_string(10);
```

产生随机数值：（同上一章）

```

DELIMITER //
CREATE FUNCTION rand_num (from_num INT ,to_num INT) RETURNS INT(11)
BEGIN
DECLARE i INT DEFAULT 0;
SET i = FLOOR(from_num +RAND()*(to_num - from_num+1)) ;
RETURN i;
END //
DELIMITER ;

#测试:
SELECT rand_num(10,100);

```

步骤4：创建存储过程

```

DELIMITER //
CREATE PROCEDURE insert_stu1( START INT , max_num INT )
BEGIN
DECLARE i INT DEFAULT 0;
SET autocommit = 0;      #设置手动提交事务
REPEAT #循环
SET i = i + 1; #赋值
INSERT INTO student (stuno, NAME ,age ,classId ) VALUES
((START+i),rand_string(6),rand_num(10,100),rand_num(10,1000));
UNTIL i = max_num
END REPEAT;
COMMIT; #提交事务
END //
DELIMITER ;

```

步骤5：调用存储过程

```

#调用刚刚写好的函数，4000000条记录，从100001号开始
CALL insert_stu1(100001,4000000);

```

4.4 测试及分析

1. 测试

```

mysql> SELECT * FROM student WHERE stuno = 3455655;
+-----+-----+-----+-----+
| id   | stuno | name  | age  | classId |
+-----+-----+-----+-----+
| 3523633 | 3455655 | oQmLUr | 19 | 39 |
+-----+-----+-----+-----+
1 row in set (2.09 sec)

mysql> SELECT * FROM student WHERE name = 'oQmLUr';
+-----+-----+-----+-----+
| id   | stuno | name  | age  | classId |
+-----+-----+-----+-----+
| 1154002 | 1243200 | OQM1UR | 266 | 28 |
| 1405708 | 1437740 | OQM1UR | 245 | 439 |
| 1748070 | 1680092 | OQM1UR | 240 | 414 |
| 2119892 | 2051914 | oQmLUr | 17 | 32 |
| 2893154 | 2825176 | OQM1UR | 245 | 435 |
| 3523633 | 3455655 | oQmLUr | 19 | 39 |
+-----+-----+-----+-----+

```

```
6 rows in set (2.39 sec)
```

从上面的结果可以看出来，查询学生编号为“3455655”的学生信息花费时间为2.09秒。查询学生姓名为“oQmLUR”的学生信息花费时间为2.39秒。已经达到了秒的数量级，说明目前查询效率是比较低的，下面的小节我们分析一下原因。

2. 分析

```
show status like 'slow_queries';
```

4.5 慢查询日志分析工具：mysqldumpslow

在生产环境中，如果要手工分析日志，查找、分析SQL，显然是个体力活，MySQL提供了日志分析工具 `mysqldumpslow`。

查看`mysqldumpslow`的帮助信息

```
mysqldumpslow --help
```

```
[root@zhangyu mysql]# mysqldumpslow --help
Usage: mysqldumpslow [ OPTS... ] [ LOGS... ]

Parse and summarize the MySQL slow query log. Options are

--verbose      verbose
--debug        debug
--help         write this text to standard output

-v             verbose
-d             debug
-s ORDER      what to sort by (al, at, ar, c, l, r, t), 'at' is default
              al: average lock time
              ar: average rows sent
              at: average query time
              c: count
              l: lock time
              r: rows sent
              t: query time
-r             reverse the sort order (largest last instead of first)
-t NUM        just show the top n queries
-a             don't abstract all numbers to N and strings to 'S'
-n NUM        abstract numbers with at least n digits within names
-g PATTERN    grep: only consider stmts that include this string
-h HOSTNAME   hostname of db server for *-slow.log filename (can be wildcard),
              default is '*', i.e. match all
-i NAME       name of server instance (if using mysql.server startup script)
-l             don't subtract lock time from total time
```

`mysqldumpslow` 命令的具体参数如下：

- `-a`: 不将数字抽象成N，字符串抽象成S

- `-s`: 是表示按照何种方式排序：

- `c`: 访问次数
- `l`: 锁定时间
- `r`: 返回记录
- **t: 查询时间**
 - `al`: 平均锁定时间
 - `ar`: 平均返回记录数
 - `at`: 平均查询时间（默认方式）
 - `ac`: 平均查询次数

- `-t`: 即为返回前面多少条的数据；

- -g: 后边搭配一个正则匹配模式，大小写不敏感的；

举例：我们想要按照查询时间排序，查看前五条 SQL 语句，这样写即可：

```
mysqldumpslow -s t -t 5 /var/lib/mysql/atguigu01-slow.log

[root@bogon ~]# mysqldumpslow -s t -t 5 /var/lib/mysql/atguigu01-slow.log

Reading mysql slow query log from /var/lib/mysql/atguigu01-slow.log
Count: 1  Time=2.39s (2s)  Lock=0.00s (0s)  Rows=13.0 (13), root[root]@localhost
  SELECT * FROM student WHERE name = 'S'

Count: 1  Time=2.09s (2s)  Lock=0.00s (0s)  Rows=2.0 (2), root[root]@localhost
  SELECT * FROM student WHERE stuno = N

Died at /usr/bin/mysqldumpslow line 162, <> chunk 2.
```

工作常用参考：

```
#得到返回记录集最多的10个SQL
mysqldumpslow -s r -t 10 /var/lib/mysql/atguigu-slow.log

#得到访问次数最多的10个SQL
mysqldumpslow -s c -t 10 /var/lib/mysql/atguigu-slow.log

#得到按照时间排序的前10条里面含有左连接的查询语句
mysqldumpslow -s t -t 10 -g "left join" /var/lib/mysql/atguigu-slow.log

#另外建议在使用这些命令时结合 | 和more 使用，否则有可能出现爆屏情况
mysqldumpslow -s r -t 10 /var/lib/mysql/atguigu-slow.log | more
```

4.6 关闭慢查询日志

MySQL服务器停止慢查询日志功能有两种方法：

方式1：永久性方式

```
[mysqld]
slow_query_log=OFF
```

或者，把slow_query_log一项注释掉或删除

```
[mysqld]
#slow_query_log =OFF
```

重启MySQL服务，执行如下语句查询慢日志功能。

```
SHOW VARIABLES LIKE '%slow%'; #查询慢查询日志所在目录
SHOW VARIABLES LIKE '%long_query_time%'; #查询超时时长
```

方式2：临时性方式

使用SET语句来设置。 (1) 停止MySQL慢查询日志功能，具体SQL语句如下。

```
SET GLOBAL slow_query_log=off;
```

(2) 重启MySQL服务，使用SHOW语句查询慢查询日志功能信息，具体SQL语句如下

```
SHOW VARIABLES LIKE '%slow%';
#以及
SHOW VARIABLES LIKE '%long_query_time%';
```

4.7 删除慢查询日志

5. 查看 SQL 执行成本：SHOW PROFILE

```
mysql > show variables like 'profiling';
```

```
mysql> show variables like 'profiling';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| profiling     | OFF   |
+-----+-----+
1 row in set (0.00 sec)
```

通过设置 `profiling='ON'` 来开启 show profile：

```
mysql > set profiling = 'ON';
```

```
mysql> set profiling = 'ON';
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> show variables like 'profiling';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| profiling     | ON    |
+-----+-----+
1 row in set (0.00 sec)
```

然后执行相关的查询语句。接着看下当前会话都有哪些 profiles，使用下面这条命令：

```
mysql > show profiles;
```

```
mysql> show profiles;
+-----+-----+-----+
| Query_ID | Duration | Query
+-----+-----+-----+
|       1 | 0.00107850 | show variables like 'profiling' |
|       2 | 0.00292675 | select * from employees
+-----+-----+-----+
2 rows in set, 1 warning (0.00 sec)
```

你能看到当前会话一共有 2 个查询。如果我们想要查看最近一次查询的开销，可以使用：

```
mysql > show profile;
```

```

mysql> show profile;
+-----+-----+
| Status          | Duration |
+-----+-----+
| starting        | 0.000102 |
| Executing hook on transaction | 0.000003 |
| starting        | 0.000007 |
| checking permissions | 0.000006 |
| Opening tables   | 0.000029 |
| init            | 0.000004 |
| System lock      | 0.000007 |
| optimizing       | 0.000003 |
| statistics       | 0.000013 |
| preparing        | 0.000011 |
| executing        | 0.002616 |
| end              | 0.000008 |
| query end        | 0.000004 |
| waiting for handler commit | 0.000006 |
| closing tables    | 0.000007 |
| freeing items     | 0.000090 |
| cleaning up       | 0.000013 |
+-----+-----+
17 rows in set, 1 warning (0.00 sec)

```

```
mysql> show profile cpu,block io for query 2;
```

```

mysql> show profile cpu,block io for query 2;
+-----+-----+-----+-----+-----+-----+
| Status          | Duration | CPU_user | CPU_system | Block_ops_in | Block_ops_out |
+-----+-----+-----+-----+-----+-----+
| starting        | 0.000102 | 0.000018 | 0.000057 | 0           | 0           |
| Executing hook on transaction | 0.000003 | 0.000000 | 0.000002 | 0           | 0           |
| starting        | 0.000007 | 0.000002 | 0.000006 | 0           | 0           |
| checking permissions | 0.000006 | 0.000001 | 0.000004 | 0           | 0           |
| Opening tables   | 0.000029 | 0.000007 | 0.000023 | 0           | 0           |
| init            | 0.000004 | 0.000001 | 0.000003 | 0           | 0           |
| System lock      | 0.000007 | 0.000002 | 0.000005 | 0           | 0           |
| optimizing       | 0.000003 | 0.000001 | 0.000002 | 0           | 0           |
| statistics       | 0.000013 | 0.000003 | 0.000010 | 0           | 0           |
| preparing        | 0.000011 | 0.000002 | 0.000008 | 0           | 0           |
| executing        | 0.002616 | 0.000341 | 0.001084 | 288         | 0           |
| end              | 0.000008 | 0.000001 | 0.000004 | 0           | 0           |
| query end        | 0.000004 | 0.000001 | 0.000003 | 0           | 0           |
| waiting for handler commit | 0.000006 | 0.000001 | 0.000005 | 0           | 0           |
| closing tables    | 0.000007 | 0.000002 | 0.000004 | 0           | 0           |
| freeing items     | 0.000090 | 0.000000 | 0.000091 | 0           | 0           |
| cleaning up       | 0.000013 | 0.000000 | 0.000012 | 0           | 0           |
+-----+-----+-----+-----+-----+-----+
17 rows in set, 1 warning (0.00 sec)

```

show profile的常用查询参数:

- ① ALL: 显示所有的开销信息。 ② BLOCK IO: 显示块IO开销。 ③ CONTEXT SWITCHES: 上下文切换开销。 ④ CPU: 显示CPU开销信息。 ⑤ IPC: 显示发送和接收开销信息。 ⑥ MEMORY: 显示内存开销信息。 ⑦ PAGE FAULTS: 显示页面错误开销信息。 ⑧ SOURCE: 显示和Source_function, Source_file, Source_line相关的开销信息。 ⑨ SWAPS: 显示交换次数开销信息。

6. 分析查询语句: EXPLAIN

6.1 概述

官网介绍

<https://dev.mysql.com/doc/refman/5.7/en/explain-output.html>

<https://dev.mysql.com/doc/refman/8.0/en/explain-output.html>

The screenshot shows the MySQL 5.7 Reference Manual page for the Extended EXPLAIN Output Format. The left sidebar has a tree view of topics, with 'Extended EXPLAIN Output Format' highlighted. The main content area discusses the extended information produced by EXPLAIN statements, noting that it's available for SELECT statements but not for other statements like DELETE, INSERT, REPLACE, and UPDATE. It also mentions that SHOW WARNINGS displays an empty result for other explainable statements. A note section explains that the syntax is still recognized for backward compatibility but is deprecated. Below this is an example of extended EXPLAIN output from the MySQL command line.

```
mysql> EXPLAIN
      SELECT t1.a, t1.b IN (SELECT t2.a FROM t2) FROM t1;
+----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len |
+----+-----+-----+-----+-----+-----+-----+-----+
| 1  | SIMPLE      | t1    |          | range | PRIMARY        | NULL | NULL    |
+----+-----+-----+-----+-----+-----+-----+-----+
```

版本情况

- MySQL 5.6.3以前只能 EXPLAIN SELECT；MySQL 5.6.3以后就可以 EXPLAIN SELECT, UPDATE, DELETE
- 在5.7以前的版本中，想要显示 partitions 需要使用 explain partitions 命令；想要显示 filtered 需要使用 explain extended 命令。在5.7版本后，默认explain直接显示partitions和filtered中的信息。

A screenshot of the MySQL command line showing the output of EXPLAIN. The columns 'partitions' and 'filtered' are highlighted with red boxes. The output shows a query with a partitioned table and its execution plan.

```
+----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len |
+----+-----+-----+-----+-----+-----+-----+-----+
| 1  | SIMPLE      | t1    |          | range | PRIMARY        | NULL | NULL    |
+----+-----+-----+-----+-----+-----+-----+-----+
```

6.2 基本语法

EXPLAIN 或 DESCRIBE语句的语法形式如下：

```
EXPLAIN SELECT select_options
或者
DESCRIBE SELECT select_options
```

如果我们想看看某个查询的执行计划的话，可以在具体的查询语句前边加一个 EXPLAIN，就像这样：

```
mysql> EXPLAIN SELECT 1;
```

EXPLAIN 语句输出的各个列的作用如下：

列名	描述
<code>id</code>	在一个大的查询语句中每个SELECT关键字都对应一个 唯一的id
<code>select_type</code>	SELECT关键字对应的那个查询的类型
<code>table</code>	表名
<code>partitions</code>	匹配的分区信息
<code>type</code>	针对单表的访问方法
<code>possible_keys</code>	可能用到的索引
<code>key</code>	实际上使用的索引
<code>key_len</code>	实际使用到的索引长度
<code>ref</code>	当使用索引列等值查询时，与索引列进行等值匹配的对象信息
<code>rows</code>	预估的需要读取的记录条数
<code>filtered</code>	某个表经过搜索条件过滤后剩余记录条数的百分比
<code>Extra</code>	一些额外的信息

6.3 数据准备

1. 建表

```
CREATE TABLE s1 (
    id INT AUTO_INCREMENT,
    key1 VARCHAR(100),
    key2 INT,
    key3 VARCHAR(100),
    key_part1 VARCHAR(100),
    key_part2 VARCHAR(100),
    key_part3 VARCHAR(100),
    common_field VARCHAR(100),
    PRIMARY KEY (id),
    INDEX idx_key1 (key1),
    UNIQUE INDEX idx_key2 (key2),
    INDEX idx_key3 (key3),
    INDEX idx_key_part(key_part1, key_part2, key_part3)
) ENGINE=INNODB CHARSET=utf8;
```

```
CREATE TABLE s2 (
    id INT AUTO_INCREMENT,
    key1 VARCHAR(100),
    key2 INT,
    key3 VARCHAR(100),
    key_part1 VARCHAR(100),
    key_part2 VARCHAR(100),
    key_part3 VARCHAR(100),
    common_field VARCHAR(100),
    PRIMARY KEY (id),
    INDEX idx_key1 (key1),
    UNIQUE INDEX idx_key2 (key2),
```

```
INDEX idx_key3 (key3),
INDEX idx_key_part(key_part1, key_part2, key_part3)
) ENGINE=INNODB CHARSET=utf8;
```

2. 设置参数 log_bin_trust_function_creators

创建函数，假如报错，需开启如下命令：允许创建函数设置：

```
set global log_bin_trust_function_creators=1;      # 不加global只是当前窗口有效。
```

3. 创建函数

```
DELIMITER //
CREATE FUNCTION rand_string1(n INT)
    RETURNS VARCHAR(255) #该函数会返回一个字符串
BEGIN
    DECLARE chars_str VARCHAR(100) DEFAULT
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ';
    DECLARE return_str VARCHAR(255) DEFAULT '';
    DECLARE i INT DEFAULT 0;
    WHILE i < n DO
        SET return_str =CONCAT(return_str,SUBSTRING(chars_str,FLLOOR(1+RAND()*52),1));
        SET i = i + 1;
    END WHILE;
    RETURN return_str;
END //
DELIMITER ;
```

4. 创建存储过程

创建往s1表中插入数据的存储过程：

```
DELIMITER //
CREATE PROCEDURE insert_s1 (IN min_num INT (10),IN max_num INT (10))
BEGIN
    DECLARE i INT DEFAULT 0;
    SET autocommit = 0;
    REPEAT
        SET i = i + 1;
        INSERT INTO s1 VALUES(
        (min_num + i),
        rand_string1(6),
        (min_num + 30 * i + 5),
        rand_string1(6),
        rand_string1(10),
        rand_string1(5),
        rand_string1(10),
        rand_string1(10));
    UNTIL i = max_num
    END REPEAT;
    COMMIT;
END //
DELIMITER ;
```

创建往s2表中插入数据的存储过程：

```
DELIMITER //
CREATE PROCEDURE insert_s2 (IN min_num INT (10),IN max_num INT (10))
BEGIN
```

```
DECLARE i INT DEFAULT 0;
SET autocommit = 0;
REPEAT
    SET i = i + 1;
    INSERT INTO s2 VALUES(
        (min_num + i),
        rand_string1(6),
        (min_num + 30 * i + 5),
        rand_string1(6),
        rand_string1(10),
        rand_string1(5),
        rand_string1(10),
        rand_string1(10));
    UNTIL i = max_num
END REPEAT;
COMMIT;
END //
DELIMITER ;
```

5. 调用存储过程

s1表数据的添加：加入1万条记录：

```
CALL insert_s1(10001, 10000);
```

s2表数据的添加：加入1万条记录：

```
CALL insert_s2(10001, 10000);
```

6.4 EXPLAIN各列作用

为了让大家有比较好的体验，我们调整了下 **EXPLAIN** 输出列的顺序。

1. table

不论我们的查询语句有多复杂，里边儿 包含了多少个表，到最后也是需要对每个表进行 单表访问 的，所以MySQL规定**EXPLAIN语句输出的每条记录都对应着某个单表的访问方法**，该条记录的table列代表着该表的表名（有时不是真实的表名字，可能是简称）。

2. id

我们写的查询语句一般都以 **SELECT** 关键字开头，比较简单的查询语句里只有一个 **SELECT** 关键字，比如下边这个查询语句：

```
SELECT * FROM s1 WHERE key1 = 'a';
```

稍微复杂一点的连接查询中也只有一个 **SELECT** 关键字，比如：

```
SELECT * FROM s1 INNER JOIN s2
ON s1.key1 = s2.key1
WHERE s1.common_field = 'a';
```

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a';
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | s1    | NULL       | ref   | idx_key1        | idx_key1  | 303    | const  | 8     | 100.00  | NULL   |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.03 sec)
```

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2;
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | s1    | NULL       | ALL   | NULL          | NULL     | NULL    | NULL  | 9688 | 100.00  | NULL   |
| 1 | SIMPLE     | s2    | NULL       | ALL   | NULL          | NULL     | NULL    | NULL  | 9954 | 100.00  | Using join buffer (Block Nested Loop) |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set, 1 warning (0.01 sec)
```

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key1 FROM s2) OR key3 = 'a';
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | PRIMARY    | s1    | NULL       | ALL   | idx_key3       | NULL     | NULL    | NULL  | 9688 | 100.00  | Using where |
| 2 | SUBQUERY   | s2    | NULL       | index | idx_key1       | idx_key1 | 303    | NULL  | 9954 | 100.00  | Using index |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set, 1 warning (0.02 sec)
```

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key2 FROM s2 WHERE common_field = 'a');
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | s2    | NULL       | ALL   | idx_key3       | NULL     | NULL    | NULL  | 9954 | 10.00   | Using where; Start temporary |
| 1 | SIMPLE     | s1    | NULL       | ref   | idx_key1       | idx_key1 | 303    | xiaohaizi.s2.key3 | 1 | 100.00 | End temporary |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set, 1 warning (0.00 sec)
```

```
mysql> EXPLAIN SELECT * FROM s1 UNION SELECT * FROM s2;
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | PRIMARY    | s1    | NULL       | ALL   | NULL          | NULL     | NULL    | NULL  | 9688 | 100.00  | NULL   |
| 2 | UNION       | s2    | NULL       | ALL   | NULL          | NULL     | NULL    | NULL  | 9954 | 100.00  | NULL   |
| NULL | UNION RESULT | <union1,2> | NULL       | ALL   | NULL          | NULL     | NULL    | NULL  | NULL  | NULL    | Using temporary |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
3 rows in set, 1 warning (0.00 sec)
```

```
mysql> EXPLAIN SELECT * FROM s1 UNION ALL SELECT * FROM s2;
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | PRIMARY    | s1    | NULL       | ALL   | NULL          | NULL     | NULL    | NULL  | 9688 | 100.00  | NULL   |
| 2 | UNION       | s2    | NULL       | ALL   | NULL          | NULL     | NULL    | NULL  | 9954 | 100.00  | NULL   |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set, 1 warning (0.01 sec)
```

小结:

- id如果相同，可以认为是一组，从上往下顺序执行
- 在所有组中，id值越大，优先级越高，越先执行
- 关注点：id号每个号码，表示一趟独立的查询，一个sql的查询趟数越少越好

3. select_type

名称	描述
SIMPLE	Simple SELECT (not using UNION or subqueries)
PRIMARY	Outermost SELECT
UNION	Second or later SELECT statement in a UNION
UNION RESULT	Result of a UNION
SUBQUERY	First SELECT in subquery
DEPENDENT SUBQUERY	First SELECT in subquery, dependent on outer query
DEPENDENT UNION	Second or later SELECT statement in a UNION, dependent on outer query
DERIVED	Derived table
MATERIALIZED	Materialized subquery
UNCACHEABLE SUBQUERY	A subquery for which the result cannot be cached and must be re-evaluated for each row of the outer query
UNCACHEABLE UNION	The second or later select in a UNION that belongs to an uncacheable subquery (see UNCACHEABLE SUBQUERY)

具体分析如下：

```
mysql> EXPLAIN SELECT * FROM s1;
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | s1    | NULL      | ALL  | NULL        | NULL | NULL   | NULL |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

当然，连接查询也算是 SIMPLE 类型，比如：

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2;
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | s1    | NULL      | ALL  | NULL        | NULL | NULL   | NULL |
| 1 | SIMPLE     | s2    | NULL      | ALL  | NULL        | NULL | NULL   | NULL |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set, 1 warning (0.01 sec)
```

- PRIMARY

```
mysql> EXPLAIN SELECT * FROM s1 UNION SELECT * FROM s2;
```

- UNION
 - UNION RESULT
 - SUBQUERY

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key1 FROM s2) OR key3 = 'a';
```

- ### • DEPENDENT SUBQUERY

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key1 FROM s2 WHERE s1.key2 = s2.key2) OR key3 = 'a';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	s1	NULL	ALL	idx_key3	NULL	NULL	NULL	9688	100.00	Using where
2	DEPENDENT SUBQUERY	s2	NULL	ref	idx_key2, idx_key1	idx_key2	5	xiaohaizi.s1.key2	1	10.00	Using where

- DEPENDENT UNION

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key1 FROM s2 WHERE key1 = 'a' UNION SELECT key1 FROM s1 WHERE key1 = 'b');
```

- DERIVED

```
mysql> EXPLAIN SELECT * FROM (SELECT key1, count(*) as c FROM s1 GROUP BY key1) AS derived_s1 where c > 1;
```

- MATERIALIZED

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key1 FROM s2);
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	s1	NULL	ALL	idx_key1	NULL	NULL	NULL	9688	100.00	Using where
1	SIMPLE	<subquery2>	NULL	eq_ref	<auto_key>	<auto_key>	303	xiaohaizi.s1.key1	1	100.00	NULL
2	MATERIALIZED	s2	NULL	index	idx_key1	idx_key1	303	NULL	9954	100.00	Using index

3 rows in set, 1 warning (0.01 sec)

- UNCACHEABLE SUBQUERY

不常用，就不多说了。

- UNCACHEABLE UNION

不常用，就不多说了。

4. partitions (可略)

- 如果想详细了解，可以如下方式测试。创建分区表：

```
-- 创建分区表,
-- 按照id分区, id<100 p0分区, 其他p1分区
CREATE TABLE user_partitions (id INT auto_increment,
    NAME VARCHAR(12), PRIMARY KEY(id))
PARTITION BY RANGE(id)(
    PARTITION p0 VALUES less than(100),
    PARTITION p1 VALUES less than MAXVALUE
);
```

```
mysql> create table user_partitions (id int auto_increment, name varchar(12), primary key(id))
    -> partition by range(id)(partition p0 values less than(100),
    -> partition p1 values less than maxvalue);
Query OK, 0 rows affected (0.11 sec)
```

```
DESC SELECT * FROM user_partitions WHERE id>200;
```

查询id大于200 (200>100, p1分区) 的记录，查看执行计划，partitions是p1，符合我们的分区规则

```
mysql> desc select * from user_partitions where id>200;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | user_partitions | p1 | range | PRIMARY | PRIMARY | 4 | NULL | 1 | 100.00 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.01 sec)
```

5. type ☆

完整的访问方法如下： system , const , eq_ref , ref , fulltext , ref_or_null , index_merge , unique_subquery , index_subquery , range , index , ALL 。

我们详细解释一下：

- system

```
mysql> CREATE TABLE t(i int) Engine=MyISAM;
Query OK, 0 rows affected (0.05 sec)

mysql> INSERT INTO t VALUES(1);
Query OK, 1 row affected (0.01 sec)
```

然后我们看一下查询这个表的执行计划：

```
mysql> EXPLAIN SELECT * FROM t;
```

id select_type table partitions type possible_keys key key_len ref rows filtered Extra
1 SIMPLE t NULL system NULL NULL NULL NULL 1 100.00 NULL

1 row in set, 1 warning (0.00 sec)

- **const**

```
mysql> EXPLAIN SELECT * FROM s1 WHERE id = 10005;
```

id select_type table partitions type possible_keys key key_len ref rows filtered Extra
1 SIMPLE s1 NULL const PRIMARY PRIMARY 4 const 1 100.00 NULL

1 row in set, 1 warning (0.01 sec)

- **eq_ref**

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s1.id = s2.id;
```

id select_type table partitions type possible_keys key key_len ref rows filtered Extra
1 SIMPLE s2 NULL ALL PRIMARY NULL NULL NULL 9895 100.00 NULL
1 SIMPLE s1 NULL eq_ref PRIMARY PRIMARY 4 temp.s2.id 1 100.00 NULL

2 rows in set, 1 warning (0.01 sec)

从执行计划的结果中可以看出，MySQL打算将s2作为驱动表，s1作为被驱动表，重点关注s1的访问方法是 **eq_ref**，表明在访问s1表的时候可以 **通过主键的等值匹配** 来进行访问。

- **ref**

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a';
```

id select_type table partitions type possible_keys key key_len ref rows filtered Extra
1 SIMPLE s1 NULL ref idx_key1 idx_key1 303 const 8 100.00 NULL

1 row in set, 1 warning (0.04 sec)

- **fulltext**

全文索引

- **ref_or_null**

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a' OR key1 IS NULL;
```

id select_type table partitions type possible_keys key key_len ref rows filtered Extra
1 SIMPLE s1 NULL ref_or_null idx_key1 idx_key1 303 const 9 100.00 Using index condition

1 row in set, 1 warning (0.01 sec)

- **index_merge**

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a' OR key3 = 'a';
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type      | possible_keys | key           | key_len | ref   | rows | filtered | Extra
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | s1    | NULL       | index_merge | idx_key1, idx_key3 | idx_key1, idx_key3 | 303,303 | NULL  | 14   | 100.00 | Using union(idx_key1, idx_key3); Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.01 sec)
```

从执行计划的 `type` 列的值是 `index_merge` 就可以看出，MySQL 打算使用索引合并的方式来执行对 `s1` 表的查询。

- `unique_subquery`

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key2 IN (SELECT id FROM s2 where s1.key1 = s2.key1) OR key3 = 'a';
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type      | possible_keys | key           | key_len | ref   | rows | filtered | Extra
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | PRIMARY    | s1    | NULL       | ALL        | idx_key3      | NULL          | NULL    | NULL  | NULL  | 9688  | 100.00 | Using where |
| 2 | DEPENDENT SUBQUERY | s2    | NULL       | unique_subquery | PRIMARY, idx_key1 | PRIMARY | 4      | func  | 1     | 10.00 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set, 2 warnings (0.00 sec)
```

- `index_subquery`

```
mysql> EXPLAIN SELECT * FROM s1 WHERE common_field IN (SELECT key3 FROM s2 where s1.key1 = s2.key1) OR key3 = 'a';
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type      | possible_keys | key           | key_len | ref   | rows | filtered | Extra
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | PRIMARY    | s1    | NULL       | ALL        | idx_key3      | NULL          | NULL    | NULL  | NULL  | 9688  | 100.00 | Using where |
| 2 | DEPENDENT SUBQUERY | s2    | NULL       | index_subquery | idx_key1, idx_key3 | idx_key3 | 303    | func  | 1     | 10.00 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set, 2 warnings (0.01 sec)
```

- `range`

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN ('a', 'b', 'c');
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type      | possible_keys | key           | key_len | ref   | rows | filtered | Extra
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | s1    | NULL       | range     | idx_key1      | idx_key1 | 303    | NULL  | 27   | 100.00 | Using index condition |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.01 sec)
```

或者：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 > 'a' AND key1 < 'b';
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type      | possible_keys | key           | key_len | ref   | rows | filtered | Extra
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | s1    | NULL       | range     | idx_key1      | idx_key1 | 303    | NULL  | 294  | 100.00 | Using index condition |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

- `index`

```
mysql> EXPLAIN SELECT key_part2 FROM s1 WHERE key_part3 = 'a';
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type      | possible_keys | key           | key_len | ref   | rows | filtered | Extra
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | s1    | NULL       | index     | NULL          | idx_key_part | 909    | NULL  | 9688 | 10.00 | Using where; Using index |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

- `ALL`

```
mysql> EXPLAIN SELECT * FROM s1;
```

小结:

结果值从最好到最坏依次是： system > const > eq_ref > ref > fulltext > ref_or_null > index_merge > unique_subquery > index_subquery > range > index > ALL 其中比较重要的几个提取出来（见上图中的蓝色）。SQL 性能优化的目标：至少要达到 range 级别，要求是 ref 级别，最好是 consts 级别。（阿里巴巴开发手册要求）

6. possible_keys和key

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 > 'z' AND key3 = 'a'
```

7. key_len ☆

```
mysql> EXPLAIN SELECT * FROM s1 WHERE id = 10005;
```

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key2 = 10126;
```

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a';
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|  1 | SIMPLE      | s1    | NULL     | ref   | idx_key1       | idx_key1 | 303    | const  | 8    | 100.00  | NULL  |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key_part1 = '1';
```

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key_part1 = 'a' AND key_part2 = 'b';
```

id select_type table partitions type possible_keys key key_len ref rows filtered Extra
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 SIMPLE s1 NULL ref idx_key_part idx_key_part 606 const,const 1 100.00 NULL

1 row in set, 1 warning (0.01 sec)

练习：

key_len的长度计算公式：

```
varchar(10)变长字段且允许NULL = 10 * ( character set:  
utf8=3, gbk=2, latin1=1)+1(NULL)+2(变长字段)
```

```
varchar(10)变长字段且不允许NULL = 10 * ( character set: utf8=3, gbk=2, latin1=1)+2(变长字段)
```

```
char(10)固定字段且允许NULL = 10 * ( character set: utf8=3, gbk=2, latin1=1)+1(NULL)
```

```
char(10)固定字段且不允许NULL = 10 * ( character set: utf8=3, gbk=2, latin1=1)
```

8.ref

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a';
```

id select_type table partitions type possible_keys key key_len ref rows filtered Extra
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 SIMPLE s1 NULL ref idx_key1 idx_key1 303 const 8 100.00 NULL

1 row in set, 1 warning (0.01 sec)

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s1.id = s2.id;
```

id select_type table partitions type possible_keys key key_len ref rows filtered Extra
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 SIMPLE s1 NULL ALL PRIMARY NULL NULL NULL 9688 100.00 NULL
1 SIMPLE s2 NULL eq_ref PRIMARY PRIMARY 4 atguigu.s1.id 1 100.00 NULL

2 rows in set, 1 warning (0.00 sec)

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s2.key1 = UPPER(s1.key1);
```

id select_type table partitions type possible_keys key key_len ref rows filtered Extra
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 SIMPLE s1 NULL ALL NULL NULL NULL NULL 9688 100.00 NULL

1 row in set, 1 warning (0.00 sec)

9. rows ☆

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 > 'z';
```

id select_type table partitions type possible_keys key key_len ref rows filtered Extra										
1 SIMPLE s1 NULL range idx_key1 idx_key1 303 NULL 266 100.00 Using index condition										
1 row in set, 1 warning (0.00 sec)										

10. filtered

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 > 'z' AND common_field = 'a';
```

id select_type table partitions type possible_keys key key_len ref rows filtered Extra										
1 SIMPLE s1 NULL range idx_key1 idx_key1 303 NULL 266 10.00 Using index condition; Using where										
1 row in set, 1 warning (0.00 sec)										

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s1.key1 = s2.key1 WHERE s1.common_field = 'a';
```

id select_type table partitions type possible_keys key key_len ref rows filtered Extra										
1 SIMPLE s1 NULL ALL idx_key1 idx_key1 303 NULL 9688 10.00 Using where										
1 SIMPLE s2 NULL ref idx_key1 idx_key1 303 xiaohaizi.s1.key1 1 100.00 NULL										
2 rows in set, 1 warning (0.00 sec)										

11. Extra ☆

```
mysql> EXPLAIN SELECT 1;
```

id select_type table partitions type possible_keys key key_len ref rows filtered Extra										
1 SIMPLE NULL NULL NULL NULL NULL NULL NULL NULL NULL No tables used										
1 row in set, 1 warning (0.00 sec)										

- Impossible WHERE

```
mysql> EXPLAIN SELECT * FROM s1 WHERE 1 != 1;
```

id select_type table partitions type possible_keys key key_len ref rows filtered Extra										
1 SIMPLE NULL NULL NULL NULL NULL NULL NULL NULL NULL Impossible WHERE										
1 row in set, 1 warning (0.01 sec)										

- Using where

```
mysql> EXPLAIN SELECT * FROM s1 WHERE common_field = 'a';
```

```
+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key   | key_len | ref   | rows | filtered | Extra      |
+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | s1    | NULL       | ALL   | NULL          | NULL  | NULL   | NULL  | 9688 | 10.00  | Using where |
+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.01 sec)
```

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a' AND common_field = 'a';
```

```
+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra      |
+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303   | const | 8    | 10.00  | Using where |
+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

- No matching min/max row

```
mysql> EXPLAIN SELECT MIN(key1) FROM s1 WHERE key1 = 'abcdefg';
```

```
+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key   | key_len | ref   | rows | filtered | Extra      |
+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | NULL  | NULL       | NULL | NULL          | NULL  | NULL   | NULL  | NULL | NULL | No matching min/max row |
+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

- Using index

```
mysql> EXPLAIN SELECT key1 FROM s1 WHERE key1 = 'a';
```

```
+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra      |
+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303   | const | 8    | 100.00 | Using index |
+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

- Using index condition

```
SELECT * FROM s1 WHERE key1 > 'z' AND key1 LIKE '%a';
```

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 > 'z' AND key1 LIKE '%b';
```

```
+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra      |
+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | s1    | NULL       | range | idx_key1      | idx_key1 | 303   | NULL  | 266  | 100.00 | Using index condition |
+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.01 sec)
```

- Using join buffer (Block Nested Loop)

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s1.common_field =
s2.common_field;
```

```
+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key   | key_len | ref   | rows | filtered | Extra      |
+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | s1    | NULL       | ALL  | NULL          | NULL  | NULL   | NULL  | 9688 | 100.00 | NULL        |
| 1 | SIMPLE     | s2    | NULL       | ALL  | NULL          | NULL  | NULL   | NULL  | 9954 | 10.00  | Using where; Using join buffer (Block Nested Loop) |
+-----+-----+-----+-----+-----+-----+-----+
2 rows in set, 1 warning (0.03 sec)
```

- Not exists

```
mysql> EXPLAIN SELECT * FROM s1 LEFT JOIN s2 ON s1.key1 = s2.key1 WHERE s2.id IS
NULL;
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref          | rows | filtered | Extra
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | s1    | NULL      | ALL  | NULL        | NULL     | NULL     | NULL          | 9688 | 100.00   | NULL
| 1 | SIMPLE     | s2    | NULL      | ref  | idx_key1    | idx_key1 | 303     | xiaohaizi.s1.key1 | 1    | 10.00    | Using where; Not exists
+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set, 1 warning (0.00 sec)
```

- Using intersect(...)、Using union(...) 和 Using sort_union(...)

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a' OR key3 = 'a';
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref          | rows | filtered | Extra
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | s1    | NULL      | index_merge | idx_key1,idx_key3 | idx_key1, idx_key3 | 303,303 | NULL          | 2    | 100.00   | Using union(idx_key1,idx_key3); Using where
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

- Zero limit

```
mysql> EXPLAIN SELECT * FROM s1 LIMIT 0;
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref          | rows | filtered | Extra
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | NULL  | NULL      | NULL | NULL        | NULL     | NULL     | NULL          | NULL | NULL     | Zero limit
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

- Using filesort

```
mysql> EXPLAIN SELECT * FROM s1 ORDER BY key1 LIMIT 10;
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref          | rows | filtered | Extra
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | s1    | NULL      | index | NULL        | idx_key1 | 303     | NULL          | 10   | 100.00   | NULL
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.03 sec)
```

```
mysql> EXPLAIN SELECT * FROM s1 ORDER BY common_field LIMIT 10;
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref          | rows | filtered | Extra
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | s1    | NULL      | ALL  | NULL        | NULL     | NULL     | NULL          | 9688 | 100.00   | Using filesort
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

- Using temporary

```
mysql> EXPLAIN SELECT DISTINCT common_field FROM s1;
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref          | rows | filtered | Extra
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | s1    | NULL      | ALL  | NULL        | NULL     | NULL     | NULL          | 9688 | 100.00   | Using temporary
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

再比如：

```
mysql> EXPLAIN SELECT common_field, COUNT(*) AS amount FROM s1 GROUP BY common_field;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	s1	NULL	ALL	NULL	NULL	NULL	NULL	9688	100.00	Using temporary

1 row in set, 1 warning (0.00 sec)

```
mysql> EXPLAIN SELECT key1, COUNT(*) AS amount FROM s1 GROUP BY key1;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	s1	NULL	index	idx_key1	idx_key1	303	NULL	9688	100.00	Using index

1 row in set, 1 warning (0.00 sec)

从 Extra 的 Using index 的提示里我们可以看出，上述查询只需要扫描 idx_key1 索引就可以搞定了，不再需要临时表了。

- 其它

其它特殊情况这里省略。

12. 小结

- EXPLAIN不考虑各种Cache
- EXPLAIN不能显示MySQL在执行查询时所作的优化工作
- EXPLAIN不会告诉你关于触发器、存储过程的信息或用户自定义函数对查询的影响情况
- 部分统计信息是估算的，并非精确值

7. EXPLAIN的进一步使用

7.1 EXPLAIN四种输出格式

这里谈谈EXPLAIN的输出格式。EXPLAIN可以输出四种格式：[传统格式](#)，[JSON格式](#)，[TREE格式](#)以及[可视化输出](#)。用户可以根据需要选择适用于自己的格式。

1. 传统格式

传统格式简单明了，输出是一个表格形式，概要说明查询计划。

```
mysql> EXPLAIN SELECT s1.key1, s2.key1 FROM s1 LEFT JOIN s2 ON s1.key1 = s2.key1 WHERE s2.common_field IS NOT NULL;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	s2	NULL	ALL	idx_key1	idx_key1	NULL	NULL	9954	90.00	Using where
1	SIMPLE	s1	NULL	ref	idx_key1	idx_key1	303	xiaohaizi.s2.key1	1	100.00	Using index

2 rows in set, 1 warning (0.00 sec)

2. JSON格式

- JSON格式：在EXPLAIN单词和真正的查询语句中间加上 `FORMAT=JSON`。

```
EXPLAIN FORMAT=JSON SELECT ...
```

我们使用 `#` 后边跟随注释的形式为大家解释了 `EXPLAIN FORMAT=JSON` 语句的输出内容，但是大家可能有疑问 `"cost_info"` 里边的成本看着怪怪的，它们是怎么计算出来的？先看 `s1` 表的 `"cost_info"` 部分：

```
"cost_info": {  
    "read_cost": "1840.84",  
    "eval_cost": "193.76",  
    "prefix_cost": "2034.60",  
    "data_read_per_join": "1M"  
}
```

- `read_cost` 是由下边这两部分组成的：
 - `IO` 成本
 - 检测 `rows × (1 - filter)` 条记录的 `CPU` 成本

小贴士： `rows` 和 `filter` 都是我们前边介绍执行计划的输出列，在 JSON 格式的执行计划中，`rows` 相当于 `rows_examined_per_scan`，`filtered` 名称不变。

- `eval_cost` 是这样计算的：
检测 `rows × filter` 条记录的成本。
- `prefix_cost` 就是单独查询 `s1` 表的成本，也就是：
`read_cost + eval_cost`
- `data_read_per_join` 表示在此次查询中需要读取的数据量。

对于 `s2` 表的 `"cost_info"` 部分是这样的：

```
"cost_info": {  
    "read_cost": "968.80",  
    "eval_cost": "193.76",  
    "prefix_cost": "3197.16",  
    "data_read_per_join": "1M"  
}
```

由于 `s2` 表是被驱动表，所以可能被读取多次，这里的 `read_cost` 和 `eval_cost` 是访问多次 `s2` 表后累加起来的值，大家主要关注里边儿的 `prefix_cost` 的值代表的是整个连接查询预计的成本，也就是单次查询 `s1` 表和多次查询 `s2` 表后的成本的和，也就是：

```
968.80 + 193.76 + 2034.60 = 3197.16
```

3. TREE格式

TREE格式是8.0.16版本之后引入的新格式，主要根据查询的 `各个部分之间的关系` 和 `各部分的执行顺序` 来描述如何查询。

```

mysql> EXPLAIN FORMAT=tree SELECT * FROM s1 INNER JOIN s2 ON s1.key1 = s2.key2 WHERE s1.common_field = 'a' \G
***** 1. row *****
EXPLAIN: -> Nested loop inner join (cost=1360.08 rows=990)
    -> Filter: ((s1.common_field = 'a') and (s1.key1 is not null)) (cost=1013.75 rows=990)
        -> Table scan on s1 (cost=1013.75 rows=9895)
            -> Single-row index lookup on s2 using idx_key2 (key2=s1.key1), with index condition: (cast(s1.key1 as double) = cast(s2.key2 as double)) (cost=0.25 rows=1)

1 row in set, 1 warning (0.00 sec)

```

4. 可视化输出

可视化输出，可以通过MySQL Workbench可视化查看MySQL的执行计划。通过点击Workbench的放大镜图标，即可生成可视化的查询计划。

上图按从左到右的连接顺序显示表。红色框表示 全表扫描，而绿色框表示使用 索引查找。对于每个表，显示使用的索引。还要注意的是，每个表格的框上方是每个表访问所发现的行数的估计值以及访问该表的成本。

7.2 SHOW WARNINGS的使用

```

mysql> EXPLAIN SELECT s1.key1, s2.key1 FROM s1 LEFT JOIN s2 ON s1.key1 = s2.key1 WHERE s2.common_field IS NOT NULL;

```

id select_type table partitions type possible_keys key key_len ref rows filtered Extra
1 SIMPLE s2 NULL ALL idx_key1 NULL NULL NULL 9954 90.00 Using where
1 SIMPLE s1 NULL ref idx_key1 idx_key1 303 xiaohaizi.s2.key1 1 100.00 Using index

2 rows in set, 1 warning (0.00 sec)

```

mysql> SHOW WARNINGS\G
***** 1. row *****
Level: Note
Code: 1003
Message: /* select#1 */ select `atguigu`.`s1`.`key1` AS `key1`, `atguigu`.`s2`.`key1` AS `key1` from `atguigu`.`s1` join `atguigu`.`s2` where ((`atguigu`.`s1`.`key1` = `atguigu`.`s2`.`key1`) and (`atguigu`.`s2`.`common_field` is not null))
1 row in set (0.00 sec)

```

8. 分析优化器执行计划：trace

```

SET optimizer_trace="enabled=on",end_markers_in_json=on;

set optimizer_trace_max_mem_size=1000000;

```

开启后，可分析如下语句：

- SELECT
- INSERT
- REPLACE

- UPDATE
- DELETE
- EXPLAIN
- SET
- DECLARE
- CASE
- IF
- RETURN
- CALL

测试：执行如下SQL语句

```
select * from student where id < 10;
```

最后，查询 information_schema.optimizer_trace 就可以知道MySQL是如何执行SQL的：

```
select * from information_schema.optimizer_trace\G
```

```
***** 1. row *****
//第1部分：查询语句
QUERY: select * from student where id < 10
//第2部分：QUERY字段对应语句的跟踪信息
TRACE: {
  "steps": [
    {
      "join_preparation": { //预备工作
        "select#": 1,
        "steps": [
          {
            "expanded_query": "/* select#1 */ select `student`.`id` AS `id`, `student`.`stuno` AS `stuno`, `student`.`name` AS `name`, `student`.`age` AS `age`, `student`.`classId` AS `classId` from `student` where (`student`.`id` < 10)"
          }
        ] /* steps */
      } /* join_preparation */
    },
    {
      "join_optimization": { //进行优化
        "select#": 1,
        "steps": [
          {
            "condition_processing": { //条件处理
              "condition": "WHERE",
              "original_condition": "(`student`.`id` < 10)",
              "steps": [
                {
                  "transformation": "equality_propagation",
                  "resulting_condition": "(`student`.`id` < 10)"
                },
                {
                  "transformation": "constant_propagation",
                  "resulting_condition": "(`student`.`id` < 10)"
                },
                {
                  "transformation": "trivial_condition_removal",
                  "resulting_condition": "(`student`.`id` < 10)"
                }
              ]
            }
          }
        ]
      }
    }
  ]
}
```

```
        ] /* steps */
    } /* condition_processing */
},
{
    "substitute_generated_columns": { //替换生成的列
    } /* substitute_generated_columns */
},
{
    "table_dependencies": [ //表的依赖关系
    {
        "table": "`student`",
        "row_may_be_null": false,
        "map_bit": 0,
        "depends_on_map_bits": [
            ] /* depends_on_map_bits */
        }
    ] /* table_dependencies */
},
{
    "ref_optimizer_key_uses": [ //使用键
    ] /* ref_optimizer_key_uses */
},
{
    "rows_estimation": [ //行判断
    {
        "table": "`student`",
        "range_analysis": {
            "table_scan": {
                "rows": 3973767,
                "cost": 408558
            } /* table_scan */, //扫描表
            "potential_range_indexes": [ //潜在的范围索引
            {
                "index": "PRIMARY",
                "usable": true,
                "key_parts": [
                    "id"
                ] /* key_parts */
            }
        ] /* potential_range_indexes */,
        "setup_range_conditions": [ //设置范围条件
        ] /* setup_range_conditions */,
        "group_index_range": {
            "chosen": false,
            "cause": "not_group_by_or_distinct"
        } /* group_index_range */,
        "skip_scan_range": {
            "potential_skip_scan_indexes": [
            {
                "index": "PRIMARY",
                "usable": false,
                "cause": "query_references_nonkey_column"
            }
        ] /* potential_skip_scan_indexes */
    } /* skip_scan_range */,
    "analyzing_range_alternatives": { //分析范围选项
        "range_scan_alternatives": [
        {

```

```
        "index": "PRIMARY",
        "ranges": [
            "id < 10"
        ] /* ranges */,
        "index_dives_for_eq_ranges": true,
        "rowid_ordered": true,
        "using_mrr": false,
        "index_only": false,
        "rows": 9,
        "cost": 1.91986,
        "chosen": true
    }
] /* range_scan_alternatives */,
"analyzing_roworder_intersect": {
    "usable": false,
    "cause": "too_few_roworder_scans"
} /* analyzing_roworder_intersect */
} /* analyzing_range_alternatives */,
"chosen_range_access_summary": {      //选择范围访问摘要
    "range_access_plan": {
        "type": "range_scan",
        "index": "PRIMARY",
        "rows": 9,
        "ranges": [
            "id < 10"
        ] /* ranges */
    } /* range_access_plan */,
    "rows_for_plan": 9,
    "cost_for_plan": 1.91986,
    "chosen": true
} /* chosen_range_access_summary */
} /* range_analysis */
}
] /* rows_estimation */
},
{
"considered_execution_plans": [      //考虑执行计划
{
    "plan_prefix": [
    ] /* plan_prefix */,
    "table": "`student`",
    "best_access_path": {      //最佳访问路径
        "considered_access_paths": [
{
            "rows_to_scan": 9,
            "access_type": "range",
            "range_details": {
                "used_index": "PRIMARY"
            } /* range_details */,
            "resulting_rows": 9,
            "cost": 2.81986,
            "chosen": true
        }
    ] /* considered_access_paths */
} /* best_access_path */,
"condition_filtering_pct": 100,      //行过滤百分比
"rows_for_plan": 9,
"cost_for_plan": 2.81986,
```

```

        "chosen": true
    }
]
/* considered_execution_plans */
},
{
    "attaching_conditions_to_tables": { //将条件附加到表上
        "original_condition": "(`student`.`id` < 10)",
        "attached_conditions_computation": [
            ] /* attached_conditions_computation */,
        "attached_conditions_summary": [ //附加条件概要
            {
                "table": "`student`",
                "attached": "(`student`.`id` < 10)"
            }
        ] /* attached_conditions_summary */
    } /* attaching_conditions_to_tables */
},
{
    "finalizing_table_conditions": [
        {
            "table": "`student`",
            "original_table_condition": "(`student`.`id` < 10)",
            "final_table_condition": "(`student`.`id` < 10)"
        }
    ] /* finalizing_table_conditions */
},
{
    "refine_plan": [ //精简计划
        {
            "table": "`student`"
        }
    ] /* refine_plan */
},
] /* steps */
} /* join_optimization */
},
{
    "join_execution": { //执行
        "select#": 1,
        "steps": [
            ] /* steps */
        } /* join_execution */
    }
] /* steps */
}
//第3部分：跟踪信息过长时，被截断的跟踪信息的字节数。
MISSING_BYTES_BEYOND_MAX_MEM_SIZE: 0 //丢失的超出最大容量的字节
//第4部分：执行跟踪语句的用户是否有查看对象的权限。当不具有权限时，该列信息为1且TRACE字段为空，一般在
//调用带有SQL SECURITY DEFINER的视图或者是存储过程的情况下，会出现此问题。
INSUFFICIENT_PRIVILEGES: 0 //缺失权限
1 row in set (0.00 sec)

```

9. MySQL监控分析视图-sys schema

9.1 Sys schema视图摘要

1. **主机相关:** 以host_summary开头, 主要汇总了IO延迟的信息。
2. **Innodb相关:** 以innodb开头, 汇总了innodb buffer信息和事务等待innodb锁的信息。
3. **I/o相关:** 以io开头, 汇总了等待I/O、I/O使用量情况。
4. **内存使用情况:** 以memory开头, 从主机、线程、事件等角度展示内存的使用情况
5. **连接与会话信息:** processlist和session相关视图, 总结了会话相关信息。
6. **表相关:** 以schema_table开头的视图, 展示了表的统计信息。
7. **索引信息:** 统计了索引的使用情况, 包含冗余索引和未使用的索引情况。
8. **语句相关:** 以statement开头, 包含执行全表扫描、使用临时表、排序等的语句信息。
9. **用户相关:** 以user开头的视图, 统计了用户使用的文件I/O、执行语句统计信息。
10. **等待事件相关信息:** 以wait开头, 展示等待事件的延迟情况。

9.2 Sys schema视图使用场景

索引情况

```
#1. 查询冗余索引
select * from sys.schema_redundant_indexes;

#2. 查询未使用过的索引
select * from sys.schema_unused_indexes;

#3. 查询索引的使用情况
select index_name, rows_selected, rows_inserted, rows_updated, rows_deleted
from sys.schema_index_statistics where table_schema='dbname' ;
```

表相关

```
# 1. 查询表的访问量
select table_schema, table_name, sum(io_read_requests+io_write_requests) as io from
sys.schema_table_statistics group by table_schema,table_name order by io desc;

# 2. 查询占用bufferpool较多的表
select object_schema,object_name,allocated,data
from sys.innodb_buffer_stats_by_table order by allocated limit 10;

# 3. 查看表的全表扫描情况
select * from sys.statements_with_full_table_scans where db='dbname' ;
```

语句相关

```
#1. 监控SQL执行的频率
select db,exec_count,query from sys.statement_analysis
order by exec_count desc;

#2. 监控使用了排序的SQL
select db,exec_count,first_seen,last_seen,query
from sys.statements_with_sorting limit 1;

#3. 监控使用了临时表或者磁盘临时表的SQL
select db,exec_count,tmp_tables,tmp_disk_tables,query
from sys.statement_analysis where tmp_tables>0 or tmp_disk_tables >0
order by (tmp_tables+tmp_disk_tables) desc;
```

IO相关

```
#1. 查看消耗磁盘IO的文件  
select file,avg_read,avg_write,avg_read+avg_write as avg_io  
from sys.io_global_by_file_by_bytes order by avg_read limit 10;
```

Innodb 相关

```
#1. 行锁阻塞情况  
select * from sys.innodb_lock_waits;
```

第10章_索引优化与查询优化

讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

1. 数据准备

学员表 插 50万 条， 班级表 插 1万 条。

步骤1：建表

```
CREATE TABLE `class` (
  `id` INT(11) NOT NULL AUTO_INCREMENT,
  `className` VARCHAR(30) DEFAULT NULL,
  `address` VARCHAR(40) DEFAULT NULL,
  `monitor` INT NULL ,
  PRIMARY KEY (`id`)
) ENGINE=INNODB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;

CREATE TABLE `student` (
  `id` INT(11) NOT NULL AUTO_INCREMENT,
  `stuno` INT NOT NULL ,
  `name` VARCHAR(20) DEFAULT NULL,
  `age` INT(3) DEFAULT NULL,
  `classId` INT(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
  #CONSTRAINT `fk_class_id` FOREIGN KEY (`classId`) REFERENCES `t_class` (`id`)
) ENGINE=INNODB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

步骤2：设置参数

- 命令开启：允许创建函数设置：

```
set global log_bin_trust_function_creators=1;      # 不加global只是当前窗口有效。
```

步骤3：创建函数

保证每条数据都不同。

```
#随机产生字符串
DELIMITER //
CREATE FUNCTION rand_string(n INT) RETURNS VARCHAR(255)
BEGIN
DECLARE chars_str VARCHAR(100) DEFAULT
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ';
DECLARE return_str VARCHAR(255) DEFAULT '';
DECLARE i INT DEFAULT 0;
WHILE i < n DO
SET return_str =CONCAT(return_str,SUBSTRING(chars_str,FL00R(1+RAND()*52),1));
SET i = i + 1;
END WHILE;
```

```

RETURN return_str;
END //
DELIMITER ;

#假如要删除
#drop function rand_string;

```

随机产生班级编号

```

#用于随机产生多少到多少的编号
DELIMITER //
CREATE FUNCTION rand_num (from_num INT ,to_num INT) RETURNS INT(11)
BEGIN
DECLARE i INT DEFAULT 0;
SET i = FLOOR(from_num +RAND()*(to_num - from_num+1)) ;
RETURN i;
END //
DELIMITER ;

#假如要删除
#drop function rand_num;

```

步骤4：创建存储过程

```

#创建往stu表中插入数据的存储过程
DELIMITER //
CREATE PROCEDURE insert_stu( START INT , max_num INT )
BEGIN
DECLARE i INT DEFAULT 0;
SET autocommit = 0;      #设置手动提交事务
REPEAT #循环
SET i = i + 1; #赋值
INSERT INTO student (stuno, name ,age ,classId ) VALUES
((START+i),rand_string(6),rand_num(1,50),rand_num(1,1000));
UNTIL i = max_num
END REPEAT;
COMMIT; #提交事务
END //
DELIMITER ;

#假如要删除
#drop PROCEDURE insert_stu;

```

创建往class表中插入数据的存储过程

```

#执行存储过程，往class表添加随机数据
DELIMITER //
CREATE PROCEDURE `insert_class`(`max_num` INT )
BEGIN
DECLARE i INT DEFAULT 0;
SET autocommit = 0;
REPEAT
SET i = i + 1;
INSERT INTO class ( classname,address,monitor ) VALUES
(rand_string(8),rand_string(10),rand_num(1,100000));
UNTIL i = max_num
END REPEAT;
COMMIT;

```

```
END //  
DELIMITER ;  
  
#假如要删除  
#drop PROCEDURE insert_class;
```

步骤5：调用存储过程

class

```
#执行存储过程，往class表添加1万条数据  
CALL insert_class(10000);
```

stu

```
#执行存储过程，往stu表添加50万条数据  
CALL insert_stu(10000, 500000);
```

步骤6：删除某表上的索引

创建存储过程

```
DELIMITER //  
CREATE PROCEDURE `proc_drop_index`(dbname VARCHAR(200), tablename VARCHAR(200))  
BEGIN  
    DECLARE done INT DEFAULT 0;  
    DECLARE ct INT DEFAULT 0;  
    DECLARE _index VARCHAR(200) DEFAULT '';  
    DECLARE _cur CURSOR FOR SELECT index_name FROM  
information_schema.STATISTICS WHERE table_schema=dbname AND table_name=tablename AND  
seq_in_index=1 AND index_name <>'PRIMARY' ;  
#每个游标必须使用不同的declare continue handler for not found set done=1来控制游标的结束  
    DECLARE CONTINUE HANDLER FOR NOT FOUND set done=2 ;  
#若没有数据返回，程序继续，并将变量done设为2  
    OPEN _cur;  
    FETCH _cur INTO _index;  
    WHILE _index<>'' DO  
        SET @str = CONCAT("drop index " , _index , " on " , tablename );  
        PREPARE sql_str FROM @str ;  
        EXECUTE sql_str;  
        DEALLOCATE PREPARE sql_str;  
        SET _index='';  
        FETCH _cur INTO _index;  
    END WHILE;  
    CLOSE _cur;  
END //  
DELIMITER ;
```

执行存储过程

```
CALL proc_drop_index("dbname", "tablename");
```

2. 索引失效案例

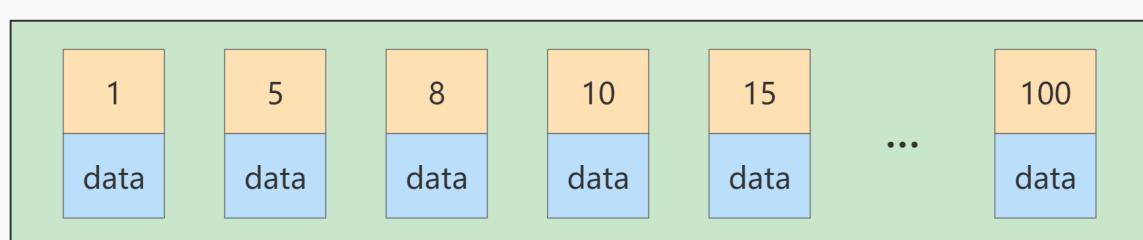
2.1 全值匹配我最爱

2.2 最佳左前缀法则

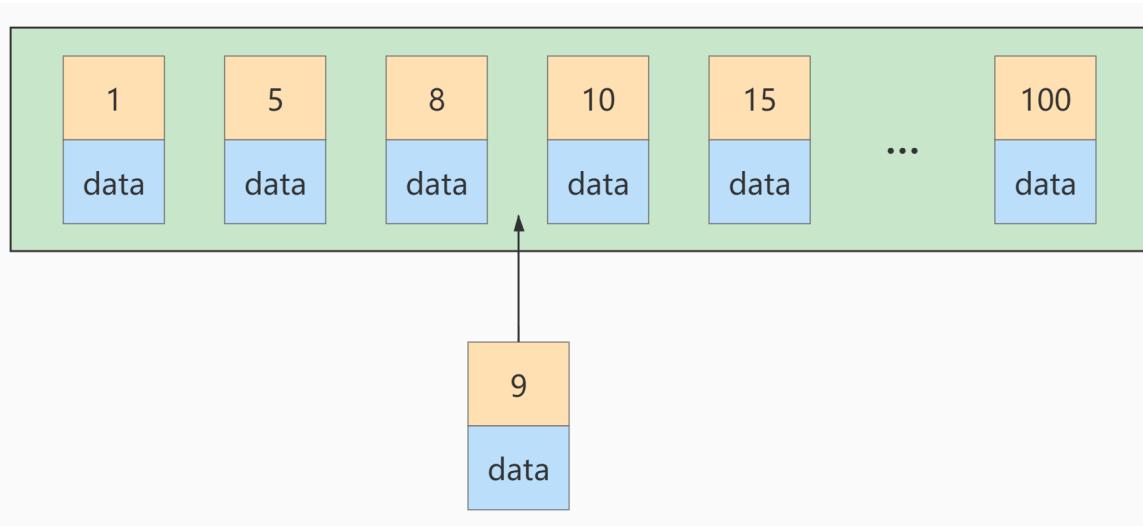
拓展：Alibaba 《Java开发手册》

索引文件具有 B-Tree 的最左前缀匹配特性，如果左边的值未确定，那么无法使用此索引。

2.3 主键插入顺序



如果此时再插入一条主键值为 9 的记录，那它插入的位置就如下图：



可这个数据页已经满了，再插进来咋办呢？我们需要把当前 [页面分裂](#) 成两个页面，把本页中的一些记录移动到新创建的这个页中。页面分裂和记录移位意味着什么？意味着：[性能损耗](#)！所以如果我们想尽量避免这样无谓的性能损耗，最好让插入的记录的 [主键值依次递增](#)，这样就不会发生这样的性能损耗了。所以我们建议：让主键具有 [AUTO_INCREMENT](#)，让存储引擎自己为表生成主键，而不是我们手动插入，比如：`person_info` 表：

```
CREATE TABLE person_info(
    id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    birthday DATE NOT NULL,
    phone_number CHAR(11) NOT NULL,
    country varchar(100) NOT NULL,
    PRIMARY KEY (id),
    KEY idx_name_birthday_phone_number (name(10), birthday, phone_number)
);
```

我们自定义的主键列 `id` 拥有 [AUTO_INCREMENT](#) 属性，在插入记录时存储引擎会自动为我们填入自增的主键值。这样的主键占用空间小，顺序写入，减少页分裂。

2.4 计算、函数、类型转换(自动或手动)导致索引失效

```
EXPLAIN SELECT SQL_NO_CACHE * FROM student WHERE student.name LIKE 'abc%';
```

```
EXPLAIN SELECT SQL_NO_CACHE * FROM student WHERE LEFT(student.name,3) = 'abc';
```

创建索引

```
CREATE INDEX idx_name ON student(NAME);
```

第一种：索引优化生效

```
mysql> EXPLAIN SELECT SQL_NO_CACHE * FROM student WHERE student.name LIKE 'abc%';
```

```
mysql> SELECT SQL_NO_CACHE * FROM student WHERE student.name LIKE 'abc%';
+-----+-----+-----+-----+
| id   | stuno | name  | age   | classId |
+-----+-----+-----+-----+
| 5301379 | 1233401 | AbCHEa | 164   |      259 |
| 7170042 | 3102064 | ABcHeB | 199   |      161 |
| 1901614 | 1833636 | ABcHeC | 226   |      275 |
| 5195021 | 1127043 | abchEC | 486   |      72  |
| 4047089 | 3810031 | AbCHFd | 268   |      210 |
| 4917074 | 849096  | ABcHfD | 264   |      442 |
| 1540859 | 141979  | abchFF | 119   |      140 |
| 5121801 | 1053823 | AbCHFg | 412   |      327 |
| 2441254 | 2373276 | abchFJ | 170   |      362 |
| 7039146 | 2971168 | ABcHgI | 502   |      465 |
| 1636826 | 1580286 | ABcHgK | 71    |      262 |
| 374344  | 474345  | abchHL | 367   |      212 |
| 1596534 | 169191  | AbCHH1 | 102   |      146 |
|
| 5266837 | 1198859 | abc1Xe | 292   |      298 |
| 8126968 | 4058990 | aBC1xE | 316   |      150 |
| 4298305 | 399962  | AbCLXF | 72    |      423 |
| 5813628 | 1745650 | aBC1xF | 356   |      323 |
| 6980448 | 2912470 | AbCLXF | 107   |      78  |
| 7881979 | 3814001 | AbCLXF | 89    |      497 |
| 4955576 | 887598  | ABcLxg | 121   |      385 |
| 3653460 | 3585482 | AbCLXJ | 130   |      174 |
| 1231990 | 1283439 | AbCLYH | 189   |      429 |
| 6110615 | 2042637 | ABcLyh | 157   |      40  |
+-----+-----+-----+-----+
401 rows in set, 1 warning (0.01 sec)
```

第二种：索引优化失效

```
mysql> EXPLAIN SELECT SQL_NO_CACHE * FROM student WHERE LEFT(student.name,3) = 'abc';
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table  | partitions | type | possible_keys | key  | key_len | ref   | rows  | filtered | Extra  |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | student | NULL       | ALL  | NULL          | NULL | NULL    | NULL  | 7737618 | 100.00  | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 2 warnings (0.00 sec)
```

```
mysql> SELECT SQL_NO_CACHE * FROM student WHERE LEFT(student.name,3) = 'abc';
```

```
+-----+-----+-----+-----+
| id   | stuno | name  | age   | classId |
+-----+-----+-----+-----+
```

```

+-----+-----+-----+-----+
| 5301379 | 1233401 | AbCHEa | 164 | 259 |
| 7170042 | 3102064 | ABcHeB | 199 | 161 |
| 1901614 | 1833636 | ABcHeC | 226 | 275 |
| 5195021 | 1127043 | abchEC | 486 | 72 |
| 4047089 | 3810031 | AbCHFd | 268 | 210 |
| 4917074 | 849096 | ABchFD | 264 | 442 |
| 1540859 | 141979 | abchFF | 119 | 140 |
| 5121801 | 1053823 | AbCHFg | 412 | 327 |
| 2441254 | 2373276 | abchFJ | 170 | 362 |
| 7039146 | 2971168 | ABcHgI | 502 | 465 |
| 1636826 | 1580286 | ABcHgK | 71 | 262 |
| 374344 | 474345 | abchHL | 367 | 212 |
| 1596534 | 169191 | AbCHH1 | 102 | 146 |
|
| ... |
| 5266837 | 1198859 | abc1Xe | 292 | 298 |
| 8126968 | 4058990 | aBC1xE | 316 | 150 |
| 4298305 | 399962 | AbCLXF | 72 | 423 |
| 5813628 | 1745650 | aBC1xF | 356 | 323 |
| 6980448 | 2912470 | AbCLXF | 107 | 78 |
| 7881979 | 3814001 | AbCLXF | 89 | 497 |
| 4955576 | 887598 | ABcLxg | 121 | 385 |
| 3653460 | 3585482 | AbCLXJ | 130 | 174 |
| 1231990 | 1283439 | AbCLYH | 189 | 429 |
| 6110615 | 2042637 | ABcLyh | 157 | 40 |
+-----+-----+-----+-----+

```

401 rows in set, 1 warning (3.62 sec)

type为“ALL”，表示没有使用到索引，查询时间为 3.62 秒，查询效率较之前低很多。

再举例：

- student表的字段stuno上设置有索引

```
CREATE INDEX idx_sno ON student(stuno);
```

```
EXPLAIN SELECT SQL_NO_CACHE id, stuno, NAME FROM student WHERE stuno+1 = 900001;
```

运行结果：

```

+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | student | NULL | ALL | NULL | NULL | NULL | NULL | 7737618 | 100.00 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)

```

- 索引优化生效：

```
EXPLAIN SELECT SQL_NO_CACHE id, stuno, NAME FROM student WHERE stuno = 900000;
```

再举例：

- student表的字段name上设置有索引

```
CREATE INDEX idx_name ON student(NAME);
```

```
EXPLAIN SELECT id, stuno, name FROM student WHERE SUBSTRING(name, 1,3)='abc';
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | student | NULL | ALL | NULL | NULL | NULL | NULL | 7737618 | 100.00 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

```
EXPLAIN SELECT id, stuno, NAME FROM student WHERE NAME LIKE 'abc%' ;
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | student | NULL | range | idx_name | idx_name | 63 | NULL | 401 | 100.00 | Using index condition |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

2.5 类型转换导致索引失效

下列哪个sql语句可以用到索引。 (假设name字段上设置有索引)

```
# 未使用到索引
```

```
EXPLAIN SELECT SQL_NO_CACHE * FROM student WHERE name=123 ;
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | student | NULL | ALL | idx_name | NULL | NULL | NULL | 7737618 | 10.00 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 6 warnings (0.00 sec)
```

```
# 使用到索引
```

```
EXPLAIN SELECT SQL_NO_CACHE * FROM student WHERE name='123' ;
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | student | NULL | ref | idx_name | idx_name | 63 | const | 1 | 100.00 | NULL |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 2 warnings (0.00 sec)
```

- name=123发生类型转换，索引失效。

2.6 范围条件右边的列索引失效

```
ALTER TABLE student DROP INDEX idx_name;
ALTER TABLE student DROP INDEX idx_age;
ALTER TABLE student DROP INDEX idx_age_classid;
```

```
EXPLAIN SELECT SQL_NO_CACHE * FROM student
WHERE student.age=30 AND student.classId>20 AND student.name = 'abc' ;
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | student | NULL | range | idx_age_classid_name | idx_age_classid_name | 10 | NULL | 31338 | 10.00 | Using index condition; Using MRR |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 2 warnings (0.00 sec)
```

```
create index idx_age_name_classid on student(age,name,classid);
```

- 将范围查询条件放置语句最后：

```
EXPLAIN SELECT SQL_NO_CACHE * FROM student WHERE student.age=30 AND student.name =
'abc' AND student.classId>20 ;
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | student | NULL      | range | idx_age_classid_name, idx_age_name_classid | idx_age_name_classid | 73 | NULL | 1 | 100.00 | Using index condition |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 2 warnings (0.00 sec)
```

2.7 不等于(!= 或者<>)索引失效

2.8 is null可以使用索引, is not null无法使用索引

```
EXPLAIN SELECT SQL_NO_CACHE * FROM student WHERE age IS NULL;
```

```
EXPLAIN SELECT SQL_NO_CACHE * FROM student WHERE age IS NOT NULL;
```

2.9 like以通配符%开头索引失效

```
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | student | NULL      | ALL   | NULL          | NULL | NULL    | NULL | 7737618 | 11.11 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 2 warnings (0.00 sec)
```

拓展：Alibaba《Java开发手册》

【强制】页面搜索严禁左模糊或者全模糊，如果需要请走搜索引擎来解决。

2.10 OR 前后存在非索引的列，索引失效

未使用到索引

```
EXPLAIN SELECT SQL_NO_CACHE * FROM student WHERE age = 10 OR classid = 100;
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | student | NULL      | ALL   | idx_age_classid_name, idx_age_name_classid | NULL | NULL    | NULL | 7737618 | 10.01 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 2 warnings (0.00 sec)
```

使用到索引

```
EXPLAIN SELECT SQL_NO_CACHE * FROM student WHERE age = 10 OR name = 'Abel';
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | student | NULL      | index_merge | idx_age_classid_name, idx_age_name_classid, idx_name_idx_age | idx_age_idx_name | 5,63 | NULL | 38569 | 100.00 | Using union(idx_age, idx_name); Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 2 warnings (0.01 sec)
```

2.11 数据库和表的字符集统一使用utf8mb4

统一使用utf8mb4(5.5.3版本以上支持)兼容性更好，统一字符集可以避免由于字符集转换产生的乱码。不同的字符集进行比较前需要进行转换会造成索引失效。

3. 关联查询优化

3.1 数据准备

3.2 采用左外连接

下面开始 EXPLAIN 分析

```
EXPLAIN SELECT SQL_NO_CACHE * FROM `type` LEFT JOIN book ON type.card = book.card;
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | type   | NULL      | ALL    | NULL          | NULL | NULL    | NULL | 20   | 100.00 | NULL
| 1 | SIMPLE     | book   | NULL      | ALL    | NULL          | NULL | NULL    | NULL | 20   | 100.00 | Using where; Using join buffer (hash join)
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set, 2 warnings (0.01 sec)
```

结论：type 有 All

添加索引优化

```
ALTER TABLE book ADD INDEX Y ( card); #【被驱动表】，可以避免全表扫描
```

```
EXPLAIN SELECT SQL_NO_CACHE * FROM `type` LEFT JOIN book ON type.card = book.card;
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref           | rows | filtered | Extra
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | type   | NULL      | ALL    | NULL          | NULL | NULL    | NULL | 20   | 100.00 | NULL
| 1 | SIMPLE     | book   | NULL      | ref   | Y             | Y    | 4       | atguigu.type.card | 1   | 100.00 | Using index
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set, 2 warnings (0.00 sec)
```

可以看到第二行的 type 变为了 ref，rows 也变成了优化比较明显。这是由左连接特性决定的。LEFT JOIN 条件用于确定如何从右表搜索行，左边一定都有，所以 **右边是我们的关键点，一定需要建立索引**。

```
ALTER TABLE `type` ADD INDEX X (card); #【驱动表】，无法避免全表扫描
```

```
EXPLAIN SELECT SQL_NO_CACHE * FROM `type` LEFT JOIN book ON type.card = book.card;
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref           | rows | filtered | Extra
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | type   | NULL      | index | NULL          | X   | 4       | NULL | 20   | 100.00 | Using index
| 1 | SIMPLE     | book   | NULL      | ref   | Y             | Y   | 4       | atguigu.type.card | 1   | 100.00 | Using index
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set, 2 warnings (0.00 sec)
```

接着：

```
DROP INDEX Y ON book;
```

```
EXPLAIN SELECT SQL_NO_CACHE * FROM `type` LEFT JOIN book ON type.card = book.card;
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref           | rows | filtered | Extra
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | type   | NULL      | index | NULL          | X   | 4       | NULL | 20   | 100.00 | Using index
| 1 | SIMPLE     | book   | NULL      | ALL   | NULL          | NULL | NULL    | NULL | 20   | 100.00 | Using where; Using join buffer (hash join)
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set, 1 warning (0.00 sec)
```

3.3 采用内连接

```
drop index X on type;
drop index Y on book; (如果已经删除了可以不用再执行该操作)
```

换成 inner join (MySQL自动选择驱动表)

```
EXPLAIN SELECT SQL_NO_CACHE * FROM type INNER JOIN book ON type.card=book.card;
```

ID	Select_Type	Table	Partitions	Type	Possible_Keys	Key	Key_Len	Ref	Rows	Filtered	Extra
1	SIMPLE	type	NULL	ALL	NULL	NULL	NULL	NULL	20	100.00	NULL
1	SIMPLE	book	NULL	ALL	NULL	NULL	NULL	NULL	20	10.00	Using where; Using join buffer (hash join)

2 rows in set, 2 warnings (0.00 sec)

添加索引优化

```
ALTER TABLE book ADD INDEX Y ( card );
```

```
EXPLAIN SELECT SQL_NO_CACHE * FROM type INNER JOIN book ON type.card=book.card;
```

ID	Select_Type	Table	Partitions	Type	Possible_Keys	Key	Key_Len	Ref	Rows	Filtered	Extra
1	SIMPLE	type	NULL	ALL	NULL	NULL	NULL	NULL	20	100.00	NULL
1	SIMPLE	book	NULL	ref	Y	Y	4	atguigu.type.card	1	100.00	Using index

2 rows in set, 2 warnings (0.00 sec)

```
ALTER TABLE type ADD INDEX X ( card );
```

```
EXPLAIN SELECT SQL_NO_CACHE * FROM type INNER JOIN book ON type.card=book.card;
```

ID	Select_Type	Table	Partitions	Type	Possible_Keys	Key	Key_Len	Ref	Rows	Filtered	Extra
1	SIMPLE	book	NULL	index	Y	Y	4	NULL	20	100.00	Using index
1	SIMPLE	type	NULL	ref	X	X	4	atguigu.book.card	1	100.00	Using index

2 rows in set, 2 warnings (0.00 sec)

接着：

```
DROP INDEX X ON `type`;
```

```
EXPLAIN SELECT SQL_NO_CACHE * FROM type INNER JOIN book ON type.card=book.card;
```

```
mysql> EXPLAIN SELECT SQL_NO_CACHE * FROM `type` INNER JOIN book ON type.card=book.card;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | type   | NULL    | ALL  | NULL        | NULL | NULL   | NULL | 20  | 100.00 | NULL
| 1 | SIMPLE     | book   | NULL    | ref  | Y           | Y    | 4      | atguigudb2.type.card | 2  | 100.00 | Using index
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set, 2 warnings (0.00 sec)
```

接着：

```
ALTER TABLE `type` ADD INDEX X ( card );
```

```
EXPLAIN SELECT SQL_NO_CACHE * FROM `type` INNER JOIN book ON type.card=book.card;
```

ID	Select_Type	Table	Partitions	Type	Possible_Keys	Key	Key_Len	Ref	Rows	Filtered	Extra
1	SIMPLE	book	NULL	index	Y	Y	4	NULL	20	100.00	Using index
1	SIMPLE	type	NULL	ref	X	X	4	atguigu.book.card	1	100.00	Using index

2 rows in set, 2 warnings (0.00 sec)

3.4 join语句原理

- Index Nested-Loop Join

我们来看一下这个语句：

```
EXPLAIN SELECT * FROM t1 STRAIGHT_JOIN t2 ON (t1.a=t2.a);
```

如果直接使用join语句，MySQL优化器可能会选择表t1或t2作为驱动表，这样会影响我们分析SQL语句的执行过程。所以，为了便于分析执行过程中的性能问题，我改用 `straight_join` 让MySQL使用固定的方式执行查询，这样优化器只会按照我们指定的方式去join。在这个语句里，t1是驱动表，t2是被驱动表。

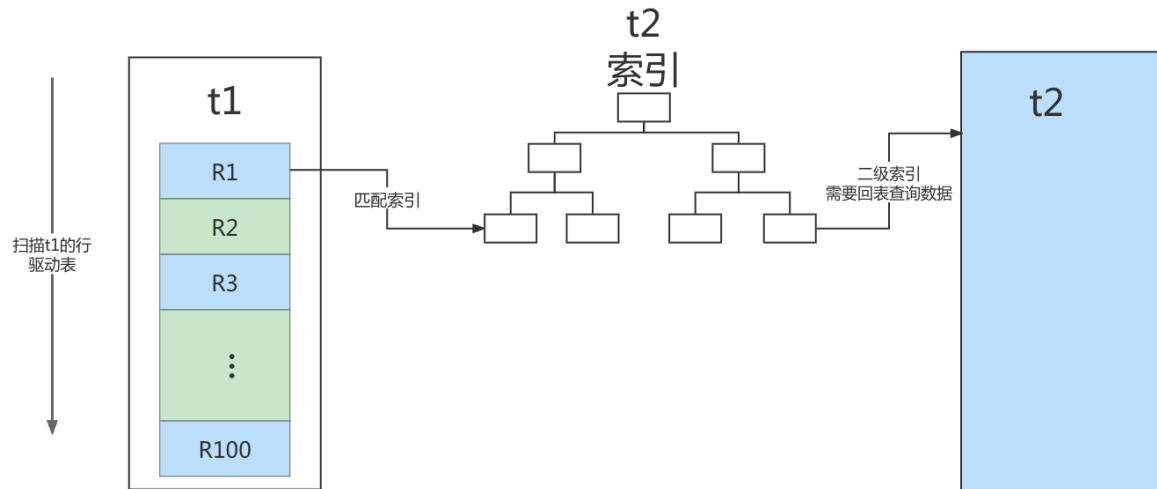
```
mysql> explain select * from t1 straight_join t2 on (t1.a=t2.a);
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id  | select_type | table | partitions | type | possible_keys | key | key_len | ref   | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1   | SIMPLE     | t1    | NULL       | ALL  | a             | a    | 5        | NULL  | 100  | 100.00  | Using where |
| 1   | SIMPLE     | t2    | NULL       | ref  | a             | a    | 5        | test.t1.a | 1    | 100.00  | NULL      |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

可以看到，在这条语句里，被驱动表t2的字段a上有索引，join过程用上了这个索引，因此这个语句的执行流程是这样的：

1. 从表t1中读入一行数据 R；
2. 从数据行R中，取出a字段到表t2里去查找；
3. 取出表t2中满足条件的行，跟R组成一行，作为结果集的一部分；
4. 重复执行步骤1到3，直到表t1的末尾循环结束。

这个过程是先遍历表t1，然后根据从表t1中取出的每行数据中的a值，去表t2中查找满足条件的记录。在形式上，这个过程就跟我们写程序时的嵌套查询类似，并且可以用上被驱动表的索引，所以我们称之为“Index Nested-Loop Join”，简称NLJ。

它对应的流程图如下所示：



在这个流程里：

1. 对驱动表t1做了全表扫描，这个过程需要扫描100行；
2. 而对于每一行R，根据a字段去表t2查找，走的是树搜索过程。由于我们构造的数据都是一一对应的，因此每次的搜索过程都只扫描一行，也是总共扫描100行；
3. 所以，整个执行流程，总扫描行数是200。

引申问题1：能不能使用join？

引申问题2：怎么选择驱动表？

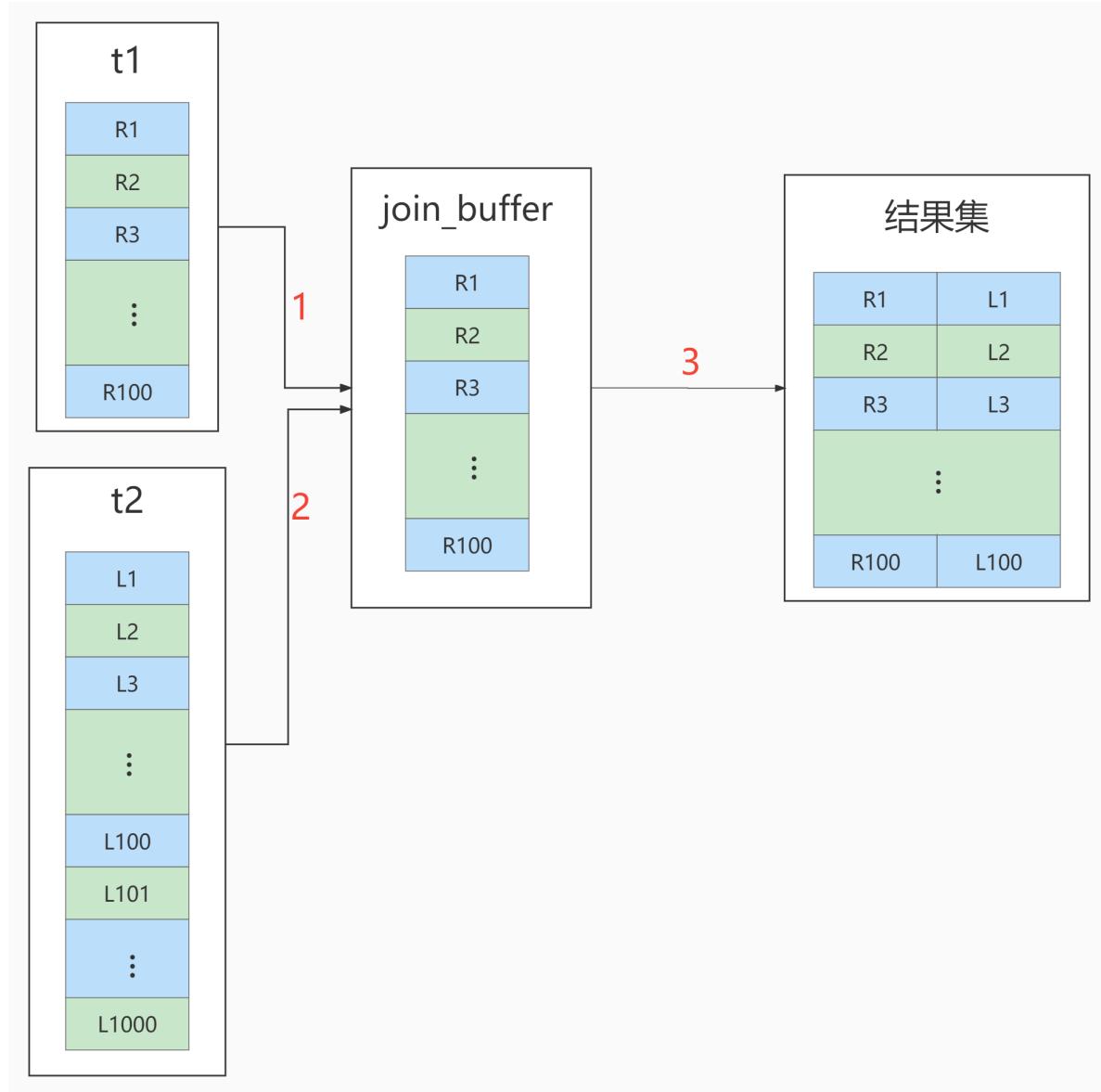
比如：N扩大1000倍的话，扫描行数就会扩大1000倍；而M扩大1000倍，扫描行数扩大不到10倍。

两个结论：

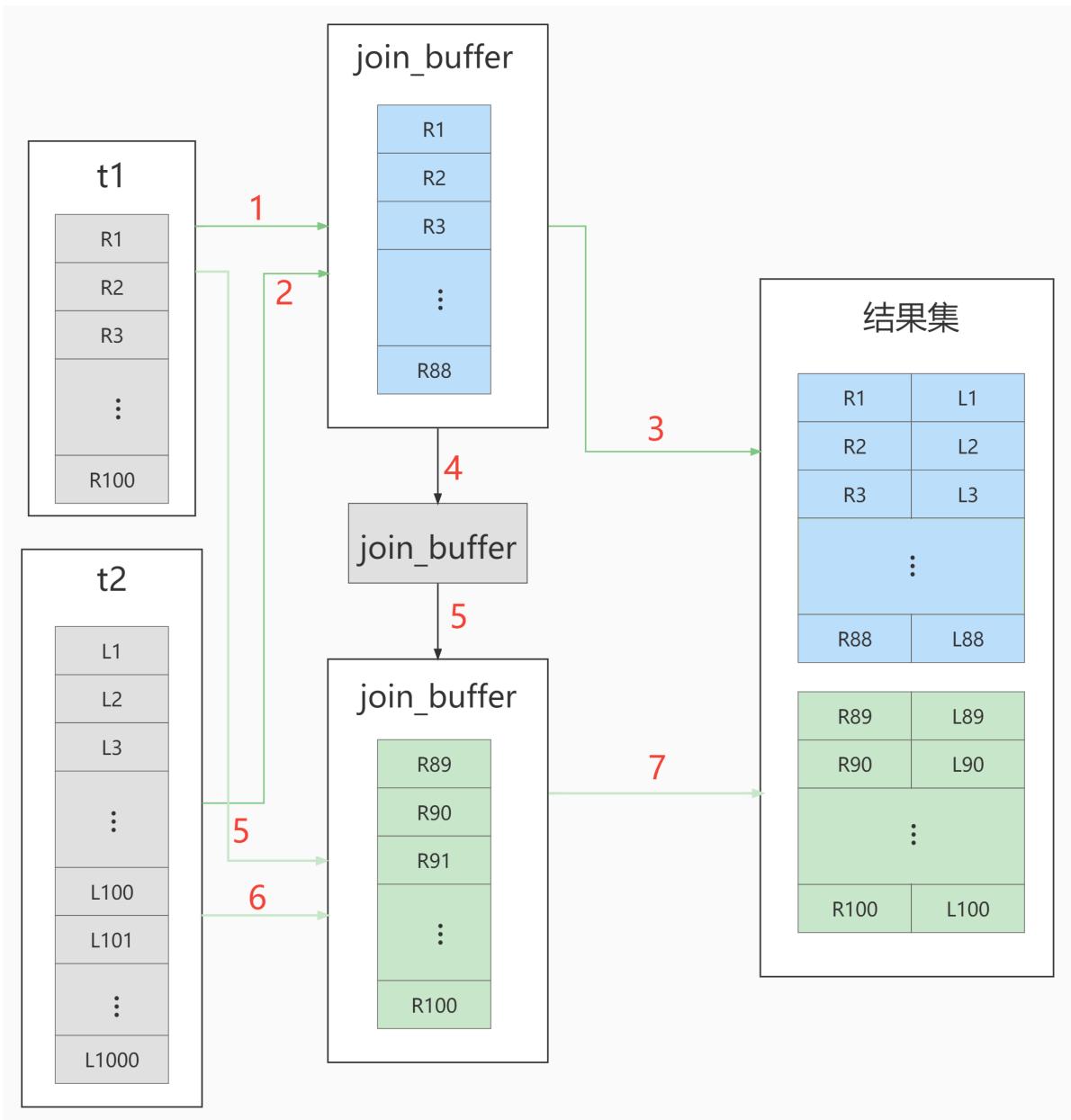
1. 使用join语句，性能比强行拆成多个单表执行SQL语句的性能要好；
2. 如果使用join语句的话，需要让小表做驱动表。

- Simple Nested-Loop Join
- Block Nested-Loop Join

这个过程的流程图如下：



执行流程图也就变成这样：



总结1：能不能使用xxx join语句？

总结2：如果要使用join，应该选择大表做驱动表还是选择小表做驱动表？

总结3：什么叫作“小表”？

在决定哪个表做驱动表的时候，应该是两个表按照各自的条件过滤，过滤完成之后，计算参与join的各个字段的总数据量，数据量小的那个表，就是“小表”，应该作为驱动表。

3.5 小结

- 保证被驱动表的JOIN字段已经创建了索引
- 需要JOIN 的字段，数据类型保持绝对一致。
- LEFT JOIN 时，选择小表作为驱动表，**大表作为被驱动表**。减少外层循环的次数。
- INNER JOIN 时，MySQL会自动将**小结果集的表选为驱动表**。选择相信MySQL优化策略。
- 能够直接多表关联的尽量直接关联，不用子查询。(减少查询的趟数)
- 不建议使用子查询，建议将子查询SQL拆开结合程序多次查询，或使用 JOIN 来代替子查询。
- 衍生表建不了索引

4. 子查询优化

MySQL从4.1版本开始支持子查询，使用子查询可以进行SELECT语句的嵌套查询，即一个SELECT查询的结果作为另一个SELECT语句的条件。**子查询可以一次性完成很多逻辑上需要多个步骤才能完成的SQL操作。**

子查询是 MySQL 的一项重要的功能，可以帮助我们通过一个 SQL 语句实现比较复杂的查询。但是，子查询的执行效率不高。原因：

- ① 执行子查询时，MySQL需要为内层查询语句的查询结果 **建立一个临时表**，然后外层查询语句从临时表中查询记录。查询完毕后，再 **撤销这些临时表**。这样会消耗过多的CPU和IO资源，产生大量的慢查询。
- ② 子查询的结果集存储的临时表，不论是内存临时表还是磁盘临时表都 **不会存在索引**，所以查询性能会受到一定的影响。
- ③ 对于返回结果集比较大的子查询，其对查询性能的影响也就越大。

在MySQL中，可以使用连接（JOIN）查询来替代子查询。连接查询 **不需要建立临时表**，其 **速度比子查询要快**，如果查询中使用索引的话，性能就会更好。

结论：尽量不要使用NOT IN 或者 NOT EXISTS，用LEFT JOIN xxx ON xx WHERE xx IS NULL替代

5. 排序优化

5.1 排序优化

问题：在 WHERE 条件字段上加索引，但是为什么在 ORDER BY 字段上还要加索引呢？

优化建议：

1. SQL 中，可以在 WHERE 子句和 ORDER BY 子句中使用索引，目的是在 WHERE 子句中 **避免全表扫描**，在 ORDER BY 子句 **避免使用 FileSort 排序**。当然，某些情况下全表扫描，或者 FileSort 排序不一定比索引慢。但总的来说，我们还是要避免，以提高查询效率。
2. 尽量使用 Index 完成 ORDER BY 排序。如果 WHERE 和 ORDER BY 后面是相同的列就使用单索引列；如果不同就使用联合索引。
3. 无法使用 Index 时，需要对 FileSort 方式进行调优。

```
INDEX a_b_c(a,b,c)

order by 能使用索引最左前缀
- ORDER BY a
- ORDER BY a,b
- ORDER BY a,b,c
- ORDER BY a DESC,b DESC,c DESC
```

如果WHERE使用索引的最左前缀定义为常量，则order by 能使用索引

```
- WHERE a = const ORDER BY b,c
- WHERE a = const AND b = const ORDER BY c
- WHERE a = const ORDER BY b,c
- WHERE a = const AND b > const ORDER BY b,c
```

不能使用索引进行排序

```
- ORDER BY a ASC,b DESC,c DESC /* 排序不一致 */
- WHERE g = const ORDER BY b,c /*丢失a索引*/
- WHERE a = const ORDER BY c /*丢失b索引*/
```

```
- WHERE a = const ORDER BY a,d /*d不是索引的一部分*/
- WHERE a in (...) ORDER BY b,c /*对于排序来说，多个相等条件也是范围查询*/
```

5.3 案例实战

ORDER BY子句，尽量使用Index方式排序，避免使用FileSort方式排序。

执行案例前先清除student上的索引，只留主键：

```
DROP INDEX idx_age ON student;
DROP INDEX idx_age_classid_stuno ON student;
DROP INDEX idx_age_classid_name ON student;

#或者
call proc_drop_index('atguigudb2','student');
```

场景：查询年龄为30岁的，且学生编号小于101000的学生，按用户名排序

```
EXPLAIN SELECT SQL_NO_CACHE * FROM student WHERE age = 30 AND stuno < 101000 ORDER BY NAME ;
```

```
+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra
+-----+-----+-----+-----+
| 1 | SIMPLE      | student | NULL      | ALL   | NULL          | NULL | NULL    | NULL | 7737618 | 3.33 | Using where; Using filesort |
+-----+-----+-----+-----+
1 row in set, 2 warnings (0.00 sec)
```

查询结果如下：

```
mysql> SELECT SQL_NO_CACHE * FROM student WHERE age = 30 AND stuno < 101000 ORDER BY NAME ;
+-----+-----+-----+-----+
| id | stuno | name | age | classId |
+-----+-----+-----+-----+
| 922 | 100923 | e1TLxD | 30 | 249 |
| 3723263 | 100412 | hKcjLb | 30 | 59 |
| 3724152 | 100827 | iHLJmh | 30 | 387 |
| 3724030 | 100776 | LgxWoD | 30 | 253 |
| 30 | 100031 | LZMOIa | 30 | 97 |
| 3722887 | 100237 | QzbJdx | 30 | 440 |
| 609 | 100610 | vbRimN | 30 | 481 |
| 139 | 100140 | ZqFbuR | 30 | 351 |
+-----+-----+-----+-----+
8 rows in set, 1 warning (3.16 sec)
```

结论：type 是 ALL，即最坏的情况。Extra 里还出现了 Using filesort, 也是最坏的情况。优化是必须的。

优化思路：

方案一：为了去掉filesort我们可以把索引建成

```
#创建新索引
CREATE INDEX idx_age_name ON student(age,NAME);
```

方案二：尽量让where的过滤条件和排序使用上索引

建一个三个字段的组合索引：

```
DROP INDEX idx_age_name ON student;

CREATE INDEX idx_age_stuno_name ON student (age,stuno,NAME);

EXPLAIN SELECT SQL_NO_CACHE * FROM student WHERE age = 30 AND stuno <101000 ORDER BY NAME ;
```

```
mysql> SELECT SQL_NO_CACHE * FROM student
    -> WHERE age = 30 AND stuno <101000 ORDER BY NAME ;
+----+-----+-----+-----+
| id | stuno | name   | age   | classId |
+----+-----+-----+-----+
| 167 | 100168 | AC1xEF | 30    |      319 |
| 323 | 100324 | bwbTpQ | 30    |      654 |
| 651 | 100652 | DRwIac | 30    |      997 |
| 517 | 100518 | HNSYqJ | 30    |      256 |
| 344 | 100345 | JuepiX | 30    |      329 |
| 905 | 100906 | JuWALd | 30    |      892 |
| 574 | 100575 | kbyqjX | 30    |      260 |
| 703 | 100704 | KJbprS | 30    |      594 |
| 723 | 100724 | OTdJKY | 30    |      236 |
| 656 | 100657 | Pfgqmj | 30    |      600 |
| 982 | 100983 | qywLqw | 30    |      837 |
| 468 | 100469 | sLEKQW | 30    |      346 |
| 988 | 100989 | UBYqJ1 | 30    |      457 |
| 173 | 100174 | UltkTN | 30    |      830 |
| 332 | 100333 | YjWiZw | 30    |      824 |
+----+-----+-----+-----+
15 rows in set, 1 warning (0.00 sec)
```

结果竟然有 filesort 的 sql 运行速度，超过了已经优化掉 filesort 的 sql，而且快了很多，几乎一瞬间就出现了结果。

结论：

- 两个索引同时存在，mysql自动选择最优的方案。（对于这个例子，mysql选择 idx_age_stuno_name）。但是，随着数据量的变化，选择的索引也会随之变化的。
- 当【范围条件】和【group by 或者 order by】的字段出现二选一时，优先观察条件字段的过滤数量，如果过滤的数据足够多，而需要排序的数据并不多时，优先把索引放在范围字段上。反之，亦然。**

思考：这里我们使用如下索引，是否可行？

```
DROP INDEX idx_age_stuno_name ON student;

CREATE INDEX idx_age_stuno ON student(age,stuno);
```

5.4 filesort算法：双路排序和单路排序

双路排序（慢）

- MySQL 4.1之前是使用双路排序，字面意思就是两次扫描磁盘，最终得到数据，读取行指针和order by列，对他们进行排序，然后扫描已经排序好的列表，按照列表中的值重新从列表中读取对应的数据输出

- 从磁盘取排序字段，在buffer进行排序，再从磁盘取其他字段。

取一批数据，要对磁盘进行两次扫描，众所周知，IO是很耗时的，所以在mysql4.1之后，出现了第二种改进的算法，就是单路排序。

单路排序（快）

从磁盘读取查询需要的 所有列，按照order by列在buffer对它们进行排序，然后扫描排序后的列表进行输出，它的效率更快一些，避免了第二次读取数据。并且把随机IO变成了顺序IO，但是它会使用更多的空间，因为它把每一行都保存在内存中了。

结论及引申出的问题

- 由于单路是后出的，总体而言好过双路
- 但是用单路有问题

优化策略

1. 尝试提高 sort_buffer_size

2. 尝试提高 max_length_for_sort_data

3. Order by 时select * 是一个大忌。最好只Query需要的字段。

6. GROUP BY优化

- group by 使用索引的原则几乎跟order by一致，group by 即使没有过滤条件用到索引，也可以直接使用索引。
- group by 先排序再分组，遵照索引建的最佳左前缀法则
- 当无法使用索引列，增大 max_length_for_sort_data 和 sort_buffer_size 参数的设置
- where效率高于having，能写在where限定的条件就不要写在having中了
- 减少使用order by，和业务沟通能不排序就不排序，或将排序放到程序端去做。Order by、group by、distinct这些语句较为耗费CPU，数据库的CPU资源是极其宝贵的。
- 包含了order by、group by、distinct这些查询的语句，where条件过滤出来的结果集请保持在1000行以内，否则SQL会很慢。

7. 优化分页查询

优化思路一

在索引上完成排序分页操作，最后根据主键关联回原表查询所需要的其他列内容。

```
EXPLAIN SELECT * FROM student t,(SELECT id FROM student ORDER BY id LIMIT 2000000,10)
a
WHERE t.id = a.id;
```

```
mysql> EXPLAIN SELECT * FROM student t,(SELECT id FROM student ORDER BY id LIMIT 2000000,10) a
-> WHERE t.id = a.id;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | PRIMARY | <derived2> | NULL | ALL | NULL | NULL | NULL | NULL | 2000010 | 100.00 | NULL |
| 1 | PRIMARY | t | NULL | eq_ref | PRIMARY | PRIMARY | 4 | a.id | 1 | 100.00 | NULL |
| 2 | DERIVED | student | NULL | index | NULL | PRIMARY | 4 | NULL | 2000010 | 100.00 | Using index |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
3 rows in set, 1 warning (0.00 sec)
```

优化思路二

该方案适用于主键自增的表，可以把Limit查询转换成某个位置的查询。

```
EXPLAIN SELECT * FROM student WHERE id > 2000000 LIMIT 10;
```

```
mysql> EXPLAIN SELECT * FROM student WHERE id > 2000000 LIMIT 10;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | student | NULL | range | PRIMARY | PRIMARY | 4 | NULL | 1994559 | 100.00 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

8. 优先考虑覆盖索引

8.1 什么是覆盖索引？

理解方式一：索引是高效找到行的一个方法，但是一般数据库也能使用索引找到一个列的数据，因此它不必读取整个行。毕竟索引叶子节点存储了它们索引的数据；当能通过读取索引就可以得到想要的数据，那就不需要读取行了。**一个索引包含了满足查询结果的数据就叫做覆盖索引。**

理解方式二：非聚簇复合索引的一种形式，它包括在查询里的SELECT、JOIN和WHERE子句用到的所有列（即建索引的字段正好是覆盖查询条件中所涉及的字段）。

简单说就是，**索引列+主键** 包含 **SELECT 到 FROM之间查询的列**。

8.2 覆盖索引的利弊

好处：

1. 避免Innodb表进行索引的二次查询（回表）
2. 可以把随机IO变成顺序IO加快查询效率

弊端：

索引字段的维护 总是有代价的。因此，在建立冗余索引来支持覆盖索引时就需要权衡考虑了。这是业务DBA，或者称为业务数据架构师的工作。

9. 如何给字符串添加索引

有一张教师表，表定义如下：

```
create table teacher(
ID bigint unsigned primary key,
email varchar(64),
...
)engine=innodb;
```

讲师要使用邮箱登录，所以业务代码中一定会出现类似于这样的语句：

```
mysql> select col1, col2 from teacher where email='xxx';
```

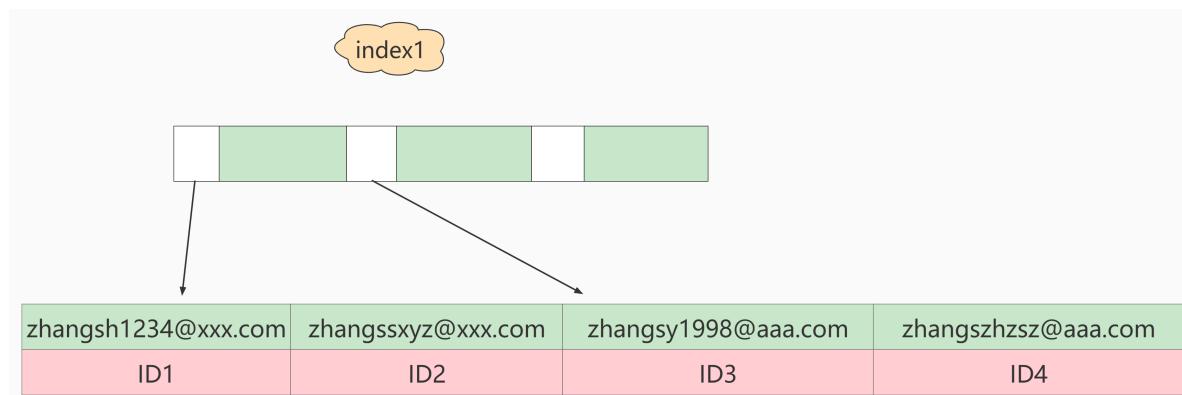
如果email这个字段上没有索引，那么这个语句就只能做 **全表扫描**。

9.1 前缀索引

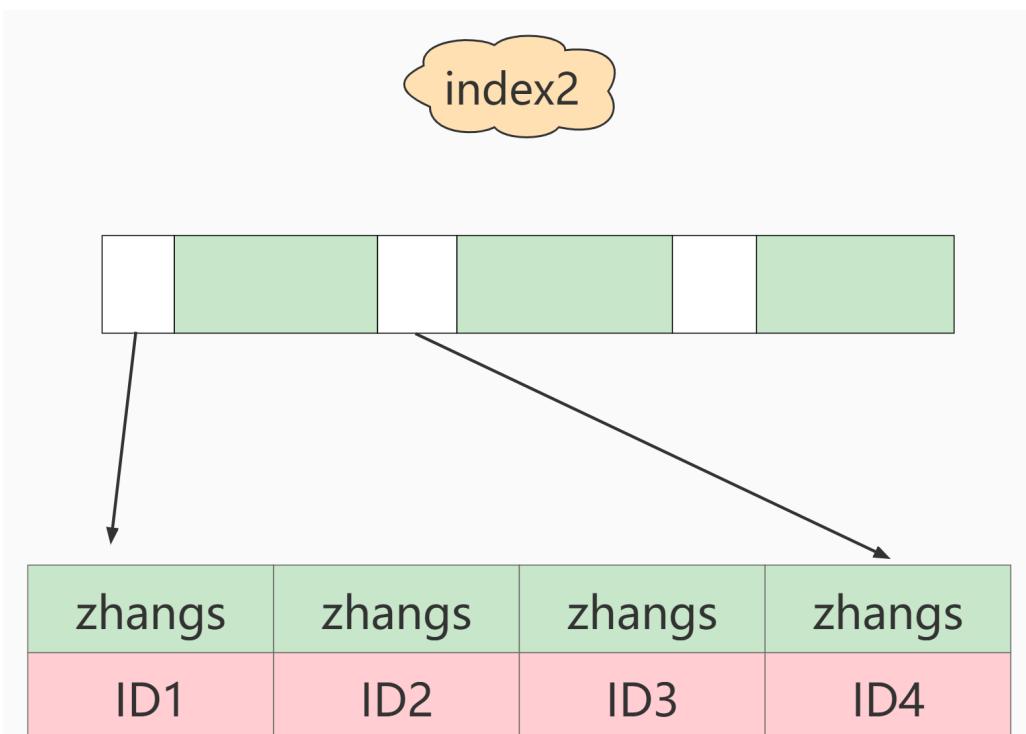
MySQL是支持前缀索引的。默认地，如果你创建索引的语句不指定前缀长度，那么索引就会包含整个字符串。

```
mysql> alter table teacher add index index1(email);
#或
mysql> alter table teacher add index index2(email(6));
```

这两种不同的定义在数据结构和存储上有什么区别呢？下图就是这两个索引的示意图。



以及



如果使用的是index1 (即email整个字符串的索引结构) , 执行顺序是这样的:

1. 从index1索引树找到满足索引值是' zhangssxyz@xxx.com '的这条记录, 取得ID2的值;
2. 到主键上查到主键值是ID2的行, 判断email的值是正确的, 将这行记录加入结果集;
3. 取index1索引树上刚刚查到的位置的下一条记录, 发现已经不满足email=' zhangssxyz@xxx.com '的条件了, 循环结束。

这个过程中, 只需要回主键索引取一次数据, 所以系统认为只扫描了一行。

如果使用的是index2 (即email(6)索引结构) , 执行顺序是这样的:

1. 从index2索引树找到满足索引值是'zhangs'的记录, 找到的第一个是ID1;
2. 到主键上查到主键值是ID1的行, 判断出email的值不是' zhangssxyz@xxx.com ', 这行记录丢弃;
3. 取index2上刚刚查到的位置的下一条记录, 发现仍然是'zhangs', 取出ID2, 再到ID索引上取整行然后判断, 这次值对了, 将这行记录加入结果集;
4. 重复上一步, 直到在idxe2上取到的值不是'zhangs'时, 循环结束。

也就是说**使用前缀索引, 定义好长度, 就可以做到既节省空间, 又不用额外增加太多的查询成本**。前面已经讲过区分度, 区分度越高越好。因为区分度越高, 意味着重复的键值越少。

9.2 前缀索引对覆盖索引的影响

结论:

使用前缀索引就用不上覆盖索引对查询性能的优化了, 这也是你在选择是否使用前缀索引时需要考虑的一个因素。

10. 索引下推

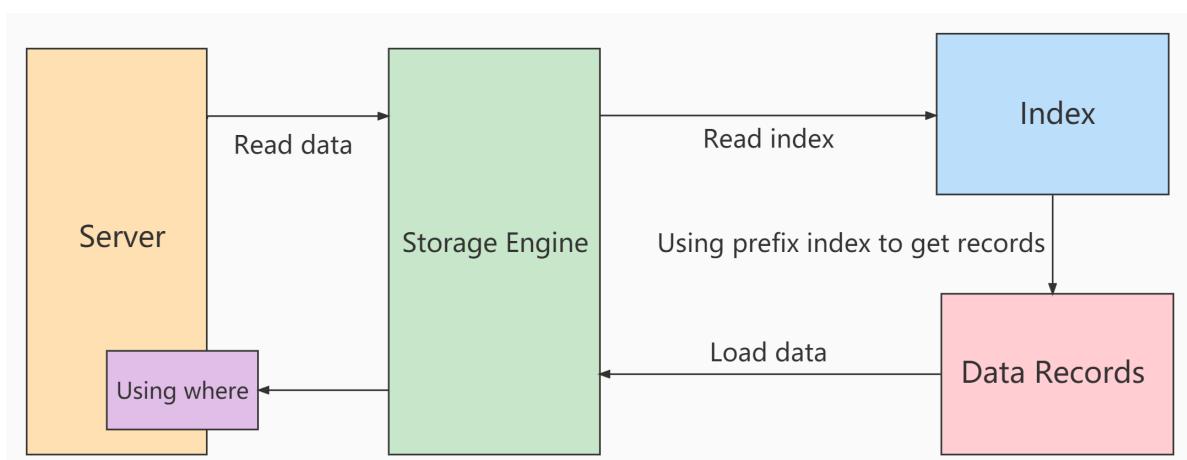
Index Condition Pushdown(ICP)是MySQL 5.6中新特性, 是一种在存储引擎层使用索引过滤数据的一种优化方式。ICP可以减少存储引擎访问基表的次数以及MySQL服务器访问存储引擎的次数。

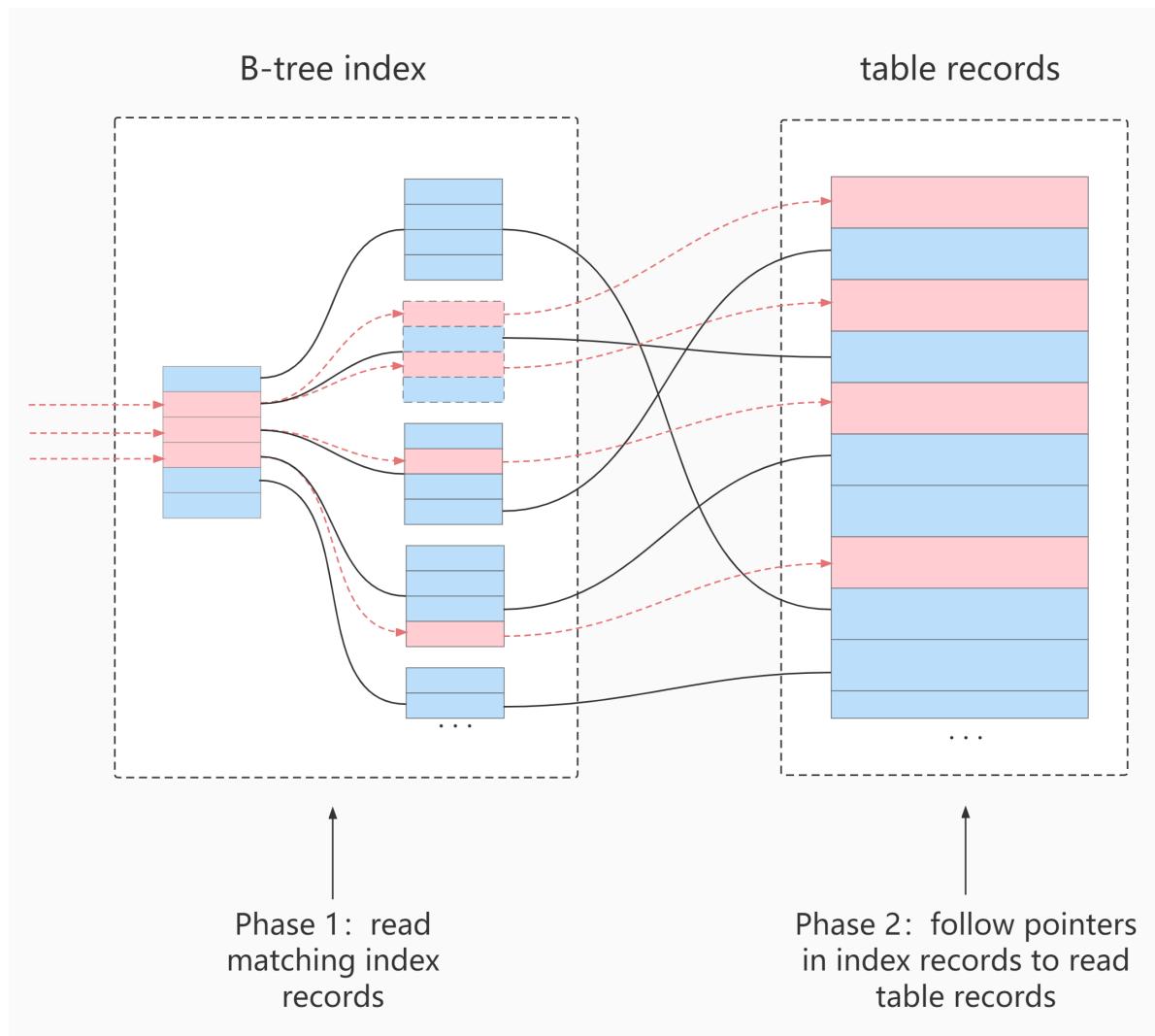
10.1 使用前后的扫描过程

在不使用ICP索引扫描的过程:

storage层: 只将满足index key条件的索引记录对应的整行记录取出, 返回给server层

server 层: 对返回的数据, 使用后面的where条件过滤, 直至返回最后一行。





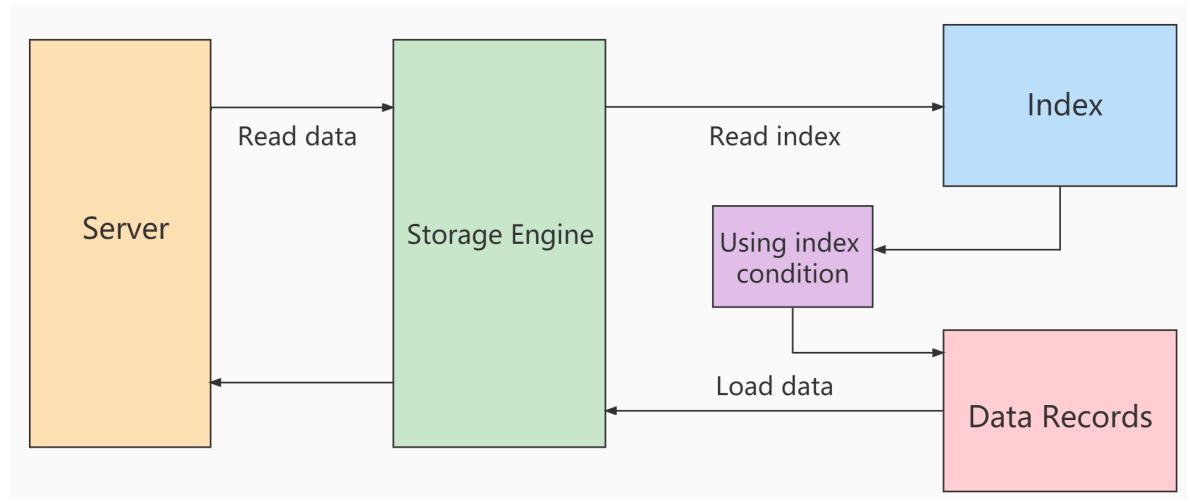
使用ICP扫描的过程：

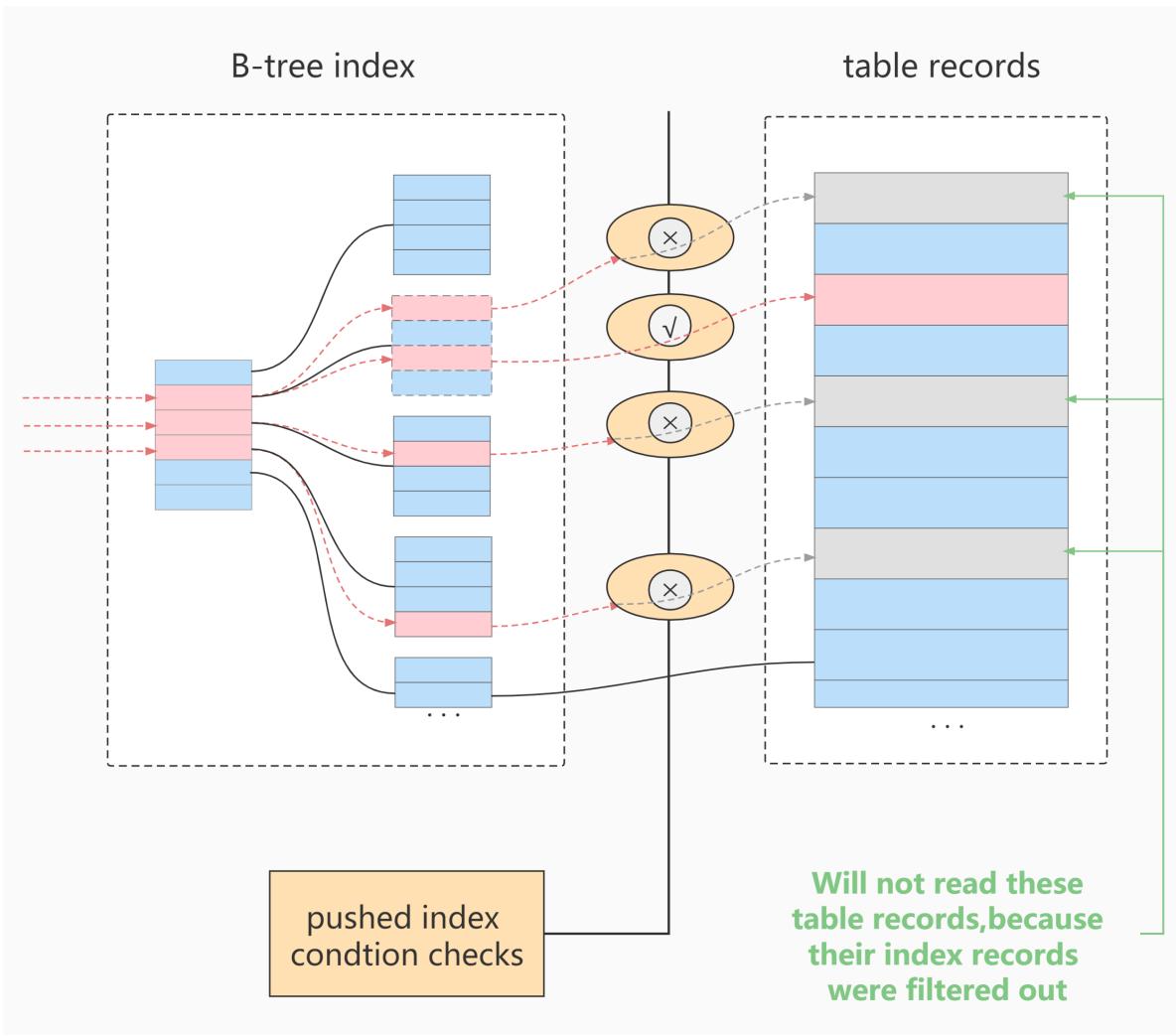
- storage层：

首先将index key条件满足的索引记录区间确定，然后在索引上使用index filter进行过滤。将满足的index filter条件的索引记录才去回表取出整行记录返回server层。不满足index filter条件的索引记录丢弃，不回表、也不会返回server层。

- server 层：

对返回的数据，使用table filter条件做最后的过滤。





使用前后的成本差别

使用前，存储层多返回了需要被index filter过滤掉的整行记录

使用ICP后，直接就去掉了不满足index filter条件的记录，省去了他们回表和传递到server层的成本。

ICP的 加速效果 取决于在存储引擎内通过 ICP筛选 掉的数据的比例。

10.2 ICP的使用条件

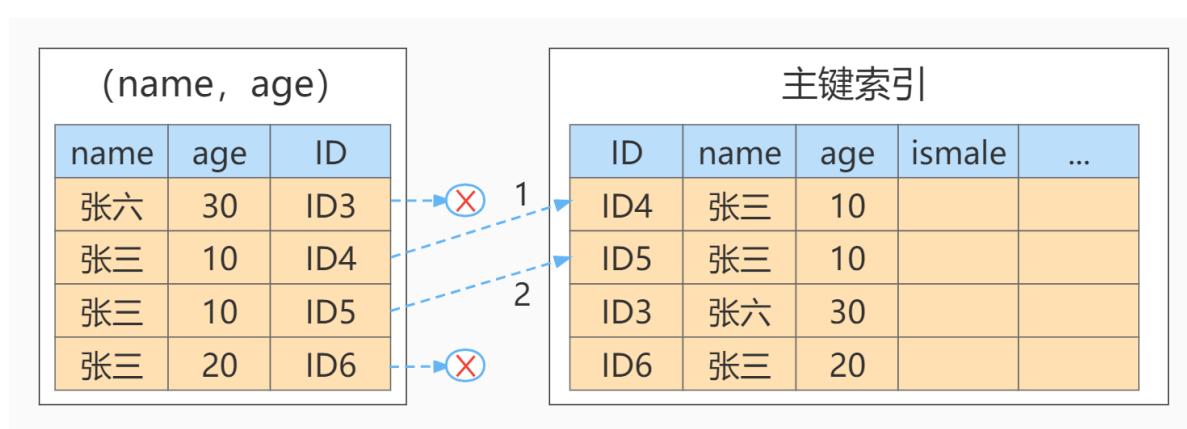
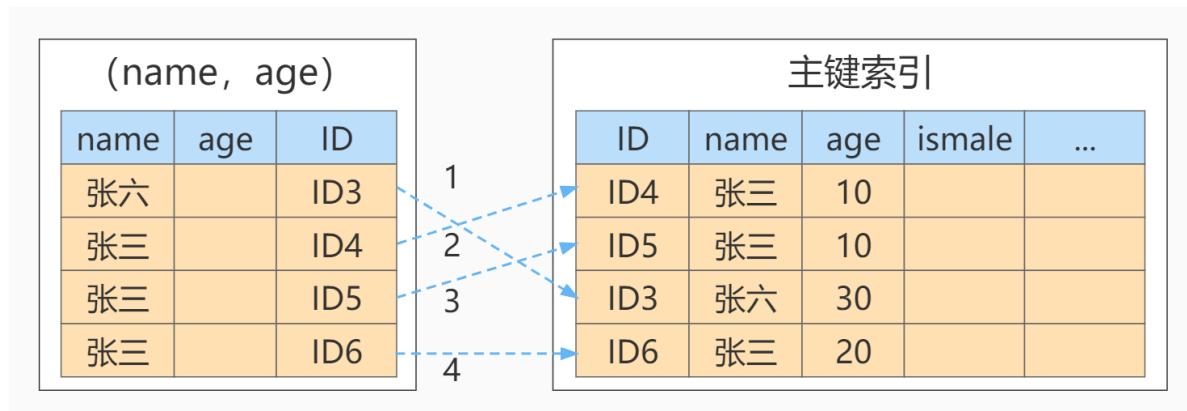
ICP的使用条件：

- ① 只能用于二级索引(secondary index)
- ②explain显示的执行计划中type值 (join 类型) 为 `range`、`ref`、`eq_ref` 或者 `ref_or_null`。
- ③ 并非全部where条件都可以用ICP筛选，如果where条件的字段不在索引列中，还是要读取整表的记录到server端做where过滤。
- ④ ICP可以用于MyISAM和InnnoDB存储引擎
- ⑤ MySQL 5.6版本的不支持分区表的ICP功能，5.7版本的开始支持。
- ⑥ 当SQL使用覆盖索引时，不支持ICP优化方法。

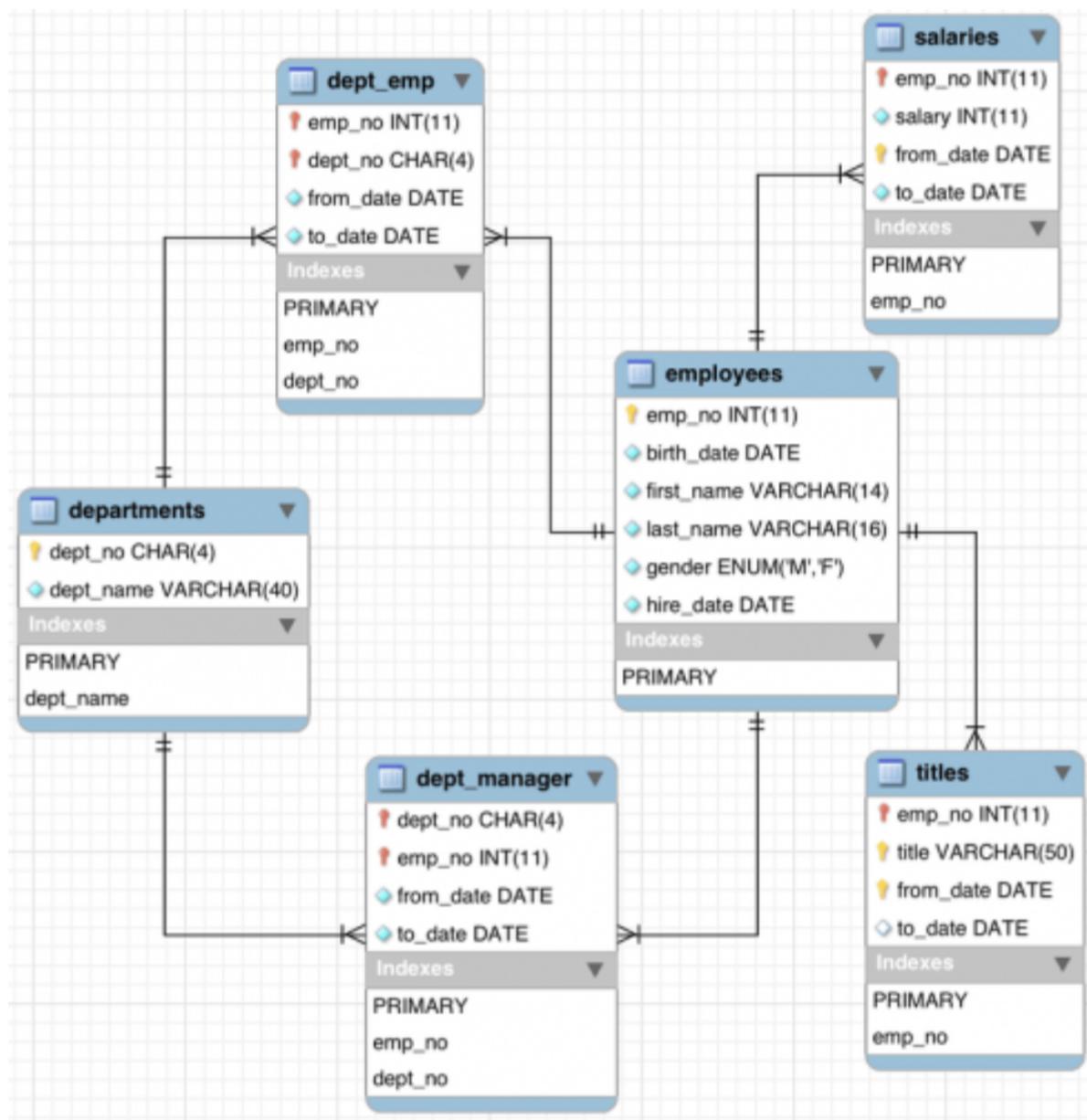
10.3 ICP使用案例

案例1

```
SELECT * FROM tuser  
WHERE NAME LIKE '张%'  
AND age = 10  
AND ismale = 1;
```



案例2



11. 普通索引 vs 唯一索引

从性能的角度考虑，你选择唯一索引还是普通索引呢？选择的依据是什么呢？

假设，我们有一个主键列为ID的表，表中有字段k，并且在k上有索引，假设字段 k 上的值都不重复。

这个表的建表语句是：

```
mysql> create table test(
    id int primary key,
    k int not null,
    name varchar(16),
    index (k)
)engine=InnoDB;
```

表中R1~R5的(ID,k)值分别为(100,1)、(200,2)、(300,3)、(500,5)和(600,6)。

11.1 查询过程

假设，执行查询的语句是 select id from test where k=5。

- 对于普通索引来说，查找到满足条件的第一个记录(5,500)后，需要查找下一个记录，直到碰到第一个不满足k=5条件的记录。
- 对于唯一索引来说，由于索引定义了唯一性，查找到第一个满足条件的记录后，就会停止继续检索。

那么，这个不同带来的性能差距会有多少呢？答案是，**微乎其微**。

11.2 更新过程

为了说明普通索引和唯一索引对更新语句性能的影响这个问题，介绍一下change buffer。

当需要更新一个数据页时，如果数据页在内存中就直接更新，而如果这个数据页还没有在内存中的话，在不影响数据一致性的前提下，InnoDB会将这些更新操作缓存在**change buffer**中，这样就不需要从磁盘中读入这个数据页了。在下次查询需要访问这个数据页的时候，将数据页读入内存，然后执行change buffer中与这个页有关的操作。通过这种方式就能保证这个数据逻辑的正确性。

将change buffer中的操作应用到原数据页，得到最新结果的过程称为**merge**。除了**访问这个数据页**会触发merge外，系统有**后台线程会定期 merge**。在**数据库正常关闭 (shutdown)**的过程中，也会执行merge操作。

如果能够将更新操作先记录在change buffer，**减少读磁盘**，语句的执行速度会得到明显的提升。而且，数据读入内存是需要占用**buffer pool**的，所以这种方式还能够**避免占用内存**，提高内存利用率。

唯一索引的更新就不能使用**change buffer**，实际上也只有普通索引可以使用。

如果要在这张表中插入一个新记录(4,400)的话，InnoDB的处理流程是怎样的？

11.3 change buffer的使用场景

1. 普通索引和唯一索引应该怎么选择？其实，这两类索引在查询能力上是没差别的，主要考虑的是对**更新性能**的影响。所以，建议你**尽量选择普通索引**。
 2. 在实际使用中会发现，**普通索引**和**change buffer**的配合使用，对于**数据量大**的表的更新优化还是很明显的。
 3. 如果所有的更新后面，都马上**伴随着对这个记录的查询**，那么你应该**关闭change buffer**。而在其他情况下，change buffer都能提升更新性能。
 4. 由于唯一索引用不上change buffer的优化机制，因此如果**业务可以接受**，从性能角度出发建议优先考虑非唯一索引。但是如果“**业务可能无法确保**”的情况下，怎么处理呢？
- 首先，**业务正确性优先**。我们的前提是“**业务代码已经保证不会写入重复数据**”的情况下，讨论性能问题。如果业务不能保证，或者业务就是要求数据库来做约束，那么没得选，必须创建唯一索引。这种情况下，本节的意义在于，如果碰上了大量插入数据慢、内存命中率低的时候，给你多提供一个排查思路。
 - 然后，在一些“**归档库**”的场景，你是可以考虑使用唯一索引的。比如，线上数据只需要保留半年，然后历史数据保存在归档库。这时候，归档数据已经是确保没有唯一键冲突了。要提高归档效率，可以考虑把表里面的唯一索引改成普通索引。

12. 其它查询优化策略

12.1 EXISTS 和 IN 的区分

问题：

不太理解哪种情况下应该使用 EXISTS，哪种情况应该用 IN。选择的标准是看能否使用表的索引吗？

12.2 COUNT(*)与COUNT(具体字段)效率

问：在 MySQL 中统计数据表的行数，可以使用三种方式：`SELECT COUNT(*)`、`SELECT COUNT(1)` 和 `SELECT COUNT(具体字段)`，使用这三者之间的查询效率是怎样的？

12.3 关于SELECT(*)

在表查询中，建议明确字段，不要使用 * 作为查询的字段列表，推荐使用`SELECT <字段列表>`查询。原因：

① MySQL 在解析的过程中，会通过 [查询数据字典](#) 将“*”按序转换成所有列名，这会大大的耗费资源和时间。

② 无法使用 [覆盖索引](#)

12.4 LIMIT 1 对优化的影响

针对的是会扫描全表的 SQL 语句，如果你可以确定结果集只有一条，那么加上 `LIMIT 1` 的时候，当找到一条结果的时候就不会继续扫描了，这样会加快查询速度。

如果数据表已经对字段建立了唯一索引，那么可以通过索引进行查询，不会全表扫描的话，就不需要加上 `LIMIT 1` 了。

12.5 多使用COMMIT

只要有可能，在程序中尽量多使用 COMMIT，这样程序的性能得到提高，需求也会因为 COMMIT 所释放的资源而减少。

COMMIT 所释放的资源：

- 回滚段上用于恢复数据的信息
- 被程序语句获得的锁
- redo / undo log buffer 中的空间
- 管理上述 3 种资源中的内部花费

13. 淘宝数据库，主键如何设计的？

聊一个实际问题：淘宝的数据库，主键是如何设计的？

某些错的离谱的答案还在网上年复一年的流传着，甚至还成为了所谓的MySQL军规。其中，一个最明显的错误就是关于MySQL的主键设计。

大部分人的回答如此自信：用8字节的 BIGINT 做主键，而不要用INT。[错！](#)

这样的回答，只站在了数据库这一层，而没有从业务的角度思考主键。主键就是一个自增ID吗？站在2022年的新年档口，用自增做主键，架构设计上可能连及格都拿不到。

13.1 自增ID的问题

自增ID做主键，简单易懂，几乎所有数据库都支持自增类型，只是实现上各自有所不同而已。自增ID除了简单，其他都是缺点，总体来看存在以下几方面的问题：

1. 可靠性不高

存在自增ID回溯的问题，这个问题直到最新版本的MySQL 8.0才修复。

2. 安全性不高

对外暴露的接口可以非常容易猜测对应的信息。比如：/User/1这样的接口，可以非常容易猜测用户ID的值为多少，总用户数量有多少，也可以非常容易地通过接口进行数据的爬取。

3. 性能差

自增ID的性能较差，需要在数据库服务器端生成。

4. 交互多

业务还需要额外执行一次类似 `last_insert_id()` 的函数才能知道刚才插入的自增值，这需要多一次的网络交互。在海量并发的系统中，多1条SQL，就多一次性能上的开销。

5. 局部唯一性

最重要的一点，自增ID是局部唯一，只在当前数据库实例中唯一，而不是全局唯一，在任意服务器间都是唯一的。对于目前分布式系统来说，这简直就是噩梦。

13.2 业务字段做主键

为了能够唯一地标识一个会员的信息，需要为 [会员信息表](#) 设置一个主键。那么，怎么为这个表设置主键，才能达到我们理想的目标呢？这里我们考虑业务字段做主键。

表数据如下：

cardno (卡号)	membername (名称)	memberphone (电话)	memberpid (身份证号)	address (地址)	sex (性别)	birthday (生日)
10000001	张三	13812345678	110123200001017890	北京	男	2000-01-01
10000002	李四	13512312312	123123199001012356	上海	女	1990-01-01

在这个表里，哪个字段比较合适呢？

• 选择卡号 (cardno)

会员卡号 (cardno) 看起来比较合适，因为会员卡号不能为空，而且有唯一性，可以用来标识一条会员记录。

```

mysql> CREATE TABLE demo.membermaster
-> (
-> cardno CHAR(8) PRIMARY KEY, -- 会员卡号为主键
-> membername TEXT,
-> memberphone TEXT,
-> memberpid TEXT,
-> memberaddress TEXT,
-> sex TEXT,
-> birthday DATETIME
-> );
Query OK, 0 rows affected (0.06 sec)

```

不同的会员卡号对应不同的会员，字段“cardno”唯一地标识某一个会员。如果都是这样，会员卡号与会员一一对应，系统是可以正常运行的。

但实际情况是，[会员卡号可能存在重复使用](#)的情况。比如，张三因为工作变动搬离了原来的地址，不再到商家的门店消费了（退还了会员卡），于是张三就不再是这个商家门店的会员了。但是，商家不想让这个会员卡空着，就把卡号是“10000001”的会员卡发给了王五。

从系统设计的角度看，这个变化只是修改了会员信息表中的卡号是“10000001”这个会员信息，并不会影响到数据一致性。也就是说，修改会员卡号是“10000001”的会员信息，系统的各个模块，都会获取到修改后的会员信息，不会出现“有的模块获取到修改之前的会员信息，有的模块获取到修改后的会员信息，而导致系统内部数据不一致”的情况。因此，从[信息系统层面](#)上看是没问题的。

但是从使用[系统的业务层面](#)来看，就有很大的问题了，会对商家造成影响。

比如，我们有一个销售流水表（trans），记录了所有的销售流水明细。2020年12月01日，张三在门店购买了一本书，消费了89元。那么，系统中就有了张三买书的流水记录，如下所示：

transactionno (流水单号)	itemnumber (商品编号)	quantity (销售数量)	price (价格)	salesvalue (销售金额)	cardno (会员卡号)	transdate (交易时间)
1	1	1	89	89	10000001	2020-12-01

接着，我们查询一下2020年12月01日的会员销售记录：

```

mysql> SELECT b.membername,c.goodsname,a.quantity,a.salesvalue,a.transdate
-> FROM demo.trans AS a
-> JOIN demo.membermaster AS b
-> JOIN demo.goodsmaster AS c
-> ON (a.cardno = b.cardno AND a.itemnumber=c.itemnumber);
+-----+-----+-----+-----+
| membername | goodsname | quantity | salesvalue | transdate |
+-----+-----+-----+-----+
| 张三       | 书         | 1.000    | 89.00     | 2020-12-01 00:00:00 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)

```

如果会员卡“10000001”又发给了王五，我们会更改会员信息表。导致查询时：

```

mysql> SELECT b.membername,c.goodsname,a.quantity,a.salesvalue,a.transdate
-> FROM demo.trans AS a
-> JOIN demo.membermaster AS b
-> JOIN demo.goodsmaster AS c
-> ON (a.cardno = b.cardno AND a.itemnumber=c.itemnumber);
+-----+-----+-----+-----+
| membername | goodsname | quantity | salesvalue | transdate      |
+-----+-----+-----+-----+
| 王五       | 书         | 1.000    | 89.00     | 2020-12-01 00:00:00 |
+-----+-----+-----+-----+
1 row in set (0.01 sec)

```

这次得到的结果是：王五在 2020 年 12 月 01 日，买了一本书，消费 89 元。显然是错误的！结论：千万不能把会员卡号当做主键。

- **选择会员电话或身份证号**

会员电话可以做主键吗？不行的。在实际操作中，手机号也存在 **被运营商收回**，重新发给别人用的情况。

那身份证号行不行呢？好像可以。因为身份证决不会重复，身份证号与一个人存在一一对应的关系。可问题是，身份证号属于 **个人隐私**，顾客不一定愿意给你。要是强制要求会员必须登记身份证号，会把很多客人赶跑的。其实，客户电话也有这个问题，这也是我们在设计会员信息表的时候，允许身份证号和电话都为空的原因。

所以，建议尽量不要用跟业务有关的字段做主键。毕竟，作为项目设计的技术人员，我们谁也无法预测在项目的整个生命周期中，哪个业务字段会因为项目的业务需求而有重复，或者重用之类的情况出现。

经验：

刚开始使用 MySQL 时，很多人都很容易犯的错误是喜欢用业务字段做主键，想当然地认为了解业务需求，但实际情况往往出乎意料，而更改主键设置的成本非常高。

13.3 淘宝的主键设计

在淘宝的电商业务中，订单服务是一个核心业务。请问，**订单表的主键** 淘宝是如何设计的呢？是自增ID吗？

打开淘宝，看一下订单信息：

2021-02-01 订单号: 1550672064762308113

中国电信官...

 和我联系



中国电信官方旗舰店 上海手机充值 100元电信话费直充
快充电信充值

¥100.00

1

申请售后
投诉商家

2021-01-01 订单号: 1481195847180308113

中国电信官...

 和我联系



中国电信官方旗舰店 上海手机充值 100元电信话费直充
快充电信充值

¥100.00

1

申请售后

2020-12-11 订单号: 1431156171142308113

仁创话费充...

 和我联系



官方上海移动50元手机话费充值 自动极速充即时到账

¥49.90

1

申请售后

2020-12-11 订单号: 1431146631521308113

中国电信官...

 和我联系



中国电信官方旗舰店 广东手机充值 30元电信话费直充快
充 电信充值

¥29.94

1

申请售后

从上图可以发现，订单号不是自增ID！我们详细看下上述4个订单号：

1550672064762308113

1481195847180308113

1431156171142308113

1431146631521308113

订单号是19位的长度，且订单的最后5位都是一样的，都是08113。且订单号的前面14位部分是单调递增的。

大胆猜测，淘宝的订单ID设计应该是：

订单ID = 时间 + 去重字段 + 用户ID后6位尾号

这样的设计能做到全局唯一，且对分布式系统查询及其友好。

13.4 推荐的主键设计

非核心业务：对应表的主键自增ID，如告警、日志、监控等信息。

核心业务：**主键设计至少应该是全局唯一且是单调递增**。全局唯一保证在各系统之间都是唯一的，单调递增是希望插入时不影响数据库性能。

这里推荐最简单的一种主键设计：UUID。

UUID的特点：

全局唯一，占用36字节，数据无序，插入性能差。

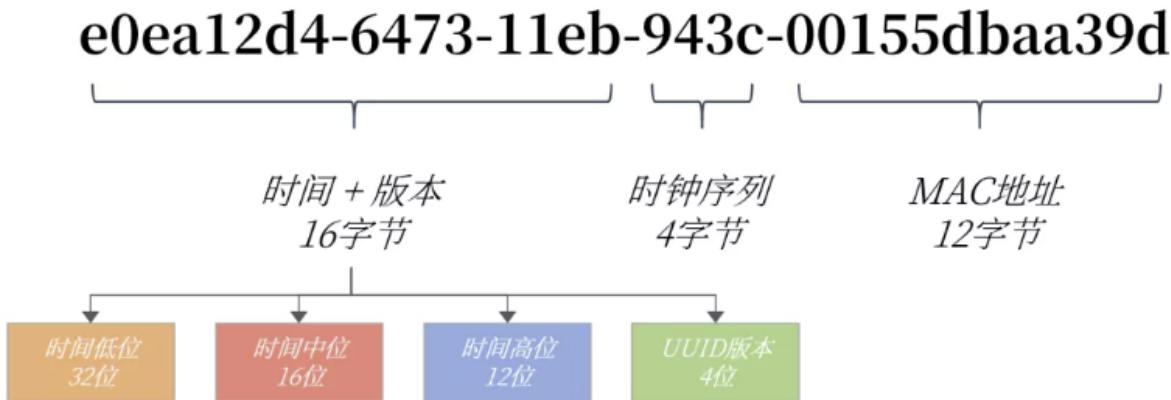
认识UUID:

- 为什么UUID是全局唯一的?
- 为什么UUID占用36个字节?
- 为什么UUID是无序的?

MySQL数据库的UUID组成如下所示:

UUID = 时间+UUID版本 (16字节) - 时钟序列 (4字节) - MAC地址 (12字节)

我们以UUID值e0ea12d4-6473-11eb-943c-00155dbaa39d举例:



为什么UUID是全局唯一的?

在UUID中时间部分占用60位，存储的类似TIMESTAMP的时间戳，但表示的是从1582-10-15 00: 00: 00.00 到现在的100ns的计数。可以看到UUID存储的时间精度比TIMESTAMPE更高，时间维度发生重复的概率降低到1/100ns。

时钟序列是为了避免时钟被回拨导致产生时间重复的可能性。MAC地址用于全局唯一。

为什么UUID占用36个字节?

UUID根据字符串进行存储，设计时还带有无用"-"字符串，因此总共需要36个字节。

为什么UUID是随机无序的呢?

因为UUID的设计中，将时间低位放在最前面，而这部分的数据是一直在变化的，并且是无序。

改造UUID

若将时间高低位互换，则时间就是单调递增的了，也就变得单调递增了。MySQL 8.0可以更换时间低位和时间高位的存储方式，这样UUID就是有序的UUID了。

MySQL 8.0还解决了UUID存在的空间占用的问题，除去了UUID字符串中无意义的"-"字符串，并且将字符串用二进制类型保存，这样存储空间降低为了16字节。

可以通过MySQL8.0提供的uuid_to_bin函数实现上述功能，同样的，MySQL也提供了bin_to_uuid函数进行转化：

```
SET @uuid = UUID();  
  
SELECT @uuid,uuid_to_bin(@uuid),uuid_to_bin(@uuid,TRUE);
```

```
mysql> SET @uuid = UUID();  
Query OK, 0 rows affected (0.00 sec)  
  
mysql> SELECT @uuid,uuid_to_bin(@uuid),uuid_to_bin(@uuid,TRUE);  
+-----+-----+-----+  
| @uuid | uuid_to_bin(@uuid) | uuid_to_bin(@uuid,TRUE) |  
+-----+-----+-----+  
| 71c8dc8a-6533-11ec-a652-000c2923a5e8 | 0x71C8DC8A653311ECA652000C2923A5E8 | 0x11EC653371C8DC8AA652000C2923A5E8 |  
+-----+-----+-----+  
1 row in set (0.00 sec)
```

通过函数`uuid_to_bin(@uuid,true)`将UUID转化为有序UUID了。全局唯一 + 单调递增，这不就是我们想要的主键！

4、有序UUID性能测试

16字节的有序UUID，相比之前8字节的自增ID，性能和存储空间对比究竟如何呢？

我们来做一个测试，插入1亿条数据，每条数据占用500字节，含有3个二级索引，最终的结果如下所示：

	时间 (秒)	表大小 (G)
自增ID	2712	240
UUID	3396	250
有序UUID	2624	243

从上图可以看到插入1亿条数据有序UUID是最快的，而且在实际业务使用中有序UUID在[业务端就可以生成](#)。还可以进一步减少SQL的交互次数。

另外，虽然有序UUID相比自增ID多了8个字节，但实际只增大了3G的存储空间，还可以接受。

在当今的互联网环境中，非常不推荐自增ID作为主键的数据库设计。更推荐类似有序UUID的全局唯一的实现。

另外在真实的业务系统中，主键还可以加入业务和系统属性，如用户的尾号，机房的信息等。这样的主键设计就更为考验架构师的水平了。

如果不是MySQL8.0 肿么办？

手动赋值字段做主键！

比如，设计各个分店的会员表的主键，因为如果每台机器各自产生的数据需要合并，就可能会出现主键重复的问题。

可以在总部 MySQL 数据库中，有一个管理信息表，在这个表中添加一个字段，专门用来记录当前会员编号的最大值。

门店在添加会员的时候，先到总部 MySQL 数据库中获取这个最大值，在这个基础上加 1，然后用这个值作为新会员的“id”，同时，更新总部 MySQL 数据库管理信息表中的当前会员编号的最大值。

这样一来，各个门店添加会员的时候，都对同一个总部 MySQL 数据库中的数据表字段进行操作，就解决了各门店添加会员时会员编号冲突的问题。

第11章_数据库的设计规范

讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

1. 为什么需要数据库设计

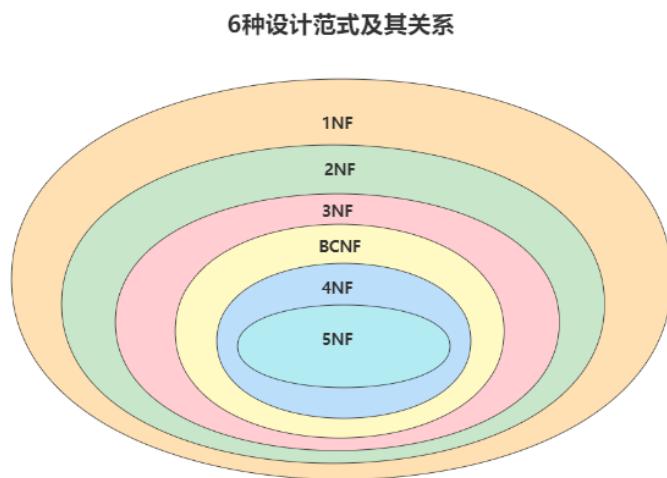
2. 范式

2.1 范式简介

在关系型数据库中，关于数据表设计的基本原则、规则就称为范式。可以理解为，一张数据表的设计结构需要满足的某种设计标准的 级别。要想设计一个结构合理的关系型数据库，必须满足一定的范式。

2.2 范式都包括哪些

目前关系型数据库有六种常见范式，按照范式级别，从低到高分别是：**第一范式（1NF）、第二范式（2NF）、第三范式（3NF）、巴斯-科德范式（BCNF）、第四范式（4NF）和第五范式（5NF，又称完美范式）。**



2.3 键和相关属性的概念

举例：

这里有两个表：

球员表(player)：球员编号 | 姓名 | 身份证号 | 年龄 | 球队编号

球队表(team)：球队编号 | 主教练 | 球队所在地

- **超键**：对于球员表来说，超键就是包括球员编号或者身份证号的任意组合，比如（球员编号）（球员编号，姓名）（身份证号，年龄）等。
- **候选键**：就是最小的超键，对于球员表来说，候选键就是（球员编号）或者（身份证号）。
- **主键**：我们自己选定，也就是从候选键中选择一个，比如（球员编号）。
- **外键**：球员表中的球队编号。
- **主属性、非主属性**：在球员表中，主属性是（球员编号）（身份证号），其他的属性（姓名）（年龄）（球队编号）都是非主属性。

2.4 第一范式(1st NF)

举例1：

假设一家公司要存储员工的姓名和联系方式。它创建一个如下表：

emp_id	emp_name	emp_address	emp_mobile
101	zhangsan	beijing	8912312390
102	lisi	liaoning	8812121212 9900012222
103	wangwu	hebei	7778881212
104	zhaoliu	shanghai	9999000012 1878120923

该表不符合 1NF，因为规则说“表的每个属性必须具有原子（单个）值”，lisi 和 zhaoliu 员工的 emp_mobile 值违反了该规则。为了使表符合 1NF，我们应该有如下表数据：

emp_id	emp_name	emp_address	emp_mobile
101	zhangsan	beijing	8912312390
102	lisi	liaoning	8812121212
102	lisi	liaoning	9900012222
103	wangwu	hebei	7778881212
104	zhaoliu	shanghai	9999000012
104	zhaoliu	shanghai	1878120923

举例2：

user 表的设计不符合第一范式

字段名称	字段类型	是否是主键	说明
id	INT	是	主键id
username	VARCHAR(30)	否	用户名
password	VARCHAR(50)	否	密码
user_info	VARCHAR(255)	否	用户信息(包含真实姓名、电话、住址)

其中，user_info 字段为用户信息，可以进一步拆分成更小粒度的字段，不符合数据库设计对第一范式的要求。将 user_info 拆分后如下：

字段名称	字段类型	是否是主键	说明
id	INT	是	主键id
username	VARCHAR(30)	否	用户名
password	VARCHAR(50)	否	密码
real_name	VARCHAR(30)	否	真实姓名
phone	VARCHAR(12)	否	联系电话
address	VARCHAR(100)	否	家庭住址

举例3：

属性的原子性是 **主观的**。例如，Employees关系中雇员姓名应当使用1个 (fullname)、2个 (firstname和lastname) 还是3个 (firstname、middlename和lastname) 属性表示呢？答案取决于应用程序。如果应用程序需要分别处理雇员的姓名部分（如：用于搜索目的），则有必要把它们分开。否则，不需要。

表1：

姓名	年龄	地址
张三	20	广东省广州市三元里78号
李四	24	广东省深圳市龙华新区

表2：

姓名	年龄	省	市	地址
张三	20	广东	广州	三元里78号
李四	24	广东	深圳	龙华新区

2.5 第二范式(2nd NF)

举例1：

成绩表 (学号, 课程号, 成绩) 关系中, (学号, 课程号) 可以决定成绩, 但是学号不能决定成绩, 课程号也不能决定成绩, 所以“(学号, 课程号) → 成绩”就是 **完全依赖关系**。

举例2：

比赛表 `player_game`, 里面包含球员编号、姓名、年龄、比赛编号、比赛时间和比赛场地等属性, 这里候选键和主键都为 (球员编号, 比赛编号), 我们可以通过候选键 (或主键) 来决定如下的关系:

(球员编号, 比赛编号) → (姓名, 年龄, 比赛时间, 比赛场地, 得分)

但是这个数据表不满足第二范式, 因为数据表中的字段之间还存在着如下的对应关系:

(球员编号) → (姓名, 年龄)

(比赛编号) → (比赛时间, 比赛场地)

对于非主属性来说, 并非完全依赖候选键。这样会产生怎样的问题呢?

- 数据冗余**：如果一个球员可以参加 m 场比赛，那么球员的姓名和年龄就重复了 $m-1$ 次。一个比赛也可能会有 n 个球员参加，比赛的时间和地点就重复了 $n-1$ 次。
- 插入异常**：如果我们想要添加一场新的比赛，但是这时还没有确定参加的球员都有谁，那么就无法插入。
- 删除异常**：如果我要删除某个球员编号，如果没有单独保存比赛表的话，就会同时把比赛信息删除掉。
- 更新异常**：如果我们调整了某个比赛的时间，那么数据表中所有这个比赛的时间都需要进行调整，否则就会出现一场比赛时间不同的情况。

为了避免出现上述的情况，我们可以把球员比赛表设计为下面的三张表。

表名	属性（字段）
球员 player 表	球员编号、姓名和年龄等属性
比赛 game 表	比赛编号、比赛时间和比赛场地等属性
球员比赛关系 player_game 表	球员编号、比赛编号和得分等属性

这样的话，每张数据表都符合第二范式，也就避免了异常情况的发生。

1NF 告诉我们字段属性需要是原子性的，而 2NF 告诉我们一张表就是一个独立的对象，一张表只表达一个意思。

举例3：

定义了一个名为 Orders 的关系，表示订单和订单行的信息：

Orders	
PK	<u>orderid</u>
PK	<u>productid</u>
	orderdate qty customerid companyname

违反了第二范式，因为有非主键属性仅依赖于候选键（或主键）的一部分。例如，可以仅通过 orderid 找到订单的 orderdate，以及 customerid 和 companyname，而没有必要再去使用 productid。

修改：

Orders 表和 OrderDetails 表如下，此时符合第二范式。

Orders		OrderDetails	
PK	<u>orderid</u>	PK,FK1	<u>orderid</u>
	orderdate customerid companyname	PK	<u>productid</u>
			qty

2.6 第三范式(3rd NF)

举例1：

部门信息表：每个部门有部门编号（dept_id）、部门名称、部门简介等信息。

员工信息表：每个员工有员工编号、姓名、部门编号。列出部门编号后就不能再将部门名称、部门简介等与部门有关的信息再加入员工信息表中。

如果不存在部门信息表，则根据第三范式（3NF）也应该构建它，否则就会有大量的数据冗余。

举例2：

字段名称	字段类型	是否是主键	说明
id	INT	是	商品主键id（主键）
category_id	INT	否	商品类别id
category_name	VARCHAR(30)	否	商品类别名称
goods_name	VARCHAR(30)	否	商品名称
price	DECIMAL(10,2)	否	商品价格

商品类别名称依赖于商品类别编号，不符合第三范式。

修改：

表1：符合第三范式的商品类别表的设计

字段名称	字段类型	是否是主键	说明
id	INT	是	商品类别主键id
category_name	VARCHAR(30)	否	商品类别名称

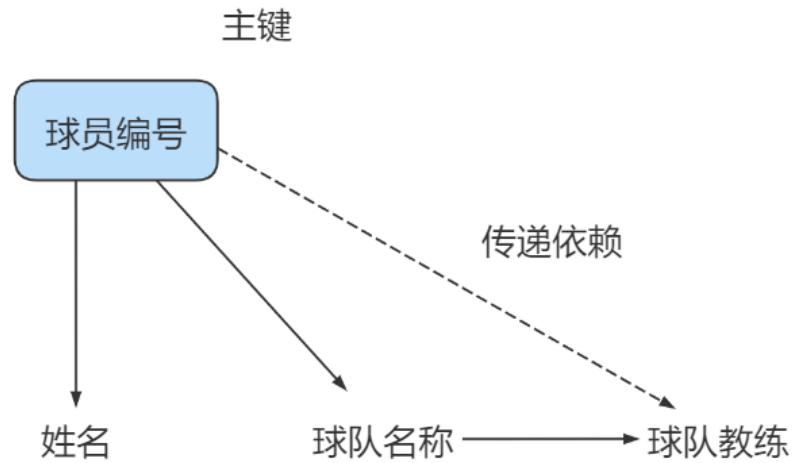
表2：符合第三范式的商品表的设计

字段名称	字段类型	是否是主键	说明
id	INT	是	商品主键id
category_id	VARCHAR(30)	否	商品类别id
goods_name	VARCHAR(30)	否	商品名称
price	DECIMAL(10,2)	否	商品价格

商品表goods通过商品类别id字段（category_id）与商品类别表goods_category进行关联。

举例3：

球员player表：球员编号、姓名、球队名称和球队主教练。现在，我们把属性之间的依赖关系画出来，如下图所示：



你能看到球员编号决定了球队名称，同时球队名称决定了球队主教练，非主属性球队主教练就会传递依赖于球员编号，因此不符合 3NF 的要求。

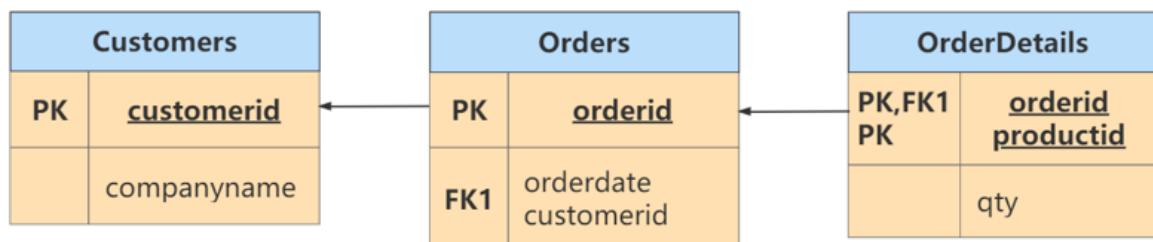
如果要达到 3NF 的要求，需要把数据表拆成下面这样：

表名	属性（字段）
球员表	球员编号、姓名和球队名称
球队表	球队名称、球队主教练

举例4：

修改第二范式中的举例3。

此时的Orders关系包含 orderid、orderdate、customerid 和 companyname 属性，主键定义为 orderid。customerid 和 companyname 均依赖于主键—orderid。例如，你需要通过orderid主键来查找代表订单中客户的customerid，同样，你需要通过 orderid 主键查找订单中客户的公司名称（companyname）。然而， customerid 和 companyname 也是互相依靠的。为满足第三范式，可以改写如下：



符合3NF后的数据模型通俗地讲，2NF和3NF通常以这句话概括：“每个非键属性依赖于键，依赖于整个键，并且除了键别无他物”。

3. 反范式化

3.1 概述

规范化 vs 性能

1. 为满足某种商业目标，数据库性能比规范化数据库更重要
2. 在数据规范化的同时，要综合考虑数据库的性能
3. 通过在给定的表中添加额外的字段，以大量减少需要从中搜索信息所需的时间
4. 通过在给定的表中插入计算列，以方便查询

3.2 应用举例

举例1：

员工的信息存储在 `employees` 表 中，部门信息存储在 `departments` 表 中。通过 `employees` 表中的 `department_id` 字段与 `departments` 表建立关联关系。如果要查询一个员工所在部门的名称：

```
select employee_id, department_name
from employees e join departments d
on e.department_id = d.department_id;
```

如果经常需要进行这个操作，连接查询就会浪费很多时间。可以在 `employees` 表中增加一个冗余字段 `department_name`，这样就不用每次都进行连接操作了。

举例2：

反范式化的 `goods` 商品信息表 设计如下：

字段名称	字段类型	是否是主键	说明
id	INT	是	商品id（主键）
category_id	VARCHAR(30)	否	商品类别id
category_name	VARCHAR(30)	否	商品类别名称
goods_name	VARCHAR(30)	否	商品名称
price	DECIMAL(10,2)	否	商品价格

举例3： 我们有 2 个表，分别是 `商品流水表 (atguigu.trans)` 和 `商品信息表 (atguigu.goodsinfo)`。商品流水表里有 400 万条流水记录，商品信息表里有 2000 条商品记录。

商品流水表：

transid (流水号唯一编号)	itemno (商品编号)	quantity (数量)	price (价格)	balance (金额)	transdate (交易日期)

商品信息表：

itemno (商品编号)	barcode (条码)	goodsname (名称)	specification (规格)	salesprice (售价)

新的商品流水表如下所示：

transid (流水号唯一编号)	itemno (商品编号)	goodsname (商品名称)	quantity (数量)	price (价格)	balance (金额)	date (交易日期)

举例4：

课程评论表 `class_comment`，对应的字段名称及含义如下：

字段	comment_id	class_id	comment_text	comment_time	stu_id
含义	课程评论ID	课程ID	评论内容	评论时间	学生ID

学生表 `student`，对应的字段名称及含义如下：

字段	stu_id	stu_name	create_time
含义	学生ID	学生昵称	注册时间

在实际应用中，我们在显示课程评论的时候，通常会显示这个学生的昵称，而不是学生 ID，因此当我们想要查询某个课程的前 1000 条评论时，需要关联 `class_comment` 和 `student`这两张表来进行查询。

实验数据：模拟两张百万量级的数据表

为了更好地进行 SQL 优化实验，我们需要给学生表和课程评论表随机模拟出百万量级的数据。我们可以通过存储过程来实现模拟数据。

反范式优化实验对比

如果我们想要查询课程 ID 为 10001 的前 1000 条评论，需要写成下面这样：

```

SELECT p.comment_text, p.comment_time, stu.stu_name
FROM class_comment AS p LEFT JOIN student AS stu
ON p.stu_id = stu.stu_id
WHERE p.class_id = 10001
ORDER BY p.comment_id DESC
LIMIT 1000;

```

运行结果（1000 条数据行）：

comment_text	comment_time	stu_name
462eed7ac6e791292a79	2021-10-14 04:53:25	stu_546655
56910cefd01f6d80f0c7	2021-10-14 04:52:35	stu_50353
.....
52f6a51769daf701bc68	2021-10-14 20:35:28	stu_698675

运行时长为 **0.395** 秒，对于网站的响应来说，这已经很慢了，用户体验会非常差。

如果我们想要提升查询的效率，可以允许适当的数据冗余，也就是在商品评论表中增加用户昵称字段，在 class_comment 数据表的基础上增加 stu_name 字段，就得到了 class_comment2 数据表。

这样一来，只需单表查询就可以得到数据集结果：

```
SELECT comment_text, comment_time, stu_name
FROM class_comment2
WHERE class_id = 10001
ORDER BY class_id DESC LIMIT 1000;
```

运行结果（1000 条数据）：

comment_text	comment_time	stu_name
462eed7ac6e791292a79	2021-10-14 04:53:25	stu_546655
56910cefd01f6d80f0c7	2021-10-14 04:52:35	stu_50353
.....
52f6a51769daf701bc68	2021-10-14 20:35:28	stu_698675

优化之后只需要扫描一次聚集索引即可，运行时间为 **0.039** 秒，查询时间是之前的 1/10。你能看到，在数据量大的情况下，查询效率会有显著的提升。

3.3 反范式的新问题

- 存储 **空间变大** 了
- 一个表中字段做了修改，另一个表中冗余的字段也需要做同步修改，否则 **数据不一致**
- 若采用存储过程来支持数据的更新、删除等额外操作，如果更新频繁，会非常 **消耗系统资源**
- 在 **数据量小** 的情况下，反范式不能体现性能的优势，可能还会让数据库的设计更加 **复杂**

3.4 反范式的适用场景

当冗余信息有价值或者能 大幅度提高查询效率 的时候，我们才会采取反范式的优化。

1. 增加冗余字段的建议

2. 历史快照、历史数据的需要

在现实生活中，我们经常需要一些冗余信息，比如订单中的收货人信息，包括姓名、电话和地址等。每次发生的 订单收货信息 都属于 历史快照， 需要进行保存，但用户可以随时修改自己的信息，这时保存这些冗余信息是非常有必要的。

反范式优化也常用在 数据仓库 的设计中，因为数据仓库通常 存储历史数据， 对增删改的实时性要求不强，对历史数据的分析需求强。这时适当允许数据的冗余度，更方便进行数据分析。

4. BCNF(巴斯范式)

1. 案例

我们分析如下表的范式情况：

仓库名	管理员	物品名	数量
北京仓	张三	iphone XR	10
北京仓	张三	iphone 7	20
上海仓	李四	iphone 7p	30
上海仓	李四	iphone 8	40

在这个表中，一个仓库只有一个管理员，同时一个管理员也只管理一个仓库。我们先来梳理下这些属性之间的依赖关系。

仓库名决定了管理员，管理员也决定了仓库名，同时（仓库名， 物品名）的属性集合可以决定数量这个属性。这样，我们就可以找到数据表的候选键。

候选键：是（管理员， 物品名）和（仓库名， 物品名）， 然后我们从候选键中选择一个作为 **主键**， 比如（仓库名， 物品名）。

主属性：包含在任一候选键中的属性，也就是仓库名，管理员和物品名。

非主属性：数量这个属性。

2. 是否符合三范式

如何判断一张表的范式呢？我们需要根据范式的等级，从低到高来进行判断。

首先，数据表每个属性都是原子性的，符合 1NF 的要求；

其次，数据表中非主属性“都与候选键全部依赖，（仓库名， 物品名）决定数量，（管理员， 物品名）决定数量。因此，数据表符合 2NF 的要求；

最后，数据表中的非主属性，不传递依赖于候选键。因此符合 3NF 的要求。

3. 存在的问题

既然数据表已经符合了 3NF 的要求，是不是就不存在问题了呢？我们来看下面的情况：

- 增加一个仓库，但是还没有存放任何物品。根据数据表实体完整性的要求，主键不能有空值，因此会出现 **插入异常**；
- 如果仓库更换了管理员，我们就可能会 **修改数据表中的多条记录**；
- 如果仓库里的商品都卖空了，那么此时仓库名称和相应的管理员名称也会随之被删除。

你能看到，即便数据表符合 3NF 的要求，同样可能存在插入、更新和删除数据的异常情况。

4. 问题解决

首先我们需要确认造成异常的原因：主属性仓库名对于候选键（管理员，物品名）是部分依赖的关系，这样就有可能导致上面的异常情况。因此引入BCNF，**它在 3NF 的基础上消除了主属性对候选键的部分依赖或者传递依赖关系。**

- 如果在关系R中，U为主键，A属性是主键的一个属性，若存在 $A \rightarrow Y$ ，Y为主属性，则该关系不属于BCNF。

根据 BCNF 的要求，我们需要把仓库管理关系 warehouse_keeper 表拆分成下面这样：

仓库表：（仓库名， 管理员）

库存表：（仓库名， 物品名， 数量）

这样就不存在主属性对于候选键的部分依赖或传递依赖，上面数据表的设计就符合 BCNF。

再举例：

有一个 **学生导师表**，其中包含字段：学生ID，专业，导师，专业GPA，这其中学生ID和专业是联合主键。

StudentId	Major	Advisor	MajGPA
1	人工智能	Edward	4.0
2	大数据	William	3.8
1	大数据	William	3.7
3	大数据	Joseph	4.0

这个表的设计满足三范式，但是这里存在另一个依赖关系，“专业”依赖于“导师”，也就是说每个导师只做一个专业方面的导师，只要知道了是哪个导师，我们自然就知道是哪个专业的了。

所以这个表的部分主键Major依赖于非主键属性Advisor，那么我们可以进行以下的调整，拆分成2个表：

学生导师表：

StudentId	Advisor	MajGPA
1	Edward	4.0
2	William	3.8
1	William	3.7
3	Joseph	4.0

导师表：

Advisor	Major
Edward	人工智能
William	大数据
Joseph	大数据

5. 第四范式

举例1：职工表(职工编号，职工孩子姓名，职工选修课程)。

在这个表中，同一个职工可能会有多个职工孩子姓名。同样，同一个职工也可能会有多个职工选修课程，即这里存在着多值事实，不符合第四范式。

如果要符合第四范式，只需要将上表分为两个表，使它们只有一个多值事实，例如：[职工表一](#)(职工编号，职工孩子姓名)，[职工表二](#)(职工编号，职工选修课程)，两个表都只有一个多值事实，所以符合第四范式。

举例2：

比如我们建立课程、教师、教材的模型。我们规定，每门课程有对应的一组教师，每门课程也有对应的一组教材，一门课程使用的教材和教师没有关系。我们建立的关系表如下：

课程ID，教师ID，教材ID；这三列作为联合主键。

为了表述方便，我们用Name代替ID，这样更容易看懂：

Course	Teacher	Book
英语	Bill	人教版英语
英语	Bill	美版英语
英语	Jay	美版英语
高数	William	人教版高数
高数	Dave	美版高数

这个表除了主键，就没有其他字段了，所以肯定满足BC范式，但是却存在[多值依赖](#)导致的异常。

假如我们下学期想采用一本新的英版高数教材，但是还没确定具体哪个老师来教，那么我们就无法在這個表中维护Course高数和Book英版高数教材的关系。

解决办法是我们把这个多值依赖的表拆解成2个表，分别建立关系。这是我们拆分后的表：

Course	Teacher
英语	Bill
英语	Jay
高数	William
高数	Dave

以及

Course	Book
英语	人教版英语
英语	美版英语
高数	人教版高数
高数	美版高数

6. 第五范式、域键范式

除了第四范式外，我们还有更高级的第五范式（又称完美范式）和域键范式（DKNF）。

在满足第四范式（4NF）的基础上，消除不是由候选键所蕴含的连接依赖。**如果关系模式R中的每一个连接依赖均由R的候选键所蕴含**，则称此关系模式符合第五范式。

函数依赖是多值依赖的一种特殊的情况，而多值依赖实际上是连接依赖的一种特殊情况。但连接依赖不像函数依赖和多值依赖可以由语义直接导出，而是在关系连接运算时才反映出来。存在连接依赖的关系模式仍可能遇到数据冗余及插入、修改、删除异常等问题。

第五范式处理的是无损连接问题，这个范式基本没有实际意义，因为无损连接很少出现，而且难以察觉。而域键范式试图定义一个终极范式，该范式考虑所有的依赖和约束类型，但是实用价值也是最小的，只存在理论研究中。

7. 实战案例

见视频讲解（https://www.bilibili.com/video/BV1iq4y1u7vj?from=search&seid=4297501441472622157&spm_id_from=333.337.0.0）

8. ER模型

ER模型中有三个要素，分别是实体、属性和关系。

实体，可以看做是数据对象，往往对应于现实生活中的真实存在的个体。在ER模型中，用矩形来表示。实体分为两类，分别是**强实体**和**弱实体**。强实体是指不依赖于其他实体的实体；弱实体是指对另一个实体有很强的依赖关系的实体。

属性，则是指实体的特性。比如超市的地址、联系电话、员工数等。在ER模型中用椭圆形来表示。

关系，则是指实体之间的联系。比如超市把商品卖给顾客，就是一种超市与顾客之间的联系。在ER模型中用菱形来表示。

注意：实体和属性不容易区分。这里提供一个原则：我们要从系统整体的角度出发去看，**可以独立存在的是实体，不可再分的是属性**。也就是说，属性不能包含其他属性。

8.2 关系的类型

在 ER 模型的 3 个要素中，关系又可以分为 3 种类型，分别是一对一、一对多、多对多。

一对一：指实体之间的关系是一一对应的，比如个人与身份证信息之间的关系就是一对一的关系。一个人只能有一个身份证信息，一个身份证信息也只属于一个人。

一对多：指一边的实体通过关系，可以对应多个另外一边的实体。相反，另外一边的实体通过这个关系，则只能对应唯一的一边的实体。比如说，我们新建一个班级表，而每个班级都有多个学生，每个学生则对应一个班级，班级对学生就是一对多的关系。

多对多：指关系两边的实体都可以通过关系对应多个对方的实体。比如在进货模块中，供货商与超市之间的关系就是多对多的关系，一个供货商可以给多个超市供货，一个超市也可以从多个供货商那里采购商品。再比如一个选课表，有许多科目，每个科目有很多学生选，而每个学生又可以选择多个科目，这就是多对多的关系。

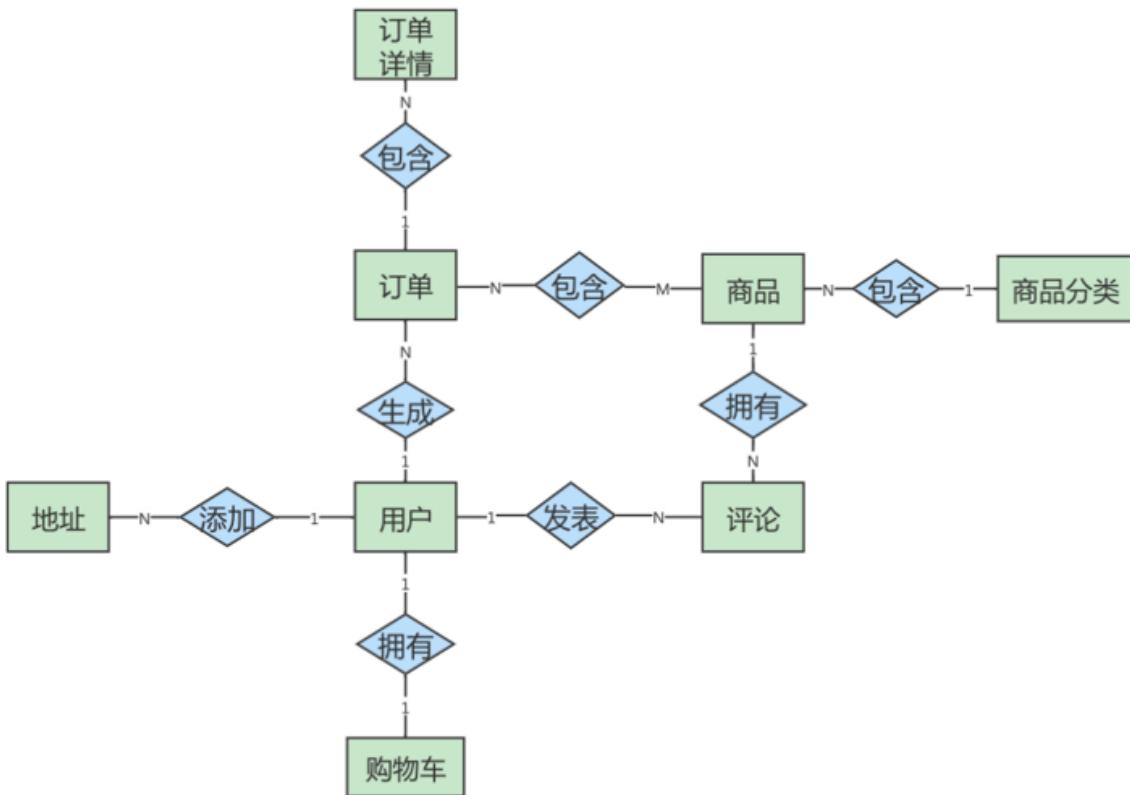
8.3 建模分析

ER 模型看起来比较麻烦，但是对我们把控项目整体非常重要。如果你只是开发一个小应用，或许简单设计几个表够用了，一旦要设计有一定规模的应用，在项目的初始阶段，建立完整的 ER 模型就非常关键了。开发应用项目的实质，其实就是 **建模**。

我们设计的案例是 **电商业务**，由于电商业务太过庞大且复杂，所以我们做了业务简化，比如针对 SKU (StockKeepingUnit, 库存量单位) 和 SPU (Standard Product Unit, 标准化产品单元) 的含义上，我们直接使用了 SKU，并没有提及 SPU 的概念。本次电商业务设计总共有 8 个实体，如下所示。

- 地址实体
- 用户实体
- 购物车实体
- 评论实体
- 商品实体
- 商品分类实体
- 订单实体
- 订单详情实体

其中，**用户** 和 **商品分类** 是强实体，因为它们不需要依赖其他任何实体。而其他属于弱实体，因为它们虽然都可以独立存在，但是它们都依赖用户这个实体，因此都是弱实体。知道了这些要素，我们就可以给电商业务创建 ER 模型了，如图：



在这个图中，地址和用户之间的添加关系，是一对多的关系，而商品和商品详情示一对1的关系，商品和订单是多对多的关系。这个 ER 模型，包括了 8 个实体之间的 8 种关系。

- (1) 用户可以在电商平台添加多个地址；
- (2) 用户只能拥有一个购物车；
- (3) 用户可以生成多个订单；
- (4) 用户可以发表多条评论；
- (5) 一件商品可以有多条评论；
- (6) 每一个商品分类包含多种商品；
- (7) 一个订单可以包含多个商品，一个商品可以在多个订单里。
- (8) 订单中又包含多个订单详情，因为一个订单中可能包含不同种类的商品

8.4 ER 模型的细化

有了这个 ER 模型，我们就可以从整体上 理解 电商的业务了。刚刚的 ER 模型展示了电商业务的框架，但是只包括了订单，地址，用户，购物车，评论，商品，商品分类和订单详情这八个实体，以及它们之间的关系，还不能对应到具体的表，以及表与表之间的关联。我们需要把 属性加上，用 椭圆 来表示，这样我们得到的 ER 模型就更加完整了。

因此，我们需要进一步去设计一下这个 ER 模型的各个局部，也就是细化下电商的具体业务流程，然后把它们综合到一起，形成一个完整的 ER 模型。这样可以帮助我们理清数据库的设计思路。

接下来，我们再分析一下各个实体都有哪些属性，如下所示。

- (1) 地址实体 包括用户编号、省、市、地区、收件人、联系电话、是否是默认地址。
- (2) 用户实体 包括用户编号、用户名称、昵称、用户密码、手机号、邮箱、头像、用户级别。
- (3) 购物车实体 包括购物车编号、用户编号、商品编号、商品数量、图片文件url。

(4) **订单实体** 包括订单编号、收货人、收件人电话、总金额、用户编号、付款方式、送货地址、下单时间。

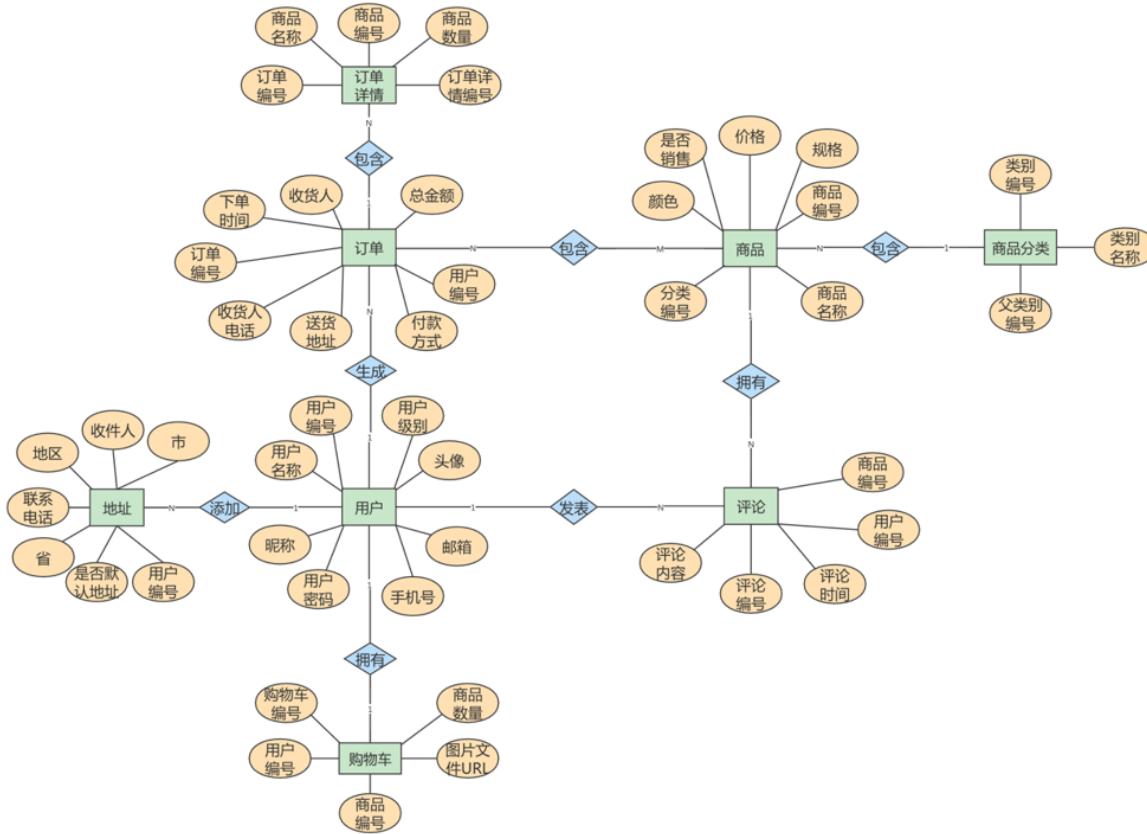
(5) **订单详情实体** 包括订单详情编号、订单编号、商品名称、商品编号、商品数量。

(6) **商品实体** 包括商品编号、价格、商品名称、分类编号、是否销售，规格、颜色。

(7) **评论实体** 包括评论id、评论内容、评论时间、用户编号、商品编号

(8) **商品分类实体** 包括类别编号、类别名称、父类别编号

这样细分之后，我们就可以重新设计电商业务了，ER 模型如图：



8.5 ER 模型转换成数据表

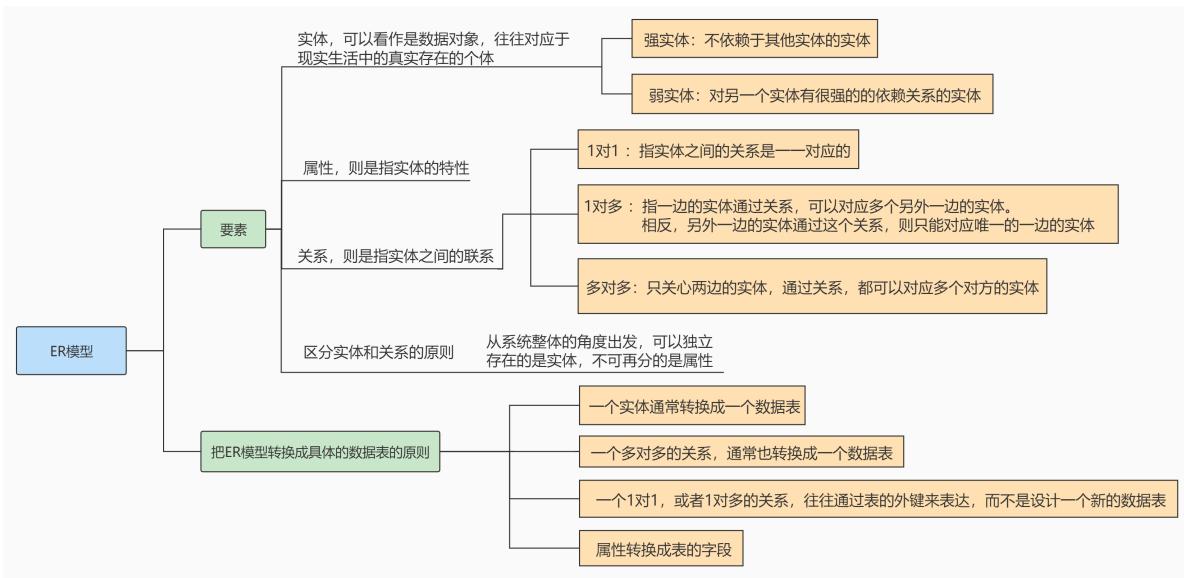
通过绘制 ER 模型，我们已经理清了业务逻辑，现在，我们就要进行非常重要的一步了：把绘制好的 ER 模型，转换成具体的数据表，下面介绍下转换的原则：

- (1) 一个 **实体** 通常转换成一个 **数据表**；
- (2) 一个 **多对多的关系**，通常也转换成一个 **数据表**；
- (3) 一个 **1 对 1**，或者 **1 对多** 的关系，往往通过表的 **外键** 来表达，而不是设计一个新的数据表；
- (4) **属性** 转换成表的 **字段**。

下面结合前面的ER模型，具体讲解一下怎么运用这些转换的原则，把 ER 模型转换成具体的数据表，从而把抽象出来的数据模型，落实到具体的数据库设计当中。

详情见视频讲解 (https://www.bilibili.com/video/BV1iq4y1u7vj?from=search&seid=4297501441472622157&spm_id_from=333.337.0.0)

其实，任何一个基于数据库的应用项目，都可以通过这种 **先建立 ER 模型，再 转换成数据表** 的方式，完成数据库的设计工作。创建 ER 模型不是目的，目的是把业务逻辑梳理清楚，设计出优秀的数据库。我建议你不是为了建模而建模，要利用创建 ER 模型的过程来整理思路，这样创建 ER 模型才有意义。



9. 数据表的设计原则

综合以上内容, 总结出数据表设计的一般原则: "三少一多"

1. **数据表的个数越少越好**
2. **数据表中的字段个数越少越好**
3. **数据表中联合主键的字段个数越少越好**
4. **使用主键和外键越多越好**

注意: 这个原则并不是绝对的, 有时候我们需要牺牲数据的冗余度来换取数据处理的效率。

10. 数据库对象编写建议

10.1 关于库

1. 【强制】库的名称必须控制在32个字符以内, 只能使用英文字母、数字和下划线, 建议以英文字母开头。
2. 【强制】库名中英文一律小写, 不同单词采用下划线分割。须见名知意。
3. 【强制】库的名称格式: 业务系统名称_子系统名。
4. 【强制】库名禁止使用关键字(如type,order等)。
5. 【强制】创建数据库时必须显式指定字符集, 并且字符集只能是utf8或者utf8mb4。

创建数据库SQL举例: CREATE DATABASE crm_fund DEFAULT CHARACTER SET 'utf8' ;

6. 【建议】对于程序连接数据库账号, 遵循权限最小原则

使用数据库账号只能在一个DB下使用, 不准跨库。程序使用的账号原则上不准有drop权限。

7. 【建议】临时库以tmp_为前缀, 并以日期为后缀;

备份库以bak_为前缀, 并以日期为后缀。

10.2 关于表、列

1. 【强制】表和列的名称必须控制在32个字符以内，表名只能使用英文字母、数字和下划线，建议以 **英文字母开头**。
2. 【强制】 **表名、列名一律小写**，不同单词采用下划线分割。须见名知意。
3. 【强制】表名要求有模块名强相关，同一模块的表名尽量使用 **统一前缀**。比如：crm_fund_item
4. 【强制】创建表时必须 **显式指定字符集** 为utf8或utf8mb4。
5. 【强制】表名、列名禁止使用关键字（如type,order等）。
6. 【强制】创建表时必须 **显式指定表存储引擎** 类型。如无特殊需求，一律为InnoDB。
7. 【强制】建表必须有comment。
8. 【强制】字段命名应尽可能使用表达实际含义的英文单词或 **缩写**。如：公司 ID，不要使用 corporation_id，而用corp_id 即可。
9. 【强制】布尔值类型的字段命名为 **is_描述**。如member表上表示是否为enabled的会员的字段命名为 is_enabled。
10. 【强制】禁止在数据库中存储图片、文件等大的二进制数据
通常文件很大，短时间内造成数据量快速增长，数据库进行数据库读取时，通常会进行大量的随机IO操作，文件很大时，IO操作很耗时。通常存储于文件服务器，数据库只存储文件地址信息。
11. 【建议】建表时关于主键：**表必须有主键** (1)强制要求主键为id，类型为int或bigint，且为 auto_increment 建议使用unsigned无符号型。 (2)标识表里每一行主体的字段不要设为主键，建议设为其他字段如user_id, order_id等，并建立unique key索引。因为如果设为主键且主键值为随机插入，则会导致innodb内部页分裂和大量随机I/O，性能下降。
12. 【建议】核心表（如用户表）必须有行数据的 **创建时间字段**（create_time）和 **最后更新时间字段**（update_time），便于查问题。
13. 【建议】表中所有字段尽量都是 **NOT NULL** 属性，业务可以根据需要定义 **DEFAULT值**。因为使用NULL值会存在每一行都会占用额外存储空间、数据迁移容易出错、聚合函数计算结果偏差等问题。
14. 【建议】所有存储相同数据的 **列名和列类型必须一致**（一般作为关联列，如果查询时关联列类型不一致会自动进行数据类型隐式转换，会造成列上的索引失效，导致查询效率降低）。
15. 【建议】中间表（或临时表）用于保留中间结果集，名称以 **tmp_** 开头。
备份表用于备份或抓取源表快照，名称以 **bak_** 开头。中间表和备份表定期清理。
16. 【示范】一个较为规范的建表语句：

```
CREATE TABLE user_info (
    `id` int unsigned NOT NULL AUTO_INCREMENT COMMENT '自增主键',
    `user_id` bigint(11) NOT NULL COMMENT '用户id',
    `username` varchar(45) NOT NULL COMMENT '真实姓名',
    `email` varchar(30) NOT NULL COMMENT '用户邮箱',
    `nickname` varchar(45) NOT NULL COMMENT '昵称',
    `birthday` date NOT NULL COMMENT '生日',
    `sex` tinyint(4) DEFAULT '0' COMMENT '性别',
    `short_introduce` varchar(150) DEFAULT NULL COMMENT '一句话介绍自己，最多50个汉字',
    `user_resume` varchar(300) NOT NULL COMMENT '用户提交的简历存放地址',
    `user_register_ip` int NOT NULL COMMENT '用户注册时的源ip',
    `create_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '创建时间',
    `update_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
    CURRENT_TIMESTAMP COMMENT '修改时间',
    `user_review_status` tinyint NOT NULL COMMENT '用户资料审核状态，1为通过，2为审核中，3为未
通过，4为还未提交审核'
```

```
PRIMARY KEY (`id`),  
UNIQUE KEY `uniq_user_id`(`user_id`),  
KEY `idx_username`(`username`),  
KEY `idx_create_time_status`(`create_time`, `user_review_status`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='网站用户基本信息'
```

17. 【建议】创建表时，可以使用可视化工具。这样可以确保表、字段相关的约定都能设置上。

实际上，我们通常很少自己写 DDL 语句，可以使用一些可视化工具来创建和操作数据库和数据表。

可视化工具除了方便，还能直接帮我们将数据库的结构定义转化成 SQL 语言，方便数据库和数据表结构的导出和导入。

10.3 关于索引

1. 【强制】InnoDB表必须主键为id int/bigint auto_increment，且主键值 禁止被更新。
2. 【强制】InnoDB和MyISAM存储引擎表，索引类型必须为 BTREE。
3. 【建议】主键的名称以 pk_ 开头，唯一键以 uni_ 或 uk_ 开头，普通索引以 idx_ 开头，一律使用小写格式，以字段的名称或缩写作为后缀。
4. 【建议】多单词组成的columnname，取前几个单词首字母，加末单词组成column_name。如：sample 表 member_id 上的索引：idx_sample_mid。
5. 【建议】单个表上的索引个数 不能超过6个。
6. 【建议】在建立索引时，多考虑建立 联合索引，并把区分度最高的字段放在最前面。
7. 【建议】在多表 JOIN 的SQL里，保证被驱动表的连接列上有索引，这样JOIN 执行效率最高。
8. 【建议】建表或加索引时，保证表里互相不存在 冗余索引。比如：如果表里已经存在key(a,b)，则key(a)为冗余索引，需要删除。

10.4 SQL编写

1. 【强制】程序端SELECT语句必须指定具体字段名称，禁止写成 *。
2. 【建议】程序端insert语句指定具体字段名称，不要写成INSERT INTO t1 VALUES(...).
3. 【建议】除静态表或小表（100行以内），DML语句必须有WHERE条件，且使用索引查找。
4. 【建议】INSERT INTO...VALUES(XX),(XX),(XX)... 这里XX的值不要超过5000个。值过多虽然上线很快，但会引起主从同步延迟。
5. 【建议】SELECT语句不要使用UNION，推荐使用UNION ALL，并且UNION子句个数限制在5个以内。
6. 【建议】线上环境，多表 JOIN 不要超过5个表。
7. 【建议】减少使用ORDER BY，和业务沟通能不排序就不排序，或将排序放到程序端去做。ORDER BY、GROUP BY、DISTINCT 这些语句较为耗费CPU，数据库的CPU资源是极其宝贵的。
8. 【建议】包含了ORDER BY、GROUP BY、DISTINCT 这些查询的语句，WHERE 条件过滤出来的结果集请保持在1000行以内，否则SQL会很慢。
9. 【建议】对单表的多次alter操作必须合并为一次

对于超过100W行的大表进行alter table，必须经过DBA审核，并在业务低高峰期执行，多个alter需整合在一起。因为alter table会产生 表锁，期间阻塞对于该表的所有写入，对于业务可能会产生极大影响。

10. 【建议】批量操作数据时，需要控制事务处理间隔时间，进行必要的sleep。

11. 【建议】事务里包含SQL不超过5个。

因为过长的事务会导致锁数据较久，MySQL内部缓存、连接消耗过多等问题。

12. 【建议】事务里更新语句尽量基于主键或UNIQUE KEY，如UPDATE... WHERE id=XX;

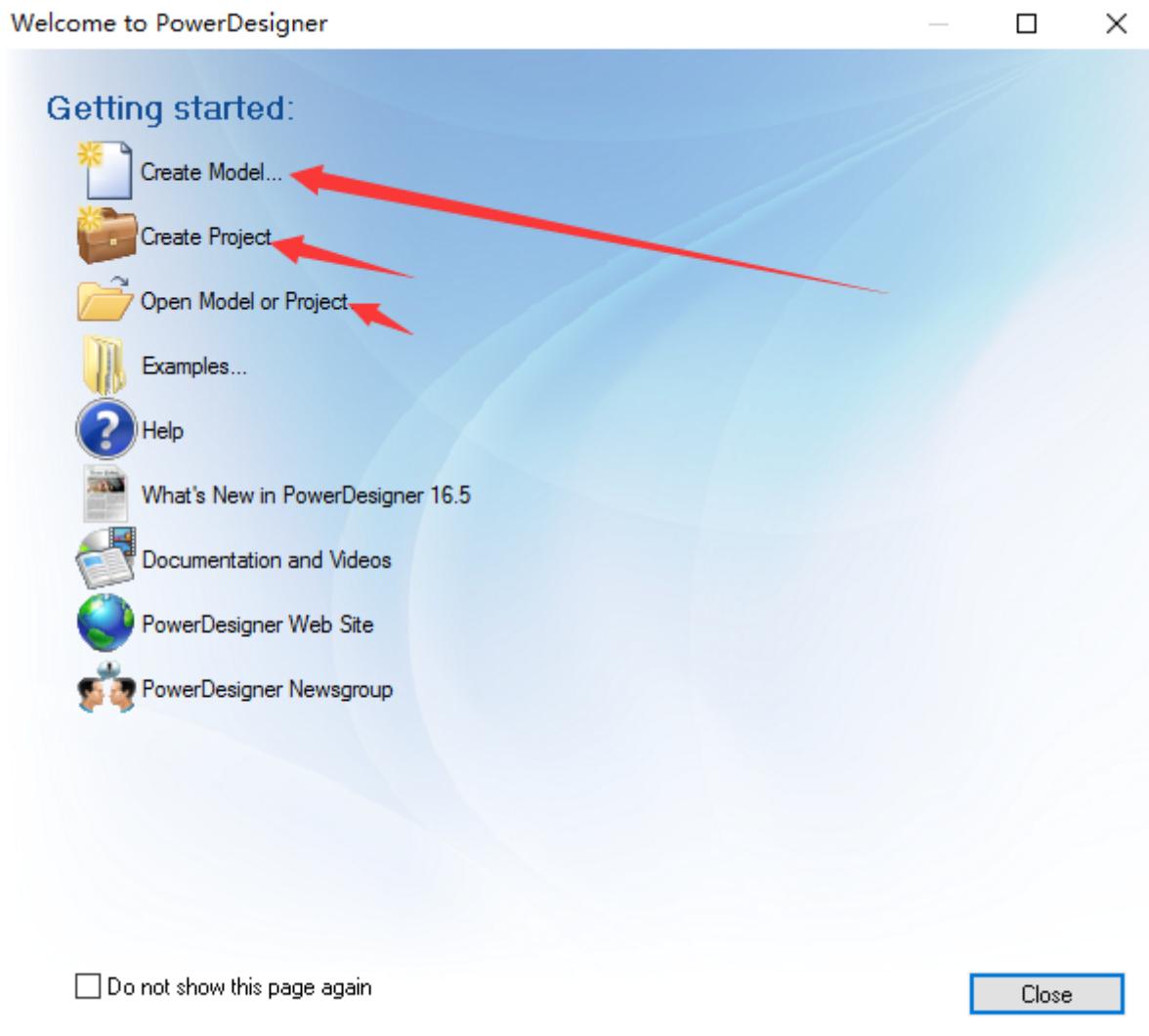
否则会产生间隙锁，内部扩大锁定范围，导致系统性能下降，产生死锁。

11. PowerDesigner的使用

PowerDesigner是一款开发人员常用的数据库建模工具，用户利用该软件可以方便地制作 **数据流程图**、**概念数据模型**、**物理数据模型**，它几乎包括了数据库模型设计的全过程，是Sybase公司为企业建模和设计提供的一套完整的集成化企业级建模解决方案。

11.1 开始界面

当前使用的PowerDesigner版本是16.5的。打开软件即是此页面，可选择Create Model,也可以选择Do Not Show page Again,自行在打开软件后创建也可以！完全看个人的喜好，在此我在后面的学习中不在显示此页面。

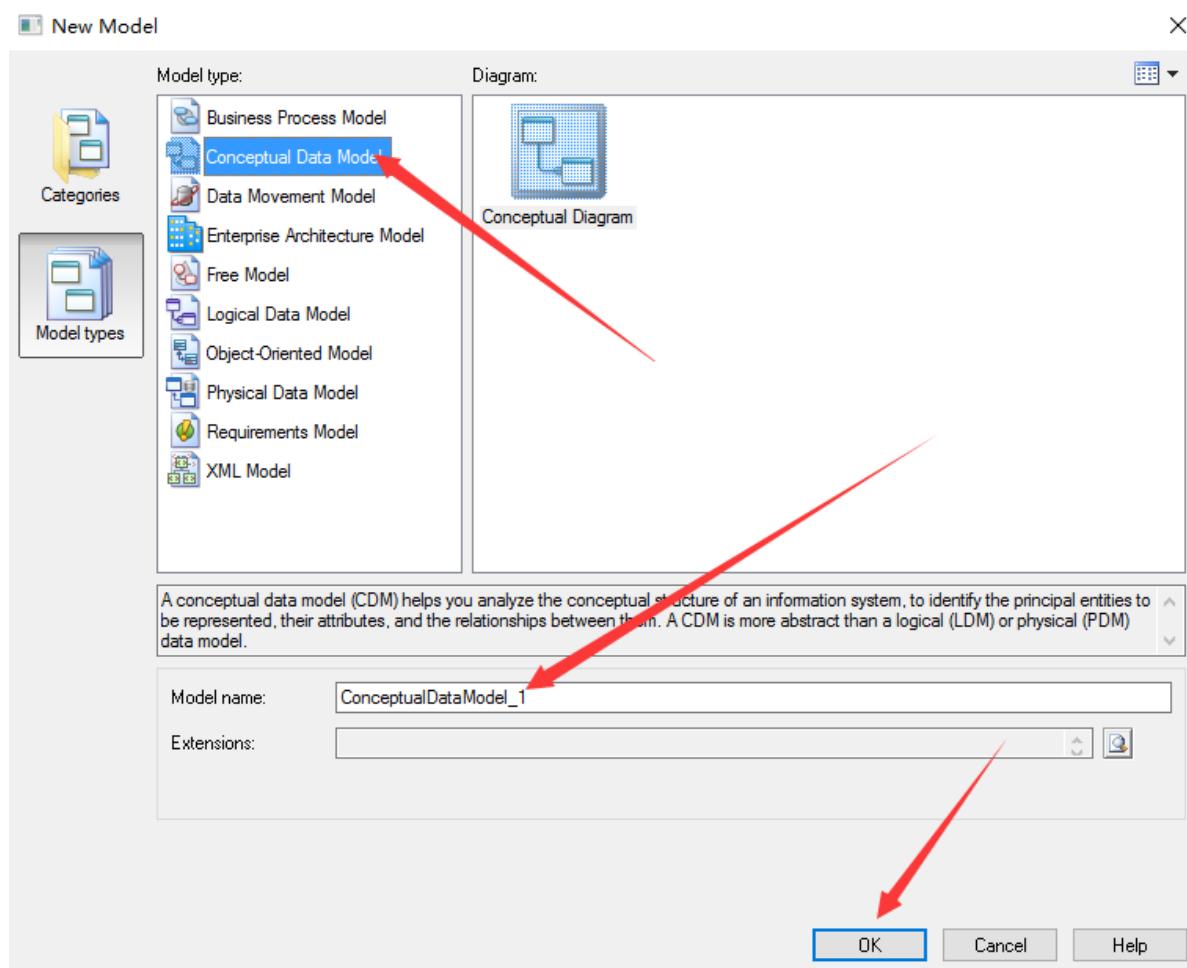


“Create Model”的作用类似于普通的一个文件，该文件可以单独存放也可以归类存放。

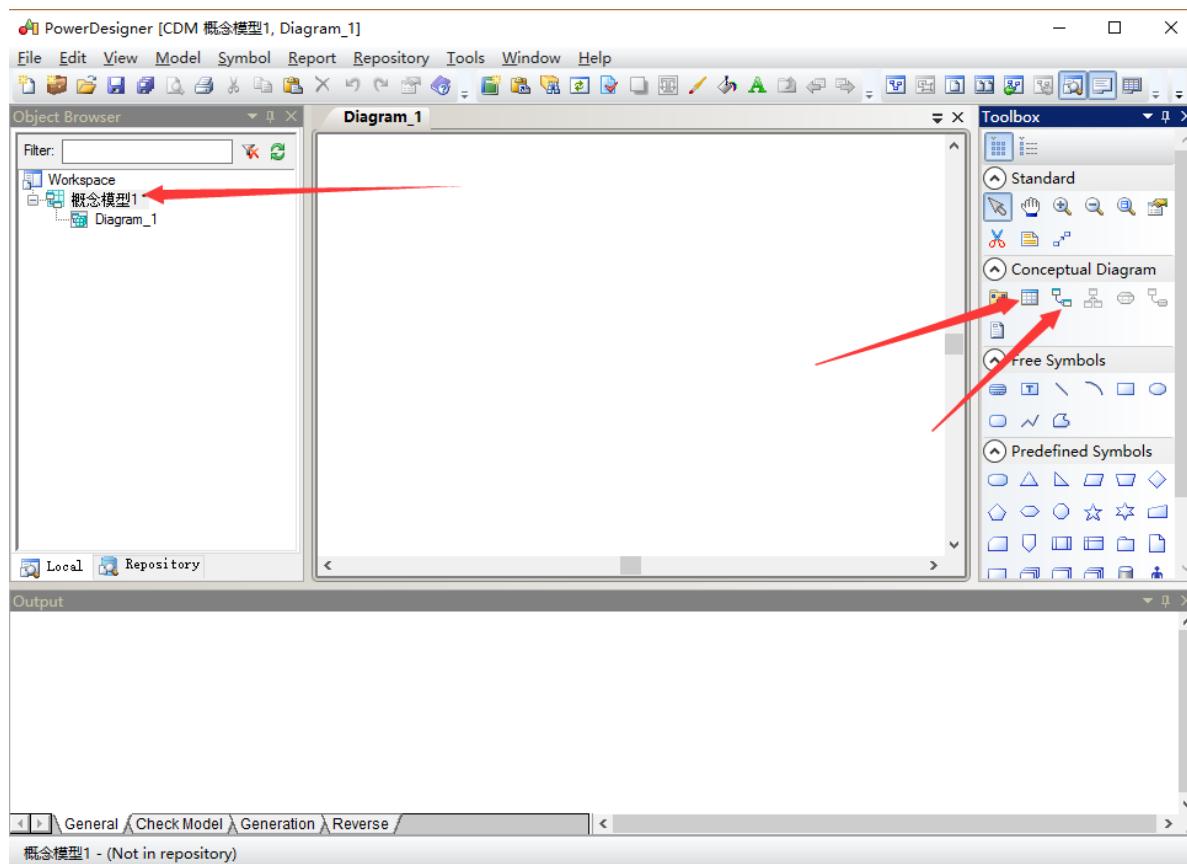
“Create Project”的作用类似于文件夹，负责把有关联关系的文件集中归类存放。

11.2 概念数据模型

常用的模型有4种，分别是 **概念模型(CDM Conceptual Data Model)**，**物理模型(PDM, Physical Data Model)**，**面向对象的模型(OOM Object Oriented Model)** 和 **业务模型(BPM Business Process Model)**，我们先创建概念数据模型。

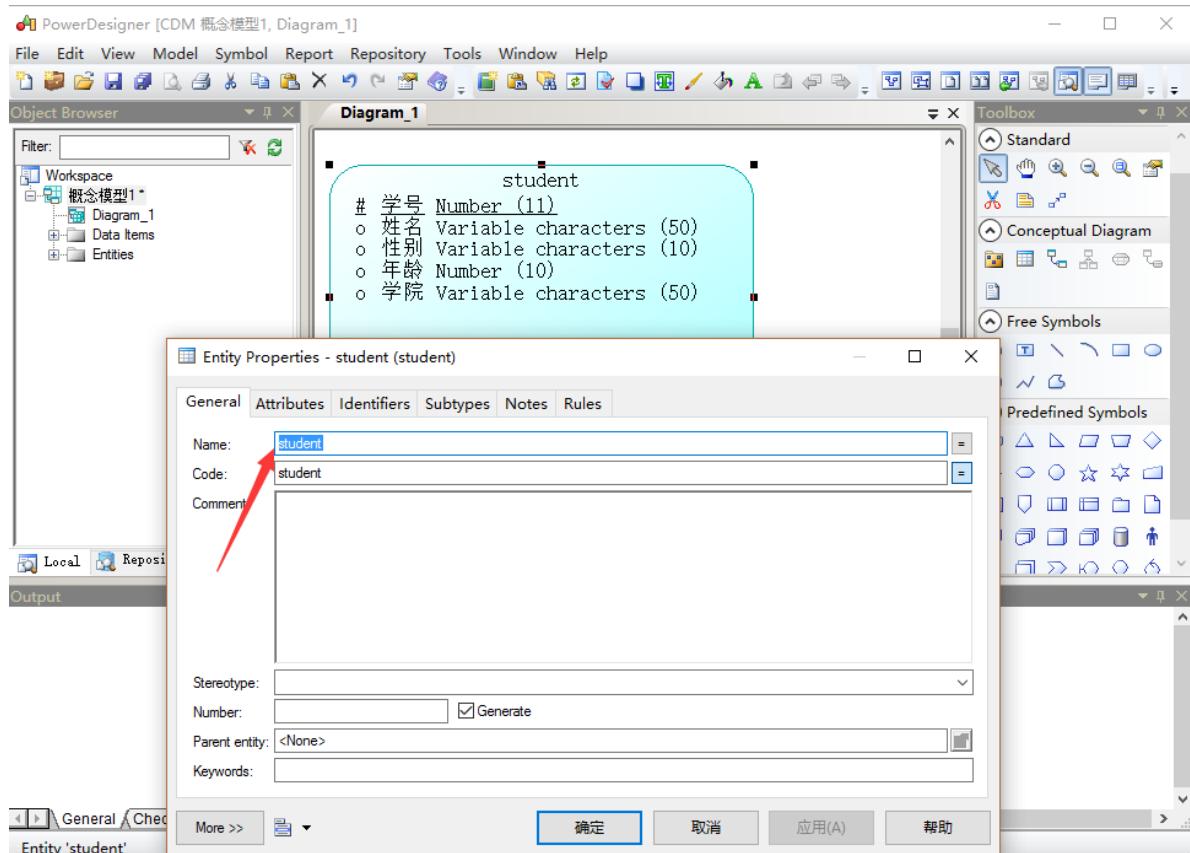


点击上面的ok，即可出现下图左边的概念模型1，可以自定义概念模型的名字，在概念模型中使用最多的就是如图所示的Entity(实体),Relationship(关系)



Entity实体

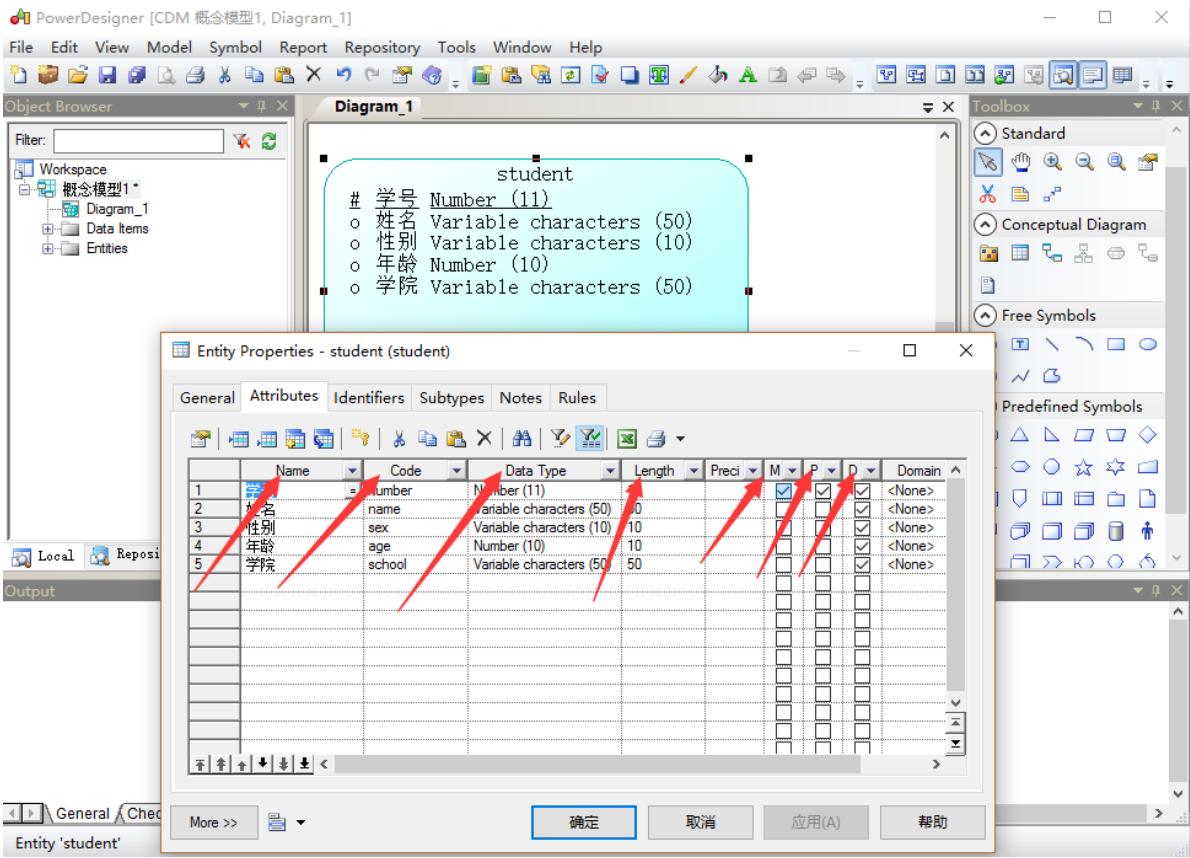
选中右边框中Entity这个功能，即可出现下面这个方框，需要注意的是书写name的时候，code自行补充，name可以是英文的也可以是中文的，但是code必须是英文的。



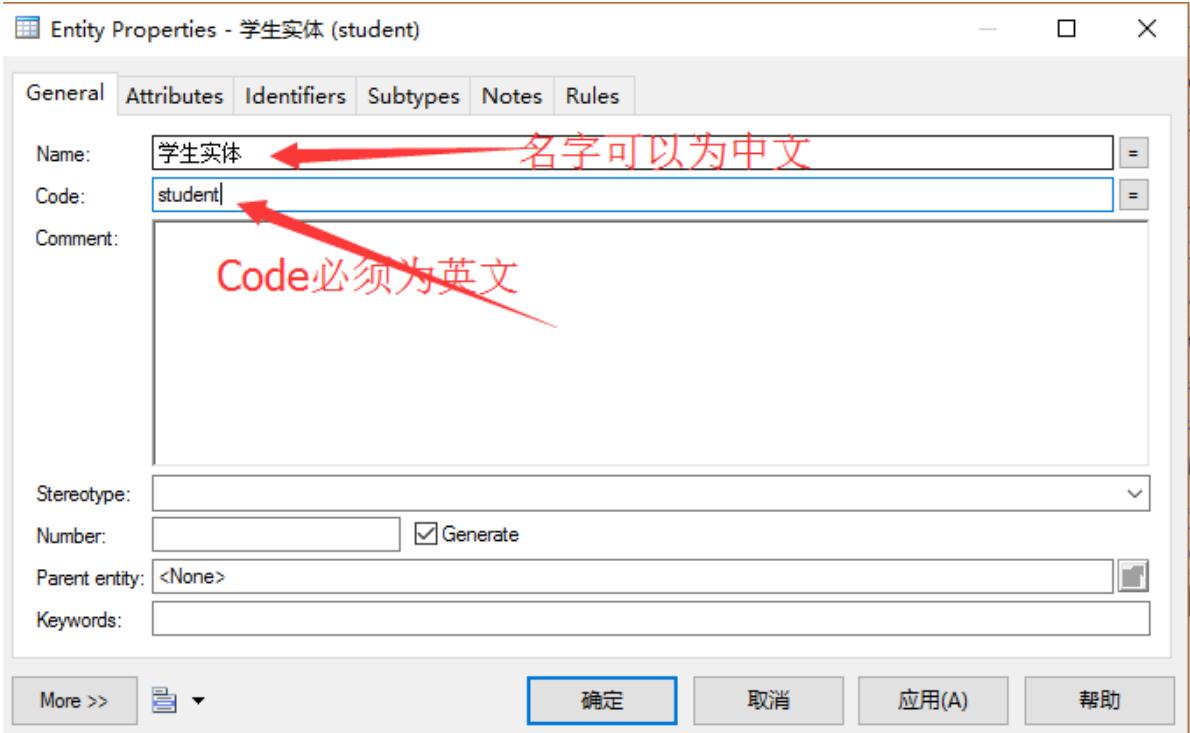
填充实体字段

General中的name和code填好后，就可以点击Attributes（属性）来设置name（名字），code(在数据库中的字段名)，Data Type(数据类型)，length(数据类型的长度)

- Name: 实体名字一般为中文，如论坛用户
- Code: 实体代号，一般用英文，如XXXUser
- Comment: 注释，对此实体详细说明
- Code属性: 代号，一般用英文UID DataType
- Domain域，表示属性取值范围如可以创建10个字符的地址域
- M:Mandatory强制属性，表示该属性必填。不能为空
- P:Primary Identifier是否是主标识符，表示实体唯一标识符
- D:Displayed显示出来，默认全部勾选

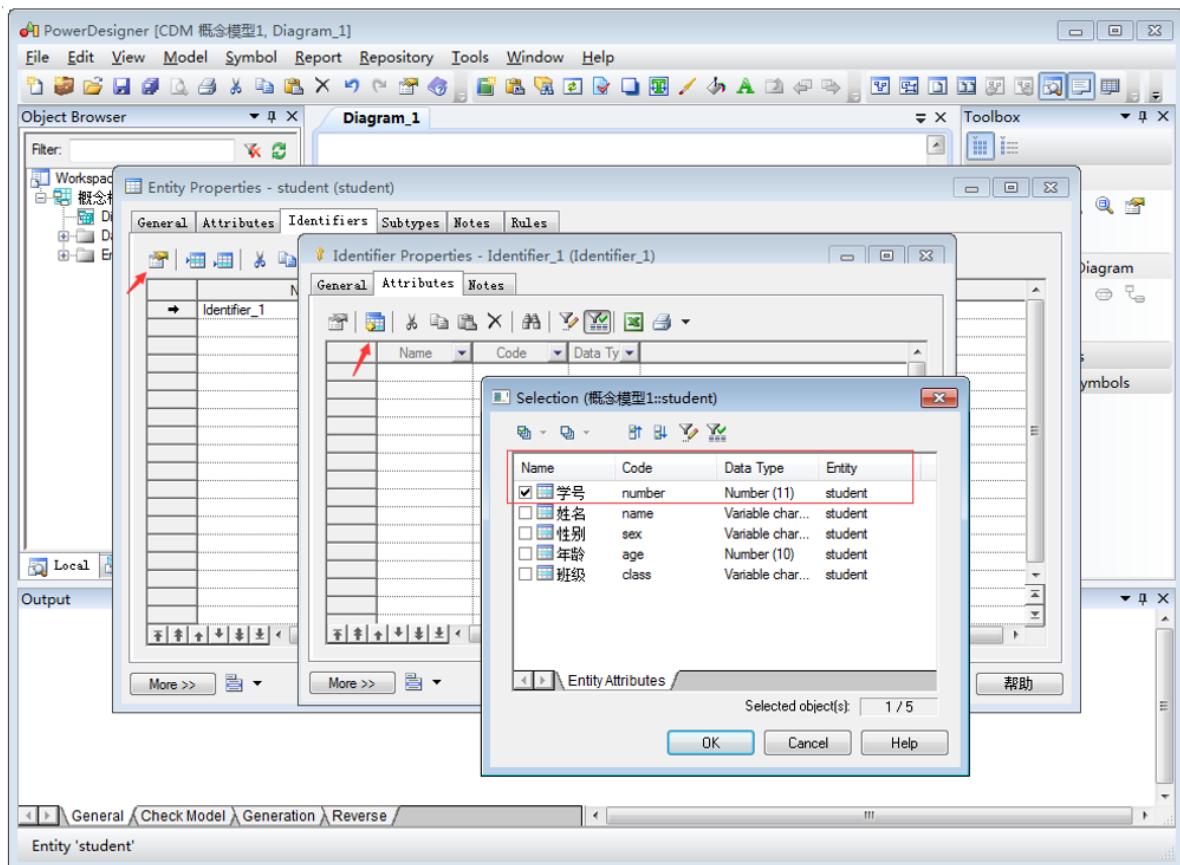


在此上图说明name和code的起名方法



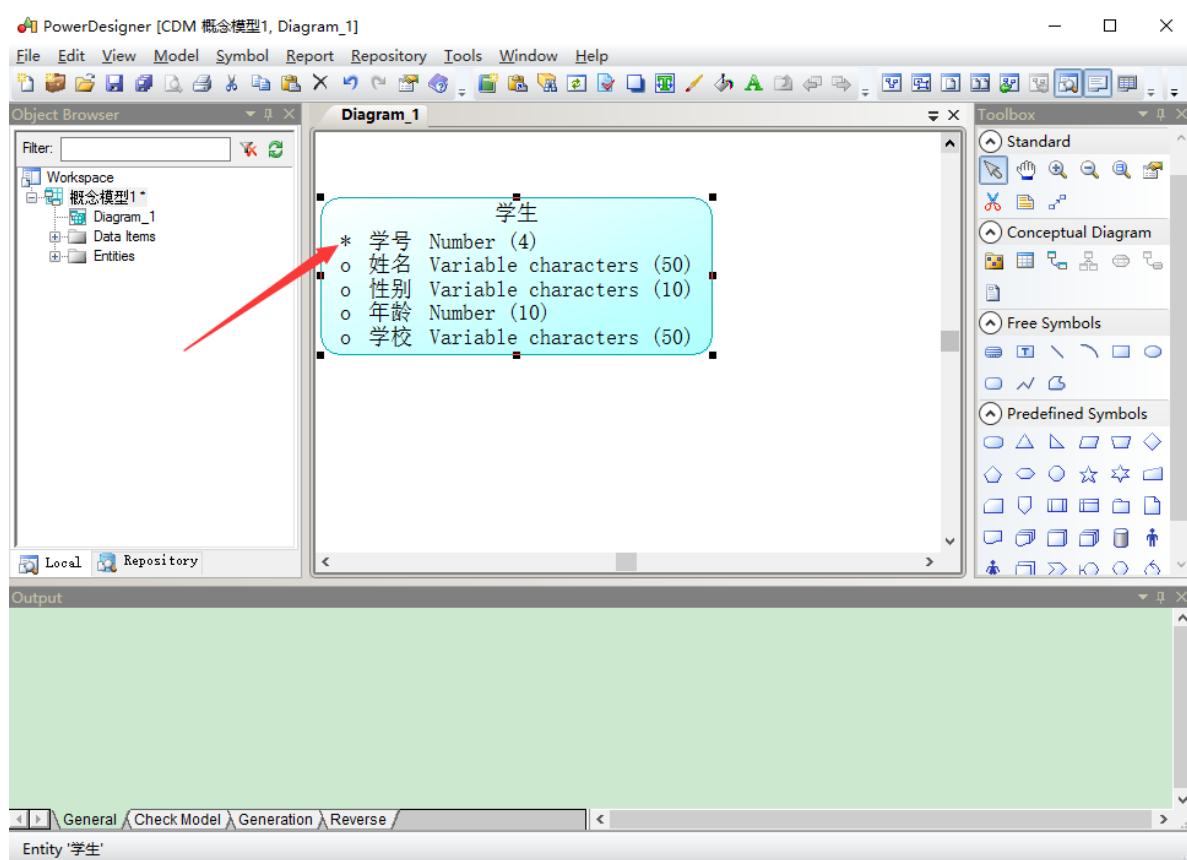
设置主标识符

如果不希望系统自动生成标识符而是手动设置的话，那么切换到Identifiers选项卡，添加一行Identifier，然后单击左上角的“属性”按钮，然后弹出的标识属性设置对话框中单击“添加行”按钮，选择该标识中使用的属性。例如将学号设置为学生实体的标识。



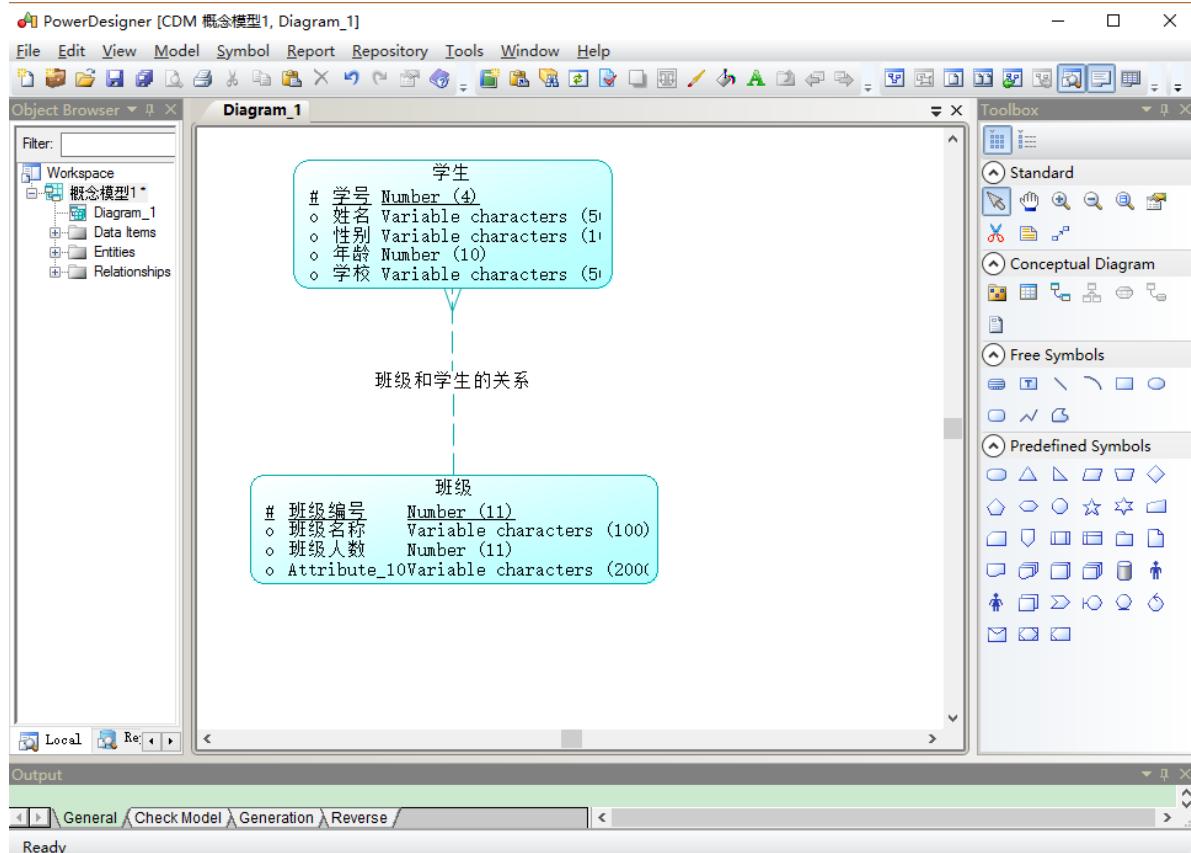
放大模型

创建好概念数据模型如图所示，但是创建好的字体很小，读者可以按着ctrl键同时滑动鼠标的可滑动按钮即可放大缩写字体，同时也可以看到主标识符有一个*号的标志，同时也显示出来了，name,Data type和length这些可见的属性

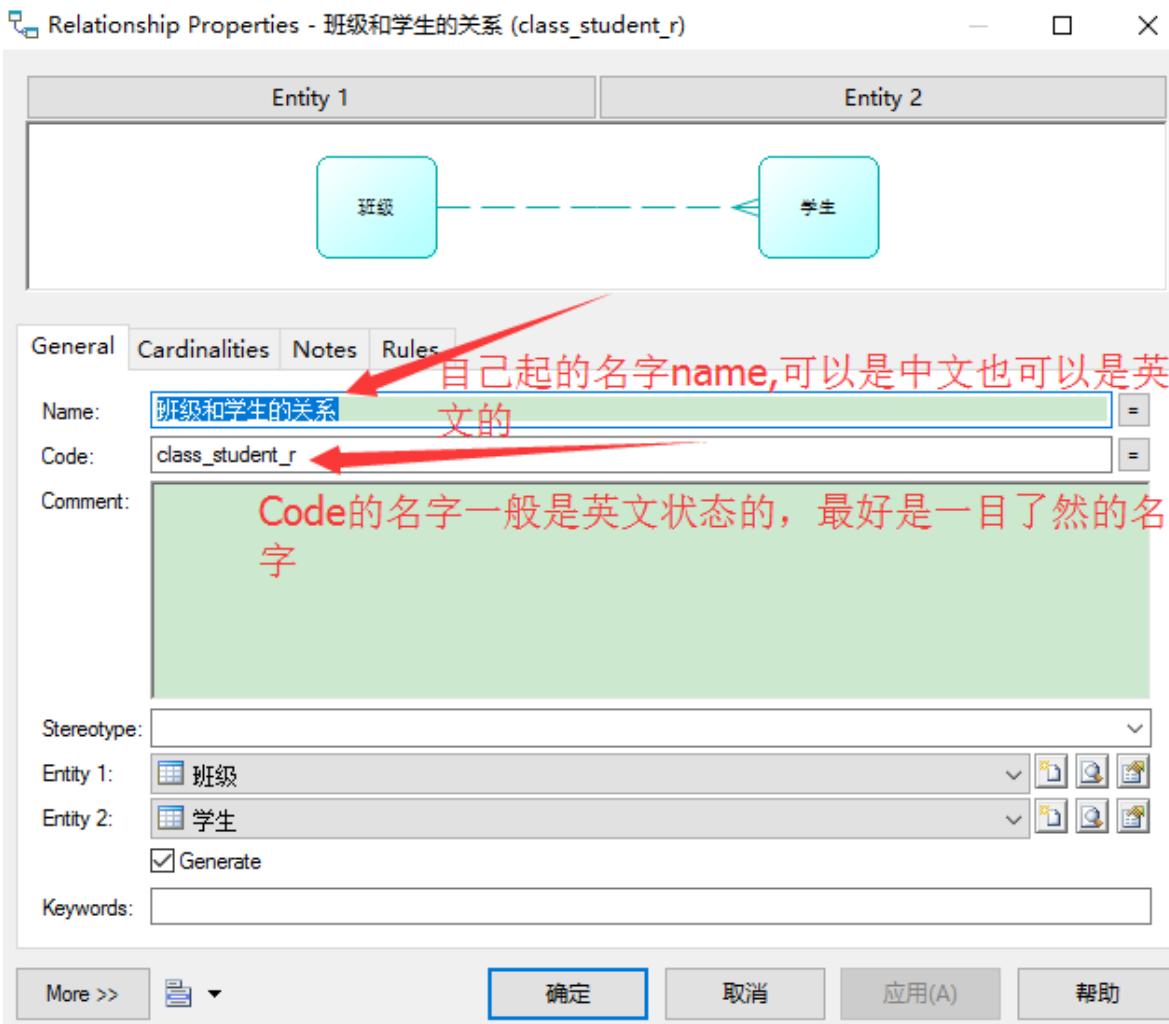


实体关系

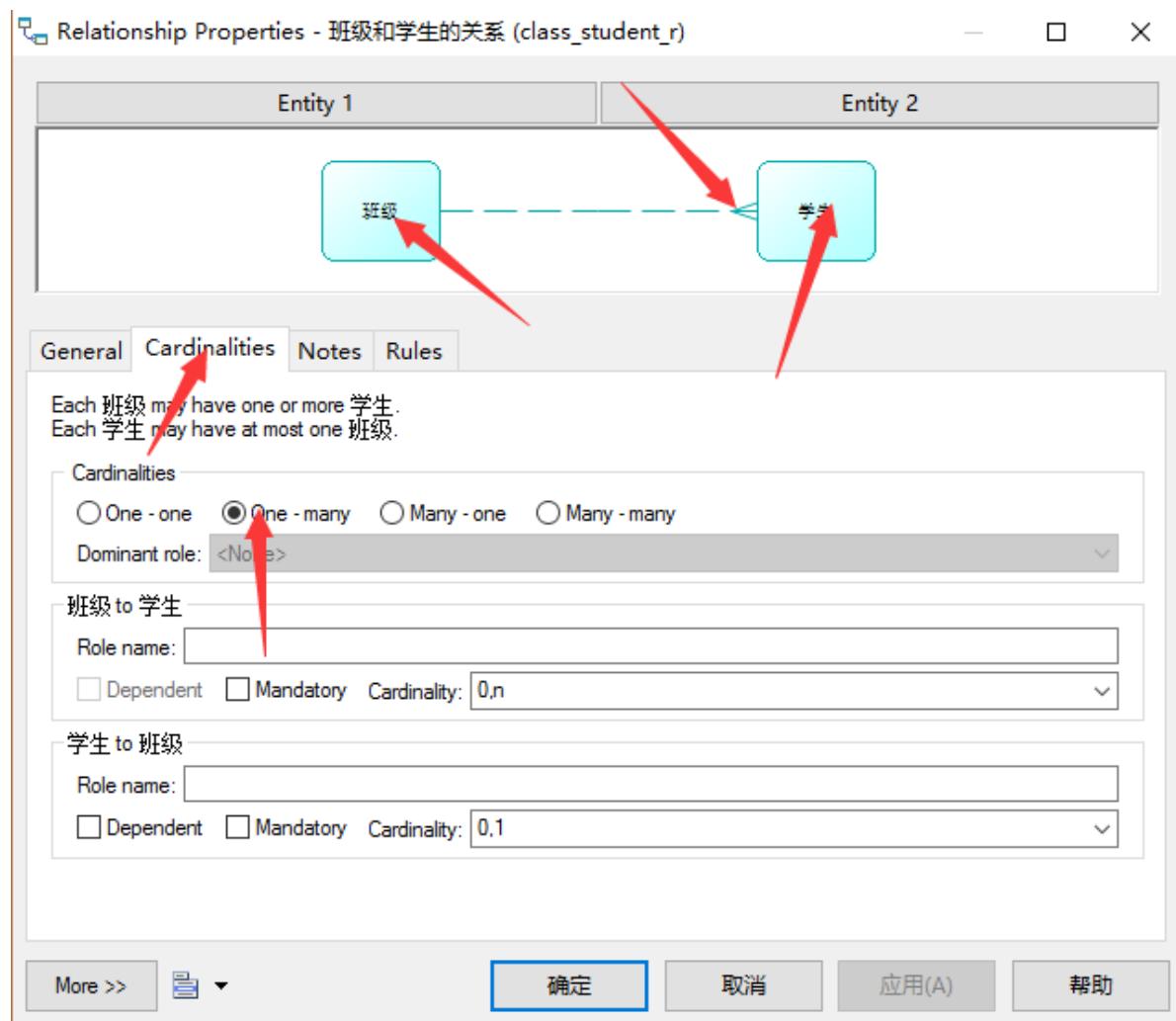
同理创建一个班级的实体（需要特别注意的是，点击完右边功能的按钮后需要点击鼠标指针状态的按钮或者右击鼠标即可，不然很容易乱操作，这点注意一下就可以了），然后使用Relationship（关系）这个按钮可以连接学生和班级之间的关系，发生一对多（班级对学生）或者多对一（学生对班级）的关系。如图所示



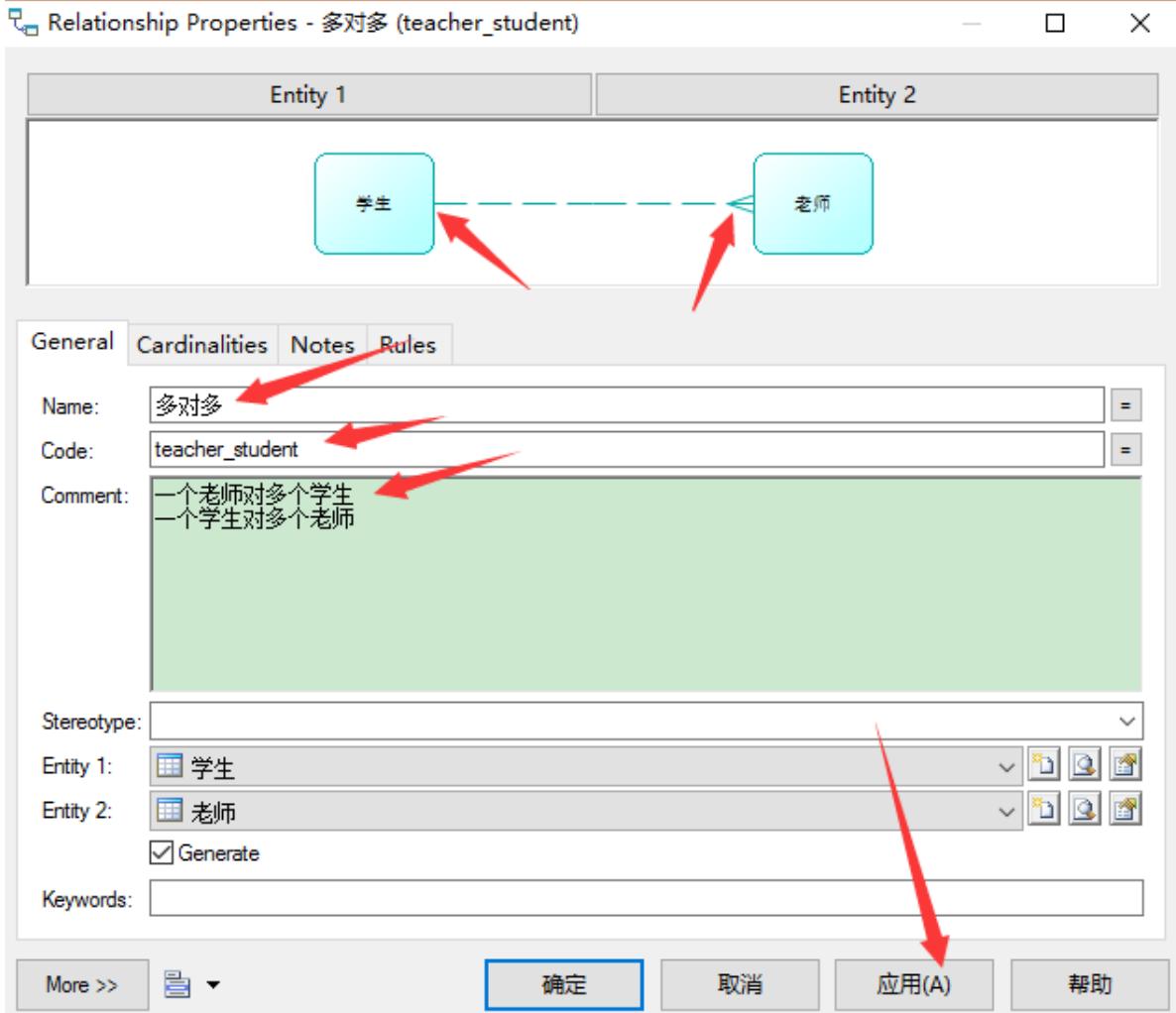
需要注意的是点击Relationship这个按钮，就把班级和学生联系起来了，就是一条线，然后双击这条线进行编辑，在General这块起name和code



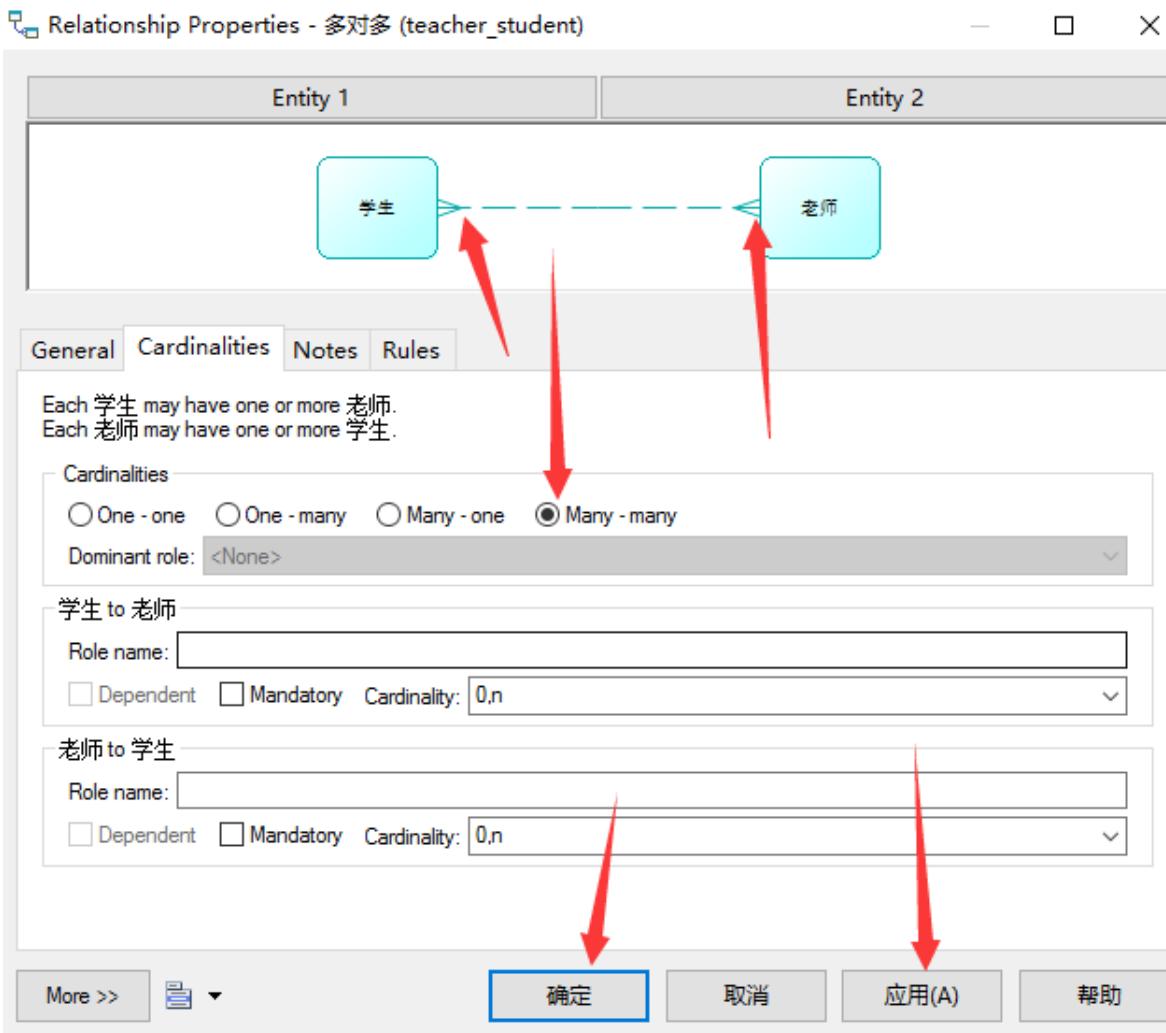
上面的name和code起好后就可以在Cardinalities这块查看班级和学生的关系，可以看到班级的一端是一条线，学生的一端是三条，代表班级对学生是一对多的关系即one对many的关系，点击应用，然后确定即可



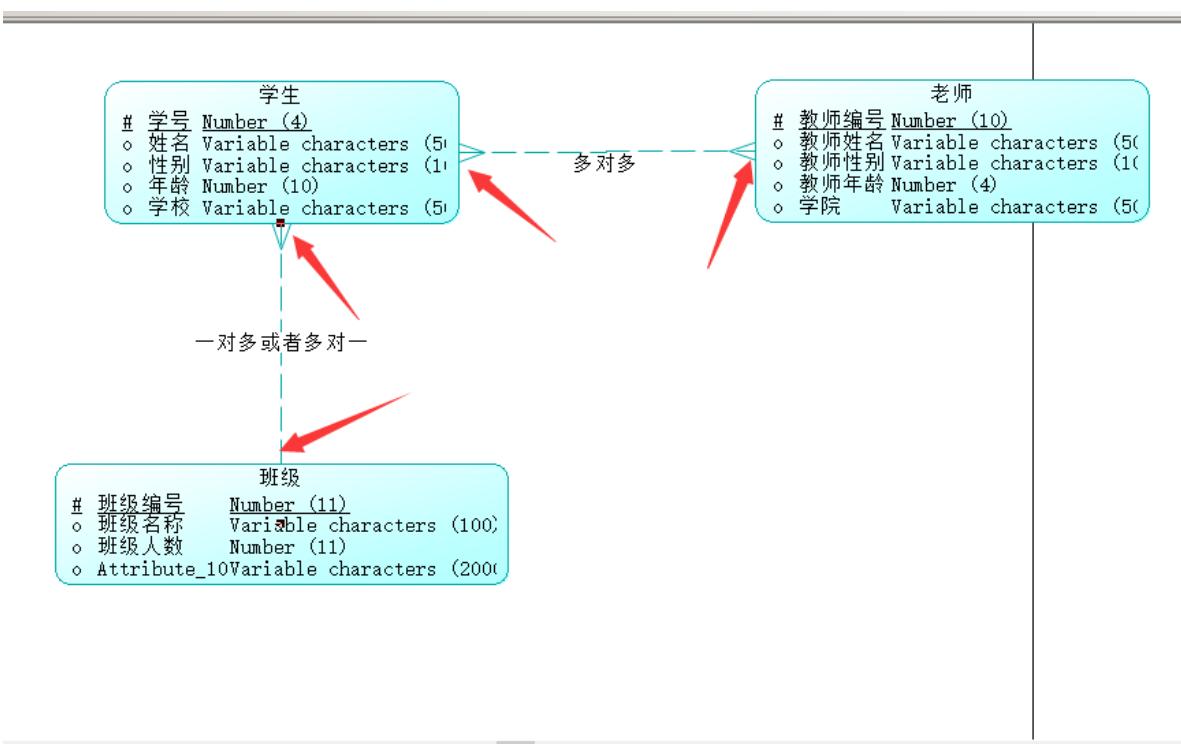
一对多和多对一练习完还有多对多的练习，如下图操作所示，老师实体和上面介绍的一样，自己将 name, data type等等修改成自己需要的即可，满足项目开发需求即可。（comment是解释说明，自己可以写相关的介绍和说明）



多对多需要注意的是自己可以手动点击按钮将关系调整称为多对多的关系many对many的关系，然后点击应用和确定即可

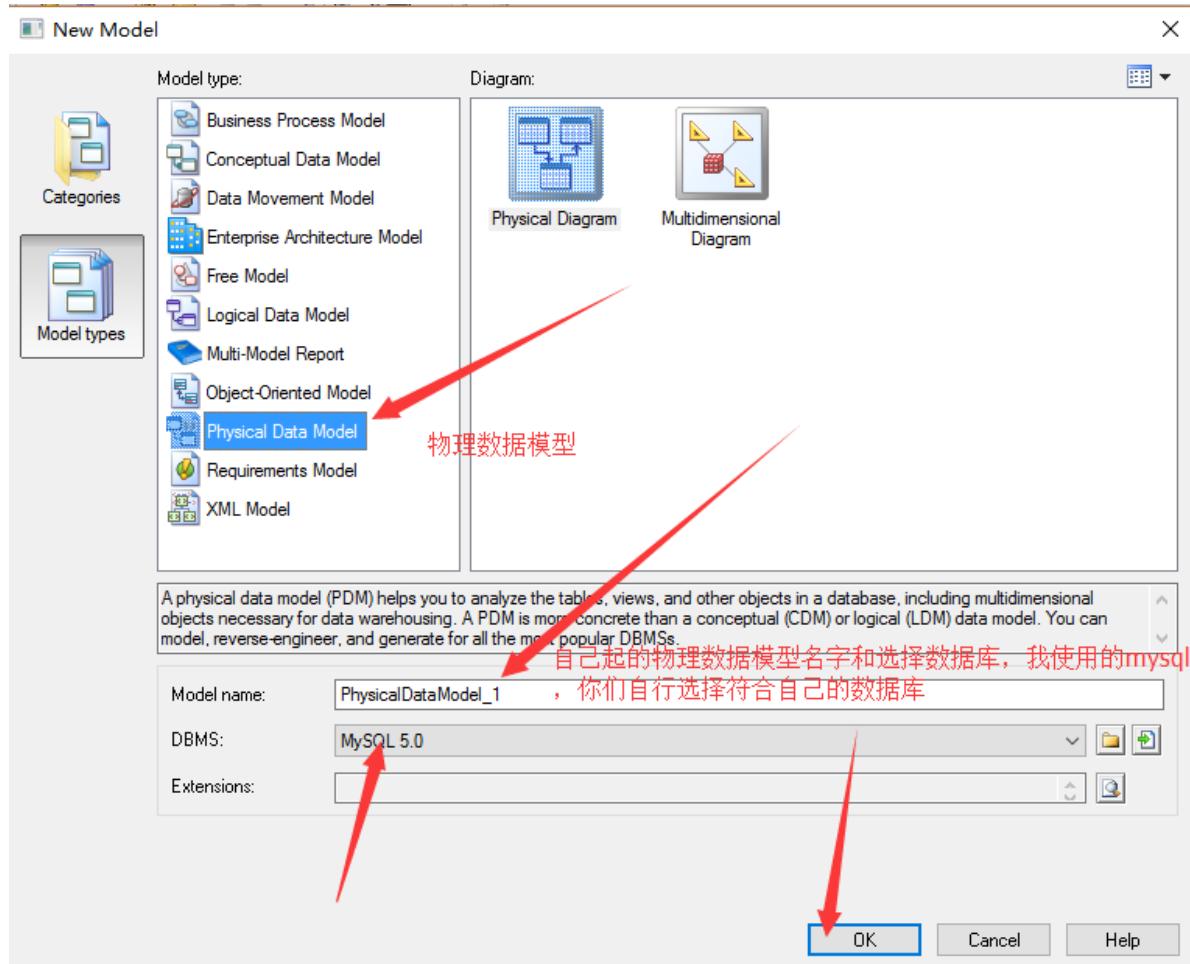


综上即可完成最简单的学生，班级，教师这种概念数据模型的设计，需要考虑数据的类型和主标识码，是否为空。关系是一对一还是一对多还是多对多的关系，自己需要先规划好再设计，然后就ok了。

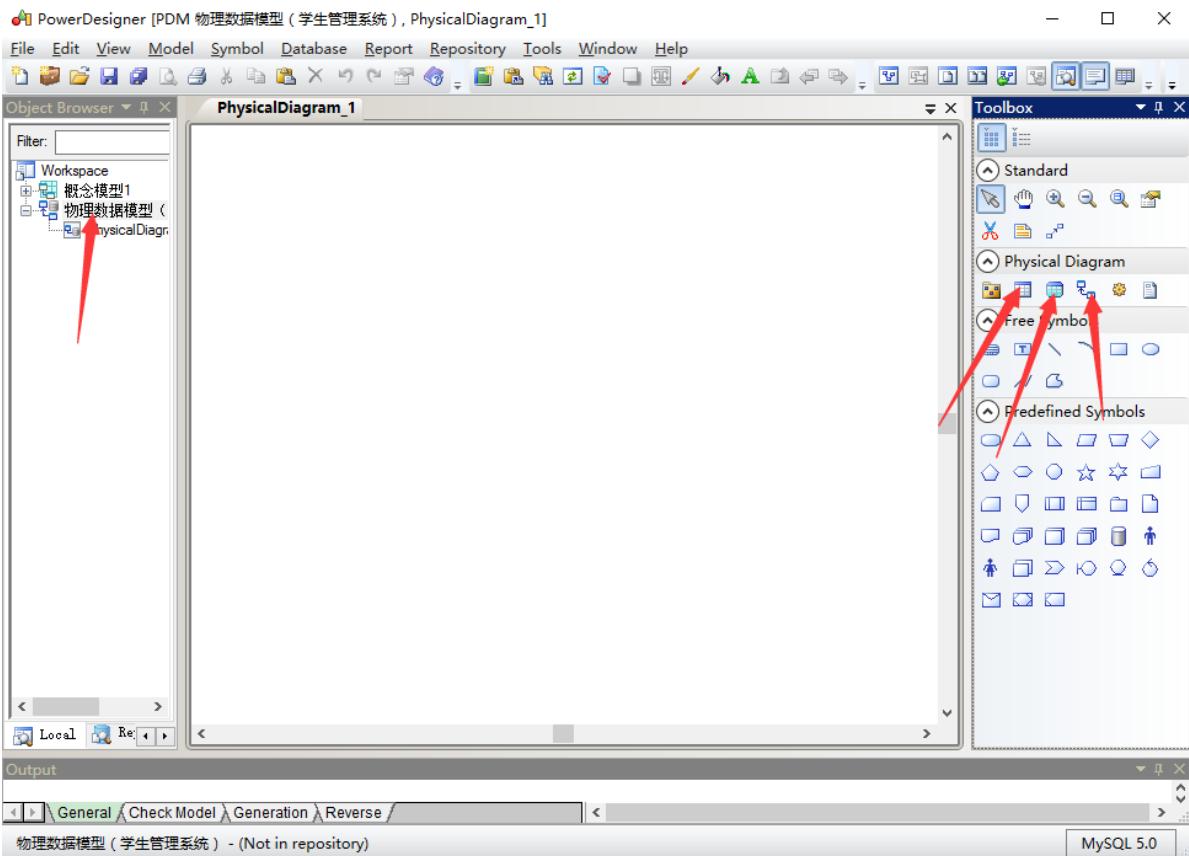


11.3 物理数据模型

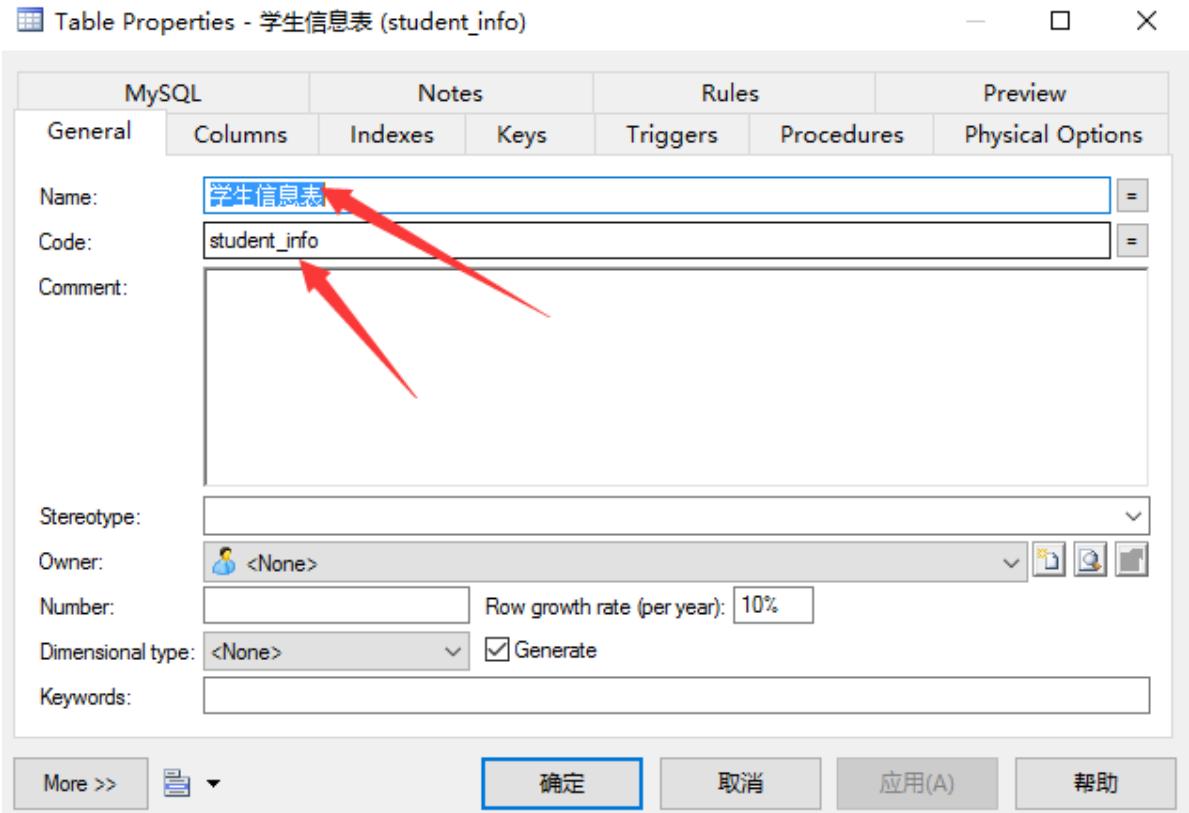
上面是概念数据模型，下面介绍一下物理数据模型，以后 经常使用 的就是物理数据模型。打开 PowerDesigner，然后点击File-->New Model然后选择如下图所示的物理数据模型，物理数据模型的名字自己起，然后选择自己所使用的数据库即可。



创建好主页面如图所示，但是右边的按钮和概念模型略有差别，物理模型最常用的三个是
table(表) , view(视图) , reference(关系) ;



鼠标先点击右边table这个按钮然后在新建的物理模型点一下，即可新建一个表，然后双击新建如下图所示，在General的name和code填上自己需要的，点击应用即可），如下图：



然后点击Columns,如下图设置，非常简单，需要注意的就是P (primary主键) , F (foreign key外键) , M (mandatory强制性的，代表不可为空) 这三个。

Table Properties - 学生信息表 (student_info)

MySQL

General Columns Indexes Keys Triggers Procedures Physical Options

	Name	Code	Data Type	Length	Precision	Scale	Null	Default	Extra
1	学号	student_id	int				<input checked="" type="checkbox"/>		
2	姓名	student_name	varchar(50)	50			<input type="checkbox"/>		
3	性别	student_sex	varchar(10)	10			<input type="checkbox"/>		
4	年龄	student_age	int				<input type="checkbox"/>		
5	学校	学校	varchar(150)	150			<input type="checkbox"/>		

More >> 确定 取消 应用(A) 帮助

在此设置学号的自增 (MYSQL里面的自增是这个AUTO_INCREMENT) , 班级编号同理, 不多赘述!

Table Properties - 学生信息表 (student_info)

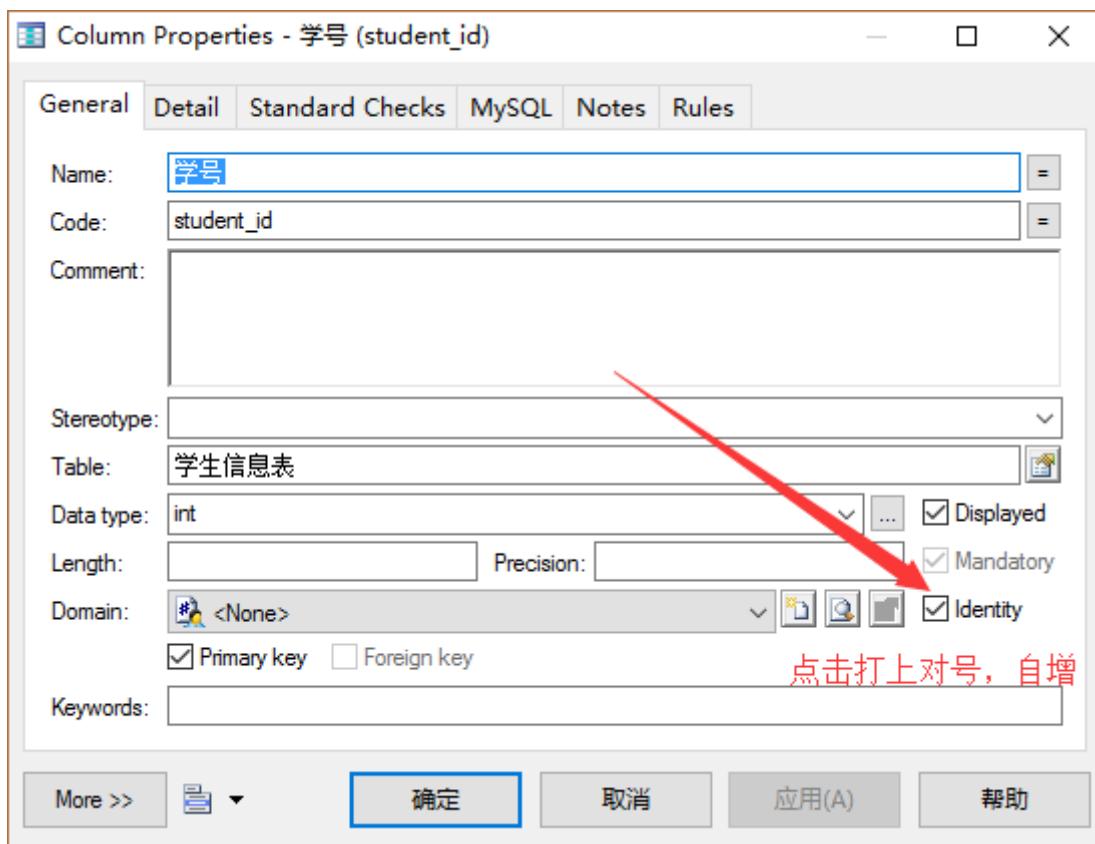
MySQL

General Columns Indexes Keys Triggers Procedures Physical Options

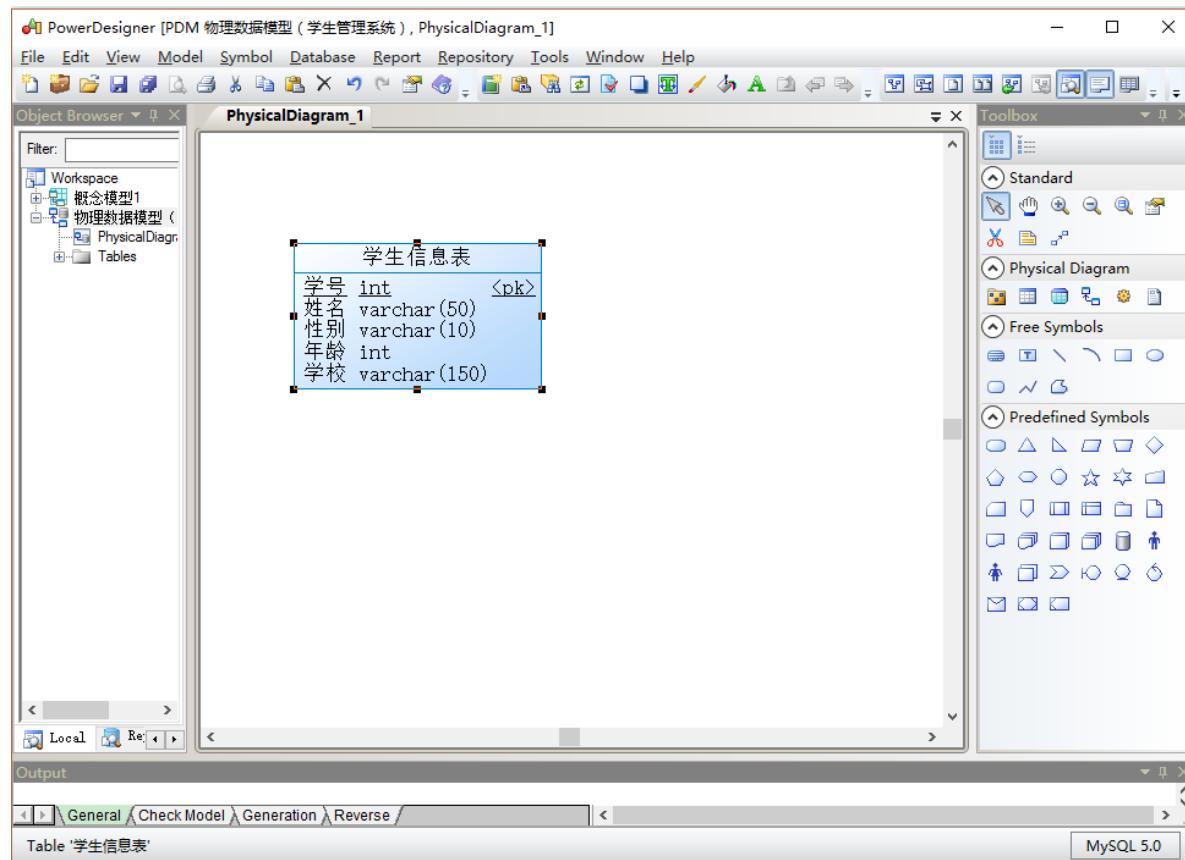
	Name	Code	Data Type	Length	Precision	Scale	Null	Default	Extra
1	学号	student_id	int				<input checked="" type="checkbox"/>		
2	姓名	student_name	varchar(50)	50			<input type="checkbox"/>		
3	性别	student_sex	varchar(10)	10			<input type="checkbox"/>		
4	年龄	student_age	int				<input type="checkbox"/>		
5	学校	学校	varchar(150)	150			<input type="checkbox"/>		

More >> 确定 取消 应用(A) 帮助

在下面的这个点上对号即可, 就设置好了自增



全部完成后如下图所示。



班级物理模型同理如下图所示创建即可

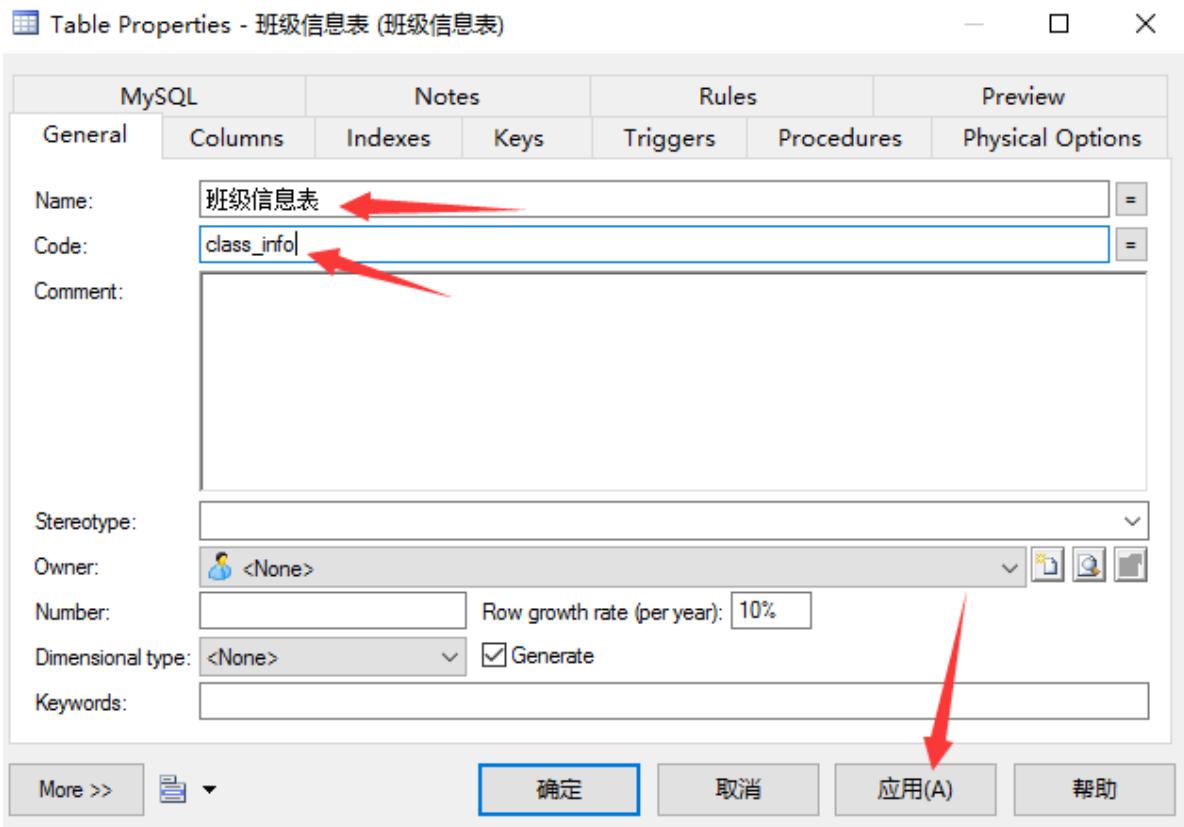


Table Properties - 班级信息表 (class_info)

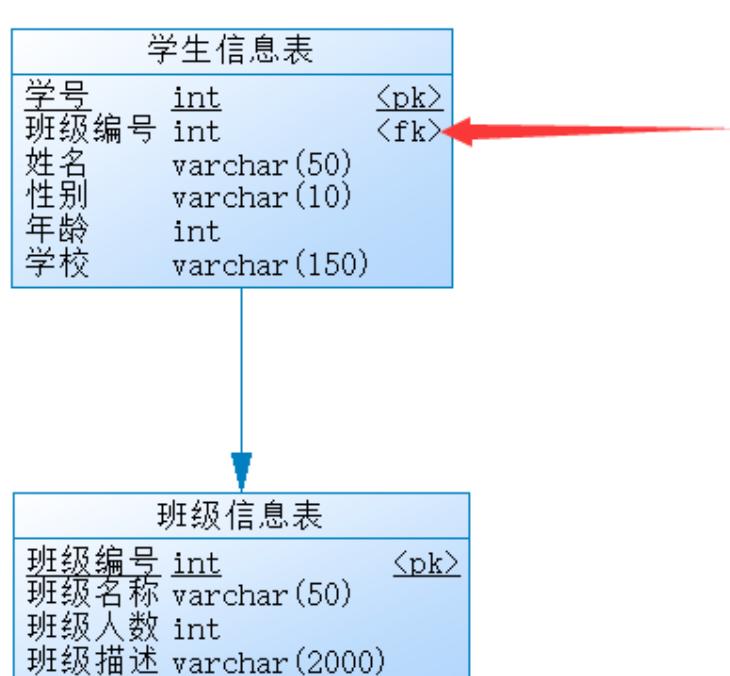
MySQL		Notes		Rules		Preview	
General	Columns	Indexes	Keys	Triggers	Procedures	Physical Options	
Name	Code	Data Type	Length	Precision	Scale	Nullable	Default Value
1. 班级编号	= class_id	int				<input checked="" type="checkbox"/>	<input type="checkbox"/>
2. 班级名称	class_name	varchar(50)	50			<input type="checkbox"/>	<input type="checkbox"/>
3. 班级人数	class_number	int				<input type="checkbox"/>	<input type="checkbox"/>
4. 班级描述	class_desc	varchar(2000)	2,000			<input type="checkbox"/>	<input type="checkbox"/>
↑ ↑ ↑ ↓ ↓ ↵ < > ↓ ↑ ↓ ↵ ↑ ↑ ↑ ↓ ↓ ↵							
More >>		<input type="button"/>		<input type="button"/> 确定		<input type="button"/> 取消	
				<input type="button"/> 应用(A)		<input type="button"/> 帮助	

完成后如下图所示

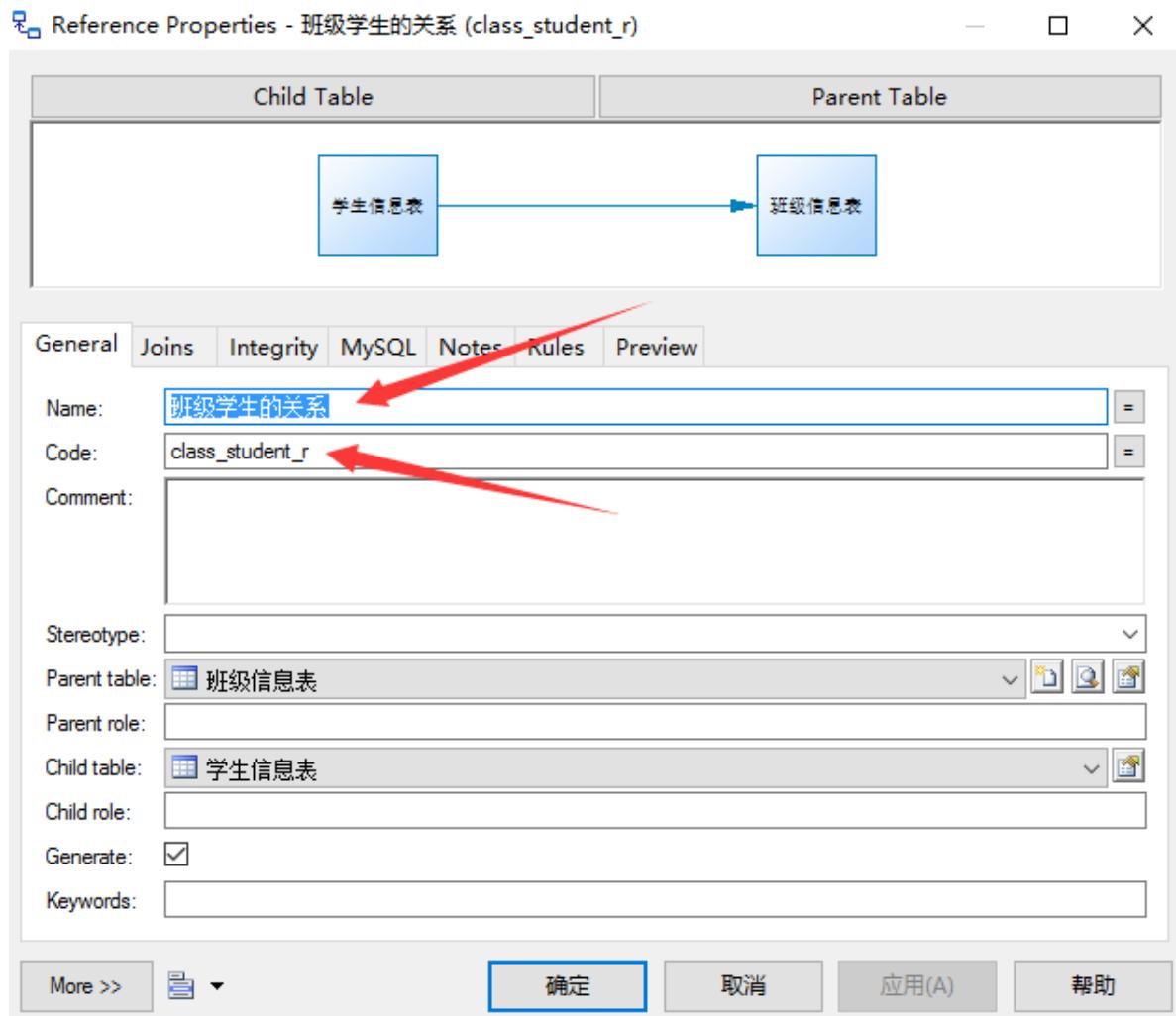
学生信息表		
学号	int	<pk>
姓名	varchar(50)	
性别	varchar(10)	
年龄	int	
学校	varchar(150)	

班级信息表		
班级编号	int	<pk>
班级名称	varchar(50)	
班级人数	int	
班级描述	varchar(2000)	

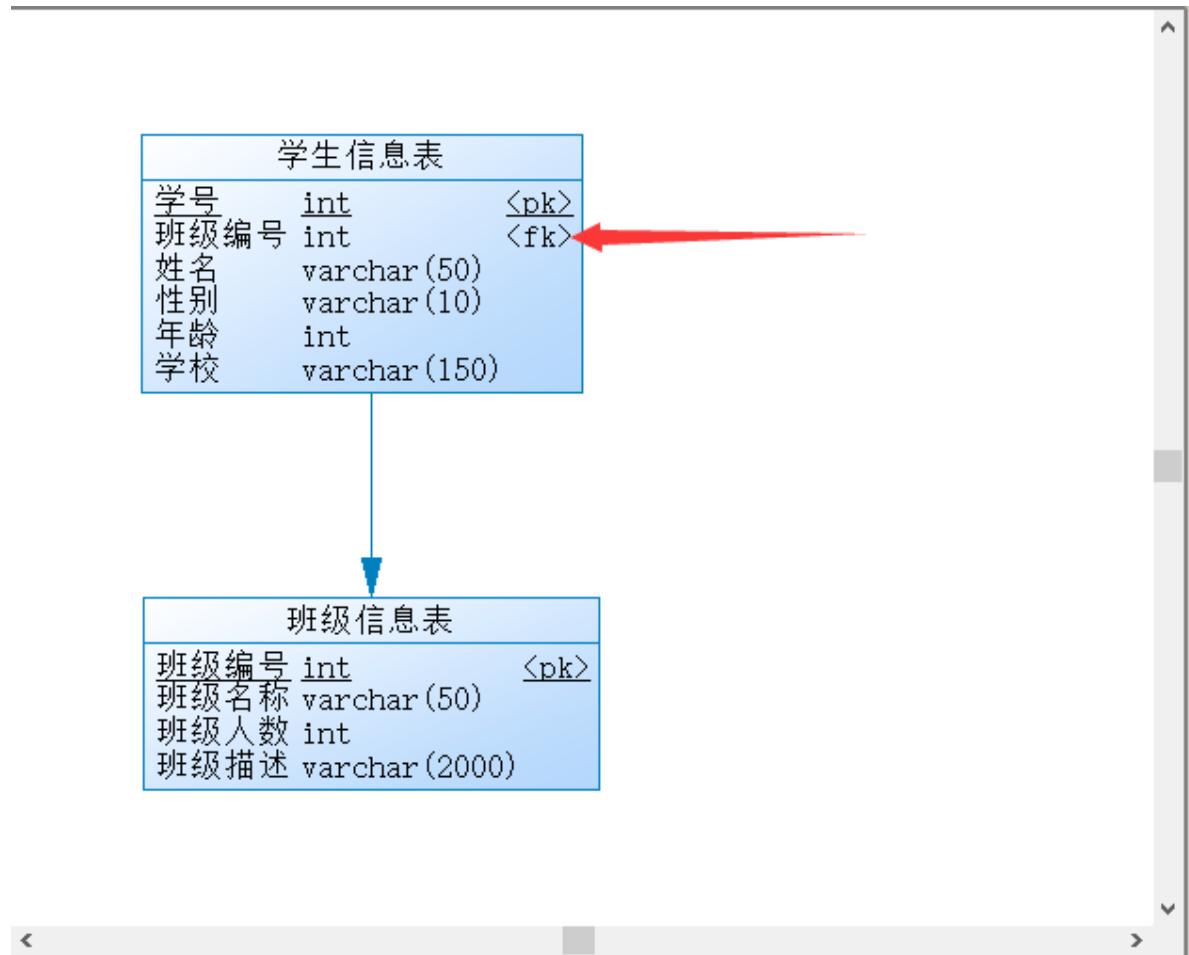
上面的设置好如上图所示，然后下面是关键的地方，点击右边按钮Reference这个按钮，因为是班级对学生是一对多的，所以鼠标从学生拉到班级如下图所示，学生表将发生变化，学生表里面增加了一行，这行是班级表的主键作为学生表的外键，将班级表和学生表联系起来。（仔细观察即可看到区别。）



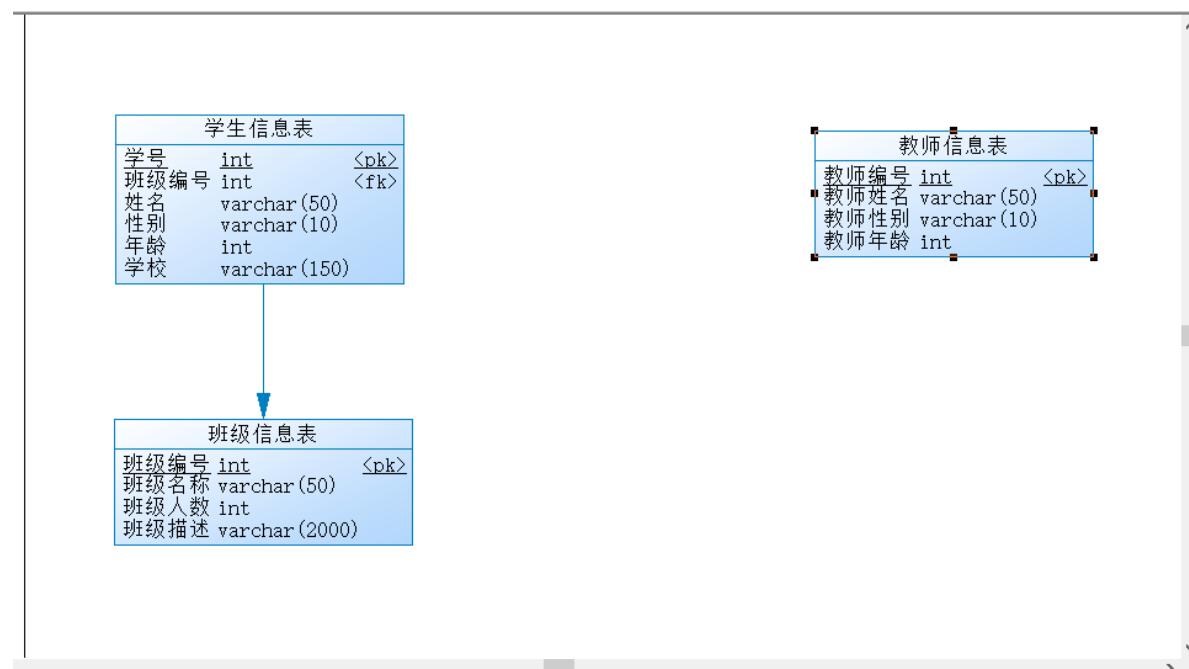
做完上面的操作，就可以双击中间的一条线，显示如下图，修改name和code即可



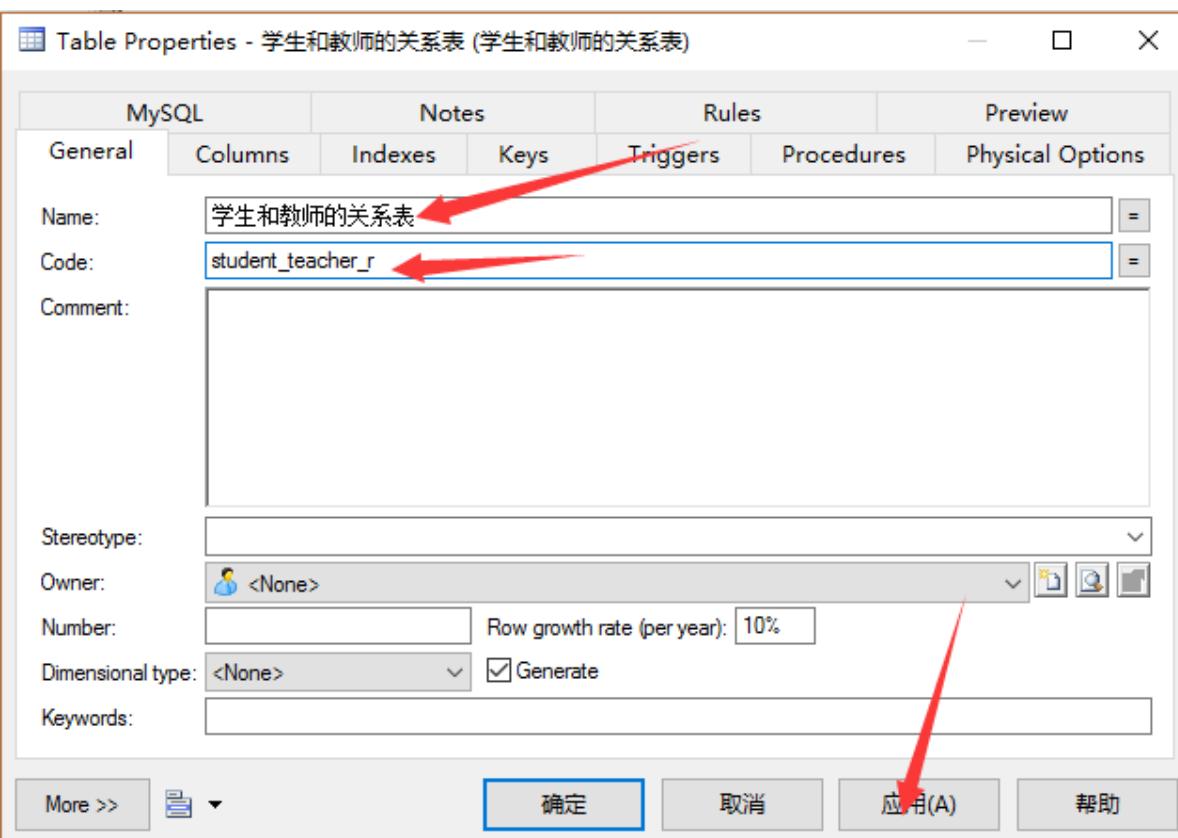
但是需要注意的是，修改完毕后显示的结果却如下图所示，并没有办法直接像概念模型那样，修改过后显示在中间的那条线上面，自己明白即可。



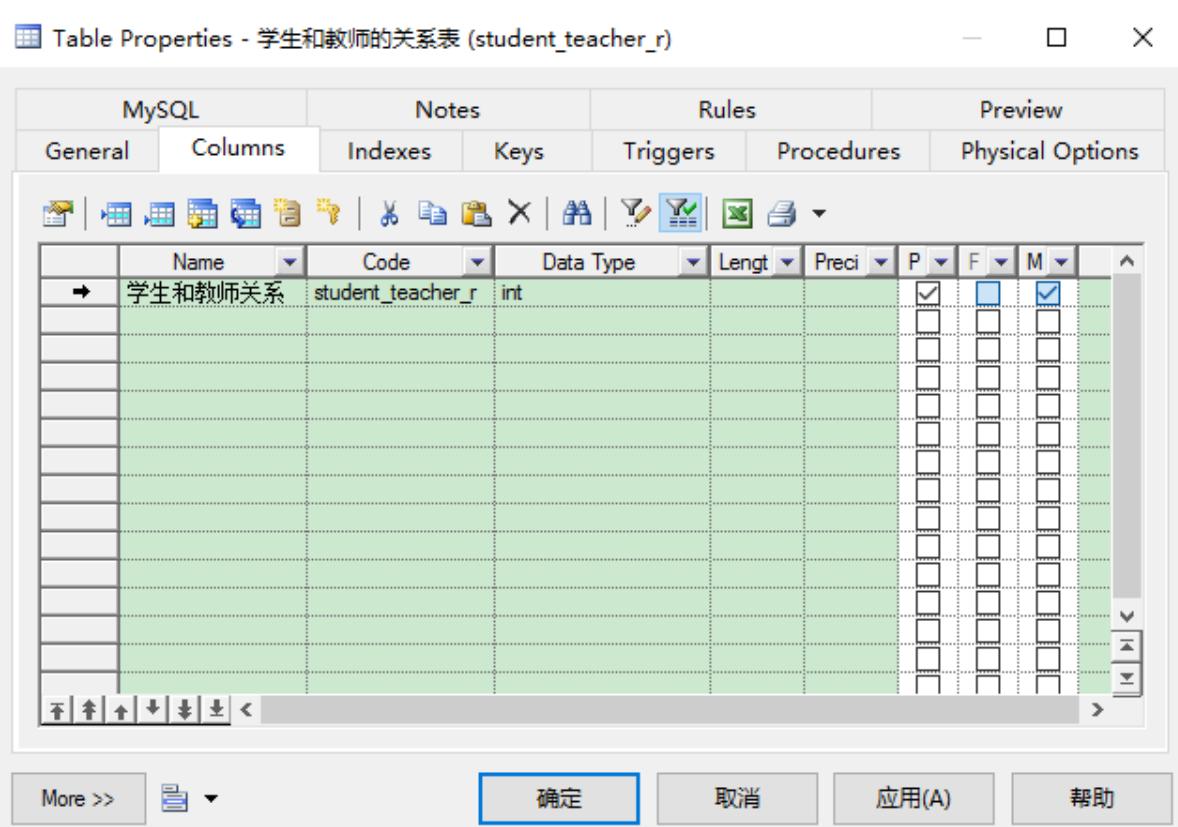
学习了多对一或者一对多的关系，接下来学习多对多的关系，同理自己建好老师表，这里不在叙述，记得老师编号自增，建好如下图所示



下面是多对多关系的关键，由于物理模型多对多的关系需要一个中间表来连接，如下图，只设置一个字段，主键，自增



点击应用，然后设置Columns，只添加一个字段

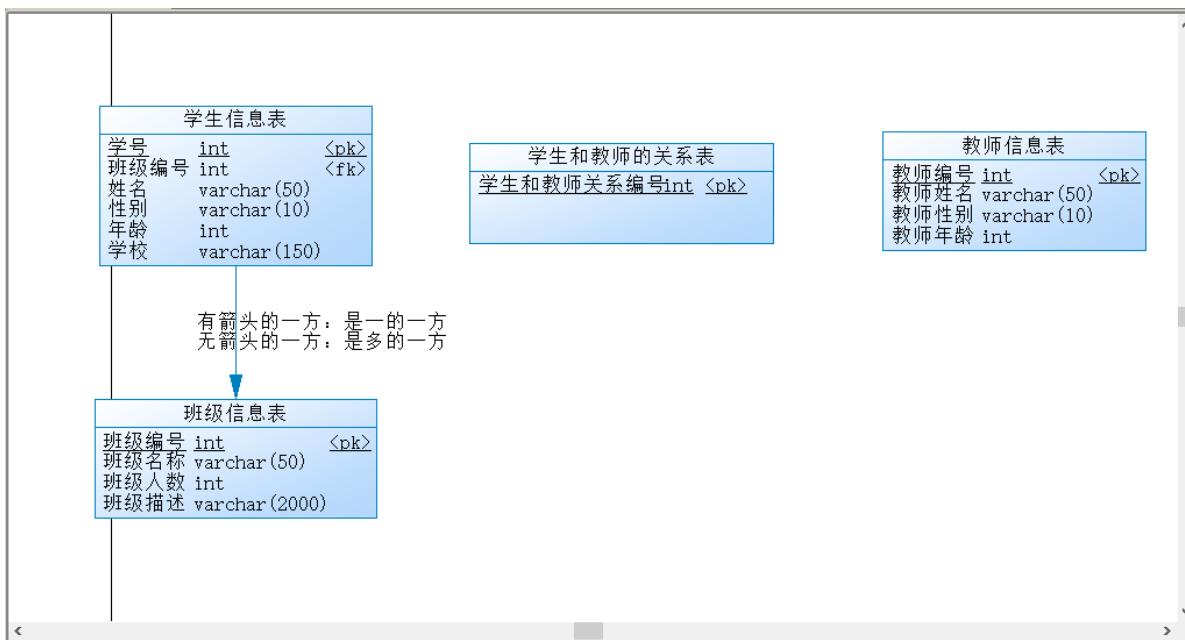


这是设置字段递增，前面已经叙述过好几次

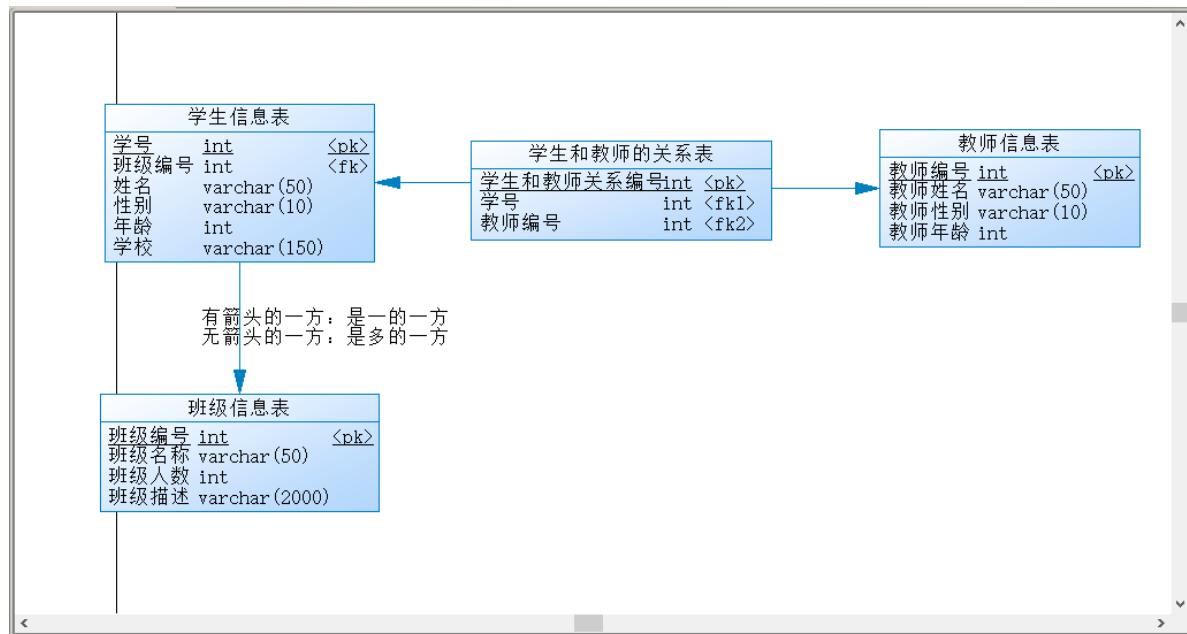
Column Properties - 学生和教师关系编号 (student_teacher_r)

General	Detail	Standard Checks	MySQL	Notes	Rules
Name:	学生和教师关系编号	=			
Code:	student_teacher_r	=			
Comment:					
Stereotype:					
Table:	学生和教师的关系表				
Data type:	int	...	<input checked="" type="checkbox"/> Displayed		
Length:		Precision:		<input checked="" type="checkbox"/> Mandatory	
Domain:	<None>	...	<input checked="" type="checkbox"/> Primary key	<input type="checkbox"/> Foreign key	<input checked="" type="checkbox"/> Identity
Keywords:					
<input type="button" value="More >>"/> <input type="button" value="确定"/> <input type="button" value="取消"/> <input type="button" value="应用(A)"/> <input type="button" value="帮助"/>					

设置好后如下图所示，需要注意的是有箭头的一方是一，无箭头的一方是多，即一对多的多对一的关系需要搞清楚，学生也可以有很多老师，老师也可以有很多学生，所以学生和老师都可以是主体；

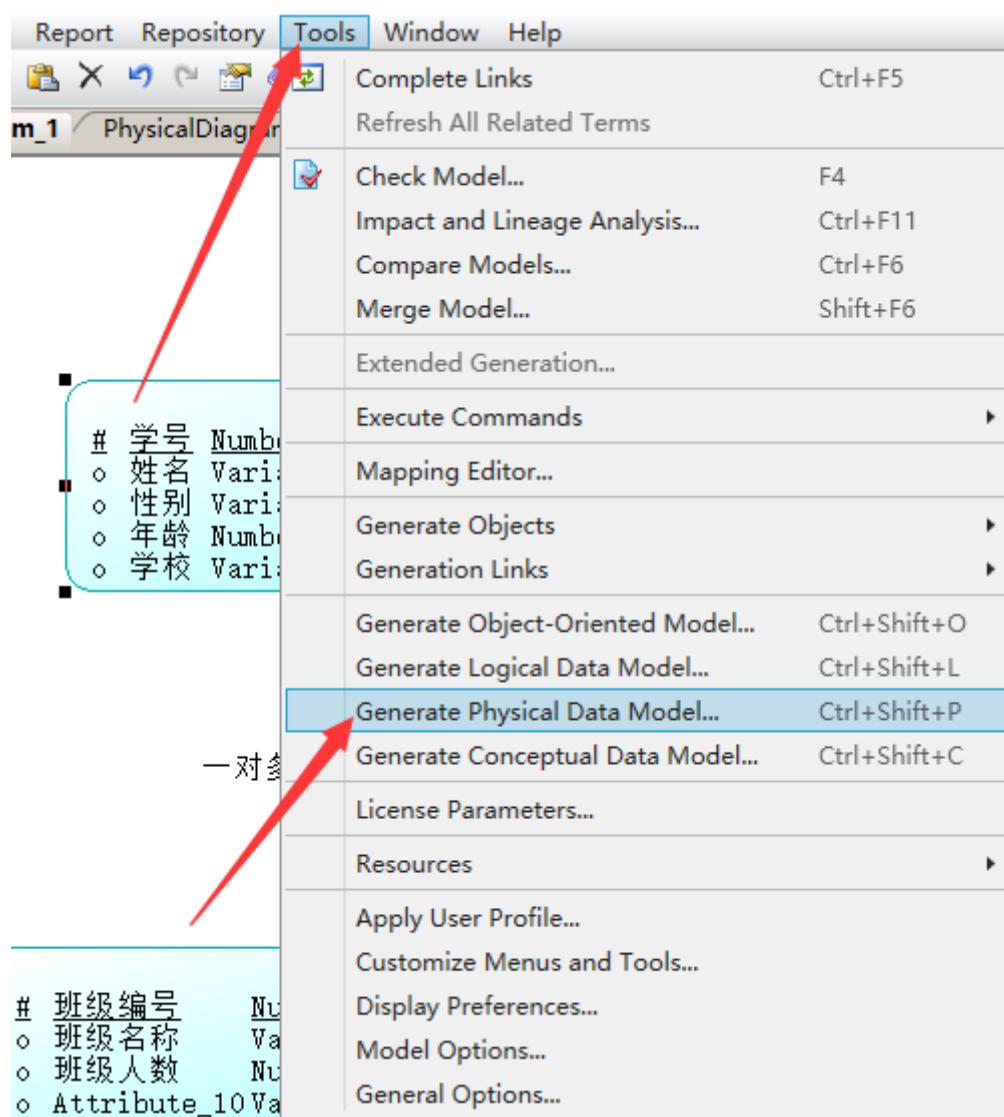


可以看到添加关系以后学生和教师的关系表前后发生的变化



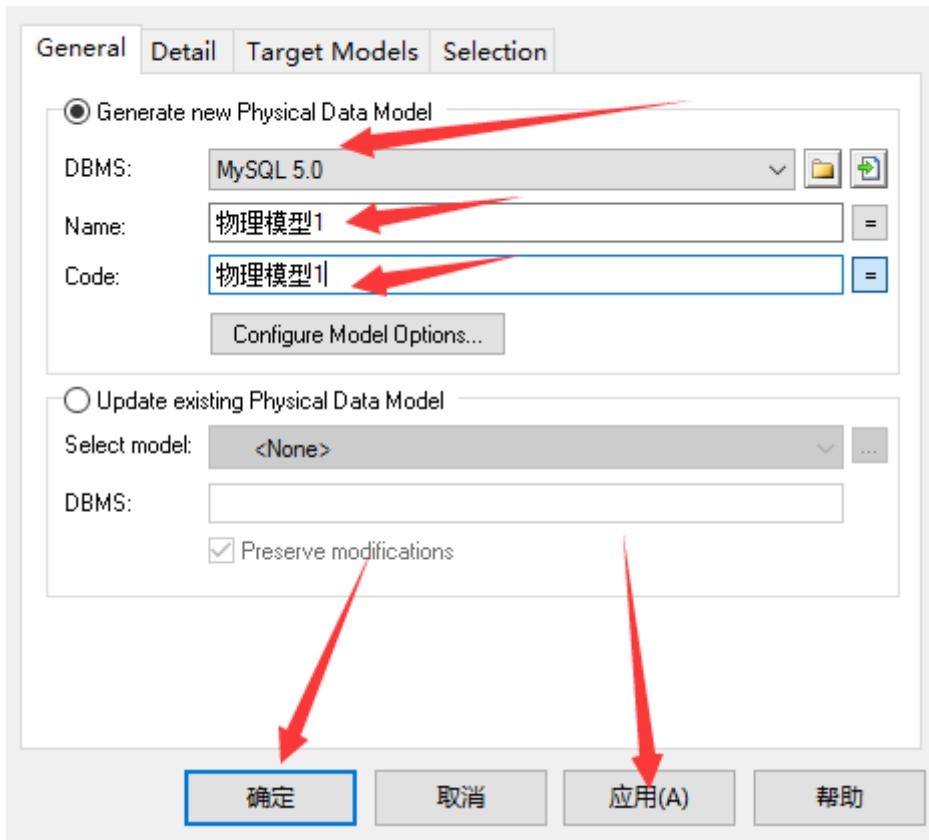
11.4 概念模型转为物理模型

1：如下图所示先打开概念模型图，然后点击Tools,如下图所示

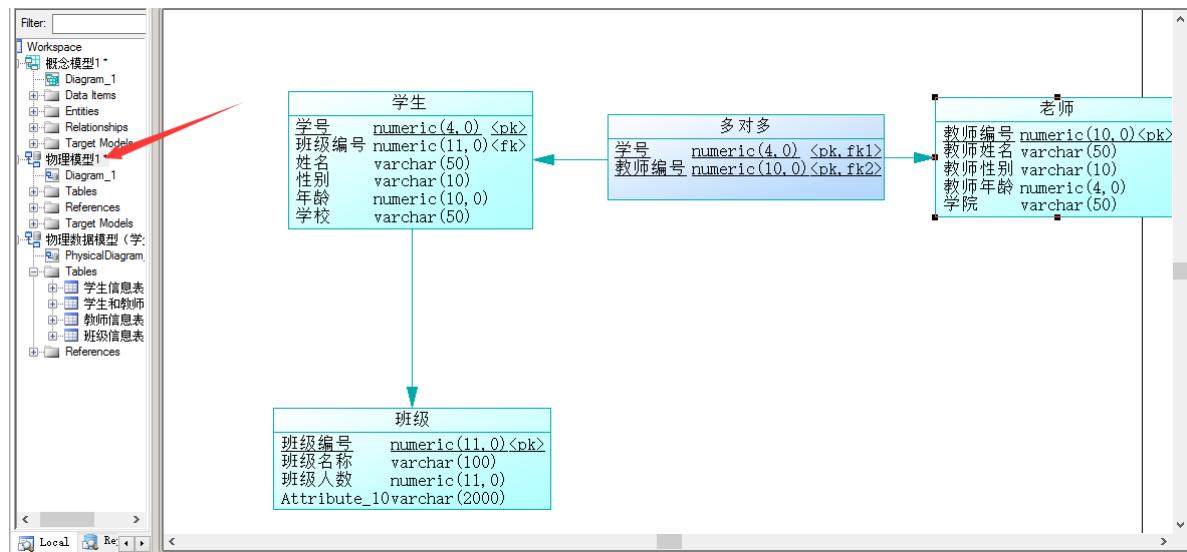


点开的页面如下所示，name和code已经从概念模型1改成物理模型1了

PDM Generation Options

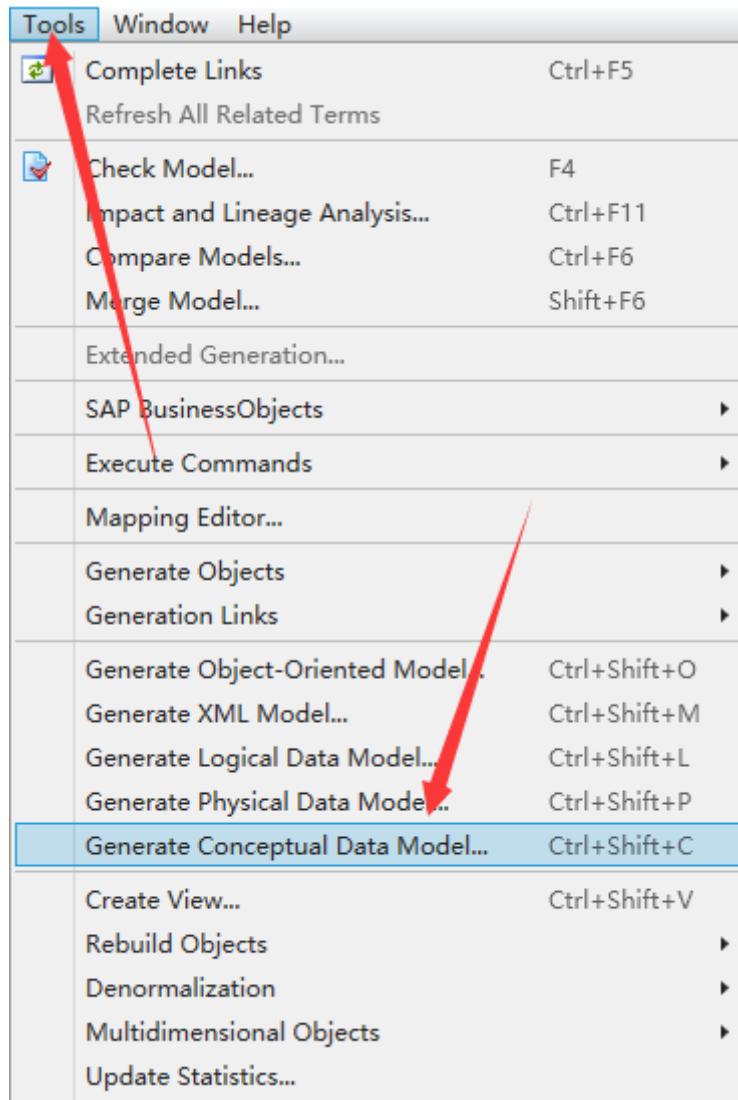


完成后如下图所示，将自行打开修改的物理模型，需要注意的是这些表的数据类型已经自行改变了，而且中间表出现两个主键，即双主键，

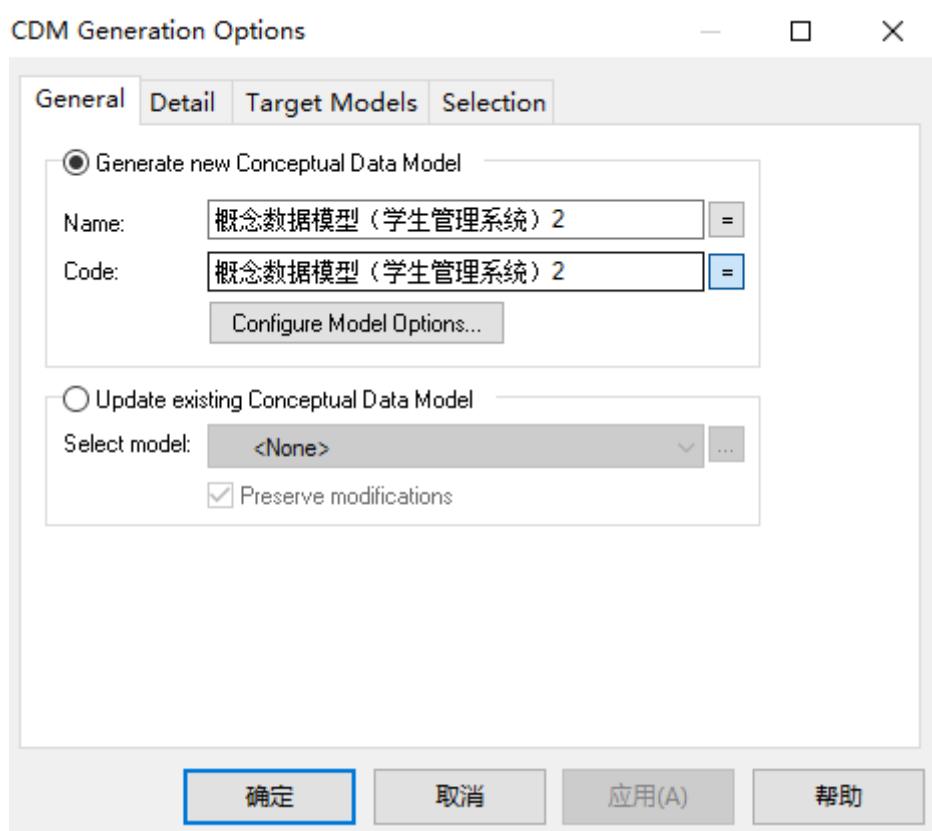


11.5 物理模型转为概念模型

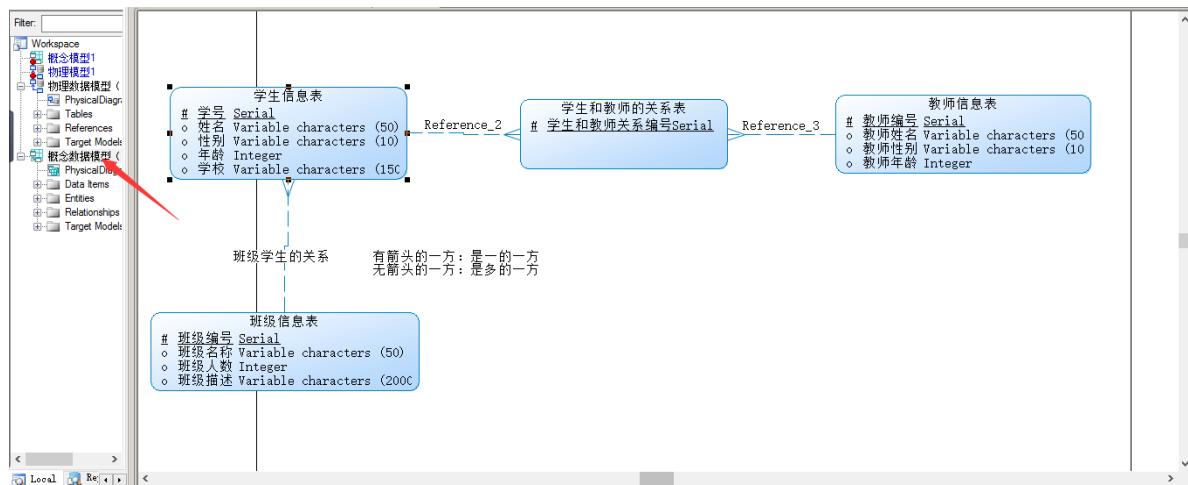
上面介绍了概念模型转物理模型，下面介绍一下物理模型转概念模型（如下图点击操作即可）



然后出现如下图所示界面，然后将物理修改为概念，点击应用确认即可

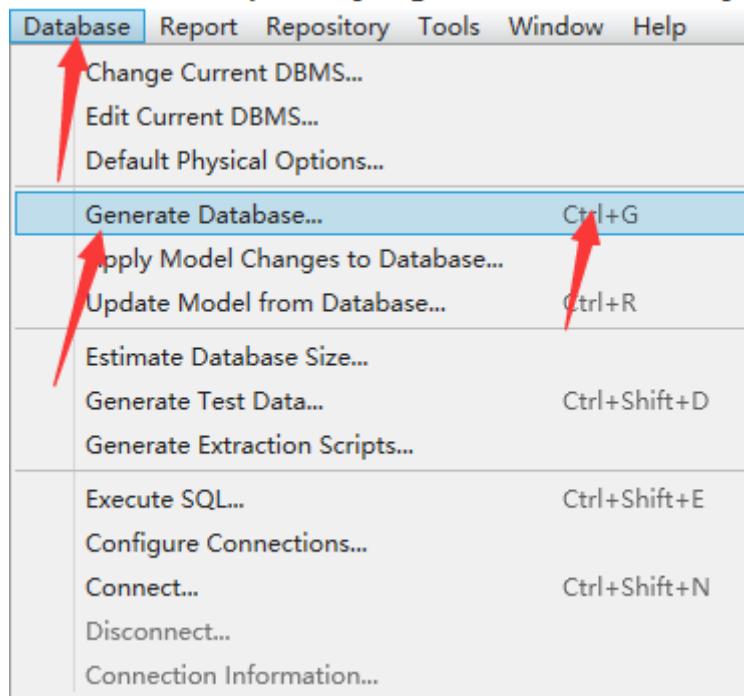


点击确认后将自行打开如下图所示的页面，自己观察有何变化，如果转换为oracle的，数据类型会发生变化，比如Varchar2等等）；

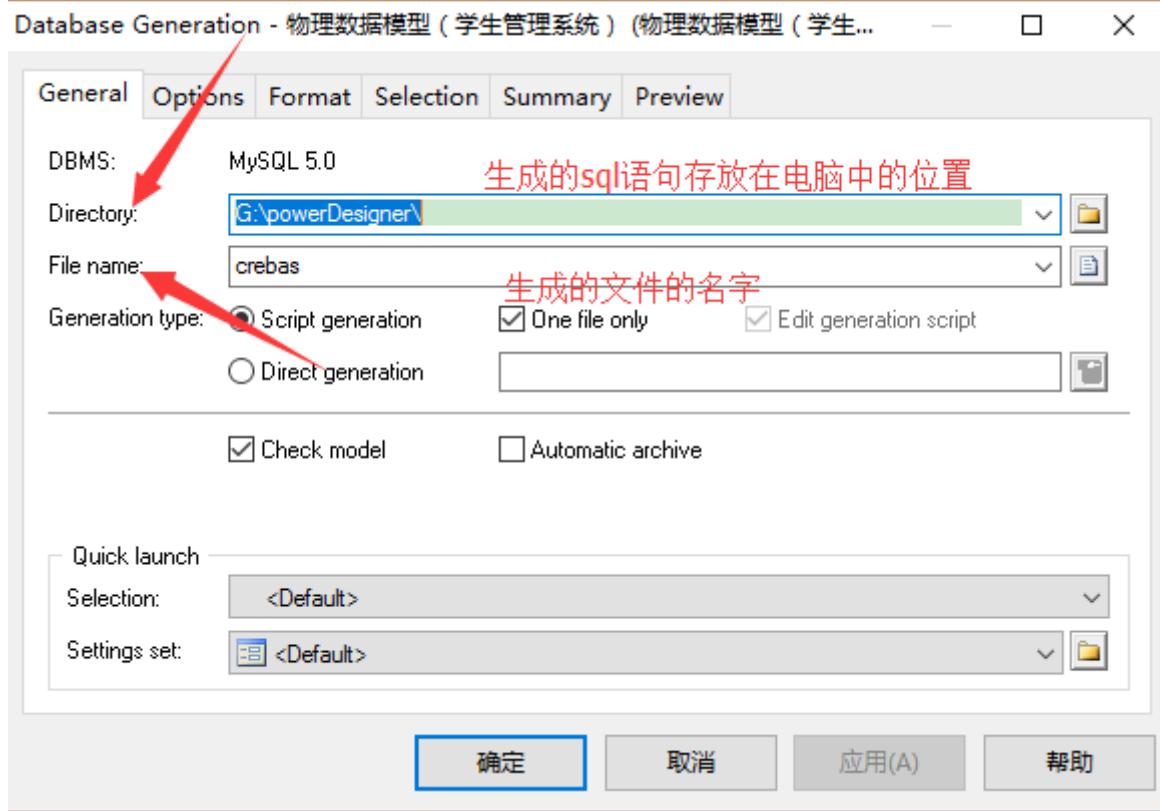


11.6 物理模型导出SQL语句

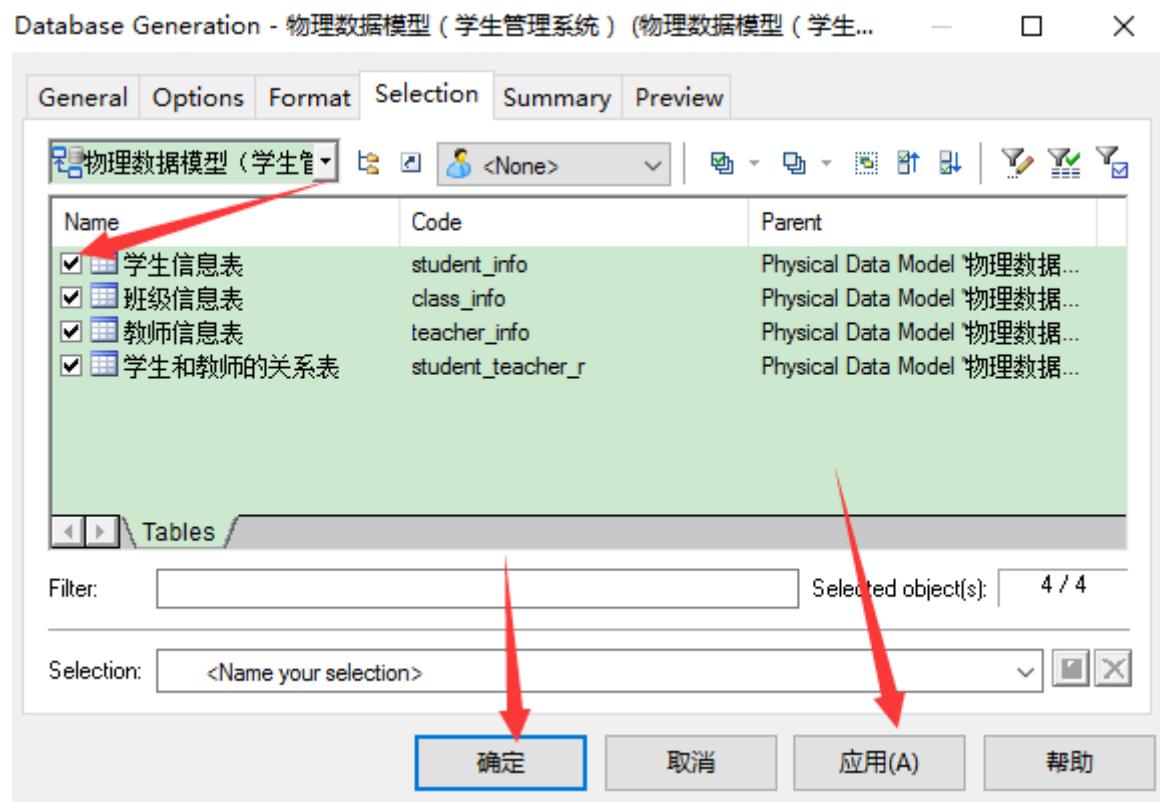
下面介绍一下物理模型导出SQL语句（点击Database按钮的Generate Database或者按ctrl+G）



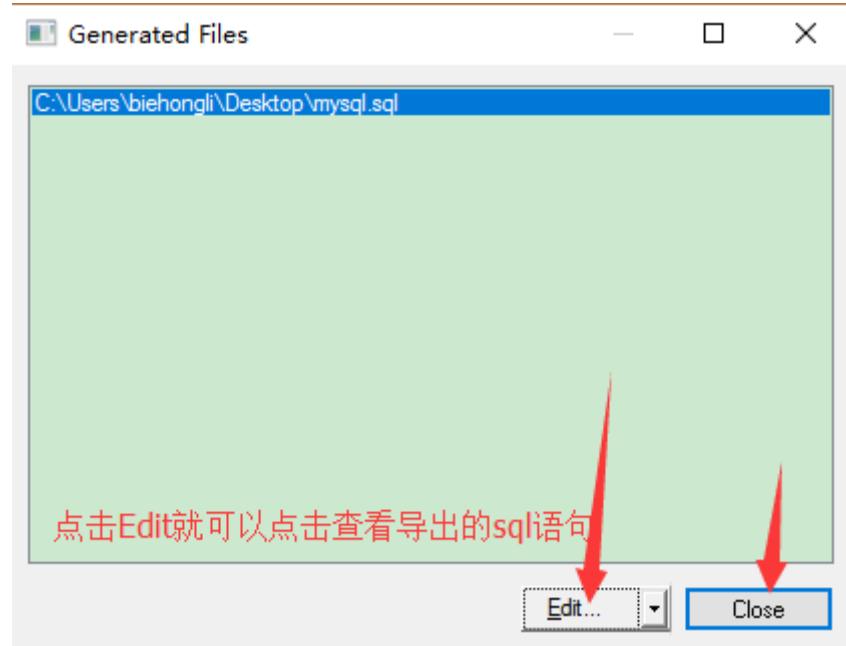
打开之后如图所示，修改好存在sql语句的位置和生成文件的名称即可



在Selection中选择需要导出的表，然后点击应用和确认即可



完成以后出现如下图所示，可以点击Edit或者close按钮



自此，就完成了导出sql语句，就可以到自己指定的位置查看导出的sql语句了；PowerDesigner在以后在项目开发过程中用来做需求分析和数据库的设计非常的方便和快捷。

第12章_数据库其它调优策略

讲师：尚硅谷-宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

1. 数据库调优的措施

1.1 调优的目标

- 尽可能节省系统资源，以便系统可以提供更大负荷的服务。（吞吐量更大）
- 合理的结构设计和参数调整，以提高用户操作响应的速度。（响应速度更快）
- 减少系统的瓶颈，提高MySQL数据库整体的性能。

1.2 如何定位调优问题

如何确定呢？一般情况下，有如下几种方式：

- 用户的反馈（主要）
- 日志分析（主要）
- 服务器资源使用监控
- 数据库内部状况监控
- 其它

除了活动会话监控以外，我们也可以对事务、锁等待等进行监控，这些都可以帮助我们对数据库的运行状态有更全面的认识。

1.4 调优的维度和步骤

我们需要调优的对象是整个数据库管理系统，它不仅包括SQL查询，还包括数据库的部署配置、架构等。从这个角度来说，我们思考的维度就不仅仅局限在SQL优化上了。通过如下的步骤我们进行梳理：

第1步：选择适合的DBMS

第2步：优化表设计

第3步：优化逻辑查询

第4步：优化物理查询

物理查询优化是在确定了逻辑查询优化之后，采用物理优化技术（比如索引等），通过计算代价模型对各种可能的访问路径进行估算，从而找到执行方式中代价最小的作为执行计划。在这个部分中，我们需要掌握的重点是对索引的创建和使用。

第5步：使用 Redis 或 Memcached 作为缓存

除了可以对 SQL 本身进行优化以外，我们还可以请外援提升查询的效率。

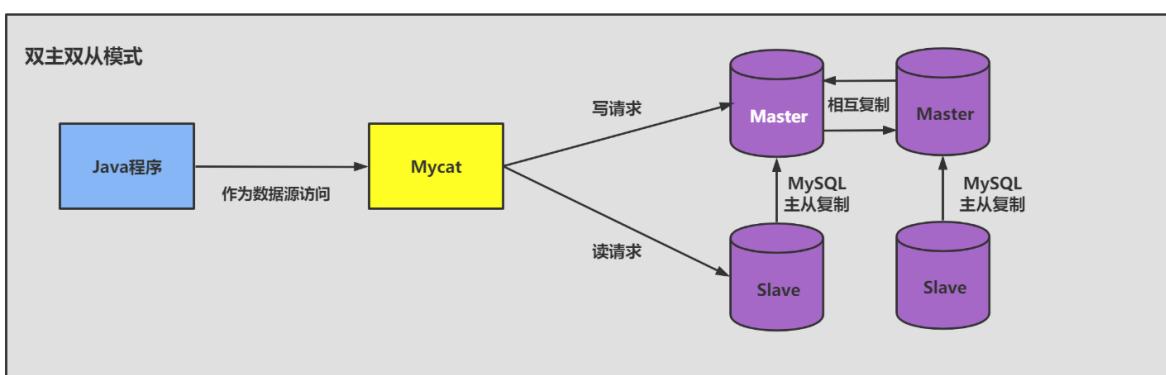
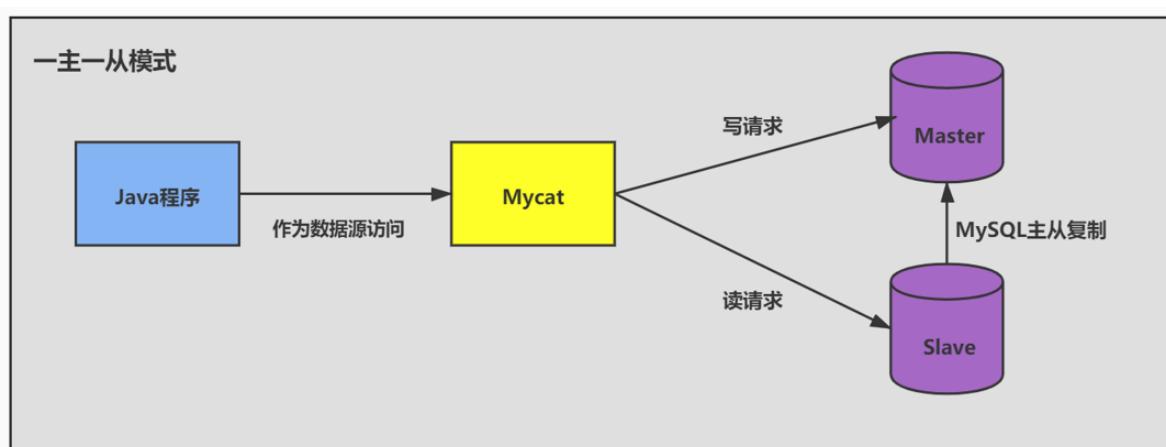
因为数据都是存放到数据库中，我们需要从数据库层中取出数据放到内存中进行业务逻辑的操作，当用户量增大的时候，如果频繁地进行数据查询，会消耗数据库的很多资源。如果我们将常用的数据直接放到内存中，就会大幅提升查询的效率。

键值存储数据库可以帮助我们解决这个问题。

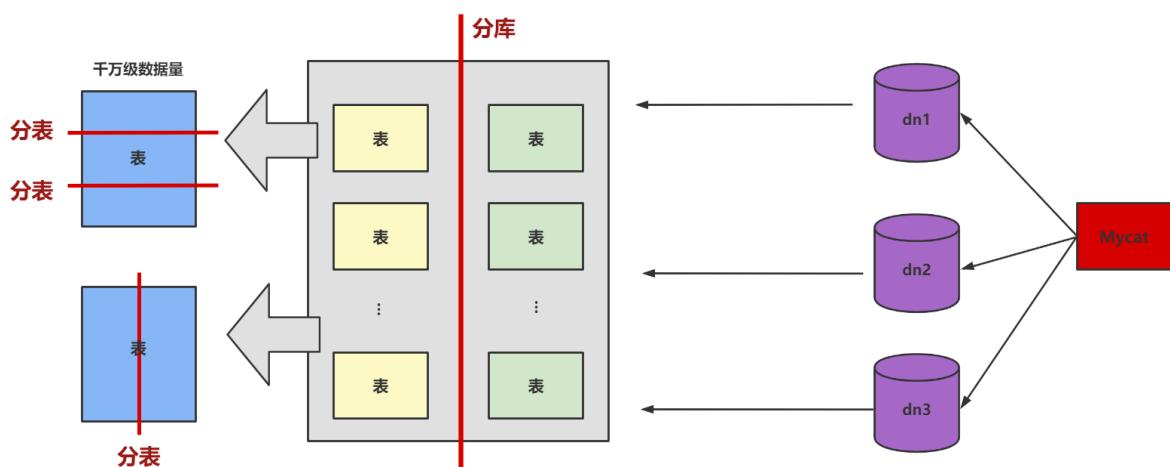
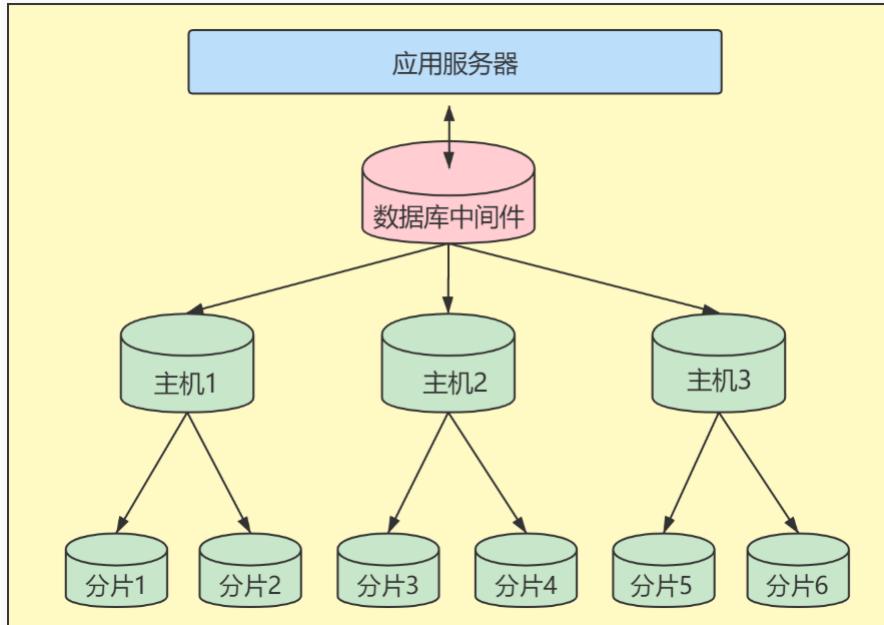
常用的键值存储数据库有 Redis 和 Memcached，它们都可以将数据存放到内存中。

第6步：库级优化

1、读写分离



2、数据分片



但需要注意的是，分拆在提升数据库性能的同时，也会增加维护和使用成本。

2. 优化MySQL服务器

2.1 优化服务器硬件

服务器的硬件性能直接决定着MySQL数据库的性能。硬件的性能瓶颈直接决定MySQL数据库的运行速度和效率。针对性能瓶颈提高硬件配置，可以提高MySQL数据库查询、更新的速度。（1）**配置较大的内存**（2）**配置高速磁盘系统**（3）**合理分布磁盘I/O**（4）**配置多处理器**

2.2 优化MySQL的参数

- **innodb_buffer_pool_size**：这个参数是MySQL数据库最重要的参数之一，表示InnoDB类型的**表**和索引的最大缓存。它不仅仅缓存**索引数据**，还会缓存**表的数据**。这个值越大，查询的速度就会越快。但是这个值太大会影响操作系统的性能。
- **key_buffer_size**：表示**索引缓冲区的大小**。索引缓冲区是所有的**线程共享**。增加索引缓冲区可以得到更好处理的索引（对所有读和多重写）。当然，这个值不是越大越好，它的大小取决于内存的大小。如果这个值太大，就会导致操作系统频繁换页，也会降低系统性能。对于内存有**4GB**左右的服务器该参数可设置为**256M**或**384M**。

- **table_cache** : 表示 同时打开的表的个数。这个值越大，能够同时打开的表的个数越多。物理内存越大，设置就越大。默认为2402，调到512-1024最佳。这个值不是越大越好，因为同时打开的表太多会影响操作系统的性能。
- **query_cache_size** : 表示 查询缓冲区的大小。可以通过在MySQL控制台观察，如果Qcache_lowmem_prunes的值非常大，则表明经常出现缓冲不够的情况，就要增加Query_cache_size的值；如果Qcache_hits的值非常大，则表明查询缓冲使用非常频繁，如果该值较小反而会影响效率，那么可以考虑不用查询缓存；Qcache_free_blocks，如果该值非常大，则表明缓冲区中碎片很多。MySQL8.0之后失效。该参数需要和query_cache_type配合使用。
 - 当query_cache_type=1时，所有的查询都将使用查询缓存区，除非在查询语句中指定SQL_NO_CACHE，如SELECT SQL_NO_CACHE * FROM tbl_name。
 - 当query_cache_type=2时，只有在查询语句中使用 SQL_CACHE 关键字，查询才会使用查询缓存区。使用查询缓存区可以提高查询的速度，这种方式只适用于修改操作少且经常执行相同的查询操作的情况。
- **sort_buffer_size** : 表示每个 需要进行排序的线程分配的缓冲区的大小。增加这个参数的值可以提高 ORDER BY 或 GROUP BY 操作的速度。默认数值是2 097 144字节（约2MB）。对于内存在4GB左右的服务器推荐设置为6-8M，如果有100个连接，那么实际分配的总共排序缓冲区大小为 $100 \times 6 = 600\text{MB}$ 。
- **join_buffer_size = 8M** : 表示 联合查询操作所能使用的缓冲区大小，和sort_buffer_size一样，该参数对应的分配内存也是每个连接独享。
- **read_buffer_size** : 表示 每个线程连续扫描时为扫描的每个表分配的缓冲区的大小（字节）。当线程从表中连续读取记录时需要用到这个缓冲区。SET SESSION read_buffer_size=n可以临时设置该参数的值。默认为64K，可以设置为4M。
- **innodb_flush_log_at_trx_commit** : 表示 何时将缓冲区的数据写入日志文件，并且将日志文件写入磁盘中。该参数对于InnoDB引擎非常重要。该参数有3个值，分别为0、1和2。该参数的默认值为1。
 - 值为 0 时，表示 每秒1次 的频率将数据写入日志文件并将日志文件写入磁盘。每个事务的commit并不会触发前面的任何操作。该模式速度最快，但不太安全，mysqld进程的崩溃会导致上一秒钟所有事务数据的丢失。
 - 值为 1 时，表示 每次提交事务时 将数据写入日志文件并将日志文件写入磁盘进行同步。该模式是最安全的，但也是最慢的一种方式。因为每次事务提交或事务外的指令都需要把日志写入(flush) 硬盘。
 - 值为 2 时，表示 每次提交事务时 将数据写入日志文件， 每隔1秒 将日志文件写入磁盘。该模式速度较快，也比0安全，只有在操作系统崩溃或者系统断电的情况下，上一秒钟所有事务数据才可能丢失。
- **innodb_log_buffer_size** : 这是 InnoDB 存储引擎的 事务日志所使用的缓冲区。为了提高性能，也是先将信息写入 Innodb Log Buffer 中，当满足 innodb_flush_log_trx_commit 参数所设置的相应条件（或者日志缓冲区写满）之后，才会将日志写到文件（或者同步到磁盘）中。
- **max_connections** : 表示 允许连接到MySQL数据库的最大数量，默认值是 151。如果状态变量connection_errors_max_connections 不为零，并且一直增长，则说明不断有连接请求因数据库连接数已达到允许最大值而失败，这是可以考虑增大max_connections 的值。在Linux 平台上，性能好的服务器，支持 500-1000 个连接不是难事，需要根据服务器性能进行评估设定。这个连接数 不是越大越好，因为这些连接会浪费内存的资源。过多的连接可能会导致MySQL服务器僵死。

- **back_log** : 用于控制MySQL监听TCP端口时设置的积压请求栈大小。如果MySQL的连接数达到max_connections时，新来的请求将会被存在堆栈中，以等待某一连接释放资源，该堆栈的数量即back_log，如果等待连接的数量超过back_log，将不被授予连接资源，将会报错。5.6.6版本之前默认值为50，之后的版本默认为 $50 + (\max_connections / 5)$ ，对于Linux系统推荐设置为小于512的整数，但最大不超过900。

如果需要数据库在较短的时间内处理大量连接请求，可以考虑适当增大back_log的值。

- **thread_cache_size** : 线程池缓存线程数量的大小，当客户端断开连接后将当前线程缓存起来，当接到新的连接请求时快速响应无需创建新的线程。这尤其对那些使用短连接的应用程序来说可以极大的提高创建连接的效率。那么为了提高性能可以增大该参数的值。默认为60，可以设置为120。

可以通过如下几个MySQL状态值来适当调整线程池的大小：

```
mysql> show global status like 'Thread%';
+-----+-----+
| Variable_name      | Value |
+-----+-----+
| Threads_cached     | 2      |
| Threads_connected  | 1      |
| Threads_created    | 3      |
| Threads_running    | 2      |
+-----+-----+
4 rows in set (0.01 sec)
```

当Threads_cached越来越少，但Threads_connected始终不降，且Threads_created持续升高，可适当增加thread_cache_size的大小。

- **wait_timeout** : 指定一个请求的最大连接时间，对于4GB左右内存的服务器可以设置为5-10。
- **interactive_timeout** : 表示服务器在关闭连接前等待行动的秒数。

这里给出一份my.cnf的参考配置：

```
[mysqld]
port = 3306 serverid = 1 socket = /tmp/mysql.sock skip-locking #避免MySQL的外部锁定，减少出错几率增强稳定性。 skip-name-resolve #禁止MySQL对外部连接进行DNS解析，使用这一选项可以消除MySQL进行DNS解析的时间。但需要注意，如果开启该选项，则所有远程主机连接授权都要使用IP地址方式，否则MySQL将无法正常处理连接请求！ back_log = 384
key_buffer_size = 256M max_allowed_packet = 4M thread_stack = 256K
table_cache = 128K sort_buffer_size = 6M read_buffer_size = 4M
read_rnd_buffer_size=16M join_buffer_size = 8M myisam_sort_buffer_size =
64M table_cache = 512 thread_cache_size = 64 query_cache_size = 64M
tmp_table_size = 256M max_connections = 768 max_connect_errors = 10000000
wait_timeout = 10 thread_concurrency = 8 #该参数取值为服务器逻辑CPU数量*2，在本例中，服务器有2颗物理CPU，而每颗物理CPU又支持H.T超线程，所以实际取值为4*2=8 skip-
networking #开启该选项可以彻底关闭MySQL的TCP/IP连接方式，如果WEB服务器是以远程连接的方式访问MySQL数据库服务器则不要开启该选项！否则将无法正常连接！ table_cache=1024
innodb_additional_mem_pool_size=4M #默认为2M innodb_flush_log_at_trx_commit=1
innodb_log_buffer_size=2M #默认为1M innodb_thread_concurrency=8 #你的服务器CPU有几个就设置为几。建议用默认一般为8 tmp_table_size=64M #默认为16M，调到64-256最佳
thread_cache_size=120 query_cache_size=32M
```

很多情况还需要具体情况具体分析！

案例分析：见视频

(https://www.bilibili.com/video/BV1iq4y1u7vj?from=search&seid=4297501441472622157&spm_id_from=333.337.0.0)

3. 优化数据库结构

3.1 拆分表：冷热数据分离

举例1： 会员members表 存储会员登录认证信息，该表中有很多字段，如id、姓名、密码、地址、电话、个人描述字段。其中地址、电话、个人描述等字段并不常用，可以将这些不常用的字段分解出另一个表。将这个表取名叫members_detail，表中有member_id、address、telephone、description等字段。这样就把会员表分成了两个表，分别为 members表 和 members_detail表。

创建这两个表的SQL语句如下：

```
CREATE TABLE members (
    id int(11) NOT NULL AUTO_INCREMENT,
    username varchar(50) DEFAULT NULL,
    password varchar(50) DEFAULT NULL,
    last_login_time datetime DEFAULT NULL,
    last_login_ip varchar(100) DEFAULT NULL,
    PRIMARY KEY(Id)
);
CREATE TABLE members_detail (
    Member_id int(11) NOT NULL DEFAULT 0,
    address varchar(255) DEFAULT NULL,
    telephone varchar(255) DEFAULT NULL,
    description text
);
```

如果需要查询会员的基本信息或详细信息，那么可以用会员的id来查询。如果需要将会员的基本信息和详细信息同时显示，那么可以将members表和members_detail表进行联合查询，查询语句如下：

```
SELECT * FROM members LEFT JOIN members_detail on members.id =
members_detail.member_id;
```

通过这种分解可以提高表的查询效率。对于字段很多且有些字段使用不频繁的表，可以通过这种分解的方式来优化数据库的性能。

3.2 增加中间表

举例1： 学生信息表 和 班级表 的SQL语句如下：

```
CREATE TABLE `class` (
    `id` INT(11) NOT NULL AUTO_INCREMENT,
    `className` VARCHAR(30) DEFAULT NULL,
    `address` VARCHAR(40) DEFAULT NULL,
    `monitor` INT NULL ,
    PRIMARY KEY (`id`)
) ENGINE=INNODB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;

CREATE TABLE `student` (
    `id` INT(11) NOT NULL AUTO_INCREMENT,
    `stuno` INT NOT NULL ,
```

```
`name` VARCHAR(20) DEFAULT NULL,  
`age` INT(3) DEFAULT NULL,  
`classId` INT(11) DEFAULT NULL,  
PRIMARY KEY (`id`)  
) ENGINE=INNODB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

现在有一个模块需要经常查询带有学生名称（name）、学生所在班级名称（className）、学生班级班长（monitor）的学生信息。根据这种情况可以创建一个 `temp_student` 表。`temp_student` 表中存储学生名称（stu_name）、学生所在班级名称（className）和学生班级班长（monitor）信息。创建表的语句如下：

```
CREATE TABLE `temp_student` (  
`id` INT(11) NOT NULL AUTO_INCREMENT,  
`stu_name` INT NOT NULL ,  
`className` VARCHAR(20) DEFAULT NULL,  
`monitor` INT(3) DEFAULT NULL,  
PRIMARY KEY (`id`)  
) ENGINE=INNODB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

接下来，从学生信息表和班级表中查询相关信息存储到临时表中：

```
insert into temp_student(stu_name,className,monitor)  
select s.name,c.className,c.monitor  
from student as s,class as c  
where s.classId = c.id
```

以后，可以直接从`temp_student`表中查询学生名称、班级名称和班级班长，而不用每次都进行联合查询。这样可以提高数据库的查询速度。

3.3 增加冗余字段

设计数据库表时应尽量遵循范式理论的规约，尽可能减少冗余字段，让数据库设计看起来精致、优雅。但是，合理地加入冗余字段可以提高查询速度。

表的规范化程度越高，表与表之间的关系就越多，需要连接查询的情况也就越多。尤其在数据量大，而且需要频繁进行连接的时候，为了提升效率，我们也可以考虑增加冗余字段来减少连接。

这部分内容在《第11章_数据库的设计规范》章节中 [反范式化小节](#) 中具体展开讲解了。这里省略。

3.4 优化数据类型

情况1：对整数类型数据进行优化。

遇到整数类型的字段可以用 `INT` 型。这样做的理由是，`INT` 型数据有足够大的取值范围，不用担心数据超出取值范围的问题。刚开始做项目的时候，首先要保证系统的稳定性，这样设计字段类型是可以的。但在数据量很大的时候，数据类型的定义，在很大程度上会影响到系统整体的执行效率。

对于 [非负型](#) 的数据（如自增ID、整型IP）来说，要优先使用无符号整型 `UNSIGNED` 来存储。因为无符号相对于有符号，同样的字节数，存储的数值范围更大。如`tinyint`有符号为-128-127，无符号为0-255，多出一倍的存储空间。

情况2：既可以使用文本类型也可以使用整数类型的字段，要选择使用整数类型。

跟文本类型数据相比，大整数往往占用 [更少的存储空间](#)，因此，在存取和比对的时候，可以占用更少的内存空间。所以，在二者皆可用的情况下，尽量使用整数类型，这样可以提高查询的效率。如：将IP地址转换成整型数据。

情况3：避免使用TEXT、BLOB数据类型

情况4：避免使用ENUM类型

情况5：使用TIMESTAMP存储时间

情况6：用DECIMAL代替FLOAT和DOUBLE存储精确浮点数

总之，遇到数据量大的项目时，一定要在充分了解业务需求的前提下，合理优化数据类型，这样才能充分发挥资源的效率，使系统达到最优。

3.5 优化插入记录的速度

1. MyISAM引擎的表：

① 禁用索引

② 禁用唯一性检查

③ 使用批量插入

```
insert into student values(1, 'zhangsan', 18, 1);
insert into student values(2, 'lisi', 17, 1);
insert into student values(3, 'wangwu', 17, 1);
insert into student values(4, 'zhaoliu', 19, 1);
```

使用一条INSERT语句插入多条记录的情形如下：

```
insert into student values
(1, 'zhangsan', 18, 1),
(2, 'lisi', 17, 1),
(3, 'wangwu', 17, 1),
(4, 'zhaoliu', 19, 1);
```

第2种情形的插入速度要比第1种情形快。

④ 使用LOAD DATA INFILE 批量导入

2. InnoDB引擎的表： ① 禁用唯一性检查

② 禁用外键检查

③ 禁止自动提交

3.6 使用非空约束

在设计字段的时候，如果业务允许，建议尽量使用非空约束

3.7 分析表、检查表与优化表

1. 分析表

MySQL中提供了ANALYZE TABLE语句分析表，ANALYZE TABLE语句的基本语法如下：

```
ANALYZE [LOCAL | NO_WRITE_TO_BINLOG] TABLE tbl_name[,tbl_name]...
```

默认的，MySQL服务会将ANALYZE TABLE语句写到binlog中，以便在主从架构中，从服务能够同步数据。可以添加参数LOCAL或者NO_WRITE_TO_BINLOG取消将语句写到binlog中。

使用ANALYZE TABLE分析表的过程中，数据库系统会自动对表加一个**只读锁**。在分析期间，只能读取表中的记录，不能更新和插入记录。ANALYZE TABLE语句能够分析InnoDB和MyISAM类型的表，但是不能作用于视图。

ANALYZE TABLE分析后的统计结果会反应到 **cardinality** 的值，该值统计了表中某一键所在的列不重复的值的个数。该值越接近表中的总行数，则在表连接查询或者索引查询时，就越优先被优化器选择使用。也就是索引列的cardinality的值与表中数据的总条数差距越大，即使查询的时候使用了该索引作为查询条件，存储引擎实际查询的时候使用的概率就越小。下面通过例子来验证下。cardinality可以通过 SHOW INDEX FROM 表名查看。

2. 检查表

MySQL中可以使用 **CHECK TABLE** 语句来检查表。CHECK TABLE语句能够检查InnoDB和MyISAM类型的表是否存在错误。CHECK TABLE语句在执行过程中也会给表加上 **只读锁**。

对于MyISAM类型的表，CHECK TABLE语句还会更新关键字统计数据。而且，CHECK TABLE也可以检查视图是否有错误，比如在视图定义中被引用的表已不存在。该语句的基本语法如下：

```
CHECK TABLE tbl_name [, tbl_name] ... [option] ...
option = {QUICK | FAST | MEDIUM | EXTENDED | CHANGED}
```

其中，tbl_name是表名；option参数有5个取值，分别是QUICK、FAST、MEDIUM、EXTENDED和CHANGED。各个选项的意义分别是：

- **QUICK**：不扫描行，不检查错误的连接。
- **FAST**：只检查没有被正确关闭的表。
- **CHANGED**：只检查上次检查后被更改的表和没有被正确关闭的表。
- **MEDIUM**：扫描行，以验证被删除的连接是有效的。也可以计算各行的关键字校验和，并使用计算出的校验和验证这一点。
- **EXTENDED**：对每行的所有关键字进行一个全面的关键字查找。这可以确保表是100%一致的，但是花的时间较长。

option只对MyISAM类型的表有效，对InnoDB类型的表无效。比如：

```
mysql> check table student;
+-----+-----+-----+-----+
| Table          | Op      | Msg_type | Msg_text |
+-----+-----+-----+-----+
| atguigudb1.student | check | status    | OK        |
+-----+-----+-----+-----+
1 row in set (1.84 sec)
```

该语句对于检查的表可能会产生多行信息。最后一行有一个状态的 **Msg_type** 值，**Msg_text** 通常为 **OK**。如果得到的不是 **OK**，通常要对其进行修复；是 **OK** 说明表已经是最新的了。表已经是最新的，意味着存储引擎对这张表不必进行检查。

3. 优化表

方式1：OPTIMIZE TABLE

MySQL中使用 **OPTIMIZE TABLE** 语句来优化表。但是，OPTIMIZE TABLE语句只能优化表中的 **VARCHAR**、**BLOB** 或 **TEXT** 类型的字段。一个表使用了这些字段的数据类型，若已经 **删除** 了表的大部分数据，或者已经对含有可变长度行的表（含有VARCHAR、BLOB或TEXT列的表）进行了很多 **更新**，则应使用OPTIMIZE TABLE来重新利用未使用的空间，并整理数据文件的 **碎片**。

OPTIMIZE TABLE语句对InnoDB和MyISAM类型的表都有效。该语句在执行过程中也会给表加上 **只读锁**。

OPTIMIZE TABLE语句的基本语法如下：

```
OPTIMIZE [LOCAL | NO_WRITE_TO_BINLOG] TABLE tbl_name [, tbl_name] ...
```

LOCAL | NO_WRITE_TO_BINLOG关键字的意义和分析表相同，都是指定不写入二进制日志。

```
mysql> optimize table student;
+-----+-----+-----+
| Table | Op   | Msg_type | Msg_text
+-----+-----+-----+
| atguigudb1.student | optimize | note    | Table does not support optimize, doing recreate + analyze instead |
| atguigudb1.student | optimize | status   | OK
+-----+-----+-----+
2 rows in set (14.44 sec)
```

执行完毕，Msg_text显示

```
'numysql.SYS_APP_USER', 'optimize', 'note', 'Table does not support optimize, doing recreate + analyze instead'
```

原因是我服务器上的MySQL是InnoDB存储引擎。

到底优化了没有呢？看官网！

<https://dev.mysql.com/doc/refman/8.0/en/optimize-table.html>

在MyISAM中，是先分析这张表，然后会整理相关的MySQL datafile，之后回收未使用的空间；在InnoDB中，回收空间是简单通过Alter table进行整理空间。在优化期间，MySQL会创建一个临时表，优化完成之后会删除原始表，然后会将临时表rename成为原始表。

说明：在多数的设置中，根本不需要运行OPTIMIZE TABLE。即使对可变长度的行进行了大量的更新，也不需要经常运行，每周一次或每月一次即可，并且只需要对特定的表运行。

3.8 小结

上述这些方法都是有利有弊的。比如：

- 修改数据类型，节省存储空间的同时，你要考虑到数据不能超过取值范围；
- 增加冗余字段的时候，不要忘了确保数据一致性；
- 把大表拆分，也意味着你的查询会增加新的连接，从而增加额外的开销和运维的成本。

因此，你一定要结合实际的业务需求进行权衡。

4. 大表优化

4.1 限定查询的范围

禁止不带任何限制数据范围条件的查询语句。比如：我们当用户在查询订单历史的时候，我们可以控制在一个月的范围内；

4.2 读/写分离

经典的数据库拆分方案，主库负责写，从库负责读。

- 一主一从模式：

一主一从模式

Java程序

作为数据源访问

Mycat

写请求

Master

读请求

Slave

MySQL主从复制

- 双主双从模式：

双主双从模式

Java程序

作为数据源访问

Mycat

写请求

Master

Master

读请求

Slave

Slave

MySQL
主从复制

MySQL
主从复制

4.3 垂直拆分

当数据量级达到 **千万级** 以上时，有时候我们需要把一个数据库切成多份，放到不同的数据库服务器上，减少对单一数据库服务器的访问压力。

应用服务器

数据库中间件

主机1

主机2

主机3

分片1

分片2

分片3

分片4

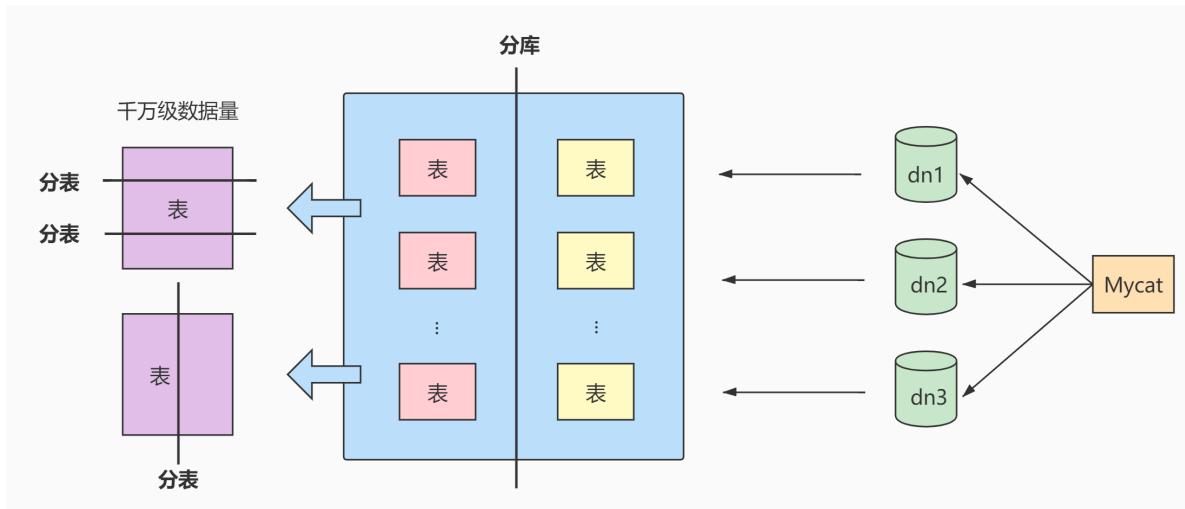
分片5

分片6

垂直拆分的优点：可以使得列数据变小，在查询时减少读取的Block数，减少I/O次数。此外，垂直分区可以简化表的结构，易于维护。

垂直拆分的缺点：主键会出现冗余，需要管理冗余列，并会引起JOIN操作。此外，垂直拆分会让事务变得更加复杂。

4.4 水平拆分



下面补充一下数据库分片的两种常见方案：

- **客户端代理：** 分片逻辑在应用端，封装在jar包中，通过修改或者封装JDBC层来实现。当当网的 Sharding-JDBC、阿里的TDDL是两种比较常用的实现。
- **中间件代理：** 在应用和数据中间加了一个代理层。分片逻辑统一维护在中间件服务中。我们现在谈的 Mycat、360的Atlas、网易的DDB等等都是这种架构的实现。

5. 其它调优策略

5.1 服务器语句超时处理

在MySQL 8.0中可以设置 **服务器语句超时的限制**，单位可以达到 **毫秒级别**。当中断的执行语句超过设置的毫秒数后，服务器将终止查询影响不大的事务或连接，然后将错误报给客户端。

设置服务器语句超时的限制，可以通过设置系统变量 **MAX_EXECUTION_TIME** 来实现。默认情况下，**MAX_EXECUTION_TIME**的值为0，代表没有时间限制。例如：

```
SET GLOBAL MAX_EXECUTION_TIME=2000;
```

```
SET SESSION MAX_EXECUTION_TIME=2000; #指定该会话中SELECT语句的超时时间
```

5.2 创建全局通用表空间

5.3 MySQL 8.0新特性：隐藏索引对调优的帮助

第13章_事务基础知识

讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

1. 数据库事务概述

1.1 存储引擎支持情况

`SHOW ENGINES` 命令来查看当前 MySQL 支持的存储引擎都有哪些，以及这些存储引擎是否支持事务。

```
mysql> show engines;
+-----+-----+-----+-----+-----+-----+
| Engine | Support | Comment          | Transactions | XA    | Savepoints |
+-----+-----+-----+-----+-----+-----+
| FEDERATED | NO    | Federated MySQL storage engine | NULL        | NULL  | NULL      |
| MEMORY    | YES   | Hash based, stored in memory, useful for temporary tables | NO         | NO    | NO       |
| InnoDB     | DEFAULT | Supports transactions, row-level locking, and foreign keys | YES        | YES   | YES      |
| PERFORMANCE_SCHEMA | YES | Performance Schema | NO         | NO    | NO       |
| MyISAM     | YES   | MyISAM storage engine | NO         | NO    | NO       |
| MRG_MYISAM | YES   | Collection of identical MyISAM tables | NO         | NO    | NO       |
| BLACKHOLE  | YES   | /dev/null storage engine (anything you write to it disappears) | NO         | NO    | NO       |
| CSV        | YES   | CSV storage engine | NO         | NO    | NO       |
| ARCHIVE    | YES   | Archive storage engine | NO         | NO    | NO       |
+-----+-----+-----+-----+-----+-----+
9 rows in set (0.00 sec)
```

能看出在 MySQL 中，只有 InnoDB 是支持事务的。

1.2 基本概念

事务：一组逻辑操作单元，使数据从一种状态变换到另一种状态。

事务处理的原则：保证所有事务都作为一个工作单元来执行，即使出现了故障，都不能改变这种执行方式。当在一个事务中执行多个操作时，要么所有的事务都被提交(`commit`)，那么这些修改就永久地保存下来；要么数据库管理系统将放弃所作的所有修改，整个事务回滚(`rollback`)到最初状态。

1.3 事务的ACID特性

- **原子性 (atomicity) :**

原子性是指事务是一个不可分割的工作单位，要么全部提交，要么全部失败回滚。

- **一致性 (consistency) :**

(国内很多网站上对一致性的阐述有误，具体你可以参考 Wikipedia 对 `Consistency` 的阐述)

根据定义，一致性是指事务执行前后，数据从一个合法性状态变换到另外一个合法性状态。这种状态是语义上的而不是语法上的，跟具体的业务有关。

那什么是合法的数据状态呢？满足预定的约束的状态就叫做合法的状态。通俗一点，这状态是由你自己来定义的（比如满足现实世界中的约束）。满足这个状态，数据就是一致的，不满足这个状态，数据就是不一致的！如果事务中的某个操作失败了，系统就会自动撤销当前正在执行的事务，返回到事务操作之前的状态。

- **隔离型 (isolation) :**

事务的隔离性是指一个事务的执行 **不能被其他事务干扰**，即一个事务内部的操作及使用的数据对 **并发** 的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。

如果无法保证隔离性会怎么样？假设A账户有200元，B账户0元。A账户往B账户转账两次，每次金额为50元，分别在两个事务中执行。如果无法保证隔离性，会出现下面的情形：

```
UPDATE accounts SET money = money - 50 WHERE NAME = 'AA';
```

```
UPDATE accounts SET money = money + 50 WHERE NAME = 'BB';
```



- **持久性 (durability) :**

持久性是指一个事务一旦被提交，它对数据库中数据的改变就是 **永久性的**，接下来的其他操作和数据库故障不应该对其有任何影响。

持久性是通过 **事务日志** 来保证的。日志包括了 **重做日志** 和 **回滚日志**。当我们通过事务对数据进行修改的时候，首先会将数据库的变化信息记录到重做日志中，然后再对数据库中对应的行进行修改。这样做的好处是，即使数据库系统崩溃，数据库重启后也能找到没有更新到数据库系统中的重做日志，重新执行，从而使事务具有持久性。

1.4 事务的状态

我们现在知道 **事务** 是一个抽象的概念，它其实对应着一个或多个数据库操作，MySQL根据这些操作所执行的不同阶段把 **事务** 大致划分成几个状态：

- **活动的 (active)**

事务对应的数据操作正在执行过程中时，我们就说该事务处在 **活动的** 状态。

- **部分提交的 (partially committed)**

当事务中的最后一个操作执行完成，但由于操作都在内存中执行，所造成的影响并 **没有刷新到磁盘** 时，我们就说该事务处在 **部分提交的** 状态。

- **失败的 (failed)**

当事务处在 **活动的** 或者 **部分提交的** 状态时，可能遇到了某些错误（数据库自身的错误、操作系统错误或者直接断电等）而无法继续执行，或者人为的停止当前事务的执行，我们就说该事务处在 **失败的** 状态。

- **中止的 (aborted)**

如果事务执行了一部分而变为 **失败的** 状态，那么就需要把已经修改的数据还原到事务执行前的状态。换句话说，就是要撤销失败事务对当前数据库造成的影响。我们把这个撤销的过程称之为 **回滚**。当 **回滚** 操作执行完毕时，也就是数据库恢复到了执行事务之前的状态，我们就说该事务处在了 **中止的** 状态。

举例：

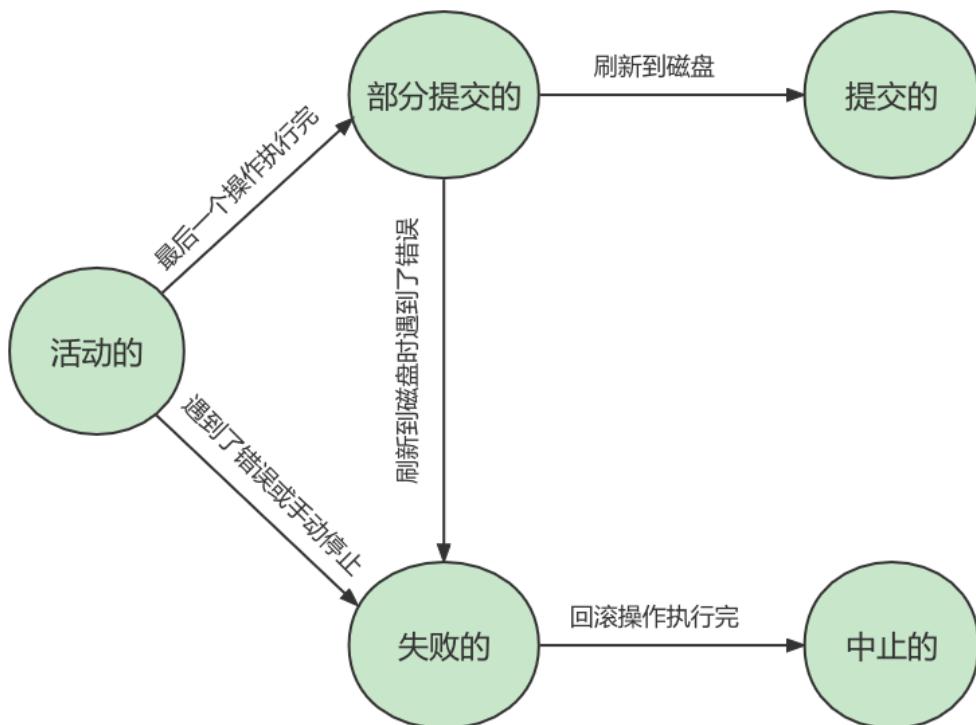
```
UPDATE accounts SET money = money - 50 WHERE NAME = 'AA';
```

```
UPDATE accounts SET money = money + 50 WHERE NAME = 'BB';
```

- **提交的 (committed)**

当一个处在 **部分提交的** 状态的事务将修改过的数据都 **同步到磁盘** 上之后，我们就可以说该事务处在了 **提交的** 状态。

一个基本的状态转换图如下所示：



2. 如何使用事务

使用事务有两种方式，分别为 **显式事务** 和 **隐式事务**。

2.1 显式事务

步骤1： `START TRANSACTION` 或者 `BEGIN`，作用是显式开启一个事务。

```
mysql> BEGIN;
#或者
mysql> START TRANSACTION;
```

`START TRANSACTION` 语句相较于 `BEGIN` 特别之处在于，后边能跟随几个 **修饰符**：

① **READ ONLY**：标识当前事务是一个 **只读事务**，也就是属于该事务的数据库操作只能读取数据，而不能修改数据。

② **READ WRITE**：标识当前事务是一个 **读写事务**，也就是属于该事务的数据库操作既可以读取数据，也可以修改数据。

③ **WITH CONSISTENT SNAPSHOT**：启动一致性读。

步骤2：一系列事务中的操作（主要是DML，不含DDL）

步骤3：提交事务或中止事务（即回滚事务）

```
# 提交事务。当提交事务后，对数据库的修改是永久性的。  
mysql> COMMIT;
```

```
# 回滚事务。即撤销正在进行的所有没有提交的修改  
mysql> ROLLBACK;  
  
# 将事务回滚到某个保存点。  
mysql> ROLLBACK TO [SAVEPOINT]
```

2.2 隐式事务

MySQL中有一个系统变量 **autocommit**：

```
mysql> SHOW VARIABLES LIKE 'autocommit';  
+-----+-----+  
| Variable_name | Value |  
+-----+-----+  
| autocommit     | ON    |  
+-----+-----+  
1 row in set (0.01 sec)
```

当然，如果我们想关闭这种 **自动提交** 的功能，可以使用下边两种方法之一：

- 显式的使用 **START TRANSACTION** 或者 **BEGIN** 语句开启一个事务。这样在本次事务提交或者回滚前会暂时关闭掉自动提交的功能。
- 把系统变量 **autocommit** 的值设置为 **OFF**，就像这样：

```
SET autocommit = OFF;  
#或  
SET autocommit = 0;
```

2.3 隐式提交数据的情况

- 数据定义语言 (Data definition language, 缩写为：DDL)**

- 隐式使用或修改mysql数据库中的表**

- 事务控制或关于锁定的语句**

① 当我们在一个事务还没提交或者回滚时就又使用 **START TRANSACTION** 或者 **BEGIN** 语句开启了另一个事务时，会 **隐式的提交** 上一个事务。即：

② 当前的 **autocommit** 系统变量的值为 **OFF**，我们手动把它调为 **ON** 时，也会 **隐式的提交** 前边语句所属的事务。

③ 使用 `LOCK TABLES`、`UNLOCK TABLES` 等关于锁定的语句也会 隐式的提交 前边语句所属的事务。

- 加载数据的语句
- 关于MySQL复制的一些语句
- 其它的一些语句

2.4 使用举例1：提交与回滚

我们看下在 MySQL 的默认状态下，下面这个事务最后的处理结果是什么。

情况1：

```
CREATE TABLE user(name varchar(20), PRIMARY KEY (name)) ENGINE=InnoDB;

BEGIN;
INSERT INTO user SELECT '张三';
COMMIT;

BEGIN;
INSERT INTO user SELECT '李四';
INSERT INTO user SELECT '李四';
ROLLBACK;

SELECT * FROM user;
```

运行结果 (1 行数据)：

```
mysql> commit;
Query OK, 0 rows affected (0.00 秒)

mysql> BEGIN;
Query OK, 0 rows affected (0.00 秒)

mysql> INSERT INTO user SELECT '李四';
Query OK, 1 rows affected (0.00 秒)

mysql> INSERT INTO user SELECT '李四';
Duplicate entry '李四' for key 'user.PRIMARY'
mysql> ROLLBACK;
Query OK, 0 rows affected (0.01 秒)

mysql> select * from user;
+-----+
| name   |
+-----+
| 张三   |
+-----+
1 行于数据集 (0.01 秒)
```

情况2：

```
CREATE TABLE user (name varchar(20), PRIMARY KEY (name)) ENGINE=InnoDB;

BEGIN;
INSERT INTO user SELECT '张三';
COMMIT;

INSERT INTO user SELECT '李四';
INSERT INTO user SELECT '李四';
ROLLBACK;
```

运行结果 (2 行数据) :

```
mysql> SELECT * FROM user;
+-----+
| name |
+-----+
| 张三 |
| 李四 |
+-----+
2 行于数据集 (0.01 秒)
```

情况3:

```
CREATE TABLE user(name varchar(255), PRIMARY KEY (name)) ENGINE=InnoDB;

SET @@completion_type = 1;
BEGIN;
INSERT INTO user SELECT '张三';
COMMIT;

INSERT INTO user SELECT '李四';
INSERT INTO user SELECT '李四';
ROLLBACK;

SELECT * FROM user;
```

运行结果 (1 行数据) :

```
mysql> SELECT * FROM user;
+-----+
| name |
+-----+
| 张三 |
+-----+
1 行于数据集 (0.01 秒)
```

当我们设置 autocommit=0 时，不论是否采用 START TRANSACTION 或者 BEGIN 的方式来开启事务，都需要用 COMMIT 进行提交，让事务生效，使用 ROLLBACK 对事务进行回滚。

当我们设置 autocommit=1 时，每条 SQL 语句都会自动进行提交。不过这时，如果你采用 START TRANSACTION 或者 BEGIN 的方式来显式地开启事务，那么这个事务只有在 COMMIT 时才会生效，在 ROLLBACK 时才会回滚。

2.5 使用举例2：测试不支持事务的engine

见视频讲解 (https://www.bilibili.com/video/BV1iq4y1u7vj?from=search&seid=4297501441472622157&spm_id_from=333.337.0.0)

2.6 使用举例3：SAVEPOINT

见视频讲解 (https://www.bilibili.com/video/BV1iq4y1u7vj?from=search&seid=4297501441472622157&spm_id_from=333.337.0.0)

3. 事务隔离级别

MySQL是一个 **客户端 / 服务器** 架构的软件，对于同一个服务器来说，可以有若干个客户端与之连接，每个客户端与服务器连接上之后，就可以称为一个会话（**Session**）。每个客户端都可以在自己的会话中向服务器发出请求语句，一个请求语句可能是某个事务的一部分，也就是对于服务器来说可能同时处理多个事务。事务有 **隔离性** 的特性，理论上在某个事务 对某个数据进行访问 时，其他事务应该进行 **排队**，当该事务提交之后，其他事务才可以继续访问这个数据。但是这样对 **性能影响太大**，我们既想保持事务的隔离性，又想让服务器在处理访问同一数据的多个事务时 **性能尽量高些**，那就看二者如何权衡取舍了。

3.1 数据准备

我们需要创建一个表：

```
CREATE TABLE student (
    studentno INT,
    name VARCHAR(20),
    class varchar(20),
    PRIMARY KEY (studentno)
) Engine=InnoDB CHARSET=utf8;
```

然后向这个表里插入一条数据：

```
INSERT INTO student VALUES(1, '小谷', '1班');
```

现在表里的数据就是这样的：

```
mysql> select * from student;
+-----+-----+-----+
| studentno | name   | class  |
+-----+-----+-----+
|       1   | 小谷   | 1班    |
+-----+-----+-----+
1 row in set (0.00 sec)
```

3.2 数据并发问题

针对事务的隔离性和并发性，我们怎么做取舍呢？先看一下访问相同数据的事务在 **不保证串行执行**（也就是执行完一个再执行另一个）的情况下可能会出现哪些问题：

1. 脏写（Dirty Write）

对于两个事务 Session A、Session B，如果事务Session A **修改了** 另一个 **未提交** 事务Session B **修改过** 的数据，那就意味着发生了 **脏写**

2. 脏读 (Dirty Read)

对于两个事务 Session A、Session B，Session A 读取 了已经被 Session B 更新 但还 没有被提交 的字段。之后若 Session B 回滚，Session A 读取 的内容就是 临时且无效 的。

Session A和Session B各开启了一个事务，Session B中的事务先将studentno列为1的记录的name列更新为'张三'，然后Session A中的事务再去查询这条studentno为1的记录，如果读到列name的值为'张三'，而Session B中的事务稍后进行了回滚，那么Session A中的事务相当于读到了一个不存在的数据，这种现象就称之为 脏读 。

3. 不可重复读 (Non-Repeatable Read)

对于两个事务Session A、Session B，Session A 读取 了一个字段，然后 Session B 更新 了该字段。之后 Session A 再次读取 同一个字段， 值就不同 了。那就意味着发生了不可重复读。

我们在Session B中提交了几个 隐式事务 （注意是隐式事务，意味着语句结束事务就提交了），这些事务都修改了studentno列为1的记录的列name的值，每次事务提交之后，如果Session A中的事务都可以查看到最新的值，这种现象也被称之为 不可重复读 。

4. 幻读 (Phantom)

对于两个事务Session A、Session B，Session A 从一个表中 读取 了一个字段，然后 Session B 在该表中 插入 了一些新的行。之后，如果 Session A 再次读取 同一个表，就会多出几行。那就意味着发生了幻读。

Session A中的事务先根据条件 studentno > 0这个条件查询表student，得到了name列值为'张三'的记录；之后Session B中提交了一个 隐式事务 ，该事务向表student中插入了一条新记录；之后Session A中的事务再根据相同的条件 studentno > 0查询表student，得到的结果集中包含Session B中的事务新插入的那条记录，这种现象也被称之为 幻读 。我们把新插入的那些记录称之为 幻影记录 。

3.3 SQL中的四种隔离级别

上面介绍了几种并发事务执行过程中可能遇到的一些问题，这些问题有轻重缓急之分，我们给这些问题按照严重性来排一下序：

脏写 > 脏读 > 不可重复读 > 幻读

我们愿意舍弃一部分隔离性来换取一部分性能在这里就体现在：设立一些隔离级别，隔离级别越低，并发问题发生的就越多。SQL标准 中设立了4个 隔离级别：

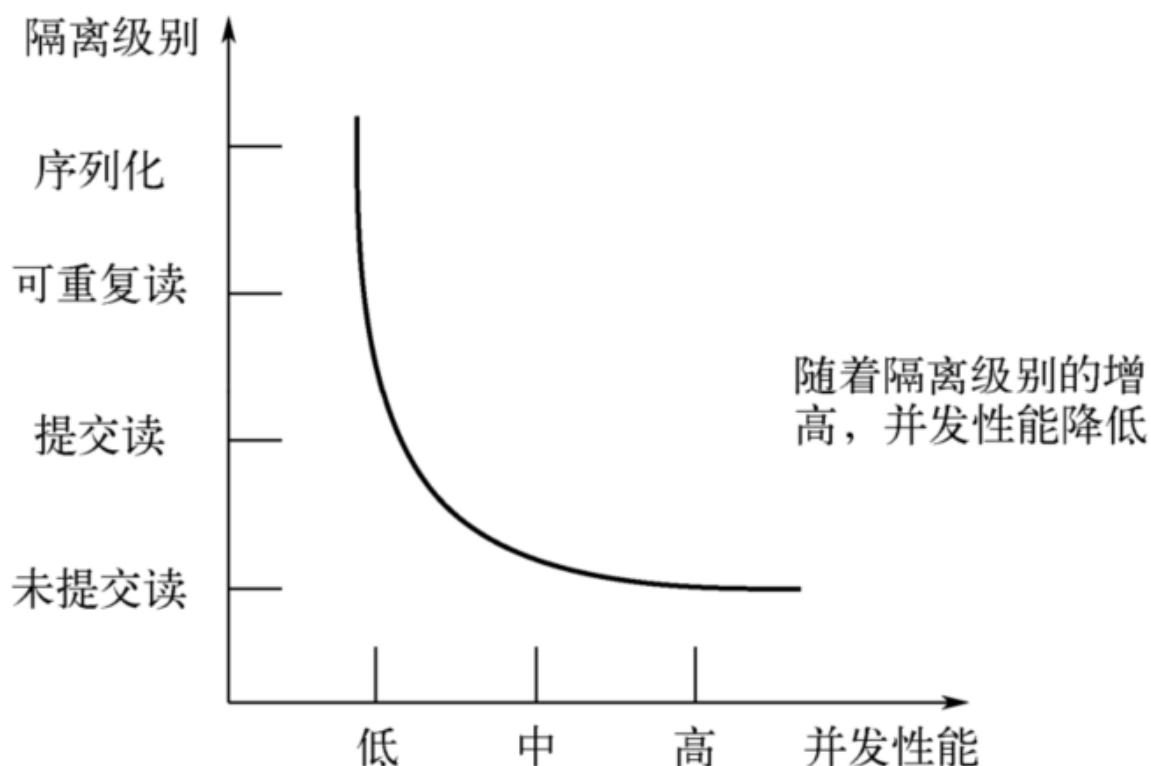
- **READ UNCOMMITTED**：读未提交，在该隔离级别，所有事务都可以看到其他未提交事务的执行结果。不能避免脏读、不可重复读、幻读。
- **READ COMMITTED**：读已提交，它满足了隔离的简单定义：一个事务只能看见已经提交事务所做的改变。这是大多数数据库系统的默认隔离级别（但不是MySQL默认的）。可以避免脏读，但不可重复读、幻读问题仍然存在。
- **REPEATABLE READ**：可重复读，事务A在读到一条数据之后，此时事务B对该数据进行了修改并提交，那么事务A再读该数据，读到的还是原来的内容。可以避免脏读、不可重复读，但幻读问题仍然存在。这是MySQL的默认隔离级别。
- **SERIALIZABLE**：可串行化，确保事务可以从一个表中读取相同的行。在这个事务持续期间，禁止其他事务对该表执行插入、更新和删除操作。所有的并发问题都可以避免，但性能十分低下。能避免脏读、不可重复读和幻读。

SQL标准 中规定，针对不同的隔离级别，并发事务可以发生不同严重程度的问题，具体情况如下：

隔离级别	脏读可能性	不可重复读可能性	幻读可能性	加锁读
READ UNCOMMITTED	Yes	Yes	Yes	No
READ COMMITTED	No	Yes	Yes	No
REPEATABLE READ	No	No	Yes	No
SERIALIZABLE	No	No	No	Yes

脏写怎么没涉及到？因为脏写这个问题太严重了，不论是哪种隔离级别，都不允许脏写的情况发生。

不同的隔离级别有不同的现象，并有不同的锁和并发机制，隔离级别越高，数据库的并发性能就越差，4种事务隔离级别与并发性能的关系如下：



3.4 MySQL支持的四种隔离级别

MySQL的默认隔离级别为REPEATABLE READ，我们可以手动修改一下事务的隔离级别。

```
# 查看隔离级别, MySQL 5.7.20的版本之前:
mysql> SHOW VARIABLES LIKE 'tx_isolation';
+-----+-----+
| Variable_name | Value      |
+-----+-----+
| tx_isolation | REPEATABLE-READ |
+-----+-----+
1 row in set (0.00 sec)

# MySQL 5.7.20版本之后, 引入transaction_isolation来替换tx_isolation

# 查看隔离级别, MySQL 5.7.20的版本之后:
mysql> SHOW VARIABLES LIKE 'transaction_isolation';
+-----+-----+
| Variable_name      | Value      |
+-----+-----+
```

```
| transaction_isolation | REPEATABLE-READ |
+-----+-----+
1 row in set (0.02 sec)
```

#或者不同MySQL版本中都可以使用的:

```
SELECT @@transaction_isolation;
```

3.5 如何设置事务的隔离级别

通过下面的语句修改事务的隔离级别:

```
SET [GLOBAL|SESSION] TRANSACTION ISOLATION LEVEL 隔离级别;
#其中, 隔离级别格式:
> READ UNCOMMITTED
> READ COMMITTED
> REPEATABLE READ
> SERIALIZABLE
```

或者:

```
SET [GLOBAL|SESSION] TRANSACTION_ISOLATION = '隔离级别'
#其中, 隔离级别格式:
> READ-UNCOMMITTED
> READ-COMMITTED
> REPEATABLE-READ
> SERIALIZABLE
```

关于设置时使用**GLOBAL**或**SESSION**的影响:

- 使用 **GLOBAL** 关键字 (在全局范围影响) :

```
SET GLOBAL TRANSACTION ISOLATION LEVEL SERIALIZABLE;
#或
SET GLOBAL TRANSACTION_ISOLATION = 'SERIALIZABLE';
```

则:

- 当前已经存在的会话无效
- 只对执行完该语句之后产生的会话起作用
- 使用 **SESSION** 关键字 (在会话范围影响) :

```
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
#或
SET SESSION TRANSACTION_ISOLATION = 'SERIALIZABLE';
```

则:

- 对当前会话的所有后续的事务有效
- 如果在事务之间执行, 则对后续的事务有效
- 该语句可以在已经开启的事务中间执行, 但不会影响当前正在执行的事务

小结:

数据库规定了多种事务隔离级别, 不同隔离级别对应不同的干扰程度, 隔离级别越高, 数据一致性就越好, 但并发性越弱。

3.6 不同隔离级别举例

演示1. 读未提交之脏读

设置隔离级别为未提交读：

时间	事务 1	事务 2
T1	set session transaction isolation level read uncommitted; start transaction;(开启事务) update account set balance = balance+100 where id=1; select * from account where id=1;#结果为 200	
T2		set session transaction isolation level read uncommitted; start transaction; select * from account where id=1;#查询余额结果为 200，脏读
T3	rollback;	
T4	commit;	
T5		select * from account where id=1;查询余额结果为 100

事务1和事务2的执行流程如下：

时间	事务 1	事务 2
T1	set session transaction isolation level read uncommitted; start transaction;(开启事务) update account set balance = balance-100 where id=1; update account set balance = balance+100 where id=2; select * from account where id=1;结果为 0	
T2		set session transaction isolation level read uncommitted; start transaction; select * from account where id=2; #结果为 100 update account set balance = balance-100 where id=2;#更新语句被阻塞
T3	rollback;	
T4		commit

演示2：读已提交

时间	事务 1	事务 2
T1	set session transaction isolation level read committed; start transaction;(开启事务) select * from account where id=2;#结果为 0	
T2		set session transaction isolation level read committed; start transaction; update account set balance = balance+100 where id=2; select * from account where id=2;#结果为 100
T3	select * from account where id=2;#结果仍然为 0，未发生脏读	
T4		commit;
T5	select * from account where id=2;#结果为 100 commit;	

设置隔离级别为可重复读，事务的执行流程如下：

时间	事务 1	事务 2
T1	set session transaction isolation level repeatable read; start transaction;(开启事务) select * from account where id=2;#结果为 0	
T2		set session transaction isolation level repeatable read; start transaction; update account set balance = balance+100 where id=2; select * from account where id=2;#结果为 100 commit;
T3		
T4	select * from account where id=2;#结果依然是 0 commit; select * from account where id=2; #结果为 100	

演示4：幻读

时间	事务 1	事务 2
T1	set session transaction isolation level repeatable read; start transaction;(开启事务) select count(*) from account where id=3;#结果为 0	
T2		set session transaction isolation level repeatable read; start transaction; insert into account (id,name,balance) values(3,"王五",0); commit;
T3	insert into account (id,name,balance) values(3,"王五",0); #主键重复，插入失败	
T4	select count(*) from account where id=3;#结果为 0	
T5	rollback;	

4. 事务的常见分类

从事务理论的角度来看，可以把事务分为以下几种类型：

- 扁平事务 (Flat Transactions)
- 带有保存点的扁平事务 (Flat Transactions with Savepoints)
- 链事务 (Chained Transactions)
- 嵌套事务 (Nested Transactions)
- 分布式事务 (Distributed Transactions)

第14章_MySQL事务日志

讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

事务有4种特性：原子性、一致性、隔离性和持久性。那么事务的四种特性到底是基于什么机制实现呢？

- 事务的隔离性由 锁机制 实现。
- 而事务的原子性、一致性和持久性由事务的 redo 日志和undo 日志来保证。
 - REDO LOG 称为 重做日志，提供再写入操作，恢复提交事务修改的页操作，用来保证事务的持久性。
 - UNDO LOG 称为 回滚日志，回滚行记录到某个特定版本，用来保证事务的原子性、一致性。

有的DBA或许会认为 UNDO 是 REDO 的逆过程，其实不然。

1. redo日志

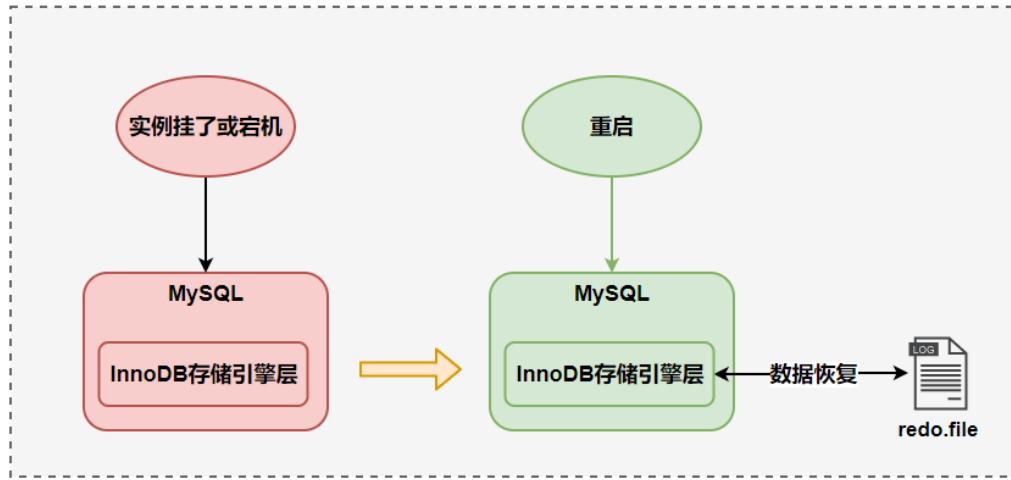
1.1 为什么需要REDO日志

一方面，缓冲池可以帮助我们消除CPU和磁盘之间的鸿沟，checkpoint机制可以保证数据的最终落盘，然而由于checkpoint 并不是每次变更的时候就触发 的，而是master线程隔一段时间去处理的。所以最坏的情况就是事务提交后，刚写完缓冲池，数据库宕机了，那么这段数据就是丢失的，无法恢复。

另一方面，事务包含 持久性 的特性，就是说对于一个已经提交的事务，在事务提交后即使系统发生了崩溃，这个事务对数据库中所做的更改也不能丢失。

那么如何保证这个持久性呢？ 一个简单的做法：在事务提交完成之前把该事务所修改的所有页面都刷新到磁盘，但是这个简单粗暴的做法有些问题

另一个解决的思路：我们只是想让已经提交了的事务对数据库中数据所做的修改永久生效，即使后来系统崩溃，在重启后也能把这种修改恢复出来。所以我们其实没有必要在每次事务提交时就把该事务在内存中修改过的全部页面刷新到磁盘，只需要把 修改 了哪些东西 记录一下 就好。比如，某个事务将系统表空间中 第10号 页面中偏移量为 100 处的那个字节的值 1 改成 2。我们只需要记录一下：将第0号表空间的10号页面的偏移量为100处的值更新为 2。



1.2 REDO日志的好处、特点

1. 好处

- redo日志降低了刷盘频率
- redo日志占用的空间非常小

2. 特点

- redo日志是顺序写入磁盘的
- 事务执行过程中，redo log不断记录

1.3 redo的组成

Redo log可以简单分为以下两个部分：

- 重做日志的缓冲（redo log buffer），保存在内存中，是易失的。

参数设置：innodb_log_buffer_size：

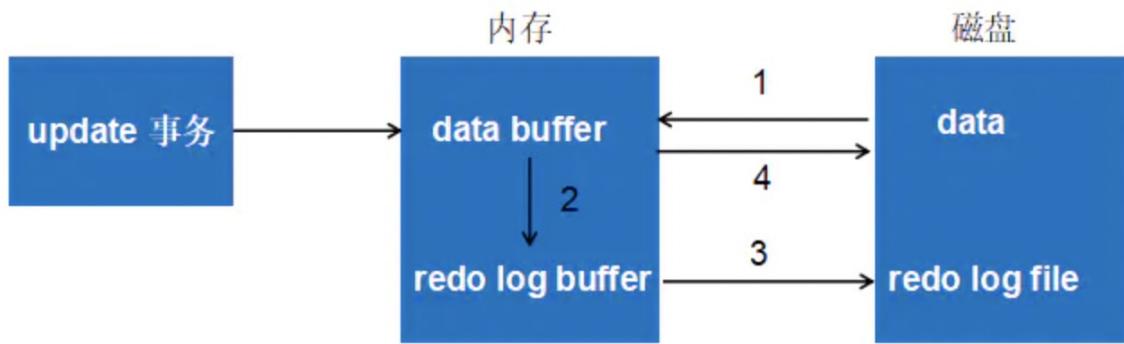
redo log buffer 大小，默认 16M，最大值是4096M，最小值为1M。

```
mysql> show variables like '%innodb_log_buffer_size';
+-----+-----+
| Variable_name      | Value   |
+-----+-----+
| innodb_log_buffer_size | 16777216 |
+-----+-----+
```

- 重做日志文件（redo log file），保存在硬盘中，是持久的。

1.4 redo的整体流程

以一个更新事务为例，redo log 流转过程，如下图所示：



第1步：先将原始数据从磁盘中读入内存中来，修改数据的内存拷贝

第2步：生成一条重做日志并写入 redo log buffer，记录的是数据被修改后的值

第3步：当事务commit时，将 redo log buffer中的内容刷新到 redo log file，对 redo log file采用追加写的方式

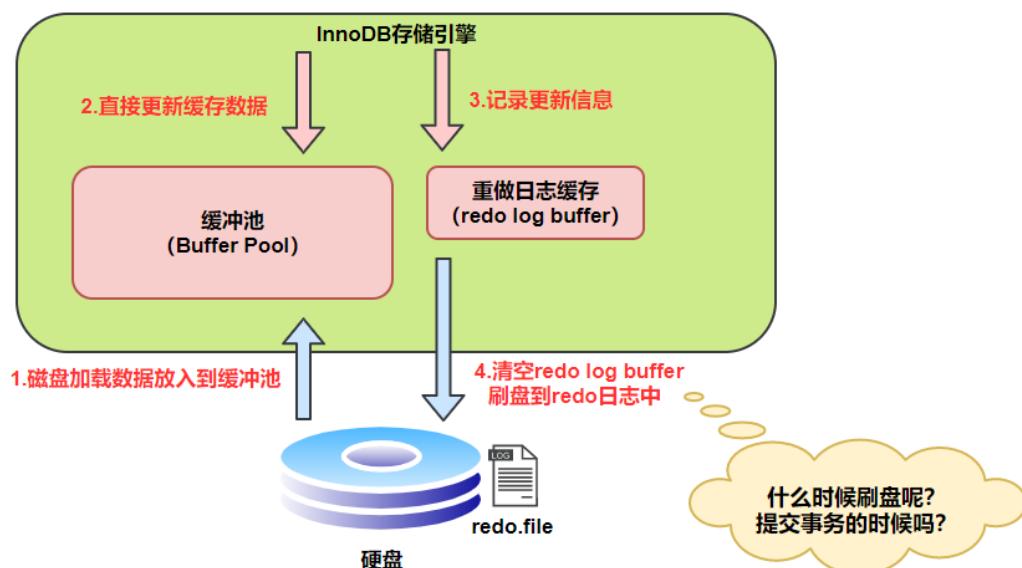
第4步：定期将内存中修改的数据刷新到磁盘中

体会：

Write-Ahead Log(预先日志持久化)：在持久化一个数据页之前，先将内存中相应的日志页持久化。

1.5 redo log的刷盘策略

redo log的写入并不是直接写入磁盘的，InnoDB引擎会在写redo log的时候先写redo log buffer，之后以一定的频率刷入到真正的redo log file 中。这里的一定频率怎么看待呢？这就是我们要说的刷盘策略。



注意，redo log buffer刷盘到redo log file的过程并不是真正的刷到磁盘中去，只是刷入到文件系统缓存（page cache）中去（这是现代操作系统为了提高文件写入效率做的一个优化），真正的写入会交给系统自己来决定（比如page cache足够大了）。那么对于InnoDB来说就存在一个问题，如果交给系统来同步，同样如果系统宕机，那么数据也丢失了（虽然整个系统宕机的概率还是比较小的）。

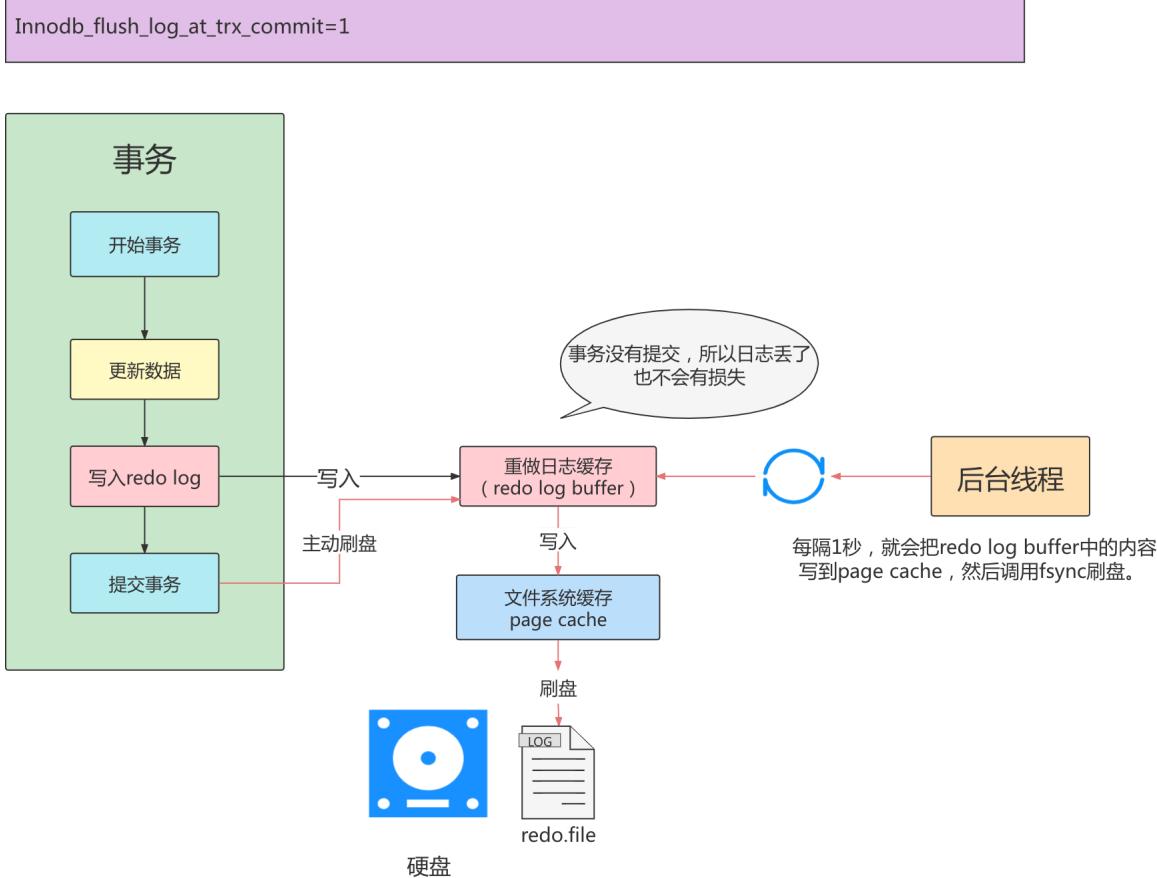
针对这种情况，InnoDB给出 `innodb_flush_log_at_trx_commit` 参数，该参数控制 commit提交事务时，如何将 redo log buffer 中的日志刷新到 redo log file 中。它支持三种策略：

- **设置为0**：表示每次事务提交时不进行刷盘操作。（系统默认master thread每隔1s进行一次重做日志的同步）

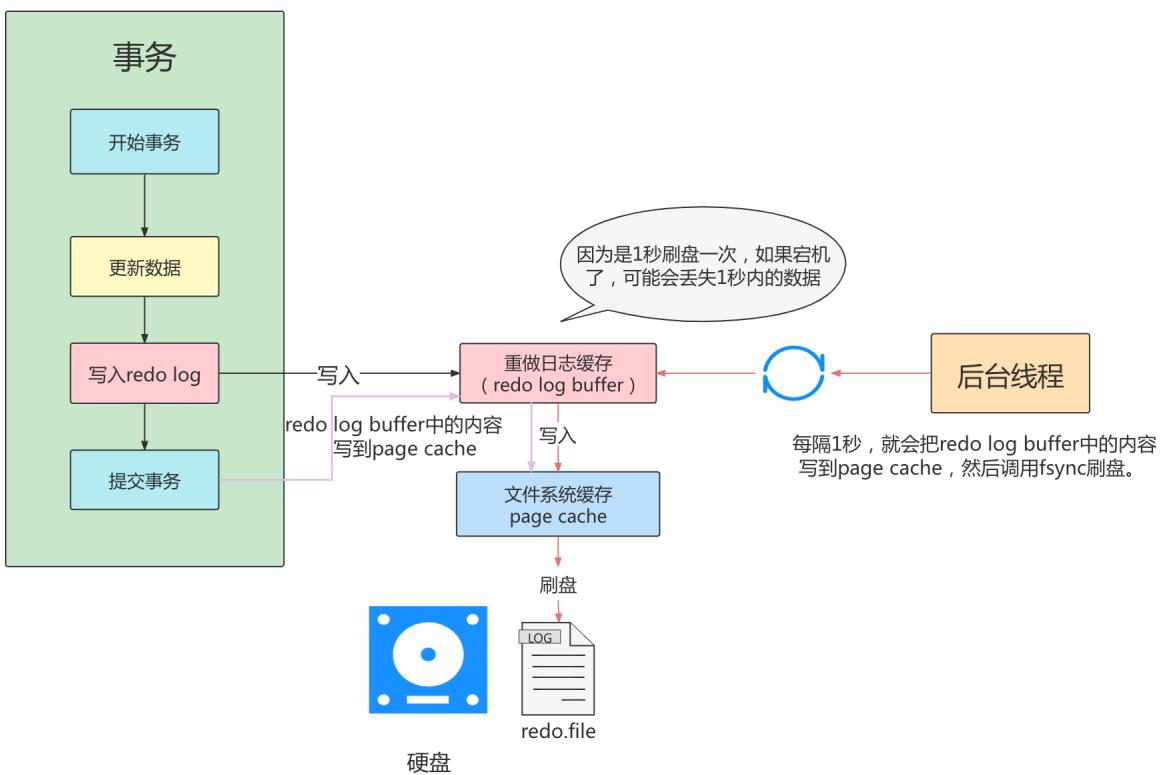
- 设置为1：表示每次事务提交时都将进行同步，刷盘操作（默认值）
- 设置为2：表示每次事务提交时都只把 redo log buffer 内容写入 page cache，不进行同步。由os自己决定什么时候同步到磁盘文件。

1.6 不同刷盘策略演示

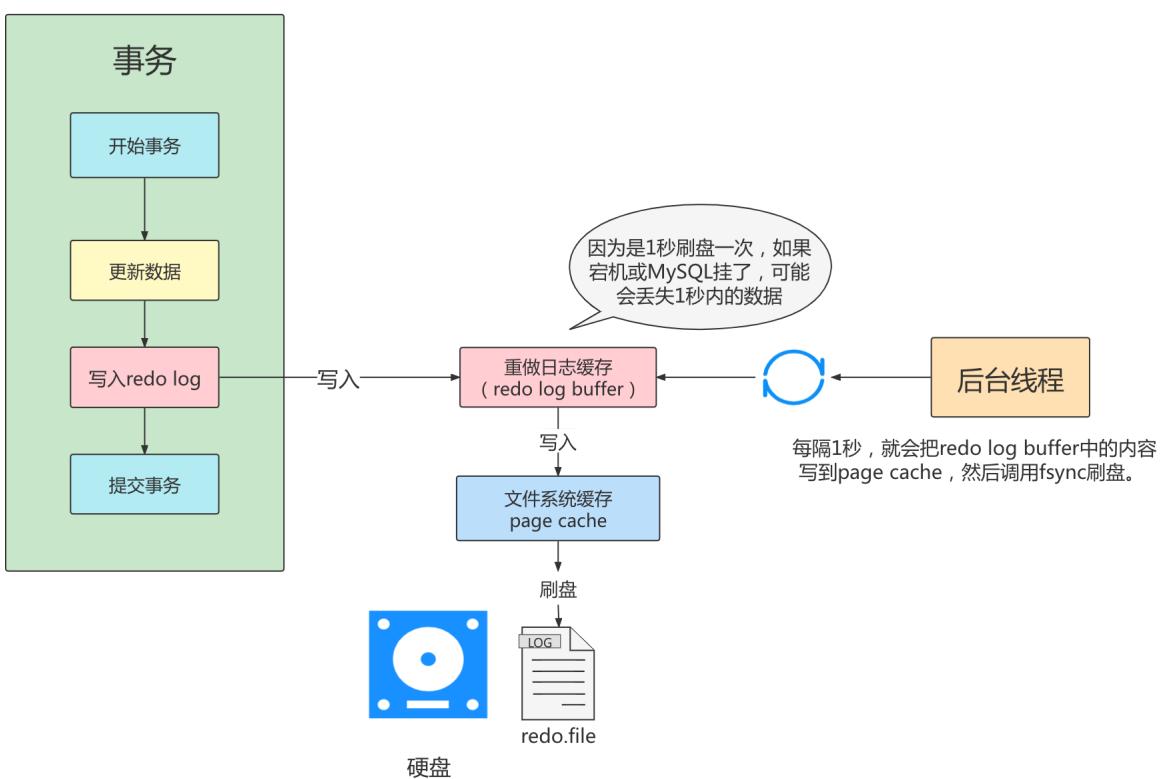
1. 流程图



Innodb_flush_log_at_trx_commit=2



Innodb_flush_log_at_trx_commit=0

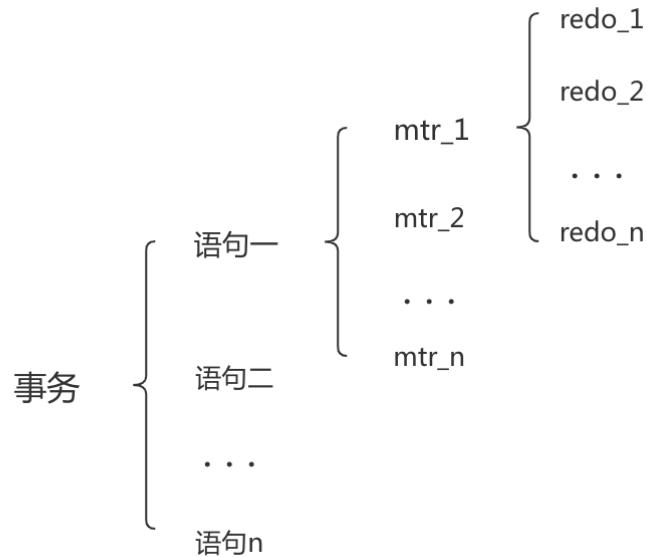


2. 举例

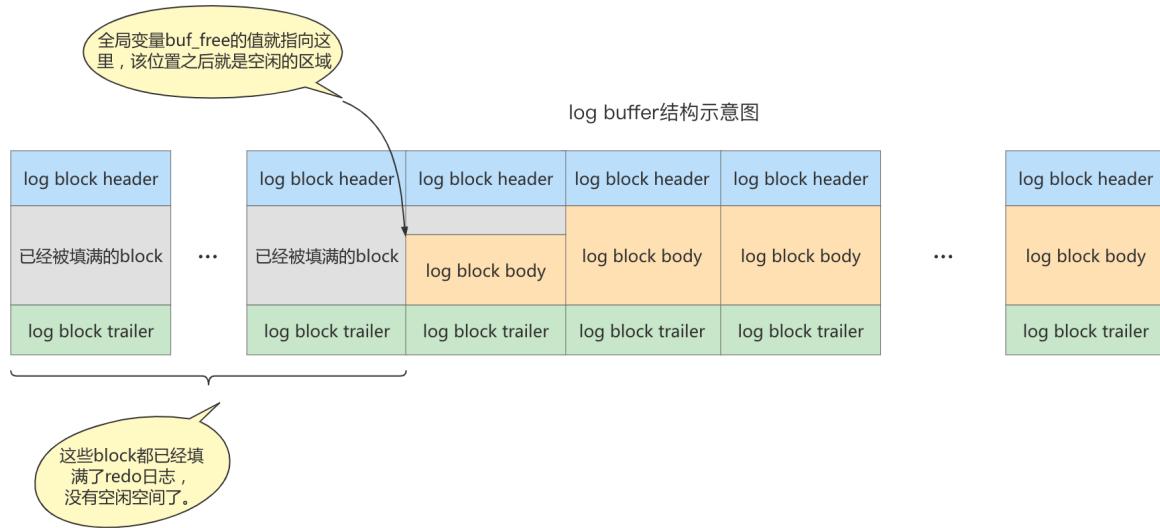
1.7 写入redo log buffer 过程

1. 补充概念：Mini-Transaction

一个事务可以包含若干条语句，每一条语句其实是由若干个 mtr 组成，每一个 mtr 又可以包含若干条 redo日志，画个图表示它们的关系就是这样：



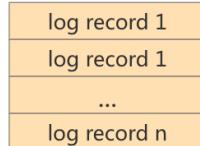
2. redo 日志写入log buffer



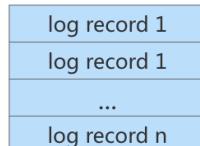
每个mtr都会产生一组redo日志，用示意图来描述一下这些mtr产生的日志情况：

事务T1的mtr

mtr_t1_1产生的一组redo日志：

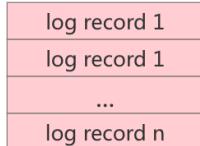


mtr_t1_2产生的一组redo日志：

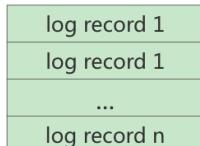


事务T2的mtr

mtr_t2_1产生的一组redo日志：

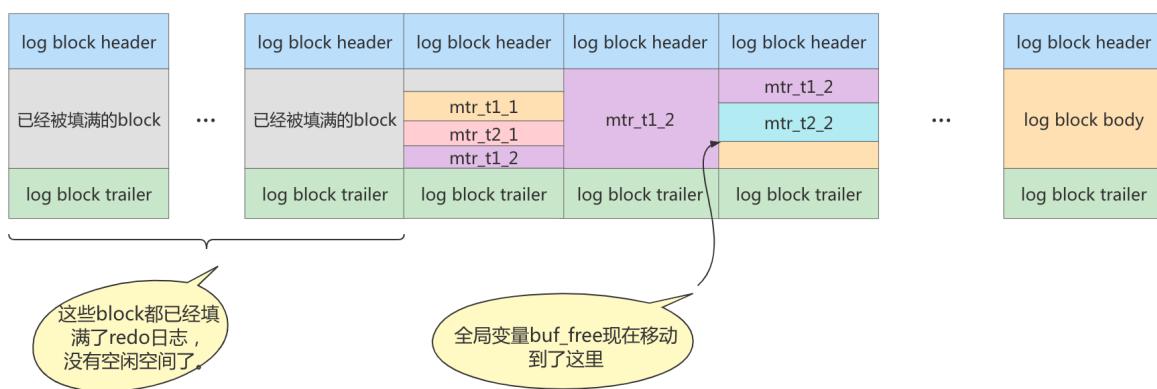


mtr_t2_2产生的一组redo日志：



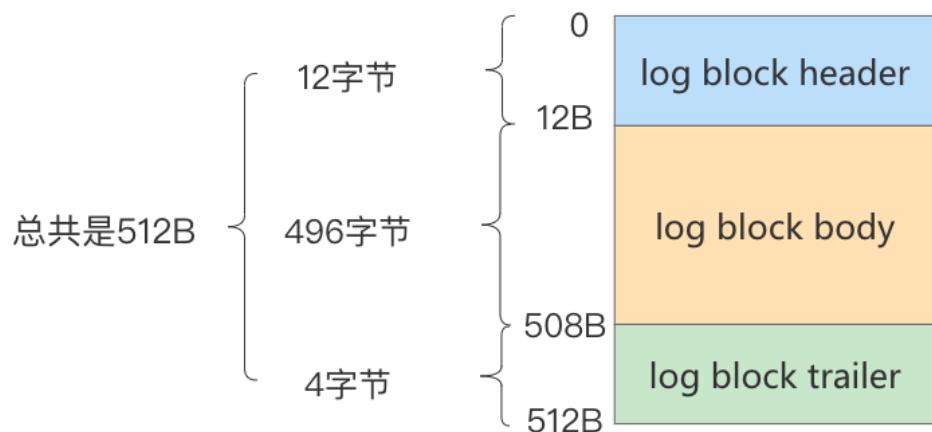
不同的事务可能是 并发 执行的，所以 T1、T2 之间的 mtr 可能是 交替执行 的。

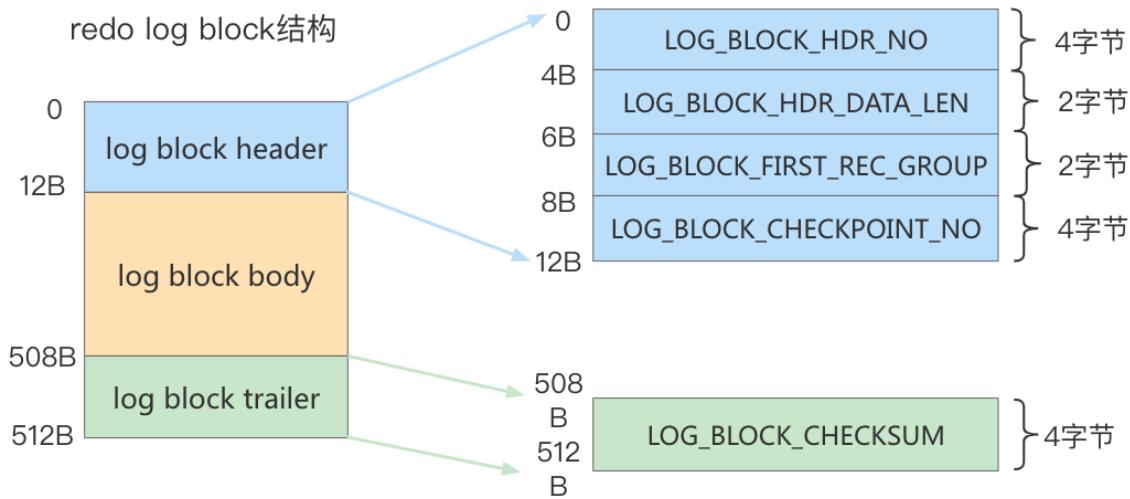
log buffer结构示意图



3. redo log block的结构图

redo log block结构





1.8 redo log file

1. 相关参数设置

- `innodb_log_group_home_dir`：指定 redo log 文件组所在的路径，默认值为 `./`，表示在数据库的数据目录下。MySQL的默认数据目录（`var/lib/mysql`）下默认有两个名为 `ib_logfile0` 和 `ib_logfile1` 的文件，log buffer中的日志默认情况下就是刷新到这两个磁盘文件中。此redo日志文件位置还可以修改。
- `innodb_log_files_in_group`：指明redo log file的个数，命名方式如：`ib_logfile0`, `ib_logfile1`... `ib_logfilen`。默认2个，最大100个。

```
mysql> show variables like 'innodb_log_files_in_group';
+-----+-----+
| Variable_name      | Value |
+-----+-----+
| innodb_log_files_in_group | 2      |
+-----+-----+
#ib_logfile0
#ib_logfile1
```

- `innodb_flush_log_at_trx_commit`：控制 redo log 刷新到磁盘的策略，默认为1。
- `innodb_log_file_size`：单个 redo log 文件设置大小，默认值为 `48M`。最大值为512G，注意最大值指的是整个 redo log 系列文件之和，即 $(\text{innodb_log_files_in_group} * \text{innodb_log_file_size})$ 不能大于最大值512G。

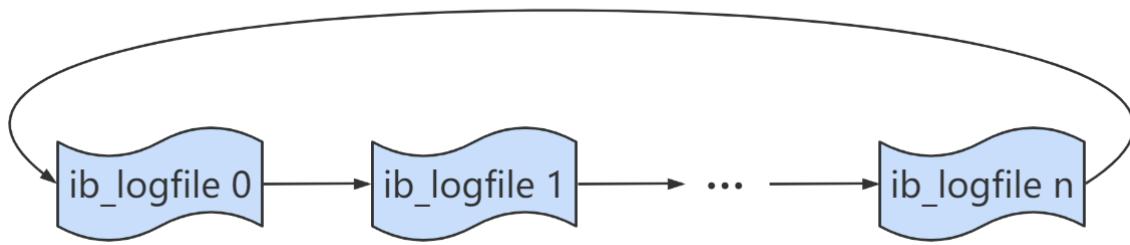
```
mysql> show variables like 'innodb_log_file_size';
+-----+-----+
| Variable_name      | Value   |
+-----+-----+
| innodb_log_file_size | 50331648 |
+-----+-----+
```

根据业务修改其大小，以便容纳较大的事务。编辑my.cnf文件并重启数据库生效，如下所示

```
[root@localhost ~]# vim /etc/my.cnf
innodb_log_file_size=200M
```

2. 日志文件组

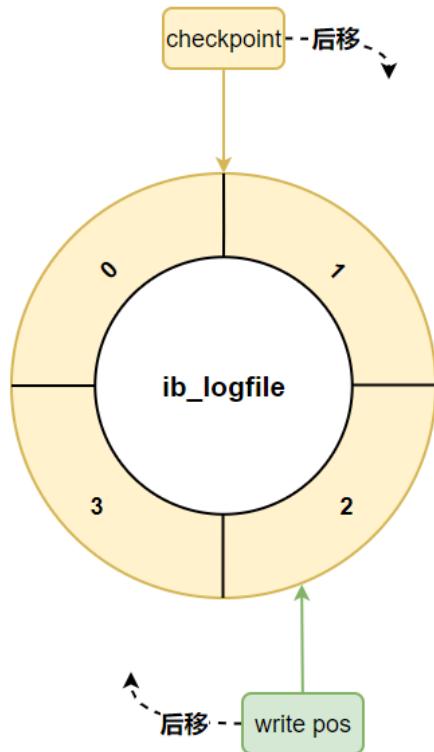
redo日志文件组示意图



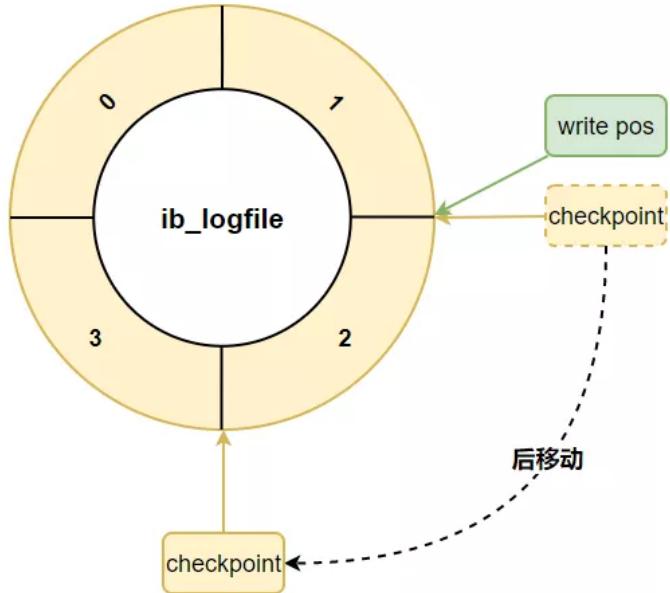
总共的redo日志文件大小其实就是：`innodb_log_file_size × innodb_log_files_in_group`。

采用循环使用的方式向redo日志文件组里写数据的话，会导致后写入的redo日志覆盖掉前边写的redo日志？当然！所以InnoDB的设计者提出了checkpoint的概念。

3. checkpoint



如果 write pos 追上 checkpoint，表示**日志文件组**满了，这时候不能再写入新的 redo log记录，MySQL 得停下来，清空一些记录，把 checkpoint 推进一下。



2. Undo日志

redo log是事务持久性的保证，undo log是事务原子性的保证。在事务中更新数据的前置操作其实是要先写入一个 undo log。

2.1 如何理解Undo日志

事务需要保证原子性，也就是事务中的操作要么全部完成，要么什么也不做。但有时候事务执行到一半会出现一些情况，比如：

- 情况一：事务执行过程中可能遇到各种错误，比如服务器本身错误，操作系统错误，甚至是突然断电导致的错误。
- 情况二：程序员可以在事务执行过程中手动输入 ROLLBACK 语句结束当前事务的执行。

以上情况出现，我们需要把数据改回原先的样子，这个过程称之为回滚，这样就可以造成一个假象：这个事务看起来什么都没做，所以符合原子性要求。

2.2 Undo日志的作用

- **作用1：回滚数据**
- **作用2：MVCC**

2.3 undo的存储结构

1. 回滚段与undo页

InnoDB对undo log的管理采用段的方式，也就是回滚段（rollback segment）。每个回滚段记录了 1024 个 undo log segment，而在每个undo log segment段中进行 undo页 的申请。

- 在 InnoDB1.1 版本之前（不包括1.1版本），只有一个 rollback segment，因此支持同时在线的事务限制为 1024。虽然对绝大多数的应用来说都已经够用。
- 从1.1版本开始InnoDB支持最大 128个 rollback segment，故其支持同时在线的事务限制提高到了 128*1024。

```

mysql> show variables like 'innodb_undo_logs';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| innodb_undo_logs | 128 |
+-----+-----+

```

2. 回滚段与事务

1. 每个事务只会使用一个回滚段，一个回滚段在同一时刻可能会服务于多个事务。
2. 当一个事务开始的时候，会制定一个回滚段，在事务进行的过程中，当数据被修改时，原始的数据会被复制到回滚段。
3. 在回滚段中，事务会不断填充盘区，直到事务结束或所有的空间被用完。如果当前的盘区不够用，事务会在段中请求扩展下一个盘区，如果所有已分配的盘区都被用完，事务会覆盖最初的盘区或者在回滚段允许的情况下扩展新的盘区来使用。
4. 回滚段存在于undo表空间中，在数据库中可以存在多个undo表空间，但同一时刻只能使用一个undo表空间。
5. 当事务提交时，InnoDB存储引擎会做以下两件事情：
 - 将undo log放入列表中，以供之后的purge操作
 - 判断undo log所在的页是否可以重用，若可以分配给下个事务使用

3. 回滚段中的数据分类

1. 未提交的回滚数据(uncommitted undo information)
2. 已经提交但未过期的回滚数据(committed undo information)
3. 事务已经提交并过期的数据(expired undo information)

2.4 undo的类型

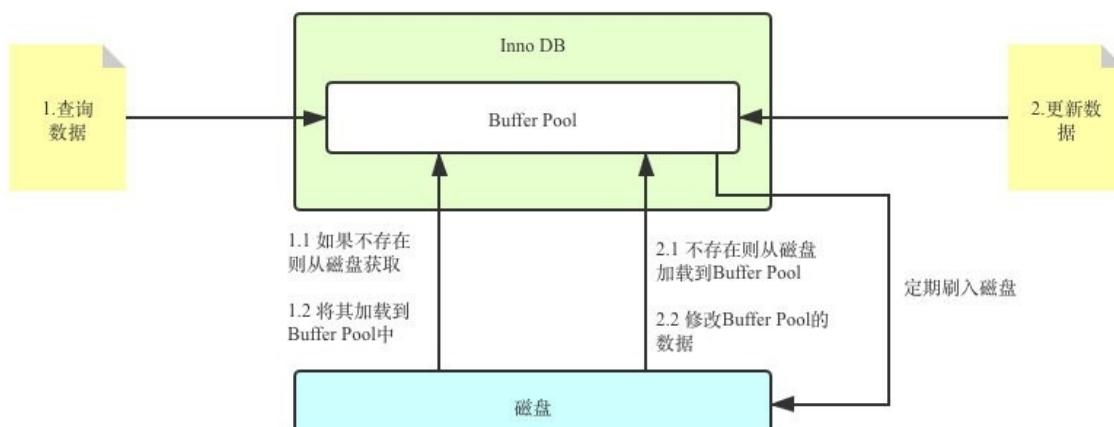
在InnoDB存储引擎中，undo log分为：

- insert undo log
- update undo log

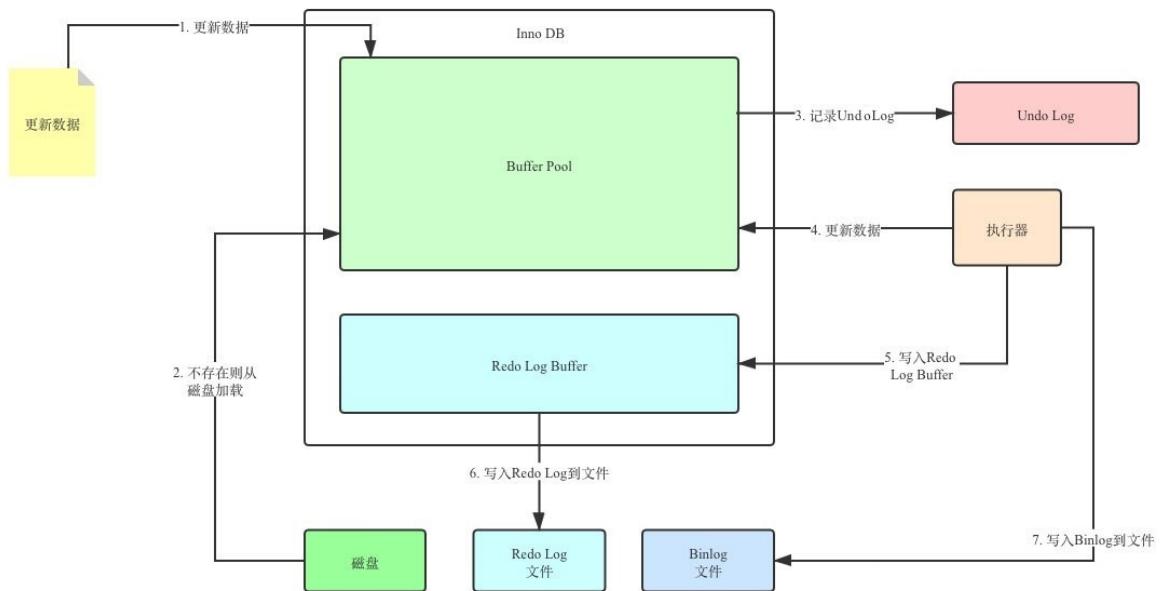
2.5 undo log的生命周期

1. 简要生成过程

只有Buffer Pool的流程：



有了Redo Log和Undo Log之后：

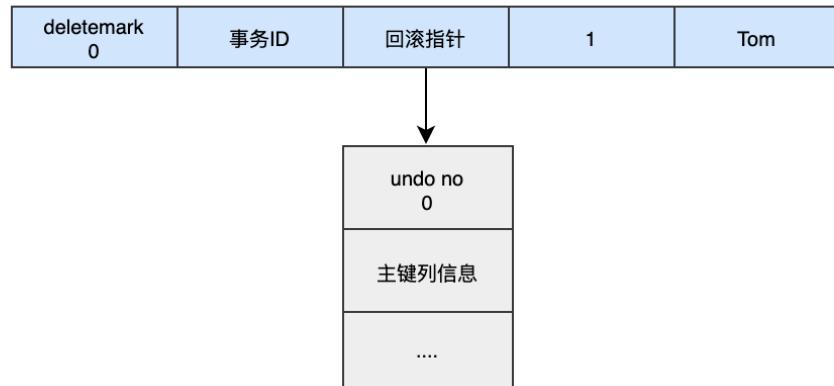


2. 详细生成过程

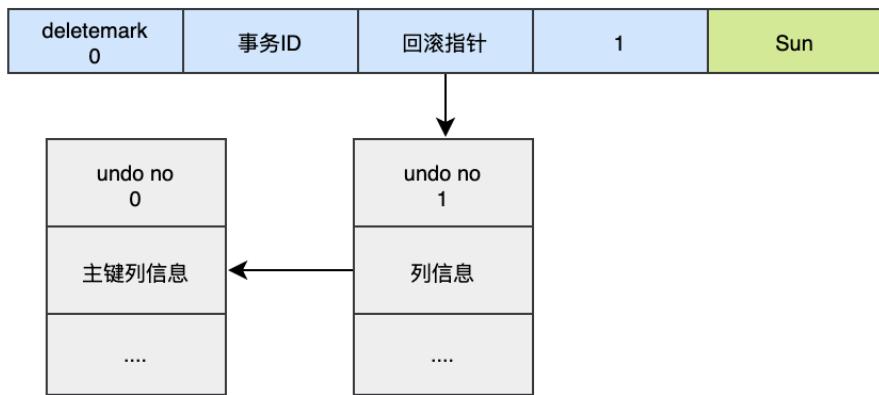
DB_ROW_ID	DB_TRX_ID	DB_ROLL_PTR	列1	列2	...	列n
-----------	-----------	-------------	----	----	-----	----

当我们执行INSERT时：

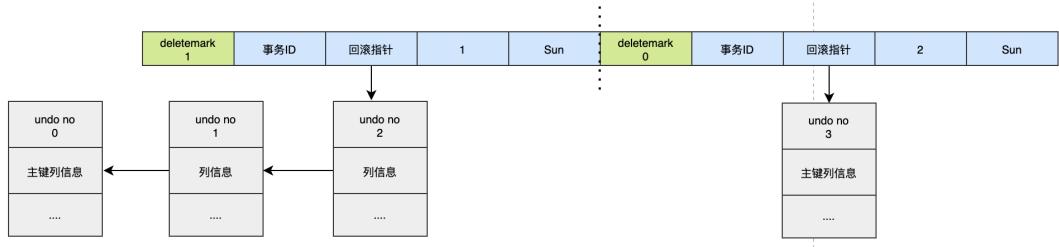
```
begin;
INSERT INTO user (name) VALUES ("tom");
```



当我们执行UPDATE时：



```
UPDATE user SET id=2 WHERE id=1;
```



3. undo log是如何回滚的

以上面的例子来说，假设执行 rollback，那么对应的流程应该是这样：

1. 通过undo no=3的日志把id=2的数据删除
2. 通过undo no=2的日志把id=1的数据的deletemark还原成0
3. 通过undo no=1的日志把id=1的数据的name还原成Tom
4. 通过undo no=0的日志把id=1的数据删除

4. undo log的删除

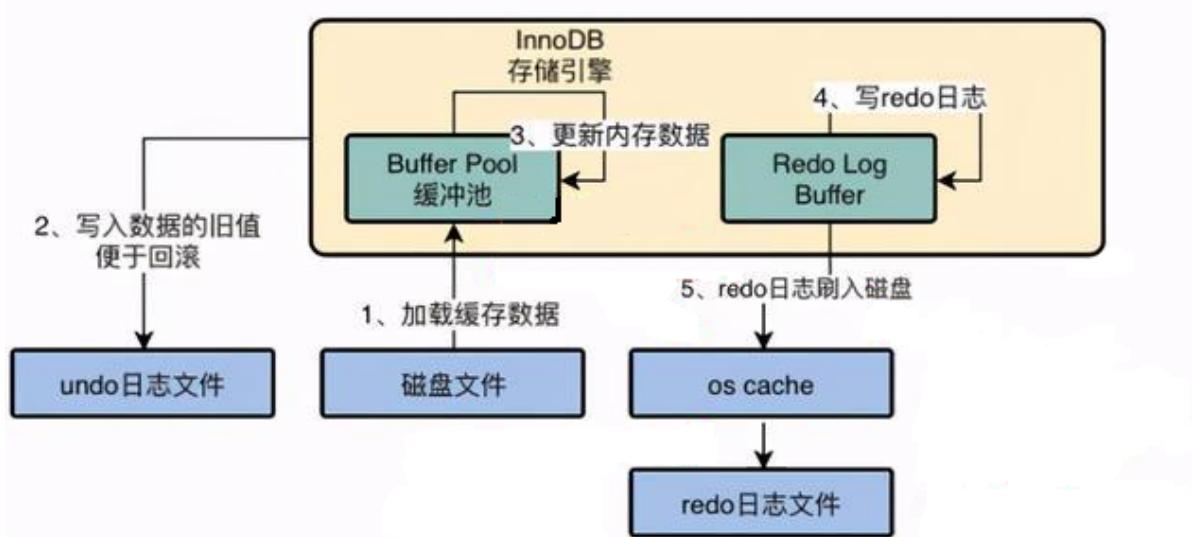
- 针对于insert undo log

因为insert操作的记录，只对事务本身可见，对其他事务不可见。故该undo log可以在事务提交后直接删除，不需要进行purge操作。

- 针对于update undo log

该undo log可能需要提供MVCC机制，因此不能在事务提交时就进行删除。提交时放入undo log链表，等待purge线程进行最后的删除。

2.6 小结



undo log是逻辑日志，对事务回滚时，只是将数据库逻辑地恢复到原来的样子。

redo log是物理日志，记录的是数据页的物理变化，undo log不是redo log的逆过程。

第15章_锁

讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

事务的 **隔离性** 由这章讲述的 **锁** 来实现。

1. 概述

在数据库中，除传统的计算资源（如CPU、RAM、I/O等）的争用以外，数据也是一种供许多用户共享的资源。为保证数据的一致性，需要对 **并发操作进行控制**，因此产生了 **锁**。同时 **锁机制** 也为实现MySQL的各个隔离级别提供了保证。**锁冲突** 也是影响数据库 **并发访问性能** 的一个重要因素。所以锁对数据库而言显得尤其重要，也更加复杂。

2. MySQL并发事务访问相同记录

并发事务访问相同记录的情况大致可以划分为3种：

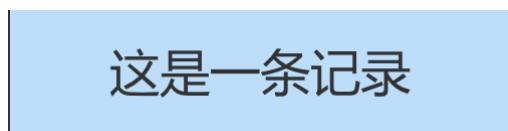
2.1 读-读情况

读-读 情况，即并发事务相继 **读取相同的记录**。读取操作本身不会对记录有任何影响，并不会引起什么问题，所以允许这种情况的发生。

2.2 写-写情况

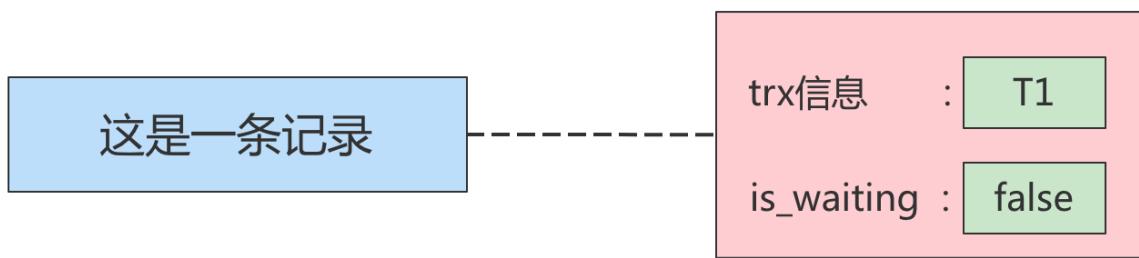
写-写 情况，即并发事务相继对相同的记录做出改动。

在这种情况下会发生 **脏写** 的问题，任何一种隔离级别都不允许这种问题的发生。所以在多个未提交事务相继对一条记录做改动时，需要让它们 **排队执行**，这个排队的过程其实是通过 **锁** 来实现的。这个所谓的锁其实是一个 **内存中的结构**，在事务执行前本来是没有锁的，也就是说一开始是没有 **锁结构** 和记录进行关联的，如图所示：

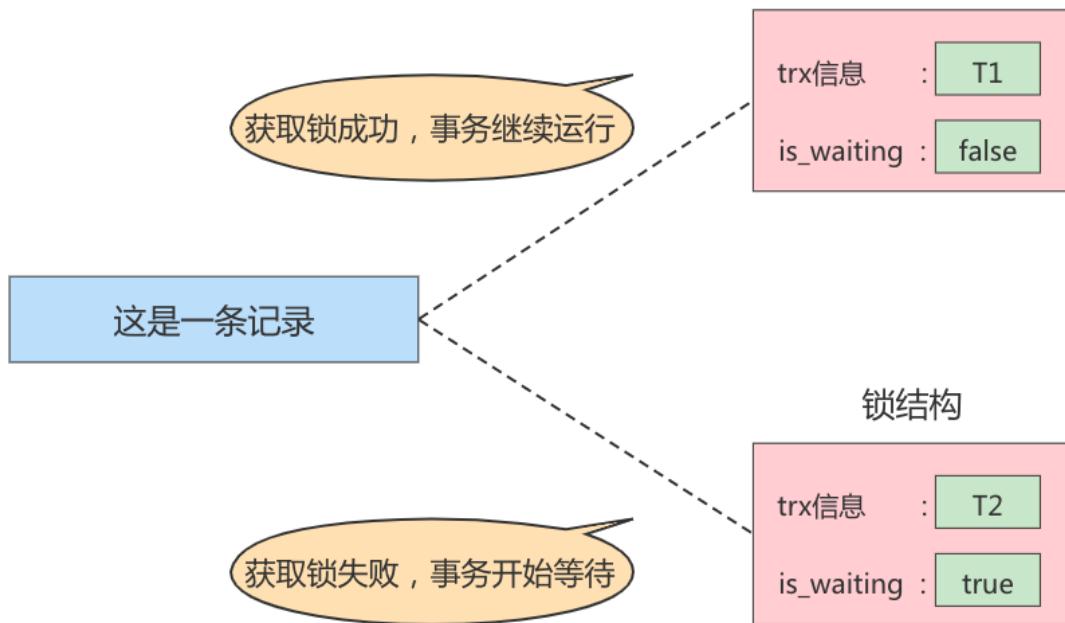


当一个事务想对这条记录做改动时，首先会看看内存中有没有与这条记录关联的 **锁结构**，当没有的时候就会在内存中生成一个 **锁结构** 与之关联。比如，事务 **T1** 要对这条记录做改动，就需要生成一个 **锁结构** 与之关联：

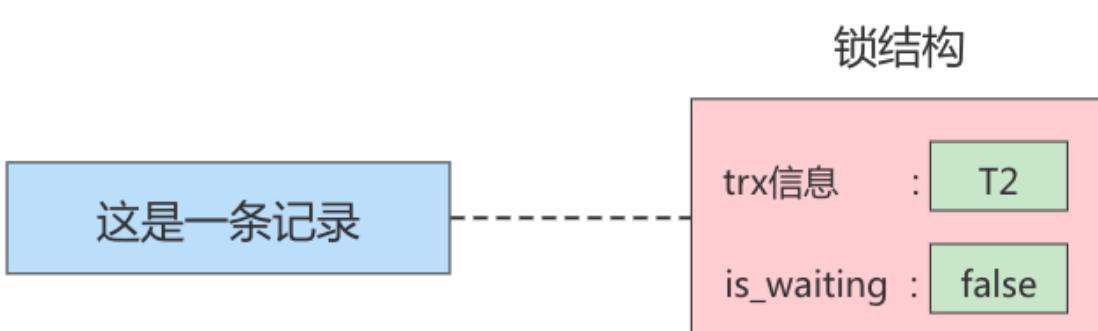
锁结构



锁结构



锁结构



小结几种说法：

- 不加锁

意思就是不需要在内存中生成对应的 锁结构，可以直接执行操作。

- 获取锁成功，或者加锁成功

意思就是在内存中生成了对应的 锁结构，而且锁结构的 `is_waiting` 属性为 `false`，也就是事务可以继续执行操作。

- 获取锁失败，或者加锁失败，或者没有获取到锁

意思就是在内存中生成了对应的 锁结构，不过锁结构的 `is_waiting` 属性为 `true`，也就是事务需要等待，不可以继续执行操作。

2.3 读-写或写-读情况

读-写 或 写-读，即一个事务进行读取操作，另一个进行改动操作。这种情况下可能发生 脏读、不可重复读、幻读 的问题。

各个数据库厂商对 SQL 标准 的支持都可能不一样。比如MySQL在 REPEATABLE READ 隔离级别上就已经解决了 幻读 问题。

2.4 并发问题的解决方案

怎么解决 脏读、不可重复读、幻读 这些问题呢？其实有两种可选的解决方案：

- 方案一：读操作利用多版本并发控制（MVCC，下章讲解），写操作进行 加锁。

普通的SELECT语句在READ COMMITTED和REPEATABLE READ隔离级别下会使用到MVCC读取记录。

- 在 READ COMMITTED 隔离级别下，一个事务在执行过程中每次执行SELECT操作时都会生成一个ReadView，ReadView的存在本身就保证了 事务不可以读取到未提交的事务所做的更改，也就是避免了脏读现象；
- 在 REPEATABLE READ 隔离级别下，一个事务在执行过程中只有 第一次执行SELECT操作 才会生成一个ReadView，之后的SELECT操作都 复用 这个ReadView，这样也就避免了不可重复读和幻读的问题。

- 方案二：读、写操作都采用 加锁 的方式。

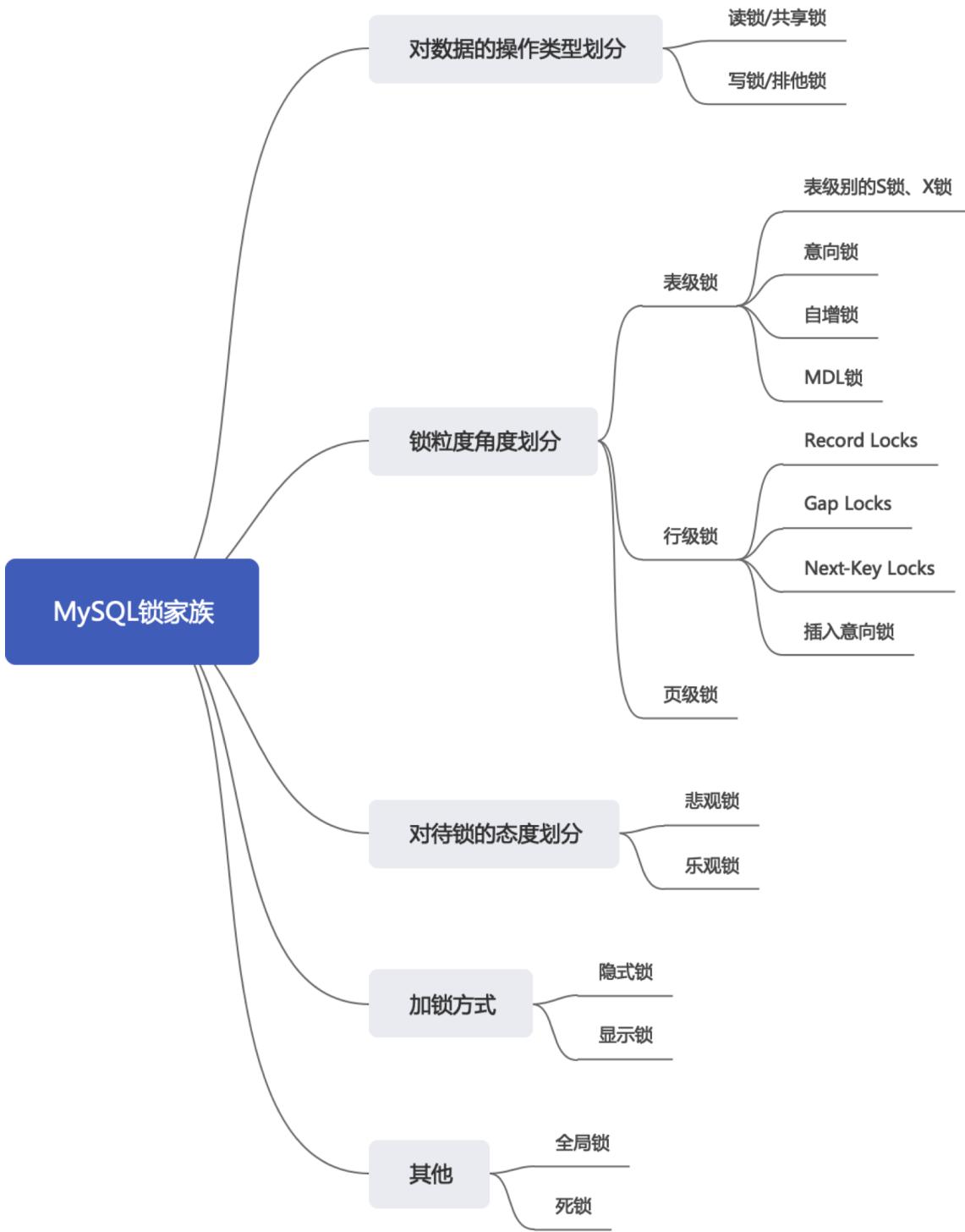
- 小结对比发现：

- 采用 MVCC 方式的话， 读-写 操作彼此并不冲突， 性能更高 。
- 采用 加锁 方式的话， 读-写 操作彼此需要 排队执行， 影响性能。

一般情况下我们当然愿意采用 MVCC 来解决 读-写 操作并发执行的问题，但是业务在某些特殊情况下，要求必须采用 加锁 的方式执行。下面就讲解下MySQL中不同类别的锁。

3. 锁的不同角度分类

锁的分类图，如下：



3.1 从数据操作的类型划分：读锁、写锁

- **读锁**：也称为 **共享锁**、英文用 **S** 表示。针对同一份数据，多个事务的读操作可以同时进行而不会互相影响，相互不阻塞的。
- **写锁**：也称为 **排他锁**、英文用 **X** 表示。当前写操作没有完成前，它会阻断其他写锁和读锁。这样就能确保在给定的时间里，只有一个事务能执行写入，并防止其他用户读取正在写入的同一资源。

需要注意的是对于 InnoDB 引擎来说，读锁和写锁可以加在表上，也可以加在行上。

3.2 从数据操作的粒度划分：表级锁、页级锁、行锁

1. 表锁 (Table Lock)

① 表级别的S锁、X锁

在对某个表执行SELECT、INSERT、DELETE、UPDATE语句时，InnoDB存储引擎是不会为这个表添加表级别的 S 锁 或者 X 锁 的。在对某个表执行一些诸如 ALTER TABLE 、 DROP TABLE 这类的 DDL 语句时，其他事务对这个表并发执行诸如SELECT、INSERT、DELETE、UPDATE的语句会发生阻塞。同理，某个事务中对某个表执行SELECT、INSERT、DELETE、UPDATE语句时，在其他会话中对这个表执行 DDL 语句也会发生阻塞。这个过程其实是通过在 server 层 使用一种称之为 元数据锁 （英文名： Metadata Locks ，简称 MDL ）结构来实现的。

一般情况下，不会使用InnoDB存储引擎提供的表级别的 S 锁 和 X 锁 。只会在一些特殊情况下，比方说 崩溃恢复 过程中用到。比如，在系统变量 autocommit=0, innodb_table_locks = 1 时， 手动 获取 InnoDB存储引擎提供的表t 的 S 锁 或者 X 锁 可以这么写：

- LOCK TABLES t READ : InnoDB存储引擎会对表 t 加表级别的 S 锁 。
- LOCK TABLES t WRITE : InnoDB存储引擎会对表 t 加表级别的 X 锁 。

不过尽量避免在使用InnoDB存储引擎的表上使用 LOCK TABLES 这样的手动锁表语句，它们并不会提供什么额外的保护，只是会降低并发能力而已。InnoDB的厉害之处还是实现了更细粒度的 行锁 ，关于 InnoDB表级别的 S 锁 和 X 锁 大家了解一下就可以了。

MySQL的表级锁有两种模式：（以MyISAM表进行操作的演示）

- 表共享读锁 (Table Read Lock)
- 表独占写锁 (Table Write Lock)

锁类型	自己可读	自己可写	自己可操作其他表	他人可读	他人可写
读锁	是	否	否	是	否, 等
写锁	是	是	否	否, 等	否, 等

② 意向锁 (intention lock)

InnoDB 支持 多粒度锁 (multiple granularity locking) ，它允许 行级锁 与 表级锁 共存，而 意向锁 就是其中的一种 表锁 。

意向锁分为两种：

- **意向共享锁** (intention shared lock, IS) : 事务有意向对表中的某些行加**共享锁** (S锁)

```
-- 事务要获取某些行的 S 锁，必须先获得表的 IS 锁。  
SELECT column FROM table ... LOCK IN SHARE MODE;
```

- **意向排他锁** (intention exclusive lock, IX) : 事务有意向对表中的某些行加**排他锁** (X锁)

```
-- 事务要获取某些行的 X 锁，必须先获得表的 IX 锁。  
SELECT column FROM table ... FOR UPDATE;
```

即：意向锁是由存储引擎 自己维护的 ， 用户无法手动操作意向锁，在为数据行加共享 / 排他锁之前，InnoDB 会先获取该数据行 所在数据表的对应意向锁 。

意向锁的并发性

意向锁不会与行级的共享 / 排他锁互斥！正因为如此，意向锁并不会影响到多个事务对不同数据行加排他锁时的并发性。（不然我们直接用普通的表锁就行了）

我们扩展一下上面 teacher 表的例子来概括一下意向锁的作用（一条数据从被锁定到被释放的过程中，可能存在多种不同锁，但是这里我们只着重表现意向锁）。

从上面的案例可以得到如下结论：

1. InnoDB 支持 多粒度锁，特定场景下，行级锁可以与表级锁共存。
2. 意向锁之间互不排斥，但除了 IS 与 S 兼容外，意向锁会与 共享锁 / 排他锁 互斥。
3. IX, IS 是表级锁，不会和行级的 X, S 锁发生冲突。只会和表级的 X, S 发生冲突。
4. 意向锁在保证并发性的前提下，实现了 行锁和表锁共存 且 满足事务隔离性 的要求。

③ 自增锁 (AUTO-INC锁)

在使用 MySQL 过程中，我们可以为表的某个列添加 AUTO_INCREMENT 属性。举例：

```
CREATE TABLE `teacher` (
  `id` int NOT NULL AUTO_INCREMENT,
  `name` varchar(255) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

由于这个表的 id 字段声明了 AUTO_INCREMENT，意味着在书写插入语句时不需要为其赋值，SQL 语句修改如下所示。

```
INSERT INTO `teacher` (name) VALUES ('zhangsan'), ('lisi');
```

上边的插入语句并没有为 id 列显式赋值，所以系统会自动为它赋上递增的值，结果如下所示。

```
mysql> select * from teacher;
+----+-----+
| id | name   |
+----+-----+
| 1  | zhangsan |
| 2  | lisi    |
+----+-----+
2 rows in set (0.00 sec)
```

现在我们看到的上面插入数据只是一种简单的插入模式，所有插入数据的方式总共分为三类，分别是“Simple inserts”，“Bulk inserts”和“Mixed-mode inserts”。

1. “Simple inserts” (简单插入)

可以 预先确定要插入的行数（当语句被初始处理时）的语句。包括没有嵌套子查询的单行和多行 INSERT...VALUES() 和 REPLACE 语句。比如我们上面举的例子就属于该类插入，已经确定要插入的行数。

2. “Bulk inserts” (批量插入)

事先不知道要插入的行数（和所需自动递增值的数量）的语句。比如 INSERT ... SELECT , REPLACE ... SELECT 和 LOAD DATA 语句，但不包括纯 INSERT。InnoDB 在每处理一行，为 AUTO_INCREMENT 列分配一个新值。

3. “Mixed-mode inserts” (混合模式插入)

这些是“Simple inserts”语句但是指定部分新行的自动递增值。例如 INSERT INTO teacher (id, name) VALUES (1, 'a'), (NULL, 'b'), (5, 'c'), (NULL, 'd')；只是指定了部分 id 的值。另一种类型的“混合模式插入”是 INSERT ... ON DUPLICATE KEY UPDATE。

innodb_autoinc_lock_mode有三种取值，分别对应与不同锁定模式：

(1) innodb_autoinc_lock_mode = 0 (“传统”锁定模式)

在此锁定模式下，所有类型的insert语句都会获得一个特殊的表级AUTO-INC锁，用于插入具有 AUTO_INCREMENT列的表。这种模式其实就如我们上面的例子，即每当执行insert的时候，都会得到一个表级锁(AUTO-INC锁)，使得语句中生成的auto_increment为顺序，且在binlog中重放的时候，可以保证 master与slave中数据的auto_increment是相同的。因为是表级锁，当在同一时间多个事务中执行insert的时候，对于AUTO-INC锁的争夺会 限制并发 能力。

(2) innodb_autoinc_lock_mode = 1 (“连续”锁定模式)

在 MySQL 8.0 之前，连续锁定模式是 默认 的。

在这个模式下，“bulk inserts”仍然使用AUTO-INC表级锁，并保持到语句结束。这适用于所有INSERT ... SELECT, REPLACE ... SELECT和LOAD DATA语句。同一时刻只有一个语句可以持有AUTO-INC锁。

对于“Simple inserts”（要插入的行数事先已知），则通过在 mutex（轻量锁）的控制下获得所需数量的自动递增值来避免表级AUTO-INC锁，它只在分配过程的持续时间内保持，而不是直到语句完成。不使用表级AUTO-INC锁，除非AUTO-INC锁由另一个事务保持。如果另一个事务保持AUTO-INC锁，则“Simple inserts”等待AUTO-INC锁，如同它是一个“bulk inserts”。

(3) innodb_autoinc_lock_mode = 2 (“交错”锁定模式)

从 MySQL 8.0 开始，交错锁模式是 默认 设置。

在此锁定模式下，自动递增值 保证 在所有并发执行的所有类型的insert语句中是 唯一 且 单调递增 的。但是，由于多个语句可以同时生成数字（即，跨语句交叉编号），**为任何给定语句插入的行生成的值可能不是连续的**。

④ 元数据锁 (MDL锁)

MySQL5.5引入了meta data lock，简称MDL锁，属于表锁范畴。MDL 的作用是，保证读写的正确性。比如，如果一个查询正在遍历一个表中的数据，而执行期间另一个线程对这个 表结构做变更，增加了一列，那么查询线程拿到的结果跟表结构对不上，肯定是不行的。

因此，**当对一个表做增删改查操作的时候，加 MDL读锁；当要对表做结构变更操作的时候，加 MDL写锁。**

2. InnoDB中的行锁

① 记录锁 (Record Locks)

记录锁也就是仅仅把一条记录锁上，官方的类型名称为：LOCK_REC_NOT_GAP。比如我们把id值为8的那条记录加一个记录锁的示意图如图所示。仅仅是锁住了id值为8的记录，对周围的数据没有影响。

聚簇索引示意图

给id值为8的记录加类型为
LOCK_REC_NOT_GAP的记录锁

id列：	1	3	8	15	20
name列：	张三	李四	王五	赵六	钱七
class列：	一班	一班	二班	二班	三班

举例如下：

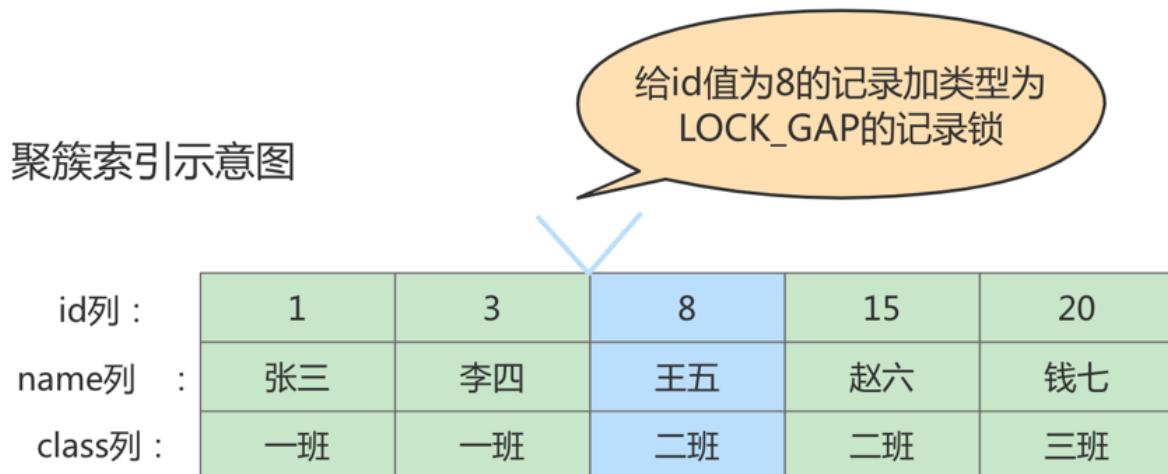
Session1	Session2
<pre>mysql>set autocommit=0; 更新但是不提交，没有手写 commit; mysql>update student set name="张三 1" where id=1; Query OK,1 row affected(0.00 sec) Rows matched: 1 Changed: 1 Warnings:0 mysql></pre>	<pre>mysql>set autocommit=0; Session2 被阻塞，只能等待 mysql>update student set name="李四 1" where id=3; Query OK, 1 rows affected (0.00 秒) mysql>update student set name="张三 1" where id=1; ERROR 1205(HY000): Lock wait timeout exceeded;try restarting transaction 再次进行更新请求 mysql>update student set name="张三 1" where id=1;</pre>
提交更新	解除阻塞，更新正常进行 <pre>mysql>update student set name="张三 1" where id=1; Query OK,1 row affected(5.36 sec) Rows matched: 1 Changed: 1 Warnings:0</pre>

记录锁是有S锁和X锁之分的，称之为 **S型记录锁** 和 **X型记录锁**。

- 当一个事务获取了一条记录的S型记录锁后，其他事务也可以继续获取该记录的S型记录锁，但不可以继续获取X型记录锁；
- 当一个事务获取了一条记录的X型记录锁后，其他事务既不可以继续获取该记录的S型记录锁，也不可以继续获取X型记录锁。

② 间隙锁 (Gap Locks)

MySQL 在 REPEATABLE READ 隔离级别下是可以解决幻读问题的，解决方案有两种，可以使用 MVCC 方案解决，也可以采用 加锁 方案解决。但是在使用加锁方案解决时有个大问题，就是事务在第一次执行读取操作时，那些幻影记录尚不存在，我们无法给这些 幻影记录 加上 记录锁 。InnoDB提出了一种称之为 **Gap Locks** 的锁，官方的类型名称为： **LOCK_GAP**，我们可以简称为 **gap锁**。比如，把id值为8的那条记录加一个gap锁的示意图如下。



图中id值为8的记录加了gap锁，意味着 不允许别的事务在id值为8的记录前边的间隙插入新记录，其实质是 id列的值(3, 8)这个区间的新记录是不允许立即插入的。比如，有另外一个事务再想插入一条id值为4的新记录，它定位到该条新记录的下一条记录的id值为8，而这条记录上又有一个gap锁，所以就会阻塞插入操作，直到拥有这个gap锁的事务提交了之后，id列的值在区间(3, 8)中的新记录才可以被插入。

gap锁的提出仅仅是为了防止插入幻影记录而提出的。

③ 临键锁 (Next-Key Locks)

有时候我们既想 锁住某条记录 , 又想 阻止 其他事务在该记录前边的 间隙插入新记录 , 所以InnoDB就提出了一种称之为 **Next-Key Locks** 的锁, 官方的类型名称为: **LOCK_ORDINARY** , 我们也可以简称为 **next-key锁** 。Next-Key Locks是在存储引擎 **innodb** 、事务级别在 **可重复读** 的情况下使用的数据库锁, innodb默认的锁就是Next-Key locks。

```
begin;
select * from student where id <=8 and id > 3 for update;
```

④ 插入意向锁 (Insert Intention Locks)

我们说一个事务在 **插入** 一条记录时需要判断一下插入位置是不是被别的事务加了 **gap锁** (**next-key锁** 也包含 **gap锁**) , 如果有的话, 插入操作需要等待, 直到拥有 **gap锁** 的那个事务提交。但是**InnoDB规定事务在等待的时候也需要在内存中生成一个锁结构**, 表明有事务想在某个 **间隙** 中 **插入** 新记录, 但是现在在等待。InnoDB就把这种类型的锁命名为 **Insert Intention Locks** , 官方的类型名称为: **LOCK_INSERT_INTENTION** , 我们称为 **插入意向锁** 。插入意向锁是一种 **Gap锁** , 不是意向锁, 在insert 操作时产生。

插入意向锁是在插入一条记录行前, 由 **INSERT** 操作产生的一种间隙锁。

事实上**插入意向锁并不会阻止别的事务继续获取该记录上任何类型的锁**。

3. 页锁

页锁就是在 **页的粒度** 上进行锁定, 锁定的数据资源比行锁要多, 因为一个页中可以有多个行记录。当我们使用页锁的时候, 会出现数据浪费的现象, 但这样的浪费最多也就是一个页上的数据行。 **页锁的开销介于表锁和行锁之间, 会出现死锁。锁定粒度介于表锁和行锁之间, 并发度一般。**

每个层级的锁数量是有限制的, 因为锁会占用内存空间, **锁空间的大小是有限的** 。当某个层级的锁数量超过了这个层级的阈值时, 就会进行 **锁升级** 。锁升级就是用更大粒度的锁替代多个更小粒度的锁, 比如 InnoDB 中行锁升级为表锁, 这样做的好处是占用的锁空间降低了, 但同时数据的并发度也下降了。

3.3 从对待锁的态度划分:乐观锁、悲观锁

从对待锁的态度来看锁的话, 可以将锁分成乐观锁和悲观锁, 从名字中也可以看出这两种锁是两种看待 **数据并发的思维方式** 。需要注意的是, 乐观锁和悲观锁并不是锁, 而是锁的 **设计思想** 。

1. 悲观锁 (Pessimistic Locking)

悲观锁是一种思想, 顾名思义, 就是很悲观, 对数据被其他事务的修改持保守态度, 会通过数据库自身的锁机制来实现, 从而保证数据操作的排它性。

悲观锁总是假设最坏的情况, 每次去拿数据的时候都认为别人会修改, 所以每次在拿数据的时候都会上锁, 这样别人想拿这个数据就会 **阻塞** 直到它拿到锁 (**共享资源每次只给一个线程使用, 其它线程阻塞, 用完后再把资源转让给其它线程**) 。比如行锁, 表锁等, 读锁, 写锁等, 都是在做操作之前先上锁, 当其他线程想要访问数据时, 都需要阻塞挂起。Java中 **synchronized** 和 **ReentrantLock** 等独占锁就是悲观锁思想的实现。

2. 乐观锁 (Optimistic Locking)

乐观锁认为对同一数据的并发操作不会总发生, 属于小概率事件, 不用每次都对数据上锁, 但是在更新的时候会判断一下在此期间别人有没有去更新这个数据, 也就是**不采用数据库自身的锁机制, 而是通过程序来实现**。在程序上, 我们可以采用 **版本号机制** 或者 **CAS机制** 实现。**乐观锁适用于多读的应用类型, 这样可以提高吞吐量**。在Java中 **java.util.concurrent.atomic** 包下的原子变量类就是使用了乐观锁

的一种实现方式：CAS实现的。

1. 乐观锁的版本号机制

在表中设计一个 版本字段 `version`，第一次读的时候，会获取 `version` 字段的取值。然后对数据进行更新或删除操作时，会执行 `UPDATE ... SET version=version+1 WHERE version=version`。此时如果已经有事务对这条数据进行了更改，修改就不会成功。

2. 乐观锁的时间戳机制

时间戳和版本号机制一样，也是在更新提交的时候，将当前数据的时间戳和更新之前取得的时间戳进行比较，如果两者一致则更新成功，否则就是版本冲突。

你能看到乐观锁就是程序员自己控制数据并发操作的权限，基本是通过给数据行增加一个戳（版本号或者时间戳），从而证明当前拿到的数据是否最新。

3. 两种锁的适用场景

从这两种锁的设计思想中，我们总结一下乐观锁和悲观锁的适用场景：

1. 乐观锁 适合 读操作多 的场景，相对来说写的操作比较少。它的优点在于 程序实现，不存在死锁 问题，不过适用场景也会相对乐观，因为它阻止不了除了程序以外的数据库操作。
2. 悲观锁 适合 写操作多 的场景，因为写的操作具有 排它性。采用悲观锁的方式，可以在数据库层面阻止其他事务对该数据的操作权限，防止 读 - 写 和 写 - 写 的冲突。

3.4 按加锁的方式划分：显式锁、隐式锁

1. 隐式锁

- **情景一：**对于聚簇索引记录来说，有一个 `trx_id` 隐藏列，该隐藏列记录着最后改动该记录的 事务 id。那么如果在当前事务中新插入一条聚簇索引记录后，该记录的 `trx_id` 隐藏列代表的就是当前事务的 事务 id，如果其他事务此时想对该记录添加 S 锁 或者 X 锁 时，首先会看一下该记录的 `trx_id` 隐藏列代表的事务是否是当前的活跃事务，如果是的话，那么就帮助当前事务创建一个 X 锁（也就是为当前事务创建一个锁结构，`is_waiting` 属性是 `false`），然后自己进入等待状态（也就是为自己也创建一个锁结构，`is_waiting` 属性是 `true`）。
- **情景二：**对于二级索引记录来说，本身并没有 `trx_id` 隐藏列，但是在二级索引页面的 Page Header 部分有一个 `PAGE_MAX_TRX_ID` 属性，该属性代表对该页面做改动的最大的 事务 id，如果 `PAGE_MAX_TRX_ID` 属性值小于当前最小的活跃 事务 id，那么说明对该页面做修改的事务都已经提交了，否则就需要在页面中定位到对应的二级索引记录，然后回表找到它对应的聚簇索引记录，然后再重复 情景一 的做法。

session 1:

```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> insert INTO student VALUES(34, "周八", "二班");
Query OK, 1 row affected (0.00 sec)
```

session 2:

```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)
mysql> select * from student lock in share mode; #执行完，当前事务被阻塞
```

执行下述语句，输出结果：

```
mysql> SELECT * FROM performance_schema.data_lock_waits\G;
***** 1. row *****
      ENGINE: INNODB
REQUESTING_ENGINE_LOCK_ID: 140562531358232:7:4:9:140562535668584
REQUESTING_ENGINE_TRANSACTION_ID: 422037508068888
    REQUESTING_THREAD_ID: 64
        REQUESTING_EVENT_ID: 6
REQUESTING_OBJECT_INSTANCE_BEGIN: 140562535668584
    BLOCKING_ENGINE_LOCK_ID: 140562531351768:7:4:9:140562535619104
    BLOCKING_ENGINE_TRANSACTION_ID: 15902
        BLOCKING_THREAD_ID: 64
        BLOCKING_EVENT_ID: 6
BLOCKING_OBJECT_INSTANCE_BEGIN: 140562535619104
1 row in set (0.00 sec)
```

隐式锁的逻辑过程如下：

- A. InnoDB的每条记录中都一个隐含的trx_id字段，这个字段存在于聚簇索引的B+Tree中。
- B. 在操作一条记录前，首先根据记录中的trx_id检查该事务是否是活动的事务(未提交或回滚)。如果是活动的事务，首先将 **隐式锁** 转换为 **显式锁** (就是为该事务添加一个锁)。
- C. 检查是否有锁冲突，如果有冲突，创建锁，并设置为waiting状态。如果没有冲突不加锁，跳到E。
- D. 等待加锁成功，被唤醒，或者超时。
- E. 写数据，并将自己的trx_id写入trx_id字段。

2. 显式锁

通过特定的语句进行加锁，我们一般称之为显示加锁，例如：

显示加共享锁：

```
select .... lock in share mode
```

显示加排它锁：

```
select .... for update
```

3.5 其它锁之：全局锁

全局锁就是对 **整个数据库实例** 加锁。当你需要让整个库处于 **只读状态** 的时候，可以使用这个命令，之后其他线程的以下语句会被阻塞：数据更新语句（数据的增删改）、数据定义语句（包括建表、修改表结构等）和更新类事务的提交语句。全局锁的典型使用 **场景** 是：做 **全库逻辑备份**。

全局锁的命令：

```
Flush tables with read lock
```

3.6 其它锁之：死锁

死锁是指两个或多个事务在同一资源上相互占用，并请求锁定对方占用的资源，从而导致恶性循环。死锁示例：

	事务1	事务2
1	start transaction; update account set money=10 where id=1;	start transaction;
2		update account set money=10 where id=2;
3	update account set money=20 where id=2;	
4		update account set money=20 where id=1;

这时候，事务1在等待事务2释放id=2的行锁，而事务2在等待事务1释放id=1的行锁。事务1和事务2在互相等待对方的资源释放，就是进入了死锁状态。当出现死锁以后，有 **两种策略**：

- 一种策略是，直接进入等待，直到超时。这个超时时间可以通过参数 `innodb_lock_wait_timeout` 来设置。
- 另一种策略是，发起死锁检测，发现死锁后，主动回滚死锁链条中的某一个事务（将持有最少行级排他锁的事务进行回滚），让其他事务得以继续执行。将参数 `innodb_deadlock_detect` 设置为 `on`，表示开启这个逻辑。

第二种策略的成本分析

方法1：如果你能确保这个业务一定不会出现死锁，可以临时把死锁检测关掉。但是这种操作本身带有一定的风险，因为业务设计的时候一般不会把死锁当做一个严重错误，毕竟出现死锁了，就回滚，然后通过业务重试一般就没问题了，这是 **业务无损** 的。而关掉死锁检测意味着可能会出现大量的超时，这是 **业务有损** 的。

方法2：控制并发度。如果并发能够控制住，比如同一行同时最多只有10个线程在更新，那么死锁检测的成本很低，就不会出现这个问题。

这个并发控制要做在 **数据库服务端**。如果你有中间件，可以考虑在 **中间件实现**；甚至有能力修改MySQL源码的人，也可以做在MySQL里面。基本思路就是，对于相同行的更新，在进入引擎之前排队，这样在InnoDB内部就不会有大量的死锁检测工作了。

4. 锁的内存结构

InnoDB 存储引擎中的 **锁结构** 如下：

InnoDB存储引擎事务锁结构

表锁特有结构：

表信息
其他信息

锁所在事务信息
索引信息
表锁/行锁信息
type_mode
其他信息
一堆比特位

行锁特有结构：

Space ID
Page Number
n_bits

结构解析：

1. 锁所在的事务信息：

不论是 表锁 还是 行锁，都是在事务执行过程中生成的，哪个事务生成了这个 锁结构，这里就记录这个事务的信息。

此 锁所在的事务信息 在内存结构中只是一个指针，通过指针可以找到内存中关于该事务的更多信息，比方说事务id等。

2. 索引信息：

对于 行锁 来说，需要记录一下加锁的记录是属于哪个索引的。这里也是一个指针。

3. 表锁 / 行锁信息：

表锁结构 和 行锁结构 在这个位置的内容是不同的：

- 表锁：

记载着是对哪个表加的锁，还有其他的一些信息。

- 行锁：

记载了三个重要的信息：

- Space ID：记录所在表空间。

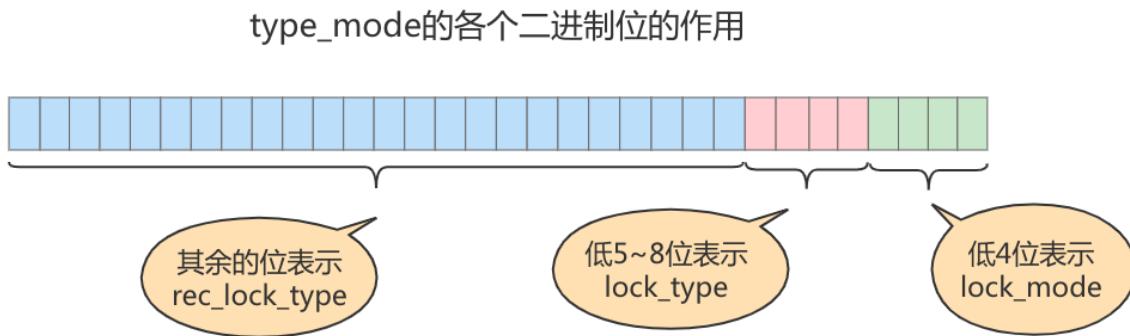
- Page Number：记录所在页号。

- n_bits：对于行锁来说，一条记录就对应着一个比特位，一个页面中包含很多记录，用不同的比特位来区分到底是哪一条记录加了锁。为此在行锁结构的末尾放置了一堆比特位，这个 n_bits 属性代表使用了多少比特位。

n_bits的值一般都比页面中记录条数多一些。主要是为了之后在页面中插入了新记录后也不至于重新分配锁结构

4. type_mode :

这是一个32位的数，被分成了 `lock_mode`、`lock_type` 和 `rec_lock_type` 三个部分，如图所示：



- 锁的模式 (`lock_mode`)，占用低4位，可选的值如下：

- `LOCK_IS` (十进制的 0)：表示共享意向锁，也就是 `IS` 锁。
- `LOCK_IX` (十进制的 1)：表示独占意向锁，也就是 `IX` 锁。
- `LOCK_S` (十进制的 2)：表示共享锁，也就是 `S` 锁。
- `LOCK_X` (十进制的 3)：表示独占锁，也就是 `X` 锁。
- `LOCK_AUTO_INC` (十进制的 4)：表示 `AUTO-INC` 锁。

在InnoDB存储引擎中，`LOCK_IS`, `LOCK_IX`, `LOCK_AUTO_INC`都算是表级锁的模式，`LOCK_S`和`LOCK_X`既可以算是表级锁的模式，也可以是行级锁的模式。

- 锁的类型 (`lock_type`)，占用第5~8位，不过现阶段只有第5位和第6位被使用：

- `LOCK_TABLE` (十进制的 16)，也就是当第5个比特位置为1时，表示表级锁。
- `LOCK_REC` (十进制的 32)，也就是当第6个比特位置为1时，表示行级锁。

- 行锁的具体类型 (`rec_lock_type`)，使用其余的位来表示。只有在 `lock_type` 的值为 `LOCK_REC` 时，也就是只有在该锁为行级锁时，才会被细分为更多的类型：

- `LOCK_ORDINARY` (十进制的 0)：表示 `next-key` 锁。
- `LOCK_GAP` (十进制的 512)：也就是当第10个比特位置为1时，表示 `gap` 锁。
- `LOCK_REC_NOT_GAP` (十进制的 1024)：也就是当第11个比特位置为1时，表示正经 `记录` 锁。
- `LOCK_INSERT_INTENTION` (十进制的 2048)：也就是当第12个比特位置为1时，表示插入意向锁。其他的类型：还有一些不常用的类型我们就不多说了。

- `is_waiting` 属性呢？基于内存空间的节省，所以把 `is_waiting` 属性放到了 `type_mode` 这个32位的数字中：

- `LOCK_WAIT` (十进制的 256)：当第9个比特位置为 1 时，表示 `is_waiting` 为 `true`，也就是当前事务尚未获取到锁，处在等待状态；当这个比特位为 0 时，表示 `is_waiting` 为 `false`，也就是当前事务获取锁成功。

5. 其他信息：

为了更好的管理系统运行过程中生成的各种锁结构而设计了各种哈希表和链表。

6. 一堆比特位：

如果是 `行锁结构` 的话，在该结构末尾还放置了一堆比特位，比特位的数量是由上边提到的 `n_bits` 属性表示的。InnoDB数据页中的每条记录在 `记录头信息` 中都包含一个 `heap_no` 属性，伪记录 `Infimum` 的 `heap_no` 值为 0，`Supremum` 的 `heap_no` 值为 1，之后每插入一条记录，`heap_no` 值就增1。`锁结构` 最后的一堆比特位就对应着一个页面中的记录，一个比特位映射一个 `heap_no`，即一个比特位映射到页内的一条记录。

5. 锁监控

关于MySQL锁的监控，我们一般可以通过检查 `InnoDB_row_lock` 等状态变量来分析系统上的行锁的竞争情况

```
mysql> show status like 'innodb_row_lock%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Innodb_row_lock_current_waits | 0 |
| Innodb_row_lock_time | 0 |
| Innodb_row_lock_time_avg | 0 |
| Innodb_row_lock_time_max | 0 |
| Innodb_row_lock_waits | 0 |
+-----+-----+
5 rows in set (0.01 sec)
```

对各个状态量的说明如下：

- `Innodb_row_lock_current_waits`: 当前正在等待锁定的数量；
- `Innodb_row_lock_time` : 从系统启动到现在锁定总时间长度； (等待总时长)
- `Innodb_row_lock_time_avg` : 每次等待所花平均时间； (等待平均时长)
- `Innodb_row_lock_time_max`: 从系统启动到现在等待最常的一次所花的时间；
- `Innodb_row_lock_waits` : 系统启动后到现在总共等待的次数； (等待总次数)

对于这5个状态变量，比较重要的3个见上面（橙色）。

其他监控方法：

MySQL把事务和锁的信息记录在了 `information_schema` 库中，涉及到的三张表分别是 `INNODB_TRX`、`INNODB_LOCKS` 和 `INNODB_LOCK_WAITS`。

MySQL5.7及之前，可以通过`information_schema.INNODB_LOCKS`查看事务的锁情况，但只能看到阻塞事务的锁；如果事务并未被阻塞，则在该表中看不到该事务的锁情况。

MySQL8.0删除了`information_schema.INNODB_LOCKS`，添加了 `performance_schema.data_locks`，可以通过`performance_schema.data_locks`查看事务的锁情况，和MySQL5.7及之前不同，`performance_schema.data_locks`不但可以看到阻塞该事务的锁，还可以看到该事务所持有的锁。

同时，`information_schema.INNODB_LOCK_WAITS`也被 `performance_schema.data_lock_waits` 所代替。

我们模拟一个锁等待的场景，以下是从这三张表收集的信息

锁等待场景，我们依然使用记录锁中的案例，当事务2进行等待时，查询情况如下：

- (1) 查询正在被锁阻塞的sql语句。

```
SELECT * FROM information_schema.INNODB_TRX\G;
```

重要属性代表含义已在上述中标注。

- (2) 查询锁等待情况

```
SELECT * FROM data_lock_waits\G;
***** 1. row *****
ENGINE: INNODB
REQUESTING_ENGINE_LOCK_ID: 139750145405624:7:4:7:139747028690608
REQUESTING_ENGINE_TRANSACTION_ID: 13845 #被阻塞的事务ID
```

```

REQUESTING_THREAD_ID: 72
REQUESTING_EVENT_ID: 26
REQUESTING_OBJECT_INSTANCE_BEGIN: 139747028690608
    BLOCKING_ENGINE_LOCK_ID: 139750145406432:7:4:7:139747028813248
    BLOCKING_ENGINE_TRANSACTION_ID: 13844 #正在执行的事务ID, 阻塞了13845
        BLOCKING_THREAD_ID: 71
        BLOCKING_EVENT_ID: 24
BLOCKING_OBJECT_INSTANCE_BEGIN: 139747028813248
1 row in set (0.00 sec)

```

(3) 查询锁的情况

```

mysql > SELECT * from performance_schema.data_locks\G;
*****
1. row ****
ENGINE: INNODB
ENGINE_LOCK_ID: 139750145405624:1068:139747028693520
ENGINE_TRANSACTION_ID: 13847
    THREAD_ID: 72
    EVENT_ID: 31
    OBJECT_SCHEMA: atguigu
    OBJECT_NAME: user
    PARTITION_NAME: NULL
    SUBPARTITION_NAME: NULL
    INDEX_NAME: NULL
OBJECT_INSTANCE_BEGIN: 139747028693520
    LOCK_TYPE: TABLE
    LOCK_MODE: IX
    LOCK_STATUS: GRANTED
    LOCK_DATA: NULL
*****
2. row ****
ENGINE: INNODB
ENGINE_LOCK_ID: 139750145405624:7:4:7:139747028690608
ENGINE_TRANSACTION_ID: 13847
    THREAD_ID: 72
    EVENT_ID: 31
    OBJECT_SCHEMA: atguigu
    OBJECT_NAME: user
    PARTITION_NAME: NULL
    SUBPARTITION_NAME: NULL
    INDEX_NAME: PRIMARY
OBJECT_INSTANCE_BEGIN: 139747028690608
    LOCK_TYPE: RECORD
    LOCK_MODE: X,REC_NOT_GAP
    LOCK_STATUS: WAITING
    LOCK_DATA: 1
*****
3. row ****
ENGINE: INNODB
ENGINE_LOCK_ID: 139750145406432:1068:139747028816304
ENGINE_TRANSACTION_ID: 13846
    THREAD_ID: 71
    EVENT_ID: 28
    OBJECT_SCHEMA: atguigu
    OBJECT_NAME: user
    PARTITION_NAME: NULL
    SUBPARTITION_NAME: NULL
    INDEX_NAME: NULL
OBJECT_INSTANCE_BEGIN: 139747028816304
    LOCK_TYPE: TABLE

```

```

LOCK_MODE: IX
LOCK_STATUS: GRANTED
LOCK_DATA: NULL
***** 4. row *****
ENGINE: INNODB
ENGINE_LOCK_ID: 139750145406432:7:4:7:139747028813248
ENGINE_TRANSACTION_ID: 13846
THREAD_ID: 71
EVENT_ID: 28
OBJECT_SCHEMA: atguigu
OBJECT_NAME: user
PARTITION_NAME: NULL
SUBPARTITION_NAME: NULL
INDEX_NAME: PRIMARY
OBJECT_INSTANCE_BEGIN: 139747028813248
LOCK_TYPE: RECORD
LOCK_MODE: X, REC_NOT_GAP
LOCK_STATUS: GRANTED
LOCK_DATA: 1
4 rows in set (0.00 sec)

ERROR:
No query specified

```

从锁的情况可以看出来，两个事务分别获取了IX锁，我们从意向锁章节可以知道，IX锁互相时兼容的。所以这里不会等待，但是事务1同样持有X锁，此时事务2也要去同一行记录获取X锁，他们之间不兼容，导致等待的情况发生。

6. 附录

间隙锁加锁规则（共11个案例）

间隙锁是在可重复读隔离级别下才会生效的：next-key lock 实际上是由间隙锁加行锁实现的，如果切换到读提交隔离级别 (read-committed) 的话，就好理解了，过程中去掉间隙锁的部分，也就是只剩下行锁的部分。而在读提交隔离级别下间隙锁就没有了，为了解决可能出现的数据和日志不一致问题，需要把 binlog 格式设置为 row 。也就是说，许多公司的配置为：读提交隔离级别加 binlog_format=row 。业务不需要可重复读的保证，这样考虑到读提交下操作数据的锁范围更小（没有间隙锁），这个选择是合理的。

next-key lock的加锁规则

总结的加锁规则里面，包含了两个““原则””、两个““优化””和一个“bug”。

1. 原则 1：加锁的基本单位是 next-key lock 。 next-key lock 是前开后闭区间。
2. 原则 2：查找过程中访问到的对象才会加锁。任何辅助索引上的锁，或者非索引列上的锁，最终都要回溯到主键上，在主键上也要加一把锁。
3. 优化 1：索引上的等值查询，给唯一索引加锁的时候，next-key lock 退化为行锁。也就是说如果 InnoDB 扫描的是一个主键、或是一个唯一索引的话，那 InnoDB 只会采用行锁方式来加锁
4. 优化 2：索引上（不一定是唯一索引）的等值查询，向右遍历时且最后一个值不满足等值条件的时候，next-keylock 退化为间隙锁。
5. 一个 bug：唯一索引上的范围查询会访问到不满足条件的第一个值为止。

我们以表test作为例子，建表语句和初始化语句如下：其中id为主键索引

```

CREATE TABLE `test` (
  `id` int(11) NOT NULL,
  `col1` int(11) DEFAULT NULL,
  `col2` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `c` (`c`)
) ENGINE=InnoDB;
insert into test values(0,0,0),(5,5,5),
(10,10,10),(15,15,15),(20,20,20),(25,25,25);

```

案例一：唯一索引等值查询间隙锁

sessionA	sessionB	sessionC
begin; update test set col2 = col2+1 where id=7;		
	insert into test values(8,8,8) (blocked)	
		update test set col2 = col2+1 where id=10; (Query OK)

由于表 test 中没有 id=7 的记录

根据原则 1，加锁单位是 next-key lock， session A 加锁范围就是 (5,10]； 同时根据优化 2，这是一个等值查询 (id=7)，而 id=10 不满足查询条件，next-key lock 退化成间隙锁，因此最终加锁的范围是 (5,10)

案例二：非唯一索引等值查询锁

sessionA	sessionB	sessionC
begin; select id from test where col1 = 5 lock in share mode;		
	update test col2 = col2+1 where id=5; (Query OK)	
		insert into test values(7,7,7) (blocked)

这里 session A 要给索引 col1 上 col1=5 的这一行加上读锁。

1. 根据原则 1，加锁单位是 next-key lock，左开右闭，5是闭上的，因此会给 (0,5] 加上 next-key lock。
2. 要注意 c 是普通索引，因此仅访问 c=5 这一条记录是不能马上停下来的（可能有 col1=5 的其他记录），需要向右遍历，查到 c=10 才放弃。根据原则 2，访问到的都要加锁，因此要给 (5,10] 加 next-key lock。
3. 但是同时这个符合优化 2：等值判断，向右遍历，最后一个值不满足 col1=5 这个等值条件，因此退化成间隙锁 (5,10)。

4. 根据原则 2，只有访问到的对象才会加锁，这个查询使用覆盖索引，并不需要访问主键索引，所以主键索引上没有加任何锁，这就是为什么 session B 的 update 语句可以执行完成。

但 session C 要插入一个 (7,7,7) 的记录，就会被 session A 的间隙锁 (5,10) 锁住。这个例子说明，锁是加在索引上的。

执行 for update 时，系统会认为你接下来要更新数据，因此会顺便给主键索引上满足条件的行加上行锁。

如果你要用 lock in share mode 来给行加读锁避免数据被更新的话，就必须得绕过覆盖索引的优化，因为覆盖索引不会访问主键索引，不会给主键索引上加锁。

案例三：主键索引范围查询锁

上面两个例子是等值查询的，这个例子是关于范围查询的，也就是说下面的语句

```
select * from test where id=10 for update
select * from test where id>=10 and id<11 for update;
```

这两条查语句肯定是等价的，但是它们的加锁规则不太一样

sessionA	sessionB	sessionC
begin; select * from test where id>= 10 and id<11 for update;		
	insert into test values(8,8,8) (Query OK) insert into test values(13,13,13); (blocked)	
		update test set col2=col2+1 where id=15; (blocked)

- 开始执行的时候，要找到第一个 id=10 的行，因此本该是 next-key lock(5,10]。根据优化 1，主键 id 上的等值条件，退化成行锁，只加了 id=10 这一行的行锁。
- 它是范围查询，范围查找就往后继续找，找到 id=15 这一行停下来，不满足条件，因此需要加 next-key lock(10,15]。

session A 这时候锁的范围就是主键索引上，行锁 id=10 和 next-key lock(10,15]。首次 session A 定位查找 id=10 的行的时候，是当做等值查询来判断的，而向右扫描到 id=15 的时候，用的是范围查询判断。

案例四：非唯一索引范围查询锁

与案例三不同的是，案例四中查询语句的 where 部分用的是字段 c，它是普通索引

这两条查语句肯定是等价的，但是它们的加锁规则不太一样

sessionA	sessionB	sessionC
begin; select * from test where col1>= 10 and col1<11 for update;		
	insert into test values(8,8,8)(blocked)	
		update test set clo2=col2+1 where id=15; (blocked)

在第一次用 col1=10 定位记录的时候，索引 c 上加了 (5,10] 这个 next-key lock 后，由于索引 col1 是非唯一索引，没有优化规则，也就是说不会蜕变为行锁，因此最终 session A 加的锁是，索引 c 上的 (5,10] 和 (10,15] 这两个 next-keylock。

这里需要扫描到 col1=15 才停止扫描，是合理的，因为 InnoDB 要扫到 col1=15，才知道不需要继续往后找了。

案例五：唯一索引范围查询锁 bug

sessionA	sessionB	sessionC
begin; select * from test where id> 10 and id<=15 for update;		
	update test set clo2=col2+1 where id=20; (blocked)	
		insert into test values(16,16,16); (blocked)

session A 是一个范围查询，按照原则 1 的话，应该是索引 id 上只加 (10,15] 这个 next-key lock，并且因为 id 是唯一键，所以循环判断到 id=15 这一行就应该停止了。

但是实现上，InnoDB 会往前扫描到第一个不满足条件的行为止，也就是 id=20。而且由于这是个范围扫描，因此索引 id 上的 (15,20] 这个 next-key lock 也会被锁上。照理说，这里锁住 id=20 这一行的行为，其实是没有必要的。因为扫描到 id=15，就可以确定不用往后再找了。

案例六：非唯一索引上存在 " " 等值 " " 的例子

这里，我给表 t 插入一条新记录：insert into t values(30,10,30);也就是说，现在表里面有两个c=10的行

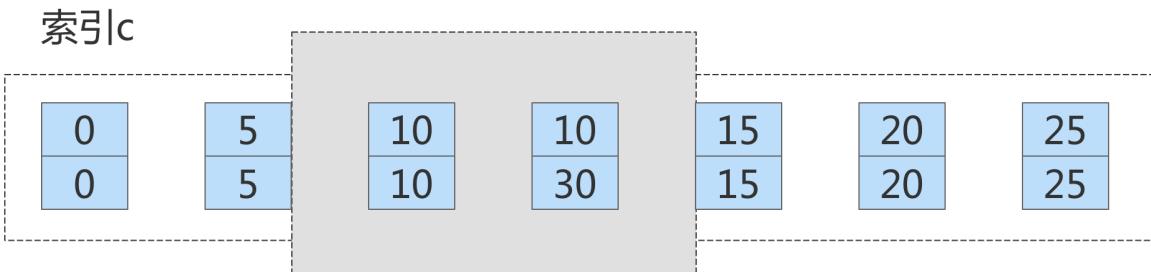
但是它们的主键值 id 是不同的（分别是 10 和 30），因此这两个c=10 的记录之间，也是有间隙的。

sessionA	sessionB	sessionC
begin; delete from test where col1=10;		
	insert into test values(12,12,12); (blocked)	
		update test set col2=col2+1 where c=15; (blocked)

这次我们用 delete 语句来验证。注意， delete 语句加锁的逻辑，其实跟 select ... for update 是类似的，也就是我在文章开始总结的两个“原则”、两个“优化”和一个“bug”。

这时， session A 在遍历的时候，先访问第一个 col1=10 的记录。同样地，根据原则 1， 这里加的是 (col1=5,id=5) 到 (col1=10,id=10) 这个 next-key lock 。

由于c是普通索引，所以继续向右查找，直到碰到 (col1=15,id=15) 这一行循环才结束。根据优化 2，这是一个等值查询，向右查找到了不满足条件的行，所以会退化成 (col1=10,id=10) 到 (col1=15,id=15) 的间隙锁。



这个 delete 语句在索引 c 上的加锁范围，就是上面图中蓝色区域覆盖的部分。这个蓝色区域左右两边都是虚线，表示开区间，即 (col1=5,id=5) 和 (col1=15,id=15) 这两行上都没有锁

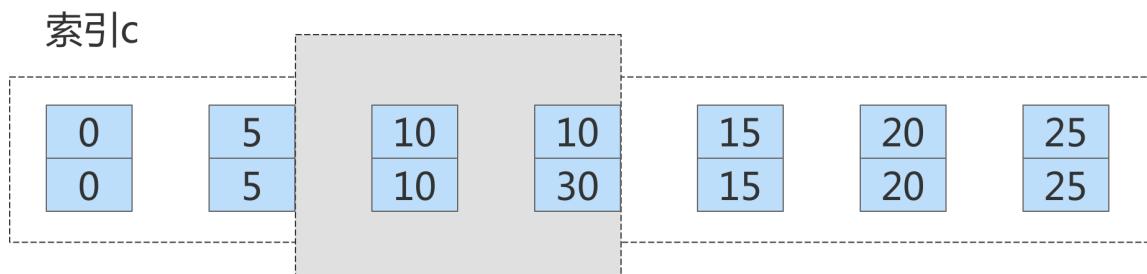
案例七： limit 语句加锁

例子 6 也有一个对照案例，场景如下所示：

sessionA	sessionB
begin; delete from test where col1=10 limit 2;	
	insert into test values(12,12,12); (Query OK)

session A 的 delete 语句加了 limit 2 。你知道表 t 里 c=10 的记录其实只有两条，因此加不加 limit 2 ，删除的效果都是一样的。但是加锁效果却不一样

这是因为，案例七里的 delete 语句明确加了 limit 2 的限制，因此在遍历到 (col1=10, id=30) 这一行之后，满足条件的语句已经有两条，循环就结束了。因此，索引 col1 上的加锁范围就变成了从 (col1=5,id=5) 到 (col1=10,id=30) 这个前开后闭区间，如下图所示：



这个例子对我们实践的指导意义就是，在删除数据的时候尽量加 limit。

这样不仅可以**控制删除数据的条数，让操作更安全，还可以减小加锁的范围。**

案例八：一个死锁的例子

sessionA	sessionB
begin; select id from test where col1=10 lock in share mode;	
	update test set col2=col2+1 where c=10; (blocked)
insert into test values(8,8,8);	
	ERROR 1213(40001):Deadlock found when trying to getlock;try restarting transaction

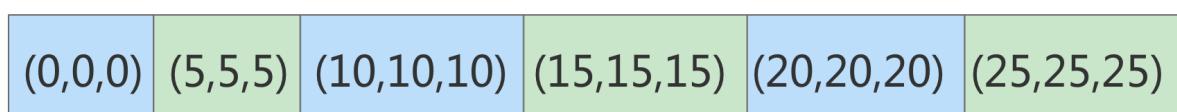
1. session A 启动事务后执行查询语句加 lock in share mode，在索引 col1 上加了 next-keylock(5,10] 和间隙锁 (10,15)（索引向右遍历退化为间隙锁）；
2. session B 的 update 语句也要在索引 c 上加 next-key lock(5,10]，进入锁等待；实际上分成了两步，先是加 (5,10) 的间隙锁，加锁成功；然后加 col1=10 的行锁，因为 sessionA 上已经给这行加上了读锁，此时申请死锁时会被阻塞
3. 然后 session A 要再插入 (8,8,8) 这一行，被 session B 的间隙锁锁住。由于出现了死锁，InnoDB 让 session B 回滚

案例九：order by索引排序的间隙锁1

如下面一条语句

```
begin;
select * from test where id>9 and id<12 order by id desc for update;
```

下图为这个表的索引id的示意图。



1. 首先这个查询语句的语义是 order by id desc，要拿到满足条件的所有行，优化器必须先找到“第一个 id<12 的值”。
2. 这个过程是通过索引树的搜索过程得到的，在引擎内部，其实是要找到 id=12 的这个值，只是最终没找到，但找到了 (10,15) 这个间隙。（id=15 不满足条件，所以 next-key lock 退化为了间隙锁 (10,

- 15)。)
- 然后向左遍历，在遍历过程中，就不是等值查询了，会扫描到 id=5 这一行，又因为区间是左开右闭的，所以会加一个next-key lock (0,5]。也就是说，在执行过程中，通过树搜索的方式定位记录的时候，用的是“等值查询”的方法。

案例十：order by索引排序的间隙锁2

sessionA	sessionB
<pre>begin; select * from test where col1>=15 and c<=20 order by col1 desc lock in share mode;</pre>	
	<pre>insert into test values(6,6,6); (blocked)</pre>

- 由于是 order by col1 desc，第一个要定位的是索引 col1 上“最右边的”col1=20 的行。这是一个非唯一索引的等值查询：

左开右闭区间，首先加上 next-key lock (15,20]。向右遍历，col1=25不满足条件，退化为间隙锁 所以会加上间隙锁(20,25) 和 next-key lock (15,20]。

- 在索引 col1 上向左遍历，要扫描到 col1=10 才停下来。同时又因为左开右闭区间，所以 next-key lock 会加到 (5,10]，这正是阻塞session B 的 insert 语句的原因。
- 在扫描过程中，col1=20、col1=15、col1=10 这三行都存在值，由于是 select *，所以会在主键 id 上加三个行锁。因此，session A 的 select 语句锁的范围就是：

 - 索引 col1 上 (5, 25)；
 - 主键索引上 id=15、20 两个行锁。

案例十一：update修改数据的例子-先插入后删除

sessionA	sessionB
<pre>begin; select col1 from test where col1>5 lock in share mode;</pre>	
	<pre>update test set col1=1 where col1=5 (Query OK) update test set col1=5 where col1=1; (blocked)</pre>

注意：根据 col1>5 查到的第一个记录是 col1=10，因此不会加 (0,5] 这个 next-key lock。

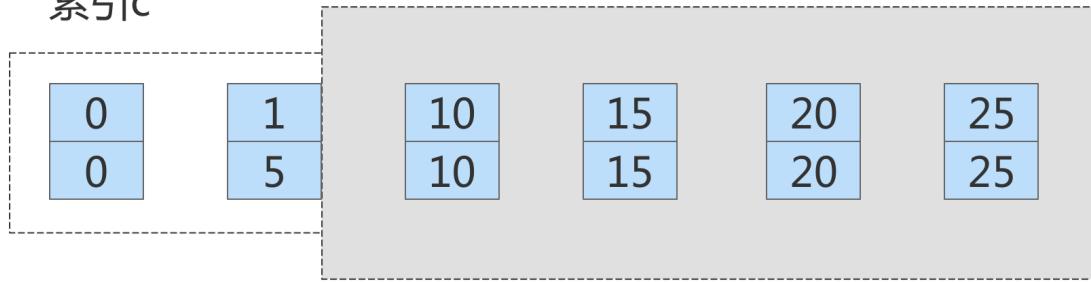
session A 的加锁范围是索引 col1 上的 (5,10]、(10,15]、(15,20]、(20,25] 和 (25,supremum]。

之后 session B 的第一个 update 语句，要把 col1=5 改成 col1=1，你可以理解为两步：

- 插入 (col1=1, id=5) 这个记录；
- 删除 (col1=5, id=5) 这个记录。

通过这个操作，session A 的加锁范围变成了图 7 所示的样子：

索引|c



好，接下来 session B 要执行 update t set col1 = 5 where col1 = 1 这个语句了，一样地可以拆成两步：

1. 插入 (col1=5, id=5) 这个记录；
2. 删除 (col1=1, id=5) 这个记录。第一步试图在已经加了间隙锁的 (1,10) 中插入数据，所以就被堵住了。

第16章_多版本并发控制

讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com> <http://www.atguigu.com/>

1. 什么是MVCC

MVCC (Multiversion Concurrency Control)，多版本并发控制。顾名思义，MVCC 是通过数据行的多个版本管理来实现数据库的并发控制。这项技术使得在InnoDB的事务隔离级别下执行一致性读操作有了保证。换言之，就是为了查询一些正在被另一个事务更新的行，并且可以看到它们被更新之前的值，这样在做查询的时候就不用等待另一个事务释放锁。

2. 快照读与当前读

MVCC在MySQL InnoDB中的实现主要是为了提高数据库并发性能，用更好的方式去处理读-写冲突，做到即使有读写冲突时，也能做到不加锁，非阻塞并发读，而这个读指的就是快照读，而非当前读。当前读实际上是一种加锁的操作，是悲观锁的实现。而MVCC本质是采用乐观锁思想的一种方式。

2.1 快照读

快照读又叫一致性读，读取的是快照数据。**不加锁的简单的SELECT都属于快照读**，即不加锁的非阻塞读；比如这样：

```
SELECT * FROM player WHERE ...
```

之所以出现快照读的情况，是基于提高并发性能的考虑，快照读的实现是基于MVCC，它在很多情况下，避免了加锁操作，降低了开销。

既然是基于多版本，那么快照读可能读到的并不一定是数据的最新版本，而有可能是之前的历史版本。

快照读的前提是隔离级别不是串行级别，串行级别下的快照读会退化成当前读。

2.2 当前读

当前读读取的是记录的最新版本（最新数据，而不是历史版本的数据），读取时还要保证其他并发事务不能修改当前记录，会对读取的记录进行加锁。加锁的SELECT，或者对数据进行增删改都会进行当前读。比如：

```
SELECT * FROM student LOCK IN SHARE MODE; # 共享锁
```

```
SELECT * FROM student FOR UPDATE; # 排他锁
```

```
INSERT INTO student values ... # 排他锁
```

```
DELETE FROM student WHERE ... # 排他锁
```

```
UPDATE student SET ... # 排他锁
```

3. 复习

3.1 再谈隔离级别

我们知道事务有 4 个隔离级别，可能存在三种并发问题：



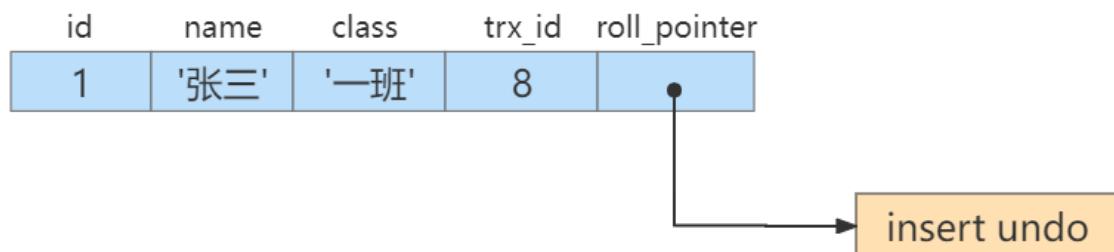
另图：



3.2 隐藏字段、Undo Log版本链

回顾一下undo日志的版本链，对于使用 InnoDB 存储引擎的表来说，它的聚簇索引记录中都包含两个必要的隐藏列。

- **trx_id**：每次一个事务对某条聚簇索引记录进行改动时，都会把该事务的 **事务 id** 赋值给 **trx_id** 隐藏列。
- **roll_pointer**：每次对某条聚簇索引记录进行改动时，都会把旧的版本写入到 **undo日志** 中，然后这个隐藏列就相当于一个指针，可以通过它来找到该记录修改前的信息。

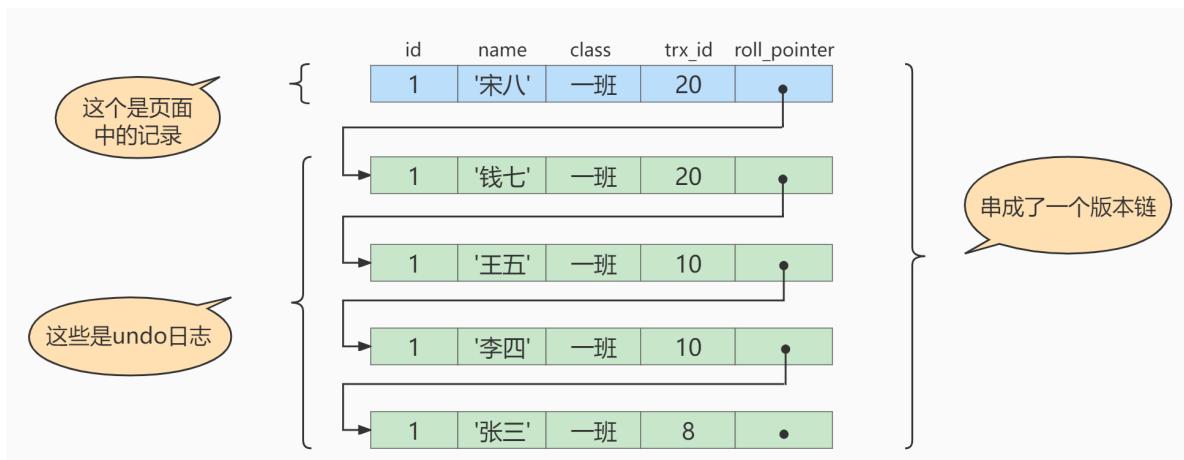


insert undo只在事务回滚时起作用，当事务提交后，该类型的undo日志就没用了，它占用的Undo Log Segment也会被系统回收（也就是该undo日志占用的Undo页面链表要么被重用，要么被释放）。

假设之后两个事务id分别为 10、20 的事务对这条记录进行 UPDATE 操作，操作流程如下：

发生时间顺序	事务10	事务20
1	BEGIN;	
2		BEGIN;
3	UPDATE student SET name="李四" WHERE id=1;	
4	UPDATE student SET name="王五" WHERE id=1;	
5	COMMIT;	
6		UPDATE student SET name="钱七" WHERE id=1;
7		UPDATE student SET name="宋八" WHERE id=1;
8		COMMIT;

每次对记录进行改动，都会记录一条undo日志，每条undo日志也都有一个 roll_pointer 属性（ INSERT 操作对应的undo日志没有该属性，因为该记录并没有更早的版本），可以将这些 undo 日志都连起来，串成一个链表：



对该记录每次更新后，都会将旧值放到一条 undo 日志 中，就算是该记录的一个旧版本，随着更新次数的增多，所有的版本都会被 roll_pointer 属性连接成一个链表，我们把这个链表称之为 版本链，版本链的头节点就是当前记录最新的值。

每个版本中还包含生成该版本时对应的 事务id 。

4. MVCC实现原理之ReadView

MVCC 的实现依赖于： **隐藏字段、Undo Log、Read View**。

4.1 什么是ReadView

4.2 设计思路

使用 `READ UNCOMMITTED` 隔离级别的事务，由于可以读到未提交事务修改过的记录，所以直接读取记录的最新版本就好了。

使用 `SERIALIZABLE` 隔离级别的事务，InnoDB规定使用加锁的方式来访问记录。

使用 `READ COMMITTED` 和 `REPEATABLE READ` 隔离级别的事务，都必须保证读到 `已经提交了的` 事务修改过的记录。假如另一个事务已经修改了记录但是尚未提交，是不能直接读取最新版本的记录的，核心问题就是需要判断一下版本链中的哪个版本是当前事务可见的，这是ReadView要解决的主要问题。

这个ReadView中主要包含4个比较重要的内容，分别如下：

1. `creator_trx_id`，创建这个 Read View 的事务 ID。

说明：只有在对表中的记录做改动时（执行`INSERT`、`DELETE`、`UPDATE`这些语句时）才会为事务分配事务id，否则在一个只读事务中的事务id值都默认为0。

2. `trx_ids`，表示在生成ReadView时当前系统中活跃的读写事务的 `事务id列表`。
3. `up_limit_id`，活跃的事务中最小的事务 ID。
4. `low_limit_id`，表示生成ReadView时系统中应该分配给下一个事务的 `id` 值。`low_limit_id` 是系统最大的事务id值，这里要注意是系统中的事务id，需要区别于正在活跃的事务ID。

注意：`low_limit_id`并不是`trx_ids`中的最大值，事务id是递增分配的。比如，现在有id为1, 2, 3这三个事务，之后id为3的事务提交了。那么一个新的读事务在生成ReadView时，`trx_ids`就包括1和2，`up_limit_id`的值就是1，`low_limit_id`的值就是4。

4.3 ReadView的规则

有了这个ReadView，这样在访问某条记录时，只需要按照下边的步骤判断记录的某个版本是否可见。

- 如果被访问版本的`trx_id`属性值与ReadView中的 `creator_trx_id` 值相同，意味着当前事务在访问它自己修改过的记录，所以该版本可以被当前事务访问。
- 如果被访问版本的`trx_id`属性值小于ReadView中的 `up_limit_id` 值，表明生成该版本的事务在当前事务生成ReadView前已经提交，所以该版本可以被当前事务访问。
- 如果被访问版本的`trx_id`属性值大于或等于ReadView中的 `low_limit_id` 值，表明生成该版本的事务在当前事务生成ReadView后才开启，所以该版本不可以被当前事务访问。
- 如果被访问版本的`trx_id`属性值在ReadView的 `up_limit_id` 和 `low_limit_id` 之间，那就需要判断一下`trx_id`属性值是不是在 `trx_ids` 列表中。
 - 如果在，说明创建ReadView时生成该版本的事务还是活跃的，该版本不可以被访问。
 - 如果不在，说明创建ReadView时生成该版本的事务已经被提交，该版本可以被访问。

4.4 MVCC整体操作流程

了解了这些概念之后，我们来看下当查询一条记录的时候，系统如何通过MVCC找到它：

1. 首先获取事务自己的版本号，也就是事务 ID；
2. 获取 ReadView；
3. 查询得到的数据，然后与 ReadView 中的事务版本号进行比较；
4. 如果不符合 ReadView 规则，就需要从 Undo Log 中获取历史快照；
5. 最后返回符合规则的数据。

在隔离级别为读已提交（Read Committed）时，一个事务中的每一次 SELECT 查询都会重新获取一次 Read View。

如表所示：

事务	说明
begin;	
select * from student where id >2;	获取一次Read View
.....	
select * from student where id >2;	获取一次Read View
commit;	

注意，此时同样的查询语句都会重新获取一次 Read View，这时如果 Read View 不同，就可能产生不可重复读或者幻读的情况。

当隔离级别为可重复读的时候，就避免了不可重复读，这是因为一个事务只在第一次 SELECT 的时候会获取一次 Read View，而后面所有的 SELECT 都会复用这个 Read View，如下表所示：

事务	说明
begin;	
select * from user where id >2;	
.....	获取一次Read View
select * from user where id >2;	
commit;	

5. 举例说明

5.1 READ COMMITTED隔离级别下

READ COMMITTED：每次读取数据前都生成一个ReadView。

现在有两个 事务id 分别为 10、20 的事务在执行：

```

# Transaction 10
BEGIN;
UPDATE student SET name="李四" WHERE id=1;
UPDATE student SET name="王五" WHERE id=1;

# Transaction 20
BEGIN;
# 更新了一些别的表的记录
...

```

此刻，表student中 `id` 为 1 的记录得到的版本链表如下所示：



假设现在有一个使用 `READ COMMITTED` 隔离级别的事务开始执行：

```

# 使用READ COMMITTED隔离级别的事务
BEGIN;

# SELECT1: Transaction 10、20未提交
SELECT * FROM student WHERE id = 1; # 得到的列name的值为'张三'

```

之后，我们把 `事务id` 为 10 的事务提交一下：

```

# Transaction 10
BEGIN;

UPDATE student SET name="李四" WHERE id=1;
UPDATE student SET name="王五" WHERE id=1;

COMMIT;

```

然后再对 `事务id` 为 20 的事务中更新一下表 `student` 中 `id` 为 1 的记录：

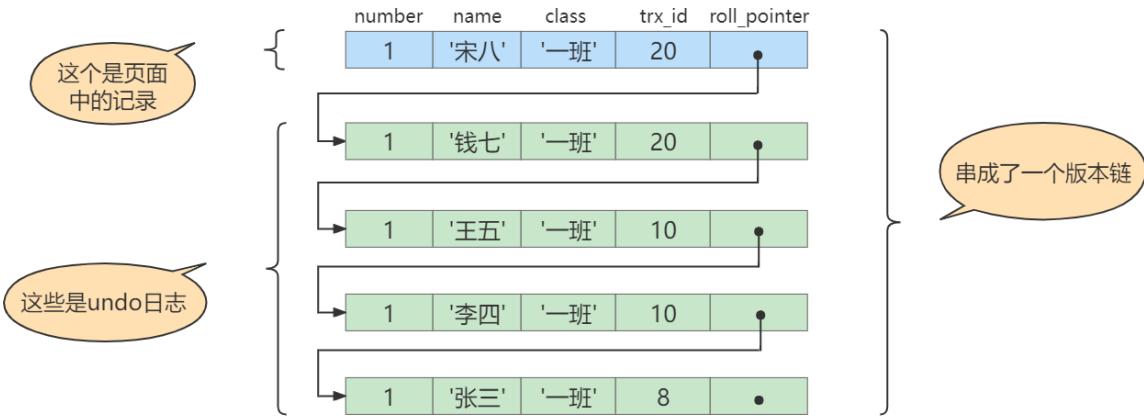
```

# Transaction 20
BEGIN;

# 更新了一些别的表的记录
...
UPDATE student SET name="钱七" WHERE id=1;
UPDATE student SET name="宋八" WHERE id=1;

```

此刻，表student中 `id` 为 1 的记录的版本链就长这样：



然后再到刚才使用 `READ COMMITTED` 隔离级别的事务中继续查找这个 `id` 为 1 的记录，如下：

```
# 使用READ COMMITTED隔离级别的事务
BEGIN;

# SELECT1: Transaction 10、20均未提交
SELECT * FROM student WHERE id = 1; # 得到的列name的值为'张三'

# SELECT2: Transaction 10提交, Transaction 20未提交
SELECT * FROM student WHERE id = 1; # 得到的列name的值为'王五'
```

5.2 REPEATABLE READ隔离级别下

使用 `REPEATABLE READ` 隔离级别的事务来说，只会在第一次执行查询语句时生成一个 `ReadView`，之后的查询就不会重复生成了。

比如，系统里有两个 事务 `id` 分别为 10、20 的事务在执行：

```
# Transaction 10
BEGIN;
UPDATE student SET name="李四" WHERE id=1;
UPDATE student SET name="王五" WHERE id=1;

# Transaction 20
BEGIN;
# 更新了一些别的表的记录
...
```

此刻，表 `student` 中 `id` 为 1 的记录得到的版本链表如下所示：



假设现在有一个使用 `REPEATABLE READ` 隔离级别的事务开始执行：

```
# 使用REPEATABLE READ隔离级别的事务
BEGIN;

# SELECT1: Transaction 10、20未提交
SELECT * FROM student WHERE id = 1; # 得到的列name的值为'张三'
```

之后，我们把 事务id 为 10 的事务提交一下，就像这样：

```
# Transaction 10
BEGIN;

UPDATE student SET name="李四" WHERE id=1;
UPDATE student SET name="王五" WHERE id=1;

COMMIT;
```

然后再到 事务id 为 20 的事务中更新一下表 student 中 id 为 1 的记录：

```
# Transaction 20
BEGIN;

# 更新了一些别的表的记录
...
UPDATE student SET name="钱七" WHERE id=1;
UPDATE student SET name="宋八" WHERE id=1;
```

此刻，表student 中 id 为 1 的记录的版本链长这样：



然后再到刚才使用 REPEATABLE READ 隔离级别的事务中继续查找这个 id 为 1 的记录，如下：

```
# 使用REPEATABLE READ隔离级别的事务
BEGIN;

# SELECT1: Transaction 10、20均未提交
SELECT * FROM student WHERE id = 1; # 得到的列name的值为'张三'

# SELECT2: Transaction 10提交, Transaction 20未提交
SELECT * FROM student WHERE id = 1; # 得到的列name的值仍为'张三'
```

5.3 如何解决幻读

接下来说明InnoDB是如何解决幻读的。

假设现在表 student 中只有一条数据，数据内容中，主键 id=1，隐藏的 trx_id=10，它的 undo log 如下图所示。

trx_id = 10	数据 id=1,name=张三	NULL
-------------	--------------------	------

假设现在有事务 A 和事务 B 并发执行，事务 A 的事务 id 为 20，事务 B 的事务 id 为 30。

步骤1：事务 A 开始第一次查询数据，查询的 SQL 语句如下。

```
select * from student where id >= 1;
```

在开始查询之前，MySQL 会为事务 A 产生一个 ReadView，此时 ReadView 的内容如下：`trx_ids=[20,30]`，`up_limit_id=20`，`low_limit_id=31`，`creator_trx_id=20`。

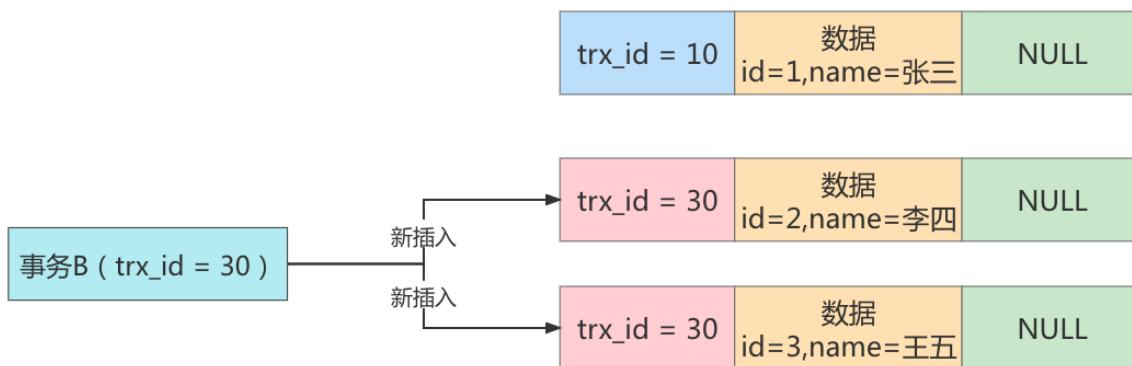
由于此时表 student 中只有一条数据，且符合 where id>=1 条件，因此会查询出来。然后根据 ReadView 机制，发现该行数据的 trx_id=10，小于事务 A 的 ReadView 里 up_limit_id，这表示这条数据是事务 A 开启之前，其他事务就已经提交了的数据，因此事务 A 可以读取到。

结论：事务 A 的第一次查询，能读取到一条数据，id=1。

步骤2：接着事务 B(trx_id=30)，往表 student 中新插入两条数据，并提交事务。

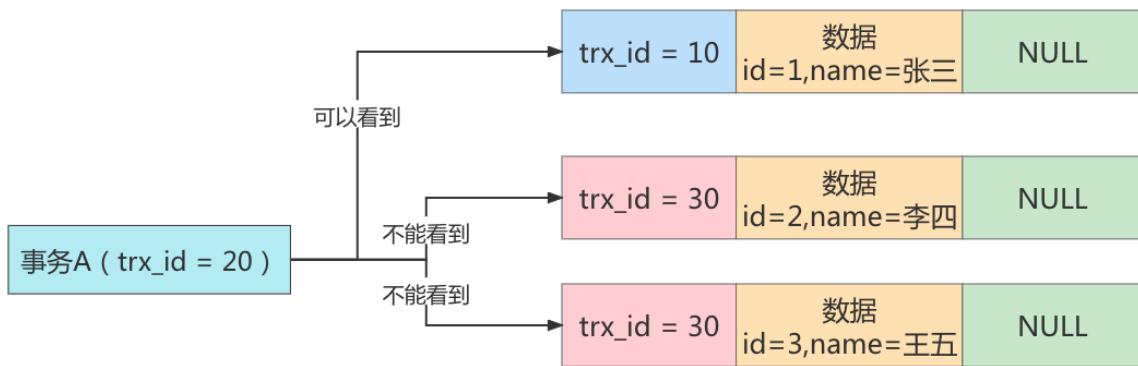
```
insert into student(id,name) values(2,'李四');  
insert into student(id,name) values(3,'王五');
```

此时表 student 中就有三条数据了，对应的 undo 如下图所示：



步骤3：接着事务 A 开启第二次查询，根据可重复读隔离级别的规则，此时事务 A 并不会再重新生成 ReadView。此时表 student 中的 3 条数据都满足 where id>=1 的条件，因此会先查出来。然后根据 ReadView 机制，判断每条数据是不是都可以被事务 A 看到。

- 1) 首先 id=1 的这条数据，前面已经说过了，可以被事务 A 看到。
- 2) 然后是 id=2 的数据，它的 trx_id=30，此时事务 A 发现，这个值处于 up_limit_id 和 low_limit_id 之间，因此还需要再判断 30 是否处于 trx_ids 数组内。由于事务 A 的 trx_ids=[20,30]，因此在数组内，这表示 id=2 的这条数据是与事务 A 在同一时刻启动的其他事务提交的，所以这条数据不能让事务 A 看到。
- 3) 同理，id=3 的这条数据，trx_id 也为 30，因此也不能被事务 A 看见。



结论：最终事务 A 的第二次查询，只能查询出 id=1 的这条数据。这和事务 A 的第一次查询的结果是一样的，因此没有出现幻读现象，所以说在 MySQL 的可重复读隔离级别下，不存在幻读问题。

6. 总结

这里介绍了 MVCC 在 READ COMMITTED、REPEATABLE READ 这两种隔离级别的事务在执行快照读操作时访问记录的版本链的过程。这样使不同事务的 读-写、写-读 操作并发执行，从而提升系统性能。

核心点在于 ReadView 的原理，READ COMMITTED、REPEATABLE READ 这两个隔离级别的一个很大不同就是生成ReadView的时机不同：

- READ COMMITTED 在每一次进行普通SELECT操作前都会生成一个ReadView
- REPEATABLE READ 只在第一次进行普通SELECT操作前生成一个ReadView，之后的查询操作都重复使用这个ReadView就好了。

第17章_其他数据库日志

讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

千万不要小看日志。很多看似奇怪的问题，答案往往就藏在日志里。很多情况下，只有通过查看日志才能发现问题的原因，真正解决问题。所以，一定要学会查看日志，养成检查日志的习惯，对提升你的数据库应用开发能力至关重要。

MySQL8.0 官网日志地址：“<https://dev.mysql.com/doc/refman/8.0/en/server-logs.html>”

1. MySQL支持的日志

1.1 日志类型

MySQL有不同类型的日志文件，用来存储不同类型的日志，分为 **二进制日志**、**错误日志**、**通用查询日志** 和 **慢查询日志**，这也是常用的4种。MySQL 8又新增两种支持的日志：**中继日志** 和 **数据定义语句日志**。使用这些日志文件，可以查看MySQL内部发生的事情。

这6类日志分别为：

- **慢查询日志：**记录所有执行时间超过long_query_time的所有查询，方便我们对查询进行优化。
- **通用查询日志：**记录所有连接的起始时间和终止时间，以及连接发送给数据库服务器的所有指令，对我们复原操作的实际场景、发现问题，甚至是审计都有很大的帮助。
- **错误日志：**记录MySQL服务的启动、运行或停止MySQL服务时出现的问题，方便我们了解服务器的状态，从而对服务器进行维护。
- **二进制日志：**记录所有更改数据的语句，可以用于主从服务器之间的数据同步，以及服务器遇到故障时数据的无损失恢复。
- **中继日志：**用于主从服务器架构中，从服务器用来存放主服务器二进制日志内容的一个中间文件。从服务器通过读取中继日志的内容，来同步主服务器上的操作。
- **数据定义语句日志：**记录数据定义语句执行的元数据操作。

除二进制日志外，其他日志都是 **文本文件**。默认情况下，所有日志创建于 **MySQL数据目录** 中。

1.2 日志的弊端

- 日志功能会 **降低MySQL数据库的性能**。
- 日志会 **占用大量的磁盘空间**。

2. 慢查询日志(slow query log)

前面章节《第09章_性能分析工具的使用》已经详细讲述。

3. 通用查询日志(general query log)

通用查询日志用来 [记录用户的所有操作](#)，包括启动和关闭MySQL服务、所有用户的连接开始时间和截止时间、发给 MySQL 数据库服务器的所有 SQL 指令等。当我们的数据发生异常时，[查看通用查询日志](#)，[还原操作时的具体场景](#)，可以帮助我们准确定位问题。

3.1 问题场景

3.2 查看当前状态

```
mysql> SHOW VARIABLES LIKE '%general%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| general_log    | OFF   | #通用查询日志处于关闭状态
| general_log_file | /var/lib/mysql/atguigu01.log | #通用查询日志文件的名称是atguigu01.log
+-----+-----+
2 rows in set (0.03 sec)
```

3.3 启动日志

方式1：永久性方式

修改my.cnf或者my.ini配置文件来设置。在[mysqld]组下加入log选项，并重启MySQL服务。格式如下：

```
[mysqld]
general_log=ON
general_log_file=[path[filename]] #日志文件所在目录路径, filename为日志文件名
```

如果不指定目录和文件名，通用查询日志将默认存储在MySQL数据目录中的hostname.log文件中，hostname表示主机名。

方式2：临时性方式

```
SET GLOBAL general_log=on; # 开启通用查询日志
```

```
SET GLOBAL general_log_file='path/filename'; # 设置日志文件保存位置
```

对应的，关闭操作SQL命令如下：

```
SET GLOBAL general_log=off; # 关闭通用查询日志
```

查看设置后情况：

```
SHOW VARIABLES LIKE 'general_log%';
```

3.4 查看日志

通用查询日志是以 [文本文件](#) 的形式存储在文件系统中的，可以使用 [文本编辑器](#) 直接打开日志文件。每台 MySQL服务器的通用查询日志内容是不同的。

- 在Windows操作系统中，使用文本文件查看器；
- 在Linux系统中，可以使用vi工具或者gedit工具查看；
- 在Mac OSX系统中，可以使用文本文件查看器或者vi等工具查看。

从 `SHOW VARIABLES LIKE 'general_log%';` 结果中可以看到通用查询日志的位置。

```

/usr/sbin/mysqld, Version: 8.0.26 (MySQL Community Server - GPL). started with:
Tcp port: 3306 Unix socket: /var/lib/mysql/mysql.sock
Time           Id Command   Argument
2022-01-04T07:44:58.052890Z      10 Query    SHOW VARIABLES LIKE '%general%'
2022-01-04T07:45:15.666672Z      10 Query    SHOW VARIABLES LIKE 'general_log%'
2022-01-04T07:45:28.970765Z      10 Query    select * from student
2022-01-04T07:47:38.706804Z      11 Connect  root@localhost on using Socket
2022-01-04T07:47:38.707435Z      11 Query    select @@version_comment limit 1
2022-01-04T07:48:21.384886Z      12 Connect  root@172.16.210.1 on using TCP/IP
2022-01-04T07:48:21.385253Z      12 Query    SET NAMES utf8
2022-01-04T07:48:21.385640Z      12 Query    USE `atguigu12`
2022-01-04T07:48:21.386179Z      12 Query    SHOW FULL TABLES WHERE Table_Type != 'VIEW'
2022-01-04T07:48:23.901778Z      13 Connect  root@172.16.210.1 on using TCP/IP
2022-01-04T07:48:23.902128Z      13 Query    SET NAMES utf8
2022-01-04T07:48:23.905179Z      13 Query    USE `atguigu`
2022-01-04T07:48:23.905825Z      13 Query    SHOW FULL TABLES WHERE Table_Type != 'VIEW'
2022-01-04T07:48:32.163833Z      14 Connect  root@172.16.210.1 on using TCP/IP
2022-01-04T07:48:32.164451Z      14 Query    SET NAMES utf8
2022-01-04T07:48:32.164840Z      14 Query    USE `atguigu`
2022-01-04T07:48:40.006687Z      14 Query    select * from account

```

在通用查询日志里面，我们可以清楚地看到，什么时候开启了新的客户端登陆数据库，登录之后做了什么SQL操作，针对的是哪个数据表等信息。

3.5 停止日志

方式1：永久性方式

修改 `my.cnf` 或者 `my.ini` 文件，把[mysqld]组下的 `general_log` 值设置为 `OFF` 或者把`general_log`一项注释掉。修改保存后，再 **重启MySQL服务**，即可生效。举例1：

```
[mysqld]
general_log=OFF
```

举例2：

```
[mysqld]
#general_log=ON
```

方式2：临时性方式

使用SET语句停止MySQL通用查询日志功能：

```
SET GLOBAL general_log=off;
```

查询通用日志功能：

```
SHOW VARIABLES LIKE 'general_log';
```

3.6 删除\刷新日志

如果数据的使用非常频繁，那么通用查询日志会占用服务器非常大的磁盘空间。数据管理员可以删除很长时间之前的查询日志，以保证MySQL服务器上的硬盘空间。

手动删除文件

```
SHOW VARIABLES LIKE 'general_log%';
```

可以看出，通用查询日志的目录默认为MySQL数据目录。在该目录下手动删除通用查询日志 atguigu01.log。

使用如下命令重新生成查询日志文件，具体命令如下。刷新MySQL数据目录，发现创建了新的日志文件。前提一定要开启通用日志。

```
mysqladmin -uroot -p flush-logs
```

4. 错误日志(error log)

4.1 启动日志

在MySQL数据库中，错误日志功能是 **默认开启** 的。而且，错误日志 **无法被禁止**。

默认情况下，错误日志存储在MySQL数据库的数据文件夹下，名称默认为 `mysqld.log` (Linux系统) 或 `hostname.err` (mac系统)。如果需要制定文件名，则需要在my.cnf或者my.ini中做如下配置：

```
[mysqld]
log-error=[path/[filename]] #path为日志文件所在的目录路径, filename为日志文件名
```

修改配置项后，需要重启MySQL服务以生效。

4.2 查看日志

MySQL错误日志是以文本文件形式存储的，可以使用文本编辑器直接查看。

查询错误日志的存储路径：

```
mysql> SHOW VARIABLES LIKE 'log_err%';
+-----+-----+
| Variable_name      | Value
+-----+-----+
| log_error          | /var/log/mysqld.log
| log_error_services | log_filter_internal; log_sink_internal
| log_error_suppression_list |
| log_error_verbosity | 2
+-----+-----+
4 rows in set (0.01 sec)
```

执行结果中可以看到错误日志文件是mysqld.log，位于MySQL默认的数据目录下。

4.3 删除\刷新日志

对于很久以前的错误日志，数据库管理员查看这些错误日志的可能性不大，可以将这些错误日志删除，以保证MySQL服务器上的 **硬盘空间**。MySQL的错误日志是以文本文件的形式存储在文件系统中的，可以 **直接删除**。

```
[root@atguigu01 log]# mysqladmin -uroot -p flush-logs
Enter password:
mysqladmin: refresh failed; error: 'Could not open file '/var/log/mysqld.log' for
error logging.'
```

官网提示：

Note

For the server to recreate a given log file after you have renamed the file externally, the file location must be writable by the server. This may not always be the case. For example, on Linux, the server might write the error log as /var/log/mysqld.log, where /var/log is owned by root and not writable by mysqld. In this case, log-flushing operations fail to create a new log file.

To handle this situation, you must manually create the new log file with the proper ownership after renaming the original log file. For example, execute these commands as root:

```
mv /var/log/mysqld.log /var/log/mysqld.log.old  
install -omysql -gmysql -m0644 /dev/null /var/log/mysqld.log
```

补充操作：

```
install -omysql -gmysql -m0644 /dev/null /var/log/mysqld.log
```

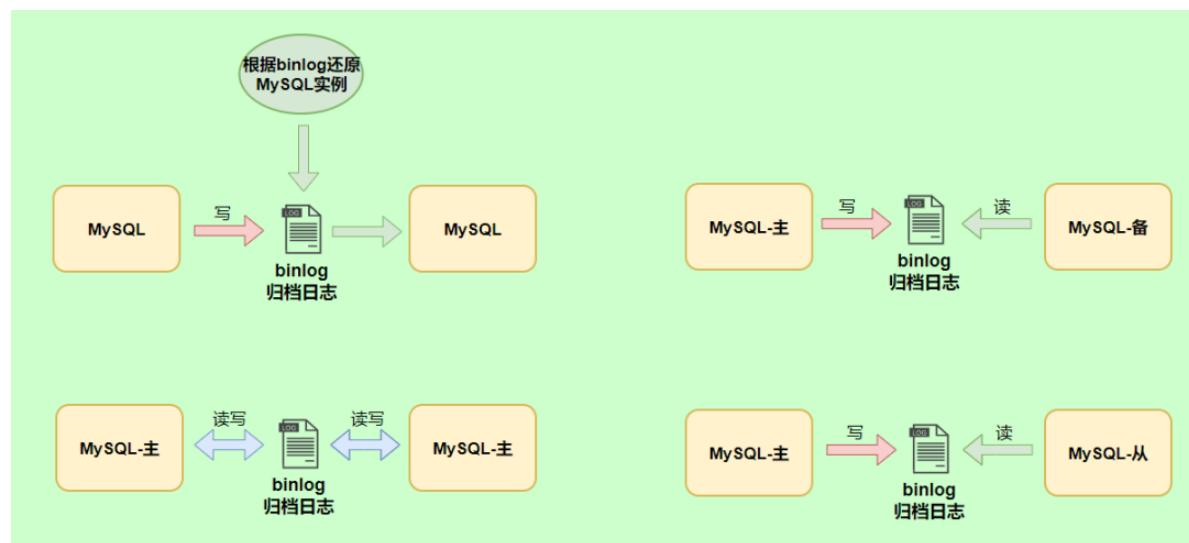
5. 二进制日志(bin log)

binlog可以说是MySQL中比较重要的日志了，在日常开发及运维过程中，经常会遇到。

binlog即binary log，二进制日志文件，也叫作变更日志（update log）。它记录了数据库所有执行的 DDL 和 DML 等数据库更新事件的语句，但是不包含没有修改任何数据的语句（如数据查询语句select、show等）。

binlog主要应用场景：

- 一是用于数据恢复
- 二是用于数据复制



5.1 查看默认情况

查看记录二进制日志是否开启：在MySQL8中默认情况下，二进制文件是开启的。

```
mysql> show variables like '%log_bin%';
+-----+-----+
| Variable_name          | Value   |
+-----+-----+
| log_bin                 | ON      |
| log_bin_basename        | /var/lib/mysql/binlog |
| log_bin_index           | /var/lib/mysql/binlog.index |
| log_bin_trust_function_creators | OFF |
| log_bin_use_v1_row_events | OFF |
| sql_log_bin              | ON      |
+-----+-----+
6 rows in set (0.00 sec)
```

5.2 日志参数设置

方式1：永久性方式

修改MySQL的 `my.cnf` 或 `my.ini` 文件可以设置二进制日志的相关参数：

```
[mysqld]
#启用二进制日志
log-bin=atguigu-bin
binlog_expire_logs_seconds=600
max_binlog_size=100M
```

重新启动MySQL服务，查询二进制日志的信息，执行结果：

```
mysql> show variables like '%log_bin%';
+-----+-----+
| Variable_name          | Value   |
+-----+-----+
| log_bin                 | ON      |
| log_bin_basename        | /var/lib/mysql/atguigu-bin |
| log_bin_index           | /var/lib/mysql/atguigu-bin.index |
| log_bin_trust_function_creators | OFF |
| log_bin_use_v1_row_events | OFF |
| sql_log_bin              | ON      |
+-----+-----+
6 rows in set (0.00 sec)
```

设置带文件夹的bin-log日志存放目录

如果想改变日志文件的目录和名称，可以对`my.cnf`或`my.ini`中的`log_bin`参数修改如下：

```
[mysqld]
log-bin="/var/lib/mysql/binlog/atguigu-bin"
```

注意：新建的文件夹需要使用`mysql`用户，使用下面的命令即可。

```
chown -R -v mysql:mysql binlog
```

方式2：临时性方式

如果不希望通过修改配置文件并重启的方式设置二进制日志的话，还可以使用如下指令，需要注意的是在MySQL8中只有 `会话级别` 的设置，没有了global级别的设置。

```

# global 级别
mysql> set global sql_log_bin=0;
ERROR 1228 (HY000): Variable 'sql_log_bin' is a SESSION variable and can't be used
with SET GLOBAL

# session级别
mysql> SET sql_log_bin=0;
Query OK, 0 rows affected (0.01 秒)

```

5.3 查看日志

当MySQL创建二进制日志文件时，先创建一个以“filename”为名称、以“.index”为后缀的文件，再创建一个以“filename”为名称、以“.000001”为后缀的文件。

MySQL服务 重新启动一次，以“.000001”为后缀的文件就会增加一个，并且后缀名按1递增。即日志文件的个数与MySQL服务启动的次数相同；如果日志长度超过了 `max_binlog_size` 的上限（默认是1GB），就会创建一个新的日志文件。

查看当前的二进制日志文件列表及大小。指令如下：

```

mysql> SHOW BINARY LOGS;
+-----+-----+-----+
| Log_name      | File_size | Encrypted |
+-----+-----+-----+
| atguigu-bin.000001 | 156       | No        |
+-----+-----+-----+
1 行于数据集 (0.02 秒)

```

下面命令将行事件以 伪SQL的形式 表现出来

```

mysqlbinlog -v "/var/lib/mysql/binlog/atguigu-bin.000002"
#2020105 9:16:37 server id 1 end_log_pos 324 CRC32 0x6b31978b  Query    thread_id=10
exec_time=0      error_code=0
SET TIMESTAMP=1641345397/*!*/;
SET @@session.pseudo_thread_id=10/*!*/;
SET @@session.foreign_key_checks=1, @@session.sql_auto_is_null=0,
@@session.unique_checks=1, @@session.autocommit=1/*!*/;
SET @@session.sql_mode=1168113696/*!*/;
SET @@session.auto_increment_increment=1, @@session.auto_increment_offset=1/*!*/;
/*!\\C utf8mb3 *//*!*/;
SET
@@session.character_set_client=33,@@session.collation_connection=33,@@session.collation_n_server=255/*!*/;
SET @@session.lc_time_names=0/*!*/;
SET @@session.collation_database=DEFAULT/*!*/;
/*!80011 SET @@session.default_collation_for_utf8mb4=255/*!*/;
BEGIN
/*!*/;
# at 324
#2020105 9:16:37 server id 1 end_log_pos 391 CRC32 0x74f89890  Table_map:
`atguigu14`.`student` mapped to number 85
# at 391
#2020105 9:16:37 server id 1 end_log_pos 470 CRC32 0xc9920491  Update_rows: table id
85 flags: STMT_END_F

BINLOG '
dfHUYRMBAAAQwAAAIcBAAAAFUAAAAAAEACWF0Z3VpZ3UxNAAHc3R1ZGVudAADAw8PBDwAHgAG

```

```

AQEAAgEhkJj4dA==
dfHUYR8BAAAATwAAANYBAAAAAFUAAAAAAEAAgAD//8AAQAAAblvKDkuIkG5LiA54+tAAEAAAAL
5byg5LiJX2JhY2sG5LiA54+tkQSSyQ==
'/*!*/;
### UPDATE `atguigu`.`student`
### WHERE
###   @1=1
###   @2='张三'
###   @3='一班'
### SET
###   @1=1
###   @2='张三_back'
###   @3='一班'
# at 470
#220105 9:16:37 server id 1 end_log_pos 501 CRC32 0xca01d30f Xid = 15
COMMIT/*!*/;

```

前面的命令同时显示binlog格式的语句，使用如下命令不显示它

```

mysqlbinlog -v --base64-output=DECODE-ROWS "/var/lib/mysql/binlog/atguigu-bin.000002"
#220105 9:16:37 server id 1 end_log_pos 324 CRC32 0x6b31978b Query thread_id=10
exec_time=0 error_code=0
SET TIMESTAMP=1641345397/*!*/;
SET @@session.pseudo_thread_id=10/*!*/;
SET @@session.foreign_key_checks=1, @@session.sql_auto_is_null=0,
@@session.unique_checks=1, @@session.autocommit=1/*!*/;
SET @@session.sql_mode=1168113696/*!*/;
SET @@session.auto_increment_increment=1, @@session.auto_increment_offset=1/*!*/;
/*!\\C utf8mb3 *//*!*/;
SET
@@session.character_set_client=33,@@session.collation_connection=33,@@session.collation_n_server=255/*!*/;
SET @@session.lc_time_names=0/*!*/;
SET @@session.collation_database=DEFAULT/*!*/;
/*!80011 SET @@session.default_collation_for_utf8mb4=255*//*!*/;
BEGIN
/*!*/;
# at 324
#220105 9:16:37 server id 1 end_log_pos 391 CRC32 0x74f89890 Table_map:
`atguigu14`.`student` mapped to number 85
# at 391
#220105 9:16:37 server id 1 end_log_pos 470 CRC32 0xc9920491 Update_rows: table id
85 flags: STMT_END_F
### UPDATE `atguigu14`.`student`
### WHERE
###   @1=1
###   @2='张三'
###   @3='一班'
### SET
###   @1=1
###   @2='张三_back'
###   @3='一班'
# at 470
#220105 9:16:37 server id 1 end_log_pos 501 CRC32 0xca01d30f Xid = 15

```

关于mysqlbinlog工具的使用技巧还有很多，例如只解析对某个库的操作或者某个时间段内的操作等。简单分享几个常用的语句，更多操作可以参考官方文档。

```

# 可查看参数帮助
mysqlbinlog --no-defaults --help

# 查看最后100行
mysqlbinlog --no-defaults --base64-output=decode-rows -vv atguigu-bin.000002 |tail -100

# 根据position查找
mysqlbinlog --no-defaults --base64-output=decode-rows -vv atguigu-bin.000002 |grep -A 20 '4939002'

```

上面这种办法读取出binlog日志的全文内容比较多，不容易分辨查看到pos点信息，下面介绍一种更为方便的查询命令：

```
mysql> show binlog events [IN 'log_name'] [FROM pos] [LIMIT [offset,] row_count];
```

- `IN 'log_name'` : 指定要查询的binlog文件名（不指定就是第一个binlog文件）
- `FROM pos` : 指定从哪个pos起始点开始查起（不指定就是从整个文件首个pos点开始算）
- `LIMIT [offset]` : 偏移量(不指定就是0)
- `row_count` : 查询总条数 (不指定就是所有行)

```

mysql> show binlog events in 'atguigu-bin.000002';
+-----+-----+-----+-----+-----+
| Log_name      | Pos | Event_type   | Server_id | End_log_pos | Info
+-----+-----+-----+-----+-----+
| atguigu-bin.000002 | 4   | Format_desc  | 1          | 125         | Server ver: 8.0.26, Binlog ver: 4
| atguigu-bin.000002 | 125 | Previous_gtids | 1          | 156         |
| atguigu-bin.000002 | 156 | Anonymous_Gtid | 1          | 235         | SET @SESSION.GTID_NEXT= 'ANONYMOUS'
| atguigu-bin.000002 | 235 | Query        | 1          | 324         | BEGIN
| atguigu-bin.000002 | 324 | Table_map    | 1          | 391         | table_id: 85 (atguigu14.student)
| atguigu-bin.000002 | 391 | Update_rows  | 1          | 470         | table_id: 85 flags: STMT_END_F
| atguigu-bin.000002 | 470 | Xid          | 1          | 501         | COMMIT /* xid=15 */
| atguigu-bin.000002 | 501 | Anonymous_Gtid | 1          | 578         | SET @SESSION.GTID_NEXT= 'ANONYMOUS'
| atguigu-bin.000002 | 578 | Query        | 1          | 721         | use `atguigu14`; create table test(id int, title varchar(100)) /* xid=19 */
| atguigu-bin.000002 | 721 | Anonymous_Gtid | 1          | 800         | SET @SESSION.GTID_NEXT= 'ANONYMOUS'
| atguigu-bin.000002 | 800 | Query        | 1          | 880         | BEGIN
| atguigu-bin.000002 | 880 | Table_map    | 1          | 943         | table_id: 89 (atguigu14.test)
| atguigu-bin.000002 | 943 | Write_rows   | 1          | 992         | table_id: 89 flags: STMT_END_F
| atguigu-bin.000002 | 992 | Xid          | 1          | 1023        | COMMIT /* xid=21 */

```

```
+-----+-----+-----+-----+
| 14 行于数据集 (0.02 秒)
```

上面我们讲了这么多都是基于binlog的默认格式，binlog格式查看

```
mysql> show variables like 'binlog_format';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| binlog_format | ROW   |
+-----+-----+
1 行于数据集 (0.02 秒)
```

除此之外，binlog还有2种格式，分别是Statement和Mixed

- **Statement**

每一条会修改数据的sql都会记录在binlog中。

优点：不需要记录每一行的变化，减少了binlog日志量，节约了IO，提高性能。

- **Row**

5.1.5版本的MySQL才开始支持row level 的复制，它不记录sql语句上下文相关信息，仅保存哪条记录被修改。

优点：row level 的日志内容会非常清楚的记录下每一行数据修改的细节。而且不会出现某些特定情况下的存储过程，或function，以及trigger的调用和触发无法被正确复制的问题。

- **Mixed**

从5.1.8版本开始，MySQL提供了Mixed格式，实际上就是Statement与Row的结合。

详细情况，下章讲解。

5.4 使用日志恢复数据

mysqlbinlog恢复数据的语法如下：

```
mysqlbinlog [option] filename|mysql -uuser -ppass;
```

这个命令可以这样理解：使用mysqlbinlog命令来读取filename中的内容，然后使用mysql命令将这些内容恢复到数据库中。

- **filename**：是日志文件名。
- **option**：可选项，比较重要的两对option参数是--start-date、--stop-date 和 --start-position、--stop-position。
 - **--start-date** 和 **--stop-date**：可以指定恢复数据库的起始时间点和结束时间点。
 - **--start-position**和**--stop-position**：可以指定恢复数据的开始位置和结束位置。

注意：使用mysqlbinlog命令进行恢复操作时，必须是编号小的先恢复，例如atguigu-bin.000001必须在atguigu-bin.000002之前恢复。

5.5 删除二进制日志

MySQL的二进制文件可以配置自动删除，同时MySQL也提供了安全的手动删除二进制文件的方法。

`PURGE MASTER LOGS` 只删除指定部分的二进制日志文件，`RESET MASTER` 删除所有的二进制日志文件。具体如下：

1. PURGE MASTER LOGS：删除指定日志文件

`PURGE MASTER LOGS`语法如下：

```
PURGE {MASTER | BINARY} LOGS TO '指定日志文件名'  
PURGE {MASTER | BINARY} LOGS BEFORE '指定日期'
```

5.6 其它场景

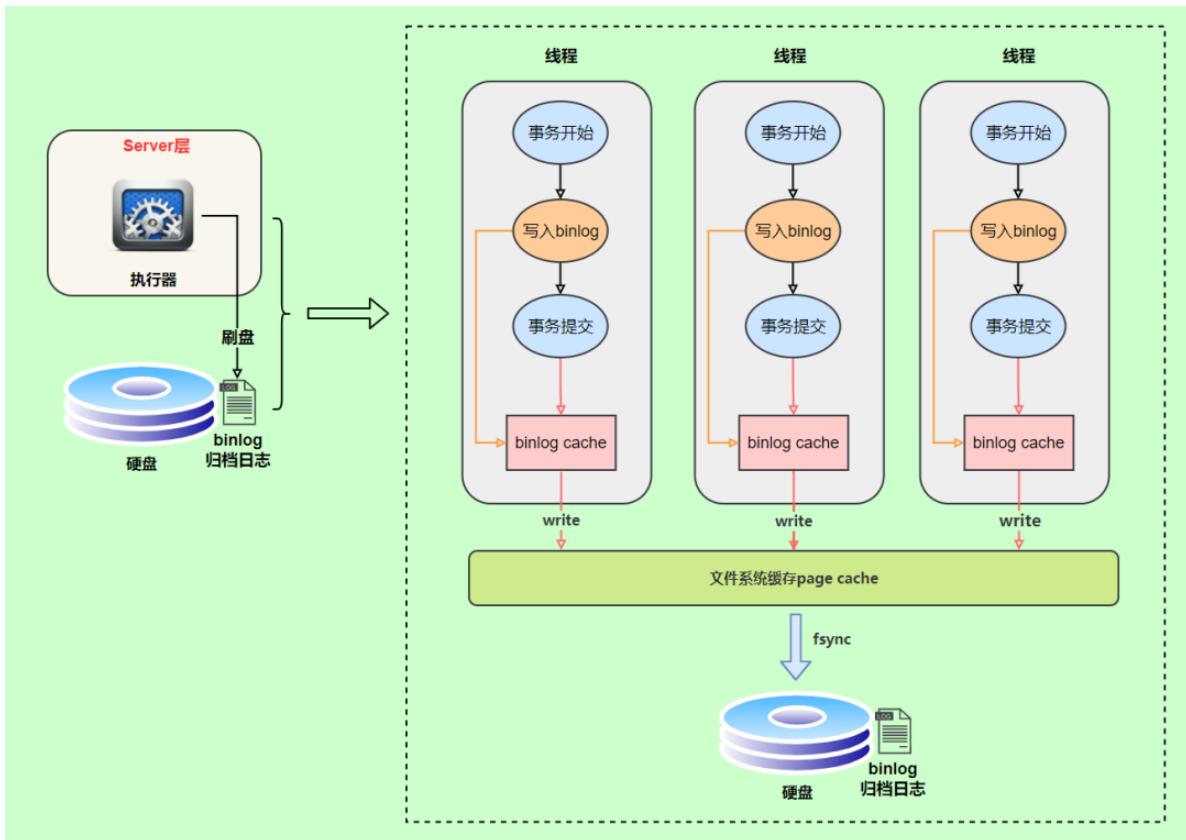
二进制日志可以通过数据库的 `全量备份` 和二进制日志中保存的 `增量信息`，完成数据库的 `无损失恢复`。但是，如果遇到数据量大、数据库和数据表很多（比如分库分表的应用）的场景，用二进制日志进行数据恢复，是很有挑战性的，因为起止位置不容易管理。

在这种情况下，一个有效的解决办法是 `配置主从数据库服务器`，甚至是 `一主多从` 的架构，把二进制日志文件的内容通过中继日志，同步到从数据库服务器中，这样就可以有效避免数据库故障导致的数据异常等问题。

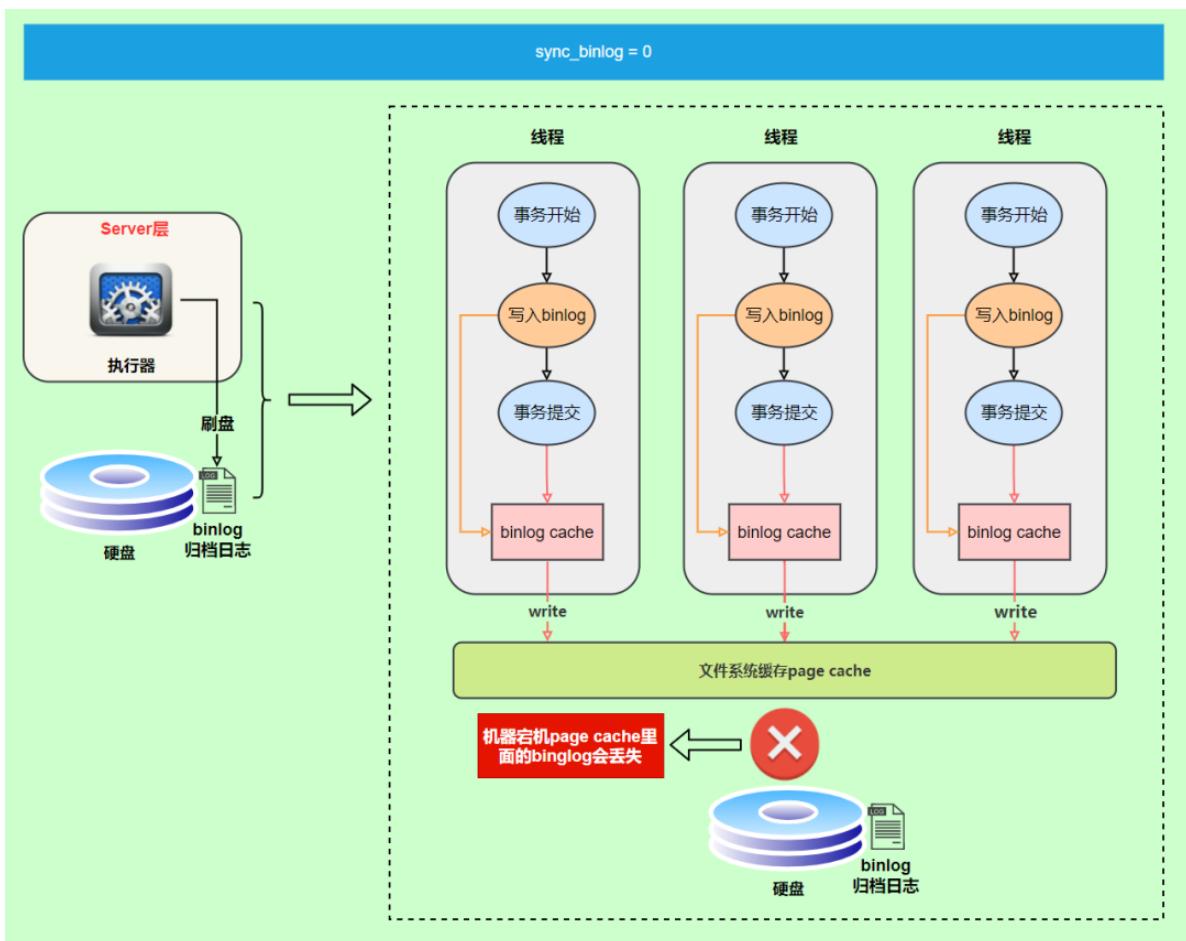
6. 再谈二进制日志(binlog)

6.1 写入机制

binlog的写入时机也非常简单，事务执行过程中，先把日志写到 `binlog cache`，事务提交的时候，再把binlog cache写到binlog文件中。因为一个事务的binlog不能被拆开，无论这个事务多大，也要确保一次性写入，所以系统会给每个线程分配一个块内存作为binlog cache。

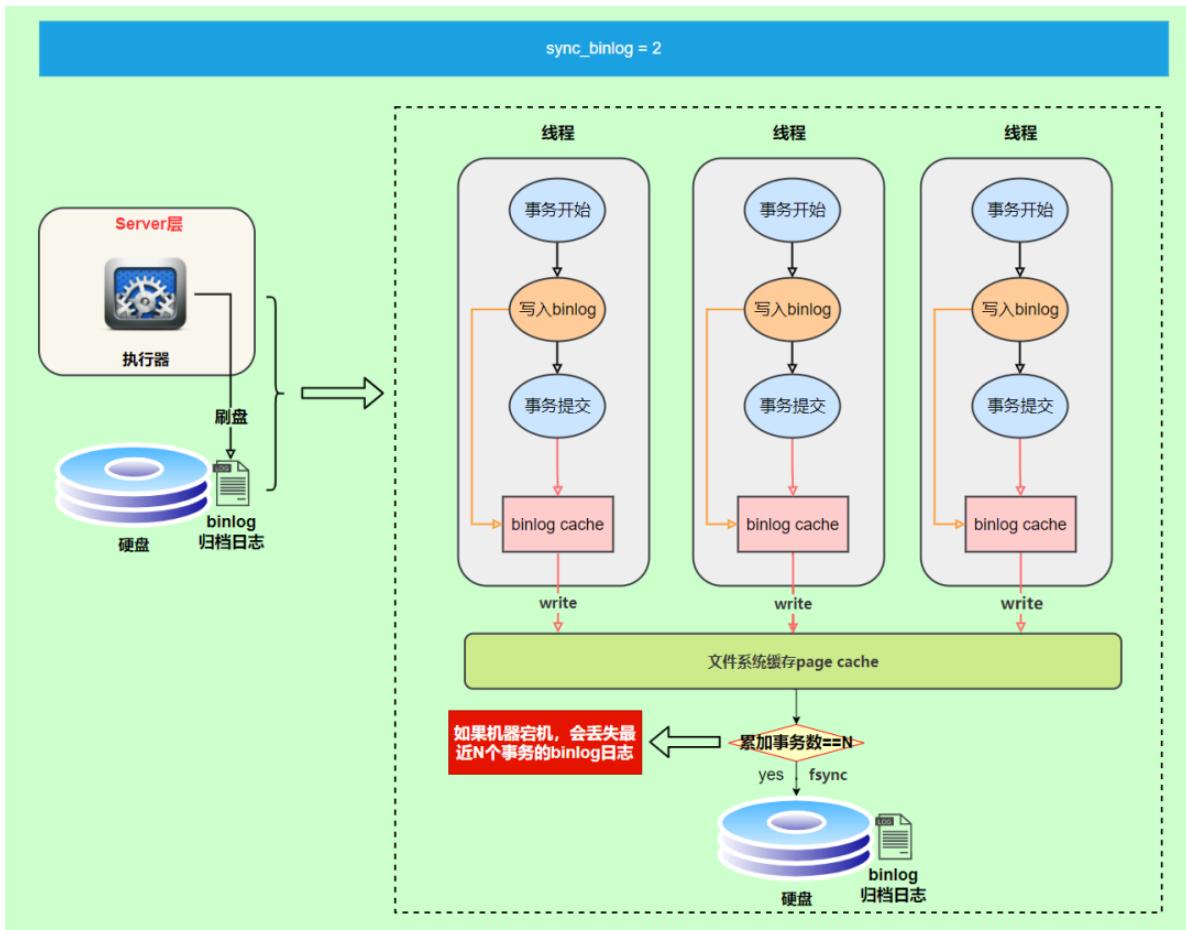


write和fsync的时机，可以由参数 `sync_binlog` 控制，默认是 0。为0的时候，表示每次提交事务都只 write，由系统自行判断什么时候执行fsync。虽然性能得到提升，但是机器宕机，page cache里面的 binlog 会丢失。如下图：



为了安全起见，可以设置为 1，表示每次提交事务都会执行fsync，就如同 **redo log 刷盘流程**一样。

最后还有一种折中方式，可以设置为N(N>1)，表示每次提交事务都write，但累积N个事务后才fsync。



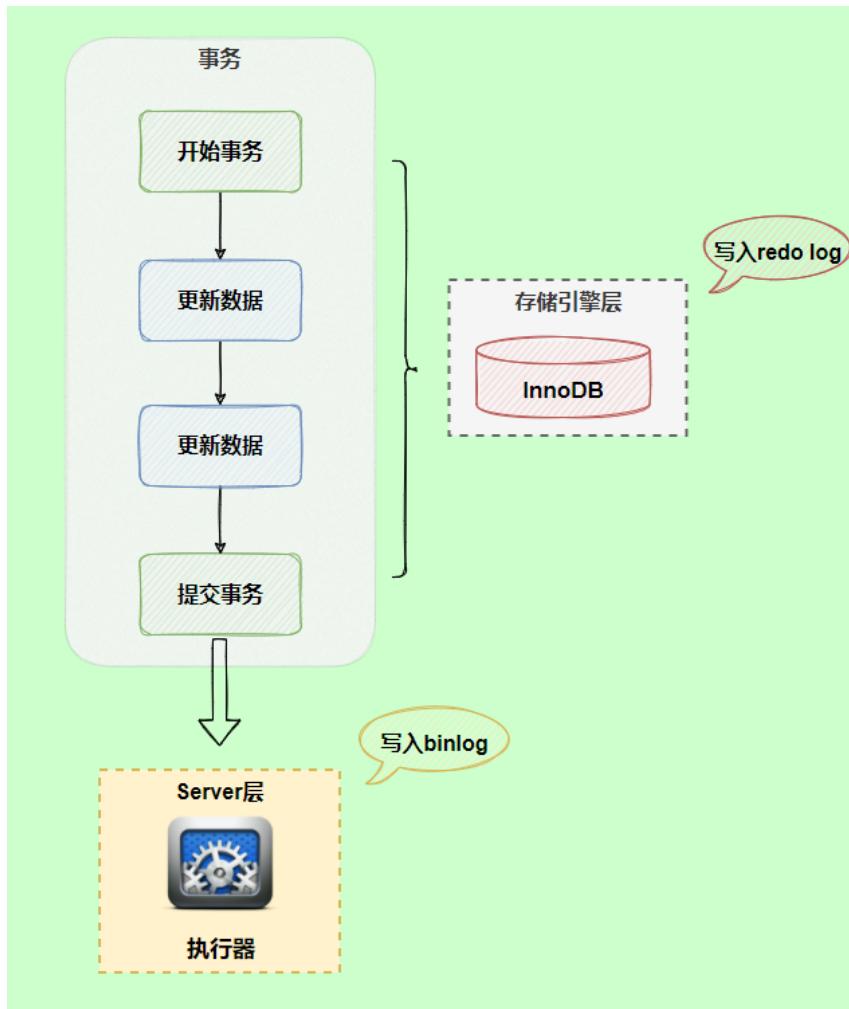
在出现IO瓶颈的场景里，将`sync_binlog`设置成一个比较大的值，可以提升性能。同样的，如果机器宕机，会丢失最近N个事务的binlog日志。

6.2 binlog与redolog对比

- redo log 它是 **物理日志**，记录内容是“在某个数据页上做了什么修改”，属于 InnoDB 存储引擎层产生的。
- 而 binlog 是 **逻辑日志**，记录内容是语句的原始逻辑，类似于“给 ID=2 这一行的 c 字段加 1”，属于 MySQL Server 层。

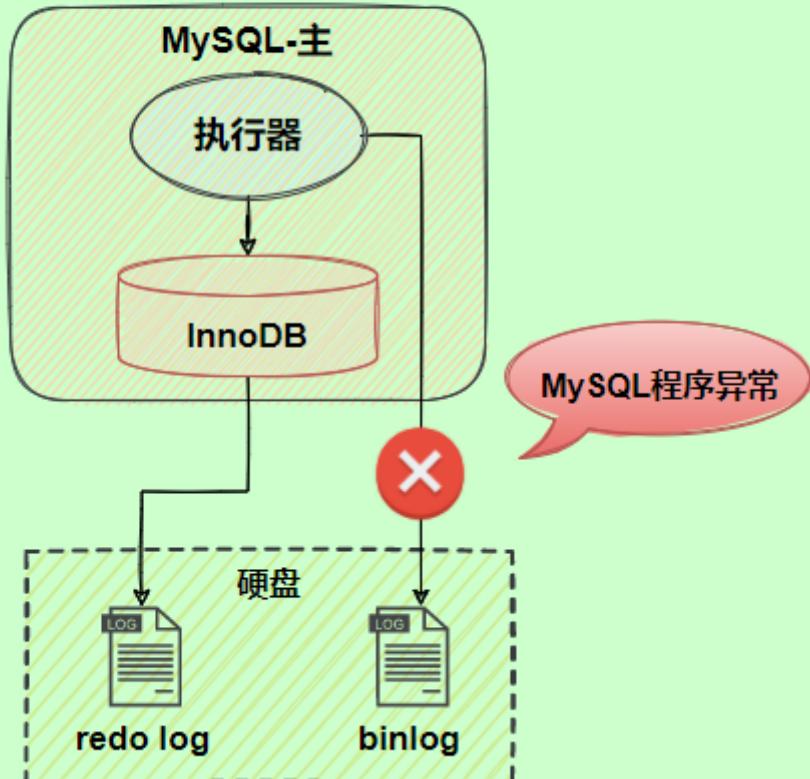
6.3 两阶段提交

在执行更新语句过程，会记录redo log与binlog两块日志，以基本的事务为单位，redo log在事务执行过程中可以不断写入，而binlog只有在提交事务时才写入，所以redo log与binlog的 **写入时机** 不一样。

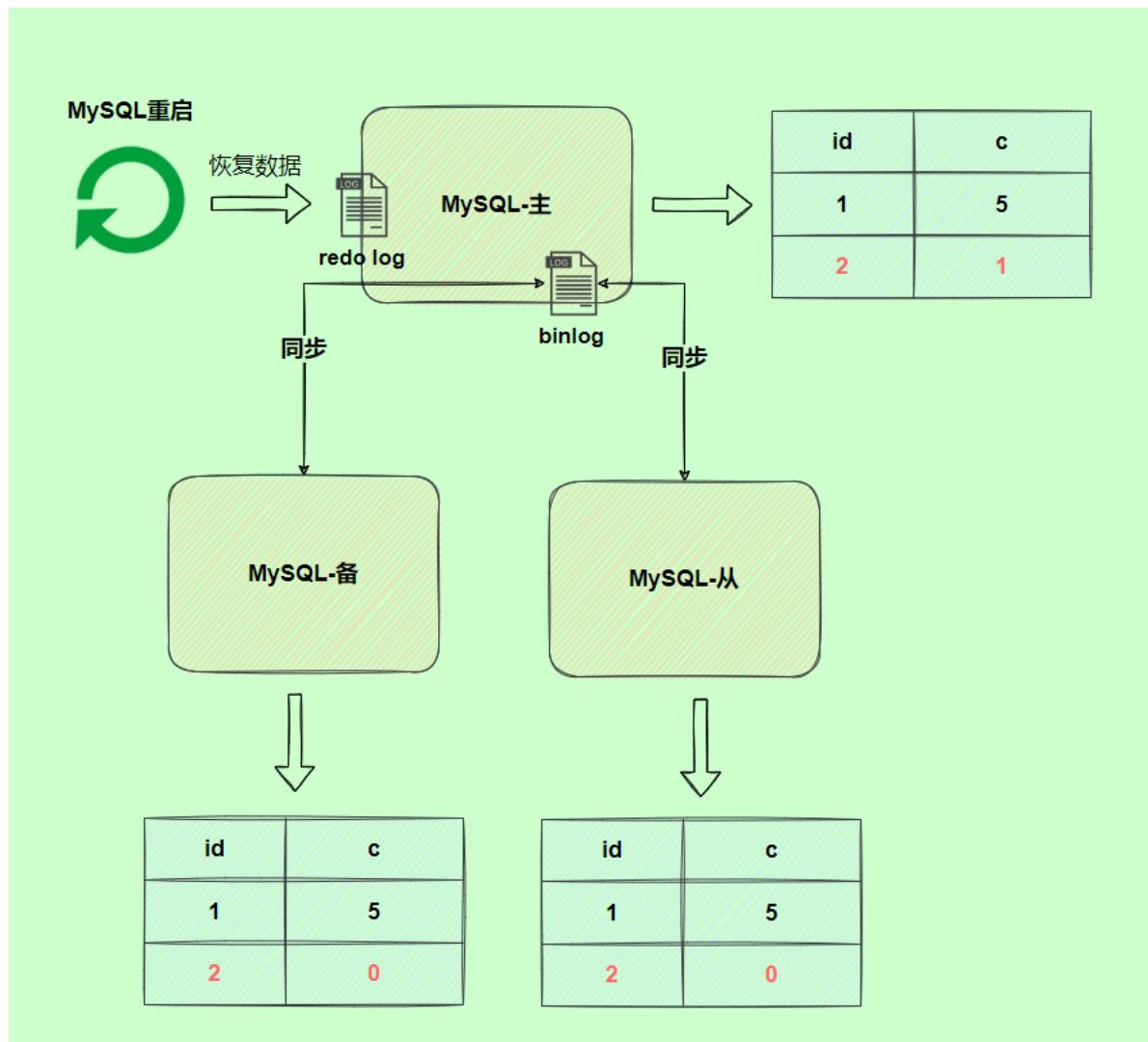


redo log与binlog两份日志之间的逻辑不一致，会出现什么问题？

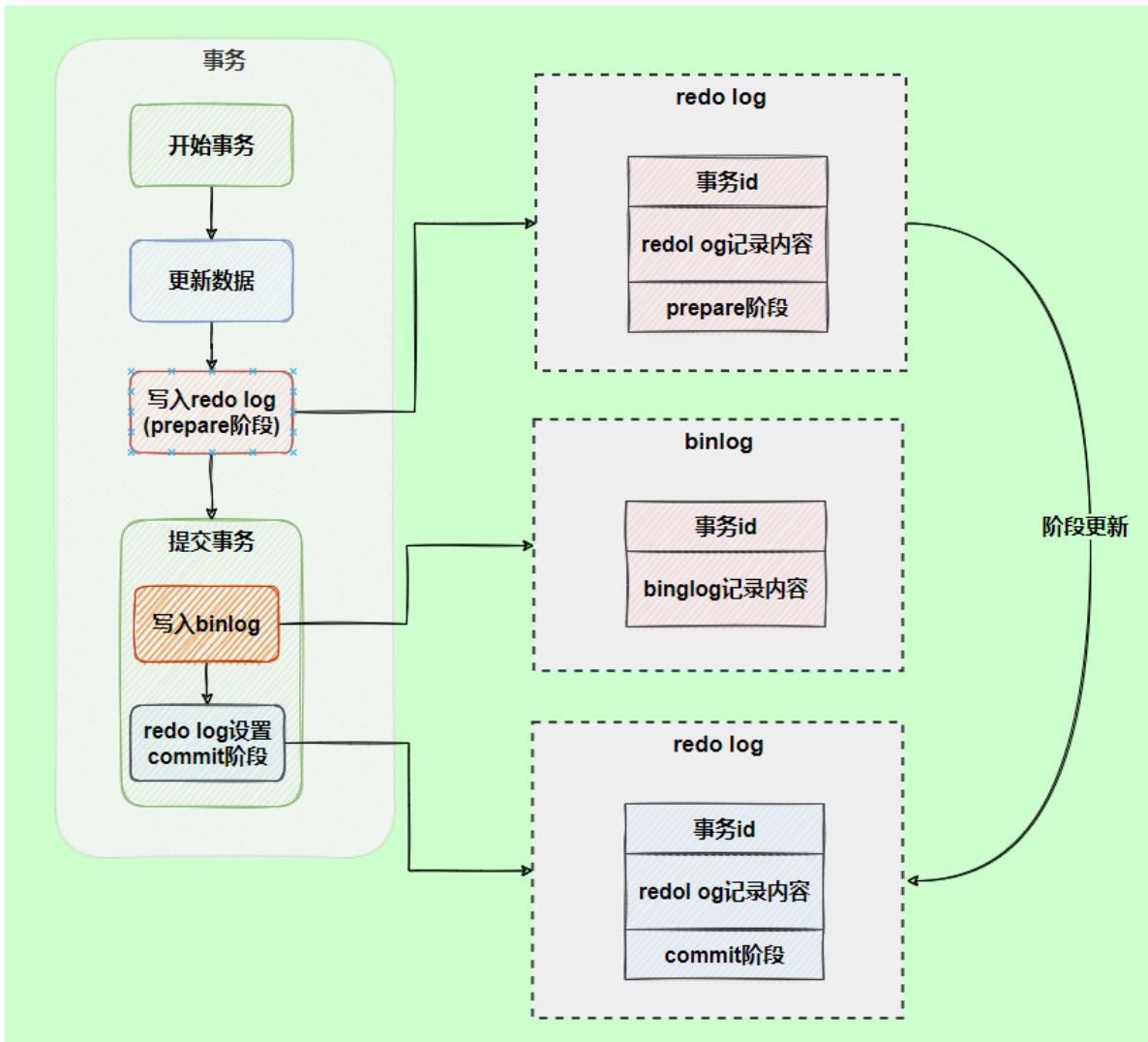
```
update T set c = 1 where id = 2
```



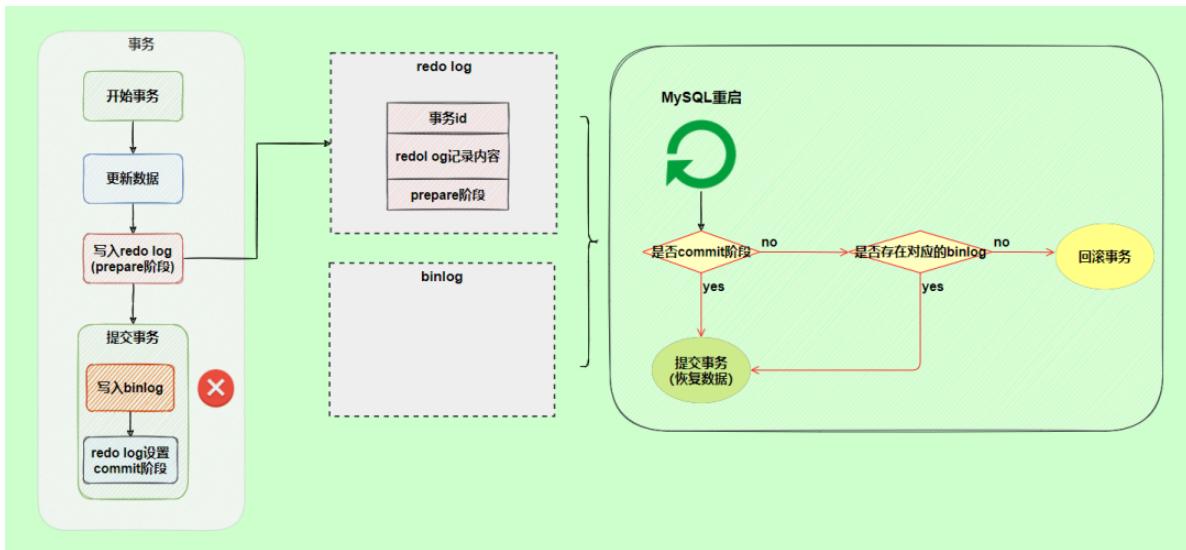
由于binlog没写完就异常，这时候binlog里面没有对应的修改记录。



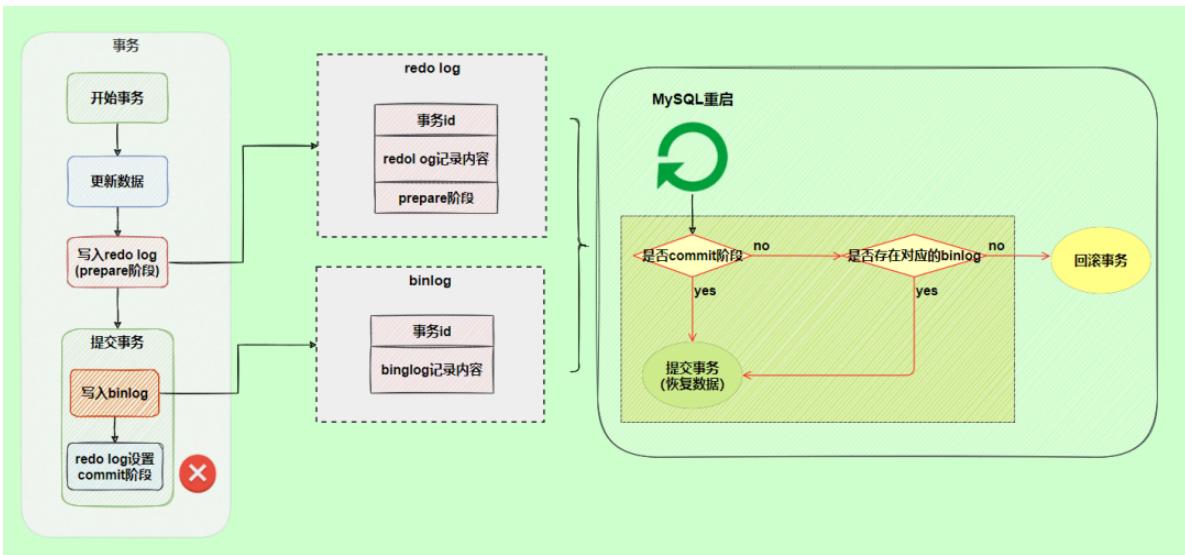
为了解决两份日志之间的逻辑一致问题，InnoDB存储引擎使用**两阶段提交**方案。



使用**两阶段提交**后，写入binlog时发生异常也不会有影响



另一个场景，redo log设置commit阶段发生异常，那会不会回滚事务呢？



并不会回滚事务，它会执行上图框住的逻辑，虽然redo log是处于prepare阶段，但是能通过事务id找到对应的binlog日志，所以MySQL认为是完整的，就会提交事务恢复数据。

7. 中继日志(relay log)

7.1 介绍

中继日志只在主从服务器架构的从服务器上存在。从服务器为了与主服务器保持一致，要从主服务器读取二进制日志的内容，并且把读取到的信息写入 **本地的日志文件** 中，这个从服务器本地的日志文件就叫 **中继日志**。然后，从服务器读取中继日志，并根据中继日志的内容对从服务器的数据进行更新，完成主从服务器的 **数据同步**。

搭建好主从服务器之后，中继日志默认会保存在从服务器的数据目录下。

文件名的格式是：**从服务器名 -relay-bin.序号**。中继日志还有一个索引文件：**从服务器名 -relay-bin.index**，用来定位当前正在使用的中继日志。

7.2 查看中继日志

中继日志与二进制日志的格式相同，可以用 **mysqlbinlog** 工具进行查看。下面是中继日志的一个片段：

```
SET TIMESTAMP=1618558728/*!*/;
BEGIN
/*!*/;
# at 950
#210416 15:38:48 server id 1  end_log_pos 832 CRC32 0xcc16d651  Table_map:
`atguigu`.`test` mapped to number 91
# at 1000
#210416 15:38:48 server id 1  end_log_pos 872 CRC32 0x07e4047c  Delete_rows: table id
91 flags: STMT_END_F  -- server id 1 是主服务器，意思是主服务器删了一行数据
BINLOG '
CD95YBMBAAAAMgAAAEADAAAAFsAAAAAAEABGR1bW8ABHR1c3QAAQMAAQEBAFHWFsw=
CD95YCABAAAACKAAAAGgDAAAAFsAAAAAAEAAgAB/wABAAAfATkBw==
/*!*/;
# at 1040
```

这一段的意思是，主服务器（“server id 1”）对表 atguigu.test 进行了 2 步操作：

定位到表 atguigu.test 编号是 91 的记录，日志位置是 832；

删除编号是 91 的记录，日志位置是 872。

7.3 恢复的典型错误

如果从服务器宕机，有的时候为了系统恢复，要重装操作系统，这样就可能会导致你的 服务器名称 与之前 不同。而中继日志里是 包含从服务器名 的。在这种情况下，就可能导致你恢复从服务器的时候，无法从宕机前的中继日志里读取数据，以为是日志文件损坏了，其实是名称不对了。

解决的方法也很简单，把从服务器的名称改回之前的名称。

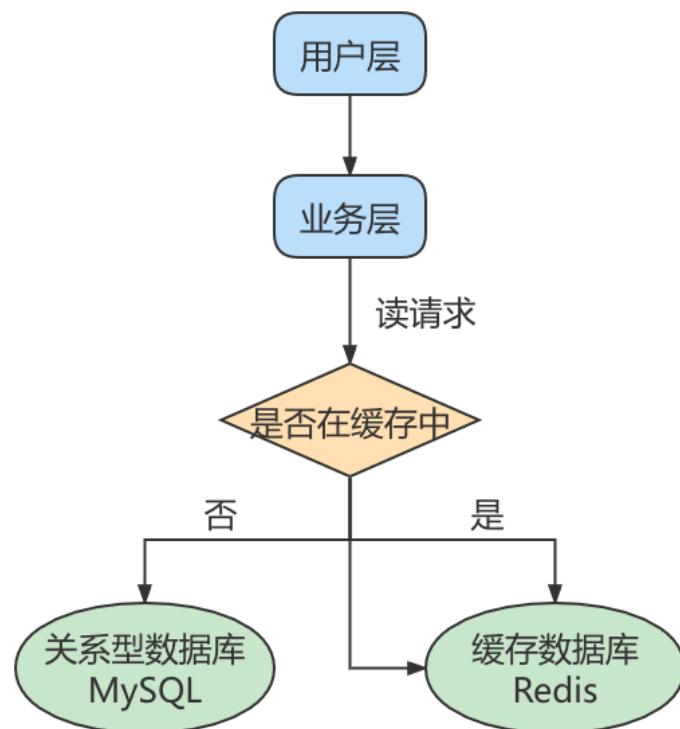
第18章_主从复制

讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

1. 主从复制概述

1.1 如何提升数据库并发能力



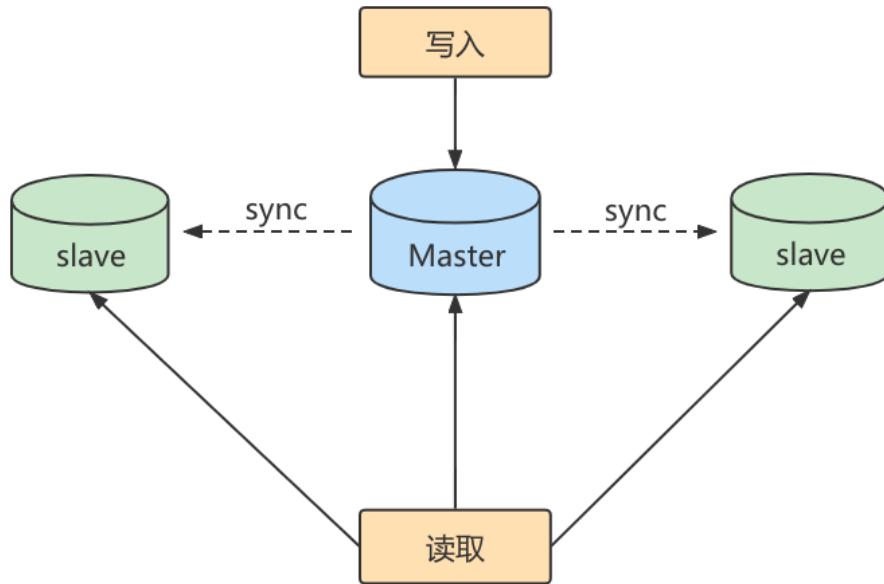
此外，一般应用对数据库而言都是“**读多写少**”，也就是说对数据库读取数据的压力比较大，有一个思路就是采用数据库集群的方案，做**主从架构**、进行**读写分离**，这样同样可以提升数据库的并发处理能力。但并不是所有的应用都需要对数据库进行主从架构的设置，毕竟设置架构本身是有成本的。

如果我们的目的在于提升数据库高并发访问的效率，那么首先考虑的是如何**优化SQL和索引**，这种方式简单有效；其次才是采用**缓存的策略**，比如使用Redis将热点数据保存在内存数据库中，提升读取的效率；最后才是对数据库采用**主从架构**，进行读写分离。

1.2 主从复制的作用

主从同步设计不仅可以提高数据库的吞吐量，还有以下3个方面的作用。

第1个作用：读写分离。



第2个作用就是数据备份。

第3个作用是具有高可用性。

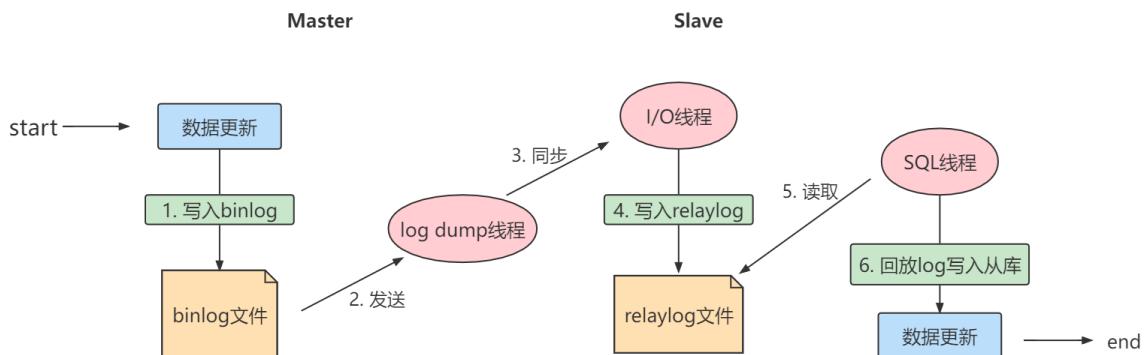
2. 主从复制的原理

Slave 会从 Master 读取 binlog 来进行数据同步。

2.1 原理剖析

三个线程

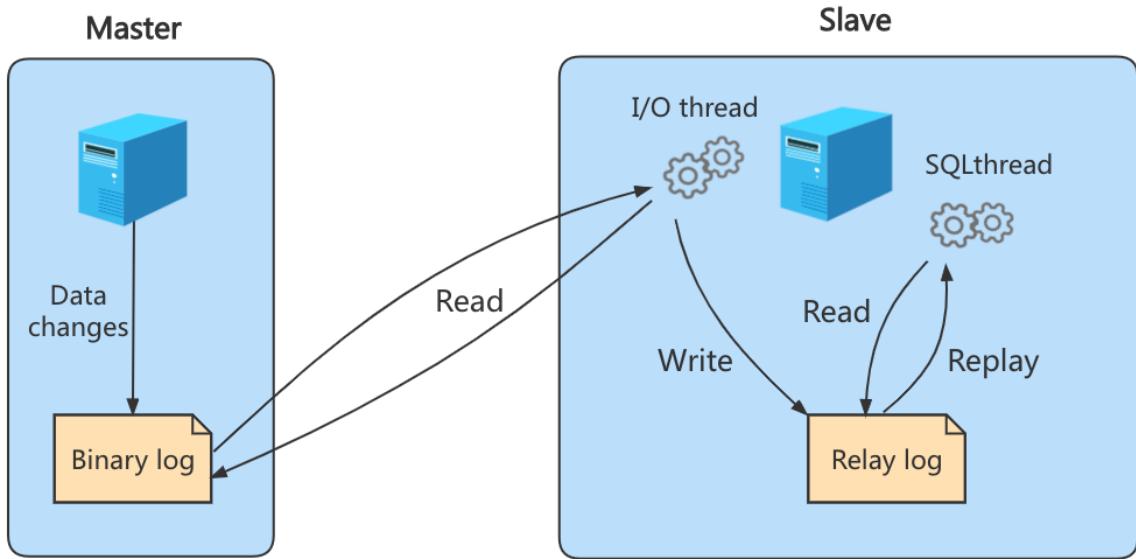
实际上主从同步的原理就是基于 binlog 进行数据同步的。在主从复制过程中，会基于 3 个线程来操作，一个主库线程，两个从库线程。



二进制日志转储线程 (Binlog dump thread) 是一个主库线程。当从库线程连接的时候，主库可以将二进制日志发送给从库，当主库读取事件 (Event) 的时候，会在 Binlog 上 **加锁**，读取完成之后，再将锁释放掉。

从库 I/O 线程 会连接到主库，向主库发送请求更新 Binlog。这时从库的 I/O 线程就可以读取到主库的二进制日志转储线程发送的 Binlog 更新部分，并且拷贝到本地的中继日志 (Relay log)。

从库 SQL 线程 会读取从库中的中继日志，并且执行日志中的事件，将从库中的数据与主库保持同步。



复制三步骤

步骤1：Master 将写操作记录到二进制日志（binlog）。

步骤2：Slave 将 Master 的binary log events拷贝到它的中继日志（relay log）；

步骤3：Slave 重做中继日志中的事件，将改变应用到自己的数据库中。MySQL复制是异步的且串行化的，而且重启后从 接入点 开始复制。

复制的问题

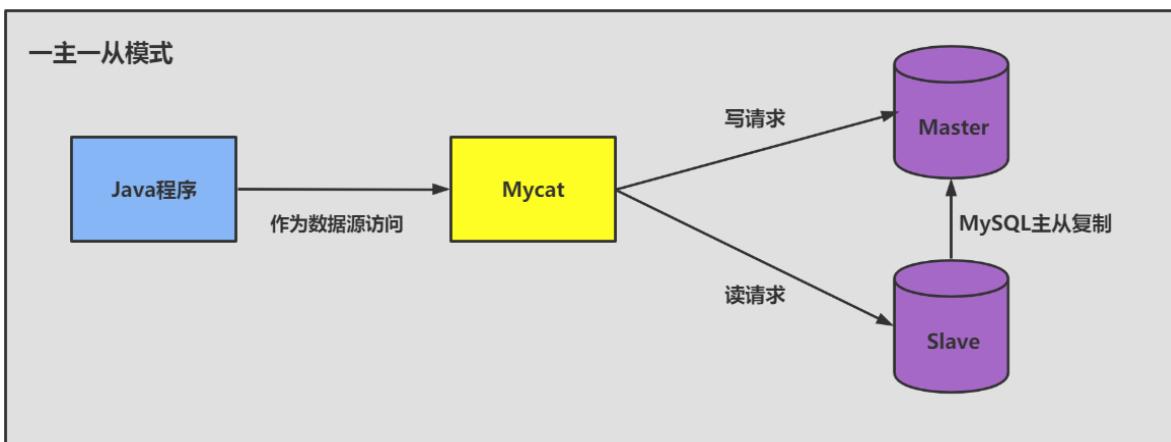
复制的最大问题： 延时

2.2 复制的基本原则

- 每个 Slave 只有一个 Master
- 每个 Slave 只能有一个唯一的服务器ID
- 每个 Master 可以有多个 Slave

3. 一主一从架构搭建

一台 主机 用于处理所有 写请求 ，一台 从机 负责所有 读请求 ，架构图如下：



3.1 准备工作

- 1、准备 2台 CentOS 虚拟机
- 2、每台虚拟机上需要安装好MySQL (可以是MySQL8.0)

说明：前面我们讲过如何克隆一台CentOS。大家可以在一台CentOS上安装好MySQL，进而通过克隆的方式复制出1台包含MySQL的虚拟机。

注意：克隆的方式需要修改新克隆出来主机的：① MAC地址 ② hostname ③ IP 地址 ④ UUID。

此外，克隆的方式生成的虚拟机（包含MySQL Server），则克隆的虚拟机MySQL Server的UUID相同，必须修改，否则在有些场景会报错。比如：`show slave status\G`，报如下的错误：

```
Last_IO_Error: Fatal error: The slave I/O thread stops because master and slave have equal MySQL server UUIDs; these UUIDs must be different for replication to work.
```

修改MySQL Server 的UUID方式：

```
vim /var/lib/mysql/auto.cnf  
systemctl restart mysqld
```

3.2 主机配置文件

建议mysql版本一致且后台以服务运行，主从所有配置项都配置在 [mysqld] 节点下，且都是小写字母。

具体参数配置如下：

- 必选

```
#【必须】主服务器唯一ID  
server-id=1  
  
#[必须]启用二进制日志,指名路径。比如：自己本地的路径/log/mysqlbin  
log-bin=atguigu-bin
```

- 可选

```
#【可选】0（默认）表示读写（主机），1表示只读（从机）  
read-only=0  
  
#设置日志文件保留的时长，单位是秒  
binlog_expire_logs_seconds=6000  
  
#控制单个二进制日志大小。此参数的最大和默认值是1GB  
max_binlog_size=200M  
  
#[可选]设置不要复制的数据库  
binlog-ignore-db=test  
  
#[可选]设置需要复制的数据库，默认全部记录。比如：binlog-do-db=atguigu_master_slave  
binlog-do-db=需要复制的主数据库名字  
  
#[可选]设置binlog格式  
binlog_format=STATEMENT
```

binlog格式设置：

格式1： **STATEMENT**模式 （基于SQL语句的复制(statement-based replication, SBR)）

`binlog_format=STATEMENT`

每一条会修改数据的sql语句会记录到binlog中。这是默认的binlog格式。

- SBR 的优点：
 - 历史悠久，技术成熟
 - 不需要记录每一行的变化，减少了binlog日志量，文件较小
 - binlog中包含了所有数据库更改信息，可以据此来审核数据库的安全等情况
 - binlog可以用于实时的还原，而不仅仅用于复制
 - 主从版本可以不一样，从服务器版本可以比主服务器版本高
- SBR 的缺点：
 - 不是所有的UPDATE语句都能被复制，尤其是包含不确定操作的时候
- 使用以下函数的语句也无法被复制：LOAD_FILE()、UUID()、USER()、FOUND_ROWS()、SYSDATE()
(除非启动时启用了 --sysdate-is-now 选项)
 - INSERT ... SELECT 会产生比 RBR 更多的行级锁
 - 复制需要进行全表扫描(WHERE 语句中没有使用到索引)的 UPDATE 时，需要比 RBR 请求更多的行级锁
 - 对于有 AUTO_INCREMENT 字段的 InnoDB 表而言，INSERT 语句会阻塞其他 INSERT 语句
 - 对于一些复杂的语句，在从服务器上的耗资源情况会更严重，而 RBR 模式下，只会对那个发生变化的记录产生影响
 - 执行复杂语句如果出错的话，会消耗更多资源
 - 数据表必须几乎和主服务器保持一致才行，否则可能会导致复制出错

② ROW模式（基于行的复制(row-based replication, RBR)）

`binlog_format=ROW`

5.1.5版本的MySQL才开始支持，不记录每条sql语句的上下文信息，仅记录哪条数据被修改了，修改成什么样了。

- RBR 的优点：
 - 任何情况都可以被复制，这对复制来说是最 **安全可靠** 的。（比如：不会出现某些特定情况下的存储过程、function、trigger的调用和触发无法被正确复制的问题）
 - 多数情况下，从服务器上的表如果有主键的话，复制就会快了很多
 - 复制以下几种语句时的行锁更少：INSERT ... SELECT、包含 AUTO_INCREMENT 字段的 INSERT、没有附带条件或者并没有修改很多记录的 UPDATE 或 DELETE 语句
 - 执行 INSERT, UPDATE, DELETE 语句时锁更少
 - 从服务器上采用 **多线程** 来执行复制成为可能
- RBR 的缺点：
 - binlog 大了很多
 - 复杂的回滚时 binlog 中会包含大量的数据
 - 主服务器上执行 UPDATE 语句时，所有发生变化的记录都会写到 binlog 中，而 SBR 只会写一次，这会导致频繁发生 binlog 的并发写问题
 - 无法从 binlog 中看到都复制了些什么语句

③ MIXED模式（混合模式复制(mixed-based replication, MBR)）

`binlog_format=MIXED`

从5.1.8版本开始，MySQL提供了Mixed格式，实际上就是Statement与Row的结合。

在Mixed模式下，一般的语句修改使用statement格式保存binlog。如一些函数，statement无法完成主从复制的操作，则采用row格式保存binlog。

MySQL会根据执行的每一条具体的sql语句来区分对待记录的日志形式，也就是在Statement和Row之间选择一种。

3.3 从机配置文件

要求主从所有配置项都配置在 `my.cnf` 的 `[mysqld]` 栏位下，且都是小写字母。

- 必选

```
# [必须] 从服务器唯一ID  
server-id=2
```

- 可选

```
# [可选] 启用中继日志  
relay-log=mysql-relay
```

重启后台mysql服务，使配置生效。

注意：主从机都关闭防火墙

```
service iptables stop #CentOS 6  
systemctl stop firewalld.service #CentOS 7
```

3.4 主机：建立账户并授权

```
# 在主机MySQL里执行授权主从复制的命令  
GRANT REPLICATION SLAVE ON *.* TO 'slave1'@'从机器数据库IP' IDENTIFIED BY 'abc123';  
#5.5, 5.7
```

注意：如果使用的是MySQL8，需要如下的方式建立账户，并授权slave：

```
CREATE USER 'slave1'@'%' IDENTIFIED BY '123456';  
  
GRANT REPLICATION SLAVE ON *.* TO 'slave1'@'%';  
  
# 此语句必须执行。否则见下面。  
ALTER USER 'slave1'@'%' IDENTIFIED WITH mysql_native_password BY '123456';  
  
flush privileges;
```

注意：在从机执行show slave status\G时报错：

```
Last_IO_Error: error connecting to master 'slave1@192.168.1.150:3306' - retry-time: 60 retries: 1  
message: Authentication plugin 'caching_sha2_password' reported error: Authentication requires  
secure connection.
```

查询Master的状态，并记录下File和Position的值。

```
show master status;
```

```
mysql> show master status;
+-----+-----+-----+-----+-----+
| File | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set |
+-----+-----+-----+-----+-----+
| mysql-bin.000007 | 154 | testdb | mysql | |
+-----+-----+-----+-----+-----+
1 row in set (0.01 sec)
```

- 记录下File和Position的值

注意：执行完此步骤后**不要再操作主服务器MySQL**，防止主服务器状态值变化。

3.5 从机：配置需要复制的主机

步骤1：从机上复制主机的命令

```
CHANGE MASTER TO
MASTER_HOST='主机的IP地址',
MASTER_USER='主机用户名',
MASTER_PASSWORD='主机用户名的密码',
MASTER_LOG_FILE='mysql-bin.具体数字',
MASTER_LOG_POS=具体值；
```

举例：

```
CHANGE MASTER TO
MASTER_HOST='192.168.1.150',MASTER_USER='slave1',MASTER_PASSWORD='123456',MASTER_LOG_F
ILE='atguigu-bin.000007',MASTER_LOG_POS=154;
```

```
mysql> CHANGE MASTER TO MASTER_HOST='192.168.140.128',
      -> MASTER_USER='slave',
      -> MASTER_PASSWORD='123123',
      -> MASTER_LOG_FILE='mysql-bin.000007',MASTER_LOG_POS=154;
Query OK, 0 rows affected, 2 warnings (0.00 sec)
```

```
mysql> CHANGE MASTER TO MASTER_HOST='192.168.124.3',
      -> MASTER_USER='zhangsan',
      -> MASTER_PASSWORD='123456',
      -> MASTER_LOG_FILE='mysqlbin.000012',MASTER_LOG_POS=4386;
ERROR 1198 (HY000): This operation cannot be performed with a running slave; run STOP SLAVE first
mysql> stop slave;
Query OK, 0 rows affected (0.00 sec)

mysql> CHANGE MASTER TO MASTER_HOST='192.168.124.3',
      -> MASTER_USER='zhangsan',
      -> MASTER_PASSWORD='123456',
      -> MASTER_LOG_FILE='mysqlbin.000012',MASTER_LOG_POS=4386;
Query OK, 0 rows affected (0.01 sec)
```

如果之前做过同步，请先停止

步骤2：

```
#启动slave同步
START SLAVE;
```

```
mysql> start slave;
Query OK, 0 rows affected (0.00 sec)
```

如果报错：

```
mysql> start slave;
ERROR 1872 (HY000): Slave failed to initialize relay log info structure from the repository
mysql> reset slave;
Query OK, 0 rows affected, 1 warning (0.03 sec)
```

可以执行如下操作，删除之前的relay_log信息。然后重新执行 CHANGE MASTER TO ...语句即可。

```
mysql> reset slave; #删除SLAVE数据库的relaylog日志文件，并重新启用新的relaylog文件
```

接着，查看同步状态：

```
SHOW SLAVE STATUS\G;
```

```
mysql> SHOW SLAVE STATUS\G;
***** 1. row *****
Slave_IO_State: Waiting for master to send event
Master_Host: 172.16.116.1
Master_User: slave01
Master_Port: 3306
Connect_Retry: 60
Master_Log_File: logbin.000001
Read_Master_Log_Pos: 154
Relay_Log_File: mysql-relay.000002
Relay_Log_Pos: 317
Relay_Master_Log_File: logbin.000001
Slave_IO_Running: Yes          这两个都是yes，说明同步配置成功了
Slave_SQL_Running: Yes
Replicate_Do_DB:
Replicate_Ignore_DB:
Replicate_Do_Table:
Replicate_Ignore_Table:
Replicate_Wild_Do_Table:
Replicate_Wild_Ignore_Table:
Last_Error:
Skip_Counter: 0
Last_Error:
Skip_Counter: 0
Exec_Master_Log_Pos: 154
Relay_Log_Space: 520
Until_Condition: None
Until_Log_File:
Until_Log_Pos: 0
Master_SSL_Allowed: No
Master_SSL_CA_File:
```

上面两个参数都是Yes，则说明主从配置成功！

显式如下的情况，就是不正确的。可能错误的原因有：

1. 网络不通
2. 账户密码错误
3. 防火墙
4. mysql配置文件问题
5. 连接服务器时语法
6. 主服务器mysql权限

```
mysql> show slave status\G
***** 1. row *****
Slave_IO_State: Connecting to master
Master_Host: 192.168.1.110
Master_User: slave
Master_Port: 3306
Connect_Retry: 60
Master_Log_File: mysql-bin.000001
Read_Master_Log_Pos: 1513
Relay_Log_File: mysql-relay.000001
Relay_Log_Pos: 4
Relay_Master_Log_File: mysql-bin.000001
Slave_IO_Running: Connecting
Slave_SQL_Running: Yes
Replicate_Do_DB:
Replicate_Ignore_DB:
Replicate_Do_Table:
Replicate_Ignore_Table:
```

3.6 测试

主机新建库、新建表、insert记录，从机复制：

```
CREATE DATABASE atguigu_master_slave;

CREATE TABLE mytbl(id INT,NAME VARCHAR(16));

INSERT INTO mytbl VALUES(1, 'zhang3');

INSERT INTO mytbl VALUES(2,@@hostname);
```

3.7 停止主从同步

- 停止主从同步命令：

```
stop slave;
```

- 如何重新配置主从

如果停止从服务器复制功能，再使用需要重新配置主从。否则会报错如下：

```
--> MASTER_PASSWORD='123123',
--> MASTER_LOG_FILE='mysql-bin.000003',MASTER_LOG_POS=722;
ERROR 3021 (HY000): This operation cannot be performed with a running slave io thread; run STOP SLAVE
IO_THREAD FOR CHANNEL '' first.
mysql> stop slave;
Query OK, 0 rows affected (0.00 sec)
```

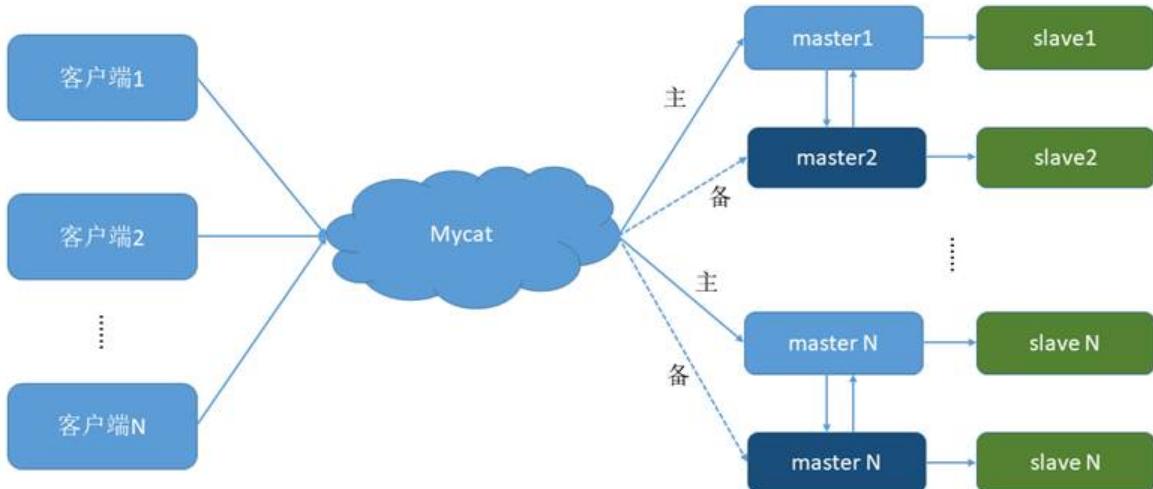
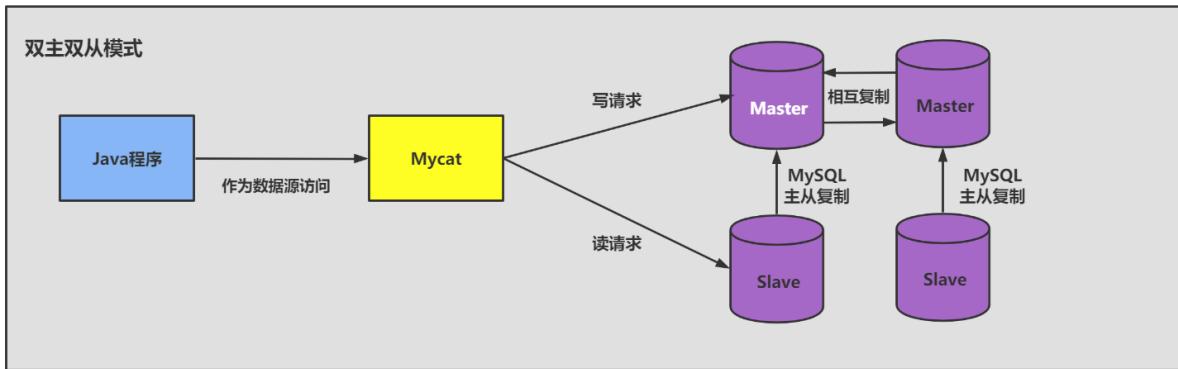
重新配置主从，需要在从机上执行：

```
stop slave;

reset master; #删除Master中所有的binglog文件，并将日志索引文件清空，重新开始所有新的日志文件(慎用)
```

3.8 后续

搭建主从复制：双主双从



4. 同步数据一致性问题

主从同步的要求:

- 读库和写库的数据一致(最终一致);
- 写数据必须写到写库;
- 读数据必须到读库(不一定);

4.1 理解主从延迟问题

进行主从同步的内容是二进制日志，它是一个文件，在进行 网络传输 的过程中就一定会 存在主从延迟 (比如 500ms)，这样就可能造成用户在从库上读取的数据不是最新的数据，也就是主从同步中的 数据不一致性 问题。

4.2 主从延迟问题原因

在网络正常的时候，日志从主库传给从库所需的时间是很短的，即T2-T1的值是非常小的。即，网络正常情况下，主备延迟的主要来源是备库接收完binlog和执行完这个事务之间的时间差。

主备延迟最直接的表现是，从库消费中继日志 (relay log) 的速度，比主库生产binlog的速度要慢。造成原因：

- 1、从库的机器性能比主库要差
- 2、从库的压力大
- 3、大事务的执行

举例1：一次性用delete语句删除太多数据

结论：后续再删除数据的时候，要控制每个事务删除的数据量，分成多次删除。

举例2：一次性用insert...select插入太多数据

举例3：大表DDL

比如在主库对一张500W的表添加一个字段耗费了10分钟，那么从节点上也会耗费10分钟。

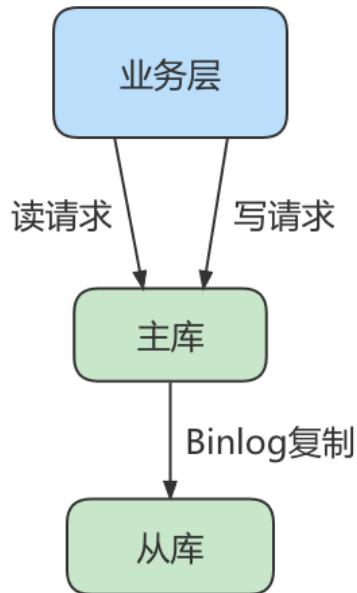
4.3 如何减少主从延迟

若想要减少主从延迟的时间，可以采取下面的办法：

1. 降低多线程大事务并发的概率，优化业务逻辑
2. 优化SQL，避免慢SQL，**减少批量操作**，建议写脚本以update-sleep这样的形式完成。
3. **提高从库机器的配置**，减少主库写binlog和从库读binlog的效率差。
4. 尽量采用**短的链路**，也就是主库和从库服务器的距离尽量要短，提升端口带宽，减少binlog传输的网络延时。
5. 实时性要求的业务读强制走主库，从库只做灾备，备份。

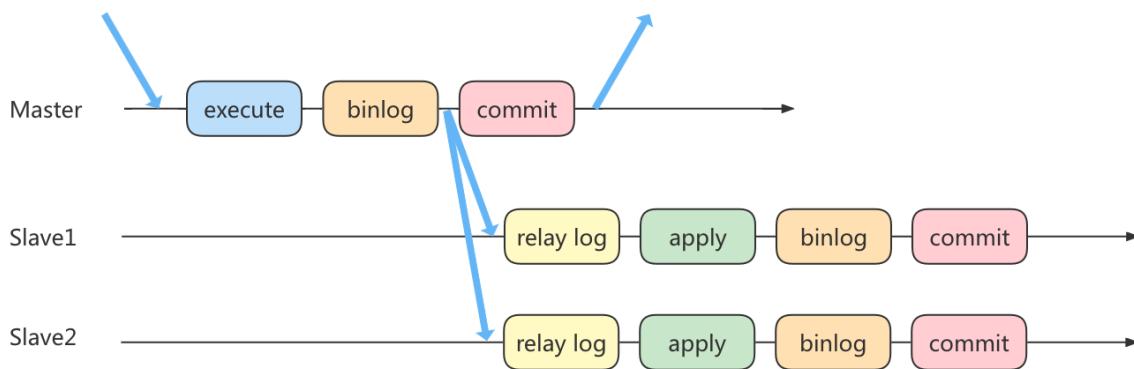
4.4 如何解决一致性问题

如果操作的数据存储在同一个数据库中，那么对数据进行更新的时候，可以对记录加写锁，这样在读取的时候就不会发生数据不一致的情况。但这时从库的作用就是**备份**，并没有起到**读写分离**，分担主库**读压力**的作用。

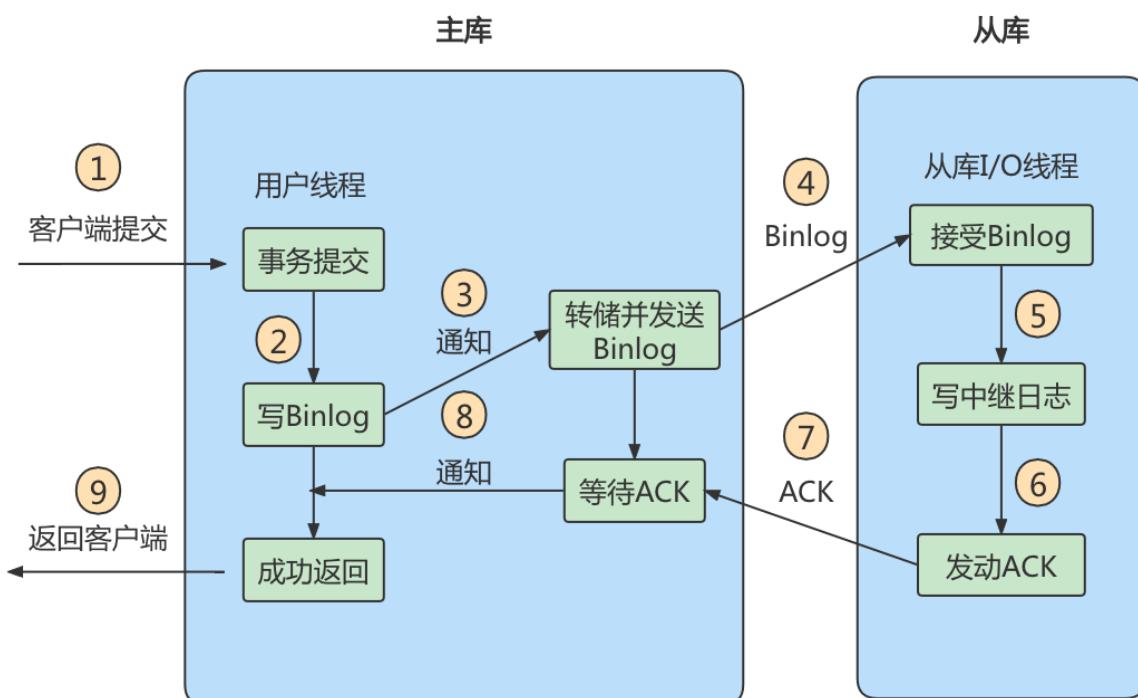


读写分离情况下，解决主从同步中数据不一致的问题，就是解决主从之间**数据复制方式**的问题，如果按照数据一致性**从弱到强**来进行划分，有以下3种复制方式。

方法 1：异步复制



方法 2：半同步复制



方法 3：组复制

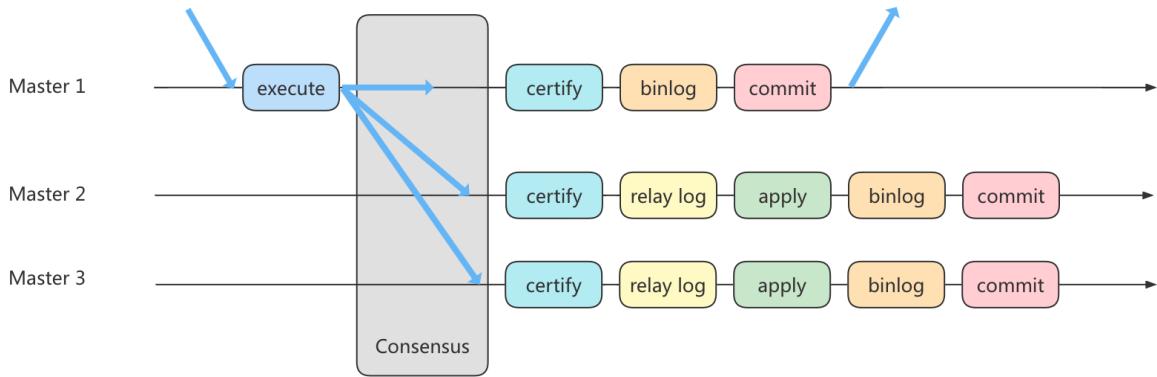
异步复制和半同步复制都无法最终保证数据的一致性问题，半同步复制是通过判断从库响应的个数来决定是否返回给客户端，虽然数据一致性相比于异步复制有提升，但仍然无法满足对数据一致性要求高的场景，比如金融领域。MGR很好地弥补了这两种复制模式的不足。

组复制技术，简称 MGR（MySQL Group Replication）。是 MySQL 在 5.7.17 版本中推出的一种新的数据复制技术，这种复制技术是基于 Paxos 协议的状态机复制。

MGR 是如何工作的

首先我们将多个节点共同组成一个复制组，在 **执行读写 (RW) 事务** 的时候，需要通过一致性协议层（Consensus 层）的同意，也就是读写事务想要进行提交，必须要经过组里“大多数人”（对应 Node 节点）的同意，大多数指的是同意的节点数量需要大于 $(N/2+1)$ ，这样才可以进行提交，而不是原发起方一个说了算。而针对 **只读 (RO) 事务** 则不需要经过组内同意，直接 COMMIT 即可。

在一个复制组内有多个节点组成，它们各自维护了自己的数据副本，并且在一致性协议层实现了原子消息和全局有序消息，从而保证组内数据的一致性。



MGR 将 MySQL 带入了数据强一致性的时代，是一个划时代的创新，其中一个重要的原因就是 MGR 是基于 Paxos 协议的。Paxos 算法是由 2013 年的图灵奖获得者 Leslie Lamport 于 1990 年提出的，有关这个算法的决策机制可以搜一下。事实上，Paxos 算法出来之后就作为 [分布式一致性算法](#) 被广泛应用，比如 Apache 的 ZooKeeper 也是基于 Paxos 实现的。

5. 知识延伸

在主从架构的配置中，如果想要采取读写分离的策略，我们可以 [自己编写程序](#)，也可以通过 [第三方的中间件](#) 来实现。

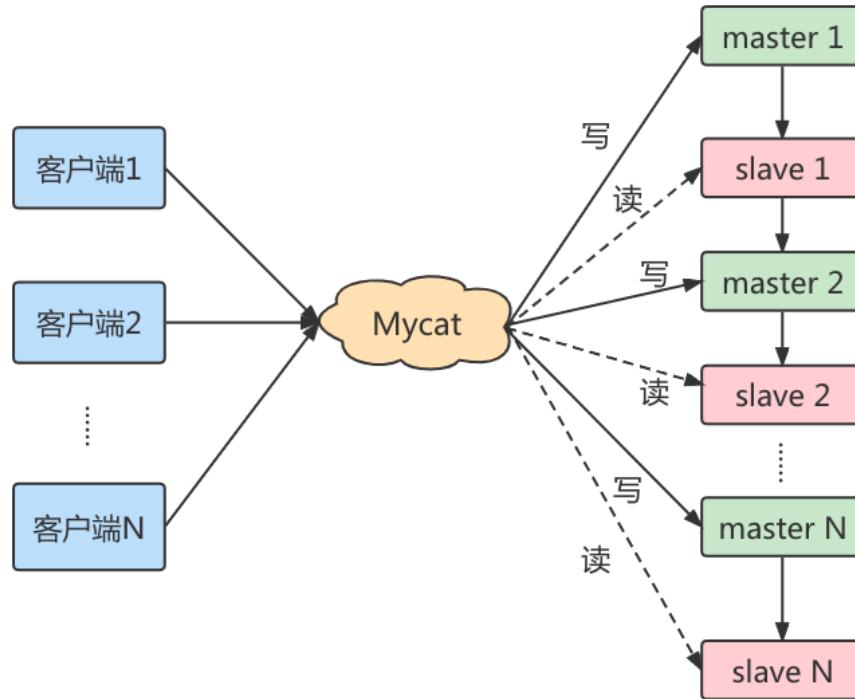
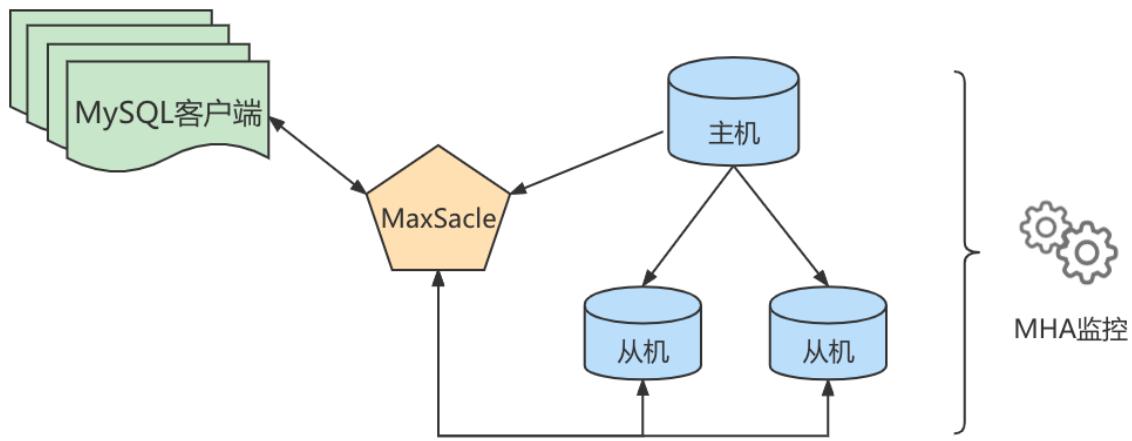
- 自己编写程序的好处就在于比较自主，我们可以自己判断哪些查询在从库上来执行，针对实时性要求高的需求，我们还可以考虑哪些查询可以在主库上执行。同时，程序直接连接数据库，减少了中间件层，相当于减少了性能损耗。
- 采用中间件的方法有很明显的劣势，[功能强大](#)，[使用简单](#)。但因为在客户端和数据库之间增加了中间件层会有一些 [性能损耗](#)，同时商业中间件也是有使用成本的。我们也可以考虑采取一些优秀的开源工具。



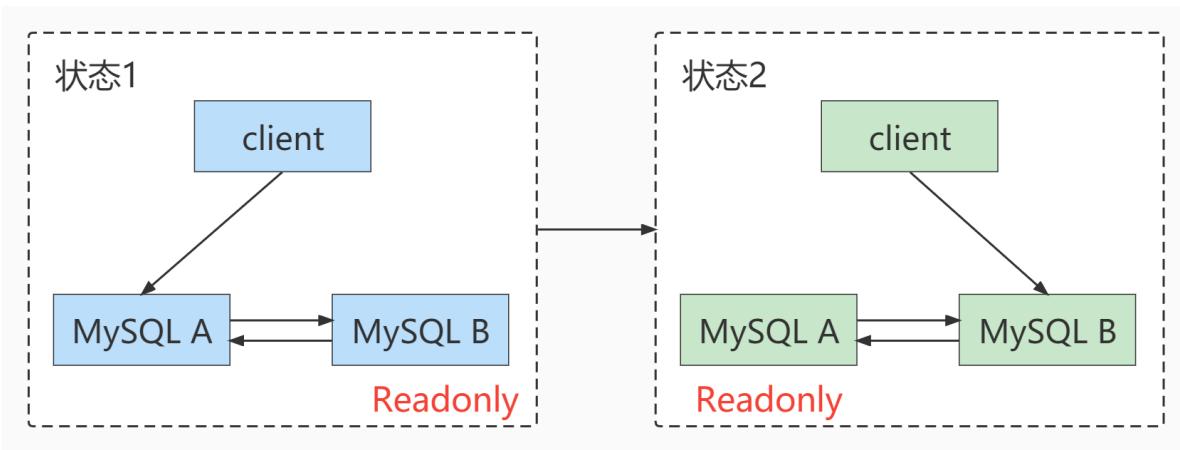
① **Cobar** 属于阿里B2B事业群，始于2008年，在阿里服役3年多，接管3000+个MySQL数据库的 schema，集群日处理在线SQL请求50亿次以上。由于Cobar发起人的离职，Cobar停止维护。

② **Mycat** 是开源社区在阿里cobar基础上进行二次开发，解决了cobar存在的问题，并且加入了许多新的功能在其中。青出于蓝而胜于蓝。

- ③ **OneProxy** 基于MySQL官方的proxy思想利用c语言进行开发的，OneProxy是一款商业 收费 的中间件。舍弃了一些功能，专注在 性能和稳定性上。
- ④ **kingshard** 由小团队用go语言开发，还需要发展，需要不断完善。
- ⑤ **Vitess** 是Youtube生产在使用，架构很复杂。不支持MySQL原生协议，使用 需要大量改造成本。
- ⑥ **Atlas** 是360团队基于mysql proxy改写，功能还需完善，高并发下不稳定。
- ⑦ **MaxScale** 是mariadb（MySQL原作者维护的一个版本）研发的中间件
- ⑧ **MySQLRoute** 是MySQL官方Oracle公司发布的中间件



主备切换：



- 主动切换
- 被动切换
- 如何判断主库出问题了？如何解决过程中的数据不一致性问题？

第19章_数据库备份与恢复

讲师：尚硅谷·宋红康（江湖人称：康师傅）

官网：<http://www.atguigu.com>

1. 物理备份与逻辑备份

物理备份：备份数据文件，转储数据库物理文件到某一目录。物理备份恢复速度比较快，但占用空间比较大，MySQL中可以用 `xtrabackup` 工具来进行物理备份。

逻辑备份：对数据库对象利用工具进行导出工作，汇总入备份文件内。逻辑备份恢复速度慢，但占用空间小，更灵活。MySQL 中常用的逻辑备份工具为 `mysqldump`。逻辑备份就是 **备份sql语句**，在恢复的时候执行备份的sql语句实现数据库数据的重现。

2. mysqldump实现逻辑备份

2.1 备份一个数据库

基本语法：

```
mysqldump -u 用户名称 -h 主机名称 -p密码 待备份的数据库名称[ tbname, [ tbname... ] ]> 备份文件名  
称.sql
```

说明：备份的文件并非一定要求后缀名为.sql，例如后缀名为.txt的文件也是可以的。

举例：使用root用户备份atguigu数据库：

```
mysqldump -uroot -p atguigu>atguigu.sql #备份文件存储在当前目录下
```

```
mysqldump -uroot -p atguigudb1 > /var/lib/mysql/atguigu.sql
```

备份文件剖析：

```
-- MySQL dump 10.13 Distrib 8.0.26, for Linux (x86_64)
--
-- Host: localhost      Database: atguigu
-- -----
-- Server version      8.0.26

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40503 SET NAMES utf8mb4 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0 */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;
```

```

/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;

-- 
-- Current Database: `atguigu`
-- 

CREATE DATABASE /*!32312 IF NOT EXISTS*/ `atguigu` /*!40100 DEFAULT CHARACTER SET
utf8mb4 COLLATE utf8mb4_0900_ai_ci */ /*!80016 DEFAULT ENCRYPTION='N' */;

USE `atguigu`;

-- 
-- Table structure for table `student`
-- 

DROP TABLE IF EXISTS `student`;
/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!50503 SET character_set_client = utf8mb4 */;
CREATE TABLE `student` (
  `studentno` int NOT NULL,
  `name` varchar(20) DEFAULT NULL,
  `class` varchar(20) DEFAULT NULL,
  PRIMARY KEY (`studentno`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3;
/*!40101 SET character_set_client = @saved_cs_client */;
INSERT INTO `student` VALUES (1,'张三_back','一班'),(3,'李四','一班'),(8,'王五','二班'),
(15,'赵六','二班'),(20,'钱七','>三班'),(22,'zhang3_update','1ban'),(24,'wang5','2ban');
/*!40000 ALTER TABLE `student` ENABLE KEYS */;
UNLOCK TABLES;
.

.

.

/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
/*!40014 SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS */;
/*!40014 SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS */;
/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
/*!40111 SET SQL_NOTES=@OLD_SQL_NOTES */;

-- Dump completed on 2022-01-07  9:58:23

```

2.2 备份全部数据库

若想用mysqldump备份整个实例，可以使用 `--all-databases` 或 `-A` 参数：

```

mysqldump -uroot -pxxxxxxx --all-databases > all_database.sql
mysqldump -uroot -pxxxxxxx -A > all_database.sql

```

2.3 备份部分数据库

使用 `--databases` 或 `-B` 参数了，该参数后面跟数据库名称，多个数据库间用空格隔开。如果指定 `databases` 参数，备份文件中会存在创建数据库的语句，如果不指定参数，则不存在。语法如下：

```
mysqldump -u user -h host -p --databases [数据库的名称1 [数据库的名称2...]] > 备份文件名  
称.sql
```

举例：

```
mysqldump -uroot -p --databases atguigu atguigu12 > two_database.sql
```

或

```
mysqldump -uroot -p -B atguigu atguigu12 > two_database.sql
```

2.4 备份部分表

比如，在表变更前做个备份。语法如下：

```
mysqldump -u user -h host -p 数据库的名称 [表名1 [表名2...]] > 备份文件名称.sql
```

举例：备份atguigu数据库下的book表

```
mysqldump -uroot -p atguigu book> book.sql
```

book.sql文件内容如下

```
mysqldump -uroot -p atguigu book> book.sql^C  
[root@node1 ~]# ls  
kk kubekey kubekey-v1.1.1-linux-amd64.tar.gz README.md test1.sql two_database.sql  
[root@node1 ~]# mysqldump -uroot -p atguigu book> book.sql  
Enter password:  
[root@node1 ~]# ls  
book.sql kk kubekey kubekey-v1.1.1-linux-amd64.tar.gz README.md test1.sql  
two_database.sql  
[root@node1 ~]# vi book.sql  
-- MySQL dump 10.13 Distrib 8.0.26, for Linux (x86_64)  
--  
-- Host: localhost Database: atguigu  
--  
-- Server version 8.0.26  
  
/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;  
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;  
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;  
/*!50503 SET NAMES utf8mb4 */;  
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;  
/*!40103 SET TIME_ZONE='+00:00' */;  
/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;  
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0 */;  
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;  
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;  
  
--  
-- Table structure for table `book`  
--
```

```

DROP TABLE IF EXISTS `book`;
/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!50503 SET character_set_client = utf8mb4 */;
CREATE TABLE `book` (
  `bookid` int unsigned NOT NULL AUTO_INCREMENT,
  `card` int unsigned NOT NULL,
  `test` varchar(255) COLLATE utf8_bin DEFAULT NULL,
  PRIMARY KEY (`bookid`),
  KEY `Y` (`card`)
) ENGINE=InnoDB AUTO_INCREMENT=101 DEFAULT CHARSET=utf8mb3 COLLATE=utf8_bin;
/*!40101 SET character_set_client = @saved_cs_client */;

-- 
-- Dumping data for table `book`
-- 

LOCK TABLES `book` WRITE;
/*!40000 ALTER TABLE `book` DISABLE KEYS */;
INSERT INTO `book` VALUES (1,9,NULL),(2,10,NULL),(3,4,NULL),(4,8,NULL),(5,7,NULL),
(6,10,NULL),(7,11,NULL),(8,3,NULL),(9,1,NULL),(10,17,NULL),(11,19,NULL),(12,4,NULL),
(13,1,NULL),(14,14,NULL),(15,5,NULL),(16,5,NULL),(17,8,NULL),(18,3,NULL),(19,12,NULL),
(20,11,NULL),(21,9,NULL),(22,20,NULL),(23,13,NULL),(24,3,NULL),(25,18,NULL),
(26,20,NULL),(27,5,NULL),(28,6,NULL),(29,15,NULL),(30,15,NULL),(31,12,NULL),
(32,11,NULL),(33,20,NULL),(34,5,NULL),(35,4,NULL),(36,6,NULL),(37,17,NULL),
(38,5,NULL),(39,16,NULL),(40,6,NULL),(41,18,NULL),(42,12,NULL),(43,6,NULL),
(44,12,NULL),(45,2,NULL),(46,12,NULL),(47,15,NULL),(48,17,NULL),(49,2,NULL),
(50,16,NULL),(51,13,NULL),(52,17,NULL),(53,7,NULL),(54,2,NULL),(55,9,NULL),
(56,1,NULL),(57,14,NULL),(58,7,NULL),(59,15,NULL),(60,12,NULL),(61,13,NULL),
(62,8,NULL),(63,2,NULL),(64,6,NULL),(65,2,NULL),(66,12,NULL),(67,12,NULL),(68,4,NULL),
(69,5,NULL),(70,10,NULL),(71,16,NULL),(72,8,NULL),(73,14,NULL),(74,5,NULL),
(75,4,NULL),(76,3,NULL),(77,2,NULL),(78,2,NULL),(79,2,NULL),(80,3,NULL),(81,8,NULL),
(82,14,NULL),(83,5,NULL),(84,4,NULL),(85,2,NULL),(86,20,NULL),(87,12,NULL),
(88,1,NULL),(89,8,NULL),(90,18,NULL),(91,3,NULL),(92,3,NULL),(93,6,NULL),(94,1,NULL),
(95,4,NULL),(96,17,NULL),(97,15,NULL),(98,1,NULL),(99,20,NULL),(100,15,NULL);
/*!40000 ALTER TABLE `book` ENABLE KEYS */;
UNLOCK TABLES;
/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;
```

可以看到，book文件和备份的库文件类似。不同的是，book文件只包含book表的DROP、CREATE和INSERT语句。

备份多张表使用下面的命令，比如备份book和account表：

```
#备份多张表
mysqldump -uroot -p atguigu book account > 2_tables_bak.sql
```

2.5 备份单表的部分数据

有些时候一张表的数据量很大，我们只需要部分数据。这时就可以使用 `--where` 选项了。`where`后面附带需要满足的条件。

举例：备份student表中id小于10的数据：

```
mysqldump -uroot -p atguigu student --where="id < 10 " > student_part_id10_low_bak.sql
```

内容如下所示，insert语句只有id小于10的部分

```
LOCK TABLES `student` WRITE;
/*!40000 ALTER TABLE `student` DISABLE KEYS */;
INSERT INTO `student` VALUES (1,100002,'JugxTY',157,280),(2,100003,'QyUcCJ',251,277),
(3,100004,'1LATUPp',80,404),(4,100005,'BmFsXI',240,171),(5,100006,'mkpSwJ',388,476),
(6,100007,'ujMgwN',259,124),(7,100008,'HBJTqX',429,168),(8,100009,'dvQSQ',61,504),
(9,100010,'HljpVJ',234,185);
```

2.6 排除某些表的备份

如果我们想备份某个库，但是某些表数据量很大或者与业务关联不大，这个时候可以考虑排除掉这些表，同样的，选项 `--ignore-table` 可以完成这个功能。

```
mysqldump -uroot -p atguigu --ignore-table=atguigu.student > no_stu_bak.sql
```

通过如下指定判定文件中没有student表结构：

```
grep "student" no_stu_bak.sql
```

2.7 只备份结构或只备份数据

只备份结构的话可以使用 `--no-data` 简写为 `-d` 选项；只备份数据可以使用 `--no-create-info` 简写为 `-t` 选项。

- 只备份结构

```
mysqldump -uroot -p atguigu --no-data > atguigu_no_data_bak.sql
#使用grep命令，没有找到insert相关语句，表示没有数据备份。
[root@node1 ~]# grep "INSERT" atguigu_no_data_bak.sql
[root@node1 ~]#
```

- 只备份数据

```
mysqldump -uroot -p atguigu --no-create-info > atguigu_no_create_info_bak.sql
#使用grep命令，没有找到create相关语句，表示没有数据结构。
[root@node1 ~]# grep "CREATE" atguigu_no_create_info_bak.sql
[root@node1 ~]#
```

2.8 备份中包含存储过程、函数、事件

mysqldump备份默认是不包含存储过程，自定义函数及事件的。可以使用 `--routines` 或 `-R` 选项来备份存储过程及函数，使用 `--events` 或 `-E` 参数来备份事件。

举例：备份整个atguigu库，包含存储过程及事件：

- 使用下面的SQL可以查看当前库有哪些存储过程或者函数

```
mysql> SELECT SPECIFIC_NAME,ROUTINE_TYPE ,ROUTINE_SCHEMA  FROM
information_schema.Routines WHERE ROUTINE_SCHEMA="atguigu";

+-----+-----+-----+
| SPECIFIC_NAME | ROUTINE_TYPE | ROUTINE_SCHEMA |
+-----+-----+-----+
| rand_num      | FUNCTION     | atguigu       |
| rand_string   | FUNCTION     | atguigu       |
| BatchInsert   | PROCEDURE    | atguigu       |
| insert_class  | PROCEDURE    | atguigu       |
| insert_order  | PROCEDURE    | atguigu       |
```

```
| insert_stu    | PROCEDURE    | atguigu      |
| insert_user   | PROCEDURE    | atguigu      |
| ts_insert     | PROCEDURE    | atguigu      |
+-----+-----+-----+
9 rows in set (0.02 sec)
```

下面备份atguigu库的数据，函数以及存储过程。

```
mysqldump -uroot -p -R -E --databases atguigu > fun_atguigu_bak.sql
```

查询备份文件中是否存在函数，如下所示，可以看到确实包含了函数。

```
grep -C 5 "rand_num" fun_atguigu_bak.sql
-- 

-- 
-- Dumping routines for database 'atguigu'
-- 

/*!50003 DROP FUNCTION IF EXISTS `rand_num` */;
/*!50003 SET @saved_cs_client      = @@character_set_client */ ;
/*!50003 SET @saved_cs_results     = @@character_set_results */ ;
/*!50003 SET @saved_col_connection = @@collation_connection */ ;
/*!50003 SET character_set_client  = utf8mb3 */ ;
/*!50003 SET character_set_results = utf8mb3 */ ;
/*!50003 SET collation_connection  = utf8_general_ci */ ;
/*!50003 SET @saved_sql_mode       = @@sql_mode */ ;
/*!50003 SET sql_mode              =
'ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION' */ ;
DELIMITER ;;
CREATE DEFINER=`root`@`%` FUNCTION `rand_num`(from_num BIGINT ,to_num BIGINT) RETURNS
bigint
BEGIN
DECLARE i BIGINT DEFAULT 0;
SET i = FLOOR(from_num +RAND()*(to_num - from_num+1)) ;
RETURN i;
END ;;
-- 
BEGIN
DECLARE i INT DEFAULT 0;
SET autocommit = 0;
REPEAT
SET i = i + 1;
INSERT INTO class ( classname,address,monitor ) VALUES
(rand_string(8),rand_string(10),rand_num());
UNTIL i = max_num
END REPEAT;
COMMIT;
END ;;
DELIMITER ;
-- 
BEGIN
DECLARE i INT DEFAULT 0;
SET autocommit = 0;      #设置手动提交事务
REPEAT  #循环
SET i = i + 1;  #赋值
INSERT INTO order_test (order_id, trans_id ) VALUES
(rand_num(1,7000000),rand_num(10000000000000000,7000000000000000));
```

```

UNTIL i = max_num
END REPEAT;
COMMIT; #提交事务
END ;;
DELIMITER ;
-- 
BEGIN
DECLARE i INT DEFAULT 0;
SET autocommit = 0;      #设置手动提交事务
REPEAT #循环
SET i = i + 1; #赋值
INSERT INTO student (stuno, name ,age ,classId ) VALUES
((START+i),rand_string(6),rand_num(),rand_num());
UNTIL i = max_num
END REPEAT;
COMMIT; #提交事务
END ;;
DELIMITER ;
-- 
BEGIN
DECLARE i INT DEFAULT 0;
SET autocommit = 0;
REPEAT
SET i = i + 1;
INSERT INTO `user` ( name,age,sex ) VALUES ("atguigu",rand_num(1,20),"male");
UNTIL i = max_num
END REPEAT;
COMMIT;
END ;;
DELIMITER ;

```

2.9 mysqldump常用选项

mysqldump其他常用选项如下：

- add-drop-database: 在每个CREATE DATABASE语句前添加DROP DATABASE语句。
- add-drop-tables: 在每个CREATE TABLE语句前添加DROP TABLE语句。
- add-locking: 用LOCK TABLES和UNLOCK TABLES语句引用每个表转储。重载转储文件时插入得更快。
- all-database, -A: 转储所有数据库中的所有表。与使用--database选项相同，在命令行中命名所有数据库。
- comment[=0|1]: 如果设置为0，禁止转储文件中的其他信息，例如程序版本、服务器版本和主机。--skip-comments与--comments=0的结果相同。默认值为1，即包括额外信息。
- compact: 产生少量输出。该选项禁用注释并启用--skip-add-drop-tables、--no-set-names、--skip-disable-keys和--skip-add-locking选项。
- compatible=name: 产生与其他数据库系统或旧的MySQL服务器更兼容的输出，值可以为ansi、MySQL323、MySQL40、postgresql、oracle、mssql、db2、maxdb、no_key_options、no_table_options或者no_field_options。
- complete_insert, -c: 使用包括列名的完整的INSERT语句。
- debug[=debug_options], -#[debug_options]: 写调试日志。

--delete, -D: 导入文本文件前清空表。

--default-character-set=charset: 使用charsets默认字符集。如果没有指定，就使用utf8。

--delete--master-logs: 在主复制服务器上，完成转储操作后删除二进制日志。该选项自动启用--master-data。

--extended-insert, -e: 使用包括几个VALUES列表的多行INSERT语法。这样使得转储文件更小，重载文件时可以加速插入。

--flush-logs, -F: 开始转储前刷新MySQL服务器日志文件。该选项要求RELOAD权限。

--force, -f: 在表转储过程中，即使出现SQL错误也继续。

--lock-all-tables, -x: 对所有数据库中的所有表加锁。在整体转储过程中通过全局锁定来实现。该选项自动关闭--single-transaction和--lock-tables。

--lock-tables, -l: 开始转储前锁定所有表。用READ LOCAL锁定表以允许并行插入MyISAM表。对于事务表（例如InnoDB和BDB），--single-transaction是一个更好的选项，因为它根本不需要锁定表。

--no-create-db, -n: 该选项禁用CREATE DATABASE /*!32312 IF NOT EXIST*/db_name语句，如果给出--database或--all-database选项，就包含到输出中。

--no-create-info, -t: 只导出数据，而不添加CREATE TABLE语句。

--no-data, -d: 不写表的任何行信息，只转储表的结构。

--opt: 该选项是速记，它可以快速进行转储操作并产生一个能很快装入MySQL服务器的转储文件。该选项默认开启，但可以用--skip-opt禁用。

--password[=password], -p[password]: 当连接服务器时使用的密码。

-port=port_num, -P port_num: 用于连接的TCP/IP端口号。

--protocol={TCP|SOCKET|PIPE|MEMORY}: 使用的连接协议。

--replace, -r --replace和--ignore: 控制替换或复制唯一键值已有记录的输入记录的处理。如果指定--replace，新行替换有相同的唯一键值的已有行；如果指定--ignore，复制已有的唯一键值的输入行被跳过。如果不指定这两个选项，当发现一个复制键值时会出现一个错误，并且忽视文本文件的剩余部分。

--silent, -s: 沉默模式。只有出现错误时才输出。

--socket=path, -S path: 当连接localhost时使用的套接字文件（为默认主机）。

--user=user_name, -u user_name: 当连接服务器时MySQL使用的用户名。

--verbose, -v: 兀长模式，打印出程序操作的详细信息。

--xml, -X: 产生XML输出。

运行帮助命令 `mysqldump --help`，可以获得特定版本的完整选项列表。

提示 如果运行mysqldump没有--quick或--opt选项，mysqldump在转储结果前将整个结果集装入内存。如果转储大数据库可能会出现问题，该选项默认启用，但可以用--skip-opt禁用。如果使用最新版本的mysqldump程序备份数据，并用于恢复到比较旧版本的MySQL服务器中，则不要使用--opt或-e选项。

3. mysql命令恢复数据

基本语法：

```
mysql -u root -p [dbname] < backup.sql
```

3.1 单库备份中恢复单库

使用root用户，将之前练习中备份的atguigu.sql文件中的备份导入数据库中，命令如下：

如果备份文件中包含了创建数据库的语句，则恢复的时候不需要指定数据库名称，如下所示

```
mysql -uroot -p < atguigu.sql
```

否则需要指定数据库名称，如下所示

```
mysql -uroot -p atguigu4< atguigu.sql
```

3.2 全量备份恢复

如果我们现在有昨天的全量备份，现在想整个恢复，则可以这样操作：

```
mysql -u root -p < all.sql
```

```
mysql -uroot -pxxxxxxx < all.sql
```

执行完后，MySQL数据库中就已经恢复了all.sql文件中的所有数据库。

3.3 从全量备份中恢复单库

可能有这样的需求，比如说我们只想恢复某一个库，但是我们有的是整个实例的备份，这个时候我们可以从全量备份中分离出单个库的备份。

举例：

```
sed -n '/^-- Current Database: `atguigu`/,/^-- Current Database: `/p' all_database.sql > atguigu.sql
```

#分离完成后我们再导入atguigu.sql即可恢复单个库

3.4 从单库备份中恢复单表

这个需求还是比较常见的。比如说我们知道哪个表误操作了，那么就可以用单表恢复的方式来恢复。

举例：我们有atguigu整库的备份，但是由于class表误操作，需要单独恢复出这张表。

```
cat atguigu.sql | sed -e '/.{H;$!d;}' -e 'x;/CREATE TABLE `class`/!d;q' > class_structure.sql
cat atguigu.sql | grep --ignore-case 'insert into `class`' > class_data.sql
#用shell语法分离出创建表的语句及插入数据的语句后 再依次导出即可完成恢复

use atguigu;
mysql> source class_structure.sql;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> source class_data.sql;
Query OK, 1 row affected (0.01 sec)
```

4. 物理备份：直接复制整个数据库

直接将MySQL中的数据库文件复制出来。这种方法最简单，速度也最快。MySQL的数据库目录位置不一定相同：

- 在Windows平台下，MySQL 8.0存放数据库的目录通常默认为“`C:\ProgramData\MySQL\MySQL Server 8.0\Data`”或者其他用户自定义目录；
- 在Linux平台下，数据库目录位置通常为`/var/lib/mysql/`；
- 在MAC OSX平台下，数据库目录位置通常为`"/usr/local/mysql/data"`

但为了保证备份的一致性。需要保证：

- 方式1：备份前，将服务器停止。
- 方式2：备份前，对相关表执行`FLUSH TABLES WITH READ LOCK`操作。这样当复制数据库目录中的文件时，允许其他客户继续查询表。同时，`FLUSH TABLES`语句来确保开始备份前将所有激活的索引页写入硬盘。

这种方式方便、快速，但不是最好的备份方法，因为实际情况可能不允许停止MySQL服务器或者锁住表，而且这种方法对InnoDB存储引擎的表不适用。对于MyISAM存储引擎的表，这样备份和还原很方便，但是还原时最好是相同版本的MySQL数据库，否则可能会存在文件类型不同的情况。

注意，物理备份完毕后，执行`UNLOCK TABLES`来结算其他客户对表的修改行为。

说明：在MySQL版本号中，第一个数字表示主版本号，主版本号相同的MySQL数据库文件格式相同。

此外，还可以考虑使用相关工具实现备份。比如，`MySQLhotcopy`工具。`MySQLhotcopy`是一个Perl脚本，它使用`LOCK TABLES`、`FLUSH TABLES`和`cp`或`scp`来快速备份数据库。它是备份数据库或单个表最快的途径，但它只能运行在数据库目录所在的机器上，并且只能备份MyISAM类型的表。多用于mysql5.5之前。

5. 物理恢复：直接复制到数据库目录

步骤：

- 1) 演示删除备份的数据库中指定表的数据
- 2) 将备份的数据库数据拷贝到数据目录下，并重启MySQL服务器

3) 查询相关表的数据是否恢复。需要使用下面的 `chown` 操作。

要求:

- 必须确保备份数据的数据库和待恢复的数据库服务器的主版本号相同。
 - 因为只有MySQL数据库主版本号相同时，才能保证这两个MySQL数据库文件类型是相同的。
- 这种方式对 MyISAM类型的表比较有效，对于InnoDB类型的表则不可用。
 - 因为InnoDB表的表空间不能直接复制。
- 在Linux操作系统下，复制到数据库目录后，一定要将数据库的用户和组变成mysql，命令如下：

```
chown -R mysql.mysql /var/lib/mysql/dbname
```

其中，两个mysql分别表示组和用户；“-R”参数可以改变文件夹下的所有子文件的用户和组；“dbname”参数表示数据库目录。

提示 Linux操作系统下的权限设置非常严格。通常情况下，MySQL数据库只有root用户和mysql用户组下的mysql用户才可以访问，因此将数据库目录复制到指定文件夹后，一定要使用chown命令将文件夹的用户组变为mysql，将用户变为mysql。

6. 表的导出与导入

6.1 表的导出

1. 使用SELECT...INTO OUTFILE导出文本文件

在MySQL中，可以使用SELECT...INTO OUTFILE语句将表的内容导出成一个文本文件。

举例：使用SELECT...INTO OUTFILE将atguigu数据库中account表中的记录导出到文本文件。（1）选择数据库atguigu，并查询account表，执行结果如下所示。

```
use atguigu;
select * from account;
mysql> select * from account;
+----+-----+-----+
| id | name  | balance |
+----+-----+-----+
| 1  | 张三  |      90 |
| 2  | 李四  |     100 |
| 3  | 王五  |       0 |
+----+-----+-----+
3 rows in set (0.01 sec)
```

(2) mysql默认对导出的目录有权限限制，也就是说使用命令行进行导出的时候，需要指定目录进行操作。

查询secure_file_priv值：

```
mysql> SHOW GLOBAL VARIABLES LIKE '%secure%';
+-----+-----+
| Variable_name      | Value       |
+-----+-----+
| require_secure_transport | OFF        |
| secure_file_priv     | /var/lib/mysql-files/ |
+-----+-----+
2 rows in set (0.02 sec)
```

(3) 上面结果中显示, secure_file_priv变量的值为/var/lib/mysql-files/, 导出目录设置为该目录, SQL语句如下。

```
SELECT * FROM account INTO OUTFILE "/var/lib/mysql-files/account.txt";
```

(4) 查看 /var/lib/mysql-files/account.txt`文件。

```
1    张三    90
2    李四    100
3    王五    0
```

2. 使用mysqldump命令导出文本文件

举例1： 使用mysqldump命令将atguigu数据库中account表中的记录导出到文本文件:

```
mysqldump -uroot -p -T "/var/lib/mysql-files/" atguigu account
```

mysqldump命令执行完毕后, 在指定的目录/var/lib/mysql-files/下生成了account.sql和account.txt文件。

打开account.sql文件, 其内容包含创建account表的CREATE语句。

```
[root@node1 mysql-files]# cat account.sql
-- MySQL dump 10.13 Distrib 8.0.26, for Linux (x86_64)
--
-- Host: localhost      Database: atguigu
-- 
-- Server version      8.0.26

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!50503 SET NAMES utf8mb4 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE=''' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;

--
-- Table structure for table `account`
-- 

DROP TABLE IF EXISTS `account`;
/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!50503 SET character_set_client = utf8mb4 */;
CREATE TABLE `account` (
  `id` int NOT NULL AUTO_INCREMENT,
  `name` varchar(255) NOT NULL,
  `balance` int NOT NULL,
  PRIMARY KEY (`id`)
```

```

) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8mb3;
/*!40101 SET character_set_client = @saved_cs_client */;

/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;

/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
/*!40111 SET SQL_NOTES=@OLD_SQL_NOTES */;

-- Dump completed on 2022-01-07 23:19:27

```

打开account.txt文件，其内容只包含account表中的数据。

```
[root@node1 mysql-files]# cat account.txt
1      张三    90
2      李四    100
3      王五    0
```

举例2：使用mysqldump将atguigu数据库中的account表导出到文本文件，使用FIELDS选项，要求字段之间使用逗号“，”间隔，所有字符类型字段值用双引号括起来：

```
mysqldump -uroot -p -T "/var/lib/mysql-files/" atguigu account --fields-terminated-by=',' --fields-optionally-enclosed-by='\"'
```

语句mysqldump语句执行成功之后，指定目录下会出现两个文件account.sql和account.txt。

打开account.sql文件，其内容包含创建account表的CREATE语句。

```
[root@node1 mysql-files]# cat account.sql
-- MySQL dump 10.13 Distrib 8.0.26, for Linux (x86_64)
--
-- Host: localhost      Database: atguigu
-- -----
-- Server version      8.0.26

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!50503 SET NAMES utf8mb4 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE=' ' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;

--
-- Table structure for table `account`
--

DROP TABLE IF EXISTS `account`;
/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!50503 SET character_set_client = utf8mb4 */;
CREATE TABLE `account` (
  `id` int NOT NULL AUTO_INCREMENT,
  `name` varchar(255) NOT NULL,
  `balance` int NOT NULL,
  PRIMARY KEY (`id`)
```

```
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8mb3;
/*!40101 SET character_set_client = @saved_cs_client */;

/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;

/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
/*!40111 SET SQL_NOTES=@OLD_SQL_NOTES */;

-- Dump completed on 2022-01-07 23:36:39
```

打开account.txt文件，其内容包含创建account表的数据。从文件中可以看出，字段之间用逗号隔开，字符串类型的值被双引号括起来。

```
[root@node1 mysql-files]# cat account.txt
1, "张三", 90
2, "李四", 100
3, "王五", 0
```

3. 使用mysql命令导出文本文件

举例1：使用mysql语句导出atguigu数据中account表中的记录到文本文件：

```
mysql -uroot -p --execute="SELECT * FROM account;" atguigu > "/var/lib/mysql-
files/account.txt"
```

打开account.txt文件，其内容包含创建account表的数据。

```
[root@node1 mysql-files]# cat account.txt
id      name      balance
1       张三      90
2       李四      100
3       王五      0
```

举例2：将atguigu数据库account表中的记录导出到文本文件，使用--vertical参数将该条件记录分为多行显示：

```
mysql -uroot -p --vertical --execute="SELECT * FROM account;" atguigu >
"/var/lib/mysql-files/account_1.txt"
```

打开account_1.txt文件，其内容包含创建account表的数据。

```
[root@node1 mysql-files]# cat account_1.txt
***** 1. row *****
    id: 1
    name: 张三
    balance: 90
***** 2. row *****
    id: 2
    name: 李四
    balance: 100
***** 3. row *****
    id: 3
    name: 王五
    balance: 0
```

举例3：将atguigu数据库account表中的记录导出到xml文件，使用--xml参数，具体语句如下。

```
mysql -uroot -p --xml --execute="SELECT * FROM account;" atguigu>"/var/lib/mysql-files/account_3.xml"

[root@node1 mysql-files]# cat account_3.xml
<?xml version="1.0"?>

<resultset statement="SELECT * FROM account"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<row>
    <field name="id">1</field>
    <field name="name">张三</field>
    <field name="balance">90</field>
</row>

<row>
    <field name="id">2</field>
    <field name="name">李四</field>
    <field name="balance">100</field>
</row>

<row>
    <field name="id">3</field>
    <field name="name">王五</field>
    <field name="balance">0</field>
</row>
</resultset>
```

说明：如果要将表数据导出到html文件中，可以使用--html选项。然后可以使用浏览器打开。

6.2 表的导入

1. 使用LOAD DATA INFILE方式导入文本文件

举例1：

使用SELECT...INTO OUTFILE将atguigu数据库中account表的记录导出到文本文件

```
SELECT * FROM atguigu.account INTO OUTFILE '/var/lib/mysql-files/account_0.txt';
```

删除account表中的数据：

```
DELETE FROM atguigu.account;
```

从文本文件account.txt中恢复数据：

```
LOAD DATA INFILE '/var/lib/mysql-files/account_0.txt' INTO TABLE atguigu.account;
```

查询account表中的数据：

```
mysql> select * from account;
+----+-----+-----+
| id | name  | balance |
+----+-----+-----+
| 1  | 张三   |      90 |
| 2  | 李四   |     100 |
| 3  | 王五   |       0 |
+----+-----+-----+
3 rows in set (0.00 sec)
```

举例2：选择数据库atguigu，使用SELECT...INTO OUTFILE将atguigu数据库account表中的记录导出到文本文件，使用FIELDS选项和LINES选项，要求字段之间使用逗号”，“间隔，所有字段值用双引号括起来：

```
SELECT * FROM atguigu.account INTO OUTFILE '/var/lib/mysql-files/account_1.txt' FIELDS
TERMINATED BY ',' ENCLOSED BY '\"';
```

删除account表中的数据：

```
DELETE FROM atguigu.account;
```

从/var/lib/mysql-files/account.txt中导入数据到account表中：

```
LOAD DATA INFILE '/var/lib/mysql-files/account_1.txt' INTO TABLE atguigu.account
FIELDS TERMINATED BY ',' ENCLOSED BY '\"';
```

查询account表中的数据，具体SQL如下：

```
select * from account;
mysql> select * from account;
+----+-----+-----+
| id | name  | balance |
+----+-----+-----+
| 1  | 张三   |      90 |
| 2  | 李四   |     100 |
| 3  | 王五   |       0 |
+----+-----+-----+
3 rows in set (0.00 sec)
```

2. 使用mysqlimport方式导入文本文件

举例：

导出文件account.txt，字段之间使用逗号”，“间隔，字段值用双引号括起来：

```
SELECT * FROM atguigu.account INTO OUTFILE '/var/lib/mysql-files/account.txt' FIELDS
TERMINATED BY ',' ENCLOSED BY '\"';
```

删除account表中的数据：

```
DELETE FROM atguigu.account;
```

使用mysqlimport命令将account.txt文件内容导入到数据库atguigu的account表中：

```
mysqlimport -uroot -p atguigu '/var/lib/mysql-files/account.txt' --fields-terminated-
by=',' --fields-optionally-enclosed-by='\"'
```

查询account表中的数据：

```
select * from account;
mysql> select * from account;
+----+-----+-----+
| id | name  | balance |
+----+-----+-----+
| 1  | 张三   |      90 |
| 2  | 李四   |     100 |
| 3  | 王五   |       0 |
+----+-----+-----+
3 rows in set (0.00 sec)
```

7. 数据库迁移

7.1 概述

数据迁移（data migration）是指选择、准备、提取和转换数据，并将数据从一个计算机存储系统永久地传输到另一个计算机存储系统的过程。此外，验证迁移数据的完整性 和 退役原来旧的数据存储，也被认为是整个数据迁移过程的一部分。

数据库迁移的原因是多样的，包括服务器或存储设备更换、维护或升级，应用程序迁移，网站集成，灾难恢复和数据中心迁移。

根据不同的需求可能要采取不同的迁移方案，但总体来讲，MySQL 数据迁移方案大致可以分为 物理迁移 和 逻辑迁移 两类。通常以尽可能 自动化 的方式执行，从而将人力资源从繁琐的任务中解放出来。

7.2 迁移方案

- 物理迁移

物理迁移适用于大数据量下的整体迁移。使用物理迁移方案的优点是比较快速，但需要停机迁移并且要求 MySQL 版本及配置必须和原服务器相同，也可能引起未知问题。

物理迁移包括拷贝数据文件和使用 XtraBackup 备份工具两种。

不同服务器之间可以采用物理迁移，我们可以在新的服务器上安装好同版本的数据库软件，创建好相同目录，建议配置文件也要和原数据库相同，然后从原数据库方拷贝来数据文件及日志文件，配置好文件组权限，之后在新服务器这边使用 mysqld 命令启动数据库。

- 逻辑迁移

逻辑迁移适用范围更广，无论是 部分迁移 还是 全量迁移，都可以使用逻辑迁移。逻辑迁移中使用最多的就是通过 mysqldump 等备份工具。

7.3 迁移注意点

1. 相同版本的数据库之间迁移注意点

指的是在主版本号相同的MySQL数据库之间进行数据库移动。

方式1：因为迁移前后MySQL数据库的 主版本号相同，所以可以通过复制数据库目录来实现数据库迁移，但是物理迁移方式只适用于MyISAM引擎的表。对于InnoDB表，不能用直接复制文件的方式备份数据库。

方式2：最常见和最安全的方式是使用 `mysqldump命令` 导出数据，然后在目标数据库服务器中使用 MySQL命令导入。

举例：

```
#host1的机器中备份所有数据库，并将数据库迁移到名为host2的机器上
mysqldump -h host1 -uroot -p --all-databases |
mysql -h host2 -uroot -p
```

在上述语句中，“|”符号表示管道，其作用是将mysqldump备份的文件给mysql命令；“--all-databases”表示要迁移所有的数据库。通过这种方式可以直接实现迁移。

2. 不同版本的数据库之间迁移注意点

例如，原来很多服务器使用5.7版本的MySQL数据库，在8.0版本推出来以后，改进了5.7版本的很多缺陷，因此需要把数据库升级到8.0版本

旧版本与新版本的MySQL可能使用不同的默认字符集，例如有的旧版本中使用latin1作为默认字符集，而最新版本的MySQL默认字符集为utf8mb4。如果数据库中有中文数据，那么迁移过程中需要对 [默认字符集进行修改](#)，不然可能无法正常显示数据。

高版本的MySQL数据库通常都会 [兼容低版本](#)，因此可以从低版本的MySQL数据库迁移到高版本的MySQL数据库。

3. 不同数据库之间迁移注意点

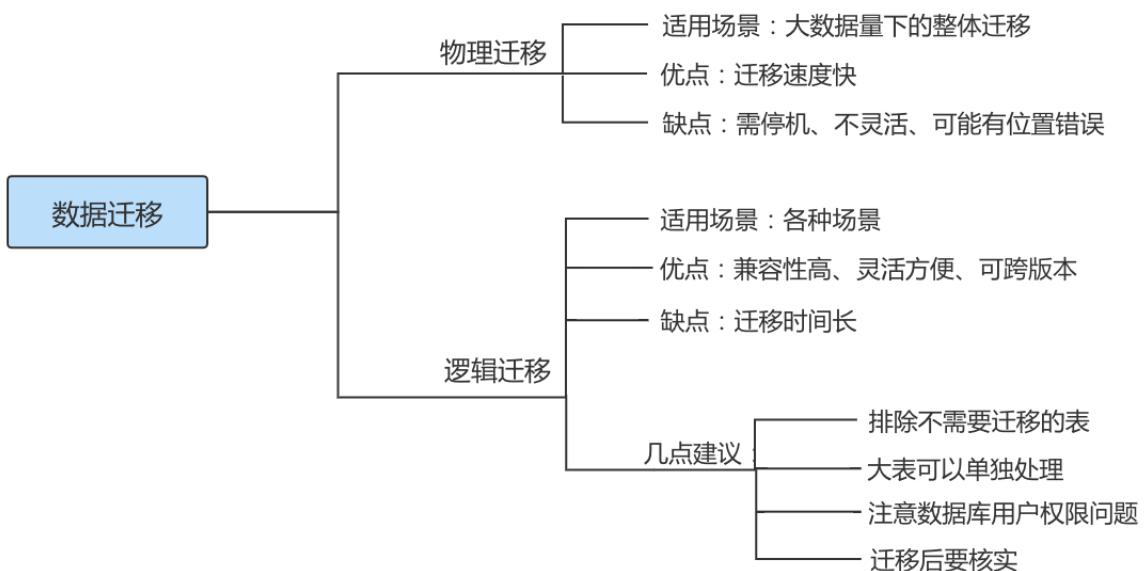
不同数据库之间迁移是指从其他类型的数据库迁移到MySQL数据库，或者从MySQL数据库迁移到其他类型的数据库。这种迁移没有普适的解决方法。

迁移之前，需要了解不同数据库的架构，[比较它们之间的差异](#)。不同数据库中定义相同类型的数据的 [关键字可能会不同](#)。例如，MySQL中日期字段分为DATE和TIME两种，而ORACLE日期字段只有DATE；SQL Server数据库中有ntext、Image等数据类型，MySQL数据库没有这些数据类型；MySQL支持的ENUM和SET类型，这些SQL Server数据库不支持。

另外，数据库厂商并没有完全按照SQL标准来设计数据库系统，导致不同的数据库系统的 [SQL语句](#) 有差别。例如，微软的SQL Server软件使用的是T-SQL语句，T-SQL中包含了非标准的SQL语句，不能和MySQL的SQL语句兼容。

不同类型数据库之间的差异造成了互相 [迁移的困难](#)，这些差异其实是商业公司故意造成的技术壁垒。但是不同类型的数据库之间的迁移并 [不是完全不可能](#)。例如，可以使用 [MyODBC](#) 实现MySQL和SQL Server之间的迁移。MySQL官方提供的工具 [MySQL Migration Toolkit](#) 也可以在不同数据之间进行数据迁移。MySQL迁移到Oracle时，需要使用mysqldump命令导出sql文件，然后，[手动更改](#) sql文件中的CREATE语句。

7.4 迁移小结



8. 删库了不敢跑，能干点啥？

8.1 delete：误删行

经验之谈：

1. 恢复数据比较安全的做法，是 **恢复出一个备份**，或者找一个从库作为 **临时库**，在这个临时库上执行这些操作，然后再将确认过的临时库的数据，恢复回主库。如果直接修改主库，可能导致对数据的 **二次破坏**。
2. 当然，针对预防误删数据的问题，建议如下：
 1. 把 `sql_safe_updates` 参数设置为 `on`。这样一来，如果我们忘记在 `delete` 或者 `update` 语句中写 `where` 条件，或者 `where` 条件里面没有包含索引字段的话，这条语句的执行就会报错。

如果确定要把一个小表的数据全部删掉，在设置了 `sql_safe_updates=on` 情况下，可以在 `delete` 语句中加上 `where` 条件，比如 `where id>=0`。

2. 代码上线前，必须经过 **SQL 审计**。

8.2 truncate/drop：误删库/表

方案：

这种情况下，要想恢复数据，就需要使用 **全量备份**，加 **增量日志** 的方式了。这个方案要求线上有定期的全量备份，并且实时备份 binlog。

在这两个条件都具备的情况下，假如有人中午12点误删了一个库，恢复数据的流程如下：

1. 取最近一次 **全量备份**，假设这个库是一天一备，上次备份是当天 **凌晨2点**；
2. 用备份恢复出一个 **临时库**；
3. 从日志备份里面，取出凌晨2点之后的日志；
4. 把这些日志，除了误删除数据的语句外，全部应用到临时库。

8.3 延迟复制备库

如果有 **非常核心** 的业务，不允许太长的恢复时间，可以考虑**搭建延迟复制的备库**。一般的主备复制结构存在的问题是，如果主库上有个表被误删了，这个命令很快也会被发给所有从库，进而导致所有从库的数据表也都一起被误删了。

延迟复制的备库是一种特殊的备库，通过 `CHANGE MASTER TO MASTER_DELAY = N` 命令，可以指定这个备库持续保持跟主库有 **N秒的延迟**。比如你把N设置为3600，这就代表了如果主库上有数据被误删了，并且在1小时内发现了这个误操作命令，这个命令就还没有在这个延迟复制的备库执行。这时候到这个备库上执行`stop slave`，再通过之前介绍的方法，跳过误操作命令，就可以恢复出需要的数据。

8.4 预防误删库/表的方法

1. **账号分离**。这样做的目的是，避免写错命令。比如：
 - 只给业务开发同学DML权限，而不给truncate/drop权限。而如果业务开发人员有DDL需求的话，可以通过开发管理系统得到支持。
 - 即使是DBA团队成员，日常也都规定只使用 **只读账号**，必要的时候才使用有更新权限的账号。
2. **制定操作规范**。比如：
 - 在删除数据表之前，必须先 **对表做改名** 操作。然后，观察一段时间，确保对业务无影响以后再删除这张表。
 - 改表名的时候，要求给表名加固定的后缀（比如加 `_to_be_deleted`），然后删除表的动作必须通过管理系统执行。并且，管理系统删除表的时候，只能删除固定后缀的表。

8.5 rm：误删MySQL实例

对于一个有高可用机制的MySQL集群来说，不用担心 **rm删除数据** 了。只是删掉了其中某一个节点的数据的话，HA系统就会开始工作，选出一个新的主库，从而保证整个集群的正常工作。我们要做的就是在这个节点上把数据恢复回来，再接入整个集群。

9.附录： MySQL常用命令

9.1 mysql

该mysql不是指mysql服务，而是指mysql的客户端工具。

语法：

```
mysql [options] [database]
```

1. 连接选项

```
#参数：  
-u, --user=name      指定用户名  
-p, --password[=name] 指定密码  
-h, --host=name      指定服务器IP或域名  
-P, --port=#         指定连接端口
```

```
#示例：  
mysql -h 127.0.0.1 -P 3306 -u root -p  
  
mysql -h127.0.0.1 -P3306 -uroot -p密码
```

2. 执行选项

```
-e, --execute=name    执行SQL语句并退出
```

此选项可以在Mysql客户端执行SQL语句，而不用连接到MySQL数据库再执行，对于一些批处理脚本，这种方式尤其方便。

```
#示例：  
mysql -uroot -p db01 -e "select * from tb_book";
```

```
[root@xaxh-server ~]# mysql -uroot -p2143 db01 -e "select * from tb_book";  
Warning: Using a password on the command line interface can be insecure.  
+----+-----+-----+  
| id | name          | publish_time | status |  
+----+-----+-----+  
| 1  | java编程思想（第二版） | 2088-08-01  | 1     |  
| 2  | solr 入门        | 2088-08-08  | 0     |  
| 3  | Mysql高级       | 2088-01-01  | 1     |  
| 4  | Netty           | 2088-08-08  | 0     |  
| 5  | lucene入门指南   | 2088-05-01  | 0     |  
| 6  | SpringCloud实战   | 2088-05-05  | 0     |  
+----+-----+-----+
```

9.2 mysqladmin

mysqladmin 是一个执行管理操作的客户端程序。可以用它来检查服务器的配置和当前状态、创建并删除数据库等。

可以通过： mysqladmin --help 指令查看帮助文档

```
create database name      Create a new database  
debug                  Instruct server to write debug information to log  
drop database name      Delete a database and all its tables  
extended-status         Gives an extended status message from the server  
flush-hosts            Flush all cached hosts  
flush-logs              Flush all logs  
flush-status            Clear status variables  
flush-tables            Flush all tables  
flush-threads           Flush the thread cache  
flush-privileges        Reload grant tables (same as reload)  
kill id,id,...          Kill mysql threads  
password [new-password] Change old password to new-password in current format  
old-password [new-password] Change old password to new-password in old format  
ping                   Check if mysqld is alive  
processlist             Show list of active threads in server  
reload                 Reload grant tables  
refresh                Flush all tables and close and open logfiles  
shutdown               Take server down  
status                 Gives a short status message from the server  
start-slave             Start slave  
stop-slave              Stop slave  
variables              Prints variables available  
version                Get version info from server
```

```
#示例：  
mysqladmin -uroot -p create 'test01';  
mysqladmin -uroot -p drop 'test01';  
mysqladmin -uroot -p version;
```

9.3 mysqlbinlog

由于服务器生成的二进制日志文件以二进制格式保存，所以如果想要检查这些文本的文本格式，就会使用到mysqlbinlog 日志管理工具。

语法：

```
mysqlbinlog [options] log-files1 log-files2 ...  
  
#选项：  
  
-d, --database=name : 指定数据库名称，只列出指定的数据库相关操作。  
  
-o, --offset=# : 忽略掉日志中的前n行命令。  
  
-r, --result-file=name : 将输出的文本格式日志输出到指定文件。  
  
-s, --short-form : 显示简单格式，省略掉一些信息。  
  
--start-datetime=date1 --stop-datetime=date2 : 指定日期间隔内的所有日志。  
  
--start-position=pos1 --stop-position=pos2 : 指定位置间隔内的所有日志。
```

9.4 mysqldump

mysqldump 客户端工具用来备份数据库或在不同数据库之间进行数据迁移。备份内容包含创建表，及插入表的SQL语句。

语法：

```
mysqldump [options] db_name [tables]  
  
mysqldump [options] --database/-B db1 [db2 db3...]  
  
mysqldump [options] --all-databases/-A
```

1. 连接选项

```
#参数：  
-u, --user=name      指定用户名  
-p, --password[=name] 指定密码  
-h, --host=name      指定服务器IP或域名  
-P, --port=#         指定连接端口
```

2. 输出内容选项

```
#参数：  
--add-drop-database    在每个数据库创建语句前加上 Drop database 语句  
--add-drop-table       在每个表创建语句前加上 Drop table 语句， 默认开启；不开启 (--  
skip-add-drop-table)  
  
-n, --no-create-db    不包含数据库的创建语句  
-t, --no-create-info   不包含数据表的创建语句  
-d --no-data          不包含数据  
  
-T, --tab=name         自动生成两个文件：一个.sql文件，创建表结构的语句；  
一个.txt文件，数据文件，相当于select into outfile
```

```
#示例 :
```

```
mysqldump -uroot -p db01 tb_book --add-drop-database --add-drop-table > a
```

```
mysqldump -uroot -p -T /tmp test city
```

```
-rw-r--r-- 1 root root 2625 Apr 17 19:45 city.sql  
-rw-rw-rw- 1 mysql mysql 50 Apr 17 19:45 city.txt
```

9.5 mysqlimport/source

mysqlimport 是客户端数据导入工具，用来导入mysqldump 加 -T 参数后导出的文本文件。

语法：

```
mysqlimport [options] db_name textfile1 [textfile2...]
```

示例：

```
mysqlimport -uroot -p test /tmp/city.txt
```

如果需要导入sql文件,可以使用mysql中的source 指令：

```
source /root/tb_book.sql
```

9.6 mysqlshow

mysqlshow 客户端对象查找工具，用来很快地查找存在哪些数据库、数据库中的表、表中的列或者索引。

语法：

```
mysqlshow [options] [db_name [table_name [col_name]]]
```

参数：

```
--count 显示数据库及表的统计信息（数据库，表 均可以不指定）
```

```
-i 显示指定数据库或者指定表的状态信息
```

示例：

```
#查询每个数据库的表的数量及表中记录的数量
mysqlshow -uroot -p --count
[root@node1 atguigu2]# mysqlshow -uroot -p --count
Enter password:
+-----+-----+-----+
| Databases | Tables | Total Rows |
+-----+-----+-----+
| atguigu | 24 | 30107483 |
| atguigu12 | 1 | 1 |
| atguigu14 | 6 | 14 |
| atguigu17 | 1 | 1 |
| atguigu18 | 0 | 0 |
| atguigu2 | 1 | 3 |
| atguigu_myisam | 1 | 4 |
| information_schema | 79 | 34034 |
| mysql | 38 | 4029 |
| performance_schema | 110 | 399957 |
| sys | 101 | 7028 |
```

```

+-----+-----+-----+
11 rows in set.

#查询test库中每个表中的字段数, 及行数
mysqlshow -uroot -p atguigu --count
[root@node1 atguigu2]# mysqlshow -uroot -p atguigu --count
Enter password:
Database: atguigu
+-----+-----+-----+
| Tables | Columns | Total Rows |
+-----+-----+-----+
| account | 3 | 3 |
| book | 3 | 100 |
| dept | 3 | 3 |
| emp | 8 | 10 |
| order1 | 2 | 5715448 |
| order2 | 2 | 8000327 |
| order_test | 2 | 8000327 |
| salgrade | 3 | 0 |
| stu2 | 6 | 5 |
| student | 5 | 8100010 |
| t1 | 3 | 210000 |
| t_class | 3 | 0 |
| test | 2 | 0 |
| test_frm | 2 | 0 |
| test_paper | 1 | 0 |
| ts1 | 2 | 79999 |
| type | 2 | 240 |
| undo_demo | 3 | 1 |
| user | 1 | 1 |
| user1 | 4 | 1000 |
+-----+-----+-----+
20 rows in set.

#查询test库中book表的详细情况
mysqlshow -uroot -p atguigu book --count
[root@node1 atguigu2]# mysqlshow -uroot -p atguigu book --count
Enter password:
Database: atguigu  Table: book  Rows: 100
+-----+-----+-----+-----+-----+-----+-----+
| Field | Type          | Collation | Null | Key | Default | Extra           |
| Privileges          | Comment   |             |       |     |         |                 |
+-----+-----+-----+-----+-----+-----+-----+
| bookid | int unsigned | NO        | PRI  |      | auto_increment |
| select,insert,update,references |           |       |     |         |                 |
| card    | int unsigned | NO        | MUL  |      |                 |
| select,insert,update,references |           |       |     |         |                 |
| test    | varchar(255) | utf8_bin  | YES   |      |                 |
| select,insert,update,references |           |       |     |         |                 |
+-----+-----+-----+-----+-----+-----+-----+

```

