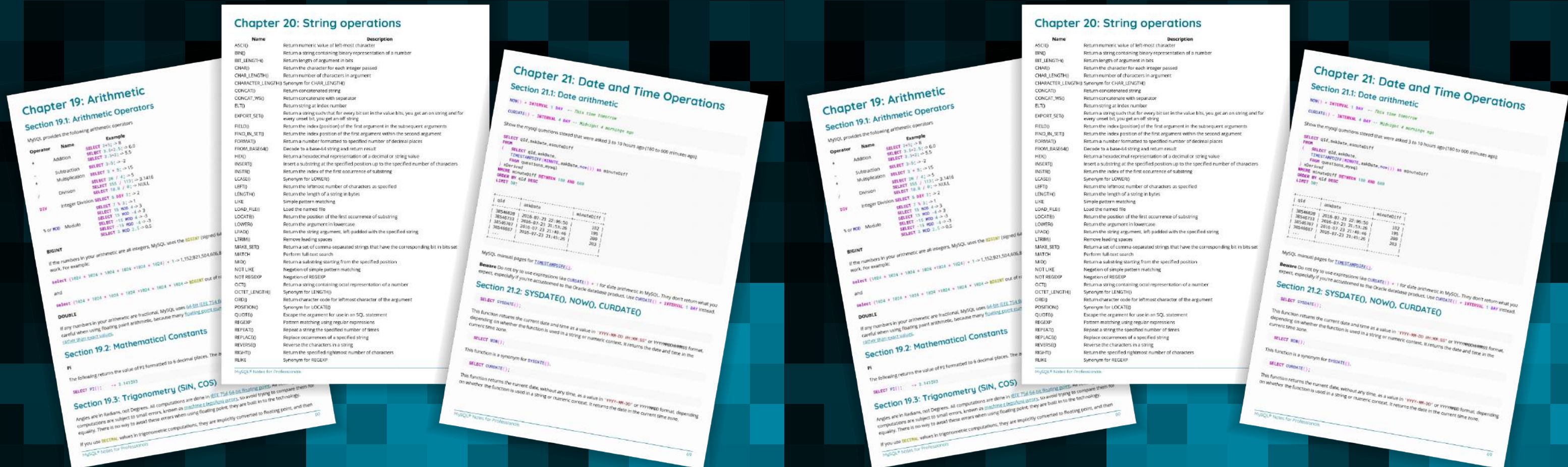


# MySQL<sup>®</sup>

## 专业人员笔记

# MySQL<sup>®</sup>

## Notes for Professionals



**100多页**  
专业提示和技巧

**100+ pages**  
of professional hints and tricks

# 目录

关于	1
第1章：MySQL入门	2
第1.1节：入门	2
第1.2节：信息模式示例	6
第2章：数据类型	7
第2.1节：CHAR(n)	7
第2.2节：DATE、DATETIME、TIMESTAMP、YEAR 和 TIME	7
第2.3节：VARCHAR(255) —— 还是不使用	8
第2.4节：INT作为AUTO_INCREMENT	8
第2.5节：其他	8
第2.6节：隐式/自动类型转换	9
第2.7节：介绍（数值型）	9
第2.8节：整数类型	10
第2.9节：定点类型	10
第2.10节：浮点类型	10
第2.11节：位值类型	11
第3章：选择（SELECT）	12
第3.1节：使用DISTINCT的SELECT	12
第3.2节：选择所有列（*）的SELECT	12
第3.3节：按列名选择的SELECT	13
第3.4节：使用LIKE（%）的SELECT	13
第3.5节：使用CASE或IF的SELECT	15
第3.6节：带别名（AS）的SELECT	15
第3.7节：带LIMIT子句的SELECT	16
第3.8节：带BETWEEN的SELECT	16
第3.9节：带WHERE的SELECT	18
第3.10节：带LIKE（_）的SELECT	18
第3.11节：带日期范围的SELECT	19
第4章：反引号	20
第4.1节：反引号的用法	20
第5章：NULL	21
第5.1节：NULL的用途	21
第5.2节：测试NULL值	21
第6章：极限与偏移	22
第6.1节：极限与偏移的关系	22
第7章：创建数据库	24
第7.1节：创建数据库、用户及权限	24
第7.2节：创建和选择数据库	26
第7.3节：我的数据库	26
第7.4节：系统数据库	27
第8章：使用变量	28
第8.1节：设置变量	28
第8.2节：在Select语句中使用变量的行号和分组	29
第9章：MySQL注释	31
第9.1节：添加注释	31
第9.2节：注释表定义	31

# Contents

About	1
Chapter 1: Getting started with MySQL	2
Section 1.1: Getting Started	2
Section 1.2: Information Schema Examples	6
Chapter 2: Data Types	7
Section 2.1: CHAR(n)	7
Section 2.2: DATE, DATETIME, TIMESTAMP, YEAR, and TIME	7
Section 2.3: VARCHAR(255) -- or not	8
Section 2.4: INT as AUTO_INCREMENT	8
Section 2.5: Others	8
Section 2.6: Implicit / automatic casting	9
Section 2.7: Introduction (numeric)	9
Section 2.8: Integer Types	10
Section 2.9: Fixed Point Types	10
Section 2.10: Floating Point Types	10
Section 2.11: Bit Value Type	11
Chapter 3: SELECT	12
Section 3.1: SELECT with DISTINCT	12
Section 3.2: SELECT all columns (*)	12
Section 3.3: SELECT by column name	13
Section 3.4: SELECT with LIKE (%)	13
Section 3.5: SELECT with CASE or IF	15
Section 3.6: SELECT with Alias (AS)	15
Section 3.7: SELECT with a LIMIT clause	16
Section 3.8: SELECT with BETWEEN	16
Section 3.9: SELECT with WHERE	18
Section 3.10: SELECT with LIKE(_)	18
Section 3.11: SELECT with date range	19
Chapter 4: Backticks	20
Section 4.1: Backticks usage	20
Chapter 5: NULL	21
Section 5.1: Uses for NULL	21
Section 5.2: Testing NULLs	21
Chapter 6: Limit and Offset	22
Section 6.1: Limit and Offset relationship	22
Chapter 7: Creating databases	24
Section 7.1: Create database, users, and grants	24
Section 7.2: Creating and Selecting a Database	26
Section 7.3: MyDatabase	26
Section 7.4: System Databases	27
Chapter 8: Using Variables	28
Section 8.1: Setting Variables	28
Section 8.2: Row Number and Group By using variables in Select Statement	29
Chapter 9: Comment MySQL	31
Section 9.1: Adding comments	31
Section 9.2: Commenting table definitions	31

第10章：INSERT	32
第10.1节：INSERT, ON DUPLICATE KEY UPDATE	32
第10.2节：插入多行	32
第10.3节：基本插入	33
第10.4节：带有AUTO INCREMENT + LAST_INSERT_ID()的INSERT	33
第10.5节：INSERT SELECT（从另一张表插入数据）	35
第10.6节：丢失的AUTO_INCREMENT ID	35
第11章：DELETE（删除）	37
第11.1节：多表删除	37
第11.2节：DELETE与TRUNCATE的比较	39
第11.3节：多表删除	39
第11.4节：基本删除	39
第11.5节：带Where子句的删除	39
第11.6节：删除表中所有行	39
第11.7节：限制删除数量	40
第12章：更新（UPDATE）	41
第12.1节：使用连接模式的更新	41
第12.2节：基本更新	41
第12.3节：批量更新	42
第12.4节：带ORDER BY和LIMIT的更新	42
第12.5节：多表更新	42
第13章：ORDER BY	44
第13.1节：上下文	44
第13.2节：基础	44
第13.3节：升序 / 降序	44
第13.4节：一些技巧	44
第14章：分组	46
第14.1节：使用 HAVING 的 GROUP BY	46
第14.2节：使用 Group Concat 的分组	46
第14.3节：使用 MIN 函数的分组	46
第14.4节：带聚合函数的 GROUP BY	47
第15章：错误1055：ONLY_FULL_GROUP_BY：某些内容未包含在GROUP BY子句中	
...	50
第15.1节：误用GROUP BY导致不可预测的结果：墨菲定律	50
第15.2节：误用GROUP BY与SELECT *, 以及如何修复	50
第15.3节：ANY_VALUE()	51
第15.4节：使用和误用GROUP BY	51
第16章：连接	53
第16.1节：连接的可视化	53
第16.2节：带子查询（“派生”表）的JOIN	53
第16.3节：全外连接	54
第16.4节：检索有订单的客户——主题的变体	55
第16.5节：连接示例	56
第17章：连接：连接3个具有相同id名称的表	57
第17.1节：在具有相同名称的列上连接3个表	57
第18章：联合（UNION）	58
第18.1节：使用UNION合并SELECT语句	58
第18.2节：合并具有不同列的数据	58
第18.3节：ORDER BY排序	58

Chapter 10: INSERT	32
Section 10.1: INSERT, ON DUPLICATE KEY UPDATE	32
Section 10.2: Inserting multiple rows	32
Section 10.3: Basic Insert	33
Section 10.4: INSERT with AUTO INCREMENT + LAST_INSERT_ID()	33
Section 10.5: INSERT SELECT (Inserting data from another Table)	35
Section 10.6: Lost AUTO_INCREMENT ids	35
Chapter 11: DELETE	37
Section 11.1: Multi-Table Deletes	37
Section 11.2: DELETE vs TRUNCATE	39
Section 11.3: Multi-table DELETE	39
Section 11.4: Basic delete	39
Section 11.5: Delete with Where clause	39
Section 11.6: Delete all rows from a table	39
Section 11.7: LIMITing deletes	40
Chapter 12: UPDATE	41
Section 12.1: Update with Join Pattern	41
Section 12.2: Basic Update	41
Section 12.3: Bulk UPDATE	42
Section 12.4: UPDATE with ORDER BY and LIMIT	42
Section 12.5: Multiple Table UPDATE	42
Chapter 13: ORDER BY	44
Section 13.1: Contexts	44
Section 13.2: Basic	44
Section 13.3: ASCending / DESCending	44
Section 13.4: Some tricks	44
Chapter 14: Group By	46
Section 14.1: GROUP BY using HAVING	46
Section 14.2: Group By using Group Concat	46
Section 14.3: Group By Using MIN function	46
Section 14.4: GROUP BY with AGGREGATE functions	47
Chapter 15: Error 1055: ONLY_FULL_GROUP_BY: something is not in GROUP BY clause	
...	50
Section 15.1: Misusing GROUP BY to return unpredictable results: Murphy's Law	50
Section 15.2: Misusing GROUP BY with SELECT *, and how to fix it	50
Section 15.3: ANY_VALUE()	51
Section 15.4: Using and misusing GROUP BY	51
Chapter 16: Joins	53
Section 16.1: Joins visualized	53
Section 16.2: JOIN with subquery (“Derived” table)	53
Section 16.3: Full Outer Join	54
Section 16.4: Retrieve customers with orders -- variations on a theme	55
Section 16.5: Joining Examples	56
Chapter 17: JOINS: Join 3 table with the same name of id.	57
Section 17.1: Join 3 tables on a column with the same name	57
Chapter 18: UNION	58
Section 18.1: Combining SELECT statements with UNION	58
Section 18.2: Combining data with different columns	58
Section 18.3: ORDER BY	58



第18.4节：通过OFFSET实现分页	58
第18.5节：合并和整合具有相同列的不同MySQL表中的数据为唯一行并执行查询	
行并执行查询	59
第18.6节：UNION ALL 和 UNION	59
第19章：算术	60
第19.1节：算术运算符	60
第19.2节：数学常数	60
第19.3节：三角函数（正弦、余弦）	60
第19.4节：取整（四舍五入、向下取整、向上取整）	62
第19.5节：幂运算（POW）	62
第19.6节：平方根（SQRT）	63
第19.7节：随机数（RAND）	63
第19.8节：绝对值和符号（ABS，SIGN）	63
第20章：字符串操作	65
第20.1节：LENGTH()	66
第20.2节：CHAR LENGTH()	66
第20.3节：HEX(str)	66
第20.4节：SUBSTRING()	66
第20.5节：UPPER() / UCASE()	67
第20.6节：STR TO DATE - 字符串转换为日期	67
第20.7节：LOWER() / LCASE()	67
第20.8节：REPLACE()	67
第20.9节：在逗号分隔列表中查找元素	67
第21章：日期和时间操作	69
第21.1节：日期运算	69
第21.2节：SYSDATE(), NOW(), CURDATE()	69
第21.3节：针对日期范围的测试	70
第21.4节：从给定的日期或日期时间表达式中提取日期	70
第21.5节：使用索引进行日期和时间查找	70
第21.6节：Now()	71
第22章：处理时区	72
第22.1节：检索特定时区的当前日期和时间	72
第22.2节：将存储的`DATE`或`DATETIME`值转换为另一个时区	72
第22.3节：在特定时区检索存储的`TIMESTAMP`值	72
第22.4节：我的服务器本地时区设置是什么？	72
第22.5节：我的服务器中有哪些time_zone值可用？	73
第23章：正则表达式	74
第23.1节：REGEXP / RLIKE	74
第24章：视图	76
第24.1节：创建视图	76
第24.2节：来自两个表的视图	77
第24.3节：删除视图	77
第24.4节：通过视图更新表	77
第25章：表的创建	78
第25.1节：带主键的表创建	78
第25.2节：基本表创建	79
第25.3节：带外键的表创建	79
第25.4节：显示表结构	80
第25.5节：克隆现有表	81

Section 18.4: Pagination via OFFSET	58
Section 18.5: Combining and merging data on different MySQL tables with the same columns into unique rows and running query	59
Section 18.6: UNION ALL and UNION	59
Chapter 19: Arithmetic	60
Section 19.1: Arithmetic Operators	60
Section 19.2: Mathematical Constants	60
Section 19.3: Trigonometry (SIN, COS)	60
Section 19.4: Rounding (ROUND, FLOOR, CEIL)	62
Section 19.5: Raise a number to a power (POW)	62
Section 19.6: Square Root (SQRT)	63
Section 19.7: Random Numbers (RAND)	63
Section 19.8: Absolute Value and Sign (ABS, SIGN)	63
Chapter 20: String operations	65
Section 20.1: LENGTH()	66
Section 20.2: CHAR LENGTH()	66
Section 20.3: HEX(str)	66
Section 20.4: SUBSTRING()	66
Section 20.5: UPPER() / UCASE()	67
Section 20.6: STR TO DATE - Convert string to date	67
Section 20.7: LOWER() / LCASE()	67
Section 20.8: REPLACE()	67
Section 20.9: Find element in comma separated list	67
Chapter 21: Date and Time Operations	69
Section 21.1: Date arithmetic	69
Section 21.2: SYSDATE(), NOW(), CURDATE()	69
Section 21.3: Testing against a date range	70
Section 21.4: Extract Date from Given Date or DateTime Expression	70
Section 21.5: Using an index for a date and time lookup	70
Section 21.6: Now()	71
Chapter 22: Handling Time Zones	72
Section 22.1: Retrieve the current date and time in a particular time zone	72
Section 22.2: Convert a stored `DATE` or `DATETIME` value to another time zone	72
Section 22.3: Retrieve stored `TIMESTAMP` values in a particular time zone	72
Section 22.4: What is my server's local time zone setting?	72
Section 22.5: What time_zone values are available in my server?	73
Chapter 23: Regular Expressions	74
Section 23.1: REGEXP / RLIKE	74
Chapter 24: VIEW	76
Section 24.1: Create a View	76
Section 24.2: A view from two tables	77
Section 24.3: DROPPING A VIEW	77
Section 24.4: Updating a table via a VIEW	77
Chapter 25: Table Creation	78
Section 25.1: Table creation with Primary Key	78
Section 25.2: Basic table creation	79
Section 25.3: Table creation with Foreign Key	79
Section 25.4: Show Table Structure	80
Section 25.5: Cloning an existing table	81

第25.6节：创建带时间戳列的表以显示最后更新	81
第25.7节：从SELECT创建表	81
第26章：修改表（ALTER TABLE）	83
第26.1节：更改存储引擎；重建表；更改file_per_table	83
第26.2节：修改表的列	83
第26.3节：更改自增值	83
第26.4节：重命名MySQL表	83
第26.5节：ALTER 表添加索引	84
第26.6节：更改主键列的类型	84
第26.7节：更改列定义	84
第26.8节：重命名MySQL数据库	84
第26.9节：交换两个MySQL数据库的名称	85
第26.10节：重命名MySQL表中的列	85
第27章：删除表	87
第27.1节：删除表	87
第27.2节：从数据库删除表	87
第28章：MySQL锁表	88
第28.1节：行级锁定	88
第28.2节：MySQL锁	89
第29章：错误代码	91
第29.1节：错误代码1064：语法错误	91
第29.2节：错误代码1175：安全更新	91
第29.3节：错误代码1215：无法添加外键约束	91
第29.4节：1067、1292、1366、1411 - 数字、日期、默认值等的无效值	93
第29.5节：错误代码1045 拒绝访问	93
第29.6节：错误代码1236 复制中的“不可能位置”	93
第29.7节：2002，2003 无法连接	94
第29.8节：126, 127, 134, 144, 145	94
第29.9节：139	94
第29.10节：1366	94
第29.11节：126, 1054, 1146, 1062, 24	95
第30章：存储过程（过程和函数）	97
第30.1节：带有IN、OUT、INOUT参数的存储过程	97
第30.2节：创建函数	98
第30.3节：游标	99
第30.4节：多个结果集	100
第30.5节：创建函数	100
第31章：索引和键	102
第31.1节：创建索引	102
第31.2节：创建唯一索引	102
第31.3节：AUTO INCREMENT键	102
第31.4节：创建复合索引	102
第31.5节：删除索引	103
第32章：全文搜索	104
第32.1节：简单全文搜索	104
第32.2节：简单布尔搜索	104
第32.3节：多列全文搜索	104
第33章：PREPARE语句	106
第33.1节：PREPARE、EXECUTE和DEALLOCATE PREPARE语句	106

Section 25.6: Table Create With TimeStamp Column To Show Last Update	81
Section 25.7: CREATE TABLE FROM SELECT	81
Chapter 26: ALTER TABLE	83
Section 26.1: Changing storage engine; rebuild table; change file_per_table	83
Section 26.2: ALTER COLUMN OF TABLE	83
Section 26.3: Change auto-increment value	83
Section 26.4: Renaming a MySQL table	83
Section 26.5: ALTER table add INDEX	84
Section 26.6: Changing the type of a primary key column	84
Section 26.7: Change column definition	84
Section 26.8: Renaming a MySQL database	84
Section 26.9: Swapping the names of two MySQL databases	85
Section 26.10: Renaming a column in a MySQL table	85
Chapter 27: Drop Table	87
Section 27.1: Drop Table	87
Section 27.2: Drop tables from database	87
Chapter 28: MySQL LOCK TABLE	88
Section 28.1: Row Level Locking	88
Section 28.2: Mysql Locks	89
Chapter 29: Error codes	91
Section 29.1: Error code 1064: Syntax error	91
Section 29.2: Error code 1175: Safe Update	91
Section 29.3: Error code 1215: Cannot add foreign key constraint	91
Section 29.4: 1067, 1292, 1366, 1411 - Bad Value for number, date, default, etc	93
Section 29.5: 1045 Access denied	93
Section 29.6: 1236 “impossible position” in Replication	93
Section 29.7: 2002, 2003 Cannot connect	94
Section 29.8: 126, 127, 134, 144, 145	94
Section 29.9: 139	94
Section 29.10: 1366	94
Section 29.11: 126, 1054, 1146, 1062, 24	95
Chapter 30: Stored routines (procedures and functions)	97
Section 30.1: Stored procedure with IN, OUT, INOUT parameters	97
Section 30.2: Create a Function	98
Section 30.3: Cursors	99
Section 30.4: Multiple ResultSets	100
Section 30.5: Create a function	100
Chapter 31: Indexes and Keys	102
Section 31.1: Create index	102
Section 31.2: Create unique index	102
Section 31.3: AUTO INCREMENT key	102
Section 31.4: Create composite index	102
Section 31.5: Drop index	103
Chapter 32: Full-Text search	104
Section 32.1: Simple FULLTEXT search	104
Section 32.2: Simple BOOLEAN search	104
Section 32.3: Multi-column FULLTEXT search	104
Chapter 33: PREPARE Statements	106
Section 33.1: PREPARE, EXECUTE and DEALLOCATE PREPARE Statements	106

第33.2节：使用添加列的ALTER TABLE	106
<b>第34章：JSON</b>	107
第34.1节：创建带主键和JSON字段的简单表	107
第34.2节：插入简单的JSON	107
第34.3节：更新JSON字段	107
第34.4节：向JSON字段插入混合数据	108
第34.5节：将数据转换为JSON类型	108
第34.6节：创建JSON对象和数组	108
<b>第35章：从JSON类型中提取值</b>	109
第35.1节：读取JSON数组值	109
第35.2节：JSON提取操作符	109
<b>第36章：MySQL 管理</b>	111
第36.1节：原子重命名与表重载	111
第36.2节：更改root密码	111
第36.3节：删除数据库	111
<b>第37章：触发器</b>	112
第37.1节：基本触发器	112
第37.2节：触发器类型	112
<b>第38章：配置与调优</b>	114
第38.1节：InnoDB性能	114
第38.2节：允许插入大量数据的参数	114
第38.3节：增加group_concat的字符串限制	114
第38.4节：最小化InnoDB配置	114
第38.5节：安全的MySQL加密	115
<b>第39章：事件</b>	116
第39.1节：创建事件	116
<b>第40章：ENUM</b>	119
第40.1节：为什么选择ENUM？	119
第40.2节：VARCHAR作为替代方案	119
第40.3节：添加新选项	119
第40.4节：NULL与NOT NULL	119
<b>第41章：使用Docker-Compose安装Mysql容器</b>	121
第41.1节：docker-compose的简单示例	121
<b>第42章：字符集与排序规则</b>	122
第42.1节：使用哪个字符集和排序规则？	122
第42.2节：在表和字段上设置字符集	122
第42.3节：声明	122
第42.4节：连接	123
<b>第43章：MyISAM引擎</b>	124
第43.1节：引擎=MyISAM	124
<b>第44章：从MyISAM转换到InnoDB</b>	125
第44.1节：基本转换	125
第44.2节：转换一个数据库中的所有表	125
<b>第45章：事务</b>	126
第45.1节：启动事务	126
第45.2节：提交（COMMIT）、回滚（ROLLBACK）和自动提交（AUTOCOMMIT）	127
第45.3节：使用JDBC驱动的事务	129
<b>第46章：日志文件</b>	132

Section 33.2: Alter table with add column	106
<b>Chapter 34: JSON</b>	107
Section 34.1: Create simple table with a primary key and JSON field	107
Section 34.2: Insert a simple JSON	107
Section 34.3: Updating a JSON field	107
Section 34.4: Insert mixed data into a JSON field	108
Section 34.5: CAST data to JSON type	108
Section 34.6: Create Json Object and Array	108
<b>Chapter 35: Extract values from JSON type</b>	109
Section 35.1: Read JSON Array value	109
Section 35.2: JSON Extract Operators	109
<b>Chapter 36: MySQL Admin</b>	111
Section 36.1: Atomic RENAME & Table Reload	111
Section 36.2: Change root password	111
Section 36.3: Drop database	111
<b>Chapter 37: TRIGGERS</b>	112
Section 37.1: Basic Trigger	112
Section 37.2: Types of triggers	112
<b>Chapter 38: Configuration and tuning</b>	114
Section 38.1: InnoDB performance	114
Section 38.2: Parameter to allow huge data to insert	114
Section 38.3: Increase the string limit for group_concat	114
Section 38.4: Minimal InnoDB configuration	114
Section 38.5: Secure MySQL encryption	115
<b>Chapter 39: Events</b>	116
Section 39.1: Create an Event	116
<b>Chapter 40: ENUM</b>	119
Section 40.1: Why ENUM?	119
Section 40.2: VARCHAR as an alternative	119
Section 40.3: Adding a new option	119
Section 40.4: NULL vs NOT NULL	119
<b>Chapter 41: Install Mysql container with Docker-Compose</b>	121
Section 41.1: Simple example with docker-compose	121
<b>Chapter 42: Character Sets and Collations</b>	122
Section 42.1: Which CHARACTER SET and COLLATION?	122
Section 42.2: Setting character sets on tables and fields	122
Section 42.3: Declaration	122
Section 42.4: Connection	123
<b>Chapter 43: MyISAM Engine</b>	124
Section 43.1: ENGINE=MyISAM	124
<b>Chapter 44: Converting from MyISAM to InnoDB</b>	125
Section 44.1: Basic conversion	125
Section 44.2: Converting All Tables in one Database	125
<b>Chapter 45: Transaction</b>	126
Section 45.1: Start Transaction	126
Section 45.2: COMMIT , ROLLBACK and AUTOCOMMIT	127
Section 45.3: Transaction using JDBC Driver	129
<b>Chapter 46: Log files</b>	132



第46.1节：慢查询日志	132	Section 46.1: Slow Query Log	132
第46.2节：列表	132	Section 46.2: A List	132
第46.3节：通用查询日志	133	Section 46.3: General Query Log	133
第46.4节：错误日志	134	Section 46.4: Error Log	134
第47章：聚类	136	Chapter 47: Clustering	136
第47.1节：消歧义	136	Section 47.1: Disambiguation	136
第48章：分区	137	Chapter 48: Partitioning	137
第48.1节：范围分区	137	Section 48.1: RANGE Partitioning	137
第48.2节：列表分区	137	Section 48.2: LIST Partitioning	137
第48.3节：哈希分区	138	Section 48.3: HASH Partitioning	138
第49章：复制	139	Chapter 49: Replication	139
第49.1节：主从复制设置	139	Section 49.1: Master - Slave Replication Setup	139
第49.2节：复制错误	141	Section 49.2: Replication Errors	141
第50章：使用mysqldump备份	143	Chapter 50: Backup using mysqldump	143
第50.1节：指定用户名和密码	143	Section 50.1: Specifying username and password	143
第50.2节：创建数据库或表的备份	143	Section 50.2: Creating a backup of a database or table	143
第50.3节：恢复数据库或表的备份	144	Section 50.3: Restoring a backup of a database or table	144
第50.4节：将数据从一个MySQL服务器传输到另一个服务器	144	Section 50.4: Tranferring data from one MySQL server to another	144
第50.5节：带压缩的远程服务器mysqldump	145	Section 50.5: mysqldump from a remote server with compression	145
第50.6节：恢复gzip压缩的mysqldump文件而不解压	145	Section 50.6: restore a gzipped mysqldump file without uncompressing	145
第50.7节：备份包含存储过程和函数的数据库	145	Section 50.7: Backup database with stored procedures and functions	145
第50.8节：压缩后直接备份到Amazon S3	145	Section 50.8: Backup direct to Amazon S3 with compression	145
第51章：mysqlimport	146	Chapter 51: mysqlimport	146
第51.1节：基本用法	146	Section 51.1: Basic usage	146
第51.2节：使用自定义字段分隔符	146	Section 51.2: Using a custom field-delimiter	146
第51.3节：使用自定义行分隔符	146	Section 51.3: Using a custom row-delimiter	146
第51.4节：处理重复键	146	Section 51.4: Handling duplicate keys	146
第51.5节：条件导入	147	Section 51.5: Conditional import	147
第51.6节：导入标准CSV	147	Section 51.6: Import a standard csv	147
第52章：LOAD DATA INFILE	148	Chapter 52: LOAD DATA INFILE	148
第52.1节：使用LOAD DATA INFILE将大量数据加载到数据库	148	Section 52.1: using LOAD DATA INFILE to load large amount of data to database	148
第52.2节：加载包含重复数据	149	Section 52.2: Load data with duplicates	149
第52.3节：将CSV文件导入MySQL表	149	Section 52.3: Import a CSV file into a MySQL table	149
第53章：MySQL联合查询	150	Chapter 53: MySQL Unions	150
第53.1节：联合操作符	150	Section 53.1: Union operator	150
第53.2节：UNION ALL	150	Section 53.2: Union ALL	150
第53.3节：带WHERE的UNION ALL	151	Section 53.3: UNION ALL With WHERE	151
第54章：MySQL客户端	152	Chapter 54: MySQL client	152
第54.1节：基础登录	152	Section 54.1: Base login	152
第54.2节：执行命令	152	Section 54.2: Execute commands	152
第55章：临时表	154	Chapter 55: Temporary Tables	154
第55.1节：创建临时表	154	Section 55.1: Create Temporary Table	154
第55.2节：删除临时表	154	Section 55.2: Drop Temporary Table	154
第56章：自定义PS1	155	Chapter 56: Customize PS1	155
第56.1节：使用当前数据库自定义MySQL PS1	155	Section 56.1: Customize the MySQL PS1 with current database	155
第56.2节：通过MySQL配置文件自定义PS1	155	Section 56.2: Custom PS1 via MySQL configuration file	155
第57章：处理稀疏或缺失数据	156	Chapter 57: Dealing with sparse or missing data	156
第57.1节：处理包含NULL值的列	156	Section 57.1: Working with columns containg NULL values	156

<b>第58章：使用各种编程语言连接UTF-8</b>	159
第58.1节：Python	159
第58.2节：PHP	159
<b>第59章：具有亚秒精度的时间</b>	160
第59.1节：获取具有毫秒精度的当前时间	160
第59.2节：以类似Javascript时间戳的形式获取当前时间	160
第59.3节：创建带有存储亚秒时间列的表	160
第59.4节：将具有毫秒精度的日期/时间值转换为文本	160
第59.5节：将Javascript时间戳存储到TIMESTAMP列中	161
<b>第60章：一对多</b>	162
第60.1节：示例公司表	162
第60.2节：获取单个经理管理的员工	162
第60.3节：获取单个员工的经理	162
<b>第61章：服务器信息</b>	164
第61.1节：SHOW VARIABLES 示例	164
第61.2节：SHOW STATUS 示例	164
<b>第62章：SSL连接设置</b>	166
第62.1节：基于Debian系统的设置	166
第62.2节：CentOS7 / RHEL7的设置	168
<b>第63章：创建新用户</b>	173
第63.1节：创建MySQL用户	173
第63.2节：指定密码	173
第63.3节：创建新用户并授予模式的所有权限	173
第63.4节：重命名用户	173
<b>第64章：通过GRANT实现安全</b>	174
第64.1节：最佳实践	174
第64.2节：主机（user@host中的主机）	174
<b>第65章：更改密码</b>	175
第65.1节：在Linux中更改MySQL root密码	175
第65.2节：在Windows中更改MySQL root密码	175
第65.3节：流程	176
<b>第66章：恢复并重置MySQL 5.7及以上版本的默认root密码</b>	177
第66.1节：服务器首次启动时会发生什么	177
第66.2节：如何使用默认密码更改root密码	177
第66.3节：当“/var/run/mysqld”UNIX套接字文件不存在时重置root密码	177
<b>第67章：恢复丢失的root密码</b>	180
第67.1节：设置root密码，启用root用户的socket和http访问	180
<b>第68章：MySQL性能技巧</b>	181
第68.1节：构建复合索引	181
第68.2节：优化InnoDB表的存储布局	181
<b>第69章：性能调优</b>	183
第69.1节：不要隐藏在函数中	183
第69.2节：OR	183
第69.3节：添加正确的索引	183
第69.4节：拥有索引	184
第69.5节：子查询	184
第69.6节：JOIN + GROUP BY	184
第69.7节：正确设置缓存	185

<b>Chapter 58: Connecting with UTF-8 Using Various Programming language.</b>	159
Section 58.1: Python	159
Section 58.2: PHP	159
<b>Chapter 59: Time with subsecond precision</b>	160
Section 59.1: Get the current time with millisecond precision	160
Section 59.2: Get the current time in a form that looks like a Javascript timestamp	160
Section 59.3: Create a table with columns to store sub-second time	160
Section 59.4: Convert a millisecond-precision date / time value to text	160
Section 59.5: Store a Javascript timestamp into a TIMESTAMP column	161
<b>Chapter 60: One to Many</b>	162
Section 60.1: Example Company Tables	162
Section 60.2: Get the Employees Managed by a Single Manager	162
Section 60.3: Get the Manager for a Single Employee	162
<b>Chapter 61: Server Information</b>	164
Section 61.1: SHOW VARIABLES example	164
Section 61.2: SHOW STATUS example	164
<b>Chapter 62: SSL Connection Setup</b>	166
Section 62.1: Setup for Debian-based systems	166
Section 62.2: Setup for CentOS7 / RHEL7	168
<b>Chapter 63: Create New User</b>	173
Section 63.1: Create a MySQL User	173
Section 63.2: Specify the password	173
Section 63.3: Create new user and grant all privileges to schema	173
Section 63.4: Renaming user	173
<b>Chapter 64: Security via GRANTS</b>	174
Section 64.1: Best Practice	174
Section 64.2: Host (of user@host)	174
<b>Chapter 65: Change Password</b>	175
Section 65.1: Change MySQL root password in Linux	175
Section 65.2: Change MySQL root password in Windows	175
Section 65.3: Process	176
<b>Chapter 66: Recover and reset the default root password for MySQL 5.7+</b>	177
Section 66.1: What happens when the initial start up of the server	177
Section 66.2: How to change the root password by using the default password	177
Section 66.3: reset root password when “ /var/run/mysqld’ for UNIX socket file don’t exists”	177
<b>Chapter 67: Recover from lost root password</b>	180
Section 67.1: Set root password, enable root user for socket and http access	180
<b>Chapter 68: MySQL Performance Tips</b>	181
Section 68.1: Building a composite index	181
Section 68.2: Optimizing Storage Layout for InnoDB Tables	181
<b>Chapter 69: Performance Tuning</b>	183
Section 69.1: Don’t hide in function	183
Section 69.2: OR	183
Section 69.3: Add the correct index	183
Section 69.4: Have an INDEX	184
Section 69.5: Subqueries	184
Section 69.6: JOIN + GROUP BY	184
Section 69.7: Set the cache correctly	185



第69.8节：负数	185
附录A：保留字	186
第A.1节：保留字引起的错误	186
学分	187
你可能也喜欢	190

Section 69.8: Negatives	185
Appendix A: Reserved Words	186
Section A.1: Errors due to reserved words	186
Credits	187
You may also like	190

欢迎免费与任何人分享此PDF，  
本书最新版本可从以下网址下载：  
<https://goalkicker.com/MySQLBook>

本MySQL® *Notes for Professionals* 书籍汇编自[Stack Overflow Documentation](#)，内容由Stack Overflow的优秀人士撰写。  
文本内容采用知识共享署名-相同方式共享许可协议发布，详见本书末尾对各章节贡献者的致谢。图片版权归各自所有者所有，除非另有说明。

本书为非官方免费书籍，旨在教育用途，与官方MySQL®组织或公司及Stack Overflow无关。所有商标和注册商标均为其各自公司所有者所有。

本书所提供的信息不保证正确或准确，使用风险自负。

请将反馈和更正发送至[web@petercv.com](mailto:web@petercv.com)

Please feel free to share this PDF with anyone for free,  
latest version of this book can be downloaded from:  
<https://goalkicker.com/MySQLBook>

This *MySQL® Notes for Professionals* book is compiled from [Stack Overflow Documentation](#), the content is written by the beautiful people at Stack Overflow. Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official MySQL® group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to [web@petercv.com](mailto:web@petercv.com)

# 第1章：MySQL入门

版本发布日期	
1.0	1995-05-23
3.19	1996-12-01
3.20	1997-01-01
3.21	1998-10-01
3.22	1999-10-01
3.23	2001-01-22
4.0	2003-03-01
4.1	2004-10-01
5.0	2005-10-01
5.1	2008-11-27
5.5	2010-11-01
5.6	2013-02-01
5.7	2015-10-01

## 第1.1节：入门

### 在MySQL中创建数据库

```
CREATE DATABASE mydb;
```

返回值：

查询成功，影响行数：1（0.05秒）

### 使用已创建的数据库mydb

```
USE mydb;
```

返回值：

数据库已更改

### 在MySQL中创建表

```
CREATE TABLE mytable
(
  id          int unsigned NOT NULL auto_increment,
  username    varchar(100) NOT NULL,
  email       varchar(100) NOT NULL,
  PRIMARY KEY (id)
);
```

CREATE TABLE mytable 将创建一个名为mytable的新表。

id int unsigned NOT NULL auto\_increment 创建了 id 列，这种类型的字段会为表中的每条记录分配一个唯一的数字 ID（意味着在这种情况下没有两行可以有相同的 id），MySQL 会

# Chapter 1: Getting started with MySQL

Version Release Date	
1.0	1995-05-23
3.19	1996-12-01
3.20	1997-01-01
3.21	1998-10-01
3.22	1999-10-01
3.23	2001-01-22
4.0	2003-03-01
4.1	2004-10-01
5.0	2005-10-01
5.1	2008-11-27
5.5	2010-11-01
5.6	2013-02-01
5.7	2015-10-01

## Section 1.1: Getting Started

### Creating a database in MySQL

```
CREATE DATABASE mydb;
```

Return value:

Query OK, 1 row affected (0.05 sec)

### Using the created database mydb

```
USE mydb;
```

Return value:

Database Changed

### Creating a table in MySQL

```
CREATE TABLE mytable
(
  id          int unsigned NOT NULL auto_increment,
  username    varchar(100) NOT NULL,
  email       varchar(100) NOT NULL,
  PRIMARY KEY (id)
);
```

CREATE TABLE mytable will create a new table called mytable.

id int unsigned NOT NULL auto\_increment creates the id column, this type of field will assign a unique numeric ID to each record in the table (meaning that no two rows can have the same id in this case), MySQL will



自动为记录的 id 字段分配一个新的唯一值（从1开始）。

返回值：

查询成功，0 行受影响（0.10 秒）

向 MySQL 表中插入一行

```
INSERT INTO mytable ( username, email )
VALUES ( "myuser", "myuser@example.com" );
```

示例返回值：

查询成功，影响行数：1 行（0.06 秒）

varchar（即字符串）也可以使用单引号插入：

```
INSERT INTO mytable ( username, email )
VALUES ( 'username', 'username@example.com' );
```

更新 MySQL 表中的一行

```
UPDATE mytable SET username="myuser" WHERE id=8
```

示例返回值：

查询成功，影响行数：1 行（0.06 秒）

int 类型的值可以在查询中不加引号插入。字符串和日期必须用单引号'或双引号"括起来。

删除 MySQL 表中的一行

```
DELETE FROM mytable WHERE id=8
```

示例返回值：

查询成功，影响行数：1 行（0.06 秒）

这将删除 id 为 8 的那一行。

在 MySQL 中根据条件选择行

```
SELECT * FROM mytable WHERE username = "myuser";
```

返回值：

+	-----+	-----+	-----+
id	用户名	邮箱	

automatically assign a new, unique value to the record's id field (starting with 1).

Return value:

Query OK, 0 rows affected (0.10 sec)

Inserting a row into a MySQL table

```
INSERT INTO mytable ( username, email )
VALUES ( "myuser", "myuser@example.com" );
```

Example return value:

Query OK, 1 row affected (0.06 sec)

The **varchar** a.k.a strings can be also be inserted using single quotes:

```
INSERT INTO mytable ( username, email )
VALUES ( 'username', 'username@example.com' );
```

Updating a row into a MySQL table

```
UPDATE mytable SET username="myuser" WHERE id=8
```

Example return value:

Query OK, 1 row affected (0.06 sec)

The **int** value can be inserted in a query without quotes. Strings and Dates must be enclosed in single quote ' or double quotes ".

Deleting a row into a MySQL table

```
DELETE FROM mytable WHERE id=8
```

Example return value:

Query OK, 1 row affected (0.06 sec)

This will delete the row having id is 8.

Selecting rows based on conditions in MySQL

```
SELECT * FROM mytable WHERE username = "myuser";
```

Return value:

+	-----+	-----+	-----+
id	username	email	

```
+-----+
| 1 | myuser | myuser@example.com |
+-----+
```

1 行记录 (0.00 秒)

显示现有数据库列表

```
SHOW databases;
```

返回值：

```
+-----+
| 数据库          |
+-----+
| information_schema|
| mydb             |
+-----+
```

2 行记录 (0.00 秒)

你可以将“information\_schema”视为一个“主数据库”，它提供对数据库元数据的访问。

显示现有数据库中的表

```
SHOW tables;
```

返回值：

```
+-----+
| mydb 中的表 |
+-----+
| mytable      |
+-----+
```

1 行记录 (0.00 秒)

显示表的所有字段

```
DESCRIBE dbName.tableName;
```

或者，如果已经使用某个数据库：

```
DESCRIBE tableName;
```

返回值：

```
+-----+-----+-----+-----+-----+-----+-----+
```

```
+-----+-----+-----+-----+
| 1 | myuser | myuser@example.com |
+-----+-----+-----+-----+
```

1 row in set (0.00 sec)

Show list of existing databases

```
SHOW databases;
```

Return value:

```
+-----+
| Databases          |
+-----+
| information_schema|
| mydb               |
+-----+
```

2 rows in set (0.00 sec)

You can think of "information\_schema" as a "master database" that provides access to database metadata.

Show tables in an existing database

```
SHOW tables;
```

Return value:

```
+-----+
| Tables_in_mydb |
+-----+
| mytable        |
+-----+
```

1 row in set (0.00 sec)

Show all the fields of a table

```
DESCRIBE dbName.tableName;
```

or, if already using a database:

```
DESCRIBE tableName;
```

Return value:

```
+-----+-----+-----+-----+-----+-----+-----+
```

字段	类型	是否为空	键	默认值	额外
fieldname	fieldvaluetype	否/是	keytype	defaultfieldvalue	

额外 可能包含 auto\_increment 例如。

键 指可能影响字段的键类型。主键 (PRI)、唯一键 (UNI) ...

n 行记录 (0.00 秒)
----------------

其中 n 是表中字段的数量。

创建用户

首先，需要创建一个用户，然后授予该用户对某些数据库/表的权限。在创建用户时，还需要指定该用户可以从哪里连接。

```
创建用户 'user'@'localhost' 通过 'some_password' 认证;
```

将创建一个只能在数据库所在的本地机器上连接的用户。

```
CREATE USER 'user'@'%' IDENTIFIED BY 'some_password';
```

将创建一个可以从任何地方连接（除了本地机器）的用户。

示例返回值：

查询成功，0 行受影响 (0.00 秒)
----------------------

添加权限

授予用户对指定数据库所有表的常用基本权限：

```
GRANT SELECT, INSERT, UPDATE ON dbName.* TO 'userName'@'localhost';
```

授予用户对所有数据库所有表的所有权限（请注意）：

```
GRANT ALL ON *.* TO 'userName'@'localhost' WITH GRANT OPTION;
```

如上所示，\*.\* 代表所有数据库和表，dbName.\* 代表指定数据库的所有表。也可以像这样指定数据库和表：  
dbName.tableName。

如果用户不需要能够授予其他用户权限，则应省略 WITH GRANT OPTION。

权限可以是以下两者之一

```
全部
```

或者以下各项的组合，每项用逗号分隔（非详尽列表）。

```
查询
```

Field	Type	Null	Key	Default	Extra
fieldname	fieldvaluetype	NO/YES	keytype	defaultfieldvalue	

Extra may contain auto\_increment for example.

Key refers to the type of key that may affect the field. Primary (PRI), Unique (UNI) ...

n row in set (0.00 sec)
-------------------------

Where n is the number of fields in the table.

Creating user

First, you need to create a user and then give the user permissions on certain databases/tables. While creating the user, you also need to specify where this user can connect from.

```
CREATE USER 'user'@'localhost' IDENTIFIED BY 'some_password';
```

Will create a user that can only connect on the local machine where the database is hosted.

```
CREATE USER 'user'@'%' IDENTIFIED BY 'some_password';
```

Will create a user that can connect from anywhere (except the local machine).

Example return value:

Query OK, 0 rows affected (0.00 sec)
--------------------------------------

Adding privileges

Grant common, basic privileges to the user for all tables of the specified database:

```
GRANT SELECT, INSERT, UPDATE ON dbName.* TO 'userName'@'localhost';
```

Grant all privileges to the user for all tables on all databases (attention with this):

```
GRANT ALL ON *.* TO 'userName'@'localhost' WITH GRANT OPTION;
```

As demonstrated above, \*.\* targets all databases and tables, dbName.\* targets all tables of the specific database. It is also possible to specify database and table like so dbName.tableName.

WITH GRANT OPTION should be left out if the user need not be able to grant other users privileges.

Privileges can be either

```
ALL
```

or a combination of the following, each separated by a comma (non-exhaustive list).

```
SELECT
```



插入  
更新  
删除  
创建  
删除

注意

通常，应尽量避免使用包含空格的列名或表名，或使用 SQL 中的保留字。  
例如，最好避免使用类似table或first name这样的名称。

如果必须使用此类名称，请将其放在反引号``中。例如：

```
创建表 `table`
(
    `first name` VARCHAR(30)
);
```

包含反引号定界符的查询可能是：

```
SELECT `first name` FROM `table` WHERE `first name` LIKE 'a%';
```

第1.2节：信息模式示例

进程列表

这将按顺序显示所有活动和休眠查询，然后按持续时间排序。

```
SELECT * FROM information_schema.PROCESSLIST ORDER BY INFO DESC, TIME DESC;
```

这对时间段提供了更多细节，因为默认单位是秒

```
SELECT ID, USER, HOST, DB, COMMAND,
TIME as time_seconds,
ROUND(TIME / 60, 2) as time_minutes,
ROUND(TIME / 60 / 60, 2) as time_hours,
STATE, INFO
FROM information_schema.PROCESSLIST ORDER BY INFO DESC, TIME DESC;
```

存储过程搜索

轻松搜索所有存储过程中的单词和通配符。

```
SELECT * FROM information_schema.ROUTINES WHERE ROUTINE_DEFINITION LIKE '%word%';
```

INSERT  
UPDATE  
DELETE  
CREATE  
DROP

Note

Generally, you should try to avoid using column or table names containing spaces or using reserved words in SQL. For example, it's best to avoid names like **table** or **first** name.

If you must use such names, put them between back-tick `` delimiters. For example:

```
CREATE TABLE `table`
(
    `first name` VARCHAR(30)
);
```

A query containing the back-tick delimiters on this table might be:

```
SELECT `first name` FROM `table` WHERE `first name` LIKE 'a%';
```

Section 1.2: Information Schema Examples

Processlist

This will show all active & sleeping queries in that order then by how long.

```
SELECT * FROM information_schema.PROCESSLIST ORDER BY INFO DESC, TIME DESC;
```

This is a bit more detail on time-frames as it is in seconds by default

```
SELECT ID, USER, HOST, DB, COMMAND,
TIME as time_seconds,
ROUND(TIME / 60, 2) as time_minutes,
ROUND(TIME / 60 / 60, 2) as time_hours,
STATE, INFO
FROM information_schema.PROCESSLIST ORDER BY INFO DESC, TIME DESC;
```

Stored Procedure Searching

Easily search thru all Stored Procedures for words and wildcards.

```
SELECT * FROM information_schema.ROUTINES WHERE ROUTINE_DEFINITION LIKE '%word%';
```

# 第2章：数据类型

## 第2.1节：CHAR(n)

CHAR(n) 是一个长度固定为 n 字符的字符串。如果它是 字符集 utf8mb4，意味着它占用的字节数正好是 4\*n 字节，无论其中包含什么文本。

大多数 CHAR(n) 的使用场景涉及包含英文字符的字符串，因此应使用 字符集 ascii。（latin1 也同样适用。）

```
country_code CHAR(2) 字符集 ascii,
postal_code  CHAR(6) 字符集 ascii,
uuid         CHAR(39) 字符集 ascii, -- 其他地方有更多讨论
```

## 第2.2节：日期（DATE）、日期时间（DATETIME）、时间戳（TIMESTAMP）、年份（YEAR）和时间（TIME）

**DATE** 数据类型包含日期但不包含时间部分。其格式为 'YYYY-MM-DD'，范围为 '1000-01-01' 到 '9999-12-31'。

**DATETIME** 类型包含时间，格式为 'YYYY-MM-DD HH:MM:SS'。其范围从 '1000-01-01 00:00:00' 到 '9999-12-31 23:59:59'。

**TIMESTAMP** 类型是一个整数类型，包含日期和时间，有效范围从 '1970-01-01 00:00:01' UTC 到 '2038-01-19 03:14:07' UTC。

YEAR 类型表示年份，范围从1901到2155。

TIME 类型表示时间，格式为 'HH:MM:SS'，范围从 '-838:59:59' 到 '838:59:59'。

存储需求：

数据类型	MySQL 5.6.4 之前	MySQL 5.6.4 及以后版本
YEAR	1 字节	1 字节
DATE	3 字节	3 字节
TIME	3 字节	3 字节 + 小数秒存储
DATETIME	8 字节	5 字节 + 小数秒存储
TIMESTAMP	4 字节	4 字节 + 小数秒存储

小数秒（自版本5.6.4起）：

小数秒精度	所需存储空间
0	0 字节
1,2	1 字节
3,4	2 字节
5,6	3 字节

请参阅 MySQL 手册页[DATE、DATETIME 和 TIMESTAMP 类型](#)，[数据类型存储需求](#)，以及[时间值中的分数秒](#)。

# Chapter 2: Data Types

## Section 2.1: CHAR(n)

CHAR(n) is a string of a *fixed* length of n *characters*. If it is CHARACTER SET utf8mb4, that means it occupies exactly 4\*n bytes, regardless of what text is in it.

Most use cases for CHAR(n) involve strings that contain English characters, hence should be CHARACTER SET ascii. (latin1 will do just as good.)

```
country_code CHAR(2) CHARACTER SET ascii,
postal_code  CHAR(6) CHARACTER SET ascii,
uuid         CHAR(39) CHARACTER SET ascii, -- more discussion elsewhere
```

## Section 2.2: DATE, DATETIME, TIMESTAMP, YEAR, and TIME

The **DATE** datatype comprises the date but no time component. Its format is 'YYYY-MM-DD' with a range of '1000-01-01' to '9999-12-31'.

The **DATETIME** type includes the time with a format of 'YYYY-MM-DD HH:MM:SS'. It has a range from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'.

The **TIMESTAMP** type is an integer type comprising date and time with an effective range from '1970-01-01 00:00:01' UTC to '2038-01-19 03:14:07' UTC.

The **YEAR** type represents a year and holds a range from 1901 to 2155.

The **TIME** type represents a time with a format of 'HH:MM:SS' and holds a range from '-838:59:59' to '838:59:59'.

Storage Requirements:

Data Type	Before MySQL 5.6.4	as of MySQL 5.6.4
YEAR	1 byte	1 byte
DATE	3 bytes	3 bytes
TIME	3 bytes	3 bytes + fractional seconds storage
DATETIME	8 bytes	5 bytes + fractional seconds storage
TIMESTAMP	4 bytes	4 bytes + fractional seconds storage

Fractional Seconds (as of Version 5.6.4):

Fractional Seconds Precision	Storage Required
0	0 bytes
1,2	1 byte
3,4	2 byte
5,6	3 byte

See the MySQL Manual Pages [DATE, DATETIME, and TIMESTAMP Types](#), [Data Type Storage Requirements](#), and [Fractional Seconds in Time Values](#).

第 2.3 节：VARCHAR(255) —— 或者不是

建议的最大长度

首先，我将提到一些总是以十六进制表示，或仅限于 ASCII 的常见字符串。对于这些，您应该指定字符集ascii（latin1 也可以），以避免浪费空间：

```
UUID CHAR(36)字符集ascii—— 或者打包成 BINARY(16)
country_code CHAR(2)字符集ascii
ip_address CHAR(39)字符集ascii—— 或者打包成 BINARY(16)
phone VARCHAR(20)字符集ascii—— 可能足够处理分机号
postal_code VARCHAR(20)字符集ascii—— （不是“zip_code”）（不知道最大值）

city VARCHAR(100) —— 这个俄罗斯城镇需要 91 个字符：
Poselok Uchebnogo Khozyaystva Srednego Professionalno-Tekhnicheskoye Uchilishche Nomer Odin
country VARCHAR(50) —— 可能足够
name VARCHAR(64) —— 可能足够；比一些政府机构允许的还多
```

为什么不直接用 255？避免对所有字段都使用 (255) 的常见做法有两个原因。

- 当一个复杂的SELECT需要创建临时表（用于子查询、UNION、GROUP BY等）时，首选是使用MEMORY引擎，它将数据放入内存中。但VARCHAR会在此过程中被转换为CHAR。这使得VARCHAR(255)字符集utf8mb4 占用 1020 字节。这可能导致需要写入磁盘，速度较慢。

- 在某些情况下，InnoDB 会查看表中列的潜在大小，并判断其过大，从而中止CREATE TABLE操作。

VARCHAR 与 TEXT

\*TEXT、CHAR 和 VARCHAR 的使用提示，以及一些最佳实践：

- 切勿使用 TINYTEXT。
- 几乎不要使用 CHAR —— 它是固定长度；每个字符是 CHARACTER SET 的最大长度（例如，utf8mb4 为每字符 4 字节）。
- 使用 CHAR 时，除非另有说明，否则使用 CHARACTER SET ascii。
- VARCHAR(n) 会在 n 字符 处截断；TEXT 会在某个 字节 数处截断。（但是，你想要截断吗？）

- \*TEXT 可能 会因临时表的处理方式而降低复杂 SELECTs 的速度。

第 2.4 节：作为 AUTO\_INCREMENT 的 INT

任何大小的 INT 都可以用于 AUTO\_INCREMENT。 UNSIGNED 总是合适的。

请记住，某些操作会“消耗” AUTO\_INCREMENT ID，这可能导致意外的间隙。示例：  
INSERT IGNORE 和 REPLACE。它们 可能 会在意识到不需要之前预分配一个ID。这是InnoDB引擎中预期的行为且是设计使然，不应因此而阻止使用它们。

第2.5节：其他

已经有单独的条目针对“FLOAT、DOUBLE和DECIMAL”以及“ENUM”。单独一页关于数据类型可能会显得笨重——我建议“字段类型”（或者应该叫“数据类型”？）作为概述，然后拆分成以下主题页面：

- 整数 (INT)
- FLOAT、DOUBLE和DECIMAL

Section 2.3: VARCHAR(255) -- or not

Suggested max len

First, I will mention some common strings that are always hex, or otherwise limited to ASCII. For these, you should specify CHARACTER SET ascii (latin1 is ok) so that it will not waste space:

```
UUID CHAR(36) CHARACTER SET ascii -- or pack into BINARY(16)
country_code CHAR(2) CHARACTER SET ascii
ip_address CHAR(39) CHARACTER SET ascii -- or pack into BINARY(16)
phone VARCHAR(20) CHARACTER SET ascii -- probably enough to handle extension
postal_code VARCHAR(20) CHARACTER SET ascii -- (not 'zip_code') (don't know the max)

city VARCHAR(100) -- This Russian town needs 91:
Poselok Uchebnogo Khozyaystva Srednego Professionalno-Tekhnicheskoye Uchilishche Nomer Odin
country VARCHAR(50) -- probably enough
name VARCHAR(64) -- probably adequate; more than some government agencies allow
```

Why not simply 255? There are two reasons to avoid the common practice of using (255) for everything.

- When a complex SELECT needs to create temporary table (for a subquery, UNION, GROUP BY, etc), the preferred choice is to use the MEMORY engine, which puts the data in RAM. But VARCHARs are turned into CHAR in the process. This makes VARCHAR(255) CHARACTER SET utf8mb4 take 1020 bytes. That can lead to needing to spill to disk, which is slower.
- In certain situations, InnoDB will look at the potential size of the columns in a table and decide that it will be too big, aborting a CREATE TABLE.

VARCHAR versus TEXT

Usage hints for \*TEXT, CHAR, and VARCHAR, plus some Best Practice:

- Never use TINYTEXT.
- Almost never use CHAR -- it is fixed length; each character is the max length of the CHARACTER SET (eg, 4 bytes/character for utf8mb4).
- With CHAR, use CHARACTER SET ascii unless you know otherwise.
- VARCHAR(n) will truncate at n characters; TEXT will truncate at some number of bytes. (But, do you want truncation?)
- \*TEXT may slow down complex SELECTs due to how temp tables are handled.

Section 2.4: INT as AUTO\_INCREMENT

Any size of INT may be used for AUTO\_INCREMENT. UNSIGNED is always appropriate.

Keep in mind that certain operations "burn" AUTO\_INCREMENT ids. This could lead to an unexpected gap. Examples: INSERT IGNORE and REPLACE. They may preallocate an id before realizing that it won't be needed. This is expected behavior and by design in the InnoDB engine and should not discourage their use.

Section 2.5: Others

There is already a separate entry for "FLOAT, DOUBLE, and DECIMAL" and "ENUM". A single page on datatypes is likely to be unwieldy -- I suggest "Field types" (or should it be called "Datatypes"?) be an overview, then split into these topic pages:

- INTs
- FLOAT, DOUBLE, and DECIMAL



- 字符串（CHAR、TEXT等）
- 二进制和BLOB
- DATETIME、TIMESTAMP及相关类型
- ENUM和SET
- 空间数据
- JSON类型（MySQL 5.7.8及以上）
- 如何表示货币，以及其他需要强行塞入现有数据类型的常见“类型”

在适当的情况下，每个主题页面除了语法和示例外，还应包括：

- 更改大小（字节）时的注意事项
- 与非MySQL引擎的对比（低优先级）
- 在主键或二级索引中使用该数据类型时的注意事项
- 其他最佳实践
- 其他性能问题

（我假设当我的建议被采纳或否决后，这个“示例”会自动销毁。）

## 第2.6节：隐式/自动类型转换

```
select '123' * 2;
```

为了使与2的乘法运算，MySQL会自动将字符串123转换为数字。

返回值：

246

数字转换从左到右开始。如果无法转换，结果为0

```
select '123ABC' * 2
```

返回值：

246

```
select 'ABC123' * 2
```

返回值：

0

## 第2.7节：介绍（数值型）

MySQL 提供了多种不同的数值类型。这些可以分为

组别	类型
整数类型	INTEGER, INT, SMALLINT, TINYINT, MEDIUMINT, BIGINT

- Strings (CHARs, TEXT, etc)
- BINARY and BLOB
- DATETIME, TIMESTAMP, and friends
- ENUM and SET
- Spatial data
- JSON type (MySQL 5.7.8+)
- How to represent Money, and other common 'types' that need shoehorning into existing datatypes

Where appropriate, each topic page should include, in addition to syntax and examples:

- Considerations when ALTERing
- Size (bytes)
- Contrast with non-MySQL engines (low priority)
- Considerations when using the datatype in a PRIMARY KEY or secondary key
- other Best Practice
- other Performance issues

(I assume this "example" will self-distruct when my suggestions have been satisfied or vetoed.)

## Section 2.6: Implicit / automatic casting

```
select '123' * 2;
```

To make the **multiplication** with 2 MySQL automatically converts the string 123 into a number.

Return value:

246

The conversion to a number starts from left to right. If the conversion is not possible the result is 0

```
select '123ABC' * 2
```

Return value:

246

```
select 'ABC123' * 2
```

Return value:

0

## Section 2.7: Introduction (numeric)

MySQL offers a number of different numeric types. These can be broken down into

Group	Types
Integer Types	INTEGER, INT, SMALLINT, TINYINT, MEDIUMINT, BIGINT

定点类型	DECIMAL, NUMERIC
浮点类型	FLOAT, DOUBLE
位值类型	BIT

第2.8节：整数类型

最小无符号值始终为0。

类型	存储 (字节)	最小值 (有符号)	最大值 (有符号)	最大值 (无符号)
TINYINT	1	-27 -128	27-1 127	28-1 255
SMALLINT2		-215 -32,768	215-1 32,767	216-1 65,535
MEDIUMINT 3		-223 -8,388,608	223-1 8,388,607	224-1 16,777,215
INT	4	-231 -2,147,483,648	231-1 2,147,483,647	232-1 4,294,967,295
BIGINT	8	-263 -9,223,372,036,854,775,808	263-1 9,223,372,036,854,775,807	264-1 18,446,744,073,709,551,615

第2.9节：定点类型

MySQL的DECIMAL和NUMERIC类型存储精确的数值数据。建议使用这些类型来保持精确的精度，例如用于货币。

十进制

这些值以二进制格式存储。在列声明中，应指定精度和小数位

精度表示存储值的有效数字位数。

小数位数表示存储在小数点后的数字位数

```
salary DECIMAL(5,2)
```

5 表示精度，2 表示小数位数。对于此示例，该列中可存储的值范围是 -999.99 到 999.99

如果省略小数位数参数，则默认为 0

该数据类型最多可存储 65 位数字。

DECIMAL(M,N) 占用的字节数大约是 M/2。

第 2.10 节：浮点类型

FLOAT 和 DOUBLE 表示近似数据类型。

类型	存储	精度	范围
FLOAT	4 字节	23 位有效位 / 约 7 位十进制数字	10^+/-38
双精度	8 字节	53 位有效数字 / 约 16 位十进制数字	10^+/-308

REAL 是 FLOAT 的同义词。 DOUBLE PRECISION 是 DOUBLE 的同义词。

Fixed Point Types	DECIMAL, NUMERIC
Floating Point Types	FLOAT, DOUBLE
Bit Value Type	BIT

Section 2.8: Integer Types

Minimal unsigned value is always 0.

Type	Storage (Bytes)	Minimum Value (Signed)	Maximum Value (Signed)	Maximum Value (Unsigned)
TINYINT	1	-27 -128	27-1 127	28-1 255
SMALLINT	2	-215 -32,768	215-1 32,767	216-1 65,535
MEDIUMINT	3	-223 -8,388,608	223-1 8,388,607	224-1 16,777,215
INT	4	-231 -2,147,483,648	231-1 2,147,483,647	232-1 4,294,967,295
BIGINT	8	-263 -9,223,372,036,854,775,808	263-1 9,223,372,036,854,775,807	264-1 18,446,744,073,709,551,615

Section 2.9: Fixed Point Types

MySQL's DECIMAL and NUMERIC types store exact numeric data values. It is recommended to use these types to preserve exact precision, such as for money.

Decimal

These values are stored in binary format. In a column declaration, the precision and scale should be specified

Precision represents the number of significant digits that are stored for values.

Scale represents the number of digits stored after the decimal

```
salary DECIMAL(5,2)
```

5 represents the precision and 2 represents the scale. For this example, the range of values that can be stored in this column is -999.99 to 999.99

If the scale parameter is omitted, it defaults to 0

This data type can store up to 65 digits.

The number of bytes taken by DECIMAL(M,N) is approximately M/2.

Section 2.10: Floating Point Types

FLOAT and DOUBLE represent approximate data types.

Type	Storage	Precision	Range
FLOAT	4 bytes	23 significant bits / ~7 decimal digits	10^+/-38
DOUBLE	8 bytes	53 significant bits / ~16 decimal digits	10^+/-308

REAL is a synonym for FLOAT. DOUBLE PRECISION is a synonym for DOUBLE.

虽然 MySQL 也允许使用 (M,D) 限定符，但不要使用它。(M,D) 表示值可以存储最多 M 位数字，其中 D 位在小数点后。数字将被四舍五入两次或截断；这会带来更多问题而非好处。

由于浮点值是近似值而非精确存储，尝试将其作为精确值进行比较可能导致问题。特别注意，FLOAT 值很少等于 DOUBLE 值。

## 第 2.11 节：位值类型

BIT 类型用于存储位字段值。 BIT(M) 允许存储最多 M 位的值，其中 M 的范围是 1 到 64

你也可以使用 bit value 表示法指定值。

```
b'111'      -> 7
b'10000000' -> 128
```

有时使用“移位”构造单个位值很方便，例如 (1 << 7) 表示 128。

在 NDB 表中，所有 BIT 列的最大总大小为 4096。

Although MySQL also permits (M,D) qualifier, do *not* use it. (M,D) means that values can be stored with up to M total digits, where D can be after the decimal. *Numbers will be rounded twice or truncated; this will cause more trouble than benefit.*

Because floating-point values are approximate and not stored as exact values, attempts to treat them as exact in comparisons may lead to problems. Note in particular that a **FLOAT** value rarely equals a **DOUBLE** value.

## Section 2.11: Bit Value Type

The **BIT** type is useful for storing bit-field values. **BIT(M)** allows storage of up to M-bit values where M is in the range of 1 to 64

You can also specify values with **bit value** notation.

```
b'111'      -> 7
b'10000000' -> 128
```

Sometimes it is handy to use 'shift' to construct a single-bit value, for example (1 << 7) for 128.

The maximum combined size of all BIT columns in an NDB table is 4096.



# 第三章：选择

SELECT 用于从一个或多个表中检索选定的行。

## 第3.1节：带 DISTINCT 的 SELECT

在 SELECT 后的 DISTINCT 子句用于消除结果集中重复的行。

创建表 `car`  
(  
  `car\_id` INT UNSIGNED NOT NULL PRIMARY KEY,  
  `name` VARCHAR(20),  
  `price` DECIMAL(8,2)  
);  
  
向 CAR (`car\_id`, `name`, `price`) 插入值 (1, 'Audi A1', '20000');  
向 CAR (`car\_id`, `name`, `price`) 插入值 (2, 'Audi A1', '15000');  
向 CAR (`car\_id`, `name`, `price`) 插入值 (3, 'Audi A2', '40000');  
向 CAR (`car\_id`, `name`, `price`) 插入值 (4, 'Audi A2', '40000');  
  
SELECT DISTINCT `name`, `price` FROM CAR;

+-----+-----+  
| name | price |  
+-----+-----+  
Audi A1	20000.00
Audi A1	15000.00
Audi A2	40000.00
+-----+-----+

DISTINCT 作用于所有列以返回结果，而非单独某一列。这一点常被新的 SQL 开发者误解。简而言之，重要的是结果集行级别的唯一性，而非列级别的唯一性。要理解这一点，可以观察上面结果集中“Audi A1”的情况。

对于较新版本的 MySQL，DISTINCT 与 ORDER BY 一起使用时会有影响。设置 ONLY\_FULL\_GROUP\_BY 会生效，详见以下 MySQL 手册页面，标题为 MySQL 对 GROUP BY 的处理。

## 第3.2节：选择所有列 (\*)

查询

```
SELECT * FROM stack;
```

结果

+-----+-----+-----+  
| id | username | password |  
+-----+-----+-----+  
| 1 | admin | admin |  
| 2 | stack | stack |  
+-----+-----+-----+  
2 行记录 (0.00 秒)

你可以通过以下方式从连接的一个表中选择所有列：

# Chapter 3: SELECT

SELECT is used to retrieve rows selected from one or more tables.

## Section 3.1: SELECT with DISTINCT

The DISTINCT clause after SELECT eliminates duplicate rows from the result set.

CREATE TABLE `car`  
(  
  `car\_id` INT UNSIGNED NOT NULL PRIMARY KEY,  
  `name` VARCHAR(20),  
  `price` DECIMAL(8,2)  
);  
  
INSERT INTO CAR (`car\_id`, `name`, `price`) VALUES (1, 'Audi A1', '20000');  
INSERT INTO CAR (`car\_id`, `name`, `price`) VALUES (2, 'Audi A1', '15000');  
INSERT INTO CAR (`car\_id`, `name`, `price`) VALUES (3, 'Audi A2', '40000');  
INSERT INTO CAR (`car\_id`, `name`, `price`) VALUES (4, 'Audi A2', '40000');  
  
SELECT DISTINCT `name`, `price` FROM CAR;

+-----+-----+  
| name | price |  
+-----+-----+  
Audi A1	20000.00
Audi A1	15000.00
Audi A2	40000.00
+-----+-----+

DISTINCT works across all columns to deliver the results, not individual columns. The latter is often a misconception of new SQL developers. In short, it is the distinctness at the row-level of the result set that matters, not distinctness at the column-level. To visualize this, look at "Audi A1" in the above result set.

For later versions of MySQL, DISTINCT has implications with its use alongside ORDER BY. The setting for ONLY\_FULL\_GROUP\_BY comes into play as seen in the following MySQL Manual Page entitled [MySQL Handling of GROUP BY](#).

## Section 3.2: SELECT all columns (\*)

Query

```
SELECT * FROM stack;
```

Result

+-----+-----+-----+  
| id | username | password |  
+-----+-----+-----+  
| 1 | admin | admin |  
| 2 | stack | stack |  
+-----+-----+-----+  
2 rows in set (0.00 sec)

You can select all columns from one table in a join by doing:

```
SELECT stack.* FROM stack JOIN Overflow ON stack.id = Overflow.id;
```

最佳实践 除非你在调试或将行数据提取到关联数组中，否则不要使用 \*，否则模式的更改（添加/删除/重新排列列）可能导致严重的应用错误。此外，如果你在结果集中列出所需的列，MySQL的查询优化器通常可以优化查询。

优点：

- 1. 当你添加/删除列时，不必修改使用了 SELECT \* 的地方
- 2. 写起来更简短
- 3. 你也能看到答案，那么SELECT\*的用法是否有正当理由？

缺点：

- 1. 你返回的数据比实际需要的更多。比如你添加了一个每行包含200k的VARBINARY列。你只在一个地方需要这条记录的数据——使用SELECT\*可能导致每10行返回2MB你不需要的数据
- 2. 明确使用了哪些数据
- 3. 指定列意味着当某列被删除时你会收到错误
- 4. 查询处理器需要做更多工作——确定表中存在哪些列（感谢 @vinodadhikary）
- 5. 你可以更容易地找到某列的使用位置
- 6. 如果使用SELECT \*，连接查询时会返回所有列
- 7. 你不能安全地使用序号引用（尽管使用序号引用列本身就是不好的做法）
- 8. 在包含TEXT字段的复杂查询中，查询可能因次优的临时表处理而变慢

第3.3节：按列名SELECT

```
CREATE TABLE stack(  
    id INT,  
    username VARCHAR(30) NOT NULL,  
    password VARCHAR(30) NOT NULL  
);  
  
INSERT INTO stack (`id`, `username`, `password`) VALUES (1, 'Foo', 'hiddenGem');  
INSERT INTO stack (`id`, `username`, `password`) VALUES (2, 'Baa', 'verySecret');
```

查询

```
SELECT id FROM stack;
```

结果

-----
id
-----
1
2
+-----+

第3.4节：使用LIKE (%)的SELECT

```
创建表 stack
```

```
SELECT stack.* FROM stack JOIN Overflow ON stack.id = Overflow.id;
```

Best Practice Do not use \* unless you are debugging or fetching the row(s) into associative arrays, otherwise schema changes (ADD/DROP/rearrange columns) can lead to nasty application errors. Also, if you give the list of columns you need in your result set, MySQL's query planner often can optimize the query.

Pros:

- 1. When you add/remove columns, you don't have to make changes where you did use SELECT \*
- 2. It's shorter to write
- 3. You also see the answers, so can SELECT \*-usage ever be justified?

Cons:

- 1. You are returning more data than you need. Say you add a VARBINARY column that contains 200k per row. You only need this data in one place for a single record - using SELECT \* you can end up returning 2MB per 10 rows that you don't need
- 2. Explicit about what data is used
- 3. Specifying columns means you get an error when a column is removed
- 4. The query processor has to do some more work - figuring out what columns exist on the table (thanks @vinodadhikary)
- 5. You can find where a column is used more easily
- 6. You get all columns in joins if you use SELECT \*
- 7. You can't safely use ordinal referencing (though using ordinal references for columns is bad practice in itself)
- 8. In complex queries with TEXT fields, the query may be slowed down by less-optimal temp table processing

Section 3.3: SELECT by column name

```
CREATE TABLE stack(  
    id INT,  
    username VARCHAR(30) NOT NULL,  
    password VARCHAR(30) NOT NULL  
);  
  
INSERT INTO stack (`id`, `username`, `password`) VALUES (1, 'Foo', 'hiddenGem');  
INSERT INTO stack (`id`, `username`, `password`) VALUES (2, 'Baa', 'verySecret');
```

Query

```
SELECT id FROM stack;
```

Result

+-----+
id
+-----+
1
2
+-----+

Section 3.4: SELECT with LIKE (%)

```
CREATE TABLE stack
```

```
( id int 自动递增 主键,
  用户名 VARCHAR(100) 非空
);

插入 stack(用户名) 值
('admin'),('k admin'),('adm'),('a adm b'),('b XadmY c'), ('adm now'), ('not here');
```

“adm” 出现在任意位置：

```
选择 * 从 stack 哪里 用户名 LIKE "%adm%";

+----+-----+
| id | 用户名 |
+----+-----+
| 1 | admin |
| 2 | k admin |
| 3 | adm |
| 4 | a adm b |
| 5 | b XadmY c |
| 6 | adm now |
+----+-----+
```

以“adm”开头：

```
SELECT * FROM stack WHERE username LIKE "adm%";

+----+-----+
| id | username |
+----+-----+
| 1 | admin |
| 3 | adm |
| 6 | adm now |
+----+-----+
```

以 "adm" 结尾：

```
SELECT * FROM stack WHERE username LIKE "%adm";

+----+-----+
| id | username |
+----+-----+
| 3 | adm |
+----+-----+
```

正如 LIKE 子句中的 % 字符匹配任意数量的字符，\_ 字符只匹配一个字符。例如，

```
SELECT * FROM stack WHERE username LIKE "adm_n";

+----+-----+
| id | username |
+----+-----+
| 1 | admin |
+----+-----+
```

```
( id int AUTO_INCREMENT PRIMARY KEY,
  username VARCHAR(100) NOT NULL
);

INSERT stack(username) VALUES
('admin'),('k admin'),('adm'),('a adm b'),('b XadmY c'), ('adm now'), ('not here');
```

"adm" anywhere:

```
SELECT * FROM stack WHERE username LIKE "%adm%";

+----+-----+
| id | username |
+----+-----+
| 1 | admin |
| 2 | k admin |
| 3 | adm |
| 4 | a adm b |
| 5 | b XadmY c |
| 6 | adm now |
+----+-----+
```

Begins with "adm":

```
SELECT * FROM stack WHERE username LIKE "adm%";

+----+-----+
| id | username |
+----+-----+
| 1 | admin |
| 3 | adm |
| 6 | adm now |
+----+-----+
```

Ends with "adm":

```
SELECT * FROM stack WHERE username LIKE "%adm";

+----+-----+
| id | username |
+----+-----+
| 3 | adm |
+----+-----+
```

Just as the % character in a LIKE clause matches any number of characters, the \_ character matches just one character. For example,

```
SELECT * FROM stack WHERE username LIKE "adm_n";

+----+-----+
| id | username |
+----+-----+
| 1 | admin |
+----+-----+
```

性能说明 如果在 username 上有索引，则

- LIKE 'adm' 的执行效果与 `= 'adm'` 相同
- LIKE 'adm%' 是一种“范围”，类似于 BETWEEN..AND.. 它可以很好地利用该列上的索引。
- LIKE '%adm'（或任何带有前导通配符的变体）无法使用任何索引。因此查询会很慢。在包含大量行的表上，速度可能慢到毫无用处。
- RLIKE（REGEXP）通常比LIKE慢，但功能更强大。
- 虽然 MySQL 在多种类型的表和列上提供了FULLTEXT索引，但这些FULLTEXT索引不会用于满足使用LIKE的查询。

### 第3.5节：带有 CASE 或 IF 的 SELECT

#### 查询

```
SELECT st.name,
st.percentage,
CASE WHEN st.percentage >= 35 THEN '通过' ELSE '不通过' END AS `备注`
FROM student AS st ;
```

#### 结果

姓名	百分比	备注
Isha	67	通过
Rucha	28	不通过
Het	35	通过
Ansh	92	通过

#### 或者使用 IF

```
SELECT st.name,
st.percentage,
IF(st.percentage >= 35, '通过', '不通过') 作为`备注`
来自 student 作为 st ;
```

#### 注意

IF(st.percentage >= 35, '通过', '不通过')

这意味着：如果 st.percentage >= 35 为真，则返回'通过'，否则返回'不通过'

### 第3.6节：使用别名（AS）的SELECT

SQL别名用于临时重命名表或列。它们通常用于提高可读性。

#### 查询

```
SELECT username 作为 val 来自 stack;
SELECT username val 来自 stack;
```

（注意：AS在语法上是可选的。）

Performance Notes If there is an index on username, then

- **LIKE** 'adm' performs the same as `= 'adm'`
- **LIKE** 'adm%' is a "range", similar to **BETWEEN**..**AND**.. It can make good use of an index on the column.
- **LIKE** '%adm' (or any variant with a *leading* wildcard) cannot use any index. Therefore it will be slow. On tables with many rows, it is likely to be so slow it is useless.
- **RLIKE (REGEXP)** tends to be slower than **LIKE**, but has more capabilities.
- While MySQL offers **FULLTEXT** indexing on many types of table and column, those **FULLTEXT** indexes are *not* used to fulfill queries using **LIKE**.

### Section 3.5: SELECT with CASE or IF

#### Query

```
SELECT st.name,
st.percentage,
CASE WHEN st.percentage >= 35 THEN 'Pass' ELSE 'Fail' END AS `Remark`
FROM student AS st ;
```

#### Result

name	percentage	Remark
Isha	67	Pass
Rucha	28	Fail
Het	35	Pass
Ansh	92	Pass

#### Or with IF

```
SELECT st.name,
st.percentage,
IF(st.percentage >= 35, 'Pass', 'Fail') AS `Remark`
FROM student AS st ;
```

#### N.B

IF(st.percentage >= 35, 'Pass', 'Fail')

This means : IF st.percentage >= 35 is **TRUE** then return 'Pass' ELSE return 'Fail'

### Section 3.6: SELECT with Alias (AS)

SQL aliases are used to temporarily rename a table or a column. They are generally used to improve readability.

#### Query

```
SELECT username AS val FROM stack;
SELECT username val FROM stack;
```

(Note: AS is syntactically optional.)



结果

```
-----+
| val  |
+-----+
| 管理员 |
| 堆栈 |
+-----+
2 行记录 (0.00 秒)
```

第3.7节：带LIMIT子句的SELECT

查询：

```
SELECT *
FROM Customers
ORDER BY CustomerID
LIMIT 3;
```

结果：

客户编号	客户名称	联系人姓名	地址	城市	邮政编码	国家
1	阿尔弗雷德的食品店	玛丽亚·安德斯	上街57号	柏林	12209	德国
2	安娜·特鲁希略 三明治与冰淇淋	安娜·特鲁希略	宪法大道222号	墨西哥联邦区	05021	墨西哥
3	安东尼奥·莫雷诺·塔克里亚	安东尼奥·莫雷诺·马塔德罗斯	2312	墨西哥联邦区	05023	墨西哥

最佳实践 使用ORDER BY 时总是搭配LIMIT；否则你得到的行将是不可预测的。

查询：

```
SELECT *
  来自 Customers
  按 CustomerID排序
  限制 2,1;
```

说明：

当LIMIT子句包含两个数字时，它被解释为LIMIT 偏移量,计数。因此，在此示例中，查询跳过两条记录并返回一条。

结果：

客户编号	客户名称	联系人姓名	地址	城市	邮政编码	国家
3	安东尼奥·莫雷诺·塔克里奥	安东尼奥·莫雷诺	马塔德罗斯2312	墨西哥城	05023	墨西哥

注意：

LIMIT 子句中的值必须是常量；不能是列值。

第3.8节：使用 BETWEEN 的 SELECT

您可以使用 BETWEEN 子句来替代“greater than equal AND less than equal”的组合条件。

Result

```
+-----+
| val  |
+-----+
| admin |
| stack |
+-----+
2 rows in set (0.00 sec)
```

Section 3.7: SELECT with a LIMIT clause

Query:

```
SELECT *
FROM Customers
ORDER BY CustomerID
LIMIT 3;
```

Result:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

Best Practice Always use ORDER BY when using LIMIT; otherwise the rows you will get will be unpredictable.

Query:

```
SELECT *
  FROM Customers
  ORDER BY CustomerID
  LIMIT 2,1;
```

Explanation:

When a LIMIT clause contains two numbers, it is interpreted as LIMIT offset,count. So, in this example the query skips two records and returns one.

Result:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

Note:

The values in LIMIT clauses must be constants; they may not be column values.

Section 3.8: SELECT with BETWEEN

You can use BETWEEN clause to replace a combination of "greater than equal AND less than equal" conditions.

数据

id	用户名
1	admin
2	root
3	toor
4	mysql
5	thanks
6	java

带操作符的查询

```
SELECT * FROM stack WHERE id >= 2 and id <= 5;
```

类似的带有 BETWEEN 的查询

```
SELECT * FROM stack WHERE id BETWEEN 2 and 5;
```

结果

id	username
2	root
3	toor
4	mysql
5	thanks

4 rows in set (0.00 sec)

注意

BETWEEN 使用 >= 和 <=, 而不是 > 和 <。

使用 NOT BETWEEN

如果你想使用否定, 可以使用 NOT。例如：

```
SELECT * FROM stack WHERE id NOT BETWEEN 2 and 5;
```

结果

id	username
1	admin
6	java

2 rows in set (0.00 sec)

Data

id	username
1	admin
2	root
3	toor
4	mysql
5	thanks
6	java

Query with operators

```
SELECT * FROM stack WHERE id >= 2 and id <= 5;
```

Similar query with BETWEEN

```
SELECT * FROM stack WHERE id BETWEEN 2 and 5;
```

Result

id	username
2	root
3	toor
4	mysql
5	thanks

4 rows in set (0.00 sec)

Note

BETWEEN uses >= and <=, not > and <.

Using NOT BETWEEN

If you want to use the negative you can use NOT. For example :

```
SELECT * FROM stack WHERE id NOT BETWEEN 2 and 5;
```

Result

id	username
1	admin
6	java

2 rows in set (0.00 sec)

注意

**NOT BETWEEN** 使用 **>** 和 **<**，而不是 **>=** 和 **<=**。也就是说，**WHERE** id **NOT BETWEEN** 2 **and** 5 等同于 **WHERE** (id **<** 2 **OR** id **>** 5)。

如果你在用于 **BETWEEN** 查询的列上有索引，MySQL 可以使用该索引进行范围扫描。

第3.9节：带 WHERE 的 SELECT

查询

```
SELECT * FROM stack WHERE username = "admin" AND password = "admin";
```

结果

+-----+-----+-----+
id   username   password
+-----+-----+-----+
1   admin   admin
+-----+-----+-----+
1 row in set (0.00 sec)

带有嵌套SELECT的WHERE子句查询

WHERE子句可以包含任何有效的SELECT语句，以编写更复杂的查询。这就是所谓的“嵌套”查询

查询

嵌套查询通常用于从查询中返回单个原子值以进行比较。

```
SELECT title FROM books WHERE author_id = (SELECT id FROM authors WHERE last_name = 'Bar' AND first_name = 'Foo');
```

选择所有没有电子邮件地址的用户名

```
SELECT * FROM stack WHERE username IN (SELECT username FROM signups WHERE email IS NULL);
```

免责声明：在比较整个结果集时，考虑使用连接（joins）以提升性能。

第3.10节：带LIKE( )的SELECT

LIKE 子句模式中的 A \_ 字符匹配单个字符。

查询

```
SELECT username FROM users WHERE users LIKE 'admin_';
```

结果

-----
username
-----
admin1

Note

**NOT BETWEEN** uses **>** and **<** and not **>=** and **<=** That is, **WHERE** id **NOT BETWEEN** 2 **and** 5 is the same as **WHERE** (id **<** 2 **OR** id **>** 5).

If you have an index on a column you use in a **BETWEEN** search, MySQL can use that index for a range scan.

Section 3.9: SELECT with WHERE

Query

```
SELECT * FROM stack WHERE username = "admin" AND password = "admin";
```

Result

+-----+-----+-----+
id   username   password
+-----+-----+-----+
1   admin   admin
+-----+-----+-----+
1 row in set (0.00 sec)

Query with a nested SELECT in the WHERE clause

The **WHERE** clause can contain any valid **SELECT** statement to write more complex queries. This is a 'nested' query

Query

Nested queries are usually used to return single atomic values from queries for comparisons.

```
SELECT title FROM books WHERE author_id = (SELECT id FROM authors WHERE last_name = 'Bar' AND first_name = 'Foo');
```

Selects all usernames with no email address

```
SELECT * FROM stack WHERE username IN (SELECT username FROM signups WHERE email IS NULL);
```

Disclaimer: Consider using [joins](#) for performance improvements when comparing a whole result set.

Section 3.10: SELECT with LIKE( )

A \_ character in a **LIKE** clause pattern matches a single character.

Query

```
SELECT username FROM users WHERE users LIKE 'admin_';
```

Result

+-----+
username
+-----+
admin1

```
| admin2 |
| admin- |
| adminA |
|-----|
```

### 第3.11节：带日期范围的SELECT

```
SELECT ... WHERE dt >= '2017-02-01'
              AND dt < '2017-02-01' + INTERVAL 1 MONTH
```

当然，这也可以用BETWEEN和包含23:59:59来实现。但这种模式有以下优点：

- 你不需要预先计算结束日期（通常是从开始日期起的精确长度）
- 你不包含两个端点（如BETWEEN所做的），也不需要输入'23:59:59'来避免包含端点。
- 它适用于DATE、TIMESTAMP、DATETIME，甚至包含微秒的DATETIME(6)。
- 它能处理闰日、年末等情况。
- 它对索引友好（BETWEEN 也是如此）。

```
| admin2 |
| admin- |
| adminA |
+-----+
```

### Section 3.11: SELECT with date range

```
SELECT ... WHERE dt >= '2017-02-01'
              AND dt < '2017-02-01' + INTERVAL 1 MONTH
```

Sure, this could be done with BETWEEN and inclusion of 23:59:59. But, the pattern has this benefits:

- You don't have pre-calculate the end date (which is often an exact length from the start)
- You don't include both endpoints (as BETWEEN does), nor type '23:59:59' to avoid it.
- It works for DATE, TIMESTAMP, DATETIME, and even the microsecond-included DATETIME(6).
- It takes care of leap days, end of year, etc.
- It is index-friendly (so is BETWEEN).



# 第4章：反引号

## 第4.1节：反引号的用法

有许多示例中在查询内使用了反引号，但对许多人来说，何时或在哪里使用反引号`仍不清楚。

反引号主要用于防止一种称为“MySQL保留字”的错误。在PHPmyAdmin中创建表时，有时会遇到警告或提示，说明你正在使用“MySQL保留字”。

例如，当你创建一个名为“group”的列时，会收到警告。这是因为你可以编写如下查询：

```
SELECT student_name, AVG(test_score) FROM student GROUP BY group
```

为了确保查询不出错，你必须使用反引号，使查询变为：

```
SELECT student_name, AVG(test_score) FROM student GROUP BY `group`
```

### 表

不仅列名可以用反引号括起来，表名也可以。例如当你需要JOIN多个表时。

```
SELECT `users`.`username`, `groups`.`group` FROM `users`
```

### 更易阅读

如您所见，在表名和列名周围使用反引号也使查询更易阅读。

例如，当您习惯全部用小写字母编写查询时：

```
select student_name, AVG(test_score) from student group by group
select `student_name`, AVG(`test_score`) from `student` group by `group`
```

请参阅MySQL手册中标题为[关键词和保留字](#)的页面。带有(R)的为保留字。其他仅为关键词。保留字需要特别注意。

# Chapter 4: Backticks

## Section 4.1: Backticks usage

There are many examples where backticks are used inside a query but for many it's still unclear when or where to use backticks ``.

Backticks are mainly used to prevent an error called "*MySQL reserved word*". When making a table in PHPmyAdmin you are sometimes faced with a warning or alert that you are using a "*MySQL reserved word*".

For example when you create a table with a column named "group" you get a warning. This is because you can make the following query:

```
SELECT student_name, AVG(test_score) FROM student GROUP BY group
```

To make sure you don't get an error in your query you have to use backticks so your query becomes:

```
SELECT student_name, AVG(test_score) FROM student GROUP BY `group`
```

### Table

Not only column names can be surrounded by backticks, but also table names. For example when you need to JOIN multiple tables.

```
SELECT `users`.`username`, `groups`.`group` FROM `users`
```

### Easier to read

As you can see using backticks around table and column names also make the query easier to read.

For example when you are used to write queries all in lower case:

```
select student_name, AVG(test_score) from student group by group
select `student_name`, AVG(`test_score`) from `student` group by `group`
```

Please see the MySQL Manual page entitled [Keywords and Reserved Words](#). The ones with an (R) are Reserved Words. The others are merely Keywords. The Reserved require special caution.

# 第5章：NULL

## 第5.1节：NULL的用途

- 尚未知的数据 - 例如end\_date, rating
- 可选数据 - 例如middle\_initial (尽管用空字符串可能更合适)
- 0/0 - 某些计算的结果, 如零除以零。
- NULL不等于"" (空字符串) 或0 (整数情况下)。
- 其他？

## 第5.2节：测试NULL值

- IS NULL / IS NOT NULL -- = NULL 并不像你预期的那样工作。
- x <=> y 是一种“空安全”比较。

在 LEFT JOIN 中, 测试 a 中没有对应 b 行的行。

```
SELECT ...  
FROM a  
LEFT JOIN b ON ...  
WHERE b.id IS NULL
```

# Chapter 5: NULL

## Section 5.1: Uses for NULL

- Data not yet known - such as end\_date, rating
- Optional data - such as middle\_initial (though that might be better as the empty string)
- 0/0 - The result of certain computations, such as zero divided by zero.
- NULL is not equal to "" (blank string) or 0 (in case of integer).
- others?

## Section 5.2: Testing NULLs

- IS NULL / IS NOT NULL -- = NULL does not work like you expect.
- x <=> y is a "null-safe" comparison.

In a LEFT JOIN tests for rows of a for which there is *not* a corresponding row in b.

```
SELECT ...  
FROM a  
LEFT JOIN b ON ...  
WHERE b.id IS NULL
```

# 第6章：限制和偏移

## 第6.1节：极限与偏移关系

考虑以下users表：

id	用户名
1	用户1
2	用户2
3	用户3
4	用户4
5	用户5

为了限制SELECT查询结果集中的行数，LIMIT子句可以与一个或两个正整数（包括零）作为参数一起使用。

### 带一个参数的LIMIT子句

当使用一个参数时，结果集将仅限制为以下方式指定的数量：

```
SELECT * FROM users ORDER BY id ASC LIMIT 2
```

id	用户名
1	用户1
2	用户2

如果参数的值为0，结果集将为空。

还要注意，ORDER BY 子句可能很重要，用于指定将要展示的结果集的前几行（当按另一列排序时）。

### 带有两个参数的LIMIT子句

当在LIMIT子句中使用两个参数时：

- 第一个参数表示结果集行将从哪一行开始展示-这个数字通常被称为offset，因为它表示受限结果集初始行之前的行。这允许参数取值为0，从而考虑非受限结果集的第一行。
- 第二个参数指定结果集中返回的最大行数（类似于单参数示例）。

因此，查询：

```
SELECT * FROM users ORDER BY id ASC LIMIT 2, 3
```

展示以下结果集：

id	用户名
3	用户3
4	用户4
5	用户5

# Chapter 6: Limit and Offset

## Section 6.1: Limit and Offset relationship

Considering the following users table:

id	username
1	User1
2	User2
3	User3
4	User4
5	User5

In order to constrain the number of rows in the result set of a [SELECT query](#), the **LIMIT** clause can be used together with one or two positive integers as arguments (zero included).

### LIMIT clause with one argument

When one argument is used, the result set will only be constrained to the number specified in the following manner:

```
SELECT * FROM users ORDER BY id ASC LIMIT 2
```

id	username
1	User1
2	User2

If the argument's value is 0, the result set will be empty.

Also notice that the **ORDER BY** clause may be important in order to specify the first rows of the result set that will be presented (when ordering by another column).

### LIMIT clause with two arguments

When two arguments are used in a **LIMIT** clause:

- the **first** argument represents the row from which the result set rows will be presented – this number is often mentioned as an **offset**, since it represents the row previous to the initial row of the constrained result set. This allows the argument to receive 0 as value and thus taking into consideration the first row of the non-constrained result set.
- the **second** argument specifies the maximum number of rows to be returned in the result set (similarly to the one argument's example).

Therefore the query:

```
SELECT * FROM users ORDER BY id ASC LIMIT 2, 3
```

Presents the following result set:

id	username
3	User3
4	User4
5	User5

请注意，当offset参数为0时，结果集将等同于只有一个参数的LIMIT子句。这意味着以下两个查询：

```
SELECT * FROM users ORDER BY id ASC LIMIT 0, 2

SELECT * FROM users ORDER BY id ASC LIMIT 2
```

产生相同的结果集：

**id 用户名**  
1 用户1  
2 用户2  
**OFFSET 关键字：替代语法**

LIMIT 子句带两个参数的替代语法是在第一个参数后使用OFFSET 关键字，格式如下：

```
SELECT * FROM users ORDER BY id ASC LIMIT 2 OFFSET 3
```

该查询将返回以下结果集：

**id 用户名**  
3 用户3  
4 用户4

注意，在此替代语法中，参数的位置被交换了：

- 第一个参数表示结果集中要返回的行数；
- 第二个参数表示偏移量。

Notice that when the **offset** argument is 0, the result set will be equivalent to a one argument **LIMIT** clause. This means that the following 2 queries:

```
SELECT * FROM users ORDER BY id ASC LIMIT 0, 2

SELECT * FROM users ORDER BY id ASC LIMIT 2
```

Produce the same result set:

**id username**  
1 User1  
2 User2  
**OFFSET keyword: alternative syntax**

An alternative syntax for the **LIMIT** clause with two arguments consists in the usage of the **OFFSET** keyword after the first argument in the following manner:

```
SELECT * FROM users ORDER BY id ASC LIMIT 2 OFFSET 3
```

This query would return the following result set:

**id username**  
3 User3  
4 User4

Notice that in this alternative syntax the arguments have their positions switched:

- the **first** argument represents the number of rows to be returned in the result set;
- the **second** argument represents the offset.



# 第7章：创建数据库

参数	详情
CREATE DATABASE	创建一个具有指定名称的数据库
CREATE SCHEMA	这是CREATE DATABASE的同义词
如果不存在	用于避免执行错误，如果指定的数据库已存在
create_specification	create_specification 选项指定数据库特性，如字符集和排序规则（数据库排序规则）

## 第7.1节：创建数据库、用户和权限

创建一个数据库。注意，缩写词SCHEMA可以作为同义词使用。

```
CREATE DATABASE Baseball; -- 创建一个名为Baseball的数据库
```

如果数据库已存在，则返回错误1007。要避免此错误，请尝试：

```
CREATE DATABASE IF NOT EXISTS Baseball;
```

同样，

```
DROP DATABASE IF EXISTS Baseball; -- 如果数据库存在则删除，避免错误1008
DROP DATABASE xyz; -- 如果xyz不存在，将发生错误1008
```

由于上述错误的可能性，DDL语句通常与IF EXISTS一起使用。

可以创建带有默认字符集和排序规则的数据库。例如：

```
CREATE DATABASE Baseball CHARACTER SET utf8 COLLATE utf8_general_ci;

SHOW CREATE DATABASE Baseball;
```

```
+-----+-----+
| Database | Create Database                                     |
+-----+-----+
| Baseball | CREATE DATABASE `Baseball` /*!40100 DEFAULT CHARACTER SET utf8 */ |
+-----+-----+
```

查看当前数据库：

```
SHOW DATABASES;
```

```
-----
| Database          |
+-----+
| information_schema |
| ajax_stuff        |
| Baseball          |
+-----+
```

设置当前活动的数据库，并查看一些信息：

```
USE Baseball; -- 设置为当前数据库
```

# Chapter 7: Creating databases

Parameter	Details
CREATE DATABASE	Creates a database with the given name
CREATE SCHEMA	This is a synonym for <b>CREATE DATABASE</b>
IF NOT EXISTS	Used to avoid execution error, if specified database already exists
create_specification	create_specification options specify database characteristics such as CHARACTER <b>SET</b> and <b>COLLATE</b> (database collation)

## Section 7.1: Create database, users, and grants

Create a DATABASE. Note that the shortened word SCHEMA can be used as a synonym.

```
CREATE DATABASE Baseball; -- creates a database named Baseball
```

If the database already exists, Error 1007 is returned. To get around this error, try:

```
CREATE DATABASE IF NOT EXISTS Baseball;
```

Similarly,

```
DROP DATABASE IF EXISTS Baseball; -- Drops a database if it exists, avoids Error 1008
DROP DATABASE xyz; -- If xyz does not exist, ERROR 1008 will occur
```

Due to the above Error possibilities, DDL statements are often used with **IF EXISTS**.

One can create a database with a default CHARACTER SET and collation. For example:

```
CREATE DATABASE Baseball CHARACTER SET utf8 COLLATE utf8_general_ci;

SHOW CREATE DATABASE Baseball;
```

```
+-----+-----+
| Database | Create Database                                     |
+-----+-----+
| Baseball | CREATE DATABASE `Baseball` /*!40100 DEFAULT CHARACTER SET utf8 */ |
+-----+-----+
```

See your current databases:

```
SHOW DATABASES;
```

```
-----
| Database          |
+-----+
| information_schema |
| ajax_stuff        |
| Baseball          |
+-----+
```

Set the currently active database, and see some information:

```
USE Baseball; -- set it as the current database
```

```
SELECT @@character_set_database as cset,@@collation_database as col;
```

```
+-----+-----+
| cset | col          |
+-----+-----+
| utf8 | utf8_general_ci |
+-----+-----+
```

以上显示了数据库的默认字符集和排序规则。

创建用户：

```
CREATE USER 'John123'@'%' IDENTIFIED BY 'OpenSesame';
```

以上创建了一个用户名为 John123 的用户，能够使用通配符 % 连接任何主机名。该用户的密码设置为 'OpenSesame'，并已被哈希处理。

并创建另一个用户：

```
CREATE USER 'John456'@'%' IDENTIFIED BY 'somePassword';
```

通过检查特殊的 mysql 数据库来显示用户是否已创建：

```
SELECT user,host,password from mysql.user where user in ('John123','John456');
```

```
+-----+-----+-----+
| user  | host | password                                     |
+-----+-----+-----+
| John123 | %    | *E6531C342ED87 ..... |
| John456 | %    | *B04E11FAAAE9A ..... |
+-----+-----+-----+
```

注意，此时用户已创建，但尚未获得使用 Baseball 数据库的任何权限。

管理用户和数据库的权限。授予用户 John123 对 Baseball 数据库的全部权限，另一个用户仅授予 SELECT 权限：

```
GRANT ALL ON Baseball.* TO 'John123'@'%' ;
GRANT SELECT ON Baseball.* TO 'John456'@'%' ;
```

验证上述操作：

```
SHOW GRANTS FOR 'John123'@'%';
```

```
+-----+
-----+
| John123@% 的授权
|
+-----+
-----+
|           | 在 *.* 上授予 'John123'@'%' 使用权限，密码为 '*E6531C342ED87 .....
|
| 授予 'John123'@'%' 对 `baseball`. * 的所有权限
|
+-----+
-----+
```

```
SELECT @@character_set_database as cset,@@collation_database as col;
```

```
+-----+-----+
| cset | col          |
+-----+-----+
| utf8 | utf8_general_ci |
+-----+-----+
```

The above shows the default CHARACTER SET and Collation for the database.

Create a user:

```
CREATE USER 'John123'@'%' IDENTIFIED BY 'OpenSesame' ;
```

The above creates a user John123, able to connect with any hostname due to the % wildcard. The Password for the user is set to 'OpenSesame' which is hashed.

And create another:

```
CREATE USER 'John456'@'%' IDENTIFIED BY 'somePassword' ;
```

Show that the users have been created by examining the special mysql database:

```
SELECT user,host,password from mysql.user where user in ('John123','John456');
```

```
+-----+-----+-----+
| user  | host | password                                     |
+-----+-----+-----+
| John123 | %    | *E6531C342ED87 ..... |
| John456 | %    | *B04E11FAAAE9A ..... |
+-----+-----+-----+
```

Note that at this point, the users have been created, but without any permissions to use the Baseball database.

Work with permissions for users and databases. Grant rights to user John123 to have full privileges on the Baseball database, and just SELECT rights for the other user:

```
GRANT ALL ON Baseball.* TO 'John123'@'%' ;
GRANT SELECT ON Baseball.* TO 'John456'@'%' ;
```

Verify the above:

```
SHOW GRANTS FOR 'John123'@'%';
```

```
+-----+
-----+
| Grants for John123@%
|
+-----+
-----+
| GRANT USAGE ON *.* TO 'John123'@'%' IDENTIFIED BY PASSWORD '*E6531C342ED87 .....
|
| GRANT ALL PRIVILEGES ON `baseball`. * TO 'John123'@'%'
|
+-----+
-----+
```

```
SHOW 授权给 'John456'@'%';
```

```
+-----+
+-----+
| John456@% 的授权
|
+-----+
+-----+
|           | 在 *.* 上授予 'John456'@'% ' 使用权限，密码为 '*B04E11FAAAE9A .....
|
| 授予 'John456'@'% ' 对 `baseball`. * 的 SELECT 权限
|
+-----+
+-----+
```

请注意，您总是会看到的 GRANT USAGE 仅仅意味着用户可以登录。它仅仅是这个意思。

## 第7.2节：创建和选择数据库

如果管理员在设置权限时为您创建了数据库，您就可以开始使用它。否则，您需要自己创建它：

```
mysql> CREATE DATABASE menagerie;
```

在Unix系统下，数据库名称区分大小写（与SQL关键字不同），因此您必须始终将数据库称为menagerie，而不是Menagerie、MENAGERIE或其他变体。表名也是如此。（在Windows系统下，此限制不适用，但在同一查询中必须始终使用相同的大小写来引用数据库和表。出于多种原因，推荐的最佳做法是始终使用创建数据库时所用的大小写。）

创建数据库并不会自动选择它供使用；您必须显式执行此操作。要将menagerie设为当前数据库，使用以下语句：

```
mysql> USE menagerie
数据库已更改
```

您的数据库只需创建一次，但每次开始mysql会话时都必须选择它以供使用。您可以通过发出USE语句来完成此操作，如示例所示。或者，您也可以在调用mysql时在命令行中选择数据库。只需在可能需要提供的任何连接参数后指定数据库名称。例如：

```
shell> mysql -h host -u user -p menagerie
输入 密码: *****
```

## 第7.3节：MyDatabase

你必须创建你自己的数据库，而不能写入任何现有的数据库。这很可能是首次连接后要做的第一件事之一。

```
CREATE DATABASE my_db;
USE my_db;
CREATE TABLE some_table;
```

```
SHOW GRANTS FOR 'John456'@'% ';
```

```
+-----+
+-----+
| Grants for John456@%
|
+-----+
+-----+
| GRANT USAGE ON *.* TO 'John456'@'% ' IDENTIFIED BY PASSWORD '*B04E11FAAAE9A .....
|
| GRANT SELECT ON `baseball`. * TO 'John456'@'% '
|
+-----+
+-----+
```

Note that the **GRANT USAGE** that you will always see means simply that the user may login. That is all that that means.

## Section 7.2: Creating and Selecting a Database

If the administrator creates your database for you when setting up your permissions, you can begin using it. Otherwise, you need to create it yourself:

```
mysql> CREATE DATABASE menagerie;
```

Under Unix, database names are case sensitive (unlike SQL keywords), so you must always refer to your database as menagerie, not as Menagerie, MENAGERIE, or some other variant. This is also true for table names. (Under Windows, this restriction does not apply, although you must refer to databases and tables using the same lettercase throughout a given query. However, for a variety of reasons, the recommended best practice is always to use the same lettercase that was used when the database was created.)

Creating a database does not select it for use; you must do that explicitly. To make menagerie the current database, use this statement:

```
mysql> USE menagerie
Database changed
```

Your database needs to be created only once, but you must select it for use each time you begin a mysql session. You can do this by issuing a USE statement as shown in the example. Alternatively, you can select the database on the command line when you invoke mysql. Just specify its name after any connection parameters that you might need to provide. For example:

```
shell> mysql -h host -u user -p menagerie
Enter password: *****
```

## Section 7.3: MyDatabase

You *must* create your own database, and not use write to any of the existing databases. This is likely to be one of the very first things to do after getting connected the first time.

```
CREATE DATABASE my_db;
USE my_db;
CREATE TABLE some_table;
```

```
INSERT INTO some_table ...;
```

你可以通过加上数据库名来引用你的表：my\_db.some\_table。

## 第7.4节：系统数据库

以下数据库是供MySQL使用的。你可以读取（**SELECT**）它们，但不得写入（**INSERT/UPDATE/DELETE**）其中的表。（有少数例外。）

- mysql -- 用于存储GRANT信息及其他一些内容的仓库。
- information\_schema -- 这里的表是“虚拟”的，实际上由内存结构体现。它们的内容包括所有表的模式。
- performance\_schema -- ?? [请接受后再编辑]
- 其他??（适用于MariaDB、Galera、TokuDB等）

```
INSERT INTO some_table ...;
```

You can reference your table by qualifying with the database name: my\_db.some\_table.

## Section 7.4: System Databases

The following databases exist for MySQL's use. You may read (**SELECT**) them, but you must not write (**INSERT/UPDATE/DELETE**) the tables in them. (There are a few exceptions.)

- mysql -- repository for **GRANT** info and some other things.
- information\_schema -- The tables here are 'virtual' in the sense that they are actually manifested by in-memory structures. Their contents include the schema for all tables.
- performance\_schema -- ?? [please accept, then edit]
- others?? (for MariaDB, Galera, TokuDB, etc)



# 第8章：使用变量

## 第8.1节：设置变量

以下是设置变量的一些方法：

- 1. 你可以使用SET将变量设置为特定的字符串、数字、日期

```
SET @var_string = 'my_var';
SET @var_num = '2'
SET @var_date = '2015-07-20';
```

- 2. 你可以使用:=将变量设置为select语句的结果

```
Select @var := '123';
（注意：当赋值变量时，如果不使用SET语法，需要使用:=，因为在其他语句中（select、update等）"="用于比较，所以在"="前加冒号表示“这不是比较，这是赋值”。）
```

- 3. 你可以使用INTO将变量设置为select语句的结果

（当我需要动态选择查询哪个分区时，这一点特别有用）

```
SET @start_date = '2015-07-20';
SET @end_date = '2016-01-31';

#获取用于作为分区名称的年月值
SET @start_yearmonth = (SELECT EXTRACT(YEAR_MONTH FROM @start_date));
SET @end_yearmonth = (SELECT EXTRACT(YEAR_MONTH FROM @end_date));

#将分区放入变量中
SELECT GROUP_CONCAT(partition_name)
FROM information_schema.partitions p
WHERE table_name = 'partitioned_table'
AND SUBSTRING_INDEX(partition_name,'P',-1) BETWEEN @start_yearmonth AND @end_yearmonth
INTO @partitions;

#将查询放入变量中。你需要这样做，因为mysql没有将我的变量识别为该位置的变量。你需要将变量的值与查询的其余部分连接起来，然后作为语句执行。

SET @query =
CONCAT('CREATE TABLE part_of_partitioned_table (PRIMARY KEY(id))
SELECT partitioned_table.*
从分区_表 PARTITION(' , @partitions, ')
连接用户 u 使用(user_id)
WHERE date(partitioned_table.date) BETWEEN ' , @start_date, ' AND ' , @end_date);

#准备来自 @query 的语句
PREPARE stmt FROM @query;
#删除表
DROP TABLE IF EXISTS tech.part_of_partitioned_table;
#使用语句创建表
EXECUTE stmt;
```

# Chapter 8: Using Variables

## Section 8.1: Setting Variables

Here are some ways to set variables:

- 1. You can set a variable to a specific, string, number, date using SET

```
SET @var_string = 'my_var';
SET @var_num = '2'
SET @var_date = '2015-07-20';
```

- 2. you can set a variable to be the result of a select statement using :=

```
Select @var := '123';
(Note: You need to use := when assigning a variable not using the SET syntax, because in other statements, (select, update...) the "=" is used to compare, so when you add a colon before the "=", you are saying "This is not a comparison, this is a SET".)
```

- 3. You can set a variable to be the result of a select statement using INTO

(This was particularly helpful when I needed to dynamically choose which Partitions to query from)

```
SET @start_date = '2015-07-20';
SET @end_date = '2016-01-31';

#this gets the year month value to use as the partition names
SET @start_yearmonth = (SELECT EXTRACT(YEAR_MONTH FROM @start_date));
SET @end_yearmonth = (SELECT EXTRACT(YEAR_MONTH FROM @end_date));

#put the partitions into a variable
SELECT GROUP_CONCAT(partition_name)
FROM information_schema.partitions p
WHERE table_name = 'partitioned_table'
AND SUBSTRING_INDEX(partition_name,'P',-1) BETWEEN @start_yearmonth AND @end_yearmonth
INTO @partitions;

#put the query in a variable. You need to do this, because mysql did not recognize my variable as a variable in that position. You need to concat the value of the variable together with the rest of the query and then execute it as a stmt.
SET @query =
CONCAT('CREATE TABLE part_of_partitioned_table (PRIMARY KEY(id))
SELECT partitioned_table.*
FROM partitioned_table PARTITION(' , @partitions, ')
JOIN users u USING(user_id)
WHERE date(partitioned_table.date) BETWEEN ' , @start_date, ' AND ' , @end_date);

#prepare the statement from @query
PREPARE stmt FROM @query;
#drop table
DROP TABLE IF EXISTS tech.part_of_partitioned_table;
#create table using statement
EXECUTE stmt;
```

第8.2节：在Select语句中使用变量的行号和分组

假设我们有一个名为team\_person的表，如下所示：

队伍	人员
A	约翰
B	史密斯
A	沃尔特
A	路易斯
C	伊丽莎白
B	韦恩

```
CREATE TABLE team_person AS SELECT 'A' 队伍, 'John' 人员
UNION ALL SELECT 'B' 队伍, 'Smith' 人员
UNION ALL SELECT 'A' 队伍, 'Walter' 人员
UNION ALL SELECT 'A' 队伍, 'Louis' 人员
UNION ALL SELECT 'C' 队伍, 'Elizabeth' 人员
UNION ALL SELECT 'B' 队伍, 'Wayne' 人员;
```

要选择带有额外row\_number列的表team\_person，可以使用以下任一方法

```
SELECT @row_no := @row_no+1 AS row_number, team, person
FROM team_person, (SELECT @row_no := 0) t;
```

或者

```
SET @row_no := 0;
SELECT @row_no := @row_no + 1 AS row_number, team, person
FROM team_person;
```

将输出以下结果：

row_number	team	person
1	A	John
2	B	Smith
3	A	Walter
4	A	Louis
5	C	Elizabeth
6	B	Wayne

Section 8.2: Row Number and Group By using variables in Select Statement

Let's say we have a table team\_person as below:

team	person
A	John
B	Smith
A	Walter
A	Louis
C	Elizabeth
B	Wayne

```
CREATE TABLE team_person AS SELECT 'A' team, 'John' person
UNION ALL SELECT 'B' team, 'Smith' person
UNION ALL SELECT 'A' team, 'Walter' person
UNION ALL SELECT 'A' team, 'Louis' person
UNION ALL SELECT 'C' team, 'Elizabeth' person
UNION ALL SELECT 'B' team, 'Wayne' person;
```

To select the table team\_person with additional row\_number column, either

```
SELECT @row_no := @row_no+1 AS row_number, team, person
FROM team_person, (SELECT @row_no := 0) t;
```

OR

```
SET @row_no := 0;
SELECT @row_no := @row_no + 1 AS row_number, team, person
FROM team_person;
```

will output the result below:

row_number	team	person
1	A	John
2	B	Smith
3	A	Walter
4	A	Louis
5	C	Elizabeth
6	B	Wayne

最后, 如果我们想按列team分组获取row\_number

```
SELECT @row_no := IF(@prev_val = t.team, @row_no + 1, 1) AS row_number
      ,@prev_val := t.team AS team
      ,t.person
FROM team_person t,
     (SELECT @row_no := 0) x,
     (SELECT @prev_val := '') y
ORDER BY t.team ASC,t.person DESC;
```

row_number	team	person
1	A	Walter
2	A	Louis
3	A	John
1	B	Wayne
2	B	Smith
1	C	Elizabeth

Finally, if we want to get the row\_number group by column team

```
SELECT @row_no := IF(@prev_val = t.team, @row_no + 1, 1) AS row_number
      ,@prev_val := t.team AS team
      ,t.person
FROM team_person t,
     (SELECT @row_no := 0) x,
     (SELECT @prev_val := '') y
ORDER BY t.team ASC,t.person DESC;
```

row_number	team	person
1	A	Walter
2	A	Louis
3	A	John
1	B	Wayne
2	B	Smith
1	C	Elizabeth

# 第9章：MySQL注释

## 第9.1节：添加注释

注释有三种类型：

```
# 该注释持续到行尾

-- 该注释持续到行尾

/* 这是一个行内注释 */

/*
这是一个
多行注释
*/
```

示例：

```
SELECT * FROM t1; -- 这是注释

CREATE TABLE stack(
    /*id_user int,
    username varchar(30),
    password varchar(30)
    */
    id int
);
```

该--方法要求--后面必须有空格，注释才会生效，否则会被解释为命令，通常会导致错误。

```
#这个注释有效
/*这个注释有效。*/
--这个注释无效。
```

## 第9.2节：注释表定义

```
CREATE TABLE menagerie.bird (
    bird_id INT NOT NULL AUTO_INCREMENT,
    species VARCHAR(300) DEFAULT NULL COMMENT '可以包含属名，但绝不能包含亚种。',
    INDEX idx_species (species) COMMENT '我们经常需要根据物种进行搜索。',
    PRIMARY KEY (bird_id)
) ENGINE=InnoDB COMMENT '该表于2月10日启用。';
```

在 COMMENT 后使用 = 是可选的。 ([官方文档](#))

这些注释与其他注释不同，会随模式一起保存，并且可以通过 SHOW CREATE TABLE 或从 information\_schema 中检索。

# Chapter 9: Comment MySQL

## Section 9.1: Adding comments

There are three types of comment:

```
# This comment continues to the end of line

-- This comment continues to the end of line

/* This is an in-line comment */

/*
This is a
multiple-line comment
*/
```

Example:

```
SELECT * FROM t1; -- this is comment

CREATE TABLE stack(
    /*id_user int,
    username varchar(30),
    password varchar(30)
    */
    id int
);
```

The -- method requires that a space follows the -- before the comment begins, otherwise it will be interpreted as a command and usually cause an error.

```
#This comment works
/*This comment works.*/
--This comment does not.
```

## Section 9.2: Commenting table definitions

```
CREATE TABLE menagerie.bird (
    bird_id INT NOT NULL AUTO_INCREMENT,
    species VARCHAR(300) DEFAULT NULL COMMENT 'You can include genus, but never subspecies.',
    INDEX idx_species (species) COMMENT 'We must search on species often.',
    PRIMARY KEY (bird_id)
) ENGINE=InnoDB COMMENT 'This table was inaugurated on February 10th.';
```

Using an = after COMMENT is optional. ([Official docs](#))

These comments, unlike the others, are saved with the schema and can be retrieved via SHOW CREATE TABLE or from information\_schema.

# 第10章：INSERT

## 第10.1节：INSERT, ON DUPLICATE KEY UPDATE

```
INSERT INTO `table_name`
  (`index_field`, `other_field_1`, `other_field_2`)
VALUES
('index_value', 'insert_value', 'other_value')
ON DUPLICATE KEY UPDATE
  `other_field_1` = 'update_value',
  `other_field_2` = VALUES(`other_field_2`);
```

这将**INSERT**到table\_name指定的值，但如果唯一键已存在，则会更新other\_field\_1为新值。

有时，在重复键更新时，使用VALUES()来访问传递给INSERT的原始值而不是直接设置值会很方便。这样，你可以通过使用**INSERT**和**UPDATE**设置不同的值。参见上面的示例，其中other\_field\_1在INSERT时设置为insert\_value而在UPDATE时设置为update\_value，而other\_field\_2始终设置为other\_value。

使重复键更新（IODKU）生效的关键是模式中包含一个唯一键，用于标识重复冲突。该唯一键可以是主键，也可以不是。它可以是单列的唯一键，或者是多列（复合键）。

## 第10.2节：插入多行

```
INSERT INTO `my_table` (`field_1`, `field_2`) VALUES
  ('data_1', 'data_2'),
  ('data_1', 'data_3'),
  ('data_4', 'data_5');
```

这是使用一个INSERT语句一次添加多行的简便方法。

这种“批量”插入比逐行插入快得多。通常，以这种方式批量插入100行比逐个插入快10倍。

### 忽略现有行

在导入大型数据集时，在某些情况下，跳过通常会因列约束（例如重复主键）而导致查询失败的行可能更为合适。这可以使用INSERT IGNORE来实现。

考虑以下示例数据库：

```
SELECT * FROM `people`;
--- 产生结果：
+----+-----+
| id | name |
+----+-----+
|  1 | john |
|  2 | anna |
+----+-----+

INSERT IGNORE INTO `people` (`id`, `name`) VALUES
('2', 'anna'), --- 如果没有IGNORE关键字，这条记录会产生错误
('3', 'mike');
```

# Chapter 10: INSERT

## Section 10.1: INSERT, ON DUPLICATE KEY UPDATE

```
INSERT INTO `table_name`
  (`index_field`, `other_field_1`, `other_field_2`)
VALUES
('index_value', 'insert_value', 'other_value')
ON DUPLICATE KEY UPDATE
  `other_field_1` = 'update_value',
  `other_field_2` = VALUES(`other_field_2`);
```

This will **INSERT** into table\_name the specified values, but if the unique key already exists, it will update the other\_field\_1 to have a new value.

Sometimes, when updating on duplicate key it comes in handy to use **VALUES()** in order to access the original value that was passed to the **INSERT** instead of setting the value directly. This way, you can set different values by using **INSERT** and **UPDATE**. See the example above where other\_field\_1 is set to insert\_value on **INSERT** or to update\_value on **UPDATE** while other\_field\_2 is always set to other\_value.

Crucial for the Insert on Duplicate Key Update (IODKU) to work is the schema containing a unique key that will signal a duplicate clash. This unique key can be a Primary Key or not. It can be a unique key on a single column, or a multi-column (composite key).

## Section 10.2: Inserting multiple rows

```
INSERT INTO `my_table` (`field_1`, `field_2`) VALUES
  ('data_1', 'data_2'),
  ('data_1', 'data_3'),
  ('data_4', 'data_5');
```

This is an easy way to add several rows at once with one **INSERT** statement.

This kind of 'batch' insert is much faster than inserting rows one by one. Typically, inserting 100 rows in a single batch insert this way is 10 times as fast as inserting them all individually.

### Ignoring existing rows

When importing large datasets, it may be preferable under certain circumstances to skip rows that would usually cause the query to fail due to a column restraint e.g. duplicate primary keys. This can be done using **INSERT IGNORE**.

Consider following example database:

```
SELECT * FROM `people`;
--- Produces:
+----+-----+
| id | name |
+----+-----+
|  1 | john |
|  2 | anna |
+----+-----+

INSERT IGNORE INTO `people` (`id`, `name`) VALUES
('2', 'anna'), --- Without the IGNORE keyword, this record would produce an error
('3', 'mike');
```



```
SELECT * FROM `people`;
--- 产生结果：
+----+-----+
| id | name |
+----+-----+
| 1  | john |
| 2  | anna |
| 3  | mike |
+----+-----+
```

重要的是要记住，INSERT IGNORE也会静默跳过其他错误，以下是MySQL官方文档的说明：

如果未指定IGNORE，触发错误的数据转换会中止语句。使用IGNORE时，无效值会被调整为最接近的值并插入；会产生警告，但语句不会中止。

注意：以下部分为完整性添加，但不被视为最佳实践（例如，如果表中添加了另一列，这将会失败）。

如果为表中的所有列指定了对应列的值，则可以在INSERT语句中忽略列列表，如下所示：

```
INSERT INTO `my_table` VALUES
    ('data_1', 'data_2'),
    ('data_1', 'data_3'),
    ('data_4', 'data_5');
```

### 第10.3节：基本插入

```
INSERT INTO `table_name` (`field_one`, `field_two`) VALUES ('value_one', 'value_two');
```

在这个简单的例子中，table\_name 是要添加数据的表，field\_one 和 field\_two 是要设置数据的字段，value\_one 和 value\_two 分别是对应 field\_one 和 field\_two 的数据。

在代码中列出要插入数据的字段是个好习惯，因为如果表结构发生变化并添加了新列，若这些列不存在，插入操作将会失败。

### 第10.4节：带有 AUTO\_INCREMENT + LAST\_INSERT\_ID() 的 INSERT

当表有一个 AUTO\_INCREMENT 主键时，通常不向该列插入数据。相反，应指定所有其他列，然后查询新生成的ID。

```
CREATE TABLE t (
  id SMALLINT UNSIGNED AUTO_INCREMENT NOT NULL,
  this ...,
  that ...,
  PRIMARY KEY(id) );

INSERT INTO t (this, that) VALUES (... , ...);
SELECT LAST_INSERT_ID() INTO @id;
INSERT INTO another_table (... , t_id, ...) VALUES (... , @id, ...);
```

```
SELECT * FROM `people`;
--- Produces:
+----+-----+
| id | name |
+----+-----+
| 1  | john |
| 2  | anna |
| 3  | mike |
+----+-----+
```

The important thing to remember is that *INSERT IGNORE* will also silently skip other errors too, here is what Mysql official documentations says:

Data conversions that would trigger errors abort the statement if IGNORE is not > specified. With IGNORE, invalid values are adjusted to the closest values and >inserted; warnings are produced but the statement does not abort.

**Note: The section below is added for the sake of completeness, but is not considered best practice (this would fail, for example, if another column was added into the table).**

If you specify the value of the corresponding column for all columns in the table, you can ignore the column list in the **INSERT** statement as follows:

```
INSERT INTO `my_table` VALUES
    ('data_1', 'data_2'),
    ('data_1', 'data_3'),
    ('data_4', 'data_5');
```

### Section 10.3: Basic Insert

```
INSERT INTO `table_name` (`field_one`, `field_two`) VALUES ('value_one', 'value_two');
```

In this trivial example, table\_name is where the data are to be added, field\_one and field\_two are fields to set data against, and value\_one and value\_two are the data to do against field\_one and field\_two respectively.

It's good practice to list the fields you are inserting data into within your code, as if the table changes and new columns are added, your insert would break should they not be there

### Section 10.4: INSERT with AUTO\_INCREMENT + LAST\_INSERT\_ID()

When a table has an **AUTO\_INCREMENT PRIMARY KEY**, normally one does not insert into that column. Instead, specify all the other columns, then ask what the new id was.

```
CREATE TABLE t (
  id SMALLINT UNSIGNED AUTO_INCREMENT NOT NULL,
  this ...,
  that ...,
  PRIMARY KEY(id) );

INSERT INTO t (this, that) VALUES (... , ...);
SELECT LAST_INSERT_ID() INTO @id;
INSERT INTO another_table (... , t_id, ...) VALUES (... , @id, ...);
```

注意 `LAST_INSERT_ID()` 是绑定到会话的，因此即使多个连接同时向同一张表插入数据，每个连接都会获得自己的ID。

您的客户端API可能有另一种获取 `LAST_INSERT_ID()` 的方法，而无需实际执行 `SELECT` 并将值返回给客户端，而不是将其保留在MySQL中的 `@变量` 中。通常这种方式更为可取。

更长、更详细的示例

IODKU的“正常”用法是基于某个 `UNIQUE` 键触发“重复键”，而不是 `AUTO_INCREMENT PRIMARY KEY`。以下示例演示了这种情况。注意INSERT语句中 `id` 没有提供。

以下是后续示例的设置：

```
CREATE TABLE iodku (
  id INT AUTO_INCREMENT NOT NULL,
  name VARCHAR(99) NOT NULL,
  misc INT NOT NULL,
  PRIMARY KEY(id),
  UNIQUE(name)
) ENGINE=InnoDB;

INSERT INTO iodku (name, misc)
VALUES
('Leslie', 123),
('Sally', 456);
```

查询成功，影响行数：2 (0.00 秒)  
记录数：2 重复：0 警告：0

id	name	misc
1	Leslie	123
2	Sally	456

IODKU 执行“更新”操作并使用LAST\_INSERT\_ID()获取相关id的情况：

```
INSERT INTO iodku (name, misc)
VALUES
('Sally', 3333)          -- 应该执行更新
ON DUPLICATE KEY UPDATE -- `name` 将触发“重复键”
  id = LAST_INSERT_ID(id),
misc = VALUES(misc);
SELECT LAST_INSERT_ID();  -- 获取现有值
```

LAST_INSERT_ID()
2

IODKU 执行“插入”操作且 LAST\_INSERT\_ID() 获取新 id 的情况：

```
INSERT INTO iodku (name, misc)
VALUES
('Dana', 789)          -- 应该插入
```

Note that `LAST_INSERT_ID()` is tied to the session, so even if multiple connections are inserting into the same table, each will get its own id.

Your client API probably has an alternative way of getting the `LAST_INSERT_ID()` without actually performing a `SELECT` and handing the value back to the client instead of leaving it in an `@variable` inside MySQL. Such is usually preferable.

Longer, more detailed, example

The "normal" usage of IODKU is to trigger "duplicate key" based on some `UNIQUE` key, not the `AUTO_INCREMENT PRIMARY KEY`. The following demonstrates such. Note that it does *not* supply the id in the INSERT.

Setup for examples to follow:

```
CREATE TABLE iodku (
  id INT AUTO_INCREMENT NOT NULL,
  name VARCHAR(99) NOT NULL,
  misc INT NOT NULL,
  PRIMARY KEY(id),
  UNIQUE(name)
) ENGINE=InnoDB;

INSERT INTO iodku (name, misc)
VALUES
('Leslie', 123),
('Sally', 456);
```

Query OK, 2 rows affected (0.00 sec)  
Records: 2 Duplicates: 0 Warnings: 0

id	name	misc
1	Leslie	123
2	Sally	456

The case of IODKU performing an "update" and `LAST_INSERT_ID()` retrieving the relevant id:

```
INSERT INTO iodku (name, misc)
VALUES
('Sally', 3333)          -- should update
ON DUPLICATE KEY UPDATE -- `name` will trigger "duplicate key"
  id = LAST_INSERT_ID(id),
misc = VALUES(misc);
SELECT LAST_INSERT_ID();  -- picking up existing value
```

LAST_INSERT_ID()
2

The case where IODKU performs an "insert" and `LAST_INSERT_ID()` retrieves the new id:

```
INSERT INTO iodku (name, misc)
VALUES
('Dana', 789)          -- Should insert
```

```
ON DUPLICATE KEY UPDATE
id = LAST_INSERT_ID(id),
misc = VALUES(misc);
SELECT LAST_INSERT_ID(); -- 获取新值
```

LAST_INSERT_ID()
3

结果表内容：

```
SELECT * FROM iodku;
```

id	name	misc
1	莱斯利	123
2	莎莉	3333 -- IODKU 修改了这个
3	达娜	789 -- IODKU 添加了这个

### 第10.5节：INSERT SELECT（从另一张表插入数据）

这是使用SELECT语句从另一张表插入数据的基本方法。

```
INSERT INTO `tableA` (`field_one`, `field_two`)
SELECT `tableB`.`field_one`, `tableB`.`field_two`
FROM `tableB`
WHERE `tableB`.clmn <> 'someValue'
ORDER BY `tableB`.`sorting_clmn`;
```

你可以 SELECT \* FROM，但此时 tableA 和 tableB 必须 有相同的列数和对应的数据类型。

带有 AUTO\_INCREMENT 的列处理方式与使用 INSERT 和 VALUES 子句时相同。

这种语法使得用其他表的数据填充（临时）表变得简单，尤其是在插入时需要对数据进行过滤时更为方便。

### 第10.6节：丢失的AUTO\_INCREMENT ID

多个“插入”函数可能会“消耗”ID。以下是一个使用InnoDB的示例（其他存储引擎的行为可能不同）：

```
创建表 Burn (
id SMALLINT 无符号 自增 非空,
name VARCHAR(99) 非空,
主键(id),
唯一(name)
) 引擎=InnoDB;

插入忽略到 Burn (name) 值 ('first'), ('second');
选择 LAST_INSERT_ID(); -- 1
选择 * 从 Burn 按 id排序;
```

```
ON DUPLICATE KEY UPDATE
id = LAST_INSERT_ID(id),
misc = VALUES(misc);
SELECT LAST_INSERT_ID(); -- picking up new value
```

LAST_INSERT_ID()
3

Resulting table contents:

```
SELECT * FROM iodku;
```

id	name	misc
1	Leslie	123
2	Sally	3333 -- IODKU changed this
3	Dana	789 -- IODKU added this

### Section 10.5: INSERT SELECT (Inserting data from another Table)

This is the basic way to insert data from another table with the SELECT statement.

```
INSERT INTO `tableA` (`field_one`, `field_two`)
SELECT `tableB`.`field_one`, `tableB`.`field_two`
FROM `tableB`
WHERE `tableB`.clmn <> 'someValue'
ORDER BY `tableB`.`sorting_clmn`;
```

You can SELECT \* FROM, but then tableA and tableB *must* have matching column count and corresponding datatypes.

Columns with AUTO\_INCREMENT are treated as in the INSERT with VALUES clause.

This syntax makes it easy to fill (temporary) tables with data from other tables, even more so when the data is to be filtered on the insert.

### Section 10.6: Lost AUTO\_INCREMENT ids

Several 'insert' functions can "burn" ids. Here is an example, using InnoDB (other Engines may work differently):

```
CREATE TABLE Burn (
id SMALLINT UNSIGNED AUTO_INCREMENT NOT NULL,
name VARCHAR(99) NOT NULL,
PRIMARY KEY(id),
UNIQUE(name)
) ENGINE=InnoDB;

INSERT IGNORE INTO Burn (name) VALUES ('first'), ('second');
SELECT LAST_INSERT_ID(); -- 1
SELECT * FROM Burn ORDER BY id;
```

```
+-----+
|  1 | first |
|  2 | second|
+-----+

插入忽略到 Burn (name) 值 ('second'); -- 重复 '忽略', 但 id=3 被烧毁
选择 LAST_INSERT_ID();                -- 仍然是 "1" -- 在这种情况下不能信任
选择 * 从 Burn 按 id排序;

+-----+
|  1 | first |
|  2 | second|
+-----+

INSERT IGNORE INTO Burn (name) VALUES ('third');
SELECT LAST_INSERT_ID();                -- 现在是 "4"
SELECT * FROM Burn ORDER BY id;        -- 注意 id=3 被跳过了

+-----+
|  1 | first |
|  2 | second|
|  4 | third | -- 注意 id=3 已被“烧掉”
+-----+
```

可以大致这样理解：首先插入操作会查看可能插入多少行。然后从该表的自增值中获取相应数量的值。最后，插入这些行，按需使用这些id，并烧掉任何剩余的id。

唯一能恢复剩余id的情况是系统关闭并重启时。重启时，实际上执行了 MAX(id) 操作。这可能会重用被烧掉的id或被 DELETEs 删除最高id时释放的id。

本质上，任何类型的 INSERT（包括 REPLACE，即 DELETE + INSERT）都可能烧掉id。在InnoDB中，全局（非会话！）变量 innodb\_autoinc\_lock\_mode 可用于控制部分行为。

当将长字符串“规范化”为 AUTO INCREMENT id 时，烧掉id的情况很容易发生。这 could 导致你选择的 INT 类型溢出。

```
+-----+
|  1 | first |
|  2 | second|
+-----+

INSERT IGNORE INTO Burn (name) VALUES ('second'); -- dup 'IGNOREd', but id=3 is burned
SELECT LAST_INSERT_ID();                -- Still "1" -- can't trust in this situation
SELECT * FROM Burn ORDER BY id;

+-----+
|  1 | first |
|  2 | second|
+-----+

INSERT IGNORE INTO Burn (name) VALUES ('third');
SELECT LAST_INSERT_ID();                -- now "4"
SELECT * FROM Burn ORDER BY id;        -- note that id=3 was skipped over

+-----+
|  1 | first |
|  2 | second|
|  4 | third | -- notice that id=3 has been 'burned'
+-----+
```

Think of it (roughly) this way: First the insert looks to see how many rows *might* be inserted. Then grab that many values from the auto\_increment for that table. Finally, insert the rows, using ids as needed, and burning any left overs.

The only time the leftover are recoverable is if the system is shutdown and restarted. On restart, effectively MAX(id) is performed. This may reuse ids that were burned or that were freed up by DELETEs of the highest id(s).

Essentially any flavor of INSERT (including REPLACE, which is DELETE + INSERT) can burn ids. In InnoDB, the global (not session!) variable innodb\_autoinc\_lock\_mode can be used to control some of what is going on.

When "normalizing" long strings into an AUTO INCREMENT id, burning can easily happen. This *could* lead to overflowing the size of the INT you chose.

# 第11章：DELETE

参数	详情
LOW_PRIORITY	如果提供了LOW_PRIORITY，删除操作将被延迟，直到没有进程从表中读取数据为止
忽略表	如果提供了IGNORE，则删除过程中遇到的所有错误将被忽略 将要删除记录的表
WHERE 条件	必须满足的删除记录条件。如果未提供条件，则表中的所有记录将被删除
ORDER BY 表达式	如果提供了ORDER BY，记录将按照给定顺序删除
LIMIT	控制从表中删除的最大记录数。给定的number_rows将被删除。

## 第11.1节：多表删除

MySQL的DELETE语句可以使用JOIN结构，也可以指定要删除的表。这对于避免嵌套查询非常有用。给定以下模式：

```
创建表 people
(
    id int 主键,
    name varchar(100) 非空,
    gender char(1) 非空
);
插入 people (id,name,gender) 值
(1,'凯西','f'),(2,'约翰','m'),(3,'保罗','m'),(4,'金','f');

创建表 pets
(
    id int 自增 主键,
    ownerId int 非空,
    name varchar(100) 非空,
    color varchar(100) 非空
);
插入 pets(ownerId,name,color) 值
(1,'罗弗','米色'),(2,'巴布尔斯','紫色'),(3,'斑点','黑白'),
(1,'罗弗2','白色');
```

id	名字	性别
1	凯西	f
2	约翰	m
3	保罗	m
4	金	f

id	所有者ID	名称	颜色
1	1	罗弗	米色
2	2	泡泡	紫色
4	1	罗弗2	白色

如果我们想要删除保罗的宠物，语句

```
DELETE p2
FROM pets p2
WHERE p2.ownerId in (
    SELECT p1.id
    FROM people p1
    WHERE p1.name = 'Paul');
```

# Chapter 11: DELETE

Parameter	Details
LOW_PRIORITY	If <b>LOW_PRIORITY</b> is provided, the delete will be delayed until there are no processes reading from the table
IGNORE	If <b>IGNORE</b> is provided, all errors encountered during the delete are ignored
table	The table from which you are going to delete records
WHERE conditions	The conditions that must be met for the records to be deleted. If no conditions are provided, then all records from the table will be deleted
ORDER BY expression	If <b>ORDER BY</b> is provided, records will be deleted in the given order
LIMIT	It controls the maximum number of records to delete from the table. Given number_rows will be deleted.

## Section 11.1: Multi-Table Deletes

MySQL's **DELETE** statement can use the **JOIN** construct, allowing also to specify which tables to delete from. This is useful to avoid nested queries. Given the schema:

```
create table people
(
    id int primary key,
    name varchar(100) not null,
    gender char(1) not null
);
insert people (id,name,gender) values
(1,'Kathy','f'),(2,'John','m'),(3,'Paul','m'),(4,'Kim','f');

create table pets
(
    id int auto_increment primary key,
    ownerId int not null,
    name varchar(100) not null,
    color varchar(100) not null
);
insert pets(ownerId,name,color) values
(1,'Rover','beige'),(2,'Bubbles','purple'),(3,'Spot','black and white'),
(1,'Rover2','white');
```

id	name	gender
1	Kathy	f
2	John	m
3	Paul	m
4	Kim	f

id	ownerId	name	color
1	1	Rover	beige
2	2	Bubbles	purple
4	1	Rover2	white

If we want to remove Paul's pets, the statement

```
DELETE p2
FROM pets p2
WHERE p2.ownerId in (
    SELECT p1.id
    FROM people p1
    WHERE p1.name = 'Paul');
```



可以重写为：

```
DELETE p2    -- 仅删除 pets 表中的行
FROM people p1
JOIN pets p2
ON p2.ownerId = p1.id
WHERE p1.name = 'Paul';
```

已删除 1 行  
Spot 已从 Pets 表中删除

p1 和 p2 是表名的别名，特别适用于表名较长且便于阅读。

要同时删除该人和宠物：

```
DELETE p1, p2    -- 从两个表中删除行
FROM people p1
JOIN pets p2
ON p2.ownerId = p1.id
WHERE p1.name = 'Paul';
```

2 行已删除  
Spot 从 Pets 中被删除  
Paul 从 People 中被删除

外键

当 DELETE 语句涉及带有外键约束的表时，优化器可能会以不遵循关系的顺序处理表。例如，在 pets 的定义中添加外键时

```
ALTER TABLE pets ADD CONSTRAINT `fk_pets_2_people` FOREIGN KEY (ownerId) references people(id) ON DELETE CASCADE;
```

引擎可能会尝试先删除 people 中的条目，再删除 pets 中的条目，从而导致以下错误：

```
错误 1451 (23000): 无法删除或更新父行：外键约束失败
(`test`.`pets`, 约束 `pets_ibfk_1` 外键 (`ownerId`) 引用 `people` (`id`))
```

此情况下的解决方案是先从 people 中删除该行，并依赖 InnoDB 的 ON DELETE 功能来传播删除操作：

```
DELETE FROM people
WHERE name = 'Paul';
```

2 行已删除  
Paul 从 People 中被删除  
Spot 从 Pets 中被级联删除

另一种解决方案是暂时禁用外键检查：

```
SET foreign_key_checks = 0;
DELETE p1, p2 FROM people p1 JOIN pets p2 ON p2.ownerId = p1.id WHERE p1.name = 'Paul';
SET foreign_key_checks = 1;
```

can be rewritten as:

```
DELETE p2    -- remove only rows from pets
FROM people p1
JOIN pets p2
ON p2.ownerId = p1.id
WHERE p1.name = 'Paul';
```

1 row deleted  
Spot is deleted from Pets

p1 and p2 are aliases for the table names, especially useful for long table names and ease of readability.

To remove both the person and the pet:

```
DELETE p1, p2    -- remove rows from both tables
FROM people p1
JOIN pets p2
ON p2.ownerId = p1.id
WHERE p1.name = 'Paul';
```

2 rows deleted  
Spot is deleted from Pets  
Paul is deleted from People

foreign keys

When the DELETE statement involoes tables with a foreing key constrain the optimizer may process the tables in an order that does not follow the relationship. Adding for example a foreign key to the definition of pets

```
ALTER TABLE pets ADD CONSTRAINT `fk_pets_2_people` FOREIGN KEY (ownerId) references people(id) ON DELETE CASCADE;
```

the engine may try to delete the entries from people before pets, thus causing the following error:

```
ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key constraint fails
(`test`.`pets`, CONSTRAINT `pets_ibfk_1` FOREIGN KEY (`ownerId`) REFERENCES `people` (`id`))
```

The solution in this case is to delete the row from people and rely on InnoDB's ON DELETE capabilities to propagate the deletion:

```
DELETE FROM people
WHERE name = 'Paul';
```

2 rows deleted  
Paul is deleted from People  
Spot is deleted on cascade from Pets

Another solution is to temporarily disable the check on foreing keys:

```
SET foreign_key_checks = 0;
DELETE p1, p2 FROM people p1 JOIN pets p2 ON p2.ownerId = p1.id WHERE p1.name = 'Paul';
SET foreign_key_checks = 1;
```

## 第11.2节：DELETE 与 TRUNCATE

```
TRUNCATE tableName;
```

这将删除所有数据并重置AUTO\_INCREMENT索引。它比在大量数据集上使用DELETE FROM tableName快得多。在开发/测试过程中非常有用。

当你截断一个表时，SQL服务器并不删除数据，而是删除表并重新创建它，从而释放页面，因此在页面被覆盖之前，有机会恢复被截断的数据。（对于innodb\_file\_per\_table=OFF，空间不能立即回收。）

## 第11.3节：多表DELETE

MySQL允许指定从哪个表删除匹配的行

```
-- 仅删除员工
DELETE e
FROM 员工 e JOIN 部门 d ON e.department_id = d.department_id
WHERE d.name = '销售'

-- 删除员工和部门
DELETE e, d
FROM 员工 e JOIN 部门 d ON e.department_id = d.department_id
WHERE d.name = '销售'

-- 从所有表中删除（在此情况下与前面相同）
DELETE
FROM 员工 e JOIN 部门 d ON e.department_id = d.department_id
WHERE d.name = '销售'
```

## 第11.4节：基本删除

```
DELETE FROM `myTable` WHERE `someColumn` = 'something'
```

WHERE 子句是可选的，但如果没有它，所有行都会被删除。

## 第11.5节：带Where子句的删除

```
DELETE FROM `table_name` WHERE `field_one` = 'value_one'
```

这将删除表中所有field\_one列内容与'value\_one'匹配的行

WHERE 子句的工作方式与select相同，因此可以使用 >、<、<> 或 LIKE 等操作符。

**注意：** 在删除查询中必须使用条件子句（WHERE，LIKE）。如果不使用任何条件子句，则该表中的所有数据将被删除。

## 第11.6节：删除表中的所有行

```
DELETE FROM table_name ;
```

这将删除表中的所有内容，所有行。这是语法的最基本示例。它还表明DELETE 语句确实应谨慎使用，因为如果省略了WHERE 子句，可能会清空整个表。

## Section 11.2: DELETE vs TRUNCATE

```
TRUNCATE tableName;
```

This will [delete](#) all the data and reset **AUTO\_INCREMENT** index. It's much faster than **DELETE FROM** tableName on a huge dataset. It can be very useful during development/testing.

When you ***truncate*** a table SQL server doesn't delete the data, it drops the table and recreates it, thereby deallocating the pages so there is a chance to recover the truncated data before the pages where overwritten. (The space cannot immediately be recouped for innodb\_file\_per\_table=OFF.)

## Section 11.3: Multi-table DELETE

MySQL allows to specify from which table the matching rows must be deleted

```
-- remove only the employees
DELETE e
FROM Employees e JOIN Department d ON e.department_id = d.department_id
WHERE d.name = 'Sales'

-- remove employees and department
DELETE e, d
FROM Employees e JOIN Department d ON e.department_id = d.department_id
WHERE d.name = 'Sales'

-- remove from all tables (in this case same as previous)
DELETE
FROM Employees e JOIN Department d ON e.department_id = d.department_id
WHERE d.name = 'Sales'
```

## Section 11.4: Basic delete

```
DELETE FROM `myTable` WHERE `someColumn` = 'something'
```

The **WHERE** clause is optional but without it all rows are deleted.

## Section 11.5: Delete with Where clause

```
DELETE FROM `table_name` WHERE `field_one` = 'value_one'
```

This will delete all rows from the table where the contents of the field\_one for that row match 'value\_one'

The **WHERE** clause works in the same way as a select, so things like >, <, <> or **LIKE** can be used.

**Notice:** It is necessary to use conditional clauses (WHERE, LIKE) in delete query. If you do not use any conditional clauses then all data from that table will be deleted.

## Section 11.6: Delete all rows from a table

```
DELETE FROM table_name ;
```

This will delete everything, all rows from the table. It is the most basic example of the syntax. It also shows that **DELETE** statements should really be used with extra care as they may empty a table, if the **WHERE** clause is omitted.

## 第11.7节：限制删除数量

```
DELETE FROM `table_name` WHERE `field_one` = 'value_one' LIMIT 1
```

这与“带Where子句的删除”示例的工作方式相同，但一旦删除了限制数量的行，删除操作将停止。

如果像这样限制删除的行数，请注意它将删除第一个符合条件的行。由于结果如果没有明确排序，可能是无序返回的，因此它可能不是你预期的那一行。

## Section 11.7: LIMITing deletes

```
DELETE FROM `table_name` WHERE `field_one` = 'value_one' LIMIT 1
```

This works in the same way as the 'Delete with Where clause' example, but it will stop the deletion once the limited number of rows have been removed.

If you are limiting rows for deletion like this, be aware that it will delete the first row which matches the criteria. It might not be the one you would expect, as the results can come back unsorted if they are not explicitly ordered.

# 第12章：更新（UPDATE）

## 第12.1节：使用Join模式的更新

考虑一个名为questions\_mysql的生产表和一个名为iwtQuestions的表（导入的工作表），表示从LOAD DATA INFILE 导入的最后一批CSV数据。导入前工作表会被清空，数据被导入，这个过程这里不展示。

使用连接导入的工作表数据来更新我们的生产数据。

```
UPDATE questions_mysql q -- 我们的生产用真实表
join iwtQuestions i -- 导入的工作表
ON i.qId = q.qId
SET q.closeVotes = i.closeVotes,
    q.votes = i.votes,
    q.answers = i.answers,
    q.views = i.views;
```

别名q和i用于简化表引用。这有助于开发和可读性。

qId，主键，代表Stackoverflow问题ID。连接匹配的行会更新四个列。

## 第12.2节：基本更新

### 更新一行

```
UPDATE customers SET email='luke_smith@email.com' WHERE id=1
```

此查询将customers表中email的内容更新为字符串luke\_smith@email.com，条件是id的值等于1。数据库表的旧内容和新内容分别在下方的左侧和右侧示意：

customers				customers			
id	firstname	lastname	email	id	firstname	lastname	email
1	Luke	Smith	luke@example.com	1	Luke	Smith	luke_smith@email.com
2	Anna	Carey	anna@example.com	2	Anna	Carey	anna@example.com
3	Todd	Winters	todd@example.com	3	Todd	Winters	todd@example.com

### 更新所有行

```
UPDATE customers SET lastname='smith'
```

此查询更新customers表中每条记录的lastname内容。数据库表的旧内容和新内容分别在下方的左侧和右侧示意：

customers				customers			
id	firstname	lastname	email	id	firstname	lastname	email
1	Luke	Smith	luke@example.com	1	Luke	Smith	luke@example.com
2	Anna	Carey	anna@example.com	2	Anna	Smith	anna@example.com
3	Todd	Winters	todd@example.com	3	Todd	Smith	todd@example.com

注意：在UPDATE查询中必须使用条件子句（WHERE）。如果不使用任何条件

# Chapter 12: UPDATE

## Section 12.1: Update with Join Pattern

Consider a production table called questions\_mysql and a table iwtQuestions (imported worktable) representing the last batch of imported CSV data from a [LOAD DATA INFILE](#). The worktable is truncated before the import, the data is imported, and that process is not shown here.

Update our production data using a join to our imported worktable data.

```
UPDATE questions_mysql q -- our real table for production
join iwtQuestions i -- imported worktable
ON i.qId = q.qId
SET q.closeVotes = i.closeVotes,
    q.votes = i.votes,
    q.answers = i.answers,
    q.views = i.views;
```

Aliases q and i are used to abbreviate the table references. This eases development and readability.

qId, the Primary Key, represents the Stackoverflow question id. Four columns are updated for matching rows from the join.

## Section 12.2: Basic Update

### Updating one row

```
UPDATE customers SET email='luke_smith@email.com' WHERE id=1
```

This query updates the content of email in the customers table to the string luke\_smith@email.com where the value of id is equal to 1. The old and new contents of the database table are illustrated below on the left and right respectively:

customers				customers			
id	firstname	lastname	email	id	firstname	lastname	email
1	Luke	Smith	luke@example.com	1	Luke	Smith	luke_smith@email.com
2	Anna	Carey	anna@example.com	2	Anna	Carey	anna@example.com
3	Todd	Winters	todd@example.com	3	Todd	Winters	todd@example.com

### Updating all rows

```
UPDATE customers SET lastname='smith'
```

This query update the content of lastname for every entry in the customers table. The old and new contents of the database table are illustrated below on the left and right respectively:

customers				customers			
id	firstname	lastname	email	id	firstname	lastname	email
1	Luke	Smith	luke@example.com	1	Luke	Smith	luke@example.com
2	Anna	Carey	anna@example.com	2	Anna	Smith	anna@example.com
3	Todd	Winters	todd@example.com	3	Todd	Smith	todd@example.com

**Notice:** It is necessary to use conditional clauses (WHERE) in UPDATE query. If you do not use any conditional

子句，则该表该属性的所有记录都会被更新。在上述示例中，customers表中lastname的新值（Smith）被设置到所有行。

### 第12.3节：批量更新

当需要用不同的值更新多行时，使用批量更新会更快。

```
UPDATE people
SET name =
    (CASE id WHEN 1 THEN '卡尔'
           WHEN 2 THEN '汤姆'
           WHEN 3 THEN '玛丽'
        END)
WHERE id IN (1,2,3);
```

通过批量更新，只需发送一个查询到服务器，而不是为每一行更新发送一个查询。cases应包含在WHERE子句中查找的所有可能参数。

### 第12.4节：带有ORDER BY和LIMIT的UPDATE

如果在更新SQL语句中指定了ORDER BY子句，则按指定的顺序更新行。

如果在SQL语句中指定了LIMIT子句，则限制可更新的行数。  
如果未指定LIMIT子句，则没有限制。

ORDER BY和LIMIT不能用于多表更新。

MySQL中带有ORDER BY和LIMIT的UPDATE语法为，

```
UPDATE [ LOW_PRIORITY ] [ IGNORE ]
tableName
SET column1 = expression1,
    column2 = expression2,
    ...
[WHERE 条件]
[ORDER BY 表达式 [ ASC | DESC ]]
[LIMIT 行数];

---> 示例
UPDATE 员工 SET isConfirmed=1 ORDER BY 入职日期 LIMIT 10
```

在上述示例中，将根据员工的入职日期顺序更新10行。

### 第12.5节：多表UPDATE

在多表UPDATE中，会更新每个指定表中满足条件的行。即使某行多次满足条件，也只更新一次。

在多表UPDATE中，不能使用ORDER BY和LIMIT。

多表UPDATE的语法是，

```
UPDATE [LOW_PRIORITY] [IGNORE]
表1, 表2, ...
SET column1 = expression1,
```

clause then all records of that table's attribute will be updated. In above example new value (Smith) of lastname in customers table set to all rows.

### Section 12.3: Bulk UPDATE

When updating multiple rows with different values it is much quicker to use a bulk update.

```
UPDATE people
SET name =
    (CASE id WHEN 1 THEN 'Karl'
           WHEN 2 THEN 'Tom'
           WHEN 3 THEN 'Mary'
        END)
WHERE id IN (1,2,3);
```

By bulk updating only one query can be sent to the server instead of one query for each row to update. The cases should contain all possible parameters looked up in the WHERE clause.

### Section 12.4: UPDATE with ORDER BY and LIMIT

If the ORDER BY clause is specified in your update SQL statement, the rows are updated in the order that is specified.

If LIMIT clause is specified in your SQL statement, that places a limit on the number of rows that can be updated. There is no limit, if LIMIT clause not specified.

ORDER BY and LIMIT cannot be used for multi table update.

Syntax for the MySQL UPDATE with ORDER BY and LIMIT is,

```
UPDATE [ LOW_PRIORITY ] [ IGNORE ]
tableName
SET column1 = expression1,
    column2 = expression2,
    ...
[WHERE conditions]
[ORDER BY expression [ ASC | DESC ]]
[LIMIT row_count];

---> Example
UPDATE employees SET isConfirmed=1 ORDER BY joiningDate LIMIT 10
```

In the above example, 10 rows will be updated according to the order of employees joiningDate.

### Section 12.5: Multiple Table UPDATE

In multiple table UPDATE, it updates rows in each specified tables that satisfy the conditions. Each matching row is updated once, even if it matches the conditions multiple times.

In multiple table UPDATE, ORDER BY and LIMIT cannot be used.

Syntax for multi table UPDATE is,

```
UPDATE [LOW_PRIORITY] [IGNORE]
table1, table2, ...
SET column1 = expression1,
```



```
column2 = expression2,
...
[WHERE 条件]
```

例如，考虑两个表，products 和 salesOrders。如果我们减少已经下单的销售订单中某个产品的数量，那么我们也需要增加 products 表中该产品的库存数量。这可以通过如下单条 SQL 更新语句完成。

```
UPDATE products, salesOrders
  SET salesOrders.Quantity = salesOrders.Quantity - 5,
      products.availableStock = products.availableStock + 5
 WHERE products.productId = salesOrders.productId
  AND salesOrders.orderId = 100 AND salesOrders.productId = 20;
```

在上述示例中，数量“5”将从 salesOrders 表中减少，并根据 WHERE 条件在 products 表中增加相同数量。

```
column2 = expression2,
...
[WHERE conditions]
```

For example consider two tables, products and salesOrders. In case, we decrease the quantity of a particular product from the sales order which is placed already. Then we also need to increase that quantity in our stock column of products table. This can be done in single SQL update statement like below.

```
UPDATE products, salesOrders
  SET salesOrders.Quantity = salesOrders.Quantity - 5,
      products.availableStock = products.availableStock + 5
 WHERE products.productId = salesOrders.productId
  AND salesOrders.orderId = 100 AND salesOrders.productId = 20;
```

In the above example, quantity '5' will be reduced from the salesOrders table and the same will be increased in products table according to the WHERE conditions.

# 第13章：ORDER BY

## 第13.1节：上下文

SELECT 语句中的子句有特定的顺序：

```
SELECT ... FROM ... WHERE ... GROUP BY ... HAVING ...
ORDER BY ... -- 这里写
LIMIT ... OFFSET ...;

( SELECT ... ) UNION ( SELECT ... ) ORDER BY ... -- 用于对UNION结果排序。

SELECT ... GROUP_CONCAT(DISTINCT x ORDER BY ... SEPARATOR ...) ...

ALTER TABLE ... ORDER BY ... -- 可能仅对MyISAM有用；不适用于InnoDB
```

## 第13.2节：基础

ORDER BY x

x 可以是任何数据类型。

- NULL值 排在非NULL值之前。
- 默认是 ASC （从低到高）
- 字符串（**VARCHAR** 等）根据声明的**COLLATION**进行排序
- 枚举（ENUM）根据其字符串的声明顺序进行排序。

## 第13.3节：升序 / 降序

```
ORDER BY x ASC -- 与默认相同
ORDER BY x DESC -- 从高到低
ORDER BY lastname, firstname -- 典型的姓名排序；使用两列
ORDER BY submit_date DESC -- 最新的排在最前
ORDER BY submit_date DESC, id ASC -- 最新的排在最前，但完全指定排序顺序。
```

- **ASC** = 升序, **DESC** = 降序
- NULL值 即使在 **DESC** 中也排在最前。
- 在上述示例中，**INDEX(x)**, **INDEX(lastname, firstname)**, **INDEX(submit\_date)** 可能显著提升性能。

但是..... 混合使用 ASC 和 DESC，如最后一个示例，无法利用复合索引带来好处。也不会 **INDEX(submit\_date DESC, id ASC)** 有帮助 -- "DESC" 在 INDEX 声明中语法上被识别，但会被忽略。

## 第13.4节：一些技巧

```
ORDER BY FIND_IN_SET(card_type, "MASTER-CARD,VISA,DISCOVER") -- 将 'MASTER-CARD' 排在最前。
ORDER BY x IS NULL, x -- 按 `x` 排序，但将 `NULL` 放在最后。
```

### 自定义排序

```
SELECT * FROM some_table WHERE id IN (118, 17, 113, 23, 72)
ORDER BY FIELD(id, 118, 17, 113, 23, 72);
```

按指定的 id 顺序返回结果。

# Chapter 13: ORDER BY

## Section 13.1: Contexts

The clauses in a **SELECT** have a specific order:

```
SELECT ... FROM ... WHERE ... GROUP BY ... HAVING ...
ORDER BY ... -- goes here
LIMIT ... OFFSET ...;

( SELECT ... ) UNION ( SELECT ... ) ORDER BY ... -- for ordering the result of the UNION.

SELECT ... GROUP_CONCAT(DISTINCT x ORDER BY ... SEPARATOR ...) ...

ALTER TABLE ... ORDER BY ... -- probably useful only for MyISAM; not for InnoDB
```

## Section 13.2: Basic

ORDER BY x

x can be any datatype.

- NULLs precede non-NULLs.
- The default is **ASC** (lowest to highest)
- Strings (**VARCHAR**, etc) are ordered according the **COLLATION** of the declaration
- ENUMs are ordered by the declaration order of its strings.

## Section 13.3: ASCending / DESCending

```
ORDER BY x ASC -- same as default
ORDER BY x DESC -- highest to lowest
ORDER BY lastname, firstname -- typical name sorting; using two columns
ORDER BY submit_date DESC -- latest first
ORDER BY submit_date DESC, id ASC -- latest first, but fully specifying order.
```

- **ASC** = ASCENDING, **DESC** = DESCENDING
- NULLs come first even for **DESC**.
- In the above examples, **INDEX(x)**, **INDEX(lastname, firstname)**, **INDEX(submit\_date)** may significantly improve performance.

But... Mixing **ASC** and **DESC**, as in the last example, cannot use a composite index to benefit. Nor will **INDEX(submit\_date DESC, id ASC)** help -- "**DESC**" is recognized syntactically in the **INDEX** declaration, but ignored.

## Section 13.4: Some tricks

```
ORDER BY FIND_IN_SET(card_type, "MASTER-CARD,VISA,DISCOVER") -- sort 'MASTER-CARD' first.
ORDER BY x IS NULL, x -- order by `x`, but put `NULLs` last.
```

### Custom ordering

```
SELECT * FROM some_table WHERE id IN (118, 17, 113, 23, 72)
ORDER BY FIELD(id, 118, 17, 113, 23, 72);
```

Returns the result in the specified order of ids.

id ...  
118 ...  
17 ...  
113 ...  
23 ...  
72 ...

如果ID已经排序，只需检索行时，这非常有用。

id ...  
118 ...  
17 ...  
113 ...  
23 ...  
72 ...

Useful if the ids are already sorted and you just need to retrieve the rows.

# 第14章：分组（Group By）

参数	详细信息
表达式1, 表达式2, ... 表达式_n	未被聚合函数封装的表达式，必须包含在GROUP BY子句中。
聚合函数	诸如SUM、COUNT、MIN、MAX或AVG等函数。
表	您希望从中检索记录的表。FROM子句中必须至少列出一个表。
WHERE 条件	可选。必须满足的条件以选择记录。

## 第14.1节：使用HAVING的GROUP BY

```
SELECT 部门, COUNT(*) AS "Man_Power"
FROM 员工
GROUP BY 部门
HAVING COUNT(*) >= 10;
```

使用 GROUP BY ... HAVING 来过滤聚合记录类似于使用 SELECT ... WHERE 来过滤单个记录。

你也可以说 HAVING Man\_Power >= 10，因为 HAVING 支持“别名”。

## 第14.2节：使用 Group Concat 的分组

Group Concat 在 MySQL 中用于获取每列多个结果的表达式的连接值。  
意思是，对于一列有多行需要选回，如 姓名(1):分数(\*)

姓名 分数  
Adam A+  
Adam A-  
Adam B  
Adam C+  
Bill D-  
约翰A-

SELECT 姓名, GROUP\_CONCAT(成绩 ORDER BY 成绩 降序 SEPARATOR ' ') AS 等级  
FROM 成绩表  
GROUP BY 姓名

结果：

+-----+-----+  
| 姓名 | 等级 |  
+-----+-----+  
亚当	C+ B A- A+
比尔	D-
约翰	A-
+-----+-----+

## 第14.3节：使用MIN函数的分组

假设有一个员工表，每行代表一名员工，包含姓名、部门和薪水。

# Chapter 14: Group By

Parameter	DETAILS
expression1, expression2, ... expression_n	The expressions that are not encapsulated within an aggregate function and must be included in the GROUP BY clause.
aggregate_function	A function such as SUM, COUNT, MIN, MAX, or AVG functions.
tables	he tables that you wish to retrieve records from. There must be at least one table listed in the FROM clause.
WHERE conditions	Optional. The conditions that must be met for the records to be selected.

## Section 14.1: GROUP BY using HAVING

```
SELECT department, COUNT(*) AS "Man_Power"
FROM employees
GROUP BY department
HAVING COUNT(*) >= 10;
```

Using GROUP BY ... HAVING to filter aggregate records is analogous to using SELECT ... WHERE to filter individual records.

You could also say HAVING Man\_Power >= 10 since HAVING understands "aliases".

## Section 14.2: Group By using Group Concat

Group Concat is used in MySQL to get concatenated values of expressions with more than one result per column.  
Meaning, there are many rows to be selected back for one column such as Name(1):Score(\*)

Name Score  
Adam A+  
Adam A-  
Adam B  
Adam C+  
Bill D-  
John A-

SELECT Name, GROUP\_CONCAT(Score ORDER BY Score desc SEPARATOR ' ') AS Grades  
FROM Grade  
GROUP BY Name

Results:

+-----+-----+  
| Name | Grades |  
+-----+-----+  
Adam	C+ B A- A+
Bill	D-
John	A-
+-----+-----+

## Section 14.3: Group By Using MIN function

Assume a table of employees in which each row is an employee who has a name, a department, and a salary.

```
SELECT 部门, MIN(薪水) AS "最低薪水"
FROM 员工表
GROUP BY 部门;
```

这将告诉你哪个部门有薪水最低的员工，以及该薪水是多少。查找每个部门薪水最低员工的姓名是另一个问题，超出本示例范围。参见“分组最大值”。

## 第14.4节：带聚合函数的GROUP BY

订单表

订单号	客户编号	客户名	总计	商品数
1	1	鲍勃	1300	10
2	3	弗雷德	500	2
3	5	特丝	2500	8
4	1	鲍勃	300	6
5	2	卡莉	800	3
6	2	卡莉	1000	12
7	3	弗雷德	100	1
8	5	特丝	11500	50
9	4	珍妮	200	2
10	1	鲍勃	500	15

- 计数

返回满足WHERE子句中特定条件的行数。

例如：每个客户的订单数量。

```
SELECT 客户, COUNT(*) 作为 订单数
FROM 订单
GROUP BY 客户
ORDER BY 客户
```

结果：

客户	订单数
鲍勃	3
卡莉	2
弗雷德	2
珍妮	1
特丝	2

- 求和

返回所选列的总和。

例如：每个客户的总计和项目之和。

```
SELECT department, MIN(salary) AS "Lowest salary"
FROM employees
GROUP BY department;
```

This would tell you which department contains the employee with the lowest salary, and what that salary is. Finding the name of the employee with the lowest salary in each department is a different problem, beyond the scope of this Example. See "groupwise max".

## Section 14.4: GROUP BY with AGGREGATE functions

Table ORDERS

orderid	customerid	customer	total	items
1	1	Bob	1300	10
2	3	Fred	500	2
3	5	Tess	2500	8
4	1	Bob	300	6
5	2	Carly	800	3
6	2	Carly	1000	12
7	3	Fred	100	1
8	5	Tess	11500	50
9	4	Jenny	200	2
10	1	Bob	500	15

- COUNT

Return the **number of rows** that satisfy a specific criteria in **WHERE** clause.

E.g.: Number of orders for each customer.

```
SELECT customer, COUNT(*) as orders
FROM orders
GROUP BY customer
ORDER BY customer
```

Result:

customer	orders
Bob	3
Carly	2
Fred	2
Jenny	1
Tess	2

- SUM

Return the **sum** of the selected column.

E.g.: Sum of the total and items for each customer.



```
SELECT customer, SUM(total) as sum_total, SUM(items) as sum_items
FROM orders
GROUP BY 客户
ORDER BY 客户
```

结果：

customer	sum_total	sum_items
Bob	2100	31
Carly	1800	15
Fred	600	3
Jenny	200	2
Tess	14000	58

• AVG

返回某列数值的平均值。

例如：每个客户的平均订单金额。

```
SELECT customer, AVG(total) as avg_total
FROM orders
GROUP BY 客户
ORDER BY 客户
```

结果：

customer	avg_total
Bob	700
Carly	900
Fred	300
Jenny	200
Tess	7000

• MAX

返回某列或表达式的最大值。

例如：每个客户的最高订单总额。

```
SELECT customer, MAX(total) as max_total
FROM orders
GROUP BY 客户
ORDER BY 客户
```

结果：

customer	max_total
----------	-----------

```
SELECT customer, SUM(total) as sum_total, SUM(items) as sum_items
FROM orders
GROUP BY customer
ORDER BY customer
```

Result:

customer	sum_total	sum_items
Bob	2100	31
Carly	1800	15
Fred	600	3
Jenny	200	2
Tess	14000	58

• AVG

Return the **average** value of a column of numeric value.

E.g.: Average order value for each customers.

```
SELECT customer, AVG(total) as avg_total
FROM orders
GROUP BY customer
ORDER BY customer
```

Result:

customer	avg_total
Bob	700
Carly	900
Fred	300
Jenny	200
Tess	7000

• MAX

Return the **highest** value of a certain column or expression.

E.g.: Highest order total for each customers.

```
SELECT customer, MAX(total) as max_total
FROM orders
GROUP BY customer
ORDER BY customer
```

Result:

customer	max_total
----------	-----------

Bob		1300	
Carly		1000	
Fred		500	
Jenny		200	
Tess		11500	
+-----+-----+			

• MIN

返回某列或表达式的最低值。

例如：每个客户的最低订单总额。

```
SELECT customer, MIN(total) as min_total
FROM orders
GROUP BY 客户
ORDER BY 客户
```

结果：

+-----+-----+			
customer		min_total	
+-----+-----+			
Bob		300	
Carly		800	
Fred		100	
Jenny		200	
Tess		2500	
+-----+-----+			

Bob		1300	
Carly		1000	
Fred		500	
Jenny		200	
Tess		11500	
+-----+-----+			

• MIN

Return the **lowest** value of a certain column or expression.

E.g.: Lowest order total for each customers.

```
SELECT customer, MIN(total) as min_total
FROM orders
GROUP BY customer
ORDER BY customer
```

Result:

+-----+-----+			
customer		min_total	
+-----+-----+			
Bob		300	
Carly		800	
Fred		100	
Jenny		200	
Tess		2500	
+-----+-----+			

# 第15章：错误1055：ONLY\_FULL\_GROUP\_BY：某些内容未包含在GROUP BY子句中.....

最近，MySQL服务器的新版本开始对以前能正常工作的查询生成1055错误。本文解释了这些错误。MySQL团队一直在努力废除对GROUP BY的非标准扩展，或者至少让编写查询的开发者更难因此而出错。

## 第15.1节：误用GROUP BY导致不可预测的结果：墨菲定律

```
SELECT item.item_id, uses.category, /* 非标准 */
       COUNT(*) number_of_uses
FROM item
JOIN uses ON item.item_id, uses.item_id
GROUP BY item.item_id
```

将显示名为item的表中的行，并显示名为uses的表中相关行的计数。它还会显示名为uses.category的列的值。

该查询在MySQL中有效（在ONLY\_FULL\_GROUP\_BY标志出现之前）。它使用了MySQL对GROUP BY的非标准扩展。

但该查询存在问题：如果uses表中有多行匹配JOIN子句中的ON条件，MySQL只会返回这些行中某一行的category列。是哪一行？查询的编写者和应用程序的用户事先无法得知。正式来说，这是不可预测的：MySQL可以返回它想要的任何值。

不可预测类似于随机，但有一个显著区别。人们可能期望随机选择会随着时间变化。因此，如果选择是随机的，你可能在调试或测试时发现它。不可预测的结果更糟：MySQL每次使用该查询时返回相同的结果，直到它不再如此。有时是MySQL服务器的新版本导致结果不同，有时是表的增长引起问题。

出错的事情总会发生，而且是在你意想不到的时候。这就是所谓的墨菲定律。

MySQL团队一直在努力让开发者更难犯这种错误。5.7版本序列中的较新MySQL版本有一个名为ONLY\_FULL\_GROUP\_BY的sql\_mode标志。当该标志被设置时，MySQL服务器会返回1055错误并拒绝执行此类查询。

## 第15.2节：误用GROUP BY与SELECT \*，以及如何修复它

有时查询看起来像这样，SELECT子句中带有\*。

```
SELECT item.*, /* 非标准 */
       COUNT(*) number_of_uses
FROM item
JOIN uses ON item.item_id, uses.item_id
GROUP BY item.item_id
```

这样的查询需要重构以符合ONLY\_FULL\_GROUP\_BY标准。

为此，我们需要一个子查询，正确使用GROUP BY来返回每个item\_id的number\_of\_uses值。

# Chapter 15: Error 1055: ONLY\_FULL\_GROUP\_BY: something is not in GROUP BY clause ...

Recently, new versions of MySQL servers have begun to generate 1055 errors for queries that used to work. This topic explains those errors. The MySQL team has been working to retire the nonstandard extension to **GROUP BY**, or at least to make it harder for query writing developers to be burned by it.

## Section 15.1: Misusing GROUP BY to return unpredictable results: Murphy's Law

```
SELECT item.item_id, uses.category, /* nonstandard */
       COUNT(*) number_of_uses
FROM item
JOIN uses ON item.item_id, uses.item_id
GROUP BY item.item_id
```

will show the rows in a table called item, and show the count of related rows in a table called uses. It will also show the value of a column called uses.category.

This query works in MySQL (before the ONLY\_FULL\_GROUP\_BY flag appeared). It uses [MySQL's nonstandard extension to GROUP BY](#).

But the query has a problem: if several rows in the uses table match the ON condition in the **JOIN** clause, MySQL returns the category column from just one of those rows. Which row? The writer of the query, and the user of the application, doesn't get to know that in advance. Formally speaking, it's *unpredictable*: MySQL can return any value it wants.

*Unpredictable* is like *random*, with one significant difference. One might expect a *random* choice to change from time to time. Therefore, if a choice were random, you might detect it during debugging or testing. The *unpredictable* result is worse: MySQL returns the same result each time you use the query, *until it doesn't*. Sometimes it's a new version of the MySQL server that causes a different result. Sometimes it's a growing table causing the problem. What can go wrong, will go wrong, and when you don't expect it. That's called [Murphy's Law](#).

The MySQL team has been working to make it harder for developers to make this mistake. Newer versions of MySQL in the 5.7 sequence have a sql\_mode flag called ONLY\_FULL\_GROUP\_BY. When that flag is set, the MySQL server returns the 1055 error and refuses to run this kind of query.

## Section 15.2: Misusing GROUP BY with SELECT \*, and how to fix it

Sometimes a query looks like this, with a \* in the **SELECT** clause.

```
SELECT item.*, /* nonstandard */
       COUNT(*) number_of_uses
FROM item
JOIN uses ON item.item_id, uses.item_id
GROUP BY item.item_id
```

Such a query needs to be refactored to comply with the ONLY\_FULL\_GROUP\_BY standard.

To do this, we need a subquery that uses **GROUP BY** correctly to return the number\_of\_uses value for each item\_id.

这个子查询简洁明了，因为它只需要查看uses表。

```
SELECT item_id, COUNT(*) number_of_uses
FROM uses
GROUP BY item_id
```

然后，我们可以将该子查询与item表连接。

```
SELECT item.*, usecount.number_of_uses
FROM item
JOIN (
    SELECT item_id, COUNT(*) number_of_uses
    FROM uses
    GROUP BY item_id
) usecount ON item.item_id = usecount.item_id
```

这使得GROUP BY子句既简单又正确，同时也允许我们使用\*通配符。

注意：尽管如此，明智的开发者通常避免使用\*通配符。通常最好在查询中列出你想要的列。

### 第15.3节：ANY\_VALUE()

```
SELECT item.item_id, ANY_VALUE(uses.tag) tag,
COUNT(*) number_of_uses
FROM item
JOIN uses ON item.item_id, uses.item_id
GROUP BY item.item_id
```

显示名为item的表中的行、相关行的计数，以及名为

你可以将ANY\_VALUE()函数看作是一种奇怪的聚合函数。它不是返回计数、求和或最大值，而是指示MySQL服务器从相关组中任意选择一个值。这是一种绕过错误1055的方法。

在生产应用中使用ANY\_VALUE()时要小心。

它实际上应该叫做SURPRISE\_ME()。它返回GROUP BY分组中某一行的值。返回哪一行的值是不可确定的。这意味着完全由MySQL服务器决定。正式来说，它返回一个不可预测的值。

服务器并不是选择一个随机值，情况比这更糟。它每次运行查询时返回相同的值，直到不再返回相同的值。它可能会变化，也可能不会，取决于表的大小变化，服务器的内存多少，服务器版本的变化，或者火星逆行（无论那是什么意思），甚至可能毫无理由地变化。

你已被警告。

### 第15.4节：使用和误用GROUP BY

```
SELECT item.item_id, item.name, /* 非SQL-92标准 */
COUNT(*) number_of_uses
FROM item
JOIN uses ON item.item_id, uses.item_id
```

This subquery is short and sweet, because it only needs to look at the uses table.

```
SELECT item_id, COUNT(*) number_of_uses
FROM uses
GROUP BY item_id
```

Then, we can join that subquery with the item table.

```
SELECT item.*, usecount.number_of_uses
FROM item
JOIN (
    SELECT item_id, COUNT(*) number_of_uses
    FROM uses
    GROUP BY item_id
) usecount ON item.item_id = usecount.item_id
```

This allows the GROUP BY clause to be simple and correct, and also allows us to use the \* specifier.

Note: nevertheless, wise developers avoid using the \* specifier in any case. It's usually better to list the columns you want in a query.

### Section 15.3: ANY\_VALUE()

```
SELECT item.item_id, ANY_VALUE(uses.tag) tag,
COUNT(*) number_of_uses
FROM item
JOIN uses ON item.item_id, uses.item_id
GROUP BY item.item_id
```

shows the rows in a table called item, the count of related rows, and one of the values in the related table called uses.

You can think of this ANY\_VALUE() function as a strange a kind of aggregate function. Instead of returning a count, sum, or maximum, it instructs the MySQL server to choose, arbitrarily, one value from the group in question. It's a way of working around Error 1055.

Be careful when using ANY\_VALUE() in queries in production applications.

It really should be called SURPRISE\_ME(). It returns the value of some row in the GROUP BY group. Which row it returns is indeterminate. That means it's entirely up to the MySQL server. Formally, it returns an unpredictable value.

The server doesn't choose a random value, it's worse than that. It returns the same value every time you run the query, until it doesn't. It can change, or not, when a table grows or shrinks, or when the server has more or less RAM, or when the server version changes, or when Mars is in retrograde (whatever that means), or for no reason at all.

You have been warned.

### Section 15.4: Using and misusing GROUP BY

```
SELECT item.item_id, item.name, /* not SQL-92 */
COUNT(*) number_of_uses
FROM item
JOIN uses ON item.item_id, uses.item_id
```

```
GROUP BY item.item_id
```

将显示名为item的表中的行，并显示名为uses的表中相关行的计数。这工作良好，但不幸的是它不是标准的SQL-92。

为什么呢？因为SELECT子句（以及ORDER BY子句）在GROUP BY查询中必须包含的列是

1. 在GROUP BY子句中提到，或者
2. 聚合函数，如COUNT()、MIN()等。

此示例的SELECT子句提到了item.name，这一列不符合上述任一条件。如果SQL模式包含ONLY\_FULL\_GROUP\_BY，MySQL 5.6及更早版本将拒绝此查询。

通过更改GROUP BY子句，可以使此示例查询符合SQL-92标准，如下所示。

```
SELECT item.item_id, item.name,
       COUNT(*) number_of_uses
FROM item
JOIN uses ON item.item_id, uses.item_id
GROUP BY item.item_id, item.name
```

后来的SQL-99标准允许SELECT语句在组键中省略未聚合的列，前提是数据库管理系统能够证明这些列与组键列之间存在函数依赖关系。因为item.name函数依赖于item.item\_id，初始示例是有效的SQL-99。MySQL在5.7版本中引入了函数依赖证明器。原始示例在ONLY\_FULL\_GROUP\_BY模式下也能正常工作。

```
GROUP BY item.item_id
```

will show the rows in a table called item, and show the count of related rows in a table called uses. This works well, but unfortunately it's not standard SQL-92.

Why not? because the **SELECT** clause (and the **ORDER BY** clause) in **GROUP BY** queries must contain columns that are

1. mentioned in the **GROUP BY** clause, or
2. aggregate functions such as **COUNT()**, **MIN()**, and the like.

This example's **SELECT** clause mentions item.name, a column that does not meet either of those criteria. MySQL 5.6 and earlier will reject this query if the SQL mode contains ONLY\_FULL\_GROUP\_BY.

This example query can be made to comply with the SQL-92 standard by changing the **GROUP BY** clause, like this.

```
SELECT item.item_id, item.name,
       COUNT(*) number_of_uses
FROM item
JOIN uses ON item.item_id, uses.item_id
GROUP BY item.item_id, item.name
```

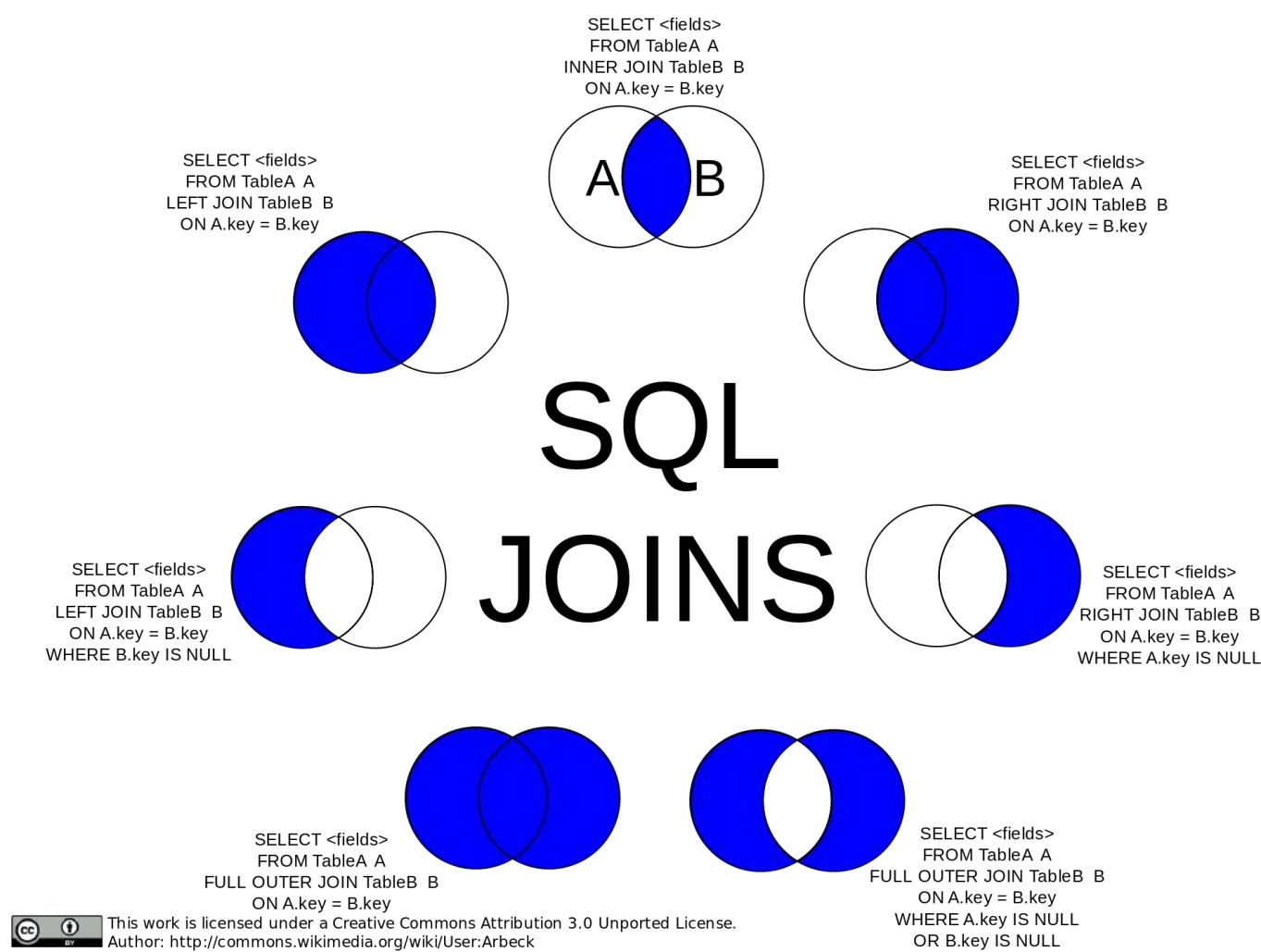
The later SQL-99 standard allows a **SELECT** statement to omit unaggregated columns from the group key if the DBMS can prove a functional dependence between them and the group key columns. Because item.name is functionally dependent on item.item\_id, the initial example is valid SQL-99. MySQL gained a [functional dependence prover](#) in version 5.7. The original example works under ONLY\_FULL\_GROUP\_BY.



# 第16章：连接

## 第16.1节：连接的可视化

如果你是一个视觉导向的人，这个维恩图可能会帮助你理解MySQL中存在的不同类型的JOIN。



## 第16.2节：带子查询的JOIN（“派生”表）

```
SELECT x, ...
FROM ( SELECT y, ... FROM ... ) AS a
JOIN tbl ON tbl.x = a.y
WHERE ...
```

这将会把子查询计算成一个临时表，然后将其JOIN到 tbl上。

在5.6版本之前，临时表上不能有索引。因此，这可能非常低效：

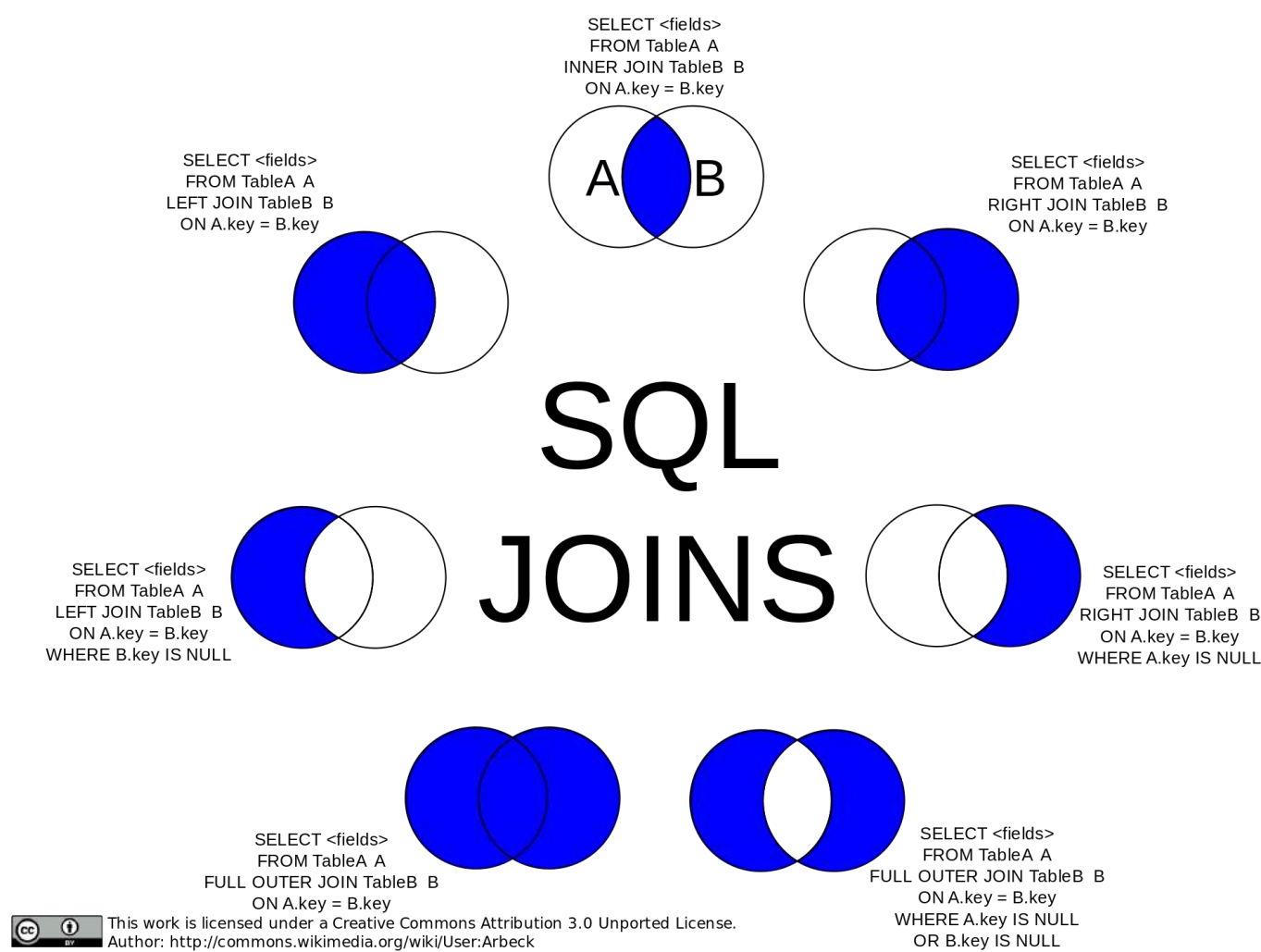
```
SELECT ...
FROM ( SELECT y, ... FROM ... ) AS a
JOIN ( SELECT x, ... FROM ... ) AS b ON b.x = a.y
WHERE ...
```

在5.6版本中，优化器会找出最佳索引并动态创建它。（这有一定的开销，所以仍然不是“完美”的。）

# Chapter 16: Joins

## Section 16.1: Joins visualized

If you are a visually oriented person, this Venn diagram may help you understand the different types of **JOINS** that exist within MySQL.



## Section 16.2: JOIN with subquery ("Derived" table)

```
SELECT x, ...
FROM ( SELECT y, ... FROM ... ) AS a
JOIN tbl ON tbl.x = a.y
WHERE ...
```

This will evaluate the subquery into a temp table, then **JOIN** that to tbl.

Prior to 5.6, there could not be an index on the temp table. So, this was potentially very inefficient:

```
SELECT ...
FROM ( SELECT y, ... FROM ... ) AS a
JOIN ( SELECT x, ... FROM ... ) AS b ON b.x = a.y
WHERE ...
```

With 5.6, the optimizer figures out the best index and creates it on the fly. (This has some overhead, so it is still not 'perfect'.)

另一个常见的范例是使用子查询来初始化某些内容：

```
查询
@n := @n + 1,
...
FROM ( SELECT @n := 0 ) AS initialize
JOIN the_real_table
ORDER BY ...
```

(注意：这在技术上是一个CROSS JOIN（笛卡尔积），因为缺少ON条件。但由于子查询只返回一行，需要与 the\_real\_table中的n行匹配，因此效率较高。)

## 第16.3节：全外连接

MySQL不支持FULL OUTER JOIN，但有方法可以模拟实现。

### 设置数据

```
-- -----
-- `owners`表结构
-- -----

如果存在则删除表`owners`；
创建表`owners` (
`owner_id` int(11) 非空 自动递增,
`owner` varchar(30) 默认 NULL,
PRIMARY KEY (`owner_id`)
) ENGINE=InnoDB AUTO_INCREMENT=10 DEFAULT CHARSET=latin1;

-- -----
-- 业主记录
-- -----

INSERT INTO `owners` VALUES ('1', 'Ben');
INSERT INTO `owners` VALUES ('2', 'Jim');
INSERT INTO `owners` VALUES ('3', 'Harry');
INSERT INTO `owners` VALUES ('6', 'John');
INSERT INTO `owners` VALUES ('9', 'Ellie');

-- -----
-- `tools` 表结构
-- -----

DROP TABLE IF EXISTS `tools`;
CREATE TABLE `tools` (
`tool_id` int(11) NOT NULL AUTO_INCREMENT,
`tool` varchar(30) DEFAULT NULL,
`owner_id` int(11) DEFAULT NULL,
PRIMARY KEY (`tool_id`)
) ENGINE=InnoDB AUTO_INCREMENT=11 DEFAULT CHARSET=latin1;

-- -----
-- 工具记录
-- -----

INSERT INTO `tools` VALUES ('1', '锤子', '9');
INSERT INTO `tools` VALUES ('2', '钳子', '1');
INSERT INTO `tools` VALUES ('3', '刀', '1');
INSERT INTO `tools` VALUES ('4', '凿子', '2');
INSERT INTO `tools` VALUES ('5', '钢锯', '1');
INSERT INTO `tools` VALUES ('6', '水平尺', null);
INSERT INTO `tools` VALUES ('7', '扳手', null);
INSERT INTO `tools` VALUES ('8', '卷尺', '9');
INSERT INTO `tools` VALUES ('9', '螺丝刀', null);
INSERT INTO `tools` VALUES ('10', '夹具', null);
```

Another common paradigm is to have a subquery to initialize something:

```
SELECT
    @n := @n + 1,
    ...
FROM ( SELECT @n := 0 ) AS initialize
JOIN the_real_table
ORDER BY ...
```

(Note: this is technically a **CROSS JOIN** (Cartesian product), as indicated by the lack of ON. However it is efficient because the subquery returns only one row that has to be matched to the n rows in the\_real\_table.)

## Section 16.3: Full Outer Join

MySQL does not support the **FULL OUTER JOIN**, but there are ways to emulate one.

### Setting up the data

```
-- -----
-- Table structure for `owners`
-- -----

DROP TABLE IF EXISTS `owners`;
CREATE TABLE `owners` (
`owner_id` int(11) NOT NULL AUTO_INCREMENT,
`owner` varchar(30) DEFAULT NULL,
PRIMARY KEY (`owner_id`)
) ENGINE=InnoDB AUTO_INCREMENT=10 DEFAULT CHARSET=latin1;

-- -----
-- Records of owners
-- -----

INSERT INTO `owners` VALUES ('1', 'Ben');
INSERT INTO `owners` VALUES ('2', 'Jim');
INSERT INTO `owners` VALUES ('3', 'Harry');
INSERT INTO `owners` VALUES ('6', 'John');
INSERT INTO `owners` VALUES ('9', 'Ellie');

-- -----
-- Table structure for `tools`
-- -----

DROP TABLE IF EXISTS `tools`;
CREATE TABLE `tools` (
`tool_id` int(11) NOT NULL AUTO_INCREMENT,
`tool` varchar(30) DEFAULT NULL,
`owner_id` int(11) DEFAULT NULL,
PRIMARY KEY (`tool_id`)
) ENGINE=InnoDB AUTO_INCREMENT=11 DEFAULT CHARSET=latin1;

-- -----
-- Records of tools
-- -----

INSERT INTO `tools` VALUES ('1', 'Hammer', '9');
INSERT INTO `tools` VALUES ('2', 'Pliers', '1');
INSERT INTO `tools` VALUES ('3', 'Knife', '1');
INSERT INTO `tools` VALUES ('4', 'Chisel', '2');
INSERT INTO `tools` VALUES ('5', 'Hacksaw', '1');
INSERT INTO `tools` VALUES ('6', 'Level', null);
INSERT INTO `tools` VALUES ('7', 'Wrench', null);
INSERT INTO `tools` VALUES ('8', 'Tape Measure', '9');
INSERT INTO `tools` VALUES ('9', 'Screwdriver', null);
INSERT INTO `tools` VALUES ('10', 'Clamp', null);
```

我们想看到什么？

我们想得到一个列表，显示谁拥有哪些工具，以及哪些工具可能没有主人。

查询语句

为此，我们可以使用UNION将两个查询合并。在第一个查询中，我们通过LEFT JOIN将工具与所有者连接。这会将所有所有者添加到结果集中，无论他们是否实际拥有工具。

在第二个查询中，我们使用RIGHT JOIN将工具连接到所有者。这样我们就能在结果集中获得所有工具，如果某个工具没有主人，其所有者列将显示为NULL。通过添加一个WHERE子句，筛选owners.owner\_idISNULL，我们定义结果为那些尚未被第一个查询返回的数据集，因为我们只查看右连接表中的数据。

由于我们使用了UNION ALL，第二个查询的结果集将附加到第一个查询的结果集后面。

```
SELECT `owners`.`owner`, tools.tool
FROM `owners`
LEFT JOIN `tools` ON `owners`.`owner_id` = `tools`.`owner_id`
UNION ALL
SELECT `owners`.`owner`, tools.tool
FROM `owners`
RIGHT JOIN `tools` ON `owners`.`owner_id` = `tools`.`owner_id`
WHERE `owners`.`owner_id` IS NULL;
```

owner	tool
Ben	Pliers
Ben	Knife
Ben	Hacksaw
Jim	Chisel
Harry	NULL
John	NULL
Ellie	Hammer
Ellie	Tape Measure
NULL	Level
NULL	Wrench
NULL	Screwdriver
NULL	Clamp

12 rows in set (0.00 sec)

第16.4节：检索有订单的客户——主题的变体

这将获取所有客户的所有订单：

```
SELECT c.CustomerName, o.OrderID
FROM Customers AS c
INNER JOIN Orders AS o
ON c.CustomerID = o.CustomerID
ORDER BY c.CustomerName, o.OrderID;
```

这将统计每个客户的订单数量：

What do we want to see?

We want to get a list, in which we see who owns which tools, and which tools might not have an owner.

The queries

To accomplish this, we can combine two queries by using UNION. In this first query we are joining the tools on the owners by using a LEFT JOIN. This will add all of our owners to our resultset, doesn't matter if they actually own tools.

In the second query we are using a RIGHT JOIN to join the tools onto the owners. This way we manage to get all the tools in our resultset, if they are owned by no one their owner column will simply contain NULL. By adding a WHERE-clause which is filtering by owners.owner\_id IS NULL we are defining the result as those datasets, which have not already been returned by the first query, as we are only looking for the data in the right joined table.

Since we are using UNION ALL the resultset of the second query will be attached to the first queries resultset.

```
SELECT `owners`.`owner`, tools.tool
FROM `owners`
LEFT JOIN `tools` ON `owners`.`owner_id` = `tools`.`owner_id`
UNION ALL
SELECT `owners`.`owner`, tools.tool
FROM `owners`
RIGHT JOIN `tools` ON `owners`.`owner_id` = `tools`.`owner_id`
WHERE `owners`.`owner_id` IS NULL;
```

owner	tool
Ben	Pliers
Ben	Knife
Ben	Hacksaw
Jim	Chisel
Harry	NULL
John	NULL
Ellie	Hammer
Ellie	Tape Measure
NULL	Level
NULL	Wrench
NULL	Screwdriver
NULL	Clamp

12 rows in set (0.00 sec)

Section 16.4: Retrieve customers with orders -- variations on a theme

This will get all the orders for all customers:

```
SELECT c.CustomerName, o.OrderID
FROM Customers AS c
INNER JOIN Orders AS o
ON c.CustomerID = o.CustomerID
ORDER BY c.CustomerName, o.OrderID;
```

This will count the number of orders for each customer:

```
SELECT c.CustomerName, COUNT(*) AS 'Order Count'
FROM Customers AS c
INNER JOIN Orders AS o
ON c.CustomerID = o.CustomerID
GROUP BY c.CustomerID;
ORDER BY c.CustomerName;
```

同样是计数，但可能更快：

```
SELECT c.CustomerName,
( SELECT COUNT(*) FROM Orders WHERE CustomerID = c.CustomerID ) AS 'Order Count'
FROM Customers AS c
按 c.CustomerName排序；
```

仅列出有订单的客户。

```
SELECT c.CustomerName,
FROM Customers AS c
WHERE EXISTS ( SELECT * FROM Orders WHERE CustomerID = c.CustomerID )
ORDER BY c.CustomerName;
```

## 第16.5节：连接示例

创建数据库表的查询

```
CREATE TABLE `user` (
`id` smallint(5) unsigned NOT NULL AUTO_INCREMENT,
`name` varchar(30) NOT NULL,
`course` smallint(5) unsigned DEFAULT NULL,
PRIMARY KEY (`id`)
) ENGINE=InnoDB;

CREATE TABLE `course` (
`id` smallint(5) unsigned NOT NULL AUTO_INCREMENT,
`name` varchar(50) NOT NULL,
PRIMARY KEY (`id`)
) ENGINE=InnoDB;
```

由于我们使用的是 InnoDB 表，并且知道 user.course 和 course.id 是相关的，我们可以指定一个外键关系：

```
ALTER TABLE `user`
ADD CONSTRAINT `FK_course`
FOREIGN KEY (`course`) REFERENCES `course` (`id`)
ON UPDATE CASCADE;
```

连接查询（内连接）

```
SELECT user.name, course.name
FROM `user`
INNER JOIN `course` on user.course = course.id;
```

```
SELECT c.CustomerName, COUNT(*) AS 'Order Count'
FROM Customers AS c
INNER JOIN Orders AS o
ON c.CustomerID = o.CustomerID
GROUP BY c.CustomerID;
ORDER BY c.CustomerName;
```

Also, counts, but probably faster:

```
SELECT c.CustomerName,
( SELECT COUNT(*) FROM Orders WHERE CustomerID = c.CustomerID ) AS 'Order Count'
FROM Customers AS c
ORDER BY c.CustomerName;
```

List only the customer with orders.

```
SELECT c.CustomerName,
FROM Customers AS c
WHERE EXISTS ( SELECT * FROM Orders WHERE CustomerID = c.CustomerID )
ORDER BY c.CustomerName;
```

## Section 16.5: Joining Examples

Query to create table on db

```
CREATE TABLE `user` (
`id` smallint(5) unsigned NOT NULL AUTO_INCREMENT,
`name` varchar(30) NOT NULL,
`course` smallint(5) unsigned DEFAULT NULL,
PRIMARY KEY (`id`)
) ENGINE=InnoDB;

CREATE TABLE `course` (
`id` smallint(5) unsigned NOT NULL AUTO_INCREMENT,
`name` varchar(50) NOT NULL,
PRIMARY KEY (`id`)
) ENGINE=InnoDB;
```

Since we’re using InnoDB tables and know that user.course and course.id are related, we can specify a foreign key relationship:

```
ALTER TABLE `user`
ADD CONSTRAINT `FK_course`
FOREIGN KEY (`course`) REFERENCES `course` (`id`)
ON UPDATE CASCADE;
```

Join Query (Inner Join)

```
SELECT user.name, course.name
FROM `user`
INNER JOIN `course` on user.course = course.id;
```

# 第17章：连接（JOINS）：连接3个具有相同id名称的表。

## 第17.1节：在具有相同名称的列上连接3个表

```
CREATE TABLE Table1 (  
  id INT UNSIGNED NOT NULL,  
  created_on DATE NOT NULL,  
  PRIMARY KEY (id)  
)  
CREATE TABLE Table2 (  
  id INT UNSIGNED NOT NULL,  
  personName VARCHAR(255) NOT NULL,  
  PRIMARY KEY (id)  
)  
CREATE TABLE Table3 (  
  id INT UNSIGNED NOT NULL,  
  accountName VARCHAR(255) NOT NULL,  
  PRIMARY KEY (id)  
)
```

创建表后，您可以执行一个查询，获取三个表中相同的 id

```
查询  
t1.id 作为 table1Id,  
  t2.id 作为 table2Id,  
  t3.id 作为 table3Id  
来自 Table1 t1  
左连接 Table2 t2 条件 t2.id = t1.id  
左连接 Table3 t3 条件 t3.id = t1.id
```

# Chapter 17: JOINS: Join 3 table with the same name of id.

## Section 17.1: Join 3 tables on a column with the same name

```
CREATE TABLE Table1 (  
  id INT UNSIGNED NOT NULL,  
  created_on DATE NOT NULL,  
  PRIMARY KEY (id)  
)  
CREATE TABLE Table2 (  
  id INT UNSIGNED NOT NULL,  
  personName VARCHAR(255) NOT NULL,  
  PRIMARY KEY (id)  
)  
CREATE TABLE Table3 (  
  id INT UNSIGNED NOT NULL,  
  accountName VARCHAR(255) NOT NULL,  
  PRIMARY KEY (id)  
)
```

after creating the tables you could do a select query to get the id's of all three tables that are the same

```
SELECT  
  t1.id AS table1Id,  
  t2.id AS table2Id,  
  t3.id AS table3Id  
FROM Table1 t1  
LEFT JOIN Table2 t2 ON t2.id = t1.id  
LEFT JOIN Table3 t3 ON t3.id = t1.id
```



## 第18章：联合（UNION）

### 第18.1节：使用 UNION 组合 SELECT 语句

您可以使用UNION关键字组合两个结构相同的查询结果。

例如，如果你想从两个不同的表中获取所有联系信息，authors 和 editors，比如，你可以使用 **UNION** 关键字，像这样：

```
select name, email, phone_number
from authors

union

select name, email, phone_number
from editors
```

单独使用 **union** 会去除重复项。如果你需要在查询中保留重复项，可以使用 **ALL** 关键字，如：**UNION ALL**。

### 第18.2节：合并不同列的数据

```
SELECT name, caption as title, year, pages FROM books
UNION
SELECT name, title, year, 0 as pages FROM movies
```

当合并两个具有不同列的记录集时，应使用默认值来模拟缺失的列。

### 第18.3节：ORDER BY

如果需要对 UNION 的结果进行排序，请使用以下模式：

```
( SELECT ... )
UNION
( SELECT ... )
ORDER BY
```

没有括号的话，最终的 ORDER BY 会属于最后一个 SELECT。

### 第18.4节：通过 OFFSET 实现分页

在给 UNION 添加 LIMIT 时，应使用以下模式：

```
( SELECT ... ORDER BY x LIMIT 10 )
UNION
( SELECT ... ORDER BY x LIMIT 10 )
ORDER BY x LIMIT 10
```

由于无法预测“10”会来自哪个 SELECT，因此需要从每个 SELECT 中获取10条记录，然后进一步筛选列表，重复使用**ORDER BY**和**LIMIT**。

对于第4页的10条记录，需要使用以下模式：

```
( SELECT ... ORDER BY x LIMIT 40 )
```

## Chapter 18: UNION

### Section 18.1: Combining SELECT statements with UNION

You can combine the results of two identically structured queries with the **UNION** keyword.

For example, if you wanted a list of all contact info from two separate tables, authors and editors, for instance, you could use the **UNION** keyword like so:

```
select name, email, phone_number
from authors

union

select name, email, phone_number
from editors
```

Using **union** by itself will strip out duplicates. If you needed to keep duplicates in your query, you could use the **ALL** keyword like so: **UNION ALL**.

### Section 18.2: Combining data with different columns

```
SELECT name, caption as title, year, pages FROM books
UNION
SELECT name, title, year, 0 as pages FROM movies
```

When combining 2 record sets with different columns then emulate the missing ones with default values.

### Section 18.3: ORDER BY

If you need to sort the results of a UNION, use this pattern:

```
( SELECT ... )
UNION
( SELECT ... )
ORDER BY
```

Without the parentheses, the final ORDER BY would belong to the last SELECT.

### Section 18.4: Pagination via OFFSET

When adding a LIMIT to a UNION, this is the pattern to use:

```
( SELECT ... ORDER BY x LIMIT 10 )
UNION
( SELECT ... ORDER BY x LIMIT 10 )
ORDER BY x LIMIT 10
```

Since you cannot predict which SELECT(s) will the "10" will come from, you need to get 10 from each, then further whittle down the list, repeating both the **ORDER BY** and **LIMIT**.

For the 4th page of 10 items, this pattern is needed:

```
( SELECT ... ORDER BY x LIMIT 40 )
```

```
联合查询
( SELECT ... ORDER BY x LIMIT 40 )
ORDER BY x LIMIT 30, 10
```

也就是说，在每个SELECT中收集4页的数据，然后在UNION中执行OFFSET。

## 第18.5节：将具有相同列的不同MySQL表中的数据合并成唯一一行并运行查询

这个UNION ALL将来自多个表的数据合并，并作为查询使用的表名别名：

```
SELECT YEAR(date_time_column), MONTH(date_time_column), MIN(
DATE(date_time_column)),
MAX(
DATE(date_time_column)), COUNT(DISTINCT (ip)), COUNT(ip),
(COUNT(ip) / COUNT(DISTINCT (ip))) AS
Ratio
FROM (
    (SELECT date_time_column, ip FROM server_log_1 WHERE state = 'action' AND log_id = 150) UNION
ALL
    (SELECT date_time_column, ip FROM server_log_2 WHERE state = 'action' AND log_id = 150) UNION
ALL
    (SELECT date_time_column, ip FROM server_log_3 WHERE state = 'action' AND log_id = 150) UNION
ALL
    (SELECT date_time_column, ip FROM server_log WHERE state = 'action' AND log_id = 150)
) AS table_all
GROUP BY YEAR(date_time_column), MONTH(date_time_column);
```

## 第18.6节：UNION ALL 和 UNION

SELECT 1,22,44 UNION SELECT 2,33,55

信息	结果1	概况	状态
1	22	44	
1	22	44	
2	33	55	

SELECT 1,22,44 UNION SELECT 2,33,55 UNION SELECT 2,33,55

结果与上述相同。

使用 UNION ALL

当

SELECT 1,22,44 UNION SELECT 2,33,55 UNION ALL SELECT 2,33,55

信息	结果1	概况	状态
1	22	44	
1	22	44	
2	33	55	
2	33	55	

```
UNION
( SELECT ... ORDER BY x LIMIT 40 )
ORDER BY x LIMIT 30, 10
```

That is, collect 4 page's worth in each **SELECT**, then do the **OFFSET** in the **UNION**.

## Section 18.5: Combining and merging data on different MySQL tables with the same columns into unique rows and running query

This **UNION ALL** combines data from multiple tables and serve as a table name alias to use for your queries:

```
SELECT YEAR(date_time_column), MONTH(date_time_column), MIN(
DATE(date_time_column)),
MAX(
DATE(date_time_column)), COUNT(DISTINCT (ip)), COUNT(ip),
(COUNT(ip) / COUNT(DISTINCT (ip))) AS
Ratio
FROM (
    (SELECT date_time_column, ip FROM server_log_1 WHERE state = 'action' AND log_id = 150) UNION
ALL
    (SELECT date_time_column, ip FROM server_log_2 WHERE state = 'action' AND log_id = 150) UNION
ALL
    (SELECT date_time_column, ip FROM server_log_3 WHERE state = 'action' AND log_id = 150) UNION
ALL
    (SELECT date_time_column, ip FROM server_log WHERE state = 'action' AND log_id = 150)
) AS table_all
GROUP BY YEAR(date_time_column), MONTH(date_time_column);
```

## Section 18.6: UNION ALL and UNION

SELECT 1,22,44 UNION SELECT 2,33,55

信息	结果1	概况	状态
1	22	44	
1	22	44	
2	33	55	

SELECT 1,22,44 UNION SELECT 2,33,55 UNION SELECT 2,33,55

The result is the same as above.

use UNION ALL

when

SELECT 1,22,44 UNION SELECT 2,33,55 UNION ALL SELECT 2,33,55

信息	结果1	概况	状态
1	22	44	
1	22	44	
2	33	55	
2	33	55	

# 第19章：算术运算

## 第19.1节：算术运算符

MySQL 提供以下算术运算符

运算符	名称	示例
+	加法	<code>SELECT 3+5; -&gt; 8</code>
		<code>SELECT 3.5+2.5; -&gt; 6.0</code>
		<code>SELECT 3.5+2; -&gt; 5.5</code>
-	减法	<code>SELECT 3-5; -&gt; -2</code>
*	乘法	<code>SELECT 3 * 5; -&gt; 15</code>
/	除法	<code>SELECT 20 / 4; -&gt; 5</code>
		<code>SELECT 355 / 113; -&gt; 3.1416</code>
		<code>SELECT 10.0 / 0; -&gt; NULL</code>
整除	整数除法	<code>SELECT 5 DIV 2; -&gt; 2</code>
% 或 MOD取模		<code>SELECT 7 % 3; -&gt; 1</code>
		<code>SELECT 15 MOD 4 -&gt; 3</code>
		<code>SELECT 15 MOD -4 -&gt; 3</code>
		<code>SELECT -15 MOD 4 -&gt; -3</code>
		<code>SELECT -15 MOD -4 -&gt; -3</code>
		<code>SELECT 3 MOD 2.5 -&gt; 0.5</code>

### BIGINT

如果你的算术运算中的数字都是整数，MySQL 会使用BIGINT（有符号64位）整数数据类型来进行计算。例如：

```
select (1024 * 1024 * 1024 * 1024 *1024 * 1024) + 1 -> 1,152,921,504,606,846,977
```

以及

```
select (1024 * 1024 * 1024 * 1024 *1024 * 1024 * 1024 -> BIGINT 超出范围错误
```

### DOUBLE

如果你的算术中有任何数字是分数，MySQL 使用64 位 IEEE 754 浮点算术。使用浮点算术时必须小心，因为许多浮点数本质上是近似值而非精确值。

## 第 19.2 节：数学常数

圆周率

下面返回PI的值，格式化为 6 位小数。实际值适用于DOUBLE；

```
SELECT PI(); -> 3.141593
```

## 第 19.3 节：三角函数（SIN，COS）

角度以弧度为单位，而非度。所有计算均在IEEE 754 64 位浮点数中完成。所有浮点计算都存在称为机器ε（epsilon）误差的小误差，因此避免尝试比较它们是否相等。使用浮点数时无法避免这些误差；它们是技术内置的。

如果在三角函数计算中使用DECIMAL值，它们会被隐式转换为浮点数，然后

# Chapter 19: Arithmetic

## Section 19.1: Arithmetic Operators

MySQL provides the following arithmetic operators

Operator	Name	Example
+	Addition	<code>SELECT 3+5; -&gt; 8</code>
		<code>SELECT 3.5+2.5; -&gt; 6.0</code>
		<code>SELECT 3.5+2; -&gt; 5.5</code>
-	Subtraction	<code>SELECT 3-5; -&gt; -2</code>
*	Multiplication	<code>SELECT 3 * 5; -&gt; 15</code>
/	Division	<code>SELECT 20 / 4; -&gt; 5</code>
		<code>SELECT 355 / 113; -&gt; 3.1416</code>
		<code>SELECT 10.0 / 0; -&gt; NULL</code>
DIV	Integer Division	<code>SELECT 5 DIV 2; -&gt; 2</code>
% or MOD	Modulo	<code>SELECT 7 % 3; -&gt; 1</code>
		<code>SELECT 15 MOD 4 -&gt; 3</code>
		<code>SELECT 15 MOD -4 -&gt; 3</code>
		<code>SELECT -15 MOD 4 -&gt; -3</code>
		<code>SELECT -15 MOD -4 -&gt; -3</code>
		<code>SELECT 3 MOD 2.5 -&gt; 0.5</code>

### BIGINT

If the numbers in your arithmetic are all integers, MySQL uses the **BIGINT** (signed 64-bit) integer data type to do its work. For example:

```
select (1024 * 1024 * 1024 * 1024 *1024 * 1024) + 1 -> 1,152,921,504,606,846,977
```

and

```
select (1024 * 1024 * 1024 * 1024 *1024 * 1024 * 1024 -> BIGINT out of range error
```

### DOUBLE

If any numbers in your arithmetic are fractional, MySQL uses [64-bit IEEE 754 floating point arithmetic](#). You must be careful when using floating point arithmetic, because many [floating point numbers are, inherently, approximations rather than exact values](#).

## Section 19.2: Mathematical Constants

### Pi

The following returns the value of PI formatted to 6 decimal places. The actual value is good to **DOUBLE**;

```
SELECT PI(); -> 3.141593
```

## Section 19.3: Trigonometry (SIN, COS)

Angles are in Radians, not Degrees. All computations are done in [IEEE 754 64-bit floating point](#). All floating point computations are subject to small errors, known as [machine ε \(epsilon\) errors](#), so avoid trying to compare them for equality. There is no way to avoid these errors when using floating point; they are built in to the technology.

If you use **DECIMAL** values in trigonometric computations, they are implicitly converted to floating point, and then

再转换回十进制。

正弦

返回以弧度表示的数字X的正弦值

```
SELECT SIN(PI()); -> 1.2246063538224e-16
```

余弦

返回以弧度表示的X的余弦值

```
SELECT COS(PI()); -> -1
```

正切

返回以弧度表示的数字X的正切值。注意结果非常接近零，但不完全是零。这是机器ε的一个例子。

```
SELECT TAN(PI()); -> -1.2246063538224e-16
```

反余弦（余弦的反函数）

如果 X 在 -1 到 1 范围内，返回 X 的反余弦值

```
SELECT ACOS(1); -> 0
SELECT ACOS(1.01); -> NULL
```

反正弦（反正弦函数）

如果 X 在 -1 到 1 范围内，返回 X 的反正弦值

```
SELECT ASIN(0.2); -> 0.20135792079033
```

反正切（反正切函数）

ATAN(x) 返回单个数字的反正切值。

```
SELECT ATAN(2); -> 1.1071487177941
```

ATAN2(X, Y) 返回两个变量 X 和 Y 的反正切值。它类似于计算 Y / X 的反正切。但它在数值上更稳健：当 X 接近零时，t 函数仍能正确工作，并且两个参数的符号用于确定结果的象限。

最佳实践建议尽可能使用ATAN2()函数而非ATAN()函数来编写公式。

```
ATAN2(1,1); -> 0.7853981633974483 (45 度)
ATAN2(1,-1); -> 2.356194490192345 (135 度)
ATAN2(0, -1); -> PI (180 度) 不要尝试 ATAN(-1 / 0)... 它不会工作
```

余切

返回 X 的余切值

back to decimal.

Sine

Returns the sine of a number X expressed in radians

```
SELECT SIN(PI()); -> 1.2246063538224e-16
```

Cosine

Returns the cosine of X when X is given in radians

```
SELECT COS(PI()); -> -1
```

Tangent

Returns the tangent of a number X expressed in radians. Notice the result is very close to zero, but not exactly zero. This is an example of machine ε.

```
SELECT TAN(PI()); -> -1.2246063538224e-16
```

Arc Cosine (inverse cosine)

Returns the arc cosine of X if X is in the range -1 to 1

```
SELECT ACOS(1); -> 0
SELECT ACOS(1.01); -> NULL
```

Arc Sine (inverse sine)

Returns the arc sine of X if X is in the range -1 to 1

```
SELECT ASIN(0.2); -> 0.20135792079033
```

Arc Tangent (inverse tangent)

ATAN(x) returns the arc tangent of a single number.

```
SELECT ATAN(2); -> 1.1071487177941
```

ATAN2(X, Y) returns the arc tangent of the two variables X and Y. It is similar to calculating the arc tangent of Y / X. But it is numerically more robust: t functions correctly when X is near zero, and the signs of both arguments are used to determine the quadrant of the result.

Best practice suggests writing formulas to use ATAN2() rather than ATAN() wherever possible.

```
ATAN2(1,1); -> 0.7853981633974483 (45 degrees)
ATAN2(1,-1); -> 2.356194490192345 (135 degrees)
ATAN2(0, -1); -> PI (180 degrees) don't try ATAN(-1 / 0)... it won't work
```

Cotangent

Returns the cotangent of X

```
SELECT COT(12); -> -1.5726734063977
```

转换

```
SELECT RADIANS(90) -> 1.5707963267948966
SELECT SIN(RADIANS(90)) -> 1
SELECT DEGREES(1), DEGREES(PI()) -> 57.29577951308232, 180
```

第19.4节：四舍五入（ROUND，FLOOR，CEIL）

将十进制数四舍五入为整数值

对于精确数值（例如 DECIMAL）：如果数字的小数点后第一位是5或更大，则此函数会将数字向远离零的方向四舍五入到下一个整数。如果该小数位是4或更小，则此函数会将数字四舍五入到最接近零的下一个整数值。

```
SELECT ROUND(4.51) -> 5
SELECT ROUND(4.49) -> 4
SELECT ROUND(-4.51) -> -5
```

对于近似数值（例如 DOUBLE）：ROUND() 函数的结果依赖于 C 库；在许多系统中，这意味着 ROUND() 使用的是四舍六入，五取偶规则：

```
SELECT ROUND(45e-1) -> 4 -- 最近的偶数是 4
SELECT ROUND(55e-1) -> 6 -- 最近的偶数是 6
```

向上取整一个数字

要向上取整一个数字，可以使用 CEIL() 或 CEILING() 函数

```
SELECT CEIL(1.23) -> 2
SELECT CEILING(4.83) -> 5
```

向下取整一个数字

要向下取整一个数字，使用 FLOOR() 函数

```
SELECT FLOOR(1.99) -> 1
```

FLOOR 和 CEIL 分别向负无穷方向和远离负无穷方向取整：

```
SELECT FLOOR(-1.01), CEIL(-1.01) -> -2 和 -1
SELECT FLOOR(-1.99), CEIL(-1.99) -> -2 和 -1
```

将十进制数四舍五入到指定的小数位数。

```
SELECT ROUND(1234.987, 2) -> 1234.99
SELECT ROUND(1234.987, -2) -> 1200
```

关于向上或向下以及“5”的讨论同样适用。

第19.5节：幂运算（POW）

要将数字 x 提升到幂 y，可以使用POW()或POWER()函数

```
SELECT COT(12); -> -1.5726734063977
```

Conversion

```
SELECT RADIANS(90) -> 1.5707963267948966
SELECT SIN(RADIANS(90)) -> 1
SELECT DEGREES(1), DEGREES(PI()) -> 57.29577951308232, 180
```

Section 19.4: Rounding (ROUND, FLOOR, CEIL)

Round a decimal number to an integer value

For exact numeric values (e.g. **DECIMAL**): If the first decimal place of a number is 5 or higher, this function will round a number to the next integer *away from zero*. If that decimal place is 4 or lower, this function will round to the next integer value *closest to zero*.

```
SELECT ROUND(4.51) -> 5
SELECT ROUND(4.49) -> 4
SELECT ROUND(-4.51) -> -5
```

For approximate numeric values (e.g. **DOUBLE**): The result of the ROUND() function depends on the C library; on many systems, this means that ROUND() uses the *round to the nearest even* rule:

```
SELECT ROUND(45e-1) -> 4 -- The nearest even value is 4
SELECT ROUND(55e-1) -> 6 -- The nearest even value is 6
```

Round up a number

To round up a number use either the CEIL() or CEILING() function

```
SELECT CEIL(1.23) -> 2
SELECT CEILING(4.83) -> 5
```

Round down a number

To round down a number, use the FLOOR() function

```
SELECT FLOOR(1.99) -> 1
```

FLOOR and CEIL go toward / away from -infinity:

```
SELECT FLOOR(-1.01), CEIL(-1.01) -> -2 and -1
SELECT FLOOR(-1.99), CEIL(-1.99) -> -2 and -1
```

Round a decimal number to a specified number of decimal places.

```
SELECT ROUND(1234.987, 2) -> 1234.99
SELECT ROUND(1234.987, -2) -> 1200
```

The discussion of up versus down and "5" applies, too.

Section 19.5: Raise a number to a power (POW)

To raise a number x to a power y, use either the POW() or POWER() functions



```
SELECT POW(2,2); => 4
SELECT POW(4,2); => 16
```

## 第19.6节：平方根（SQRT）

使用SQRT()函数。如果数字为负，则返回NULL

```
SELECT SQRT(16); -> 4
SELECT SQRT(-3); -> NULL
```

## 第19.7节：随机数（RAND）

### 生成一个随机数

要生成一个介于0和1之间的伪随机浮点数，请使用RAND()函数

假设你有以下查询

```
SELECT i, RAND() FROM t;
```

这将返回类似如下内容

i	RAND()
1	0.6191438870682
2	0.93845168309142
3	0.83482678498591

### 范围内的随机数

要生成范围在 a <= n <= b 之间的随机数，可以使用以下公式

```
FLOOR(a + RAND() * (b - a + 1))
```

例如，这将生成一个介于7到12之间的随机数

```
SELECT FLOOR(7 + (RAND() * 6));
```

随机返回表中行的一种简单方法：

```
SELECT * FROM tbl ORDER BY RAND();
```

这些是伪随机数。

MySQL中的伪随机数生成器不是加密安全的。也就是说，如果你使用MySQL生成随机数作为秘密，一个知道你使用MySQL的有心人将比你想象的更容易猜出你的秘密。

## 第19.8节：绝对值和符号（ABS，SIGN）

返回数字的绝对值

```
SELECT ABS(2); -> 2
SELECT ABS(-46); -> 46
```

```
SELECT POW(2,2); => 4
SELECT POW(4,2); => 16
```

## Section 19.6: Square Root (SQRT)

Use the SQRT() function. If the number is negative, NULL will be returned

```
SELECT SQRT(16); -> 4
SELECT SQRT(-3); -> NULL
```

## Section 19.7: Random Numbers (RAND)

### Generate a random number

To generate a pseudorandom floating point number between 0 and 1, use the RAND() function

Suppose you have the following query

```
SELECT i, RAND() FROM t;
```

This will return something like this

i	RAND()
1	0.6191438870682
2	0.93845168309142
3	0.83482678498591

### Random Number in a range

To generate a random number in the range a <= n <= b, you can use the following formula

```
FLOOR(a + RAND() * (b - a + 1))
```

For example, this will generate a random number between 7 and 12

```
SELECT FLOOR(7 + (RAND() * 6));
```

A simple way to randomly return the rows in a table:

```
SELECT * FROM tbl ORDER BY RAND();
```

These are **pseudorandom** numbers.

The pseudorandom number generator in MySQL is not cryptographically secure. That is, if you use MySQL to generate random numbers to be used as secrets, a determined adversary who knows you used MySQL will be able to guess your secrets more easily than you might believe.

## Section 19.8: Absolute Value and Sign (ABS, SIGN)

Return the absolute value of a number

```
SELECT ABS(2); -> 2
SELECT ABS(-46); -> 46
```

数字的符号将其与0进行比较。

符号结果	示例
-1	n < 0选择SIGN(42); -> 1
0	n = 0选择SIGN(0); -> 0
1	n > 0选择SIGN(-3); -> -1
选择SIGN(-423421); -> -1	

The [sign](#) of a number compares it to 0.

Sign Result	Example
-1	n < 0 <b>SELECT SIGN(42);</b> -> 1
0	n = 0 <b>SELECT SIGN(0);</b> -> 0
1	n > 0 <b>SELECT SIGN(-3);</b> -> -1
<b>SELECT SIGN(-423421);</b> -> -1	

# 第20章：字符串操作

名称	描述
ASCII()	返回最左侧字符的数值
BIN()	返回包含数字二进制表示的字符串
BIT_LENGTH()	返回参数的位数长度
CHAR()	返回传入的每个整数对应的字符
CHAR_LENGTH()	返回参数中的字符数
CHARACTER_LENGTH()	是 CHAR_LENGTH() 的同义词
CONCAT()	返回连接后的字符串
CONCAT_WS()	返回带分隔符的连接结果
ELT()	返回指定索引位置的字符串
EXPORT_SET()	返回一个字符串，对于值位中每个置位的位，返回一个“开”字符串，对于每个未置位的位，返回一个“关”字符串
FIELD()	返回第一个参数在后续参数中的索引（位置）
FIND_IN_SET()	返回第一个参数在第二个参数中的索引位置
FORMAT()	返回格式化为指定小数位数的数字
FROM_BASE64()	解码为Base64字符串并返回结果
HEX()	返回十进制或字符串值的十六进制表示
INSERT()	在指定位置插入子字符串，最多插入指定数量的字符
INSTR()	返回子字符串首次出现的位置索引
LCASE()	LOWER() 的同义词
LEFT()	返回指定的最左边的字符数
LENGTH()	返回字符串的字节长度
LIKE	简单的模式匹配
LOAD_FILE()	加载指定的文件
LOCATE()	返回子字符串首次出现的位置
LOWER()	返回参数的小写形式
LPAD()	返回用指定字符串左填充的字符串参数
LTRIM()	去除前导空格
MAKE_SET()	返回一个由逗号分隔的字符串集合，这些字符串对应于bits中被设置的位
MATCH	执行全文搜索
MID()	返回从指定位置开始的子字符串
NOT LIKE	简单模式匹配的否定
NOT REGEXP	REGEXP 的否定
OCT()	返回包含数字八进制表示的字符串
OCTET_LENGTH()	LENGTH() 的同义词
ORD()	返回参数中最左边字符的字符编码
POSITION()	LOCATE() 的同义词
QUOTE()	对参数进行转义以用于 SQL 语句
REGEXP	使用正则表达式进行模式匹配
REPEAT()	重复字符串指定的次数
REPLACE()	替换指定字符串的出现
REVERSE()	反转字符串中的字符
RIGHT()	返回指定的最右边字符数
RLIKE	REGEXP 的同义词

# Chapter 20: String operations

Name	Description
ASCII()	Return numeric value of left-most character
BIN()	Return a string containing binary representation of a number
BIT_LENGTH()	Return length of argument in bits
CHAR()	Return the character for each integer passed
CHAR_LENGTH()	Return number of characters in argument
CHARACTER_LENGTH()	Synonym for CHAR_LENGTH()
CONCAT()	Return concatenated string
CONCAT_WS()	Return concatenate with separator
ELT()	Return string at index number
EXPORT_SET()	Return a string such that for every bit set in the value bits, you get an on string and for every unset bit, you get an off string
FIELD()	Return the index (position) of the first argument in the subsequent arguments
FIND_IN_SET()	Return the index position of the first argument within the second argument
FORMAT()	Return a number formatted to specified number of decimal places
FROM_BASE64()	Decode to a base-64 string and return result
HEX()	Return a hexadecimal representation of a decimal or string value
INSERT()	Insert a substring at the specified position up to the specified number of characters
INSTR()	Return the index of the first occurrence of substring
LCASE()	Synonym for LOWER()
LEFT()	Return the leftmost number of characters as specified
LENGTH()	Return the length of a string in bytes
LIKE	Simple pattern matching
LOAD_FILE()	Load the named file
LOCATE()	Return the position of the first occurrence of substring
LOWER()	Return the argument in lowercase
LPAD()	Return the string argument, left-padded with the specified string
LTRIM()	Remove leading spaces
MAKE_SET()	Return a set of comma-separated strings that have the corresponding bit in bits set
MATCH	Perform full-text search
MID()	Return a substring starting from the specified position
NOT LIKE	Negation of simple pattern matching
NOT REGEXP	Negation of REGEXP
OCT()	Return a string containing octal representation of a number
OCTET_LENGTH()	Synonym for LENGTH()
ORD()	Return character code for leftmost character of the argument
POSITION()	Synonym for LOCATE()
QUOTE()	Escape the argument for use in an SQL statement
REGEXP	Pattern matching using regular expressions
REPEAT()	Repeat a string the specified number of times
REPLACE()	Replace occurrences of a specified string
REVERSE()	Reverse the characters in a string
RIGHT()	Return the specified rightmost number of characters
RLIKE	Synonym for REGEXP

RPAD()	将字符串追加指定次数
RTRIM()	删除尾部空格
SOUNDEX()	返回一个soundex字符串
SOUNDS LIKE	比较发音
SPACE()	返回指定数量的空格字符串
STRCMP()	比较两个字符串
SUBSTR()	返回指定的子字符串
SUBSTRING()	返回指定的子字符串
SUBSTRING_INDEX()	返回指定分隔符出现次数之前的字符串子串
TO_BASE64()	返回转换为Base64字符串的参数
TRIM()	去除字符串开头和结尾的空格
UCASE()	UPPER()的同义词
UNHEX()	返回包含数字的十六进制表示的字符串
UPPER()	转换为大写
WEIGHT_STRING()	返回字符串的权重字符串

## 第20.1节：LENGTH()

返回字符串的字节长度。由于某些字符可能使用多个字节编码，如果你想要字符长度，请参见CHAR\_LENGTH()

语法：LENGTH(str)

```
LENGTH('foobar') -- 6
LENGTH("fööbar") -- 8 -- 与CHAR_LENGTH(...) = 6对比
```

## 第20.2节：CHAR\_LENGTH()

返回字符串中的字符数

语法：CHAR\_LENGTH(str)

```
CHAR_LENGTH('foobar') -- 6
CHAR_LENGTH("fööbar") -- 6 -- 与 LENGTH(...) = 8 对比
```

## 第20.3节：HEX(str)

将参数转换为十六进制。此函数用于字符串。

```
HEX('fööbar') -- 66F6F6626172 -- 在“字符集 latin1”中，因为“F6”是ö的十六进制表示
HEX('fööbar') -- 66C3B6C3B6626172 -- 在“字符集 utf8 或 utf8mb4”中，因为“C3B6”是ö的十六进制表示
```

## 第20.4节：SUBSTRING()

SUBSTRING（或等效的 SUBSTR）返回从指定位置开始的子字符串，并且可选地指定长度

语法：SUBSTRING(str, start\_position)

```
SELECT SUBSTRING('foobarbaz', 4); -- 'barbaz'
```

RPAD()	Append string the specified number of times
RTRIM()	Remove trailing spaces
SOUNDEX()	Return a soundex string
SOUNDS LIKE	Compare sounds
SPACE()	Return a string of the specified number of spaces
STRCMP()	Compare two strings
SUBSTR()	Return the substring as specified
SUBSTRING()	Return the substring as specified
SUBSTRING_INDEX()	Return a substring from a string before the specified number of occurrences of the delimiter
TO_BASE64()	Return the argument converted to a base-64 string
TRIM()	Remove leading and trailing spaces
UCASE()	Synonym for UPPER()
UNHEX()	Return a string containing hex representation of a number
UPPER()	Convert to uppercase
WEIGHT_STRING()	Return the weight string for a string

## Section 20.1: LENGTH()

Return the length of the string in bytes. Since some characters may be encoded using more than one byte, if you want the length in characters see CHAR\_LENGTH()

Syntax: LENGTH(str)

```
LENGTH('foobar') -- 6
LENGTH("fööbar") -- 8 -- contrast with CHAR_LENGTH(...) = 6
```

## Section 20.2: CHAR\_LENGTH()

Return the number of characters in the string

Syntax: CHAR\_LENGTH(str)

```
CHAR_LENGTH('foobar') -- 6
CHAR_LENGTH("fööbar") -- 6 -- contrast with LENGTH(...) = 8
```

## Section 20.3: HEX(str)

Convert the argument to hexadecimal. This is used for strings.

```
HEX('fööbar') -- 66F6F6626172 -- in "CHARACTER SET latin1" because "F6" is hex for ö
HEX('fööbar') -- 66C3B6C3B6626172 -- in "CHARACTER SET utf8 or utf8mb4" because "C3B6" is hex for ö
```

## Section 20.4: SUBSTRING()

SUBSTRING (or equivalent: SUBSTR) returns the substring starting from the specified position and, optionally, with the specified length

Syntax: SUBSTRING(str, start\_position)

```
SELECT SUBSTRING('foobarbaz', 4); -- 'barbaz'
```

```
SELECT SUBSTRING('foobarbaz' FROM 4); -- 'barbaz'

-- 使用负索引
SELECT SUBSTRING('foobarbaz', -6); -- 'barbaz'
SELECT SUBSTRING('foobarbaz' FROM -6); -- 'barbaz'
```

语法：SUBSTRING(字符串,起始位置, 长度)

```
SELECT SUBSTRING('foobarbaz', 4, 3); -- 'bar'
SELECT SUBSTRING('foobarbaz', FROM 4 FOR 3); -- 'bar'

-- 使用负索引
SELECT SUBSTRING('foobarbaz', -6, 3); -- 'bar'
SELECT SUBSTRING('foobarbaz' FROM -6 FOR 3); -- 'bar'
```

## 第20.5节：UPPER() / UCASE()

将字符串参数转换为大写

语法：UPPER(str)

```
UPPER('fOoBar') -- 'FOOBAR'
UCASE('fOoBar') -- 'FOOBAR'
```

## 第20.6节：STR\_TO\_DATE - 将字符串转换为日期

有一列字符串类型，名为my\_date\_field，值例如07/25/2016，以下语句演示了STR\_TO\_DATE函数的使用：

```
SELECT STR_TO_DATE(my_date_field, '%m/%d/%Y') FROM my_table;
```

你也可以将此函数用作WHERE子句的一部分。

## 第20.7节：LOWER() / LCASE()

将字符串参数转换为小写

语法：LOWER(str)

```
LOWER('fOoBar') -- 'foobar'
LCASE('fOoBar') -- 'foobar'
```

## 第20.8节：REPLACE()

将字符串参数转换为小写

语法：REPLACE(str, from\_str, to\_str)

```
REPLACE('foobarbaz', 'bar', 'BAR') -- 'fooBARbaz'
REPLACE('foobarbaz', 'zzz', 'ZZZ') -- 'foobarbaz'
```

## 第20.9节：在逗号分隔列表中查找元素

```
SELECT FIND_IN_SET('b','a,b,c');
```

```
SELECT SUBSTRING('foobarbaz' FROM 4); -- 'barbaz'

-- using negative indexing
SELECT SUBSTRING('foobarbaz', -6); -- 'barbaz'
SELECT SUBSTRING('foobarbaz' FROM -6); -- 'barbaz'
```

Syntax: SUBSTRING(str, start\_position, length)

```
SELECT SUBSTRING('foobarbaz', 4, 3); -- 'bar'
SELECT SUBSTRING('foobarbaz', FROM 4 FOR 3); -- 'bar'

-- using negative indexing
SELECT SUBSTRING('foobarbaz', -6, 3); -- 'bar'
SELECT SUBSTRING('foobarbaz' FROM -6 FOR 3); -- 'bar'
```

## Section 20.5: UPPER() / UCASE()

Convert in uppercase the string argument

Syntax: UPPER(str)

```
UPPER('fOoBar') -- 'FOOBAR'
UCASE('fOoBar') -- 'FOOBAR'
```

## Section 20.6: STR\_TO\_DATE - Convert string to date

With a column of one of the string types, named my\_date\_field with a value such as [the string] 07/25/2016, the following statement demonstrates the use of the STR\_TO\_DATE function:

```
SELECT STR_TO_DATE(my_date_field, '%m/%d/%Y') FROM my_table;
```

You could use this function as part of WHERE clause as well.

## Section 20.7: LOWER() / LCASE()

Convert in lowercase the string argument

Syntax: LOWER(str)

```
LOWER('fOoBar') -- 'foobar'
LCASE('fOoBar') -- 'foobar'
```

## Section 20.8: REPLACE()

Convert in lowercase the string argument

Syntax: REPLACE(str, from\_str, to\_str)

```
REPLACE('foobarbaz', 'bar', 'BAR') -- 'fooBARbaz'
REPLACE('foobarbaz', 'zzz', 'ZZZ') -- 'foobarbaz'
```

## Section 20.9: Find element in comma separated list

```
SELECT FIND_IN_SET('b','a,b,c');
```



返回值：

2
---

```
SELECT FIND_IN_SET('d','a,b,c');
```

返回值：

0
---

Return value:

2
---

```
SELECT FIND_IN_SET('d','a,b,c');
```

Return value:

0
---

# 第21章：日期和时间操作

## 第21.1节：日期运算

```
NOW() + INTERVAL 1 DAY -- 明天此时

CURDATE() - INTERVAL 4 DAY -- 四天前的午夜
```

显示3到10小时前（180到600分钟前）提出的mysql问题：

```
SELECT qId,askDate,minuteDiff
FROM
(
  SELECT qId,askDate,
    TIMESTAMPDIFF(MINUTE,askDate,now()) as minuteDiff
    FROM questions_mysql
) xDerived
WHERE minuteDiff BETWEEN 180 AND 600
ORDER BY qId DESC
LIMIT 50;
```

qId	askDate	minuteDiff
38546828	2016-07-23 22:06:50	182
38546733	2016-07-23 21:53:26	195
38546707	2016-07-23 21:48:46	200
38546687	2016-07-23 21:45:26	203
...		

MySQL 手册页关于TIMESTAMPDIFF()。

**注意** 不要尝试使用类似 CURDATE() + 1 这样的表达式进行 MySQL 中的日期运算。它们不会返回你期望的结果，尤其是如果你习惯于 Oracle 数据库产品。请改用 CURDATE() + INTERVAL 1 DAY。

## 第 21.2 节：SYSDATE()、NOW()、CURDATE()

```
SELECT SYSDATE();
```

该函数返回当前日期和时间，格式为'YYYY-MM-DD HH:MM:SS'或YYYYMMDDHHMMSS，具体取决于函数是在字符串还是数字上下文中使用。它返回当前时区的日期和时间。

```
SELECT NOW();
```

该函数是 SYSDATE()的同义词。

```
SELECT CURDATE();
```

该函数返回当前日期，不包含时间，格式为'YYYY-MM-DD'或YYYYMMDD，具体取决于函数是在字符串还是数字上下文中使用。它返回当前时区的日期。

# Chapter 21: Date and Time Operations

## Section 21.1: Date arithmetic

```
NOW() + INTERVAL 1 DAY -- This time tomorrow

CURDATE() - INTERVAL 4 DAY -- Midnight 4 mornings ago
```

Show the mysql questions stored that were asked 3 to 10 hours ago (180 to 600 minutes ago):

```
SELECT qId,askDate,minuteDiff
FROM
(
  SELECT qId,askDate,
    TIMESTAMPDIFF(MINUTE,askDate,now()) as minuteDiff
    FROM questions_mysql
) xDerived
WHERE minuteDiff BETWEEN 180 AND 600
ORDER BY qId DESC
LIMIT 50;
```

qId	askDate	minuteDiff
38546828	2016-07-23 22:06:50	182
38546733	2016-07-23 21:53:26	195
38546707	2016-07-23 21:48:46	200
38546687	2016-07-23 21:45:26	203
...		

MySQL manual pages for [TIMESTAMPDIFF\(\)](#).

**Beware** Do not try to use expressions like CURDATE() + 1 for date arithmetic in MySQL. They don't return what you expect, especially if you're accustomed to the Oracle database product. Use CURDATE() + INTERVAL 1 DAY instead.

## Section 21.2: SYSDATE(), NOW(), CURDATE()

```
SELECT SYSDATE();
```

This function returns the current date and time as a value in 'YYYY-MM-DD HH:MM:SS' or YYYYMMDDHHMMSS format, depending on whether the function is used in a string or numeric context. It returns the date and time in the current time zone.

```
SELECT NOW();
```

This function is a synonym for SYSDATE().

```
SELECT CURDATE();
```

This function returns the current date, without any time, as a value in 'YYYY-MM-DD' or YYYYMMDD format, depending on whether the function is used in a string or numeric context. It returns the date in the current time zone.

第21.3节：针对日期范围的测试

虽然使用BETWEEN ... AND ... 来表示日期范围非常诱人，但这存在问题。相反，这种模式避免了大多数问题：

```
WHERE x >= '2016-02-25'
      AND x < '2016-02-25' + INTERVAL 5 DAY
```

- 优点：
- BETWEEN 是“包含”的，因此包括了结束日期或时间秒数。
  - 23:59:59 如果你的 DATETIME 有微秒精度，这种写法既笨拙又错误。
  - 这种模式避免了处理闰年和其他日期计算的问题。
  - 无论 x 是 DATE、DATETIME 还是 TIMESTAMP 都适用。

第21.4节：从给定的日期或日期时间表达式中提取日期

```
SELECT DATE('2003-12-31 01:02:03');
```

输出将是：

2003-12-31

第21.5节：使用索引进行日期和时间查找

许多现实世界的数据库表包含大量行，其DATETIME或TIMESTAMP列的值跨越很长时间，包括数年甚至数十年。通常需要使用WHERE子句来检索该时间跨度内的某个子集。例如，我们可能想从表中检索2016年9月1日的行。

一种低效的做法是：

```
WHERE DATE(x) = '2016-09-01' /* 慢！ */
```

这很低效，因为它对列的值应用了一个函数——DATE()。这意味着MySQL必须检查每个 x的值，且无法使用索引。

更好的做法是：

```
WHERE x >= '2016-09-01'
      AND x < '2016-09-01' + INTERVAL 1 DAY
```

这会选择 x在指定日期当天的所有值，直到但不包括（因此使用<）第二天的午夜。

如果表在 x列上有索引，数据库服务器就可以对索引执行范围扫描。这意味着它可以快速找到第一个相关的 x值，然后顺序扫描索引直到找到最后一个相关值。索引范围扫描比DATE(x) = 所需的全表扫描效率高得多。

2016-09-01.

不要被这个诱惑，尽管它看起来更高效。

```
WHERE x BETWEEN '2016-09-01' AND '2016-09-01' + INTERVAL 1 DAY /* 错误！ */
```

Section 21.3: Testing against a date range

Although it is very tempting to use BETWEEN ... AND ... for a date range, it is problematical. Instead, this pattern avoids most problems:

```
WHERE x >= '2016-02-25'
      AND x < '2016-02-25' + INTERVAL 5 DAY
```

- Advantages:
- BETWEEN is 'inclusive' thereby including the final date or second.
  - 23:59:59 is clumsy and wrong if you have microsecond resolution on a DATETIME.
  - This pattern avoid dealing with leap years and other data calculations.
  - It works whether x is DATE, DATETIME or TIMESTAMP.

Section 21.4: Extract Date from Given Date or DateTime Expression

```
SELECT DATE('2003-12-31 01:02:03');
```

The output will be:

2003-12-31

Section 21.5: Using an index for a date and time lookup

Many real-world database tables have many rows with DATETIME OR TIMESTAMP column values spanning a lot of time, including years or even decades. Often it's necessary to use a WHERE clause to retrieve some subset of that timespan. For example, we might want to retrieve rows for the date 1-September-2016 from a table.

An inefficient way to do that is this:

```
WHERE DATE(x) = '2016-09-01' /* slow! */
```

It's inefficient because it applies a function -- DATE() -- to the values of a column. That means MySQL must examine each value of x, and an index cannot be used.

A better way to do the operation is this

```
WHERE x >= '2016-09-01'
      AND x < '2016-09-01' + INTERVAL 1 DAY
```

This selects a range of values of x lying anywhere on the day in question, up until but *not including* (hence <) midnight on the next day.

If the table has an index on the x column, then the database server can perform a range scan on the index. That means it can quickly find the first relevant value of x, and then scan the index sequentially until it finds the last relevant value. An index range scan is much more efficient than the full table scan required by DATE(x) = '2016-09-01.

Don't be tempted to use this, even though it looks more efficient.

```
WHERE x BETWEEN '2016-09-01' AND '2016-09-01' + INTERVAL 1 DAY /* wrong! */
```

它的效率与范围扫描相同，但会选择 x 值恰好落在2016年9月2日午夜的行，这不是你想要的。

## 第21.6节：Now()

```
Select Now();
```

显示当前服务器的日期和时间。

```
Update `footable` set mydatefield = Now();
```

这将用服务器配置时区的当前服务器日期和时间更新字段 mydatefield，例如

```
'2016-07-21 12:00:00'
```

It has the same efficiency as the range scan, but it will select rows with values of x falling exactly at midnight on 2-Sept-2016, which is not what you want.

## Section 21.6: Now()

```
Select Now();
```

Shows the current server date and time.

```
Update `footable` set mydatefield = Now();
```

This will update the field mydatefield with current server date and time in server's configured timezone, e.g.

```
'2016-07-21 12:00:00'
```

# 第22章：处理时区

## 第22.1节：检索特定时区的当前日期和时间

这会获取本地时间（印度标准时间）下的NOW()值，然后再获取UTC时间。

```
SELECT NOW();
SET time_zone='Asia/Kolkata';
SELECT NOW();
SET time_zone='UTC';
SELECT NOW();
```

## 第22.2节：将存储的`DATE`或`DATETIME`值转换为另一个时区

如果你有一个存储的DATE或DATETIME（某列中的值），它是相对于某个时区存储的，但在MySQL中，时区信息不会随值一起存储。因此，如果你想转换到另一个时区，可以，但必须知道原始时区。使用CONVERT\_TZ()可以进行转换。此示例显示了加州本地时间的销售记录。

```
SELECT CONVERT_TZ(date_sold,'UTC','America/Los_Angeles') date_sold_local
FROM sales
WHERE state_sold = 'CA'
```

## 第22.3节：检索特定时区的存储`TIMESTAMP`值

这真的很简单。所有TIMESTAMP值都存储为协调世界时，并且在每次呈现时总是转换为当前的time\_zone设置。

```
SET SESSION time_zone='America/Los_Angeles';
SELECT timestamp_sold
FROM sales
WHERE state_sold = 'CA'
```

这是为什么？TIMESTAMP值基于久经考验的UNIX time\_t数据类型。那些UNIX时间戳是以自1970-01-01 00:00:00 UTC以来的秒数存储的。

注意TIMESTAMP值以协调世界时存储。DATE和DATETIME值则存储为存储时所处的本地时间。

## 第22.4节：我的服务器本地时区设置是什么？

每台服务器都有一个默认的全局time\_zone设置，由服务器机器的所有者配置。你可以通过以下方式查询当前时区设置：

```
SELECT @@time_zone
```

不幸的是，这通常返回值为SYSTEM，意味着MySQL时间由服务器操作系统的时区设置控制。

这组查询（是的，这是一种技巧）会返回服务器时区设置与UTC之间的分钟偏移量

# Chapter 22: Handling Time Zones

## Section 22.1: Retrieve the current date and time in a particular time zone

This fetches the value of NOW() in local time, in India Standard Time, and then again in UTC.

```
SELECT NOW();
SET time_zone='Asia/Kolkata';
SELECT NOW();
SET time_zone='UTC';
SELECT NOW();
```

## Section 22.2: Convert a stored `DATE` or `DATETIME` value to another time zone

If you have a stored DATE or DATETIME (in a column somewhere) it was stored with respect to some time zone, but in MySQL the time zone is *not* stored with the value. So, if you want to convert it to another time zone, you can, but you must know the original time zone. Using CONVERT\_TZ() does the conversion. This example shows rows sold in California in local time.

```
SELECT CONVERT_TZ(date_sold,'UTC','America/Los_Angeles') date_sold_local
FROM sales
WHERE state_sold = 'CA'
```

## Section 22.3: Retrieve stored `TIMESTAMP` values in a particular time zone

This is really easy. All TIMESTAMP values are stored in universal time, and always converted to the present time\_zone setting whenever they are rendered.

```
SET SESSION time_zone='America/Los_Angeles';
SELECT timestamp_sold
FROM sales
WHERE state_sold = 'CA'
```

Why is this? TIMESTAMP values are based on the venerable UNIX time\_t data type. Those UNIX timestamps are stored as a number of seconds since 1970-01-01 00:00:00 UTC.

**Notice** TIMESTAMP values are stored in universal time. DATE and DATETIME values are stored in whatever local time was in effect when they were stored.

## Section 22.4: What is my server's local time zone setting?

Each server has a default global time\_zone setting, configured by the owner of the server machine. You can find out the current time zone setting this way:

```
SELECT @@time_zone
```

Unfortunately, that usually yields the value SYSTEM, meaning the MySQL time is governed by the server OS's time zone setting.

This sequence of queries (yes, [it's a hack](#)) gives you back the offset in minutes between the server's time zone



。

```
CREATE TEMPORARY TABLE times (dt DATETIME, ts TIMESTAMP);
SET time_zone = 'UTC';
INSERT INTO times VALUES(NOW(), NOW());
SET time_zone = 'SYSTEM';
SELECT dt, ts, TIMESTAMPDIFF(MINUTE, dt, ts)offset FROM times;
DROP TEMPORARY TABLE times;
```

这是如何工作的？临时表中两个不同数据类型的列是关键。 DATETIME 数据类型总是以本地时间存储在表中，而 TIMESTAMP 则以 UTC 存储。因此，当 INSERT 语句执行时，time\_zone 设置为 UTC，存储了两个相同的日期/时间值。

然后，SELECT 语句在 time\_zone 设置为服务器本地时间时执行。 TIMESTAMP 总是在 SELECT 语句中从其存储的 UTC 形式转换为本地时间。 DATETIME 则不会。因此，TIMESTAMPDIFF(MINUTE...) 操作 计算了本地时间和协调世界时之间的差异。

## 第22.5节：我的服务器中有哪些 time\_zone 值可用？

要获取 MySQL 服务器实例中可能的 time\_zone 值列表，请使用此命令。

```
SELECT mysql.time_zone_name.name
```

通常，这会显示由 Paul Eggert 在 [Internet Assigned Numbers Authority](#) 维护的 [ZoneInfo](#) 时区列表。全球大约有600个时区。

类 Unix 操作系统（例如 Linux 发行版、BSD 发行版和现代 Mac OS 发行版）会定期接收更新。在操作系统上安装这些更新可以让运行在该系统上的 MySQL 实例跟踪时区和夏令时/标准时切换的变化。

如果您获得的时区名称列表非常短，说明您的服务器配置不完整或正在运行Windows。以下是服务器管理员安装和维护ZoneInfo列表的说明。

setting and UTC.

```
CREATE TEMPORARY TABLE times (dt DATETIME, ts TIMESTAMP);
SET time_zone = 'UTC';
INSERT INTO times VALUES(NOW(), NOW());
SET time_zone = 'SYSTEM';
SELECT dt, ts, TIMESTAMPDIFF(MINUTE, dt, ts)offset FROM times;
DROP TEMPORARY TABLE times;
```

How does this work? The two columns in the temporary table with different data types is the clue. DATETIME data types are always stored in local time in tables, and TIMESTAMPS in UTC. So the INSERT statement, performed when the time\_zone is set to UTC, stores two identical date / time values.

Then, the SELECT statement, is done when the time\_zone is set to server local time. TIMESTAMPS are always translated from their stored UTC form to local time in SELECT statements. DATETIMES are not. So the [TIMESTAMPDIFF\(MINUTE...\) operation](#) computes the difference between local and universal time.

## Section 22.5: What time\_zone values are available in my server?

To get a list of possible time\_zone values in your MySQL server instance, use this command.

```
SELECT mysql.time_zone_name.name
```

Ordinarily, this shows the [ZoneInfo list of time zones](#) maintained by Paul Eggert at the [Internet Assigned Numbers Authority](#). Worldwide there are approximately 600 time zones.

Unix-like operating systems (Linux distributions, BSD distributions, and modern Mac OS distributions, for example) receive routine updates. Installing these updates on an operating system lets the MySQL instances running there track the changes in time zone and daylight / standard time changeovers.

If you get a much shorter list of time zone names, your server is either incompletely configured or running on Windows. [Here are instructions](#) for your server administrator to install and maintain the ZoneInfo list.

# 第23章：正则表达式

正则表达式是一种指定复杂搜索模式的强大方式。

## 第23.1节：REGEXP / RLIKE

REGEXP（或其同义词RLIKE）操作符允许基于正则表达式进行模式匹配。

考虑以下employee表：

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	PHONE_NUMBER	SALARY
100	Steven	King	515.123.4567	24000.00
101	Neena	Kochhar	515.123.4568	17000.00
102	Lex	De Haan	515.123.4569	17000.00
103	Alexander	Hunold	590.423.4567	9000.00
104	Bruce	Ernst	590.423.4568	6000.00
105	David	Austin	590.423.4569	4800.00
106	Valli	Pataballa	590.423.4560	4800.00
107	Diana	Lorentz	590.423.5567	4200.00
108	Nancy	Greenberg	515.124.4569	12000.00
109	Daniel	Faviet	515.124.4169	9000.00
110	John	Chen	515.124.4269	8200.00

### 模式 ^

选择所有FIRST\_NAME以N开头的员工。

### 查询

```
SELECT * FROM employees WHERE FIRST_NAME REGEXP '^N'
-- 模式以^开头-----^
```

### 模式 \$\*\*

选择所有电话号码以4569结尾的员工。

### 查询

```
SELECT * FROM employees WHERE PHONE_NUMBER REGEXP '4569$'
-- 模式以此结尾-----^
```

### NOT REGEXP

选择所有名字不以N开头的员工。

### 查询

```
SELECT * FROM employees WHERE FIRST_NAME NOT REGEXP '^N'
-- 模式不以此开头-----^
```

# Chapter 23: Regular Expressions

A regular expression is a powerful way of specifying a pattern for a complex search.

## Section 23.1: REGEXP / RLIKE

The **REGEXP** (or its synonym, **RLIKE**) operator allows pattern matching based on regular expressions.

Consider the following employee table:

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	PHONE_NUMBER	SALARY
100	Steven	King	515.123.4567	24000.00
101	Neena	Kochhar	515.123.4568	17000.00
102	Lex	De Haan	515.123.4569	17000.00
103	Alexander	Hunold	590.423.4567	9000.00
104	Bruce	Ernst	590.423.4568	6000.00
105	David	Austin	590.423.4569	4800.00
106	Valli	Pataballa	590.423.4560	4800.00
107	Diana	Lorentz	590.423.5567	4200.00
108	Nancy	Greenberg	515.124.4569	12000.00
109	Daniel	Faviet	515.124.4169	9000.00
110	John	Chen	515.124.4269	8200.00

### Pattern ^

Select all employees whose FIRST\_NAME starts with **N**.

### Query

```
SELECT * FROM employees WHERE FIRST_NAME REGEXP '^N'
-- Pattern start with-----^
```

### Pattern \$\*\*

Select all employees whose PHONE\_NUMBER ends with **4569**.

### Query

```
SELECT * FROM employees WHERE PHONE_NUMBER REGEXP '4569$'
-- Pattern end with-----^
```

### NOT REGEXP

Select all employees whose FIRST\_NAME *does not* start with **N**.

### Query

```
SELECT * FROM employees WHERE FIRST_NAME NOT REGEXP '^N'
-- Pattern does not start with-----^
```

正则表达式包含

选择所有姓氏包含in且名字包含a的员工。

查询

```
SELECT * FROM employees WHERE FIRST_NAME REGEXP 'a' AND LAST_NAME REGEXP 'in'
-- 无 ^ 或 $, 模式可出现在任意位置 -----^
```

[ ] 中的任意字符

选择所有名字以A、B或C开头的员工。

查询

```
SELECT * FROM employees WHERE FIRST_NAME REGEXP '^[ABC]'
-----^^_--^
```

模式或 |

选择所有FIRST\_NAME以A或B或C开头且以 r、 e或 i结尾的员工。

查询

```
SELECT * FROM employees WHERE FIRST_NAME REGEXP '^[ABC]||[rei]$$'
-- -----^^_--^^_--^^
```

计算正则表达式匹配次数

考虑以下查询：

```
SELECT FIRST_NAME, FIRST_NAME REGEXP '^N' as matching FROM employees
```

FIRST\_NAME REGEXP '^N' 的值为 1 或 0，取决于 FIRST\_NAME 是否匹配 ^N。

为了更好地理解：

```
查询
FIRST_NAME,
IF(FIRST_NAME REGEXP '^N', '匹配 ^N', '不匹配 ^N') as matching
FROM employees
```

最后，统计匹配和不匹配行的总数，使用：

```
查询
IF(FIRST_NAME REGEXP '^N', '匹配 ^N', '不匹配 ^N') 作为 matching,
COUNT(*)
FROM employees
GROUP BY matching
```

Regex Contain

Select all employees whose LAST\_NAME contains **in** and whose FIRST\_NAME contains a.

Query

```
SELECT * FROM employees WHERE FIRST_NAME REGEXP 'a' AND LAST_NAME REGEXP 'in'
-- No ^ or $, pattern can be anywhere -----^
```

Any character between [ ]

Select all employees whose FIRST\_NAME starts with **A** or **B** or **C**.

Query

```
SELECT * FROM employees WHERE FIRST_NAME REGEXP '^[ABC]'
-----^^_--^
```

Pattern or |

Select all employees whose FIRST\_NAME starts with **A** or **B** or **C** and ends with **r**, **e**, or **i**.

Query

```
SELECT * FROM employees WHERE FIRST_NAME REGEXP '^[ABC]||[rei]$$'
-- -----^^_--^^_--^^
```

Counting regular expression matches

Consider the following query:

```
SELECT FIRST_NAME, FIRST_NAME REGEXP '^N' as matching FROM employees
```

FIRST\_NAME REGEXP '^N' is 1 or 0 depending on the fact that FIRST\_NAME matches ^N.

To visualize it better:

```
SELECT
FIRST_NAME,
IF(FIRST_NAME REGEXP '^N', 'matches ^N', 'does not match ^N') as matching
FROM employees
```

Finally, count total number of matching and non-matching rows with:

```
SELECT
IF(FIRST_NAME REGEXP '^N', 'matches ^N', 'does not match ^N') as matching,
COUNT(*)
FROM employees
GROUP BY matching
```

# 第24章：视图

参数	详细信息
view_name	视图名称
	SELECT语句，SQL语句将被封装在视图中。它可以是从一个或多个表中获取数据的SELECT语句。

## 第24.1节：创建视图

**权限**

CREATE VIEW 语句需要对视图具有 CREATE VIEW 权限，并且对 SELECT 语句所选的每一列具有某种权限。对于在 SELECT 语句其他部分使用的列，必须具有 SELECT 权限。如果存在 OR REPLACE 子句，还必须对视图具有 DROP 权限。根据本节后文描述，CREATE VIEW 可能还需要 SUPER 权限，具体取决于 DEFINER 的值。

当引用视图时，会进行权限检查。

视图属于某个数据库。默认情况下，新视图会创建在默认数据库中。要显式地在指定数据库中创建视图，请使用完全限定名。

例如：

db\_name.view\_name

```
mysql> CREATE VIEW test.v AS SELECT * FROM t;
```

注意 - 在同一数据库内，基表和视图共享同一命名空间，因此基表和视图不能同名。

视图可以：

- 由多种类型的 SELECT 语句创建
- 引用基表或其他视图
- 使用连接、UNION 和子查询
- SELECT 语句甚至可以不引用任何表

### 另一个示例

以下示例定义了一个视图，该视图从另一个表中选择两列以及基于这些列计算的表达式：

```
mysql> 创建表 t (数量 INT, 价格 INT);
mysql> 插入到 t 值(3, 50);
mysql> 创建视图 v 作为选择 数量, 价格, 数量*价格 作为值 从 t;
mysql> 选择 * 从 v;
```

```
+-----+-----+-----+
| 数量 | 价格 | 值   |
+-----+-----+-----+
|    3 |    50 | 150 |
+-----+-----+-----+
```

限制

# Chapter 24: VIEW

Parameters	Details
view_name	Name of View
SELECT statement	SQL statements to be packed in the views. It can be a SELECT statement to fetch data from one or more tables.

## Section 24.1: Create a View

**Privileges**

The CREATE VIEW statement requires the CREATE VIEW privilege for the view, and some privilege for each column selected by the SELECT statement. For columns used elsewhere in the SELECT statement, you must have the SELECT privilege. If the OR REPLACE clause is present, you must also have the DROP privilege for the view. CREATE VIEW might also require the SUPER privilege, depending on the DEFINER value, as described later in this section.

When a view is referenced, privilege checking occurs.

A view belongs to a database. By default, a new view is created in the default database. To create the view explicitly in a given database, use a fully qualified name

For Example:

db\_name.view\_name

```
mysql> CREATE VIEW test.v AS SELECT * FROM t;
```

Note - Within a database, base tables and views share the same namespace, so a base table and a view cannot have the same name.

A VIEW can:

- be created from many kinds of SELECT statements
- refer to base tables or other views
- use joins, UNION, and subqueries
- SELECT need not even refer to any tables

### Another Example

The following example defines a view that selects two columns from another table as well as an expression calculated from those columns:

```
mysql> CREATE TABLE t (qty INT, price INT);
mysql> INSERT INTO t VALUES(3, 50);
mysql> CREATE VIEW v AS SELECT qty, price, qty*price AS value FROM t;
mysql> SELECT * FROM v;
```

```
+-----+-----+-----+
| qty  | price | value |
+-----+-----+-----+
|    3 |    50 |  150 |
+-----+-----+-----+
```

Restrictions

- 在 MySQL 5.7.7 之前，SELECT 语句的 FROM 子句中不能包含子查询。
  - SELECT 语句不能引用系统变量或用户定义变量。
  - 在存储程序中，SELECT 语句不能引用程序参数或局部变量。
  - SELECT 语句不能引用预处理语句参数。
  - 定义中引用的任何表或视图必须存在。视图创建后，可以删除一个表或视图，前提是定义所指的内容。在这种情况下，使用视图会导致错误。要检查视图定义中此类问题，请使用 CHECK TABLE 语句。
  - 定义不能引用临时表，也不能创建临时视图。
- 
- 不能将触发器与视图关联。
  - SELECT 语句中列名的别名会根据最大列长度 64 个字符进行检查（而非最大别名长度 256 个字符）。
  - 视图（VIEW）可能会优化，也可能不会像等效的 SELECT 那样优化。它不太可能优化得更好。

## 第 24.2 节：来自两张表的视图

当视图可以用来从多个表中提取数据时，它最为有用。

```
CREATE VIEW myview AS
SELECT a.*, b.extra_data FROM main_table a
LEFT OUTER JOIN other_table b
ON a.id = b.id
```

在 MySQL 中，视图不是物化的。如果你现在执行简单查询SELECT \* FROM myview，MySQL 实际上会在后台执行 LEFT JOIN。

视图一旦创建，可以与其他视图或表进行连接

## 第24.3节：删除视图

-- 在当前数据库中创建并删除视图。

```
CREATE VIEW few_rows_from_t1 AS SELECT * FROM t1 LIMIT 10;
DROP VIEW few_rows_from_t1;
```

-- 创建并删除引用不同数据库中表的视图。

```
CREATE VIEW table_from_other_db AS SELECT x FROM db1.foo WHERE x IS NOT NULL;
DROP VIEW table_from_other_db;
```

## 第24.4节：通过视图更新表

视图VIEW的行为非常类似于表。虽然你可以UPDATE表，但你可能也可能不能通过视图更新该表。一般来说，如果视图中的SELECT足够复杂以至于需要临时表，那么UPDATE是不允许的。

像GROUP BY、UNION、HAVING、DISTINCT以及某些子查询会阻止视图可更新。  
[详情见参考手册。](#)

- Before MySQL 5.7.7, the SELECT statement cannot contain a subquery in the FROM clause.
- The SELECT statement cannot refer to system variables or user-defined variables.
- Within a stored program, the SELECT statement cannot refer to program parameters or local variables.
- The SELECT statement cannot refer to prepared statement parameters.
- Any table or view referred to in the definition must exist. After the view has been created, it is possible to drop a table or view that the definition refers to. In this case, use of the view results in an error. To check a view definition for problems of this kind, use the CHECK TABLE statement.
- The definition cannot refer to a TEMPORARY table, and you cannot create a TEMPORARY view.
- You cannot associate a trigger with a view.
- Aliases for column names in the SELECT statement are checked against the maximum column length of 64 characters (not the maximum alias length of 256 characters).
- A **VIEW** may or may not optimize as well as the equivalent **SELECT**. It is unlikely to optimize any better.

## Section 24.2: A view from two tables

A view is most useful when it can be used to pull in data from more than one table.

```
CREATE VIEW myview AS
SELECT a.*, b.extra_data FROM main_table a
LEFT OUTER JOIN other_table b
ON a.id = b.id
```

In mysql views are not materialized. If you now perform the simple query **SELECT \* FROM** myview, mysql will actually perform the LEFT JOIN behind the scene.

A view once created can be joined to other views or tables

## Section 24.3: DROPPING A VIEW

-- Create and drop a view in the current database.

```
CREATE VIEW few_rows_from_t1 AS SELECT * FROM t1 LIMIT 10;
DROP VIEW few_rows_from_t1;
```

-- Create and drop a view referencing a table in a different database.

```
CREATE VIEW table_from_other_db AS SELECT x FROM db1.foo WHERE x IS NOT NULL;
DROP VIEW table_from_other_db;
```

## Section 24.4: Updating a table via a VIEW

A **VIEW** acts very much like a table. Although you can **UPDATE** a table, you may or may not be able to update a view into that table. In general, if the **SELECT** in the view is complex enough to require a temp table, then **UPDATE** is not allowed.

Things like **GROUP BY**, **UNION**, **HAVING**, **DISTINCT**, and some subqueries prevent the view from being updatable. Details in [reference manual](#).



# 第25章：表的创建

## 第25.1节：带主键的表创建

```
CREATE TABLE Person (  
  PersonID      INT UNSIGNED NOT NULL,  
  LastName      VARCHAR(66) NOT NULL,  
  FirstName     VARCHAR(66),  
  Address       VARCHAR(255),  
  City          VARCHAR(66),  
  PRIMARY KEY  (PersonID)  
);
```

主键 是一个 NOT NULL的单列或多列标识符，用于唯一标识表中的一行。会创建一个索引，且如果未显式声明为 NOT NULL，MySQL会自动且隐式地将其声明为NOT NULL。

一个表只能有一个 PRIMARY KEY，且建议每个表都应有一个。InnoDB会在缺失时自动创建一个（如MySQL文档所示），尽管这并不理想。

通常， AUTO\_INCREMENT INT，也称为“代理键”，用于轻量级索引优化和与其他表的关联。该值通常在添加新记录时递增1，起始默认值为1。

然而，尽管名称如此，其目的并非保证值递增，仅保证其顺序性和唯一性。

如果表中所有行被删除，自动递增的 INT值不会重置为默认起始值，除非使用 TRUNCATE TABLE语句截断表。

### 定义某一列为主键（内联定义）

如果主键由单个列组成，PRIMARY KEY 子句可以与列定义放在同一行：

```
CREATE TABLE Person (  
  PersonID      INT UNSIGNED NOT NULL PRIMARY KEY,  
  LastName      VARCHAR(66) NOT NULL,  
  FirstName     VARCHAR(66),  
  Address       VARCHAR(255),  
  City          VARCHAR(66)  
);
```

这种命令形式更简短且易于阅读。

### 定义多列主键

也可以定义由多个列组成的主键。例如，这可能用于外键关系的子表。多列主键通过在单独的PRIMARY KEY子句中列出参与的列来定义。这里不允许使用内联语法，因为内联中只能声明一个列为PRIMARY KEY。例如：

```
CREATE TABLE invoice_line_items (  
  LineNum      SMALLINT UNSIGNED NOT NULL,  
  InvoiceNum    INT UNSIGNED NOT NULL,  
  -- 其他列在此处
```

# Chapter 25: Table Creation

## Section 25.1: Table creation with Primary Key

```
CREATE TABLE Person (  
  PersonID      INT UNSIGNED NOT NULL,  
  LastName      VARCHAR(66) NOT NULL,  
  FirstName     VARCHAR(66),  
  Address       VARCHAR(255),  
  City          VARCHAR(66),  
  PRIMARY KEY  (PersonID)  
);
```

A **primary key** is a **NOT NULL** single or a multi-column identifier which uniquely identifies a row of a table. An index is created, and if not explicitly declared as **NOT NULL**, MySQL will declare them so silently and implicitly.

A table can have only one **PRIMARY KEY**, and each table is recommended to have one. InnoDB will automatically create one in its absence, (as seen in [MySQL documentation](#)) though this is less desirable.

Often, an **AUTO\_INCREMENT INT** also known as "surrogate key", is used for thin index optimization and relations with other tables. This value will (normally) increase by 1 whenever a new record is added, starting from a default value of 1.

However, despite its name, it is not its purpose to guarantee that values are incremental, merely that they are sequential and unique.

An auto-increment **INT** value will not reset to its default start value if all rows in the table are deleted, unless the table is truncated using **TRUNCATE TABLE** statement.

### Defining one column as Primary Key (inline definition)

If the primary key consists of a single column, the **PRIMARY KEY** clause can be placed inline with the column definition:

```
CREATE TABLE Person (  
  PersonID      INT UNSIGNED NOT NULL PRIMARY KEY,  
  LastName      VARCHAR(66) NOT NULL,  
  FirstName     VARCHAR(66),  
  Address       VARCHAR(255),  
  City          VARCHAR(66)  
);
```

This form of the command is shorter and easier to read.

### Defining a multiple-column Primary Key

It is also possible to define a primary key comprising more than one column. This might be done e.g. on the child table of a foreign-key relationship. A multi-column primary key is defined by listing the participating columns in a separate **PRIMARY KEY** clause. Inline syntax is not permitted here, as only one column may be declared **PRIMARY KEY** inline. For example:

```
CREATE TABLE invoice_line_items (  
  LineNum      SMALLINT UNSIGNED NOT NULL,  
  InvoiceNum    INT UNSIGNED NOT NULL,  
  -- Other columns go here
```

```
PRIMARY KEY (InvoiceNum, LineNum),
FOREIGN KEY (InvoiceNum) REFERENCES -- 引用某表的属性
);
```

请注意，主键的列应以逻辑排序顺序指定，这可能与列定义的顺序不同，如上例所示。

较大的索引需要更多的磁盘空间、内存和I/O。因此，键应尽可能小（尤其是复合键）。在InnoDB中，每个“二级索引”都包含一份PRIMARY KEY列的副本。

## 第25.2节：基本表创建

CREATE TABLE 语句用于在MySQL数据库中创建表。

```
CREATE TABLE Person (
  `PersonID`      整数 非 空 主键,
  `LastName`      变长字符串(80),
  `FirstName`     变长字符串(80),
  `Address`       文本,
  `City`          变长字符串(100)
) 存储引擎=InnoDB;
```

每个字段定义必须包含：

- 1. 字段名：一个有效的字段名。确保用反引号`括起来。这可以确保字段名中可以使用例如空格字符。
- 2. 数据类型[长度]：如果字段是CHAR或VARCHAR，则必须指定字段长度。
- 3. 属性 NULL | NOT NULL：如果指定了NOT NULL，则任何尝试在该字段存储NULL值的操作都会失败。
- 4. 查看更多关于数据类型及其属性的信息 [here](#)。\_\_\_\_\_

Engine=... 是一个可选参数，用于指定表的存储引擎。如果未指定存储引擎，表将使用服务器的默认存储引擎创建（通常是 InnoDB 或 MyISAM）。

### 设置默认值

此外，在合适的情况下，您可以使用 DEFAULT 为每个字段设置默认值：

```
CREATE TABLE Address (
  `AddressID`     INTEGER NOT NULL PRIMARY KEY,
  `Street`        VARCHAR(80),
  `City`          VARCHAR(80),
  `Country`       VARCHAR(80) DEFAULT "United States",
  `Active`        BOOLEAN DEFAULT 1,
) Engine=InnoDB;
```

如果在插入过程中未指定Street字段，则检索时该字段将为NULL。插入时如果未指定Country字段，则默认值为“United States”。

您可以为所有列类型设置默认值，except 对于 BLOB、TEXT、GEOMETRY 和 JSON 字段除外。

## 第25.3节：带外键的表创建

```
CREATE TABLE Account (
  AccountID      INT UNSIGNED NOT NULL,
  AccountNo      INT UNSIGNED NOT NULL,
```

```
PRIMARY KEY (InvoiceNum, LineNum),
FOREIGN KEY (InvoiceNum) REFERENCES -- references to an attribute of a table
);
```

Note that the columns of the primary key *should* be specified in logical sort order, which *may* be different from the order in which the columns were defined, as in the example above.

Larger indexes require more disk space, memory, and I/O. Therefore keys should be as small as possible (especially regarding composed keys). In InnoDB, every 'secondary index' includes a copy of the columns of the PRIMARY KEY.

## Section 25.2: Basic table creation

The CREATE TABLE statement is used to create a table in a MySQL database.

```
CREATE TABLE Person (
  `PersonID`      INTEGER NOT NULL PRIMARY KEY,
  `LastName`      VARCHAR(80),
  `FirstName`     VARCHAR(80),
  `Address`       TEXT,
  `City`          VARCHAR(100)
) Engine=InnoDB;
```

Every field definition must have:

- 1. Field name: A valid field Name. Make sure to enclose the names in ` -chars. This ensures that you can use eg space-chars in the fieldname.
- 2. Data type [Length]: If the field is CHAR or VARCHAR, it is mandatory to specify a field length.
- 3. Attributes NULL | NOT NULL: If NOT NULL is specified, then any attempt to store a NULL value in that field will fail.
- 4. See more on data types and their attributes [here](#).

Engine=... is an optional parameter used to specify the table's storage engine. If no storage engine is specified, the table will be created using the server's default table storage engine (usually InnoDB or MyISAM).

### Setting defaults

Additionally, where it makes sense you can set a default value for each field by using DEFAULT:

```
CREATE TABLE Address (
  `AddressID`     INTEGER NOT NULL PRIMARY KEY,
  `Street`        VARCHAR(80),
  `City`          VARCHAR(80),
  `Country`       VARCHAR(80) DEFAULT "United States",
  `Active`        BOOLEAN DEFAULT 1,
) Engine=InnoDB;
```

If during inserts no Street is specified, that field will be NULL when retrieved. When no Country is specified upon insert, it will default to "United States".

You can set default values for all column types, [except](#) for BLOB, TEXT, GEOMETRY, and JSON fields.

## Section 25.3: Table creation with Foreign Key

```
CREATE TABLE Account (
  AccountID      INT UNSIGNED NOT NULL,
  AccountNo      INT UNSIGNED NOT NULL,
```

```
PersonID INT UNSIGNED,  
PRIMARY KEY (AccountID),  
FOREIGN KEY (PersonID) REFERENCES Person (PersonID)  
) ENGINE=InnoDB;
```

外键： 外键（FK）是引用表中的单列或多列复合列。该 FK 确认存在于被引用表中。强烈建议被引用表中确认 FK 的键为主键，但这并非强制要求。它用作对被引用表的快速查找，且不需要唯一，实际上可以是该表的最左索引。

外键关系涉及一个保存核心数据值的父表，以及一个具有相同值指向其父表的子表。FOREIGN KEY 子句在子表中指定。父表和子表必须使用相同的存储引擎，且不能是 TEMPORARY 临时表。

外键和被引用键中的对应列必须具有相似的数据类型。整数类型的大小和符号必须相同。字符串类型的长度不必相同。对于非二进制（字符）字符串列，字符集和排序规则必须相同。

注意： 外键约束仅在 InnoDB 存储引擎下支持（不支持 MyISAM 或 MEMORY）。使用其他引擎的数据库设置将接受此 CREATE TABLE 语句，但不会遵守外键约束。（尽管较新的 MySQL 版本默认使用 InnoDB，但明确指定仍是良好实践。）

## 第25.4节：显示表结构

如果你想查看表的模式信息，可以使用以下任意一种方法：

```
SHOW CREATE TABLE child; -- 选项1  
  
CREATE TABLE `child` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `fullName` varchar(100) NOT NULL,  
  `myParent` int(11) NOT NULL,  
  主键 (`id`),  
  索引 `mommy_daddy` (`myParent`),  
  约束 `mommy_daddy` 外键 (`myParent`) 引用 `parent` (`id`)  
  删除时级联更新时级联  
) ENGINE=InnoDB 默认 字符集=utf8;
```

如果从 mysql 命令行工具使用，输出会更简洁：

```
SHOW CREATE TABLE child \G
```

一种较不具描述性的表格结构展示方式：

```
mysql> 创建表 Tab1(id int, name varchar(30));  
查询成功, 0 行受影响 (0.03 秒)
```

```
mysql> 描述 Tab1; -- 选项 2
```

字段	类型	允许空	键	默认值	额外
id	int(11)	是		NULL	
name	varchar(30)	是		NULL	

```
PersonID INT UNSIGNED,  
PRIMARY KEY (AccountID),  
FOREIGN KEY (PersonID) REFERENCES Person (PersonID)  
) ENGINE=InnoDB;
```

**Foreign key:** A Foreign Key (FK) is either a single column, or multi-column composite of columns, in a *referencing* table. This FK is confirmed to exist in the *referenced* table. It is highly recommended that the *referenced* table key confirming the FK be a Primary Key, but that is not enforced. It is used as a fast-lookup into the *referenced* where it does not need to be unique, and in fact can be a left-most index there.

Foreign key relationships involve a parent table that holds the central data values, and a child table with identical values pointing back to its parent. The FOREIGN KEY clause is specified in the child table. The parent and child tables must use the same storage engine. They must not be [TEMPORARY](#) tables.

Corresponding columns in the foreign key and the referenced key must have similar data types. The size and sign of integer types must be the same. The length of string types need not be the same. For nonbinary (character) string columns, the character set and collation must be the same.

**Note:** foreign-key constraints are supported under the InnoDB storage engine (not MyISAM or MEMORY). DB set-ups using other engines will accept this **CREATE TABLE** statement but will not respect foreign-key constraints. (Although newer MySQL versions default to **InnoDB**, but it is good practice to be explicit.)

## Section 25.4: Show Table Structure

If you want to see the schema information of your table, you can use one of the following:

```
SHOW CREATE TABLE child; -- Option 1  
  
CREATE TABLE `child` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `fullName` varchar(100) NOT NULL,  
  `myParent` int(11) NOT NULL,  
  PRIMARY KEY (`id`),  
  KEY `mommy_daddy` (`myParent`),  
  CONSTRAINT `mommy_daddy` FOREIGN KEY (`myParent`) REFERENCES `parent` (`id`)  
  ON DELETE CASCADE ON UPDATE CASCADE  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

If used from the mysql commandline tool, this is less verbose:

```
SHOW CREATE TABLE child \G
```

A less descriptive way of showing the table structure:

```
mysql> CREATE TABLE Tab1(id int, name varchar(30));  
Query OK, 0 rows affected (0.03 sec)
```

```
mysql> DESCRIBE Tab1; -- Option 2
```

Field	Type	Null	Key	Default	Extra
id	int(11)	YES		NULL	
name	varchar(30)	YES		NULL	

DESCRIBE 和 DESC 都会给出相同的结果。

要一次查看数据库中所有表的 DESCRIBE 执行情况，请参见此 示例。

## 第25.5节：克隆现有表

表可以按如下方式复制：

```
创建表 ClonedPersons LIKE Persons;
```

新表将具有与原表完全相同的结构，包括索引和列属性。

除了手动创建表外，还可以通过从另一个表中选择数据来创建表：

```
创建表 ClonedPersons SELECT * FROM Persons;
```

您可以使用SELECT语句的任何常规功能来修改数据：

```
创建表 ModifiedPersons
SELECT PersonID, FirstName + LastName 作为 FullName 从 Persons
WHERE LastName 不为NULL;
```

使用SELECT创建表时，主键和索引不会被保留。您必须重新声明它们：

```
创建表 ModifiedPersons (主键 (PersonID))
SELECT PersonID, FirstName + LastName 作为 FullName 从 Persons
WHERE LastName 不为NULL;
```

## 第25.6节：创建带有时间戳列以显示最后更新的表

TIMESTAMP列将显示该行的最后更新时间。

```
创建表 `TestLastUpdate` (
  `ID` INT NULL,
  `Name` VARCHAR(50) NULL,
  `Address` VARCHAR(50) NULL,
  `LastUpdate` 时间戳 空 默认 当前时间戳 更新时 当前时间戳
)
COMMENT='最后更新'
;
```

## 第25.7节：从SELECT创建表

您可以通过在CREATE TABLE语句末尾添加SELECT语句，从另一个表创建表：

```
CREATE TABLE stack (
  id_user INT,
  username VARCHAR(30),
  password VARCHAR(30)
);
```

在同一数据库中创建表：

Both **DESCRIBE** and **DESC** gives the same result.

To see **DESCRIBE** performed on all tables in a database at once, see this [Example](#).

## Section 25.5: Cloning an existing table

A table can be replicated as follows:

```
CREATE TABLE ClonedPersons LIKE Persons;
```

The new table will have exactly the same structure as the original table, including indexes and column attributes.

As well as manually creating a table, it is also possible to create table by selecting data from another table:

```
CREATE TABLE ClonedPersons SELECT * FROM Persons;
```

You can use any of the normal features of a **SELECT** statement to modify the data as you go:

```
CREATE TABLE ModifiedPersons
SELECT PersonID, FirstName + LastName AS FullName FROM Persons
WHERE LastName IS NOT NULL;
```

Primary keys and indexes will not be preserved when creating tables from **SELECT**. You must redeclare them:

```
CREATE TABLE ModifiedPersons (PRIMARY KEY (PersonID))
SELECT PersonID, FirstName + LastName AS FullName FROM Persons
WHERE LastName IS NOT NULL;
```

## Section 25.6: Table Create With TimeStamp Column To Show Last Update

The TIMESTAMP column will show when the row was last updated.

```
CREATE TABLE `TestLastUpdate` (
  `ID` INT NULL,
  `Name` VARCHAR(50) NULL,
  `Address` VARCHAR(50) NULL,
  `LastUpdate` TIMESTAMP NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
)
COMMENT='Last Update'
;
```

## Section 25.7: CREATE TABLE FROM SELECT

You can create one table from another by adding a **SELECT** statement at the end of the **CREATE TABLE** statement:

```
CREATE TABLE stack (
  id_user INT,
  username VARCHAR(30),
  password VARCHAR(30)
);
```

Create a table in the same database:

```
-- 从同一数据库中的另一个表创建包含所有属性的表
CREATE TABLE stack2 AS SELECT * FROM stack;
```

```
-- 从同一数据库中的另一个表创建包含部分属性的表
CREATE TABLE stack3 AS SELECT username, password FROM stack;
```

从不同数据库创建表：

```
-- 从另一个数据库中的表创建包含所有属性的表
CREATE TABLE stack2 AS SELECT * FROM second_db.stack;
```

```
-- 从另一个数据库的表创建一个新表，带有部分属性
CREATE TABLE stack3 AS SELECT username, password FROM second_db.stack;
```

注意

要创建一个与另一个数据库中已存在的表相同的表，您需要像这样指定数据库名称：

```
FROM NAME_DATABASE.name_table
```

```
-- create a table from another table in the same database with all attributes
CREATE TABLE stack2 AS SELECT * FROM stack;
```

```
-- create a table from another table in the same database with some attributes
CREATE TABLE stack3 AS SELECT username, password FROM stack;
```

Create tables from different databases:

```
-- create a table from another table from another database with all attributes
CREATE TABLE stack2 AS SELECT * FROM second_db.stack;
```

```
-- create a table from another table from another database with some attributes
CREATE TABLE stack3 AS SELECT username, password FROM second_db.stack;
```

N.B

To create a table same of another table that exist in another database, you need to specifies the name of the database like this:

```
FROM NAME_DATABASE.name_table
```



# 第26章：ALTER TABLE

## 第26.1节：更改存储引擎；重建表；更改file\_per\_table

例如，如果 t1 当前不是InnoDB表，此语句将其存储引擎更改为InnoDB：

```
ALTER TABLE t1 ENGINE = InnoDB;
```

如果表已经是InnoDB，这将重建表及其索引，效果类似于OPTIMIZE TABLE。您可能会获得一些磁盘空间的改善。

如果innodb\_file\_per\_table 的值当前与 t1 创建时生效的值不同，这将转换为（或从）file\_per\_table。

## 第26.2节：修改表的列

```
创建数据库 stackoverflow;

使用 stackoverflow;

创建表 stack(
id_user int 非空,
username varchar(30) 非空,
password varchar(30) 非空
);

修改表 stack 添加列 submit date 非空; -- 添加新列
修改表 stack 删除列 submit; -- 删除列
修改表 stack 修改 submit DATETIME 非空; -- 修改列类型
修改表 stack 更改 submit submit_date DATETIME 非空; -- 更改列类型和名称
修改表 stack 添加列 mod_id INT 非空 在 id_user之后; -- 在现有列之后添加新列
```

## 第26.3节：更改自增值

更改自增值在你不想在大量删除后在AUTO\_INCREMENT列中出现间隙时非常有用。

例如，你的表中有很多不需要的（广告）行，你删除了它们，并且想要修复自增值中的间隙。假设AUTO\_INCREMENT列的最大值现在是100。你可以使用以下语句来修复自增值。

```
修改表 your_table_name AUTO_INCREMENT = 101;
```

## 第26.4节：重命名MySQL表

重命名表可以通过一条命令完成：

```
RENAME TABLE `<旧名称>` TO `<新名称>`;
```

以下语法实现完全相同的功能：

```
ALTER TABLE `<旧名称>` RENAME TO `<新名称>`;
```

# Chapter 26: ALTER TABLE

## Section 26.1: Changing storage engine; rebuild table; change file\_per\_table

For example, if t1 is currently not an InnoDB table, this statement changes its storage engine to InnoDB:

```
ALTER TABLE t1 ENGINE = InnoDB;
```

If the table is already InnoDB, this will rebuild the table and its indexes and have an effect similar to **OPTIMIZE TABLE**. You may gain some disk space improvement.

If the value of innodb\_file\_per\_table is currently different than the value in effect when t1 was built, this will convert to (or from) file\_per\_table.

## Section 26.2: ALTER COLUMN OF TABLE

```
CREATE DATABASE stackoverflow;

USE stackoverflow;

Create table stack(
    id_user int NOT NULL,
    username varchar(30) NOT NULL,
    password varchar(30) NOT NULL
);

ALTER TABLE stack ADD COLUMN submit date NOT NULL; -- add new column
ALTER TABLE stack DROP COLUMN submit; -- drop column
ALTER TABLE stack MODIFY submit DATETIME NOT NULL; -- modify type column
ALTER TABLE stack CHANGE submit submit_date DATETIME NOT NULL; -- change type and name of column
ALTER TABLE stack ADD COLUMN mod_id INT NOT NULL AFTER id_user; -- add new column after existing column
```

## Section 26.3: Change auto-increment value

Changing an auto-increment value is useful when you don't want a gap in an AUTO\_INCREMENT column after a massive deletion.

For example, you got a lot of unwanted (advertisement) rows posted in your table, you deleted them, and you want to fix the gap in auto-increment values. Assume the MAX value of AUTO\_INCREMENT column is 100 now. You can use the following to fix the auto-increment value.

```
ALTER TABLE your_table_name AUTO_INCREMENT = 101;
```

## Section 26.4: Renaming a MySQL table

Renaming a table can be done in a single command:

```
RENAME TABLE `<old name>` TO `<new name>`;
```

The following syntax does exactly the same:

```
ALTER TABLE `<old name>` RENAME TO `<new name>`;
```

如果重命名临时表，必须使用ALTER TABLE版本的语法。

步骤：

- 将上面语句中的<旧名称>和<新名称>替换为相应的值。注意：如果表被移动到不同的数据库，可以对<旧名称>和/或<新名称>使用数据库名.表名的语法。
- 在MySQL命令行或MySQL Workbench等客户端中对相关数据库执行该命令。注意：用户必须对旧表拥有 ALTER 和 DROP 权限，对新表拥有 CREATE 和 INSERT 权限。

## 第26.5节：ALTER 表添加索引

为了提高性能，可能需要为列添加索引

```
ALTER TABLE 表名 ADD INDEX `index_name` (`column_name`)
```

修改以添加复合（多列）索引

```
ALTER TABLE 表名 ADD INDEX `index_name` (`col1`,`col2`)
```

## 第26.6节：更改主键列的数据类型

```
ALTER TABLE fish_data.fish DROP PRIMARY KEY;
ALTER TABLE fish_data.fish MODIFY COLUMN fish_id DECIMAL(20,0) NOT NULL PRIMARY KEY;
```

如果不先删除主键就尝试修改该列的数据类型，将会导致错误。

## 第26.7节：更改列定义

要更改数据库列的定义，例如，如果我们有以下数据库模式，可以使用下面的查询

```
用户 (
  名字 VARCHAR(20),
  姓氏 VARCHAR(20),
  年龄 CHAR(2)
)
```

要将年龄列的类型从char更改为int，我们使用以下查询：

```
ALTER TABLE users CHANGE age age TINYINT UNSIGNED NOT NULL;
```

通用格式为：

```
ALTER TABLE 表名 CHANGE 列名 新列定义
```

## 第26.8节：重命名MySQL数据库

MySQL没有单独的命令来重命名数据库，但可以通过备份和恢复的简单方法来实现这一点：

```
mysqladmin -uroot -p<password> create <新名称>
mysqldump -uroot -p<password> --routines <旧名称> | mysql -uroot -pmypassword <新名称>
mysqladmin -uroot -p<password> drop <旧名称>
```

步骤：

If renaming a temporary table, the ALTER TABLE version of the syntax must be used.

Steps:

- Replace <old name> and <new name> in the line above with the relevant values. *Note: If the table is being moved to a different database, the dbname.tablename syntax can be used for <old name> and/or <new name>.*
- Execute it on the relevant database in the MySQL command line or a client such as MySQL Workbench. *Note: The user must have ALTER and DROP privileges on the old table and CREATE and INSERT on the new one.*

## Section 26.5: ALTER table add INDEX

To improve performance one might want to add indexes to columns

```
ALTER TABLE TABLE_NAME ADD INDEX `index_name` (`column_name`)
```

altering to add composite (multiple column) indexes

```
ALTER TABLE TABLE_NAME ADD INDEX `index_name` (`col1`,`col2`)
```

## Section 26.6: Changing the type of a primary key column

```
ALTER TABLE fish_data.fish DROP PRIMARY KEY;
ALTER TABLE fish_data.fish MODIFY COLUMN fish_id DECIMAL(20,0) NOT NULL PRIMARY KEY;
```

An attempt to modify the type of this column without first dropping the primary key would result in an error.

## Section 26.7: Change column definition

The change the definition of a db column, the query below can be used for example, if we have this db schema

```
users (
  firstname VARCHAR(20),
  lastname VARCHAR(20),
  age CHAR(2)
)
```

To change the type of age column from char to int, we use the query below:

```
ALTER TABLE users CHANGE age age TINYINT UNSIGNED NOT NULL;
```

General format is:

```
ALTER TABLE table_name CHANGE column_name new_column_definition
```

## Section 26.8: Renaming a MySQL database

There is no single command to rename a MySQL database but a simple workaround can be used to achieve this by backing up and restoring:

```
mysqladmin -uroot -p<password> create <new name>
mysqldump -uroot -p<password> --routines <old name> | mysql -uroot -pmypassword <new name>
mysqladmin -uroot -p<password> drop <old name>
```

Steps:

1. 将以上内容复制到文本编辑器中。
2. 替换所有对<old name>、<new name>和<password>的引用（+ 可选地使用root以使用不同的用户）带有相关的值。
3. 在命令行上逐条执行（假设 MySQL 的“bin”文件夹已加入路径，提示时输入“y”）。

替代步骤：

将每个表从一个数据库重命名（移动）到另一个数据库。对每个表执行此操作：

```
RENAME TABLE `<旧数据库>`.`<表名>` TO `<新数据库>`.`<表名>;`
```

你可以通过类似以下方式创建这些语句

```
SELECT CONCAT('RENAME TABLE old_db.', table_name, ' TO ',
              'new_db.', table_name)
FROM information_schema.TABLES
WHERE table_schema = 'old_db';
```

警告。不要试图通过简单地在文件系统中移动文件来进行任何表或数据库的操作。这在早期仅使用 MyISAM 时可以正常工作，但在使用 InnoDB 和表空间的新环境中则不可行。尤其是在“数据字典”从文件系统迁移到系统 InnoDB 表中（可能在下一个主要版本中）时更是如此。移动（而非仅仅 DROP）InnoDB 表的 PARTITION 需要使用“可传输表空间”。在不久的将来，甚至可能没有文件可供操作。

## 第26.9节：交换两个 MySQL 数据库的名称

以下命令可用于交换两个 MySQL 数据库（<db1> 和 <db2>）的名称：

```
mysqladmin -uroot -p<password> create swaptemp
mysqldump -uroot -p<password> --routines <db1> | mysql -uroot -p<password> swaptemp
mysqladmin -uroot -p<password> drop <db1>
mysqladmin -uroot -p<password> create <db1>
mysqldump -uroot -p<password> --routines <db2> | mysql -uroot -p<password> <db1>
mysqladmin -uroot -p<password> drop <db2>
mysqladmin -uroot -p<password> create <db2>
mysqldump -uroot -p<password> --routines swaptemp | mysql -uroot -p<password> <db2>
mysqladmin -uroot -p<password> drop swaptemp
```

步骤：

1. 将以上内容复制到文本编辑器中。
2. 将所有对 <db1>、<db2> 和 <password>（+ 可选的 root 用于使用不同用户）的引用替换为相应的值。相关值。
3. 在命令行上逐条执行（假设 MySQL 的“bin”文件夹已加入路径，提示时输入“y”）。

## 第26.10节：重命名 MySQL 表中的列

重命名列可以通过一条语句完成，但除了新名称外，还必须指定“列定义”（即其数据类型和其他可选属性，如是否允许为空、自增等）。

```
ALTER TABLE `<table name>` CHANGE `<old name>` `<new name>` <column definition>;
```

1. Copy the lines above into a text editor.
2. Replace all references to <old name>, <new name> and <password> (+ optionally root to use a different user) with the relevant values.
3. Execute one by one on the command line (assuming the MySQL "bin" folder is in the path and entering "y" when prompted).

Alternative Steps:

Rename (move) each table from one db to the other. Do this for each table:

```
RENAME TABLE `<old db>`.`<name>` TO `<new db>`.`<name>;`
```

You can create those statements by doing something like

```
SELECT CONCAT('RENAME TABLE old_db.', table_name, ' TO ',
              'new_db.', table_name)
FROM information_schema.TABLES
WHERE table_schema = 'old_db';
```

Warning. Do not attempt to do any sort of table or database by simply moving files around on the filesystem. This worked fine in the old days of just MyISAM, but in the new days of InnoDB and tablespaces, it won't work. Especially when the "Data Dictionary" is moved from the filesystem into system InnoDB tables, probably in the next major release. Moving (as opposed to just DR0Pping) a PARTITION of an InnoDB table requires using "transportable tablespaces". In the near future, there won't even be a file to reach for.

## Section 26.9: Swapping the names of two MySQL databases

The following commands can be used to swap the names of two MySQL databases (<db1> and <db2>):

```
mysqladmin -uroot -p<password> create swaptemp
mysqldump -uroot -p<password> --routines <db1> | mysql -uroot -p<password> swaptemp
mysqladmin -uroot -p<password> drop <db1>
mysqladmin -uroot -p<password> create <db1>
mysqldump -uroot -p<password> --routines <db2> | mysql -uroot -p<password> <db1>
mysqladmin -uroot -p<password> drop <db2>
mysqladmin -uroot -p<password> create <db2>
mysqldump -uroot -p<password> --routines swaptemp | mysql -uroot -p<password> <db2>
mysqladmin -uroot -p<password> drop swaptemp
```

Steps:

1. Copy the lines above into a text editor.
2. Replace all references to <db1>, <db2> and <password> (+ optionally root to use a different user) with the relevant values.
3. Execute one by one on the command line (assuming the MySQL "bin" folder is in the path and entering "y" when prompted).

## Section 26.10: Renaming a column in a MySQL table

Renaming a column can be done in a single statement but as well as the new name, the "column definition" (i.e. its data type and other optional properties such as nullability, auto incrementing etc.) must also be specified.

```
ALTER TABLE `<table name>` CHANGE `<old name>` `<new name>` <column definition>;
```

步骤：

- 1. 打开 MySQL 命令行或 MySQL Workbench 等客户端。
- 2. 运行以下语句：SHOW CREATE TABLE <表名>; (将<表名>替换为相关的值)。
- 3. 记下要重命名的列的整个列定义（即出现在之后的所有内容列名，但在与下一个列名用逗号分隔之前）。
- 4. 将上面行中的 <old name>、<new name> 和 <column definition> 替换为相关的值并然后执行它。

Steps:

- 1. Open the MySQL command line or a client such as MySQL Workbench.
- 2. Run the following statement: **SHOW CREATE TABLE <table name>;** (replacing **<table name>** with the relevant value).
- 3. Make a note of the entire column definition for the column to be renamed (*i.e. everything that appears after the name of the column but before the comma separating it from the next column name*).
- 4. Replace **<old name>**, **<new name>** and **<column definition>** in the line above with the relevant values and then execute it.

# 第27章：删除表

参数	详情
临时的	可选。它指定 DROP TABLE 语句只应删除临时表。
如果存在	可选。如果指定，当某个表不存在时，DROP TABLE 语句不会引发错误。

## 第27.1节：删除表

删除表用于从数据库中删除表。

创建表：

创建一个名为tbl的表，然后删除该表

```
CREATE TABLE tbl(  
  id INT NOT NULL AUTO_INCREMENT,  
  title VARCHAR(100) NOT NULL,  
  author VARCHAR(40) NOT NULL,  
  submission_date DATE,  
  PRIMARY KEY (id)  
);
```

删除表：

DROP TABLE tbl;

请注意

删除表将完全从数据库中删除该表及其所有信息，且无法恢复。

## 第27.2节：从数据库中删除表

DROP TABLE Database.table\_name

# Chapter 27: Drop Table

Parameters	Details
TEMPORARY	Optional. It specifies that only temporary tables should be dropped by the DROP TABLE statement.
IF EXISTS	Optional. If specified, the DROP TABLE statement will not raise an error if one of the tables does not exist.

## Section 27.1: Drop Table

Drop Table is used to delete the table from database.

Creating Table:

Creating a table named tbl and then deleting the created table

```
CREATE TABLE tbl(  
  id INT NOT NULL AUTO_INCREMENT,  
  title VARCHAR(100) NOT NULL,  
  author VARCHAR(40) NOT NULL,  
  submission_date DATE,  
  PRIMARY KEY (id)  
);
```

Dropping Table:

DROP TABLE tbl;

PLEASE NOTE

Dropping table will completely delete the table from the database and all its information, and it will not be recovered.

## Section 27.2: Drop tables from database

DROP TABLE Database.table\_name



# 第28章：MySQL 锁表

## 第28.1节：行级锁定

如果表使用InnoDB，MySQL会自动使用行级锁定，使多个事务可以同时同一表进行读写，而不会相互等待。

如果两个事务尝试修改同一行且都使用行级锁定，其中一个事务会等待另一个事务完成。

行级锁定也可以通过对预期修改的每一行使用SELECT ... FOR UPDATE语句来实现。

考虑两个连接以详细说明行级锁定

连接1

```
START TRANSACTION;
SELECT ledgerAmount FROM accDetails WHERE id = 1 FOR UPDATE;
```

在连接1中，通过SELECT ... FOR UPDATE语句获得了行级锁。

连接2

```
UPDATE accDetails SET ledgerAmount = ledgerAmount + 500 WHERE id=1;
```

当有人尝试在连接2中更新同一行时，将等待连接1完成事务，或者根据innodb\_lock\_wait\_timeout设置（默认50秒）显示错误消息。

```
错误代码：1205。锁等待超时；请尝试重新启动事务
```

要查看此锁的详细信息，请运行SHOW ENGINE INNODB STATUS

```
---事务1973004，活动7秒，正在更新
使用的MySQL表1个，锁定1个
锁等待2个锁结构，堆大小360，1个行锁
MySQL线程ID 4，操作系统线程句柄0x7f996beac700，查询ID 30 localhost root 更新
UPDATE accDetails SET ledgerAmount = ledgerAmount + 500 WHERE id=1
----- 事务已等待7秒以获取此锁：
```

连接2

```
UPDATE accDetails SET ledgerAmount = ledgerAmount + 250 WHERE id=2;
```

```
影响了1行
```

但是，在连接2中更新其他某些行时，将不会出现任何错误。

连接1

```
UPDATE accDetails SET ledgerAmount = ledgerAmount + 750 WHERE id=1;
COMMIT;
```

# Chapter 28: MySQL LOCK TABLE

## Section 28.1: Row Level Locking

If the tables use InnoDB, MySQL automatically uses row level locking so that multiple transactions can use same table simultaneously for read and write, without making each other wait.

If two transactions trying to modify the same row and both uses row level locking, one of the transactions waits for the other to complete.

Row level locking also can be obtained by using **SELECT ... FOR UPDATE** statement for each rows expected to be modified.

Consider two connections to explain Row level locking in detail

Connection 1

```
START TRANSACTION;
SELECT ledgerAmount FROM accDetails WHERE id = 1 FOR UPDATE;
```

In connection 1, row level lock obtained by **SELECT ... FOR UPDATE** statement.

Connection 2

```
UPDATE accDetails SET ledgerAmount = ledgerAmount + 500 WHERE id=1;
```

When some one try to update same row in connection 2, that will wait for connection 1 to finish transaction or error message will be displayed according to the innodb\_lock\_wait\_timeout setting, which defaults to 50 seconds.

```
Error Code: 1205. Lock wait timeout exceeded; try restarting transaction
```

To view details about this lock, run **SHOW ENGINE INNODB STATUS**

```
---TRANSACTION 1973004, ACTIVE 7 sec updating
mysql tables in use 1, locked 1
LOCK WAIT 2 lock struct(s), heap size 360, 1 row lock(s)
MySQL thread id 4, OS thread handle 0x7f996beac700, query id 30 localhost root update
UPDATE accDetails SET ledgerAmount = ledgerAmount + 500 WHERE id=1
----- TRX HAS BEEN WAITING 7 SEC FOR THIS LOCK TO BE GRANTED:
```

Connection 2

```
UPDATE accDetails SET ledgerAmount = ledgerAmount + 250 WHERE id=2;
```

```
1 row(s) affected
```

But while updating some other row in connection 2 will be executed without any error.

Connection 1

```
UPDATE accDetails SET ledgerAmount = ledgerAmount + 750 WHERE id=1;
COMMIT;
```

影响了1行

现在行锁已释放，因为连接1中的事务已提交。

连接2

UPDATE accDetails SET ledgerAmount = ledgerAmount + 500 WHERE id=1;

影响了1行

在连接1通过完成事务释放行锁后，连接2中的更新操作无任何错误地执行。

## 第28.2节：MySQL锁

表锁对于ENGINE=MyISAM可能是一个重要工具，但对于ENGINE=InnoDB则很少有用。如果你想在InnoDB中使用表锁，应该重新考虑你的事务处理方式。

MySQL允许客户端会话显式获取表锁，以便与其他会话协作访问表，或在会话需要独占访问表期间防止其他会话修改表。一个会话只能为自己获取或释放锁，不能为其他会话获取锁或释放其他会话持有的锁。

锁可以用来模拟事务，或在更新表时提高速度。本节后面将对此进行更详细的说明。

命令：**LOCK TABLES** table\_name **READ|WRITE**;

你只能为单个表分配一种锁类型；

示例（读锁）：

**锁定表 table\_name 读取;**

示例（写锁）：

**锁定表 table\_name 写入;**

要查看锁是否已应用，请使用以下命令

**显示 打开 表;**

要刷新/移除所有锁，请使用以下命令：

**解锁表;**

示例：

**锁定表 products 写入：**  
**插入到 products(id,product\_name) 选择 id,old\_product\_name 从 old\_products;**  
**解锁表;**

上述示例中，任何外部连接在解锁产品表之前都无法向产品表写入任何数据

1 row(s) affected

Now row lock is released, because transaction is committed in Connection 1.

Connection 2

**UPDATE** accDetails **SET** ledgerAmount = ledgerAmount + 500 **WHERE** id=1;

1 row(s) affected

The update is executed without any error in Connection 2 after Connection 1 released row lock by finishing the transaction.

## Section 28.2: Mysql Locks

Table locks can be an important tool for **ENGINE=MyISAM**, but are rarely useful for **ENGINE=InnoDB**. If you are tempted to use table locks with InnoDB, you should rethink how you are working with transactions.

MySQL enables client sessions to acquire table locks explicitly for the purpose of cooperating with other sessions for access to tables, or to prevent other sessions from modifying tables during periods when a session requires exclusive access to them. A session can acquire or release locks only for itself. One session cannot acquire locks for another session or release locks held by another session.

Locks may be used to emulate transactions or to get more speed when updating tables. This is explained in more detail later in this section.

Command:**LOCK TABLES** table\_name **READ|WRITE**;

you can assign only lock type to a single table;

Example (READ LOCK):

**LOCK TABLES** table\_name **READ**;

Example (WRITE LOCK):

**LOCK TABLES** table\_name **WRITE**;

To see lock is applied or not, use following Command

**SHOW OPEN TABLES;**

To flush/remove all locks, use following command:

**UNLOCK TABLES;**

EXAMPLE:

**LOCK TABLES** products **WRITE**;  
**INSERT INTO** products(id,product\_name) **SELECT** id,old\_product\_name **FROM** old\_products;  
**UNLOCK TABLES;**

Above example any external connection cannot write any data to products table until unlocking table product

示例：

```
锁定表 products 读取：
插入到 products(id,product_name) 选择 id,old_product_name 从 old_products;
解锁表；
```

上述示例中，任何外部连接在解锁 products 表之前都无法读取该表中的任何数据

EXAMPLE:

```
LOCK TABLES products READ:
INSERT INTO products(id,product_name) SELECT id,old_product_name FROM old_products;
UNLOCK TABLES;
```

Above example any external connection cannot read any data from products table until unlocking table product

# 第29章：错误代码

## 第29.1节：错误代码1064：语法错误

```
select LastName, FirstName,
from Person
```

返回消息：

错误代码：1064。您的SQL语法有误；请检查与您的MySQL服务器版本对应的手册，以获取在第2行“from Person”附近使用的正确语法。

收到MySQL的“1064错误”消息意味着查询无法被解析，存在语法错误。换句话说，它无法理解该查询。

错误消息中的引用从MySQL无法解析的查询的第一个字符开始。在此示例中，MySQL无法理解上下文中的from Person。此情况下，from Person前面有一个多余的逗号。该逗号告诉MySQL在SELECT子句中期待另一个列描述。

语法错误总是显示为... near '...'。引号开头的内容非常接近错误所在的位置。要定位错误，请查看引号中的第一个标记和引号前的最后一个标记。

有时你会看到... near "；也就是说，引号中没有内容。这意味着MySQL无法识别的第一个字符正好在语句的末尾或开头。这表明查询中包含不平衡的引号（'或"）或不平衡的括号，或者你没有正确终止语句。

如果是存储过程的情况，可能是你忘记正确使用DELIMITER。

因此，当你遇到错误1064时，查看查询文本，找到错误信息中提到的位置。目视检查该位置附近的查询文本。

如果你请别人帮忙排查错误1064，最好同时提供整个查询文本和错误信息文本。

## 第29.2节：错误代码1175：安全更新

当尝试更新或删除记录时，如果没有包含使用KEY列的WHERE子句，就会出现此错误。

如果仍要执行删除或更新，请输入：

```
SET SQL_SAFE_UPDATES = 0;
```

要重新启用安全模式，请输入：

```
SET SQL_SAFE_UPDATES = 1;
```

## 第29.3节：错误代码1215：无法添加外键

# Chapter 29: Error codes

## Section 29.1: Error code 1064: Syntax error

```
select LastName, FirstName,
from Person
```

Returns message:

Error Code: 1064. You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'from Person' at line 2.

Getting a "1064 error" message from MySQL means the query cannot be parsed without syntax errors. In other words it can't make sense of the query.

The quotation in the error message begins with the first character of the query that MySQL can't figure out how to parse. In this example MySQL can't make sense, in context, of **from** Person. In this case, there's an extra comma immediately before **from** Person. The comma tells MySQL to expect another column description in the **SELECT** clause

A syntax error always says ... near '...' . The thing at the beginning of the quotes is very near where the error is. To locate an error, look at the first token in the quotes and at the last token before the quotes.

Sometimes you will get ... near '' ; that is, nothing in the quotes. That means the first character MySQL can't figure out is right at the end or the beginning of the statement. This suggests the query contains unbalanced quotes (' or ") or unbalanced parentheses or that you did not terminate the statement before correctly.

In the case of a Stored Routine, you may have forgotten to properly use DELIMITER.

So, when you get Error 1064, look at the text of the query, and find the point mentioned in the error message. Visually inspect the text of the query right around that point.

If you ask somebody to help you troubleshoot Error 1064, it's best to provide both the text of the whole query and the text of the error message.

## Section 29.2: Error code 1175: Safe Update

This error appears while trying to update or delete records without including the **WHERE** clause that uses the **KEY** column.

To execute the delete or update anyway - type:

```
SET SQL_SAFE_UPDATES = 0;
```

To enable the safe mode again - type:

```
SET SQL_SAFE_UPDATES = 1;
```

## Section 29.3: Error code 1215: Cannot add foreign key

约束

当表结构不足以支持开发者要求的外键（FK）快速查验时，会出现此错误。

```
CREATE TABLE `gtType` (  
  `type` char(2) NOT NULL,  
  `description` varchar(1000) NOT NULL,  
  PRIMARY KEY (`type`)  
) ENGINE=InnoDB;  
  
CREATE TABLE `getTogethers` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `type` char(2) NOT NULL,  
  `eventDT` datetime NOT NULL,  
  `location` varchar(1000) NOT NULL,  
  PRIMARY KEY (`id`),  
  KEY `fk_gt2type` (`type`), -- 见下面注释1  
  CONSTRAINT `gettogethers_ibfk_1` FOREIGN KEY (`type`) REFERENCES `gtType` (`type`)  
) ENGINE=InnoDB;
```

注1：如果由于下一行中的外键（FK）定义需要，类似这样的键（KEY）将自动创建。开发者可以跳过它，如果有必要，KEY（也称为索引）将被添加。下面在 someOther 中展示了开发者跳过它的示例。

到目前为止一切正常，直到下面的调用。

```
创建表 `someOther` (  
  `id` int(11) 非空 自动递增,  
  `someDT` datetime 非空,  
  主键 (`id`),  
  约束 `someOther_dt` 外键 (`someDT`) 引用 `getTogethers` (`eventDT`)  
) 引擎=InnoDB;
```

错误代码：1215。无法添加外键约束

在这种情况下，失败是因为被引用表 *referenced* 表 getTogethers 中缺少索引，无法快速查找一个 eventDT。将在下一条语句中解决。

```
创建索引 `gt_eventdt` 在 getTogethers (`eventDT`);
```

表 getTogethers 已被修改，现在创建 someOther 将成功。

摘自 [MySQL 手册页 使用外键约束（FOREIGN KEY Constraints）](#)：

MySQL 要求外键和被引用键上必须有索引，以便外键检查可以快速进行，且不需要全表扫描。在引用表中，必须有一个索引，其中外键列按相同顺序作为首列列出。如果不存在这样的索引，引用表上会自动创建该索引。

外键中的对应列和被引用键中的列必须具有相似的数据类型。整数类型的大小和符号必须相同。字符串类型的长度不必相同。对于非二进制（字符）字符串列，字符集和排序规则必须相同。

constraint

This error occurs when tables are not adequately structured to handle the speedy lookup verification of Foreign Key (FK) requirements that the developer is mandating.

```
CREATE TABLE `gtType` (  
  `type` char(2) NOT NULL,  
  `description` varchar(1000) NOT NULL,  
  PRIMARY KEY (`type`)  
) ENGINE=InnoDB;  
  
CREATE TABLE `getTogethers` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `type` char(2) NOT NULL,  
  `eventDT` datetime NOT NULL,  
  `location` varchar(1000) NOT NULL,  
  PRIMARY KEY (`id`),  
  KEY `fk_gt2type` (`type`), -- see Note1 below  
  CONSTRAINT `gettogethers_ibfk_1` FOREIGN KEY (`type`) REFERENCES `gtType` (`type`)  
) ENGINE=InnoDB;
```

Note1: a KEY like this will be created automatically if needed due to the FK definition in the line that follows it. The developer can skip it, and the KEY (a.k.a. index) will be added if necessary. An example of it being skipped by the developer is shown below in someOther.

So far so good, until the below call.

```
CREATE TABLE `someOther` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `someDT` datetime NOT NULL,  
  PRIMARY KEY (`id`),  
  CONSTRAINT `someOther_dt` FOREIGN KEY (`someDT`) REFERENCES `getTogethers` (`eventDT`)  
) ENGINE=InnoDB;
```

Error Code: 1215. Cannot add foreign key constraint

In this case it fails due to the lack of an index in the *referenced* table getTogethers to handle the speedy lookup of an eventDT. To be solved in next statement.

```
CREATE INDEX `gt_eventdt` ON getTogethers (`eventDT`);
```

Table getTogethers has been modified, and now the creation of someOther will succeed.

From the MySQL Manual Page [Using FOREIGN KEY Constraints](#):

MySQL requires indexes on foreign keys and referenced keys so that foreign key checks can be fast and not require a table scan. In the referencing table, there must be an index where the foreign key columns are listed as the first columns in the same order. Such an index is created on the referencing table automatically if it does not exist.

Corresponding columns in the foreign key and the referenced key must have similar data types. The size and sign of integer types must be the same. The length of string types need not be the same. For nonbinary (character) string columns, the character set and collation must be the same.



InnoDB 允许外键引用任何索引列或列组。然而，在被引用的表中，必须存在一个索引，其中被引用的列按相同顺序作为最前面的列列出。

注意上述关于最前面（最左侧）列的最后一点，以及没有主键要求（尽管强烈建议）的说明。

在成功创建一个引用（子）表后，任何自动为您创建的键都可以通过如下命令查看：

```
SHOW CREATE TABLE someOther;
```

如上文文档所述，出现此错误的其他常见情况包括，但应特别指出：

- 看似微不足道的INT有符号与INT无符号之间的差异。
- 开发人员难以理解多列（复合）键和最前面（最左侧）排序要求。

## 第29.4节：1067、1292、1366、1411 - 数字、日期、默认值等的无效值

**1067** 这可能与TIMESTAMP默认值有关，默认值随着时间发生了变化。详见TIMESTAMP默认值相关内容。日期和时间页面。（尚不存在）

**1292/1366 双精度/整数** 检查是否有字母或其他语法错误。检查列是否对齐；也许你认为你放入的是**VARCHAR**，但它与数字列对齐。

1292 日期时间 检查是否过于早或过于晚。检查是否在夏令时变更当天凌晨2点到3点之间。检查语法错误，例如+00时区相关内容。

1292 变量 检查你尝试设置的变量允许的值。

1292 加载数据 查看“错误”所在的行。检查转义符等。查看数据类型。

1411 STR\_TO\_DATE 日期格式错误？

## 第29.5节：1045 拒绝访问

参见“授权”和“恢复root密码”中的讨论。

## 第29.6节：1236 复制中的“不可能位置”

通常这意味着主服务器崩溃且sync\_binlog关闭。解决方法是在从服务器上**更改主服务器到**下一个二进制日志文件的POS=**0**（查看主服务器）。

原因：主服务器在刷新其二进制日志（binlog）之前将复制项发送给从服务器（当 sync\_binlog=OFF时）。如果主服务器在刷新之前崩溃，从服务器已经在逻辑上越过了二进制日志的文件末尾。当主服务器重新启动时，它会开始一个新的二进制日志，因此切换到该二进制日志的开头是可用的最佳解决方案。

一个更长期的解决方案是 sync\_binlog=ON，如果你能承受它带来的额外I/O开销。

InnoDB permits a foreign key to reference any index column or group of columns. However, in the referenced table, there must be an index where the referenced columns are listed as the first columns in the same order.

Note that last point above about first (left-most) columns and the lack of a Primary Key requirement (though highly advised).

Upon successful creation of a *referencing* (child) table, any keys that were automatically created for you are visible with a command such as the following:

```
SHOW CREATE TABLE someOther ;
```

Other common cases of experiencing this error include, as mentioned above from the docs, but should be highlighted:

- Seemingly trivial differences in **INT** which is signed, pointing toward **INT UNSIGNED**.
- Developers having trouble understanding multi-column (composite) KEYS and first (left-most) ordering requirements.

## Section 29.4: 1067, 1292, 1366, 1411 - Bad Value for number, date, default, etc

**1067** This is probably related to **TIMESTAMP** defaults, which have changed over time. See **TIMESTAMP** defaults in the Dates & Times page. (which does not exist yet)

**1292/1366 DOUBLE/Integer** Check for letters or other syntax errors. Check that the columns align; perhaps you think you are putting into a **VARCHAR** but it is aligned with a numeric column.

**1292 DATETIME** Check for too far in past or future. Check for between 2am and 3am on a morning when Daylight savings changed. Check for bad syntax, such as **+00** timezone stuff.

**1292 VARIABLE** Check the allowed values for the VARIABLE you are trying to **SET**.

**1292 LOAD DATA** Look at the line that is 'bad'. Check the escape symbols, etc. Look at the datatypes.

**1411 STR\_TO\_DATE** Incorrectly formatted date?

## Section 29.5: 1045 Access denied

See discussions in "GRANT" and "Recovering root password".

## Section 29.6: 1236 "impossible position" in Replication

Usually this means that the Master crashed and that sync\_binlog was OFF. The solution is to **CHANGE MASTER to** POS=**0** of the next binlog file (see the Master) on the Slave.

The cause: The Master sends replication items to the Slave before flushing to its binlog (when sync\_binlog=OFF). If the Master crashes before the flush, the Slave has already logically moved past the end of file on the binlog. When the Master starts up again, it starts a new binlog, so **CHANGEing** to the beginning of that binlog is the best available solution.

A longer term solution is sync\_binlog=**ON**, if you can afford the extra I/O that it causes.

(如果你正在使用GTID， .....?)

## 第29.7节：2002，2003 无法连接

检查是否有防火墙阻止了3306端口。

一些可能的诊断和/或解决方案

- 服务器实际上是否正在运行？
- 执行 "service firewalld stop" 和 "systemctl disable firewalld"
- 然后 telnet master 3306
- 检查bind-address
- 检查 skip-name-resolve
- 检查socket。

## 第29.8节：126，127，134，144，145

当你尝试访问MySQL数据库中的记录时，可能会收到这些错误信息。这些错误信息是由于MySQL数据库损坏引起的。以下是类型

MySQL错误代码126 = 索引文件损坏  
MySQL错误代码127 = 记录文件损坏  
MySQL错误代码134 = 记录已被删除（或记录文件损坏）  
MySQL错误代码144 = 表损坏且上次修复失败  
MySQL错误代码145 = 表被标记为损坏，应进行修复

MySQL错误、病毒攻击、服务器崩溃、不当关闭、表损坏是导致此损坏的原因。当它损坏时，将无法访问，您将无法再访问它们。为了恢复访问，最好的方法是从更新的备份中恢复数据。然而，如果您没有更新的或任何有效的备份，那么您可以尝试MySQL修复。

如果表的引擎类型是MyISAM，先执行CHECK TABLE，然后执行REPAIR TABLE。

然后认真考虑转换为InnoDB，这样此错误就不会再次发生。

### 语法

CHECK TABLE <表名> ////检查数据库损坏程度  
REPAIR TABLE <表名> ////修复表

## 第29.9节：139

错误139可能意味着表定义中的字段数量和大小超过某些限制。解决方法：

- 重新考虑模式
- 规范化某些字段
- 垂直分割表

## 第29.10节：1366

这通常意味着客户端和服务端之间的字符集处理不一致。有关更多帮助，请参见...

(If you are running with GTID, ...?)

## Section 29.7: 2002, 2003 Cannot connect

Check for a Firewall issue blocking port 3306.

Some possible diagnostics and/or solutions

- Is the server actually running?
- "service firewalld stop" and "systemctl disable firewalld"
- telnet master 3306
- Check the bind-address
- check skip-name-resolve
- check the socket.

## Section 29.8: 126, 127, 134, 144, 145

When you try access the records from MySQL database, you may get these error messages. These error messages occurred due to corruption in MySQL database. Following are the types

MySQL error code 126 = Index file is crashed  
MySQL error code 127 = Record-file is crashed  
MySQL error code 134 = Record was already deleted (or record file crashed)  
MySQL error code 144 = Table is crashed and last repair failed  
MySQL error code 145 = Table was marked as crashed and should be repaired

MySQL bug, virus attack, server crash, improper shutdown, damaged table are the reason behind this corruption. When it gets corrupted, it becomes inaccessible and you cannot access them anymore. In order to get accessibility, the best way to retrieve data from an updated backup. However, if you do not have updated or any valid backup then you can go for MySQL Repair.

If the table engine type is MyISAM, apply **CHECK TABLE**, then **REPAIR TABLE** to it.

Then think seriously about converting to InnoDB, so this error won't happen again.

### Syntax

CHECK TABLE <table name> ////To check the extent of database corruption  
REPAIR TABLE <table name> ////To repair table

## Section 29.9: 139

Error 139 may mean that the number and size of the fields in the table definition exceeds some limit. Workarounds:

- Re-think the schema
- Normalize some fields
- Vertically partition the table

## Section 29.10: 1366

This usually means that the character set handling was not consistent between client and server. See ... for further assistance.

第29.11节：126，1054，1146，1062，24

（休息一下）包含这4个错误编号后，我认为本页已经涵盖了用户遇到的约50%的典型错误。

（是的，这个“示例”需要修订。）

24 无法打开文件（打开的文件过多）

open\_files\_limit 来源于操作系统设置。table\_open\_cache 需要小于该值。

这些可能导致该错误：

- 在存储过程中未能DEALLOCATE PREPARE。
- 带有大量分区的分区表，并且innodb\_file\_per\_table = ON。建议单个表的分区数不超过50个（出于各种原因）。（当“原生分区”可用时，此建议可能会改变。）

显而易见的解决方法是提高操作系统限制：为了允许更多文件，修改ulimit或/etc/security/limits.conf文件，或者在sysctl.conf（kern.maxfiles 和 kern.maxfilesperproc）中，或其他（取决于操作系统）。然后增加open\_files\_limit和table\_open\_cache。

从5.6.8版本开始，open\_files\_limit会根据max\_connections自动调整，但修改默认值也是可以的。

1062 - 重复条目

此错误主要由以下两个原因引起

- 重复值 - 错误代码：1062。键‘PRIMARY’的重复条目‘12’

主键列是唯一的，不接受重复条目。因此，当你尝试插入一个已经存在于表中的新行时，会产生此错误。

解决方法是将主键列设置为AUTO\_INCREMENT。当你尝试插入新行时，忽略主键列或向主键插入NULL值。

```
CREATE TABLE userDetails(  
  userId INT(10) NOT NULL AUTO_INCREMENT,  
  firstName VARCHAR(50),  
  lastName VARCHAR(50),  
  isActive INT(1) 默认值 0,  
  主键 (userId) );  
  
--->和现在插入时  
INSERT INTO userDetails VALUES (NULL , 'John', 'Doe', 1);
```

- 唯一数据字段 - 错误代码：1062。重复条目‘A’用于键‘code’

您可能将某列设置为唯一，并尝试插入一个该列已存在的值的新行，这将导致此错误。

Section 29.11: 126, 1054, 1146, 1062, 24

(taking a break) With the inclusion of those 4 error numbers, I think this page will have covered about 50% of the typical errors users get.

(Yes, this 'Example' needs revision.)

24 Can't open file (Too many open files)

open\_files\_limit comes from an OS setting. table\_open\_cache needs to be less than that.

These can cause that error:

- Failure to DEALLOCATE PREPARE in a stored procedure.
- PARTITIONed table(s) with a large number of partitions and innodb\_file\_per\_table = ON. Recommend not having more than 50 partitions in a given table (for various reasons). (When "Native Partitions" become available, this advice may change.)

The obvious workaround is to set increase the OS limit: To allow more files, change ulimit or /etc/security/limits.conf or in sysctl.conf (kern.maxfiles & kern.maxfilesperproc) or something else (OS dependent). Then increase open\_files\_limit and table\_open\_cache.

As of 5.6.8, open\_files\_limit is auto-sized based on max\_connections, but it is OK to change it from the default.

1062 - Duplicate Entry

This error occur mainly because of the following two reasons

- Duplicate Value - Error Code: 1062. Duplicate entry '12' for key 'PRIMARY'

The primary key column is unique and it will not accept the duplicate entry. So when you are trying to insert a new row which is already present in you table will produce this error.

To solve this, Set the primary key column as AUTO\_INCREMENT. And when you are trying to insert a new row, ignore the primary key column or insert NULL value to primary key.

```
CREATE TABLE userDetails(  
  userId INT(10) NOT NULL AUTO_INCREMENT,  
  firstName VARCHAR(50),  
  lastName VARCHAR(50),  
  isActive INT(1) DEFAULT 0,  
  PRIMARY KEY (userId) );  
  
--->and now while inserting  
INSERT INTO userDetails VALUES (NULL , 'John', 'Doe', 1);
```

- Unique data field - Error Code: 1062. Duplicate entry 'A' for key 'code'

You may assigned a column as unique and trying to insert a new row with already existing value for that column will produce this error.

为解决此错误，使用INSERT IGNORE代替普通的INSERT。如果您尝试插入的新行没有重复现有记录，MySQL会照常插入。如果记录重复，IGNORE关键字会丢弃该记录而不产生任何错误。

```
INSERT IGNORE INTO userDetails VALUES (NULL , 'John', 'Doe', 1);
```

To overcome this error, use **INSERT IGNORE** instead of normal **INSERT**. If the new row which you are trying to insert doesn't duplicate an existing record, MySQL inserts it as usual. If the record is a duplicate, the **IGNORE** keyword discard it without generating any error.

```
INSERT IGNORE INTO userDetails VALUES (NULL , 'John', 'Doe', 1);
```

第30章：存储例程（存储过程和函数）

参数	详情
RETURNS	指定函数可以返回的数据类型。
返回	遵循RETURN语法的实际变量或值是返回到函数被调用的位置的内容。

第30.1节：带有IN、OUT、INOUT参数的存储过程

```
DELIMITER $$

如果存在则删除存储过程 sp_nested_loop$$
创建存储过程 sp_nested_loop(IN i INT, IN j INT, OUT x INT, OUT y INT, INOUT z INT)
BEGIN
    声明 a INTEGER 默认值 0;
    声明 b INTEGER 默认值 0;
    声明 c INTEGER 默认值 0;
    当 a < i 时执行
        当 b < j 时执行
            设置 c = c + 1;
            设置 b = b + 1;
        结束 WHILE;
        设置 a = a + 1;
        SET b = 0;
    END WHILE;
    SET x = a, y = c;
    SET z = x + y + z;
END $$
DELIMITER ;
```

调用（CALL）存储过程：

```
SET @z = 30;
call sp_nested_loop(10, 20, @x, @y, @z);
SELECT @x, @y, @z;
```

结果：

+-----+-----+-----+
@x     @y     @z
+-----+-----+-----+
10     200     240
+-----+-----+-----+

一个IN参数将值传入过程。过程可能会修改该值，但当过程返回时，修改对调用者不可见。

一个OUT参数将值从过程传回调用者。其初始值在过程内为NULL，且当过程返回时，其值对调用者可见。

一个INOUT参数由调用者初始化，过程可以修改它，且过程所做的任何更改在过程返回时对调用者可见。

Chapter 30: Stored routines (procedures and functions)

Parameter	Details
RETURNS	Specifies the data type that can be returned from a function.
RETURN	Actual variable or value following the RETURN syntax is what is returned to where the function was called from.

Section 30.1: Stored procedure with IN, OUT, INOUT parameters

```
DELIMITER $$

DROP PROCEDURE IF EXISTS sp_nested_loop$$
CREATE PROCEDURE sp_nested_loop(IN i INT, IN j INT, OUT x INT, OUT y INT, INOUT z INT)
BEGIN
    DECLARE a INTEGER DEFAULT 0;
    DECLARE b INTEGER DEFAULT 0;
    DECLARE c INTEGER DEFAULT 0;
    WHILE a < i DO
        WHILE b < j DO
            SET c = c + 1;
            SET b = b + 1;
        END WHILE;
        SET a = a + 1;
        SET b = 0;
    END WHILE;
    SET x = a, y = c;
    SET z = x + y + z;
END $$
DELIMITER ;
```

Invokes (CALL) the stored procedure:

```
SET @z = 30;
call sp_nested_loop(10, 20, @x, @y, @z);
SELECT @x, @y, @z;
```

Result:

+-----+-----+-----+
@x     @y     @z
+-----+-----+-----+
10     200     240
+-----+-----+-----+

An IN parameter passes a value into a procedure. The procedure might modify the value, but the modification is not visible to the caller when the procedure returns.

An OUT parameter passes a value from the procedure back to the caller. Its initial value is NULL within the procedure, and its value is visible to the caller when the procedure returns.

An INOUT parameter is initialized by the caller, can be modified by the procedure, and any change made by the procedure is visible to the caller when the procedure returns.



## 第30.2节：创建函数

下面的（简单）示例函数仅返回常量INT值12。

```
分隔符||
创建函数 functionname()
返回INT
开始
返回12;
结束;
||
DELIMITER ;
```

第一行定义了分隔符字符（分隔符||）将被更改为什么，这需要在创建函数之前设置，否则如果保持默认的;，那么函数体中第一个出现的;将被视为CREATE语句的结束，这通常不是期望的结果。

在CREATE FUNCTION执行完毕后，应将分隔符恢复为默认的;，如上例函数代码后所示（分隔符;）。

执行此功能如下：

```
SELECT functionname();
```

functionname()	
	12

一个稍微复杂一些（但仍然简单）的例子，接受一个参数并加上一个常数：

```
DELIMITER $$
CREATE FUNCTION add_2 ( my_arg INT )
RETURNS INT
开始
RETURN (my_arg + 2);
结束;
$$
DELIMITER ;

SELECT add_2(12);
```

add_2(12)	
	14

注意对DELIMITER指令使用了不同的参数。实际上你可以使用任何不出现在CREATE语句主体中的字符序列，但通常的做法是使用成对的非字母数字字符，如\\、||或\$\$。

在创建或更新函数、存储过程或触发器之前和之后更改分隔符是一个好习惯，因为有些图形界面不要求更改分隔符，而通过命令行运行查询则总是需要设置分隔符。

## Section 30.2: Create a Function

The following (trivial) example function simply returns the constant **INT** value 12.

```
DELIMITER ||
CREATE FUNCTION functionname()
RETURNS INT
BEGIN
    RETURN 12;
END;
||
DELIMITER ;
```

The first line defines what the delimiter character(DELIMITER ||) is to be changed to, this is needed to be set before a function is created otherwise if left it at its default ; then the first ; that is found in the function body will be taken as the end of the **CREATE** statement, which is usually not what is desired.

After the **CREATE FUNCTION** has run you should set the delimiter back to its default of ; as is seen after the function code in the above example (DELIMITER ;).

Execution this function is as follows:

```
SELECT functionname();
```

functionname()	
	12

A slightly more complex (but still trivial) example takes a parameter and adds a constant to it:

```
DELIMITER $$
CREATE FUNCTION add_2 ( my_arg INT )
RETURNS INT
BEGIN
    RETURN (my_arg + 2);
END;
$$
DELIMITER ;

SELECT add_2(12);
```

add_2(12)	
	14

Note the use of a different argument to the DELIMITER directive. You can actually use any character sequence that does not appear in the **CREATE** statement body, but the usual practice is to use a doubled non-alphanumeric character such as \\, || or \$\$.

It is good practice to always change the parameter before and after a function, procedure or trigger creation or update as some GUI's don't require the delimiter to change whereas running queries via the command line always require the delimiter to be set.

第30.3节：游标

游标使您能够逐行遍历查询结果。DECLARE命令用于初始化游标并将其与特定的SQL查询关联：

```
DECLARE student 游标 FOR SELECT name FROM studend;
```

假设我们销售某些类型的产品。我们想统计每种类型的产品有多少。

我们的数据：

```
CREATE TABLE product
(
  id      INT(10) UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
  type    VARCHAR(50)      NOT NULL,
  name    VARCHAR(255)     NOT NULL

);
CREATE TABLE product_type
(
  name    VARCHAR(50) NOT NULL PRIMARY KEY
);
创建表 product_type_count
(
  type    VARCHAR(50)      NOT NULL PRIMARY KEY,
  count   INT(10) UNSIGNED NOT NULL DEFAULT 0
);

INSERT INTO product_type (name) VALUES
  ('dress'),
  ('food');

INSERT INTO product (type, name) VALUES
  ('dress', 'T-shirt'),
  ('dress', 'Trousers'),
  ('food', 'Apple'),
  ('food', 'Tomatoes'),
  ('food', 'Meat');
```

我们可以使用带游标的存储过程来实现该目标：

```
DELIMITER //
DROP PROCEDURE IF EXISTS product_count;
CREATE PROCEDURE product_count()
  开始
  DECLARE p_type VARCHAR(255);
  DECLARE p_count INT(10) UNSIGNED;
  DECLARE done INT DEFAULT 0;
  DECLARE product CURSOR FOR
    SELECT
      类型,
      COUNT(*)
    FROM product
    GROUP BY 类型;
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;

  TRUNCATE product_type;

  OPEN product;
```

Section 30.3: Cursors

Cursors enable you to itterate results of query one by line. **DECLARE** command is used to init cursor and associate it with a specific SQL query:

```
DECLARE student CURSOR FOR SELECT name FROM studend;
```

Let's say we sell products of some types. We want to count how many products of each type are exists.

Our data:

```
CREATE TABLE product
(
  id      INT(10) UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
  type    VARCHAR(50)      NOT NULL,
  name    VARCHAR(255)     NOT NULL

);
CREATE TABLE product_type
(
  name    VARCHAR(50) NOT NULL PRIMARY KEY
);
CREATE TABLE product_type_count
(
  type    VARCHAR(50)      NOT NULL PRIMARY KEY,
  count   INT(10) UNSIGNED NOT NULL DEFAULT 0
);

INSERT INTO product_type (name) VALUES
  ('dress'),
  ('food');

INSERT INTO product (type, name) VALUES
  ('dress', 'T-shirt'),
  ('dress', 'Trousers'),
  ('food', 'Apple'),
  ('food', 'Tomatoes'),
  ('food', 'Meat');
```

We may achieve the goal using stored procedure with using cursor:

```
DELIMITER //
DROP PROCEDURE IF EXISTS product_count;
CREATE PROCEDURE product_count()
  BEGIN
    DECLARE p_type VARCHAR(255);
    DECLARE p_count INT(10) UNSIGNED;
    DECLARE done INT DEFAULT 0;
    DECLARE product CURSOR FOR
      SELECT
        type,
        COUNT(*)
      FROM product
      GROUP BY type;
    DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;

    TRUNCATE product_type;

    OPEN product;
```

```
REPEAT
  FETCH product
  INTO p_type, p_count;
  IF NOT done
  THEN
    INSERT INTO product_type_count
    SET
      type = p_type,
      count = p_count;
  END IF;
UNTIL done
END REPEAT;

CLOSE product;
END //
DELIMITER ;
```

当您可以这样调用存储过程时：

```
CALL product_count();
```

结果将存储在 product\_type\_count 表中：

type	count
dress	2
food	3

虽然这是一个 CURSOR 的好例子，但请注意，整个存储过程的主体可以仅用以下语句替代

```
INSERT INTO product_type_count
  (type, count)
SELECT type, COUNT(*)
FROM product
GROUP BY type;
```

这将运行得更快。

### 第30.4节：多个结果集

与SELECT语句不同，存储过程会返回多个结果集。这需要使用不同的代码来收集Perl、PHP等中CALL的结果。

（这里或其他地方需要具体代码！）

### 第30.5节：创建函数

```
DELIMITER $$
CREATE
  DEFINER='db_username'@'hostname_or_IP'
  FUNCTION `function_name` (optional_param data_type(length_if_applicable))
  RETURNS data_type
开始
  /*
  SQL语句写在这里
  */
END$$
```

```
REPEAT
  FETCH product
  INTO p_type, p_count;
  IF NOT done
  THEN
    INSERT INTO product_type_count
    SET
      type = p_type,
      count = p_count;
  END IF;
UNTIL done
END REPEAT;

CLOSE product;
END //
DELIMITER ;
```

When you may call procedure with:

```
CALL product_count();
```

Result would be in product\_type\_count table:

type	count
dress	2
food	3

While that is a good example of a CURSOR, notice how the entire body of the procedure can be replaced by just

```
INSERT INTO product_type_count
  (type, count)
SELECT type, COUNT(*)
FROM product
GROUP BY type;
```

This will run a lot faster.

### Section 30.4: Multiple ResultSets

Unlike a SELECT statement, a Stored Procedure returns multiple result sets. The requires different code to be used for gathering the results of a CALL in Perl, PHP, etc.

(Need specific code here or elsewhere!)

### Section 30.5: Create a function

```
DELIMITER $$
CREATE
  DEFINER='db_username'@'hostname_or_IP'
  FUNCTION `function_name` (optional_param data_type(length_if_applicable))
  RETURNS data_type
BEGIN
  /*
  SQL Statements goes here
  */
END$$
```

DELIMITER ;

RETURNS 数据类型可以是任何 MySQL 数据类型。

DELIMITER ;

The RETURNS data\_type is any MySQL datatype.

# 第31章：索引和键

## 第31.1节：创建索引

```
-- 为表 'my_table' 中的列 'name' 创建索引
CREATE INDEX idx_name ON my_table(name);
```

## 第31.2节：创建唯一索引

唯一索引防止在表中插入重复数据。NULL值可以插入到构成唯一索引的列中（因为根据定义，NULL值与任何其他值不同，包括另一个NULL值）

```
-- 为表 'my_table' 中的列 'name' 创建唯一索引
CREATE UNIQUE INDEX idx_name ON my_table(name);
```

## 第31.3节：AUTO\_INCREMENT键

```
CREATE TABLE (
id INT UNSIGNED NOT NULL AUTO_INCREMENT,
...
PRIMARY KEY(id),
... );
```

主要说明：

- 当你在INSERT时未指定该值，或指定为NULL时，id从1开始自动递增1。
- 这些id始终彼此不同，但...
- 除了在任何时刻唯一外，不要对id的值做任何假设（如无间隙、连续生成、不重复使用等）。

细节说明：

- 服务器重启时，‘下一个’值被‘计算’为MAX(id)+1。
- 如果关闭或崩溃前的最后一次操作是删除最高的ID，则该ID可能会被重新使用（这取决于引擎）。因此，不要相信auto\_increments是永久唯一的；它们只在任何时刻是唯一的。
- 对于多主机或集群解决方案，请参见auto\_increment\_offset和auto\_increment\_increment。
- 将其他字段设为PRIMARY KEY并仅做INDEX(id)是可以的。（这在某些情况下是一种优化。）
- 将AUTO\_INCREMENT用作"PARTITION键"很少有益；请使用其他方式。
- 各种操作可能“消耗”值。这种情况发生在预分配了值但未使用时：INSERT IGNORE（遇到重复键时）、REPLACE（即DELETE加INSERT）等操作。ROLLBACK也是导致ID出现间隙的原因之一。
- 在复制中，不能保证ID按升序到达从库。虽然ID是按连续顺序分配的，但InnoDB语句是按COMMIT顺序发送到从库的。

## 第31.4节：创建复合索引

这将创建mystring和mydatetime两个键的复合索引，加快同时在WHERE子句中使用这两列的查询速度。

# Chapter 31: Indexes and Keys

## Section 31.1: Create index

```
-- Create an index for column 'name' in table 'my_table'
CREATE INDEX idx_name ON my_table(name);
```

## Section 31.2: Create unique index

A unique index prevents the insertion of duplicated data in a table. NULL values can be inserted in the columns that form part of the unique index (since, by definition, a NULL value is different from any other value, including another NULL value)

```
-- Creates a unique index for column 'name' in table 'my_table'
CREATE UNIQUE INDEX idx_name ON my_table(name);
```

## Section 31.3: AUTO\_INCREMENT key

```
CREATE TABLE (
id INT UNSIGNED NOT NULL AUTO_INCREMENT,
...
PRIMARY KEY(id),
... );
```

Main notes:

- Starts with 1 and increments by 1 automatically when you fail to specify it on INSERT, or specify it as NULL.
- The ids are always distinct from each other, but...
- Do not make any assumptions (no gaps, consecutively generated, not reused, etc) about the values of the id other than being unique at any given instant.

Subtle notes:

- On restart of server, the 'next' value is 'computed' as MAX(id)+1.
- If the last operation before shutdown or crash was to delete the highest id, that id may be reused (this is engine-dependent). So, do not trust auto\_increments to be permanently unique; they are only unique at any moment.
- For multi-master or clustered solutions, see auto\_increment\_offset and auto\_increment\_increment.
- It is OK to have something else as the PRIMARY KEY and simply do INDEX(id). (This is an optimization in some situations.)
- Using the AUTO\_INCREMENT as the "PARTITION key" is rarely beneficial; do something different.
- Various operations may "burn" values. This happens when they pre-allocate value(s), then don't use them: INSERT IGNORE (with dup key), REPLACE (which is DELETE plus INSERT) and others. ROLLBACK is another cause for gaps in ids.
- In Replication, you cannot trust ids to arrive at the slave(s) in ascending order. Although ids are assigned in consecutive order, InnoDB statements are sent to slaves in COMMIT order.

## Section 31.4: Create composite index

This will create a composite index of both keys, mystring and mydatetime and speed up queries with both columns in the WHERE clause.



```
CREATE INDEX idx_mycol_myothercol ON my_table(mycol, myothercol)
```

注意：顺序很重要！如果查询条件中没有同时包含这两列，则只能使用最左前缀索引。在这种情况下，包含mycol的WHERE查询会使用该索引，而仅搜索myothercol且不搜索mycol的查询则不会使用。更多信息请参见这篇博客文章。

注意： 由于BTREE的工作方式，通常以范围查询的列应放在最右侧。  
例如，DATETIME 列通常以 WHERE datecol > '2016-01-01 00:00:00' 的方式查询。BTREE索引能非常高效地处理范围查询，但前提是被范围查询的列必须是复合索引中的最后一列。

## 第31.5节：删除索引

```
-- 删除表 'my_table' 中列 'name' 的索引
DROP INDEX idx_name ON my_table;
```

```
CREATE INDEX idx_mycol_myothercol ON my_table(mycol, myothercol)
```

**Note:** The order is important! If the search query does not include both columns in the WHERE clause, it can only use the leftmost index. In this case, a query with mycol in the WHERE will use the index, a query searching for myothercol without also searching for mycol will not. For more information [check out this blog post](#).

**Note:** Due to the way BTREE's work, columns that are usually queried in ranges should go in the rightmost value. For example, DATETIME columns are usualy queried like WHERE datecol > '2016-01-01 00:00:00'. BTREE indexes handle ranges very efficiently but only if the column being queried as a range is the last one in the composite index.

## Section 31.5: Drop index

```
-- Drop an index for column 'name' in table 'my_table'
DROP INDEX idx_name ON my_table;
```

# 第32章：全文搜索

MySQL提供全文搜索功能。它会在包含文本的列中搜索与单词和短语最匹配的内容。

## 第32.1节：简单全文搜索

```
SET @searchTerm= '数据库编程';
SELECT MATCH (Title) AGAINST (@searchTerm IN NATURAL LANGUAGE MODE) Score,
       ISBN, Author, Title
  来自 书籍
WHERE MATCH (Title) AGAINST (@searchTerm IN NATURAL LANGUAGE MODE)
ORDER BY MATCH (Title) AGAINST (@searchTerm IN NATURAL LANGUAGE MODE) DESC;
```

给定一个名为book的表，包含名为ISBN、'Title'和'Author'的列，这将查找匹配这些词的书籍 'Database Programming'。它会先显示最匹配的结果。

要使其工作，必须在Title列上有全文索引：

```
ALTER TABLE book ADD FULLTEXT INDEX Fulltext_title_index (Title);
```

## 第32.2节：简单的BOOLEAN搜索

```
SET @searchTerm= 'Database Programming -Java';
SELECT MATCH (Title) AGAINST (@searchTerm IN BOOLEAN MODE) Score,
       ISBN, Author, Title
  来自 书籍
WHERE MATCH (Title) AGAINST (@searchTerm IN BOOLEAN MODE)
ORDER BY MATCH (Title) AGAINST (@searchTerm IN BOOLEAN MODE) DESC;
```

给定一个名为book的表，包含名为ISBN、Title和Author的列，这将搜索标题中包含单词 'Database' 和 'Programming' 但不包含单词 'Java' 的书籍。

要使其工作，必须在 Title 列上有全文索引：

```
ALTER TABLE book ADD FULLTEXT INDEX Fulltext_title_index (Title);
```

## 第 32.3 节：多列 FULLTEXT 搜索

```
SET @searchTerm= 'Date Database Programming';
SELECT MATCH (Title, Author) AGAINST (@searchTerm IN NATURAL LANGUAGE MODE) Score,
       ISBN, Author, Title
  来自 书籍
WHERE MATCH (Title, Author) AGAINST (@searchTerm IN NATURAL LANGUAGE MODE)
ORDER BY MATCH (Title, Author) AGAINST (@searchTerm IN NATURAL LANGUAGE MODE) DESC;
```

给定一个名为 book 的表，包含列 ISBN、Title 和 Author，本查询查找匹配“Date Database Programming”词条的书籍。它优先显示最佳匹配。最佳匹配包括由 C. J. Date 教授撰写的书籍。

（但是，最佳匹配之一也是《The Date Doctor's Guide to Dating : How to Get from First Date to Perfect Mate 》。这显示了 FULLTEXT 搜索的一个限制：它并不试图理解诸如词性或索引词的含义等内容。）

要使其工作，必须在 Title 和 Author 列上有全文索引：

# Chapter 32: Full-Text search

MySQL offers FULLTEXT searching. It searches tables with columns containing text for the best matches for words and phrases.

## Section 32.1: Simple FULLTEXT search

```
SET @searchTerm= 'Database Programming';
SELECT MATCH (Title) AGAINST (@searchTerm IN NATURAL LANGUAGE MODE) Score,
       ISBN, Author, Title
  FROM book
WHERE MATCH (Title) AGAINST (@searchTerm IN NATURAL LANGUAGE MODE)
ORDER BY MATCH (Title) AGAINST (@searchTerm IN NATURAL LANGUAGE MODE) DESC;
```

Given a table named book with columns named ISBN, 'Title', and 'Author', this finds books matching the terms 'Database Programming'. It shows the best matches first.

For this to work, a fulltext index on the Title column must be available:

```
ALTER TABLE book ADD FULLTEXT INDEX Fulltext_title_index (Title);
```

## Section 32.2: Simple BOOLEAN search

```
SET @searchTerm= 'Database Programming -Java';
SELECT MATCH (Title) AGAINST (@searchTerm IN BOOLEAN MODE) Score,
       ISBN, Author, Title
  FROM book
WHERE MATCH (Title) AGAINST (@searchTerm IN BOOLEAN MODE)
ORDER BY MATCH (Title) AGAINST (@searchTerm IN BOOLEAN MODE) DESC;
```

Given a table named book with columns named ISBN, Title, and Author, this searches for books with the words 'Database' and 'Programming' in the title, but not the word 'Java'.

For this to work, a fulltext index on the Title column must be available:

```
ALTER TABLE book ADD FULLTEXT INDEX Fulltext_title_index (Title);
```

## Section 32.3: Multi-column FULLTEXT search

```
SET @searchTerm= 'Date Database Programming';
SELECT MATCH (Title, Author) AGAINST (@searchTerm IN NATURAL LANGUAGE MODE) Score,
       ISBN, Author, Title
  FROM book
WHERE MATCH (Title, Author) AGAINST (@searchTerm IN NATURAL LANGUAGE MODE)
ORDER BY MATCH (Title, Author) AGAINST (@searchTerm IN NATURAL LANGUAGE MODE) DESC;
```

Given a table named book with columns named ISBN, Title, and Author, this finds books matching the terms 'Date Database Programming'. It shows the best matches first. The best matches include books written by Prof. C. J. Date.

(But, one of the best matches is also *The Date Doctor's Guide to Dating : How to Get from First Date to Perfect Mate*. This shows up a limitation of FULLTEXT search: it doesn't pretend to understand such things as parts of speech or the meaning of the indexed words.)

For this to work, a fulltext index on the Title and Author columns must be available:

```
ALTER TABLE book ADD FULLTEXT INDEX Fulltext_title_author_index (Title, Author);
```

```
ALTER TABLE book ADD FULLTEXT INDEX Fulltext_title_author_index (Title, Author);
```

# 第33章：PREPARE语句

## 第33.1节：准备、执行和释放预处理语句

[PREPARE](#) 准备一个语句以供执行

[EXECUTE](#) 执行一个预处理语句

[DEALLOCATE PREPARE](#) 释放一个预处理语句

```
SET @s = 'SELECT SQRT(POW(?,2) + POW(?,2)) AS hypotenuse';
PREPARE stmt2 FROM @s;
SET @a = 6;
SET @b = 8;
EXECUTE stmt2 USING @a, @b;
```

结果：

-----
hypotenuse
-----
10
+-----+

最后，

```
DEALLOCATE PREPARE stmt2;
```

注意事项：

- 必须使用 @变量，而不是 DECLARE 声明的变量来FROM @s
- Prepare 等的主要用途是构造查询，用于绑定无法实现的情况，例如插入表名。

## 第33.2节：使用添加列修改表

```
SET v_column_definition := CONCAT(
    v_column_name
    , ' ', v_column_type
    , ' ', v_column_options
);

SET @stmt := CONCAT('ALTER TABLE ADD COLUMN ', v_column_definition);

PREPARE stmt FROM @stmt;
EXECUTE stmt;
DEALLOCATE PREPARE stmt;
```

# Chapter 33: PREPARE Statements

## Section 33.1: PREPARE, EXECUTE and DEALLOCATE PREPARE Statements

[PREPARE](#) prepares a statement for execution

[EXECUTE](#) executes a prepared statement

[DEALLOCATE PREPARE](#) releases a prepared statement

```
SET @s = 'SELECT SQRT(POW(?,2) + POW(?,2)) AS hypotenuse';
PREPARE stmt2 FROM @s;
SET @a = 6;
SET @b = 8;
EXECUTE stmt2 USING @a, @b;
```

Result:

+-----+
hypotenuse
+-----+
10
+-----+

Finally,

```
DEALLOCATE PREPARE stmt2;
```

Notes:

- You must use @variables, not DECLAREd variables for FROM @s
- A primary use for Prepare, etc, is to 'construct' a query for situations where binding will not work, such as inserting the table name.

## Section 33.2: Alter table with add column

```
SET v_column_definition := CONCAT(
    v_column_name
    , ' ', v_column_type
    , ' ', v_column_options
);

SET @stmt := CONCAT('ALTER TABLE ADD COLUMN ', v_column_definition);

PREPARE stmt FROM @stmt;
EXECUTE stmt;
DEALLOCATE PREPARE stmt;
```

# 第34章：JSON

从 MySQL 5.7.8 开始，MySQL 支持原生 JSON 数据类型，能够高效访问 JSON（JavaScript 对象表示法）文档中的数据。<https://dev.mysql.com/doc/refman/5.7/en/json.html>

## 第34.1节：创建带有主键和JSON字段的简单表

```
CREATE TABLE table_name (  
  id INT NOT NULL AUTO_INCREMENT,  
  json_col JSON,  
  PRIMARY KEY(id)  
);
```

## 第34.2节：插入简单的JSON

```
INSERT INTO  
table_name (json_col)  
VALUES  
( '{"City": "加勒", "Description": "世界上最棒的城市"}');
```

这就是最简单的用法，但请注意，由于JSON字典的键必须用双引号括起来，整个内容应使用单引号包裹。如果查询成功，数据将以二进制格式存储。

## 第34.3节：更新JSON字段

在前面的例子中，我们看到如何将混合数据类型插入到 JSON 字段中。如果我们想更新该字段怎么办？我们将向前面例子中名为variations的数组中添加scheveningen。

```
更新  
myjson  
设置  
dict=JSON_ARRAY_APPEND(dict,'$.variations','scheveningen')  
哪里  
id = 2;
```

注意事项：

- 1. 我们json字典中的\$.variations数组。\$符号代表json文档。完整的MySQL 识别的 JSON 路径说明请参阅<https://dev.mysql.com/doc/refman/5.7/en/json-path-syntax.html>
- 2. 由于我们还没有使用 JSON 字段查询的示例，此示例使用主键。

现在如果我们执行 SELECT \* FROM myjson 就会看到

```
| id | dict |  
+---+-----+  
| 2  | {"opening": "西西里防御", "variations": ["佩利坎", "龙式", "纳伊多夫", "斯凯文宁根"]} |  
+---+-----+  
1 行记录 (0.00 秒)
```

# Chapter 34: JSON

As of MySQL 5.7.8, MySQL supports a native JSON data type that enables efficient access to data in JSON (JavaScript Object Notation) documents. <https://dev.mysql.com/doc/refman/5.7/en/json.html>

## Section 34.1: Create simple table with a primary key and JSON field

```
CREATE TABLE table_name (  
  id INT NOT NULL AUTO_INCREMENT,  
  json_col JSON,  
  PRIMARY KEY(id)  
);
```

## Section 34.2: Insert a simple JSON

```
INSERT INTO  
table_name (json_col)  
VALUES  
( '{"City": "Galle", "Description": "Best damn city in the world"}');
```

That's simple as it can get but note that because JSON dictionary keys have to be surrounded by double quotes the entire thing should be wrapped in single quotes. If the query succeeds, the data will be stored in a binary format.

## Section 34.3: Updating a JSON field

In the previous example we saw how mixed data types can be inserted into a JSON field. What if we want to update that field? We are going to add *scheveningen* to the array named variations in the previous example.

```
UPDATE  
myjson  
SET  
dict=JSON_ARRAY_APPEND(dict,'$.variations','scheveningen')  
WHERE  
id = 2;
```

Notes:

- 1. The \$.variations array in our json dictionary. The \$ symbol represents the json documentation. For a full explanation of json paths recognized by mysql refer to <https://dev.mysql.com/doc/refman/5.7/en/json-path-syntax.html>
- 2. Since we don't yet have an example on querying using json fields, this example uses the primary key.

Now if we do SELECT \* FROM myjson we will see

```
+---+-----+  
| id | dict |  
+---+-----+  
| 2  | {"opening": "Sicilian", "variations": ["pelikan", "dragon", "najdorf", "scheveningen"]} |  
+---+-----+  
1 row in set (0.00 sec)
```



## 第34.4节：将混合数据插入JSON字段

这会将一个json字典插入到之前示例中创建的表中，其中一个成员是字符串数组。

```
INSERT INTO myjson(dict)
VALUES({'opening':"西西里防御","variations":["pelikan","dragon","najdorf"]});
```

注意，再次提醒，使用单引号和双引号时需要小心。整个内容必须用单引号包裹。

## 第34.5节：将数据转换为JSON类型

这将有有效的json字符串转换为MySQL的JSON类型：

```
SELECT CAST('[1,2,3]' as JSON) ;
SELECT CAST({'opening':"西西里防御","variations":["pelikan","dragon","najdorf"]} as JSON);
```

## 第34.6节：创建Json对象和数组

JSON\_OBJECT 创建 JSON 对象：

```
SELECT JSON_OBJECT('key1',col1 , 'key2',col2 , 'key3',col3) as myobj;
```

JSON\_ARRAY 也创建 JSON 数组：

```
SELECT JSON_ARRAY(col1,col2,'col3') as myarray;
```

注意：myobj.key3 和 myarray[2] 是作为固定字符串的 "col3"。

还有混合的 JSON 数据：

```
SELECT JSON_OBJECT("opening","Sicilian", "variations",JSON_ARRAY("pelikan","dragon","najdorf"))
as mymixed ;
```

## Section 34.4: Insert mixed data into a JSON field

This inserts a json dictionary where one of the members is an array of strings into the table that was created in another example.

```
INSERT INTO myjson(dict)
VALUES( {'opening':"Sicilian", "variations":["pelikan","dragon","najdorf"]} );
```

Note, once again, that you need to be careful with the use of single and double quotes. The whole thing has to be wrapped in single quotes.

## Section 34.5: CAST data to JSON type

This converts valid json strings to MySQL JSON type:

```
SELECT CAST('[1,2,3]' as JSON) ;
SELECT CAST( '{"opening":"Sicilian", "variations":["pelikan","dragon","najdorf"]}' as JSON);
```

## Section 34.6: Create Json Object and Array

JSON\_OBJECT creates JSON Objects:

```
SELECT JSON_OBJECT('key1',col1 , 'key2',col2 , 'key3','col3') as myobj;
```

JSON\_ARRAY creates JSON Array as well:

```
SELECT JSON_ARRAY(col1,col2,'col3') as myarray;
```

Note: myobj.key3 and myarray[2] are "col3" as fixed string.

Also mixed JSON data:

```
SELECT JSON_OBJECT("opening","Sicilian", "variations",JSON_ARRAY("pelikan","dragon","najdorf"))
as mymixed ;
```

# 第35章：从 JSON 类型中提取值

参数	描述
json_doc	有效的 JSON 文档
路径	成员路径

MySQL 5.7.8 及以上版本支持原生 JSON 类型。虽然你有不同的方法创建 JSON 对象，但访问和读取成员的方式也各不相同。

主要函数是JSON\_EXTRACT，因此->和->>操作符更为友好。

## 第35.1节：读取 JSON 数组值

创建 @myjson 变量为 JSON 类型（详情请阅读）：

```
SET @myjson = CAST('["A","B",{ "id":1,"label":"C"}]' as JSON) ;
```

选择 一些成员！

```
查询
JSON_EXTRACT( @myjson , '$[1]' ) ,
JSON_EXTRACT( @myjson , '$[*].label' ) ,
JSON_EXTRACT( @myjson , '$[1].*' ) ,
JSON_EXTRACT( @myjson , '$[2].*' )
;
-- 结果值：
\'"B\'", \'["C"]\', NULL, [1, \'"C"\']
-- 视觉效果：
"B", ["C"], NULL, [1, "C"]
```

## 第35.2节：JSON提取操作符

通过->或->>操作符提取path，其中->>是未加引号的值：

```
查询
myjson_col->>'${1}' , myjson_col->'${1}' ,
myjson_col->>'${[*].label' ,
myjson_col->>'${1}.*' ,
myjson_col->>'${2}.*'
FROM tablename ;
-- 可视化：
B, "B" , ["C"], NULL, [1, "C"]
--^^^ ^^^
```

因此col->>path等同于JSON\_UNQUOTE(JSON\_EXTRACT(col,path))：

与->操作符一样，->>操作符在EXPLAIN的输出中总是被展开，正如下例所示：

```
mysql> EXPLAIN SELECT c->>'$.name' AS name
->      FROM jemp WHERE g > 2\G
***** 第1行 *****
```

# Chapter 35: Extract values from JSON type

Parameter	Description
json_doc	valid JSON document
path	members path

MySQL 5.7.8+ supports native JSON type. While you have different ways to create json objects, you can access and read members in different ways, too.

Main function is JSON\_EXTRACT, hence -> and ->> operators are more friendly.

## Section 35.1: Read JSON Array value

Create @myjson variable as JSON type (read more):

```
SET @myjson = CAST('["A","B",{ "id":1,"label":"C"}]' as JSON) ;
```

SELECT some members!

```
SELECT
JSON_EXTRACT( @myjson , '$[1]' ) ,
JSON_EXTRACT( @myjson , '$[*].label' ) ,
JSON_EXTRACT( @myjson , '$[1].*' ) ,
JSON_EXTRACT( @myjson , '$[2].*' )
;
-- result values:
\'"B\'", \'["C"]\', NULL, [1, \'"C"\']
-- visually:
"B", ["C"], NULL, [1, "C"]
```

## Section 35.2: JSON Extract Operators

Extract path by -> or ->> Operators, while ->> is UNQUOTED value:

```
SELECT
myjson_col->>'${1}' , myjson_col->'${1}' ,
myjson_col->>'${[*].label' ,
myjson_col->>'${1}.*' ,
myjson_col->>'${2}.*'
FROM tablename ;
-- visual:
B, "B" , ["C"], NULL, [1, "C"]
--^^^ ^^^
```

So col->>path is equal to JSON\_UNQUOTE(JSON\_EXTRACT(col,path))：

As with ->, the ->> operator is always expanded in the output of EXPLAIN, as the following example demonstrates:

```
mysql> EXPLAIN SELECT c->>'$.name' AS name
->      FROM jemp WHERE g > 2\G
***** 1. row *****
```

```
id: 1
select_type: SIMPLE
table: jemp
partitions: NULL
type: range
possible_keys: i
key: i
key_len: 5
ref: NULL
rows: 2
filtered: 100.00
Extra: 使用 where
1 行在结果集中, 1 条警告 (0.00 秒)
```

```
mysql> SHOW WARNINGS\G
***** 第1行 *****
级别: 注意
代码: 1003
信息: /* select#1 */ 选择
json_unquote(json_extract(`jtest`.`jemp`.`c`, '$.name')) 作为 `name` 从
`jtest`.`jemp` 中选择, 其中 (`jtest`.`jemp`.`g` > 2)
1 行记录 (0.00 秒)
```

[阅读关于内联路径提取\(+\)](#)

```
id: 1
select_type: SIMPLE
table: jemp
partitions: NULL
type: range
possible_keys: i
key: i
key_len: 5
ref: NULL
rows: 2
filtered: 100.00
Extra: Using where
1 row in set, 1 warning (0.00 sec)
```

```
mysql> SHOW WARNINGS\G
***** 1. row *****
Level: Note
Code: 1003
Message: /* select#1 */ select
json_unquote(json_extract(`jtest`.`jemp`.`c`, '$.name')) AS `name` from
`jtest`.`jemp` where (`jtest`.`jemp`.`g` > 2)
1 row in set (0.00 sec)
```

Read about [inline path extract\(+\)](#)

# 第36章：MySQL 管理

## 第36.1节：原子重命名与表重载

重命名表 t 为 t\_old, t\_copy 为 t;

在执行重命名表操作时，其他会话无法访问涉及的表，因此重命名操作不会受到并发问题的影响。

原子重命名特别适用于在不等待DELETE和加载完成的情况下，完全重载表：

```
CREATE TABLE new LIKE real;
通过任何方式加载`new` - LOAD DATA、INSERT或其他
重命名表real为old, new为real;
删除表old;
```

## 第36.2节：更改root密码

```
mysqladmin -u root -p'old-password' password 'new-password'
```

## 第36.3节：删除数据库

用于脚本化删除所有表并删除数据库：

```
mysqladmin -u[用户名] -p[密码] drop [数据库]
```

请极其谨慎使用。

作为SQL脚本删除数据库（你需要对该数据库有DROP权限）：

```
DROP DATABASE database_name
```

或

```
删除模式 database_name
```

# Chapter 36: MySQL Admin

## Section 36.1: Atomic RENAME & Table Reload

```
RENAME TABLE t TO t_old, t_copy TO t;
```

No other sessions can access the tables involved while RENAME TABLE executes, so the rename operation is not subject to concurrency problems.

Atomic Rename is especially for completely reloading a table without waiting for **DELETE** and load to finish:

```
CREATE TABLE new LIKE real;
load `new` by whatever means - LOAD DATA, INSERT, whatever
RENAME TABLE real TO old, new TO real;
DROP TABLE old;
```

## Section 36.2: Change root password

```
mysqladmin -u root -p'old-password' password 'new-password'
```

## Section 36.3: Drop database

Useful for scripting to drop all tables and deletes the database:

```
mysqladmin -u[username] -p[password] drop [database]
```

Use with extreme caution.

To **DROP** database as a SQL Script (you will need DROP privilege on that database):

```
DROP DATABASE database_name
```

or

```
DROP SCHEMA database_name
```

# 第37章：触发器

## 第37.1节：基本触发器

创建表

```
mysql> 创建表 account (acct_num 整数, amount 十进制(10,2));
```

查询成功, 0 行受影响 (0.03 秒)

创建触发器

```
mysql> 创建触发器 ins_sum 在插入之前于 account
-> 针对每行设置 @sum = @sum + NEW.amount;
```

查询成功, 0 行受影响 (0.06 秒)

CREATE TRIGGER 语句创建了一个名为 ins\_sum 的触发器，该触发器与 account 表相关联。它还包括指定触发器动作时间、触发事件以及触发器激活时执行操作的子句

插入值

要使用触发器，先将累加器变量（@sum）设置为零，执行INSERT语句，然后查看该变量之后的值：

```
mysql> SET @sum = 0;
mysql> INSERT INTO account VALUES(137,14.98),(141,1937.50),(97,-100.00);
mysql> SELECT @sum AS '插入的总金额';
```

插入的总金额
1852.48

在这种情况下，INSERT语句执行后@sum的值是14.98 + 1937.50 - 100，即1852.48。

删除触发器

```
mysql> DROP TRIGGER test.ins_sum;
```

如果删除一个表，该表的任何触发器也会被删除。

## 第37.2节：触发器的类型

时机

有两种触发器动作时间修饰符：

- BEFORE触发器在执行请求之前激活，
- AFTER触发器在更改后触发。

触发事件

# Chapter 37: TRIGGERS

## Section 37.1: Basic Trigger

Create Table

```
mysql> CREATE TABLE account (acct_num INT, amount DECIMAL(10,2));
```

Query OK, 0 rows affected (0.03 sec)

Create Trigger

```
mysql> CREATE TRIGGER ins_sum BEFORE INSERT ON account
-> FOR EACH ROW SET @sum = @sum + NEW.amount;
```

Query OK, 0 rows affected (0.06 sec)

The CREATE TRIGGER statement creates a trigger named ins\_sum that is associated with the account table. It also includes clauses that specify the trigger action time, the triggering event, and what to do when the trigger activates

Insert Value

To use the trigger, set the accumulator variable (@sum) to zero, execute an INSERT statement, and then see what value the variable has afterward:

```
mysql> SET @sum = 0;
mysql> INSERT INTO account VALUES(137,14.98),(141,1937.50),(97,-100.00);
mysql> SELECT @sum AS 'Total amount inserted';
```

Total amount inserted
1852.48

In this case, the value of @sum after the INSERT statement has executed is 14.98 + 1937.50 - 100, or 1852.48.

Drop Trigger

```
mysql> DROP TRIGGER test.ins_sum;
```

If you drop a table, any triggers for the table are also dropped.

## Section 37.2: Types of triggers

Timing

There are two trigger action time modifiers：

- BEFORE trigger activates before executing the request,
- AFTER trigger fire after change.

Triggering event



触发器可以附加的三种事件有：

- 插入
- 更新
- 删除

插入前触发器示例

```
DELIMITER $$

CREATE TRIGGER insert_date
  BEFORE INSERT ON stack
  FOR EACH ROW
  BEGIN
    -- 在插入之前设置请求中的 insert_date 字段
    SET NEW.insert_date = NOW();
  END;

$$
DELIMITER ;
```

更新前触发器示例

```
DELIMITER $$

CREATE TRIGGER update_date
  BEFORE UPDATE ON stack
  FOR EACH ROW
  BEGIN
    -- 在更新前设置请求中的 update_date 字段
    SET NEW.update_date = NOW();
  END;

$$
DELIMITER ;
```

删除后触发器示例

```
DELIMITER $$

CREATE TRIGGER deletion_date
  AFTER DELETE ON stack
  FOR EACH ROW
  BEGIN
    -- 成功删除后添加日志条目
    INSERT INTO log_action(stack_id, deleted_date) VALUES(OLD.id, NOW());
  END;

$$
DELIMITER ;
```

There are three events that triggers can be attached to:

- INSERT
- UPDATE
- DELETE

Before Insert trigger example

```
DELIMITER $$

CREATE TRIGGER insert_date
  BEFORE INSERT ON stack
  FOR EACH ROW
  BEGIN
    -- set the insert_date field in the request before the insert
    SET NEW.insert_date = NOW();
  END;

$$
DELIMITER ;
```

Before Update trigger example

```
DELIMITER $$

CREATE TRIGGER update_date
  BEFORE UPDATE ON stack
  FOR EACH ROW
  BEGIN
    -- set the update_date field in the request before the update
    SET NEW.update_date = NOW();
  END;

$$
DELIMITER ;
```

After Delete trigger example

```
DELIMITER $$

CREATE TRIGGER deletion_date
  AFTER DELETE ON stack
  FOR EACH ROW
  BEGIN
    -- add a log entry after a successful delete
    INSERT INTO log_action(stack_id, deleted_date) VALUES(OLD.id, NOW());
  END;

$$
DELIMITER ;
```

# 第38章：配置与调优

## 第38.1节：InnoDB性能

my.cnf中可以设置数百个参数。对于MySQL的“轻量级”用户来说，这些参数不会那么重要。

一旦您的数据库变得不再简单，建议设置以下参数：

```
innodb_buffer_pool_size
```

该参数应设置为可用内存的大约70%（如果您有至少4GB内存；如果是小型虚拟机或老旧机器，则比例应更小）。该设置控制InnoDB引擎使用的缓存大小，因此对InnoDB的性能非常重要。

## 第38.2节：允许插入大数据的参数

如果您需要在列中存储图片或视频，则需要根据您的应用需求调整该值

max\_allowed\_packet = 10M

M表示兆字节，G表示千兆字节，K表示千字节

## 第38.3节：增加group\_concat的字符串长度限制

group\_concat用于连接一个组中的非空值。结果字符串的最大长度可以通过group\_concat\_max\_len选项设置：

```
SET [GLOBAL | SESSION] group_concat_max_len = val;
```

设置GLOBAL变量将确保永久更改，而设置SESSION变量将为当前会话设置值。

## 第38.4节：最小InnoDB配置

这是使用InnoDB表的MySQL服务器的最低限度设置。使用InnoDB时，不需要查询缓存。当表或数据库被DROP时回收磁盘空间。如果您使用的是SSD，刷新操作是多余的（SSD不是顺序存储）。

```
default_storage_engine = InnoDB
query_cache_type = 0
innodb_file_per_table = 1
innodb_flush_neighbors = 0
```

### 并发

确保我们可以通过将innodb\_thread\_concurrency设置为无限大（0）来创建超过默认4个线程；这让InnoDB根据最佳执行情况自行决定。

```
innodb_thread_concurrency = 0
innodb_read_io_threads = 64
innodb_write_io_threads = 64
```

# Chapter 38: Configuration and tuning

## Section 38.1: InnoDB performance

There are hundreds of settings that can be placed in my.cnf. For the 'lite' user of MySQL, they won't matter as much.

Once your database becomes non-trivial, it is advisable to set the following parameters:

```
innodb_buffer_pool_size
```

This should be set to about 70% of *available* RAM (if you have at least 4GB of RAM; a smaller percentage if you have a tiny VM or antique machine). The setting controls the amount of cache used by the InnoDB ENGINE. Hence, it is very important for performance of InnoDB.

## Section 38.2: Parameter to allow huge data to insert

If you need to store images or videos in the column then we need to change the value as needed by your application

max\_allowed\_packet = 10M

M is Mb, G in Gb, K in Kb

## Section 38.3: Increase the string limit for group\_concat

group\_concat is used to concatenate non-null values in a group. The maximum length of the resulting string can be set using the group\_concat\_max\_len option:

```
SET [GLOBAL | SESSION] group_concat_max_len = val;
```

Setting the GLOBAL variable will ensure a permanent change, whereas setting the SESSION variable will set the value for the current session.

## Section 38.4: Minimal InnoDB configuration

This is a bare minimum setup for MySQL servers using InnoDB tables. Using InnoDB, query cache is not required. Reclaim disk space when a table or database is **DRO**ped. If you're using SSDs, flushing is a redundant operation (SDDs are not sequential).

```
default_storage_engine = InnoDB
query_cache_type = 0
innodb_file_per_table = 1
innodb_flush_neighbors = 0
```

### Concurrency

Make sure we can create more than than the default 4 threads by setting innodb\_thread\_concurrency to infinity (0); this lets InnoDB decide based on optimal execution.

```
innodb_thread_concurrency = 0
innodb_read_io_threads = 64
innodb_write_io_threads = 64
```

硬盘利用率

设置 MySQL 的 IOPS 容量（正常负载）和 capacity\_max（绝对最大值）。默认值 200 适用于机械硬盘，但如今 SSD 的 IOPS 可达数千，您可能需要调整此数值。

您可以运行许多测试来确定 IOPS。如果您运行的是专用的 MySQL 服务器，上述数值应接近该限制。如果您在同一台机器上运行其他服务，应适当分配资源。

```
innodb_io_capacity = 2500
innodb_io_capacity_max = 3000
```

内存利用率

设置 MySQL 可用的内存。虽然经验法则是 70-80%，但这实际上取决于您的实例是否专用于 MySQL 以及可用内存的多少。如果有大量内存可用，不要浪费内存（即资源）。

```
innodb_buffer_pool_size = 10G
```

第 38.5 节：安全的 MySQL 加密

默认的加密方式 aes-128-ecb 使用电子密码本（ECB）模式，该模式不安全，绝不应使用。请改为在配置文件中添加以下内容：

```
block_encryption_mode = aes-256-cbc
```

Hard drive utilization

Set the capacity (normal load) and capacity\_max (absolute maximum) of IOPS for MySQL. The default of 200 is fine for HDDs, but these days, with SSDs capable of thousands of IOPS, you are likely to want to adjust this number. There are many tests you can run to determine IOPS. The values above should be nearly that limit *if you are running a dedicated MySQL server*. If you are running any other services on the same machine, you should apportion as appropriate.

```
innodb_io_capacity = 2500
innodb_io_capacity_max = 3000
```

RAM utilization

Set the RAM available to MySQL. Whilst the rule of thumb is 70-80%, this really depends on whether or not your instance is dedicated to MySQL, and how much RAM is available. Don't *waste* RAM (i.e. resources) if you have a lot available.

```
innodb_buffer_pool_size = 10G
```

Section 38.5: Secure MySQL encryption

The default encryption aes-128-ecb uses Electronic Codebook (ECB) mode, which is insecure and should never be used. Instead, add the following to your configuration file:

```
block_encryption_mode = aes-256-cbc
```

# 第 39 章：事件

## 第39.1节：创建事件

MySQL 有其事件（EVENT）功能，用于避免在调度大量与 SQL 相关而非文件相关的任务时，复杂的 cron 交互。参见手册页面 [here](#)。可以将事件视为定期运行的存储过程。

为了节省调试事件相关问题的时间，请记住必须开启全局事件调度器才能处理事件。

```
SHOW VARIABLES WHERE variable_name='event_scheduler';
```

Variable_name	Value
event_scheduler	OFF

如果它是关闭状态，则不会触发任何事件。所以请将其打开：

```
SET GLOBAL event_scheduler = ON;
```

### 测试用的模式

```
create table theMessages
(
  id INT AUTO_INCREMENT PRIMARY KEY,
  userId INT NOT NULL,
  message VARCHAR(255) NOT NULL,
  updateDt DATETIME NOT NULL,
  KEY(updateDt)
);

INSERT theMessages(userId,message,updateDt) VALUES (1,'message 123','2015-08-24 11:10:09');
INSERT theMessages(userId,message,updateDt) VALUES (7,'message 124','2015-08-29');
INSERT theMessages(userId,message,updateDt) VALUES (1,'message 125','2015-09-03 12:00:00');
INSERT theMessages(userId,message,updateDt) VALUES (1,'message 126','2015-09-03 14:00:00');
```

以上插入语句用于展示起点。请注意，下面创建的两个事件将会清理数据行。

### 创建两个事件，第一个每天运行，第二个每10分钟运行一次

忽略它们实际执行的操作（相互竞争）。重点在于间隔（INTERVAL）和调度。

```
DROP EVENT IF EXISTS `delete7DayOldMessages`;
DELIMITER $$
CREATE EVENT `delete7DayOldMessages`
  ON SCHEDULE EVERY 1 DAY STARTS '2015-09-01 00:00:00'
  ON COMPLETION PRESERVE
DO BEGIN
  DELETE FROM theMessages
    WHERE datediff(now(),updateDt)>6; -- 不是非常精确，昨天但少于24小时仍算1天

  -- 其他代码

END$$
```

# Chapter 39: Events

## Section 39.1: Create an Event

MySQL has its EVENT functionality for avoiding complicated cron interactions when much of what you are scheduling is SQL related, and less file related. See the Manual page [here](#). Think of Events as Stored Procedures that are scheduled to run on recurring intervals.

To save time in debugging Event-related problems, keep in mind that the global event handler must be turned on to process events.

```
SHOW VARIABLES WHERE variable_name='event_scheduler';
```

Variable_name	Value
event_scheduler	OFF

With it OFF, nothing will trigger. So turn it on:

```
SET GLOBAL event_scheduler = ON;
```

### Schema for testing

```
create table theMessages
(
  id INT AUTO_INCREMENT PRIMARY KEY,
  userId INT NOT NULL,
  message VARCHAR(255) NOT NULL,
  updateDt DATETIME NOT NULL,
  KEY(updateDt)
);

INSERT theMessages(userId,message,updateDt) VALUES (1,'message 123','2015-08-24 11:10:09');
INSERT theMessages(userId,message,updateDt) VALUES (7,'message 124','2015-08-29');
INSERT theMessages(userId,message,updateDt) VALUES (1,'message 125','2015-09-03 12:00:00');
INSERT theMessages(userId,message,updateDt) VALUES (1,'message 126','2015-09-03 14:00:00');
```

The above inserts are provided to show a starting point. Note that the 2 events created below will clean out rows.

### Create 2 events, 1st runs daily, 2nd runs every 10 minutes

Ignore what they are actually doing (playing against one another). The point is on the INTERVAL and scheduling.

```
DROP EVENT IF EXISTS `delete7DayOldMessages`;
DELIMITER $$
CREATE EVENT `delete7DayOldMessages`
  ON SCHEDULE EVERY 1 DAY STARTS '2015-09-01 00:00:00'
  ON COMPLETION PRESERVE
DO BEGIN
  DELETE FROM theMessages
    WHERE datediff(now(),updateDt)>6; -- not terribly exact, yesterday but <24hrs is still 1 day

  -- Other code here

END$$
```

```
DELIMITER ;
```

...

```
DROP EVENT IF EXISTS `Every_10_Minutes_Cleanup`;
DELIMITER $$
CREATE EVENT `Every_10_Minutes_Cleanup`
  ON SCHEDULE EVERY 10 MINUTE STARTS '2015-09-01 00:00:00'
  ON COMPLETION PRESERVE
DO BEGIN
  DELETE FROM theMessages
  WHERE TIMESTAMPDIFF(HOUR, updateDt, now())>168; -- 消息超过1周 (168小时)

  -- 其他代码
END$$
DELIMITER ;
```

显示事件状态（不同方法）

```
SHOW EVENTS FROM my_db_name; -- 按模式名（数据库名）列出所有事件
SHOW EVENTS;
SHOW EVENTS\G; -- <----- 我喜欢mysql>提示符下的这个
```

```
***** 1. 行 *****
数据库: my_db_name
名称: delete7DayOldMessages
定义者: root@localhost
时区: SYSTEM
类型: 定期
执行时间: NULL
间隔值: 1
间隔单位: 天
开始时间: 2015-09-01 00:00:00
结束时间: NULL
状态: 启用
发起者: 1
character_set_client: utf8
collation_connection: utf8_general_ci
数据库排序规则: utf8_general_ci
***** 第2行 *****

数据库: my_db_name
名称: Every_10_Minutes_Cleanup
定义者: root@localhost
时区: SYSTEM
类型: 周期性
执行时间: NULL
间隔值: 10
间隔单位: 分钟
开始时间: 2015-09-01 00:00:00
结束时间: NULL
状态: 启用
发起者: 1
character_set_client: utf8
collation_connection: utf8_general_ci
数据库排序规则: utf8_general_ci
共2行, 查询耗时0.06秒
```

需要考虑的随机内容

```
DROP EVENT someEventName; -- 删除事件及其代码
```

```
DELIMITER ;
```

...

```
DROP EVENT IF EXISTS `Every_10_Minutes_Cleanup`;
DELIMITER $$
CREATE EVENT `Every_10_Minutes_Cleanup`
  ON SCHEDULE EVERY 10 MINUTE STARTS '2015-09-01 00:00:00'
  ON COMPLETION PRESERVE
DO BEGIN
  DELETE FROM theMessages
  WHERE TIMESTAMPDIFF(HOUR, updateDt, now())>168; -- messages over 1 week old (168 hours)

  -- Other code here
END$$
DELIMITER ;
```

Show event statuses (different approaches)

```
SHOW EVENTS FROM my_db_name; -- List all events by schema name (db name)
SHOW EVENTS;
SHOW EVENTS\G; -- <----- I like this one from mysql> prompt
```

```
***** 1. row *****
Db: my_db_name
Name: delete7DayOldMessages
Definer: root@localhost
Time zone: SYSTEM
Type: RECURRING
Execute at: NULL
Interval value: 1
Interval field: DAY
Starts: 2015-09-01 00:00:00
Ends: NULL
Status: ENABLED
Originator: 1
character_set_client: utf8
collation_connection: utf8_general_ci
Database Collation: utf8_general_ci
***** 2. row *****

Db: my_db_name
Name: Every_10_Minutes_Cleanup
Definer: root@localhost
Time zone: SYSTEM
Type: RECURRING
Execute at: NULL
Interval value: 10
Interval field: MINUTE
Starts: 2015-09-01 00:00:00
Ends: NULL
Status: ENABLED
Originator: 1
character_set_client: utf8
collation_connection: utf8_general_ci
Database Collation: utf8_general_ci
2 rows in set (0.06 sec)
```

Random stuff to consider

```
DROP EVENT someEventName; -- Deletes the event and its code
```



ON COMPLETION PRESERVE -- 事件处理完成后保留事件，否则删除。

事件类似触发器。它们不是由用户程序调用，而是被调度执行。因此，它们成功或失败时不会有提示。

手册页面的链接显示了相当多的区间选择灵活性，如下所示：

间隔：  
  
数量 {年 | 季度 | 月 | 日 | 小时 | 分钟 |  
周 | 秒 | 年月 | 日小时 | 日分钟 |  
日秒 | 小时分钟 | 小时秒 | 分钟秒}

事件是处理系统中重复和计划任务的强大机制。它们可以包含尽可能多的语句、DDL 和 DML 例程，以及复杂的连接。请参阅 MySQL 手册页中题为“存储程序的限制”的部分。

ON COMPLETION PRESERVE -- When the event is done processing, retain it. Otherwise, it is deleted.

Events are like triggers. They are not called by a user's program. Rather, they are scheduled. As such, they succeed or fail silently.

The link to the Manual Page shows quite a bit of flexibility with interval choices, shown below:

interval:  
  
quantity {YEAR | QUARTER | MONTH | DAY | HOUR | MINUTE |  
WEEK | SECOND | YEAR\_MONTH | DAY\_HOUR | DAY\_MINUTE |  
DAY\_SECOND | HOUR\_MINUTE | HOUR\_SECOND | MINUTE\_SECOND}

Events are powerful mechanisms that handle recurring and scheduled tasks for your system. They may contain as many statements, DDL and DML routines, and complicated joins as you may reasonably wish. Please see the MySQL Manual Page entitled [Restrictions on Stored Programs](#).

# 第40章：ENUM

## 第40.1节：为什么使用 ENUM？

ENUM 提供了一种为行提供属性的方法。具有少量非数字选项的属性效果最佳。示例：

```
reply ENUM('yes', 'no')
gender ENUM('male', 'female', 'other', 'decline-to-state')
```

这些值是字符串：

```
INSERT ... VALUES ('yes', 'female')
SELECT ... --> yes female
```

## 第40.2节：VARCHAR作为替代方案

假设我们有

```
类型 ENUM('fish','mammal','bird')
```

一种替代方案是

```
类型 VARCHAR(20) 注释 "fish, bird, etc"
```

这相当开放，因为可以轻松添加新类型。

比较，以及是否优于ENUM：

- （相同）插入：只需提供字符串
- （较差？）插入时拼写错误不会被发现
- （相同）查询：返回实际字符串
- （较差）占用更多空间

## 第40.3节：添加新选项

```
ALTER TABLE tbl MODIFY COLUMN type ENUM('fish','mammal','bird','insect');
```

备注

- 与所有修改列（MODIFY COLUMN）的情况一样，必须包含NOT NULL以及任何原本存在的其他限定符，否则它们将会丢失。
- 如果你在列表的末尾添加，并且列表项少于256个，ALTER操作仅通过更改模式来完成。也就是说，不会进行耗时的表复制。（旧版本的MySQL没有此优化。）

## 第40.4节：NULL与NOT NULL

示例展示了当NULL和“错误值”存储到可为空和不可为空的列中时会发生什么。同时展示了通过+0进行数值类型转换的用法。

```
创建表 enum (
e      ENUM('yes', 'no')      NOT NULL,
enull  ENUM('x', 'y', 'z')    NULL
```

# Chapter 40: ENUM

## Section 40.1: Why ENUM?

ENUM provides a way to provide an attribute for a row. Attributes with a small number of non-numeric options work best. Examples:

```
reply ENUM('yes', 'no')
gender ENUM('male', 'female', 'other', 'decline-to-state')
```

The values are strings:

```
INSERT ... VALUES ('yes', 'female')
SELECT ... --> yes female
```

## Section 40.2: VARCHAR as an alternative

Let's say we have

```
type ENUM('fish', 'mammal', 'bird')
```

An alternative is

```
type VARCHAR(20)  COMMENT "fish, bird, etc"
```

This is quite open-ended in that new types are trivially added.

Comparison, and whether better or worse than ENUM:

- (same) INSERT: simply provide the string
- (worse?) On INSERT a typo will go unnoticed
- (same) SELECT: the actual string is returned
- (worse) A lot more space is consumed

## Section 40.3: Adding a new option

```
ALTER TABLE tbl MODIFY COLUMN type ENUM('fish', 'mammal', 'bird', 'insect');
```

Notes

- As with all cases of MODIFY COLUMN, you must include **NOT NULL**, and any other qualifiers that originally existed, else they will be lost.
- If you add to the *end* of the list *and* the list is under 256 items, the **ALTER** is done by merely changing the schema. That is there will not be a lengthy table copy. (Old versions of MySQL did not have this optimization.)

## Section 40.4: NULL vs NOT NULL

Examples of what happens when NULL and 'bad-value' are stored into nullable and not nullable columns. Also shows usage of casting to numeric via +0.

```
CREATE TABLE enum (
e      ENUM('yes', 'no')      NOT NULL,
enull  ENUM('x', 'y', 'z')    NULL
```

```
);
插入数据到 enum (e, enull)
VALUES
('yes', 'x'),
('no', 'y'),
(NULL, NULL),
('bad-value', 'bad-value');
```

查询成功, 影响了4行, 3条警告 (0.00 秒)  
记录数: 4 重复: 0 警告: 3

mysql>SHOW WARNINGS;

级别	代码	信息
警告	1048	列 'e' 不能为 null
警告	1265	第4行列 'e' 的数据被截断
警告	1265	第4行列 'enull' 的数据被截断

3 行记录 (0.00 秒)

插入后表中的内容是什么。这里使用 "+0" 来转换为数值，查看存储的内容。

mysql>SELECT e, e+0 FROM enum;

e	e+0
yes	1
no	2
	0 -- NULL
	0 -- 'bad-value'

4 行记录 (0.00 秒)

mysql>SELECT enull, enull+0 FROM enum;

enull	enull+0
x	1
y	2
NULL	NULL
	0 -- 'bad-value'

4 行记录 (0.00 秒)

```
);
INSERT INTO enum (e, enull)
VALUES
('yes', 'x'),
('no', 'y'),
(NULL, NULL),
('bad-value', 'bad-value');
```

Query OK, 4 rows affected, 3 warnings (0.00 sec)  
Records: 4 Duplicates: 0 Warnings: 3

mysql>SHOW WARNINGS;

Level	Code	Message
Warning	1048	Column 'e' cannot be null
Warning	1265	Data truncated for column 'e' at row 4
Warning	1265	Data truncated for column 'enull' at row 4

3 rows in set (0.00 sec)

What is in the table after those inserts. This uses "+0" to cast to numeric see what is stored.

mysql>SELECT e, e+0 FROM enum;

e	e+0
yes	1
no	2
	0 -- NULL
	0 -- 'bad-value'

4 rows in set (0.00 sec)

mysql>SELECT enull, enull+0 FROM enum;

enull	enull+0
x	1
y	2
NULL	NULL
	0 -- 'bad-value'

4 rows in set (0.00 sec)

# 第41章：使用 Docker-Compose 安装 Mysql 容器

## 第41.1节：使用 docker-compose 的简单示例

这是一个使用 docker 创建 mysql 服务器的简单示例

1.- 创建 **docker-compose.yml**：

注意：如果你想为所有项目使用同一个容器，应该在你的 HOME\_PATH 中创建一个路径。如果你想为每个项目单独创建，可以在项目中创建一个 docker 目录。

```
版本： '2'
服务：
cabin_db：
  镜像: mysql:latest
  卷：
    - ".mysql-data/db:/var/lib/mysql"
  restart: always
端口：
  - 3306:3306
环境变量：
MYSQL_ROOT_PASSWORD: rootpw
MYSQL_DATABASE: cabin
MYSQL_USER: cabin
MYSQL_PASSWORD: cabinpw
```

2.- 运行它：

```
cd PATH_TO_DOCKER-COMPOSE.YML
docker-compose up -d
```

3.- 连接到服务器

```
mysql -h 127.0.0.1 -u root -P 3306 -p rootpw
```

太好了！！

4.- 停止服务器

```
docker-compose stop
```

# Chapter 41: Install Mysql container with Docker-Compose

## Section 41.1: Simple example with docker-compose

This is an simple example to create a mysql server with docker

1.- create **docker-compose.yml**:

**Note:** If you want to use same container for all your projects, you should create a PATH in your HOME\_PATH. If you want to create it for every project you could create a **docker** directory in your project.

```
version: '2'
services:
  cabin_db:
    image: mysql:latest
    volumes:
      - ".mysql-data/db:/var/lib/mysql"
    restart: always
    ports:
      - 3306:3306
    environment:
      MYSQL_ROOT_PASSWORD: rootpw
      MYSQL_DATABASE: cabin
      MYSQL_USER: cabin
      MYSQL_PASSWORD: cabinpw
```

2.- run it:

```
cd PATH_TO_DOCKER-COMPOSE.YML
docker-compose up -d
```

3.- connect to server

```
mysql -h 127.0.0.1 -u root -P 3306 -p rootpw
```

Hurray!!

4.- stop server

```
docker-compose stop
```

# 第42章：字符集和排序规则

## 第42.1节：选择哪个字符集和排序规则？

有数十种字符集和数百种排序规则。（一个排序规则只属于一个字符集。）  
请参见 `SHOW COLLATION`；的输出结果。

通常只有4种字符集比较重要：

```
ascii -- 基本的7位编码。
latin1 -- ascii，加上大多数西欧语言所需的字符。
utf8 -- utf8的1、2和3字节子集。 这不包括表情符号和部分中文字符。
utf8mb4 -- 完整的UTF8字符集，涵盖所有现有语言。
```

所有字符集都包含英文字符，编码方式相同。utf8是utf8mb4的子集。

最佳实践...

- 对于可能包含多种语言的任何TEXT或VARCHAR列，使用utf8mb4。
- 对于十六进制字符串（UUID、MD5等）和简单代码（国家代码、邮政编码等），使用ascii（latin1也可以）。

utf8mb4直到版本5.5.3才出现，因此在此之前utf8是最好的选择。

在MySQL之外，“UTF8”与MySQL的utf8mb4含义相同，而不是MySQL的utf8。

排序规则以字符集名称开头，通常以\_ci结尾表示“大小写和重音不敏感”，或以\_bin结尾表示“仅比较位”。

“最新”的utf8mb4排序规则是utf8mb4\_unicode\_520\_ci，基于Unicode 5.20。如果你只处理单一语言，可能会选择例如utf8mb4\_polish\_ci，它会根据波兰语习惯稍微调整字母顺序。

## 第42.2节：设置表和字段的字符集

你可以为表设置字符集，也可以为单个字段使用CHARACTER SET和CHARSET语句设置字符集：

```
CREATE TABLE Address (
  `AddressID`    INTEGER NOT NULL PRIMARY KEY,
  `Street`       VARCHAR(80) CHARACTER SET ASCII,
  `City`         VARCHAR(80),
  `Country`      VARCHAR(80) 默认值 "United States",
  `Active`       BOOLEAN 默认值 1,
) 引擎=InnoDB 默认 字符集=UTF8;
```

城市和国家将使用UTF8，因为我们将其设置为表的默认字符集。街道则使用ASCII，因为我们特别指定了这样做。

设置正确的字符集高度依赖于您的数据集，但也能大幅提升处理您数据的系统之间的可移植性。

## 第42.3节：声明

```
创建表 foo ( ...
名称 字符集 utf8mb4
```

# Chapter 42: Character Sets and Collations

## Section 42.1: Which CHARACTER SET and COLLATION?

There are dozens of character sets with hundreds of collations. (A given collation belongs to only one character set.)  
See the output of `SHOW COLLATION`;

There are usually only 4 CHARACTER SETs that matter:

```
ascii -- basic 7-bit codes.
latin1 -- ascii, plus most characters needed for Western European languages.
utf8 -- the 1-, 2-, and 3-byte subset of utf8. This excludes Emoji and some of Chinese.
utf8mb4 -- the full set of UTF8 characters, covering all current languages.
```

All include English characters, encoded identically. utf8 is a subset of utf8mb4.

Best practice...

- Use utf8mb4 for any TEXT or VARCHAR column that can have a variety of languages in it.
- Use ascii (latin1 is ok) for hex strings (UUID, MD5, etc) and simple codes (country\_code, postal\_code, etc).

utf8mb4 did not exist until version 5.5.3, so utf8 was the best available before that.

Outside of MySQL, "UTF8" means the same things as MySQL's utf8mb4, not MySQL's utf8.

Collations start with the charset name and usually end with \_ci for "case and accent insensitive" or \_bin for "simply compare the bits."

The 'latest' utf8mb4 collation is utf8mb4\_unicode\_520\_ci, based on Unicode 5.20. If you are working with a single language, you might want, say, utf8mb4\_polish\_ci, which will rearrange the letters slightly, based on Polish conventions.

## Section 42.2: Setting character sets on tables and fields

You can set a [character set](#) both per table, as well as per individual field using the CHARACTER SET and CHARSET statements:

```
CREATE TABLE Address (
  `AddressID`    INTEGER NOT NULL PRIMARY KEY,
  `Street`       VARCHAR(80) CHARACTER SET ASCII,
  `City`         VARCHAR(80),
  `Country`      VARCHAR(80) DEFAULT "United States",
  `Active`       BOOLEAN DEFAULT 1,
) Engine=InnoDB default charset=UTF8;
```

City and Country will use UTF8, as we set that as the default character set for the table. Street on the other hand will use ASCII, as we've specifically told it to do so.

Setting the right character set is highly dependent on your dataset, but can also highly improve portability between systems working with your data.

## Section 42.3: Declaration

```
CREATE TABLE foo ( ...
  name CHARACTER SET utf8mb4
```



```
... );
```

## 第42.4节：连接

使用字符集的关键是告诉MySQL服务器客户端字节的编码。以下是一种方法：

```
设置 NAMES utf8mb4;
```

每种语言（PHP、Python、Java 等）都有自己的方式，通常建议使用SET NAMES。

例如：SET NAMES utf8mb4，配合声明为CHARACTER SET latin1的列——这将在INSERT时从latin1转换为utf8mb4，SELECT时再转换回去。

```
... );
```

## Section 42.4: Connection

Vital to using character sets is to tell the MySQL-server what encoding the client's bytes are. Here is one way:

```
SET NAMES utf8mb4;
```

Each language (PHP, Python, Java, ...) has its own way the it usually preferable to SET NAMES.

For example: SET NAMES utf8mb4, together with a column declared CHARACTER SET latin1 -- this will convert from latin1 to utf8mb4 when INSERTing and convert back when SELECTing.

# 第43章：MyISAM引擎

## 第43.1节：ENGINE=MyISAM

```
CREATE TABLE foo (  
    ...  
) ENGINE=MyISAM;
```

# Chapter 43: MyISAM Engine

## Section 43.1: ENGINE=MyISAM

```
CREATE TABLE foo (  
    ...  
) ENGINE=MyISAM;
```

# 第44章：从MyISAM转换到InnoDB

## 第44.1节：基本转换

```
ALTER TABLE foo ENGINE=InnoDB;
```

这会转换表，但不会处理引擎之间的任何差异。大多数差异不会产生影响，尤其是对于小表。但对于更繁忙的表，应考虑其他因素。[转换注意事项](#)

## 第44.2节：转换一个数据库中的所有表

要轻松转换一个数据库中的所有表，请使用以下命令：

```
SET @DB_NAME = DATABASE();

SELECT CONCAT('ALTER TABLE `', table_name, '` ENGINE=InnoDB;') AS sql_statements
FROM information_schema.tables
WHERE table_schema = @DB_NAME
AND `ENGINE` = 'MyISAM'
AND `TABLE_TYPE` = 'BASE TABLE';
```

注意：你应该连接到你的数据库以使 DATABASE() 函数生效，否则它将返回 NULL。此情况主要适用于服务器自带的标准 mysql 客户端，因为它允许连接时不指定数据库。

运行此 SQL 语句以检索数据库中所有 MyISAM 表。

最后，复制输出内容并执行其中的 SQL 查询。

# Chapter 4 4: Converting from MyISAM to InnoDB

## Section 4 4.1: Basic conversion

```
ALTER TABLE foo ENGINE=InnoDB;
```

This converts the table, but does not take care of any differences between the engines. Most differences will not matter, especially for small tables. But for busier tables, other considerations should be considered. [Conversion considerations](#)

## Section 4 4.2: Converting All Tables in one Database

To easily convert all tables in one database, use the following:

```
SET @DB_NAME = DATABASE();

SELECT CONCAT('ALTER TABLE `', table_name, '` ENGINE=InnoDB;') AS sql_statements
FROM information_schema.tables
WHERE table_schema = @DB_NAME
AND `ENGINE` = 'MyISAM'
AND `TABLE_TYPE` = 'BASE TABLE';
```

**NOTE:** You should be connected to your database for DATABASE() function to work, otherwise it will return **NULL**. This mostly applies to standard mysql client shipped with server as it allows to connect without specifying a database.

Run this SQL statement to retrieve all the MyISAM tables in your database.

Finally, copy the output and execute SQL queries from it.

# 第45章：事务

## 第45.1节：启动事务

事务是一组顺序执行的 SQL 语句，如 select、insert、update 或 delete，这些语句作为一个单一的工作单元执行。

换句话说，除非组内的每个单独操作都成功，否则事务永远不会完成。如果事务中的任何操作失败，整个事务将失败。

银行交易是解释这点的最佳例子。考虑两个账户之间的转账。为了实现这一点，你必须编写执行以下操作的SQL语句

- 1. 检查第一个账户中请求金额的可用性
- 2. 从第一个账户扣除请求金额
- 3. 将其存入第二个账户

如果这些过程中的任何一个失败，整个过程都应恢复到之前的状态。

### ACID：事务的属性

事务具有以下四个标准属性

- **原子性**：确保工作单元内的所有操作都成功完成；否则，事务将在失败点中止，之前的操作将回滚到原先状态。
- **一致性**：确保数据库在成功提交事务后正确地改变状态。
- **隔离性**：使事务能够独立运行且彼此透明。
- **持久性**：确保已提交事务的结果或效果在系统故障时依然存在。

事务以语句**START TRANSACTION**或**BEGIN WORK**开始，以**COMMIT**或**ROLLBACK**语句结束。开始和结束语句之间的SQL命令构成了事务的主体。

```
START TRANSACTION;
SET @transAmt = '500';
SELECT @availableAmt:=ledgerAmt FROM accTable WHERE customerId=1 FOR UPDATE;
UPDATE accTable SET ledgerAmt=ledgerAmt-@transAmt WHERE customerId=1;
UPDATE accTable SET ledgerAmt=ledgerAmt+@transAmt WHERE customerId=2;
COMMIT;
```

使用START TRANSACTION时，自动提交保持禁用状态，直到你用COMMIT或ROLLBACK结束事务。然后自动提交模式恢复到之前的状态。

FOR UPDATE表示（并锁定）事务期间的行。

在事务未提交期间，该事务对其他用户不可用。

### 事务涉及的一般步骤

- 通过执行SQL命令BEGIN WORK或START TRANSACTION开始事务。
- 运行所有的 SQL 语句。
- 检查是否所有操作都按照你的要求执行。
- 如果是，则发出COMMIT命令，否则发出ROLLBACK命令，将所有操作回滚到之前的状态。

# Chapter 45: Transaction

## Section 45.1: Start Transaction

A transaction is a sequential group of SQL statements such as select,insert,update or delete, which is performed as one single work unit.

In other words, a transaction will never be complete unless each individual operation within the group is successful. If any operation within the transaction fails, the entire transaction will fail.

Bank transaction will be best example for explaining this. Consider a transfer between two accounts. To achieve this you have to write SQL statements that do the following

- 1. Check the availability of requested amount in the first account
- 2. Deduct requested amount from first account
- 3. Deposit it in second account

If anyone these process fails, the whole should be reverted to their previous state.

### ACID : Properties of Transactions

Transactions have the following four standard properties

- **Atomicity**: ensures that all operations within the work unit are completed successfully; otherwise, the transaction is aborted at the point of failure, and previous operations are rolled back to their former state.
- **Consistency**: ensures that the database properly changes states upon a successfully committed transaction.
- **Isolation**: enables transactions to operate independently of and transparent to each other.
- **Durability**: ensures that the result or effect of a committed transaction persists in case of a system failure.

Transactions begin with the statement **START TRANSACTION** or **BEGIN WORK** and end with either a **COMMIT** or a **ROLLBACK** statement. The SQL commands between the beginning and ending statements form the bulk of the transaction.

```
START TRANSACTION;
SET @transAmt = '500';
SELECT @availableAmt:=ledgerAmt FROM accTable WHERE customerId=1 FOR UPDATE;
UPDATE accTable SET ledgerAmt=ledgerAmt-@transAmt WHERE customerId=1;
UPDATE accTable SET ledgerAmt=ledgerAmt+@transAmt WHERE customerId=2;
COMMIT;
```

With **START TRANSACTION**, autocommit remains disabled until you end the transaction with **COMMIT** or **ROLLBACK**. The autocommit mode then reverts to its previous state.

The FOR **UPDATE** indicates (and locks) the row(s) for the duration of the transaction.

While the transaction remains uncommitted, this transaction will not be available for others users.

### General Procedures involved in Transaction

- Begin transaction by issuing SQL command **BEGIN WORK** or **START TRANSACTION**.
- Run all your SQL statements.
- Check whether everything is executed according to your requirement.
- If yes, then issue **COMMIT** command, otherwise issue a **ROLLBACK** command to revert everything to the previous state.

- 即使使用了或可能最终使用 Galera/PXC，也要在COMMIT后检查错误。

## 第45.2节：COMMIT、ROLLBACK 和 AUTOCOMMIT

### AUTOCOMMIT

MySQL 会自动提交不属于事务的语句。任何未以BEGIN或START TRANSACTION开头的UPDATE、DELETE或INSERT语句的结果将立即对所有连接可见。

AUTOCOMMIT变量默认设置为true。可以通过以下方式更改，

```
--->将 autocommit 设置为false
SET AUTOCOMMIT=false;
--或者
SET AUTOCOMMIT=0;

--->将自动提交设置为true
SET AUTOCOMMIT=true;
--或者
SET AUTOCOMMIT=1;
```

查看AUTOCOMMIT状态

```
SELECT @@autocommit;
```

### 提交

如果AUTOCOMMIT设置为false且事务未提交，变更仅对当前连接可见。

执行COMMIT语句提交变更后，结果对所有连接可见。

我们通过两个连接来说明这一点

连接1

```
--->在将自动提交设置为false之前，向新表中添加一行
mysql> INSERT INTO testTable VALUES (1);

--->将自动提交设置为= false
mysql> SET autocommit=0;

mysql> INSERT INTO testTable VALUES (2), (3);
mysql> SELECT * FROM testTable;
-----
| tId |
-----
|  1  |
|  2  |
|  3  |
+-----+
```

连接2

```
mysql> SELECT * FROM testTable;
-----
| tId |
```

- Check for errors even after COMMIT if you are using, or might eventually use, Galera/PXC.

## Section 45.2: COMMIT , ROLLBACK and AUTOCOMMIT

### AUTOCOMMIT

MySQL automatically commits statements that are not part of a transaction. The results of any UPDATE,DELETE or INSERT statement not preceded with a BEGIN or START TRANSACTION will immediately be visible to all connections.

The AUTOCOMMIT variable is set true by default. This can be changed in the following way,

```
--->To make autocommit false
SET AUTOCOMMIT=false;
--or
SET AUTOCOMMIT=0;

--->To make autocommit true
SET AUTOCOMMIT=true;
--or
SET AUTOCOMMIT=1;
```

To view AUTOCOMMIT status

```
SELECT @@autocommit;
```

### COMMIT

If AUTOCOMMIT set to false and the transaction not committed, the changes will be visible only for the current connection.

After COMMIT statement commits the changes to the table, the result will be visible for all connections.

We consider two connections to explain this

Connection 1

```
--->Before making autocommit false one row added in a new table
mysql> INSERT INTO testTable VALUES (1);

--->Making autocommit = false
mysql> SET autocommit=0;

mysql> INSERT INTO testTable VALUES (2), (3);
mysql> SELECT * FROM testTable;
+-----+
| tId |
+-----+
|  1  |
|  2  |
|  3  |
+-----+
```

Connection 2

```
mysql> SELECT * FROM testTable;
+-----+
| tId |
```



```
-----
|  1 |
+-----+
---> 在 autocommit=false 之前插入的行仅在此处可见
```

连接1

```
mysql> COMMIT;
--->现在COMMIT在连接1中执行
mysql> SELECT * FROM testTable;

-----
| tId |
-----
|  1 |
|  2 |
|  3 |
+-----+
```

连接2

```
mysql> SELECT * FROM testTable;

-----
| tId |
+-----+
|  1 |
|  2 |
|  3 |
+-----+
--->现在所有三行数据都显示在这里
```

回滚

如果查询执行过程中出现任何错误，ROLLBACK用于撤销更改。详见下面的说明

```
--->在将自动提交设置为false之前，向新表中添加一行
mysql> INSERT INTO testTable VALUES (1);

--->将自动提交设置为= false
mysql> SET autocommit=0;

mysql> INSERT INTO testTable VALUES (2), (3);
mysql> SELECT * FROM testTable;

-----
| tId |
-----
|  1 |
|  2 |
|  3 |
+-----+
```

我们现在正在执行ROLLBACK

```
--->回滚已执行现在
mysql> ROLLBACK;

mysql> SELECT * FROM testTable;

-----
| tId |
-----
```

```
+-----+
|  1 |
+-----+
---> Row inserted before autocommit=false only visible here
```

Connection 1

```
mysql> COMMIT;
--->Now COMMIT is executed in connection 1
mysql> SELECT * FROM testTable;

+-----+
| tId |
+-----+
|  1 |
|  2 |
|  3 |
+-----+
```

Connection 2

```
mysql> SELECT * FROM testTable;

+-----+
| tId |
+-----+
|  1 |
|  2 |
|  3 |
+-----+
--->Now all the three rows are visible here
```

ROLLBACK

If anything went wrong in your query execution, **ROLLBACK** is used to revert the changes. See the explanation below

```
--->Before making autocommit false one row added in a new table
mysql> INSERT INTO testTable VALUES (1);

--->Making autocommit = false
mysql> SET autocommit=0;

mysql> INSERT INTO testTable VALUES (2), (3);
mysql> SELECT * FROM testTable;

+-----+
| tId |
+-----+
|  1 |
|  2 |
|  3 |
+-----+
```

Now we are executing **ROLLBACK**

```
--->Rollback executed now
mysql> ROLLBACK;

mysql> SELECT * FROM testTable;

+-----+
| tId |
+-----+
```

```
| 1 |
+----+
--->回滚移除了所有未提交的行
```

一旦执行COMMIT，ROLLBACK将不会产生任何效果

```
mysql> INSERT INTO testTable VALUES (2), (3);
mysql> SELECT * FROM testTable;
mysql> COMMIT;

-----
| tId |
+----+
| 1 |
| 2 |
| 3 |
+----+

--->回滚已执行现在
mysql> ROLLBACK;

mysql> SELECT * FROM testTable;
-----
| tId |
+----+
| 1 |
| 2 |
| 3 |
+----+
--->回滚未移除任何行
```

如果AUTOCOMMIT设置为true，则COMMIT和ROLLBACK无效

## 第45.3节：使用JDBC驱动的事务

使用JDBC驱动的事务用于控制事务何时提交和回滚。  
使用JDBC驱动创建与MySQL服务器的连接

[MySQL的JDBC驱动](#)可在此下载

让我们从使用JDBC驱动获取数据库连接开始

```
Class.forName("com.mysql.jdbc.Driver");
Connection con = DriverManager.getConnection(DB_CONNECTION_URL,DB_USER,USER_PASSWORD);
--->连接 url 示例 "jdbc:mysql://localhost:3306/testDB");
```

字符集：这表示客户端将使用什么字符集向服务器发送 SQL 语句。它还指定服务器应使用什么字符集将结果发送回客户端。

这应在创建与服务器的连接时提及。因此连接字符串应如下所示，

```
jdbc:mysql://localhost:3306/testDB?useUnicode=true&characterEncoding=utf8
```

有关字符集和排序规则的更多详细信息，请参见此处当你打开连

接时，AUTOCOMMIT模式默认设置为true，应该将其更改为false以开始事务。

```
| 1 |
+----+
--->Rollback removed all rows which all are not committed
```

Once COMMIT is executed, then ROLLBACK will not cause anything

```
mysql> INSERT INTO testTable VALUES (2), (3);
mysql> SELECT * FROM testTable;
mysql> COMMIT;

-----
| tId |
+----+
| 1 |
| 2 |
| 3 |
+----+

--->Rollback executed now
mysql> ROLLBACK;

mysql> SELECT * FROM testTable;
-----
| tId |
+----+
| 1 |
| 2 |
| 3 |
+----+
--->Rollback not removed any rows
```

If AUTOCOMMIT is set true, then COMMIT and ROLLBACK is useless

## Section 45.3: Transaction using JDBC Driver

Transaction using JDBC driver is used to control how and when a transaction should commit and rollback.  
Connection to MySQL server is created using JDBC driver

[JDBC driver for MySQL](#) can be downloaded here

Lets start with getting a connection to database using JDBC driver

```
Class.forName("com.mysql.jdbc.Driver");
Connection con = DriverManager.getConnection(DB_CONNECTION_URL,DB_USER,USER_PASSWORD);
--->Example for connection url "jdbc:mysql://localhost:3306/testDB");
```

**Character Sets**: This indicates what character set the client will use to send SQL statements to the server. It also specifies the character set that the server should use for sending results back to the client.

This should be mentioned while creating connection to server. So the connection string should be like,

```
jdbc:mysql://localhost:3306/testDB?useUnicode=true&characterEncoding=utf8
```

See this for more details about Character Sets and Collations

When you open connection, the AUTOCOMMIT mode is set to true by default, that should be changed false to start transaction.

```
con.setAutoCommit(false);
```

你应始终在打开连接后立即调用setAutoCommit()方法。

否则使用**START TRANSACTION**或**BEGIN WORK**来开始一个新事务。使用**START TRANSACTION**或**BEGIN WORK**时，无需将AUTOCOMMIT设置为false。它会自动被禁用。

现在你可以开始事务了。下面是一个完整的 JDBC 事务示例。

```
package jdbcTest;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class accTrans {

    public static void doTransfer(double transAmount,int customerIdFrom,int customerIdTo) {

        Connection con = null;
        PreparedStatement pstmt = null;
        ResultSet rs = null;

        try {
            String DB_CONNECTION_URL =
"jdbc:mysql://localhost:3306/testDB?useUnicode=true&characterEncoding=utf8";

            Class.forName("com.mysql.jdbc.Driver");
con = DriverManager.getConnection(DB_CONNECTION_URL,DB_USER,USER_PASSWORD);

            --->将自动提交设置为false
con.setAutoCommit(false);
            ---> 或使用 con.START TRANSACTION / con.BEGIN WORK

            --->开始事务的SQL语句
            --->检查金额可用性
            double availableAmt    = 0;
pstmt = con.prepareStatement("SELECT ledgerAmt FROM accTable WHERE customerId=? FOR
UPDATE");
pstmt.setInt(1, customerIdFrom);
            rs = pstmt.executeQuery();
            if(rs.next())
availableAmt    = rs.getDouble(1);

            if(availableAmt >= transAmount)
            {
                ---> 执行转账
                --->从 customerIdFrom 扣款
pstmt = con.prepareStatement("UPDATE accTable SET ledgerAmt=ledgerAmt-? WHERE
customerId=?");
pstmt.setDouble(1, transAmount);
                pstmt.setInt(2, customerIdFrom);
                pstmt.executeUpdate();

                ---> 在 customerIdTo 中存入金额
pstmt = con.prepareStatement("UPDATE accTable SET ledgerAmt=ledgerAmt+? WHERE
customerId=?");
pstmt.setDouble(1, transAmount);
                pstmt.setInt(2, customerIdTo);
                pstmt.executeUpdate();
            }
        }
    }
}
```

```
con.setAutoCommit(false);
```

You should always call setAutoCommit() method right after you open a connection.

Otherwise use **START TRANSACTION** or **BEGIN WORK** to start a new transaction. By using **START TRANSACTION** or **BEGIN WORK**, no need to change AUTOCOMMIT false. That will be automatically disabled.

Now you can start transaction. See a complete JDBC transaction example below.

```
package jdbcTest;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class accTrans {

    public static void doTransfer(double transAmount,int customerIdFrom,int customerIdTo) {

        Connection con = null;
        PreparedStatement pstmt = null;
        ResultSet rs = null;

        try {
            String DB_CONNECTION_URL =
"jdbc:mysql://localhost:3306/testDB?useUnicode=true&characterEncoding=utf8";

            Class.forName("com.mysql.jdbc.Driver");
con = DriverManager.getConnection(DB_CONNECTION_URL,DB_USER,USER_PASSWORD);

            --->set auto commit to false
con.setAutoCommit(false);
            ---> or use con.START TRANSACTION / con.BEGIN WORK

            --->Start SQL Statements for transaction
            --->Checking availability of amount
            double availableAmt    = 0;
pstmt = con.prepareStatement("SELECT ledgerAmt FROM accTable WHERE customerId=? FOR
UPDATE");
pstmt.setInt(1, customerIdFrom);
            rs = pstmt.executeQuery();
            if(rs.next())
                availableAmt    = rs.getDouble(1);

            if(availableAmt >= transAmount)
            {
                ---> Do Transfer
                ---> taking amount from cutomerIdFrom
pstmt = con.prepareStatement("UPDATE accTable SET ledgerAmt=ledgerAmt-? WHERE
customerId=?");
pstmt.setDouble(1, transAmount);
                pstmt.setInt(2, customerIdFrom);
                pstmt.executeUpdate();

                ---> depositing amount in cutomerIdTo
pstmt = con.prepareStatement("UPDATE accTable SET ledgerAmt=ledgerAmt+? WHERE
customerId=?");
pstmt.setDouble(1, transAmount);
                pstmt.setInt(2, customerIdTo);
                pstmt.executeUpdate();
            }
        }
    }
}
```

```

con.commit();
}
--->如果你之前执行过任何插入、更新或删除操作
----> 这次可用性检查，则包含此 else部分
/*else { --->如果可用性低于要求，则回滚事务
    con.rollback();
}*/

} 捕获 (SQLException ex) {
    --->在发生任何错误时回滚事务
    con.rollback();
} finally {
    try {
        if(rs != null) rs.close();
        if(pstmt != null) pstmt.close();
        if(con != null) con.close();
    }
}
}

public static void main(String[] args) {
    doTransfer(500, 1020, 1021);
    -->doTransfer(transAmount, customerIdFrom, customerIdTo);
}
}

```

JDBC 事务确保事务块内的所有 SQL 语句都成功执行，如果事务块内的任一 SQL 语句失败，则中止并回滚事务块内的所有操作。

```

        con.commit();
    }
    --->If you performed any insert,update or delete operations before
    ----> this availability check, then include this else part
    /*else { --->Rollback the transaction if availability is less than required
        con.rollback();
    }*/

} catch (SQLException ex) {
    ---> Rollback the transaction in case of any error
    con.rollback();
} finally {
    try {
        if(rs != null) rs.close();
        if(pstmt != null) pstmt.close();
        if(con != null) con.close();
    }
}
}

public static void main(String[] args) {
    doTransfer(500, 1020, 1021);
    -->doTransfer(transAmount, customerIdFrom, customerIdTo);
}
}

```

JDBC transaction make sure of all SQL statements within a transaction block are executed successful, if either one of the SQL statement within transaction block is failed, abort and rollback everything within the transaction block.

# 第46章：日志文件

## 第46.1节：慢查询日志

慢查询日志包含执行时间达到long\_query\_time秒的查询日志事件。例如，执行时间最长可达10秒。要查看当前设置的时间阈值，请执行以下命令：

```
SELECT @@long_query_time;
+-----+
| @@long_query_time |
+-----+
|          10.000000 |
+-----+
```

它可以作为全局变量设置，在my.cnf或my.ini文件中。也可以通过连接设置，尽管这不常见。该值可以设置在0到10（秒）之间。应该使用什么值？

- 10太高，几乎没用；
- 2是一个折中；
- 0.5及其他小数也是可能的；
- 0会捕获所有查询；这可能会非常快地填满磁盘，但非常有用。

慢查询的捕获要么开启，要么关闭。并且还指定了日志文件。以下内容捕获了这些概念：

```
SELECT @@slow_query_log; -- 当前是否启用捕获？（1=开启, 0=关闭）
SELECT @@slow_query_log_file; -- 捕获文件名。位于datadir目录下
SELECT @@datadir; -- 查看当前捕获文件的位置

SET GLOBAL slow_query_log=0; -- 关闭
-- 备份慢查询日志捕获文件。然后删除它。
SET GLOBAL slow_query_log=1; -- 重新开启（会创建新的空文件）
```

更多信息请参见MySQL手册页慢查询日志注意：上述关于开启/关闭慢查询日志的

信息在5.6版本（？）中有所更改；旧版本有另一种机制。

查看是什么拖慢系统速度的“最佳”方法：

```
long_query_time=...
开启慢查询日志
运行几个小时
关闭慢查询日志（或提高阈值）
运行 pt-query-digest 查找“最差”的几个查询。或者使用 mysqldumpslow -s t
```

## 第46.2节：列表

- 通用日志 - 所有查询 - 见变量 general\_log
- 慢查询日志 - 查询时间超过 long\_query\_time - slow\_query\_log\_file
- 二进制日志 - 用于复制和备份 - log\_bin\_basename
- 中继日志 - 也用于复制
- general 错误日志 - mysqld.err

# Chapter 46: Log files

## Section 46.1: Slow Query Log

The Slow Query Log consists of log events for queries taking up to long\_query\_time seconds to finish. For instance, up to 10 seconds to complete. To see the time threshold currently set, issue the following:

```
SELECT @@long_query_time;
+-----+
| @@long_query_time |
+-----+
|          10.000000 |
+-----+
```

It can be set as a GLOBAL variable, in my.cnf or my.ini file. Or it can be set by the connection, though this is unusual. The value can be set between 0 to 10 (seconds). What value to use?

- 10 is so high as to be almost useless;
- 2 is a compromise;
- 0.5 and other fractions are possible;
- 0 captures everything; this could fill up disk dangerously fast, but can be very useful.

The capturing of slow queries is either turned on or off. And the file logged to is also specified. The below captures these concepts:

```
SELECT @@slow_query_log; -- Is capture currently active? (1=On, 0=Off)
SELECT @@slow_query_log_file; -- filename for capture. Resides in datadir
SELECT @@datadir; -- to see current value of the location for capture file

SET GLOBAL slow_query_log=0; -- Turn Off
-- make a backup of the Slow Query Log capture file. Then delete it.
SET GLOBAL slow_query_log=1; -- Turn it back On (new empty file is created)
```

For more information, please see the MySQL Manual Page [The Slow Query Log](#)

Note: The above information on turning on/off the slowlog was changed in 5.6(?); older version had another mechanism.

The "best" way to see what is slowing down your system:

```
long_query_time=...
turn on the slowlog
run for a few hours
turn off the slowlog (or raise the cutoff)
run pt-query-digest to find the 'worst' couple of queries. Or mysqldumpslow -s t
```

## Section 46.2: A List

- General log - all queries - see VARIABLE general\_log
- Slow log - queries slower than long\_query\_time - slow\_query\_log\_file
- Binlog - for replication and backup - log\_bin\_basename
- Relay log - also for replication
- general errors - mysqld.err



- 启动/停止日志 - mysql.log（不太重要）- log\_error
- InnoDB 重做日志 - iblog\*

查看变量basedir和datadir，了解许多日志的默认位置

部分日志由其他变量控制开启/关闭。部分日志写入文件，部分写入表。

（给审阅者的备注：此处需要更多细节和解释。）

文档编写者：请包括每种日志类型的默认位置和名称，适用于Windows和\*nix系统。（或者至少尽可能多地提供。）

## 第46.3节：通用查询日志

通用查询日志包含来自客户端连接、断开连接和查询的一般信息列表。它对于调试非常宝贵，但会对性能造成阻碍（引用？）。

下面是通用查询日志的示例视图：

```
36 Query insert [questions]_c23(qId,ownerId,title,votes,answers,isClosed,closeVotes,views,ownerRep,
  comments,answeredAccepted,askDate,closeDate,lastScanDate,ign,bn,pvtc,
  mainTagForImport,prepStatus,touches,status,status_bef_change,cv_bef_change,max_cv_reached)
  values(38666373, 1322183, 'How to post a numeric value in c#', 0, 1, 0, 0, 50, 1,
  0, 0, '2016-07-29 19:40:32', null, now(), 0, 0, 0,
  'c%23',0,1,'0','',0,0)
  on duplicate key update title='How to post a numeric value in c#', votes=0, answers=1, views=50,ownerRep=1,
  answeredAccepted=0,lastScanDate=now(), touches=touches+1,status='0'
```

确定当前是否正在捕获通用日志：

```
SELECT @@general_log; -- 1 = 捕获处于激活状态；0 = 未激活。
```

确定捕获文件的文件名：

```
SELECT @@general_log_file; -- 捕获文件的完整路径
```

如果未显示文件的完整路径，则该文件存在于datadir目录中。

Windows示例：

```
+-----+
| @@general_log_file |
+-----+
| C:\ProgramData\MySQL\MySQL Server 5.7\Data\GuySmiley.log |
+-----+
```

Linux：

```
+-----+
| @@general_log_file |
+-----+
| /var/lib/mysql/ip-ww-xx-yy-zz.log |
+-----+
```

当对general\_log\_file全局变量进行更改时，新的日志会保存在datadir中。然而，通过检查该变量可能不再能反映完整路径。

- start/stop - mysql.log (not very interesting) - log\_error
- InnoDB redo log - iblog\*

See the variables basedir and datadir for default location for many logs

Some logs are turned on/off by other VARIABLES. Some are either written to a file or to a table.

(Note to reviewers: This needs more details and more explanation.)

Documenters: please include the default location and name for each log type, for both Windows and \*nix. (Or at least as much as you can.)

## Section 46.3: General Query Log

The General Query Log contains a listing of general information from client connects, disconnects, and queries. It is invaluable for debugging, yet it poses as a hindrance to performance (citation?).

An example view of a General Query Log is seen below:

```
36 Query insert [questions]_c23(qId,ownerId,title,votes,answers,isClosed,closeVotes,views,ownerRep,
  comments,answeredAccepted,askDate,closeDate,lastScanDate,ign,bn,pvtc,
  mainTagForImport,prepStatus,touches,status,status_bef_change,cv_bef_change,max_cv_reached)
  values(38666373, 1322183, 'How to post a numeric value in c#', 0, 1, 0, 0, 50, 1,
  0, 0, '2016-07-29 19:40:32', null, now(), 0, 0, 0,
  'c%23',0,1,'0','',0,0)
  on duplicate key update title='How to post a numeric value in c#', votes=0, answers=1, views=50,ownerRep=1,
  answeredAccepted=0,lastScanDate=now(), touches=touches+1,status='0'
```

To determine if the General Log is currently being captured:

```
SELECT @@general_log; -- 1 = Capture is active; 0 = It is not.
```

To determine the filename of the capture file:

```
SELECT @@general_log_file; -- Full path to capture file
```

If the fullpath to the file is not shown, the file exists in the datadir.

Windows example:

```
+-----+
| @@general_log_file |
+-----+
| C:\ProgramData\MySQL\MySQL Server 5.7\Data\GuySmiley.log |
+-----+
```

Linux:

```
+-----+
| @@general_log_file |
+-----+
| /var/lib/mysql/ip-ww-xx-yy-zz.log |
+-----+
```

When changes are made to the general\_log\_file GLOBAL variable, the new log is saved in the datadir. However, the fullpath may no longer be reflected by examining the variable.

如果配置文件中没有general\_log\_file的条目，默认会使用@@hostname.log，保存在datadir中。

最佳实践是关闭捕获。将日志文件保存到备份目录，文件名应反映捕获的开始/结束日期时间。如果该文件没有通过文件系统move操作被移动，则删除之前的文件。为日志文件建立一个新文件名并开启捕获（如下所示）。最佳实践还包括仔细判断当前是否需要捕获。通常，捕获仅在调试时开启。

备份日志的典型文件系统文件名可能是：

```
/LogBackup/GeneralLog_20160802_1520_to_20160802_1815.log
```

日期和时间是文件名的一部分，表示一个时间范围。

对于Windows，请注意以下带有设置更改的序列。

```
SELECT @@general_log; -- 0. 未被捕获
SELECT @@general_log_file; -- C:\ProgramData\MySQL\MySQL Server 5.6\Data\GuySmiley.log
SELECT @@datadir; -- C:\ProgramData\MySQL\MySQL Server 5.7\Data\
SET GLOBAL general_log_file='GeneralLogBegin_20160803_1420.log'; -- 日期时间提示
SET GLOBAL general_log=1; -- 开启实际日志捕获。文件创建在 `datadir` 下
SET GLOBAL general_log=0; -- 关闭日志记录
```

Linux 类似。这些代表动态更改。服务器任何重启都会读取配置文件设置。

关于配置文件，请考虑以下相关变量设置：

```
[mysqld]
general_log_file = /path/to/currentquery.log
general_log      = 1
```

此外，变量 log\_output 可以配置为 **TABLE** 输出，而不仅仅是 FILE。有关详情，请参见 [Destinations](#)。

请参阅 MySQL 手册页通用查询日志。

## 第46.4节：错误日志

错误日志记录了服务器的启动和停止信息，以及遇到的关键事件。

下面是其内容的一个示例：

```
2016-08-02 20:40:39 2420 [Note] Shutting down plugin 'binlog'
2016-08-02 20:40:39 2420 [Note] mysqld: Shutdown complete

2016-08-02 20:43:11 2888 [Note] Plugin 'FEDERATED' is disabled.
2016-08-02 20:43:11 2888 [Note] InnoDB: Using atomics to ref count buffer pool pages
2016-08-02 20:43:11 2888 [Note] InnoDB: The InnoDB memory heap is disabled
```

变量log\_error保存错误日志文件的路径。

如果配置文件中没有log\_error的条目，系统将默认其值为@@hostname.err，位于datadir目录下。注意log\_error不是动态变量。因此，修改需要通过cnf或ini文件更改并重启服务器（或参见本页底部手册链接中的“刷新并重命名错误日志文件”）。

In the case of no entry for general\_log\_file in the configuration file, it will default to @@hostname.log in the datadir.

Best practices are to turn OFF capture. Save the log file to a backup directory with a filename reflecting the begin/end datetime of the capture. Deleting the prior file if a filesystem *move* did not occur of that file. Establish a new filename for the log file and turn capture ON (all show below). Best practices also include a careful determination if you even want to capture at the moment. Typically, capture is ON for debugging purposes only.

A typical filesystem filename for a backed-up log might be:

```
/LogBackup/GeneralLog_20160802_1520_to_20160802_1815.log
```

where the date and time are part to the filename as a range.

For Windows note the following sequence with setting changes.

```
SELECT @@general_log; -- 0. Not being captured
SELECT @@general_log_file; -- C:\ProgramData\MySQL\MySQL Server 5.6\Data\GuySmiley.log
SELECT @@datadir; -- C:\ProgramData\MySQL\MySQL Server 5.7\Data\
SET GLOBAL general_log_file='GeneralLogBegin_20160803_1420.log'; -- datetime clue
SET GLOBAL general_log=1; -- Turns on actual log capture. File is created under `datadir`
SET GLOBAL general_log=0; -- Turn logging off
```

Linux is similar. These would represent dynamic changes. Any restart of the server would pick up configuration file settings.

As for the configuration file, consider the following relevant variable settings:

```
[mysqld]
general_log_file = /path/to/currentquery.log
general_log      = 1
```

In addition, the variable log\_output can be configured for **TABLE** output, not just FILE. For that, please see [Destinations](#).

Please see the MySQL Manual Page [The General Query Log](#).

## Section 46.4: Error Log

The Error Log is populated with start and stop information, and critical events encountered by the server.

The following is an example of its contents:

```
2016-08-02 20:40:39 2420 [Note] Shutting down plugin 'binlog'
2016-08-02 20:40:39 2420 [Note] mysqld: Shutdown complete

2016-08-02 20:43:11 2888 [Note] Plugin 'FEDERATED' is disabled.
2016-08-02 20:43:11 2888 [Note] InnoDB: Using atomics to ref count buffer pool pages
2016-08-02 20:43:11 2888 [Note] InnoDB: The InnoDB memory heap is disabled
```

The variable log\_error holds the path to the log file for error logging.

In the absence of a configuration file entry for log\_error, the system will default its values to @@hostname.err in the datadir. Note that log\_error is not a dynamic variable. As such, changes are done through a cnf or ini file changes and a server restart (or by seeing "Flushing and Renaming the Error Log File" in the Manual Page link at the bottom here).

错误日志无法被禁用。它们对于系统健康和故障排查非常重要。此外，与通用查询日志相比，错误日志的条目较少。

全局变量log\_warnings设置详细程度，具体取决于服务器版本。以下代码片段说明了这一点：

```
SELECT @@log_warnings; -- 记录之前的设置
SET GLOBAL log_warnings=2; -- 设置大于1会增加输出 (参见服务器版本)
```

如上所示，log\_warnings是一个动态变量。

配置文件中cnf和ini文件的更改可能如下所示。

```
[mysqld]
log_error      = /path/to/CurrentError.log
log_warnings   = 2
```

MySQL 5.7.2将警告级别的详细程度扩展到了3，并添加了全局变量log\_error\_verbosity。同样，它是在5.7.2版本中引入的。[该变量](#)可以动态设置，也可以作为变量检查，或者通过cnf或ini配置文件设置。

截至MySQL 5.7.2版本：

```
[mysqld]
log_error      = /path/to/CurrentError.log
log_warnings   = 2
log_error_verbosity = 3
```

请参阅MySQL手册页中名为The Error Log的[章节](#)，特别是关于刷新和重命名错误日志文件的部分，以及其Error Log Verbosity章节，涉及与log\_warnings和error\_log\_verbosity相关的版本内容。

Logging cannot be disabled for errors. They are important for system health while troubleshooting problems. Also, entries are infrequent compared to the General Query Log.

The GLOBAL variable log\_warnings sets the level for verbosity which varies by server version. The following snippet illustrates:

```
SELECT @@log_warnings; -- make a note of your prior setting
SET GLOBAL log_warnings=2; -- setting above 1 increases output (see server version)
```

log\_warnings as seen above is a dynamic variable.

Configuration file changes in cnf and ini files might look like the following.

```
[mysqld]
log_error      = /path/to/CurrentError.log
log_warnings   = 2
```

MySQL 5.7.2 expanded the warning level verbosity to 3 and added the GLOBAL log\_error\_verbosity. Again, it was [introduced](#) in 5.7.2. It can be set dynamically and checked as a variable or set via cnf or ini configuration file settings.

As of MySQL 5.7.2:

```
[mysqld]
log_error      = /path/to/CurrentError.log
log_warnings   = 2
log_error_verbosity = 3
```

Please see the MySQL Manual Page entitled [The Error Log](#) especially for Flushing and Renaming the Error Log file, and its *Error Log Verbosity* section with versions related to log\_warnings and error\_log\_verbosity.

# 第47章：集群

## 第47.1节：消歧义

“MySQL Cluster”的消歧义...

- NDB 集群——一种专门的、主要基于内存的引擎。使用不广泛。
- Galera 集群，也称为 Percona XtraDB 集群、PXC 或带有 Galera 的 MariaDB。-- 一个非常好的 MySQL 高可用性解决方案；它超越了复制。

请参阅关于这些“集群”变体的各个页面。

有关“聚簇索引”，请参阅PRIMARY KEY相关页面。

# Chapter 47: Clustering

## Section 47.1: Disambiguation

"MySQL Cluster" disambiguation...

- NDB Cluster -- A specialized, mostly in-memory, engine. Not widely used.
- Galera Cluster aka Percona XtraDB Cluster aka PXC aka MariaDB with Galera. -- A very good High Availability solution for MySQL; it goes beyond Replication.

See individual pages on those variants of "Cluster".

For "clustered index" see page(s) on **PRIMARY KEY**.



# 第48章：分区

## 第48.1节：范围分区

按范围分区的表是这样分区的：每个分区包含分区表达式值位于给定范围内的行。范围应当是连续的但不重叠，使用 VALUESLESS THAN操作符定义。以下几个示例中，假设你正在创建一个表，用于存储一个拥有20家视频商店（编号1到20）的连锁店的人员记录，如下所示：

```
CREATE TABLE employees (  
  id INT NOT NULL,  
  fname VARCHAR(30),  
  lname VARCHAR(30),  
  hired DATE NOT NULL DEFAULT '1970-01-01',  
    separated DATE NOT NULL DEFAULT '9999-12-31',  
    job_code INT NOT NULL,  
  store_id INT NOT NULL  
);
```

该表可以根据您的需求通过多种方式进行范围分区。一种方法是使用 store\_id 列。例如，您可以通过添加一个 PARTITION BY RANGE

子句，将表分成4个分区，如下所示：

```
ALTER TABLE employees PARTITION BY RANGE (store_id) (  
  PARTITION p0 VALUES LESS THAN (6),  
  PARTITION p1 VALUES LESS THAN (11),  
  PARTITION p2 VALUES LESS THAN (16),  
  PARTITION p3 VALUES LESS THAN MAXVALUE  
);
```

MAXVALUE 表示一个总是大于最大可能整数值的整数值（用数学语言来说，它作为一个上确界）。

基于 [MySQL 官方文档](#)。

## 第48.2节：列表分区

列表分区在许多方面类似于范围分区。与按 RANGE 分区一样，每个分区必须被明确地定义。两种分区类型的主要区别在于，列表分区中，每个分区是基于某列值是否属于某个值列表中的成员来定义和选择的，而不是属于一组连续的值范围内的某个范围。这是通过使用 PARTITION BY LIST(expr) 来实现的，其中 expr 是一个列值或基于列值的表达式并返回一个整数值，然后通过 VALUES IN (value\_list) 定义每个分区，value\_list 是一个用逗号分隔的整数列表。

对于以下示例，我们假设要分区的表的基本定义已由 CREATE TABLE 语句如下所示：

```
CREATE TABLE employees (  
  id INT NOT NULL,  
  fname VARCHAR(30),  
  lname VARCHAR(30),  
  hired DATE NOT NULL DEFAULT '1970-01-01',  
    separated DATE NOT NULL DEFAULT '9999-12-31',
```

# Chapter 48: Partitioning

## Section 48.1: RANGE Partitioning

A table that is partitioned by range is partitioned in such a way that each partition contains rows for which the partitioning expression value lies within a given range. Ranges should be contiguous but not overlapping, and are defined using the VALUES LESS THAN operator. For the next few examples, suppose that you are creating a table such as the following to hold personnel records for a chain of 20 video stores, numbered 1 through 20:

```
CREATE TABLE employees (  
  id INT NOT NULL,  
  fname VARCHAR(30),  
  lname VARCHAR(30),  
  hired DATE NOT NULL DEFAULT '1970-01-01',  
    separated DATE NOT NULL DEFAULT '9999-12-31',  
    job_code INT NOT NULL,  
  store_id INT NOT NULL  
);
```

This table can be partitioned by range in a number of ways, depending on your needs. One way would be to use the store\_id column. For instance, you might decide to partition the table 4 ways by adding a PARTITION BY RANGE clause as shown here:

```
ALTER TABLE employees PARTITION BY RANGE (store_id) (  
  PARTITION p0 VALUES LESS THAN (6),  
  PARTITION p1 VALUES LESS THAN (11),  
  PARTITION p2 VALUES LESS THAN (16),  
  PARTITION p3 VALUES LESS THAN MAXVALUE  
);
```

MAXVALUE represents an integer value that is always greater than the largest possible integer value (in mathematical language, it serves as a least upper bound).

based on [MySQL official document](#).

## Section 48.2: LIST Partitioning

List partitioning is similar to range partitioning in many ways. As in partitioning by RANGE, each partition must be explicitly defined. The chief difference between the two types of partitioning is that, in list partitioning, each partition is defined and selected based on the membership of a column value in one of a set of value lists, rather than in one of a set of contiguous ranges of values. This is done by using PARTITION BY LIST(expr) where expr is a column value or an expression based on a column value and returning an integer value, and then defining each partition by means of a VALUES IN (value\_list), where value\_list is a comma-separated list of integers.

For the examples that follow, we assume that the basic definition of the table to be partitioned is provided by the CREATE TABLE statement shown here:

```
CREATE TABLE employees (  
  id INT NOT NULL,  
  fname VARCHAR(30),  
  lname VARCHAR(30),  
  hired DATE NOT NULL DEFAULT '1970-01-01',  
    separated DATE NOT NULL DEFAULT '9999-12-31',
```



```
job_code INT,
  store_id INT
);
```

假设有20家录像店分布在4个加盟店中，如下表所示。

地区	店铺编号
北部	3, 5, 6, 9, 17
东部	1, 2, 10, 11, 19, 20
西部	4, 12, 13, 14, 18
中部	7, 8, 15, 16

将此表划分为同一区域的门店行存储在同一个分区中

```
ALTER TABLE employees PARTITION BY LIST(store_id) (
  PARTITION pNorth VALUES IN (3,5,6,9,17),
  PARTITION pEast  VALUES IN (1,2,10,11,19,20),
  PARTITION pWest  VALUES IN (4,12,13,14,18),
  PARTITION pCentral VALUES IN (7,8,15,16)
);
```

基于 [MySQL 官方文档](#)。

## 第48.3节：HASH分区

HASH分区主要用于确保数据在预定数量的分区之间均匀分布。使用范围或列表分区时，必须明确指定给定列值或列值集合存储在哪个分区；而使用哈希分区时，MySQL会为你处理此事，你只需指定一个基于列值的列值或表达式进行哈希，以及分区表要划分的分区数量。

以下语句创建了一个在store\_id列上使用哈希分区并划分为4个分区的表：

```
CREATE TABLE employees (
  id INT NOT NULL,
  fname VARCHAR(30),
  lname VARCHAR(30),
  hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT,
  store_id INT
)
PARTITION BY HASH(store_id)
PARTITIONS 4;
```

如果不包含PARTITIONS子句，分区数量默认为1。

基于 [MySQL 官方文档](#)。

```
job_code INT,
  store_id INT
);
```

Suppose that there are 20 video stores distributed among 4 franchises as shown in the following table.

Region	Store ID Numbers
North	3, 5, 6, 9, 17
East	1, 2, 10, 11, 19, 20
West	4, 12, 13, 14, 18
Central	7, 8, 15, 16

To partition this table in such a way that rows for stores belonging to the same region are stored in the same partition

```
ALTER TABLE employees PARTITION BY LIST(store_id) (
  PARTITION pNorth VALUES IN (3,5,6,9,17),
  PARTITION pEast  VALUES IN (1,2,10,11,19,20),
  PARTITION pWest  VALUES IN (4,12,13,14,18),
  PARTITION pCentral VALUES IN (7,8,15,16)
);
```

based on [MySQL official document](#).

## Section 48.3: HASH Partitioning

Partitioning by HASH is used primarily to ensure an even distribution of data among a predetermined number of partitions. With range or list partitioning, you must specify explicitly into which partition a given column value or set of column values is to be stored; with hash partitioning, MySQL takes care of this for you, and you need only specify a column value or expression based on a column value to be hashed and the number of partitions into which the partitioned table is to be divided.

The following statement creates a table that uses hashing on the store\_id column and is divided into 4 partitions:

```
CREATE TABLE employees (
  id INT NOT NULL,
  fname VARCHAR(30),
  lname VARCHAR(30),
  hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT,
  store_id INT
)
PARTITION BY HASH(store_id)
PARTITIONS 4;
```

If you do not include a PARTITIONS clause, the number of partitions defaults to 1.

based on [MySQL official document](#).

# 第49章：复制

## 第49.1节：主从复制设置

考虑两个用于复制设置的MySQL服务器，一个作为主服务器，另一个作为从服务器。

我们将配置主服务器，使其记录所有执行的操作日志。我们将配置从服务器，使其查看主服务器上的日志，并且每当主服务器的日志发生变化时，从服务器应执行相同的操作。

### 主服务器配置

首先，我们需要在主服务器上创建一个用户。该用户将被从服务器用来与主服务器建立连接。

```
CREATE USER 'user_name'@'%' IDENTIFIED BY 'user_password';
GRANT REPLICATION SLAVE ON *.* TO 'user_name'@'%' ;
FLUSH PRIVILEGES;
```

根据您的用户名和密码更改user\_name和user\_password。

现在应编辑my.inf（Linux中为my.cnf）文件。在[mysqld]部分包含以下行。

```
server-id = 1
log-bin = mysql-bin.log
binlog-do-db = your_database
```

第一行用于为该MySQL服务器分配一个ID。

第二行告诉MySQL开始在指定的日志文件中写入日志。在Linux中，这可以配置为log-bin = /home/mysql/logs/mysql-bin.log。如果你在一个已经使用过复制功能的MySQL服务器上启动复制，确保该目录中没有任何复制日志。

第三行用于配置我们将要写入日志的数据库。你应该替换your\_database为你的数据库名称。

确保没有启用skip-networking，并重启MySQL服务器（主服务器）

### 从服务器配置

my.inf文件也应在从服务器中编辑。在[mysqld]部分包含以下行。

```
server-id = 2
master-host = master_ip_address
master-connect-retry = 60

master-user = user_name
master-password = user_password
replicate-do-db = your_database

relay-log = slave-relay.log
relay-log-index = slave-relay-log.index
```

第一行用于为此MySQL服务器分配一个ID。该ID应当是唯一的。

# Chapter 49: Replication

## Section 49.1: Master - Slave Replication Setup

Consider 2 MySQL Servers for replication setup, one is a Master and the other is a Slave.

We are going to configure the Master that it should keep a log of every action performed on it. We are going to configure the Slave server that it should look at the log on the Master and whenever changes happens in log on the Master, it should do the same thing.

### Master Configuration

First of all, we need to create a user on the Master. This user is going to be used by Slave to create a connection with the Master.

```
CREATE USER 'user_name'@'%' IDENTIFIED BY 'user_password';
GRANT REPLICATION SLAVE ON *.* TO 'user_name'@'%' ;
FLUSH PRIVILEGES;
```

Change user\_name and user\_password according to your Username and Password.

Now my.inf (my.cnf in Linux) file should be edited. Include the following lines in [mysqld] section.

```
server-id = 1
log-bin = mysql-bin.log
binlog-do-db = your_database
```

The first line is used to assign an ID to this MySQL server.

The second line tells MySQL to start writing a log in the specified log file. In Linux this can be configured like log-bin = /home/mysql/logs/mysql-bin.log. If you are starting replication in a MySQL server in which replication has already been used, make sure this directory is empty of all replication logs.

The third line is used to configure the database for which we are going to write log. You should replace your\_database with your database name.

Make sure skip-networking has not been enabled and restart the MySQL server(Master)

### Slave Configuration

my.inf file should be edited in Slave also. Include the following lines in [mysqld] section.

```
server-id = 2
master-host = master_ip_address
master-connect-retry = 60

master-user = user_name
master-password = user_password
replicate-do-db = your_database

relay-log = slave-relay.log
relay-log-index = slave-relay-log.index
```

The first line is used to assign an ID to this MySQL server. This ID should be unique.

第二行是主服务器的IP地址。请根据您的主系统IP进行更改。

第三行用于设置重试限制的秒数。

接下来的两行告诉从服务器用户名和密码，从而使其能够连接主服务器。

下一行设置需要复制的数据库。

最后两行用于分配relay-log和relay-log-index文件名。

确保skip-networking未被启用，然后重启MySQL服务器（从服务器）

将数据复制到从服务器

如果主服务器上的数据不断增加，我们必须阻止主服务器上的所有数据库访问，以防止任何数据被添加。这可以通过在主服务器上运行以下语句来实现。

FLUSH TABLES WITH READ LOCK;

如果没有数据添加到服务器，可以跳过上述步骤。

我们将使用mysqldump对主服务器的数据进行备份

mysqldump your\_database -u root -p > D://Backup/backup.sql;

根据您的设置更改your\_database和备份目录。您现在将在指定位置拥有一个名为backup.sql的文件。

如果您的从库中不存在该数据库，请执行以下命令创建

CREATE DATABASE `your\_database`;

现在我们需要将备份导入从库MySQL服务器。

mysql -u root -p your\_database <D://Backup/backup.sql  
---->根据您的设置更改`your\_database`和备份目录

启动复制

要启动复制，我们需要找到主服务器上的日志文件名和日志位置。因此，在主服务器上运行以下命令

SHOW MASTER STATUS;

这将给你如下输出

文件	位置	Binlog_Do_DB	Binlog_Ignore_DB
mysql-bin.000001	130	your_database	

然后在从服务器上运行以下命令

SLAVE STOP;  
CHANGE MASTER TO MASTER\_HOST='master\_ip\_address', MASTER\_USER='user\_name',

The second line is the I.P address of the Master server. Change this according to your Master system I.P.

The third line is used to set a retry limit in seconds.

The next two lines tell the username and password to the Slave, by using which it connect the Master.

Next line set the database it needs to replicate.

The last two lines used to assign relay-log and relay-log-index file names.

Make sure skip-networking has not been enabled and restart the MySQL server(Slave)

Copy Data to Slave

If data is constantly being added to the Master, we will have to prevent all database access on the Master so nothing can be added. This can be achieved by run the following statement in Master.

FLUSH TABLES WITH READ LOCK;

If no data is being added to the server, you can skip the above step.

We are going to take data backup of the Master by using mysqldump

mysqldump your\_database -u root -p > D://Backup/backup.sql;

Change your\_database and backup directory according to your setup. You will now have a file called backup.sql in the given location.

If your database not exists in your Slave, create that by executing the following

CREATE DATABASE `your\_database`;

Now we have to import backup into Slave MySQL server.

mysql -u root -p your\_database <D://Backup/backup.sql  
---->Change `your\_database` and backup directory according to your setup

Start Replication

To start replication, we need to find the log file name and log position in the Master. So, run the following in Master

SHOW MASTER STATUS;

This will give you an output like below

File	Position	Binlog_Do_DB	Binlog_Ignore_DB
mysql-bin.000001	130	your_database	

Then run the following in Slave

SLAVE STOP;  
CHANGE MASTER TO MASTER\_HOST='master\_ip\_address', MASTER\_USER='user\_name',

```
MASTER_PASSWORD='user_password', MASTER_LOG_FILE='mysql-bin.000001', MASTER_LOG_POS=130;  
SLAVE START;
```

首先我们停止从服务器。然后告诉它准确地在主服务器日志文件中的位置。对于MASTER\_LOG\_FILE名称和MASTER\_LOG\_POS, 使用我们在主服务器上运行SHOW MASTER STATUS命令得到的值。

你应该更改MASTER\_HOST中的主服务器IP, 并相应地更改用户名和密码。

从服务器现在将处于等待状态。可以通过运行以下命令查看从服务器的状态

```
SHOW SLAVE STATUS;
```

如果您之前在主服务器上执行了FLUSH TABLES WITH READ LOCK, 请通过运行以下命令释放表锁

```
解锁表;
```

现在主服务器会记录其上执行的每个操作的日志, 且从服务器会查看主服务器上的日志。  
每当主服务器上的日志发生变化时, 从服务器会进行复制。

## 第49.2节：复制错误

每当从服务器执行查询时出现错误, MySQL会自动停止复制以识别问题并进行修复。这主要是因为某个事件导致了重复键, 或者未找到某行, 无法更新或删除。您可以跳过此类错误, 尽管不推荐这样做。

要跳过导致从服务器挂起的单个查询, 请使用以下语法

```
SET GLOBAL sql_slave_skip_counter = N;
```

该语句跳过主服务器的下N个事件。该语句仅在从服务器线程未运行时有效, 否则会产生错误。

```
STOP SLAVE;  
SET GLOBAL sql_slave_skip_counter=1;  
START SLAVE;
```

在某些情况下这是可以的。但如果该语句是多语句事务的一部分, 情况就更复杂了, 因为跳过产生错误的语句将导致整个事务被跳过。

如果您想跳过更多产生相同错误代码的查询, 并且确定跳过这些错误不会导致您的从库不一致, 且您想跳过所有这些错误, 您可以在您的my.cnf中添加一行来跳过该错误代码。

例如, 您可能想跳过所有可能遇到的重复错误

```
1062 | 错误 '重复条目 'xyz' 对键 1' 的查询
```

然后在您的my.cnf中添加以下内容

```
slave-skip-errors = 1062
```

您也可以跳过其他类型的错误或所有错误代码, 但请确保跳过这些错误不会导致您的从库不一致。以下是语法和示例

```
MASTER_PASSWORD='user_password', MASTER_LOG_FILE='mysql-bin.000001', MASTER_LOG_POS=130;  
SLAVE START;
```

First we stop the Slave. Then we tell it exactly where to look in the Master log file. For MASTER\_LOG\_FILE name and MASTER\_LOG\_POS, use the values which we got by running SHOW MASTER STATUS command on the Master.

You should change the I.P of the Master in MASTER\_HOST, and change the user and password accordingly.

The Slave will now be waiting. The status of the Slave can be viewed by run the following

```
SHOW SLAVE STATUS;
```

If you previously executed FLUSH TABLES WITH READ LOCK in Master, release the tables from lock by run the following

```
UNLOCK TABLES;
```

Now the Master keep a log for every action performed on it and the Slave server look at the log on the Master. Whenever changes happens in log on the Master, Slave replicate that.

## Section 49.2: Replication Errors

Whenever there is an error while running a query on the slave, MySQL stop replication automatically to identify the problem and fix it. This mainly because an event caused a duplicate key or a row was not found and it cannot be updated or deleted. You can skip such errors, even if this is not recommended

To skip just one query that is hanging the slave, use the following syntax

```
SET GLOBAL sql_slave_skip_counter = N;
```

This statement skips the next N events from the master. This statement is valid only when the slave threads are not running. Otherwise, it produces an error.

```
STOP SLAVE;  
SET GLOBAL sql_slave_skip_counter=1;  
START SLAVE;
```

In some cases this is fine. But if the statement is part of a multi-statement transaction, it becomes more complex, because skipping the error producing statement will cause the whole transaction to be skipped.

If you want to skip more queries which producing same error code and if you are sure that skipping those errors will not bring your slave inconsistent and you want to skip them all, you would add a line to skip that error code in your my.cnf.

For example you might want to skip all duplicate errors you might be getting

```
1062 | Error 'Duplicate entry 'xyz' for key 1' on query
```

Then add the following to your my.cnf

```
slave-skip-errors = 1062
```

You can skip also other type of errors or all error codes, but make sure that skipping those errors will not bring your slave inconsistent. The following are the syntax and examples

```
slave-skip-errors=[err_code1,err_code2,...|all]
```

```
slave-skip-errors=1062,1053
```

```
slave-skip-errors=all
```

```
slave-skip-errors=ddl_exist_errors
```

```
slave-skip-errors=[err_code1,err_code2,...|all]
```

```
slave-skip-errors=1062,1053
```

```
slave-skip-errors=all
```

```
slave-skip-errors=ddl_exist_errors
```



# 第50章：使用mysqldump备份

选项	效果
--	# 服务器登录选项
-h (--host)	要连接的主机（IP 地址或主机名）。默认是localhost（127.0.0.1）示例：-h localhost
-u (--user)	MySQL 用户
-p (--password)	MySQL 密码。重要：使用-p时，选项和密码之间不能有空格。示例：-pMyPassword
--	# 转储选项
--add-drop-database	在每个CREATE DATABASE语句之前添加一个DROP DATABASE语句。如果你想要替换服务器中的数据库。
--add-drop-table	在每个CREATE TABLE语句之前添加一个DROP TABLE语句。如果您想要替换服务器中的表，这非常有用。
--no-create-db	在转储中抑制CREATE DATABASE语句。当您确定要转储的数据库已经存在于将要加载转储的服务器时，这非常有用。
-t (--no-create-info)	在转储中抑制所有CREATE TABLE语句。当您想要转储时，这非常有用 仅使用表中的数据，并将使用转储文件填充另一个数据库/服务器中的相同表。
-d (--no-data)	不要写入表信息。这只会导出CREATE TABLE语句。适用于创建“模板”数据库
-R (--routines)	在导出中包含存储过程/函数。
-K (--disable-keys)	在插入数据之前禁用每个表的键，数据插入后启用键。这只会加快带有非唯一索引的MyISAM表的插入速度。

## 第50.1节：指定用户名和密码

```
> mysqldump -u 用户名 -p [其他选项]
输入 密码：
```

如果需要在命令行中指定密码（例如在脚本中），可以在-p选项后直接添加密码无空格：

```
> mysqldump -u 用户名 -ppassword [其他选项]
```

如果密码包含空格或特殊字符，请根据您的shell/系统使用转义。

可选的扩展形式是：

```
> mysqldump --user=username --password=password [other options]
```

(由于安全问题，不建议在命令行中显式指定密码。)

## 第50.2节：创建数据库或数据表的备份

创建整个数据库的快照：

```
mysqldump [options] db_name > filename.sql
```

创建多个数据库的快照：

```
mysqldump [options] --databases db_name1 db_name2 ... > filename.sql
```

# Chapter 50: Backup using mysqldump

Option	Effect
--	# Server login options
-h (--host)	Host (IP address or hostname) to connect to. Default is localhost (127.0.0.1) Example: -h localhost
-u (--user)	MySQL user
-p (--password)	MySQL password. <b>Important:</b> When using -p, there must not be a space between the option and the password. Example: -pMyPassword
--	# Dump options
--add-drop-database	Add a <b>DROP DATABASE</b> statement before each <b>CREATE DATABASE</b> statement. Useful if you want to replace databases in the server.
--add-drop-table	Add a <b>DROP TABLE</b> statement before each <b>CREATE TABLE</b> statement. Useful if you want to replace tables in the server.
--no-create-db	Suppress the <b>CREATE DATABASE</b> statements in the dump. This is useful when you're sure the database(s) you're dumping already exist(s) in the server where you'll load the dump.
-t (--no-create-info)	Suppress all <b>CREATE TABLE</b> statements in the dump. This is useful when you want to dump only the data from the tables and will use the dump file to populate identical tables in another database / server.
-d (--no-data)	Do not write table information. This will only dump the <b>CREATE TABLE</b> statements. Useful for creating "template" databases
-R (--routines)	Include stored procedures / functions in the dump.
-K (--disable-keys)	Disable keys for each table before inserting the data, and enable keys after the data is inserted. This speeds up inserts only in MyISAM tables with non-unique indexes.

## Section 50.1: Specifying username and password

```
> mysqldump -u username -p [other options]
Enter password:
```

If you need to specify the password on the command line (e.g. in a script), you can add it after the -p option *without* a space:

```
> mysqldump -u username -ppassword [other options]
```

If you password contains spaces or special characters, remember to use escaping depending on your shell / system.

Optionally the extended form is:

```
> mysqldump --user=username --password=password [other options]
```

(Explicitly specifying the password on the commandline is Not Recommended due to security concerns.)

## Section 50.2: Creating a backup of a database or table

Create a snapshot of a whole database:

```
mysqldump [options] db_name > filename.sql
```

Create a snapshot of multiple databases:

```
mysqldump [options] --databases db_name1 db_name2 ... > filename.sql
```

```
mysqldump [options] --all-databases > filename.sql
```

创建一个或多个数据表的快照：

```
mysqldump [options] db_name table_name... > filename.sql
```

创建一个快照排除一个或多个表：

```
mysqldump [选项] 数据库名 --ignore-table=表1 --ignore-table=表2 ... > 文件名.sql
```

文件扩展名.sql完全是风格问题。任何扩展名都可以使用。

## 第50.3节：恢复数据库或表的备份

```
mysql [选项] 数据库名 < 文件名.sql
```

注意：

- 数据库名需要是已存在的数据库；
- 您认证的用户必须有足够权限执行文件名.sql中的所有命令；
- 文件扩展名.sql完全是风格问题。任何扩展名都可以使用。
- 您不能指定要加载的表名，尽管可以指定要导出的表名。此操作必须在文件名.sql中完成。

或者，在MySQL命令行工具中，您可以使用source命令恢复（或运行任何其他脚本）：

```
源文件名.sql
```

或

```
\. filename.sql
```

## 第50.4节：从一个MySQL服务器传输数据到另一个服务器

如果你需要将数据库从一台服务器复制到另一台服务器，有两种选择：

**选项1：**

1. 在源服务器上存储转储文件
2. 将转储文件复制到目标服务器
3. 在目标服务器上加载转储文件

在源服务器上：

```
mysqldump [选项] > dump.sql
```

在目标服务器上，复制转储文件并执行：

```
mysql [选项] < dump.sql
```

**选项2：**

```
mysqldump [options] --all-databases > filename.sql
```

Create a snapshot of one or more tables:

```
mysqldump [options] db_name table_name... > filename.sql
```

Create a snapshot *excluding* one or more tables:

```
mysqldump [options] db_name --ignore-table=tbl1 --ignore-table=tbl2 ... > filename.sql
```

The file extension .sql is fully a matter of style. Any extension would work.

## Section 50.3: Restoring a backup of a database or table

```
mysql [options] db_name < filename.sql
```

Note that:

- db\_name needs to be an existing database;
- your authenticated user has sufficient privileges to execute all the commands inside your filename.sql;
- The file extension .sql is fully a matter of style. Any extension would work.
- You cannot specify a table name to load into even though you could specify one to dump from. This must be done within filename.sql.

Alternatively, when in the **MySQL Command line tool**, you can restore (or run any other script) by using the source command:

```
source filename.sql
```

or

```
\. filename.sql
```

## Section 50.4: Tranferring data from one MySQL server to another

If you need to copy a database from one server to another, you have two options:

**Option 1:**

1. Store the dump file in the source server
2. Copy the dump file to your destination server
3. Load the dump file into your destination server

On the source server:

```
mysqldump [options] > dump.sql
```

On the destination server, copy the dump file and execute:

```
mysql [options] < dump.sql
```

**Option 2:**

如果目标服务器可以连接到主机服务器，你可以使用管道将数据库从一个服务器复制到另一个服务器：

在目标服务器上

```
mysqldump [连接到源服务器的选项] | mysql [选项]
```

同样，该脚本也可以在源服务器上运行，推送到目标服务器。无论哪种情况，这通常比选项1快得多。

## 第50.5节：带压缩的远程服务器mysqldump

为了使用传输线上的压缩以加快传输速度，向mysqldump传递--compress选项。例如：

```
mysqldump -h db.example.com -u username -p --compress dbname > dbname.sql
```

重要提示：如果你不想锁定源数据库，也应该包含--lock-tables=false。但这样可能无法获得内部一致的数据库镜像。

如果还想保存为压缩文件，可以通过管道传给gzip。

```
mysqldump -h db.example.com -u username -p --compress dbname | gzip --stdout > dbname.sql.gz
```

## 第50.6节：恢复一个gzip压缩的mysqldump文件而不解压

```
gunzip -c dbname.sql.gz | mysql dbname -u username -p
```

注意：-c 表示将输出写入标准输出。

## 第50.7节：备份包含存储过程和函数的数据库

默认情况下，mysqldump 不会生成存储过程和函数，你需要添加参数--routines（或-R）：

```
mysqldump -u username -p -R db_name > dump.sql
```

使用--routines时，创建和修改时间戳不会被保留，取而代之的是你应该导出并重新加载mysql.proc的内容。

## 第50.8节：压缩备份直接到Amazon S3

如果你想对大型MySQL安装进行完整备份且本地存储空间不足，可以直接将备份导出并压缩到Amazon S3桶中。最好不要在命令中包含数据库密码，这也是一种良好做法：

```
mysqldump -u root -p --host=localhost --opt --skip-lock-tables --single-transaction \
--verbose --hex-lob --routines --triggers --all-databases |
gzip -9 | s3cmd put - s3://s3-bucket/db-server-name.sql.gz
```

系统会提示输入密码，之后备份开始。

If the destination server can connect to the host server, you can use a pipeline to copy the database from one server to the other:

On the destination server

```
mysqldump [options to connect to the source server] | mysql [options]
```

Similarly, the script could be run on the source server, pushing to the destination. In either case, it is likely to be significantly faster than Option 1.

## Section 50.5: mysqldump from a remote server with compression

In order to use compression over the wire for a faster transfer, pass the --compress option to mysqldump. Example:

```
mysqldump -h db.example.com -u username -p --compress dbname > dbname.sql
```

Important: If you don't want to lock up the *source* db, you should also include --lock-tables=false. But you may not get an internally consistent db image that way.

To also save the file compressed, you can pipe to gzip.

```
mysqldump -h db.example.com -u username -p --compress dbname | gzip --stdout > dbname.sql.gz
```

## Section 50.6: restore a gzipped mysqldump file without uncompressing

```
gunzip -c dbname.sql.gz | mysql dbname -u username -p
```

Note: -c means write output to stdout.

## Section 50.7: Backup database with stored procedures and functions

By default stored procedures and functions or not generated by mysqldump, you will need to add the parameter --routines (or -R):

```
mysqldump -u username -p -R db_name > dump.sql
```

When using --routines the creation and change time stamps are not maintained, instead you should dump and reload the contents of mysql.proc.

## Section 50.8: Backup direct to Amazon S3 with compression

If you wish to make a complete backup of a large MySQL installation and do not have sufficient local storage, you can dump and compress it directly to an Amazon S3 bucket. It's also a good practice to do this without having the DB password as part of the command:

```
mysqldump -u root -p --host=localhost --opt --skip-lock-tables --single-transaction \
--verbose --hex-lob --routines --triggers --all-databases |
gzip -9 | s3cmd put - s3://s3-bucket/db-server-name.sql.gz
```

You are prompted for the password, after which the backup starts.

# 第51章：mysqlimport

参数	描述
<code>--delete -D</code>	导入文本文件前清空表
<code>--fields-optionally-enclosed-by</code>	定义引用字段的字符
<code>--fields-terminated-by</code>	字段分隔符
<code>--ignore -i</code>	遇到重复键时忽略该行
<code>--lines-terminated-by</code>	定义行终止符
<code>--密码 -p</code>	密码
<code>--端口 -P</code>	端口
<code>--替换 -r</code>	在键重复的情况下覆盖旧的条目行
<code>--用户 -u</code>	用户名
<code>--条件 -w</code>	指定一个条件

## 第51.1节：基本用法

给定制表符分隔的文件employee.txt

```
1 亚瑟·登特
2 马文
3 扎福德·比布尔布罗克斯
```

```
$ mysql --user=user --password=password mycompany -e 'CREATE TABLE employee(id INT, name VARCHAR(100), PRIMARY KEY (id))'

$ mysqlimport --user=user --password=password mycompany employee.txt
```

## 第51.2节：使用自定义字段分隔符

给定文本文件 employee.txt

```
1|亚瑟·登特
2|马文
3|扎福德·比布尔布罗克斯
```

```
$ mysqlimport --fields-terminated-by='|' mycompany employee.txt
```

## 第51.3节：使用自定义行分隔符

此示例适用于类似Windows的行尾：

```
$ mysqlimport --lines-terminated-by='\r' mycompany employee.txt
```

## 第51.4节：处理重复键

给定表Employee

id	名称
----	----

# Chapter 51: mysqlimport

Parameter	Description
<code>--delete -D</code>	empty the table before importing the text file
<code>--fields-optionally-enclosed-by</code>	define the character that quotes the fields
<code>--fields-terminated-by</code>	field terminator
<code>--ignore -i</code>	ignore the ingested row in case of duplicate-keys
<code>--lines-terminated-by</code>	define row terminator
<code>--password -p</code>	password
<code>--port -P</code>	port
<code>--replace -r</code>	overwrite the old entry row in case of duplicate-keys
<code>--user -u</code>	username
<code>--where -w</code>	specify a condition

## Section 51.1: Basic usage

Given the tab-separated file employee.txt

```
1 \t Arthur Dent
2 \t Marvin
3 \t Zaphod Beeblebrox
```

```
$ mysql --user=user --password=password mycompany -e 'CREATE TABLE employee(id INT, name VARCHAR(100), PRIMARY KEY (id))'

$ mysqlimport --user=user --password=password mycompany employee.txt
```

## Section 51.2: Using a custom field-delimiter

Given the text file employee.txt

```
1|Arthur Dent
2|Marvin
3|Zaphod Beeblebrox
```

```
$ mysqlimport --fields-terminated-by='|' mycompany employee.txt
```

## Section 51.3: Using a custom row-delimiter

This example is useful for windows-like endings:

```
$ mysqlimport --lines-terminated-by='\r\n' mycompany employee.txt
```

## Section 51.4: Handling duplicate keys

Given the table Employee

id	Name
----	------

3 尤登·弗兰克斯

以及文件employee.txt

- 1 亚瑟·登特
- 2 马文
- 3 扎福德·比布尔布罗克斯

--ignore选项将在遇到重复键时忽略该条目

```
$ mysqlimport --ignore mycompany employee.txt
```

id	名称
1	亚瑟·登特
2	马文
3	尤登·弗兰克斯

选项 --replace 将覆盖旧条目

```
$ mysqlimport --replace mycompany employee.txt
```

id	名称
1	亚瑟·登特
2	马文
3	扎福德·比布尔布罗克斯

## 第51.5节：条件导入

```
$ mysqlimport --where="id>2" mycompany employee.txt
```

## 第51.6节：导入标准csv

```
$ mysqlimport
  --字段-可选-由="'"包围
  --字段-以=,结束
  --行-以="\r"结束    mycompany e
mployee.csv
```

3 Yooden Vranx

And the file employee.txt

- 1 \t Arthur Dent
- 2 \t Marvin
- 3 \t Zaphod Beeblebrox

The --ignore option will ignore the entry on duplicate keys

```
$ mysqlimport --ignore mycompany employee.txt
```

id	Name
1	Arthur Dent
2	Marvin
3	Yooden Vranx

The --replace option will overwrite the old entry

```
$ mysqlimport --replace mycompany employee.txt
```

id	Name
1	Arthur Dent
2	Marvin
3	Zaphod Beeblebrox

## Section 51.5: Conditional import

```
$ mysqlimport --where="id>2" mycompany employee.txt
```

## Section 51.6: Import a standard csv

```
$ mysqlimport
  --fields-optionally-enclosed-by='"'
  --fields-terminated-by=,
  --lines-terminated-by="\r\n"
mycompany employee.csv
```



# 第52章：LOAD DATA INFILE

## 第52.1节：使用LOAD DATA INFILE加载大量数据到数据库

假设你有一个以';'分隔的CSV文件需要加载到数据库中，考虑以下示例。

```
1;max;male;manager;12-7-1985
2;jack;male;executive;21-8-1990
.
```

```
CREATE TABLE employee (
  `id` INT NOT NULL,
  `name` VARCHAR NOT NULL,
  `sex` VARCHAR NOT NULL,
  `designation` VARCHAR NOT NULL,
  `dob` VARCHAR NOT NULL );
```

使用以下查询将值插入该表中。

```
LOAD DATA INFILE '文件路径/文件_名.txt'
  导入到表 employee
  字段分隔符为 ';' //指定分隔值的分隔符行分隔符为 '\r'

(id,姓名,性别,职位,出生日期)
```

考虑日期格式非标准的情况。

```
1;max;男;经理;17-一月-1985
2;jack;男;主管;01-二月-1992
.
```

```
LOAD DATA INFILE '文件路径/文件_名.txt'
  导入到表 employee
  字段分隔符为 ';' //指定分隔值的分隔符行分隔符为 '\r'

(id,姓名,性别,职位,@出生日期)
设置 日期 = STR_TO_DATE(@日期, '%d-%b-%Y');
```

此 LOAD DATA INFILE 示例未指定所有可用功能。

您可以在[此处](#)查看更多关于 LOAD DATA INFILE 的参考资料。

# Chapter 52: LOAD DATA INFILE

## Section 52.1: using LOAD DATA INFILE to load large amount of data to database

Consider the following example assuming that you have a ';' delimited CSV to load into your database.

```
1;max;male;manager;12-7-1985
2;jack;male;executive;21-8-1990
.
.
.
1000000;marta;female;accountant;15-6-1992
```

Create the table for insertion.

```
CREATE TABLE `employee` (
  `id` INT NOT NULL,
  `name` VARCHAR NOT NULL,
  `sex` VARCHAR NOT NULL,
  `designation` VARCHAR NOT NULL,
  `dob` VARCHAR NOT NULL );
```

Use the following query to insert the values in that table.

```
LOAD DATA INFILE 'path of the file/file_name.txt'
  INTO TABLE employee
  FIELDS TERMINATED BY ';' //specify the delimiter separating the values
  LINES TERMINATED BY '\r\n'
(id,name,sex,designation,dob)
```

Consider the case where the date format is non standard.

```
1;max;male;manager;17-Jan-1985
2;jack;male;executive;01-Feb-1992
.
.
.
1000000;marta;female;accountant;25-Apr-1993
```

In this case you can change the format of the dob column before inserting like this.

```
LOAD DATA INFILE 'path of the file/file_name.txt'
  INTO TABLE employee
  FIELDS TERMINATED BY ';' //specify the delimiter separating the values
  LINES TERMINATED BY '\r\n'
(id,name,sex,designation,@dob)
SET date = STR_TO_DATE(@date, '%d-%b-%Y');
```

This example of LOAD DATA INFILE does not specify all the available features.

You can see more references on LOAD DATA INFILE [here](#).

## 第52.2节：加载包含重复项的数据

如果您使用LOAD DATA INFILE命令将现有数据导入表中，您常常会发现导入因重复而失败。解决此问题有几种可能的方法。

### LOAD DATA LOCAL

如果服务器启用了此选项，则可以用它来加载存在于客户端计算机上的文件，而不是服务器上的文件。一个副作用是唯一值的重复行会被忽略。

```
LOAD DATA LOCAL INFILE '文件路径/文件_名.txt'
INTO TABLE employee
```

### LOAD DATA INFILE 'fname' REPLACE

当使用replace关键字时，重复的唯一键或主键将导致现有行被新行替换

```
LOAD DATA INFILE '文件路径/文件_名.txt'
REPLACE INTO TABLE employee
```

### LOAD DATA INFILE 'fname' IGNORE

与REPLACE相反，现有行将被保留，新行将被忽略。这种行为类似于上面描述的LOCAL。但文件不必存在于客户端计算机上。

```
LOAD DATA INFILE '文件路径/文件_名.txt'
IGNORE INTO TABLE employee
```

### 通过中间表加载

有时忽略或替换所有重复项可能不是理想的选择。您可能需要根据其他列的内容做出决策。在这种情况下，最佳选择是先加载到中间表，然后再从中转移数据。

```
INSERT INTO employee SELECT * FROM intermediary WHERE ...
```

## 第52.3节：将CSV文件导入MySQL表

下面的命令将CSV文件导入具有相同列的MySQL表，同时遵守CSV的引用和转义规则。

```
load data infile '/tmp/file.csv'
into table my_table
fields terminated by ','
optionally enclosed by '"'
escaped by '"'
lines terminated by '\n'
ignore 1 lines; -- 跳过表头行
```

## Section 52.2: Load data with duplicates

If you use the **LOAD DATA INFILE** command to populate a table with existing data, you will often find that the import fails due to duplicates. There are several possible ways to overcome this problem.

### LOAD DATA LOCAL

If this option has been enabled in your server, it can be used to load a file that exists on the client computer rather than the server. A side effect is that duplicate rows for unique values are ignored.

```
LOAD DATA LOCAL INFILE 'path of the file/file_name.txt'
INTO TABLE employee
```

### LOAD DATA INFILE 'fname' REPLACE

When the replace keyword is used duplicate unique or primary keys will result in the existing row being replaced with new ones

```
LOAD DATA INFILE 'path of the file/file_name.txt'
REPLACE INTO TABLE employee
```

### LOAD DATA INFILE 'fname' IGNORE

The opposite of **REPLACE**, existing rows will be preserved and new ones ignored. This behavior is similar to **LOCAL** described above. However the file need not exist on the client computer.

```
LOAD DATA INFILE 'path of the file/file_name.txt'
IGNORE INTO TABLE employee
```

### Load via intermediary table

Sometimes ignoring or replacing all duplicates may not be the ideal option. You may need to make decisions based on the contents of other columns. In that case the best option is to load into an intermediary table and transfer from there.

```
INSERT INTO employee SELECT * FROM intermediary WHERE ...
```

## Section 52.3: Import a CSV file into a MySQL table

The following command imports CSV files into a MySQL table with the same columns while respecting CSV quoting and escaping rules.

```
load data infile '/tmp/file.csv'
into table my_table
fields terminated by ','
optionally enclosed by '"'
escaped by '"'
lines terminated by '\n'
ignore 1 lines; -- skip the header row
```

# 第53章：MySQL联合查询

## 第53.1节：联合操作符

UNION 运算符用于合并两个或多个 SELECT 语句的结果集（仅包含不同的值）。

查询：（选择“客户”和“供应商”表中所有不同的城市（仅不同值））

```
SELECT City FROM Customers
UNION
SELECT City FROM Suppliers
ORDER BY City;
```

结果：

记录数：10
城市
-----
亚琛
阿尔伯克基
安克雷奇
安纳西
巴塞罗那
巴尔基西梅托
本德
贝加莫
柏林
伯尔尼

## 第53.2节：UNION ALL

使用UNION ALL选择“客户”和“供应商”表中的所有城市（包括重复值）。

查询：

```
SELECT City FROM Customers
UNION ALL
SELECT City FROM Suppliers
ORDER BY City;
```

结果：

记录数：12
城市
-----
亚琛
阿尔伯克基
安克雷奇
安娜堡
安纳西
巴塞罗那
巴尔基西梅托
本德
贝加莫

# Chapter 53: MySQL Unions

## Section 53.1: Union operator

The UNION operator is used to combine the result-set (*only distinct values*) of two or more SELECT statements.

**Query:** (To selects all the different cities (*only distinct values*) from the "Customers" and the "Suppliers" tables)

```
SELECT City FROM Customers
UNION
SELECT City FROM Suppliers
ORDER BY City;
```

Result:

Number of Records: 10
City
-----
Aachen
Albuquerque
Anchorage
Annecy
Barcelona
Barquisimeto
Bend
Bergamo
Berlin
Bern

## Section 53.2: Union ALL

UNION ALL to select all (duplicate values also) cities from the "Customers" and "Suppliers" tables.

Query:

```
SELECT City FROM Customers
UNION ALL
SELECT City FROM Suppliers
ORDER BY City;
```

Result:

Number of Records: 12
City
-----
Aachen
Albuquerque
Anchorage
Ann Arbor
Annecy
Barcelona
Barquisimeto
Bend
Bergamo

柏林  
柏林  
伯尔尼

### 第53.3节：带WHERE的UNION ALL

使用UNION ALL从“Customers”和“Suppliers”表中选择所有（包括重复值）德国城市。这里在where子句中指定Country="Germany"。

查询：

```
SELECT City, Country FROM Customers
WHERE Country='Germany'
UNION ALL
SELECT City, Country FROM Suppliers
WHERE Country='Germany'
ORDER BY City;
```

结果：

记录数：14

城市	国家
亚琛	德国
柏林	德国
柏林	德国
勃兰登堡	德国
库内瓦尔德	德国
库克斯港	德国
法兰克福	德国
法兰克福（美因河畔）	德国
科隆	德国
莱比锡	德国
曼海姆	德国
慕尼黑	德国
明斯特	德国
斯图加特	德国

Berlin  
Berlin  
Bern

### Section 53.3: UNION ALL With WHERE

UNION ALL to select all(duplicate values also) German cities from the "Customers" and "Suppliers" tables. Here Country="Germany" is to be specified in the where clause.

Query:

```
SELECT City, Country FROM Customers
WHERE Country='Germany'
UNION ALL
SELECT City, Country FROM Suppliers
WHERE Country='Germany'
ORDER BY City;
```

Result:

Number of Records: 14

City	Country
Aachen	Germany
Berlin	Germany
Berlin	Germany
Brandenburg	Germany
Cunewalde	Germany
Cuxhaven	Germany
Frankfurt	Germany
Frankfurt a.M.	Germany
Köln	Germany
Leipzig	Germany
Mannheim	Germany
München	Germany
Münster	Germany
Stuttgart	Germany

# 第54章：MySQL客户端

参数	描述
-D --数据库=名称	数据库名称
--分隔符=字符串	设置语句分隔符。默认分隔符为';'
-e --执行='命令'	执行命令
-h --主机=名称	连接的主机名
-p --password=name	password 注意：-p和密码之间没有空格
-p （无密码）	将提示输入密码
-P --port=#	端口号
-s --silent	静默模式，输出更少。使用作为列分隔符
--ss	类似于-s，但省略列名
-S --socket=path	指定连接本地实例时使用的套接字（Unix）或命名管道（Windows）
--跳过-列-名称	省略列名
-u --用户=名称	用户名
-U --安全-更新 --i-am-a-虚拟 使用变量 sql_safe_updates=ON 登录。这将只允许 DELETE 和 明确使用键的 UPDATE	
-V --版本	打印版本并退出

## 第54.1节：基础登录

从命令行访问 MySQL：

```
mysql --user=username --password=pwd --host=hostname test_db
```

这可以简化为：

```
mysql -u 用户名 -p 密码 -h 主机名 test_db
```

省略password值时，MySQL会在首次输入时提示输入所需密码。如果指定了password，客户端会给出“不安全”的警告：

```
mysql -u=用户名 -p -h=主机名 test_db
```

对于本地连接，可以使用--socket来指向套接字文件：

```
mysql --user=用户名 --password=密码 --host=localhost --socket=/路径/到/mysql.sock test_db
```

省略socket参数会导致客户端尝试连接本地机器上的服务器。服务器必须正在运行才能连接。

## 第54.2节：执行命令

这组示例展示了如何执行存储在字符串或脚本文件中的命令，而无需交互式提示。这在shell脚本需要与数据库交互时尤其有用。

从字符串执行命令

```
$ mysql -uroot -proot test -e'select * from people'
```

# Chapter 54: MySQL client

Parameter	Description
-D --database=name	name of the database
--delimiter=str	set the statement delimiter. The default one is ';'
-e --execute='command'	execute command
-h --host=name	hostname to connect to
-p --password=name	password Note: there is no space between -p and the password
-p (without password)	the password will be prompted for
-P --port=#	port number
-s --silent	silent mode, produce less output. Use \t as column separator
--ss	like -s, but omit column names
-S --socket=path	specify the socket (Unix) or named pipe (Windows) to use when connecting to a local instance
--skip-column-names	omit column names
-u --user=name	username
-U --safe-updates --i-am-a-dummy	login with the variable sql_safe_updates=ON. This will allow only DELETE and UPDATE that explicitly use keys
-V --version	print the version and exit

## Section 54.1: Base login

To access MySQL from the command line:

```
mysql --user=username --password=pwd --host=hostname test_db
```

This can be shortened to:

```
mysql -u username -p password -h hostname test_db
```

By omitting the password value MySQL will ask for any required password as the first input. If you specify password the client will give you an 'insecure' warning:

```
mysql -u=username -p -h=hostname test_db
```

For local connections --socket can be used to point to the socket file:

```
mysql --user=username --password=pwd --host=localhost --socket=/path/to/mysql.sock test_db
```

Omitting the socket parameter will cause the client to attempt to attach to a server on the local machine. The server must be running to connect to it.

## Section 54.2: Execute commands

This set of example show how to execute commands stored in strings or script files, without the need of the interactive prompt. This is especially useful to when a shell script needs to interact with a database.

Execute command from a string

```
$ mysql -uroot -proot test -e'select * from people'
```



```
+---+-----+-----+
| id | name  | gender |
+---+-----+-----+
|  1 | Kathy | f      |
|  2 | John  | m      |
+---+-----+-----+
```

要将输出格式化为制表符分隔的表格，请使用--silent参数：

```
$ mysql -uroot -proot test -s -e'select * from people'
```

```
id      name  gender
1       Kathy  f
2       John   m
```

要省略表头：

```
$ mysql -uroot -proot test -ss -e'select * from people'
```

```
1       Kathy  f
2       John   m
```

从脚本文件执行：

```
$ mysql -uroot -proot test < my_script.sql
```

```
$ mysql -uroot -proot test -e'source my_script.sql'
```

将输出写入文件

```
$ mysql -uroot -proot test < my_script.sql > out.txt
```

```
$ mysql -uroot -proot test -s -e'select * from people' > out.txt
```

```
+---+-----+-----+
| id | name  | gender |
+---+-----+-----+
|  1 | Kathy | f      |
|  2 | John  | m      |
+---+-----+-----+
```

To format the output as a tab-separated grid, use the --silent parameter:

```
$ mysql -uroot -proot test -s -e'select * from people'
```

```
id      name  gender
1       Kathy  f
2       John   m
```

To omit the headers:

```
$ mysql -uroot -proot test -ss -e'select * from people'
```

```
1       Kathy  f
2       John   m
```

Execute from script file:

```
$ mysql -uroot -proot test < my_script.sql
```

```
$ mysql -uroot -proot test -e'source my_script.sql'
```

Write the output on a file

```
$ mysql -uroot -proot test < my_script.sql > out.txt
```

```
$ mysql -uroot -proot test -s -e'select * from people' > out.txt
```

# 第55章：临时表

## 第55.1节：创建临时表

临时表对于保存临时数据非常有用。MySQL 3.23及以上版本支持临时表选项。

临时表将在会话结束或连接关闭时自动销毁。用户也可以手动删除临时表。

同名临时表可以在多个连接中同时使用，因为临时表仅对创建该表的客户端可用和可访问。

临时表可以通过以下类型创建

```
--->基本临时表创建
CREATE TEMPORARY TABLE tempTable1(
    id INT NOT NULL AUTO_INCREMENT,
    title VARCHAR(100) NOT NULL,
    PRIMARY KEY ( id )
);

--->通过select查询创建临时表
CREATE TEMPORARY TABLE tempTable1
SELECT ColumnName1,ColumnName2,... FROM table1;
```

您可以在创建表的同时添加索引：

```
CREATE TEMPORARY TABLE tempTable1
( PRIMARY KEY(ColumnName2) )
SELECT ColumnName1,ColumnName2,... FROM table1;
```

可以使用 IF NOT EXISTS 关键字，如下所示，以避免出现 '表已存在' 错误。但在这种情况下，如果您使用的表名在当前会话中已存在，则不会创建表。

```
CREATE TEMPORARY TABLE IF NOT EXISTS tempTable1
SELECT ColumnName1,ColumnName2,... FROM table1;
```

## 第55.2节：删除临时表

删除临时表用于删除您在当前会话中创建的临时表。

```
DROP TEMPORARY TABLE tempTable1

DROP TEMPORARY TABLE IF EXISTS tempTable1
```

使用 IF EXISTS 可以防止因表可能不存在而导致的错误

# Chapter 55: Temporary Tables

## Section 55.1: Create Temporary Table

Temporary tables could be very useful to keep temporary data. Temporary tables option is available in MySQL version 3.23 and above.

Temporary table will be automatically destroyed when the session ends or connection is closed. The user can also drop temporary table.

Same temporary table name can be used in many connections at the same time, because the temporary table is only available and accessible by the client who creates that table.

The temporary table can be created in the following types

```
--->Basic temporary table creation
CREATE TEMPORARY TABLE tempTable1(
    id INT NOT NULL AUTO_INCREMENT,
    title VARCHAR(100) NOT NULL,
    PRIMARY KEY ( id )
);

--->Temporary table creation from select query
CREATE TEMPORARY TABLE tempTable1
SELECT ColumnName1,ColumnName2,... FROM table1;
```

You can add indexes as you build the table:

```
CREATE TEMPORARY TABLE tempTable1
( PRIMARY KEY(ColumnName2) )
SELECT ColumnName1,ColumnName2,... FROM table1;
```

IF NOT EXISTS key word can be used as mentioned below to avoid 'table already exists' error. But in that case table will not be created, if the table name which you are using already exists in your current session.

```
CREATE TEMPORARY TABLE IF NOT EXISTS tempTable1
SELECT ColumnName1,ColumnName2,... FROM table1;
```

## Section 55.2: Drop Temporary Table

Drop Temporary Table is used to delete the temporary table which you are created in your current session.

```
DROP TEMPORARY TABLE tempTable1

DROP TEMPORARY TABLE IF EXISTS tempTable1
```

Use IF EXISTS to prevent an error occurring for tables that may not exist

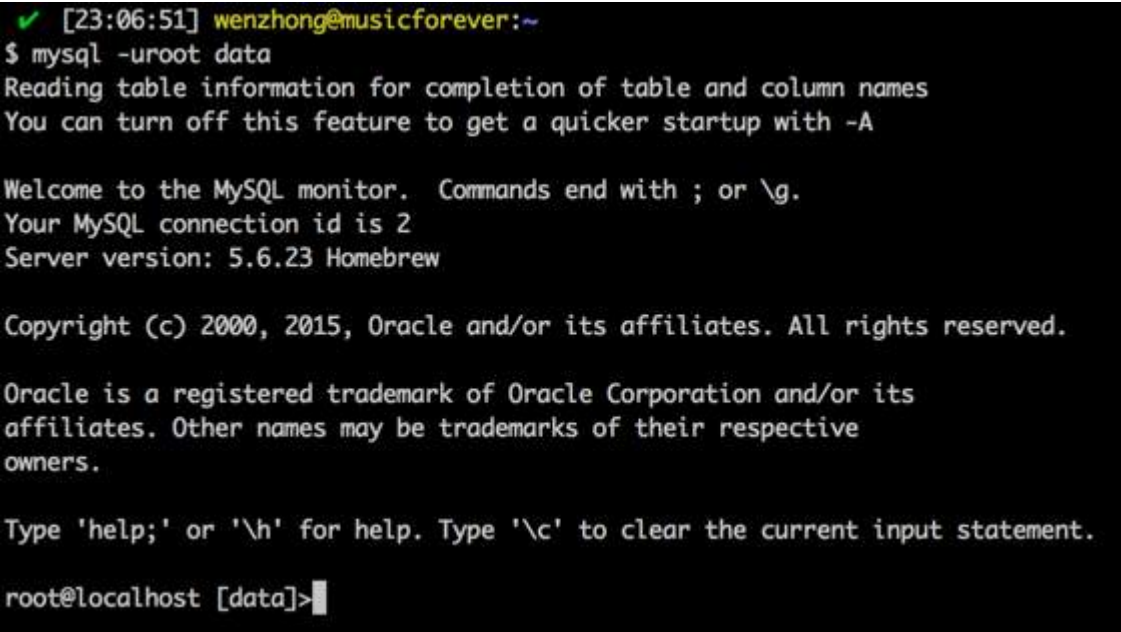
# 第56章：自定义 PS1

## 第56.1节：使用当前数据库自定义MySQL PS1

在 .bashrc 或 .bash\_profile 中添加：

```
export MYSQL_PS1="\u@\h [\d]>"
```

使 MySQL 客户端提示符显示当前的 user@host [database]。



## 第 56.2 节：通过 MySQL 配置文件自定义 PS1

在 mysqld.cnf 或同等文件中：

```
[mysql]
prompt = '\u@\h [\d]> '
```

这样可以达到类似效果，无需处理 .bashrc 文件。

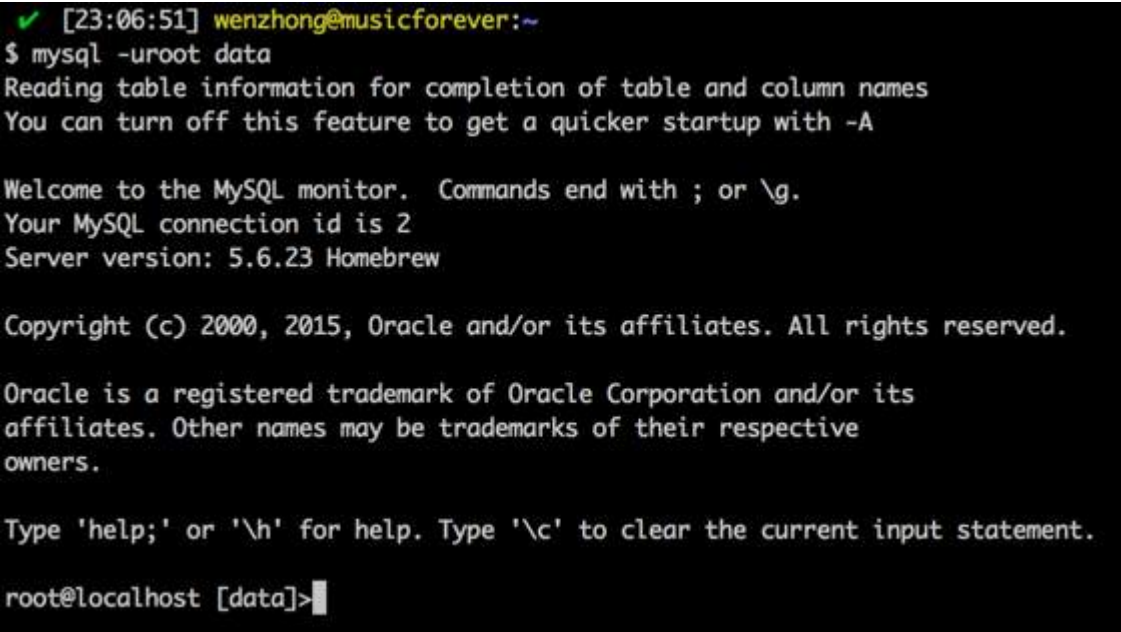
# Chapter 56: Customize PS1

## Section 56.1: Customize the MySQL PS1 with current database

In the .bashrc or .bash\_profile, adding:

```
export MYSQL_PS1="\u@\h [\d]>"
```

make the MySQL client PROMPT show current user@host [database].



## Section 56.2: Custom PS1 via MySQL configuration file

In mysqld.cnf or equivalent:

```
[mysql]
prompt = '\u@\h [\d]> '
```

This achieves a similar effect, without having to deal with .bashrc's.

# 第 57 章：处理稀疏或缺失的数据

## 第 57.1 节：处理包含 NULL 值的列

在 MySQL 及其他 SQL 方言中，NULL 值具有特殊属性。

考虑以下包含求职者、他们曾工作的公司以及离职日期的表。NULL 表示求职者仍在该公司工作：

```
CREATE TABLE example
('applicant_id' INT, 'company_name' VARCHAR(255), 'end_date' DATE);
```

applicant_id	company_name	end_date
1	Google	NULL
1	Initech	2013-01-31
2	Woodworking.com	2016-08-25
2	NY Times	2013-11-10
3	NFL.com	2014-04-13

你的任务是编写一个查询，返回所有 2016-01-01 之后的行，包括仍在公司工作的员工（即 end\_date 为NULL的员工）。以下的 select 语句：

```
SELECT * FROM example WHERE end_date > '2016-01-01';
```

未能包含任何带有NULL值的行：

applicant_id	company_name	end_date
2	Woodworking.com	2016-08-25

根据MySQL 文档，使用算术运算符 <、>、= 和 <> 进行比较时，结果本身返回NULL，而不是布尔值TRUE或FALSE。因此，end\_date 为NULL的行既不大于 2016-01-01，也不小于 2016-01-01。

这可以通过使用关键字 IS NULL 来解决：

```
SELECT * FROM example WHERE end_date > '2016-01-01' OR end_date IS NULL;
```

applicant_id	company_name	end_date
1	Google	NULL
2	Woodworking.com	2016-08-25

处理 NULL 变得更加复杂，当任务涉及聚合函数如 MAX() 和 GROUP

# Chapter 57: Dealing with sparse or missing data

## Section 57.1: Working with columns containg NULL values

In MySQL and other SQL dialects, NULL values have special properties.

Consider the following table containing job applicants, the companies they worked for, and the date they left the company. NULL indicates that an applicant still works at the company:

```
CREATE TABLE example
('applicant_id' INT, 'company_name' VARCHAR(255), 'end_date' DATE);
```

applicant_id	company_name	end_date
1	Google	NULL
1	Initech	2013-01-31
2	Woodworking.com	2016-08-25
2	NY Times	2013-11-10
3	NFL.com	2014-04-13

Your task is to compose a query that returns all rows after 2016-01-01, including any employees that are still working at a company (those with NULL end dates). This select statement:

```
SELECT * FROM example WHERE end_date > '2016-01-01';
```

fails to include any rows with NULL values:

applicant_id	company_name	end_date
2	Woodworking.com	2016-08-25

Per the MySQL documentation, comparisons using the arithmetic operators <, >, =, and <> themselves return NULL instead of a boolean TRUE or FALSE. Thus a row with a NULL end\_date is neither greater than 2016-01-01 nor less than 2016-01-01.

This can be solved by using the keywords IS NULL:

```
SELECT * FROM example WHERE end_date > '2016-01-01' OR end_date IS NULL;
```

applicant_id	company_name	end_date
1	Google	NULL
2	Woodworking.com	2016-08-25

Working with NULLs becomes more complex when the task involves aggregation functions like MAX() and a GROUP

**BY** 子句时。如果你的任务是为每个 applicant\_id 选择最近的雇佣日期，以下查询看起来是一个合乎逻辑的初步尝试：

```
SELECT applicant_id, MAX(end_date) FROM example GROUP BY applicant_id;
```

applicant_id	MAX(end_date)
1	2013-01-31
2	2016-08-25
3	2014-04-13

然而，知道 **NULL** 表示申请人仍在公司工作，结果的第一行是不准确的。使用 **CASE WHEN** 提供了一个解决 **NULL** 问题的变通方法：

```
查询
applicant_id,
CASE WHEN MAX(end_date is null) = 1 THEN 'present' ELSE MAX(end_date) END
max_date
FROM example
GROUP BY applicant_id;
```

applicant_id	max_date
1	present
2	2016-08-25
3	2014-04-13

该结果可以与原始的example表连接，以确定申请人最后工作的公司：

```
查询
data.applicant_id,
data.company_name,
data.max_date
FROM (
  查询
  *,
  CASE WHEN end_date is null THEN 'present' ELSE end_date END max_date
  FROM example
) data
INNER JOIN (
  SELECT
applicant_id,
CASE WHEN MAX(end_date is null) = 1 THEN 'present' ELSE MAX(end_date) END max_date
FROM
例子
按 applicant_id 分组
) j
在 data.applicant_id = j.applicant_id 且 data.max_date = j.max_date;
```

applicant_id	公司名称	最大日期
1		
2		
3		

**BY** clause. If your task were to select the most recent employed date for each applicant\_id, the following query would seem a logical first attempt:

```
SELECT applicant_id, MAX(end_date) FROM example GROUP BY applicant_id;
```

applicant_id	MAX(end_date)
1	2013-01-31
2	2016-08-25
3	2014-04-13

However, knowing that **NULL** indicates an applicant is still employed at a company, the first row of the result is inaccurate. Using **CASE WHEN** provides a workaround for the **NULL** issue:

```
SELECT
  applicant_id,
  CASE WHEN MAX(end_date is null) = 1 THEN 'present' ELSE MAX(end_date) END
  max_date
FROM example
GROUP BY applicant_id;
```

applicant_id	max_date
1	present
2	2016-08-25
3	2014-04-13

This result can be joined back to the original example table to determine the company at which an applicant last worked:

```
SELECT
  data.applicant_id,
  data.company_name,
  data.max_date
FROM (
  SELECT
    *,
    CASE WHEN end_date is null THEN 'present' ELSE end_date END max_date
  FROM example
) data
INNER JOIN (
  SELECT
    applicant_id,
    CASE WHEN MAX(end_date is null) = 1 THEN 'present' ELSE MAX(end_date) END max_date
  FROM
    example
  GROUP BY applicant_id
) j
ON data.applicant_id = j.applicant_id AND data.max_date = j.max_date;
```

applicant_id	company_name	max_date
1		
2		
3		



1	谷歌	现在
2	Woodworking.com	2016-08-25
3	NFL.com	2014-04-13

这些只是使用 MySQL 处理NULL值的几个示例。

1	Google	present
2	Woodworking.com	2016-08-25
3	NFL.com	2014-04-13

These are just a few examples of working with **NULL** values in MySQL.

# 第58章：使用各种编程语言连接UTF-8。

## 第58.1节：Python

源代码中的第1行或第2行（使代码中的字面量为utf8编码）：

```
# -*- coding: utf-8 -*-
```

连接：

```
db = MySQLdb.connect(主机=DB_HOST, 用户=DB_USER, 密码=DB_PASS, 数据库=DB_NAME,
                      字符集="utf8mb4", 使用_unicode=True)
```

对于网页，以下任意一个：

```
<meta charset="utf-8" />
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
```

## 第58.2节：PHP

在php.ini中（PHP 5.6之后的默认设置）：

```
default_charset UTF-8
```

构建网页时：

```
header('Content-type: text/plain; charset=UTF-8');
```

连接到 MySQL 时：

```
(针对 mysql:) 不要使用 mysql_* API !
(针对 mysqli:) $mysqli_obj->set_charset('utf8mb4');
(针对 PDO:) $db = new PDO('dblib:host=host;dbname=db;charset=utf8', $user, $pwd);
```

在代码中，不要使用任何转换例程。

对于数据输入，

```
<form accept-charset="UTF-8">
```

对于 JSON，为避免 \uxxxx：

```
$t = json_encode($s, JSON_UNESCAPED_UNICODE);
```

# Chapter 58: Connecting with UTF-8 Using Various Programming language.

## Section 58.1: Python

1st or 2nd line in source code (to have literals in the code utf8-encoded):

```
# -*- coding: utf-8 -*-
```

Connection:

```
db = MySQLdb.connect(host=DB_HOST, user=DB_USER, passwd=DB_PASS, db=DB_NAME,
                      charset="utf8mb4", use_unicode=True)
```

For web pages, one of these:

```
<meta charset="utf-8" />
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
```

## Section 58.2: PHP

In php.ini (this is the default after PHP 5.6):

```
default_charset UTF-8
```

When building a web page:

```
header('Content-type: text/plain; charset=UTF-8');
```

When connecting to MySQL:

```
(for mysql:) Do not use the mysql_* API!
(for mysqli:) $mysqli_obj->set_charset('utf8mb4');
(for PDO:) $db = new PDO('dblib:host=host;dbname=db;charset=utf8', $user, $pwd);
```

In code, do not use any conversion routines.

For data entry,

```
<form accept-charset="UTF-8">
```

For JSON, to avoid \uxxxx:

```
$t = json_encode($s, JSON_UNESCAPED_UNICODE);
```

# 第59章：具有亚秒精度的时间

## 第59.1节：获取毫秒精度的当前时间

```
SELECT NOW(3)
```

这就行得通。

## 第59.2节：以类似Javascript时间戳的形式获取当前时间

Javascript时间戳基于久经考验的UNIX time\_t 数据类型，显示自1970-01-01 00:00:00 UTC以来的毫秒数。

该表达式以Javascript时间戳整数的形式获取当前时间。（无论当前的时区设置如何，都能正确获取。）

```
ROUND(UNIX_TIMESTAMP(NOW(3)) * 1000.0, 0)
```

如果你在某列中存储了TIMESTAMP值，可以使用

UNIX\_TIMESTAMP()函数将它们作为整数Javascript时间戳检索。

```
SELECT ROUND(UNIX_TIMESTAMP(column) * 1000.0, 0)
```

如果你的列包含DATETIME列，并且你将它们作为Javascript时间戳检索，那么这些时间戳将被存储时所在时区的时区偏移量所影响。

## 第59.3节：创建一个用于存储亚秒时间的列的表

```
CREATE TABLE times (
  dt DATETIME(3),
  ts TIMESTAMP(3)
);
```

创建一个带有毫秒精度日期/时间字段的表。

```
INSERT INTO times VALUES (NOW(3), NOW(3));
```

向表中插入包含毫秒精度的NOW()值的行。

```
INSERT INTO times VALUES ('2015-01-01 16:34:00.123','2015-01-01 16:34:00.128');
```

插入特定的毫秒精度值。

注意如果使用该函数插入高精度时间值，必须使用NOW(3)而不是NOW()。

## 第59.4节：将毫秒精度的日期/时间值转换为文本

%f是DATE\_FORMAT()函数的分数精度格式说明符。\_\_\_\_\_

# Chapter 59: Time with subsecond precision

## Section 59.1: Get the current time with millisecond precision

```
SELECT NOW(3)
```

does the trick.

## Section 59.2: Get the current time in a form that looks like a Javascript timestamp

Javascript timestamps are based on the venerable UNIX time\_t data type, and show the number of milliseconds since 1970-01-01 00:00:00 UTC.

This expression gets the current time as a Javascript timestamp integer. (It does so correctly regardless of the current time\_zone setting.)

```
ROUND(UNIX_TIMESTAMP(NOW(3)) * 1000.0, 0)
```

If you have TIMESTAMP values stored in a column, you can retrieve them as integer Javascript timestamps using the UNIX\_TIMESTAMP() function.

```
SELECT ROUND(UNIX_TIMESTAMP(column) * 1000.0, 0)
```

If your column contains DATETIME columns and you retrieve them as Javascript timestamps, those timestamps will be offset by the time zone offset of the time zone they're stored in.

## Section 59.3: Create a table with columns to store sub-second time

```
CREATE TABLE times (
  dt DATETIME(3),
  ts TIMESTAMP(3)
);
```

makes a table with millisecond-precision date / time fields.

```
INSERT INTO times VALUES (NOW(3), NOW(3));
```

inserts a row containing NOW() values with millisecond precision into the table.

```
INSERT INTO times VALUES ('2015-01-01 16:34:00.123', '2015-01-01 16:34:00.128');
```

inserts specific millisecond precision values.

**Notice** that you must use NOW(3) rather than NOW() if you use that function to insert high-precision time values.

## Section 59.4: Convert a millisecond-precision date / time value to text

%f is the fractional precision format specifier for [the DATE\\_FORMAT\(\) function](#).

```
SELECT DATE_FORMAT(NOW(3), '%Y-%m-%d %H:%i:%s.%f')
```

显示类似2016-11-19 09:52:53.248000的带有分数微秒的值。因为我们使用了NOW(3)，分数部分的最后三位数字是0。

## 第59.5节：将Javascript时间戳存储到TIMESTAMP列中

如果你有一个Javascript时间戳值，例如1478960868932，你可以像这样将其转换为MySQL的分数时间值：

```
FROM_UNIXTIME(1478960868932 * 0.001)
```

使用这种表达式将你的Javascript时间戳存储到MySQL表中非常简单。操作如下：

```
INSERT INTO table (col) VALUES (FROM_UNIXTIME(1478960868932 * 0.001))
```

（显然，你还需要插入其他列。）

```
SELECT DATE_FORMAT(NOW(3), '%Y-%m-%d %H:%i:%s.%f')
```

displays a value like 2016-11-19 09:52:53.248000 with fractional microseconds. Because we used NOW(3), the final three digits in the fraction are 0.

## Section 59.5: Store a Javascript timestamp into a TIMESTAMP column

If you have a Javascript timestamp value, for example 1478960868932, you can convert that to a MySQL fractional time value like this:

```
FROM_UNIXTIME(1478960868932 * 0.001)
```

It's simple to use that kind of expression to store your Javascript timestamp into a MySQL table. Do this:

```
INSERT INTO table (col) VALUES (FROM_UNIXTIME(1478960868932 * 0.001))
```

(Obviously, you'll want to insert other columns.)

# 第60章：一对多

一对多（1:M）的概念涉及行与行之间的连接，特别是指一个表中的单行对应另一个表中的多行的情况。

1:M是单向的，也就是说，每当你查询1:M关系时，可以使用“一个”行来选择另一个表中的“多个”行，但不能使用单个“多个”行来选择超过一个“一个”行。

## 第60.1节：示例公司表

考虑这样一家公司：每位经理都管理着一名或多名员工，而每位员工只有一名经理。

这导致了两个表：

员工			
员工编号	名字	姓氏	经理编号
E01	约翰尼	苹果树	M02
E02	艾琳	麦克尔莫	M01
E03	科尔比	Paperwork	M03
E04	罗恩	桑斯万	M01

经理们		
MGR_ID	名字	姓氏
M01	响亮	麦昆
M02	专横	裤子
M03	桶	琼斯

## 第60.2节：获取由单一经理管理的员工

```
SELECT e.emp_id , e.first_name , e.last_name FROM employees e INNER JOIN managers m ON m.mgr_id = e.mgr_id WHERE m.mgr_id = 'M01' ;
```

结果为：

EMP_ID	名字	姓氏
E02	艾琳	麦克尔莫尔
E04	罗恩	桑斯万

最终，对于我们查询的每个经理，都会返回一个或多个员工。

## 第60.3节：获取单个员工的经理

查看此示例时，请参考上述示例表。

```
SELECT m.mgr_id , m.first_name , m.last_name FROM managers m INNER JOIN employees e ON e.mgr_id = m.mgr_id WHERE e.emp_id = 'E03' ;
```

MGR_ID	名字	姓氏
M03	桶	琼斯

# Chapter 60: One to Many

The idea of one to many (1:M) concerns the joining of rows to each other, specifically cases where a single row in one table corresponds to many rows in another.

1:M is one-directional, that is, any time you query a 1:M relationship, you can use the 'one' row to select 'many' rows in another table, but you cannot use a single 'many' row to select more than a single 'one' row.

## Section 60.1: Example Company Tables

Consider a company where every employee who is a manager, manages 1 or more employees, and every employee has only 1 manager.

This results in two tables:

EMPLOYEES			
EMP_ID	FIRST_NAME	LAST_NAME	MGR_ID
E01	Johnny	Appleseed	M02
E02	Erin	Macklemore	M01
E03	Colby	Paperwork	M03
E04	Ron	Sonswan	M01

MANAGERS		
MGR_ID	FIRST_NAME	LAST_NAME
M01	Loud	McQueen
M02	Bossy	Pants
M03	Barrel	Jones

## Section 60.2: Get the Employees Managed by a Single Manager

```
SELECT e.emp_id , e.first_name , e.last_name FROM employees e INNER JOIN managers m ON m.mgr_id = e.mgr_id WHERE m.mgr_id = 'M01' ;
```

Results in:

EMP_ID	FIRST_NAME	LAST_NAME
E02	Erin	Macklemore
E04	Ron	Sonswan

Ultimately, for every manager we query for, we will see 1 or more employees returned.

## Section 60.3: Get the Manager for a Single Employee

Consult the above example tables when looking at this example.

```
SELECT m.mgr_id , m.first_name , m.last_name FROM managers m INNER JOIN employees e ON e.mgr_id = m.mgr_id WHERE e.emp_id = 'E03' ;
```

MGR_ID	FIRST_NAME	LAST_NAME
M03	Barrel	Jones



由于这是上述示例的逆向操作，我们知道对于查询的每个员工，我们只会看到一个对应的经理。

As this is the inverse of the above example, we know that for every employee we query for, we will only ever see one corresponding manager.

# 第61章：服务器信息

参数	说明
GLOBAL	显示整个服务器配置的变量。可选。
SESSION	显示仅为本会话配置的变量。可选。

## 第61.1节：SHOW VARIABLES 示例

要获取所有服务器变量，请在您首选的界面（PHPMyAdmin或其他）的SQL窗口中，或在MySQL命令行界面中运行此查询

```
SHOW VARIABLES;
```

您可以指定是要会话变量还是全局变量，方法如下：

会话变量：

```
SHOW SESSION VARIABLES;
```

全局变量：

```
SHOW GLOBAL VARIABLES;
```

像其他SQL命令一样，您可以为查询添加参数，例如LIKE命令：

```
SHOW [GLOBAL | SESSION] VARIABLES LIKE 'max_join_size';
```

或者，使用通配符：

```
SHOW [GLOBAL | SESSION] VARIABLES LIKE '%size%';
```

您也可以使用 WHERE 参数过滤 SHOW 查询的结果，方法如下：

```
SHOW [GLOBAL | SESSION] VARIABLES WHERE VALUE > 0;
```

## 第61.2节：SHOW STATUS 示例

要获取数据库服务器状态，请在您喜欢的界面（PHPMyAdmin 或其他）的 SQL 窗口中，或在 MySQL 命令行界面运行此查询。

```
SHOW STATUS;
```

您可以指定希望接收服务器的 SESSION 状态或 GLOBAL 状态，如下所示：会话状态：

```
SHOW SESSION STATUS;
```

全局状态：

```
SHOW GLOBAL STATUS;
```

像其他SQL命令一样，您可以为查询添加参数，例如LIKE命令：

# Chapter 61: Server Information

Parameters	Explanation
GLOBAL	Shows the variables as they are configured for the entire server. Optional.
SESSION	Shows the variables that are configured for this session only. Optional.

## Section 61.1: SHOW VARIABLES example

To get all the server variables run this query either in the SQL window of your preferred interface (PHPMyAdmin or other) or in the MySQL CLI interface

```
SHOW VARIABLES;
```

You can specify if you want the session variables or the global variables as follows:

Session variables:

```
SHOW SESSION VARIABLES;
```

Global variables:

```
SHOW GLOBAL VARIABLES;
```

Like any other SQL command you can add parameters to your query such as the LIKE command:

```
SHOW [GLOBAL | SESSION] VARIABLES LIKE 'max_join_size';
```

Or, using wildcards:

```
SHOW [GLOBAL | SESSION] VARIABLES LIKE '%size%';
```

You can also filter the results of the SHOW query using a WHERE parameter as follows:

```
SHOW [GLOBAL | SESSION] VARIABLES WHERE VALUE > 0;
```

## Section 61.2: SHOW STATUS example

To get the database server status run this query in either the SQL window of your preferred interface (PHPMyAdmin or other) or on the MySQL CLI interface.

```
SHOW STATUS;
```

You can specify whether you wish to receive the SESSION or GLOBAL status of your sever like so: Session status:

```
SHOW SESSION STATUS;
```

Global status:

```
SHOW GLOBAL STATUS;
```

Like any other SQL command you can add parameters to your query such as the LIKE command:

```
SHOW [GLOBAL | SESSION] STATUS LIKE 'Key%';
```

或者使用 Where 命令：

```
SHOW [GLOBAL | SESSION] STATUS WHERE VALUE > 0;
```

GLOBAL 和 SESSION 之间的主要区别在于，使用 GLOBAL 修饰符时，命令显示的是服务器及其所有连接的汇总信息，而 SESSION 修饰符只显示当前连接的值。

```
SHOW [GLOBAL | SESSION] STATUS LIKE 'Key%' ;
```

Or the Where command:

```
SHOW [GLOBAL | SESSION] STATUS WHERE VALUE > 0;
```

The main difference between GLOBAL and SESSION is that with the GLOBAL modifier the command displays aggregated information about the server and all of it's connections, while the SESSION modifier will only show the values for the current connection.

# 第62章：SSL连接设置

## 第62.1节：基于Debian系统的设置

（假设已安装MySQL并且正在使用 `sudo`。）

### 生成CA和SSL密钥

确保已安装OpenSSL及其库：

```
apt-get -y install openssl
apt-get -y install libssl-dev
```

接下来创建并进入用于存放SSL文件的目录：

```
mkdir /home/ubuntu/mysqlcerts
cd /home/ubuntu/mysqlcerts
```

要生成密钥，先创建一个证书颁发机构（CA）来签署密钥（自签名）：

```
openssl genrsa 2048 > ca-key.pem
openssl req -new -x509 -nodes -days 3600 -key ca-key.pem -out ca.pem
```

每个提示输入的值不会影响配置。接下来为服务器创建密钥，并使用之前的CA进行签名：

```
openssl req -newkey rsa:2048 -days 3600 -nodes -keyout server-key.pem -out server-req.pem
openssl rsa -in server-key.pem -out server-key.pem

openssl x509 -req -in server-req.pem -days 3600 -CA ca.pem -CAkey ca-key.pem -set_serial 01 -out server-cert.pem
```

然后为客户端创建密钥：

```
openssl req -newkey rsa:2048 -days 3600 -nodes -keyout client-key.pem -out client-req.pem
openssl rsa -in client-key.pem -out client-key.pem
openssl x509 -req -in client-req.pem -days 3600 -CA ca.pem -CAkey ca-key.pem -set_serial 01 -out client-cert.pem
```

为了确保一切设置正确，请验证密钥：

```
openssl verify -CAfile ca.pem server-cert.pem client-cert.pem
```

### 将密钥添加到 MySQL

打开 MySQL 配置文件。例如：

```
vim /etc/mysql/mysql.conf.d/mysqld.cnf
```

在 `[mysqld]` 部分，添加以下选项：

```
ssl-ca = /home/ubuntu/mysqlcerts/ca.pem
ssl-cert = /home/ubuntu/mysqlcerts/server-cert.pem
ssl-key = /home/ubuntu/mysqlcerts/server-key.pem
```

# Chapter 62: SSL Connection Setup

## Section 62.1: Setup for Debian-based systems

(This assumes MySQL has been installed and that `sudo` is being used.)

### Generating a CA and SSL keys

Make sure OpenSSL and libraries are installed:

```
apt-get -y install openssl
apt-get -y install libssl-dev
```

Next make and enter a directory for the SSL files:

```
mkdir /home/ubuntu/mysqlcerts
cd /home/ubuntu/mysqlcerts
```

To generate keys, create a certificate authority (CA) to sign the keys (self-signed):

```
openssl genrsa 2048 > ca-key.pem
openssl req -new -x509 -nodes -days 3600 -key ca-key.pem -out ca.pem
```

The values entered at each prompt won't affect the configuration. Next create a key for the server, and sign using the CA from before:

```
openssl req -newkey rsa:2048 -days 3600 -nodes -keyout server-key.pem -out server-req.pem
openssl rsa -in server-key.pem -out server-key.pem

openssl x509 -req -in server-req.pem -days 3600 -CA ca.pem -CAkey ca-key.pem -set_serial 01 -out server-cert.pem
```

Then create a key for a client:

```
openssl req -newkey rsa:2048 -days 3600 -nodes -keyout client-key.pem -out client-req.pem
openssl rsa -in client-key.pem -out client-key.pem
openssl x509 -req -in client-req.pem -days 3600 -CA ca.pem -CAkey ca-key.pem -set_serial 01 -out client-cert.pem
```

To make sure everything was set up correctly, verify the keys:

```
openssl verify -CAfile ca.pem server-cert.pem client-cert.pem
```

### Adding the keys to MySQL

Open the MySQL configuration file. For example:

```
vim /etc/mysql/mysql.conf.d/mysqld.cnf
```

Under the `[mysqld]` section, add the following options:

```
ssl-ca = /home/ubuntu/mysqlcerts/ca.pem
ssl-cert = /home/ubuntu/mysqlcerts/server-cert.pem
ssl-key = /home/ubuntu/mysqlcerts/server-key.pem
```

重启 MySQL。例如：

```
service mysql restart
```

测试 SSL 连接

以相同方式连接，传入额外选项 `ssl-ca`、`ssl-cert` 和 `ssl-key`，使用生成的客户端密钥。例如，假设 `cd /home/ubuntu/mysqlcerts`：

```
mysql --ssl-ca=ca.pem --ssl-cert=client-cert.pem --ssl-key=client-key.pem -h 127.0.0.1 -u superman -p
```

登录后，验证连接确实是安全的：

```
superman@127.0.0.1 [None]> SHOW VARIABLES LIKE '%ssl%';
+-----+-----+
| Variable_name | Value                               |
+-----+-----+
| have_openssl  | YES                                |
| have_ssl      | YES                                |
| ssl_ca        | /home/ubuntu/mysqlcerts/ca.pem     |
| ssl_capath    |                                     |
| ssl_cert      | /home/ubuntu/mysqlcerts/server-cert.pem |
| ssl_cipher    |                                     |
| ssl_crl       |                                     |
| ssl_crlpath   |                                     |
| ssl_key       | /home/ubuntu/mysqlcerts/server-key.pem |
+-----+-----+
```

你也可以检查：

```
superman@127.0.0.1 [None]> STATUS;
...
SSL:                               Cipher in use is DHE-RSA-AES256-SHA
...
```

强制使用 SSL

这是通过GRANT，使用REQUIRE SSL实现的：

```
授予 'superman'@'127.0.0.1' 对所有数据库和表的所有权限，密码为 'pass'，要求使用 SSL；
刷新权限；
```

现在，superman 必须通过 SSL 连接。

如果你不想管理客户端密钥，可以使用之前的客户端密钥，并自动为所有客户端使用该密钥。打开 MySQL 配置文件，例如：

```
vim /etc/mysql/mysql.conf.d/mysqld.cnf
```

在 [client] 部分，添加以下选项：

```
ssl-ca = /home/ubuntu/mysqlcerts/ca.pem
ssl-cert = /home/ubuntu/mysqlcerts/client-cert.pem
ssl-key = /home/ubuntu/mysqlcerts/client-key.pem
```

现在 superman 只需输入以下命令即可通过 SSL 登录：

Restart MySQL. For example:

```
service mysql restart
```

Test the SSL connection

Connect in the same way, passing in the extra options `ssl-ca`, `ssl-cert`, and `ssl-key`, using the generated client key. For example, assuming `cd /home/ubuntu/mysqlcerts`:

```
mysql --ssl-ca=ca.pem --ssl-cert=client-cert.pem --ssl-key=client-key.pem -h 127.0.0.1 -u superman -p
```

After logging in, verify the connection is indeed secure:

```
superman@127.0.0.1 [None]> SHOW VARIABLES LIKE '%ssl%';
+-----+-----+
| Variable_name | Value                               |
+-----+-----+
| have_openssl  | YES                                |
| have_ssl      | YES                                |
| ssl_ca        | /home/ubuntu/mysqlcerts/ca.pem     |
| ssl_capath    |                                     |
| ssl_cert      | /home/ubuntu/mysqlcerts/server-cert.pem |
| ssl_cipher    |                                     |
| ssl_crl       |                                     |
| ssl_crlpath   |                                     |
| ssl_key       | /home/ubuntu/mysqlcerts/server-key.pem |
+-----+-----+
```

You could also check:

```
superman@127.0.0.1 [None]> STATUS;
...
SSL:                               Cipher in use is DHE-RSA-AES256-SHA
...
```

Enforcing SSL

This is via **GRANT**, using **REQUIRE SSL**:

```
GRANT ALL PRIVILEGES ON *.* TO 'superman'@'127.0.0.1' IDENTIFIED BY 'pass' REQUIRE SSL;
FLUSH PRIVILEGES;
```

Now, superman *must* connect via SSL.

If you don't want to manage client keys, use the client key from earlier and automatically use that for all clients. Open MySQL configuration file, for example:

```
vim /etc/mysql/mysql.conf.d/mysqld.cnf
```

Under the [client] section, add the following options:

```
ssl-ca = /home/ubuntu/mysqlcerts/ca.pem
ssl-cert = /home/ubuntu/mysqlcerts/client-cert.pem
ssl-key = /home/ubuntu/mysqlcerts/client-key.pem
```

Now superman only has to type the following to login via SSL:



```
mysql -h 127.0.0.1 -u superman -p
```

从其他程序连接，例如在 Python 中，通常只需在 connect 函数中添加一个额外参数。Python 示例：

```
import MySQLdb
ssl = {'cert': '/home/ubuntu/mysqlcerts/client-cert.pem', 'key': '/home/ubuntu/mysqlcerts/client-key.pem'}
conn = MySQLdb.connect(host='127.0.0.1', user='superman', passwd='imsoawesome', ssl=ssl)
```

参考文献及进一步阅读：

- <https://www.percona.com/blog/2013/06/22/setting-up-mysql-ssl-and-secure-connections/>
- <https://lowendbox.com/blog/getting-started-with-mysql-over-ssl/>
- <http://xmodulo.com/enable-ssl-mysql-server-client.html>
- <https://ubuntuforums.org/showthread.php?t=1121458>

## 第62.2节：CentOS7 / RHEL7的设置

本示例假设有两台服务器：

- 1.dbserver（数据库所在服务器）
- 2.appclient（应用程序所在服务器）

仅供参考，两台服务器均启用了SELinux强制模式。

首先，登录到数据库服务器

创建一个临时目录用于生成证书。

```
mkdir /root/certs/mysql/ && cd /root/certs/mysql/
```

创建服务器证书

```
openssl genrsa 2048 > ca-key.pem
openssl req -sha1 -new -x509 -nodes -days 3650 -key ca-key.pem > ca-cert.pem
openssl req -sha1 -newkey rsa:2048 -days 730 -nodes -keyout server-key.pem > server-req.pem
openssl rsa -in server-key.pem -out server-key.pem
openssl x509 -sha1 -req -in server-req.pem -days 730 -CA ca-cert.pem -CAkey ca-key.pem -set_serial 01 > server-cert.pem
```

将服务器证书移动到 /etc/pki/tls/certs/mysql/

目录路径假设为 CentOS 或 RHEL（其他发行版请根据需要调整）：

```
mkdir /etc/pki/tls/certs/mysql/
```

务必设置文件夹和文件的权限。mysql 需要完全的所有权和访问权限。

```
chown -R mysql:mysql /etc/pki/tls/certs/mysql
```

现在配置 MySQL/MariaDB

```
# vi /etc/my.cnf
# i
```

```
mysql -h 127.0.0.1 -u superman -p
```

Connecting from another program, for example in Python, typically only requires an additional parameter to the connect function. A Python example:

```
import MySQLdb
ssl = {'cert': '/home/ubuntu/mysqlcerts/client-cert.pem', 'key': '/home/ubuntu/mysqlcerts/client-key.pem'}
conn = MySQLdb.connect(host='127.0.0.1', user='superman', passwd='imsoawesome', ssl=ssl)
```

References and further reading:

- <https://www.percona.com/blog/2013/06/22/setting-up-mysql-ssl-and-secure-connections/>
- <https://lowendbox.com/blog/getting-started-with-mysql-over-ssl/>
- <http://xmodulo.com/enable-ssl-mysql-server-client.html>
- <https://ubuntuforums.org/showthread.php?t=1121458>

## Section 62.2: Setup for CentOS7 / RHEL7

This example assumes two servers:

- 1. dbserver (where our database lives)
- 2. appclient (where our applications live)

FWIW, both servers are SELinux enforcing.

First, log on to dbserver

Create a temporary directory for creating the certificates.

```
mkdir /root/certs/mysql/ && cd /root/certs/mysql/
```

Create the server certificates

```
openssl genrsa 2048 > ca-key.pem
openssl req -sha1 -new -x509 -nodes -days 3650 -key ca-key.pem > ca-cert.pem
openssl req -sha1 -newkey rsa:2048 -days 730 -nodes -keyout server-key.pem > server-req.pem
openssl rsa -in server-key.pem -out server-key.pem
openssl x509 -sha1 -req -in server-req.pem -days 730 -CA ca-cert.pem -CAkey ca-key.pem -set_serial 01 > server-cert.pem
```

Move server certificates to /etc/pki/tls/certs/mysql/

Directory path assumes CentOS or RHEL (adjust as needed for other distros):

```
mkdir /etc/pki/tls/certs/mysql/
```

Be sure to set permissions on the folder and files. mysql needs full ownership and access.

```
chown -R mysql:mysql /etc/pki/tls/certs/mysql
```

Now configure MySQL/MariaDB

```
# vi /etc/my.cnf
# i
```

```
[mysqld]
bind-address=*
ssl-ca=/etc/pki/tls/certs/ca-cert.pem
ssl-cert=/etc/pki/tls/certs/server-cert.pem
ssl-key=/etc/pki/tls/certs/server-key.pem
# :wq
```

然后

```
systemctl restart mariadb
```

别忘了打开防火墙，允许来自 appclient（使用 IP 1.2.3.4）的连接

```
firewall-cmd --zone=drop --permanent --add-rich-rule 'rule family="ipv4" source address="1.2.3.4"
service name="mysql" accept'
# 我将所有东西都强制放到投放区。根据需要调整上述命令的参数。
```

现在重启 firewalld

```
service firewalld restart
```

接下来，登录到 dbserver 的 mysql 服务器：

```
mysql -uroot -p
```

执行以下命令为客户端创建用户。注意 GRANT 语句中的 REQUIRE SSL。

```
GRANT ALL PRIVILEGES ON *.* TO 'iamsecure'@'appclient' IDENTIFIED BY 'dingdingding' REQUIRE SSL;
FLUSH PRIVILEGES;
# 退出 mysql
```

你应该仍然在第一步的 /root/certs/mysql 目录下。如果不在，请切换回该目录以执行下面的命令之一。

创建客户端证书

```
openssl req -sha1 -newkey rsa:2048 -days 730 -nodes -keyout client-key.pem > client-req.pem
openssl rsa -in client-key.pem -out client-key.pem
openssl x509 -sha1 -req -in client-req.pem -days 730 -CA ca-cert.pem -CAkey ca-key.pem -set_serial
01 > client-cert.pem
```

**注意：**我为服务器和客户端证书使用了相同的通用名称。实际情况可能有所不同。

确保你仍然在 /root/certs/mysql/ 目录下执行以下命令

将服务器和客户端的 CA 证书合并到一个文件中：

```
cat server-cert.pem client-cert.pem > ca.pem
```

确保你能看到两个证书：

```
cat ca.pem
```

服务器端工作暂时结束。

打开另一个终端并

```
[mysqld]
bind-address=*
ssl-ca=/etc/pki/tls/certs/ca-cert.pem
ssl-cert=/etc/pki/tls/certs/server-cert.pem
ssl-key=/etc/pki/tls/certs/server-key.pem
# :wq
```

Then

```
systemctl restart mariadb
```

Don't forget to open your firewall to allow connections from appclient (using IP 1.2.3.4)

```
firewall-cmd --zone=drop --permanent --add-rich-rule 'rule family="ipv4" source address="1.2.3.4"
service name="mysql" accept'
# I force everything to the drop zone. Season the above command to taste.
```

Now restart firewalld

```
service firewalld restart
```

Next, log in to dbserver's mysql server:

```
mysql -uroot -p
```

Issue the following to create a user for the client. note REQUIRE SSL in GRANT statement.

```
GRANT ALL PRIVILEGES ON *.* TO 'iamsecure'@'appclient' IDENTIFIED BY 'dingdingding' REQUIRE SSL;
FLUSH PRIVILEGES;
# quit mysql
```

You should still be in /root/certs/mysql from the first step. If not, cd back to it for one of the commands below.

Create the client certificates

```
openssl req -sha1 -newkey rsa:2048 -days 730 -nodes -keyout client-key.pem > client-req.pem
openssl rsa -in client-key.pem -out client-key.pem
openssl x509 -sha1 -req -in client-req.pem -days 730 -CA ca-cert.pem -CAkey ca-key.pem -set_serial
01 > client-cert.pem
```

**Note:** I used the same common name for both server and client certificates. YMMV.

Be sure you're still /root/certs/mysql/ for this next command

Combine server and client CA certificate into a single file:

```
cat server-cert.pem client-cert.pem > ca.pem
```

Make sure you see two certificates:

```
cat ca.pem
```

**END OF SERVER SIDE WORK FOR NOW.**

Open another terminal and

```
ssh appclient
```

像之前一样，为客户端证书创建一个永久存放位置

```
mkdir /etc/pki/tls/certs/mysql/
```

现在，将客户端证书（在 dbserver 上创建）放到 appclient 上。你可以通过 scp 传输，或者逐个复制粘贴文件。

```
scp dbserver
# 从 dbserver 复制文件到 appclient
# 退出 scp
```

同样，务必设置文件夹和文件的权限。mysql 需要完全的所有权和访问权限。

```
chown -R mysql:mysql /etc/pki/tls/certs/mysql
```

你应该有三个文件，每个文件的所有者都是 mysql 用户：

```
/etc/pki/tls/certs/mysql/ca.pem
/etc/pki/tls/certs/mysql/client-cert.pem
/etc/pki/tls/certs/mysql/client-key.pem
```

现在编辑 appclient 上 MariaDB/MySQL 配置文件中的 [client] 部分。

```
vi /etc/my.cnf
# i
[client]
ssl-ca=/etc/pki/tls/certs/mysql/ca.pem
ssl-cert=/etc/pki/tls/certs/mysql/client-cert.pem
ssl-key=/etc/pki/tls/certs/mysql/client-key.pem
# :wq
```

重启 appclient 的 mariadb 服务：

```
systemctl restart mariadb
```

仍然在客户端这里

这应该返回：ssl TRUE

```
mysql --ssl --help
```

现在，登录到 appclient 的 mysql 实例

```
mysql -uroot -p
```

下面两个变量都应该显示 YES

```
show variables LIKE '%ssl';
have_openssl      YES
have_ssl          YES
```

起初我看到

```
have_openssl 否
```

```
ssh appclient
```

As before, create a permanent home for the client certificates

```
mkdir /etc/pki/tls/certs/mysql/
```

Now, place the client certificates (created on dbserver) on appclient. You can either scp them over, or just copy and paste the files one by one.

```
scp dbserver
# copy files from dbserver to appclient
# exit scp
```

Again, be sure to set permissions on the folder and files. mysql needs full ownership and access.

```
chown -R mysql:mysql /etc/pki/tls/certs/mysql
```

You should have three files, each owned by user mysql:

```
/etc/pki/tls/certs/mysql/ca.pem
/etc/pki/tls/certs/mysql/client-cert.pem
/etc/pki/tls/certs/mysql/client-key.pem
```

Now edit appclient's MariaDB/MySQL config in the [client] section.

```
vi /etc/my.cnf
# i
[client]
ssl-ca=/etc/pki/tls/certs/mysql/ca.pem
ssl-cert=/etc/pki/tls/certs/mysql/client-cert.pem
ssl-key=/etc/pki/tls/certs/mysql/client-key.pem
# :wq
```

Restart appclient's mariadb service:

```
systemctl restart mariadb
```

still on the client here

This should return: ssl TRUE

```
mysql --ssl --help
```

Now, log in to appclient's mysql instance

```
mysql -uroot -p
```

Should see YES to both variables below

```
show variables LIKE '%ssl';
have_openssl      YES
have_ssl          YES
```

Initially I saw

```
have_openssl NO
```

快速查看 mariadb.log 发现：

SSL 错误：无法从 '/etc/pki/tls/certs/mysql/client-cert.pem' 获取证书

问题是 client-cert.pem 及其所在文件夹归 root 所有。解决方法是将

/etc/pki/tls/certs/mysql/ 的所有权设置为 mysql。

```
chown -R mysql:mysql /etc/pki/tls/certs/mysql
```

如有需要，请从上一步立即重启 mariadb

现在我们准备测试安全连接  
我们仍然在 appclient 上

尝试使用上面创建的账户连接到 dbserver 的 mysql 实例。

```
mysql -h dbserver -u iamsecure -p
# 输入密码 dingdingding (希望你已经改成别的密码)
```

只要运气好，你应该能无错误地登录。

要确认你已启用SSL连接，请在MariaDB/MySQL提示符下执行以下命令：

```
\s
```

这是反斜杠加s，也就是status命令

这将显示你的连接状态，应该类似如下内容：

```
连接ID：          4
当前数据库：
当前用户：        iamsecure@appclient
SSL：             使用的加密套件是DHE-RSA-AES256-GCM-SHA384
当前分页器：      stdout
使用输出文件：    "
使用分隔符：      ;
服务器：          MariaDB
服务器版本：      5.X.X-MariaDB MariaDB服务器
协议版本：        10
连接：通过 TCP/IP 连接到 dbserver
服务器字符集：latin1
数据库字符集：latin1
客户端字符集：utf8
连接字符集：utf8
TCP端口：3306
运行时间：42 分 13 秒
```

如果在尝试连接时遇到权限被拒绝错误，请检查上面的 GRANT 语句，确保没有多余的字符或引号。

如果遇到 SSL 错误，请回顾本指南，确保步骤顺序正确。

此方法在 RHEL7 上有效，可能在 CentOS7 上也适用。无法确认这些步骤在其他环境中是否有效。

A quick look into mariadb.log revealed:

SSL error: Unable to get certificate from '/etc/pki/tls/certs/mysql/client-cert.pem'

The problem was that root owned client-cert.pem and the containing folder. The solution was to set ownership of /etc/pki/tls/certs/mysql/ to mysql.

```
chown -R mysql:mysql /etc/pki/tls/certs/mysql
```

Restart mariadb if needed from the step immediately above

NOW WE ARE READY TO TEST THE SECURE CONNECTION  
We're still on appclient here

Attempt to connect to dbserver's mysql instance using the account created above.

```
mysql -h dbserver -u iamsecure -p
# enter password dingdingding (hopefully you changed that to something else)
```

With a little luck you should be logged in without error.

To confirm you are connected with SSL enabled, issue the following command from the MariaDB/MySQL prompt:

```
\s
```

That's a backslash s, aka status

That will show the status of your connection, which should look something like this:

```
Connection id:          4
Current database:
Current user:           iamsecure@appclient
SSL:                   Cipher in use is DHE-RSA-AES256-GCM-SHA384
Current pager:          stdout
Using outfile:          ''
Using delimiter:        ;
Server:                 MariaDB
Server version:         5.X.X-MariaDB MariaDB Server
Protocol version:       10
Connection:             dbserver via TCP/IP
Server characterset:    latin1
Db characterset:        latin1
Client characterset:    utf8
Conn. characterset:     utf8
TCP port:               3306
Uptime:                 42 min 13 sec
```

If you get permission denied errors on your connection attempt, check your GRANT statement above to make sure there aren't any stray characters or ' marks.

If you have SSL errors, go back through this guide to make sure the steps are orderly.

This worked on RHEL7 and will likely work on CentOS7, too. Cannot confirm whether these exact steps will work elsewhere.

希望这能为其他人节省一些时间和烦恼。

Hope this saves someone else a little time and aggravation.



# 第63章：创建新用户

## 第63.1节：创建 MySQL 用户

创建新用户需要遵循以下简单步骤：

步骤 1：以 root 用户身份登录 MySQL

```
$ mysql -u root -p
```

步骤 2：我们将看到 mysql 命令提示符

```
mysql> CREATE USER 'my_new_user'@'localhost' IDENTIFIED BY 'test_password';
```

这里，我们已经成功创建了新用户，但该用户没有任何权限，因此要赋予用户权限，请使用以下命令：

```
mysql> GRANT ALL PRIVILEGES ON my_db.* TO 'my_new_user'@'localhost' IDENTIFIED BY 'my_password';
```

## 第 63.2 节：指定密码

基本用法是：

```
mysql> CREATE USER 'my_new_user'@'localhost' IDENTIFIED BY 'test_password';
```

但是在不建议以明文硬编码密码的情况下，也可以使用指令 PASSWORD，指定由 PASSWORD() 函数返回的哈希值：

```
mysql> select PASSWORD('test_password'); -- 返回 *4414E26EDED6D661B5386813EBBA95065DBC4728
mysql> CREATE USER 'my_new_user'@'localhost' IDENTIFIED BY PASSWORD
'*4414E26EDED6D661B5386813EBBA95065DBC4728';
```

## 第63.3节：创建新用户并授予对

schema的所有权限

```
授予所有权限于 schema_name.* 给 'new_user_name'@'%', 认证方式为 'newpassword';
```

注意：这可以用来创建新的root用户

## 第63.4节：重命名用户

```
重命名用户 'user'@'% ' 为 'new_name`@'%';
```

如果误创建了用户，可以更改其名称

# Chapter 63: Create New User

## Section 63.1: Create a MySQL User

For creating new user, We need to follow simple steps as below :

Step 1: Login to MySQL as root

```
$ mysql -u root -p
```

Step 2 : We will see mysql command prompt

```
mysql> CREATE USER 'my_new_user'@'localhost' IDENTIFIED BY 'test_password';
```

Here, We have successfully created new user, But this user won't have any permissions, So to assign permissions to user use following command :

```
mysql> GRANT ALL PRIVILEGES ON my_db.* TO 'my_new_user'@'localhost' identified by 'my_password';
```

## Section 63.2: Specify the password

The basic usage is:

```
mysql> CREATE USER 'my_new_user'@'localhost' IDENTIFIED BY 'test_password';
```

However for situations where is not advisable to hard-code the password in cleartext it is also possible to specify directly, using the directive PASSWORD, the hashed value as returned by the PASSWORD() function:

```
mysql> select PASSWORD('test_password'); -- returns *4414E26EDED6D661B5386813EBBA95065DBC4728
mysql> CREATE USER 'my_new_user'@'localhost' IDENTIFIED BY PASSWORD
'*4414E26EDED6D661B5386813EBBA95065DBC4728';
```

## Section 63.3: Create new user and grant all priviliges to schema

```
grant all privileges on schema_name.* to 'new_user_name'@'%' identified by 'newpassword';
```

Attention: This can be used to create new root user

## Section 63.4: Renaming user

```
rename user 'user'@'%' to 'new_name`@'%';
```

If you create a user by mistake, you can change his name

# 第64章：通过GRANT实现安全

## 第64.1节：最佳实践

限制root（以及任何其他具有SUPER权限的用户）

```
GRANT ... TO root@localhost ...
```

这可以防止其他服务器的访问。你应该只把SUPER权限授予极少数人，并且他们应该清楚自己的责任。应用程序不应该拥有SUPER权限。

限制应用程序登录到它使用的那个数据库：

```
GRANT ... ON dbname.* ...
```

这样，入侵应用程序代码的人无法越过dbname。还可以通过以下任一方式进一步细化：

```
GRANT SELECT ON dbname.* ... -- “只读”
GRANT ... ON dbname.tblname ... -- “仅限一张表”
```

只读权限可能还需要一些“安全”的操作，比如

```
GRANT SELECT, CREATE TEMPORARY TABLE ON dbname.* ... -- “只读”
```

正如你说，没有绝对的安全。我的观点是，你可以做一些事情来减缓黑客的攻击速度。（诚实的人犯错也是同理。）

在极少数情况下，你可能需要应用程序执行只有root才能做的操作。这可以通过带有SECURITY DEFINER（由root定义）的“存储过程”来实现。这样只会暴露存储过程所执行的操作，例如，针对某张特定表的某个特定动作。

## 第64.2节：主机（user@host中的主机）

“主机”可以是主机名或IP地址。同时，也可以包含通配符。

```
GRANT SELECT ON db.* TO sam@'my.domain.com' IDENTIFIED BY 'foo';
```

示例：注意：这些通常需要加引号

```
localhost -- 与 mysqld 相同的机器
'my.domain.com' -- 特定域名；这涉及查找
'11.22.33.44' -- 特定IP地址
'192.168.1.%' -- IP地址后缀的通配符。（192.168.% 和 10.% 以及 11.% 是“内部”IP地址。）
```

使用 localhost 依赖于服务器的安全性。最佳实践是 root 只允许通过 localhost 访问。在某些情况下，这些表示相同的含义：0.0.0.1 和 ::1。

# Chapter 64: Security via GRANTS

## Section 64.1: Best Practice

Limit root (and any other SUPER-privileged user) to

```
GRANT ... TO root@localhost ...
```

That prevents access from other servers. You should hand out SUPER to very few people, and they should be aware of their responsibility. The application should not have SUPER.

Limit application logins to the one database it uses:

```
GRANT ... ON dbname.* ...
```

That way, someone who hacks into the application code can't get past dbname. This can be further refined via either of these:

```
GRANT SELECT ON dbname.* ... -- “read only”
GRANT ... ON dbname.tblname ... -- “just one table”
```

The readonly may also need 'safe' things like

```
GRANT SELECT, CREATE TEMPORARY TABLE ON dbname.* ... -- “read only”
```

As you say, there is no absolute security. My point here is there you can do a few things to slow hackers down. (Same goes for honest people goofing.)

In rare cases, you may need the application to do something available only to root. this can be done via a "Stored Procedure" that has SECURITY DEFINER (and root defines it). That will expose only what the SP does, which might, for example, be one particular action on one particular table.

## Section 64.2: Host (of user@host)

The "host" can be either a host name or an IP address. Also, it can involve wild cards.

```
GRANT SELECT ON db.* TO sam@'my.domain.com' IDENTIFIED BY 'foo';
```

Examples: Note: these usually need to be quoted

```
localhost -- the same machine as mysqld
'my.domain.com' -- a specific domain; this involves a lookup
'11.22.33.44' -- a specific IP address
'192.168.1.%' -- wild card for trailing part of IP address. (192.168.% and 10.% and 11.% are "internal" ip addresses.)
```

Using localhost relies on the security of the server. For best practice root should only be allowed in through localhost. In some cases, these mean the same thing: 0.0.0.1 and ::1.

# 第65章：更改密码

## 第65.1节：在Linux中更改MySQL root密码

更改MySQL root用户密码：

步骤1： 停止MySQL服务器。

- 在 Ubuntu 或 Debian 中：  
sudo /etc/init.d/mysql stop
- 在 CentOS、Fedora 或 Red Hat 企业版 Linux 中：  
sudo /etc/init.d/mysqld stop

步骤 2： 启动 MySQL 服务器，跳过权限系统。

```
sudo mysqld_safe --skip-grant-tables &
```

或者，如果 mysqld\_safe 不可用，

```
sudo mysqld --skip-grant-tables &
```

步骤 3： 连接到 MySQL 服务器。

```
mysql -u root
```

步骤4： 为root用户设置新密码。

```
版本 > 5.7
FLUSH PRIVILEGES;
ALTER USER 'root'@'localhost' IDENTIFIED BY 'new_password';
FLUSH PRIVILEGES;
exit;
```

```
版本 ≤ 5.7
FLUSH PRIVILEGES;
SET PASSWORD FOR 'root'@'localhost' = PASSWORD('new_password');
FLUSH PRIVILEGES;
exit;
```

注意：ALTER USER 语法是在 MySQL 5.7.6 中引入的。

步骤 5: 重启 MySQL 服务器。

- 在 Ubuntu 或 Debian 中：  
sudo /etc/init.d/mysql stop  
sudo /etc/init.d/mysql start
- 在 CentOS、Fedora 或 Red Hat 企业版 Linux 中：  
sudo /etc/init.d/mysqld stop  
sudo /etc/init.d/mysqld start

## 第 65.2 节：在 Windows 中更改 MySQL root 密码

当我们想要更改 Windows 中的 root 密码时，需要按照以下步骤操作：

步骤 1： 使用以下任一方法启动命令提示符：

# Chapter 65: Change Password

## Section 65.1: Change MySQL root password in Linux

To change MySQL's root user password:

Step 1: Stop the MySQL server.

- in Ubuntu or Debian:  
sudo /etc/init.d/mysql stop
- in CentOS, Fedora or Red Hat Enterprise Linux:  
sudo /etc/init.d/mysqld stop

Step 2: Start the MySQL server without the privilege system.

```
sudo mysqld_safe --skip-grant-tables &
```

or, if mysqld\_safe is unavailable,

```
sudo mysqld --skip-grant-tables &
```

Step 3: Connect to the MySQL server.

```
mysql -u root
```

Step 4: Set a new password for root user.

```
Version > 5.7
FLUSH PRIVILEGES;
ALTER USER 'root'@'localhost' IDENTIFIED BY 'new_password';
FLUSH PRIVILEGES;
exit;
```

```
Version ≤ 5.7
FLUSH PRIVILEGES;
SET PASSWORD FOR 'root'@'localhost' = PASSWORD('new_password');
FLUSH PRIVILEGES;
exit;
```

Note: The ALTER USER syntax was introduced in MySQL 5.7.6.

Step 5: Restart the MySQL server.

- in Ubuntu or Debian:  
sudo /etc/init.d/mysql stop  
sudo /etc/init.d/mysql start
- in CentOS, Fedora or Red Hat Enterprise Linux:  
sudo /etc/init.d/mysqld stop  
sudo /etc/init.d/mysqld start

## Section 65.2: Change MySQL root password in Windows

When we want to change root password in windows, We need to follow following steps :

Step 1 : Start your Command Prompt by using any of below method :

按下 Ctrl+R 或者进入 开始 菜单 > 运行 ，然后输入 cmd 并按回车

步骤 2 ： 将目录切换到 MYSQL 安装的位置，我这里是

```
C:\> cd C:\mysql\bin
```

步骤 3 ： 现在需要启动 mysql 命令提示符

```
C:\mysql\bin> mysql -u root mysql
```

步骤 4 ： 执行查询以更改 root 密码

```
mysql> SET PASSWORD FOR root@localhost=PASSWORD('my_new_password');
```

## 第 65.3 节：流程

1. 停止 MySQL (mysqld) 服务器/守护进程。
2. 启动 MySQL 服务器进程时使用 --skip-grant-tables 选项，这样它将不会提示输入密码：  
mysqld\_safe --跳过-授权-表 &
3. 以 root 用户连接到 MySQL 服务器：mysql -u root
4. 修改密码：
  - (5.7.6 及更新版本)：ALTER USER 'root'@'localhost' IDENTIFIED BY 'new-password';
  - (5.7.5 及更早版本，或 MariaDB)：SET PASSWORD FOR 'root'@'localhost' = PASSWORD('new-password'); flush privileges; quit;
5. 重启 MySQL 服务器。

注意：此方法仅在您物理上位于同一服务器时有效。

在线文档：<http://dev.mysql.com/doc/refman/5.7/en/resetting-permissions.html>

Perss Crt1+R or Goto Start Menu > Run and then type cmd and hit enter

Step 2 : Change your directory to where MYSQL is installed, In my case it's

```
C:\> cd C:\mysql\bin
```

Step 3 : Now we need to start mysql command prompt

```
C:\mysql\bin> mysql -u root mysql
```

Step 4 : Fire query to change root password

```
mysql> SET PASSWORD FOR root@localhost=PASSWORD('my_new_password');
```

## Section 65.3: Process

1. Stop the MySQL (mysqld) server/daemon process.
2. Start the MySQL server process the --skip-grant-tables option so that it will not prompt for a password:  
mysqld\_safe --skip-grant-tables &
3. Connect to the MySQL server as the root user: mysql -u root
4. Change password:
  - (5.7.6 and newer): ALTER USER 'root'@'localhost' IDENTIFIED BY 'new-password';
  - (5.7.5 and older, or MariaDB): SET PASSWORD FOR 'root'@'localhost' = PASSWORD('new-password'); flush privileges; quit;
5. Restart the MySQL server.

Note: this will work only if you are physically on the same server.

Online Doc: <http://dev.mysql.com/doc/refman/5.7/en/resetting-permissions.html>

# 第66章：恢复并重置 MySQL 5.7+ 的默认 root 密码

MySQL 5.7 之后，安装 MySQL 时有时不需要创建 root 账户或设置 root 密码。默认情况下，当我们启动服务器时，默认密码会存储在 `mysqld.log` 文件中。我们需要使用该密码登录系统，并且需要更改它。

## 第66.1节：服务器初次启动时会发生什么

假设服务器的数据目录为空：

- 服务器被初始化。
- SSL证书和密钥文件会在数据目录中生成。
- `validate_password` 插件被安装并启用。
- 超级用户账户 `'root'@'localhost'` 被创建。超级用户的密码被设置并存储在错误日志文件中。

## 第66.2节：如何使用默认密码更改root密码

要查看默认的“root”密码：

```
shell> sudo grep 'temporary password' /var/log/mysqld.log
```

请尽快使用生成的临时密码登录并更改root密码，为超级用户账户设置自定义密码：

```
shell> mysql -uroot -p

mysql> ALTER USER 'root'@'localhost' IDENTIFIED BY 'MyNewPass5!';
```

注意：MySQL 的 `validate_password` 插件默认已安装。该插件要求密码至少包含一个大写字母、一个小写字母、一个数字和一个特殊字符，且密码总长度至少为 8 个字符。

## 第 66.3 节：当 UNIX 套接字文件 "/var/run/mysqld" 不存在时重置 root 密码

如果我忘记密码，就会出现错误。

```
$ mysql -u root -p
```

请输入密码：

错误 1045 (28000)：用户 `'root'@'localhost'` 访问被拒绝（使用密码：是）

我尝试先查看状态来解决该问题：

```
$ systemctl status mysql.service
```

mysql.service - MySQL 社区服务器 已加载：已加载 (/lib/systemd/system/mysql.service; 启用; 供应商

# Chapter 66: Recover and reset the default root password for MySQL 5.7+

After MySQL 5.7, when we install MySQL sometimes we don't need to create a root account or give a root password. By default when we start the server, the default password is stored in the `mysqld.log` file. We need to login in to the system using that password and we need to change it.

## Section 66.1: What happens when the initial start up of the server

Given that the data directory of the server is empty:

- The server is initialized.
- SSL certificate and key files are generated in the data directory.
- The `validate_password` plugin is installed and enabled.
- The superuser account `'root'@'localhost'` is created. The password for the superuser is set and stored in the error log file.

## Section 66.2: How to change the root password by using the default password

To reveal the default "root" password:

```
shell> sudo grep 'temporary password' /var/log/mysqld.log
```

Change the root password as soon as possible by logging in with the generated temporary password and set a custom password for the superuser account:

```
shell> mysql -uroot -p

mysql> ALTER USER 'root'@'localhost' IDENTIFIED BY 'MyNewPass5!';
```

**Note:** MySQL's `validate_password` plugin is installed by default. This will require that passwords contain at least one upper case letter, one lower case letter, one digit, and one special character, and that the total password length is at least 8 characters.

## Section 66.3: reset root password when " /var/run/mysqld" for UNIX socket file don't exists"

if I forget the password then I'll get error.

```
$ mysql -u root -p
```

Enter password:

ERROR 1045 (28000): Access denied for user `'root'@'localhost'` (using password: YES)

I tried to solve the issue by first knowing the status:

```
$ systemctl status mysql.service
```

mysql.service - MySQL Community Server Loaded: loaded (/lib/systemd/system/mysql.service; enabled; vendor



preset: en 活动状态：active（运行中），自 2017-06-08 星期四 14:31:33 IST 起；38秒前

然后我使用了代码 `mysqld_safe --skip-grant-tables &` 但出现错误：

`mysqld_safe` UNIX 套接字文件的目录 `'/var/run/mysqld'` 不存在。

```
$ systemctl stop mysql.service
$ ps -eaf|grep mysql
$ mysqld_safe --skip-grant-tables &
```

我解决了：

```
$ mkdir -p /var/run/mysqld
$ chown mysql:mysql /var/run/mysqld
```

现在我使用相同的代码 `mysqld_safe --skip-grant-tables &` 并得到

`mysqld_safe` 使用 `/var/lib/mysql` 中的数据库启动 `mysqld` 守护进程

如果我使用 `$ mysql -u root` 会得到：

服务器版本：5.7.18-0ubuntu0.16.04.1 (Ubuntu)

版权所有 (c) 2000, 2017, Oracle 及/或其关联公司。保留所有权利。

Oracle是Oracle公司及/或其附属公司的注册商标。其他名称可能是其各自所有者的商标。

输入 'help;' 或 '\h' 获取帮助。输入 '\c' 以清除当前输入语句。

mysql>

现在是修改密码的时候了：

```
mysql> use mysql
mysql> describe user;
```

正在读取表信息以完成表和列名。您可以通过 `-A` 关闭此功能以加快启动速度

数据库已更改

```
mysql> FLUSH PRIVILEGES;
mysql> SET PASSWORD FOR root@'localhost' = PASSWORD('newpwd');
```

或者，如果您有一个可以从任何地方连接的 `mysql root` 账户，您也应该执行：

```
UPDATE mysql.user SET Password=PASSWORD('newpwd') WHERE User='root';
```

preset: en Active: active (running) since Thu 2017-06-08 14:31:33 IST; 38s ago

Then I used the code `mysqld_safe --skip-grant-tables &` but I get the error:

`mysqld_safe` Directory `'/var/run/mysqld'` for UNIX socket file don't exists.

```
$ systemctl stop mysql.service
$ ps -eaf|grep mysql
$ mysqld_safe --skip-grant-tables &
```

I solved:

```
$ mkdir -p /var/run/mysqld
$ chown mysql:mysql /var/run/mysqld
```

Now I use the same code `mysqld_safe --skip-grant-tables &` and get

`mysqld_safe` Starting `mysqld` daemon with databases from `/var/lib/mysql`

If I use `$ mysql -u root` I'll get :

Server version: 5.7.18-0ubuntu0.16.04.1 (Ubuntu)

Copyright (c) 2000, 2017, Oracle and/or its affiliates. All rights reserved.

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

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

mysql>

Now time to change password:

```
mysql> use mysql
mysql> describe user;
```

Reading table information for completion of table and column names You can turn off this feature to get a quicker startup with `-A`

Database changed

```
mysql> FLUSH PRIVILEGES;
mysql> SET PASSWORD FOR root@'localhost' = PASSWORD('newpwd');
```

or If you have a `mysql root` account that can connect from everywhere, you should also do:

```
UPDATE mysql.user SET Password=PASSWORD('newpwd') WHERE User='root';
```

替代方法：

```
USE mysql
UPDATE user SET Password = PASSWORD('newpwd')
WHERE Host = 'localhost' AND User = 'root';
```

如果你有一个可以从任何地方访问的root账户：

```
USE mysql
UPDATE user SET Password = PASSWORD('newpwd')
WHERE Host = '%' AND User = 'root';`enter code here`
```

现在需要退出mysql并停止/启动

```
FLUSH PRIVILEGES;
sudo /etc/init.d/mysql stop
sudo /etc/init.d/mysql start
```

现在再次使用`mysql -u root -p`，并使用新密码登录

```
| mysql>
```

Alternate Method:

```
USE mysql
UPDATE user SET Password = PASSWORD('newpwd')
WHERE Host = 'localhost' AND User = 'root';
```

And if you have a root account that can access from everywhere:

```
USE mysql
UPDATE user SET Password = PASSWORD('newpwd')
WHERE Host = '%' AND User = 'root';`enter code here`
```

now need to quit from mysql and stop/start

```
FLUSH PRIVILEGES;
sudo /etc/init.d/mysql stop
sudo /etc/init.d/mysql start
```

now again `mysql -u root -p` and use the new password to get

```
| mysql>
```

# 第67章：恢复丢失的root密码

## 第67.1节：设置root密码，启用root用户以进行socket和http访问

解决问题：使用密码YES登录用户root时访问被拒绝 停止MySQL：

```
sudo systemctl stop mysql
```

重启MySQL，跳过授权表：

```
sudo mysqld_safe --skip-grant-tables
```

登录：

```
mysql -u root
```

在SQL命令行中，查看是否存在用户：

```
select User, password,plugin FROM mysql.user ;
```

更新用户（plugin设为null以启用所有插件）：

```
update mysql.user set password=PASSWORD('mypassword'), plugin = NULL WHERE User = 'root';
exit;
```

在Unix命令行中停止不带授权表的MySQL，然后带授权表重启：

```
sudo service mysql 停止
sudo service mysql 启动
```

# Chapter 67: Recover from lost root password

## Section 67.1: Set root password, enable root user for socket and http access

Solves problem of: access denied for user root using password YES Stop MySQL:

```
sudo systemctl stop mysql
```

Restart MySQL, skipping grant tables:

```
sudo mysqld_safe --skip-grant-tables
```

Login:

```
mysql -u root
```

In SQL shell, look if users exist:

```
select User, password,plugin FROM mysql.user ;
```

Update the users (plugin null enables for all plugins):

```
update mysql.user set password=PASSWORD('mypassword'), plugin = NULL WHERE User = 'root';
exit;
```

In Unix shell stop MySQL without grant tables, then restart with grant tables:

```
sudo service mysql stop
sudo service mysql start
```

# 第68章：MySQL性能技巧

## 第68.1节：构建复合索引

在许多情况下，复合索引的性能优于单列索引。要构建一个最优的复合索引，请按以下顺序填充列。

- = 来自 **WHERE** 子句的列优先。（例如，**INDEX**(a,b,...) 用于 **WHERE** a=12 **AND** b='xyz' ...）
- IN 列；优化器可能能够跳跃式地通过索引。
- 一个“范围”（例如 x BETWEEN 3 AND 9, name LIKE 'J%') 它不会使用第一个范围列之后的任何列。
- 所有 GROUP BY 中的列，按顺序
- 所有 ORDER BY 中的列，按顺序。仅当全部为 ASC 或全部为 DESC，或使用8.0版本时有效。

注意事项和例外：

- 不要重复任何列。
- 跳过任何不适用的情况。
- 如果你没有使用WHERE的所有列，就没有必要继续使用GROUP BY等。
- 有些情况下，仅对ORDER BY列建立索引而忽略WHERE是有用的。
- 不要在函数中“隐藏”列（例如DATE(x) = ...无法使用索引中的x）。
- “前缀”索引（例如text\_col(99)）通常不会有帮助，甚至可能有害。

[更多细节和技巧。](#)

## 第68.2节：优化InnoDB表的存储布局

1. 在InnoDB中，拥有一个较长的主键（无论是单列的长值，还是由多列组成的长复合值）会浪费大量磁盘空间。行的主键值会在所有指向该行的二级索引记录中重复存储。如果你的主键较长，建议创建一个AUTO\_INCREMENT列作为主键。
2. 对于存储可变长度字符串或包含大量NULL值的列，使用VARCHAR数据类型代替CHAR。CHAR(N)列总是占用N个字符的存储空间，即使字符串较短或值为NULL。较小的表更适合缓冲池，能减少磁盘I/O。

当使用COMPACT行格式（InnoDB的默认格式）和可变长度字符集（如utf8或sjis）时，CHAR(N)列占用的空间是可变的，但至少为N字节。

3. 对于较大或包含大量重复文本或数字数据的表，考虑使用COMPRESSED行格式。这样可以减少将数据加载到缓冲池或执行全表扫描时的磁盘I/O。在做出最终决定前，测量使用COMPRESSED与COMPACT行格式所能达到的压缩率。注意：基准测试很少显示超过2:1的压缩率，并且COMPRESSED格式在缓冲池中有较大开销。
4. 一旦数据达到稳定大小，或者增长的表增加了几十或几百兆字节，考虑使用OPTIMIZE TABLE语句来重组表并压缩任何浪费的空间。重组后的表在执行全表扫描时需要更少的磁盘I/O。这是一种简单直接的技术，当改进索引使用或调整应用代码等其他方法不可行时，可以提升性能。注意：无论表大小如何，OPTIMIZE TABLE应尽量少用，因为它代价较高，且很少能带来足够的改进。InnoDB在保持其B+树结构中较少浪费空间方面表现相当不错。

# Chapter 68: MySQL Performance Tips

## Section 68.1: Building a composite index

In many situations, a composite index performs better than an index with a single column. To build an optimal composite index, populate it with columns in this order.

- = column(s) from the **WHERE** clause first. (eg, **INDEX**(a,b,...) for **WHERE** a=12 **AND** b='xyz' ...)
- IN column(s); the optimizer may be able to leapfrog through the index.
- One "range" (eg x BETWEEN 3 AND 9, name LIKE 'J%') It won't use anything past the first range column.
- All the columns in **GROUP BY**, in order
- All the columns in **ORDER BY**, in order. Works only if all are **ASC** or all are **DESC** or you are using 8.0.

Notes and exceptions:

- Don't duplicate any columns.
- Skip over any cases that don't apply.
- If you don't use all the columns of **WHERE**, there is no need to go on to **GROUP BY**, etc.
- There are cases where it is useful to index only the **ORDER BY** column(s), ignoring **WHERE**.
- Don't "hide" a column in a function (eg DATE(x) = ... cannot use x in the index.)
- 'Prefix' indexing (eg, text\_col(99)) is unlikely to be helpful; may hurt.

[More details and tips](#) .

## Section 68.2: Optimizing Storage Layout for InnoDB Tables

1. In InnoDB, having a long PRIMARY KEY (either a single column with a lengthy value, or several columns that form a long composite value) wastes a lot of disk space. The primary key value for a row is duplicated in all the secondary index records that point to the same row. Create an AUTO\_INCREMENT column as the primary key if your primary key is long.
2. Use the VARCHAR data type instead of CHAR to store variable-length strings or for columns with many NULL values. A CHAR(N) column always takes N characters to store data, even if the string is shorter or its value is NULL. Smaller tables fit better in the buffer pool and reduce disk I/O.

When using COMPACT row format (the default InnoDB format) and variable-length character sets, such as utf8 or sjis, CHAR(N) columns occupy a variable amount of space, but still at least N bytes.

3. For tables that are big, or contain lots of repetitive text or numeric data, consider using COMPRESSED row format. Less disk I/O is required to bring data into the buffer pool, or to perform full table scans. Before making a permanent decision, measure the amount of compression you can achieve by using COMPRESSED versus COMPACT row format. *Caveat:* Benchmarks rarely show better than 2:1 compression and there is a lot of overhead in the buffer\_pool for COMPRESSED.
4. Once your data reaches a stable size, or a growing table has increased by tens or some hundreds of megabytes, consider using the OPTIMIZE TABLE statement to reorganize the table and compact any wasted space. The reorganized tables require less disk I/O to perform full table scans. This is a straightforward technique that can improve performance when other techniques such as improving index usage or tuning application code are not practical. *Caveat:* Regardless of table size, OPTIMIZE TABLE should only rarely be performed. This is because it is costly, and rarely improves the table enough to be worth it. InnoDB is reasonably good at keeping its B+Trees free of a lot of wasted space.

OPTIMIZE TABLE 会复制表的数据部分并重建索引。其好处来自于索引内数据的更好打包，以及表空间和磁盘上的碎片减少。这些好处因每个表中的数据而异。你可能会发现某些表有显著的提升，而其他表则没有，或者提升会随着时间推移而减少，直到你下一次优化该表。如果表很大，或者重建的索引无法全部放入缓冲池中，该操作可能会很慢。向表中添加大量数据后的第一次运行通常比后续运行慢得多。

OPTIMIZE TABLE copies the data part of the table and rebuilds the indexes. The benefits come from improved packing of data within indexes, and reduced fragmentation within the tablespaces and on disk. The benefits vary depending on the data in each table. You may find that there are significant gains for some and not for others, or that the gains decrease over time until you next optimize the table. This operation can be slow if the table is large or if the indexes being rebuilt do not fit into the buffer pool. The first run after adding a lot of data to a table is often much slower than later runs.



# 第69章：性能调优

## 第69.1节：不要在函数中隐藏

一个常见错误是将带索引的列隐藏在函数调用中。例如，以下情况索引无法发挥作用：

```
WHERE DATE(dt) = '2000-01-01'
```

相反，假设有 INDEX(dt)，则以下写法可以使用索引：

```
WHERE dt = '2000-01-01' -- 如果 `dt` 的数据类型是 `DATE`
```

这适用于 DATE、DATETIME、TIMESTAMP，甚至 DATETIME(6)（微秒）：

```
WHERE dt >= '2000-01-01'
AND dt < '2000-01-01' + INTERVAL 1 DAY
```

## 第69.2节：OR

通常情况下，OR 会破坏优化。

```
WHERE a = 12 OR b = 78
```

不能使用INDEX(a,b)，且可能使用也可能不使用通过“索引合并”的INDEX(a)、INDEX(b)。索引合并总比没有好，但仅仅稍好一点。

```
WHERE x = 3 OR x = 5
```

被转换为

```
WHERE x IN (3, 5)
```

这可能使用包含 x 的索引。

## 第69.3节：添加正确的索引

这是一个庞大的话题，但它也是最重要的“性能”问题。

新手的主要教训是学习“复合”索引。这里有一个快速示例：

```
INDEX(last_name, first_name)
```

非常适合以下情况：

```
WHERE last_name = '...'
WHERE first_name = '...' AND last_name = '...' -- (WHERE中的顺序无关紧要)
```

但不适合

```
WHERE first_name = '...' -- INDEX中的顺序确实很重要
WHERE last_name = '...' OR first_name = '...' -- “OR” 是致命的
```

# Chapter 69: Performance Tuning

## Section 69.1: Don't hide in function

A common mistake is to hide an indexed column inside a function call. For example, this can't be helped by an index:

```
WHERE DATE(dt) = '2000-01-01'
```

Instead, given INDEX(dt) then these may use the index:

```
WHERE dt = '2000-01-01' -- if `dt` is datatype `DATE`
```

This works for DATE, DATETIME, TIMESTAMP, and even DATETIME(6) (microseconds):

```
WHERE dt >= '2000-01-01'
AND dt < '2000-01-01' + INTERVAL 1 DAY
```

## Section 69.2: OR

In general OR kills optimization.

```
WHERE a = 12 OR b = 78
```

cannot use INDEX(a, b), and may or may not use INDEX(a), INDEX(b) via "index merge". Index merge is better than nothing, but only barely.

```
WHERE x = 3 OR x = 5
```

is turned into

```
WHERE x IN (3, 5)
```

which may use an index with x in it.

## Section 69.3: Add the correct index

This is a huge topic, but it is also the most important "performance" issue.

The main lesson for a novice is to learn of "composite" indexes. Here's a quick example:

```
INDEX(last_name, first_name)
```

is excellent for these:

```
WHERE last_name = '...'
WHERE first_name = '...' AND last_name = '...' -- (order in WHERE does not matter)
```

but not for

```
WHERE first_name = '...' -- order in INDEX _does_ matter
WHERE last_name = '...' OR first_name = '...' -- "OR" is a killer
```

第69.4节：拥有索引

加快任何非微小表查询速度最重要的事情是拥有合适的索引。

```
WHERE a = 12 --> INDEX(a)
WHERE a > 12 --> INDEX(a)

WHERE a = 12 AND b > 78 --> INDEX(a,b) 比INDEX(b,a)更有用
WHERE a > 12 AND b > 78 --> INDEX(a) 或INDEX(b)；无法同时处理b的两个范围

ORDER BY x --> INDEX(x)
ORDER BY x, y --> INDEX(x,y) 按该顺序
ORDER BY x DESC, y ASC --> 没有索引能帮忙—因为混合了ASC和DESC
```

第69.5节：子查询

子查询有多种形式，且它们的优化潜力不同。首先，注意子查询可以是“相关的”或“无关的”。相关意味着它们依赖于子查询外部的某些值。这通常意味着子查询必须为每个外部值重新计算。

这种相关子查询通常表现不错。注意：它必须最多返回1个值。它常作为LEFT JOIN的替代方案，尽管不一定比LEFT JOIN更快。

```
SELECT a, b, ( SELECT ... FROM t WHERE t.x = u.x ) AS c
FROM u ...
SELECT a, b, ( SELECT MAX(x) ... ) AS c
FROM u ...
SELECT a, b, ( SELECT x FROM t ORDER BY ... LIMIT 1 ) AS c
FROM u ...
```

这通常是无关的：

```
SELECT ...
FROM ( SELECT ... ) AS a
JOIN b ON ...
```

关于FROM-SELECT的说明：

- 如果返回1行，那很好。
- 一个好的范例（同样是“1行”）是子查询为(SELECT@n:=0)，从而初始化一个`@变量以供查询的其余部分使用。
- 如果返回多行且JOIN也是(SELECT...)返回多行，那么效率可能非常差。5.6之前，没有索引，因此变成了CROSS JOIN；5.6及以后版本会推断临时表上的最佳索引并生成它，但在完成SELECT后会将其丢弃。

第69.6节：JOIN + GROUP BY

一个常见导致查询效率低下的问题大致如下：

```
SELECT ...
FROM a
JOIN b ON ...
WHERE ...
GROUP BY a.id
```

首先，JOIN会扩展行数；然后GROUP BY将其缩减回 a中的行数。

Section 69.4: Have an INDEX

The most important thing for speeding up a query on any non-tiny table is to have a suitable index.

```
WHERE a = 12 --> INDEX(a)
WHERE a > 12 --> INDEX(a)

WHERE a = 12 AND b > 78 --> INDEX(a,b) is more useful than INDEX(b,a)
WHERE a > 12 AND b > 78 --> INDEX(a) or INDEX(b)；no way to handle both ranges

ORDER BY x --> INDEX(x)
ORDER BY x, y --> INDEX(x,y) in that order
ORDER BY x DESC, y ASC --> No index helps - because of mixing ASC and DESC
```

Section 69.5: Subqueries

Subqueries come in several flavors, and they have different optimization potential. First, note that subqueries can be either "correlated" or "uncorrelated". Correlated means that they depend on some value from outside the subquery. This generally implies that the subquery *must* be re-evaluated for each outer value.

This correlated subquery is often pretty good. Note: It must return at most 1 value. It is often useful as an alternative to, though not necessarily faster than, a LEFT JOIN.

```
SELECT a, b, ( SELECT ... FROM t WHERE t.x = u.x ) AS c
FROM u ...
SELECT a, b, ( SELECT MAX(x) ... ) AS c
FROM u ...
SELECT a, b, ( SELECT x FROM t ORDER BY ... LIMIT 1 ) AS c
FROM u ...
```

This is usually uncorrelated:

```
SELECT ...
FROM ( SELECT ... ) AS a
JOIN b ON ...
```

Notes on the FROM-SELECT:

- If it returns 1 row, great.
- A good paradigm (again "1 row") is for the subquery to be ( SELECT @n := 0 ), thereby initializing an `@variable for use in the rest or the query.
- If it returns many rows *and* the JOIN also is ( SELECT ... ) with many rows, then efficiency can be terrible. Pre-5.6, there was no index, so it became a CROSS JOIN; 5.6+ involves deducing the best index on the temp tables and then generating it, only to throw it away when finished with the SELECT.

Section 69.6: JOIN + GROUP BY

A common problem that leads to an inefficient query goes something like this:

```
SELECT ...
FROM a
JOIN b ON ...
WHERE ...
GROUP BY a.id
```

First, the JOIN expands the number of rows; then the GROUP BY whittles it back down the the number of rows in a.

解决这个爆炸-压缩问题可能没有好的选择。一个可能的方案是将JOIN转换为SELECT中的相关子查询。这也消除了GROUP BY。

## 第69.7节：正确设置缓存

innodb\_buffer\_pool\_size 应该约为可用内存的70%。

## 第69.8节：负面情况

以下是一些不太可能提升性能的做法。它们源于过时的信息和/或天真想法。

- InnoDB已经改进到MyISAM不太可能更好的程度。
- 分区（PARTITIONing）很少能带来性能提升；甚至可能会降低性能。
- 将query\_cache\_size设置得超过100M通常会降低性能。
- 在my.cnf中增加大量参数值可能导致“交换”，这是一种严重的性能问题。
- “前缀索引”（例如INDEX(foo(20))) 通常是无用的。
- OPTIMIZE TABLE几乎总是无用的。（而且它会锁定表。）

There may not be any good choices to solve this explode-implode problem. One possible option is to turn the JOIN into a correlated subquery in the SELECT. This also eliminates the GROUP BY.

## Section 69.7: Set the cache correctly

innodb\_buffer\_pool\_size should be about 70% of available RAM.

## Section 69.8: Negatives

Here are some things that are not likely to help performance. They stem from out-of-date information and/or naivety.

- InnoDB has improved to the point where MyISAM is unlikely to be better.
- PARTITIONing rarely provides performance benefits; it can even hurt performance.
- Setting query\_cache\_size bigger than 100M will usually *hurt* performance.
- Increasing lots of values in my.cnf may lead to 'swapping', which is a *serious* performance problem.
- "Prefix indexes" (such as INDEX( foo(20) )) are generally useless.
- OPTIMIZE TABLE is almost always useless. (And it involves locking the table.)

# 附录 A：保留字

MySQL 有一些特殊名称，称为保留字。保留字只有在用反引号 ( ` ) 包裹时，才能用作表、列等的标识符，否则会导致错误。

为避免此类错误，要么不要使用保留字作为标识符，要么将有问题的标识符用反引号包裹。

## 第 A.1 节：因保留字引起的错误

尝试从名为order的表中查询，如下所示

```
select * from order
```

会出现错误：

错误代码：1064。您的 SQL 语法有误；请检查与您的 MySQL 服务器版本对应的手册，以获取在 'order' 附近使用的正确语法，错误发生在第 1 行

MySQL 中的保留关键字需要用反引号 ( ` ) 转义

```
select * from `order`
```

区分关键字和表名或列名。

另见：[由于在MySQL中将保留字用作表名或列名导致的语法错误。](#)

# Appendix A: Reserved Words

MySQL has some special names called *reserved words*. A reserved word can be used as an identifier for a table, column, etc. only if it's wrapped in backticks ( ` ), otherwise it will give rise to an error.

To avoid such errors, either don't use reserved words as identifiers or wrap the offending identifier in backticks.

## Section A.1: Errors due to reserved words

When trying to select from a table called order like this

```
select * from order
```

the error rises:

Error Code: 1064. You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'order' at line 1

Reserved keywords in MySQL need to be escaped with backticks ( ` )

```
select * from `order`
```

to distinguish between a keyword and a table or column name.

See also: [Syntax error due to using a reserved word as a table or column name in MySQL.](#)

致谢

非常感谢所有来自Stack Overflow文档的人员帮助提供此内容，  
更多更改可发送至[web@petercv.com](mailto:web@petercv.com)以发布或更新新内容

<a href="#">0x49D1</a>	第10章
<a href="#">4thfloorstudios</a>	第12章
<a href="#">一名程序员</a>	第62章
<a href="#">AJ</a>	第21章、第24章和第30章
<a href="#">阿布巴卡尔</a>	第20章
<a href="#">亚当</a>	第14章
<a href="#">agold</a>	第50章
<a href="#">亚历克斯·雷卡雷</a>	第31章
<a href="#">alex9311</a>	第25章
<a href="#">阿尔瓦罗·弗拉尼奥·拉隆多</a>	第6章
<a href="#">阿曼·丹达</a>	第1章
<a href="#">阿米纳达夫</a>	第63章
<a href="#">安迪</a>	第1章
<a href="#">andygeers</a>	第25章
<a href="#">阿尼·梅农</a>	第3章和第53章
<a href="#">animuson</a>	第6章
<a href="#">aries12</a>	第52章
<a href="#">arushi</a>	第68章
<a href="#">Aryo</a>	第25章
<a href="#">Asaph</a>	第50章和第52章
<a href="#">Asjad Athick</a>	第3章
<a href="#">阿塔福德</a>	第1章
<a href="#">巴克卢克</a>	第67章
<a href="#">巴兰卡</a>	第19、25、31和50章
<a href="#">巴茨</a>	第11、20、50、51、54和63章
<a href="#">本·维斯内斯</a>	第31章
<a href="#">本沃斯</a>	第3、16和25章
<a href="#">巴文·索兰基</a>	第3章
<a href="#">bhrached</a>	第52章
<a href="#">Blag</a>	第37章
<a href="#">CGritton</a>	第10章
<a href="#">ChintaMoney</a>	第38章
<a href="#">Chip</a>	第3章
<a href="#">Chris</a>	第12章
<a href="#">CodeWarrior</a>	第1章和第38章
<a href="#">CPHPython</a>	第6章和第25章
<a href="#">dakab</a>	第2章
<a href="#">达米安·耶里克</a>	第15章
<a href="#">达尔文·冯·科拉克斯</a>	第25章和第30章
<a href="#">迪尼杜·赫瓦格</a>	第10章
<a href="#">迪彭·沙阿</a>	第1章
<a href="#">迪维娅</a>	第24章
<a href="#">德鲁</a>	第2、3、4、7、10、11、12、21、25、29、30、39和46章
<a href="#">e4c5</a>	第24、34、37和52章
<a href="#">尤金</a>	第38、56和62章
<a href="#">falsefive</a>	第60章
<a href="#">菲利普·马丁斯</a>	第14章

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content,  
more changes can be sent to [web@petercv.com](mailto:web@petercv.com) for new content to be published or updated

<a href="#">0x49D1</a>	Chapter 10
<a href="#">4thfloorstudios</a>	Chapter 12
<a href="#">a coder</a>	Chapter 62
<a href="#">AJ</a>	Chapters 21, 24 and 30
<a href="#">Abubakkar</a>	Chapter 20
<a href="#">Adam</a>	Chapter 14
<a href="#">agold</a>	Chapter 50
<a href="#">Alex Recarey</a>	Chapter 31
<a href="#">alex9311</a>	Chapter 25
<a href="#">Alvaro Flaño Larrondo</a>	Chapter 6
<a href="#">Aman Dhanda</a>	Chapter 1
<a href="#">Aminadav</a>	Chapter 63
<a href="#">Andy</a>	Chapter 1
<a href="#">andygeers</a>	Chapter 25
<a href="#">Ani Menon</a>	Chapters 3 and 53
<a href="#">animuson</a>	Chapter 6
<a href="#">aries12</a>	Chapter 52
<a href="#">arushi</a>	Chapter 68
<a href="#">Aryo</a>	Chapter 25
<a href="#">Asaph</a>	Chapters 50 and 52
<a href="#">Asjad Athick</a>	Chapter 3
<a href="#">Athafoud</a>	Chapter 1
<a href="#">BacLuc</a>	Chapter 67
<a href="#">Barranka</a>	Chapters 19, 25, 31 and 50
<a href="#">Batsu</a>	Chapters 11, 20, 50, 51, 54 and 63
<a href="#">Ben Visness</a>	Chapter 31
<a href="#">Benvorth</a>	Chapters 3, 16 and 25
<a href="#">Bhavin Solanki</a>	Chapter 3
<a href="#">bhrached</a>	Chapter 52
<a href="#">Blag</a>	Chapter 37
<a href="#">CGritton</a>	Chapter 10
<a href="#">ChintaMoney</a>	Chapter 38
<a href="#">Chip</a>	Chapter 3
<a href="#">Chris</a>	Chapter 12
<a href="#">CodeWarrior</a>	Chapters 1 and 38
<a href="#">CPHPython</a>	Chapters 6 and 25
<a href="#">dakab</a>	Chapter 2
<a href="#">Damian Yerrick</a>	Chapter 15
<a href="#">Darwin von Corax</a>	Chapters 25 and 30
<a href="#">Dinidu Hewage</a>	Chapter 10
<a href="#">Dipen Shah</a>	Chapter 1
<a href="#">Divya</a>	Chapter 24
<a href="#">Drew</a>	Chapters 2, 3, 4, 7, 10, 11, 12, 21, 25, 29, 30, 39 and 46
<a href="#">e4c5</a>	Chapters 24, 34, 37 and 52
<a href="#">Eugene</a>	Chapters 38, 56 and 62
<a href="#">falsefive</a>	Chapter 60
<a href="#">Filipe Martins</a>	Chapter 14



<a href="#">弗洛里安·根泽尔</a>	第13章和第36章
<a href="#">FMashiro</a>	第17章和第61章
<a href="#">福尔格斯R</a>	第11章
<a href="#">gabe3886</a>	第10章和第11章
<a href="#">哈迪克·坎贾里亚 ツ</a>	第63章和第65章
<a href="#">HCarrasko</a>	第25章
<a href="#">Horen</a>	第10章
<a href="#">雨果·巴夫</a>	第10章
<a href="#">伊恩·格雷戈里</a>	第6章
<a href="#">伊恩·肯尼</a>	第10章
<a href="#">jan_kiran</a>	第38章
<a href="#">杰伊·里佐</a>	第1章
<a href="#">让·维托尔</a>	第25章
<a href="#">约翰·M</a>	第25章
<a href="#">约翰·L·贝文</a>	第26章
<a href="#">尤尔根·D</a>	第1、2、20、29、30和70章
<a href="#">卡伦吉</a>	第50章
<a href="#">卡尔蒂克·坎纳普尔</a>	第3章
<a href="#">库拉姆</a>	第11和12章
<a href="#">科卢纳尔</a>	第8、20、26、30、31和33章
<a href="#">克鲁蒂·帕特尔</a>	第3章
<a href="#">拉希鲁</a>	第66章
<a href="#">利乔</a>	第14章
<a href="#">刘研刘研</a>	第26章
<a href="#">llanato</a>	第30章
<a href="#">卢卡斯·保利洛</a>	第29章
<a href="#">马吉德</a>	第48章
<a href="#">玛丽娜·K.</a>	第24章
<a href="#">马塔斯·瓦伊特凯维丘斯</a>	第26、36和37章
<a href="#">马特·S</a>	第6和21章
<a href="#">马修·惠特</a>	第18章
<a href="#">马修</a>	第50章
<a href="#">matthiasunt</a>	第16章
<a href="#">mayojava</a>	第26章
<a href="#">Md. Nahiduzzaman Rose</a>	第1章
<a href="#">MohaMad</a>	第2章、第34章和第35章
<a href="#">molavec</a>	第41章
<a href="#">内特·沃恩</a>	第57章
<a href="#">内森尼尔·福特</a>	第54章
<a href="#">尼基塔·库尔廷</a>	第24章
<a href="#">诺亚·范德阿</a>	第27章
<a href="#">O. 琼斯</a>	第1、3、15、19、21、22、29、32和59章
<a href="#">优化的比卡什</a>	第16章
<a href="#">熊猫</a>	第1和25章
<a href="#">帕尔斯·帕特尔</a>	第1、11、25和27章
<a href="#">帕尔萨森</a>	第66章
<a href="#">菲利普</a>	第10和16章
<a href="#">庞纳拉苏</a>	第3、7、11、12、24、28、29、44、45、49、50和55章
<a href="#">R.K123</a>	第7、24和27章
<a href="#">拉面厨师</a>	第50章
<a href="#">棘轮</a>	第24和37章
<a href="#">理性开发者</a>	第36章
<a href="#">雷内</a>	第39章

<a href="#">Florian Genser</a>	Chapters 13 and 36
<a href="#">FMashiro</a>	Chapters 17 and 61
<a href="#">ForguesR</a>	Chapter 11
<a href="#">gabe3886</a>	Chapters 10 and 11
<a href="#">Hardik Kanjariya ツ</a>	Chapters 63 and 65
<a href="#">HCarrasko</a>	Chapter 25
<a href="#">Horen</a>	Chapter 10
<a href="#">Hugo Buff</a>	Chapter 10
<a href="#">Ian Gregory</a>	Chapter 6
<a href="#">Ian Kenney</a>	Chapter 10
<a href="#">jan_kiran</a>	Chapter 38
<a href="#">JayRizzo</a>	Chapter 1
<a href="#">Jean Vitor</a>	Chapter 25
<a href="#">John M</a>	Chapter 25
<a href="#">JohnLBevan</a>	Chapter 26
<a href="#">juergen d</a>	Chapters 1, 2, 20, 29, 30 and 70
<a href="#">KalenGi</a>	Chapter 50
<a href="#">KartikKannapur</a>	Chapter 3
<a href="#">Khurram</a>	Chapters 11 and 12
<a href="#">kolunar</a>	Chapters 8, 20, 26, 30, 31 and 33
<a href="#">Kruti Patel</a>	Chapter 3
<a href="#">Lahiru</a>	Chapter 66
<a href="#">Lijo</a>	Chapter 14
<a href="#">LiuYan 刘研</a>	Chapter 26
<a href="#">llanato</a>	Chapter 30
<a href="#">Lucas Paolillo</a>	Chapter 29
<a href="#">Majid</a>	Chapter 48
<a href="#">Marina K.</a>	Chapter 24
<a href="#">Matas Vaitkevicius</a>	Chapters 26, 36 and 37
<a href="#">Matt S</a>	Chapters 6 and 21
<a href="#">Matthew Whitt</a>	Chapter 18
<a href="#">Matthew</a>	Chapter 50
<a href="#">matthiasunt</a>	Chapter 16
<a href="#">mayojava</a>	Chapter 26
<a href="#">Md. Nahiduzzaman Rose</a>	Chapter 1
<a href="#">MohaMad</a>	Chapters 2, 34 and 35
<a href="#">molavec</a>	Chapter 41
<a href="#">Nate Vaughan</a>	Chapter 57
<a href="#">Nathaniel Ford</a>	Chapter 54
<a href="#">Nikita Kurtin</a>	Chapter 24
<a href="#">Noah van der Aa</a>	Chapter 27
<a href="#">O. Jones</a>	Chapters 1, 3, 15, 19, 21, 22, 29, 32 and 59
<a href="#">Optimised Bikash</a>	Chapter 16
<a href="#">Panda</a>	Chapters 1 and 25
<a href="#">Parth Patel</a>	Chapters 1, 11, 25 and 27
<a href="#">ParthaSen</a>	Chapter 66
<a href="#">Philipp</a>	Chapters 10 and 16
<a href="#">Ponnarasu</a>	Chapters 3, 7, 11, 12, 24, 28, 29, 44, 45, 49, 50 and 55
<a href="#">R.K123</a>	Chapters 7, 24 and 27
<a href="#">RamenChef</a>	Chapter 50
<a href="#">ratchet</a>	Chapters 24 and 37
<a href="#">RationalDev</a>	Chapter 36
<a href="#">rene</a>	Chapter 39


<a href="#">理查德·汉密尔顿</a>	第2章和第19章
<a href="#">里克</a>	第16章、第25章和第42章
<a href="#">里克·詹姆斯</a>	第2、3、5、6、7、9、10、11、13、14、16、18、19、20、21、24、25、26、27、28、29、30、31、33、36、38、40、42、43、44、45、46、47、48、50、52、53、54、55、58、64、65、68和69章
<a href="#">理穗</a>	第10章和第18章
<a href="#">罗德里戈·达尔蒂·达·科斯塔</a>	第68章
<a href="#">罗曼·文森特</a>	第1章
<a href="#">萨罗杰·萨斯马尔</a>	第25章
<a href="#">SeeuD1</a>	第3章
<a href="#">Sevle</a>	第12章
<a href="#">skytreader</a>	第25章
<a href="#">斯特凡·罗金</a>	第25章
<a href="#">史蒂夫·钱伯斯</a>	第26章
<a href="#">still_learning</a>	第7章和第50章
<a href="#">strangeqargo</a>	第10章和第11章
<a href="#">草莓</a>	第25章
<a href="#">苏米特·古普塔</a>	第5章和第21章
<a href="#">SuperDJ</a>	第4章
<a href="#">塔里克</a>	第18章
<a href="#">苏塔·昂</a>	第14章和第26章
<a href="#">蒂莫西</a>	第25章
<a href="#">图沙尔·帕特尔</a>	第27章
<a href="#">user2314737</a>	第23章和第70章
<a href="#">user3617558</a>	第16章
<a href="#">user5389107</a>	第1章和第10章
<a href="#">user6655061</a>	第8章
<a href="#">userlond</a>	第30章
<a href="#">vijeeshin</a>	第28章
<a href="#">维克多</a>	第65章
<a href="#">WAF</a>	第9、14、20、24、26、37、50和63章
<a href="#">王恩正</a>	第18章
<a href="#">文中</a>	第56章
<a href="#">whrrgarbl</a>	第14章
<a href="#">冬季</a>	第33章
<a href="#">YCF_L</a>	第1、3、9、23、25、26、27和37章
<a href="#">超立方体</a>	第11章
<a href="#">尤科夫</a>	第44章
<a href="#">尤里·费多罗夫</a>	第25章
<a href="#">齐柏林飞艇</a>	第10章

<a href="#">Richard Hamilton</a>	Chapters 2 and 19
<a href="#">Rick</a>	Chapters 16, 25 and 42
<a href="#">Rick James</a>	Chapters 2, 3, 5, 6, 7, 9, 10, 11, 13, 14, 16, 18, 19, 20, 21, 24, 25, 26, 27, 28, 29, 30, 31, 33, 36, 38, 40, 42, 43, 44, 45, 46, 47, 48, 50, 52, 53, 54, 55, 58, 64, 65, 68 and 69
<a href="#">Riho</a>	Chapters 10 and 18
<a href="#">Rodrigo Darti da Costa</a>	Chapter 68
<a href="#">Romain Vincent</a>	Chapter 1
<a href="#">Saroj Sasmal</a>	Chapter 25
<a href="#">SeeuD1</a>	Chapter 3
<a href="#">Sevle</a>	Chapter 12
<a href="#">skytreader</a>	Chapter 25
<a href="#">Stefan Rogin</a>	Chapter 25
<a href="#">Steve Chambers</a>	Chapter 26
<a href="#">still_learning</a>	Chapters 7 and 50
<a href="#">strangeqargo</a>	Chapters 10 and 11
<a href="#">Strawberry</a>	Chapter 25
<a href="#">Sumit Gupta</a>	Chapters 5 and 21
<a href="#">SuperDJ</a>	Chapter 4
<a href="#">Tarik</a>	Chapter 18
<a href="#">Thuta Aung</a>	Chapters 14 and 26
<a href="#">Timothy</a>	Chapter 25
<a href="#">Tushar patel</a>	Chapter 27
<a href="#">user2314737</a>	Chapters 23 and 70
<a href="#">user3617558</a>	Chapter 16
<a href="#">user5389107</a>	Chapters 1 and 10
<a href="#">user6655061</a>	Chapter 8
<a href="#">userlond</a>	Chapter 30
<a href="#">vijeeshin</a>	Chapter 28
<a href="#">Viktor</a>	Chapter 65
<a href="#">WAF</a>	Chapters 9, 14, 20, 24, 26, 37, 50 and 63
<a href="#">wangengzheng</a>	Chapter 18
<a href="#">Wenzhong</a>	Chapter 56
<a href="#">whrrgarbl</a>	Chapter 14
<a href="#">winter</a>	Chapter 33
<a href="#">YCF_L</a>	Chapters 1, 3, 9, 23, 25, 26, 27 and 37
<a href="#">ypercube□□</a>	Chapter 11
<a href="#">yukoff</a>	Chapter 44
<a href="#">Yury Fedorov</a>	Chapter 25
<a href="#">zeppelin</a>	Chapter 10

你可能也喜欢

### CSS

Notes for Professionals



200+ pages  
of professional hints and tricks

GoalKicker.com  
Free Programming Books

### Linux

Notes for Professionals



50+ pages  
of professional hints and tricks

GoalKicker.com  
Free Programming Books

### HTML5

Notes for Professionals



100+ pages  
of professional hints and tricks

GoalKicker.com  
Free Programming Books

You may also like

### CSS

Notes for Professionals



200+ pages  
of professional hints and tricks

GoalKicker.com  
Free Programming Books

### Linux

Notes for Professionals



50+ pages  
of professional hints and tricks

GoalKicker.com  
Free Programming Books

### HTML5

Notes for Professionals



100+ pages  
of professional hints and tricks

GoalKicker.com  
Free Programming Books

### Microsoft SQL Server

Notes for Professionals



200+ pages  
of professional hints and tricks

GoalKicker.com  
Free Programming Books

### MongoDB

Notes for Professionals



60+ pages  
of professional hints and tricks

GoalKicker.com  
Free Programming Books

### Oracle Database

Notes for Professionals



100+ pages  
of professional hints and tricks

GoalKicker.com  
Free Programming Books

### Microsoft SQL Server

Notes for Professionals



200+ pages  
of professional hints and tricks

GoalKicker.com  
Free Programming Books

### MongoDB

Notes for Professionals



60+ pages  
of professional hints and tricks

GoalKicker.com  
Free Programming Books

### Oracle Database

Notes for Professionals




100+ pages  
of professional hints and tricks

GoalKicker.com  
Free Programming Books

### PHP

Notes for Professionals



400+ pages  
of professional hints and tricks

GoalKicker.com  
Free Programming Books

### PostgreSQL

Notes for Professionals



60+ pages  
of professional hints and tricks

GoalKicker.com  
Free Programming Books

### SQL

Notes for Professionals



100+ pages  
of professional hints and tricks

GoalKicker.com  
Free Programming Books

### PHP

Notes for Professionals



400+ pages  
of professional hints and tricks

GoalKicker.com  
Free Programming Books

### PostgreSQL

Notes for Professionals



60+ pages  
of professional hints and tricks

GoalKicker.com  
Free Programming Books

### SQL

Notes for Professionals



100+ pages  
of professional hints and tricks

GoalKicker.com  
Free Programming Books