1MySql高级

day1

序号	Day01	Day02	Day03	Day04
1	Linux系统安装MySQL	体系结构	应用优化	MySQL 常用工具
2	索引	存储引擎	查询缓存优化	MySQL 日志
3	视图	优化SQL步骤	内存管理及优化	MySQL 主从复制
4	存储过程和函数	索引使用	MySQL锁问题	综合案例
5	触发器	SQL优化	常用SQL技巧	

在linux(cent os)上安装mysql

```
1、删除linux上的mysql
        1、查询linux上，mysql的版本：rpm -qa | grep -i mysq1
        2、删除对应的mysql：rpm -e (mysq1 -libs-5.1.71-1.e16.x86_64) --nodeps
2、上传mysql安装包到linux：按下alt+t，进入sftp命令下
        put (D:/setup/MySQL-5.6.22-1.e16.1686.rpm-bundle.tar)
3、在lunix目录下，查看上传的安装包(ll)
        解压
            创建目录：mkdir mysql
            解压到指定目录下：tar -xvf (MySQL-5.6.22-1.e16.i686.rpm-bundle.tar) -C
mysq1/

4、安装mysql所需的第三方的依赖：
        yum -y install 1ibaio.so.1 1ibgcc_s.so.1 1ibstdc++.so.6 libncurses.so.5
--setopt=protected_multilib=false
        yum update 1ibstdc++-4.4.7-4.e16.x86_64(更新操作)

5、安装mysql的客户端：rpm -ivh MySQL-client-5.6.22-1.e16.i686.rpm
6、安装mysql的服务端：rpm -ivh MySQL-server-5.6.22-1.e16.i686.rpm


7、启动mysql：
        查看mysql的状态：service mysql status
        启动mysql：service mysql start
        停止mysql：service mysql stop
        查看mysql安装时的随机密码：cat /root/.mysql_secret
        登录：mysql uroot P密码
        更改登录密码：set password password('新密码');
        退出：ctrl-c -- exit!

8、mysql授权远程登录：
        grant a11 privileges on *.* to 'root' @'%' identified by '密码';
        需要刷新一次：flush privileges;
        查看防火墙状态：service iptables status
        关闭防火墙：service iptables stop
```

mysql安装时的随机密码提示

查看mysql安装时的随机密码：

索引

存储引擎

和大多数的数据库不同, MySQL中有-一个存储引擎的概念,针对不同的存储需求可以选择最优的存储引擎。

存储引擎就是存储数据,建立索引,更新查询数据等等技术的实现方式。存储引擎是基于表的,而不是基于库的。所以存储引擎也可被称为表类型。

Oracle , SqlServer等数据库只有一种存储引擎。 MySQL提供了插件式的存储引擎架构。所以MySQL存在多种存储引擎,可以根据需要使用相应引擎,或者编写存储引擎。

MySQL5.0支持的存储引擎包含: InnoDB . MyISAM、 BDB、MEMORY、MERGE、EXAMPLE、 NDB Cluster. ARCHIVE、 CSV、BLACKHOLE、FEDERATED等 ,其中InnoDB和BDB提供事务安全表,其他存储引擎是非事务安全表。

mysql再5.5版本之后(包括)：默认使用的是InnoDB

```
查看mysql版本：select version();
查看所有的存储引擎：show engines（engine：存储引擎的名称；support：支持的存储引擎）
查看mysql默认的存储引擎：show VARIABLES LIKE '%STORAGE_engine%'
```

各种存储引擎特性

常用的存储引擎，并对比各个存储引擎之间的区别；常用InnoDB 、MyISAM； MEMORY 、MERGE了解

特点	InnoDB	MyISAM	MEMORY	MERGE	NDB
存储限制	64TB	有	有	有	有
事务安全	**支持**				
锁机制	**行锁(适合高并发)**	**表锁**	表锁	表锁	行锁
B树索引	支持	支持	支持	支持	支持
哈希索引			支持		
全文索引	支持(5.6版本后)	支持			
集群索引	支持				
数据索引	支持		支持		支持
索引缓存	支持	支持	支持	支持	支持
数据可压缩		支持			
空间使用	高	低	N/A	低	低
内存使用	高	低	中等	低	高
批量插入速度	低	高	高	高	高
支持外键	**支持**				

各种存储引擎的特性

InnoDB存储引擎是Mysql的默认存储引擎。InnoDB存储引擎提供了具有提交、回滚、崩溃恢复能力的事务安全。但是对比MyISAM的存储引擎, InnoDB写的处理效率差一些,并且会占用更多的磁盘空间以保留数据和索引。

InnoDB存储引擎不同于其他存储引擎的特点

```
CREATE TABLE goods_innodb (
  id int NOT NULL AUTO_INCREMENT,
  name varchar(20) NOT NULL,
  PRIMARY KEY (id)
) ENGINE=innodb DEFAULT CHARSET=utf8
```

```
-- 开启事务
START TRANSACTION;

-- 插入数据
insert into goods_innodb (id, name) values (null, 'Meta21');

-- 提交(在mysql中，开启事务，若处于未提交，那么命令是差不到的)
COMMIT;
```

InnoDB

MySQL支持外键的存储引擎只有InnoDB，在创建外键的时候，要求父表必须有对应的索引，子表在创建外键的时候,也会自动的创建对应的索引。

下面两张表中，country_innodb是父表, country_id为主键索引, city_innodb表是子表, country_id字段为外键,对应于country_innodb表的主键country_id

```
-- 创建两张新表
create table country_innodb(
country_id int NOT NULL AUTO_INCREMENT,
country_name varchar(100) NOT NULL,
primary key(country_id)
)ENGINE=InnoDB DEFAULT CHARSET=utf8;


create table city_innodb(
city_id int NOT NULL AUTO_INCREMENT,
city_name varchar(50) NOT NULL,
country_id int NOT NULL,
primary key(city_id),
key idx_fk_country_id(country_id),
CONSTRAINT fk_city_country FOREIGN KEY(country_id) REFERENCES
country_innodb(country_id) ON DELETE
RESTRICT ON UPDATE CASCADE
)ENGINE=InnoDB DEFAULT CHARSET=utf8;


ON DELETE RESTRICT:删除主表数据时，如果有关联记录，不删除
ON UPDATE CASCADE:更新主表时，如果子表有 关联记录，更新字表记录
```

在创建索引时，可以指定在删除、更新父表时,对子表进行的相应操作.包括RESTRICT. CASCADE、SET NULL和NO ACTION。

RESTRICT和NO ACTION相同，是指限制在子表有关联记录的情况下，父表不能更新;

CASCADE表示父表在更新或者删除时,更新或者删除子表对应的记录;

SET NULL则表示父表在更新或者删除的时候,子表的对应字段被SET NULL。

针对.上面创建的两个表，子表的外键指定是ON DELETE RESTRICT ON UPDATE CASCADE方式的，那么在主表删除记录的时候，如果子表有对应记录，则不允许删除，主表在更新记录的时候,如果子表有对应记录，则子表对应更新。

```
ON DELETE RESTRICT:删除主表数据时，如果有关联记录，不删除
ON UPDATE CASCADE:更新主表时，如果子表有 关联记录，更新字表记录

-- 尝试删除有关联的主键数据
DELETE FROM country_innodb WHERE country_id = 2;

Cannot delete or update a parent row: a foreign key constraint fails
(`optimization_test`.`city_innodb`, CONSTRAINT `fk_city_country` FOREIGN KEY
(`country_id`) REFERENCES `country_innodb` (`country_id`) ON UPDATE CASCADE)
翻译:
无法删除或更新父行：外键约束失败（`optimization\utest`.`city_innodb`，约束
`fk_city\ucountry`外键（`country\uid`）引用了update CASCADE上的
country_innodb`（`country_id`）
```

```
-- 尝试更新主表
UPDATE country_innodb SET country_id = 10 WHERE country_id = 1;
```

存储方式

InnoDB存储表和索引|有以下两种方式:

①.使用共享表空间存储，这种方式创建的表的**表结构保存在.frm文件**中，数据和索引保存在innodb_data_ home_ dir 和innodb_data_file_path定义的表空间中,可以是多个文件。（**.ibd文件存储数据**）

②.使用多表空间存储.这种方式创建的表的表结构仍然存在.frm 文件中,但是每个表的数据和索引单独保存在.ibd 中。

MyISAM

MyISAM不支持事务、也不支持外键,其优势是访问的速度快,对事务的完整性没有要求或者以SELECT、INSERT为主的应用基本上都可以使用这个引擎来创建表。有以下两个比较重要的特点

```
-- 创建myisam存储引擎的表
CREATE TABLE goods_myisam (
    id INT NOT NULL AUTO_INCREMENT,
NAME VARCHAR ( 20 ) NOT NULL,
PRIMARY KEY ( id )) ENGINE = myisam DEFAULT CHARSET = utf8;

-- 插入一条数据
INSERT INTO goods_myisam (id, name) VALUES (NULL, 'Meta20');

-- 开启事务
START TRANSACTION;

-- 再插入一条数据
INSERT INTO goods_myisam (id, name) VALUES (NULL, 'Meta21');

-- 查询此表的所有数据
SELECT * FROM goods_myisam;      #未提交，依然可以查询到；因为myisam存储引擎没有事务
```

文件存储方式

每个MyISAM在磁盘上存储成3个文件,其文件名都和表名相同,但拓展名分别是:

.frm (存储表定义):

.MYD(MYData ,存储数据) ;

.MYI(MYIndex ,存储索引) ;

MEMORY（内存）

Memory存储引擎将表的数据存放在内存中。每个MEMORY表实际对应一个磁盘文件 ,格式是**.frm** , 该文件中**只存储表的结构**,而其**数据文件,都是存储在内存中**,这样有利于数据的快速处理,提高整个表的效率。MEMORY类型的表访问非常地快,因为他的数据是存放在内存中的,并且默认使用HASH索引，但是服务一旦关闭 ,表中的数据就会丢失。

MERGE

MERGE存储引擎**是一组MyISAM表的组合** ,这些MyISAM表必须结构完全相同, MERGE表本身并没有存储数据,对MERGE类型的表可以进行查询、更新、删除操作,这些操作实际上是对内部的MyISAM表进行的。

对于MERGE类型表的插入操作,是通过INSERT _METHOD子句定义插入的表,可以有3个不同的值,使用FIRST或LAST值使得插入操作被相应地作用在第一-或者最后一 个表上,不定义这个子句或者定义为NO , 表示不能对这个MERGE表执行插入操作。

可以对MERGE表进行DROP操作,但是这个操作只是删除MERGE表的定义,对内部的表是没有任何影响的。

案例：创建3个测试表payment 2006 , payment _2007 , payment. ,all ,其中payment al是前两个表的MERGE表:

```
-- 创建两张以myisam存储引擎的表
CREATE TABLE order_1990 (
order_id INT,
order_money DOUBLE (10,2),
order_address VARCHAR (50),
PRIMARY KEY (order_id)
) ENGINE = myisam DEFAULT charset = utf8;



CREATE TABLE order_1991 (
order_id INT,
order_money DOUBLE (10,2),
order_address varCHAR (50),
PRIMARY KEY ( order_id )
) ENGINE = myisam DEFAULT charset = utf8;

-- 再创建一张以MERGE存储引擎的表
CREATE TABLE order_all (
order_id INT,
order_money DOUBLE (10, 2),
order_address VARCHAR ( 50 ),
PRIMARY KEY ( order_id )
) ENGINE = MERGE UNION
= ( order_1990, order_1991 ) INSERT_METHOD = LAST DEFAULT charset = utf8;
```

```
-- 分别向myisam存储引擎的表中，插入两条记录（数据结构相同，数据可以不同）
insert into order_1990 values(1,100.0, '北京');
insert into order_1990 values(2,100.0,'上海');

insert into order_1991 values(10,200.0,'北京');
insert into order_1991 values(11,200.0,'上海');


-- 查询三张表中所有的数据
select * from order_1990;
select * from order_1991;
select * from order_all;    #前两张myisam存储引擎的总数据
```

优化SQL步骤

在应用的的开发过程中,由于初期数据量小,开发人员写SQL语句时更重视功能上的实现，但是当应用系统正式上线后,随着生产数据量的急剧增长,很多SQL语句开始逐渐显露出性能问题,对生产的影响也越来越大,此时这些有问题的SQL语句就成为整个系统性能的瓶颈,因此我们必须要对它们进行优化,本章将详细介绍在MySQL中优化SQL语句的方法。

当面对一个有SQL性能问题的数据库时,我们应该从何处入手来进行系统的分析,使得能够尽快定位问题SQL并尽快解决问题。

查看SQL执行频率

MySQL客户端连接成功后,通过show [session|global] status命令可以提供服务器状态信息。show [session |global] status可以根据需要加上参数"session"或者'global"来显示session级(当前连接)的计结果和global级(自数据库上次启动至今)的统计结果。如果不写,默认使用参数是"session"

下面的命令显示了当前session中所有统计参数的值:

```
查询当前按连接使用的信息：SHOW STATUS LIKE 'Com_______'
查询全局的信息：SHOW global STATUS LIKE 'Com_______'
innodb的操作的数量：SHOW global STATUS LIKE 'Innodb_rows_%'
```

定位低效率执行SQL

可以通过以下两种方式定位执行效率较低的SQL语句。

慢查询日志:通过慢查询日志定位那些执行效率较低的SQL语询,用-log slow-queriesl-file_ name]选项启动时, mysqld写一个包含所有执行时间超过long_query_time秒的SQL语句的日志文件。具体可以查看本书第26章中日志管理的相关部分。

show processlist :慢查询日志在查询结束以后才纪录,所以在应用反映执行效率出现问题的时候查询慢查询日志并不能定位问题,可以使用show processlist命令查看当前MySQL在进行的线程,包括线程的状态、是否锁表等.可以实时地查看SQL的执行情况,同时对一些锁表操作进行优化。

```
SHOW PROCESSLIST
#    time：此状态的持续的时长(秒)
#    status：当前连接sql语句的状态
```

1、id列,用户登录mysql时,系统分配的"connection_id" ,可以使用函数connection_id()查看

2、user列,显示当前用户.如果不是root ,这个命令就只显示用户权限范围的sql语句

3、host列,显示这个语句是从哪个ip的哪个端口上发的,可以用来跟踪出现问题语句的用户

4、db列，显示这个进程目前连接的是哪个数据库

5、command列,显示当前连接的执行的命令,一般取值为休眠(sleep) ,查询(query) ， 连接(connect)等

6、time列,显示这个状态持续的时间,单位是秒

7、state列，显示使用当前连接的sql语句的状态,很重要的列。stat e描述的是语句执行中的某一个状态。一个sql语句,以查询为例,可能需要经过copying to tmp tab1e. sorting resu1t. sending data等状态才可以完成

索引

索引的概述

MySQL官方对索引的定义为：索引（index）是**帮助MySQL高效获取数据的数据结构**（有序）。在数据之外，数据库系统还维护者满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据， 这样就可以在这些数据结构上实现高级查找算法，这种数据结构就是索引。

左边是数据表，一共有两列七条记录，最左边的是数据记录的物理地址（注意逻辑上相邻的记录在磁盘上也并不是一定物理相邻的）。为了加快Col2的查找，可以维护一个右边所示的二叉查找树，每个节点分别包含索引键值和一个指向对应数据记录物理地址的指针，这样就可以运用二叉查找快速获取到相应数据。

一般来说索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储在磁盘上。索引是数据库中用来提高性能的最常用的工具。

索引的优势和劣势

优势

1） 类似于书籍的目录索引，提高数据检索的效率，降低数据库的IO成本。

2） 通过索引列对数据进行排序，降低数据排序的成本，降低CPU的消耗。

劣势

1） 实际上索引也是一张表，该表中保存了主键与索引字段，并指向实体类的记录，所以索引列也是要占用空间的。

2） 虽然索引大大提高了查询效率，同时却也降低更新表的速度，如对表进行INSERT、UPDATE、DELETE。因为更新表时，MySQL 不仅要保存数据，还要保存一下索引文件每次更新添加了索引列的字段，都会调整因为更新所带来的键值变化后的索引信息。

索引结构

索引是在MySQL的存储引擎层中实现的，而不是在服务器层实现的。所以每种存储引擎的索引都不一定完全相同，也不是所有的存储引擎都支持所有的索引类型的。MySQL目前提供了以下4种索引：

- BTREE 索引： 最常见的索引类型，大部分索引都支持 B 树索引。
- HASH 索引：只有Memory引擎支持 ， 使用场景简单 。
- R-tree 索引（空间索引）： 空间索引是MyISAM引擎的一个特殊索引类型，主要用于地理空间数据类型，通常使用较少，不做特别介绍。
- Full-text （全文索引） ： 全文索引也是MyISAM的一个特殊索引类型，主要用于全文索引，InnoDB从Mysql5.6版本开始支持全文索引。

MyISAM、InnoDB、Memory三种存储引擎对各种索引类型的支持

索引	InnoDB引擎	MyISAM引擎	Memory引擎
BTREE索引	支持	支持	支持
HASH 索引	不支持	不支持	支持
R-tree 索引	不支持	支持	不支持
Full-text	5.6版本之后支持	支持	不支持

我们平常所说的索引，如果没有特别指明，都是指B+树（多路搜索树，并不一定是二叉的）结构组织的索引。其中聚集索引、复合索引、前缀索引、唯一索引默认都是使用 B+tree 索引，统称为 索引。

BTREE 结构

BTree又叫多路平衡搜索树，一颗m叉的BTree特性如下：

- 树中每个节点最多包含m个孩子。
- 除根节点与叶子节点外，每个节点至少有[ceil(m/2)]个孩子。
- 若根节点不是叶子节点，则至少有两个孩子。
- 所有的叶子节点都在同一层。
- 每个非叶子节点由n个key与n+1个指针组成，其中[ceil(m/2)-1] <= n <= m-1

以5叉BTree为例，key的数量：公式推导[ceil(m/2)-1] <= n <= m-1。所以 2 <= n <=4 。当n>4时，中间节点分裂到父节点，两边节点分裂。

插入 C N G A H E K Q M F W L T Z D P R X Y S 数据为例。

到此，该BTREE树就已经构建完成了， BTREE树 和 二叉树 相比， 查询数据的效率更高， 因为对于相同的数据量 来说，BTREE的层级结构比二叉树小，因此搜索速度快。

B+TREE结构

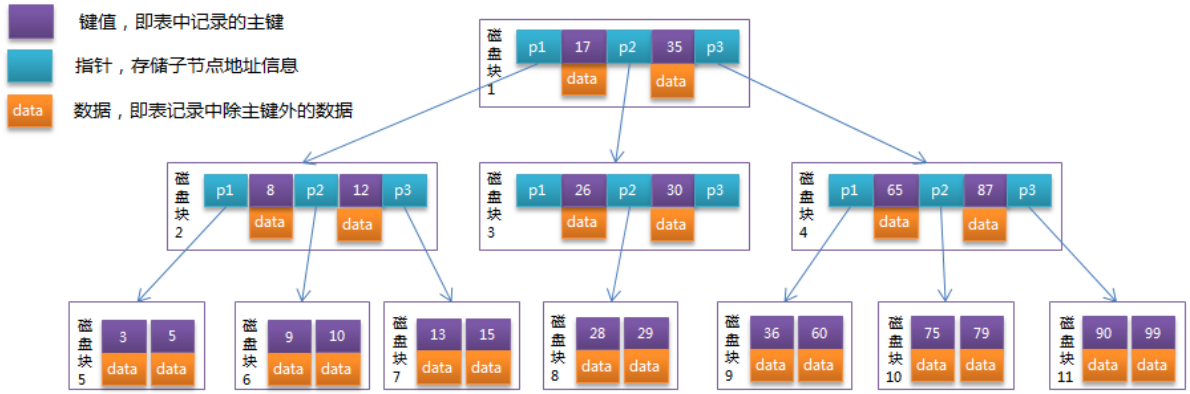
B+Tree为BTree的变种, B+Tree与BTree的区别为:

1. n叉B+Tree最多含有n个key ,而BTree最多含有n-1个key
2. B+Tree的叶子节点保存所有的key信息,依key大小顺序排列。
3. 所有的非叶子节点都可以看作是key的索引部分。

MySQL中的B+Tree

MySq|索引数据结构对经典的B+ Tree进行了优化。在原B+ Tree的基础上,增加一个指向相邻叶子节点的链表指针,就形成了带有顺序指针的B+Tree ,提高区间访问的性能。

MySQL中的B+Tree索引结构示意图:



存储引擎的选择

在选择存储引擎时,应该根据应用系统的特点选择合适的存储引擎。对于复杂的应用系统,还可以根据实际情况选择多种存储引擎进行组合。以下是几种常用的存储引擎的使用环境。

- InnoDB :是Mysq|的默认存储引擎,用于事务处理应用程序,支持外键。如果应用对事务的完整性有比较高的要求,在并发条件下要求数据的一致性,数据操作除了插入和查询意外,还包含很多的更新、删除操作,那么InnoDB存储引擎是比较合适的选择。InnoDB存储引擎除了有效的降低由于删除和更新导致的锁定.还可以确保事务的完整提交和回滚.对于类似于计费系统或者财务系统等对数据准确性要求比较高的系统，InnoDB是最合适的选择。
- MyISAM :如果应用是以读操作和插入操作为主,只有很少的更新和删除操作,并且对事务的完整性、并发性要求不是很高,那么选择这个存储引擎是非常合适的。
- MEMORY :将所有数据保存在RAM中,在需要快速定位记录和其他类似数据环境下,可以提供几块的访问。MEMORY的缺陷就是对表的大小有限制,太大的表无法缓存在内存中.其次是要确保表的数据可以恢复,数据库异常终止后表中的数据是可以恢复的。MEMORY表通常用于更新不太频繁的小表,用以快速得到访问结果。
- MERGE :用于将一系列等同的MyISAM表以逻辑方式组合在一 起,并作为-一个对象引用他们。MERGE表的优点在于可以突破对单个MyISAM表的大小限制,并且通过将不同的表分布在多个磁盘上,可以有效的改善MERGE表的访问效率。这对于存储诸如数据仓储等VLDB环境十分合

索引分类

单值(单列)索引：即一个索引只包含单个列,一个表可以有多个单列索引

唯一索引:索引列的值必须唯一 ,但允许有空值

复合索引: 即一个索引包含多个列

索引语法

索引在创建表的时候,可以同时创建，也可以随时增加新的索引。

准备环境

```
-- 创建数据库
create database demo_01 default charset=utf8mb4;

-- 创建两张表结构
CREATE TABLE city(
city_id int(11) NOT NULL AUTO_INCREMENT,
city_name varchar(50) NOT NULL,
country_id INT ( 11 ) NOT NULL,
PRIMARY KEY ( city_id )
) ENGINE = INNODB DEFAULT CHARSET = utf8;

CREATE TABLE country (
country_id int(11) NOT NULL AUTO_INCREMENT,
country_name varchar(100) NOT NULL,
```

```
PRIMARY KEY (country_id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

-- 为两张表增加一些数据
insert into `city` (`city_id`, `city_name`, `country_id`) values(1,'西安',1);
insert into `city` (`city_id`, `city_name`, `country_id`) values(2,'NewYork',2);
insert into `city` (`city_id`, `city_name`, `country_id`) values(3,'北京',1);
insert into `city` (`city_id`, `city_name`, `country_id`) values(4,'上海',1);

insert into `country` (`country_id`, `country_name`) values(1,'China');
insert into `country` (`country_id`, `country_name`) values(2,'America');
insert into `country` (`country_id`, `country_name`) values(3,'Japan');
insert into `country` (`country_id`, `country_name`) values(4,'UK');
```

创建索引

语法：　（主键，默认有主键索引）

```
#   create  (可指定索引类型,也可以不指定)唯一|全文|空间    index   索引的名称
#   USING   索引的类型(指定索引类型，默认B+树)
#   ON  表名  (字段名.....)

CREATE  [UNIQUE|FULLTEXT|SPATIAL]  INDEX index_name
[USING  index_type]
ON tbl_name(index_col_name,...)

-- 例如，为city_name列创建索引
CREATE INDEX idx_city_name on city(city_name)
```

查看索引

```
#   show index from table_name;
#   查看city表中的所有索引
SHOW INDEX FROM city
每个索引以***隔开(编译器软件内不可用)：SHOW INDEX FROM city\G
```

删除索引

```
#           索引名称        表名
DROP INDEX index_name ON tbl_name;
-- 删除city表的idx_city_name的索引
DROP INDEX idx_city_name ON city;
```

ALTER命令

通过alter命令，修改表，建立索引

```
#    该语句添加一个主键，这意味着索引值必须是唯一的，且不能为NULL
添加主键索引：alter  table  tb_name  add  primary  key(column_list);

#    这条语句创建索引的值必须是唯一的（除了NULL外，NULL可能会出现多次），括号内是字段列表(列名)
添加一个唯一索引：alter  table  tb_name  add  unique index_name(column_list);

#    添加普通索引，      索引值可以出现多次。
alter  table  tb_name  add  index index_name(column_list);

#    该语句指定了索引为FULLTEXT，      用于全文索引
alter  table  tb_name  add  fulltext  index_name(column_list);
```

索引设计原则

索引的设计可以遵循一些已有的原则，创建索引的时候请尽量考虑符合这些原则，便于提升索引的使用效率，更高 效的使用索引。

- **对查询频次较高，且数据量比较大**的表建立索引。
- **索引字段的选择**，最佳候选列应当从**where子句的条件中提取**，如果where子句中的组合比较多，那么应当挑 选最常用、过滤效果最好的列的组合。
- 使用唯一索引，区分度越高，使用索引的效率越高。
- 索引可以有效的提升查询数据的效率，但索引数量不是多多益善，索引越多，维护索引的代价自然也就水涨 船高。对于插入、更新、删除等DML操作比较频繁的表来说，索引过多，会引入相当高的维护代价，降低 DML操作的效率，增加相应操作的时间消耗。另外索引过多的话，MySQL也会犯选择困难病，虽然最终仍然 会找到一个可用的索引，但无疑提高了选择的代价。
- **使用短索引**，索引创建之后也是使用硬盘来存储的，因此提升索引访问的I/O效率，也可以提升总体的访问效 率。假如构成索引的字段总长度比较短，那么在给定大小的存储块内可以存储更多的索引值，相应的可以有 效的提升MySQL访问索引的I/O效率。
- 利用最左前缀，N个列组合而成的组合索引，那么相当于是创建了N个索引，如果查询时where子句中使用了 组成该索引的前几个字段，那么这条查询SQL可以利用组合索引来提升查询效率。

```
-- 创建复合索引
--                索引名称                    表名            表名中所需添加索引的字段
CREATE INDEX idx_name_email_status ON tb_ se11 er (NAME, email, STATUS);

就相当于 (包含了第一个字段name)
对name 创建索引;
对name , email 创建了索引;
对name , email, status 创建了索引;
```

视图

视图的概述

视图（View）是一种虚拟存在的表。视图并不在数据库中实际存在，行和列数据来自定义视图的查询中使用的表， 并且是在使用视图时动态生成的。通俗的讲，**视图就是一条SELECT语句执行后返回的结果集**。所以我们在创建视 图的时候，主要的工作就落在创建这条SQL查询语句上。

视图相对于普通的表的优势主要包括以下几项。

- 简单：使用视图的用户完全不需要关心后面对应的表的结构、关联条件和筛选条件，对用户来说已经是过滤 好的复合条件的结果集。
- 安全：使用视图的用户只能访问他们被允许查询的结果集，对表的权限管理并不能限制到某个行某个列，但 是通过视图就可以简单的实现。
- 数据独立：一旦视图的结构确定了，可以屏蔽表结构变化对用户的影响，源表增加列对视图没有影响；源表 修改列名，则可以通过修改视图来解决，不会造成对访问者的影响。

创建视图和修改视图

创建视图的语法

```
CREATE [OR REPLACE] [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}] # REPLACE替换
VIEW view_name [(column_list)]
AS select_statement          #   AS 后是查询语句
[WITH [CASCADED | LOCAL] CHECK OPTION]

-- 创建视图案例
--                视图名称              as后跟查询语句；此处是外连接
CREATE VIEW view_city_country AS SELECT c.*, t.country_name FROM city c, country
t WHERE c.country_id = t.country_id
```

修改视图的语法

```
ALTER [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
VIEW view_name [(column_list)]
AS select_statement
[WITH [CASCADED | LOCAL] CHECK OPTION]

选项:
WITH [CASCADED | LOCAL] CHECK OPTION 决定了是否允许更新数据使记录不再满足视图的条件。
LOCAL :      只要满足本视图的条件就可以更新。
CASCADED :      必须满足所有针对该视图的所有视图的条件才可以更新。      默认值
```

更新视图

更新的是，基础表的信息

```
#        视图名                    字段名          新的值          条件
UPDATE view_city_country SET city_name = '西安市' WHERE city_id = 1
```

查看视图

```
-- 查询指定视图名称
SELECT * FROM view_city_country

show tables;         -- mysql5.1之后，包含了视图层
SHOW CREATE VIEW view_city_country  --  查看创建视图时的语句
```

删除视图

```
DROP VIEW [IF EXISTS] view_name [, view_name] ...[RESTRICT | CASCADE]

#例如:
DROP VIEW IF EXISTS VIEW city_country_view;
```

存储过程和函数

存储过程和函数概述

存储过程和函数是事先经过编译并存储在数据库中的一段SQL语句的集合,调用存储过程和函数可以简化应用开发人员的很多工作,减少数据在数据库和应用服务器之间的传输,对于提高数据处理的效率是有好处的。(就是执行sql语句的集合)

存储过程和函数的区别在于函数必须有返回值,而存储过程没有。

函数:是一个有返回值的过程;

过程:是一个没有返回值的函数;

创建存储过程

```
create procedure procedure_name([proc_parameter[...]])
begin
    -- SQL语句
end;
```

例如:

```
create procedure pro_test1()
begin
    select 'hello MySQL';   # 此处不加;会报异常
end
```

更换mysql的分隔符为$:delimiter $

该关键字用来声明SQL语句的分隔符,告诉MySQL解释器,该段命令是否已经结束了, mysql是否可以执行了。默认情况下, delimiter是号。在命令行客户端中,如果有一行命令以分号结束 ,那么回车后, mysql将会执行该命令。

调用存储过程

```
call pro_test1();
```

查看存储过程

```
-- 查询db_ name数据库中的所有的存储过程         数据库名称
select name from mysql.proc where db = 'db_name';

-- 查询存储过程的状态信息(名称)；加上/G，每个存储过程以****隔开
show procedure status;

-- 查询某个存储过程的定义(语法)  数据库名称.存储过程名称
show create procedure test.pro_test1 \G
```

删除存储过程

```
--              存储过程名称
drop procedure pro_test1;
```

存储过程的语法结构

存储过程是可以编程的,意味着可以使用变量,表达式,控制结构，来完成比较复杂的功能。

变量

declare

通过DECLARE可以定义一个局部变量,该变量的作用范围只能在BEGIN..END块中。

查询存储过程依然是用：call 存储过程名称;

调用存储过程时，会执行书写时的sql语句，也就是说select...into方式，会根据表的总数更新

```
--              可以是多个变量
declare var_name[,...] type [default value]
```

例如：

```
--                  存储过程的名称
CREATE PROCEDURE pro_test1()
BEGIN
--      变量名 类型       默认值
DECLARE num int DEFAULT 10;
-- 查看创建的num的值
SELECT CONCAT('num的值:',num);
END
```

set

直接赋值使用SET ,可以赋常量或者赋表达式,具体语法如下

```
--  变量名      值；有多个以,隔开
set var_name = expr [, var_name = expr] ...
```

例如：

```
CREATE PROCEDURE pro_test2()
BEGIN
DECLARE num int DEFAULT 0;
SET num = num + 10;
SELECT num;
END
```

也可以通过select...into方式进行赋值

```
CREATE PROCEDURE pro_test3()
BEGIN
--           不给默认值
DECLARE num int;
--       将查询到的总条数，放入num变量中
SELECT COUNT(*) INTO num FROM city;
SELECT CONCAT('city表中的记录数：', num);
END
```

---

if条件判断

语法结构

```
--  条件                成立   执行sql语句
if search_condittion then statement_list
--           满足此条件         成立       执行此sql语句
    [elseif search_condition then statement_list] ...
--  否则   执行此sql语句
    [else statement_list]
end if;
```

需求

```
根据定义的身高变量，判定当前身高的身材类似
180及以上------------>身材高挑
170 - 180 ---------->标准身材
170以下 ------------->一般身材
```

代码块：**注意**：代码的结束符;不能少

```
-- 创建存储过程为  pro_test4
CREATE PROCEDURE pro_test4()
BEGIN
-- 设置身高的类型，及默认值
    DECLARE height int DEFAULT 175;
```

```
-- 设置身高的描述，类型及默认值
    DECLARE description VARCHAR(50) DEFAULT '';
-- 身高大于等于180，将描述存储到，描述的变量中
    IF height >= 180 THEN
        SET description = '身材高挑';
-- 身高大于等于170，并且小于180时，将描述存储到，描述的变量中
    ELSEIF height >= 170 AND height < 180 THEN
        SET description = '标准身材';
-- 前面的条件都不成立，将描述存储到，描述的变量中
    ELSE
        SET description = '一般身材';
-- 结束if判断
    END IF;
-- 输出身高的变量值，和对于的描述
    SELECT CONCAT('身高：', height, '----->', description);
-- 结束存储过程的创建
END
```

参数传递

语法格式：

```
--               存储过程名称
create procedure procedure_name([in/out/inout] 参数名 参数类型)
...

in：该参数可以作为输入，也就是需要调用方传入值，默认
out：该参数作为输出,也就是该参数可以作为返回值
inout：既可以作为输入参数,也可以作为输出参数
```

in - 输入

需求：

```
根据传递的身高变量，判定当前身高的所属身材类型
```

代码块：

调用此带参数的存储过程时，(参数)：CALL pro_test5(198);

```
-- 创建存储过程为  pro_test5
CREATE PROCEDURE pro_test5(IN height INT)
BEGIN
-- 设置身高的描述，类型及默认值
    DECLARE description VARCHAR(50) DEFAULT '';
-- 身高大于等于180，将描述存储到，描述的变量中
    IF height >= 180 THEN
        SET description = '身材高挑';
-- 身高大于等于170，并且小于180时，将描述存储到，描述的变量中
    ELSEIF height >= 170 AND height < 180 THEN
        SET description = '标准身材';
-- 前面的条件都不成立，将描述存储到，描述的变量中
    ELSE
```

```
        SET description = '一般身材';
-- 结束if判断
    END IF;
-- 输出身高的变量值，和对于的描述
    SELECT CONCAT('身高：', height, '---->', description);
-- 结束存储过程的创建
END
```

out - 输出

需求：

> 根据传入的身高变量，获取当前身高的所属的身材类型

代码块：

调用：

CALL pro_test6(188, @description);　　两个参数，身高，@需要输出的变量名
　　SELECT @description　　查询@变量名，得到输出的值

```
-- 创建存储过程pro_test6  (输入参数 height int类型，输出参数 description 长度为100的字符串类型)
CREATE PROCEDURE pro_test6(IN height INT, OUT description VARCHAR(100))
BEGIN
-- 身高大于等于180，将描述存储到，描述的变量中
    IF height >= 180 THEN
    SET description = '身材高挑';
    -- 身高大于等于170，并且小于180时，将描述存储到，描述的变量中
    ELSEIF height >= 170 AND height < 180 THEN
        SET description = '标准身材';
-- 前面的条件都不成立，将描述存储到，描述的变量中
    ELSE
        SET description = '一般身材';
-- 结束if判断
    END IF;
-- 结束存储过程的创建
END
```

补充

@description:这种变量要在变量名称前面加上”@"符号,叫做用户会话变量,代表整个会话过程他都是有作用的,这个类似于全局变量一样。

@@global.sort_buffer_size :这种在变量前加上"@@"符号,叫做系统变量

case结构

```
#方式一：
--      (条件)值
CASE case_value
--  等于       (条件)值
    WHEN when_value THEN
-- 需要执行的sql语句
    statement_list
--  可以有多个when语句
    ...
--  否则
    ELSE
--      执行此sql语句
        statement_list
END CASE;

#方式二
CASE
--      此表达式为true                执行此sql语句
    WHEN search_condition THEN statement_list
--  可以有多条when语句
    ...
--  前面的都不成立，执行此sql语句
    ELSE statement_list
END CASE;
```

需求：给定一个月份，计算出所在的季节

```
CREATE PROCEDURE pro_test7(mon int)
BEGIN
    DECLARE result VARCHAR(10);
    CASE
    WHEN mon >= 1 AND mon <= 3 THEN
        SET result = '第一季度';
    WHEN mon >= 4 AND mon <= 6 THEN
        SET result = '第二季度';
    WHEN mon >= 7 AND mon <= 9 THEN
        SET result = '第三季度';
    ELSE
        SET result = '第四季度';
    END CASE;
    SELECT CONCAT('传递的月份为：',mon,'，计算出的结果为：', result) AS content;
END

CALL pro_test7(9)
```

while循环

语法：

```
WHILE search_condition DO
    statement_list
END WHILE;
```

需求：计算从1加到n的值

```
CREATE PROCEDURE pro_test8(n INT)
BEGIN
-- 声明两个变量，total：总和，num：累加数
    DECLARE total INT DEFAULT 0;
    DECLARE num INT DEFAULT 1;
-- 当num <= 输入的值
    WHILE num <= n DO
-- total加上循环的数num
        SET total = total + num;
-- num循环一次，要累加一次
        SET num = num + 1;
-- 结束循环
    END WHILE;

    SELECT total as content;
END

CALL pro_test8(3);
```

---

repeat结构

有条件的循环控制语句,当满足条件的时候退出循环。while 是满足条件才执行, repeat是满足条件就退出循环

语法：

```
REPEAT
--  循环的sql语句
    statement_list
--  直到       满足此条件，结束条件后没有分号
    UNTIL search_condition
--  结束循环
END REPEAT;
```

需求：计算从1加到n的值

```
CREATE PROCEDURE pro_test9(n INT)
    BEGIN
--  创建total变量，为int类型，默认值为0
        DECLARE total INT DEFAULT 0;
--  开始循环
```

```
        REPEAT
--   设置total + 输入的值
        SET total = total + n;
-- 循环一次，n减一
        SET n = n - 1;
--   直到n=0，结束循环；此处不能有分号结束符
        UNTIL n = 0
    END REPEAT;
    SELECT total as content;
END

CALL pro_test9(3);
```

---

loop语句

LOOP实现简单的循环,退出循环的条件需要使用其他的语句定义,通常可以使用LEAVE语句实现,具体语法如下:

语法:

```
-- loop循环的别名
[begin_label:] LOOP
--       循环语句
    statement_list

    IF exit_condition THEN
        LEAVE label;
    END IF;
END LOOP label;
```

---

leave语句

用来从标注的流程构造中退出,通常和BEGIN ... END或者循环一起使用。下面是一个使用LOOP和LEAVE的简单例子,退出循环

需求：计算从1加到n的值

```
CREATE PROCEDURE pro_test10(n INT)
BEGIN
--   设置int类型的变量total，默认值为0
    DECLARE total INT DEFAULT 0;
-- 别名：c
    c: LOOP
--   循环语句
    SET total = total + n;
    SET n = n - 1;
--   当输入的值，小于等于0时
    IF n <= 0 THEN
--   结束别名为c的循环
        LEAVE c;
--   结束if语句
```

```
    END IF;
--      结束loop循环
    END LOOP c;
-- 查询total的值
    SELECT total AS content;
END

CALL pro_test10(3);
```

游标/光标

游标是用来存储查询结果集的数据类型,在存储过程和函数中可以使用光标对结果集进行循环的处理。光标的使用包括光标的声明、OPEN、FETCH和CLOSE ,其语法分别如下。

语法:

```
-- 声明光标:
--      游标名称                    sql语句(查询的结果)
DECLARE cursor_name CURSOR FOR select_statement

-- OPEN光标:
OPEN cursor_name;

-- FETCH光标:fetch:抓取
FETCH cursor_name INTO var_name [, var_name] ...

-- CLOSE光标:
CLOSE cursor_name;
```

需求:查询emp表的数据,并逐行获取进行展示

```
-- 创建员工表,主键为id,其他字段为:名字,年龄,薪资
CREATE TABLE emp (
    id INT (11) NOT NULL auto_increment,
    NAME VARCHAR (50) NOT NULL COMMENT '姓名',
    age INT (11) COMMENT '年龄',
    salary INT (11) COMMENT '薪水',
    PRIMARY KEY (`id`)
) ENGINE = INNODB DEFAULT charset = utf8;

-- 给员工表插入几条数据
INSERT INTO emp (id, NAME, age, salary)
VALUES (NULL, '金毛狮王', 55, 3800),
    (NULL, '白眉鹰王', 60, 4000),
    (NULL, '青翼蝠王', 38, 2800),
    (NULL, '紫衫龙王', 42, 1800);
```

```
-- 创建存储过程,查询emp表的数据,且逐行获取进行展示
CREATE PROCEDURE pro_test11()
BEGIN
-- 创建4个变量,和表中对应的字段类型保持一致
```

```
    DECLARE e_id INT (11);
    DECLARE e_name VARCHAR (50);
    DECLARE e_age INT (11);
    DECLARE e_salary INT (11);
--      游标名称(封装的是后面sql语句的结果集)            此查询sql语句的结果集
    DECLARE emp_result CURSOR FOR SELECT * FROM emp;
--  开启游标
    OPEN emp_result;
--  抓取游标,几个变量(小于赋值给其他变量，用于查看遍历的结果),对应的表中的字段名称,类型也是一致
    FETCH emp_result INTO e_id, e_name, e_age, e_salary;
    SELECT CONCAT('id=', e_id, ',name=', e_name, ',age=', e_age, '薪资=',
e_salary);

--  游标抓取一次,得到一行的结果
    FETCH emp_result INTO e_id, e_name, e_age, e_salary;
    SELECT CONCAT('id=', e_id, ',name=', e_name, ',age=', e_age, '薪资=',
e_salary);

    FETCH emp_result INTO e_id, e_name, e_age, e_salary;
    SELECT CONCAT('id=', e_id, ',name=', e_name, ',age=', e_age, '薪资=',
e_salary);

    FETCH emp_result INTO e_id, e_name, e_age, e_salary;
    SELECT CONCAT('id=', e_id, ',name=', e_name, ',age=', e_age, '薪资=',
e_salary);

--  关掉游标
    CLOSE emp_result;
END

CALL pro_test11();
```

使用循环，优化以上写法

```
CREATE PROCEDURE pro_test12()
BEGIN
-- 创建4个变量,和表中对应的字段类型保持一致
    DECLARE e_id INT (11);
    DECLARE e_name VARCHAR (50);
    DECLARE e_age INT (11);
    DECLARE e_salary INT (11);
--  定义变量,判断光标是否还有后续指向
    DECLARE has_data INT DEFAULT 1;

--                  游标名称                                    此查询sql语句的结果集
    DECLARE emp_result CURSOR FOR SELECT * FROM emp;
    --  声明  退出  处理:未找到的声明退出处理程序 ;这一句只能是写在游标之后
    DECLARE EXIT HANDLER FOR NOT FOUND SET has_data = 0;

--  开启游标
    OPEN emp_result;
--  开启repeat循环
    REPEAT
--  抓取游标,几个变量,对应的表中的字段名称,类型也是一致
        FETCH emp_result INTO e_id, e_name, e_age, e_salary;
```

```
        SELECT CONCAT('id=', e_id, ',name=', e_name, ',age=', e_age, '薪资=',
e_salary);
--  退出repeat循环的条件,退出条件,不能有分号
        UNTIL has_data = 0
    END REPEAT;
--  关掉游标
    CLOSE emp_result;
END

CALL pro_test12();
```

存储函数

存储过程没有返回值，也可以返回结果(in:输入；out:输出)

语法：

```
-- 存储函数          函数名称          参数列表
CREATE FUNCTION function_name([param type, ...])
--  返回值的类型，函数是有返回值的，所以一定要有返回类型
RETURNS type
BEGIN

END
```

需求：定义一个存储函数，获取满足条件(city)的总记录数

```
CREATE FUNCTION fun1(countryId INT)
RETURNS INT
BEGIN
--  定义cnum变量,记录返回值
    DECLARE cnum INT;
--  在条件成立的情况下,将查询到的总数赋值到cnum变量中
    SELECT COUNT(*) INTO cnum FROM city WHERE country_id = countryId;
--  返回结果
    RETURN cnum;
END
-- 调用存储函数，因为函数是有返回值的，所以用select
SELECT fun1(2);


-- 删除存储函数
DROP FUNCTION fun1;
```

触发器

介绍

触发器是与表有关的数据库对象，指在 insert/update/delete 之前或之后，触发并执行触发器中定义的SQL语句集合。触发器的这种特性**可以协助应用在数据库端确保数据的完整性，日志记录，数据校验等操作**。

使用别名 OLD 和 NEW 来引用触发器中发生变化的记录内容，这与其他的数据库是相似的。现在触发器还只支持行级触发，不支持语句级触发

触发器类型	NEW 和 OLD的使用
INSERT 型触发器	NEW 表示将要或者已经新增的数据
UPDATE 型触发器	OLD 表示修改之前的数据，NEW 表示将要或已经修改后的数据
DELETE 型触发器	OLD 表示将要或者已经删除的数据

创建触发器

语法:mysql中只有行级触发器，Oracle既有行级触发器，也有语句级触发器

```
-- 创建    触发器      触发器名称
CREATE TRIGGER trigger_name
-- 之前/之后          插入/更新/删除------>在插入/更新/删除   的      之前/之后
BEFORE/AFTER     INSERT/UPDATE/DELETE

-- 表名
ON tb1_name

--   行级触发器
[FOR EACH ROW]

BEGIN
-- 触发器的内容
    trigger_stmt;

END;
```

需求：通过触发器记录emp表的数据变更日志emp_logs， 包含增加，修改;

```
-- 创建日志表
CREATE TABLE emp_logs (
    id INT ( 11 ) NOT NULL auto_increment,
    operation VARCHAR ( 20 ) NOT NULL COMMENT '操作类型, insert/update/delete',
    operate_time datetime NOT NULL COMMENT '操作时间',
    operate_id INT ( 11 ) NOT NULL COMMENT '操作表的ID',
    operate_params VARCHAR ( 500 ) COMMENT '操作参数',
PRIMARY KEY ( `id` )
) ENGINE = INNODB DEFAULT charset = utf8;
```

```
-- 插入时的触发器
```

```
CREATE TRIGGER emp_insert_trigger
-- 插入之后执行
AFTER INSERT
-- 对应emp员工表
ON emp
-- 声明为行级触发器
FOR EACH ROW
BEGIN
--                          员工日志表的字段
id自增长为null,明确类型insert,时间为当前时间,new是行 当前行的id,
    INSERT INTO emp_logs(id, operation, operate_time, operate_id,
operate_params) VALUES (NULL, 'INSERT', NOW(), new.id, CONCAT('插入后的(id:',
new.id, ',name:', new.NAME, ',age:', new.age, ',salary:', new.salary, ')'));
END



INSERT INTO emp (id, NAME, age, salary)
VALUES
    (NULL, '光明左使', 30, 3500);
```

```
-- 更新时的触发器
CREATE TRIGGER emp_update_trigger
-- 插入之后执行
AFTER UPDATE
-- 对应emp员工表
ON emp
-- 声明为行级触发器
FOR EACH ROW
BEGIN
--                          员工日志表的字段
    INSERT INTO emp_logs(id, operation, operate_time, operate_id,
operate_params) VALUES (NULL, 'UPDATE', NOW(), new.id, CONCAT('修改前的(id:',
old.id, ',name:', old.NAME, ',age:', old.age, ',salary:', old.salary, '),修改后的
(id:', new.id, ',name:', new.NAME, ',age:', new.age, ',salary:', new.salary,
')'));
END


UPDATE emp SET age = 39 WHERE id = 3;
```

```
-- 删除时的触发器
CREATE TRIGGER emp_delete_trigger
-- 插入之后执行
AFTER DELETE
-- 对应emp员工表
ON emp
-- 声明为行级触发器
FOR EACH ROW
BEGIN
--                          员工日志表的字段
    INSERT INTO emp_logs(id, operation, operate_time, operate_id,
operate_params) VALUES (NULL, 'DELETE', NOW(), old.id, CONCAT('删除前(id:',
old.id, ',name:', old.NAME, ',age:', old.age, ',salary:', old.salary, ')'));
END
```

```
-- 删除id=6的数据
DELETE FROM emp WHERE id = 8;
```

删除触发器

```
DROP TRIGGER [schema_name.] trigger_name;
```

查看触发器

```
-- 查看全部的触发器；加上/G可在控制台分隔查看
SHOW TRIGGERS;

-- 在MySQL中所有的触发器都存在INFORMATION_ SCHEMA数据库中，可以通过SELECT语句查看触发器
SELECT * FROM information_schema.`TRIGGERS`;
```

---

day2---mysql的体系结构概述

---

explain分析执行计划

通过以上步骤查询到效率低的SQL语句后,可以通过EXPLAIN或者DESC命令获取MySQL如何执行SELECT语句的信息,包括在SELECT语句执行过程中表如何连接和连接的顺序。

插叙SQL语句的执行计划

```
explain select 台from tb_ item where id = 1;
```

字段	含义
id	select查询的序列号,是一组数字,表示的是查询中执行select子句或者是操作表的顺序。
select_type	表示SELECT的类型,常见的取值有SIMPLE (简单表，即不使用表连接或者子查询)、PRIMARY (主查询,即外层的查询)、UNION (UNION中的第二个或者后面的查询语句). SUBQUERY (子查询中的第一个SELECT)等
table	输出结果集的表
type	表示表的连接类型,性能由好到差的连接类型为(system --> const --- eq_ref ----->ref ----- ref_or_null---->index_ merge --> index_subquery --> range ---> index ---> all)
possible_keys	表示查询时,可能使用的索引
key	表示实际使用的索引
key_len	索引字段的长度
rows	扫描行的数量
extra	执行情况的说明和描述

环境准备