数据库

基本概念

数据库范式

- 第一范式：1NF是对属性的原子性约束，要求属性具有原子性，不可再分解；即实体中的某个属性不能有多个值或者不能有重复的属性
- 第二范式：2NF是对记录的惟一性约束，要求记录有惟一标识，即实体的惟一性； 非主属性完全依赖于主关键字，消除部分依赖。如果存在依赖于主关键字一部分的属性，那我们需要将这个属性和主关键字的这一部分单独拎出来创建另一个表（所有单关键字的数据库表都符合第二范式，因为不可能存在组合关键字。如果依赖于一个组合关键属性，我们需要对这个组合进行拆分，和前面说的一样 ）

> 假定选课关系表为SelectCourse(学号，姓名，年龄，课程名称，成绩，学分)，关键字为组合关键字(学号，课程名称)，因为存在如下决定关系：
>
> (学号，课程名称）→（姓名，年龄，成绩，学分）
>
> 这个数据库表不满足第二范式，因为存在如下决定关系：
>
> (课程名称）→（学分）
>
> (学号）→（姓名，年龄）
>
> 即存在组合关键字中的字段决定非关键字的情况。
>
> 由于不符合2NF，这个选课关系表会存在如下问题：
>
> (1）数据冗余：
>
> 同一门课程由n个学生选修，"学分"就重复n-1次；同一个学生选修了m门课程，姓名和年龄就重复了m-1次。
>
> (2）更新异常：
>
> 若调整了某门课程的学分，数据表中所有行的"学分"值都要更新，否则会出现同一门课程学分不同的情况。
>
> (3）插入异常：

假设要开设一门新的课程，暂时还没有人选修。这样，由于还没有"学号"关键字，课程名称和学分也无法记录入数据库。

（4）删除异常：

假设一批学生已经完成课程的选修，这些选修记录就应该从数据库表中删除。但是，与此同时，课程名称和学分信息也被删除了。很显然，这也会导致插入异常。

把选课关系表SelectCourse改为如下三个表：

学生：Student(学号，姓名，年龄)；

课程：Course(课程名称，学分)；

选课关系：SelectCourse(学号，课程名称，成绩)。

这样的数据库表是符合第二范式的， 消除了数据冗余、更新异常、插入异常和删除异常。

另外，所有单关键字的数据库表都符合第二范式，因为不可能存在组合关键字。

- 第三范式：3NF是对字段冗余性的约束，即任何字段不能由其他字段派生出来，它要求字段没有冗余。简而言之，第三范式（3NF）要求一个数据库表中不包含已在其它表中已包含的非主关键字信息。也就是属性不依赖与非主属性

假定学生关系表为Student(学号，姓名，年龄，所在学院，学院地点，学院电话)，关键字为单一关键字"学号"，因为存在如下决定关系：
（学号）→（姓名，年龄，所在学院，学院地点，学院电话）

这个数据库是符合2NF的，但是不符合3NF，因为存在如下决定关系：

（学号）→（所在学院）→（学院地点，学院电话）

即存在非关键字段"学院地点"、"学院电话"对关键字段"学号"的传递函数依赖。

它也会存在数据冗余、更新异常、插入异常和删除异常的情况.

把学生关系表分为如下两个表：

学生：(学号，姓名，年龄，所在学院)；

学院：(学院，地点，电话)

视图

基本概念

在 SQL 中，视图是基于 SQL 语句的结果集的可视化的表。

视图包含行和列，就像一个真实的表。视图中的字段就是来自一个或多个数据库中的真实的表中的字段。我们可以向视图添加 SQL 函数、WHERE 以及 JOIN 语句，我们也可以提交数据，就像这些来自于某个单一的表。

基本语法

CREATE VIEW view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition

之后可以对视图类似数据库表进行select的操作。

作用

1、 视图能简化用户操作
2、 视图使用户能以多种角度看待同一数据
3、 视图对重构数据库提供了一定程度的逻辑独立性
4、视图能够对机密数据提供安全保护

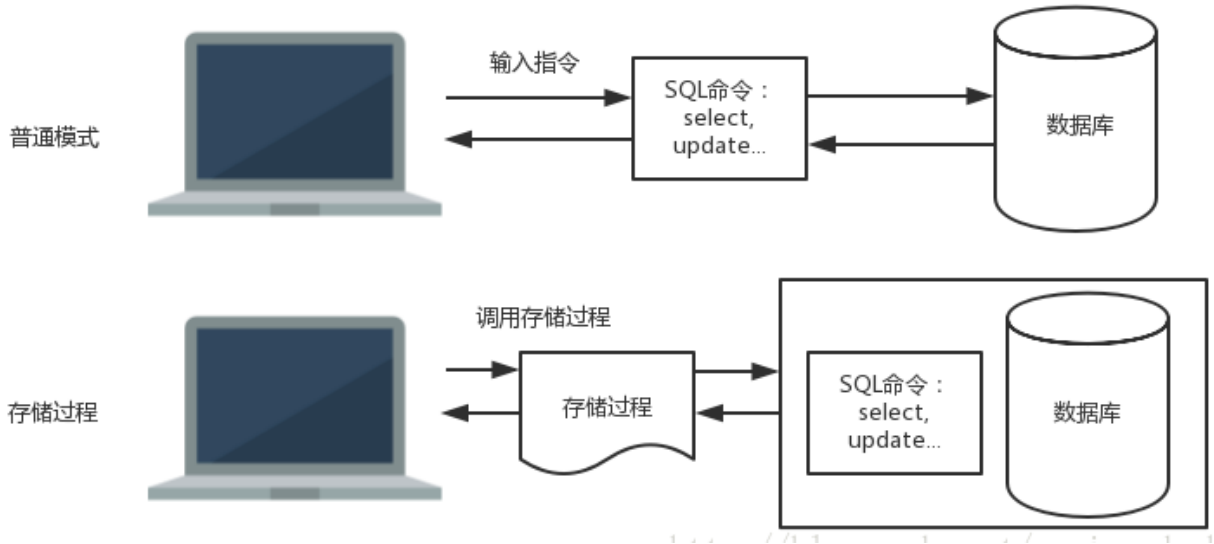
和表的区别

1、视图是已经编译好的sql语句。而表不是
2、视图没有实际的物理记录。而表有
3、表是内容，视图是窗口
4、表只用物理空间而视图不占用物理空间，视图只是逻辑概念的存在，表可以及时对它进行修改，但视图只能有创建的语句来修改
5、表是内模式，视图是外模式
6、视图是查看数据表的一种方法，可以查询数据表中某些字段构成的数据，只是一些SQL语句的集合。从安全的角度说，视图可以不给用户接触数据表，从而不知道表结构。
7、表属于全局模式中的表，是实表；视图属于局部模式的表，是虚表。
8、视图的建立和删除只影响视图本身，不影响对应的基本表。

理解存储过程

基本概念

存储过程（Stored Procedure）是在大型数据库系统中，一组为了完成特定功能的SQL 语句集，存储在数据库中，经过第一次编译后再次调用不需要再次编译，用户通过指定存储过程的名字并给出参数（如果该存储过程带有参数）来执行它。存储过程是数据库中的一个重要对象。



如图所示，在普通模式下获取数据，用户需要输入SQL命令与数据库进行交互，而存储过程是编写好的SQL命令，存储在数据库中，用户操作的时候只需要调用存储过程，而不用重新输入冗余繁杂的SQL命令

存储过程的优势

- 存储过程可以重复使用，大大减小开发人员的负担；
- 对于网络上的服务器，可以大大减小网络流量，因为只需要传递存储过程的名称即可；
- 可以防止对表的直接访问，只需要赋予用户存储过程的访问权限。

存储过程全操作

创建存储过程

```
/*参数种类1 ：IN 或者OUT
 *参数1 ：通常是指参数名
 */
>CREATE PROCEDURE 存储过程名(参数种类1 参数1 数据类型1,[...])
 BEGIN
 具体的procedure(处理)
 END
```

查看数据库中的存储过程

```
>SHOW PROCEDURE STATUS\G
```

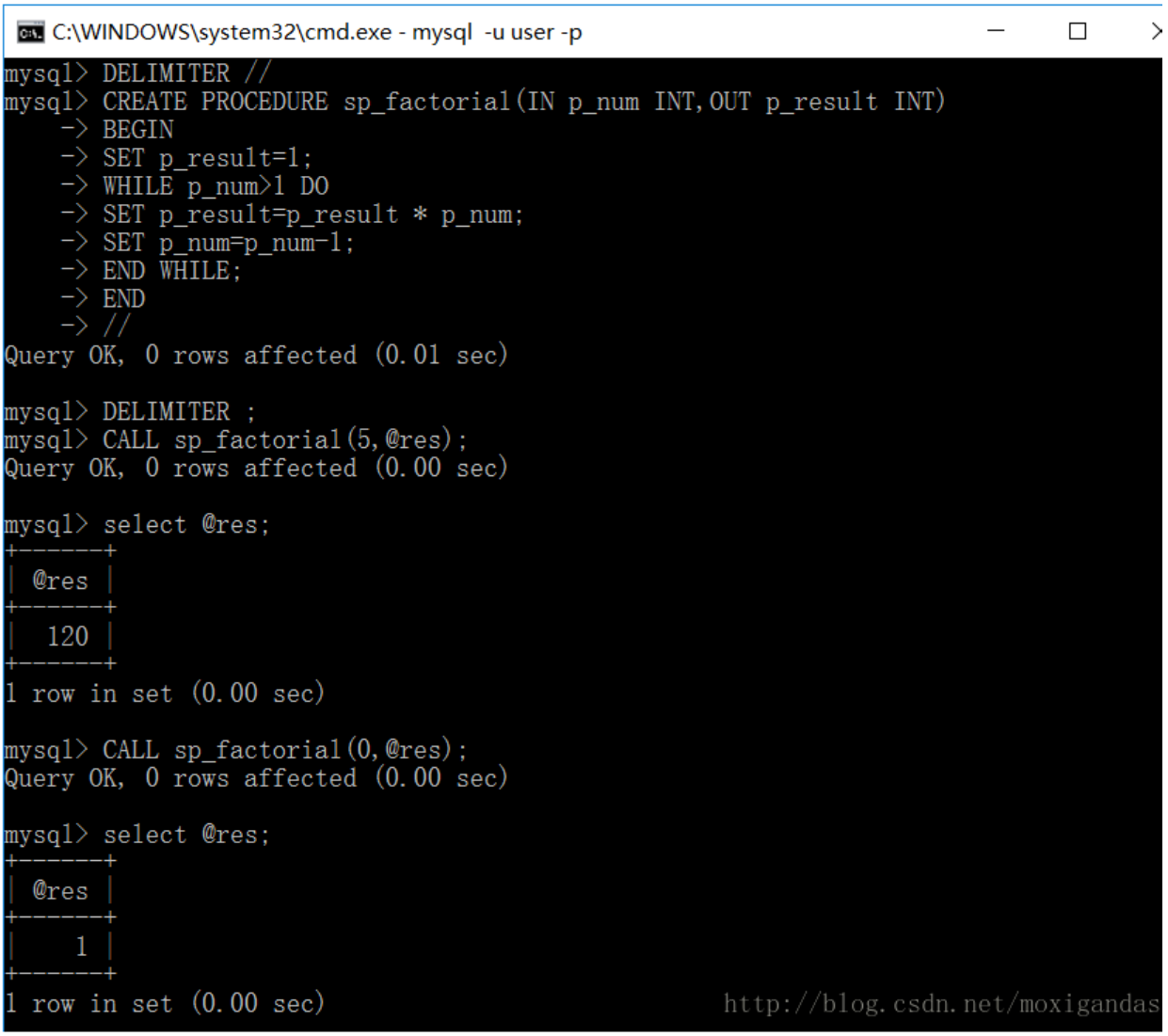
查看具体的存储过程

```
>SHOW CREATE PROCEDURE 存储过程名\G
```

调用存储过程

```
CALL 存储过程名(参数,...)
```

如果参数类型是out，在调用时，要在输出参数前面加上@，这样将输出结果保存在“@变量名”中。



删除存储过程

```
DROP PROCEDURE 存储过程名;
```

触发器

触发器是与表有关的数据库对象，在满足定义条件时触发，并执行触发器中定义的语句集合

触发条件：insert，update,delete;
触发时间：before，after
执行体：在begin和end之间的语句集合
触发频率：针对每一行执行
触发器定义在表上，操作也在表上

```
//模板
CREATE
    [DEFINER = { user | CURRENT_USER }]
TRIGGER trigger_name
trigger_time trigger_event
ON tbl_name FOR EACH ROW
    [trigger_order]
trigger_body

trigger_time: { BEFORE | AFTER }

trigger_event: { INSERT | UPDATE | DELETE }

trigger_order: { FOLLOWS | PRECEDES } other_trigger_name


//例子一
mysql> CREATE TRIGGER trig1 AFTER INSERT
    -> ON work FOR EACH ROW
    -> INSERT INTO time VALUES(NOW());


//例子二：new和old代表更新前后的数据
mysql> CREATE TRIGGER upd_check BEFORE UPDATE ON account
    -> FOR EACH ROW
    -> BEGIN
    ->     IF NEW.amount < 0 THEN
    ->         SET NEW.amount = 0;
    ->     ELSEIF NEW.amount > 100 THEN
    ->         SET NEW.amount = 100;
    ->     END IF;
    -> END$$
```

基本的CRUD语句

插入

```
INSERT INTO tb_name (field1, field2, ..., fieldn) VAULES(value1, value2, ...,valuen);
```

更新

```
UPDATE tb_name SET field1=value1, field2=value2, ..., fieldn=valuen [WHERE condition];

//更新多个表的数据
UPDATE tb_name1, tb_name2, ..., tb_namen SET tb_name1.field=value, ...,
tb_namen.field=value [WHERE condition];
```

删除

```
DELETE FROM tb_name [WHERE condition];

//删除多个表的数据--真正要删除的记录在 FROM 前的 tb_name 中，FROM 后的 tb_name 是用于 WHERE 中的条件判断。

DELETE tb_name1, tb_name2, ..., tb_namen FROM tb_name1, tb_name2, ..., tb_namen [WHERE condition];
```

查询

```
SELECT column_name FROM tb_name;

//查询不重复记录
SELECT distinct column_name FROM tb_name;

//条件查询
SELECT column_name FROM tb_name WHERE condition;

//排序
SELECT * FROM tb_name [WHERE condition] [ORDER BY field1 [DESC|ASC], field2 [DESC|ASC], ..., fieldn [DESC|ACS]];

//限制---offset_start 表示偏移量， row_conut 表示显示的行数。
SELECT * FROM tb_name [LIMIT offset_start, row_count];
```

聚合

```
/*
function_name 表示要做的聚合操作，又称聚合函数。常用的有 sum()、 count(*)、 max() 和 min()。
GROUP BY 关键字表示要进行分类聚合的字段。
WITH ROLLUP 表明是否对分类聚合后的结果进行再汇总。
HAVING 关键字表示对分类后的结果再进行条件的过滤。
注意：HAVING 和 WHERE 的区别在于，HAVING 是对聚合后的结果进行条件的过滤，而 WHERE 是对聚合前的记录进行过滤
*/

SELECT [field1, field2, ..., fieldn] function_name
FROM tb_name
[WHERE condition]
[GROUP BY field1, field2, ..., fieldn
[WITH ROLLUP]
[HAVING condition]];
```

连接查询

内连接 查询两个表的交集，不满足条件的都不显示

```
//隐式写法
SELECT <selectList> FROM A,B WHERE A.列=B.列

//显示写法
SELECT <selectList> FROM A [INNER] JOIN B ON A.列= B.列
```

外连接
左外连接：查询出JOIN左边表的全部数据查询出啦，JOIN右边的表不匹配的数据使用NULL来填充数据。
右外连接：查询出JOIN右边表的全部数据查询出啦，JOIN左边的表不匹配的数据使用NULL来填充数据。

```
SELECT <selectList>
FROM A LEFT/RIGHT[OUTER]JOIN B
ON (A.column_name=B.column_name
```

子查询

where from

in

查询优化

事务

事务其实就是单个数据逻辑单元组成的对象操作集合

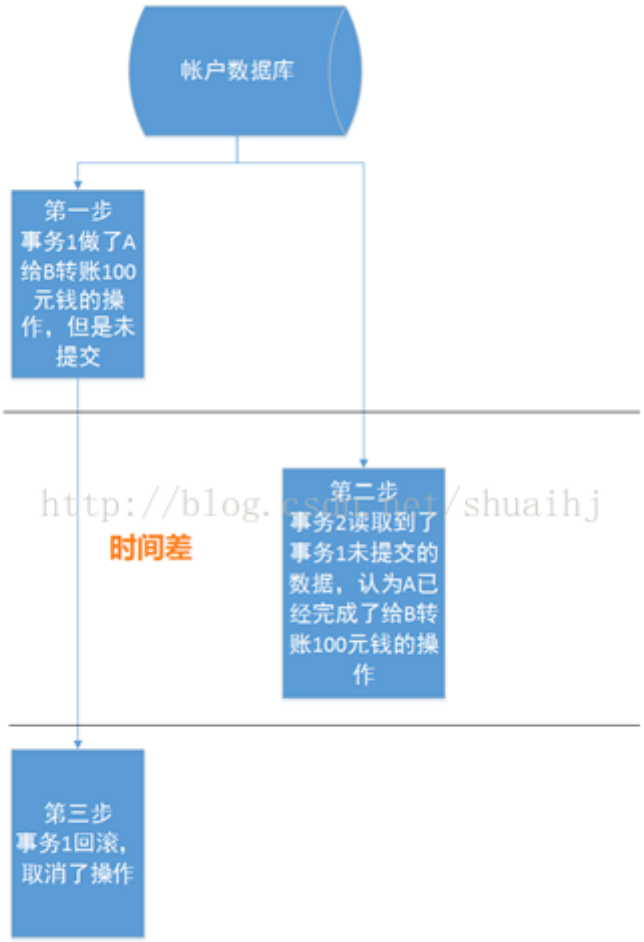
ACID的基本概念

- 原子性（Atomicity）一个事务必须被视为一个不可分割的最小工作单元，整个事务中的所有操作要么全部提交成功，要么全部失败回滚，对于一个事务来说，不可能只执行其中的一部分操作。
- 一致性（Consistency）数据库总是从一个一致性的状态转换到另一个一致性的状态。
  - 拿转账来说，a给b转账500，转账前两人共1000元，转账后两人还是1000元。
- 隔离性（Isolation）一个事务所做的修改在最终提交以前，对其他事务是不可见的。
  - 也就是对于任意并发的，ab两个事务，在事务a看来，b要么在a开始之前就开始了，要么在a结束之后才开始，并不能感受到并发执行这个状态。
- 持久性（Durability）一旦事务提交，则其所做的修改会永久保存到数据库。

事务之间的影响

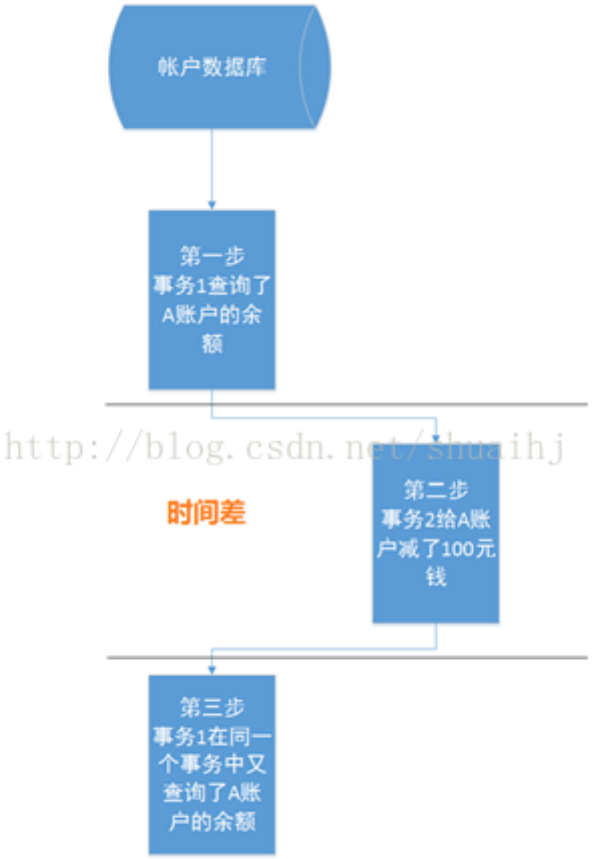
脏读

脏读意味着一个事务读取了另一个事务未提交的数据，而这个数据是有可能回滚的；如下案例，此时如果事务1回滚，则B账户必将有损失。



不可重复读

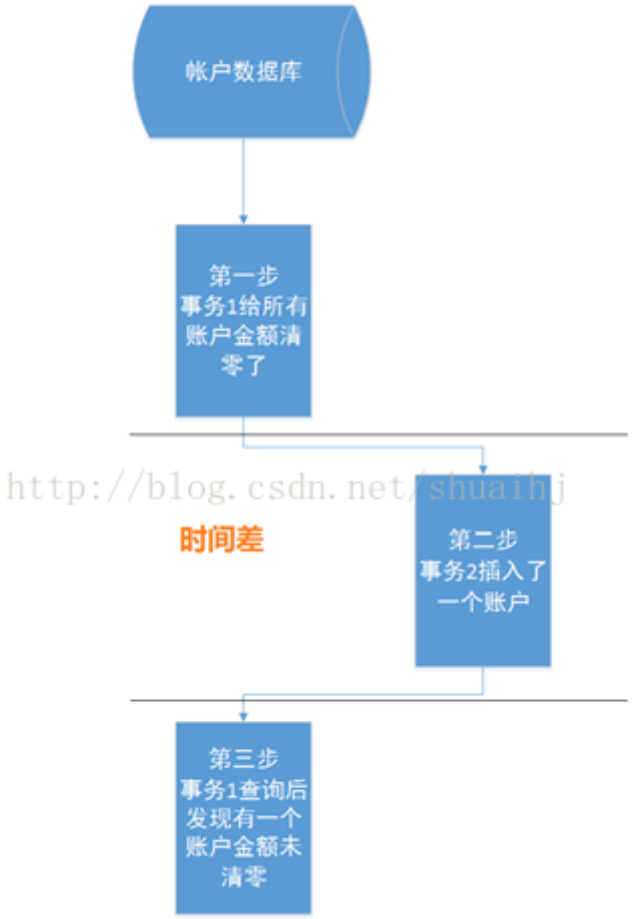
不可重复读是指在对于数据库中的某个数据，一个事务范围内多次查询却返回了不同的数据值，这是由于在查询间隔，被另一个事务修改并提交了。



幻读/虚读

幻读是事务非独立执行时发生的一种现象。例如事务T1对一个表中所有的行的某个数据项做了从“1”修改为“2”的操作，这时事务T2又对这个表中插入了一行数据项，而这个数据项的数值还是为“1”并且提交给数据库。而操作事务T1的用户如果再查看刚刚修改的数据，会发现还有一行没有修改，其实这行是从事务T2中添加的，就好像产生幻觉一样，这就是发生了幻读。

幻读和不可重复读都是读取了另一条已经提交的事务（这点就脏读不同），所不同的是不可重复读查询的都是同一个数据项，而幻读针对的是一批数据整体（比如数据的个数）。



丢失更新

两个事务同时读取同一条记录，A先修改记录，B也修改记录（B是不知道A修改过），B提交数据后B的修改结果覆盖了A的修改结果。

四大隔离级别及对应问题

数据库的事务隔离级别（TRANSACTION ISOLATION LEVEL）是一个数据库上很基本的一个概念。为什么会有事务隔离级别，SQL Server上实现了哪些事务隔离级别？事务隔离级别的前提是一个多用户、多进程、多线程的并发系统，在这个系统中为了保证数据的一致性和完整性，我们引入了事务隔离级别这个概念，对一个单用户、单线程的应用来说则不存在这个问题。

隔离级别	脏读	丢失更新	不可重复读	幻读	并发模型	更新冲突检测
未提交读：Read Uncommited	是	是	是	是	悲观	否
已提交读：Read commited	否	是	是	是	悲观	否
可重复读：Repeatable Read	否	否	否	是	悲观	否
可串行读：Serializable	否	否	否	否	悲观	否

可以在数据库中设置某个表的隔离级别，也可以在jdbc或者其他的数据库连接引擎中设置隔离级别

实现原理

read_uncommited：

- 事务对被读取的数据不加锁
- 事务在更新某数据的瞬间（发生更新的瞬间），必须对其行级加锁，直到事务结束才释放
  - 事务1读取某行记录时，事务2也能对这行记录进行读取、更新；当事务2对该记录进行更新时，事务1再次读取该记录，能读到事务2对该记录的修改版本，即使该修改尚未被提交。
  - 事务1更新某行记录时，事务2不能对这行记录做更新，直到事务1结束。

read_commited:

- 事务对当前被读取的数据加行级共享锁（当读到时才加锁），一旦读完该行，立即释放该行级共享锁；
- 事务在更新某数据的瞬间（就是发生更新的瞬间），必须先对其加 行级排他锁，直到事务结束才释放
  - 事务1读取某行记录时，事务2也能对这行记录进行读取、更新；当事务2对该记录进行更新时，事务1再次读取该记录，读到的只能是事务2对其更新前的版本，要不就是事务2提交后的版本。
  - 事务1更新某行记录时，事务2不能对这行记录做更新，直到事务1结束。

repeated read：

- 事务在读取某数据的瞬间（就是开始读取的瞬间），必须先对其加行级共享锁，直到事务结束才释放；
- 事务在更新某数据的瞬间（就是发生更新的瞬间），必须先对其加行级排他锁，直到事务结束才释放。
  - 事务1读取某行记录时，事务2也能对这行记录进行读取、更新；当事务2对该记录进行更新时，事务1再次读取该记录，读到的仍然是第一次读取的那个版本
  - 事务1更新某行记录时，事务2不能对这行记录做更新，直到事务1结束。

serialzable

- 事务在读取数据时，必须先对其加 表级共享锁 ，直到事务结束才释放；
- 事务在更新数据时，必须先对其加 表级排他锁 ，直到事务结束才释放。
  - 事务1正在读取A表中的记录时，则事务2也能读取A表，但不能对A表做更新、新增、删除，直到事务1结束。
  - 事务1正在更新A表中的记录时，则事务2不能读取A表的任意记录，更不可能对A表做更新、新增、删除，直到事务1结束。

封锁的类型以及粒度，两段锁协议，隐式锁定和显式锁定

mysql中的锁

在计算机科学中，锁是在执行多线程时用于强行限制资源访问的同步机制，即用于在并发控制中保证对互斥要求的满足。

在数据库的锁机制中，咱们介绍过在 DBMS 中，可以按照锁的粒度把数据库锁分为行级锁（InnoDB 引擎）、表级锁（MyISAM 引擎）和页级锁（BDB 引擎）。

行级锁
行级锁是Mysql中锁定粒度最细的一种锁，表示只针对当前操作的行进行加锁。行级锁能大大减少数据库操作的冲突。其加锁粒度最小，但加锁的开销也最大。行级锁分为共享锁 和 排他锁。

特点 :开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度也最高。

表级锁
表级锁是MySQL中锁定粒度最大的一种锁，表示对当前操作的整张表加锁，它实现简单，资源消耗较少，被大部分MySQL引擎支持。最常使用的MYISAM与INNODB都支持表级锁定。表级锁定分为表共享读锁（共享锁）与表独占写锁（排他锁）。

特点: 开销小，加锁快；不会出现死锁；锁定粒度大，发出锁冲突的概率最高，并发度最低。

页级锁 页级锁是MySQL中锁定粒度介于行级锁和表级锁中间的一种锁。表级锁速度快，但冲突多，行级冲突少，但速度慢。所以取了折衷的页级，一次锁定相邻的一组记录。BDB支持页级锁

特点 :开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般

行级锁与死锁

原因

MyISAM中是不会产生死锁的，因为MyISAM总是一次性获得所需的全部锁，要么全部满足，要么全部等待。而在InnoDB中，锁是逐步获得的，就造成了死锁的可能。

在MySQL中，行级锁并不是直接锁记录，而是锁索引。索引分为主键索引和非主键索引两种，如果一条sql语句操作了主键索引，MySQL就会锁定这条主键索引；如果一条语句操作了非主键索引，MySQL会先锁定该非主键索引，再锁定相关的主键索引。 在UPDATE、DELETE操作时，MySQL不仅锁定WHERE条件扫描过的所有索引记录，而且会锁定相邻的键值，即所谓的next-key locking。

当两个事务同时执行，一个锁住了主键索引，在等待其他相关索引。另一个锁定了非主键索引，在等待主键索引。这样就会发生死锁。

发生死锁后，InnoDB一般都可以检测到，并使一个事务释放锁回退，另一个获取锁完成事务。

避免死锁办法

- 如果不同程序会并发存取多个表，尽量约定以相同的顺序访问表，可以大大降低死锁机会。
- 在同一个事务中，尽可能做到一次锁定所需要的所有资源，减少死锁产生概率；
- 对于非常容易产生死锁的业务部分，可以尝试使用升级锁定颗粒度，通过表级锁定来减少死锁产生的概率；

两次锁协议

- **一次性锁协议**，事务开始时，即一次性申请所有的锁，之后不会再申请任何锁，如果其中某个锁不可用，则整个申请就不成功，事务就不会执行，在事务尾端，一次性释放所有的锁。一次性锁协议不会产生死锁的问题，但事务的并发度不高。
- **两阶段锁协议**，整个事务分为两个阶段，前一个阶段为加锁，后一个阶段为解锁。在加锁阶段，事务只能加锁，也可以操作数据，但不能解锁，直到事务释放第一个锁，就进入解锁阶段，此过程中事务只能解锁，也可以操作数据，不能再加锁。两阶段锁协议使得事务具有较高的并发度，因为解锁不必发生在事务结尾。它的不足是没有解决死锁的问题，因为它在加锁阶段没有顺序要求。如两个事务分别申请了A, B锁，接着又申请对方的锁，此时进入死锁状态。

显示锁/隐式锁

mysql锁分为隐式锁和显式锁。当多个客户端并发访问同一个数据的时候，为了保证数据的一致性，数据库管理系统会自动的为该数据加锁、解锁，这种被称为隐式锁。隐式锁无需开发人员维护（包括锁粒度、加锁时机、解锁时机等）

当时在某些特殊的情况下需要开发人员手动的进行加锁、解锁，这种锁方式被称为显式锁。对于显式锁而言，开发人员不仅要确定锁的粒度，还需要确定加锁的时机（何时加锁）、解锁的时机（何时解锁）以及所的类型。

```
//mybaits中
<mapper namespace="hello.UserMapper">
    <select id="getUser" parameterType="int" resultType="hello.User">
        LOCK TABLE users READ;
        select * from users where id=#{id};
    </select>
</mapper>


//数据库中
Lock tables xxxxx read local, yyyy read local;
do something;
Unlock tables;
```

乐观锁与悲观锁

悲观锁

当我们要对一个数据库中的一条数据进行修改的时候，为了避免同时被其他人修改，最好的办法就是直接对该数据进行加锁以防止并发。

这种借助数据库锁机制在修改数据之前先锁定，再修改的方式被称之为悲观并发控制（又名“悲观锁”，Pessimistic Concurrency Control，缩写“PCC”）。

类似java中的synchronized。

流程

- 在对任意记录进行修改前，先尝试为该记录加上排他锁（exclusive locking）。

- 如果加锁失败，说明该记录正在被修改，那么当前查询可能要等待或者抛出异常。 具体响应方式由开发者根据实际需要决定。
- 如果成功加锁，那么就可以对记录做修改，事务完成后就会解锁了。
- 其间如果有其他对该记录做修改或加排他锁的操作，都会等待我们解锁或直接抛出异常。

乐观锁

乐观锁（Optimistic Lock），就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在提交更新的时候会判断一下在此期间别人有没有去更新这个数据。乐观锁适用于读多写少的应用场景，这样可以提高吞吐量。

假设不会发生并发冲突，只在提交操作时检查是否违反数据完整性

类似于java中的CAS概念，通过新增字段（比如版本，比如前后时间），在更新操作发生时去进行对比，如果相同才进行这个操作

手段

- 使用数据版本（Version）记录机制实现，这是乐观锁最常用的一种实现方式。何谓数据版本？即为数据增加一个版本标识，一般是通过为数据库表增加一个数字类型的 “version” 字段来实现。当读取数据时，将version字段的值一同读出，数据每更新一次，对此version值加一。当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的version值进行比对，如果数据库表当前版本号与第一次取出来的version值相等，则予以更新，否则认为是过期数据。
- 使用时间戳（timestamp）。乐观锁定的第二种实现方式和第一种差不多，同样是在需要乐观锁控制的table中增加一个字段，名称无所谓，字段类型使用时间戳（timestamp），和上面的version类似，也是在更新提交的时候检查当前数据库中数据的时间戳和自己更新前取到的时间戳进行对比，如果一致则OK，否则就是版本冲突

MVCC原理，当前读与快照读。

MVCC(Multi Version Concurrency Control的简称)，代表多版本并发控制。与MVCC相对的，是基于锁的并发控制，Lock-Based Concurrency Control)。

MVCC(Multi Version Concurrency Control的简称)，代表多版本并发控制。与MVCC相对的，是基于锁的并发控制，Lock-Based Concurrency Control)。

MVCC的实现（Innodb）

InnoDB的MVCC,是通过在每行记录后面保存两个隐藏的列来实现的,这两个列，分别保存了这个行的创建时间，一个保存的是行的删除时间。这里存储的并不是实际的时间值,而是系统版本号(可以理解为事务的ID)，每开始一个新的事务，系统版本号就会自动递增，事务开始时刻的系统版本号会作为事务的ID.

在 read uncommit 隔离级别下，每次都是读取最新版本的数据行，所以不能用 MVCC 的多版本，而 serializable 隔离级别每次读取操作都会为记录加上读锁，也和 MVCC 不兼容，所以只有 RC 和 RR 这两个隔离级别才有 MVCC。

快照读 RR 和 RC 隔离级别都实现了 MVCC 来满足读写并行，但是读的实现方式是不一样的：RC 总是读取记录的最新版本，如果该记录被锁住，则读取该记录最新的一次快照，而 RR 是读取该记录事务开始时的那个版本。虽然这两种读取方式不一样，但是它们读取的都是快照数据，并不会被写操作阻塞，所以这种读操作称为 快照读（Snapshot Read），有时候也叫做 非阻塞读（Nonlocking Read）

当前读

除了 快照读 ，MySQL 还提供了另一种读取方式：当前读（Current Read），有时候又叫做 加锁读（Locking Read） 或者 阻塞读（Blocking Read），这种读操作读的不再是数据的快照版本，而是数据的最新版本，并会对数据加锁，根据加锁的不同，又分成两类：

SELECT ... LOCK IN SHARE MODE：加 S 锁
SELECT ... FOR UPDATE：加 X 锁
INSERT / UPDATE / DELETE：加 X 锁
当前读在 RR 和 RC 两种隔离级别下的实现也是不一样的：RC 只加记录锁，RR 除了加记录锁，还会加间隙锁，用于解决幻读问题。

不同隔离级别下InnoDB的读操作

	快照读	当前读
读未提交		读取最新版本
读已提交	读取最新一份快照	读取最新版本，并加记录锁
可重复读	读取事务开始时的快照	读取最新版本，并加记录锁 + 间隙锁
可序列化		读取最新版本，并加记录锁 + 间隙锁

索引

数据库索引，是数据库管理系统中一个排序的数据结构，索引的实现通常使用B树及其变种B+树。

在数据之外，数据库系统还维护着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

索引包含从表中一个或多个列生成的键，以及映射到指定数据的存储位置的指针，也就是说索引由键和 指针组成。

索引基础概念

索引的作用

- 通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。
- 可以大大加快数据的检索速度，这也是创建索引的最主要的原因。
- 可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。
- 在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。

- 通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能

索引的缺点

- 创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。
- 索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。
- 当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

适用场景

- 在经常需要搜索的列上，可以加快搜索的速度；
- 在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构；
- 在经常用在连接的列上，这些列主要是一些外键，可以加快连接的速度；
- 在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的；
- 在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；
- 在经常使用在WHERE子句中的列上面创建索引，加快条件的判断速度。

不适用场景

- 在查询中很少适用或者参考的列不应该创建索引
- 对于那些只有很少数据值的列（比如在人事表中对性别创建索引，意义不大）
- 对于定义为text，image和bit数据类型的列（这些列要么数据量大，要么取值很少）
- 当修改性能需求远大于检索性能需求时（因为索引会降低修改更新的速度，增加维护成本）

索引分类

普通索引

普通索引（由关键字KEY或INDEX定义的索引）的唯一任务是加快对数据的访问速度。因此，应该只为那些最经常出现在查询条件（WHEREcolumn=）或排序条件（ORDERBYcolumn）中的数据列创建索引。只要有可能，就应该选择一个数据最整齐、最紧凑的数据列（如一个整数类型的数据列）来创 建索引。

允许数据重复

唯一索引

唯一索引是不允许其中任何两行具有相同索引值的索引。 当现有数据中存在重复的键值时，大多数数据库不允许将新创建的唯一索引与表一起保存。数据库还可能防止添加将在表中创建重复键值的新数据。例如，如果在 employee 表中职员的姓 (lname) 上创建了唯一索引，则任何两个员工都不能同姓。

主键索引
在数据库关系图中为表定义主键将自动创建主键索引，主键索引是唯一索引的特定类型。该索引要求主键中的每个值都唯一。当在查询中使用主键索引时，它还允许对数据的快速访问

全文索引

目前只有MyISAM引擎支持。其可以在CREATE TABLE ，ALTER TABLE ，CREATE INDEX 使用，不过目前只有 CHAR、VARCHAR ，TEXT 列上可以创建全文索引。

全文索引并不是和MyISAM一起诞生的，它的出现是为了解决WHERE name LIKE “%word%"这类针对文本的模糊查询效率较低的问题。在没有全文索引之前，这样一个查询语句是要进行遍历数据表操作的，可见，在数据量较大时是极其的耗时的，如果没有异步IO处理，进程将被挟持，很浪费时间。

FULLTEXT索引也是按照分词原理建立索引的。

布尔模式：允许一些特殊字符用于标记一些具体的要求，如+表示一定要有，-表示一定没有，*表示通用匹配符，类似正则语法；
自然语言模式，就是简单的单词匹配；含表达式的自然语言模式，就是先用自然语言模式处理，对返回的结果，再进行表达式匹配。

```
//创建带有全文索引的表
CREATE TABLE article (
    id INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY,
    title VARCHAR(200),
    content TEXT,
    FULLTEXT (title, content) --在title和content列上创建全文索引
);

//对现有表添加全文索引
ALTER TABLE article
ADD FULLTEXT INDEX fulltext_article (title, content)


//普通的模糊查询
SELECT * FROM article WHERE content LIKE '%查询字符串%'
//调用全文索引进行模糊查询
SELECT * FROM article WHERE MATCH(title, content) AGAINST('查询字符串')
```

聚簇索引

这里要理解InnoDB数据库存储的文件结构，Innodb的存储文件有两个.frm和.idb。其中.frm是表的定义文件，而idb是数据文件。从这里我们可以得出，InnoDB的聚簇索引，数据信息就直接存在于叶节点中。

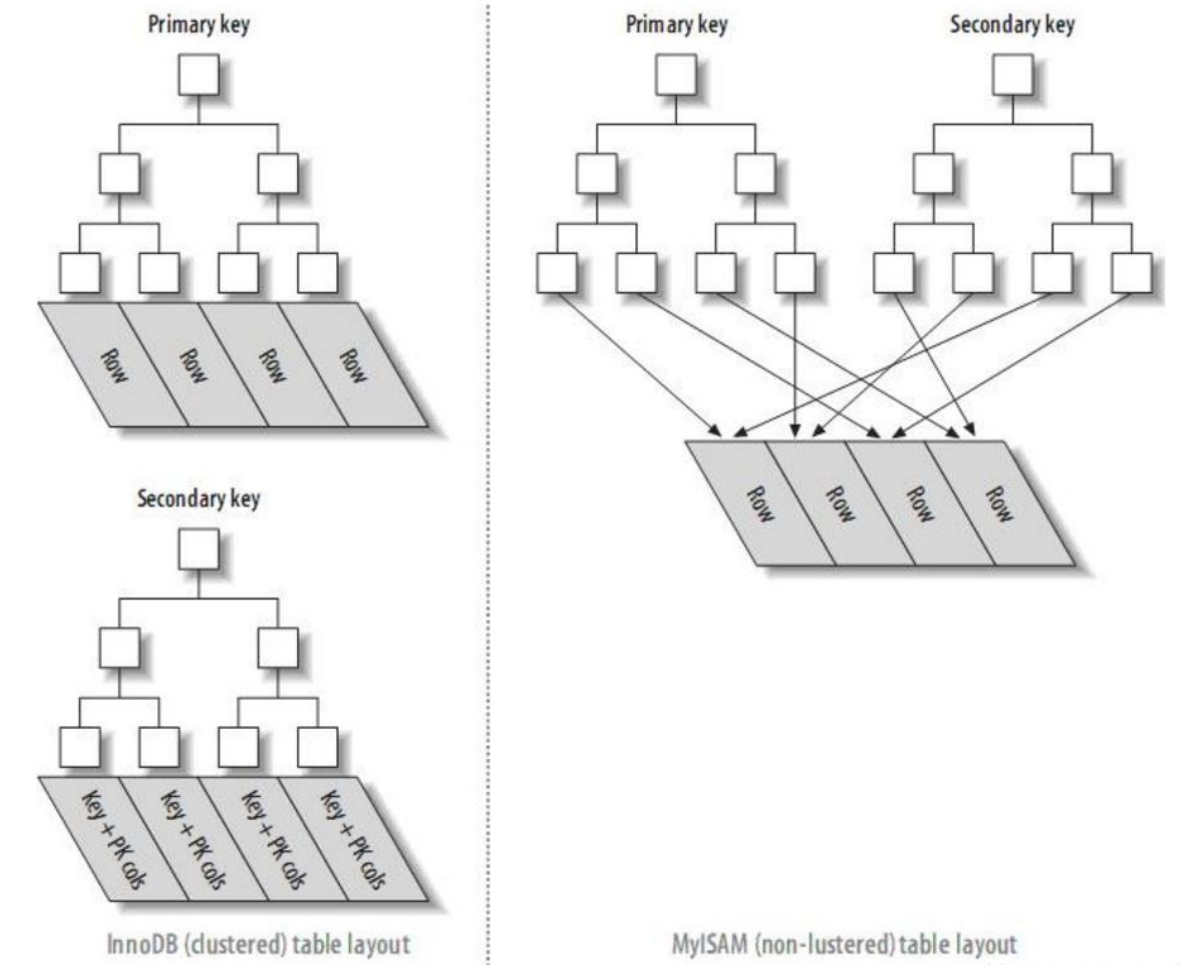
所以聚簇索引就是将主键组织到一个B+树上，行数据就存储在叶子节点上。当使用“where id = 14”这样的条件查找逐渐时，则按照B+树的检索算法即可查找到对应的叶节点，之后获得行数据。若对非主键列进行条件搜索，则需要现在辅助索引B+树中检索该列名，到期其叶子节点获得对应的主键，然后再去主索引B+树去进行检索操作，到达叶子结点获取整行数据。

然后聚簇索引的逻辑上顺序和刚开始插入的顺序是一致的。不是物理上的一致。

辅助索引
辅助索引页级别不包含行的全部数据。叶节点除了包含键值以外，每个叶级别中的索引行中还包含了一个书签，该书签用来告诉InnoDB哪里可以找到与索引相对应的行数据。其中存的就是聚集索引的键。 辅助索引的存在并不影响数据在聚集索引的结构组织。InnoDB会遍历辅助索引并通过叶级别的指针获得指向主键索引的主键，然后通过主键索引找到一个完整的行记录。当然如果只是需要辅助索引的值和主键索引的值，那么只需要查找辅助索引就可以查询出索要的数据，就不用再去查主键索引了。

非聚簇索引 这里要从使用非聚簇索引的Myisam引擎底层文件说起，他包括三个存储文件。.frm,.MYD.MYI三个，其中frm是表的定义文件，MYD是数据文件，MYI是索引文件。所以从这里我们可以看出非聚簇索引的B+树叶子结点保存的是指向数据文件的索引（类似一个指针），而不直接是行数据。而且非聚簇索引的辅助索引表和主索引表没有明显的区别，保存的都是指向数据的指针。

INNODB和MYISAM的主键索引与二级索引的对比：



索引的底层实现

这个关注下B树和B+树的结构就可以了。

底层引擎

InnoDB和Myisam的区别

-	MyISAM	InnoDB
存储结构	每个MyISAM在磁盘上存储成三个文件。第一个文件的名字以表的名字开始，扩展名指出文件类型。.frm文件存储表定义。数据文件的扩展名为.MYD (MYData)。索引文件的扩展名是.MYI (MYIndex)。	所有的表都保存在同一个数据文件中（也可能是多个文件，或者是独立的表空间文件），InnoDB表的大小只受限于操作系统文件的大小，一般为2GB。
存储空间	可被压缩，存储空间较小。支持三种不同的存储格式：静态表(默认，但是注意数据末尾不能有空格，会被去掉)、动态表、压缩表。	需要更多的内存和存储，它会在主内存中建立其专用的缓冲池用于高速缓冲数据和索引。
可移植性、备份及恢复	数据是以文件的形式存储，所以在跨平台的数据转移中会很方便。在备份和恢复时可单独针对某个表进行操作。	免费的方案可以是拷贝数据文件、备份 binlog，或者用 mysqldump，在数据量达到几十G的时候就相对痛苦了。
事务支持	强调的是性能，每次查询具有原子性,其执行数度比InnoDB类型更快，但是不提供事务支持。	提供事务支持事务，外部键等高级数据库功能。 具有事务(commit)、回滚(rollback)和崩溃修复能力(crash recovery capabilities)的事务安全(transaction-safe (ACID compliant))型表。
AUTO_INCREMENT	可以和其他字段一起建立联合索引。引擎的自动增长列必须是索引，如果是组合索引，自动增长可以不是第一列，他可以根据前面几列进行排序后递增。	InnoDB中必须包含只有该字段的索引。引擎的自动增长列必须是索引，如果是组合索引也必须是组合索引的第一列。
表锁差异	只支持表级锁，用户在操作myisam表时，select，update，delete，insert语句都会给表自动加锁，如果加锁以后的表满足insert并发的情况下，可以在表的尾部插入新的数据。	支持事务和行级锁，是innodb的最大特色。行锁大幅度提高了多用户并发操作的新能。但是InnoDB的行锁，只是在WHERE的主键是有效的，非主键的WHERE都会锁全表的。
全文索引	支持 FULLTEXT类型的全文索引	不支持FULLTEXT类型的全文索引，但是innodb可以使用sphinx插件支持全文索引，并且效果更好。
表主键	允许没有任何索引和主键的表存在，索引都是保存行的地址。	如果没有设定主键或者非空唯一索引，就会自动生成一个6字节的主键(用户不可见)，数据是主索引的一部分，附加索引保存的是主索引的值。
表的具体行数	保存有表的总行数，如果select count(*) from table;会直接取出出该值。	没有保存表的总行数，如果使用select count(*) from table；就会遍历整个表，消耗相当大，但是在加了wehre条件后，myisam和innodb处理的方式都一样。
CURD操作	如果执行大量的SELECT，MyISAM是更好的选择。	如果你的数据执行大量的INSERT或UPDATE，出于性能方面的考虑，应该使用InnoDB表。DELETE 从性能上InnoDB更优，但DELETE FROM table时，InnoDB不会重新建立表，而是一行一行的删除，在innodb上如果要清空保存有大量数据的表，最好使用truncate table这个命令。
外键	不支持	支持

适用场景

MyISAM适合：
（1）做很多count 的计算；
（2）插入不频繁，查询非常频繁，如果执行大量的SELECT，MyISAM是更好的选择；
（3）没有事务。
(4) 硬件有限制

InnoDB适合：
（1）可靠性要求比较高，或者要求事务；
（2）表更新和查询都相当的频繁，并且表锁定的机会比较大的情况指定数据引擎的创建；
（3）如果你的数据执行大量的INSERT或UPDATE，出于性能方面的考虑，应该使用InnoDB表；
（4）DELETE FROM table时，InnoDB不会重新建立表，而是一行一行的 删除；
（5）并发性高

其他

主从复制原理，作用实现

水平切分与垂直切分

与NoSql的比较