数据库事务及其他

一、事务

1. 事务简介

事务主要用于处理操作量大、复杂度高、并且关联性强的数据。

比如说, 在人员管理系统中, 你删除一个人员, 你即需要删除人员的基本资料, 也要删除和该人员相关的信息, 如信箱, 文章等等, 这样, 这些数据库操作语句就构成一个事务!

在 MySQL 中只有 Innodb 存储引擎支持事务。

事务处理可以用来维护数据库的完整性, 保证成批的 SQL 语句要么全部执行, 要么全部不执行。主要针对 insert, update, delete 语句而设置

2. 事务四大特性

在写入或更新资料的过程中, 为保证事务 (transaction) 是正确可靠的, 所必须具备的四个特性 (ACID):

1. 原子性 (Atomicity):
    - 事务中的所有操作, 要么全部完成, 要么全部不完成, 不会结束在中间某个环节。
    - 事务在执行过程中发生错误, 会被回滚 (Rollback) 到事务开始前的状态, 就像这个事务从来没有执行过一样。
2. 一致性 (Consistency):

    在事务开始之前和事务结束以后, 数据库的完整性没有被破坏。
    这表示写入的资料必须完全符合所有的预设规则, 这包含资料的精确度、串联性以及后续数据库可以自发性地完成预定的工作。
3. 隔离性 (Isolation):

    数据库允许多个并发事务同时对其数据进行读写和修改的能力, 隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。
    事务隔离分为不同级别, 包括:

    1. 读取未提交 (Read uncommitted)
        - 所有事务都可以看到其他未提交事务的执行结果
        - 本隔离级别很少用于实际应用，因为它的性能也不比其他级别好多少
        - 该级别引发的问题是——脏读(Dirty Read)：读取到了未提交的数据
    2. 读提交 (read committed)
        - 这是大多数数据库系统的默认隔离级别（但不是MySQL默认的）
        - 它满足了隔离的简单定义：一个事务只能看见已经提交事务做的改变
        - 这种隔离级别出现的问题是: 不可重复读(Nonrepeatable Read):

            不可重复读意味着我们在同一个事务中执行完全相同的 select 语句时可能看到不一样的结果。

导致这种情况的原因可能有：

- 有一个交叉的事务有新的commit，导致了数据的改变;
- 一个数据库被多个实例操作时,同一事务的其他实例在该实例处理其间可能会有新的commit

3. 可重复读 (repeatable read)
    - 这是MySQL的默认事务隔离级别
    - 它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行
    - 此级别可能出现的问题: 幻读(Phantom Read)：当用户读取某一范围的数据行时，另一个事务又在该范围内插入了新行，当用户再读取该范围的数据行时，会发现有新的“幻影” 行
    - InnoDB 通过多版本并发控制 (MVCC，Multiversion Concurrency Control) 机制解决幻读问题；
    - InnoDB 还通过间隙锁解决幻读问题
4. 串行化 (Serializable)
    - 这是最高的隔离级别
    - 它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简言之,它是在每个读的数据行上加上共享锁。MySQL锁总结
    - 在这个级别，可能导致大量的超时现象和锁竞争

4. 持久性 (Durability)：

    事务处理结束后, 对数据的修改就是永久的, 即便系统故障也不会丢失。

3. 语法与使用

- 开启事务: `BEGIN` 或 `START TRANSACTION`
- 提交事务: `COMMIT`, 提交会让所有修改生效
- 回滚: `ROLLBACK`, 撤销正在进行的所有未提交的修改
- 创建保存点: `SAVEPOINT identifier`
- 删除保存点: `RELEASE SAVEPOINT identifier`
- 把事务回滚到保存点: `ROLLBACK TO identifier`
- 设置事务的隔离级别: `SET TRANSACTION`

    InnoDB 提供的隔离级别有

    - `READ`
    - `UNCOMMITTED`
    - `READ COMMITTED`
    - `REPEATABLE READ`
    - `SERIALIZABLE`

4. 示例

```
create table `abc` (
    id int unsigned primary key auto_increment,
    name varchar(32) unique,
    age int unsigned
```

```
) charset=utf8;

begin;
insert into abc (name, age) values ('aa', 11);
insert into abc (name, age) values ('bb', 22);
-- 在事务中查看一下数据
-- 同时另开一个窗口，连接到 MySQL 查看一下数据是否一样
select * from abc;
commit;

begin;
insert into abc (name, age) values ('cc', 33);
insert into abc (name, age) values ('dd', 44);
update abc set age=77 where name='aa';
-- 在事务中查看一下数据
select * from abc;
rollback;

select * from abc;  -- 事务结束后在查看一下数据
```

二、锁

锁是计算机协调多个进程或线程并发访问某一资源的机制。

锁保证数据并发访问的一致性、有效性；

锁冲突也是影响数据库并发访问性能的一个重要因素。

锁是Mysql在服务器层和存储引擎层的的并发控制。

分类

- 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。
  - 行级锁只有 InnoDB 引擎支持。
  - 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。
  - 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
  - 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。
  - 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。
- 共享锁 (读锁)
- 其他用户可以并发读取数据，但任何事务都不能对数据进行修改，直到已释放所有共享锁。
- 排他锁 (写锁)
  - 如果事务 T 对数据 A 加上排他锁后，则其他事务不能再对 A 加任任何类型的封锁。

    - 持有排他锁的事务既能读数据，又能修改数据。
- 乐观锁(Optimistic Lock)

  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。 乐观锁不能解决脏读的问题。

  乐观锁, 顾名思义，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，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库如果提供类似于write_condition机制的其实都是提供的乐观锁。

- 悲观锁(Pessimistic Lock)

  假定会发生并发冲突，屏蔽一切可能违反数据完整性的操作。

  悲观锁，顾名思义，就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会block直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

三、存储过程

存储过程（Stored Procedure）是一种在数据库中存储复杂程序，以便外部程序调用的一种数据库对象。

存储过程是为了完成特定功能的SQL语句集，经编译创建并保存在数据库中，用户可通过指定存储过程的名字并给定参数(需要时)来调用执行。

存储过程思想上很简单，就是数据库 SQL 语言层面的代码封装与重用。

1. 优点
    - 存储过程可封装，并隐藏复杂的商业逻辑。
    - 存储过程可以回传值，并可以接受参数。
    - 存储过程无法使用 SELECT 指令来运行，因为它是子程序，与查看表，数据表或用户定义函数不同。
    - 存储过程可以用在数据检验，强制实行商业逻辑等。
2. 缺点
    - 存储过程，往往定制化于特定的数据库上，因为支持的编程语言不同。当切换到其他厂商的数据库系统时，需要重写原有的存储过程。
    - 存储过程的性能调校与撰写，受限于各种数据库系统。

语法

1. 声明语句结束符，可以自定义:

   存储过程中有很多的SQL语句，SQL语句的后面为了保证语法结构必须要有分号（;），但是默认情况下分号表示客户端代码发送到服务器执行。必须更改结束符

   ```
   DELIMITER $$
   -- 或者
   DELIMITER //
   ```

2. 声明存储过程:

```
CREATE PROCEDURE demo_in_parameter(IN p_in int)
```

3. 存储过程开始和结束符号:

```
BEGIN .... END
```

4. 变量赋值:

```
SET @p_in=1
```

5. 变量定义:

```
DECLARE l_int int unsigned default 4000000;
```

6. 创建mysql存储过程、存储函数:

```
create procedure 存储过程名(参数)
```

7. 存储过程体:

```
create function 存储函数名(参数)
```

使用

1. 简单用法

```
-- 定义
-- 如果存储过程中就一条SQL语句，begin…end两个关键字可以省略
create procedure get_info()
select * from student;

-- 调用
call get_info();
```

2. 复杂一点的

```
delimiter // -- 定义前，将分隔符改成 //
create procedure foo(in uid int)
begin
select * from student where `id`=uid;
update student set `city`='北京' where `id`=uid;
end//
delimiter;  -- 定义完以后可以将分隔符改回 分号

call foo(3);
```

延伸阅读

- https://www.zhihu.com/question/19749126
- https://segmentfault.com/q/1010000004907411

四、Python操作

1. 安装: `pip install pymysql`
2. 使用

```
import pymysql

db = pymysql.connect(host='localhost',
                     user='user',
                     password='passwd',
                     db='db',
                     charset='utf8')

try:
    with db.cursor() as cursor:
        # 插入
        sql = "INSERT INTO `users` (`email`, `password`) VALUES (%s, %s)"
        cursor.execute(sql, ('webmaster@python.org', 'very-secret'))
    # 需要手动提交才会执行
    db.commit()

    with db.cursor() as cursor:
        # 读取记录
        sql = "SELECT `id`, `password` FROM `users` WHERE `email`=%s"
        cursor.execute(sql, ('webmaster@python.org',))
        result = cursor.fetchone()
        print(result)
finally:
    db.close()
```

五、数据备份与恢复

1. 备份

```
mysqldump -h localhost -u root -p123456 dbname > dbname.sql
```

2. 恢复

```
-h localhost -u root -p123456 dbname < ./dbname.sql
```