课程作业

课程名称 ： **数据库系统原理**
作业次数 ： **作业#8**
学号 ： **21281280**
姓名 ： **柯劲帆**
班级 ： **物联网2101班**
指导老师 ： **郝爽**
修改日期 ： **2024年6月3日**

1. 题目1

> **备份与日志初步实验**
>
> 1. 了解你所使用的数据库平台的单表数据备份和整库备份方法，进行相应备份操作，并尝试利用备份数据在另一个机器上恢复数据，并在实验报告中描述上述过程。
> 2. 掌握数据库日志的概念，并说明数据备份、日志与故障恢复之间的关系。
> 3. 查阅资料，在你所使用的数据库中找到能记录数据修改操作的日志文件，针对某个表执行插入或修改操作，请在相应日志文件中找对应的插入或修改操作日志记录，至少解释其中的一条日志数据样例。

1.1. 备份方法

1.1.1. 备份操作

1. **单表数据备份**：

   使用 `mysqldump` 工具进行单表备份。

   ```
   sudo mysqldump -p DBLab_1 BOOK > BOOK_backup.sql
   ```

   备份文件 `BOOK_backup.sql` 包含 `BOOK` 表的结构和数据。

2. **整库备份**：

   使用 `mysqldump` 工具进行整库备份。

```
sudo mysqldump -p DBLab_1 > DBLab_backup.sql
```

备份文件 `DBLab_backup.sql` 将包含 `DBLab` 数据库的所有表的结构和数据。

1.1.2. 恢复操作：

1. **恢复单表数据**：

   在另一台机器上创建相应的数据库并恢复备份数据。

```
sudo mysql -p DBLab_1 < BOOK_backup.sql
```

2. **恢复整库数据**：

   在另一台机器上创建相应的数据库。

```
create database DBLab_8_1;
-- Query OK, 1 row affected (0.02 sec)
exit;
```

   恢复备份数据。

```
sudo mysql -p DBLab_8_1 < DBLab_backup.sql
```

1.2. 数据库日志的概念

数据库日志的概念：

数据库日志是记录数据库系统中所有事务和数据库操作的文件。它主要有两种类型：

1. **重做日志（Redo Log）**：用于记录所有已提交事务对数据库所做的修改。这些日志在系统崩溃后可以用来重新应用这些修改，从而恢复数据库。
2. **撤销日志（Undo Log）**：用于记录未提交事务的反向操作，用于在事务回滚时撤销未完成的事务。

数据备份、日志与故障恢复之间的关系：

- **数据备份**：是对数据库当前状态的一个完整拷贝，用于在发生数据丢失或损坏时进行恢复。
- **日志**：记录了数据库的所有事务操作，可以用于重做或撤销操作。
- **故障恢复**：依靠日志和备份共同完成。在数据库崩溃后，可以先通过最近的备份恢复数据库，然后通过重做日志应用自备份以来的所有事务，以恢复到故障前的最新状态。

1.3. 数据修改操作的日志文件

查找数据修改操作的日志文件：

在 MySQL 中，二进制日志（Binary Log）记录了所有修改数据库数据的操作。

查看二进制日志文件：

1. 启用二进制日志：

   在 MySQL 配置文件中

```
sudo vim /etc/mysql/mysql.conf.d/mysqld.cnf
```

添加以下内容

```
[mysqld]
log-bin=mysql-bin
```

并重启 MySQL

```
sudo systemctl restart mysql
```

2. 执行插入或修改操作：

```
INSERT INTO Faculty (`ID`, `IDCode`, `Password`, RolesID, `Name`,
EngName, Gender)
VALUES (3, 'GD0001', 'GD0001', 3, 'CCC', 'CCC', 1);
-- Query OK, 1 row affected (0.04 sec)
```

3. 查看二进制日志文件：

使用 `mysqlbinlog` 工具查看二进制日志文件：

```
mysqlbinlog mysql-bin.000001
```

输出内容截取：

```
# at 391
#240603 20:37:07 server id 1  end_log_pos 462 CRC32 0x3b5c7ec3
Write_rows: table id 92 flags: STMT_END_F
BINLOG '
87hdZhMBAAAASwAAAIcBAAAAAFwAAAAAAAEACURCTGFiXzhfMQAHRmFjdWx0eQAHAw8PAw8PA
Qj9 AvOC/QL9AigBAQACASFJuihd
87hdZh4BAAAARwAAAM4BAAAAAFwAAAAAAAEAAgAH/wADAAAABgBHRDAwMDEGAEdEMDAwMQMAA
AAD AENDQwMAQ0NDAcN+XDs=' /*!*/;
```

这部分是 `Table_map` 事件，表示 `DBLab_8_1.Faculty` 表被映射为内部表 ID 92。

```
# at 391
#240603 20:37:07 server id 1  end_log_pos 462 CRC32 0x3b5c7ec3
Write_rows: table id 92 flags: STMT_END_F
BINLOG '
87hdZhMBAAAASwAAAIcBAAAAAFwAAAAAAAEACURCTGFiXzhfMQAHRmFjdWx0eQAHAw8PAw8PA
Qj9 AvOC/QL9AigBAQACASFJuihd
87hdZh4BAAAARwAAAM4BAAAAAFwAAAAAAAEAAgAH/wADAAAABgBHRDAwMDEGAEdEMDAwMQMAA
AAD AENDQwMAQ0NDAcN+XDs=' /*!*/;
```

这部分是 `Write_rows` 事件，记录了对 `Faculty` 表的一行插入操作。具体的二进制数据用 `BINLOG` 标记记录。

解释日志数据样例：

- `# at 391`：表示此事件在二进制日志文件中的起始位置为 391。
- `#240603 20:37:07`：表示事件发生的时间是 2024年6月3日20:37:07。

- `server id 1`：表示生成该日志事件的服务器 ID 为 1。
- `end_log_pos 462`：表示此事件在二进制日志文件中的结束位置为 462。
- `CRC32 0x3b5c7ec3`：表示该事件的 CRC32 校验和，用于校验数据的完整性。
- `Write_rows`：表示这是一个写行事件，即插入或更新操作。
- `table id 92`：表示被修改的表的内部 ID 为 92。这个 ID 在之前的 Table_map 事件中定义，与表 `DBLab_8_1.Faculty` 映射。
- `flags: STMT_END_F`：表示该事件的标志，`STMT_END_F` 表示这是一个语句结束标志。
- `BINLOG '...'`：这一部分是编码后的二进制数据，表示实际的行数据。解码这些数据可以恢复被写入或更新的行的内容。

通过这种方式，可以在日志文件中找到特定的插入或修改操作，并解释其中的相关信息。这对于数据库的审计和故障恢复非常有用。

2. 问题2

> **并发控制实验**
>
> 1. 通过取消所用DBMS的查询分析器的自动提交功能，创建两个不同用户，分别登录查询分析器，同时打开两个客户端；
> 2. 通过SQL语言设计具体例子展示不同隔离级别的应用场景，验证各种隔离级别的并发控制效果，即是否存在并发操作带来的数据不一致问题，包括丢失修改、不可重复读和读“脏”数据等。

2.1. 创建用户和赋予权限

在 MySQL 中已有两个用户，分别是 `root` 和 `kejingfan`：

```
CREATE USER 'kejingfan'@'%' IDENTIFIED BY 'xxxxxxxx';
GRANT ALL PRIVILEGES ON *.* TO 'kejingfan'@'%';
FLUSH PRIVILEGES;
```

然后，在两个不同的客户端中分别以 `root` 和 `kejingfan` 登录：

```
# 终端 1 IP: 192.168.31.8
sudo mysql -p

# 终端 2 IP: 192.168.31.16
mysql -u kejingfan -h 192.168.31.8 -p
```

取消自动提交功能：

```
SET autocommit=0;
```

2.2. 隔离级别

创建示例表并插入数据：

```sql
DROP TABLE IF EXISTS acounts;

CREATE TABLE accounts (
    id INT PRIMARY KEY,
    balance DECIMAL(10, 2)
);

INSERT INTO accounts (id, balance) VALUES (1, 1000.00), (2, 2000.00);
```

演示不同隔离级别：

2.2.1. 读未提交（READ UNCOMMITTED）

读“脏”数据：

1. **用户1**：设置隔离级别为读未提交，开始一个事务，修改数据但不提交。

```sql
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
-- Query OK, 0 rows affected (0.01 sec)
START TRANSACTION;
-- Query OK, 0 rows affected (0.04 sec)
UPDATE accounts SET balance = 500.00 WHERE id = 1;
-- Query OK, 0 rows affected (0.00 sec)
-- Rows matched: 1  Changed: 0  Warnings: 0
```

2. **用户2**：在用户1未提交的情况下读取数据。

```sql
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
-- Query OK, 0 rows affected (0.01 sec)
START TRANSACTION;
-- Query OK, 0 rows affected (0.00 sec)
SELECT * FROM accounts WHERE id = 1;
-- +----+---------+
-- | id | balance |
-- +----+---------+
-- |  1 |  500.00 |
-- +----+---------+
-- 1 row in set (0.00 sec)
```

用户2可以看到用户1未提交的修改，这就是读“脏”数据。

2.2.2. 读已提交（READ COMMITTED）

不可重复读：

1. **用户1**：设置隔离级别为读已提交，开始一个事务，读取数据。

```
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
-- Query OK, 0 rows affected (0.00 sec)
START TRANSACTION;
-- Query OK, 0 rows affected (0.01 sec)
SELECT * FROM accounts WHERE id = 1;
-- +----+---------+
-- | id | balance |
-- +----+---------+
-- |  1 |  500.00 |
-- +----+---------+
-- 1 row in set (0.01 sec)
```

2. **用户2**：修改数据，不提交。

```
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
-- Query OK, 0 rows affected (0.01 sec)
START TRANSACTION;
-- Query OK, 0 rows affected (0.00 sec)
UPDATE accounts SET balance = 800.00 WHERE id = 1;
-- Query OK, 1 row affected (0.00 sec)
-- Rows matched: 1  Changed: 1  Warnings: 0
```

3. **用户1**：再次读取相同数据，发现数据未被修改。

```
SELECT * FROM accounts WHERE id = 1;
+----+---------+
| id | balance |
+----+---------+
|  1 |  500.00 |
+----+---------+
1 row in set (0.01 sec)
```

4. **用户2**：提交数据。

```
COMMIT;
-- Query OK, 0 rows affected (0.03 sec)
```

5. **用户1**：再次读取相同数据，发现数据被修改。

```
SELECT * FROM accounts WHERE id = 1;
-- +----+---------+
-- | id | balance |
-- +----+---------+
-- |  1 |  800.00 |
-- +----+---------+
-- 1 row in set (0.00 sec)
```

用户1连续两次读取数据，发现读取的数据不一致，体现出不可重复读。

2.2.3. 可重复读（REPEATABLE READ）

幻读：

1. **用户1**：设置隔离级别为可重复读，开始一个事务，读取数据。

```
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
-- Query OK, 0 rows affected (0.00 sec)
START TRANSACTION;
-- Query OK, 0 rows affected (0.00 sec)
SELECT * FROM accounts;
-- +----+---------+
-- | id | balance |
-- +----+---------+
-- |  1 |  800.00 |
-- |  2 | 2000.00 |
-- +----+---------+
-- 2 rows in set (0.00 sec)
```

2. **用户2**：插入一条新记录。

```
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
-- Query OK, 0 rows affected (0.00 sec)
START TRANSACTION;
-- Query OK, 0 rows affected (0.01 sec)
INSERT INTO accounts (id, balance) VALUES (3, 1500.00);
-- Query OK, 1 row affected (0.00 sec)
```

3. **用户1**：再次读取数据，发现新记录未出现。

```
SELECT * FROM accounts;
-- +----+---------+
-- | id | balance |
-- +----+---------+
-- |  1 |  800.00 |
-- |  2 | 2000.00 |
-- +----+---------+
-- 2 rows in set (0.00 sec)
```

4. **用户2**：提交。

```
COMMIT;
-- Query OK, 0 rows affected (0.02 sec)
```

5. **用户1**：读取。

```
SELECT * FROM accounts;
-- +----+---------+
-- | id | balance |
-- +----+---------+
-- |  1 |  800.00 |
-- |  2 | 2000.00 |
-- +----+---------+
```

```
-- 2 rows in set (0.00 sec)
COMMIT;
-- Query OK, 0 rows affected (0.00 sec)
SELECT * FROM accounts;
-- +----+---------+
-- | id | balance |
-- +----+---------+
-- |  1 |  800.00 |
-- |  2 | 2000.00 |
-- |  3 | 1500.00 |
-- +----+---------+
-- 3 rows in set (0.00 sec)
```

我们预期的幻读并没有出现，这是因为MySQL对幻读问题有修复。

> 如果出现幻读，情况将会是：
>
> 在用户2 `COMMIT;` 之后，无需用户2 `COMMIT;` 操作，用户2就能看到 `accounts` 表中的行数量发生了变化。

2.2.4. 串行化（SERIALIZABLE）

可以防止所有并发问题：

1. **用户1**：设置隔离级别为串行化，开始一个事务，读取数据。

```
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
-- Query OK, 0 rows affected (0.00 sec)
START TRANSACTION;
-- Query OK, 0 rows affected (0.00 sec)
SELECT * FROM accounts WHERE id = 1;
-- +----+---------+
-- | id | balance |
-- +----+---------+
-- |  1 |  900.00 |
-- +----+---------+
-- 1 row in set (0.00 sec)
```

2. **用户2**：尝试在用户1未提交的情况下修改数据，未被阻塞。

```
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
-- Query OK, 0 rows affected (0.00 sec)
START TRANSACTION;
-- Query OK, 0 rows affected (0.00 sec)
UPDATE accounts SET balance = 800.00 WHERE id = 1;
-- Query OK, 0 rows affected (0.00 sec)
-- Rows matched: 1  Changed: 0  Warnings: 0
```

3. **用户1**：尝试在用户2未提交的情况下修改数据，被阻塞。

```
UPDATE accounts SET balance = 700.00 WHERE id = 1;
-- 被阻塞
```

4. **用户2**：提交。

```
COMMIT;
-- Query OK, 0 rows affected (0.04 sec)
```

5. **用户1**：阻塞被放行，查询数据，结果为自己修改的值。

```
-- UPDATE accounts SET balance = 700.00 WHERE id = 1;
-- Query OK, 1 row affected (45.56 sec)
-- Rows matched: 1  Changed: 1  Warnings: 0
SELECT * FROM accounts WHERE id = 1;
-- +----+---------+
-- | id | balance |
-- +----+---------+
-- |  1 |  700.00 |
-- +----+---------+
-- 1 row in set (0.00 sec)
```

6. **用户2**：（此时用户1在第3步中的修改操作还未提交）尝试在用户1未提交的情况下查询数据，被阻塞。

```
SELECT * FROM accounts WHERE id = 1;
-- ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting
transaction
```

串行化可以防止所有并发问题，但性能最差，因为操作会被阻塞。

> **总结：**
>
> 不同的隔离级别可以解决不同的并发问题：
>
> - 读未提交：可能会读到“脏”数据。
> - 读已提交：可能会出现不可重复读。
> - 可重复读：可能会出现幻读。
> - 串行化：防止所有并发问题，但性能最差。
>
> 这些隔离级别和并发控制措施确保了数据库在并发环境下的数据一致性和完整性。