一、什么是事务

Transaction 事务

现在我们假设几个场景：如果A向B转账。

第一步：A的钱数减去50元，第二步：B的钱数增加50元。

如果在第一步完成后，第二步开始前，断电了.....

再次开启时，A少了50，B并没有增加50，这找谁说理去。

为了数据库的安全性，提出了事务的概念。

首先来看看与事务相关的sql语句：

```
设置隔离级别：
set session transaction_isolation = 'READ-COMMITTED'

开启事务：
begin / start transaction

提交事务：
commit

回滚：
rollback

保存点：
savepoint name

释放保存点：
release savepoint savepoint_name

事务回滚到保存点：
rollback to savepoint_name

设置禁止自动提交：
set autocommit = 0;//禁止自动提交
set autocommit = 1;//开启自动提交
```

再来看看常见的事务模型：

只读事务：

```
start transaction;
select a from table_name;
commit;
```

写事务：

```
start transaction;
select a from table_name;
insert into table_name(a,b) values(1,1);
commit;
```

自动提交事务：

```
set autocommit = 1;
insert into table_name(a,b) values(1,1);
```

Tips：mysql默认配置自动提交，oralce默认为非自动提交

隐式提交事务：

```
set autocommit = 0;   （mysql默认配置自动提交）
insert into table_name(a,b) values(1,1);
create table table_name_2(a int key, b int);
```

Tips：DDL语句都具有隐式提交属性。

今天想跟大家一起研究下事务内部到底是怎么实现的，在讲解前我想先抛出个问题： **事务想要做到什么效果?**

按我理解，无非是要做到**可靠性**以及**并发处理。**

可靠性：数据库要保证当insert或update操作时抛异常或者数据库crash的时候需要保障数据的操作前后的一致，想要做到这个，我需要知道我修改之前和修改之后的状态，所以就有了undo log和redo log。

并发处理：也就是说当多个并发请求过来，并且其中有一个请求是对数据修改操作的时候会有影响，为了避免读到脏数据，所以需要对事务之间的读写进行隔离，至于隔离到啥程度得看业务系统的场景了，实现这个就得用MySQL 的隔离级别。

事务的处理机制，就是要保证用户的数据操作对数据是"安全的"。

那么怎样才算是安全的，只有在带着ACID四个性质的事务处理机制是安全的。

那么ACID是如何实现的呢？是由MVCC，锁机制，log日志共同协作完成的。

二、事务特性简介

A原子性

事务要么成功，要么失败

背后的机制是undo log实现的，及逻辑日志，如果操作失败或事务未完成，即可进行回滚操作

C一致性
数据库中的数据从一个一致性状态转变成另一个一致性状态

事务除了ACID还有三个重要的属性：

可串行化：从理论上保证了并发事务的调度等价于一个串行调度，用串行结果必然，满足一致性来表示并发事务的调度带来的结果也是满足于一致性的。

可恢复性：事务不会读到其他事务未提交的内容（避免脏读），那么也可以引申为，事务的提交顺序对数据的一致性没有影响

严格性：发送写操作的事务提交或终止操作高于其他

I隔离性

事务和事务之间不会相互影响

隔离级别是由锁作为底层进行实现的，但是并发太低，此时使用快照隔离实现的MVCC（多版本并发控制）,来提高并发

所以在锁机制（隔离级别）和MVCC的共同作用下，实现了数据库的隔离性。

D持久性

事务对数据库的修改是持久的

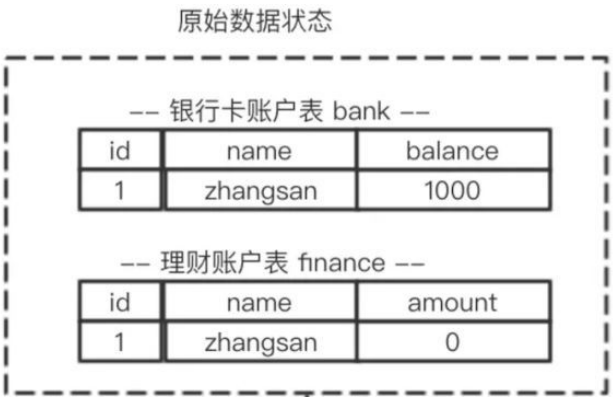
是Redo log机制确保了这个性质，因为数据都是保存在磁盘中的，如果每次commit都去读写磁盘，那么一定会影响程序的并发和效率，所以有了Redo log日志保证数据的持久性。

三、 redo log 与 undo log介绍

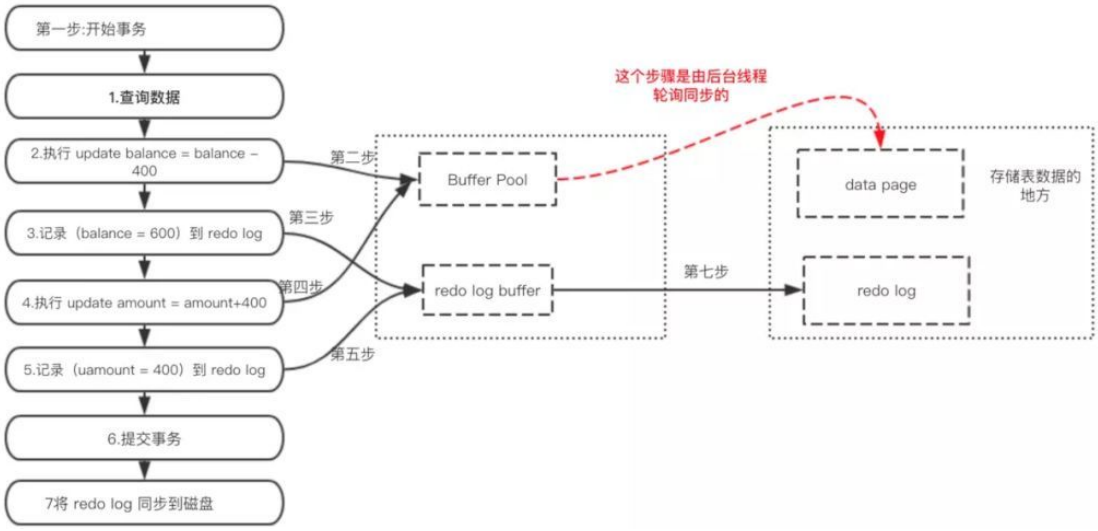
1. redo log

什么是redo log？

redo log叫做**重做**日志，是用来实现事务的持久性。该日志文件由两部分组成：重做日志缓冲（redo log buffer）以及重做日志文件（redo log）,前者是在内存中，后者在磁盘中。当**事务提交之后**会把所有修改信息都会存到该日志中。假设有个表叫做tb1(id,username) 现在要插入数据（3，ceshi）




```
start transaction;
select balance from bank where name="zhangsan";
// 生成 重做日志 balance=600
update bank set balance = balance - 400;
// 生成 重做日志 amount=400
update finance set amount = amount + 400;
commit;
```



redo log 有什么作用?

mysql 为了提升性能不会把每次的修改都实时同步到磁盘，而是会先存到Boffer Pool(缓冲池)里头，把这个当作缓存来用。然后使用后台线程去做**缓冲池和磁盘之间的同步。**

那么问题来了，如果还没来的同步的时候宕机或断电了怎么办？还没来得及执行上面图中红色的操作。这样会导致丢部分已提交事务的修改信息！

所以引入了redo log来记录已成功提交事务的修改信息，并且会把redo log持久化到磁盘，系统重启之后在读取redo log恢复最新数据。

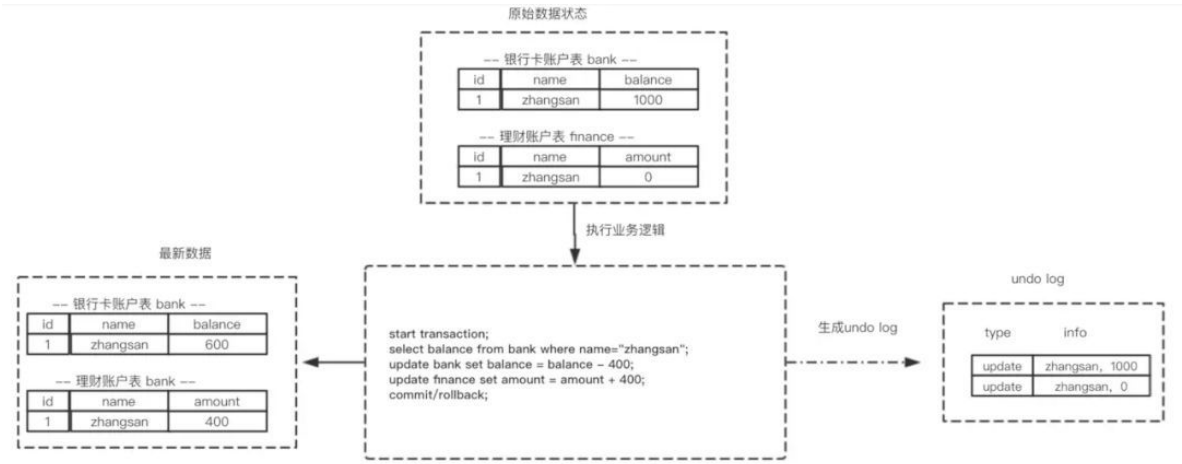
总结： redo log是用来恢复数据的 用于保障，已提交事务的持久化特性

2.undo log

什么是 undo log ?

undo log 叫做回滚日志，用于记录数据**被修改前**的信息。他正好跟前面所说的重做日志所记录的相反，重做日志记录数据被修改后的信息。undo log主要记录的是数据的逻辑变化，为了在发生错误时回滚之前的操作，需要将之前的操作都记录下来，然后在发生错误时才可以回滚。

还用上面那两张表



每次写入数据或者修改数据之前都会把修改前的信息记录到 undo log。

undo log 有什么作用?

undo log 记录事务修改之前版本的数据信息，因此假如由于系统错误或者rollback操作而回滚的话可以根据undo log的信息来进行回滚到没被修改前的状态。

总结： undo log是用来回滚数据的用于保障 未提交事务的原子性

四、mysql锁技术以及MVCC基础

1. mysql锁技术

当有多个请求来读取表中的数据时可以不采取任何操作，但是多个请求里有读请求，又有修改请求时必须有一种措施来进行并发控制。不然很有可能会造成不一致。 **读写锁** 解决上述问题很简单，只需用两种锁的组合来对读写请求进行控制即可，这两种锁被称为：

共享锁(shared lock),又叫做"读锁" 读锁是可以共享的，或者说多个读请求可以共享一把锁读数据，不会造成阻塞。

排他锁(exclusive lock),又叫做"写锁" 写锁会排斥其他所有获取锁的请求，一直阻塞，直到写入完成释放锁。

	读锁(S)	**写锁（X)**
读锁	并行	不可并行
写锁	不可并行	不可并行

总结： 通过读写锁，可以做到读读可以并行，但是不能做到写读，写写并行 事务的隔离性就是根据读写锁来实现的！！！这个后面再说。

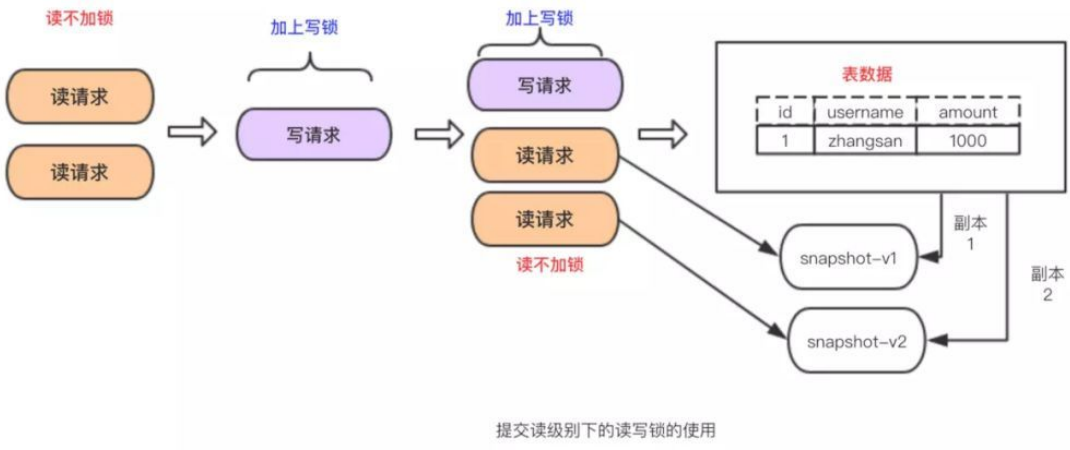
2. MVCC基础

MVCC (MultiVersion Concurrency Control) 叫做多版本并发控制。

他的主要实现思想是通过**数据多版本**来做到**读写分离**。从而实现不加锁读进而做到读写并行。

MVCC在mysql中的实现依赖的是undo log与read view

- undo log :undo log 中记录某行数据的多个版本的数据。
- read view :用来判断当前版本数据的可见性



提交读级别下的读写锁的使用

3. 表锁类型矩阵

INNODB存储引擎的表锁主要分为如下几种：

1、LOCK_IS 意向共享锁

2、LOCK_IX 意向排他锁

3、LOCK_S 共享锁

4、LOCK_X 排他锁

5、LOCK_AUTO_INC 自增长锁，含有自增长列的表才会加该类型的锁。

1、2两种类型的锁在表锁上用的比较频繁，3、4类型的锁只有在lock table语句里才会用到。

上述锁之间的矩阵关系如下图：

	LOCK_IS	LOCK_IX	LOCK_S	LOCK_X	LOCK_AI
LOCK_IS	+	+	+	-	+
LOCK_IX	+	+	-	-	+
LOCK_S	+	-	+	-	-
LOCK_X	-	-	-	-	-
LOCK_AI	+	+	-	-	-

注意，“+”号代表兼容，“-”号代表互斥

上述锁之间的权重关系如下图：

col / row	LOCK_IS	LOCK_IX	LOCK_S	LOCK_X	LOCK_AI
LOCK_IS	+	−	−	−	−
LOCK_IX	+	+	−	−	−
LOCK_S	+	−	+	−	−
LOCK_X	+	+	+	+	+
LOCK_AI	−	−	−	−	+

注意：

“+”号代表：row的权重 >= col的权重

“-”号代表：row的权重 < col的权重

4. 行锁类型矩阵

行锁总的类型可以分为如下两种：

LOCK_S：共享锁

LOCK_X：排他锁

其中每一种类型又可以细分为如下几种类型：

LOCK_GAP：间隙锁，锁住前一条记录到该记录之间的间隙，不包括前一条记录和本条记录。

LOCK_REC_NOT_GAP：记录锁，只锁住本条记录

LOCK_ORDINARY：next-key锁，锁住间隙+本条记录，不包括前一条记录

LOCK_INSERT_INTENTION：插入意向锁，具体由LOCK_GAP锁打上插入意向标记实现。

上述各种锁之间的矩阵关系如下：

已加类型 / 待加类型	LOCK_S_GAP	LOCK_S_REC_NOT_GAP	LOCK_S_ORDINARY	LOCK_S_INSERT_INTENTION	LOCK_X_GAP	LOCK_X_REC_NOT_GAP	LOCK_X_ORDINARY	LOCK_X_INSERT_INTENTION
LOCK_S_GAP	+	+	+	+	+	+	+	+
LOCK_S_REC_NOT_GAP	+	+	+	+	+	−	−	+
LOCK_S_ORDINARY	+	+	+	+	+	−	−	+
LOCK_S_INSERT_INTENTION	+	+	+	+	−	+	−	+
LOCK_X_GAP	+	+	+	+	+	+	+	+
LOCK_X_REC_NOT_GAP	+	−	−	+	+	−	−	+
LOCK_X_ORDINARY	+	−	−	+	+	−	−	+
LOCK_X_INSERT_INTENTION	−	+	−	+	−	+	−	+

五、事务的ACID实现

前面讲的重做日志，回滚日志以及锁技术就是实现事务的基础。

- 事务的原子性是通过 undo log 来实现的
- 事务的持久性性是通过 redo log 来实现的
- 事务的隔离性是通过 (读写锁+MVCC)来实现的
- 而事务的终极大 boss **一致性**是通过原子性，持久性，隔离性来实现的！！！

原子性，持久性，隔离性折腾半天的目的也是为了保障数据的一致性！

总之，ACID只是个概念，事务最终目的是要保障数据的可靠性，一致性。

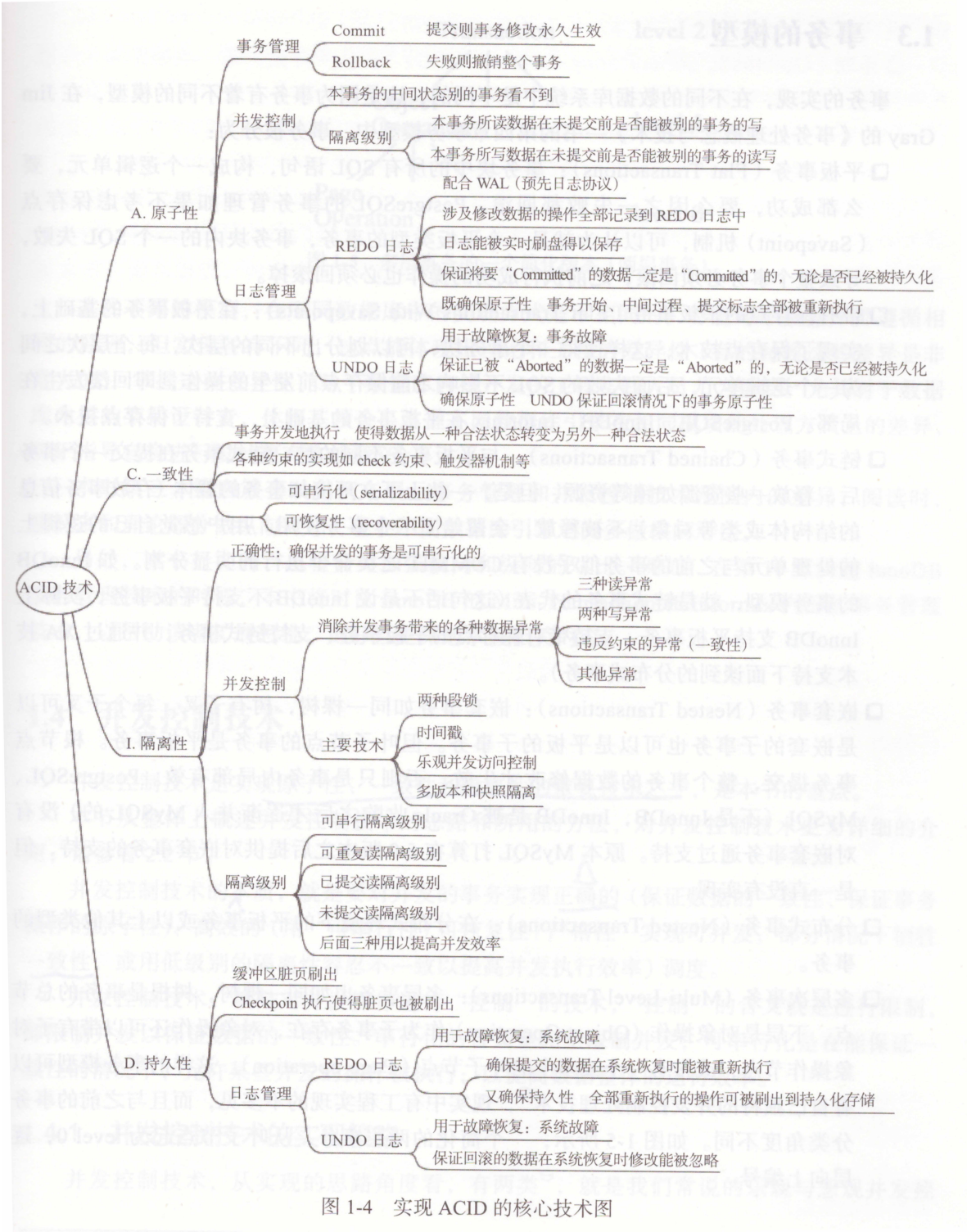


图 1-4 实现 ACID 的核心技术图

1.原子性的实现

什么是原子性：

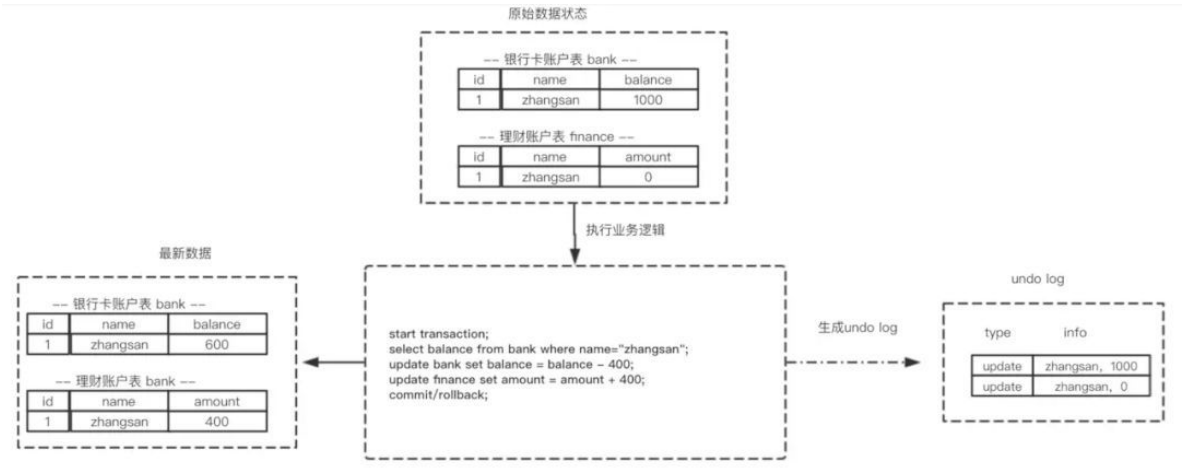
一个事务必须被视为不可分割的最小工作单位，一个事务中的所有操作要么全部成功提交，要么全部失败回滚，对于一个事务来说不可能只执行其中的部分操作，这就是事务的原子性。

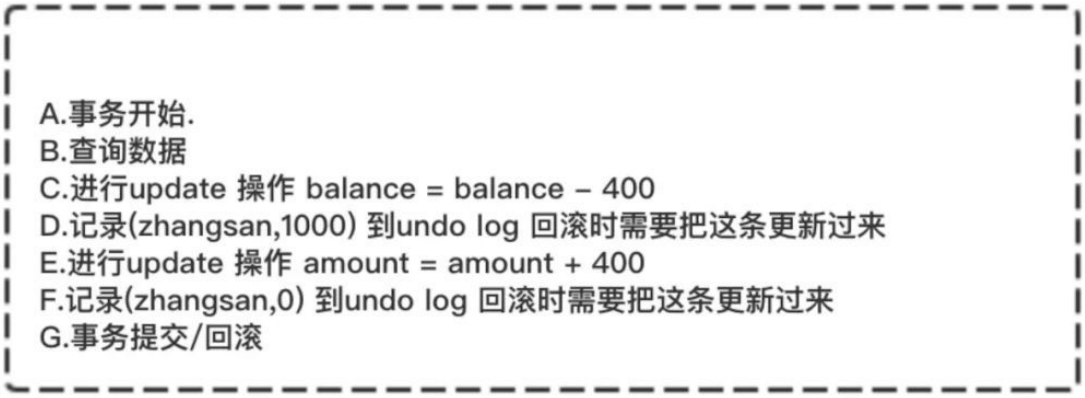
上面这段话取自《高性能MySQL》这本书对原子性的定义，原子性可以概括为就是要实现要么全部失败，要么全部成功。

以上概念相信大家伙儿都了解，那么数据库是怎么实现的呢？ 就是通过**回滚**操作。 所谓回滚操作就是当发生错误异常或者显式的执行rollback语句时需要把数据还原到原先的模样，所以这时候就需要用到undo log来进行回滚，接下来看一下undo log在实现事务原子性时怎么发挥作用的

1.1 undo log 的生成

假设有两个表 bank和finance，表中原始数据如图所示，当进行插入，删除以及更新操作时生成的undo log如下面图所示：





从上图可以了解到数据的变更都伴随着回滚日志的产生： (1) 产生了被修改前数据(zhangsan,1000) 的回滚日志

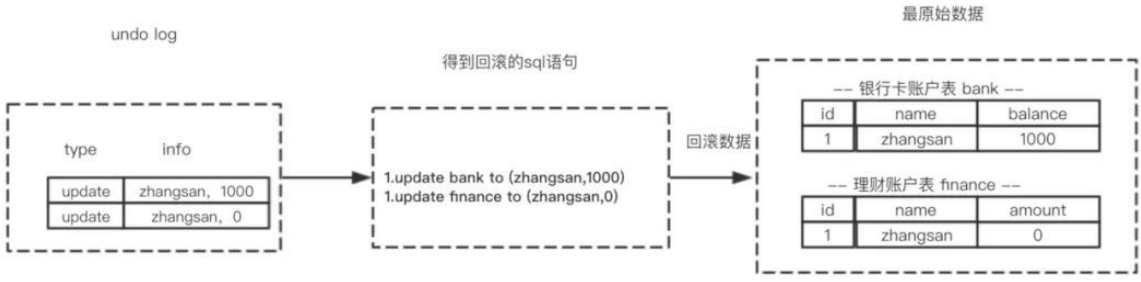
(2) 产生了被修改前数据(zhangsan,0) 的回滚日志

根据上面流程可以得出如下结论： **1.每条数据变更(insert/update/delete)操作都伴随一条undo log的生成,并且回滚日志必须先于数据持久化到磁盘上 2.所谓的回滚就是根据回滚日志做逆向操作，比如delete的逆向操作为insert，insert的逆向操作为delete，update的逆向为update等。**

思考：为什么先写日志后写数据库？ ---稍后做解释

1.2 根据undo log 进行回滚

为了做到同时成功或者失败，当系统发生错误或者执行rollback操作时需要根据undo log 进行回滚



回滚操作就是要还原到原来的状态，undo log记录了数据被修改前的信息以及新增和被删除的数据信息，根据undo log生成回滚语句，比如：

(1) 如果在回滚日志里有新增数据记录，则生成删除该条的语句

(2) 如果在回滚日志里有删除数据记录，则生成生成该条的语句

(3) 如果在回滚日志里有修改数据记录，则生成修改到原先数据的语句

2.持久性的实现

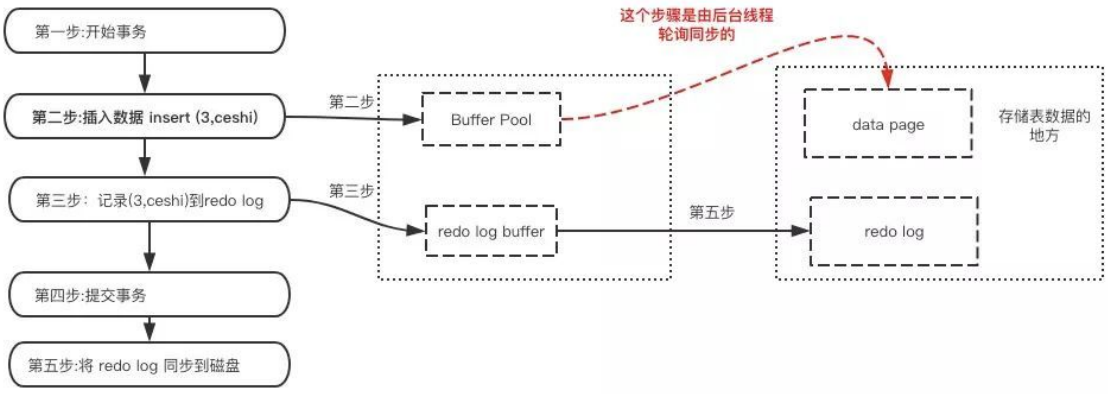
事务一旦提交，其所作做的修改会永久保存到数据库中，此时即使系统崩溃修改的数据也不会丢失。

先了解一下MySQL的数据存储机制，MySQL的表数据是存放在磁盘上的，因此想要存取的时候都要经历磁盘IO,然而即使是使用SSD磁盘IO也是非常消耗性能的。 为此，为了提升性能InnoDB提供了缓冲池(Buffer Pool)，Buffer Pool中包含了磁盘数据页的映射，可以当做缓存来使用： **读数据**：会首先从缓冲池中读取，如果缓冲池中没有，则从磁盘读取在放入缓冲池； **写数据**：会首先写入缓冲池，缓冲池中的数据会定期同步到磁盘中；

上面这种缓冲池的措施虽然在性能方面带来了质的飞跃，但是它也带来了新的问题，当MySQL系统宕机，断电的时候可能会丢数据！！！

因为我们的数据已经提交了，但此时是在缓冲池里头，还没来得及在磁盘持久化，所以我们急需一种机制需要存一下已提交事务的数据，为恢复数据使用。

于是 redo log就派上用场了。下面看下redo log是什么时候产生的



既然redo log也需要存储，也涉及磁盘IO为啥还用它?

（1）redo log 的存储是顺序存储，而缓存同步是随机操作。

（2）缓存同步是以数据页为单位的，每次传输的数据大小大于redo log。

3.隔离性实现

隔离性是事务ACID特性里最复杂的一个。在SQL标准里定义了四种隔离级别，每一种级别都规定一个事务中的修改，哪些是事务之间可见的，哪些是不可见的。

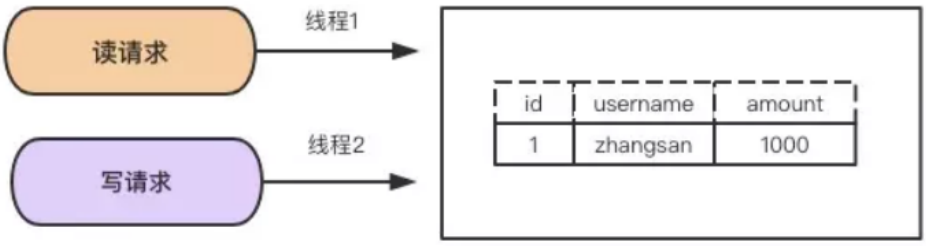
级别越低的隔离级别可以执行越高的并发，但同时实现复杂度以及开销也越大。

Mysql 隔离级别有以下四种（级别由低到高）：

- **READ UNCOMMITED** (未提交读)
- **READ COMMITED** (提交读)
- **REPEATABLE READ** (可重复读)
- **SERIALIZABLE** (可重复读)

只要彻底理解了隔离级别以及他的实现原理就相当于理解了ACID里的隔离型。前面说过原子性，隔离性，持久性的目的都是为了要做到一致性，但隔离型跟其他两个有所区别，原子性和持久性是为了要实现数据的可性保障靠，比如要做到宕机后的恢复，以及错误后的回滚。

那么隔离性是要做到什么呢？ **隔离性是要管理多个并发读写请求的访问顺序。** 这种顺序包括**串行**或者是**并行** 说明一点，写请求不仅仅是指insert操作，又包括update操作。



总之，从隔离性的实现可以看出这是一场数据的可靠性与性能之间的权衡。

- 可靠性性高的，并发性能低(比如 Serializable)
- 可靠性低的，并发性能高(比如 Read Uncommited)

READ UNCOMMITTED

在READ UNCOMMITTED隔离级别下，事务中的修改即使还没提交，对其他事务是可见的。事务可以读取未提交的数据，造成脏读。

因为读不会加任何锁，所以写操作在读的过程中修改数据，所以会造成脏读。好处是可以提升并发处理性能，能做到**读写并行**。

换句话说，读的操作不能排斥写请求。

优点：读写并行，性能高 缺点：造成脏读。

示例：脏读

```
--session 1
begin;
select * from tb1 where a=2;
+---+------+
| a | b    |
+---+------+
| 2 |   2 |
+---+------+
update tb1 set b=20 where a=2;
--session 2
set session tx_isolation='READ-UNCOMMITTED';
begin;
select * from tb1 where a=2;
+---+------+
| a | b    |
+---+------+
| 2 |   20 |
+---+------+
```

READ COMMITTED

一个事务的修改在他提交之前的所有修改，对其他事务都是不可见的。其他事务能读到已提交的修改变化。在很多场景下这种逻辑是可以接受的。

InnoDB在 READ COMMITTED，使用排它锁,读取数据不加锁而是使用了MVCC机制。或者换句话说他采用了**读写分离机制**。 但是该级别会产生**不可重读**以及**幻读**问题。

> 什么是不可重读?

在一个事务内多次读取的结果不一样。

> 为什么会产生不可重复读?

这跟 READ COMMITTED 级别下的MVCC机制有关系，在该隔离级别下每次 select的时候新**生成一个版本号**，所以每次select的时候读的不是一个副本而是不同的副本。

在每次select之间有其他事务**更新**了我们读取的数据并提交了，那就出现了不可重复读

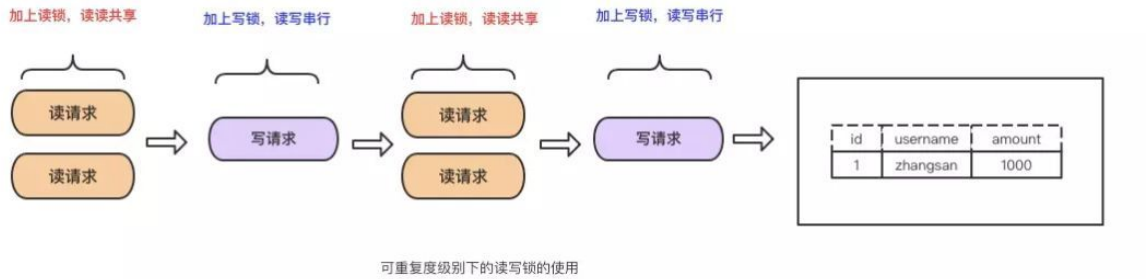
示例：不可重复读问题

```
--session 1
begin;
select * from tb1 where a=2;
+---+------+
| a | b    |
+---+------+
| 2 |   2 |
+---+------+
update tb1 set b=20 where a=2;
--session 2
set session tx_isolation='READ-COMMITTED';
begin;
select * from tb1 where a=2;
+---+------+
| a | b    |
+---+------+
| 2 |   2 |
+---+------+
--session 1
commit;
--session 2
select * from tb1 where a=2;
+---+------+
| a | b    |
+---+------+
| 2 |   20 |
+---+------+
```

REPEATABLE READ(Mysql默认隔离级别)

在一个事务内的多次读取的结果是一样的。这种级别下可以避免，脏读，不可重复读等查询问题。mysql 有两种机制可以达到这种隔离级别的效果，分别是采用读写锁以及MVCC。

采用读写锁实现:



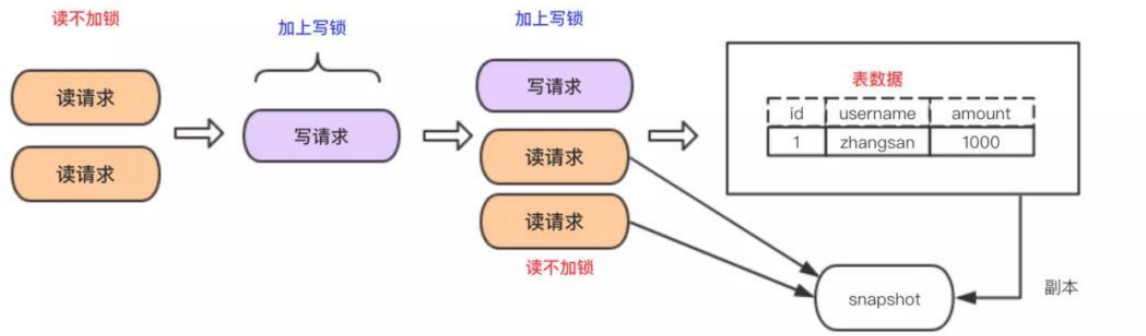
可重复度级别下的读写锁的使用

为什么能可重复度？只要没释放读锁，在次读的时候还是可以读到第一次读的数据。

优点：实现起来简单

缺点：无法做到读写并行

采用MVCC实现：



可重复读级别下的读写锁的使用

为什么能可重复度？因为多次读取只生成一个版本，读到的自然是相同数据。

优点：读写并行

缺点：实现的复杂度高

但是在该隔离级别下仍会存在幻读的问题，

示例：幻读---特指查到新插入的记录

```
--预置数据
create table tb1(a int key, b int);
insert into tb1(a,b) values(1,1),(2,2),(3,3),(4,4),(5,5);
--session 1
set session tx_isolation='READ-COMMITTED';
begin;
select * from tb1;
+---+------+
| a | b    |
+---+------+
| 1 |    1 |
| 2 |    2 |
| 3 |    3 |
| 4 |    4 |
| 5 |    5 |
+---+------+
--session 2
begin;
```

```
select * from tb1;
+---+------+
| a | b    |
+---+------+
| 1 |    1 |
| 2 |    2 |
| 3 |    3 |
| 4 |    4 |
| 5 |    5 |
+---+------+
insert into tb1(a,b) values(6,6);
commit;
--session 1
select * from tb1;  ---产生幻读现象
+---+------+
| a | b    |
+---+------+
| 1 |    1 |
| 2 |    2 |
| 3 |    3 |
| 4 |    4 |
| 5 |    5 |
| 6 |    6 |
+---+------+
```

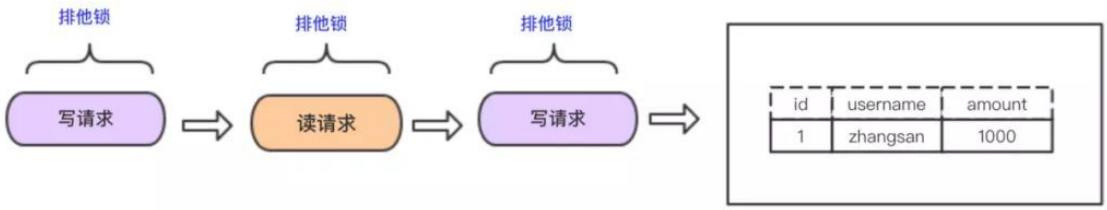
但是mysql在RR隔离级别下，通过GAP锁的方式，解决了幻读的问题。

```
--预置数据
create table tb1(a int key, b int);
insert into tb1(a,b) values(1,1),(2,2),(3,3),(4,4),(5,5);
--session 1
set session tx_isolation='REPEATABLE-READ';
begin;
select * from tb1;
+---+------+
| a | b    |
+---+------+
| 1 |    1 |
| 2 |    2 |
| 3 |    3 |
| 4 |    4 |
| 5 |    5 |
+---+------+
--session 2
begin;
select * from tb1;
+---+------+
| a | b    |
+---+------+
| 1 |    1 |
| 2 |    2 |
| 3 |    3 |
| 4 |    4 |
| 5 |    5 |
+---+------+
insert into tb1(a,b) values(6,6);
commit;
```

```
--session 1
select * from tb1;      ---无幻读现象
+---+------+
| a | b    |
+---+------+
| 1 |    1 |
| 2 |    2 |
| 3 |    3 |
| 4 |    4 |
| 5 |    5 |
+---+------+
```

SERIALIZABLE

该隔离级别理解起来最简单，实现也最单。在该隔离级别下除了不会造成数据不一致问题，没其他优点。



序列化读级别下的读写锁的使用

表1-1：ANSI SQL隔离级别

隔离级别	脏读可能性	不可重复读可能性	幻读可能性	加锁读
READ UNCOMMITTED	Yes	Yes	Yes	No
READ COMMITTED	No	Yes	Yes	No
REPEATABLE READ	No	No	Yes	No
SERIALIZABLE	No	No	No	Yes

4.一致性的实现

> 数据库总是从一个一致性的状态转移到另一个一致性的状态.

下面举个例子:zhangsan 从银行卡转400到理财账户

```
start transaction;
select balance from bank where name="zhangsan";
// 生成 重做日志 balance=600
update bank set balance = balance - 400;
// 生成 重做日志 amount=400
update finance set amount = amount + 400;
commit;
```

1.假如执行完 `update bank set balance = balance - 400;` 之发生异常了，银行卡的钱也不能平白无辜的减少，而是回滚到最初状态。

2.又或者事务提交之后，缓冲池还没同步到磁盘的时候宕机了，这也是不能接受的，应该在重启的时候恢复并持久化。

3.假如有并发事务请求的时候也应该做好事务之间的可见性问题，避免造成脏读，不可重复读，幻读等。在涉及并发的情况下往往在性能和一致性之间做平衡，做一定的取舍，所以隔离性也是对一致性的一种破坏。