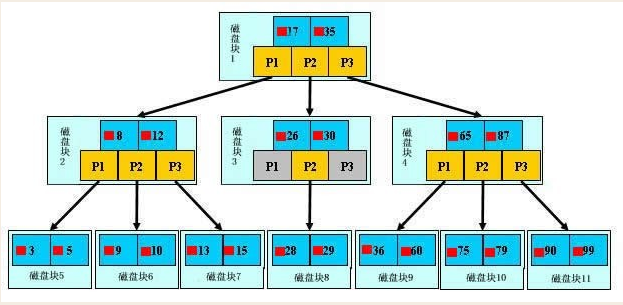
# 索引

https://www.cnblogs.com/aaabbbcccddd/p/14864982.html

## 索引的数据结构



如上图，是一颗b+树，关于b+树的定义可以参见B+树，这里只说一些重点，浅蓝色的块我们称之为一个磁盘块，可以看到每个磁盘块包含几个数据项（深蓝色所示）和指针（黄色所示），如磁盘块1包含数据项17和35，包含指针P1、P2、P3，P1表示小于17的磁盘块，P2表示在17和35之间的磁盘块，P3表示大于35的磁盘块。真实的数据存在于叶子节点即3、5、9、10、13、15、28、29、36、60、75、79、90、99。非叶子节点只不存储真实的数据，只存储指引搜索方向的数据项，如17、35并不真实存在于数据表中。

###b+树的查找过程

如图所示，如果要查找数据项29，那么首先会把磁盘块1由磁盘加载到内存，此时发生一次IO，在内存中用二分查找确定29在17和35之间，锁定磁盘块1的P2指针，内存时间因为非常短（相比磁盘的IO）可以忽略不计，通过磁盘块1的P2指针的磁盘地址把磁盘块3由磁盘加载到内存，发生第二次IO，29在26和30之间，锁定磁盘块3的P2指针，通过指针加载磁盘块8到内存，发生第三次IO，同时内存中做二分查找找到29，结束查询，总计三次IO。真实的情况是，3层的b+树可以表示上百万的数据，如果上百万的数据查找只需要三次IO，性能提高将是巨大的，如果没有索引，每个数据项都要发生一次IO，那么总共需要百万次的IO，显然成本非常非常高。

1. 索引字段要尽量的小：通过上面的分析，我们知道IO次数取决于b+数的高度h，假设当前数据表的数据为N，每个磁盘块的数据项的数量是m，则有h=㏒(m+1)N，当数据量N一定的情况下，m越大，h越小；而m = 磁盘块的大小 / 数据项的大小，磁盘块的大小也就是一个数据页的大小，是固定的，如果数据项占的空间越小，数据项的数量越多，树的高度越低。这就是为什么每个数据项，即索引字段要尽量的小，比如int占4字节，要比bigint8字节少一半。这也是为什么b+树要求把真实的数据放到叶子节点而不是内层节点，一旦放到内层节点，磁盘块的数据项会大幅度下降，导致树增高。当数据项等于1时将会退化成线性表。

数据页之间是双向链表，数据页中的数据是单向链表

## 联合索引

索引的最左匹配特性（即从左往右匹配）：当b+树的数据项是复合的数据结构，比如(name,age,sex)的时候，b+数是按照从左到右的顺序来建立搜索树的，比如当(张三,20,F)这样的数据来检索的时候，b+树会优先比较name来确定下一步的所搜方向，如果name相同再依次比较age和sex，最后得到检索的数据；但当(20,F)这样的没有name的数据来的时候，b+树就不知道下一步该查哪个节点，因为建立搜索树的时候name就是第一个比较因子，必须要先根据name来搜索才能知道下一步去哪里查询。比如当(张三,F)这样的数据来检索时，b+树可以用name来指定搜索方向，但下一个字段age的缺失，所以只能把名字等于张三的数据都找到，然后再匹配性别是F的数据了， 这个是非常重要的性质，即索引的最左匹配特性。

最左前缀匹配原则，非常重要的原则，

create index ix\_name\_email on s1(name,email,)

- 最左前缀匹配：必须按照从左到右的顺序匹配

select \* from s1 where name='egon'; #可以

select \* from s1 where name='egon' and email='asdf'; #可以

select \* from s1 where email='alex@oldboy.com'; #不可以

mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，

比如a = 1 and b = 2 and c > 3 and d = 4 如果建立(a,b,c,d)顺序的索引，

d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。

**总结**：联合索引（name，age），为使索引生效，要保证name在查询条件中。如果查询条件中只有age，没有name，无法命中索引。联合索引是有先后顺序的，比如（a,b,c,d），abcd依次当做索引过滤，当c为范围查询时，d不做为索引过滤。

针对sql：select \* from s1 where id>3 and name='egon' and email='alex333@oldboy.com' and gender='male';

建立索引应该为：create index idx on s1(name,email,gender,id);

## 索引的选择

尽量选择区分度高的列作为索引,区分度的公式是count(distinct col)/count(\*)，表示字段不重复的比例，比例越大我们扫描的记录数越少，唯一键的区分度是1

## 索引失效的情况

- like '%xx'

select \* from tb1 where email like '%cn';

- 使用函数

select \* from tb1 where reverse(email) = 'wupeiqi';

- or

select \* from tb1 where nid = 1 or name = 'seven@live.com';

特别的：当or条件中有未建立索引的列才失效，以下会走索引

select \* from tb1 where nid = 1 or name = 'seven';

select \* from tb1 where nid = 1 or name = 'seven@live.com' and email = 'alex'

- 类型不一致

如果列是字符串类型，传入条件是必须用引号引起来，不然...

select \* from tb1 where email = 999;

普通索引的不等于不会走索引

- !=

select \* from tb1 where email != 'alex'

特别的：如果是主键，则还是会走索引

select \* from tb1 where nid != 123

- >

select \* from tb1 where email > 'alex'

特别的：如果是主键或索引是整数类型，则还是会走索引

select \* from tb1 where nid > 123

select \* from tb1 where num > 123

#排序条件为索引，则select字段必须也是索引字段，否则无法命中

- order by

select name from s1 order by email desc;

当根据索引排序时候，select查询的字段如果不是索引，则不走索引

select email from s1 order by email desc;

特别的：如果对主键排序，则还是走索引：

select \* from tb1 order by nid desc;

- 组合索引最左前缀

如果组合索引为：(name,email)

name and email -- 使用索引

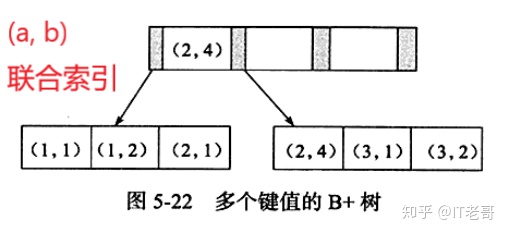
name -- 使用索引

email -- 不使用索引

- count(1)或count(列)代替count(\*)在mysql中没有差别了

- create index xxxx on tb(title(19)) #text类型，必须制定长度

## 联合索引结构



从本质上来说，联合索引也是一个B+树，和单值索引不同的是，联合索引的键值对不是1，而是大于1个。

a, b 排序分析

a顺序：1，1，2，2，3，3

b顺序：1，2，1，4，1，2

可以发现a字段是有序排列，b字段是无序排列（因为B+树只能选一个字段来构建有序的树）

一不小心又会发现，在a相等的情况下，b字段是有序的。

### 联合索引生效

Select \* from table where a = 2 and b = 2;

第一次查询 a = 2，(2,1),(2,2) ,(2,3) ,(2,4) ,(2,5)。B变成了有序数据，可以用二分高效查询。

**总结：**几个列建立的联合索引，同时生效时，对每个列做二分查找。

### 联合索引失效

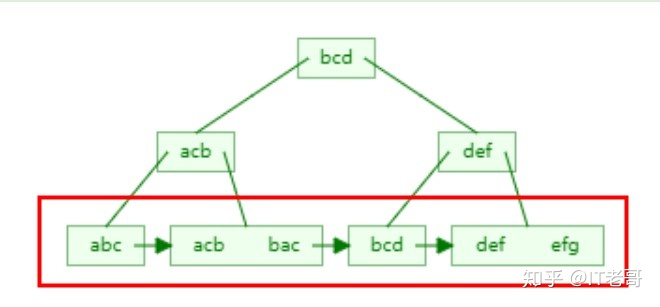
Select \* from table where a > 2 and b = 2;

a > 2 查询可以使用二分，所以命中索引，查询结果：(3,1),(3,2) ,(3,3) ,(4,1) ,(4,2)。

B数据为无序数组 [1,2,3,1,2]，没法用二分查找，只能遍历。

**总结**：联合索引一直生效，直到碰到某一列为范围查询，剩下的列就都不会命中索引，只能遍历。即：遵循最佳左前缀匹配

## %造成索引失效



与联合索引的结构相似，遵循最佳左前缀匹配，第一个字母有序，在第一个字母相同时，第二字母有序，第三个同理。

所以：

a% 能匹配到：abc,acb。能够命中索引。

%a 就等于遍历

%a% 也等于遍历，符合该值的索引能出现在任何位置

## 回表与规避回表

**回表**：普通索引只保存了索引值和主键值，当查询 select \* from table where name = ‘aaa’，当找到name=aaa这条数据的主键值，再使用主键值去主键索引里查 \* 的数据。

所以如果触发回表，不仅要查普通索引的数，还要再查一遍主键索引的数，效率不高。

**规避回表**：创建索引 （name,age），当查询select \* from table where name = ‘aaa’ and age = 20 ，依然要回表。

所以修改sql，select id,name,age from table where name = ‘aaa’ and age = 20。

由于id，name，age在普通索引（name，age）里都有值，不用回表，提高效率。

参考：https://zhuanlan.zhihu.com/p/137647823

# 事务

参考： <https://www.cnblogs.com/ffdsj/p/12266539.html>

<https://zhuanlan.zhihu.com/p/117476959>

## 事务解决的问题

**脏读**

事务A读取了事务B更新的数据，然后B回滚操作，那么A读取到的数据是脏数据。

**可重复读**

可重复读指的是在一个事务内，最开始读到的数据和事务结束前的任意时刻读到的同一批数据都是一致的。通常针对数据更新（UPDATE）操作。

**不可重复读**

对比可重复读，不可重复读指的是在同一事务内，不同的时刻读到的同一批数据可能是不一样的，可能会受到其他事务的影响，比如其他事务改了这批数据并提交了。通常针对数据更新（UPDATE）操作。

**幻读**

幻读是针对数据插入（INSERT）操作来说的。假设事务A对某些行的内容作了更改，但是还未提交，此时事务B插入了与事务A更改前的记录相同的记录行，并且在事务A提交之前先提交了，而这时，在事务A中查询，会发现好像刚刚的更改对于某些数据未起作用，但其实是事务B刚插入进来的，让用户感觉很魔幻，感觉出现了幻觉，这就叫幻读。

**总结**：

脏读，不可重复读，幻读都是当a事务执行中，b事务对a事务操作的数据进行了修改。脏读由回滚引起，不可重复读由修改引起，幻读由新增或删除引起。

不可重复读需要锁行，幻读需要锁表。

## 事务等级

SQL 标准定义了四种隔离级别，MySQL 全都支持。这四种隔离级别分别是：

读未提交（READ UNCOMMITTED）

读提交 （READ COMMITTED）

可重复读 （REPEATABLE READ）

串行化 （SERIALIZABLE）



## 设置事务级别

我们可以通过以下语句查看当前数据库的隔离级别，通过下面语句可以看出我使用的 MySQL 的隔离级别是 REPEATABLE-READ，也就是可重复读，这也是 MySQL 的默认级别。

# 查看事务隔离级别 5.7.20 之后

show variables like 'transaction\_isolation';

SELECT @@transaction\_isolation

# 5.7.20 之后

SELECT @@tx\_isolation

show variables like 'tx\_isolation'

+---------------+-----------------+

| Variable\_name | Value |

+---------------+-----------------+

| tx\_isolation | REPEATABLE-READ |

+---------------+-----------------+

稍后，我们要修改数据库的隔离级别，所以先了解一下具体的修改方式。

修改隔离级别的语句是：set [作用域] transaction isolation level [事务隔离级别]，

SET [SESSION | GLOBAL] TRANSACTION ISOLATION LEVEL {READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE}。

其中作用于可以是 SESSION 或者 GLOBAL，GLOBAL 是全局的，而 SESSION 只针对当前回话窗口。隔离级别是 {READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE} 这四种，不区分大小写。

比如下面这个语句的意思是设置全局隔离级别为读提交级别。

mysql> set global transaction isolation level read committed;

## 执行事务

事务的执行过程如下，以 begin 或者 start transaction 开始，然后执行一系列操作，最后要执行 commit 操作，事务才算结束。当然，如果进行回滚操作(rollback)，事务也会结束。

需要注意的是，begin 命令并不代表事务的开始，事务开始于 begin 命令之后的第一条语句执行的时候。例如下面示例中，select \* from xxx 才是事务的开始，

begin;

select \* from xxx;

commit; -- 或者 rollback;

另外，通过以下语句可以查询当前有多少事务正在运行。

select \* from information\_schema.innodb\_trx;

执行前准备数据：

CREATE TABLE `user` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

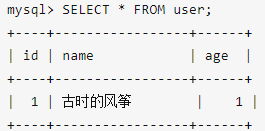
`name` varchar(30) DEFAULT NULL,

`age` tinyint(4) DEFAULT NULL,

PRIMARY KEY (`id`)

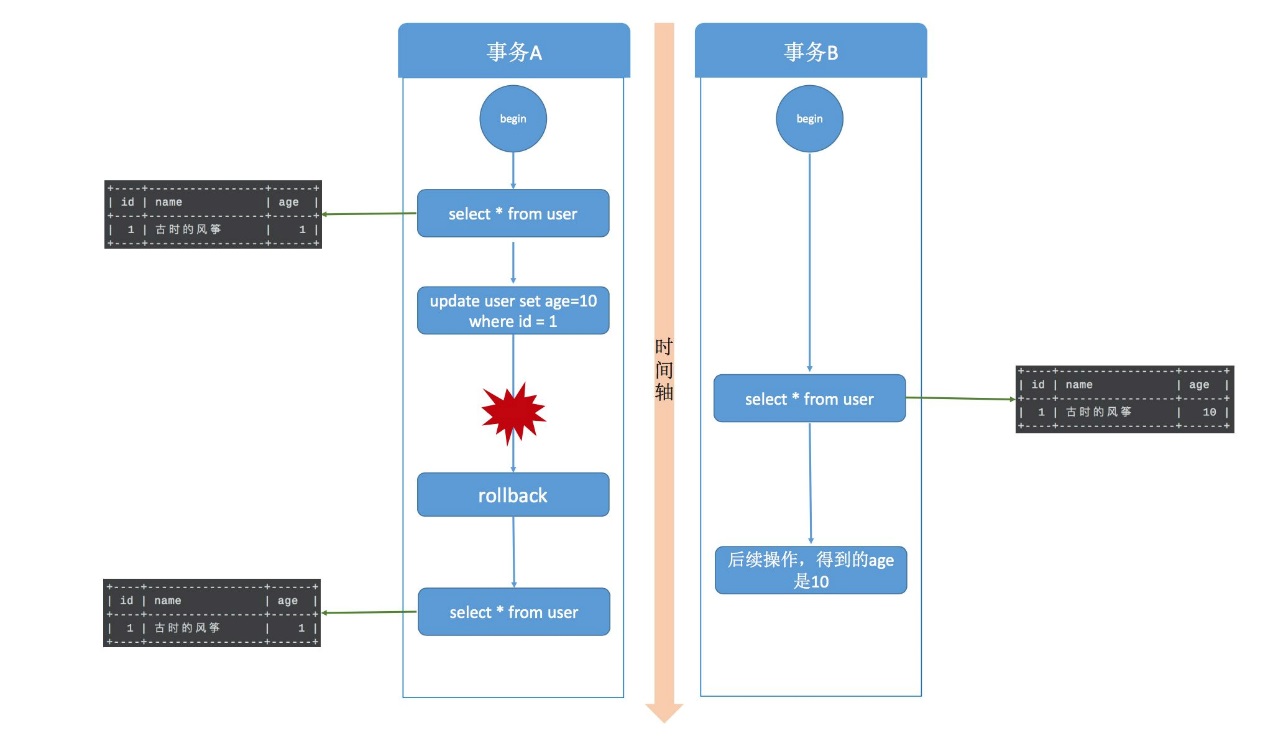
) ENGINE=InnoDB AUTO\_INCREMENT=2 DEFAULT CHARSET=utf8

只有一条数据 insert into user values(1,’古时的风筝’,1)



### 读未提交

启动两个事务，分别为事务A和事务B，在事务A中使用 update 语句，修改 age 的值为10，初始是1 ，在执行完 update 语句之后，在事务B中查询 user 表，会看到 age 的值已经是 10 了，这时候事务A还没有提交，而此时事务B有可能拿着已经修改过的 age=10 去进行其他操作了。在事务B进行操作的过程中，很有可能事务A由于某些原因，进行了事务回滚操作，那其实事务B得到的就是脏数据了，拿着脏数据去进行其他的计算，那结果肯定也是有问题的。

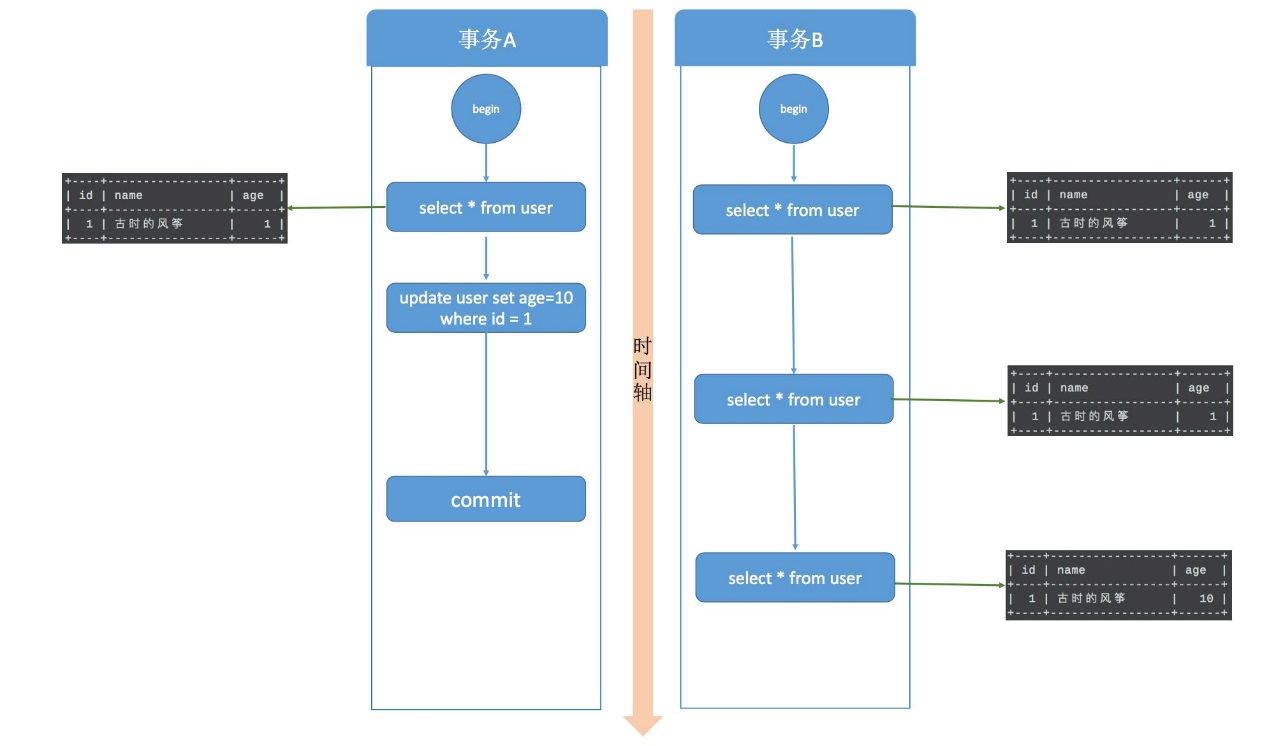


读未提交，其实就是可以读到其他事务未提交的数据，但没有办法保证你读到的数据最终一定是提交后的数据，如果中间发生回滚，那就会出现脏数据问题，读未提交没办法解决脏数据问题。更别提可重复读和幻读了，想都不要想。

### 读提交

同样开启事务A和事务B两个事务，在事务A中使用 update 语句将 id=1 的记录行 age 字段改为 10。此时，在事务B中使用 select 语句进行查询，我们发现在事务A提交之前，事务B中查询到的记录 age 一直是1，直到事务A提交，此时在事务B中 select 查询，发现 age 的值已经是 10 了。

这就出现了一个问题，在同一事务中(本例中的事务B)，事务的不同时刻同样的查询条件，查询出来的记录内容是不一样的，事务A的提交影响了事务B的查询结果，这就是不可重复读，也就是读提交隔离级别。



每个 select 语句都有自己的一份快照，而不是一个事务一份，所以在不同的时刻，查询出来的数据可能是不一致的。

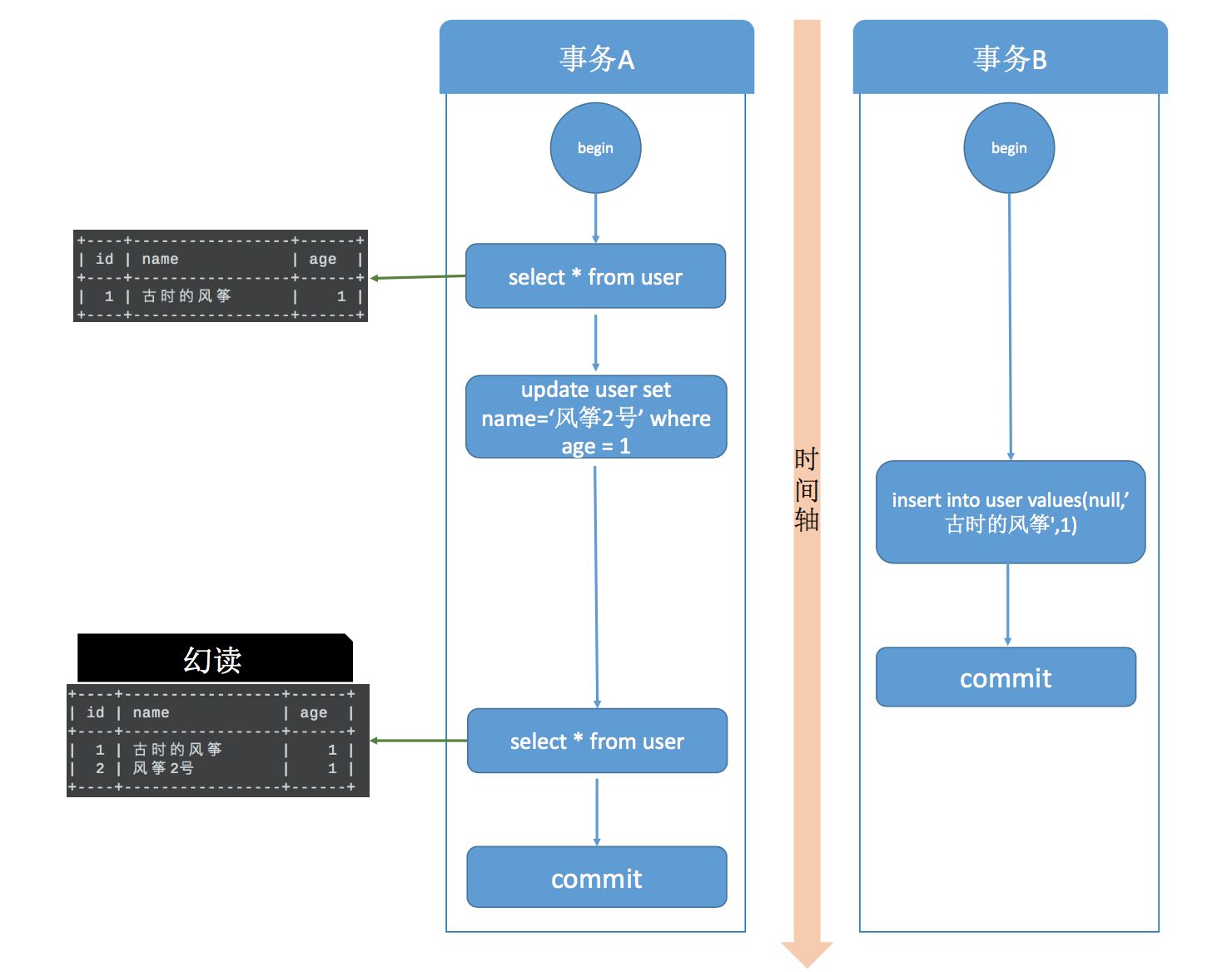
读提交解决了脏读的问题，但是无法做到可重复读，也没办法解决幻读。

### 可重复读

事务A开始后，执行 update 操作，将 age = 1 的记录的 name 改为“风筝2号”；

事务B开始后，在事务执行完 update 后，执行 insert 操作，插入记录 age =1，name = 古时的风筝，这和事务A修改的那条记录值相同，然后提交。

事务B提交后，事务A中执行 select，查询 age=1 的数据，这时，会发现多了一行，并且发现还有一条 name = 古时的风筝，age = 1 的记录，这其实就是事务B刚刚插入的，这就是幻读。



要说明的是，当你在 MySQL 中测试幻读的时候，并不会出现上图的结果，幻读并没有发生，MySQL 的可重复读隔离级别其实解决了幻读问题，这会在后面的内容说明

### 串行化

串行化是4种事务隔离级别中隔离效果最好的，解决了脏读、可重复读、幻读的问题，但是效果最差，它将事务的执行变为顺序执行，与其他三个隔离级别相比，它就相当于单线程，后一个事务的执行必须等待前一个事务结束。

## 事务隔离

首先说读未提交，它是性能最好，也可以说它是最野蛮的方式，因为它压根儿就不加锁，所以根本谈不上什么隔离效果，可以理解为没有隔离。

再来说串行化。读的时候加共享锁，也就是其他事务可以并发读，但是不能写。写的时候加排它锁，其他事务不能并发写也不能并发读。

最后说读提交和可重复读。这两种隔离级别是比较复杂的，既要允许一定的并发，又想要兼顾的解决问题。

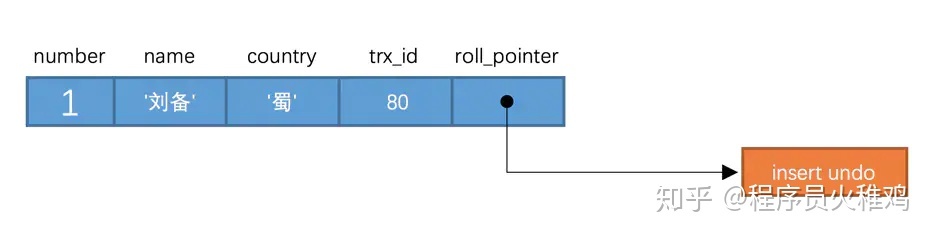
### 版本链

在InnoDB中，每行记录实际上都包含了两个隐藏字段：事务id(trx\_id)和回滚指针(roll\_pointer)。

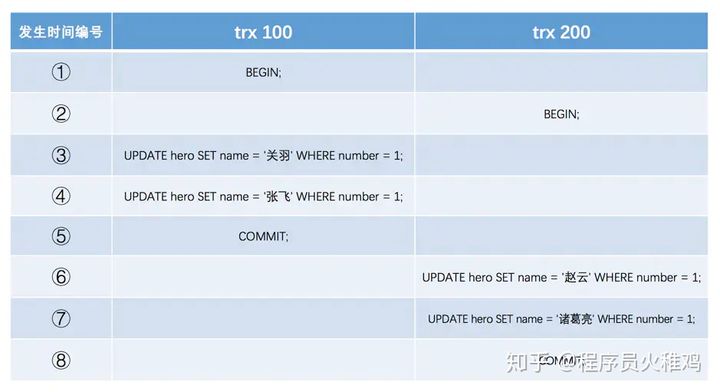
trx\_id：事务id。每次修改某行记录时，都会把该事务的事务id赋值给trx\_id隐藏列。

roll\_pointer：回滚指针。每次修改某行记录时，都会把undo日志地址赋值给roll\_pointer隐藏列。

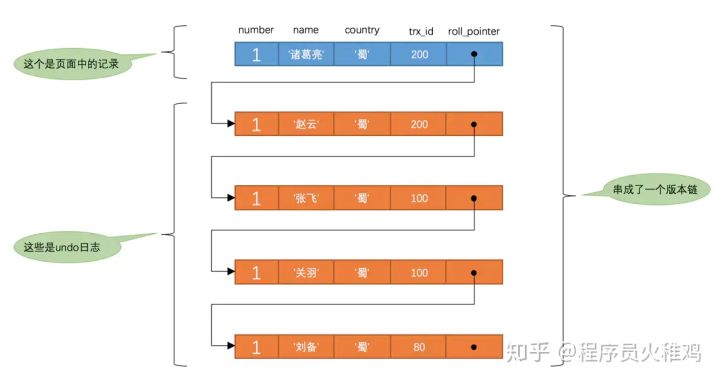
假设hero表中只有一行记录，当时插入的事务id为80。此时，该条记录的示例图如下：



假设之后两个事务id分别为100、200的事务对这条记录进行UPDATE操作，操作流程如下：



由于每次变动都会先把undo日志记录下来，并用roll\_pointer指向undo日志地址。因此可以认为，对该条记录的修改日志串联起来就形成了一个版本链，版本链的头节点就是当前记录最新的值。如下



### readview视图

如果数据库隔离级别是未提交读（READ UNCOMMITTED），那么读取版本链中最新版本的记录即可。如果是是串行化（SERIALIZABLE），事务之间是加锁执行的，不存在读不一致的问题。但是如果是已提交读（READ COMMITTED）或者可重复读（REPEATABLE READ），就需要遍历版本链中的每一条记录，判断该条记录是否对当前事务可见，直到找到为止(遍历完还没找到就说明记录不存在)。InnoDB通过ReadView实现了这个功能。ReadView中主要包含以下4个内容：

m\_ids：表示在生成ReadView时当前系统中活跃的读写事务的事务id列表。

min\_trx\_id：表示在生成ReadView时当前系统中活跃的读写事务中最小的事务id，也就是m\_ids中的最小值。

max\_trx\_id：表示生成ReadView时系统中应该分配给下一个事务的id值。

creator\_trx\_id：表示生成该ReadView事务的事务id。

有了ReadView之后，我们可以基于以下步骤判断某个版本的记录是否对当前事务可见。

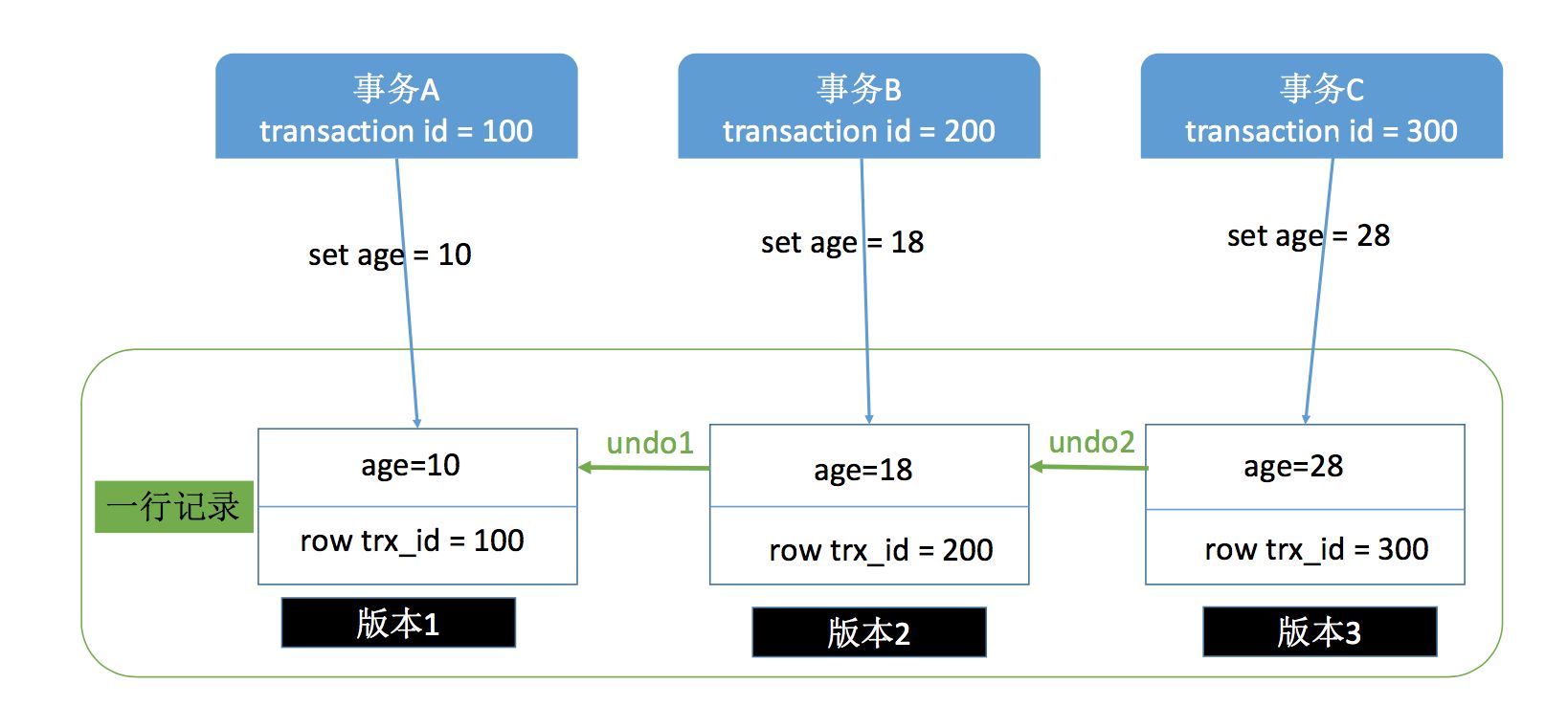
1. 如果被访问版本的trx\_id属性值与ReadView中的creator\_trx\_id值相同，意味着当前事务在访问它自己修改过的记录，所以该版本可以被当前事务访问。
2. 如果被访问版本的trx\_id属性值小于ReadView中的min\_trx\_id值，表明生成该版本的事务在当前事务生成ReadView前已经提交，所以该版本可以被当前事务访问。
3. 如果被访问版本的trx\_id属性值大于或等于ReadView中的max\_trx\_id值，表明生成该版本的事务在当前事务生成ReadView后才开启，所以该版本不可以被当前事务访问。
4. 如果被访问版本的trx\_id属性值在ReadView的min\_trx\_id和max\_trx\_id之间，那就需要判断一下trx\_id属性值是不是在m\_ids列表中，如果在，说明创建ReadView时生成该版本的事务还是活跃的，该版本不可以被访问；如果不在，说明创建ReadView时生成该版本的事务已经被提交，该版本可以被访问。

在MySQL中，READ COMMITTED和REPEATABLE READ隔离级别的的一个非常大的区别就是它们生成ReadView的时机不同。READ COMMITTED在每次读取数据前都会生成一个ReadView，这样就能保证每次都能读到其它事务已提交的数据。REPEATABLE READ 只在第一次读取数据时生成一个ReadView，这样就能保证后续读取的结果完全一致。

### 可重复读（mvcc）

为了解决不可重复读，或者为了实现可重复读，MySQL 采用了 MVCC (多版本并发控制) 的方式。

我们在数据库表中看到的一行记录可能实际上有多个版本，每个版本的记录除了有数据本身外，还要有一个表示版本的字段，记为 row trx\_id，而这个字段就是使其产生的事务的 id，事务 ID 记为 transaction id，它在事务开始的时候向事务系统申请，按时间先后顺序递增。



按照上面这张图理解，一行记录现在有 3 个版本，每一个版本都记录这使其产生的事务 ID，比如事务A的transaction id 是100，那么版本1的row trx\_id 就是 100，同理版本2和版本3。

在上面介绍读提交和可重复读的时候都提到了一个词，叫做快照，学名叫做一致性视图，这也是可重复读和不可重复读的关键，可重复读是在事务开始的时候生成一个当前事务全局性的快照，而读提交则是每次执行语句的时候都重新生成一次快照。

对于一个快照来说，它能够读到那些版本数据，要遵循以下规则：

* 当前事务内的更新，可以读到；
* 版本未提交，不能读到；
* 版本已提交，但是却在快照创建后提交的，不能读到；
* 版本已提交，且是在快照创建前提交的，可以读到；

利用上面的规则，再返回去套用到读提交和可重复读的那两张图上就很清晰了。还是要强调，两者主要的区别就是在快照的创建上，可重复读仅在事务开始是创建一次，而读提交每次执行语句的时候都要重新创建一次。

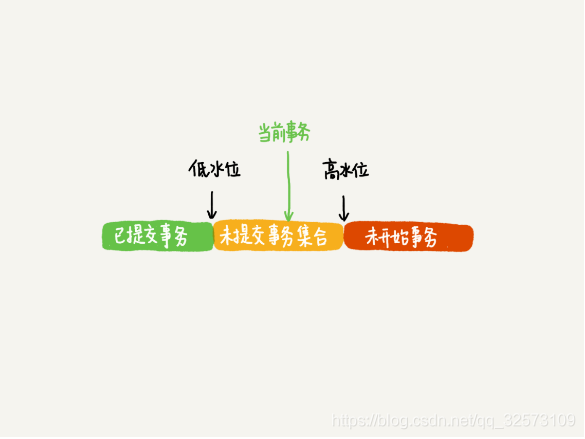
**总结**：

1. 读提交，每一句语句前创建快照，所以能够避免其他事务回滚造成的脏读。但是每次的快照内容都可能不同，所以可能造成不可重复读。
2. 可重复读，事务开始时生成快照，同时避免回滚和不可重复读，每次读快照的数据都是相同的。

ps：能说读的正确性解决了，但是其他事务做了修改，如果当前事务拿着快照数据做修改，会覆盖掉其他事务的操作，暂时没看到解决方案（不对）

pps：可重复读会对相关行数据加入行锁，所以修改的数据会锁住，其他事务无法修改。但是其他事务可以新增数据，还是可能在造成幻读。

3.依据下图



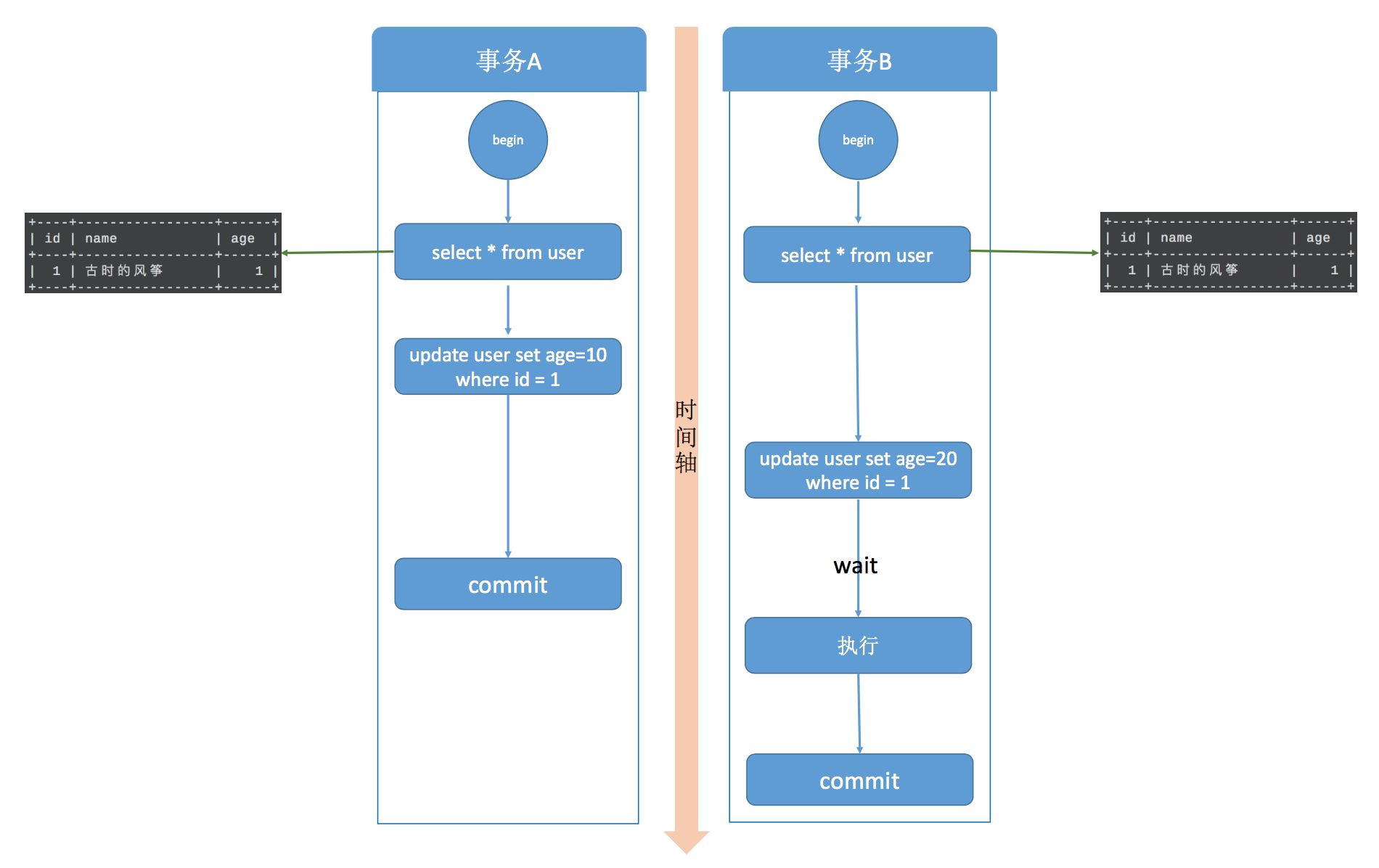
事务只能读事务编号小于自己的数据，即快照读，快照读无加锁操作。

但做update操作时，事务可以使用当前读，即读取最新版本的行数据。当前读操作会加锁。

### 并发写问题

存在这的情况，两个事务，对同一条数据做修改。最后结果应该是哪个事务的结果呢，肯定要是时间靠后的那个对不对。并且更新之前要先读数据，这里所说的读和上面说到的读不一样，更新之前的读叫做“当前读”，总是当前版本的数据，也就是多版本中最新一次提交的那版。

假设事务A执行 update 操作， update 的时候要对所修改的行加行锁，这个行锁会在提交之后才释放。而在事务A提交之前，事务B也想 update 这行数据，于是申请行锁，但是由于已经被事务A占有，事务B是申请不到的，此时，事务B就会一直处于等待状态，直到事务A提交，事务B才能继续执行，如果事务A的时间太长，那么事务B很有可能出现超时异常。如下图所示。



加锁的过程要分有索引和无索引两种情况，比如下面这条语句

update user set age=11 where id = 1

id 是这张表的主键，是有索引的情况，那么 MySQL 直接就在索引数中找到了这行数据，然后干净利落的加上行锁就可以了。

而下面这条语句

update user set age=11 where age=10

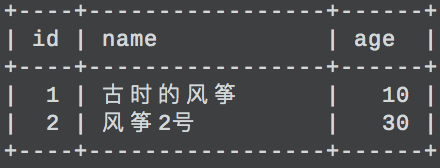
表中并没有为 age 字段设置索引，所以， MySQL 无法直接定位到这行数据。那怎么办呢，当然也不是加表锁了。MySQL 会为这张表中所有行加行锁，没错，是所有行。但是呢，在加上行锁后，MySQL 会进行一遍过滤，发现不满足的行就释放锁，最终只留下符合条件的行。虽然最终只为符合条件的行加了锁，但是这一锁一释放的过程对性能也是影响极大的。所以，如果是大表的话，建议合理设计索引，如果真的出现这种情况，那很难保证并发度。

### 解决幻读

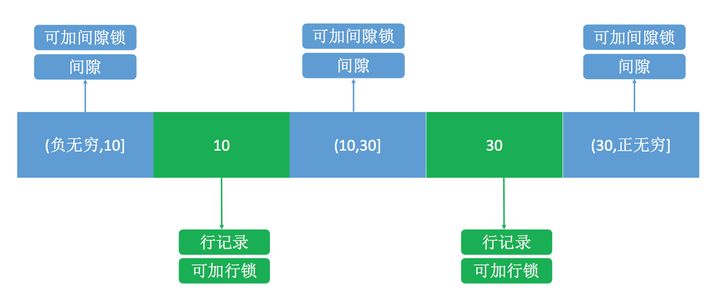
上面介绍可重复读的时候，那张图里标示着出现幻读的地方实际上在 MySQL 中并不会出现，MySQL 已经在可重复读隔离级别下解决了幻读的问题。

前面刚说了并发写问题的解决方式就是行锁，而解决幻读用的也是锁，叫做间隙锁，MySQL 把行锁和间隙锁合并在一起，解决了并发写和幻读的问题，这个锁叫做 Next-Key锁。

假设现在表中有两条记录，并且 age 字段已经添加了索引，两条记录 age 的值分别为 10 和 30。

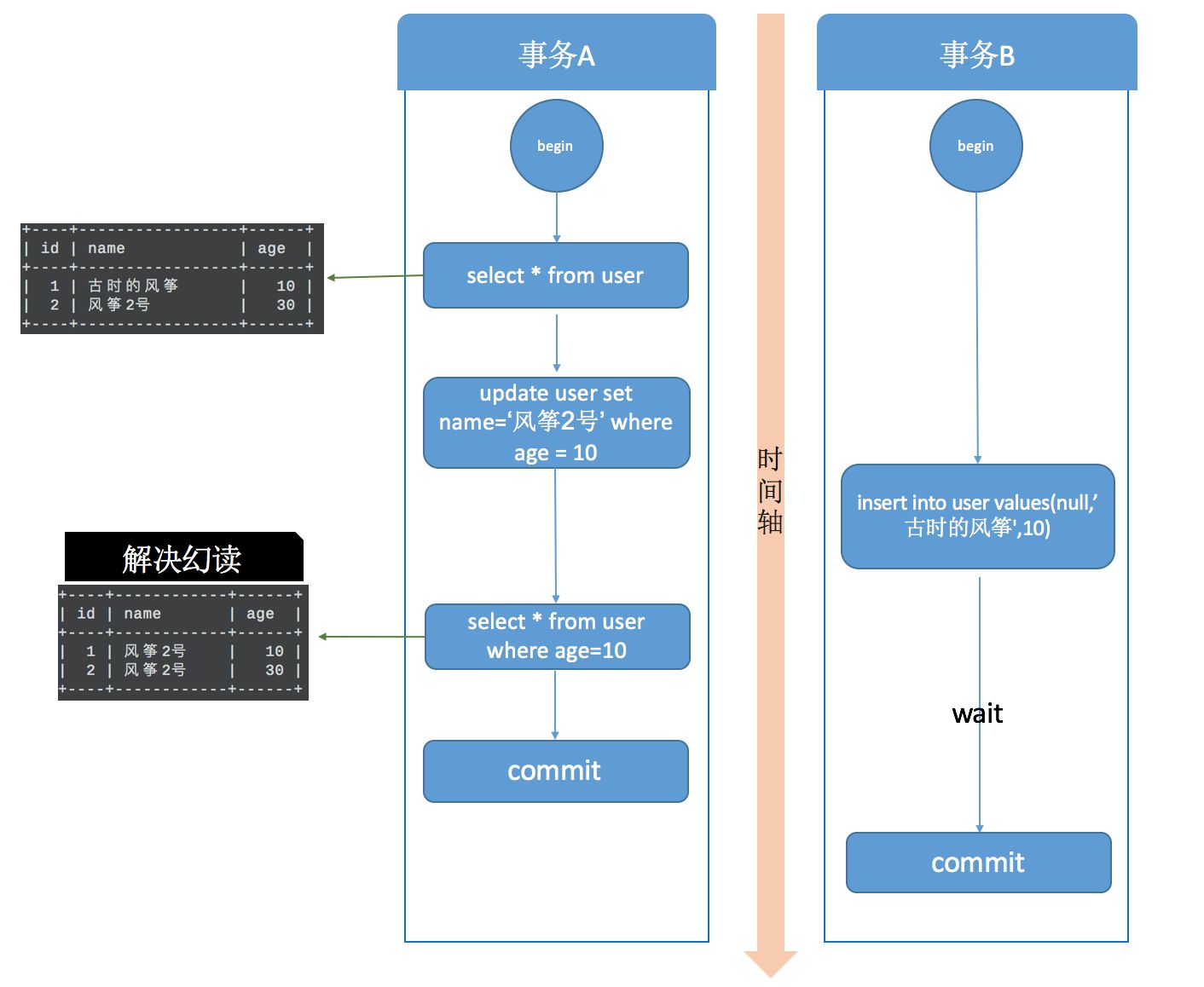


此时，在数据库中会为索引维护一套B+树，用来快速定位行记录。B+索引树是有序的，所以会把这张表的索引分割成几个区间。



如图所示，分成了3 个区间，(负无穷,10]、(10,30]、(30,正无穷]，在这3个区间是可以加间隙锁的。

之后，我用下面的两个事务演示一下加锁过程。



在事务A提交之前，事务B的插入操作只能等待，这就是间隙锁起得作用。当事务A执行update user set name='风筝2号’ where age = 10; 的时候，由于条件 where age = 10 ，数据库不仅在 age =10 的行上添加了行锁，而且在这条记录的两边，也就是(负无穷,10]、(10,30]这两个区间加了间隙锁，从而导致事务B插入操作无法完成，只能等待事务A提交。不仅插入 age = 10 的记录需要等待事务A提交，age<10、10<age<30 的记录页无法完成，而大于等于30的记录则不受影响，这足以解决幻读问题了。

这是有索引的情况，如果 age 不是索引列，那么数据库会为整个表加上间隙锁。所以，如果是没有索引的话，不管 age 是否大于等于30，都要等待事务A提交才可以成功插入。

### 当前读

在MVCC并发控制中，读操作可以分成两类：快照读 (snapshot read)与当前读 (current read)。快照读，读取的是记录的可见版本 (有可能是历史版本)，不用加锁。当前读，读取的是记录的最新版本，并且，当前读返回的记录，都会加上锁，保证其他事务不会再并发修改这条记录。

在一个支持MVCC并发控制的系统中，哪些读操作是快照读？哪些操作又是当前读呢？以MySQL InnoDB为例：

快照读：简单的select操作，属于快照读，不加锁。(当然，也有例外，下面会分析)

select \* from table where ?;

当前读：特殊的读操作，插入/更新/删除操作，属于当前读，需要加锁。（表的改删，数据的增删改）

select \* from table where ? lock in share mode;

select \* from table where ? for update;

insert into table values (…);

update table set ? where ?;

delete from table where ?;

所有以上的语句，都属于当前读，读取记录的最新版本。并且，读取之后，还需要保证其他并发事务不能修改当前记录，对读取记录加锁。其中，除了第一条语句，对读取记录加S锁 (共享锁)外，其他的操作，都加的是X锁 (排它锁)。