什么是MySQL

关系型数据库，端口号3306。

MyISAM和InnoDB区别

MyISAM是MySQL的默认数据库引擎（5.5版之前）。虽然性能极佳，而且提供了大量的特性，包括全文索引、压缩、空间函数等，但MyISAM不支持事务和行级锁，而且最大的缺陷就是崩溃后无法安全恢复。不过，5.5版本之后，MySQL引入了InnoDB（事务性数据库引擎），MySQL 5.5版本后默认的存储引擎为InnoDB。

大多数时候我们使用的都是 InnoDB 存储引擎，但是在某些情况下使用 MyISAM 也是合适的比如读密集的情况下。（如果你不介意 MyISAM 崩溃恢复问题的话）。
两者的对比：

1. **是否支持行级锁** : MyISAM 只有表级锁(table-level locking)，而InnoDB 支持行级锁(row-level locking)和表级锁,默认为行级锁。
2. **是否支持事务和崩溃后的安全恢复： MyISAM** 强调的是性能，每次查询具有原子性,其执行速度比InnoDB类型更快，但是不提供事务支持。但是**InnoDB** 提供事务支持，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。
3. **是否支持外键：** MyISAM不支持，而InnoDB支持。
4. **是否支持MVCC** ： 仅 InnoDB 支持。应对高并发事务, MVCC比单纯的加锁更高效;MVCC只在 `READ COMMITTED` 和 `REPEATABLE READ` 两个隔离级别下工作;MVCC可以使用 乐观(optimistic)锁 和 悲观(pessimistic)锁来实现;各数据库中MVCC实现并不统一。推荐阅读：MySQL-InnoDB-MVCC多版本并发控制
5. ……
   《MySQL高性能》上面有一句话这样写到:

   > 不要轻易相信“MyISAM比InnoDB快”之类的经验之谈，这个结论往往不是绝对的。在很多我们已知场景中，InnoDB的速度都可以让MyISAM望尘莫及，尤其是用到了聚簇索引，或者需要访问的数据都可以放入内存的应用。

关于count(*)
MyISAM会直接存储总行数，InnoDB则不会，需要按行扫描。是指在没有where条件的筛选的时候。
InnoDB的行锁是实现在索引上的，而不是锁在物理行记录上。潜台词是，如果访问没有命中索引，也无法使用行锁，将要退化为表锁。

InnoDB的每一个表都会有聚集索引：
(1)如果表定义了PK，则**PK**就是聚集索引；
(2)如果表没有定义PK，则**第一个非空unique列**是聚集索引；
(3)否则，InnoDB会**创建一个隐藏的row-id**作为聚集索引；

区别：

1. InnoDB 支持事务，MyISAM 不支持事务。这是 MySQL 将默认存储引擎从 MyISAM 变成 InnoDB 的重要原因之一；
2. InnoDB 支持外键，而 MyISAM 不支持。对一个包含外键的 InnoDB 表转为 MYISAM 会失败；
3. InnoDB 是聚集索引，MyISAM 是非聚集索引。聚簇索引的文件存放在主键索引的叶子节点上，因此 InnoDB 必须要有主键，通过主键索引效率很高。但是辅助索引需要两次查询，先查询到主键，然后再通过主键查询到数据。因此，主键不应该过大，因为主键太大，其他索引也都会很大。而 MyISAM 是非聚集索引，数据文件是分离的，索引保存的是数据文件的指针。主键索引和辅助索引是独立的。
4. InnoDB 不保存表的具体行数，执行 select count(*) from table 时需要全表扫描。而MyISAM 用一个变量保存了整个表的行数，执行上述语句时只需要读出该变量即可，速度很快；
5. InnoDB 最小的锁粒度是行锁，MyISAM 最小的锁粒度是表锁。一个更新语句会锁住整张表，导致其他查询和更新都会被阻塞，因此并发访问受限。这也是 MySQL 将默认存储引擎从 MyISAM 变成 InnoDB 的重要原因之一；

如何选择：

1. 是否要支持事务，如果要请选择 InnoDB，如果不需要可以考虑 MyISAM；
2. 如果表中绝大多数都只是读查询，可以考虑 MyISAM，如果既有读写也挺频繁，请使用InnoDB。
3. 系统奔溃后，MyISAM恢复起来更困难，能否接受，不能接受就选 InnoDB；
4. MySQL5.5版本开始Innodb已经成为Mysql的默认引擎(之前是MyISAM)，说明其优势是有目共睹的。如果你不知道用什么存储引擎，那就用InnoDB，至少不会差。

Innodb 创建表后生成的文件有：

- frm:创建表的语句
- idb:表里面的数据+索引文件

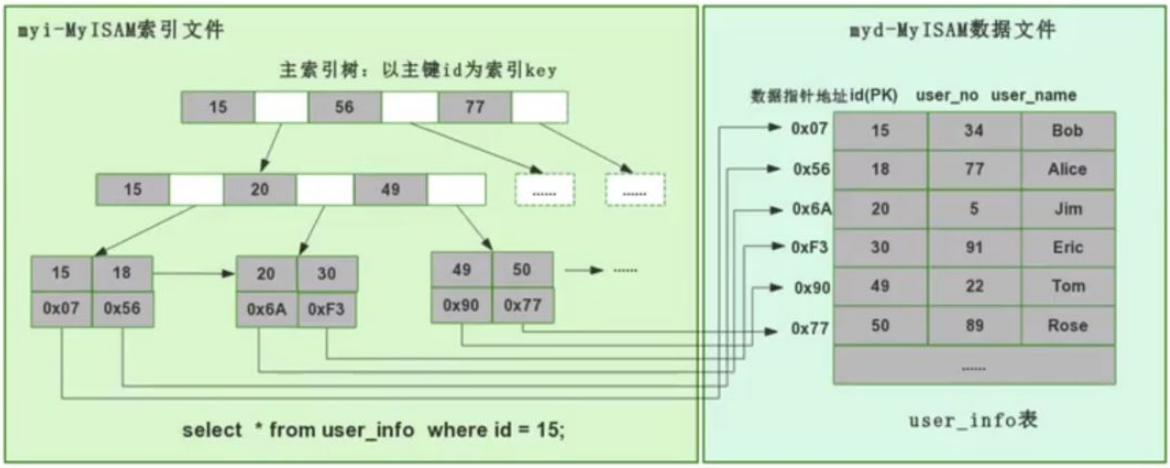
Myisam 创建表后生成的文件有

- frm:创建表的语句
- MYD:表里面的数据文件（myisam data）
- MYI:表里面的索引文件（myisam index）

从生成的文件看来，这两个引擎底层数据和索引的组织方式并不一样，MyISAM 引擎把数据和索引分开了，一人一个文件，这叫做非聚集索引方式；Innodb 引擎把数据和索引放在同一个文件里了，这叫做聚集索引方式。下面将从底层实现角度分析这两个引擎是怎么依靠 B+树这个数据结构来组织引擎实现的。

1. MyISAM 引擎的底层实现（非聚集索引方式）

MyISAM 用的是非聚集索引方式，即数据和索引落在不同的两个文件上。MyISAM 在建表时以主键作为 KEY 来建立主索引 B+树，树的叶子节点存的是对应数据的物理地址。我们拿到这个物理地址后，就可以到 MyISAM 数据文件中直接定位到具体的数据记录了。

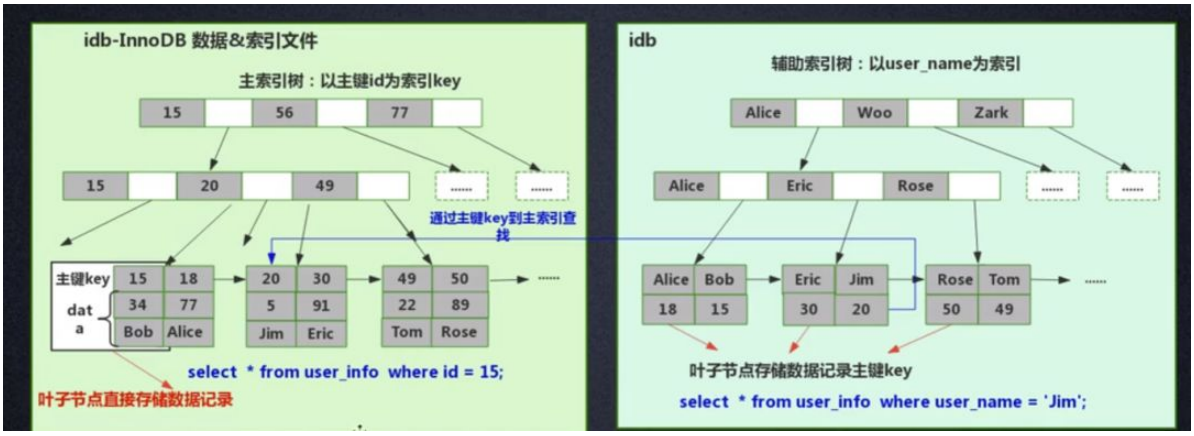


当我们为某个字段添加索引时，我们同样会生成对应字段的索引树，该字段的索引树的叶子节点同样是记录了对应数据的物理地址，然后也是拿着这个物理地址去数据文件里定位到具体的数据记录。

2. Innodb 引擎的底层实现（聚集索引方式）

InnoDB 是聚集索引方式，因此数据和索引都存储在同一个文件里。首先 InnoDB 会根据主键 ID 作为 KEY 建立索引 B+树，如左下图所示，而 B+树的叶子节点存储的是主键 ID 对应的数据，比如在执行 select * from user_info where id=15 这个语句时，InnoDB 就会查询这颗主键 ID 索引 B+树，找到对应的 user_name='Bob'。

这是建表的时候 InnoDB 就会自动建立好主键 ID 索引树，这也是为什么 Mysql 在建表时要求必须指定主键的原因。当我们为表里某个字段加索引时 InnoDB 会怎么建立索引树呢？比如我们要给 user_name 这个字段加索引，那么 InnoDB 就会建立 user_name 索引 B+树，节点里存的是 user_name 这个 KEY，叶子节点存储的数据的是主键 KEY。注意，叶子存储的是主键 KEY！拿到主键 KEY 后，InnoDB 才会去主键索引树里根据刚在 user_name 索引树找到的主键 KEY 查找到对应的数据。



问题来了，为什么 InnoDB 只在主键索引树的叶子节点存储了具体数据，但是其他索引树却不存具体数据呢，而要多此一举先找到主键，再在主键索引树找到对应的数据呢?

其实很简单，因为 InnoDB 需要节省存储空间。一个表里可能有很多个索引，InnoDB 都会给每个加了索引的字段生成索引树，如果每个字段的索引树都存储了具体数据，那么这个表的索引数据文件就变得非常巨大（数据极度冗余了）。从节约磁盘空间的角度来说，真的没有必要每个字段索引树都存具体数据，通过这种看似“多此一举”的步骤，在牺牲较少查询的性能下节省了巨大的磁盘空间，这是非常有值得的。

在进行 InnoDB 和 MyISAM 特点对比时谈到，MyISAM 查询性能更好，从上面索引文件数据文件的设计来看也可以看出原因：MyISAM 直接找到物理地址后就可以直接定位到数据记录，但是 InnoDB 查询到叶子节点后，还需要再查询一次主键索引树，才可以定位到具体数据。等于 MyISAM 一步就查到了数据，但是 InnoDB 要两步，那当然 MyISAM 查询性能更高。

字符集及校对规则

?

索引

MySQL索引使用的数据结构主要有**BTree索引** 和 **哈希索引** 。对于哈希索引来说，底层的数据结构就是哈希表，因此在绝大多数需求为单条记录查询的时候，可以选择哈希索引，查询性能最快；其余大部分场景，建议选择BTree索引。

MySQL的BTree索引使用的是B树中的B+Tree，但对于主要的两种存储引擎的实现方式是不同的。

- **MyISAM:** B+Tree叶节点的data域存放的是数据记录的地址。在索引检索的时候，首先按照B+Tree搜索算法搜索索引，如果指定的Key存在，则取出其 data 域的值，然后以 data 域的值为地址读取相应的数据记录。这被称为“非聚簇索引”。
- **InnoDB:** 其数据文件本身就是索引文件。相比MyISAM，索引文件和数据文件是分离的，其表数据文件本身就是按B+Tree组织的一个索引结构，树的叶节点data域保存了完整的数据记录。这个索引的key是数据表的主键，因此InnoDB表数据文件本身就是主索引。这被称为“聚簇索引（或聚集索引）”。而其余的索引都作为辅助索引，辅助索引的data域存储相应记录主键的值而不是地址，这也是和MyISAM不同的地方。**在根据主索引搜索时，直接找到key所在的节点即可取出数据；在根据辅助索引查找时，则需要先取出主键的值，再走一遍主索引。 因此，在设计表的时候，不建议使用过长的字段作为主键，也不建议使用非单调的字段作为主键，这样会造成主索引频繁分裂。**
  PS：整理自《Java工程师修炼之道》

更多关于索引的内容可以查看文档首页MySQL目录下关于索引的详细总结。

InnoDB引擎有“自适应哈希索引”（Adaptive Hash Index），当它注意到某些索引值被引用的非常频繁时，会在内存中基于B-Tree索引之上再创建一个哈希索引，让B-Tree索引也拥有哈希索引的一些优点。

查询缓存

执行查询语句的时候，会先查询缓存。不过，MySQL 8.0 版本后移除，因为这个功能不太实用

三大范式

第一范式

该范式是为了排除 重复组 的出现，因此要求数据库的每个列的值域都由原子值组成；每个字段的值都只能是单一值。1971 年埃德加·科德提出了第一范式。即表中所有字段都是不可再分的。解决方案：想要消除重复组的话，只要把每笔记录都转化为单一记录即可。

关系的所有属性都不可再分，即数据项不可分，强调数据表的原子性。

第二范式

属性完全依赖于主键。表中必须存在业务主键，并且非主键依赖于全部业务主键。解决方案：拆分将依赖的字段单独成表。

第三范式

表中的非主键列之间不能相互依赖，将不与 PK 形成依赖关系的字段直接提出单独成表即可。

第三范式(3NF)要求-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。例如，存在一个部门信息表,其中每个部有部门门编号(dept id)、部门]名称、部门简介等信息。那么在的员工信息表中列出部i]编号后就不能再将部门]名称、部门简介等与部门有关的信息再加入员工信息表中。如果不存在部门]信息表,则根据第三范式(3NF)也应该构建它，否则就会有大量的数据冗余。**简而言之，第三范式就是属性不依赖于其它非主属性。也就是说，如果存在非主属性对于码的传递函数依赖，则不符合3NF的要求。**

事务

事务是逻辑上的一组操作，要么都执行，要么都不执行。

事务最经典也经常被拿出来说例子就是转账了。假如小明要给小红转账1000元，这个转账会涉及到两个关键操作就是：将小明的余额减少1000元，将小红的余额增加1000元。万一在这两个操作之间突然出现错误比如银行系统崩溃，导致小明余额减少而小红的余额没有增加，这样就不对了。事务就是保证这两个关键操作要么都成功，要么都要失败。

事务的四大特性（ACID）

1. **原子性（Atomicity）：** 事务是最小的执行单位，不允许分割。事务的原子性确保动作要么全部完成，要么完全不起作用；InnoDB通过`redo/undo`来实现。
2. **一致性（Consistency）：** 执行事务后，数据库从一个正确的状态变化到另一个正确的状态；| 一致性，应该是个整体概念，保证所有的mysql对象（数据，索引，约束，日志，用户）在事务执行前后都具有完整的特性，应该是mysql所有的功能都为此服务吧！
3. **隔离性（Isolation）：** 并发访问数据库时，一个用户的事务不被其他事务所干扰，各并发事务之间数据库是独立的；隔离性侧重指事务之间相互隔离，不受影响，这个与事务设置的隔离级别有密切的关系。
4. **持久性（Durability）：** 一个事务被提交之后。它对数据库中数据的改变是持久的，即使数据库发生故障也不应该对其有任何影响。

并发事务带来哪些问题?

在典型的应用程序中，多个事务并发运行，经常会操作相同的数据来完成各自的任务（多个用户对同一数据进行操作）。并发虽然是必须的，但可能会导致以下的问题。

- **脏读（Dirty read）：**当一个事务正在访问数据并且对数据进行了修改，而这种修改还没有提交到数据库中，这时另外一个事务也访问了这个数据，然后使用了这个数据。因为这个数据是还没有提交的数据，那么另外一个事务读到的这个数据是“脏数据”，依据“脏数据”所做的操作可能是不正确的。
- **丢失修改（Lost to modify）：**指在一个事务读取一个数据时，另外一个事务也访问了该数据，那么在第一个事务中修改了这个数据后，第二个事务也修改了这个数据。这样第一个事务内的修改结果就被丢失，因此称为丢失修改。 例如：事务1读取某表中的数据A=20，事务2也读取A=20，事务1修改A=A-1，事务2也修改A=A-1，最终结果A=19，事务1的修改被丢失。
- **不可重复读（Unrepeatableread）：**指在一个事务内多次读同一数据。在这个事务还没有结束时，另一个事务也访问该数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修改导致第一个事务两次读取的数据可能不太一样。这就发生了在一个事务内两次读到的数据是不一样的情况，因此称为不可重复读。
- **幻读（Phantom read）：**幻读与不可重复读类似。它发生在一个事务（T1）读取了几行数据，接着另一个并发事务（T2）插入了一些数据时。在随后的查询中，第一个事务（T1）就会发现多了一些原本不存在的记录，就好像发生了幻觉一样，所以称为幻读。

不可重复读和幻读区别：

不可重复读的重点是修改比如多次读取一条记录发现其中某些列的值被修改，幻读的重点在于新增或者删除比如多次读取一条记录发现记录增多或减少了。

事务隔离级别有哪些？MySQL的默认隔离级别是?

SQL 标准定义了四个隔离级别：

- **READ-UNCOMMITTED(读取未提交)：** 最低的隔离级别，允许读取尚未提交的数据变更，**可能会导致脏读、幻读或不可重复读。**

- **READ-COMMITTED(读取已提交)：** 允许读取并发事务已经提交的数据，**可以阻止脏读，但是幻读或不可重复读仍有可能发生。**
- **REPEATABLE-READ(可重复读)：** 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，**可以阻止脏读和不可重复读，但幻读仍有可能发生。**
- **SERIALIZABLE(可串行化)：** 最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，**该级别可以防止脏读、不可重复读以及幻读。**

隔离级别	脏读	不可重复读	幻影读
READ-UNCOMMITTED	√	√	√
READ-COMMITTED	×	√	√
REPEATABLE-READ	×	×	√
SERIALIZABLE	×	×	×

MySQL InnoDB 存储引擎的默认支持的隔离级别是 **REPEATABLE-READ（可重读）**。我们可以通过 `SELECT @@tx_isolation;` 命令来查看，MySQL 8.0 该命令改为 `SELECT @@transaction_isolation;`

这里需要注意的是：与 SQL 标准不同的地方在于 InnoDB 存储引擎在 **REPEATABLE-READ（可重读）** 事务隔离级别下使用的是Next-Key Lock 锁算法，因此可以避免幻读的产生，这与其他数据库系统(如 SQL Server) 是不同的。所以说InnoDB 存储引擎的默认支持的隔离级别是 **REPEATABLE-READ（可重读）** 已经可以完全保证事务的隔离性要求，即达到了 SQL标准的 **SERIALIZABLE(可串行化)** 隔离级别。因为隔离级别越低，事务请求的锁越少，所以大部分数据库系统的隔离级别都是 **READ-COMMITTED(读取提交内容)**，但是你要知道的是InnoDB 存储引擎默认使用 **REPEAaTABLE-READ（可重读）** 并不会有任何性能损失。

InnoDB 存储引擎在 **分布式事务** 的情况下一般会用到 **SERIALIZABLE(可串行化)** 隔离级别。

锁机制与InnoDB锁算法

MyISAM和InnoDB存储引擎使用的锁：

- MyISAM采用表级锁(table-level locking)。
- InnoDB支持行级锁(row-level locking)和表级锁,默认为行级锁

表级锁和行级锁对比：

- **表级锁：** MySQL中锁定 **粒度最大** 的一种锁，对当前操作的整张表加锁，实现简单，资源消耗也比较少，加锁快，不会出现死锁。其锁定粒度最大，触发锁冲突的概率最高，并发度最低，MyISAM 和 InnoDB引擎都支持表级锁。
- **行级锁：** MySQL中锁定 **粒度最小** 的一种锁，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，并发度高，但加锁的开销也最大，加锁慢，会出现死锁。

InnoDB存储引擎的锁的算法有三种：

- Record lock：单个行记录上的锁
- Gap lock：间隙锁，锁定一个范围，不包括记录本身
- Next-key lock：record+gap 锁定一个范围，包含记录本身

相关知识点：

1. innodb对于行的查询使用next-key lock
2. Next-locking keying为了解决Phantom Problem幻读问题
3. 当查询的索引含有唯一属性时，将next-key lock降级为record key
4. Gap锁设计的目的是为了阻止多个事务将记录插入到同一范围内，而这会导致幻读问题的产生
5. 有两种方式显式关闭gap锁：（除了外键约束和唯一性检查外，其余情况仅使用record lock） A. 将事务隔离级别设置为RC B. 将参数innodb_locks_unsafe_for_binlog设置为1

大表优化

当MySQL单表记录数过大时，数据库的CRUD性能会明显下降，一些常见的优化措施如下：

1. **限定数据的范围**

务必禁止不带任何限制数据范围条件的查询语句。比如：我们当用户在查询订单历史的时候，我们可以控制在一个月的范围内；

2. **读/写分离**

经典的数据库拆分方案，主库负责写，从库负责读；

3. **垂直分区(待完善)**

根据数据库里面数据表的相关性进行拆分。 例如，用户表中既有用户的登录信息又有用户的基本信息，可以将用户表拆分成两个单独的表，甚至放到单独的库做分库。
简单来说垂直拆分是指数据表列的拆分，把一张列比较多的表拆分为多张表。

- **垂直拆分的优点：** 可以使得列数据变小，在查询时减少读取的Block数，减少I/O次数。此外，垂直分区可以简化表的结构，易于维护。
- **垂直拆分的缺点：** 主键会出现冗余，需要管理冗余列，并会引起Join操作，可以通过在应用层进行Join来解决。此外，垂直分区会让事务变得更加复杂；

4. **水平分区(待完善)**

保持数据表结构不变，通过某种策略存储数据分片。这样每一片数据分散到不同的表或者库中，达到了分布式的目的。 水平拆分可以支撑非常大的数据量。
水平拆分是指数据表行的拆分，表的行数超过200万行时，就会变慢，这时可以把一张的表的数据拆成多张表来存放。举个例子：我们可以将用户信息表拆分成多个用户信息表，这样就可以避免单一表数据量过大对性能造成影响。

水平拆分可以支持非常大的数据量。需要注意的一点是：分表仅仅是解决了单一表数据过大的问题，但由于表的数据还是在同一台机器上，其实对于提升MySQL并发能力没有什么意义，所以 **水平拆分最好分库** 。

水平拆分能够 **支持非常大的数据量存储，应用端改造也少**，但 **分片事务难以解决** ，跨节点Join性能较差，逻辑复杂。《Java工程师修炼之道》的作者推荐 **尽量不要对数据进行分片，因为拆分会带来逻辑、部署、运维的各种复杂度** ，一般的数据表在优化得当的情况下支撑千万以下的数据量是没有太大问题的。如果实在要分片，尽量选择客户端分片架构，这样可以减少一次和中间件的网络I/O。

下面补充一下数据库分片的两种常见方案：

- **客户端代理： 分片逻辑在应用端，封装在jar包中，通过修改或者封装JDBC层来实现。** 当当网的 **Sharding-JDBC** （推荐） 、阿里的TDDL是两种比较常用的实现。
- **中间件代理： 在应用和数据中间加了一个代理层。分片逻辑统一维护在中间件服务中。** 我们现在谈的 **Mycat** 、360的Atlas、网易的DDB等等都是这种架构的实现。

详细内容可以参考： MySQL大表优化方案: https://segmentfault.com/a/1190000006158186

什么是数据库连接池？为什么需要连接池？

池化设计应该不是一个新名词。我们常见的如java线程池、jdbc连接池、redis连接池等就是这类设计的代表实现。这种设计会初始预设资源，解决的问题就是抵消每次获取资源的消耗，如创建线程的开销，获取远程连接的开销等。就好比你去食堂打饭，打饭的大妈会先把饭盛好几份放那里，你来了就直接拿着饭盒加菜即可，不用再临时又盛饭又打菜，效率就高了。除了初始化资源，池化设计还包括如下这些特征：池子的初始值、池子的活跃值、池子的最大值等，这些特征可以直接映射到java线程池和数据库连接池的成员属性中。这篇文章对池化设计思想介绍的还不错，直接复制过来，避免重复造轮子了。

数据库连接本质就是一个 socket 的连接。数据库服务端还要维护一些缓存和用户权限信息之类的 所以占用了一些内存。我们可以把数据库连接池是看做是维护的数据库连接的缓存，以便将来需要对数据库的请求时可以重用这些连接。为每个用户打开和维护数据库连接，尤其是对动态数据库驱动的网站应用程序的请求，既昂贵又浪费资源。**在连接池中，创建连接后，将其放置在池中，并再次使用它，因此不必建立新的连接。如果使用了所有连接，则会建立一个新连接并将其添加到池中。** 连接池还减少了用户必须等待建立与数据库的连接的时间。

分库分表之后，id主键如何处理？

因为要是分成多个表之后，每个表都是从 1 开始累加，这样是不对的，我们需要一个全局唯一的 id 来支持。

生成全局 id 有下面这几种方式：

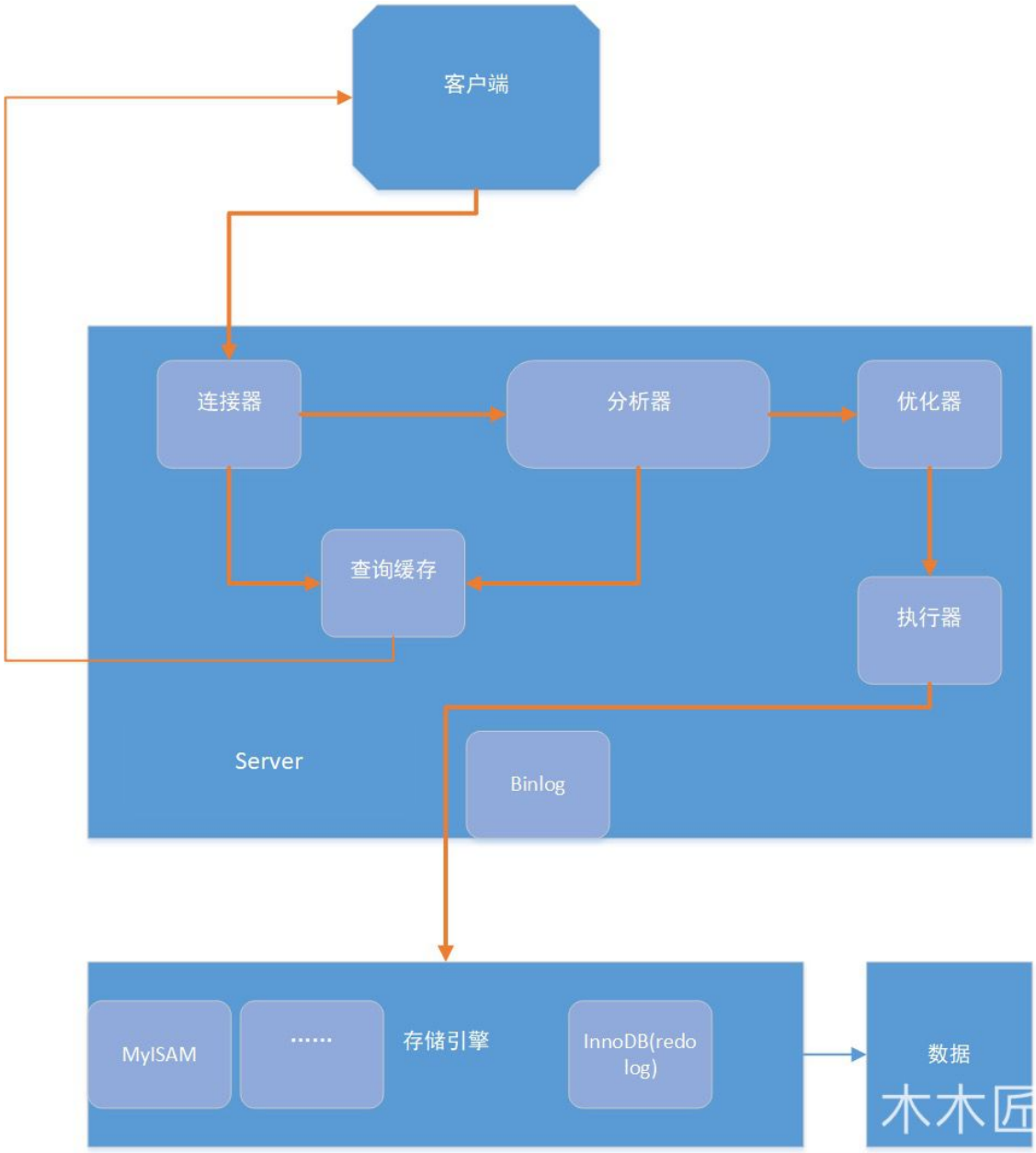
- **UUID**：不适合作为主键，因为太长了，并且无序不可读，查询效率低。比较适合用于生成唯一的名字的标示比如文件的名字。
- **数据库自增 id** : 两台数据库分别设置不同步长，生成不重复ID的策略来实现高可用。这种方式生成的 id 有序，但是需要独立部署数据库实例，成本高，还会有性能瓶颈。
- **利用 redis 生成 id :** 性能比较好，灵活方便，不依赖于数据库。但是，引入了新的组件造成系统更加复杂，可用性降低，编码更加复杂，增加了系统成本。
- **Twitter的snowflake算法** ： Github 地址：https://github.com/twitter-archive/snowflake。
- **美团的Leaf分布式ID生成系统** ： Leaf 是美团开源的分布式ID生成器，能保证全局唯一性、趋势递增、单调递增、信息安全，里面也提到了几种分布式方案的对比，但也需要依赖关系数据库、Zookeeper等中间件。感觉还不错。美团技术团队的一篇文章：https://tech.meituan.com/2017/04/21/mt-leaf.html 。
- .....

一条SQL语句的执行

基本架构

先简单介绍一下下图涉及的一些组件的基本作用帮助大家理解这幅图，在 1.2 节中会详细介绍到这些组件的作用。

- **连接器：** 身份认证和权限相关(登录 MySQL 的时候)。
- **查询缓存:** 执行查询语句的时候，会先查询缓存（MySQL 8.0 版本后移除，因为这个功能不太实用）。
- **分析器:** 没有命中缓存的话，SQL 语句就会经过分析器，分析器说白了就是要先看你的 SQL 语句要干嘛，再检查你的 SQL 语句语法是否正确。
- **优化器：** 按照 MySQL 认为最优的方案去执行。
- **执行器:** 执行语句，然后从存储引擎返回数据。



简单来说 MySQL 主要分为 Server 层和存储引擎层：

- **Server 层**：主要包括连接器、查询缓存、分析器、优化器、执行器等，所有跨存储引擎的功能都在这一层实现，比如存储过程、触发器、视图，函数等，还有一个通用的日志模块 binglog 日志模块。
- **存储引擎**： 主要负责数据的存储和读取，采用可以替换的插件式架构，支持 InnoDB、MyISAM、Memory 等多个存储引擎，其中 InnoDB 引擎有自有的日志模块 redolog 模块。**现在最常用的存储引擎是 InnoDB，它从 MySQL 5.5.5 版本开始就被当做默认存储引擎了**

1.1.1 连接层

最上层是一些客户端和连接服务，包含本地 sock 通信和大多数基于客户端/服务端工具实现的类似于 tcp/ip 的通信。主要完成一些类似于连接处理、授权认证、及相关的安全方案。在该层上引入了线程池的概念，为通过认证安全接入的客户端提供线程。同样在该层上可以实现基于 SSL 的安全链接。服务器也会为安全接入的每个客户端验证它所具有的操作权限。

1.1.2 服务层

Management Serveices & Utilities	系统管理和控制工具
SQL Interface:	SQL接口。接受用户的SQL命令，并且返回用户需要查询的结果。比如select from就是调用SQL Interface
Parser	解析器。 SQL命令传递到解析器的时候会被解析器验证和解析
Optimizer	查询优化器。 SQL语句在查询之前会使用查询优化器对查询进行优化，比如有where条件时，优化器来决定先投影还是先过滤。
Cache和Buffer	查询缓存。如果查询缓存有命中的查询结果，查询语句就可以直接去查询缓存中取数据。这个缓存机制是由一系列小缓存组成的。比如表缓存，记录缓存，key缓存，权限缓存等

1.1.3 引擎层

存储引擎层，存储引擎真正的负责了 MySQL 中数据的存储和提取，服务器通过 API 与存储引擎进行通信。不同的存储引擎具有的功能不同，这样我们可以根据自己的实际需要进行选取。

1.1.4 存储层

数据存储层，主要是将数据存储在运行于裸设备的文件系统之上，并完成与存储引擎的交互。

提高并发

提高并发的演进思路，就在如此：

- **普通锁**，本质是串行执行
- **读写锁**，可以实现读读并发
- **数据多版本**，可以实现读写并发

简单的锁住太过粗暴，连“读任务”也无法并行，任务执行过程本质上是串行的。于是出现了**共享锁**和**排他锁**。

- 共享锁（**S**hare Locks，记为S锁），读取数据时加S锁
- 排他锁（e**X**clusive Locks，记为X锁），修改数据时加X锁

共享锁与排他锁的玩法是：

- 共享锁之间不互斥，简记为：读读可以并行
- 排他锁与任何锁互斥，简记为：写读，写写不可以并行

可以看到，一旦写数据的任务没有完成，数据是不能被其他任务读取的，这对并发度有较大的影响。
画外音：对应到数据库，可以理解为，写事务没有提交，读相关数据的select也会被阻塞。

MVCC

MVCC，全称Multi-Version Concurrency Control，即多版本并发控制。MVCC是一种并发控制的方法，一般在数据库管理系统中，实现对数据库的并发访问，在编程语言中实现事务内存。

- **当前读。**像select lock in share mode(共享锁), select for update ; update, insert ,delete(排他锁)这些操作都是一种当前读，为什么叫当前读？就是它读取的是记录的最新版本，读取时还要保证其他并发事务不能修改当前记录，会对读取的记录进行加锁。
- **快照读。**像不加锁的select操作就是快照读，即不加锁的非阻塞读；快照读的前提是隔离级别不是串行级别，串行级别下的快照读会退化成当前读；之所以出现快照读的情况，是基于提高并发性能的考虑，快照读的实现是基于多版本并发控制，即MVCC,可以认为MVCC是行锁的一个变种，但它在很多情况下，避免了加锁操作，降低了开销；既然是基于多版本，即快照读可能读到的并不一定是数据的最新版本，而有可能是之前的历史版本。**除非显示加锁，普通的select语句都是快照读。**

说白了MVCC就是为了实现读-写冲突不加锁，而这个读指的就是快照读, 而非当前读，当前读实际上是一种加锁的操作，是悲观锁的实现

> MVCC的好处?

多版本并发控制（MVCC）是一种用来解决读-写冲突的无锁并发控制，也就是为事务分配单向增长的时间戳，为每个修改保存一个版本，版本与事务时间戳关联，读操作只读该事务开始前的数据库的快照。
所以MVCC可以为数据库解决以下问题

- 在并发读写数据库时，可以做到在读操作时不用阻塞写操作，写操作也不用阻塞读操作，提高了数据库并发读写的性能
- 同时还可以解决脏读，幻读，不可重复读等事务隔离问题，但不能解决更新丢失问题

实现原理

隐式字段，undo日志，Read View。

1. 隐式字段
每行记录除了我们自定义的字段外，还有数据库隐式定义的DB_TRX_ID,DB_ROLL_PTR,DB_ROW_ID等字段。

- DB_TRX_ID，最近修改(修改/插入)事务ID：记录创建这条记录/最后一次修改该记录的事务ID
- DB_ROLL_PTR，回滚指针，指向这条记录的上一个版本（存储于rollback segment里）
- DB_ROW_ID，隐含的自增ID（隐藏主键），如果数据表没有主键，InnoDB会自动以DB_ROW_ID产生一个聚簇索引
- 实际还有一个删除flag隐藏字段, 既记录被更新或删除并不代表真的删除，而是删除flag变了

person表的某条记录

name	age	DB_ROW_ID(隐式主键)	DB_TRX_ID(事务ID)	DB_ROLL_PTR(回滚指针)
Jerry	24	1	1	0x12446545

如上图，DB_ROW_ID是数据库默认为该行记录生成的唯一隐式主键，DB_TRX_ID是当前操作该记录的事务ID,而DB_ROLL_PTR是一个回滚指针，用于配合undo日志，指向上一个旧版本。

2. undo 日志

> 首先插一句redo 日志。 为什么要有redo日志?

数据库事务提交后，必须将更新后的数据刷到磁盘上，以保证ACID特性。磁盘**随机写**性能较低，如果每次都刷盘，会极大影响数据库的吞吐量。优化方式是，将修改行为先写到redo日志里（此时变成了**顺序写**），再定期将数据刷到磁盘上，这样能极大提高性能。
*画外音：这里的架构设计方法是，**随机写优化为顺序写**。*

假如某一时刻，数据库崩溃，还没来得及刷盘的数据，在数据库重启后，会重做redo日志里的内容，以保证已提交事务对数据产生的影响都刷到磁盘上。**一句话**，redo日志用于保障，**已提交事务**的ACID特性。

> 为什么要有undo日志?

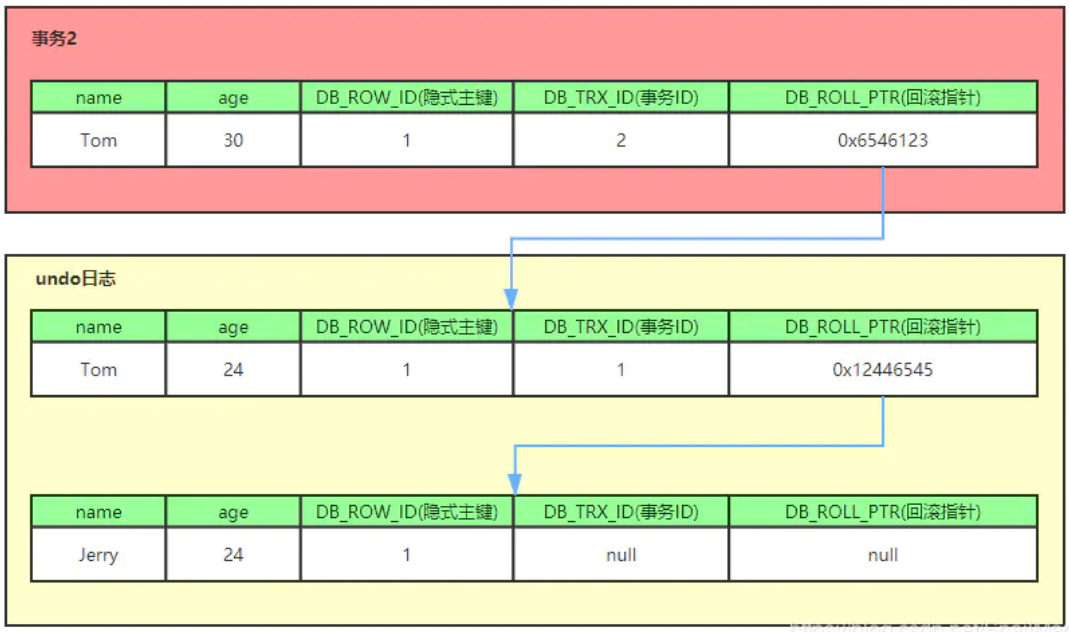
数据库事务未提交时，会将事务修改数据的镜像（即修改前的旧版本）存放到undo日志里，当事务回滚时，或者数据库奔溃时，可以利用undo日志，即旧版本数据，撤销未提交事务对数据库产生的影响。
画外音：更细节的，
对于insert操作，**undo日志记录新数据的PK(ROW_ID)，回滚时直接删除；***
对于delete/update操作*，*undo日志记录旧数据row，回滚时直接恢复；*
他们分别存放在不同的buffer里。

一句话，undo日志用于保障，**未提交事务**不会对数据库的ACID特性产生影响。

PS：还有，undolog才是原子性的关键。提供redolog，应该主要目的是提升磁盘的IO开销吧，如果直接写入磁盘，IO开销，会很大。如果先将操作记录到redolog中，可以顺序的记录，批量的记录，再一起同步到磁盘上，速度会比直接写磁盘快些。 mysql在生成redolog时，会使用 innodb log buffer，先缓冲到内存中，再同步到redolog上。速度会更快。

> 回滚段，rollback segment

存储undo日志的地方，是回滚段。undo日志和回滚段和InnoDB的MVCC密切相关。*假设事务提交，回滚段里的undo日志可以删除。*



从上面，我们就可以看出，不同事务或者相同事务的对同一记录的修改，会导致该记录的undo log成为一条记录版本线性表，既链表，undo log的链首就是最新的旧记录，链尾就是最早的旧记录。

3. Read View（读视图）

什么是Read View，说白了Read View就是事务进行快照读操作的时候生产的读视图(Read View)，在该事务执行的快照读的那一刻，会生成数据库系统当前的一个快照，记录并维护系统当前活跃事务的ID(当每个事务开启时，都会被分配一个ID, 这个ID是递增的，所以最新的事务，ID值越大)。
所以我们知道 Read View主要是用来做可见性判断的, 即当我们某个事务执行快照读的时候，对该记录创建一个Read View读视图，把它比作条件用来判断当前事务能够看到哪个版本的数据，既可能是当前最新的数据，也有可能是该行记录的undo log里面的某个版本的数据。
Read View遵循一个可见性算法，主要是将要被修改的数据的最新记录中的DB_TRX_ID（即当前事务ID）取出来，与系统当前其他活跃事务的ID去对比（由Read View维护），如果DB_TRX_ID跟Read View的属性做了某些比较，不符合可见性，那就通过DB_ROLL_PTR回滚指针去取出Undo Log中的DB_TRX_ID再比较，即遍历链表的DB_TRX_ID（从链首到链尾，即从最近的一次修改查起），直到找到满足特定条件的DB_TRX_ID, 那么这个DB_TRX_ID所在的旧记录就是当前事务能看见的最新老版本

RC,RR级别下的InnoDB快照读有什么不同?

正是Read View生成时机的不同，从而造成RC,RR级别下快照读的结果的不同

- 在RR级别下的某个事务的对某条记录的第一次快照读会创建一个快照及Read View, 将当前系统活跃的其他事务记录起来，此后在调用快照读的时候，还是使用的是同一个Read View，所以只要当前事务在其他事务提交更新之前使用过快照读，那么之后的快照读使用的都是同一个Read View，所以对之后的修改不可见；

- 即RR级别下，快照读生成Read View时，Read View会记录此时所有其他活动事务的快照，这些事务的修改对于当前事务都是不可见的。而早于Read View创建的事务所做的修改均是可见
- 而在RC级别下的，事务中，每次快照读都会新生成一个快照和Read View, 这就是我们在RC级别下的事务中可以看到别的事务提交的更新的原因

总之在RC隔离级别下，是每个快照读都会生成并获取最新的Read View；
而在RR隔离级别下，则是同一个事务中的第一个快照读才会创建Read View, 之后的快照读获取的都是同一个Read View。

RR下，事务在第一个Read操作时，会建立Read View
RC下，事务在每次Read操作时，都会建立Read View

MVCC能解决幻读吗?

https://blog.csdn.net/qq_35590091/article/details/107734005

不能笼统的说能不能解决，因为有的情况下可以解决，但是有的情况下解决不了。
mysql里面实际上有两种读，一种是"**快照读**"，比如我们使用select进行查询，就是快照读，在"快照读"的情况下是可以解决"幻读"的问题的。使用的就是MVCC。

锁

数据库使用锁是为了支持对共享资源进行并发访问，提供数据的完整性和一致性。

> lock 和 latch

latch一般称为闩锁（轻量级锁），因为其要求锁定的时间必须非常短。若持续的时间长，则应用的性能会非常差。在InnoDB存储引擎中，latch又分为mutex（互斥锁）和rwlock（读写锁）。其目的是用来保证并发线程操作临界资源的正确性，并且通常没有死锁检测的机制。

lock的对象是事务，用来锁定的是数据库中的对象，如表、页、行。并且一般lock的对象仅在事务commit或rollback后进行释放（不同事务隔离级别释放的时间可能不同）。此外，lock，正如在大多数数据库中一样，是有死锁机制的。下图显示lock与latch的区别：

	lock	latch
对象	事务	线程
保护	数据库内容	内存数据结构
持续时间	整个事务过程	临界资源
模式	行锁、表锁、意向锁	读写锁、互斥量
死锁	通过 waits-for graph、time out 等机制进行死锁检测与处理	无死锁检测与处理机制。仅通过应用程序加锁的顺序（lock leveling）保证无死锁的情况发生
存在于	Lock Manager 的哈希表中	每个数据结构的对象中

锁的类型

- 共享锁，允许事务读，
- 排他锁，允许删除或更行一行数据。只有共享锁之间可以兼容，其余S和X，X和X都不兼容。

意向锁，是将锁定的对象分为多个层次，意味着事务想在更细的粒度上进行加锁。
（数据库层级关系：数据库A - 表1&表2&表3&表4 - 页 - 记录）
如果想对页上的记录r进行上X锁，那么就要对数据库A、表、页上意向IX锁。InnoDB支持多粒度锁(multiple granularity locking)，它允许行级锁与表级锁共存。**意向锁**是指，未来的某个时刻，事务可能要加共享/排它锁了，先提前声明一个意向。

对已有数据行的**修改与删除**，必须加强互斥锁X锁，那对于**数据的插入**，是否还需要加这么强的锁，来实施互斥呢？插入意向锁，孕育而生。**插入意向锁**，是间隙锁(Gap Locks)的一种（所以，也是实施在索引上的），它是专门针对insert操作的。它的玩法是：多个事务，在同一个索引，同一个范围区间插入记录时，如果插入的位置不冲突，不会阻塞彼此。

自增锁

自增锁是一种特殊的**表级别锁**（table-level lock），专门针对事务插入AUTO_INCREMENT类型的列。最简单的情况，如果一个事务正在往表中插入记录，所有其他事务的插入必须等待，以便第一个事务插入的行，是连续的主键值。

记录锁，间隙锁，临键锁

记录锁，它封锁索引记录，例如：`select * from t where id=1 for update;`它会在id=1的索引记录上加锁，以阻止其他事务插入，更新，删除id=1的这一行。需要说明的是：`select * from t where id=1;`则是**快照读**(SnapShot Read)，它并不加锁.

间隙锁（Gap Lock），锁定一个范围，不包括记录本身。它封锁索引记录中的间隔，或者第一条索引记录之前的范围，又或者最后一条索引记录之后的范围。间隙锁的**主要目的**，就是为了防止其他事务在间隔中插入数据，以导致“~~不可重复读~~幻读？”。如果把事务的隔离级别降级为**读提交**(Read Committed, RC)，间隙锁则会自动失效。

临键锁（Next-Key Lock），Gap Lock + Record Lock，是记录锁与间隙锁的组合，它的封锁范围，既包含索引记录，又包含索引区间。更具体的，临键锁会封锁索引记录本身，以及索引记录之前的区间。如果一个会话占有了索引记录R的共享/排他锁，其他会话不能立刻在R之前的区间插入新的索引记录。如果一个索引中有10，11，13，20这四个值，那么可能被锁住的区间为：(-∞, 10] (10,11] (11,13] (13, 20] (20, +∞)。
临键锁的主要目的，也是为了避免**幻读**(Phantom Read)。如果把事务的隔离级别降级为RC，临键锁则也会失效。

并发中的一致性

小概率事件，使用CAS乐观锁，Compare ans Set，即要修改时需要再次比对当前的状态是否为初始状态，只有是初始状态时才能修改。比如有100块余额，事务1扣款80，余额为20，事务2扣款70，余额为30，事务1扣完剩20，事务2计算完逻辑值准备设定余额时，需要再检查一下状态，由旧的100变成了现在的20，不一致了，所以不能扣款。

但这会发生**ABA问题**，先A，后B，后来又有个事务改成了A，但此A非彼A，发生问题。ABA问题导致的原因，是CAS过程中只简单进行了“值”的校验，再有些情况下，“值”相同不会引入错误的业务逻辑（例如余额），有些情况下，“值”虽然相同，却已经不是原来的数据了（例如堆栈）。因此，CAS不能只比对“值”，还必须确保是原来的数据，才能修改成功。

常见的实践是，将“值”比对，升级为“版本号”的比对，一个数据一个版本，**版本变化，即使值相同，也不应该修改成功。**

秒杀业务的优化

（1） 将请求尽量拦截在系统上游，而不要让锁冲突落到数据库。

传统秒杀系统之所以挂，是因为请求都压到了后端数据层，数据读写锁冲突严重，并发高响应慢，几乎所有请求都超时，访问流量大，下单成功的有效流量小。一趟火车2000张票，200w个人同时来买，没有人能买成功，请求有效率为0。

（2）充分利用缓存。

秒杀买票，这是一个典型的读多写少的业务场景：

- 车次查询，读，量大
- 余票查询，读，量大
- 下单和支付，写，量小

一趟火车2000张票，200w个人同时来买，最多2000个人下单成功，其他人都是查询库存，写比例只有0.1%，读比例占99.9%，非常适合使用缓存来优化。

秒杀业务，常见的系统分层架构如何？

秒杀业务，可以使用典型的服务化分层架构：

- **端**（浏览器/APP），最上层，面向用户
- **站点层**，访问后端数据，拼装html/json返回
- **服务层**，屏蔽底层数据细节，提供数据访问
- **数据层**，DB存储库存，当然也有缓存

一、端上的请求拦截（浏览器/APP）

每隔x秒才向后台发送一次数据请求。

二、站点层的请求拦截

如何抗住程序员写for循环调用http接口，首先要确定用户的唯一标识，对于频繁访问的用户予以拦截。
在站点层，对同一个uid的请求进行计数和限速，例如：一个uid，5秒只准透过1个请求，这样又能拦住99%的for循环请求。

> 一个uid，5s只透过一个请求，其余的请求怎么办？

缓存，页面缓存，5秒内到达站点层的其他请求，均返回上次返回的页面。
*画外音：车次查询和余票查询都能够这么做，既能保证用户体验（至少没有返回**404页面），又能保证系统的健壮性（利用页面缓存，把请求拦截在站点层了）。*

OK，通过计数、限速、页面缓存拦住了99%的普通程序员，但仍有些高端程序员，例如黑客，控制了10w个肉鸡，手里有10w个uid，同时发请求，这下怎么办？

三、服务层的请求拦截

并发的请求已经到了服务层，如何进拦截？
服务层非常清楚业务的库存，非常清楚数据库的抗压能力，可以根据这两者进行削峰限速。

例如，业务服务很清楚的知道，一列火车只有2000张车票，此时透传10w个请求去数据库，是没有意义的。
画外音：假如数据库每秒只能抗500个写请求，就只透传500个。

用什么削峰？请求队列。

对于写请求，做请求队列，每次只透传有限的写请求去数据层（下订单，支付这样的写业务）。
只有2000张火车票，即使10w个请求过来，也只透传2000个去访问数据库：

- 如果前一批请求均成功，再放下一批
- 如果前一批请求库存已经不足，则后续请求全部返回“已售罄”

对于读请求，怎么优化？

cache抗，不管是memcached还是redis，单机抗个每秒10w应该都是没什么问题的。
画外音：缓存做水平扩展，很容易线性扩容。

如此削峰限流，只有非常少的写请求，和非常少的读缓存mis的请求会透到数据层去，又有99%的请求被拦住了。

四、数据库层

没啥压力了。

> 按照上面的优化方案，其实压力最大的反而是站点层，假设真实有效的请求数是每秒100w，这部分的压力怎么处理?

解决方向有两个：

（1）站点层水平扩展，通过加机器扩容，一台抗5000，200台搞定；

（2）服务降级，抛弃请求，例如抛弃50%；

原则是要保护系统，不能让所有用户都失败。