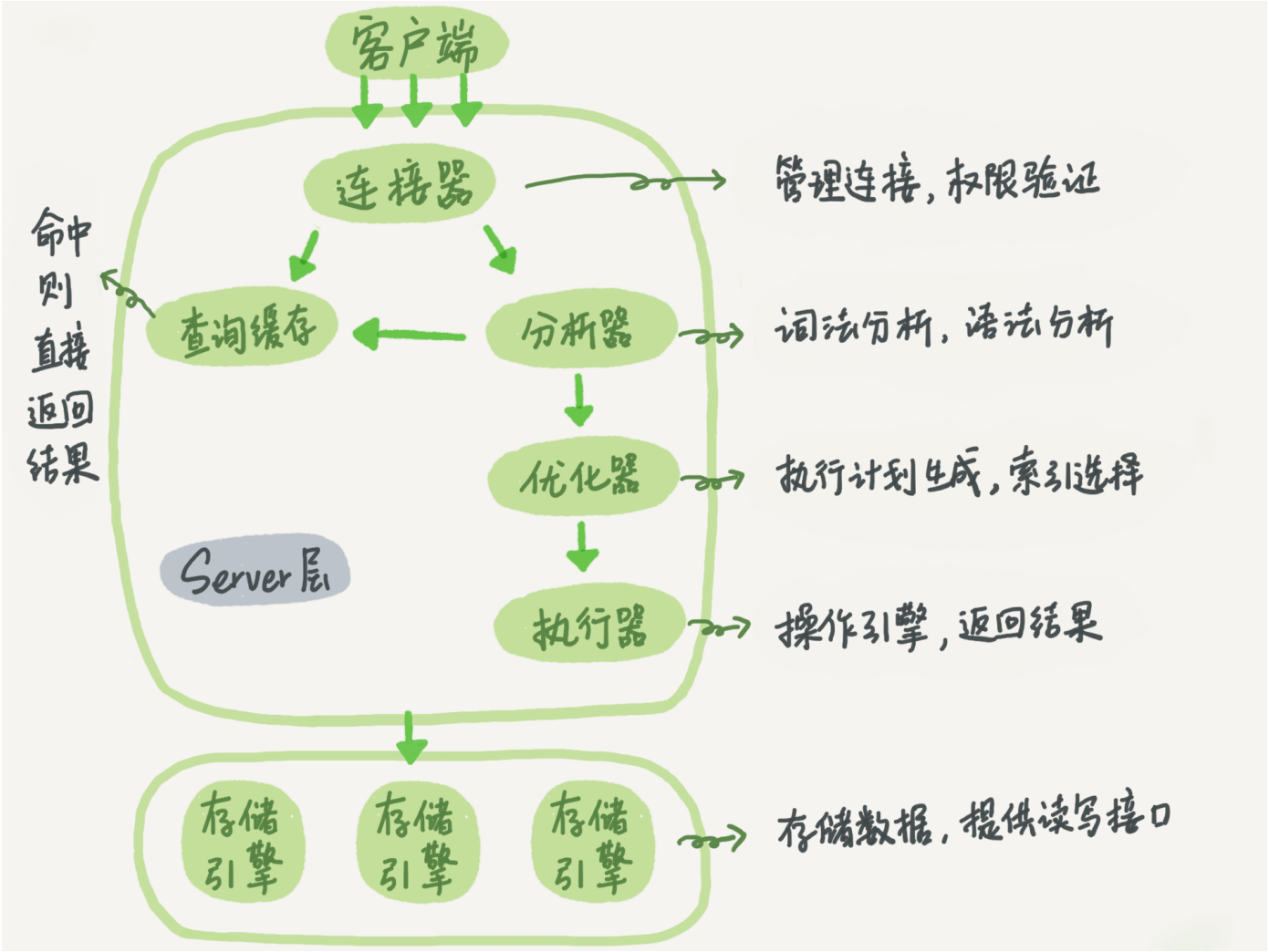
MySQL（残酷学习版

MySQL 的整体架构



连接器

连接器就是负责管理连接的，包括权限验证等等流程，因为连接是 TCP 的可能还包括连接状态的维护。

分析器

分析器的作用是对 SQL 进行词法分析，语法分析，抽出 BST，并交给后续的组件。

优化器

优化器是 MySQL 中对 SQL 语法的分析器以及索引的选择器，会根据解析出进来的 SQL 语句结合索引以及取样数据选择索引。

因为其中还包含数据的影响，所以即使符合最左前缀匹配也无法 100% 确定是否真的会走索引。

例如，如果优化器根据数据推测全表扫描的速度大于走索引再回表的速度，那么就会直接放弃索引。

执行器

执行器就相当于是一个调度器，会根据表选择的存储引擎调用不同的存储引擎的接口。

执行器对于上层的优化器屏蔽了底层不同存储引擎带来的查询语法上的差异性。

MySQL 的存储引擎是可插拔式的，在创建表的时候就可以使用不同的存储引擎。

早期的 MySQL 还会有查询缓存层，但是在4.x版本中就已经被删除了。

> Q: 为什么要删除查询缓存?

查询缓存是以查询语句为 Key，作为命中的要求，所以命中率并不会高，而且大量的缓存在业务逻辑层实现灵活性更高，更加可控，也就实在没必要在数据库中增加缓存机制。

MySQL Transaction（事务

事务的特性有如下四种，**原子性（Atomicity），一致性（Consistency），隔离性（Isolation），持久性（Durability）**。

原子性指的是事务的操作作为一个不可再分的整体，**要不同时完成要不同时失败**。

隔离性指的是**多个事务并发执行的时候，互相之间的可见性**，多个事务之间互相干扰的情况。

> 隔离性并不是说强制的完全不能看到，类似 InnoDB 提供了多种的隔离级别，低级别隔离也有使用场景。

持久性好理解，**就是保证数据不丢失，在事务提交之后，事务造成的变更就是永久性的。**

一致性指的是事务执行的前后，数据库中的数据都处于一种稳定状态，可能不太好理解，简单举例可以参考转账的操作，转账前后总额是不会增加的。

InnoDB 下事务的隔离级别

MySQL InnoDB 中提供了四种隔离级别：

- READ UNCOMMITTED 读未提交
- READ COMMITED 读已提交
- REPEATABLE READ 可重复读（Default）
- SERIALIZABLE 序列化

四种隔离级别分别解决了不同的并发问题：

隔离解别	脏读	不可重复读	幻读
Read Uncommitted	Y	Y	Y
Read Committed	N	Y	Y
Repeatable(default)	N	N	Y（InnoDB - N）
Serializable	N	N	N

> 简单理解一下这三种并发问题：

> **脏读（读未提交），事务读取到了其它事务中未提交的数据。**
>
> **不可重复读，事务前后多次读取内容不一致。**
>
> **幻读，事务前后多次读取总量不一致。**

RC 解决脏读依靠的就是锁和MVCC。

> MVCC 在 InnoDB 的 RR 和 RC 级别下表现是不一样的，RR 级别下，MVCC 以第一次 SELECT 查到的数据为主不会再创建新的快照，但是 RC 级别下，MVCC 机制每次都会创建新的快照，所以也会存在前后数据不一致的情况。

InnoDB 中 ACID 的实现

首先是原子性，**InnoDB 使用 Undo Log 实现了原子性**，基本原则就是在失败之后回滚之前的操作。

> InnoDB 的 Undo Log 会根据数据行的版本指针组成一个链表，回滚时可以根据链表向上追溯。

隔离性，InnoDB 的隔离性是依靠 MVCC 和 锁来实现的。

InnoDB 中提供了多种不同的隔离级别，每个隔离级别使用不同类型的锁和 MVCC 表现形式来支持隔离性。

> 锁定义了事务并发时访问权限，MVCC 减少了部分上锁的情况（主要还是增加性能。

持久性，InnoDB 中的持久性是依靠的 redo log 以及 undo log 实现的。

> 并说不清为什么还需要 undo log，但是因为 WAL 机制，redo log 是在数据修改前就已经持久化的，在事务提交的时候可以保证 redo log 已经落盘了，大部分情况下 redo log 就已经能保障数据的持久性了。

一致性... emmm 不太清楚，应该是其他的东西一起保证的。

InnoDB 中的MVCC（多版本并发控制）

MVCC（**Multi-Version Concurrency Control**） 在我看来是 InnoDB 中一个非常重要的特性，很大程度上提高了 MySQL 的并发性能。

MVCC 机制在数据行中保留了多版本的数据，使用数据行隐藏字段 roll_pointer（回滚指针）串联起一个版本链，可以顺着版本链回滚数据行。

> 因为是链是逐个遍历直到找到当前事务可以看到的数据行，所以当链很长的时候可能会拖慢查询。

基于 MVCC，InnoDB 引入了一个快照读的概念，相对应的还有当前读，快照读指的是当前查询语句读取的是快照的内容，当前读读取的就是当前的真实数据。

> 这里的快照不同于 Redis 的 RDB，是基于隐藏字段 trx_id 实现的可读范围标识。

InnoDB 的行记录包含了两个隐藏字段：

字段名	含义
trx_id	事务Id，由存储引擎统一下发，确保递增
roll_pointer	回滚指针

回滚指针指向的是当前行上次的数据，以此形成一个版本链，如果需要回滚到最先版本的数据，需要顺着 roll_pointer 一直往上。

MVCC 根据 trx_id 的大小界定出可见范围。

事务开始时，会额外保存当前最大和最小的 trx_id，并且保存当前未提交的事务 trx_id 数组。

> 这里的事务开始是指第一次查询，而非 start tran。

小于最小的 trx_id 标识已经提交，所以可见，大于最大的 trx_id 表示开始时还未开启，所以不可见。

如果在中间，则判断 trx_id 数组是否包含来标识是否可读。

MVCC 特性仅仅在 RC 和 RR 级别下生效，并且在两个级别下的表现形式不同。

在 RC 级别下，每次查询都会创建一个快照，而在 RR 级别下，只有第一次查询会创建一个快照。（Important

> 这就导致了事务的表现不同。
>
> 在 RC 级别下，事务可以查看到别的事务已经提交的数据，这样就造成了不可重复读，也无法避免脏读。
>
> 而在 RR 级别下，事务只会以第一次查询语句为准创建快照，所以 RR 级别下不会出现所谓的不可重复读问题。

参考文档

- 相见恨晚，MVCC 这么理解，早就通关了
- 『浅入深出』MySQL 中事务的实现

MySQL Index（索引

索引就是用来加速查询速度的特殊结构。

InnoDB 中的索引结构

常见的索引结构有B+树，B树或者 Hash 索引，倒排索引。

> InnoDB 中以B+树为主，存在自适应 Hash索引，提供特殊形式的查询优化。

B+ 树就是平衡的搜索树，可以简单理解为是二叉搜索树（BST）或者二叉平衡树（AVL）的变种。

AVL **就是任意节点左右子树的高度差不超过1的树，**为了维持这种特性需要大量的旋转操作（添加最多旋转两次，但是删除最多需要 Lg(N)，并且随着深度的增加搜索的效率也会慢慢降低，在动辄千万亿万的数据的数据库中，二叉搜索树明显是不合适的

M 阶的B+树，根节点的节点数为[2,M]，索引节点的节点数为[M/2,M]，**而且保证了数据的有序性质**，所以层次更低，查询速度更快也更平稳。

> 因为B+树数据有序性的特点，所以如果不使用单调递增的索引键，在插入和删除操作时候就会存在页分裂和页合并的问题，十分影响效率。

B 树和 B+树 的区别如下：

1. B+ 树的所有数据都在子节点中，这样的**查询更加稳定**，所有的查询都需要树高度次的查询。
2. B+ 树所有的子节点**组成一个链表**，这样非常便于**实现范围查询**。

相对于 B 树和 Hash 来说，B+树更加符合磁盘的特性。

> 为什么说 B+ 树更加符合磁盘特性呢?

相比于 Hash 来说，B+ 树的层级更低（往往只有三四层，对于 B+ 树的存储来说，往往一个节点就是一个数据页，因此正常情况下 3～4 次磁盘 IO 就可以获取到想要的内容，并且 B+ 树将所有的叶子节点连成链，更加适合范围查询。

相比于 B 树，除去叶子节点成链不说，B+ 树的非叶子节点不保存数据，具有更稳定的查询效率，B 树虽然在某些查询中可以更快速，但是整体查询并不稳定，读取同样大小的索引页 B+ 树有更多的索引项。

> MySQL 为什么不使用 Hash 表或者跳表作为索引实现?

Hash 表只适合等值查询，几乎无法做范围查询。

> 为什么不使用跳表的原因如下：

MySQL 的主要数据还是保存在磁盘中，相对于跳表，B+ 树更加适配磁盘的特性，每个索引块可以保存在一个盘页。

（以 Redis 为例，如果纯内存的数据库跳表应该和 B+ 树访问速度差不多。

索引分类

聚集索引和非聚集索引

聚集索引并**不是一种单独的索引类型，而是一种数据的存储方式**，在 InnoDB 中，主键索引就是聚集索引，**所有的数据都保存在主键索引的叶子节点中，数据按照主键的顺序排列存储。**

InnoDB 中，主键索引决定了数据的物理存储顺序，应该更能理解主键的乱序插入带来的页分裂等等问题了。

> 如果没有明确定义表的主键，MySQL 也会挑选一个唯一键作为主键，如果没有唯一键则会生1成一个 rowId 作为主键。

非聚集索引就是非聚集索引，**和聚集索引相反的它的逻辑顺序和物理的存储顺序就是完全无关的。**

InnoDB 的实现中，次级索引都是非聚集索引，保存的是主键。

> 所以 InnoDB 中存在回表操作，就是在一个索引树中无法完全确定数据是否可用时，先返回主键，查询完整的数据再来判断。
>
> 增加单索引中字段，索引下推，索引联合都可以起到减少回表的作用。

MyISAM 中的非聚集索引实现不同，MyISAM 中所有的索引树都是非聚集索引，包括主键在内，保存的都是数据的真实地址。

> MyISAM 和 InnoDB 的不同在这里就有体现：
>
> MyISAM 支持没有主键，理论上来说 MyISAM 的主键索引和次级索引没有任何区别。
>
> MyISAM 的索引中保存的都是数据地址，而 InnoDB 的次级索引保存的主键。

稠密索引和稀疏索引

稠密索引会为每一个键值建立一个索引记录，可以i加快查询速度，但是需要更多的空战占用以及维护成本。（类似 MySQL 中的主键索引

稀疏索引不会为每一个键值建立索引，这种索引往往出现在有序的排序中，例如跳表结构就是稀疏索引的典型实现（Mongo 以及 Kafka 都算是稀疏索引，Mongo 的文档可能会缺失某些字段？Kafka 是以时间戳为序间隔一定长度建立索引项

唯一索引和非唯一索引

唯一索引就是在表内需要保证字段值全局唯一的索引。

> 唯一索引是保证不重复调用或者记录唯一的有效手段。
>
> 比如希望点赞数不重复被记录，那么就可以将帖子Id和用户Id组成一个唯一索引，确保一个用户只能对一个帖子点赞一次。

在 InnoDB 中唯一索引还会导致一些另外的问题，有好也有坏，但影响其实都不大，仅做了解：

1. 首先等值查询时，如果查找字段有唯一索引，那么查询到一条记录就会返回，而非唯一索引会顺着链表继续查询到一条不相等的记录。
2. 在插入或者修改数据的时候，InnoDB 的 Change Buffer 可能有效的减少随机读操作，而唯一索引无法使用该特性，因为在修改或者插入前都需要判断是否唯一

> Q：什么是 Change Buffer?

Change Buffer 早期又称为 Insert Buffer，在数据插入时生效，后面扩展到数据的修改。

Change Buffer 主要优化非唯一辅助索引的维护成本。

在涉及到数据修改时，如果记录所在数据页在内存中则直接修改，如果不在可能需要先加载再修改，此时这个加载过程就是随机读的过程，相对于顺序读而言随机读的效率低了不止一点点。

所以在修改的时候，InnoDB 会把这些更新操作缓存到 Changge Buffer 中，日志正常保存，即使宕机也能根据日志恢复。

保存在 Change Buffer 的数据在下一次读取到数据页时合并，也就是 Merge 过程。

前缀索引

前缀索引是指在一个长字符串字段中，可以选取其中N字节长度的前缀作为索引。

> 长字符串的索引除了使用前缀索引，还可以直接独立一个字段做hash，搜索会更加全面。

索引使用的相关算法

最左前缀匹配

最左前缀匹配在联合索引中是一个非常重要的概念，**就是依据左前缀判断是否可以使用该索引。**

> 简单的例子，联合索引[a,b,c]，可是使用该索引的查询条件是[a]，[a,b]，[a,b,c]，但是绝对不包括[b,c]等不以a开头的查询条件

本质上来说，联合索引在 InnoDB 中的数据结构仍然是一棵 B+ 树，并且索引节点保存以声明顺序所表示的索引数据。

> 例如[a,b,c]，在索引树中的排序就是先按照a排序，a相同按照b排序，b相同按照c排序。
>
> ！！！利用索引有序的结构，可以完美的优化查询语句中的排序，但是在联合索引中，如果搜索条件是[a,b]并且按照b排序就不会出现文件排序，因为在a相同时，b本身就是有序的。
>
> 但是在搜索条件为[a,c]时，当a相同时，c并非有序，所以查询会出现 file sort。

覆盖索引

覆盖索引是指在索引树中的内容已经包含了需要查找所需要所有字段，所以可以直接返回而跳过回表。

> 回表可以简单理解为使用二级索引查询获得主键之后，为了获得更多的数据而需要再一次扫描主键索引树。
>
> ！！一般来说扫描二级索引树获得的主键，会返回给 Server 层，由 Server 再次发起查询。

有些时候大量的回表会导致查询的效率十分低下，此时适当冗余索引字段也不失为一个好办法。

索引下推

索引下推是在 MySQL 5.6 引入的对索引使用方式的优化，在次级索引树的遍历过程中，尽量多的使用索引树中的字段。

> 在5.6之前，[a,b,c]索引查询[a,c]，只能使用到a字段，c字段就需要回表之后判断，如果a的筛率不高就会有大量的回表，
>
> 而在5.6以后，c字段也能下推判断，进一步的判断也减少了回表的记录数，加快了查询速度。

索引联合

索引联合了解的不多，在使用or的等值查询过程中可能会用到索引联合，搜索两棵索引树在做值的整合，相当于 union all 吧。

> 虽然索引有这好那好，但是走哪个索引还是依据优化器的，优化器也是根据抽样统计信息的，偶尔也可能出错。

建立索引的思考

1. 联合索引的字段排序（a，b，c 的联合索引，b 相对于 a 有序，c 相对于 b 有序，如果需要以a排序就可以建立（b，c，a 或者 c，b，a）索引，消除排序
2. 字段的区分度（比如 sex，存它干嘛呢，撑死了三个值
3. 实用程度（？，有些使用频率低的 SQL，可能并不需要特定的索引，索引也需要消耗一定的空间，并且降低更新和插入的效率。

MySQL Log（日志

MySQL 中存在多种日志，比如 **binlog ，redo log 以及 undo log。**

redo log 和 undo log 属于 InnoDB 层的日志，而 binlog 属于 MySQL Server 层的日志。

binlog 主要用于主从复制，数据归档（可以单独根据 binlog 实现数据恢复，但不能保证 crash-safe

redo log 和 undo log 共同实现原子性，在正常的回滚下可能仅仅需要 undo log 来进行行记录的回滚，但是如果是经过 crash 则需要 redo log 来判断事务是否已经提交。

undo log 在 InnoDB 中另外实现了 MVCC。

binlog（归档日志

binlog 属于归档日志，在 MySQL 中属于 Server 层日志， **MySQL 中所有的存储引擎都会记录该日志。**

binlog 记录了所有的**数据库变更操作**，包括 UPDATE，INSERT，DELETE，也包括表结构的修改 ALTER TABLE 等。

binlog 的主要作用是 1. 主从复制 2. 数据归档（奔溃后的适度恢复）。

但是 binlog 并不能提供 crash-safe 的保证。

日志格式

binlog 有如下三种格式：

1. statement

statement 完整的保存执行的语句，但是因为 now() 等即时函数的存在复制的异常，所以用于复制的情况下会出现异常。

1. row

row 记录的是表中数据 完整的变更，比如 now() 就会直接记录当前时间，数据较为准确，不会受语句上下文环境的影响

但是相对的日志文件会比较大，因为 statement 一个删除语句，row 会保存所有的行记录。

1. mixed

mixed 基本上就是混合两种的情况。

相关配置

再说两个 binlog 相关的主要配置：

1. binlog_cache_size

该值表示在 MySQL 中 binlog 缓冲区的大小，用于缓存事务产生的 binlog。

binlog cache 是线程私有的，不同线程之间不共享缓冲区。

因为一个事务中可能涉及多个更新语句，并且多个更新语句不能拆分写入，因此需要单独一个缓冲区。

1. sync_binlog

binlog 的 fsync 刷盘策略，有如下几种配置形式：

- 0 - 系统自由决定何时刷盘，所有 binlog 只做 write
- 1 - 每次都需要刷盘
- n - 每次提交事务都会 write，但是n次的操作之后才会进行刷盘

为0时性能最好，但是如果系统宕机，会丢失未落盘的内容。

redo log（重做日志

redo log 是 InnoDB 的日志，根据 Write Ahead Log（WAL）机制和 force log at commit 机制保证数据的持久性。

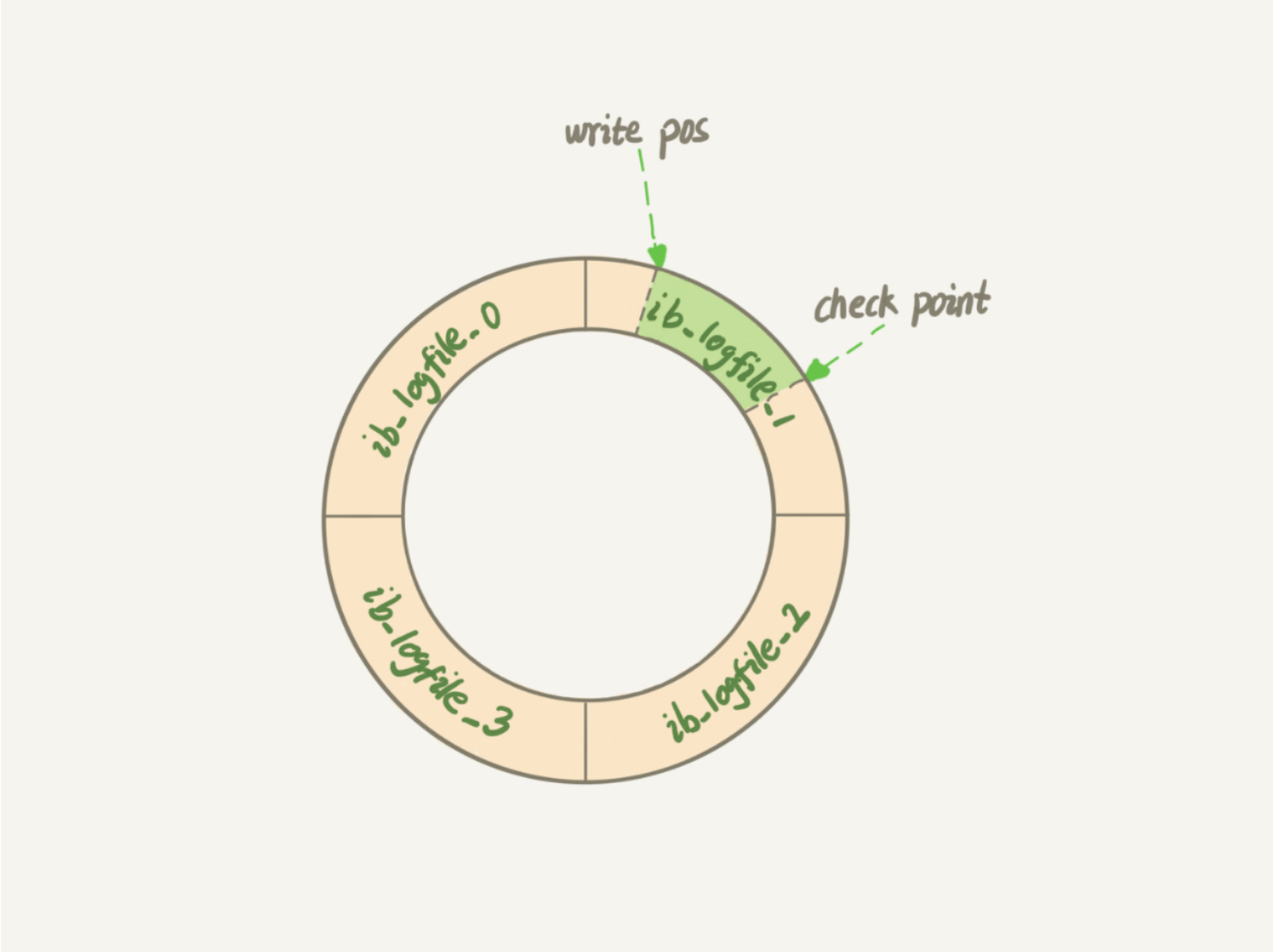
WAL（Write Ahead Log ）就是预写日志技术，在 InnoDB 中所有的修改都需要写日志，再做修改内存数据，同时在提交事务的时候也需要先将日志落盘。

所以理论上 redo log 可以单独提供 crash-safe 的保证。

redo log 中记录的是每次修改的物理日志，即每个数据页的修改（包含主键索引和次级索引。

redo log buffer

InnoDB 中的 redo log 有一个固定的大小的缓冲区，并且首尾相连组成一个环，环上有两个主要的指针: check_point 和 write_pos。



在 write_pos 和 check_point 之间的就是日志的可写范围，如果刷盘不及时导致 write_pos 追上了 check_point，就会开启强制的刷盘（所以在 MySQL 大量写的时候会有瞬间抖动的现象。

另外在 MySQL 的后台线程也会定时刷盘，在正常关闭 MySQL 的时候也会将 redo log 落盘。

redo log 的配置

redo log 的刷盘策略也有参数控制 - **innodb_fluish_log_at_trx_commit**（这个非常非常重要！。

1. innodb_fluish_log_at_trx_commit

该参数为1时，每次的 redo log 都会调用 fsync，真正落盘持久化保存。

undo log（回滚日志

undo log 在 InnoDB 中用于实现 MVCC 和原子性。

InnoDB 在修改行记录都会带有几个隐藏字段：

1. TRX_ID - 修改的事务Id（事务 Id 由 MySQL 统一下发保证全局唯一并且递增。
2. ROLL_PTR - 回滚指针，指向旧版本数据，数据行根据指针组成一个单向链表（新 -> 旧

MVCC 的实现就是在查询的时候沿着 ROLL_PTR 遍历到一个当前事务可见的行记录并返回（因此也存在 undo log 记录太多，导致查询缓慢的问题。

原子性的实现就是直接替换当前行记录，修改都是会上锁的所以不存在多个修改并行的情况。

undo log 在 5.6 之后记录在单独的表空间，并且使用回滚段作为组织的形式。

> 所以 undo log 并算不上 InnoDB WAL 机制的实现，因为 undo log 自身的持久化都要基于 redo log。

undo log 不会一直存在，当事务提交的时候 undo log 就没有作用了（已经提交了，当前事务不需要回滚了），但是是否要删除还得看对于目前还存活的事务 undo log 是否可用。

最终的删除还是根据后台的 purge 线程决定。

binlog 和 redo log 的 2PC（二阶段提交

binlog 和 redo log 需要做二阶段提交，保证双方日志的一致性，保证经过 binlog 复制的操作不会丢失或者被回滚。

之所以要保证一致性的原因是因为 binlog 作为归档日志以及复制功能基础，如果 binlog 已经写入的数据，redo log 回滚，就会导致主从或者恢复前后的数据不一致。

二阶段提交的流程如下：

1. 准备阶段（Storage Engine（InnoDB） Transaction Prepare Phase）

该阶段生成 XID（事务Id），进入 PREPARED 阶段，此时 binlog 不需要落盘，但 redo log 需要先落盘。

该阶段可能执行多次，每次修改都需要将 redo log 落盘。

1. 提交阶段（Storage Engine（InnoDB）Commit Phase）

如果将事务提交，则将 binlog 落盘，如果回滚则使用 undo log 进行回滚。

1. 完成阶段

事务提交或者回滚都需要看情况清除对应的 undo log。

binlog 在 2PC 中充当了事务的协调者（Transaction Coordinator），并且以 binlog 是否写入来判断事务是否成功，使用 XID 建立当前日志之间的对应关系。

在恢复的时候，redo log 检查到最近的 checkpoint，然后查看之后的日志，需要确定事务是否已经提交则通过 XID 找到对应的 binlog 俩判断 commit 状态。

group commit（组提交策略

沙尘暴也阻挡不了学习的脚步-- 面试官：你竟然不知道MySQL的组提交

checkpoint（检查点

checkpoint 就是将脏页刷回磁盘的机制。

当 MySQL 重启后会第一时间定位到最后的 checkpoint，在 checkpoint 之前的数据就不需要做恢复，只需要对其后的数据做恢复（按照 redo log。

checkpoint 可以分为以下两种：

- Sharp Checkpoint

在当 MySQL 关闭的时候，需要将所有的脏页刷回磁盘，此时 checkpoint 会直接拉到日志最末尾。

- Fuzzy Checkpoint

基本就是刷脏页的触发时机，包含后台定时线程触发，redo log buffer 里 write_point 追上 checkpoint 触发，LRU 空闲页面不够的刷盘。

LSN（Log Sequence Number，日志序列号

LSN 表示的就是日志的序号，在 InnoDB 中占8个字节。

表空间中的数据页、缓存页、内存中的 rodo log、磁盘中的 redo log 以及 checkponit 都有LSN标记。

崩溃恢复的流程

基于 Redo Log 和 Undo Log 的 MySQL 崩溃恢复流程

reference

- Redo log,Undo log 和 Binlog
- 详细分析MySQL事务日志(redo log和undo log)

InnoDB 特性

一、 Change Buffer（修改缓存

Change Buffer 的主要作用就是**缓存对二级（辅助）非唯一索引的修改**（早期只在 Insert 操作中生效，称为 Insert Buffer。

Changer Buffer 属于日志的一种，在 InnoDB 底层的 Buffer Pool 中会占据一定的空间。

如果没有 Change Buffer，在一次数据更新中会需要将数据所有的索引树加载到 Buffer Pool 之后再做更新（因为 Redo Log 的存在，所以此时 Buffer Pool 不需要立即刷到磁盘中。

Change Buffer 会在适当的时候进行 Merge，例如当索引页被加载到 Buffer Pool 的时候，或者服务空闲的时候，服务关闭之前等等。

Change Buffer 的机制可以和 redo log 做类比，redo log 减少了随机写的操作，而 Change Buffer 减少了随机读的操作（对于磁盘操作顺序操作比随机操作快了好几倍。

二、Double Write（两次写

InnoDB 的两次写是为了防止部分页刷新的问题。

默认的 InnoDB 内部的 Buffer Pool 的页大小为 16kb，但是系统写文件却大部分以 4kb 为单位，此时可能就出现页数据没有被完全写入就奔溃的情况。

MySQL 在磁盘共享空间中会创建一个 Double Write 的区域用于存放临时数据。

所有的脏数据写入会分为两次，一次写入 Double Write 的磁盘区，而后在将脏页具体刷盘。

三、Flush Neighbor Page（刷新邻接页

刷脏页的时候连带着将附近的一起刷了（处处透露着优化。

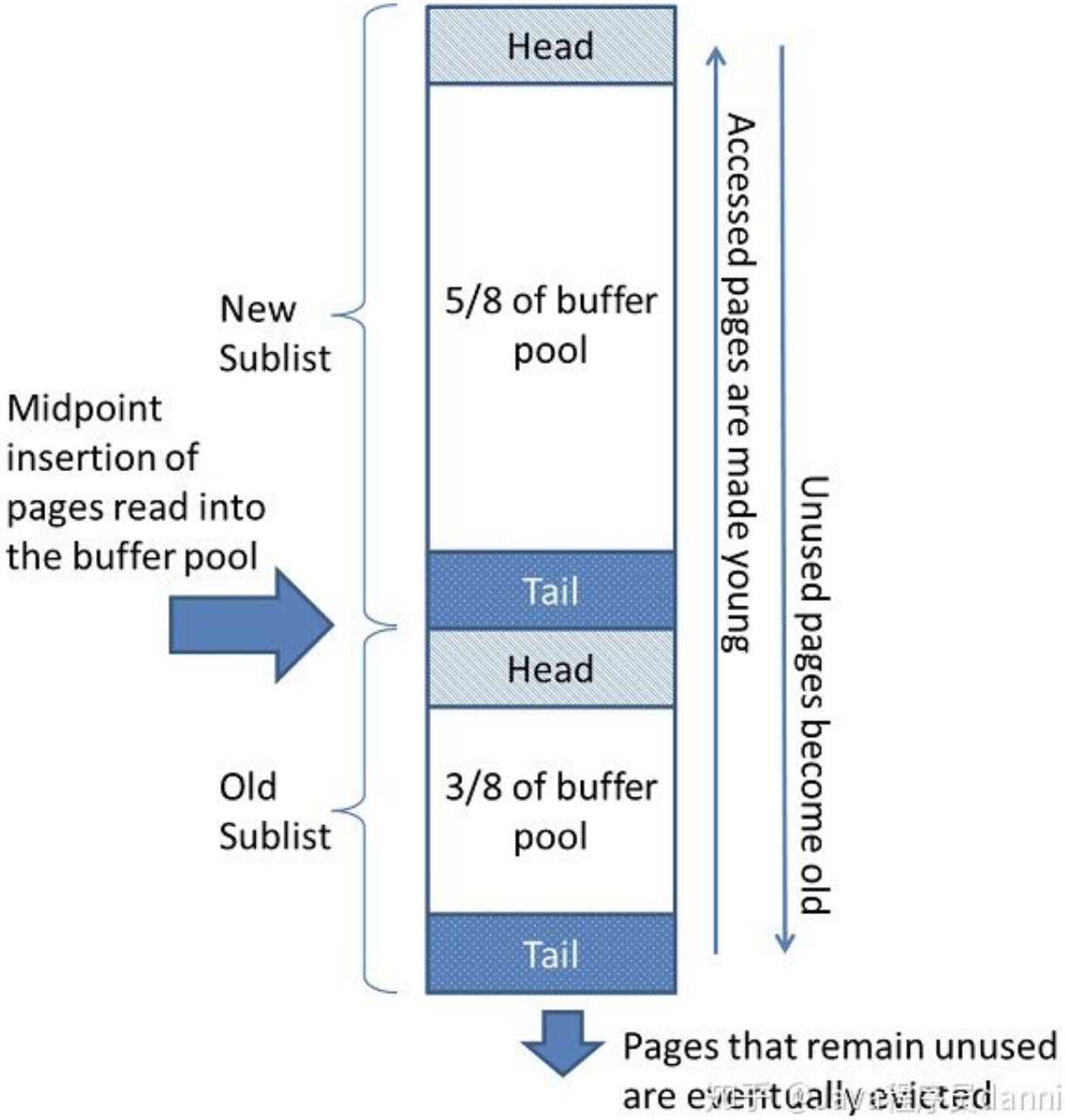
四、自适应 Hash 索引

TODO

InnoDB 的内存管理（LRU

InnoDB 底层会申请一片的 Buffer Pool 用于保存数据页以及其他数据，默认情况下一页为 16kb（通过 innodb_page_size 控制。

数据页会根据 LRU 算法进行保存，InnoDB 的 LRU 经过一定的优化，链表将其前 3/8 的部分作为热数据区，后面的属于冷数据区，中间点可以称之为 midpoint。



读取的新数据并不是直接添加到链表的头部，而是添加到冷区头部，在一定的时间内被访问才会进入到热区。

再次基础上，LRU 还优化了热区的移动逻辑，热区前1/4的数据被访问时不会再被移动（用来减少指针移动带来的锁。

此类优化适合数据库查询相结合的，因为部分查询可能会大量查询到无用的数据页（类似全表扫描），如果一股脑全部填充到首部会将真实的热数据冲散。

页分裂问题

InnoDB 中每张表都要求必须有一个主键Id（没有就隐式生成一个 row_id，并且主键索引就是聚簇索引，因此都是按照主键 Id 的顺序存放的数据。

当前主键 Id 递增时，每次都是在最后一页新增一行数据，如果超出则新申请一页，但当在两个主键中插入一个中间值时，此时如果页面数据也是满的就可能产生页分裂的情况。

页分裂的情况会导致部分数据需要被拷贝到新的数据页，因此也会显著降低插入的效率。

MySQL 锁相关

MySQL 中的锁需要根据存储引擎的不同来说。

在 MyISAM 中仅仅只有表锁，所有的插入操作都需要事先锁表，而 InnoDB 支持表锁，行锁，甚至多级上锁。

InnoDB 的基础锁类型

InnoDB 中根据划分依据的不同存在多种不同的锁。

根据锁的粒度，或者说锁的目标来划分，存在以下几种锁：**行锁，表锁，间隙锁。**

根据锁的排他性或者锁的目标行为来划分，存在**写锁和读锁。**

二阶段锁协议

二阶段锁协议的简单理解就是随机上锁，最终（事务提交）解锁。

在 InnoDB 中加锁的过程是根据语句的执行过程慢慢加的。

> 例如 `INSERT INTO ... SELECT ... FROM` 语句，该语句如果用于迁表，那么就会感觉到上锁的过程是跟随语句的执行过程慢慢发展到锁表。

而锁的释放是在事务提交之后一次性释放的。

> 中间可能会有一些优化，类似于 AUTO_INCREMENT 带来的插入锁，会提前释放，并且一些行锁也可能提前释放，但总得来说大部分的锁都还是在事务提交时被释放的。

读锁和写锁

读锁和写锁是锁的两个程度，读锁就是所谓的排他锁，而写锁就是所谓的共享锁。

> Javaer 可以直接联想 ReadWriteLock。

读锁和读锁之间相互兼容，写锁排斥一切。

因为 MVCC 的存在，所以 MySQL 的查询一般来说是不会上锁的（因此，就算在写也是可以读到内容，并不是读写相融。

强行上锁可以使用以下语句加锁：

```
// 读锁
SELECT * FROM tableName WHERE ... LOCK IN SHARE MODE
// 写锁
SELECT * FROM tableName WHERE ...  FOR UPDATE
```

划分读锁和写锁的意义就在于，让两个读锁可以通知执行，增加并发度（基于 MVCC 的无锁化实现才是性能提升的最大原因。

行锁和表锁以及意向锁

根据粒度划分，InnoDB 中存在行锁，也就是对**表中的单行记录上锁**，也有表锁，可以对**整张表上锁**。

> ！！！ InnoDB 中并没有真正意义上的表锁，就是直接对表上锁的那种，而是通过行锁+间隙锁的形式锁表。
>
> 多粒度的锁也是 InnoDB 的特性之一，MyISAM 就只有表锁。

对于常规的 CURD 语句，判断行锁还是表锁，简单来看就是是否走索引，不走索引的 CRUD 语句都会经过一个全表扫描的过程，扫描过程中慢慢的就会锁表。

InnoDb 支持多粒度上锁，即表锁和行锁，如果表锁和行锁都为读锁，那也不会冲突，而如何在上表锁的时候判断是否在表中存在行锁就会出现问题，总不能扫表来判断是否有锁吧，此时就出现了意向锁。

意向锁是为了兼容多粒度的锁而设计的，在上读锁的同时会给对应的表上读的意向锁，此时上写的表锁会被意向锁卡住。

GAP 锁

GAP 锁的锁定目标就是两个索引记录之间的区域（左开右闭），GAP 锁的目的就是为了防止其他的事务在间隙（GAP）范围内插入数据。

GAP 锁是共享锁，也就是说两个事务可以同时对相同的 GAP 上锁（咩有写的 GAP 锁。

GAP 锁仅仅在 RR 级别下生效。

Next-Key Lock

Next-Key Lock 就是**行锁和 GAP 锁的结合**，GAP 锁锁定的是命中的索引记录之前的间隙。

Next-Key Lock 的存在使 InnoDB 在 RR 级别下面就可以解决幻读问题。

死锁

死锁出现的情况就是互相持有对象需要的锁。

> 例如，持有A资源，等待B资源的线程和持有B资源，等待A资源的线程会造成死锁。

死锁的必要条件：

1. 资源互斥 - 只有一个对象可以使用资源
2. 占有等待 - 在等待另外的资源期间，已有资源并不会释放
3. 不可强占 - 资源不可强行剥夺，即无法强行获取别的所持有的资源
4. 循环等待 - 若干对象循环持有对方所需要的资源

如何避免死锁（减少死锁的发生：

1. 缩小事务范围

> MySQL 的上锁是逐步的，扫描索引树的时候逐步上锁，并且在事务提交的时候才会释放，所以缩小事务范围可以有效减少死锁的发生。
>
> 因为事务的解锁统一在事务的提交的时候，所以即使不同表的更新也会造成死锁。

1. 尽量使用主键索引更新语句

> 避免对索引树的扫描导致一次更新覆盖太多的行。

1. 以相同的顺序更新

> 死锁的原因是在更新多条记录的时候，互相持有部分记录的锁（单条记录的更新不会有死锁的问题。
>
> 所以将更新的顺序改为一致就可以解决死锁的问题，改死锁为等待（同个事务下更新的执行可以认为是无关先后顺序的，都是在提交的一刻生效。
>
> 对应的场景有 IM 中群聊会话的更新。

rference

- 浅谈数据库并发控制 - 锁和 MVCC
- 史上最全的select加锁分析(Mysql)

排序算法

MySQL 中的排序算法包括三种：

1. 全字段排序

全字段排序就是将全部需要的字段放入 sort_buffer 统一排序后返回。

1. rowId 排序

在排序内容较多的时候，可能仅使用 rowId + 排序字段进行排序，然后回表查询另外的内容。

此时的效率可能非常低，因为先根据筛选字段查询 rowId 以及 排序字段（此时可能已经经过一次回表，而排序结束之后可能再次使用 rowId 进行二次回表。

1. 索引树排序

MySQL 索引本身就是有序的，因此如果排序条件满足索引（最左匹配原则，则可以直接使用索引中的顺序。

explain 的 Extra 字段中可能出现 filesort 标记，表示出现额外排序（并不一定是磁盘排序。

相关的还有分页问题，大数据量分页的时候可能会非常的慢，因为例如 limit 1000000,1000002; 此时会将 1000002 的数据全部先排序然后在选去后两条。

此时的优化应该减少待排序内容，使用索引或者子查询。

联表查询算法

联表查询包含如下几种形式：

1. 全连接/内连接查询

全链接查询最后的数据集只会保存驱动和被驱动表都匹配的数据。

例如 select * from a,b where a.id = b.id。

此时 a 和 b 的 id 在对方表中无匹配项的就不会被返回。

1. 左连接查询
2. 右连接查询

> 普通的 A join B，会是 MySQL 自行选择驱动表，而使用 A straight_join B，会固定 A 为驱动表。
>
> 驱动表可以简单理解为先查询的数据表，会根据驱动表的数据去匹配被驱动表。

联表查询的时候应该是小表作为驱动，小表的判断依据是单个表执行完 WHERE 语句之后剩余的数据集。

Index Nested-Loop Join

TODO

Simple Nested-Loop Join

TODO

Block Nested-Loop Join

TODO

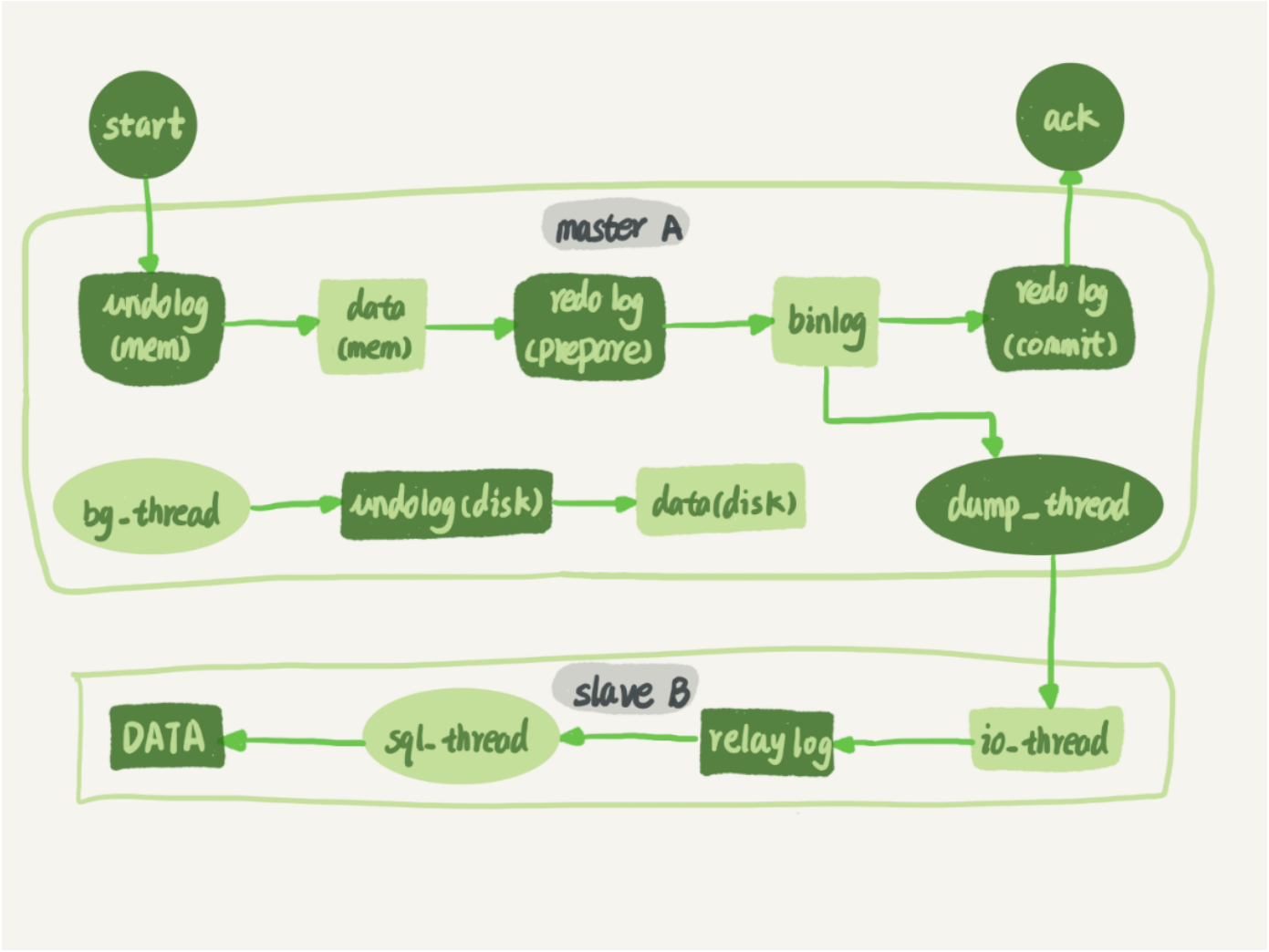
相关参数

参数名	参数作用
join_buffer_size	join buffer 的大小（在合适的范围之内，Join Buffer 肯定是越大越好

分页算法

TODO

Master-Slave Replication（主从复制



（还是 MySQL 45讲里面的图片。

复制的作用

1. 多机备份，数据安全性保证（在单机突然爆炸的情况下也能保证数据安全
2. 读性能的水平扩展（主从分离之后，可以将读写进一步分离
3. 数据的异步化处理（例如阿里 canal，可以监听 binlog 用来进一步处理

基本复制流程

MySQL 的主从复制是基于 Server 层的 binlog 实现的复制功能（可以类比于 Redis 的复制，binlog 转成 Redis 的复制缓冲区。

在 master 接收到 slave 传递的 start slave 指令后就开始复制过程，此时 slave 需要指定 binlog 的日志偏移量。

复制过程主要涉及的有以下几个线程（线程池）：

1. Dump Thread

master 侧的线程，负责从 binlog 中读取日志记录并推送到 Slave，注意读取的是 binlog 的磁盘文件（binlog 的缓存区是线程私有的也读取不到。

master 会为每个从节点创建一个 Dump Thread，从不同的起始点开始读取日志文件（因此一主多从多架构对主的要求很高。

1. I/O Thread

slave 侧的线程，负责接收从 master 请求来的日志数据，并写入 relay log（relay log 就是中转日志，负责缓存从 master 接收的日志数据。

1. SQL Thread Group

旧版本的 MySQL 可能就是单个线程，在5.?之后变成了线程组，但是因为 SQL 语句可能存在上下文语境，因此并发执行需要额外判断。

该线程组用来执行从 relay log 解析出来的 SQL 语句。

和 Redis 不同的是，MySQL 支持 Master-Master（主主）架构，此时需要双方各自指定自身的 server_id 防止日志的无限复制。

另外的 MySQL 还支持级联复制，Slave 可以复制 Slave 节点的数据，主节点只需要创建一个 Dump Thread 去扩散日志，其他的从节点都从一级从节点复制。

Replication Model（复制模式

复制模式可以对比 Kafka 的 asks 参数策略，不同的模式反映了不同的一致性和安全性保证。

Async-Model（异步模式

异步模式下，master 不会主动推送 binlog 到从节点，在接收到客户端的 SQL 以后，本地执行完毕就会返回结果，并不会关心从库是否已经接收。

该情况下，master 和 slave 可能存在明显的时间延迟，导致读写不一致，并且在 master 宕机之后如果以 slave 为新 master，可能出现数据丢失的情况。

Semi-Sync（半同步模式

半同步模式下，master 节点在执行完 SQL 之后会等待至少一个从库确定接收到对应 binlog 信息后才会返回结果。

此时写的性能至少延迟两个 TTL，并且写得性能完全看最快的节点。

Sync（全同步模式

全同步模式则是进一步强化复制过程，需要全部的 slave 都已经复制 binlog 才会返回。

GTID 模式

GTID 模式是对复制进度的表示优化，之前的流程中 slave 需要指定 binlog 的复制偏移量来获取之后的日志，但是这个比较难以界定（鬼知道我最后一条日志在 binlog 的哪里），不仅难找而且容易遗漏，所以就出现了 GTID 模式。

GTID 就是 Global Transaction Identifier 即全局事务 Id，此时每个在主库上执行的事务都会指定一个唯一的 ID（全局递增，GTID 的组成由 server_id 和 transaction_id）。

GTID 模式下，通过 GTID 代替了之前的 binlog 偏移量，可以清楚的界定出复制的进度，在事务提交的过程中也会一起记录，在 SQL 线程回放的过程中也会对比本地的 binlog 判断是否已经执行保证 SQL 复制的幂等性。

reference

- MySQL 主从复制原理不再难
- 【MySQL】主从复制实现原理详解

Explain 分析

> Explain 是 MySQL 中常规的 SQL 解析工具，能展示出SQL的部分执行逻辑和过程。
>
> 分析 Explain 的输出就能帮助我们优化和改进 SQL 语句。

示例

```
mysql> explain select * from servers;
+----+-------------+---------+------+---------------+------+---------+------+------+-------+
| id | select_type | table   | type | possible_keys | key  | key_len | ref  | rows | Extra |
+----+-------------+---------+------+---------------+------+---------+------+------+-------+
|  1 | SIMPLE      | servers | ALL  | NULL          | NULL | NULL    | NULL |    1 | NULL  |
+----+-------------+---------+------+---------------+------+---------+------+------+-------+
1 row in set (0.03 sec)
```

explain 用于展示 SQL 语句的执行计划，可以将其作为 SQL 优化的辅助工具。

主要关注的几个字段如下：

possible_keys

可能选择索引名称。

这里会将表结构中可以用到的所有索引列出，然后从中选择效率最高的执行（可能选择错误）。

如果 possible_keys 为空，表示没有任何索引可以使用，所以都会作全表扫描处理。

key / key_len

最终选择的索引，以及索引的长度。

key_len 肯定是越小越好，类型上 int 的匹配优于字符串匹配。

rows

扫描行数。

每个查询语句可能扫描的记录行数，InnoDB 中该行数只是一个粗略值（经抽样统计得出）。

extra

额外信息，表示为了完成查询 MySQL 需要做的额外的事情（这里是不是指的 Server 层需要做的事情。

额外信息	出现含义	如何解决
using where	表示在 Server 层需要额外的判断	一般来说不需要关心，不会太影响查询效率
using index	只需要读取索引文件就可以获取全部的数据，而不需要读取数据文件，表示不需要进行回表，或者直接使用索引覆盖。	可
using filesort	需要进行额外排序（不一定包含文件排序	可以利用联合索引的相对顺序避免排序
using_index_condition	使用了索引下推	

MySQL - explain output format

Q & A

InnoDB 和 MyISAM 的区别:

InnoDB	MYSQL
支持事务	不支持事务
聚簇索引（主键索引就是聚簇索引，所以必须包含主键，没有就帮你隐式创建一个	非聚簇索引（可以没有主键
count 需要扫索引树（有 MVCC 也没法记准确的	会在表中记录当前行数
支持外键（虽然没啥卵用	不支持外键
多级锁机制（行锁，表锁，Gap 锁	表锁（一个烂的摆

如果解决深度分页问题?

索引失效的常见情况

索引失效的情况	失效原因
索引列存在函数调用	注意是对索引列的函数，对索引的函数操作可能会影响索引的有序性
隐式的类型转换	和上条类似，MySQL 中通常也是使用函数来进行类型转换（在 MySQL 中，字符串和数字做比较的话，是将字符串转换成数字。
不满足最左前缀	如果存在索引下推勉强能用
左模糊匹配	类似 LIKE %XX，对于字符串类型，索引的顺序是按照字典序排列的，因此左模糊匹配也会

索引的原理就是按照有序性进行二分（一次性排除大量数据无用数据），所以在改变了有序性之后索引失效就是理所当然的。

Reference

- 官方文档