主讲老师：Fox

> 有道笔记地址：https://note.youdao.com/s/8QztzaFw

1. Mysql怎么学

索引 事务 （锁机制 日志） 集群架构

Mysql专题课程安排

1、全面理解Mysql架构（直播）

前置基础（必须掌握）：

2、深入理解Mysql索引底层数据结构与算法(录播)

3、Explain详解与索引最佳实践(录播)

4、Mysql索引优化实战一(录播)

5、Mysql索引优化实战二(录播)

进阶（深入理解）：

6、深入理解Mysql事务隔离级别与锁机制(录播)

7、深入理解MVCC与BufferPool缓存机制(录播)

8、深入理解Mysql各种日志实现机制(录播)

9、MySQL全局优化与Mysql 8.0新增特性详解（直播）

集群架构

10、深入理解Mysql8集群架构一（直播）

11、深入理解Mysql8集群架构二（直播）

Mysql快速安装

> 官方文档：https://dev.mysql.com/doc/refman/8.0/en/installing.html

Centos7安装mysql8

Docker 安装 MySQL8.0

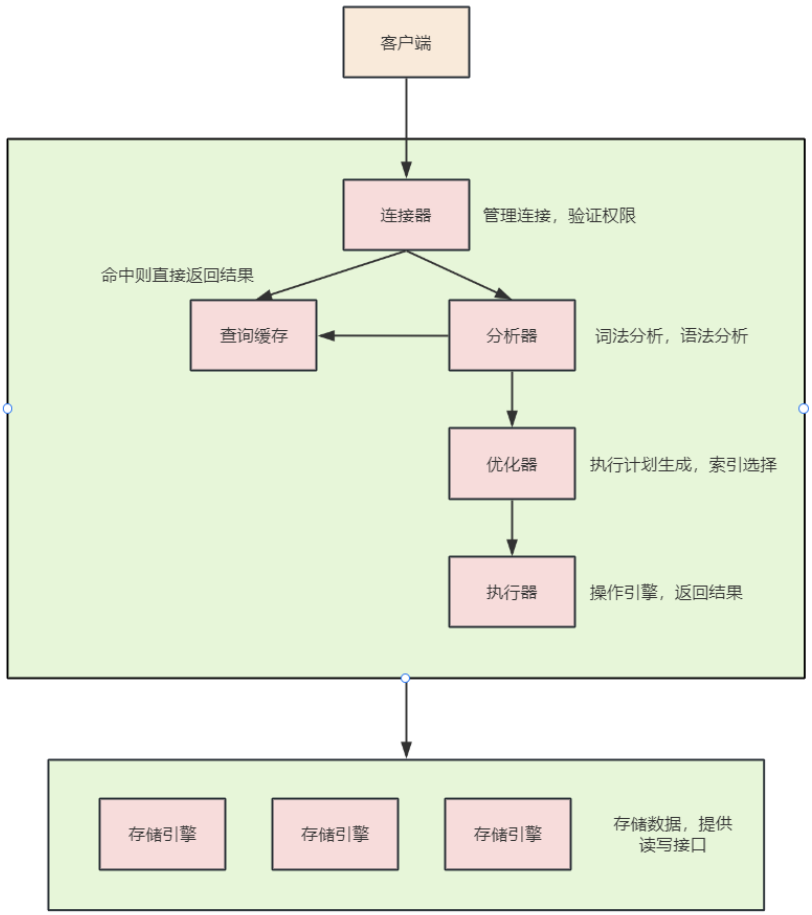
2. 一条SQL查询语句是如何执行的?

我们经常说，看一个事儿千万不要直接陷入细节里，你应该先鸟瞰其全貌，这样能够帮助你从高维度理解问题。同样，对于 MySQL 的学习也是这样。平时我们使用数据库，看到的通常都是一个整体。比如，你有个最简单的表，表里有一个 ID 字段，在执行下面这个查询语句时：我们看到的只是输入一条语句，返回一个结果，却不知道这条语句在 MySQL 内部的执行过程。

```
mysql> select * from user where id=10;
```

Select执行流程

下面是 MySQL 的基本架构示意图，从中我们可以清楚地看到 SQL 语句在 MySQL的各个功能模块中的执行过程。



客户端

连接工具（Navacat、SQLyog、JDBC）都归纳为MySQL客户端(Client)，主要用于发送执行sql语句的请求。

服务端

大体来说，MySQL 服务端可以分为 Server 层和存储引擎层两部分。Server 层包括连接器、查询缓存、分析器、优化器、执行器等，涵盖 MySQL 的大多数核心服务功能，以及所有的内置函数（如日期、时间、数学和加密函数等），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、视图等。

而存储引擎层负责数据的存储和检索。其架构模式是插件式的，支持 InnoDB、MyISAM、Memory 等多个存储引擎。现在最常用的存储引擎是 InnoDB，它从 MySQL5.5.5 版本开始成为了默认存储引擎。

也就是说，你执行 create table 建表的时候，如果不指定引擎类型，默认使用的就是InnoDB。不过，你也可以通过指定存储引擎的类型来选择别的引擎，比如在 create table语句中使用 engine=memory, 来指定使用内存引擎创建表。

Server 层

- 负责处理 SQL 语句、解析、优化、缓存等。
- 负责权限管理、用户认证等。
- 提供了各种 SQL 函数和存储过程。
- 提供了复制、备份、恢复等高级功能。
- Server 层有自己的日志系统，称为 **binlog（归档日志）**。binlog 记录了所有修改数据库数据的 SQL 语句（如 INSERT、UPDATE、DELETE 等）的信息，但不包括 SELECT 和 SHOW 这类查询语句。binlog 主要用于复制和恢复操作。

存储引擎层

- 负责数据的存储和检索。
- MySQL 支持多种存储引擎，如 InnoDB、MyISAM、Memory 等，每种引擎都有其特点和适用场景。
- InnoDB 是 MySQL 的默认存储引擎，它支持事务、行级锁定和外键约束。InnoDB 有自己的日志系统，称为 **redo log（重做日志）** 和 **undo log（撤销日志）**。redo log 用于保证事务的持久性，在数据库崩溃后可以用来恢复数据；undo log 用于支持事务的原子性和多版本并发控制（MVCC）。

连接器

第一步，你会先连接到这个数据库上，这时候接待你的就是连接器。连接器负责跟客户端建立连接、获取权限、维持和管理连接。连接命令一般是这么写的：

```
mysql -h$ip -P$port -u$user -p
```

输完命令之后，你就需要在交互对话里面输入密码。

连接命令中的 mysql 是客户端工具，用来跟服务端建立连接。在完成经典的 TCP 握手后，连接器就要开始认证你的身份，这个时候用的就是你输入的用户名和密码。

- 如果用户名或密码不对，你就会收到一个"Access denied for user"的错误，然后客户端程序结束执行。
- 如果用户名密码认证通过，连接器会到权限表里面查出你拥有的权限。之后，这个连接里面的权限判断逻辑，都将依赖于此时读到的权限。

```
root@a8e549f22a60:/# mysql -uroot -p12345
mysql: [Warning] Using a password on the command line interface can be insecure.
ERROR 1045 (28000): Access denied for user 'root'@'localhost' (using password: YES)
```

客户端如果太长时间没动静，连接器就会自动将它断开。这个时间是由参数 wait_timeout控制的，默认值是 8 小时。如果在连接被断开之后，客户端再次发送请求的话，就会收到一个错误提醒： Lost

connection to MySQL server during query。这时候如果你要继续，就需要重连，然后再执行请求了。

```
# 查看数据库的连接状态
show processlist;
#查看当前的wait_timeout参数值
SHOW VARIABLES LIKE 'wait_timeout';
```

注意：建立连接的过程通常是比较复杂的，建议在使用中要尽量减少建立连接的动作，尽量使用长连接。为了提升数据库并发性，可以建立一个数据库连接池。

> 长连接：连接成功后，如果客户端持续有请求，则一直使用同一个连接。
> 短连接：每次执行完很少的几次查询就断开连接，下次查询再重新建立一个。

连接器常见的问题：

全部使用长连接后，有时候 MySQL 占用内存涨得特别快，因为 MySQL 在执行过程中临时使用的内存是管理在连接对象里面的，这些资源会在连接断开的时候才释放，所以如果长连接累积下来，可能导致内存占用太大，被系统强行杀掉（OOM）。从现象看就是 MySQL 异常重启了

解决方案：

- 定期断开长连接。使用一段时间，或者程序里面判断执行过一个占用内存的大查询后，断开连接，之后要查询再重连。
- MySQL 5.7 以上版本，可以在每次执行一个比较大的操作后，通过执行 mysql_reset_connection 来重新初始化连接资源。这个过程不需要重连和重新做权限验证，但是会将连接恢复到刚刚创建完时的状态。

在Java 中与 MySQL 数据库交互通常使用 JDBC (Java Database Connectivity) API，它提供了自己的连接管理和错误处理机制。请注意，频繁地创建和关闭连接可能会对性能产生负面影响，特别是在高负载的情况下。因此，在生产环境中，通常会使用连接池来管理数据库连接，这样可以复用现有的连接而不是频繁地创建和销毁它们。

查询缓存

在MySQL5.7版本，连接完成后就会直接查询缓存，查询此语句是否执行过。执行逻辑就会来到第二步：查询缓存。

MySQL 拿到一个查询请求后，会先到查询缓存看看，之前是不是执行过这条语句。之前执行过的语句及其结果可能会以 key-value 对的形式，被直接缓存在内存中。key 是查询的语句，value 是查询的结果。如果你的查询能够直接在这个缓存中找到 key，那么这个value 就会被直接返回给客户端。

如果语句不在查询缓存中，就会继续后面的执行阶段。执行完成后，执行结果会被存入查询缓存中。

你可以看到，如果查询命中缓存，MySQL 不需要执行后面的复杂操作，就可以直接返回结果，这个效率会很高。

注意，MySQL 8.0 版本直接将查询缓存的整块功能删掉了。

> MySQL 为什么在 8.0 版本中移除了查询缓存功能

相关参数说明

- query_cache_size：用于缓存的大小
- query_cache_type：设置使用缓存的场景：
  - 0（OFF）：全不使用query cache
  - 1（ON）：除显式要求不使用 query cache 之外的所有的 select 都使用query cache，通过sql_no_cache 显示指定不使用缓存
  - 2（DEMOND）：只有显示要求才使用query cache，通过 sql_cache 显示指定使用缓存

查询缓存常见的问题：

但是大多数情况下不建议使用查询缓存，为什么呢？因为查询缓存往往弊大于利。对于更新压力大的数据库来说，查询缓存的命中率会非常低。除非你的业务就是有一张静态表，很长时间才会更新一次。比如，一个系统配置表，那这张表上的查询才适合使用查询缓存。

解决方案：

MySQL 提供了按需使用的方式。可以将参数 query_cache_type 设置成 DEMAND，对于默认的 SQL 语句都将不使用查询缓存。

你可以通过在 MySQL 命令行界面（CLI）中执行以下命令来设置 `query_cache_type` 为 `DEMAND`：

```
SET GLOBAL query_cache_type = DEMAND;
```

或者，你可以在 MySQL 的配置文件（通常是 `my.cnf` 或 `my.ini`）中设置：

```
[mysqld]
query_cache_type = DEMAND
```

修改配置文件后，你需要重启 MySQL 服务来使更改生效。

而对于你确定要使用查询缓存的语句，可以用 SQL_CACHE 显式指定，如下：

```
# 只有这条带有 SQL_CACHE 提示的查询会被缓存。
mysql> select SQL_CACHE * from user where id = 1;
```

分析器

若查询缓存未命中，则会执行分析器，来分析查询语句是否合法。

分析器先会做“词法分析”。MySQL 从你输入的"select"这个关键字识别出来，这是一个查询语句。它也要把字符串“user”识别成“表名 user”，把字符串“id”识别成“列 id”。

词法分析：

- 主要负责从 SQL 语句中提取关键字，比如：查询的表，字段名，查询条件等等。
- 词法分析阶段是从 information_schema 里面获得表的结构信息的。

做完了这些识别以后，就要做“语法分析”。根据词法分析的结果，语法分析器会根据语法规则，判断你输入的这个 SQL 语句是否满足 MySQL 语法。

语法分析：

- 判断输入的SQL 语句是否满足 MySQL 语法
- 如果 SQL 语句不对，就会返回 You have an error in your SQL syntax 的错误提醒，一般语法错误会提示第一个出现错误的位置，所以你要关注的是紧接“use near”的内容。

```
mysql> elect * from user where ID=1;
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version
for the right syntax to use near 'elect * from user where ID=1' at line 1
```

优化器

经过了分析器，若语句正确，就会进入优化器。优化器的作用是在基于同一个查询语句的多个查询方案中找出效率最高的。比如，在表里面有多个索引的时候，决定使用哪个索引；在一个语句有多表关联（join）的时候，决定各个表的连接顺序。

比如你执行下面这样的语句，这个语句是执行两个表的 join：

```
# USING等价于JOIN中on
mysql> select * from t1 join t2 using(ID) where t1.c=10 and t2.d=20;
```

理论上有两种执行方案：

方案1：可以先从表 t1 里面取出 c=10 的记录的 ID 值，再根据 ID 值关联到表 t2，再判断 t2里面 d 的值是否等于 20。

方案2：可以先从表 t2 里面取出 d=20 的记录的 ID 值，再根据 ID 值关联到 t1，再判断 t1里面 c 的值是否等于 10。

这两种执行方案的逻辑结果是一样的，但是执行的效率会有不同，而优化器的作用就是决定选择使用哪一个方案。例如，如果t1表非常大而t2表非常小，那么优化器可能会选择方案2，因为它可以先快速地从t2中筛选出d=20的记录，然后使用这些ID值去t1中进行关联，这样可以大大减少t1表的扫描量。

优化器阶段完成后，这个语句的执行方案就确定下来了，然后进入执行器阶段。

执行器

MySQL 通过分析器知道了你要做什么，通过优化器知道了该怎么做，于是就进入了执行器阶段，开始执行语句。

开始执行的时候，要先判断一下你对这个表有没有执行查询的权限，如果没有，就会返回没有权限的错误。如果有权限，就打开表继续执行。这是一种安全机制，确保只有被授权的用户才能访问和操作数据。

```
select * from T where ID=10;
ERROR 1142 (42000): SELECT command denied to user 'b'@'localhost' for table 'T'
```

注意：

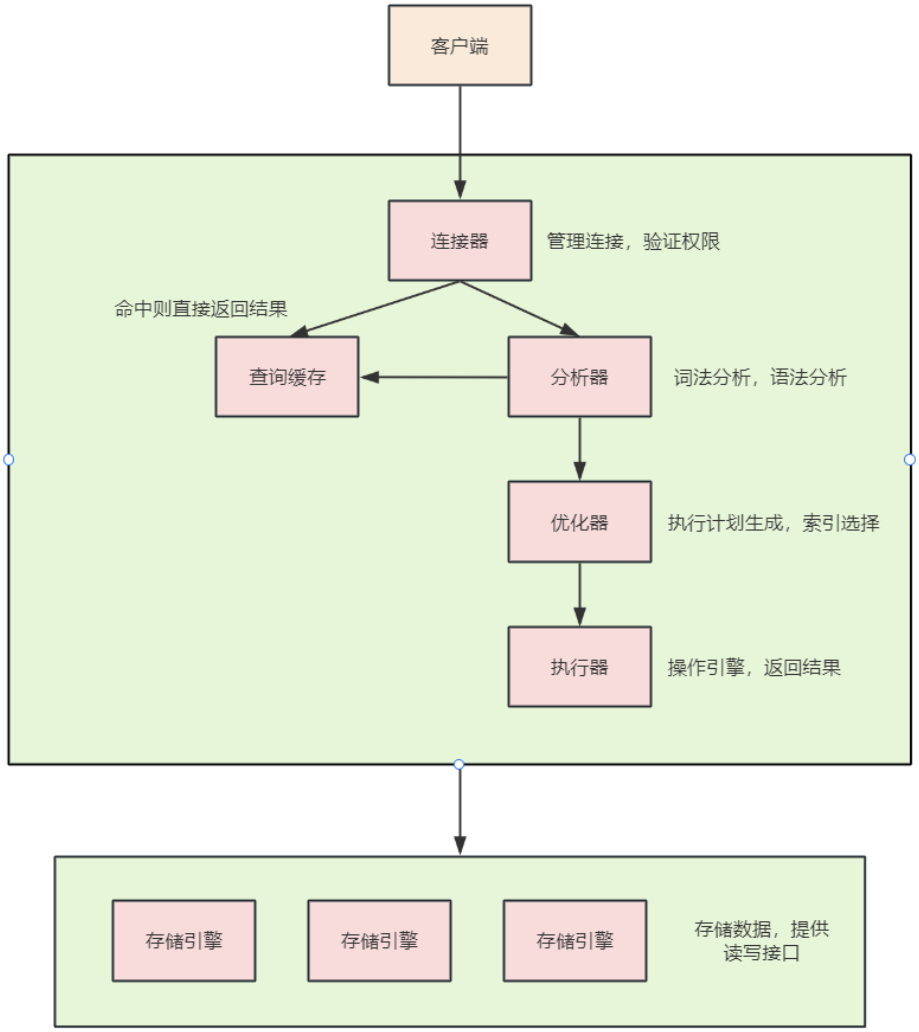
- 如果命中查询缓存，会在查询缓存返回结果的时候，做权限验证。
- 在语法分析过程中，解析器会进行一些初步的权限检查 precheck，例如验证用户是否有权访问指定的数据库和表。
- 有些时候，SQL语句要操作的表不只是SQL字面上那些。SQL执行过程中可能会有触发器这种在运行时才能确定的过程，precheck是不能对这种运行时涉及到的表进行权限校验的，所以需要在执行器阶段进行权限检查。

打开表的时候，执行器就会根据表的引擎定义，去使用这个引擎提供的接口：

1. 调用引擎接口取这个表的第一行，判断是否满足条件，如果不是则跳过，如果是则将这行存在结果集中
2. 调用引擎接口取下一行，重复相同的判断逻辑，直到取到这个表的最后一行
3. 执行器将上述遍历过程中所有满足条件的行组成的记录集作为结果集返回给客户端

至此，这个语句就执行完成了。

小结



思考题：

如果表 T 中没有字段 k，而你执行了这个语句 select * from T where k=1, 那肯定是会报“不存在这个列”的错误： “Unknown column ‘k’ in‘where clause’”。请问是在哪个阶段报出的错误?

分析器

3. 一条SQL更新语句是如何执行的?

前面我们系统了解了一个查询语句的执行流程，并介绍了执行过程中涉及的处理模块。

那么，一条更新语句的执行流程又是怎样的呢?

我们还是从一个表的一条更新语句说起，下面是这个表的创建语句，这个表有一个主键 ID和一个整型字段 c：

```
mysql> create table T(ID int primary key, c int);
```

如果要将 ID=2 这一行的值加 1，SQL 语句就会这么写：

```
mysql> update T set c=c+1 where ID=2;
```

查询语句的那一套流程，更新语句也是同样会走一遍。

- 执行语句前要先连接数据库，这是连接器的工作。
- 前面我们说过，在一个表上有更新的时候，跟这个表有关的查询缓存会失效，所以这条语句就会把表 T 上所有缓存结果都清空。这也就是我们一般不建议使用查询缓存的原因。
- 接下来，分析器会通过词法和语法解析知道这是一条更新语句。
- 优化器决定要使用 ID 这个索引。
- 然后，执行器负责具体执行，找到这一行，然后更新。

与查询流程不一样的是，更新流程还涉及两个重要的日志模块，它们正是我们要讨论的主角：redo log（重做日志）和 binlog（归档日志）。

redo log

不知道你还记不记得《孔乙己》这篇文章，酒店掌柜有一个粉板（白漆的木板），专门用来记录客人的赊账记录。如果赊账的人不多，那么他可以把顾客名和账目写在板上。但如果赊账的人多了，粉板总会有记不下的时候，这个时候掌柜一定还有一个专门记录赊账的账本。

如果有人要赊账或者还账的话，掌柜一般有两种做法：

- 一种做法是直接把账本翻出来，把这次赊的账加上去或者扣除掉；
- 另一种做法是先在粉板上记下这次的账，等打烊以后再把账本翻出来核算。

在生意红火柜台很忙时，掌柜一定会选择后者，因为前者操作实在是太麻烦了。首先，你得找到这个人的赊账总额那条记录。你想想，密密麻麻几十页，掌柜要找到那个名字，可能还得带上老花镜慢慢找，找到之后再拿出算盘计算，最后再将结果写回到账本上。这整个过程想想都麻烦。相比之下，还是先在粉板上记一下方便。你想想，如果掌柜没有粉板的帮助，每次记账都得翻账本，效率是不是低得让人难以忍受？

同样，在 MySQL 里也有这个问题，如果每一次的更新操作都需要写进磁盘，然后磁盘也要找到对应的那条记录，然后再更新，整个过程 IO 成本、查找成本都很高。为了解决这个问题，MySQL 的设计者就用了类似酒店掌柜粉板的思路来提升更新效率。

而粉板和账本配合的整个过程，其实就是 MySQL 里经常说到的 **WAL 技术（预写式日志）**，WAL 的全称是 Write-Ahead Logging，它的关键点就是先写日志，再写磁盘，也就是先写粉板，等不忙的时候再写账本。

具体来说，当有一条记录需要更新的时候，InnoDB 引擎就会先把记录写到 redo log（粉板）里面，并更新内存，这个时候更新就算完成了。同时，InnoDB 引擎会在适当的时候，将这个操作记录更新到磁盘里面，而这个更新往往是在系统比较空闲的时候做，这就像打烊以后掌柜做的事。如果今天赊账的不多，掌柜可以等打烊后再整理。

但如果某天赊账的特别多，粉板写满了，又怎么办呢？这个时候掌柜只好放下手中的活儿，把粉板中的一部分赊账记录更新到账本中，然后把这些记录从粉板上擦掉，为记新账腾出空间。

与此类似，InnoDB 的 redo log 是固定大小的，比如可以配置为一组 4 个文件，每个文件的大小是 1GB，那么这块“粉板”总共就可以记录 4GB 的操作。从头开始写，写到末尾就又回到开头循环写，如下面这个图所示。

write pos 是当前记录的位置，一边写一边后移，写到第 3 号文件末尾后就回到 0 号文件开头。

checkpoint 是当前要擦除的位置，也是往后推移并且循环的，擦除记录前要把记录更新到数据文件。

write pos 和 checkpoint 之间的是“粉板”上还空着的部分，可以用来记录新的操作。如果 write pos 追上 checkpoint，表示“粉板”满了，这时候不能再执行新的更新，得停下来先擦掉一些记录，把 checkpoint 推进一下。

有了 redo log，InnoDB 就可以保证即使数据库发生异常重启，之前提交的记录都不会丢失，这个能力称为crash-safe。

binlog

MySQL 整体来看，其实就有两块：一块是 Server 层，它主要做的是MySQL 功能层面的事情；还有一块是引擎层，负责存储相关的具体事宜。上面我们聊到的粉板 redo log 是 InnoDB 引擎特有的日志，而 Server 层也有自己的日志，称为binlog（归档日志）。

我想你肯定会问，为什么会有两份日志呢?

因为最开始 MySQL 里并没有 InnoDB 引擎。MySQL 自带的引擎是 MyISAM，但是MyISAM 没有 crash-safe 的能力，binlog 日志只能用于归档。而 InnoDB 是另一个公司以插件形式引入 MySQL 的，既然只依靠 binlog 是没有 crash-safe 能力的，所以InnoDB 使用另外一套日志系统——也就是 redo log 来实现 crash-safe 能力。

这两种日志有以下三点不同。

1. redo log 是 InnoDB 引擎特有的；binlog 是 MySQL 的 Server 层实现的，所有引擎都可以使用。

2. redo log 是物理日志，记录的是“在某个数据页上做了什么修改”；binlog 是逻辑日志，记录的是这个语句的原始逻辑，比如“给 ID=2 这一行的 c 字段加 1 ”。redo log 是从数据页的角度来看的，它关心的是数据在磁盘上的物理布局和如何高效地修改这些数据。binlog 是从 SQL 语句的角度来看的，它关心的是执行了哪些操作以及这些操作的内容。
3. redo log 是循环写的，空间固定会用完；binlog 是可以追加写入的。“追加写”是指binlog 文件写到一定大小后会切换到下一个，并不会覆盖以前的日志。

两者结合起来，使得 MySQL 既能保证事务的 ACID 属性，又能支持高效的数据复制和恢复。当执行一个事务时，相关的更改会先写入 redo log，然后执行 SQL 语句并将更改写入 binlog。这样，即使在数据库系统崩溃的情况下，也可以通过这两个日志来恢复数据。

> - innodb_flush_log_at_trx_commit 这个参数设置成1 的时候，表示每次事务的 redo log 都直接持久化到磁盘，这样可以保证 MySQL 异常重启之后数据不丢失。
> - sync_binlog 这个参数设置成 1 的时候，表示每次事务的 binlog 都持久化到磁盘。这个参数我也建议你设置成 1，这样可以保证 MySQL 异常重启之后 binlog 不丢失。

```
mysql> show variables like 'innodb_flush_log_at_trx_commit';
+--------------------------------+-------+
| Variable_name                  | Value |
+--------------------------------+-------+
| innodb_flush_log_at_trx_commit | 1     |
+--------------------------------+-------+
```

思考：mysql如何恢复误删数据，比如某天下午发现中午十二点有一次误删表，需要找回数据，应该怎么做?

当需要恢复到指定的某一秒时，你可以这么做：

- 首先，找到最近的一次全量备份（要做定期备份，比如一天一备），如果你运气好，可能就是昨天晚上的一个备份，从这个备份恢复到临时库；
- 然后，从备份的时间点开始，将备份的 binlog 依次取出来，重放到中午误删表之前的那个时刻。

这样你的临时库就跟误删之前的线上库一样了，然后你可以把表数据从临时库取出来，按需要恢复到线上库去。

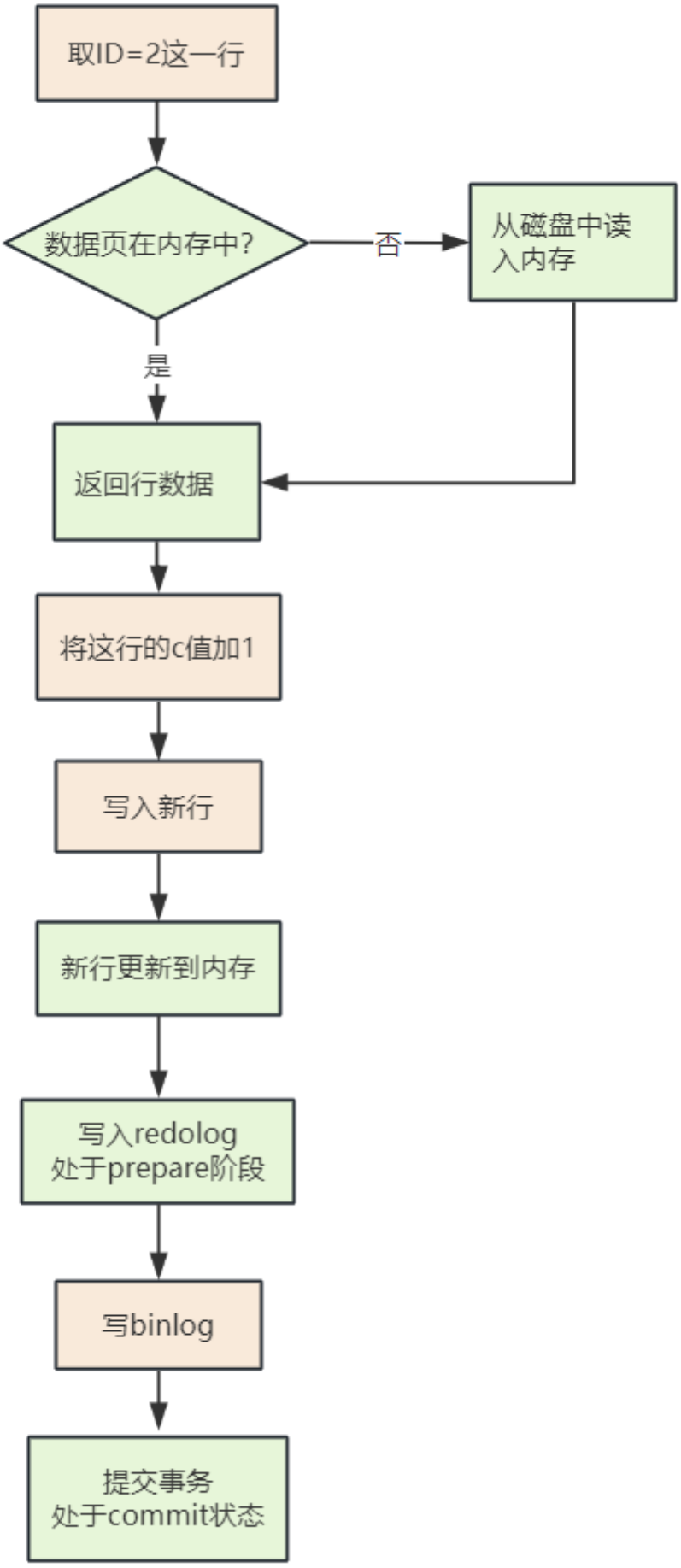
Update执行流程

有了对这两个日志的概念性理解，我们再来看执行器和 InnoDB 引擎在执行这个简单的update 语句时的内部流程。

1. 执行器先找引擎取 ID=2 这一行。ID 是主键，引擎直接用树搜索找到这一行。如果ID=2 这一行所在的数据页本来就在内存中，就直接返回给执行器；否则，需要先从磁盘读入内存，然后再返回。
2. 执行器拿到引擎给的行数据，把这个值加上 1，比如原来是 N，现在就是 N+1，得到新的一行数据，再调用引擎接口写入这行新数据。

3. 引擎将这行新数据更新到内存中，同时将这个更新操作记录到 redo log 里面，此时redo log 处于 prepare 状态。然后告知执行器执行完成了，随时可以提交事务。
4. 执行器生成这个操作的 binlog，并把 binlog 写入磁盘。
5. 执行器调用引擎的提交事务接口，引擎把刚刚写入的 redo log 改成提交（commit）状态，更新完成。

下图是update 语句 update T set c=c+1 where ID=2的执行流程图



思考题：上图哪些是在执行器中执行的，哪些是在 InnoDB 引擎中执行的?

Mysql内部两阶段提交（内部XA）

这是为了让两份日志之间的逻辑一致，mysql会采用两阶段提交。

由于 redo log 和 binlog 是两个独立的逻辑，如果不用两阶段提交，要么就是先写完redo log 再写binlog，或者采用反过来的顺序。我们看看这两种方式会有什么问题。

仍然用前面的 update 语句update T set c=c+1 where ID=2来做例子。假设当前 ID=2 的行，字段 c 的值是 0，再假设执行 update 语句过程中在写完第一个日志后，第二个日志还没有写完期间发生了crash，会出现什么情况呢?

场景1：`

假设在 redo log 写完，binlog 还没有写完的时候，MySQL 进程异常重启。由于我们前面说过的，redo log 写完之后，系统即使崩溃，仍然能够把数据恢复回来，所以恢复后这一行 c 的值是 1。 但是由于 binlog 没写完就 crash 了，这时候 binlog 里面就没有记录这个语句。因此，之后备份日志的时候，存起来的 binlog 里面就没有这条语句。 然后你会发现，如果需要用这个 binlog 来恢复临时库的话，由于这个语句的 binlog 丢失，这个临时库就会少了这一次更新，恢复出来的这一行 c 的值就是 0，与原库的值不同。

场景2： 先写 binlog 后写 redo log

如果在 binlog 写完之后 crash，由于 redo log 还没写，崩溃恢复以后这个事务无效，所以这一行 c 的值是 0。但是 binlog 里面已经记录了“把 c 从 0 改成 1”这个日志。所以，在之后用 binlog 来恢复的时候就多了一个事务出来，恢复出来的这一行 c 的值就是 1，与原库的值不同。

可以看到，如果不使用“两阶段提交”，那么数据库的状态就有可能和用它的日志恢复出来的库的状态不一致。

思考题： binlog 写完，redo log 还没 commit前发生 crash，那崩溃恢复的时候 MySQL 会怎么处理?

我们先来看一下崩溃恢复时的判断规则。

1. 如果 redo log 里面的事务是完整的，也就是已经有了 commit 标识，则直接提交；
2. 如果 redo log 里面的事务只有完整的 prepare，则判断对应的事务 binlog 是否存在并完整：
   a. 如果是，则提交事务；
   b. 否则，回滚事务。

所以，binlog 写完，redo log 还没 commit前发生 crash， 对应的就是 2(a) 的情况，崩溃恢复过程中事务会被提交。

追问： redo log 和 binlog 是怎么关联起来的?

它们有一个共同的数据字段，叫 XID。崩溃恢复的时候，会按顺序扫描 redo log：

- 如果碰到既有 prepare、又有 commit 的 redo log，就直接提交；

- 如果碰到只有 parepare、而没有 commit 的 redo log，就拿着 XID 去 binlog 找对应的事务。

追问：redo log buffer 是什么？是先修改内存，还是先写 redo log文件?

在一个事务的更新过程中，日志是要写多次的。比如下面这个事务：

```
begin;
insert into t1 ...
insert into t2 ...
commit;
```

这个事务要往两个表中插入记录，插入数据的过程中，生成的日志都得先保存起来，但又不能在还没 commit 的时候就直接写到 redo log 文件里。

所以，redo log buffer 就是一块内存，用来先存 redo 日志的。也就是说，在执行第一个insert 的时候，数据的内存被修改了，redo log buffer 也写入了日志。

但是，真正把日志写到 redo log 文件（文件名是 ib_logfile+ 数字），是在执行 commit语句的时候做的。