MySQL架构篇

一、环境说明

1.MySQL服务器环境

1. Linux虚拟机：CentOS 7
2. MySQL：MySQL5.7.30

2、mysql文件结构

- MySQL是通过文件系统对数据和索引进行存储的。
- MySQL从物理结构上可以分为日志文件和数据索引文件。

MySQL在Linux中的数据索引文件和日志文件通常放在/var/lib/mysql目录下。

1.日志文件（顺序IO）

MySQL通过日志记录了数据库操作信息和错误信息。常用的日志文件包括**错误日志、二进制日志、查询日志、慢查询日志和事务Redo 日志、中继日志**等。

可以通过命令查看当前数据库中的日志使用信息：

```
mysql> show variables like 'log_%';
```

1）错误日志（errorlog）

默认是开启的，而且从5.5.7以后无法关闭错误日志，错误日志记录了运行过程中**遇到的所有严重的错误信息**,以及 MySQL**每次启动和关闭的详细信息。**

默认的错误日志名称：hostname.err。

错误日志所记录的信息是可以通过**log-error和log-warnings**来定义的，其中log-err是定义是否启用错误日志的功能和错误日志的存储位置，log-warnings是定义是否将警告信息也定义至错误日志中。

```
#可以直接定义为文件路径，也可以为ON|OFF
log_error=/var/log/mysqld.log
#只能使用1|0来定义开关启动，默认是启动的
log_warings=1
```

2）二进制日志（bin log）

默认是关闭的，需要通过以下配置进行开启。：

```
log-bin=mysql-bin
```

其中mysql-bin是binlog日志文件的basename，binlog日志文件的完整名称：mysql-bin-000001.log

binlog记录了**数据库所有的ddl语句和dml语句，但不包括select语句内容**，语句以事件的形式保存，描述了数据的变更顺序，binlog还包括了每个更新语句的执行时间信息。如果是DDL语句，则直接记录到binlog日志，而**DML语句，必须通过事务提交才能记录到binlog日志中。**

binlog主要用于**实现mysql主从复制、数据备份、数据恢复。**

3）通用查询日志（general query log）

默认情况下通用查询日志是关闭的。

由于通用查询日志会记录用户的所有操作，其中还包含增删查改等信息，在并发操作大的环境下会产生大量的信息从而导致不必要的磁盘IO，会影响mysql的性能的。如若不是为了调试数据库的目的**建议不要开启查询日志。**

```
mysql> show global variables like 'general_log';
```

开启方式：

```
#启动开关
general_log={ON|OFF}
#日志文件变量，而general_log_file如果没有指定，默认名是host_name.log
general_log_file=/PATH/TO/file
#记录类型
log_output={TABLE|FILE|NONE}
```

4）慢查询日志（slow query log）

默认是关闭的。

需要通过以下设置进行开启：

```
#开启慢查询日志
slow_query_log=ON
#慢查询的阈值
long_query_time=10
#日志记录文件如果没有给出file_name值， 默认为主机名，后缀为-slow.log。如果给出了文件名，但不是绝对路径名，文件则写入数据目录。
slow_query_log_file= file_name
```

记录执行时间超过**long_query_time**秒的所有查询，便于收集查询时间比较长的SQL语句

查询多少SQL超过了慢查询时间的阈值： `SHOW GLOBAL STATUS LIKE '%Slow_queries%';`

2.数据文件（随机IO）

- 查看MySQL数据文件：

```
SHOW VARIABLES LIKE '%datadir%';
```

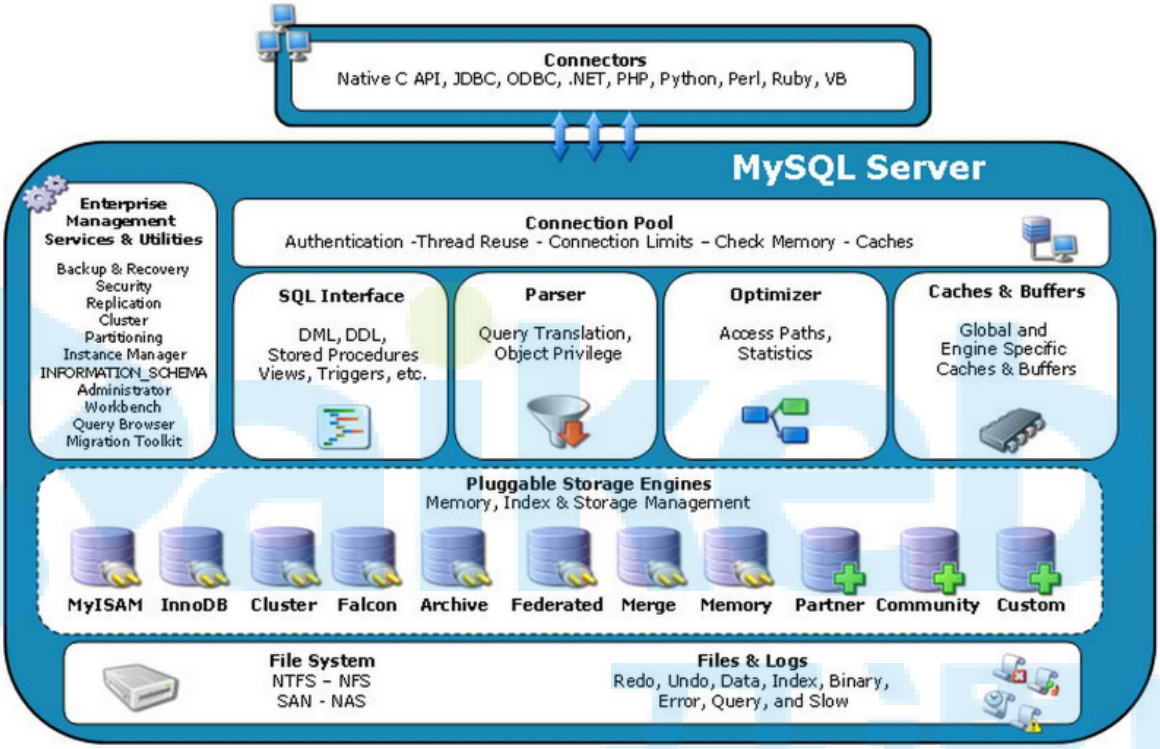
1）InnoDB数据文件

- **.frm文件**：主要存放与表相关的数据信息,主要包括**表结构的定义信息**
- **.ibd**：使用**独享表空间**存储**表数据和索引**信息，一张表对应一个ibd文件。
- **ibdata文件**：使用**共享表空间**存储**表数据和索引**信息，所有表共同使用一个或者多个ibdata文件。

2）MyIsam数据文件

- **.frm文件**：主要存放与表相关的数据信息,主要包括**表结构的定义信息**
- **.myd文件**：主要用来存储**表数据信息。**
- **.myi文件**：主要用来存储**表数据文件中任何索引的数据树。**

二、逻辑架构图



1.Connectors

连接器，指的是不同语言中与SQL的交互

2.Management Serveices & Utilities

系统管理和控制工具

3.Connection Pool: 连接池

- 管理用户连接，等待处理连接请求。
- 负责监听对 MySQL Server 的各种请求，接收连接请求，转发所有连接请求到线程管理模块。每一个连接上 MySQL Server 的客户端请求都会被分配（或创建）一个连接线程为其单独服务。
- 而连接线程的主要工作就是负责 MySQL Server 与客户端的通信，接受客户端的命令请求，传递 Server 端的结果信息等。线程管理模块则负责管理维护这些连接线程。包括线程的创建，线程的 cache 等。

4.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

5.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。**

主要功能：

a . 将SQL语句进行词法分析和语法分析，解析成**语法树**，然后按照不同的操作类型进行分类，然后做出针对性的转发到后续步骤，以后SQL语句的传递和处理就是基于这个结构的。

b. 如果在分解过程中遇到错误，那么就说明这个sql语句是不合理的。

6.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。**explain语句查看的SQL语句执行计划，就是由查询优化器生成的。

7.Cache和Buffer： 查询缓存

他的主要功能是将客户端提交给MySQL的 select请求的返回结果集 cache 到内存中，与该 query 的一个 hash 值 做一个对应。该 Query 所取数据的基表发生任何数据的变化之后， MySQL 会自动使该 query 的Cache 失效。在读写比例非常高的应用系统中， Query Cache 对性能的提高是非常显著的。当然它对内存的消耗也是非常大的。

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

8.Pluggable Storage Engines：存储引擎

与其他数据库例如Oracle 和SQL Server等数据库中只有一种存储引擎不同的是，MySQL有一个被称为“Pluggable Storage Engine Architecture”(可插拔的存储引擎架构)的特性，也就意味着MySQL数据库提供了多种存储引擎。

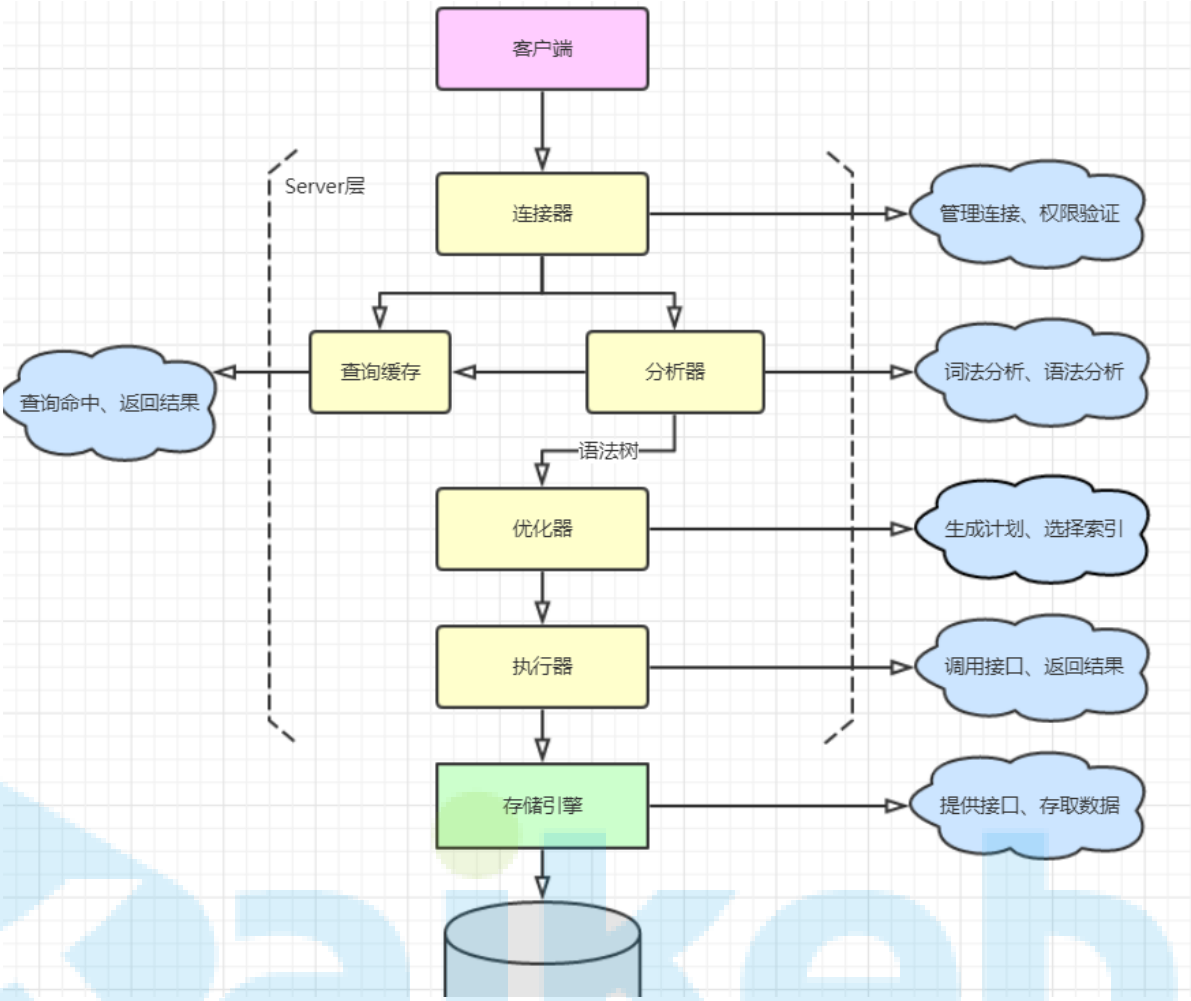
而且存储引擎是针对表的，用户可以根据不同的需求为数据表选择不同的存储引擎，用户也可以根据自己的需要编写自己的存储引擎。也就是说，同一数据库不同的表可以选择不同的存储引擎

```
creat table xxx()engine=InnoDB/Memory/MyISAM
```

简而言之，存储引擎就是如何存储数据、如何为存储的数据建立索引和如何更新、查询数据等技术的实现方法。

三、MySqlServer层对象

1.Sql语句执行流程



待分析SQL语句如下：

```
mysql> select customer_id,first_name,last_name from customer where
customer_id=14;
```

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。

所以今天我想和你一起把 MySQL 拆解一下，看看里面都有哪些“零件”，希望借由这个拆解过程，让你对 MySQL 有更深入的理解。这样当我们碰到 MySQL 的一些异常或者问题时，就能够直戳本质，更为快速地定位并解决问题。

上面给出的是 MySQL 的基本架构示意图，从中你可以清楚地看到 SQL 语句在 MySQL 的各个功能模块中的执行过程。

大体来说，MySQL 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**负责数据的存储和提取。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 MySQL 5.5.5 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。不同存储引擎的表数据存取方式不同，支持的功能也不同，在后面的文章中，我们会讨论到引擎的选择。

从图中不难看出，不同的存储引擎共用一个**Server 层**，也就是从连接器到执行器的部分。你可以先对每个组件的名字有个印象，接下来我会结合开头提到的那条 SQL 语句，带你走一遍整个执行流程，依次看下每个组件的作用。

2.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。虽然密码也可以直接跟在 -p 后面写在命令行中，但这样可能会导致你的密码泄露。如果你连的是生产服务器，强烈建议你不要这么做。

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。

这就意味着，一个用户成功建立连接后，即使你用管理员账号对这个用户的权限做了修改，也不会影响已经存在连接的权限。修改完成后，只有再新建的连接才会使用新的权限设置。

连接完成后，如果你没有后续的动作，这个连接就处于空闲状态，你可以在 show processlist 命令中看到它。文本中这个图是 show processlist 的结果，其中的 Command 列显示为“Sleep”的这一行，就表示现在系统里面有一个空闲连接。

```
mysql> show processlist;
+----+------+-----------------+------+---------+------+----------+------------------+
| Id | User | Host            | db   | Command | Time | State    | Info             |
+----+------+-----------------+------+---------+------+----------+------------------+
|  5 | root | localhost:27710 | test | Sleep   |   16 |          | NULL             |
|  6 | root | localhost:27712 | test | Query   |    0 | starting | show processlist |
+----+------+-----------------+------+---------+------+----------+------------------+
2 rows in set (0.00 sec)
```

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。

数据库里面，长连接是指连接成功后，如果客户端持续有请求，则一直使用同一个连接。短连接则是指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。

建立连接的过程通常是比较复杂的，所以我建议你在使用中要尽量减少建立连接的动作，也就是尽量使用长连接。

但是全部使用长连接后，你可能会发现，有些时候 MySQL 占用内存涨得特别快，这是因为 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。

怎么解决这个问题呢？你可以考虑以下两种方案。

1. 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。
2. 如果你用的是 MySQL 5.7 或更新版本，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。

3.查询缓存

连接建立完成后，你就可以执行 select 语句了。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hash之后的值，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。

但是大多数情况下我会建议你不要使用查询缓存，为什么呢？因为查询缓存往往弊大于利。

查询缓存的失效非常频繁，只要有对一个表的更新，这个表上所有的查询缓存都会被清空。因此很可能你费劲地把结果存起来，还没使用呢，就被一个更新全清空了。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。

好在 MySQL 也提供了这种“按需使用”的方式。你可以将参数 query_cache_type 设置成 DEMAND，这样对于默认的 SQL 语句都不使用查询缓存。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，像下面这个语句一样：

```
mysql> select sql_cache * from city where city_id = 1;
```

查询缓存命中次数

```
SHOW STATUS LIKE 'Qcache_hits'
```

果然没有命中，是因为`query_cache_type`默认是关闭的：

```
mysql> show variables like 'query_cache_type';
+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| query_cache_type | OFF   |
+------------------+-------+

1 row in set (0.00 sec)
```

值为`0`或`OFF`会禁止使用缓存。
值为`1`或`ON`将启用缓存，但以`SELECT SQL_NO_CACHE`开头的语句除外。
值为`2`或`DEMAND`时，只缓存以`SELECT SQL_CACHE`开头的语句。

需要修改配`my.cnf`置文件，在文件中增加如下内容开启缓存：

```
query_cache_type=2
```

清空查询缓存

可以使用下面三个SQL来清理查询缓存：

1、FLUSH QUERY CACHE; // 清理查询缓存内存碎片。

2、RESET QUERY CACHE; // 从查询缓存中移出所有查询。

3、FLUSH TABLES; //关闭所有打开的表，同时该操作将会清空查询缓存中的内容。

需要注意的是，MySQL 8.0 版本直接将查询缓存的整块功能删掉了，也就是说 8.0 开始彻底没有这个功能了。

4.分析器

如果没有命中查询缓存，就要开始真正执行语句了。首先，MySQL 需要知道你要做什么，SQL语句是如何被识别的呢？因此需要对接收到 SQL 语句做解析。

这个阶段就是 MySQL 的 Parser 解析器和 Preprocessor预处理模块的功能。

如果查询缓存没有命中，接下来就需要进入正式的查询阶段了。客户端程序发送过来的请求事实上只是一段文本而已，所以MySQL服务器程序首先需要对这段文本做分析，判断请求的语法是否正确，然后从文本中将要查询的表、列和各种查询条件都提取出来，本质上是对一个SQL语句编译的过程，涉及词法解析、语法分析、语义分析等阶段。

解析器先会做“词法分析”。

词法分析就是把一个完整的 SQL 语句分割成一个个的字符串，比如这条简单的SQL语句

```
select customer_id,first_name,last_name from customer where customer_id=14;
```

会被分割成10个字符串

```
select，customer_id,first_name,last_name，from，customer，where，customer_id，
=，14
```

MySQL 同时需要识别出这个SQL语句中的字符串分别是什么，代表什么。MySQL 从"select"这个关键字识别出来，这是一个查询语句，把字符串“customer”识别成“表名 customer”，把字符串“customer_id 识别成“列 customer_id”。

做完了这些识别以后，解析器就要做“语法分析”。

分析器的第二步是根据词法分析的结果，语法分析器会根据语法规则做语法检查，判断你输入的这个 SQL 语句是否满足 MySQL 语法。

如果你的语句不对，就会收到“You have an error in your SQL syntax”的错误提醒，比如下面这个语句 select 少打了开头的字母“s”。

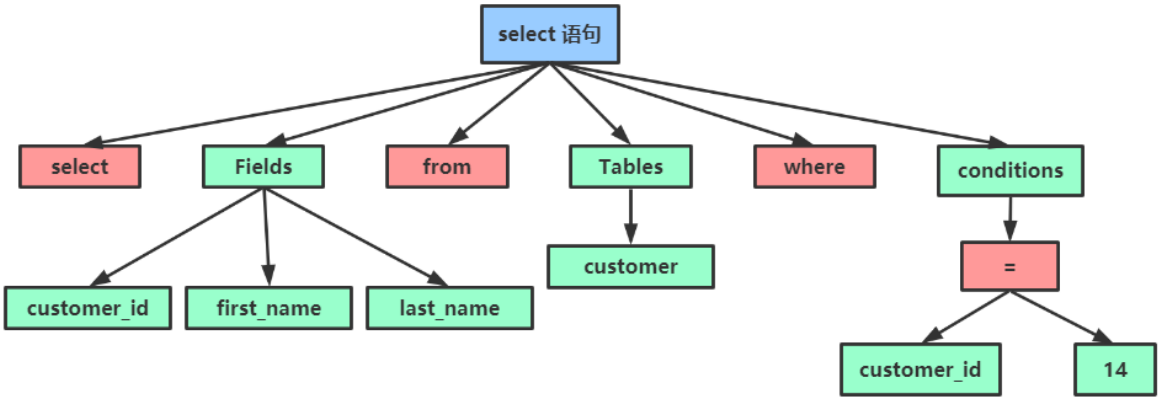
```
mysql> elect customer_id,first_name,last_name from customer where
customer_id=14;

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 customer where customer_id=14' at line 1
```

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。

如果语法正确，就会根据 MySQL 定义的语法规则，根据 SQL 语句生成一个数据结构，这个数据结构我们把它叫做解析树。

select customer_id,first_name,last_name from customer where customer_id=14;



预处理器

预处理器则会进一步去检查解析树是否合法，比如表名是否存在，语句中表的列是否存在等等，在这一步MySQL会检验用户是否有表的操作权限。预处理之后会得到一个新的解析树。

5.优化器

经过了解析器和预处理器，得到了解析树后，MySQL已经知道你要做什么了，那是不是这时就可以执行语句了呢?

这里我们有一个问题，一条 查询 语句是不是就只有一种执行方式？数据库最终执行的 SQL 是不是就是我们发送的 SQL？答案是否定的。一条 SQL 语句是可以有多种执行方式的，它们最终返回结果是相同的。但是在这么多种执行方式，我们最终选择哪一种去执行？选择的判断标准是什么呢？ 这个是就是优化器的作用。

在开始执行SQL之前，还要先经过优化器的处理。

查询优化器的作用就是根据解析树生成不同的执行计划，然后选择一种最优的执行计划，MySQL 里面使用的是基于成本模型的优化器，哪种执行计划执行时成本最小就用哪种。而且它是io_cost和cpu_cost的开销总和，它通常也是我们评价一个查询的执行效率的一个常用指标。

```
#查看上次查询成本开销
show status like 'Last_query_cost';
```

优化器都做哪些优化处理呢？比如

1. 当有多个索引可用的时候，决定使用哪个索引；

2. 在一个语句有多表关联（join）的时候，决定各个表的连接顺序，以哪个表为基准表。

（1）比如sakila数据库中表customer上执行下面的语句，这个语句用到了两个索引last_name和address_id。

```
#sakila数据库中表customer
select * from customer where last_name='WHITE' and address_id=18
```

- 既可以使用索引last_name查询，然后过滤列address_id
- 也可以使用索引address_id查询，然后过滤列last_name。

  两种执行计划的结果是一样的，但是执行效率会有所不同，**而优化器的作用就是决定选择使用哪一个方案。**

我们使用explain工具可以查看优化器的执行计划，可以看到优化器选用的是第二个执行计划。

```
mysql> explain select * from customer where last_name='WHITE' and address_id=18;
+----+-------------+----------+------------+------+-------------------------------+----------------+---------+-------+
| id | select_type | table    | partitions | type | possible_keys                 | key            | key_len | ref   |
+----+-------------+----------+------------+------+-------------------------------+----------------+---------+-------+
|  1 | SIMPLE      | customer | NULL       | ref  | idx_fk_address_id,idx_last_name | idx_fk_address_id | 2       | const |
+----+-------------+----------+------------+------+-------------------------------+----------------+---------+-------+
1 row in set, 1 warning (0.00 sec)
```

至于关于优化器使用成本模型选择索引，预估每个执行计划的成本，然后选择最优的执行计划。至于优化器是如何根据成本模型选择索引的，有没有可能存在选错的可能，我们会在后面的索引部分内容展开说明。

（2）而对于我们的查询语句，只用到了一个索引：主键索引，主键索引就是最优的执行计划。

```
1 explain select * from customer where customer_id=14;
```

```
mysql> explain select * from customer where customer_id=14;
+----+-------------+----------+------------+-------+---------------+---------+---------+-------+
| id | select_type | table    | partitions | type  | possible_keys | key     | key_len | ref   |
+----+-------------+----------+------------+-------+---------------+---------+---------+-------+
|  1 | SIMPLE      | customer | NULL       | const | PRIMARY       | PRIMARY | 2       | const |
+----+-------------+----------+------------+-------+---------------+---------+---------+-------+
1 row in set, 1 warning (0.00 sec)
```

优化器也不是万能的，我们不能完全依赖于MySQL的优化器，并不是再垃圾的 SQL 语句都能被优化，MySQL也不是每次都选择到最优的执行计划，大家在编写 SQL 语句的时候还是要注意，要对自己编写的SQL有一个预判，不能将所有的工作都交给MySQL。

6.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，得到了一个查询计划。于是就进入了执行器阶段，开始执行语句。

（1）开始执行的时候，要**先判断一下你对这个表customer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。（在工程实现上，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。

比如我们新建一个用户lesson1_test，只有表actor的查询权限，没有表customer的查询权限。

```
1 CREATE USER `lesson1_test`@`localhost` IDENTIFIED BY '123456';
2 GRANT Select ON TABLE `sakila`.`actor` TO `lesson1_test`@`localhost`;
```

使用这个用户lesson1_test连接mysql，执行下面的查询语句，就会返回没有权限的错误。

```
1 mysql> select * from customer where customer_id=14;
2
3 ERROR 1142 (42000): SELECT command denied to user 'lesson1_test'@'localhost'
  for table 'customer'
```

（2）**如果有权限，就使用指定的存储引擎打开表开始查询。执行器会根据表的引擎定义，去使用这个引擎提供的查询接口，提取数据。**

比如我们这个例子中的表 customer中，customer_id 字段是主键，那么执行器的执行流程是这样的：

1. 调用 InnoDB 引擎接口，从主键索引中检索customer_id=14的记录。
2. 主键索引等值查询只会查询出一条记录，直接将该记录返回客户端。

   至此，这个语句就执行完成了。

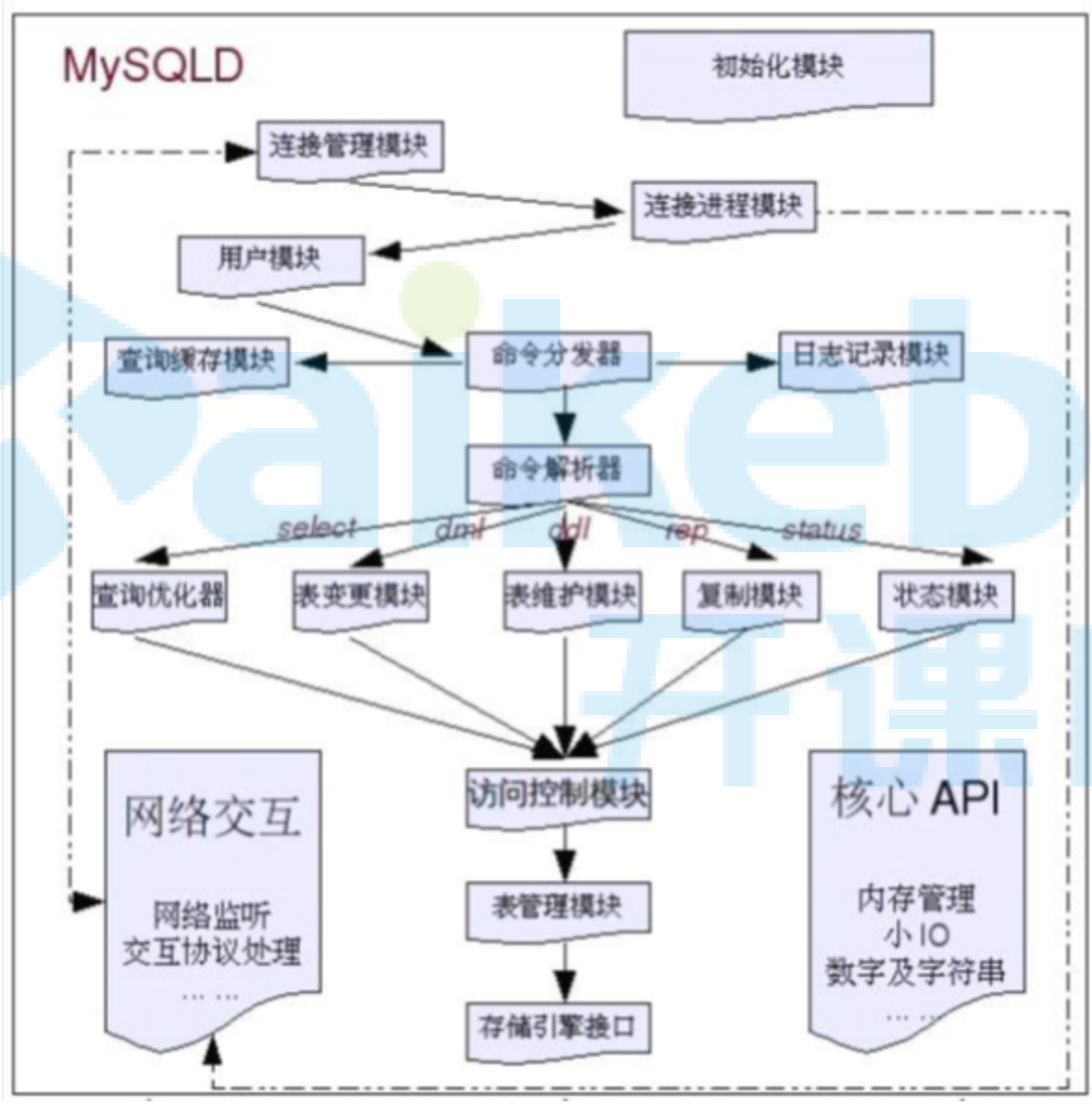
InnoDB主键索引和非主键索引的检索流程，我们在后面的索引部分内容会展开说明，这里就不做过多说明了。

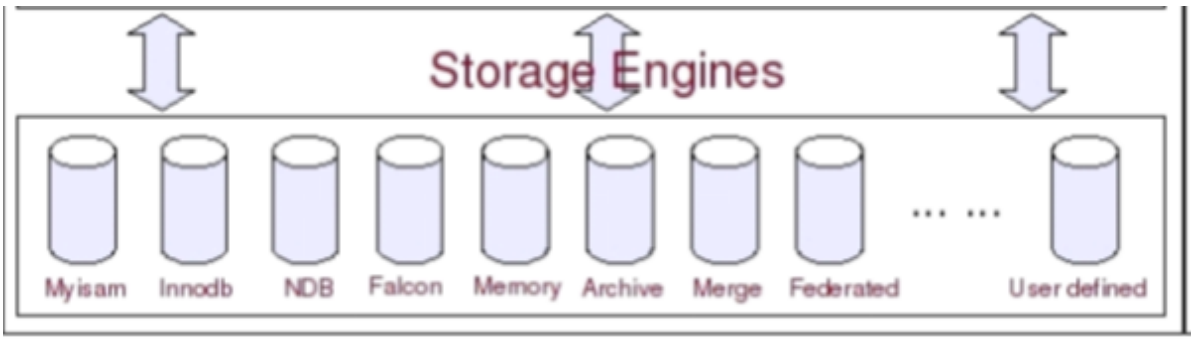
假设customer_id 字段不是索引，这时查询只能全表扫描。那么执行器的执行流程是这样的：

1. 调用 InnoDB 引擎接口取这个表的第一行，判断customer_id 值是不是 14，如果不是则跳过，如果是则将这行缓存在结果集中；
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结果集返回给客户端。

至此，这个语句就执行完成了。

7.详细流程图(参考)





四、InnoDB存储引擎

1、MySQL存储引擎种类

存储引擎	说明
MyISAM	高速引擎，拥有较高的插入，查询速度，**但不支持事务**
InnoDB	**5.5版本后MySQL的默认数据库，支持事务和行级锁定**，比MyISAM处理速度稍慢
ISAM	MyISAM的前身，MySQL5.0以后不再默认安装
MRG_MyISAM（MERGE）	将多个表联合成一个表使用，在超大规模数据存储时很有用
Memory	**内存存储引擎，拥有极高的插入，更新和查询效率。**但是会占用和数据量成正比的内存空间。只在内存上保存数据，意味着数据可能会丢失
Falcon	一种新的存储引擎，支持事物处理，传言可能是InnoDB的替代者
Archive	将数据压缩后进行存储，非常适合存储大量的独立的，作为历史记录的数据，但是只能进行插入和查询操作
CSV	CSV 存储引擎是基于 CSV 格式文件存储数据(应用于跨平台的数据交换)

- **查看存储引擎：**

```
mysql> show engines;
```

- **InnoDB和MyISAM存储引擎区别：**

	Innodb	Myisam
存储文件	.frm 表定义文件 .ibd 数据文件和索引文件	.frm 表定义文件 .myd 数据文件 .myi 索引文件
锁	表锁、行锁	表锁
事务	支持	不支持
CRDU	读、写	读多
count	扫表	专门存储的地方
索引结构	B+ Tree	B+ Tree

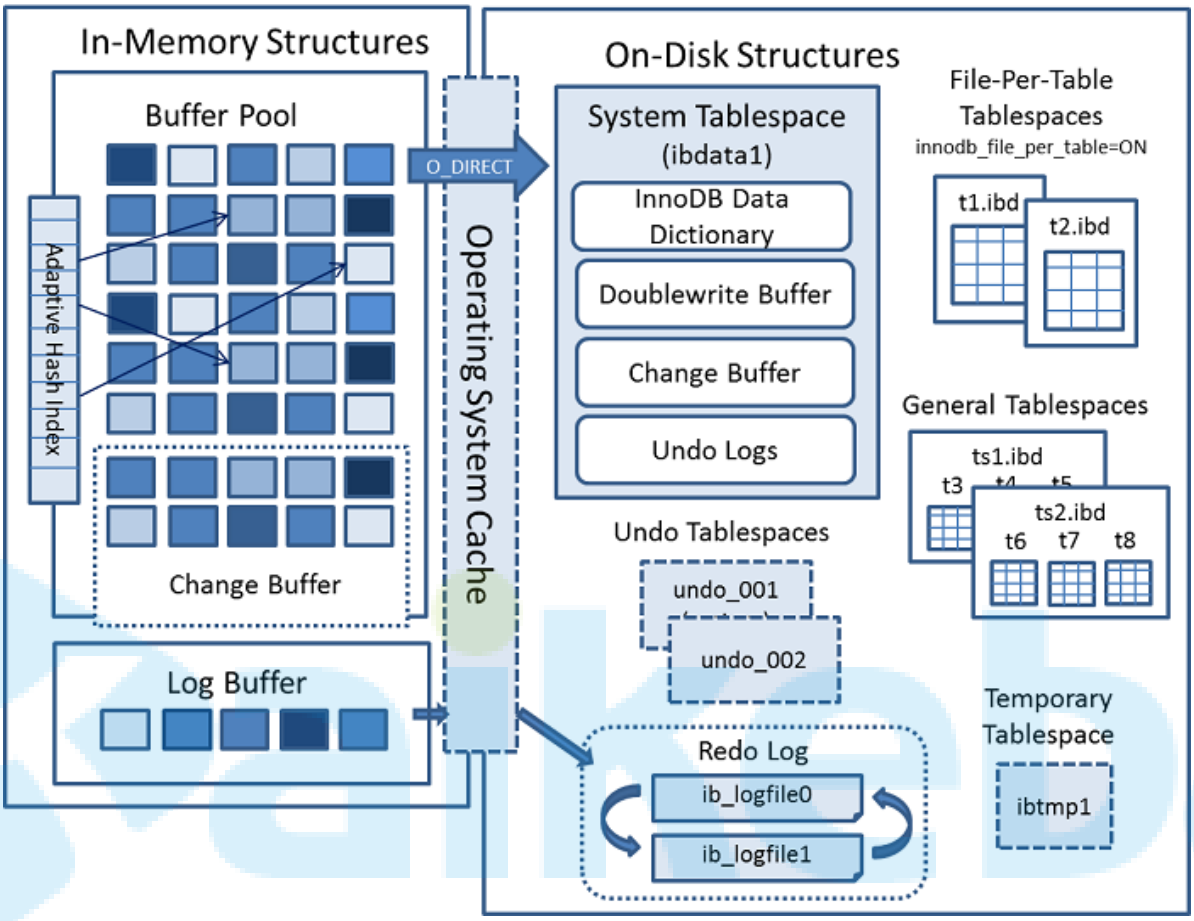
XtraDB存储引擎是由Percona公司提供的存储引擎，该公司还出品了Percona Server这个产品，它是基于MySQL开源代码进行修改之后的产品。其中MariaDB中包含了XtraDB引擎。

阿里对于Percona Server服务器进行修改，衍生了自己的数据库（alisql）。

- **引擎的选择**

那么多引擎我们怎么选择？大部分情况下，InnoDB都是正确的选择。对于如何选择MySQL引擎这件事上可以归纳为一句话：**除非需要用到某些InnoDB不具备的特性，并且没有其他办法可以替代，否则都应该选择InnoDB引擎。**

2、InnoDB架构图

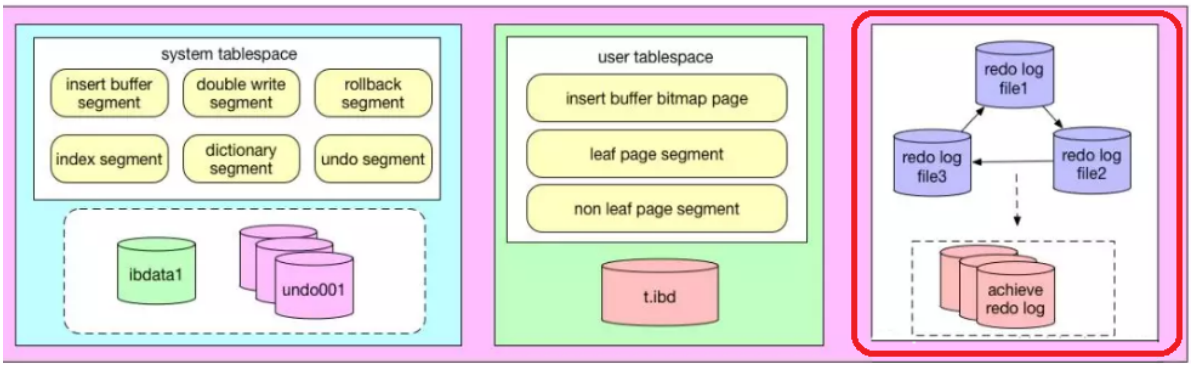


上图详细显示了InnoDB存储引擎的体系架构，从图中可见，InnoDB存储引擎由内存池，后台线程和磁盘文件三大部分组成。接下来我们就来简单了解一下内存相关的概念和原理。

3、InnoDB磁盘文件

> InnoDB的主要的磁盘文件主要分为三大块：**一是系统表空间，二是用户表空间，三是redo日志文件和归档文件。**
>
> 二进制文件(binlog)等文件是MySQL Server层维护的文件，所以未列入InnoDB的磁盘文件中。

1）重做日志文件



哪些文件是重做日志文件？

> 默认情况下，在InnoDB存储引擎的数据目录下会有两个名为**ib_logfile0**和**ib_logfile1**的文件，这就是InnoDB的**重做日志文件(redo log file)**，它记录了对于InnoDB存储引擎的事务日志。
> achieve redo log（重做日志归档）在InnoDB非常古老的版本（MySQL 4.0.6之前的版本）才存在，不过这个功能当时没什么卵用，所以取消了。自从MySQL 8.0.17发布后，又恢复了redo log归档（redo log archiving）功能。我们学习的版本中5.7.29中没有此功能。

重做日志文件的作用是什么?

- 当InnoDB的数据存储文件发生错误时，重做日志文件就能派上用场。InnoDB存储引擎可以使用重做日志文件将数据恢复为正确状态，以此来保证数据的正确性和完整性。
- 为了得到更高的可靠性，用户可以设置多个镜像日志组，将不同的文件组放在不同的磁盘上，以此来提高重做日志的高可用性。

重做日志文件组是如何写入数据的?

每个InnoDB存储引擎至少有1个重做日志文件组(group)，每个文件组下至少有2个重做日志文件，如默认的ib_logfile0和ib_logfile1。

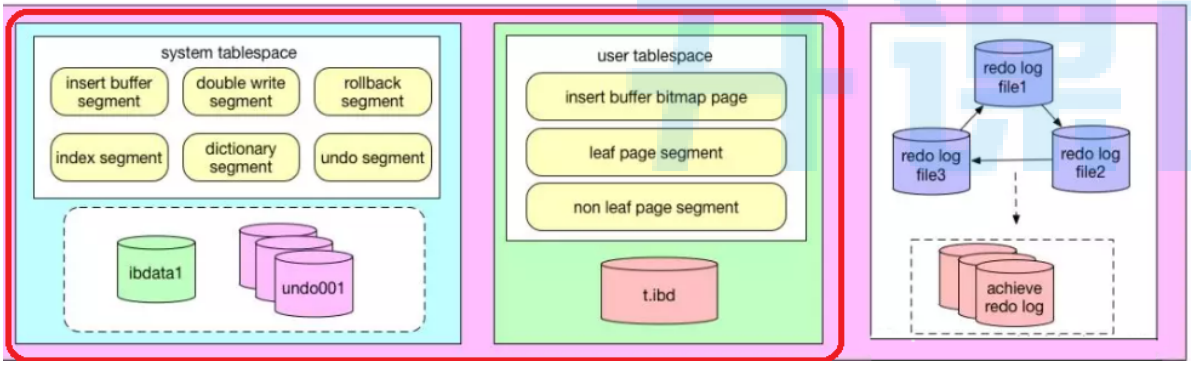
> 在日志组中每个重做日志文件的大小一致，并以循环写入的方式运行。
>
> InnoDB存储引擎先写入重做日志文件1，当文件被写满时，会切换到重做日志文件2，再当重做日志文件2也被写满时，再切换到重做日志文件1。

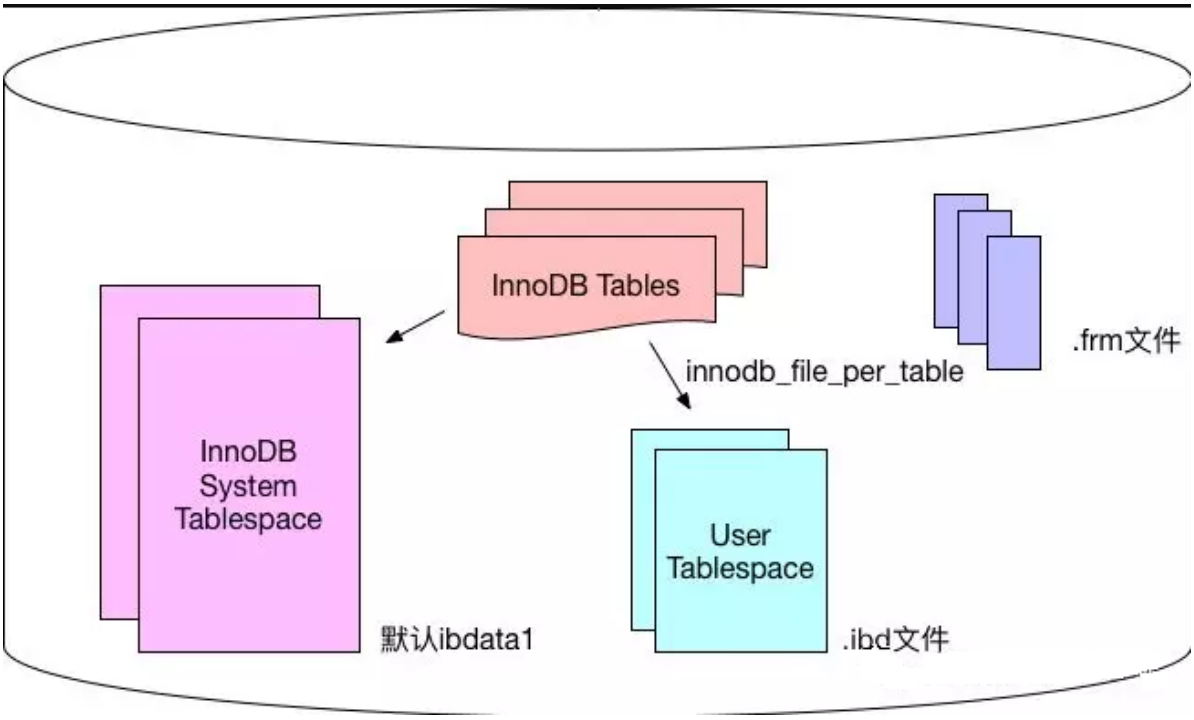
如何设置重做日志文件大小?

用户可以使用innodb_log_file_size来设置重做日志文件的大小，这对InnoDB存储引擎的性能有着非常大的影响。

> 如果重做日志文件设置的太大，数据丢失时，恢复时可能需要很长的时间;
>
> 另一方面，如果设置的太小，重做日志文件太小会导致依据checkpoint的检查需要频繁刷新脏页到磁盘中，导致性能的抖动。

2）系统表空间和用户表空间





系统表空间存储哪些数据?

> 系统表空间是一个共享的表空间，因为它是被多个表共享的。
>
> InnoDB系统表空间包含InnoDB数据字典(元数据以及相关对象)、double write buffer、change buffer、undo logs的存储区域。
>
> 系统表空间也默认包含任何用户在系统表空间创建的表数据和索引数据。

系统表空间配置解析

> 系统表空间是由一个或者多个数据文件组成。
>
> 默认情况下，一个初始大小为10MB，名为ibdata1的系统数据文件在MySQL的data目录下被创建。用户可以使用`innodb_data_file_path`对数据文件的大小和数量进行配置。

`innodb_data_file_path`的格式如下:

```
innodb_data_file_path=datafile1[,datafile2]...
```

示例(这里将/db/ibdata1和/dr2/db/ibdata2两个文件组成系统表空间):

```
innodb_data_file_path=/db/ibdata1:1000M;/dr2/db/ibdata2:1000M:autoextend
```

> 如果这两个文件位于不同的磁盘上，磁盘的负载可能被平均，因此可以提高数据库的整体性能。
>
> 两个文件的文件名之后都跟了属性，表示文件ibdata1的大小为1000MB，文件ibdata2的大小为1000MB，而且用完空间之后可以自动增长(autoextend)。

如何使用用户表空间?

> 如果设置了参数**innodb_file_per_table**，则用户可以将每个基于InnoDB存储引擎的表**产生一个独立的用户表空间。用户表空间的命名规则为：表名.ibd。**

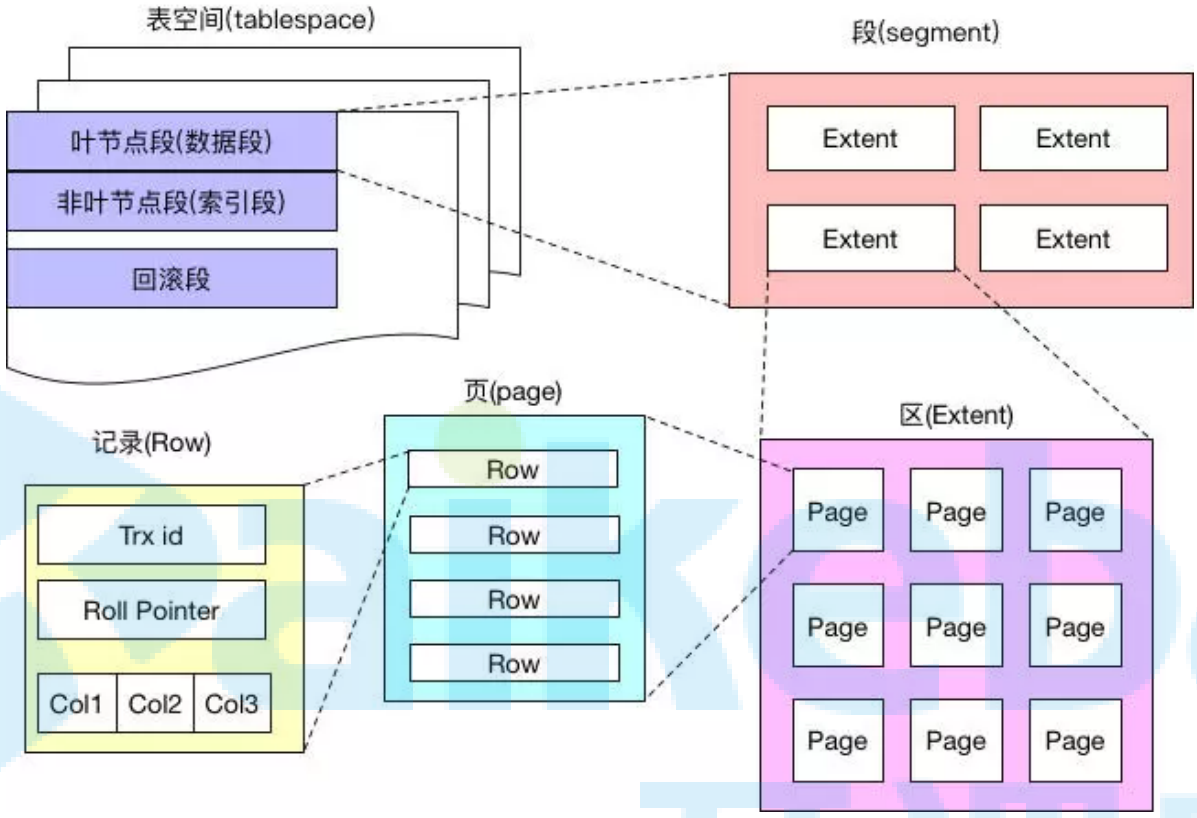
> 通过这种方式，用户不用将所有数据都存放于默认的系统表空间中。

用户表空间存储哪些数据?

> 用户表空间只存储该表的数据、索引信息，其余信息还是存放在默认的系统表空间中。

3）InnoDB逻辑存储结构

InnoDB存储引擎逻辑存储结构可分为五级：表空间、段、区、页、行。



1.表空间

从InnoDB存储引擎的逻辑存储结构看，所有数据都被逻辑地存放在一个空间中，称之为表空间(tablespace)。

从功能上来看，InnoDB存储引擎的表空间分为系统表空间，独占表空间，通用表空间，临时表空间，Undo表空间。

如果开启了独立表空间innodb_file_per_table=1，每张表的数据都会存储到一个独立的表空间，即一个单独的.ibd文件。

InnoDB 存储引擎有一个共享表空间，叫做系统表空间，对一个磁盘上的文件名为ibdata1。如果设置了参数innodb_file_per_table=0，关闭了独占表空间，则所有基于InnoDB存储引擎的表数据都会记录到系统表空间。

2.段

表空间是由各个段组成的，常见的段有数据段、索引段、回滚段等。

如果开启了独立表空间innodb_file_per_table=1，每张表的数据都会存储到一个独立的表空间，即一个单独的.ibd文件。一个用户表空间里面由很多个段组成，创建一个索引时会创建两个段：数据段和索引段。

数据段存储着索引树中叶子节点的数据。

索引段存储着索引树中非叶子节点的数据。

一个段会包含多个区，至少会有一个区，段扩展的最小单位是区。

3.区

一个区由64个连续的页组成，一个区的大小=1M=64个页(16K)。为了保证区中页的连续性，区扩展时InnoDB 存储引擎会一次性从磁盘申请4 ~ 5个区。

4.页

InnoDB 每个页默认大小时是 16KB，页是 InnoDB管理磁盘的最小单位，也InnoDB中磁盘和内存交互的最小单位。

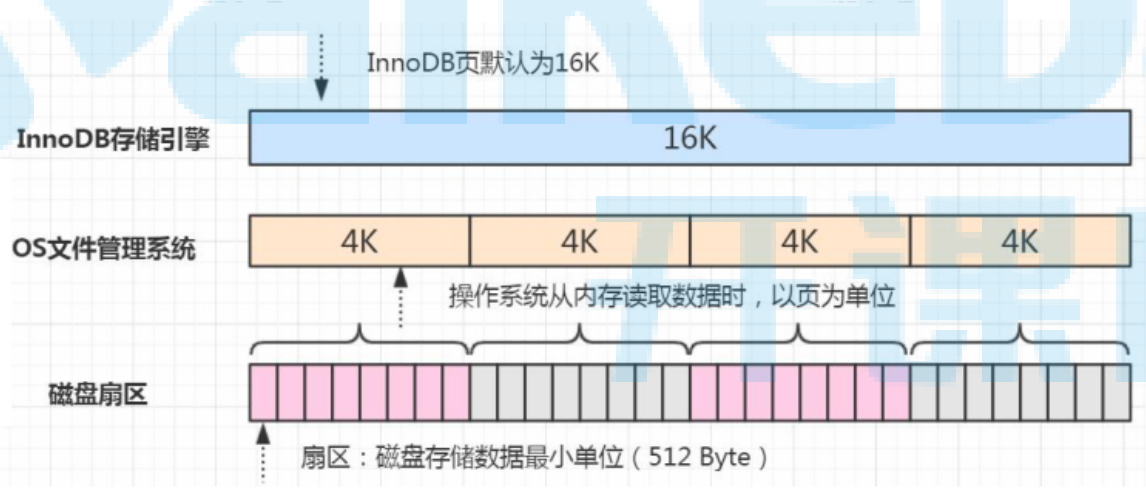
```
show global variables like 'innodb_page_size';
```

索引树上一个节点就是一个页，MySQL规定一个页上最少存储2个数据项。如果向一个页插入数据时，这个页已将满了，就会从区中分配一个新页。如果向索引树叶子节点中间的一个页中插入数据，如果这个页是满的，就会发生页分裂。

操作系统管理磁盘的最小单位也是页，是操作系统读写磁盘最小单位，Linux中页一般是4K，可以通过命令查看。

```
#默认 4096 4K
> getconf PAGE_SIZE
```

所以InnoDB从磁盘中读取一个数据页时，操作系统会分4次从磁盘文件中读取数据到内存。写入也是一样的，需要分4次从内存写入到磁盘中。



5.行

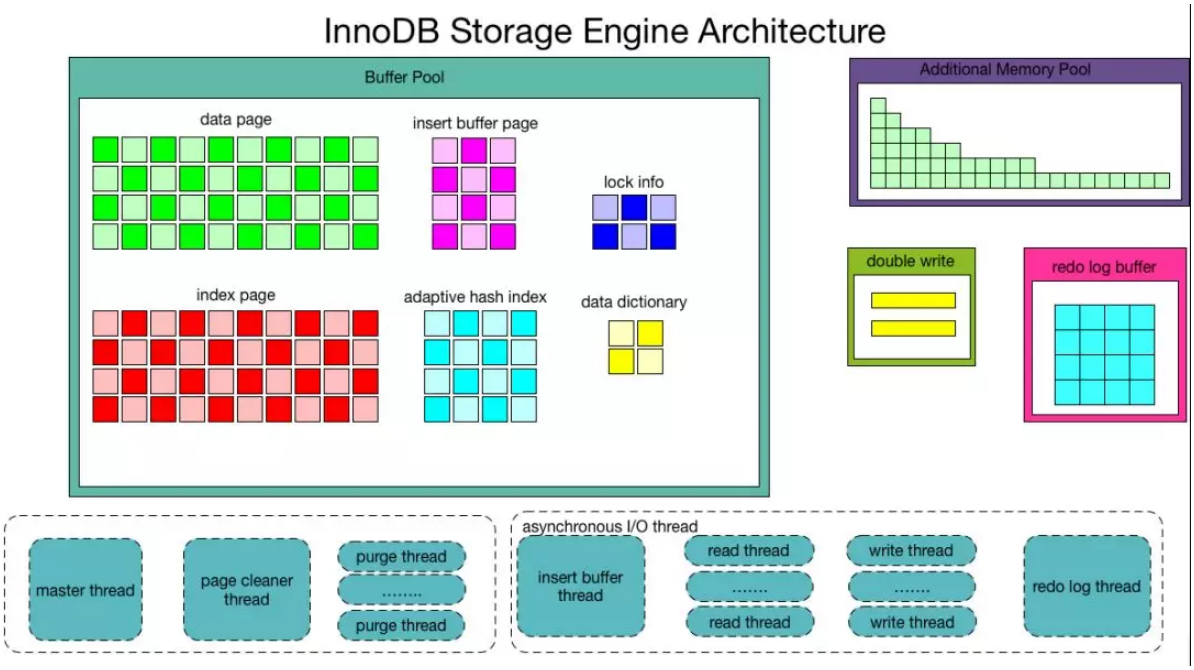
InnoDB的数据是以行为单位存储的，1个页中包含多个行。在MySQL5.7中，InnoDB提供了4种行格式：Compact、Redundant、Dynamic和Compressed行格式，Dynamic为MySQL5.7默认的行格式。

InnoDB行格式官网：https://dev.mysql.com/doc/refman/5.7/en/innodb-row-format.html

创建表时可以指定行格式：

```
CREATE TABLE t1 (c1 INT) ROW_FORMAT=DYNAMIC;
ALTER TABLE tablename ROW_FORMAT=行格式名称;
#修改默认行格式
SET GLOBAL innodb_default_row_format=DYNAMIC;
#查看表行格式
SHOW TABLE STATUS LIKE 'student'\G;
```

4、InnoDB内存结构



1）Buffer Pool缓冲池

概述

> InnoDB存储引擎是基于磁盘存储的，并将其中的记录按照页的方式进行管理。但是由于CPU速度和磁盘速度之间的鸿沟，基于磁盘的数据库系统通常使用**缓冲池**记录来提高数据库的的整体性能。

所以，缓冲池的大小直接影响着数据库的整体性能，可以通过配置参数 `innodb_buffer_pool_size` 来设置。

> 具体来看，缓冲池中缓存的数据页类型有：索引页、数据页、undo页、插入缓冲(insert buffer)、自适应哈希索引(adaptive hash index)、InnoDB存储的锁信息(lock info)和数据字典信息(data dictionary)。

在架构图上可以看到，InnoDB存储引擎的内存区域除了有缓冲池之外，还有重做日志缓冲和额外内存池。InnoDB存储引擎首先将重做日志信息先放到这个缓冲区中，然后按照一定频率将其刷新到重做日志文件中。重做日志缓冲一般不需要设置的很大，该值可由配置参数 `innodb_log_buffer_size` 控制。

数据页和索引页

InnoDB存储引擎工作时，需要以**Page页**为最小单位去将磁盘中的数据加载到内存中，与数据库相关的所有内容都存储在Page结构里。

Page分为几种类型，数据页和索引页就是其中最为重要的两种类型。

更新缓冲（插入缓冲）

> 主要针对次要索引的数据插入存在的问题而设计。

我们都知道，在InnoDB引擎上进行插入操作时，一般需要按照主键顺序进行插入，这样才能获得较高的插入性能。当一张表中存在**次要索引**时，在插入时，数据页的存放还是按照主键进行顺序存放，但是对于次要索引叶节点的插入不再是顺序的了，这时就需要离散的访问次要索引页，由于随机读取的存在导致插入操作性能下降。

InnoDB为此设计了**Change Buffer**来进行插入优化。对于次要索引的插入或者更新操作，不是每一次都直接插入到索引页中，而是先判断插入的非主键索引是否在缓冲池中，若在，则直接插入；若不在，则先放入到一个Change Buffer中。看似数据库这个非主键的索引已经插到叶节点，而实际没有，这时存放在另外一个位置。然后再以一定的频率和情况进行Change Buffer和非聚簇索引页子节点的合并操作。这时通常能够将多个插入合并到一个操作中，这样就大大提高了对于非聚簇索引的插入性能。

自适应哈希索引

> InnoDB会根据访问的频率和模式，为热点页建立哈希索引，来提高查询效率。

哈希(hash)是一种非常快的查找方法,在一般情况下这种查找的时间复杂度为O(1),即一般仅需要一次查找就能定位数据。而B+树的查找次数,取决于B+树的高度,在生产环境中,B+树的高度一般为3~4层,故需要3~4次的查询。
InnoDB存储引擎会监控对表上各索引页的查询。如果观察到建立哈希索引可以带来速度提升,则建立哈希索引,称之为自适应哈希索引(Adaptive Hash Index,AHI) AHI是通过缓冲池的B+树页构造而来,因此建立的速度很快,而且不需要对整张表构建哈希索引。 InnoDB存储引擎会自动根据访问的频率和模式来自动地为某些热点页建立哈希索引。
AHI有一个要求,即对这个页的连续访问模式必须是一样的。例如对于(a,b)这样的联合索引页,其访问模式可以是以下情况:

- WHERE a=xxx
- WHERE a= xxx and b=xxx

访问模式一样指的是查询的条件一样,若交替进行上述两种查询,那么 InnoDB存储引擎不会对该页构造AH此外AH还有如下的要求:

- 以该模式访问了100次
- 页通过该模式访问了N次,其中N=页中记录*1/16

根据 InnoDB存储引擎官方的文档显示,启用AHI后,读取和写入速度可以提高2倍,辅助索引的连接操作性能可以提高5倍。毫无疑问,AHI是非常好的优化模式,其设计思想是数据库自优化的(self-tuning),即无需DBA对数据库进行人为调整。

```
mysql> show engine innodb status \G

-------------------------------------
INSERT BUFFER AND ADAPTIVE HASH INDEX
-------------------------------------
Ibuf: size 1, free list len 0, seg size 2, 0 merges
merged operations:
 insert 0, delete mark 0, delete 0
discarded operations:
 insert 0, delete mark 0, delete 0
Hash table size 34673, node heap has 0 buffer(s)
Hash table size 34673, node heap has 0 buffer(s)
Hash table size 34673, node heap has 0 buffer(s)
Hash table size 34673, node heap has 0 buffer(s)
Hash table size 34673, node heap has 0 buffer(s)
Hash table size 34673, node heap has 0 buffer(s)
Hash table size 34673, node heap has 0 buffer(s)
Hash table size 34673, node heap has 0 buffer(s)
0.00 hash searches/s, 0.00 non-hash searches/s

```

可以通过参数innodb_adaptive_hash_index来考虑禁用或启动此特性，默认是开启状态。

```
mysql> show variables like 'innodb_adaptive_hash_index';
+-----------------------------+-------+
| Variable_name               | Value |
+-----------------------------+-------+
| innodb_adaptive_hash_index | ON    |
+-----------------------------+-------+
1 row in set (0.00 sec)
```

锁信息

我们都知道，InnoDB存储引擎会在行级别上对表数据进行上锁。不过InnoDB也会在数据库内部其他很多地方使用锁，从而允许对多种不同资源提供并发访问。数据库系统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。关于锁的具体知识我们之后再进行详细学习。

数据字典信息

InnoDB有自己的表缓存，可以称为表定义缓存或者数据字典(Data Dictionary)。当InnoDB打开一张表，就增加一个对应的对象到数据字典。

数据字典是对数据库中的数据、库对象、表对象等的元信息的集合。在MySQL中，数据字典信息内容就包括表结构、数据库名或表名、字段的数据类型、视图、索引、表字段信息、存储过程、触发器等内容。MySQL INFORMATION_SCHEMA库提供了对数据局元数据、统计信息、以及有关MySQL server的访问信息（例如：数据库名或表名，字段的数据类型和访问权限等）。该库中保存的信息也可以称为MySQL的数据字典。

2）额外内存池（Addtional memory pool）

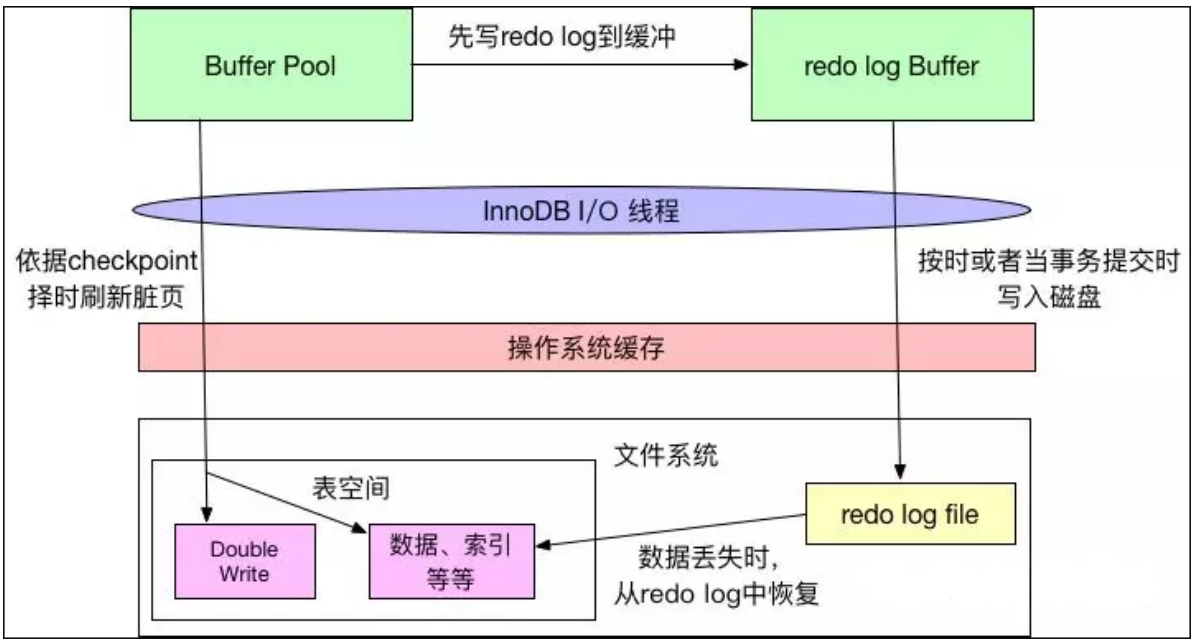
额外内存池是InnoDB存储引擎用来存放数据字典信息以及一些内部数据结构的内存空间，控制参数为：

```
innodb_additional_mem_pool_size
```

这个参数我们平时调整的可能不是太多，很多人都使用了默认值，可能很多人都不是太熟悉这个参数的作用。所以当我们一个MySQL Instance中的数据库对象非常多的时候，是需要适当调整该参数的大小以确保所有数据都能存放在内存中提高访问效率的。这个参数大小是否足够还是比较容易知道的，因为当过小的时候，MySQL 会记录 Warning 信息到数据库的 error log 中，这时候你就知道该调整这个参数大小了。

> 注：此参数在 MySQL 5.7.4 中移除。

3）Redo log Buffer重做日志缓冲



> 如上图所示，InnoDB在缓冲池中变更数据时，会首先将相关变更写入重做日志缓冲中，然后再按时或者当事务提交时写入磁盘，这符合Force-log-at-commit原则；
>
> 当重做日志写入磁盘后，缓冲池中的变更数据才会依据checkpoint机制择时写入到磁盘中，这符合WAL原则。

在checkpoint择时机制中，就有重做日志文件写满的判断，所以，如前文所述，如果重做日志文件太小，经常被写满，就会频繁导致checkpoint将更改的数据写入磁盘，导致性能抖动。

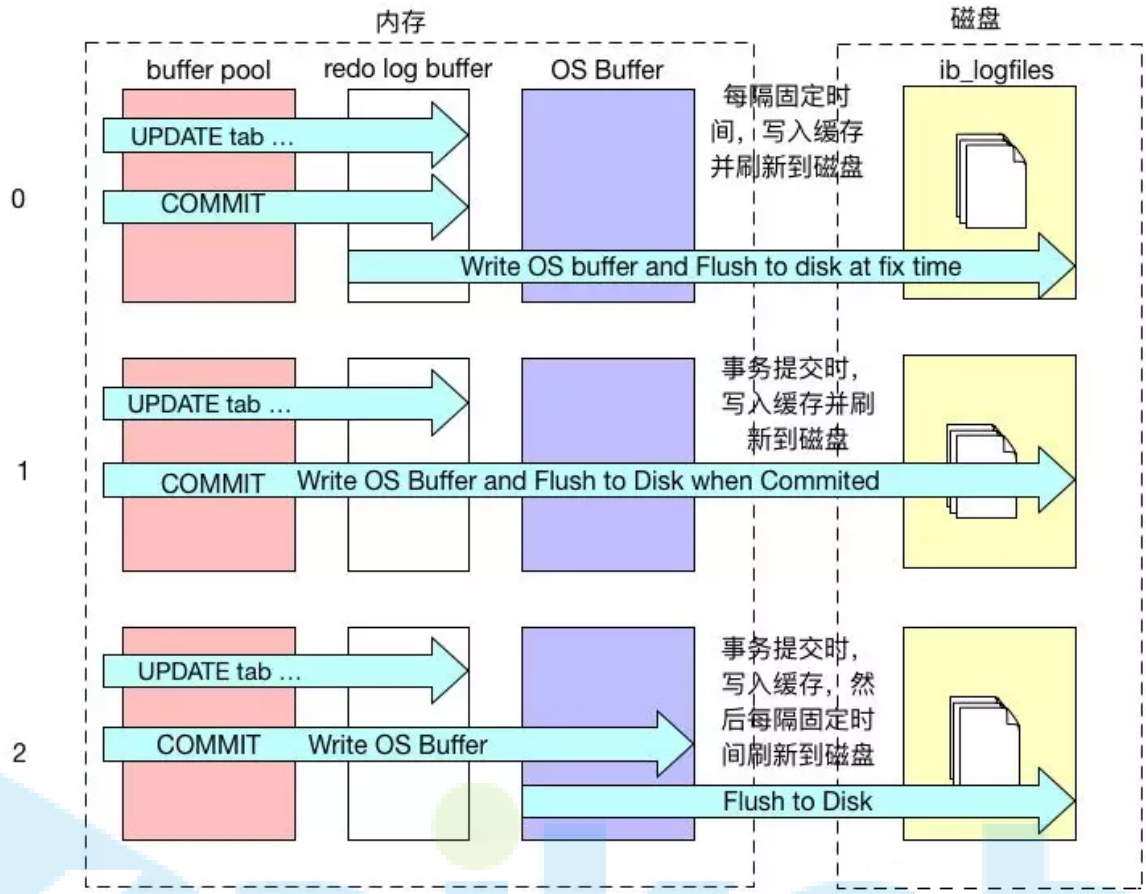
操作系统的文件系统是带有缓存的，当InnoDB向磁盘写入数据时，有可能只是写入到了文件系统的缓存中，没有真正的"落袋为安"。

InnoDB的innodb_flush_log_at_trx_commit属性可以控制每次事务提交时InnoDB的行为。

- 当属性值为0时，事务提交时，不会对重做日志进行写入操作，而是等待主线程按时写入每秒写入一次；
- 当属性值为1时，事务提交时，会将重做日志写入文件系统缓存，并且调用文件系统的fsync，将文件系统缓冲中的数据真正写入磁盘存储，确保不会出现数据丢失；
- 当属性值为2时，事务提交时，也会将日志文件写入文件系统缓存，但是不会调用fsync，而是让文件系统自己去判断何时将缓存写入磁盘。

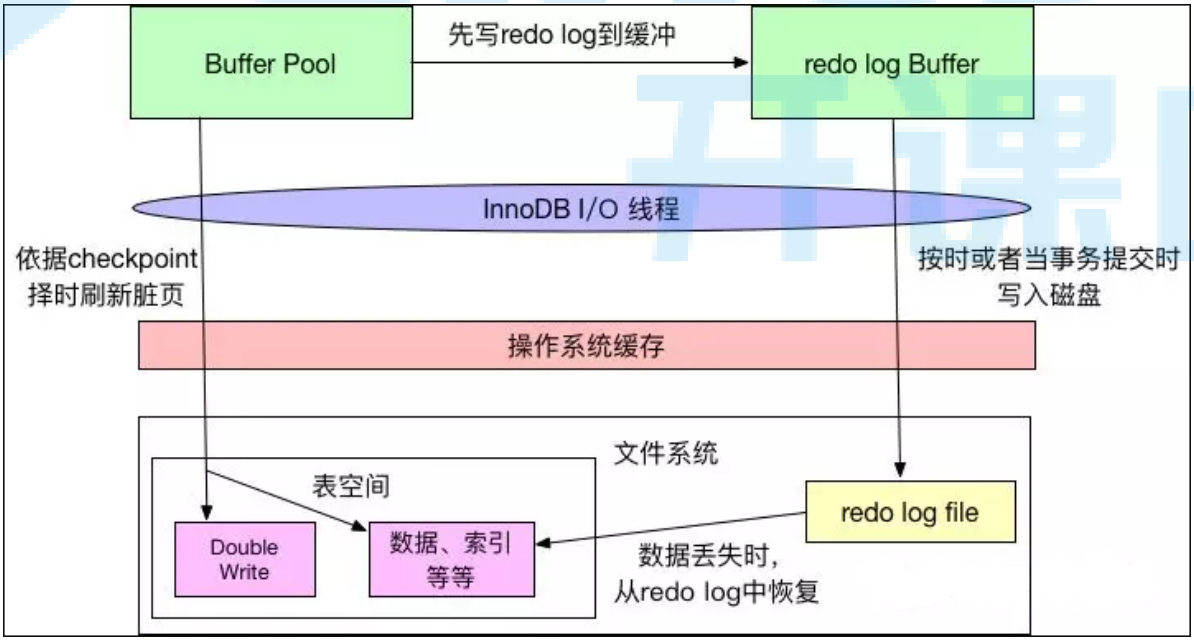
> innodb_flush_log_at_trx_commit是InnoDB性能调优的一个基础参数，涉及InnoDB的写入效率和数据安全。当参数值为0时，写入效率最高，但是数据安全最低；参数值为1时，写入效率最低，但是数据安全最高；参数值为2时，二者都是中等水平。一般建议将该属性值设置为1，以获得较高的数据安全性，而且也只有设置为1，才能保证事务的持久性。

日志的刷盘机制如下图所示：



4）内存数据落盘

1.整体思路分析



> InnoDB内存缓冲池中的数据page要完成持久化的话，是通过两个流程来完成的，一个是脏页落盘；一个是预写redo log日志。

当缓冲池中的页的版本比磁盘要新时，数据库需要将新版本的页从缓冲池刷新到磁盘。但是如果每次一个页发送变化，就进行刷新，那么性能开发是非常大的，于是InnoDB采用了**Write Ahead Log（WAL）**策略和**Force Log at Commit**机制实现事务级别下数据的持久性。

WAL要求数据的变更写入到磁盘前，首先必须将内存中的日志写入到磁盘；

Force-log-at-commit要求当一个事务提交时，所有产生的日志都必须刷新到磁盘上，如果日志刷新成功后，缓冲池中的数据刷新到磁盘前数据库发生了宕机，那么重启时，数据库可以从日志中恢复数据。

为了确保每次日志都写入到重做日志文件，在每次将重做日志缓冲写入重做日志后，必须调用一次**fsync**操作，将缓冲文件从文件系统缓存中真正写入磁盘。

可以通过`innodb_flush_log_at_trx_commit`来控制重做日志刷新到磁盘的策略。

2.脏页落盘：

在数据库中进行**读取操作**，将从磁盘中读到的页放在缓冲池中，下次再读相同的页时，首先判断该页是否在缓冲池中。若在缓冲池中，称该页在缓冲池中被命中，直接读取该页。否则，读取磁盘上的页。

对于数据库中页的**修改操作**，则首先修改在缓冲池中的页，然后再以一定的频率刷新到磁盘上。页从缓冲池刷新回磁盘的操作并不是在每次页发生更新时触发，而是通过一种称为**CheckPoint的机制**刷新回磁盘。

3.重做日志落盘：

那Log Buffer什么时候写入到redo log?

Log Buffer写入磁盘的时机，由参数 innodb_flush_log_at_trx_commit 控制，默认是 1，表示事务提交后立即落盘。

https://dev.mysql.com/doc/refman/5.7/en/innodb-parameters.html#sysvar_innodb_flush_log_at_trx_commit

```
1 show VARIABLES like 'innodb_flush_log_at_trx_commit';
```

用户程序写入数据到磁盘文件时，需要调用操作系统的接口，操作系统本身是有缓冲区的，之后依赖操作系统机制不时的将缓存中刷新到磁盘文件中。用户程序可以执行fsync操作将操作系统缓冲区的数据刷入到磁盘文件中。

- 0：MySQL每秒一次将数据从log buffer写入日志文件并同时fsync刷新到磁盘中。

  每次事务提交时，不会立即把 log buffer 里的数据写入到redo log日志文件的。如果MySQL崩溃或者服务器宕机，此时内存里的数据会全部丢失，最多会丢失1秒的事务。

- 1：每次事务提交时，MySQL将数据将从log buffer写入日志文件并同时fsync刷新到磁盘中。

  该模式为系统默认，MySQL崩溃已经提交的事务不会丢失，要完全符合ACID，必须使用默认设置1。

- 2：每次事务提交时，MySQL将数据从log buffer写入日志文件，MySQL每秒执行一次fsync操作将数据同步到磁盘中。

> 每次事务提交时，都会将数据刷新到操作系统缓冲区，可以认为已经持久化磁盘，如果MySQL崩溃已经提交的事务不会丢失。但是如果服务器宕机或者意外断电，操作系统缓存内的数据会丢失，所以最多丢失1秒的事务。
>
> 只有设置为1是最安全但是性能消耗的方式，可以真正地保证事务的持久性，但是由于MySQL执行刷新操作 fsync() 是阻塞的，直到完成后才会返回，我们知道写磁盘的速度是很慢的，因此 MySQL 的性能会明显地下降。
>
> 0和2的性能最好的模式，综合安全性和性能的考虑，在**业务中经常使用的2这种模式**，在MySQL异常重启时不会丢失数据，只有在服务器意外宕机时才会丢失1秒的数据，这种情况几率是很低的，相对于性能来说，这时可以容忍的。

5）CheckPoint检查点机制

1.简介

> **思考一下这个场景**：如果重做日志可以无限地增大，同时缓冲池也足够大，那么是不需要将缓冲池中页的新版本刷新回磁盘。因为当发生宕机时，完全可以通过重做日志来恢复整个数据库系统中的数据到宕机发生的时刻。

但是这需要两个前提条件：

1. 缓冲池可以缓存数据库中所有的数据；
2. 重做日志可以无限增大

因此Checkpoint（检查点）技术就诞生了，目的是解决以下几个问题：1、缩短数据库的恢复时间；2、缓冲池不够用时，将脏页刷新到磁盘；3、重做日志不可用时，刷新脏页。

- 当数据库发生宕机时，数据库不需要重做所有的日志，因为Checkpoint之前的页都已经刷新回磁盘。数据库只需对Checkpoint后的重做日志进行恢复，这样就大大缩短了恢复的时间。
- 当缓冲池不够用时，根据LRU算法会溢出最近最少使用的页，若此页为脏页，那么需要强制执行Checkpoint，将脏页也就是页的新版本刷回磁盘。
- 当重做日志出现不可用时，因为当前事务数据库系统对重做日志的设计都是循环使用的，并不是让其无限增大的。重做日志可以被重用的部分是指这些重做日志已经不再需要，当数据库发生宕机时，数据库恢复操作不需要这部分的重做日志，因此这部分就可以被覆盖重用。如果重做日志还需要使用，那么必须强制Checkpoint，将缓冲池中的页至少刷新到当前重做日志的位置。

对于InnoDB存储引擎而言，是通过LSN（Log Sequence Number）来标记版本的。

LSN是8字节的数字，每个页有LSN，重做日志中也有LSN，Checkpoint也有LSN。可以通过命令`SHOW ENGINE INNODB STATUS`来观察：

```
mysql> show engine innodb status \G
---
LOG
---
Log sequence number 9849834
Log flushed up to   9849834
Pages flushed up to 9849834
Last checkpoint at  9849825
0 pending log flushes, 0 pending chkp writes
19 log i/o's done, 0.00 log i/o's/second
```

Checkpoint发生的时间、条件及脏页的选择等都非常复杂。而Checkpoint所做的事情无外乎是将缓冲池中的脏页刷回到磁盘，不同之处在于每次刷新多少页到磁盘，每次从哪里取脏页，以及什么时间触发Checkpoint。

2.Checkpoint分类

在InnoDB存储引擎内部，有两种Checkpoint，分别为：Sharp Checkpoint、Fuzzy Checkpoint

> sharp checkpoint：在关闭数据库的时候，将buffer pool中的脏页全部刷新到磁盘中。
> fuzzy checkpoint：数据库正常运行时，在不同的时机，将部分脏页写入磁盘。仅刷新部分脏页到磁盘，也是为了避免一次刷新全部的脏页造成的性能问题。

Fuzzy Checkpoint：

> 1、Master Thread Checkpoint；
>
> 2、FLUSH_LRU_LIST Checkpoint；
>
> 3、Async/Sync Flush Checkpoint；
>
> 4、Dirty Page too much Checkpoint

1、Master Thread Checkpoint

在Master Thread中，会以每秒或者每10秒一次的频率，将部分脏页从内存中刷新到磁盘，这个过程是异步的。正常的用户线程对数据的操作不会被阻塞。

2、FLUSH_LRU_LIST Checkpoint

> FLUSH_LRU_LIST checkpoint是在单独的page cleaner线程中执行的。
>
> MySQL对缓存的管理是通过buffer pool中的LRU列表实现的，LRU 空闲列表中要保留一定数量的空闲页面，来保证buffer pool中有足够的空闲页面来相应外界对数据库的请求。
>
> 当这个空间页面数量不足的时候，发生FLUSH_LRU_LIST checkpoint。
>
> 空闲页的数量由innodb_lru_scan_depth参数表来控制的，因此在空闲列表页面数量少于配置的值的时候，会发生checkpoint，剔除部分LRU列表尾端的页面。

```
mysql> show variables like 'innodb_lru_scan_depth';
+------------------------+-------+
| Variable_name          | Value |
+------------------------+-------+
| innodb_lru_scan_depth | 2000  |
+------------------------+-------+
1 row in set (0.00 sec)
```

因为InnoDB存储引擎需要保证LRU列表中需要有差不多100个空闲页可供使用。

在InnoDB1.1.x版本之前，需要检查LRU列表中是否有足够的可用空间操作发生在用户查询线程中，显然这会阻塞用户的查询操作。倘若没有100个可用空闲页，那么InnoDB存储引擎会将LRU列表尾端的页移除。如果这些页中有脏页，那么需要进行Checkpoint，而这些页是来自LRU列表的，因此称为FLUSH_LRU_LIST Checkpoint。

而从MySQL 5.6版本，也就是InnoDB1.2.x版本开始，这个检查被放在了一个单独的Page Cleaner线程中进行，并且用户可以通过参数innodb_lru_scan_depth控制LRU列表中可用页的数量，该值默认为1024.

3、Async/Sync Flush Checkpoint

> Async/Sync Flush checkpoint是在单独的page cleaner线程中执行的。
>
> Async/Sync Flush checkpoint 发生在重做日志不可用的时候，将buffer pool中的一部分脏页刷新到磁盘中，在脏页写入磁盘之后，事务对应的重做日志也就可以释放了。

> 关于redo_log文件的的大小，可以通过`innodb_log_file_size`来配置。

```
mysql> show variables like 'innodb_log_file_size';
+----------------------+-----------+
| Variable_name        | Value     |
+----------------------+-----------+
| innodb_log_file_size | 268435456 |
+----------------------+-----------+
1 row in set (0.01 sec)
```

对于是执行Async Flush checkpoint还是Sync Flush checkpoint，由`checkpoint_age`以及`async_water_mark`和`sync_water_mark`来决定。

```
##即checkpoint_age等于最新的lsn减去已经刷新到磁盘的lsn的值
checkpoint_age = redo_lsn-checkpoint_lsn
async_water_mark = 75%*innodb_log_file_size
sync_water_mark = 90%*innodb_log_file_size
```

> 1. 当checkpoint_age<async_water_mark的时候，无需执行Flush checkpoint。也就说，redo log剩余空间超过25%的时候，无需执行Async/Sync Flush checkpoint。
> 2. 当async_water_mark<checkpoint_age<sync_water_mark的时候，执行Async Flush checkpoint，也就说，redo log剩余空间不足25%，但是大于10%的时候，执行Async Flush checkpoint，刷新到满足条件1
> 3. 当checkpoint_age>sync_water_mark的时候，执行sync Flush checkpoint。也就说，redo log剩余空间不足10%的时候，执行Sync Flush checkpoint，刷新到满足条件1。
> 在mysql 5.6之后，不管是Async Flush checkpoint还是Sync Flush checkpoint，都不会阻塞用户的查询进程。

总结：

> 由于磁盘是一种相对较慢的存储设备，内存与磁盘的交互是一个相对较慢的过程
> 由于innodb_log_file_size定义的是一个相对较大的值，正常情况下，由前面两种checkpoint刷新脏页到磁盘，在前面两种checkpoint刷新脏页到磁盘之后，脏页对应的redo log空间随即释放，一般不会发生Async/Sync Flush checkpoint。同时也要意识到，为了避免频繁低发生Async/Sync Flush checkpoint，也应该将innodb_log_file_size配置的相对较大一些。

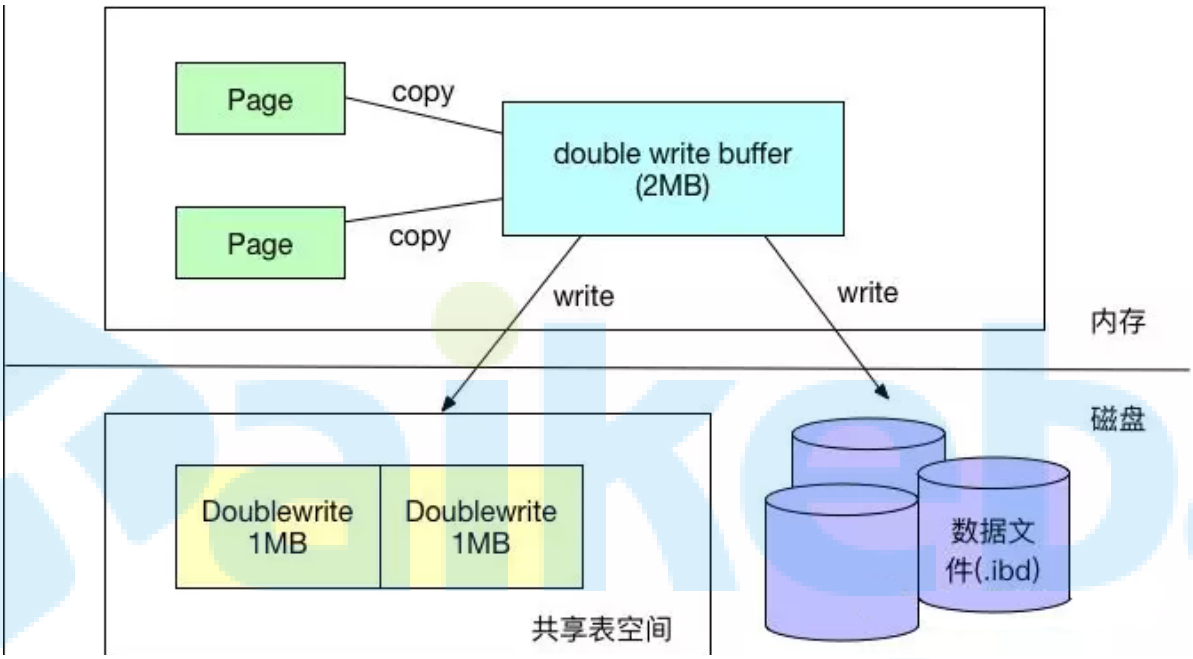
4、Dirty Page too much

> Dirty Page too much Checkpoint是在Master Thread 线程中每秒一次的频率实现的。
>
> Dirty Page too much 意味着buffer pool中的脏页过多，执行checkpoint脏页刷入磁盘，保证buffer pool中有足够的可用页面。
>
> Dirty Page 由innodb_max_dirty_pages_pct配置，innodb_max_dirty_pages_pct的默认值在innodb 1.0之前是90%，之后是75%。

```
mysql> show variables like 'innodb_max_dirty_pages_pct';
+-----------------------------+-----------+
| Variable_name               | Value     |
+-----------------------------+-----------+
| innodb_max_dirty_pages_pct | 75.000000 |
+-----------------------------+-----------+
1 row in set (0.01 sec)
```

6）Double Write双写

如果说Insert Buffer给InnoDB存储引擎带来了性能上的提升，那么Double Write带给InnoDB存储引擎的是数据页的可靠性。

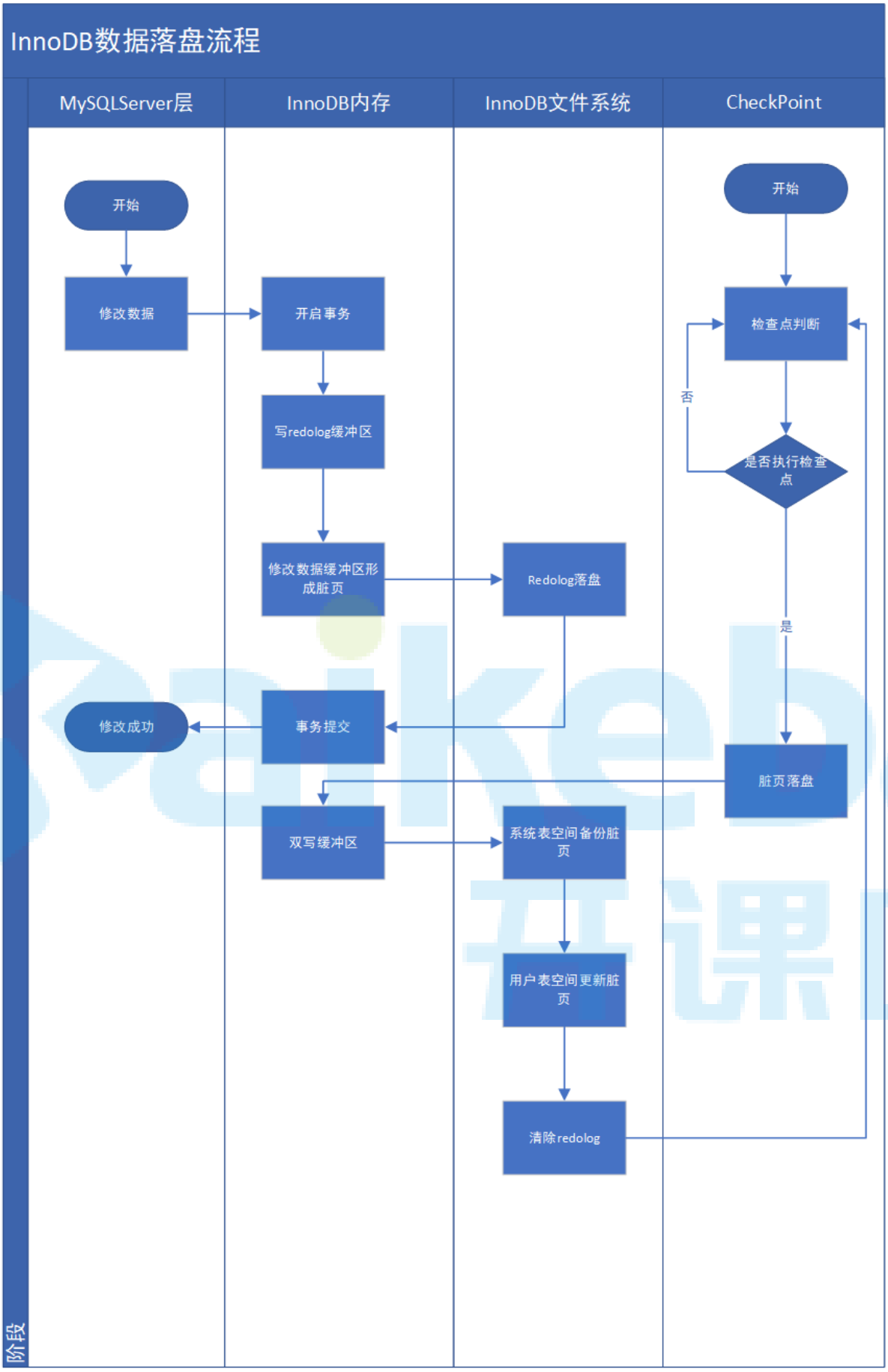


如上图所示，**Double Write由两部分组成，一部分是内存中的double write buffer，大小为2MB，另一部分是物理磁盘上共享表空间连续的128个页，大小也为2MB。**

在对缓冲池的脏页进行刷新时，并不直接写磁盘，而是通过memcpy函数将脏页先复制到内存中的double write buffer区域，之后通过double write buffer再分两次，每次1MB顺序地写入共享表空间的物理磁盘上，然后马上调用fsync函数，同步磁盘，避免操作系统缓冲写带来的问题。在完成double write页的写入后，再讲double wirite buffer中的页写入各个表空间文件中。

如果操作系统在将页写入磁盘的过程中发生了崩溃，在恢复过程中，InnoDB存储引擎可以从共享表空间中的double write中找到该页的一个副本，将其复制到表空间文件中，再应用重做日志。

5、总结：InnoDB数据落盘


InnoDB数据落盘流程
MySQLServer层
InnoDB内存
InnoDB文件系统
CheckPoint
开始
修改数据
开启事务
写redolog缓冲区
修改数据缓冲区形成脏页
Redolog落盘
事务提交
修改成功
双写缓冲区
系统表空间备份脏页
用户表空间更新脏页
清除redolog
开始
检查点判断
否
是否执行检查点
是
脏页落盘
阶段