图文 专栏大纲

手机观看

2101 人次阅读 2020-01-04 20:05:45

详情 评论

week 2

008 **案例实战：**互联网公司每秒10万并发的数据库如何进行性能测试？

009 **案例实战：**如何对生产环境中的数据库进行360°无死角压测？

010 **案例实战：**在数据库的压测过程中，如何360°无死角观察性能表现？

011 **案例实战：**如何使用可视化工具对MySQL服务器进行监控

012 了解MySQL架构之后，来看看一条insert语句的底层执行原理？

week 3

017 继续探索：InnoDB存储引擎是如何基于表的概念来存储多个数据页的？（上）

week 4

018 继续探索：InnoDB存储引擎是如何基于表的概念来存储多个数据页的？（下）

019 **案例实战：**每隔一段时间就会出现的MySQL性能异常抖动的优化

020 **案例实战：**系统访问数据库时，连接池异常的问题优化

021 **案例实战：**偶然性发生的数据库查询性能异常抖动的问题

优化

022 **案例实战：**MySQL无法抗下高并发连接的问题优化

023 关于MySQL写入数据的优秀提问与答疑总结

024 **定期复习：**用一张思维导图梳理MySQL写入数据的原理

week 5

025 **案例实战：**对生产环境下的MySQL进行360°全方位调优

026 MySQL作为一个数据库，到底会不会数据丢失：redo日志详解（上）

027 MySQL作为一个数据库，到底会不会数据丢失：redo日志详解（下）

028 我们删除数据时，MySQL会立即在物理磁盘上删除吗？

029 删库跑路？网上流行的段子到底怎么破？如何恢复误删的数据

030 关于MySQL是否会丢失数据的优秀提问与答疑总结

031 **定期复习：**一张思维导图给你梳理MySQL数据丢失问题

week 6

032 **案例实战：**生产环境下的MySQL数据冷备份方案以及崩溃恢复

033 **案例实战：**企业级的MySQL数据热备份方案以及极速恢复

034 **案例实战：**企业级的精细化数据库和数据表的备份方案和恢复方案

035 **案例实战：**高级工程师必备的生产级数据库克隆技术方案

036 **案例实战：**高级工程师必备的生产级表复制技术方案

week 7

week 8

044 **案例实战：**对线上的千万级大表加字段时，性能极慢问题的优化

045 **案例实战：**线上系统报出MySQL死锁异常，如何进行排查、定位和解决?

046 **案例实战：**线上系统insert语句导致过多锁的时候，如何进行性能优化?

047 **案例实战：**线上系统的更新语句因为锁导致性能过慢时，如何优化?

week 9

053 当我们使用select查询数据时，如何使用索引来提升性能?

054 关于更新语句的锁性能问题的案例实战的优秀提问与答疑总结

055 **定期复习：**一张思维导图给你梳理MySQL锁性能问题的案例实战

week 10

056 索引的花式使用规则：普通索引、唯一索引、聚簇索引以及各种索引类型（上）

057 索引的花式使用规则：普通索引、唯一索引、聚簇索引以

及各种索引类型（下）

058 **案例实战：**社交电商场景下的商品表结构设计实战（上）

059 **案例实战：**社交电商场景下的商品表结构设计实战（下）

060 **案例实战：**社交电商场景下的交易系统的表结构设计实战（上）

week 11

061 **案例实战：**社交电商场景下的交易系统的表结构设计实战（下）

062 **案例实战：**新零售电商场景下的营销系统的表结构设计实

战（上）

063 **案例实战：**新零售电商场景下的营销系统的表结构设计实战（下）

064 如果我们要进行多表join查询，MySQL底层是如何实现的

065 使用order by语句对数据进行排序时，MySQL底层是如何实现的？

066 关于MySQL的索引以及查询原理的优秀提问与答疑总结

067 **定期复习：**一张思维导图给你梳理MySQL的索引原理以及查询原理

week 12

week 13

073 我们想对一条SQL语句调优时，如何使用explain查看执行计划？（下）

074 如何对复杂的多表关联join语句进行性能调优？

075 使用count语句进行计数的时候，如何优化执行性能？

076 **案例实战：**百万级日活用户场景下的用户系统复杂SQL调优实战

077 **案例实战：**亿级数据量的商品系统的复杂SQL调优实战

078 关于SQL优化原理的优秀提问与答疑总结

079 **定期复习：**一张思维导图梳理MySQL优化原理

week 14

080 **案例实战：**一线电商大厂复杂购物车功能的SQL调优实战

081 **案例实战：**每天几十万订单场景下的订单系统的SQL调优实战

082 **案例实战：**数十亿数据量级的评论系统的SQL调优实战

083 MySQL单机并发过高时，如何通过主从架构实现读写分离

084 深入探究MySQL主从集群同步的底层原理是如何实现的?

（上）

085 关于SQL调优案例的优秀提问与答疑总结

086 **定期复习：**一张思维导图给你梳理SQL调优案例

week 15

087 深入探究MySQL主从集群同步的底层原理是如何实现的？（下）

088 如果要让MySQL实现高可用架构，避免单点故障，应该怎么做？（上）

089 如果要让MySQL实现高可用架构，避免单点故障，应该怎

089 如果要让MySQL实现高可用架构，避免单点故障，应该怎么做？（下）

090 **案例实战：**数据库主从集群的数据不一致问题，应该如何解决？

091 **案例实战：**如果因为线上网络故障导致从库数据大规模延迟，应该如何解决？

092 关于MySQL读写分离架构的优秀提问与答疑总结

093 **定期复习：**一张思维导图给你梳理MySQL读写分离与高可用架构

week 16

094 **案例实战：**如果主库突然宕机，此时数据库主从集群的稳

定性如何保障?

095 **案例实战：**数据库主从集群下的各种常见问题的解决方案详解（上）

096 **案例实战：**数据库主从集群下的各种常见问题的解决方案详解（下）

097 单表数据量过大时，如何使用MySQL分区表进行优化?

098 为什么互联网公司一般都不会使用MySQL的分区表方案?

099 关于MySQL读写分离架构案例实战的优秀提问与答疑总结

100 **定期复习：**用一张思维导图给你梳理MySQL读写分离架构的案例实战

week 17

101 当MySQL写入并发过大以及数据量过大的时候，如何设计分库分表架构?

102 如何挑选一个数据库中间件去实现分库分表技术方案?

103 分库之后，如何进行分布式的id生成：各种技术方案详解对比

104 对国内大厂开源的分布式id生成机制的详解：时钟回拨等问题如何解决?

105 **案例实战：**一线互联网公司的数据库架构是如何设计的?

106 关于MySQL分库分表方案的优秀提问与答疑总结

106 关于MySQL分库分表方案的优劣提问与合规总结

107 **定期复习：**一张思维导图给你梳理MySQL的分库分表解决方案

week 18

108 **案例实战：**大型电商网站上亿数据量的用户表如何进行水平拆分？

109 **案例实战：**互联网公司是如何进行大宽表的垂直拆分方案设计的？

110 **案例实战：**当完成基于分库分表的系统架构改造之后，如何进行旧数据迁移？

111 **案例实战：**一线电商公司的订单系统是如何进行数据库设计的？

112 **案例实战：**下一个难题，如果需要进行跨库的分页操作，应该怎么做？

week 19

113 **案例实战：**大型社区网站中的核心数据表的动态属性变更方案如何设计？

114 **案例实战：**上亿用户的社交APP的核心数据表应该如何设计？

115 **案例实战：**当分库分表技术方案运行几年后，再次进行扩

图文 01 天天写CRUD，你知道你的系统是如何跟MySQL打交道的吗？ 手机观看

2565 人次阅读 2020-01-13 07:00:00

详情 评论

天天写CRUD，你知道你的系统是如何跟MySQL打交道的吗？

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

1、Java工程师眼中的数据库是什么东西？

从今天开始，我们将要开始一个MySQL的专栏，一起来研究MySQL数据库的底层原理和各种实践案例，以及互联网公司的技术方案。

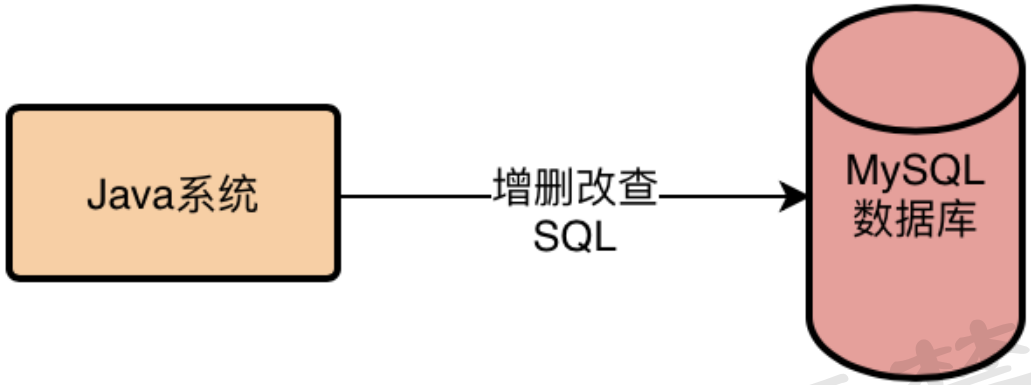
现在我们先来看看，在一个Java工程师眼中的数据库是什么东西？

平时我们在做Java系统时，一般情况下都会连接到一个MySQL数据库上去，执行各种增删改查的语句。

据我所知，目前行业里大部分的Java工程师对MySQL的了解和掌握程度，大致就停留在这么一个阶段：对MySQL可以建库建表建索引，然后就是执行增删改查去更新和查询里面的数据！

所以我们看下面的图，很多Java工程师眼中的数据库大致就是下面这样子。

（**附加说明**：我在写《从0开始带你成为JVM实战高手》专栏时，采用的是亿图图示这个画图工具，现在为了统一画图风格，本专栏会改成跟原子弹大侠的《从0开始带你成为消息中间件实战高手》专栏一样的画图工具）



但是实际在使用MySQL的过程中，大家总会遇到这样那样的一些问题，比如死锁异常、SQL性能太差、异常报错，等等。

很多Java工程师在遇到MySQL数据库的一些问题时，一般都会上网搜索博客，然后自己尝试捣鼓着解决一下，最后解决了问题，自己可能也没搞明白里面的原理。

因此我们就是要带着大家去探索MySQL底层原理的方方面面，以及探索在解决MySQL各种生产实战问题的时候，如何基于MySQL底层原理去进行分析、排查和定位。

2、MySQL驱动到底是什么东西?

大家都知道，我们如果要在Java系统中去访问一个MySQL数据库，必须得在系统的依赖中加入一个MySQL驱动，有了这个MySQL驱动才能跟MySQL数据库建立连接，然后执行各种各样的SQL语句。

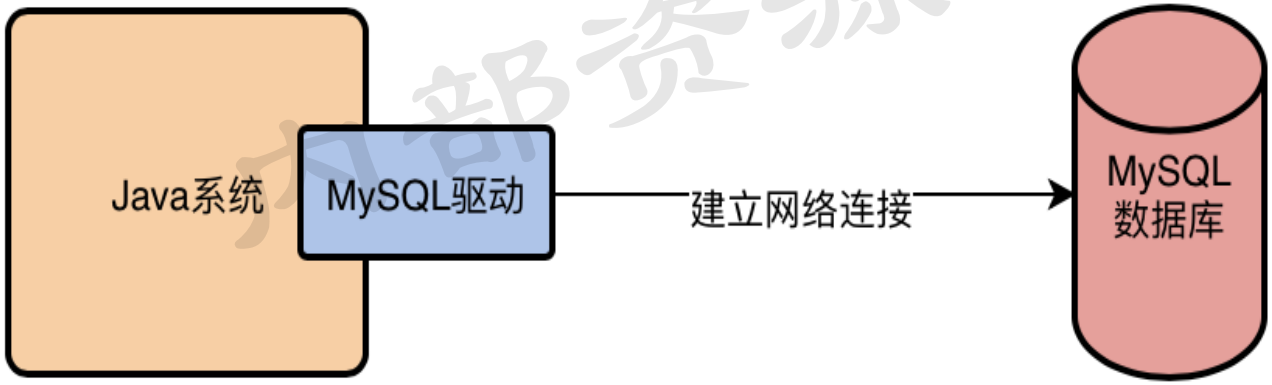
那么这个MySQL驱动到底是个什么东西?

我们先来看下面的一段maven配置，这段maven配置中就引入了一个MySQL驱动。这里的mysql-connector-java就是面向Java语言的MySQL驱动。

```
<dependency>
   <groupId>MySQL</groupId>
   <artifactId>mysql-connector-java</artifactId>
   <version>5.1.46</version>
</dependency>
```

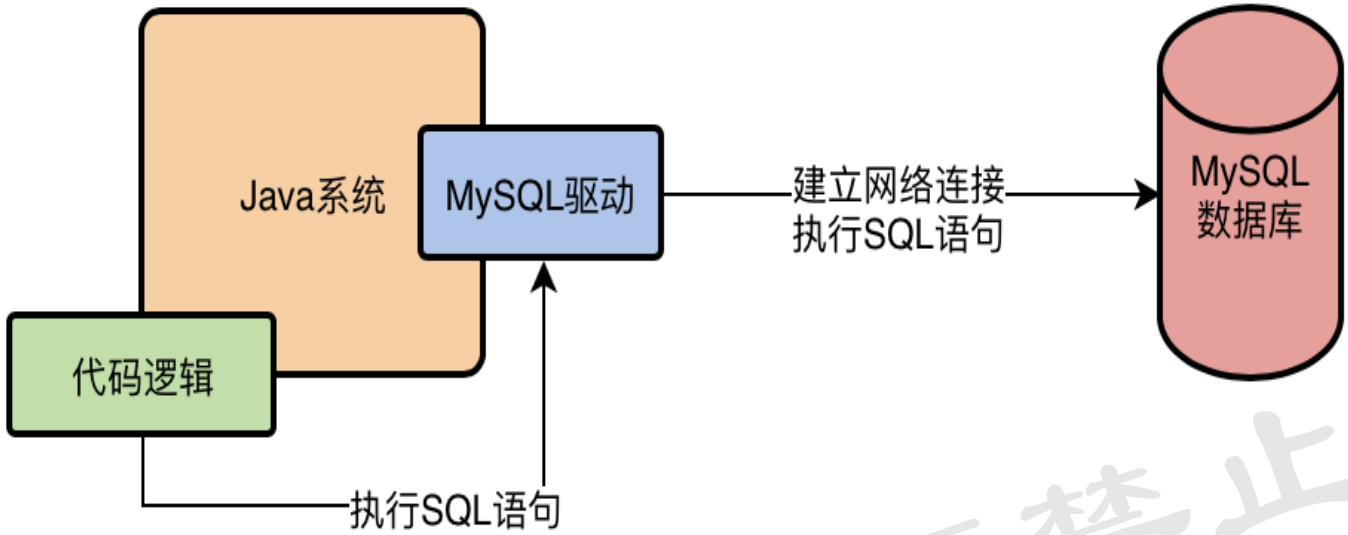
大家都知道，如果我们要访问数据库，必须得跟数据库建立一个网络连接，那么这个连接由谁来建立呢?

其实答案就是这个MySQL驱动，他会在底层跟数据库建立网络连接，有网络连接，接着才能去发送请求给数据库服务器！我们看下图。



然后当我们跟数据库之间有了网络连接之后，我们的Java代码才能基于这个连接去执行各种各样的增删改查SQL语句

我们看下图

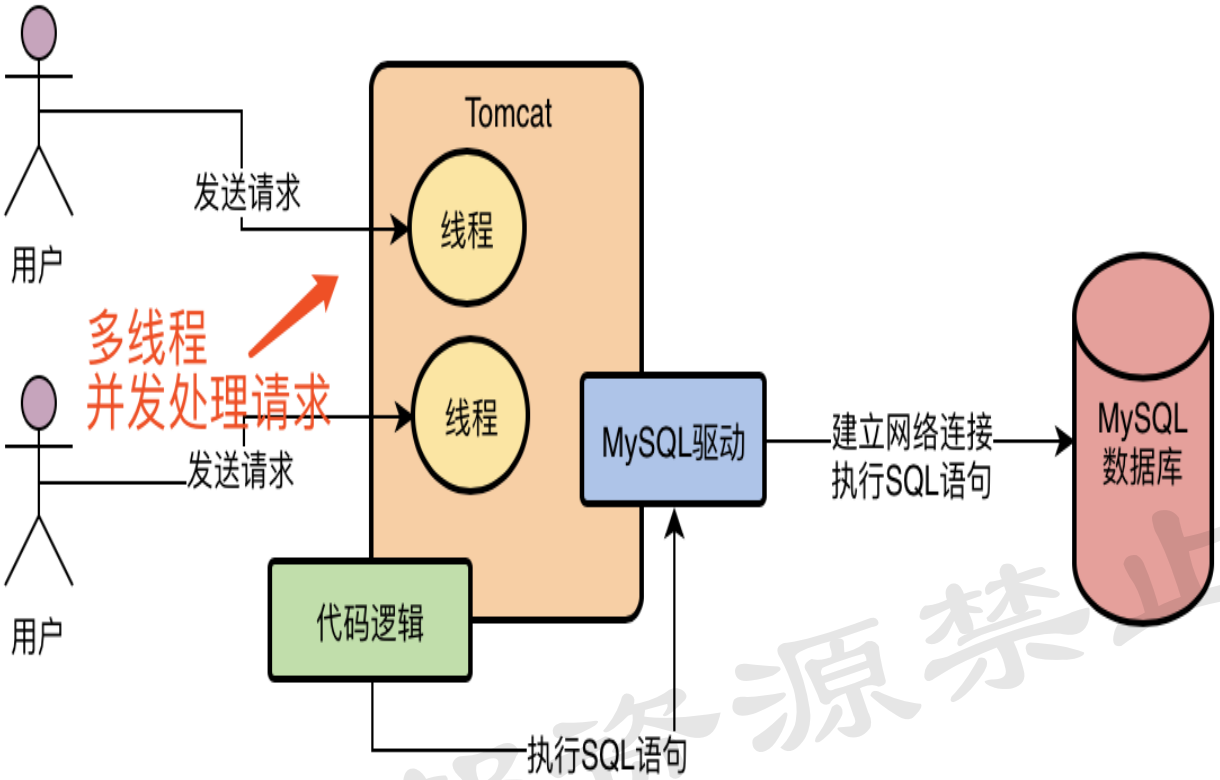


所以对于Java语言开发的系统，MySQL会提供Java版本的MySQL驱动，对于PHP、Perl、.NET、Python、Ruby等各种常见的编程语言，MySQL都会提供对应语言的MySQL驱动，让各种语言编写的系统通过MySQL驱动去访问数据库。

3、数据库连接池到底是用来干什么的?

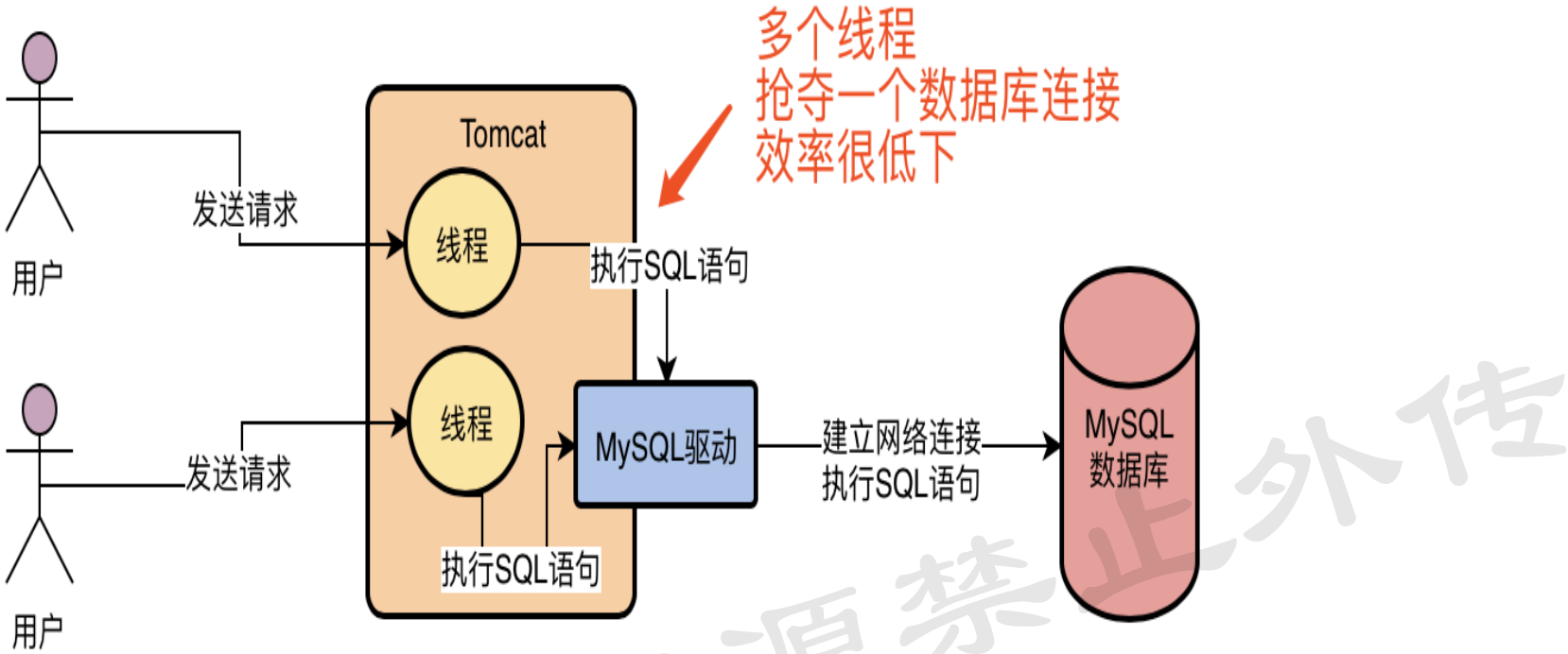
接着我们来思考一个问题，一个Java系统难道只会跟数据库建立一个连接吗?

这个肯定是不行的，因为我们要明白一个道理，假设我们用Java开发了一个Web系统，是部署在Tomcat中的，那么Tomcat本身肯定是有多个线程来并发的处理同时接收到的多个请求的，我们看下图。



这个时候，如果Tomcat中的多个线程并发处理多个请求的时候，都要去抢夺一个连接去访问数据库的话，那效率肯定是很低下的

我们看下面的图

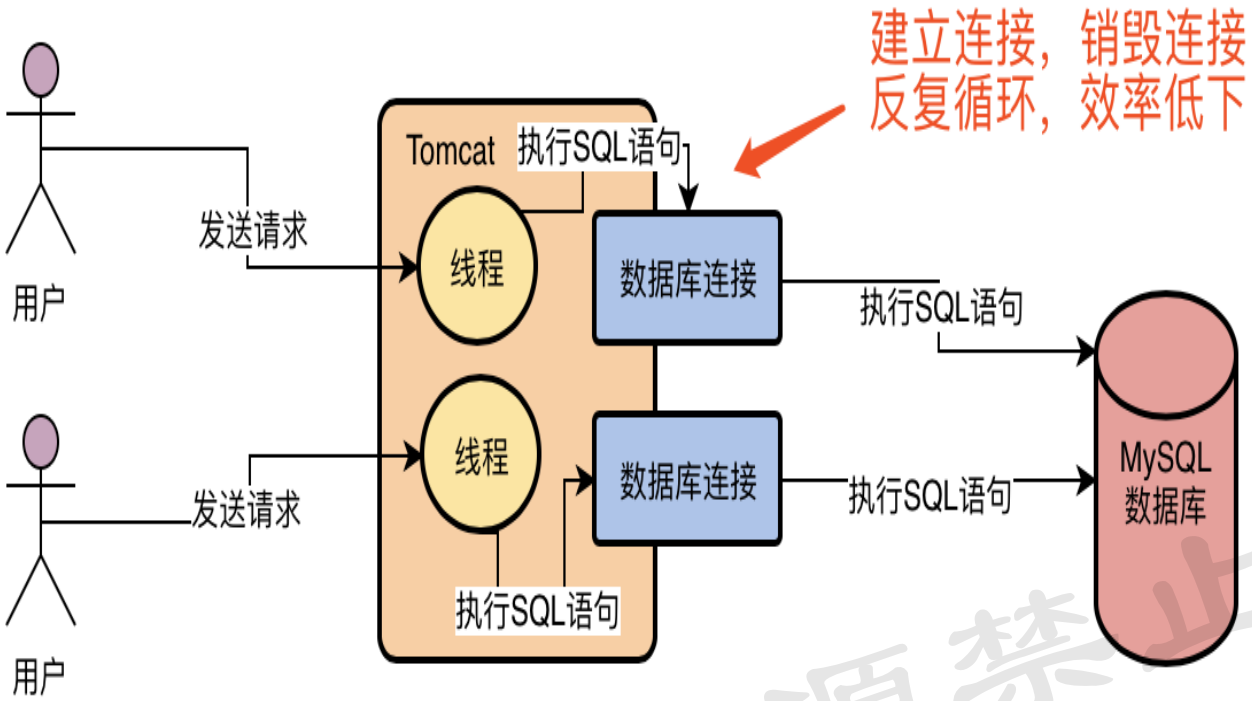


那么如果Tomcat中的每个线程在每次访问数据库的时候，都基于MySQL驱动去创建一个数据库连接，然后执行SQL语句，然后执行完之后再销毁这个数据库连接，这样行不行呢？

可能Tomcat中上百个线程会并发的频繁创建数据库连接，执行SQL语句，然后频繁的销毁数据库连接。

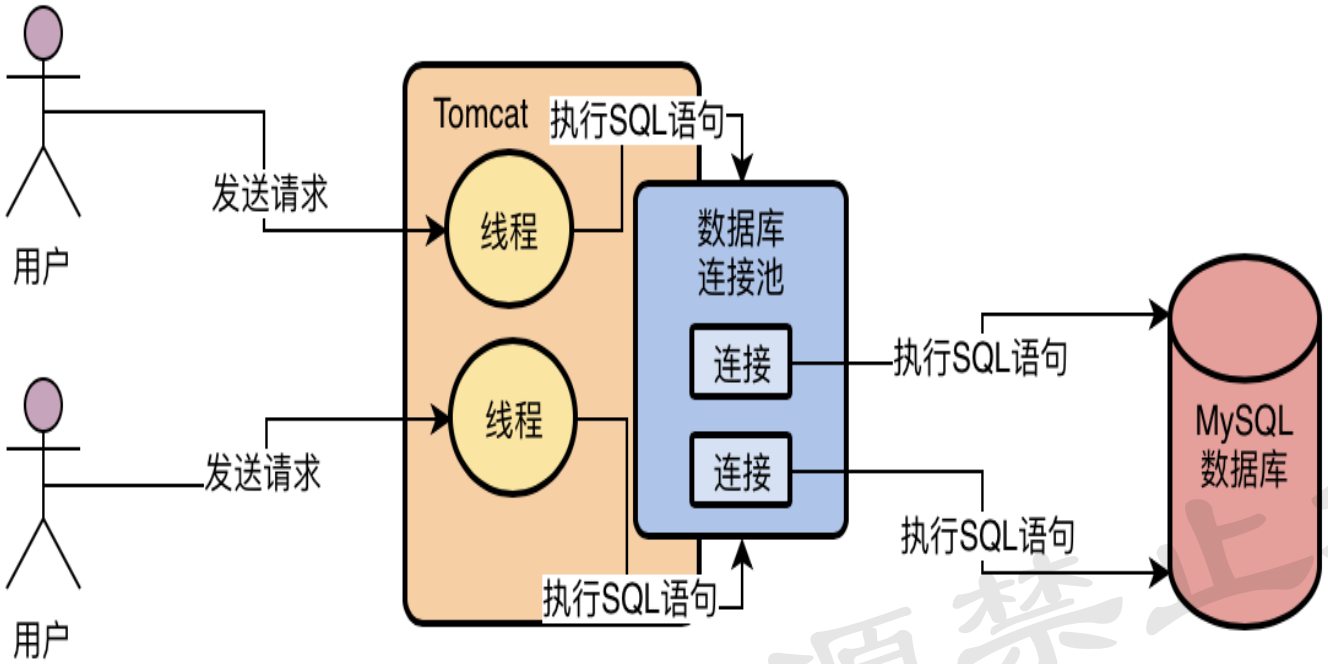
上述这个过程反复循环执行，大家觉得可行吗？

这也是非常不好的，因为每次建立一个数据库连接都很耗时，好不容易建立好了连接，执行完了SQL语句，你还把数据库连接给销毁了，下一次再重新建立数据库连接，那肯定是效率很低下的！如下图。



所以一般我们必须要使用一个数据库连接池，也就是说在一个池子里维持多个数据库连接，让多个线程使用里面的不同的数据库连接去执行SQL语句，然后执行完SQL语句之后，不要销毁这个数据库连接，而是把连接放回池子里，后续还可以继续使用。

基于这样的一个数据库连接池的机制，就可以解决多个线程并发的使用多个数据库连接去执行SQL语句的问题，而且还避免了数据库连接使用完之后就销毁的问题，我们看下图的说明。



常见的数据库连接池有DBCP，C3P0，Druid，等等，大家如果有兴趣的话，可以去搜索一下数据库连接池的使用例子和代码，甚至探索一下数据库连接池的底层原理，但这个不是我们专栏的重点，我们就不会拓展了。

毕竟我们专栏主要还是会专注讲解MySQL数据库本身的内容，只不过在开头的时候，需要大家对Java系统与数据库的交互方式有一个了解。

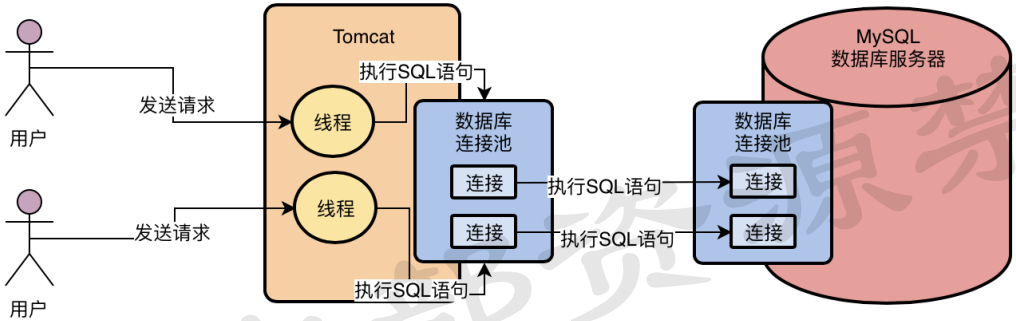
其实不光是Java系统，如果你是一个Python、Ruby、.NET、PHP的程序员，这个系统与数据库的交互本质都是一样的，都是基于数据库连接池去与数据库进行交互。

4、MySQL数据库的连接池是用来干什么的?

现在我们已经知道，我们任何一个系统都会有一个数据库连接池去访问数据库，也就是说这个系统会有多个数据库连接，供多线程并发的使用。同时我们可能会有多个系统同时去访问一个数据库，这都是有可能的。

所以当我们把目光转移到MySQL的时候，我们要来思考一个问题，那就是肯定会有很多系统要与MySQL数据库建立很多个连接，那么MySQL也必然要维护与系统之间的多个连接，所以**MySQL架构体系中的第一个环节，就是连接池。**

我们看下面的图，实际上MySQL中的连接池就是维护了与系统之间的多个数据库连接。除此之外，你的系统每次跟MySQL建立连接的时候，还会根据你传递过来的账号和密码，进行账号密码的验证，库表权限的验证。



5、小作业：自己试一试写代码建立MySQL连接

当我们看完今天的内容后，大家可以用自己工作中经常使用的编程语言，来写一下跟MySQL建立连接的代码，想必写完之后，再对照今天的内容，感受会更深一些。

另外，大家可以基于数据库连接池框架，去写一下对应的代码例子，感受一下你建立多个数据库连接让多个线程并发访问数据库的效果。

End

狸猫技术窝精品专栏及课程推荐：

- 《从零开始带你成为消息中间件实战高手》
- 《21天互联网Java进阶面试训练营》（分布式篇）
- 《互联网Java工程师面试突击》（第1季）
- 《互联网Java工程师面试突击》（第3季）
- 《从零开始带你成为JVM实战高手》

图文 02 为了执行SQL语句，你知道MySQL用了什么样的架构设计吗？ 手机观看

2348 人次阅读 2020-01-14 07:00:00

详情 评论

为了执行SQL语句，你知道MySQL用了什么样的架构设计吗？

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

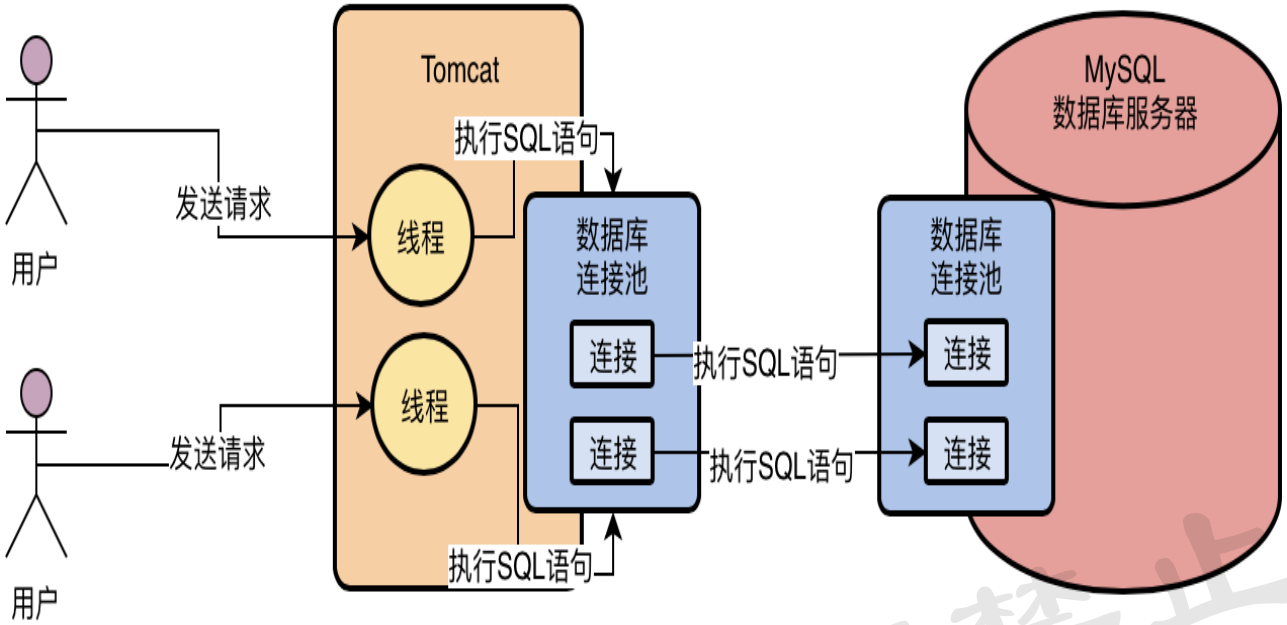
如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

1、把MySQL当个黑盒子一样执行SQL语句

上一讲我们已经说到，我们的系统采用数据库连接池的方式去并发访问数据库，然后数据库自己其实也会维护一个连接池，其中管理了各种系统跟这台数据库服务器建立的所有连接

我们先看下图回顾一下



当我们的系统只要能从数据库连接池获取到一个数据库连接之后，我们就可以执行增删改查的SQL语句了

从上图其实我们就可以看到，我们可以通过数据库连接把要执行的SQL语句发送给MySQL数据库。

然后呢？大部分同学了解到这个程度就停下来了，然后大家觉得要关注的可能主要就是数据库里的表结构，建了哪些索引，然后就按照SQL语法去编写增删改查SQL语句，把MySQL当个黑盒子去执行SQL语句就可以了。

我们只知道执行了insert语句之后，在表里会多出来一条数据；执行了update语句之后，会对表里的数据进行更改；执行了delete语句之后，会把表里的一条数据删除掉；执行了select语句之后，会从表里查询一些数据出来。

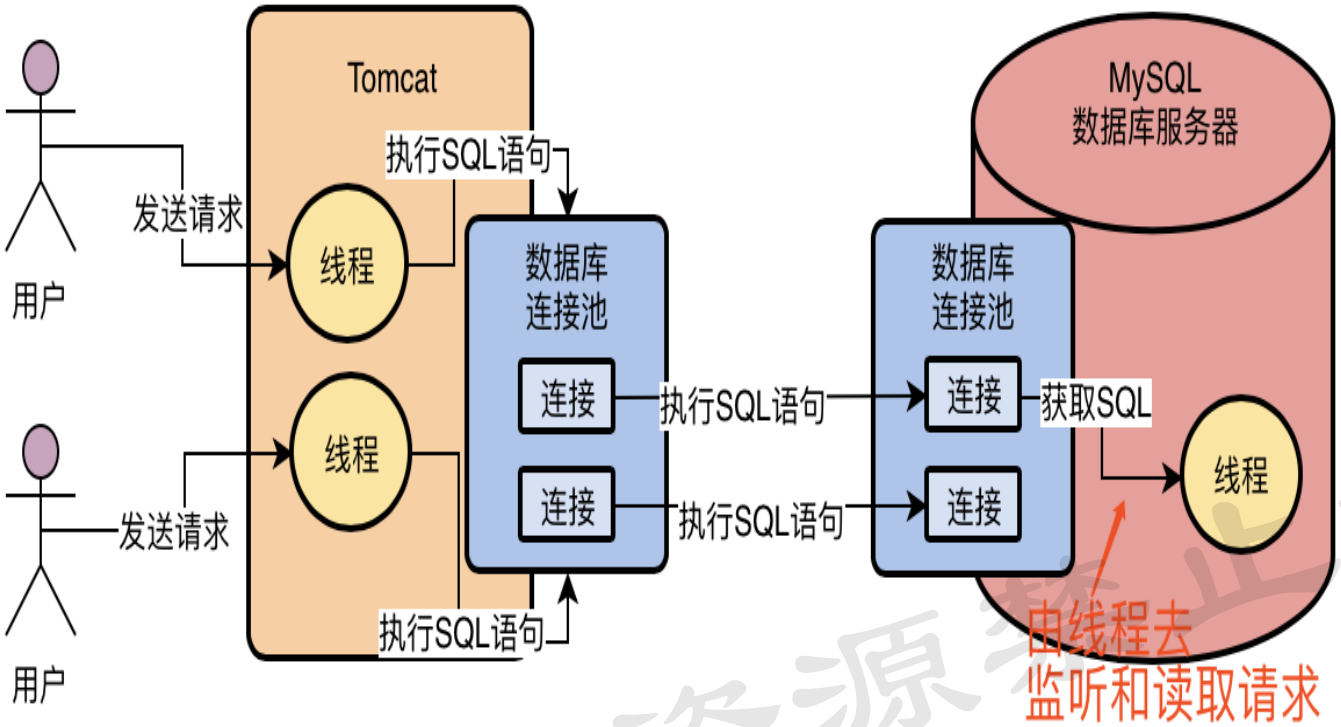
如果语句性能有点差？没关系，在表里建几个索引就可以了！可能这就是目前行业内很多工程师对数据库的一个认知，完全当他是个黑盒子，来建表以及执行SQL语句。

但是大家既然跟着我开始学习了，从现在开始就要打破这种把数据库当黑盒子的认知程度，要深入底层，去探索数据库的工作原理以及生产问题的优化手段！

2、一个不变的原则：网络连接必须让线程来处理

现在假设我们的数据库服务器的连接池中的某个连接接收到了网络请求，假设就是一条SQL语句，那么大家先思考一个问题，谁负责从这个连接中去监听网络请求？谁负责从网络连接里把请求数据读取出来？

我想很多人恐怕都没思考过这个问题，但是如果大家对计算机基础知识有一个简单了解的话，应该或多或少知道一点，那就是网络连接必须得分配给一个线程去进行处理，由一个线程来监听请求以及读取请求数据，比如从网络连接中读取和解析出来一条我们的系统发送过去的SQL语句，如下图所示：



3、SQL接口：负责处理接收到的SQL语句

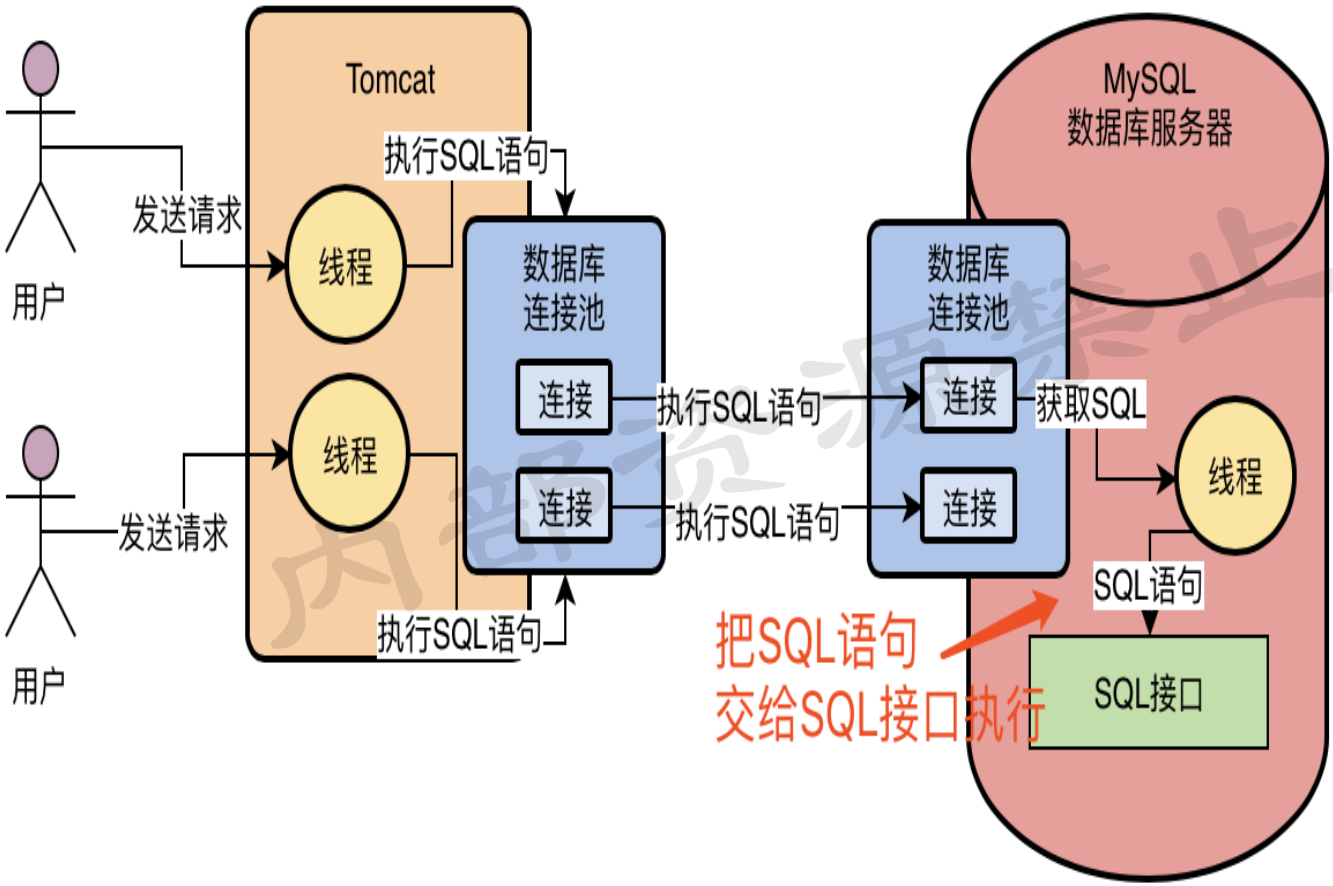
接着我们来思考一下，当MySQL内部的工作线程从一个网络连接中读取出来一个SQL语句之后，此时会如何来执行这个SQL语句呢?

其实SQL是一项伟大的发明，他发明了简单易用的数据读写的语法和模型，哪怕是个产品经理，或者是运营专员，甚至是销售专员，即使他不会技术，他也能轻松学会使用SQL语句。

但如果你要去执行这个SQL语句，去完成底层数据的增删改查，那这就是一项极度复杂的任务了！

所以MySQL内部首先提供了一个组件，就是SQL接口（SQL Interface），他是一套执行SQL语句的接口，专门用于执行我们发送给MySQL的那些增删改查的SQL语句

因此MySQL的工作线程接收到SQL语句之后，就会转交给SQL接口去执行，如下图。



4、查询解析器：让MySQL能看懂SQL语句

接着下一个问题来了，SQL接口怎么执行SQL语句呢？你直接把SQL语句交给MySQL，他能看懂和理解这些SQL语句吗？

比如我们来举一个例子，现在我们有这么一个SQL语句：

```
select id,name,age from users where id=1
```

这个SQL语句，我们用人脑是直接就可以处理一下，只要懂SQL语法的人，立马大家就知道他是什么意思，但是MySQL自己本身也是一个系统，是一个数据库管理系统，他没法直接理解这些SQL语句！

所以此时有一个关键的组件要出场了，那就是**查询解析器**

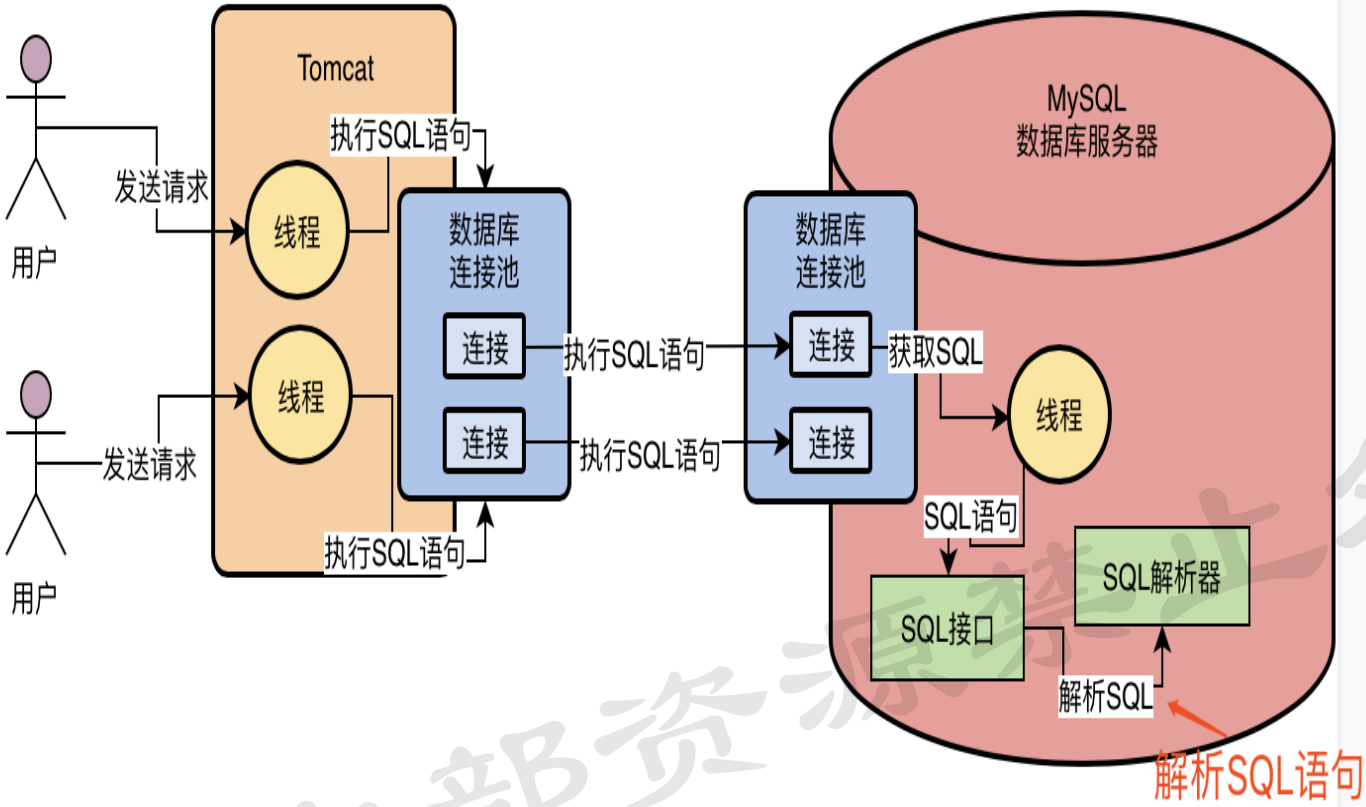
这个查询解析器（Parser）就是负责对SQL语句进行解析的，比如对上面那个SQL语句进行一下拆解，拆解成以下几个部分：

我们现在要从“users”表里查询数据

查询“id”字段的值等于1的那行数据

对查出来的那行数据要提取里面的“id,name,age”三个字段。

所谓的SQL解析，就是按照既定的SQL语法，对我们按照SQL语法规则编写的SQL语句进行解析，然后理解这个SQL语句要干什么事情，如下图所示：



5、查询优化器：选择最优的查询路径

当我们通过解析器理解了SQL语句要干什么之后，接着会找查询优化器（Optimizer）来选择一个最优的查询路径。

可能有同学这里就不太理解什么是最优的查询路径了，这个看起来确实很抽象，当然，这个查询优化器的工作原理，后续将会是我们分析的重点，大家现在不用去纠结他的原理。

但是我们可以用一个极为通俗简单的例子，让大家理解一下所谓的最优查询路径是什么。

就用我们刚才讲的那个例子好了，我们现在理解了一个SQL想要干这么一个事儿：我们现在要从“users”表里查询数据，查询“id”字段的值等于1的那行数据，对查出来的那行数据要提取里面的“id,name,age”三个字段。

事是明白了，但是到底应该怎么来实现呢？

你看，要完成这个事儿我们有以下几个查询路径（**纯属用于大家理解的例子，不代表真实的MySQL原理，但是通过这个例子，大家肯定能理解所谓最优查询路径的意思**）：

直接定位到“users”表中的“id”字段等于1的一行数据，然后查出来那行数据的“id,name,age”三个字段的值就可以了

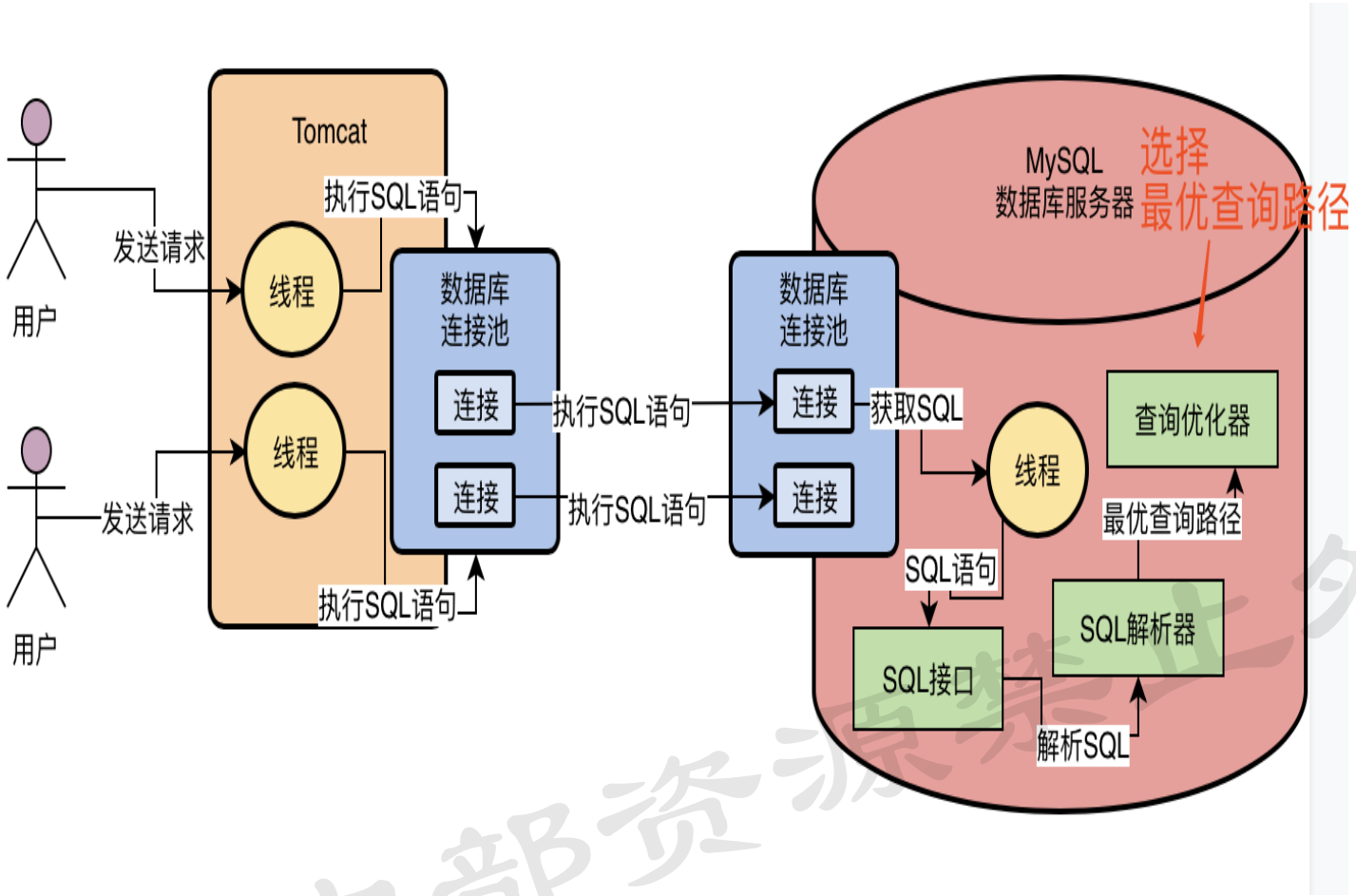
先把“users”表中的每一行数据的“id,name,age”三个字段的值都查出来，然后从这批数据里过滤出来“id”字段等于1的那行数据的“id,name,age”三个字段

上面这就是一个最简单的SQL语句的两种实现路径，其实我们会发现，要完成这个SQL语句的目标，两个路径都可以做到，但是哪一种更好呢？显然感觉上是第一种查询路径更好一些。

所以查询优化器大概就是干这个的，他会针对你编写的几十行、几百行甚至上千行的复杂SQL语句生成查询路径树，然后从里面选择一条最优的查询路径出来。

相当于他会告诉你，你应该按照一个什么样的步骤和顺序，去执行哪些操作，然后一步一步的把SQL语句就给完成了。

我们来一起看看下面的图：



6、调用存储引擎接口，真正执行SQL语句

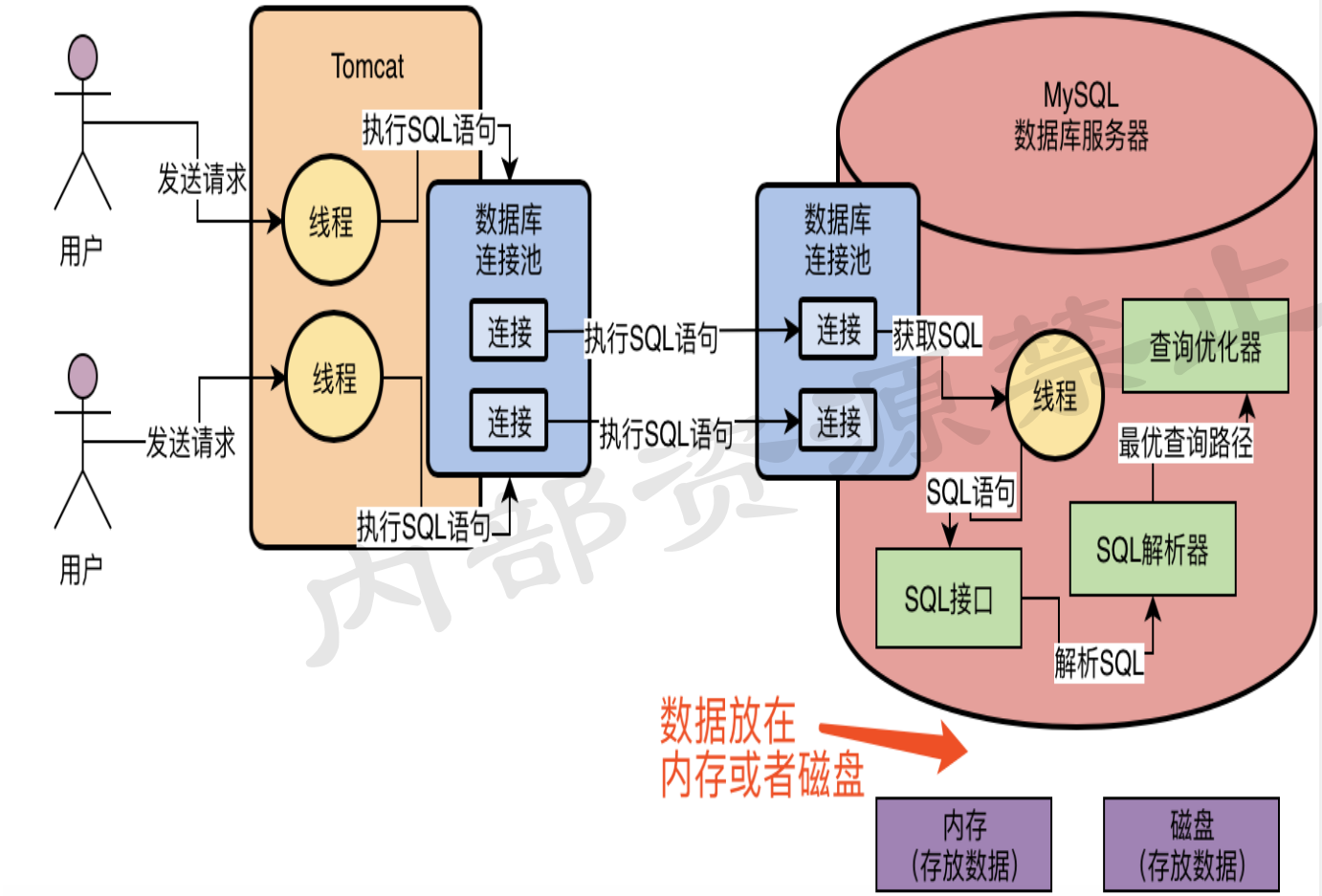
最后一步，就是把查询优化器选择的最优查询路径，也就是你到底应该按照一个什么样的顺序和步骤去执行这个SQL语句的计划，把这个计划交给底层的存储引擎去真正的执行。这个存储引擎是MySQL的架构设计中很有特色的一个环节。

不知道大家是否思考过，真正在执行SQL语句的时候，要不然是更新数据，要不然是查询数据，那么数据你觉得存放在哪里？

说白了，数据库也不是什么神秘莫测的东西，你可以把他理解为本身就是一个类似你平时写的图书馆管理系统、电信计费系统、电商订单系统之类的系统罢了。

数据库自己就是一个编程语言写出来的系统而已，然后启动之后也是一个进程，执行他里面的各种代码，也就是我们上面所说的那些东西。所以对数据库而言，我们的数据要不然是放在内存里，要不然是放在磁盘文件里，没什么特殊的地方！

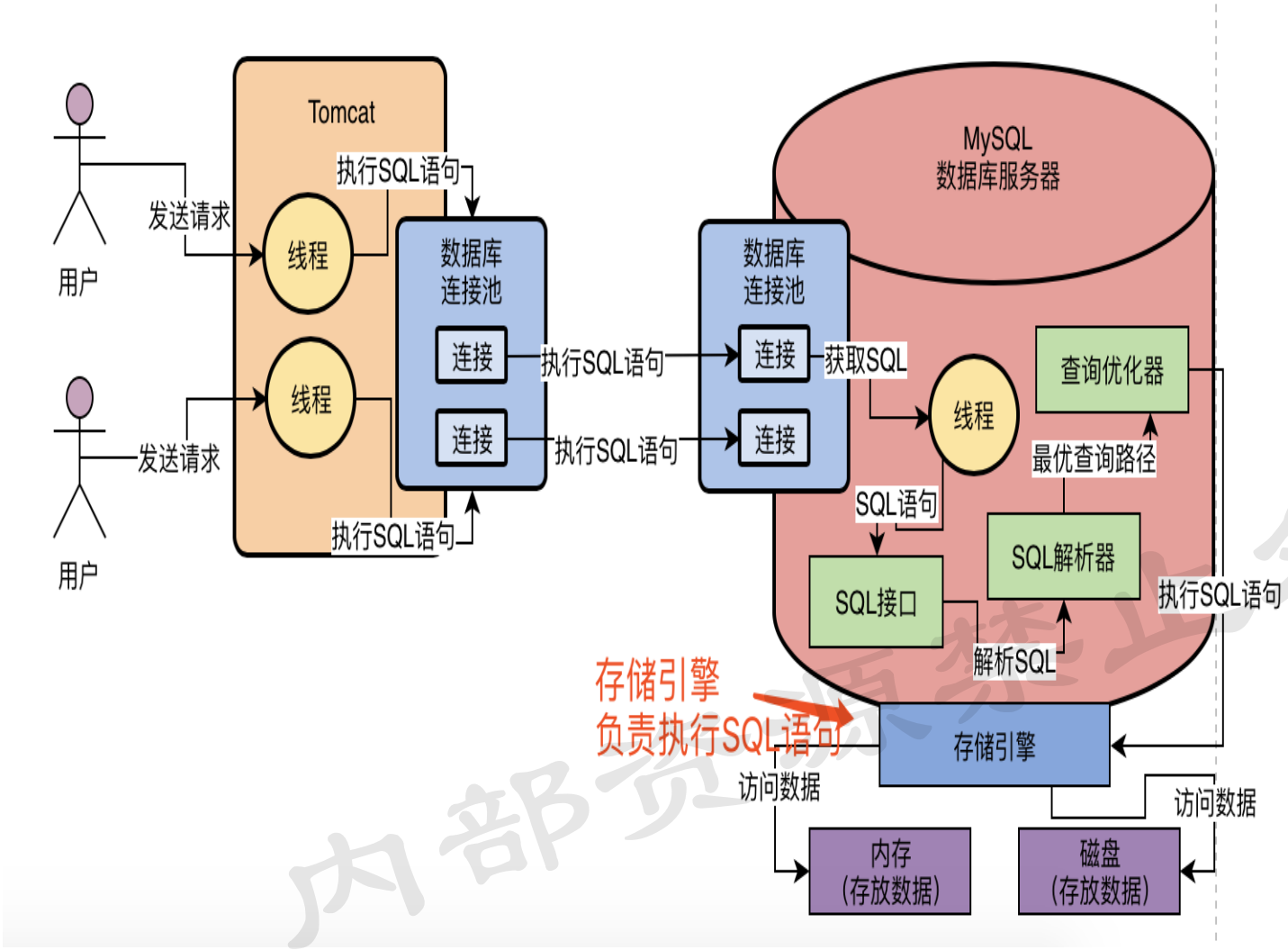
所以我们来思考一下，假设我们的数据有的存放在内存里，有的存放在磁盘文件里，如下图所示。



那么现在问题来了，我们已经知道一个SQL语句要如何执行了，但是我们现在怎么知道哪些数据在内存里？哪些数据在磁盘里？我们执行的时候是更新内存的数据？还是更新磁盘的数据？我们如果更新磁盘的数据，是先查询哪个磁盘文件，再更新哪个磁盘文件？

是不是感觉一头雾水

所以这个时候就需要存储引擎了，存储引擎其实就是执行SQL语句的，他会按照一定的步骤去查询内存缓存数据，更新磁盘数据，查询磁盘数据，等等，执行诸如此类的一系列的操作，如下图所示。



MySQL的架构设计中，SQL接口、SQL解析器、查询优化器其实都是通用的，他就是一套组件而已。

但是存储引擎的话，他是支持各种各样的存储引擎的，比如我们常见的InnoDB、MyISAM、Memory等等，我们是可以选择使用哪种存储引擎来负责具体的SQL语句执行的。

当然现在MySQL一般都是使用InnoDB存储引擎的，至于存储引擎的原理，后续我们也会深入一步一步分析，大家不必着急。

7、执行器：根据执行计划调用存储引擎的接口

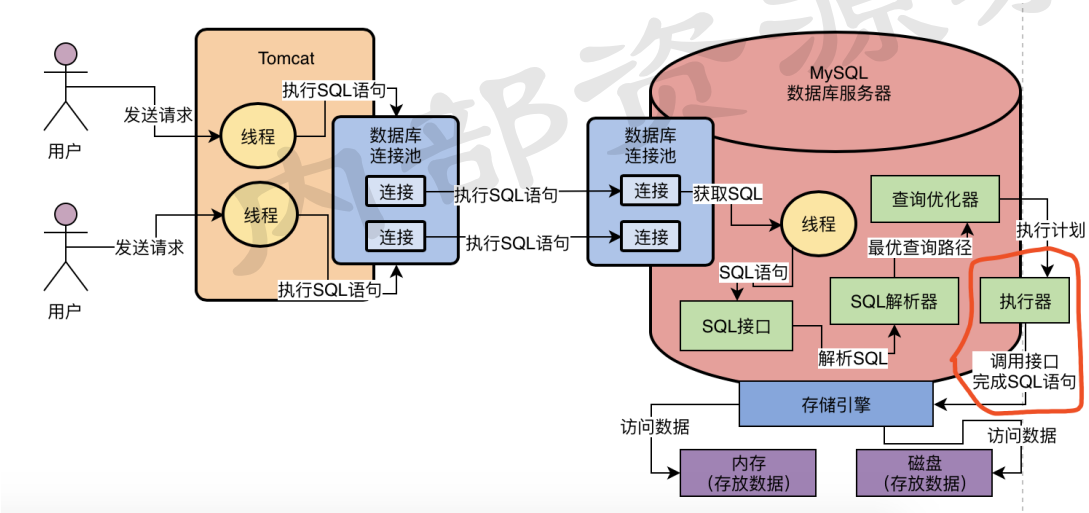
那么看完存储引擎之后，我们回过头来思考一个问题，存储引擎可以帮助我们去访问内存以及磁盘上的数据，那么是谁来调用存储引擎的接口呢?

其实我们现在还漏了一个执行器的概念，这个执行器会根据优化器选择的执行方案，去调用存储引擎的接口按照一定的顺序和步骤，就把SQL语句的逻辑给执行了。

举个例子，比如执行器可能会先调用存储引擎的一个接口，去获取“users”表中的第一行数据，然后判断一下这个数据的“id”字段的值是否等于我们期望的一个值，如果不是的话，那就继续调用存储引擎的接口，去获取“users”表的下一行数据。

就是基于上述的思路，**执行器就会去根据我们的优化器生成的一套执行计划，然后不停的调用存储引擎的各种接口去完成SQL语句的执行计划**，大致就是不停的更新或者提取一些数据出来

我们看下图的示意



8、小思考题：打开脑洞，你觉得不同的存储引擎是用来干什么的?

今天给大家留一个小的思考题，就是你先别管MySQL有哪些存储引擎，你就从业务场景来出发考虑，有的场景可能是高并发的更新，有的场景可能是大规模数据查询，有的场景可能是允许丢失数据的

那么你觉得如果让你来设计存储引擎，你觉得应该有哪些存储引擎，分别适用于什么场景？

希望大家在评论区留言，踊跃的思考和回复。

End

图文 03 用一次数据更新流程，初步了解InnoDB存储引擎的架构设计

手机观看

详情 评论

用一次数据更新流程，初步了解InnoDB存储引擎的架构设计

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

1、更新语句在MySQL中是如何执行的?

之前我们已经分析了MySQL架构上的整体设计原理，现在对一条SQL语句从我们的系统层面发送到MySQL中，然后一步一步执行这条SQL的流程，都有了一个整体的了解。

我们已经知道了，MySQL最常用的就是InnoDB存储引擎，那么我们今天借助一条更新语句的执行，来初步的了解一下InnoDB存储引擎的架构设计。

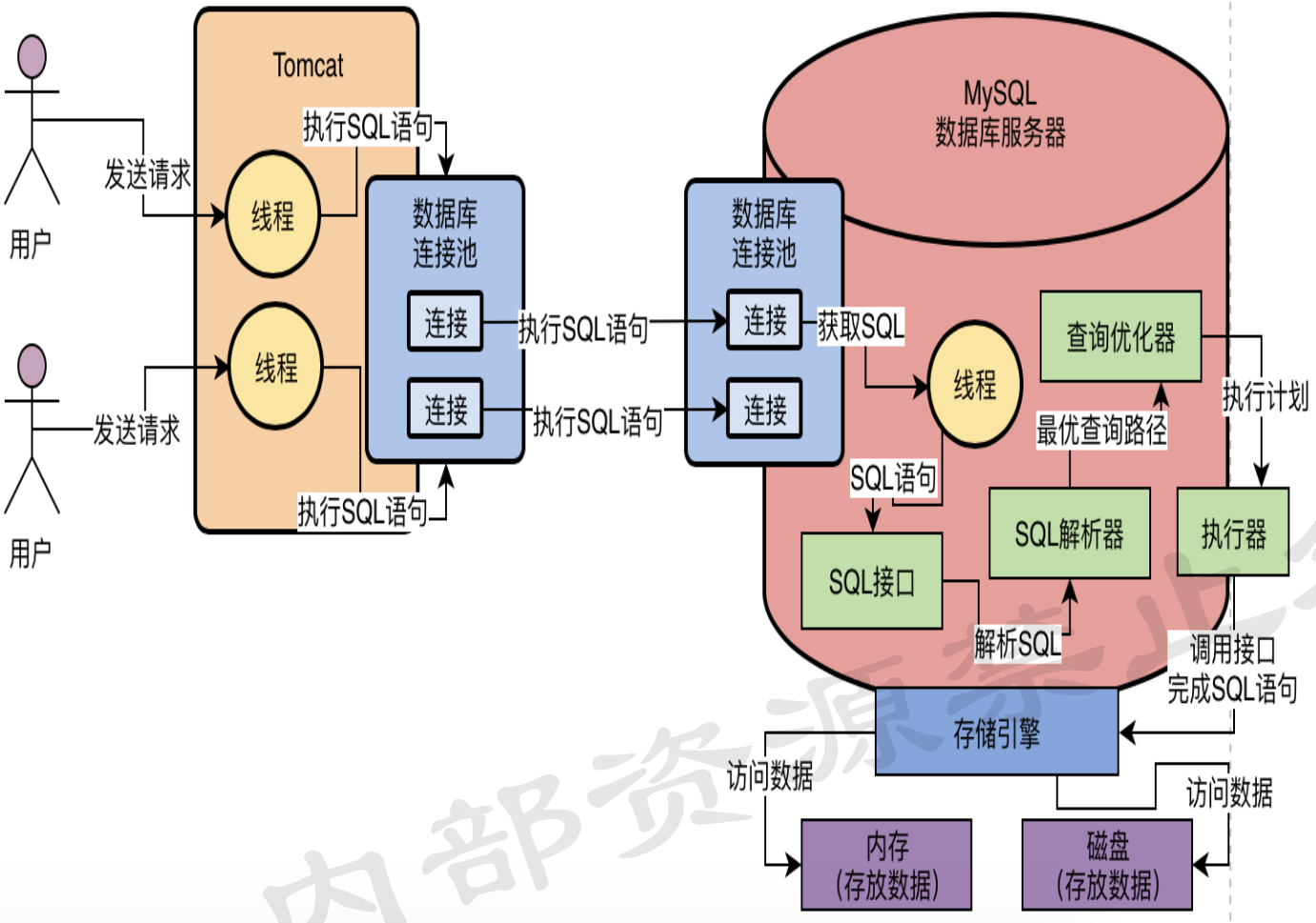
首先假设我们有一条SQL语句是这样的：

```
update users set name='xxx' where id=10
```

那么我们先想一下这条SQL语句是如何执行的？

首先肯定是我们的系统通过一个数据库连接发送到了MySQL上，然后肯定会经过SQL接口、解析器、优化器、执行器几个环节，解析SQL语句，生成执行计划，接着去由执行器负责这个计划的执行，调用InnoDB存储引擎的接口去执行。

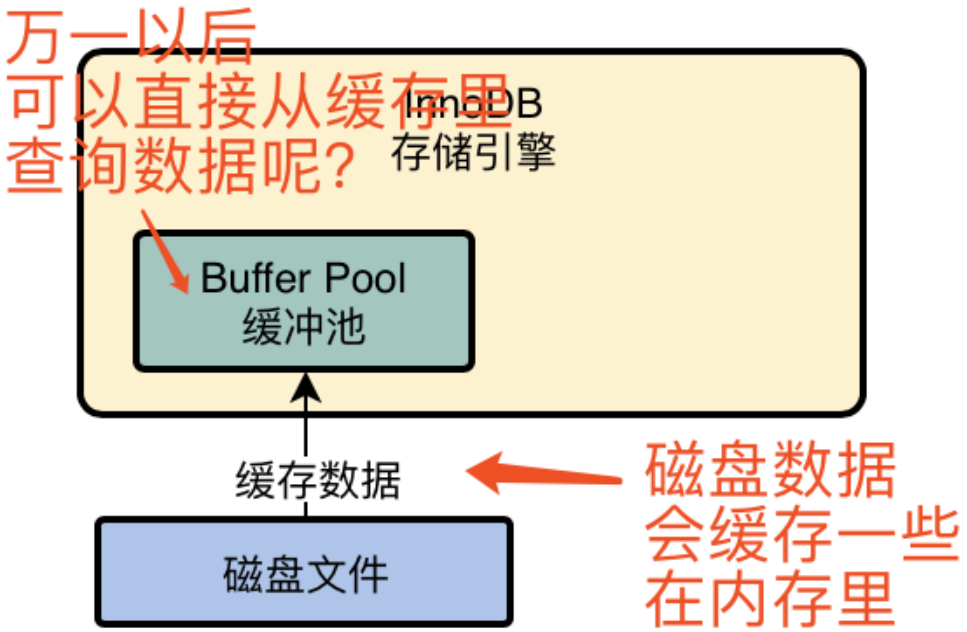
所以先看下图，大致还是会走下图的这个流程



今天我们就来探索一下这个存储引擎里的架构设计，以及如何基于存储引擎完成一条更新语句的执行

2、InnoDB的重要内存结构：缓冲池

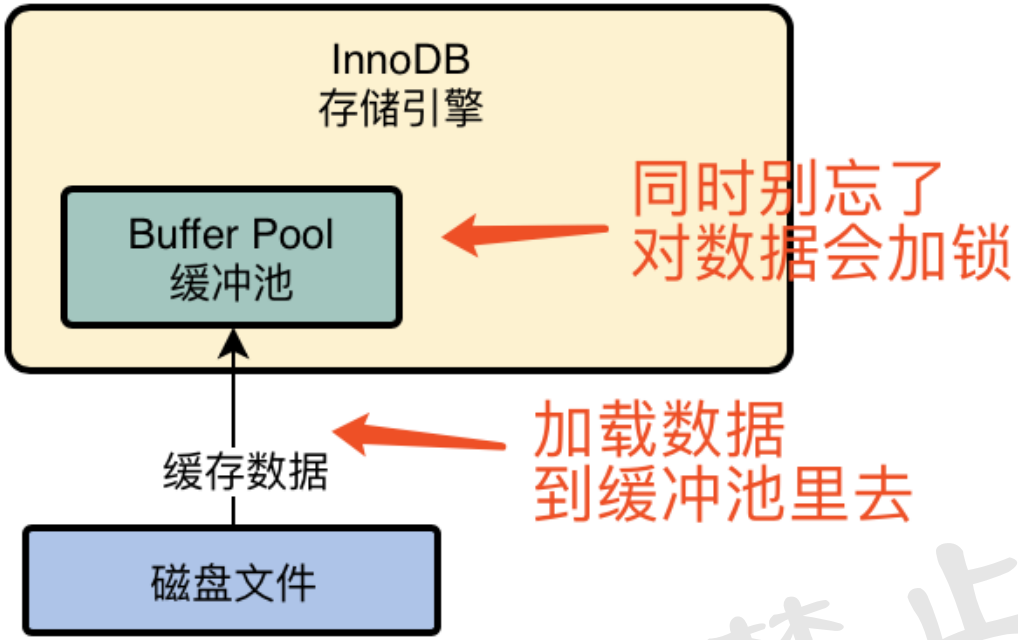
InnoDB存储引擎中有一个非常重要的放在内存里的组件，就是缓冲池（Buffer Pool），这里面会缓存很多的数据，以便于以后在查询的时候，万一你要是内存缓冲池里有数据，就可以不用去查磁盘了，我们看下图。



所以当我们的InnoDB存储引擎要执行更新语句的时候，比如对“id=10”这一行数据，他其实会先将“id=10”这一行数据看看是否在缓冲池里，如果不在的话，那么会直接从磁盘里加载到缓冲池里来，而且接着还会对这行记录加独占锁。

因为我们想一下，在我们更新“id=10”这一行数据的时候，肯定是不允许别人同时更新的，所以必须要对这行记录加独占锁

至于锁的详细分析，我们后续也会有，大家不用着急，在这里先初步了解即可，我们看下面的图

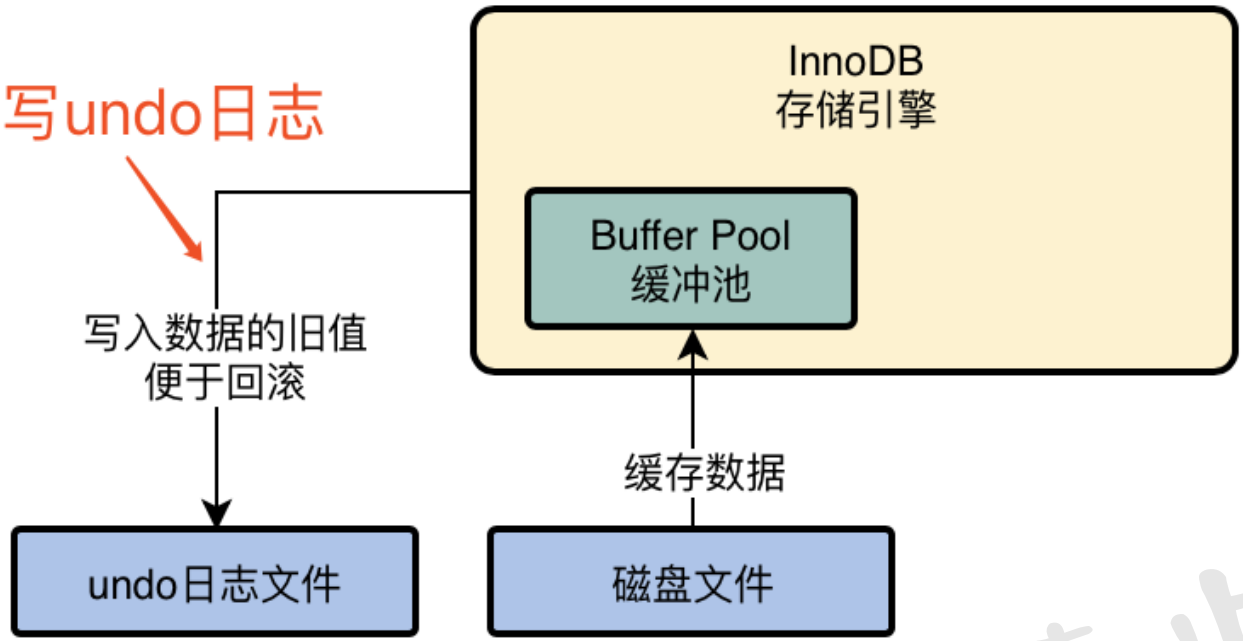


3、undo日志文件：如何让你更新的数据可以回滚?

接着下一步，假设“id=10”这行数据的name原来是“zhangsan”，现在我们要更新为“xxx”，那么此时我们得先把要更新的原来的值“zhangsan”和“id=10”这些信息，写入到undo日志文件中去。

其实稍微对数据库 有一点了解的同学都应该知道，如果我们执行一个更新语句，要是他是在一个事务里的话，那么事务提交之前我们都是可以对数据进行回滚的，也就是把你更新为“xxx”的值回滚到之前的“zhangsan”去。

所以为了考虑到未来可能要回滚数据的需要，这里会把你更新前的值写入undo日志文件，我们看下图。



4、更新buffer pool中的缓存数据

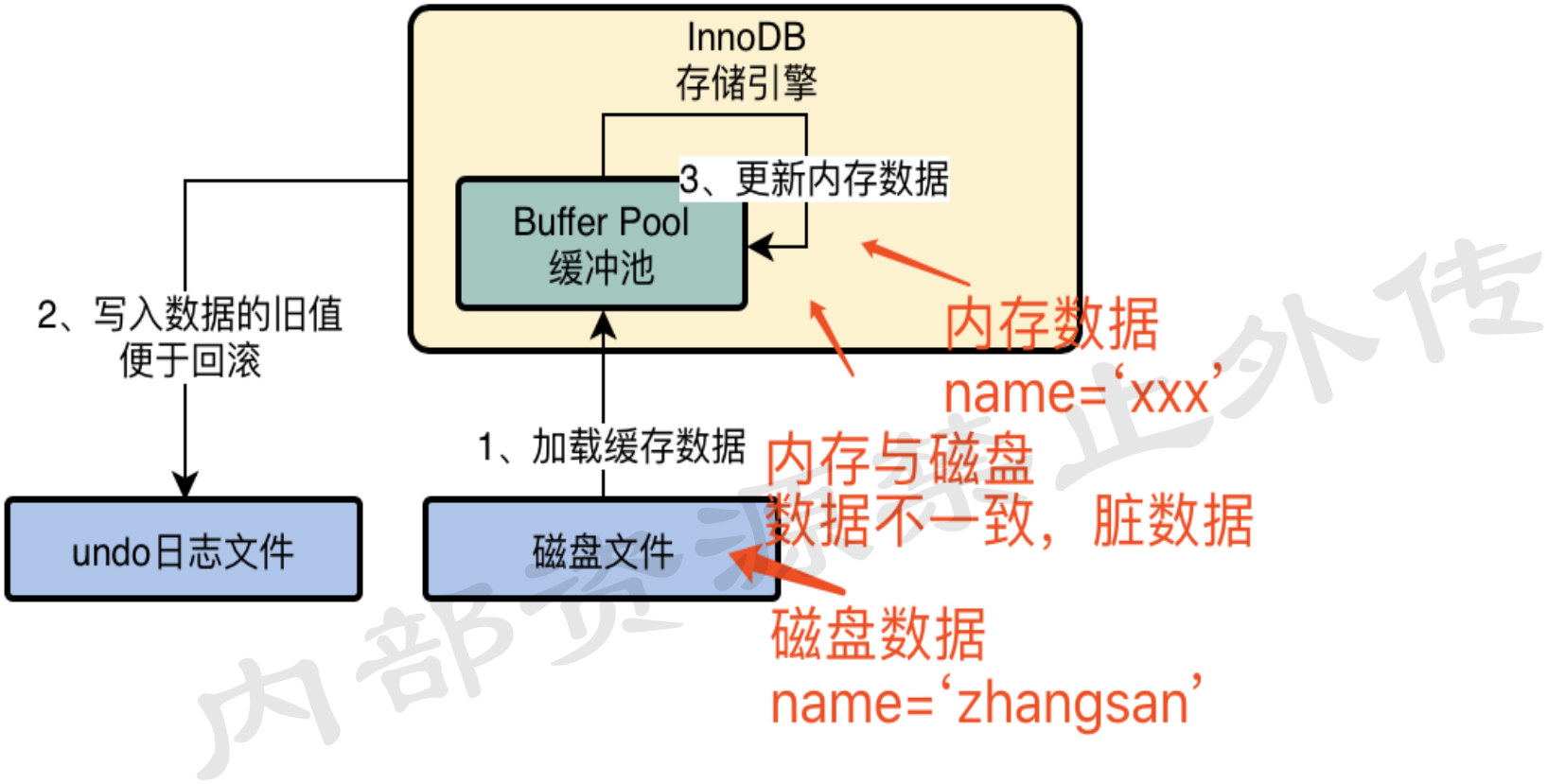
当我们把要更新的那行记录从磁盘文件加载到缓冲池，同时对他加锁之后，而且还把更新前的旧值写入undo日志文件之后，我们就可以正式开始更新这行记录了，更新的时候，先是会更新缓冲池中的记录，此时这个数据就是脏数据了。

这里所谓的更新内存缓冲池里的数据，意思就是把内存里的“id=10”这行数据的name字段修改为“xxx”

那么为什么说此时这行数据就是脏数据了呢?

因为这个时候磁盘上“id=10”这行数据的name字段还是“zhangsan”，但是内存里这行数据已经被修改了，所以就会叫他是脏数据。

我们看下图，我同时把几个步骤的序号标记出来了。



5、Redo Log Buffer：万一系统宕机，如何避免数据丢失？

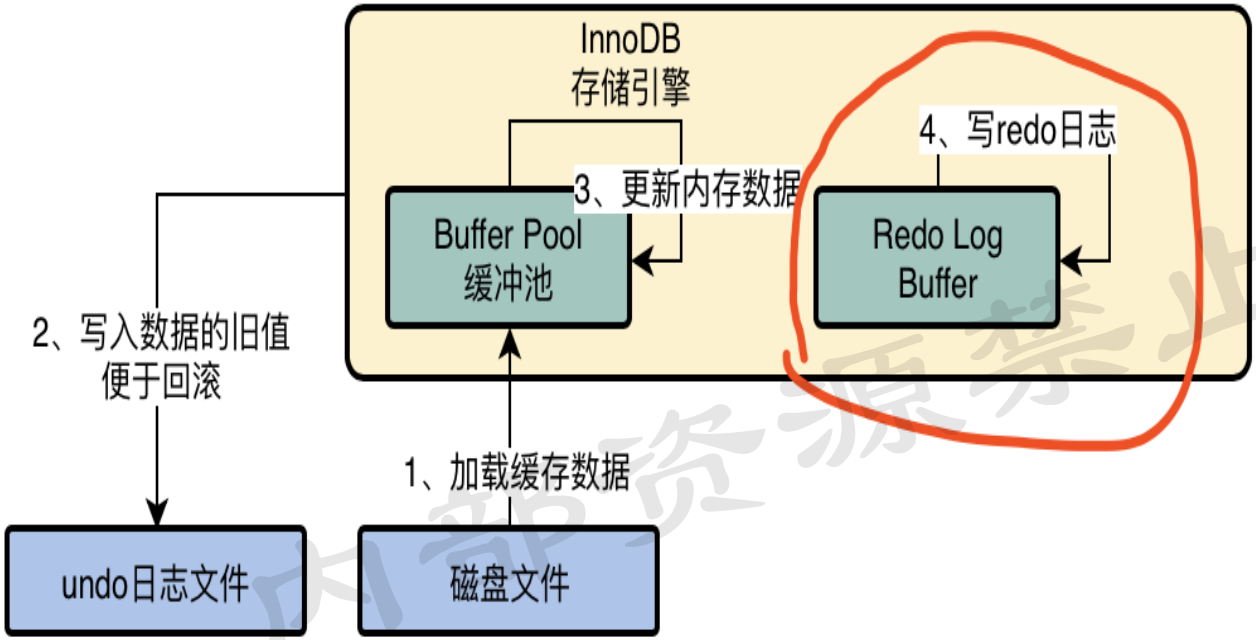
接着我们来思考一个问题，按照上图的说明，现在已经把内存里的数据进行了修改，但是磁盘上的数据还没修改

那么此时万一MySQL所在的机器宕机了，必然会导致内存里修改过的数据丢失，这可怎么办呢？

这个时候，就必须要把对内存所做的修改写入到一个**Redo Log Buffer**里去，这也是内存里的一个缓冲区，是用来存放redo日志的

所谓的redo日志，就是记录下来你对数据做了什么修改，比如对“id=10这行记录修改了name字段的值为xxx”，这就是一个日志。

我们先看下图的示意



这个redo日志其实是用来在MySQL突然宕机的时候，用来恢复你更新过的数据的，但是我们现在还没法直接讲解redo是如何使用的，毕竟现在redo日志还仅仅停留在内存缓冲里

大家稍安勿躁，继续往下看

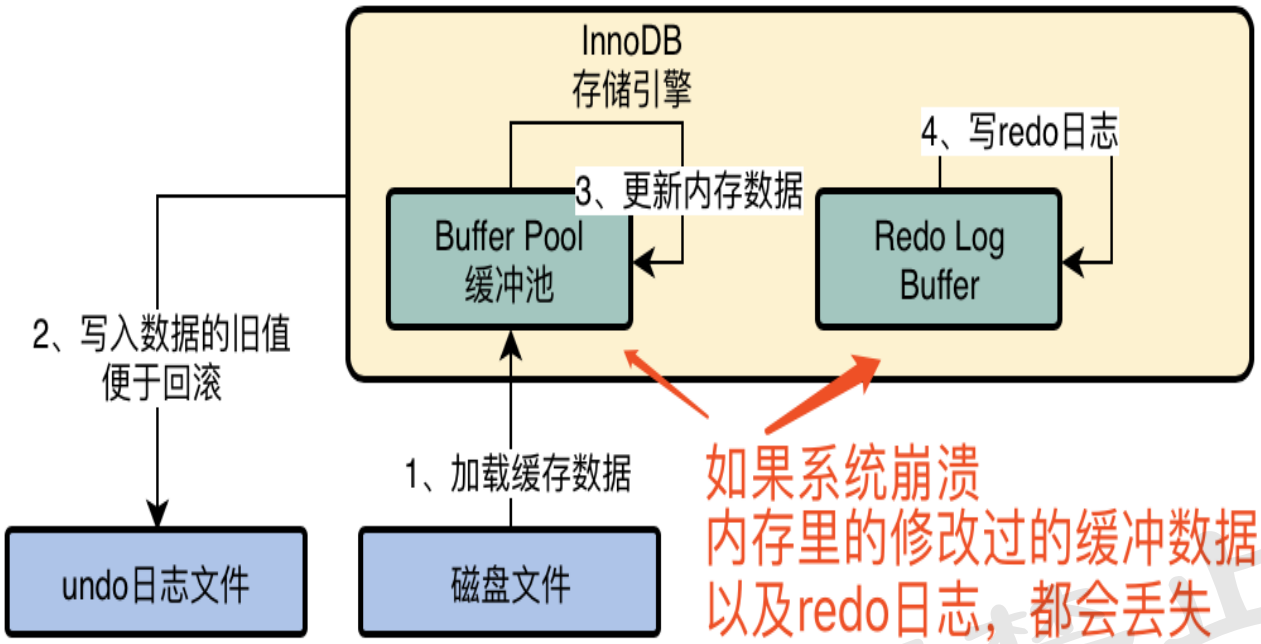
6、如果还没提交事务，MySQL宕机了怎么办?

这里我们假设每个人看专栏的人，都对MySQL的基本SQL语法、事务的基本概念以及索引的基本概念有一个基础的了解，因为但凡一个后端工程师，要跟数据库打交道，必然会跟这些概念有一定的了解。

所以我们都知道，其实在数据库中，哪怕执行一条SQL语句，其实也可以是一个独立的事务，只有当你提交事务之后，SQL语句才算执行结束。

所以这里我们都知道，到目前为止，其实还没有提交事务，那么此时如果MySQL崩溃，必然导致内存里Buffer Pool中的修改过的数据都丢失，同时你写入Redo Log Buffer中的redo日志也会丢失

我们看下图



那么此时数据丢失要紧吗?

其实是不要紧的，因为你一条更新语句，没提交事务，就代表他没执行成功，此时MySQL宕机虽然导致内存里的数据都丢失了，但是你会发现，磁盘上的数据依然还停留在原样子。

也就是说，“id=1”的那行数据的name字段的值还是老的值，“zhangsan”，所以此时你的这个事务就是执行失败了，没能成功完成更新，你会收到一个数据库的异常。然后当mysql重启之后，你会发现你的数据并没有任何变化。

所以此时如果mysql宕机，不会有任何的问题。

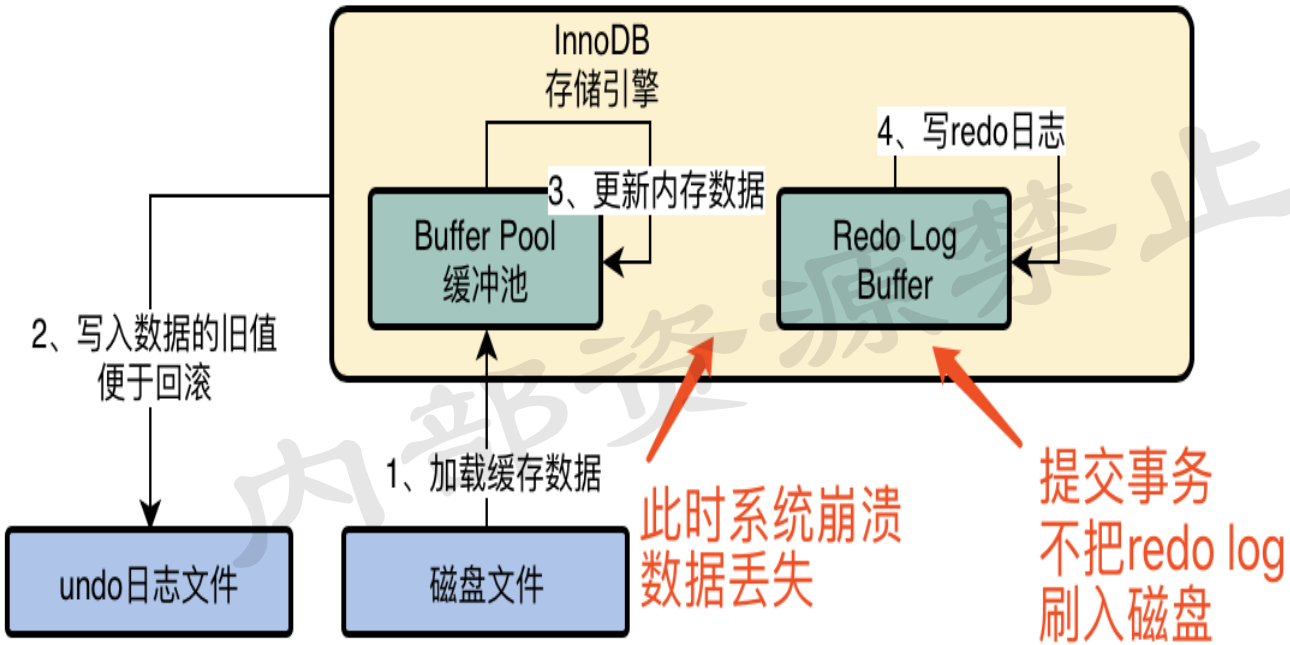
7、提交事务的时候将redo日志写入磁盘中

接着我们想要提交一个事务了，此时就会根据一定的策略把redo日志从redo log buffer里刷入到磁盘文件里去。

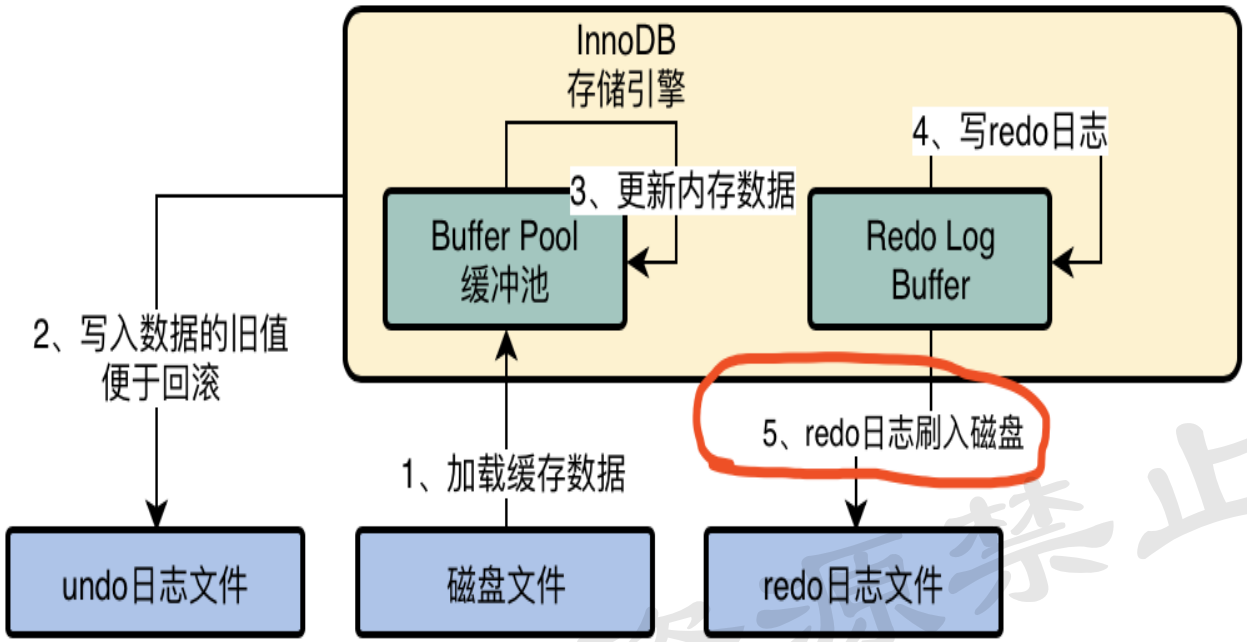
此时这个策略是通过innodb_flush_log_at_trx_commit来配置的，他有几个选项。

当这个参数的值为0的时候，那么你提交事务的时候，不会把redo log buffer里的数据刷入磁盘文件的，此时可能你都提交事务了，结果mysql宕机了，然后此时内存里的数据全部丢失。

相当于你提交事务成功了，但是由于MySQL突然宕机，导致内存中的数据和redo日志都丢失了，我们看下图：



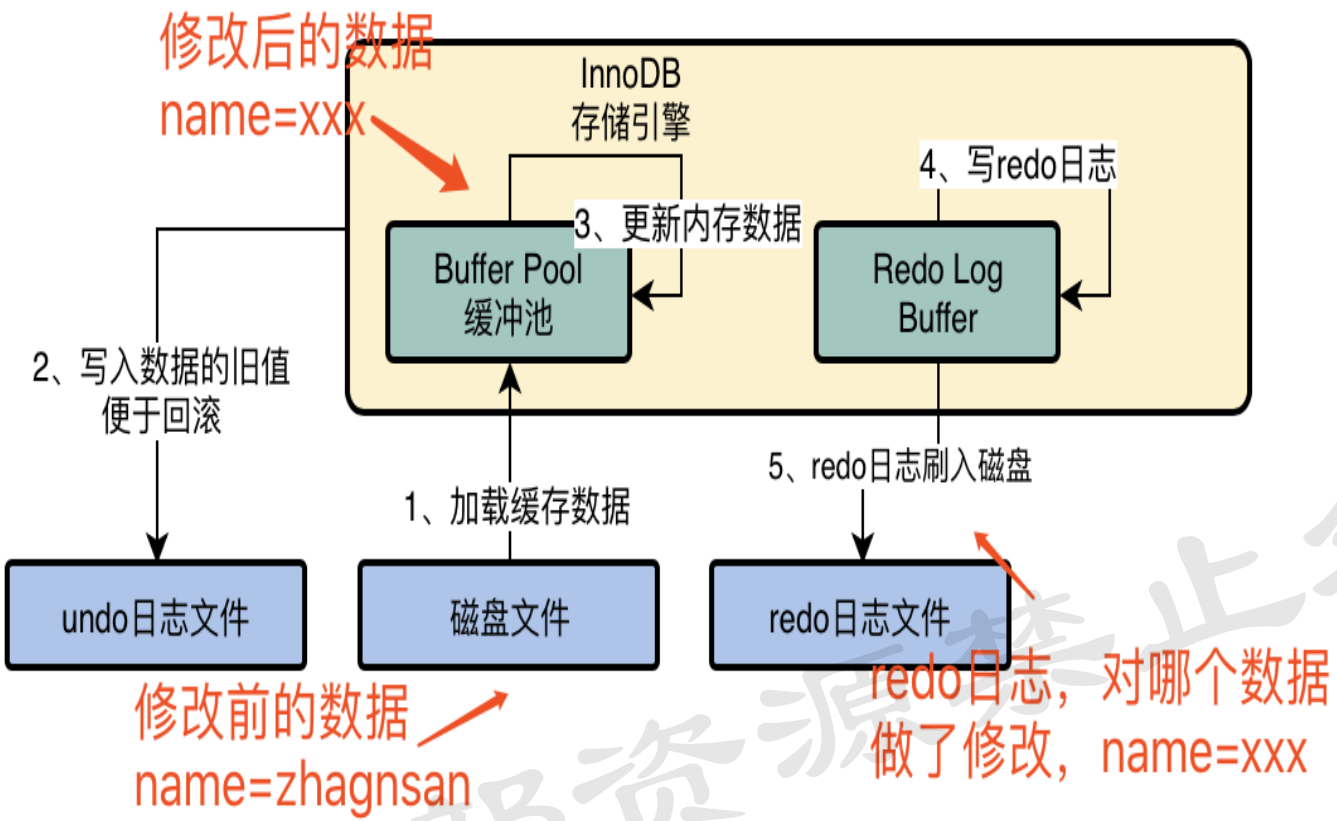
当这个参数的值为1的时候，你提交事务的时候，就必须把redo log从内存刷入到磁盘文件里去，只要事务提交成功，那么redo log就必然在磁盘里了，我们看下图：



那么只要提交事务成功之后，redo日志一定在磁盘文件里，此时你肯定会有一条redo日志说了，“我此时对哪个数据做了一个什么修改，比如name字段修改为xxx了”。

然后哪怕此时buffer pool中更新过的数据还没刷新到磁盘里去，此时内存里的数据是已经更新过的“name=xxx”，然后磁盘上的数据还是没更新过的“name=zhangsan”。

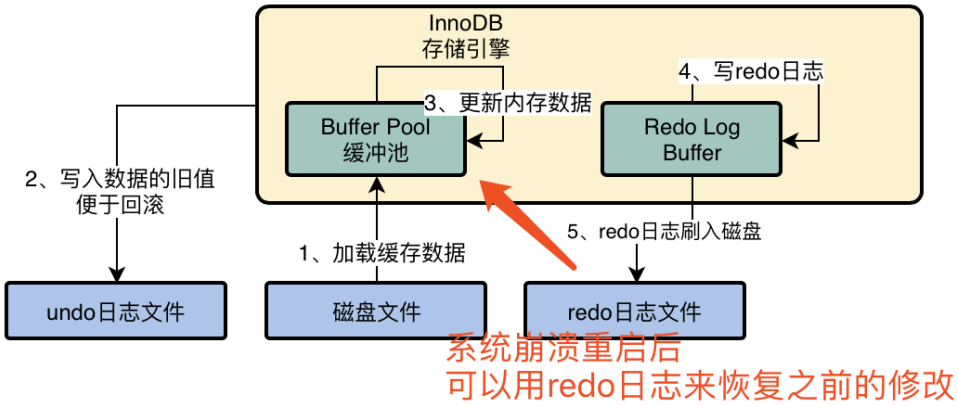
我们看下图，提交事务之后，可能处于的一个状态。



此时如果说提交事务后处于上图的状态，然后mysql系统突然崩溃了，此时会如何？会丢失数据吗？

肯定不会啊，因为虽然内存里的修改成name=xxx的数据会丢失，但是redo日志里已经说了，对某某数据做了修改name=xxx。

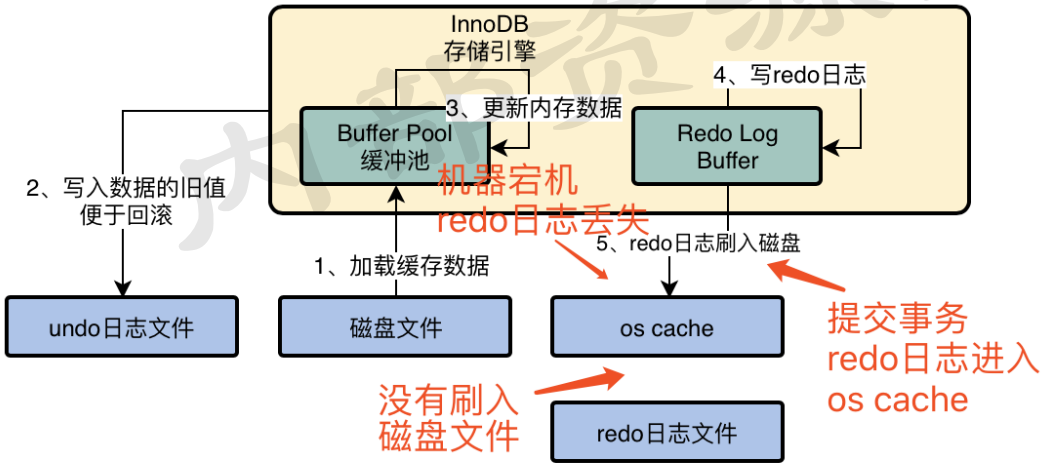
所以此时mysql重启之后，他可以根据redo日志去恢复之前做过的修改，我们看下图。



最后来看看，如果innodb_flush_log_at_trx_commit参数的值是2呢?

他的意思就是，提交事务的时候，把redo日志写入磁盘文件对应的os cache缓存里去，而不是直接进入磁盘文件，可能1秒后才会把os cache里的数据写入到磁盘文件里去。

这种模式下，你提交事务之后，redo log可能仅仅停留在os cache内存缓存里，没实际进入磁盘文件，万一此时你要是机器宕机了，那么os cache里的redo log就会丢失，同样会让你感觉提交事务了，结果数据丢了，看下图。



8、小思考题：三种redo日志刷盘策略到底选择哪一种?

今天给大家留一个小的思考题，大家觉得在提交事务的时候，我们对redo日志的刷盘策略应该选择哪一种？每一种刷盘策略的优缺点分别是什么？为什么?

欢迎大家在评论区留言和我交流

End

图文 04 借着更新语句在InnoDB存储引擎中的执行流程，聊聊binlog是什么？ 手机观看

详情 评论

借着更新语句在InnoDB存储引擎中的执行流程，聊聊binlog是什么？

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

1、上一讲思考题解答：redo日志刷盘策略的选择建议

先给大家解释一下上一讲的思考题，我给大家的一个建议，其实对于redo日志的三种刷盘策略，我们通常建议是设置为1

也就是说，提交事务的时候，redo日志必须是刷入磁盘文件里的。

这样可以严格的保证提交事务之后，数据是绝对不会丢失的，因为有redo日志在磁盘文件里可以恢复你做的所有修改。

如果要是选择0的话，可能你提交事务之后，mysql宕机，那么此时redo日志没有刷盘，导致内存里的redo日志丢失，你提交的事务更新的数据就丢失了；

如果要是选择2的话，如果机器宕机，虽然之前提交事务的时候，redo日志进入os cache了，但是还没进入磁盘文件，此时机器宕机还是会导致os cache里的redo日志丢失。

所以对于数据库这样严格的系统而言，一般建议redo日志刷盘策略设置为1，保证事务提交之后，数据绝对不能丢失。

2、MySQL binlog到底是什么东西?

接着我们来看看MySQL binlog到底是个什么东西?

实际上我们之前说的redo log，他是一种偏向物理性质的重做日志，因为他里面记录的是类似这样的东西，“对哪个数据页中的什么记录，做了个什么修改”。

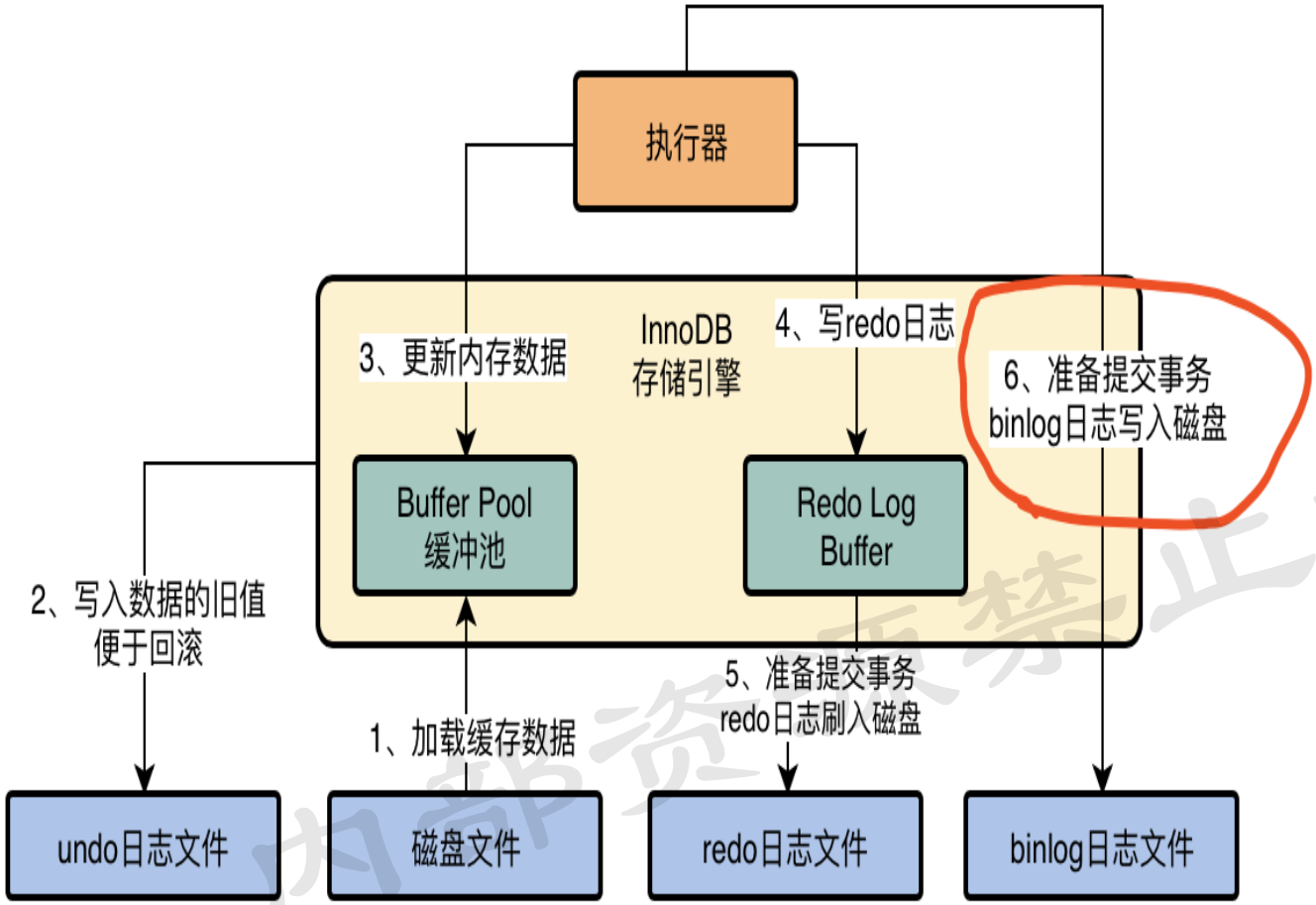
而且redo log本身是属于InnoDB存储引擎特有的一个东西。

而binlog叫做归档日志，他里面记录的是偏向于逻辑性的日志，类似于“对users表中的id=10的一行数据做了更新操作，更新以后的值是什么”

binlog不是InnoDB存储引擎特有的日志文件，是属于mysql server自己的日志文件。

3、提交事务的时候，同时会写入binlog

所以其实我们上一讲讲到，在我们提交事务的时候，会把redo log日志写入磁盘文件中去。然后其实在提交事务的时候，我们同时还会把这次更新对应的binlog日志写入到磁盘文件中去，如下图所示。



大家可以在这个图里看到一些变动，就是我把跟InnoDB存储引擎进行交互的组件加入了之前提过的执行器这个组件，他会负责跟InnoDB进行交互，包括从磁盘里加载数据到Buffer Pool中进行缓存，包括写入undo日志，包括更新Buffer Pool里的数据，以及写入redo log buffer，redo log刷入磁盘，写binlog，等等。

实际上，执行器是非常核心的一个组件，负责跟存储引擎配合完成一个SQL语句在磁盘与内存层面的全部数据更新操作。

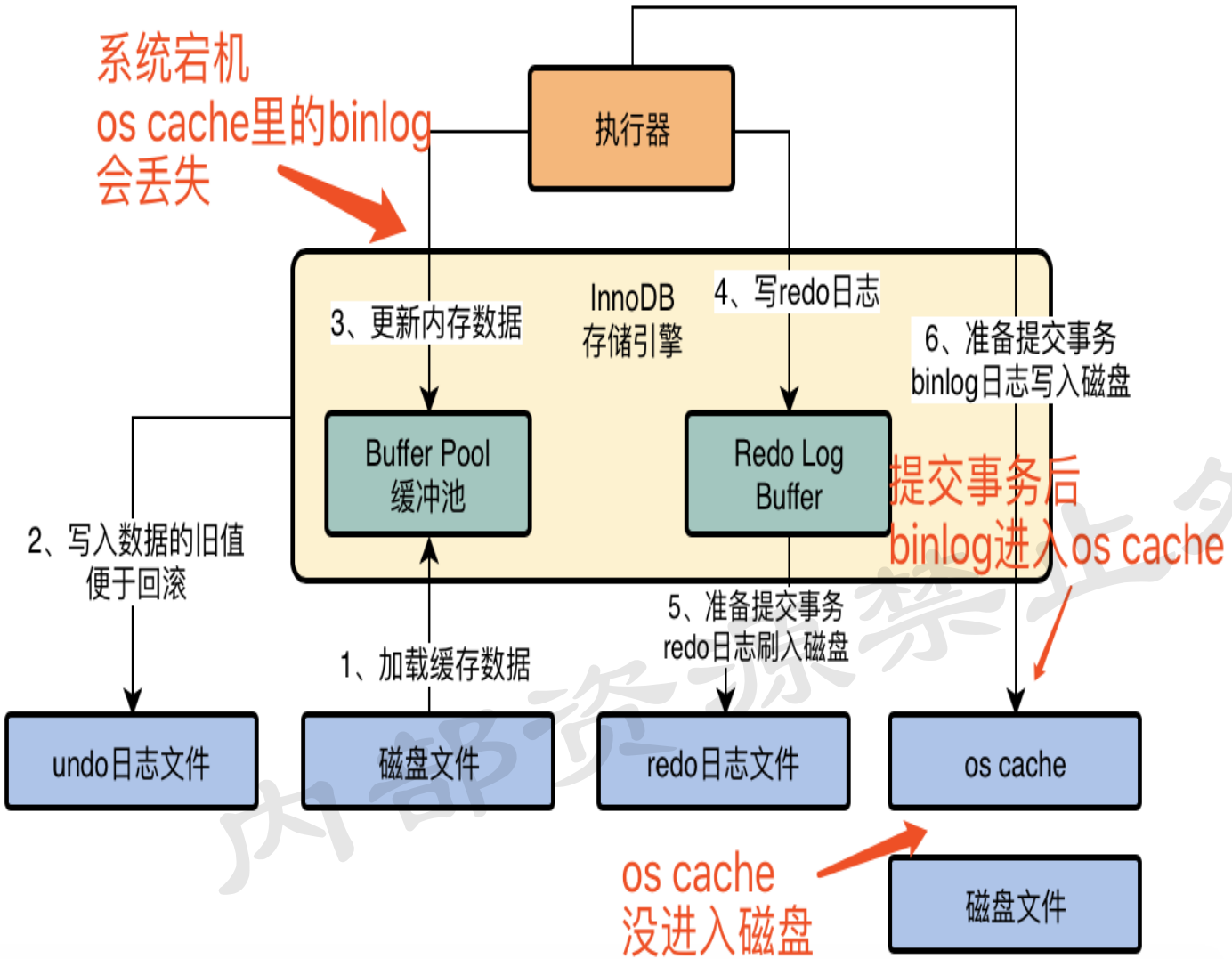
而且我们在上图可以看到，我把一次更新语句的执行，拆分为了两个阶段，上图中的1、2、3、4几个步骤，其实本质是你执行这个更新语句的时候干的事。

然后上图中的5和6两个步骤，是从你提交事务开始的，属于提交事务的阶段了。

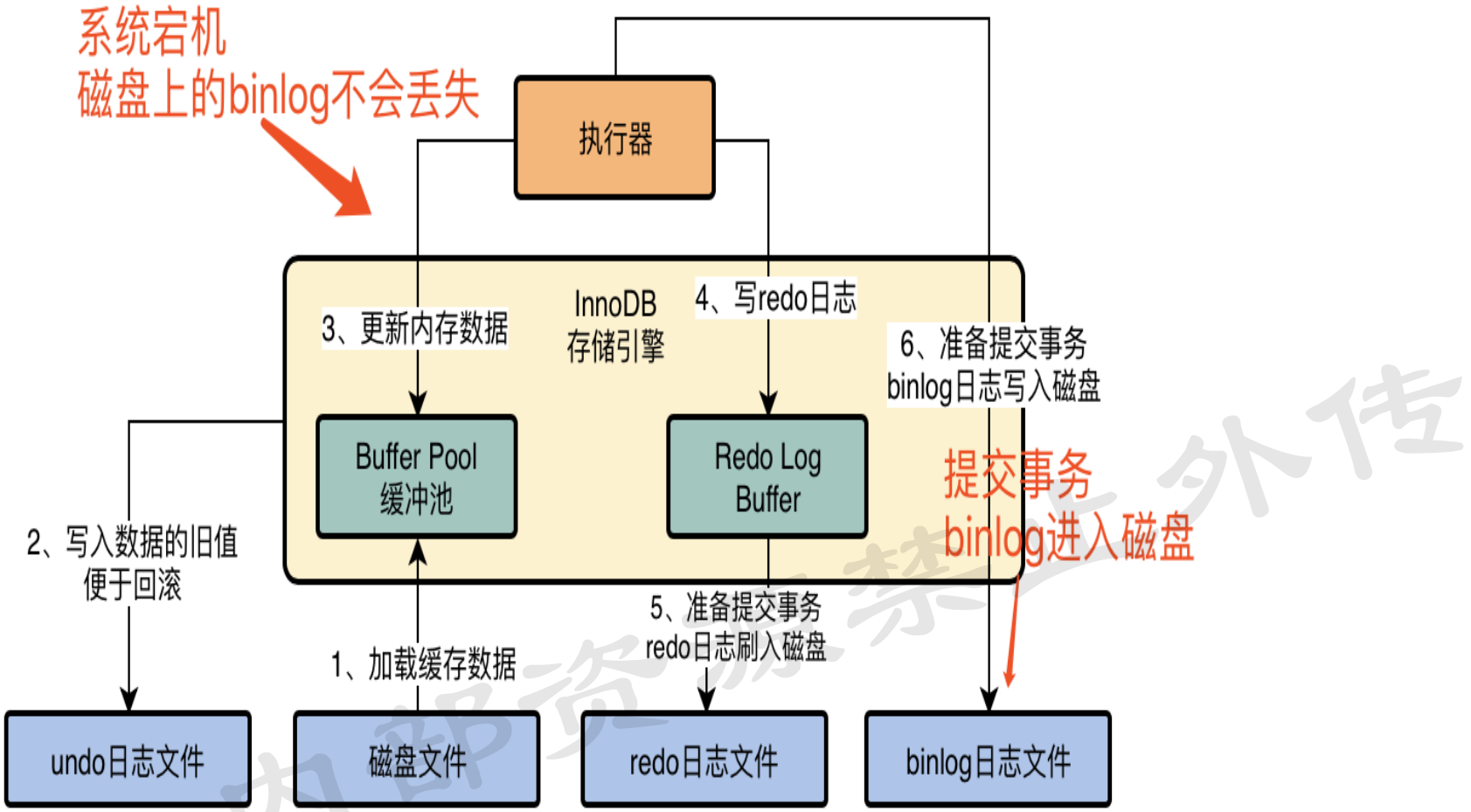
4、binlog日志的刷盘策略分析

对于binlog日志，其实也有不同的刷盘策略，有一个**sync_binlog**参数可以控制binlog的刷盘策略，他的默认值是0，此时你把binlog写入磁盘的时候，其实不是直接进入磁盘文件，而是进入os cache内存缓存。

所以跟之前分析的一样，如果此时机器宕机，那么你在os cache里的binlog日志是会丢失的，我们看下图的示意



如果要是把**sync_binlog**参数设置为1的话，那么此时会强制在提交事务的时候，把binlog直接写入到磁盘文件里去，那么这样提交事务之后，哪怕机器宕机，磁盘上的binlog是不会丢失的，如下图所示

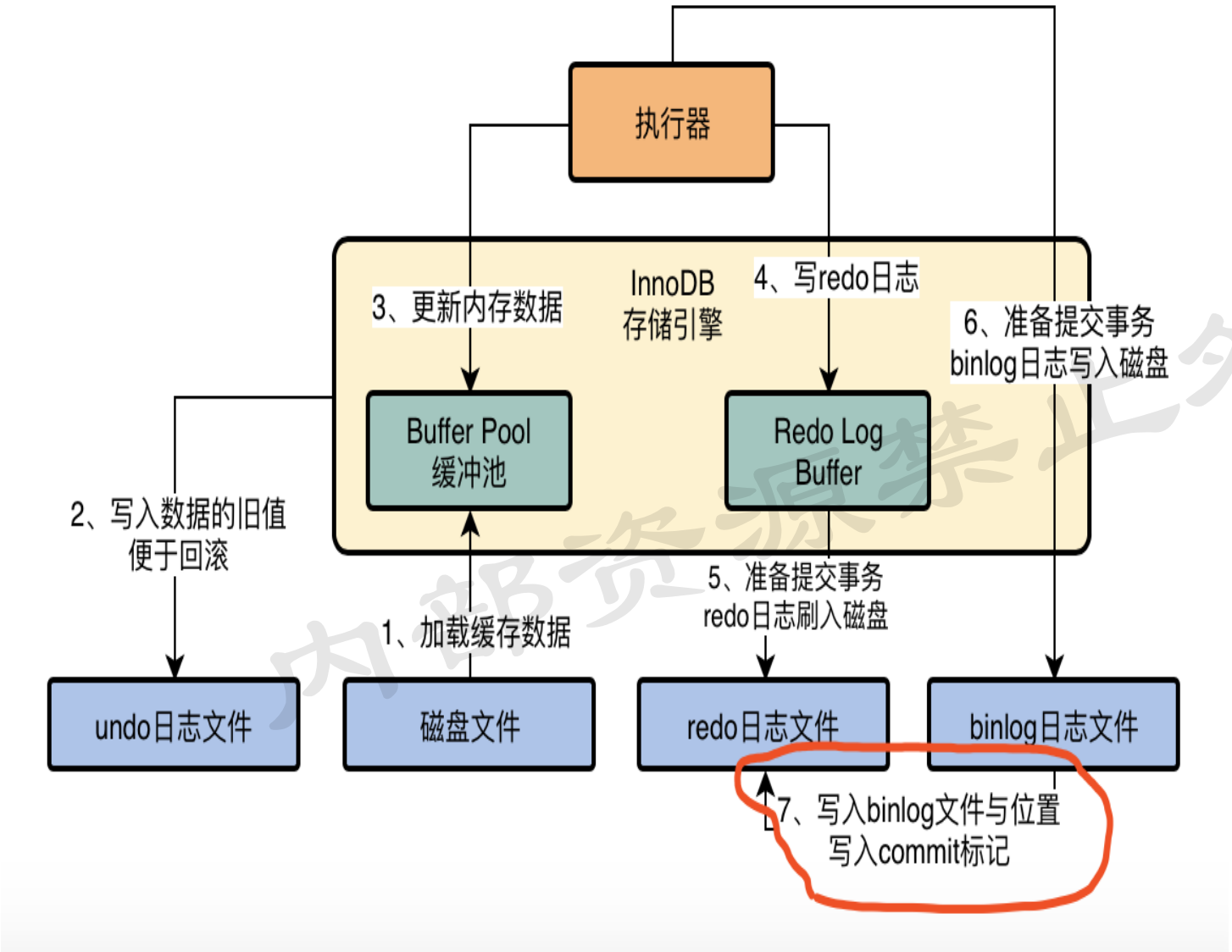


5、基于binlog和redo log完成事务的提交

当我们把binlog写入磁盘文件之后，接着就会完成最终的事务提交，此时会把本次更新对应的binlog文件名称和这次更新的binlog日志在文件里的位置，都写入到redo log日志文件里去，同时在redo log日志文件里写入一个commit标

记。

在完成这个事情之后，才算最终完成了事务的提交，我们看下图的示意。



6、最后一步在redo日志中写入commit标记的意义是什么?

这时候肯定有同学会问了，最后在redo日志中写入commit标记有什么意义呢?

说白了，他其实是用来保持redo log日志与binlog日志一致的。

我们来举个例子，假设我们在提交事务的时候，一共有上图中的5、6、7三个步骤，必须是三个步骤都执行完毕，才算是提交了事务。那么在我们刚完成步骤5的时候，也就是redo log刚刷入磁盘文件的时候，mysql宕机了，此时怎么办?

这个时候因为没有最终的事务commit标记在redo日志里，所以此次事务可以判定为不成功。不会说redo日志文件里有这次更新的日志，但是binlog日志文件里没有这次更新的日志，不会出现数据不一致的问题。

如果要是完成步骤6的时候，也就是binlog写入磁盘了，此时mysql宕机了，怎么办?

同理，因为没有redo log中的最终commit标记，因此此时事务提交也是失败的。

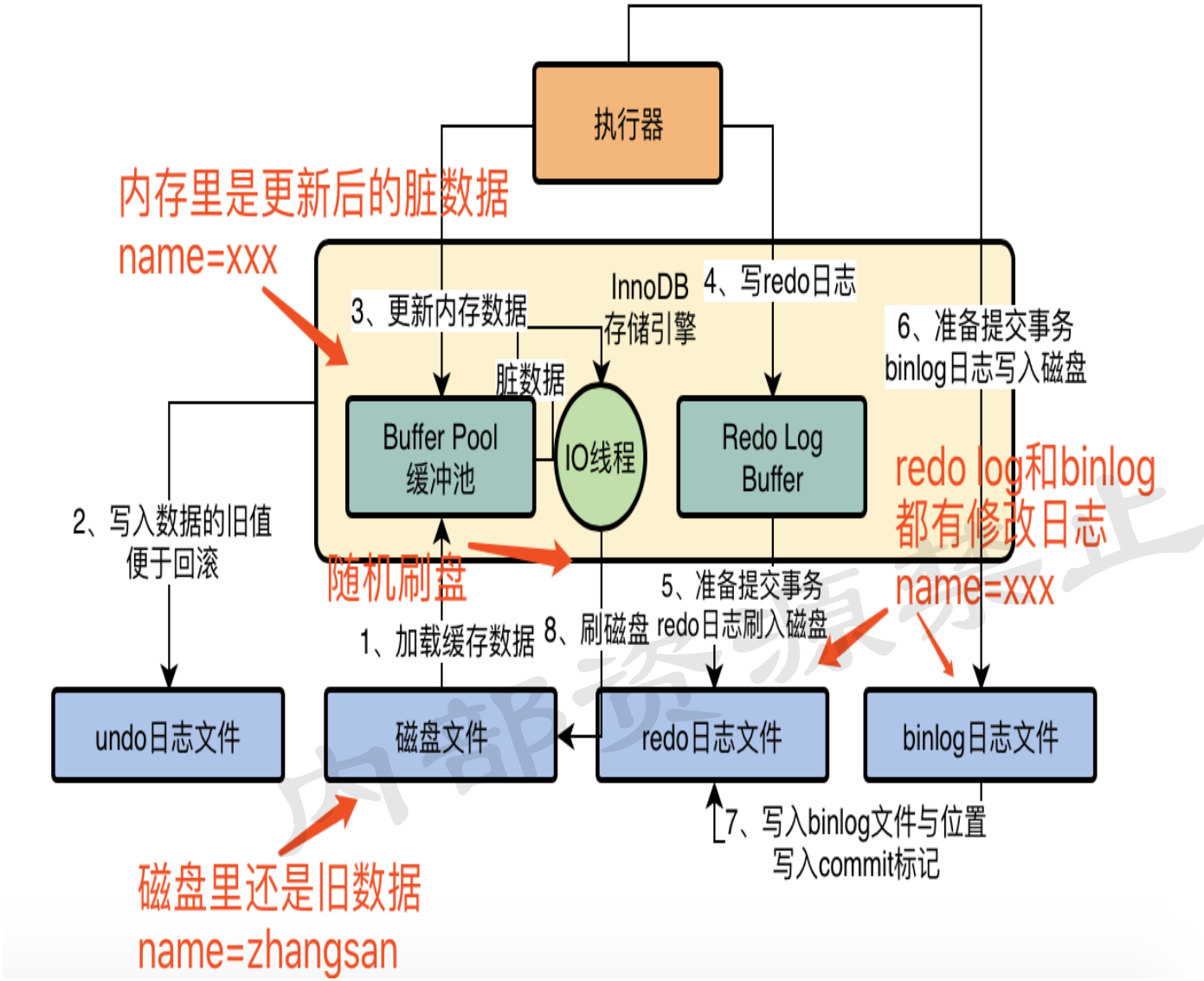
必须是在redo log中写入最终的事务commit标记了，然后此时事务提交成功，而且redo log里有本次更新对应的日志，binlog里也有本次更新对应的日志 ，redo log和binlog完全是一致的。

7、后台IO线程随机将内存更新后的脏数据刷回磁盘

现在我们假设已经提交事务了，此时一次更新“update users set name='xxx' where id=10”，他已经把内存里的buffer pool中的缓存数据更新了，同时磁盘里有redo日志和binlog日志，都记录了把我们指定的“id=10”这行数据修改了“name='xxx'”。

此时我们会思考一个问题了，但是这个时候磁盘上的数据文件里的“id=10”这行数据的name字段还是等于zhangsan这个旧的值啊!

所以MySQL有一个后台的IO线程，会在之后某个时间里，随机的把内存buffer pool中的修改后的脏数据给刷回到磁盘上的数据文件里去，我们看下图:



当上图中的IO线程把buffer pool里的修改后的脏数据刷回磁盘的之后，磁盘上的数据才会跟内存里一样，都是name=xxx这个修改以后的值了！

在你IO线程把脏数据刷回磁盘之前，哪怕mysql宕机崩溃也没关系，因为重启之后，会根据redo日志恢复之前提交事务做过的修改到内存里去，就是id=10的数据的name修改为了xxx，然后等适当时机，IO线程自然还是会把这个修改后的数据刷到磁盘上的数据文件里去的

8、基于更新数据的流程，总结一下InnoDB存储引擎的架构原理

大家通过一次更新数据的流程，就可以清晰地看到，InnoDB存储引擎主要就是包含了一些buffer pool、redo log buffer等内存里的缓存数据，同时还包含了一些undo日志文件，redo日志文件等东西，同时mysql server自己还有binlog日志文件。

在你执行更新的时候，每条SQL语句，都会对应修改buffer pool里的缓存数据、写undo日志、写redo log buffer几个步骤；

但是当你提交事务的时候，一定会把redo log刷入磁盘，binlog刷入磁盘，完成redo log中的事务commit标记；最后后台的IO线程会随机的把buffer pool里的脏数据刷入磁盘里去。

9、思考题：执行更新操作的时候，为什么不能执行修改磁盘上的数据？

好了，今天的文章接近尾声，咱们再来思考一个问题：

> 为什么MySQL在更新数据的时候，要大费周章的搞这么多事情，包括buffer pool、redo log、undo log、binlog、事务提交、脏数据。引入了一大堆的概念，有复杂的流程和步骤。
>
> 为什么他反而最关键的修改磁盘里的数据，要通过IO线程不定时的去执行？
>
> 为什么他不干脆直接就每次执行SQL语句，直接就更新磁盘里的数据得了？

希望大家踊跃思考，在评论区给出自己的答案。

End

狸猫技术窝精品专栏及课程推荐：

- 《从零开始带你成为消息中间件实战高手》
- 《21天互联网Java进阶面试训练营》（分布式篇）
- 《互联网Java工程师面试突击》（第1季）
- 《互联网Java工程师面试突击》（第3季）
- 《从零开始带你成为JVM实战高手》

图文 05 生产经验：真实生产环境下的数据库机器配置如何规划？

手机观看

688 人次阅读 2020-02-03 07:30:00

详情 评论

生产经验：真实生产环境下的数据库机器配置如何规划?

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

1、当你了解数据库的架构原理之后，就该了解一下自己数据库的规划

之前我们用了4篇文章给大家整体分析了一下MySQL数据库的工作原理，相信很多朋友都已经对数据库的整体架构原理有了一定的了解，毕竟在这之前，可能大部分人对MySQL数据库的了解还停留在执行SQL语句的程度。

当我们初步了解了数据库的架构设计原理后，接着其实应该了解的第一件事，就是我们平时在工作中，如何规划生产环境下的数据库。因为我想很多人如果平时主要是负责一些没什么并发量，用户量也就几十个或者几百个人的系统，那么根本就不会去关注数据库的规划这件事情。

对很多Java工程师而言，要不然是自己找一台linux机器装一个MySQL，然后就让自己的系统连接上去直接就开始使用了，要不然就是让DBA或者运维工程师帮自己去找一台机器装一个MySQL或者Oracle，然后自己就可以直接使用了。

但是在我们的专栏中，我们希望能够教会大家较为专业化的数据库使用经验，包括数据库的整体架构原理，还有就是如何规划生产环境下的数据库，包括当你有一个生产库之后，要做的事情就是设计压测方案，包括对你的数据库进行压测，包括对你的数据库部署可视化监控系统，等等。

当你做好这一系列的事情之后，接着才应该是开发你的Java系统，去操作你的数据库，实现各种各样的业务功能和逻辑。

2、生产数据库一般用什么配置的机器?

现在我们来看第一个问题，假设你在生产环境中需要部署一个数据库，此时首先你就需要一个机器来部署这个数据库。那么我们要考虑的事情就是，部署一个生产环境的数据库，一般需要什么样配置的机器呢?

接下来我们将会给大家说一些我们的经验值，直接告诉大家什么样配置的机器部署的MySQL数据库，大致适合多高的并发访问量

当你了解这个经验值之后，未来当你在负责系统的开发，申请数据库的时候，你就知道生产环境下的数据库大致需要什么样的机器配置了，大致可以抗下多少并发访问了。

首先我们先明确一点，如果你负责的系统就是一个没什么并发访问量，用户就几十个人或者几百个人的系统，那么其实你选择什么样的机器去部署数据库，影响都不是很大，哪怕是你用我们自己平时用的个人笔记本电脑去部署一个MySQL数据库，其实也能支撑那种低并发系统的运行。

因为那种系统可能每隔几分钟才会有一波请求发到数据库上去，而且数据库里一张表也许就几百条、几千条或者几万条数据，数据量很小，并发量很小，操作频率很低，用户量很小，并发量很小，只不过可能系统的业务逻辑很复杂而已。对于这类系统的数据库机器选型，就不在我们的考虑范围之内了。

我们主要关注的是有一定并发量的互联网类的系统，对数据库可能会产生每秒几百，每秒几千，甚至每秒上万的并发请求量，对于这类场景下，我们应该选择什么样的机器去部署数据库，才能比较好的抗下我们的系统压力。

3、普通的Java应用系统部署在机器上能抗多少并发?

通常来说，根据我们的经验值而言，Java应用系统部署的时候常选用的机器配置大致是2核4G和4核8G的较多一些，数据库部署的时候常选用的机器配置最低在8核16G以上，正常在16核32G

那么以我们大量的高并发线上系统的生产经验观察下来而言，一般Java应用系统部署在4核8G的机器上，每秒钟抗下500左右的并发访问量，差不多是比较合适的，当然这个也不一定。因为你得考虑一下，假设你每个请求花费1s可以处理完，那么你一台机器每秒也许只可以处理100个请求，但是如果你每个请求只要花费100ms就可以处理完，那么你一台机器每秒也许就可以处理几百个请求。

所以一台机器能抗下每秒多少请求，往往是跟你每个请求处理耗费多长时间是关联的，但是大体上来说，根据我们大量的经验观察而言，4核8G的机器部署普通的Java应用系统，每秒大致就是抗下几百的并发访问，从每秒一两百请求到每秒七八百请求，都是有可能的，关键是看你每个请求处理需要耗费多长时间。

4、高并发场景下，数据库应该用什么样的机器?

对于数据库而言，我们刚才也说过了，通常推荐的数据库至少是选用8核16G以的机器，甚至是16核32G的机器更加合适一些。

因为大家要考虑一个问题，对于我们的Java应用系统，主要耗费时间的是Java系统和数据库之间的网络通信。对Java系统自己而言，如果你仅仅只是系统内部运行一些普通的业务逻辑，纯粹在自己内存中完成一些业务逻辑，这个性能是极高极高的。

对于你Java系统接收到的每个请求，耗时最多的还是发送网络请求到数据库上去，等待数据库执行一些SQL语句，返回结果给你。

所以其实我们常说你有一个Java系统压力很大，负载很高，但是其实你要明白一点，你这个Java系统其实主要的压力和复杂都是集中在你依赖的那个MySQL数据库上的!

因为你执行大量的增删改查的SQL语句的时候，MySQL数据库需要对内存和磁盘文件进行大量的IO操作，所以数据库往往是负载最高的！这个问题我们在之前4篇文章里，通过MySQL数据库架构原理的分析，都已经讲解过了。

而你的Java系统一般并不需要你去直接大量的读写本地文件进行耗时的IO操作吧？是不是，想必做过Java开发的朋友一下子就会想明白这个道理。

所以往往对一个数据库而言，都是选用8核16G的机器作为起步，最好是选用16核32G的机器更加合适一些，因为数据库需要执行大量的磁盘IO操作，他的每个请求都比较耗时一些，所以机器的配置自然需要高一些了。

然后通过我们之前的经验而言，一般8核16G的机器部署的MySQL数据库，每秒抗个一两千并发请求是没问题的，但是如果你的并发量再高一些，假设每秒有几千并发请求，那么可能数据库就会有点危险了，因为数据库的CPU、磁盘、IO、内存的负载都会很高，弄不数据库压力过大就会宕机。

对于16核32G的机器部署的MySQL数据库而言，每秒抗个两三千，甚至三四千的并发请求也都是可以的，但是如果你达到每秒上万请求，那么数据库的CPU、磁盘、IO、内存的负载瞬间都会飙升到很高，数据库也是可能会扛不住宕机的。

所以这就是对于数据库，我们一般推荐选用的机器的配置，以及他大致可以抗下多高的并发请求量的经验分享。

另外对于数据库而言，如果可以的话，最好是采用SSD固态硬盘而不是普通的机械硬盘，因为数据库最大的复杂就在于大量的磁盘IO，他需要大量的读写磁盘文件，所以如果能使用SSD固态硬盘，那么你的数据库每秒能抗的并发请求量就会更高一些。

5、今日思考题

今天想留给大家一个小的思考题：假设你开发的Java系统部署在一台4核8G的机器上，那么我们假设这个Java系统处理一个请求非常非常快，每个请求只需要0.01ms就可以处理完了，那你觉得这一台机器部署的Java系统，可以实现每秒抗下几千并发请求吗？可以实现每秒抗下几万并发请求吗？

请大家思考思考这个问题，把你的思考结果在评论区发出来，跟大家一起交流，同时也多看看评论区下的别人的留言。

End

狸猫技术窝精品专栏及课程推荐：

- 《从零开始带你成为消息中间件实战高手》
- 《21天互联网Java进阶面试训练营》（分布式篇）
- 《互联网Java工程师面试突击》（第1季）
- 《互联网Java工程师面试突击》（第3季）
- 《从零开始带你成为JVM实战高手》

图文 06 生产经验：互联网公司的生产环境数据库是如何进行性能测试的？ 手机观看

573 人次阅读 2020-02-03 07:30:00

详情 评论

生产经验：互联网公司的生产环境数据库是如何进行性能测试的？

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

1、申请了机器之后，你作为Java架构师就要心里有数

上一篇文章我们讲到了在真实的项目中，第一件事情就是申请数据库机器，一般来说我们需要申请8核16G或者16核32G的高配置机器下来，甚至要机器全部搭配SSD固态硬盘，然后让DBA兄弟在申请下来的机器上安装和部署一个MySQL，同时启动MySQL数据库。

当然如何安装和部署MySQL，以及如何启动MySQL，都是非常简单的，大家网络上随便一搜索就会看到大量类似的东西，那不是我们专栏要讲的东西。然后MySQL在生产环境下的各种纷繁复杂的高级参数的调整，暂时我们还不会立马涉及到，那些是属于MySQL DBA需要搞定的事情。

但是我们后续随着专栏的推进，会讲解一部分MySQL生产环境中的高阶参数的调优和配置，有一些是跟我们开发Java应用系统密切相关的东西，我们作为开发人员，也是需要了解MySQL一些高阶参数的调优的，有时候在我们优化系统性能的时候，可能就需要跟DBA一起配合进行调优。

但是简单来说，我们作为一个项目的核心Java工程师甚至Java架构师，必须要选择自己的数据库使用什么配置的机器，心里大致明白这个配置的机器部署的数据库，大致能帮我们抗下每秒多少并发请求。

比如你申请的是8核16G的机器来部署MySQL，那你作为项目的Java架构师，心里大致就该知道你这个数据库后续每秒抗个一两千请求还是可以的，如果你申请的是16核32G的机器，那你心里就知道妥妥可以抗个每秒两三千，甚至三四千的请求，你心里就有数了，这是你要做到的

2、把机器交给专业的DBA，让他部署MySQL

其次你要知道的是，你申请一台机器下来之后，接着这台机器在有一定规模的公司里，一定是交给公司专业的DBA去安装、部署和启动MySQL的，DBA这个时候会按照他过往的经验，用自己的MySQL生产调优参数模板，直接放到MySQL里去，然后用一个参数模板去启动这个MySQL，往往这里很多参数都是调优过的。

而且DBA还可能会对linux机器的一些OS内核参数进行一定的调整，比如说最大文件句柄之类的参数，这些参数往往也都是需要调整的。

接着当DBA搞定这台机器上的数据库之后，就会交给你来使用，你就知道这台机器的地址和用户名密码，然后就的Java系统就可以直接连接上去，就可以执行各种各样的SQL语句去实现业务逻辑了。

3、有了数据库之后，还需要先进行压测

当你手头有一个可以使用的数据库之后，你觉得就可以直接基于他开发Java系统了吗？

并不是这样的！这么做在一个互联网公司里往往会显得比较的业余，因为你首先得先对这个数据库进行一个较为基本的基准压测。

也就是说，你得基于一些工具模拟一个系统每秒发出1000个请求到数据库上去，观察一下他的CPU负载、磁盘IO负载、网络IO负载、内存复杂，然后数据库能否每秒处理掉这1000个请求，还是每秒只能处理500个请求？这个过程，就是压测。

你不光用工具每秒发送1000个请求，还可以模拟每秒发送2000个请求，甚至3000个请求，逐步的测试出来，这个数据库在目前的机器配置之下，他大致的一个负载压力如何，性能表现如何，每秒最多可以抗多少请求。

可能有的人会提出疑问了，他会说：老师，为什么刚开始就要对数据库搞一个基准压测？你完全可以等Java系统都开发完毕了，然后直接让Java系统连接上MySQL数据库，然后直接对Java系统进行压测啊！

如果有人提出这个问题，那就有所不知了，数据库的压测和他上面的Java系统的压测，其实是两回事儿，首先你得知道你的数据库最大能抗多大压力，然后你再去看你的Java系统能抗多大压力。

因为有一种可能是，你的数据库每秒可以抗下2000个请求，但是你的Java系统每秒只能抗下500个请求，这也是有可能的。所以你不能光是针对Java系统去进行压测，在那之前也得先对数据库进行压测，心里得有个数。

4、傻傻分不清楚：QPS和TPS到底有什么区别?

既然要压测了，那么肯定得先明白一点，我们压测数据库，最终是想看看这个数据库在现有的机器配置之下，每秒可以抗下多少个请求呢？这个每秒抗下多少个请求，其实是有专业术语的，分别是QPS和TPS。

就QPS而言，他的英文全称是：Query Per Second。

其实就是英文字面意思已经很明确了，QPS就是说，你的这个数据库每秒可以处理多少个请求，你大致可以理解为，一次请求就是一条SQL语句，也就是说这个数据库每秒可以处理多少个SQL语句。

对于QPS而言，其实你的一些Java系统或者中间件系统在进行压测的时候，也可以使用这个指标，也就是说，你的Java系统每秒可以处理多少个请求。

然后另外一个术语是TPS，他的英文全称是：Transaction Per Second。其实就是每秒可处理的事务量，这个TPS往往是用在数据库中较多一些，其实从字面意思就能看的出来，他就是说数据库每秒会处理多少次事务提交或者回滚。

因为大家应该都对数据库有一个基本的了解，就是他的事务到底是什么？

简单来说，一个事务就会包含多个SQL语句，这些SQL最好要么就是事务提交，大家一起成功了，要么就是最好事务回滚，大家一起失败了，这就是事务。

所以TPS往往指的是一个数据库每秒里有多少个事务执行完毕了，事务提交或者回滚都算是事务执行完毕了，所以TPS衡量的是一个数据库每秒处理完的事务的数量。有一些人往往会把TPS理解为是数据库每秒钟处理请求的数量，其实这是不太严谨的。

5、IO相关的压测性能指标

接着再给大家讲几个压测的时候要关注的IO相关的性能指标，大家也要对他做一个了解：

（1）IOPS：这个指的是机器的随机IO并发处理的能力，比如机器可以达到200 IOPS，意思就是说每秒可以执行200个随机IO读写请求。

这个指标是很关键的，因为之前我们在数据库架构原理中讲解过，你在内存中更新的脏数据库，最后都会由后台IO线程在不确定的时间，刷回到磁盘里去，这就是随机IO的过程。如果说IOPS指标太低了，那么会导致你内存里的脏数据刷回磁盘的效率就会不高。

（2）吞吐量：这个指的是机器的磁盘存储每秒可以读写多少字节的数据量

这个指标也是很关键的，因为大家通过之前的学习都知道，我们平时在执行各种SQL语句的时候，提交事务的时候，其实都是大量的会写redo log之类的日志的，这些日志都会直接写磁盘文件。

所以一台机器他的存储每秒可以读写多少字节的数据量，就决定了他每秒可以把多少redo log之类的日志写入到磁盘里去。一般来说我们写redo log之类的日志，都是对磁盘文件进行顺序写入的，也就是一行接着一行的写，不会说进行随机的读写，那么一般普通磁盘的顺序写入的吞吐量每秒都可以达到200MB左右。

所以通常而言，机器的磁盘吞吐量都是足够承载高并发请求的。

（3）latency：这个指标说的是往磁盘里写入一条数据的延迟。

这个指标同样很重要，因为我们执行SQL语句和提交事务的时候，都需要顺序写redo log磁盘文件，所以此时你写一条日志到磁盘文件里去，到底是延迟1ms，还是延迟100us，这就对你的数据库的SQL语句执行性能是有影响的。

一般来说，当然是你的磁盘读写延迟越低，那么你的数据库性能就越高，你执行每个SQL语句和事务的时候速度就会越快。

6、压测的时候要关注的其他性能指标

除了上面说的QPS、TPS、IOPS、吞吐量、latency这些指标之外，在压测的时候还需要关注机器的其他一些性能指标。

（1）CPU负载：CPU负载是一个很重要的性能指标，因为假设你数据库压测到了每秒处理3000请求了，可能其他的性能指标都还正常，但是此时CPU负载特别高，那么也说明你的数据库不能继续往下压测更高的QPS了，否则CPU是吃不消的。

（2）网络负载：这个主要是要看看你的机器带宽情况下，在压测到一定的QPS和TPS的时候，每秒钟机器的网卡会输入多少MB数据，会输出多少MB数据，因为有可能你的网络带宽最多每秒传输100MB的数据，那么可能你的QPS到1000的时候，网卡就打满了，已经每秒传输100MB的数据了，此时即使其他指标都还算正常，但是你也不能继续压测下去了

（3）内存负载：这个就是看看在压测到一定情况下的时候，你的机器内存耗费了多少，如果说机器内存耗费过高了，说明也不能继续压测下去了

7、后续的压测实战说明

接下来下一篇文章，我就会在我自己的电脑上安装一个MySQL数据库，然后交给大家如何使用方便的压测工具，对数据库进行一定的压测，压测的同时，应该通过哪些便捷的工具，去观察压测过程中的机器表现和各项指标。

8、今日思考题

今天想请每个人思考一下QPS和TPS两个术语，想请大家说说自己在看今天的文章之前，自己对QPS和TPS是怎么理解的，在你们公司里是否有相关的系统和数据库的QPS/TPS的统计？今天看完这篇文章之后，你对QPS和TPS两个术语的理解是否有所变化呢？

另外，我再给大家出另外一个思考题，假设现在你负责一个交易系统，对于这个交易系统，他拆分为了很多服务，一笔交易的完成需要多个服务协作完成，也就是说一次交易请求需要调用多个服务才能完成。

那么你觉得对于每个服务而言，他每秒处理的请求数量是QPS还是TPS呢？对于整个交易系统而言，他每秒钟处理的交易笔数是QPS还是TPS呢？请大家谈谈你的看法。

请大家在评论区写下自己的心得体会，同时多看看其他人在评论区发表的留言，多多交流。

End

狸猫技术窝精品专栏及课程推荐：

- 《从零开始带你成为消息中间件实战高手》
- 《21天互联网Java进阶面试训练营》（分布式篇）
- 《互联网Java工程师面试突击》（第1季）
- 《互联网Java工程师面试突击》（第3季）
- 《从零开始带你成为JVM实战高手》

图文 07 生产经验：如何对生产环境中的数据库进行360度无死角压测？ 手机观看

527 人次阅读 2020-02-04 08:08:38

详情 评论

生产经验：如何对生产环境中的数据库进行360度无死角压测？

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

1、昨日思考题解答

先给大家分析昨天的第一个小思考题：给你一台4核8G的机器，他可以抗到每秒几千甚至每秒几万的并发请求吗？

其实这个是不一定的，因为一台机器到底每秒钟可以抗下多少并发请求，跟CPU、内存、磁盘IO、网络带宽，都是有关系的。

举个例子，之前在我们的一个项目的生产环境中，据我们观察，一台4核8G的机器如果每秒抗下500+的请求，那么他的CPU负载就已经很高了，基本上最多可能也就是去抗下每秒1000+的请求，而且那个时候CPU负载基本会打满，机器有挂掉的风险。

另外如果你的系统的业务逻辑特别的吃内存，也许你一台4核8G的机器跑到每秒几百的请求，内存使用率就很高了，而且JVM GC频率可能会非常的高，所以此时也很难继续提升并发请求了。

所以其实你一台机器是不可能无限制的让他增加可以抗下的并发请求的。

再来看下一个思考题：关于QPS和TPS的。我上次问大家了，如果一个交易系统拆分为了很多服务，那么每个服务每秒接收的并发请求是QPS还是TPS呢?

这个明显是QPS，因为每个服务就负责干自己的一些事儿，其实对他来说，每秒并发请求数量就是QPS。

那么对于多个服务组成的一个大的交易系统而言，这个交易系统每秒可以完成多少笔交易，这是QPS还是TPS呢?

其实这个你可以认为是TPS的概念，因为一笔交易需要调用多个服务来完成，所以一笔交易的完成其实就类似数据库里的一个事务，他涵盖了很多服务的请求调用，所以每秒完成多少笔交易，你可以用TPS来形容。

比如你说交易系统的TPS是300，就是说每秒可以完成300笔交易。那么比如交易系统中的服务A的QPS是500，就是交易系统中的一个服务A每秒可以处理500个请求。

2、一款非常好用的数据库压测工具

上一篇文章给大家讲解了我们在压测的过程中要关注哪些东西，这一篇文章就来带着大家一步一步的利用一个工具进行数据库压测。

先给大家介绍一个非常好用的数据库压测工具，就是**sysbench**，这个工具可以自动帮你在数据库里构造出来大量的数据，你想要多少数据，他就自动给你构造出来多少条数据。

然后这个工具接着可以模拟几千个线程并发的访问你的数据库，模拟使用各种各样的SQL语句来访问你的数据库，包括模拟出来各种事务提交到你的数据库里去，甚至可以模拟出几十万的TPS去压测你的数据库。

所以一般来说，如果你要进行数据库的压测，就是直接使用sysbench工具就可以了，这一篇文章我来带着大家学习一下这个压测工具的使用，大家学习完这一讲，完全可以自己本地装一个MySQL数据库，然后自己压测一下试一试。

3、在linux上安装sysbench工具

首先你需要有一台linux机器，如果你只有一个windows笔记本电脑，可以在里面装一个linux的虚拟机，然后你可以用如下的命令设置一下yum repo仓库，接着基于yum来安装sysbench就可以了，安装完成以后验证一下是否成功。

curl -s https://packagecloud.io/install/repositories/akopytov/sysbench/script.rpm.sh | sudo bash

sudo yum -y install sysbench

sysbench --version

如果上面可以看到sysbench的版本号，就说明安装成功了。

4、数据库压测的测试用例

接着我们需要在自己的数据库里创建好一个测试库，我们可以取个名字叫做test_db，同时创建好对应的测试账号，可以叫做test_user，密码也是test_user，让这个用户有权限可以访问test_db。

然后我们将要基于sysbench构建20个测试表，每个表里有100万条数据，接着使用10个并发线程去对这个数据库发起访问，连续访问5分钟，也就是300秒，然后对其进行压力测试。

5、基于sysbench构造测试表和测试数据

sysbench --db-driver=mysql --time=300 --threads=10 --report-interval=1 --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=test_user --mysql-password=test_user --mysql-db=test_db --tables=20 --table_size=1000000 oltp_read_write --db-ps-mode=disable prepare

上面我们构造了一个sysbench命令，给他加入了很多的参数，现在我们来 解释一下这些参数，相信很多参数大家自己看到也就大致明白什么意思了：

--db-driver=mysql：这个很简单，就是说他基于mysql的驱动去连接mysql数据库，你要是oracle，或者sqlserver，那自然就是其他的数据库的驱动了

--time=300：这个就是说连续访问300秒

--threads=10：这个就是说用10个线程模拟并发访问

--report-interval=1：这个就是说每隔1秒输出一下压测情况

--mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=test_user --mysql-password=test_user：这一大串，就是说连接到哪台机器的哪个端口上的MySQL库，他的用户名和密码是什么

--mysql-db=test_db --tables=20 --table_size=1000000：这一串的意思，就是说在test_db这个库里，构造20个测试表，每个测试表里构造100万条测试数据，测试表的名字会是类似于sbtest1，sbtest2这个样子的

oltp_read_write：这个就是说，执行oltp数据库的读写测试

--db-ps-mode=disable：这个就是禁止ps模式

最后有一个prepare，意思是参照这个命令的设置去构造出来我们需要的数据库里的数据，他会自动创建20个测试表，每个表里创建100万条测试数据，所以这个工具是非常的方便的。

6、对数据库进行360度的全方位测试

测试数据库的综合读写TPS，使用的是oltp_read_write模式（大家看命令中最后不是prepare，是run了，就是运行压测）：

```
sysbench --db-driver=mysql --time=300 --threads=10 --report-interval=1 --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=test_user --mysql-password=test_user --mysql-db=test_db --tables=20 --table_size=1000000 oltp_read_write --db-ps-mode=disable run
```

测试数据库的只读性能，使用的是oltp_read_only模式（大家看命令中的oltp_read_write已经变为oltp_read_only了）：

```
sysbench --db-driver=mysql --time=300 --threads=10 --report-interval=1 --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=test_user --mysql-password=test_user --mysql-db=test_db --tables=20 --table_size=1000000 oltp_read_only --db-ps-mode=disable run
```

测试数据库的删除性能，使用的是oltp_delete模式：

```
sysbench --db-driver=mysql --time=300 --threads=10 --report-interval=1 --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=test_user --mysql-password=test_user --mysql-db=test_db --tables=20 --
```

table_size=1000000 oltp_delete --db-ps-mode=disable run

测试数据库的更新索引字段的性能，使用的是oltp_update_index模式：

```
sysbench --db-driver=mysql --time=300 --threads=10 --report-interval=1 --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=test_user --mysql-password=test_user --mysql-db=test_db --tables=20 --table_size=1000000 oltp_update_index --db-ps-mode=disable run
```

测试数据库的更新非索引字段的性能，使用的是oltp_update_non_index模式：

```
sysbench --db-driver=mysql --time=300 --threads=10 --report-interval=1 --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=test_user --mysql-password=test_user --mysql-db=test_db --tables=20 --table_size=1000000 oltp_update_non_index --db-ps-mode=disable run
```

测试数据库的更新非索引字段的性能，使用的是oltp_update_non_index模式：

```
sysbench --db-driver=mysql --time=300 --threads=10 --report-interval=1 --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=test_user --mysql-password=test_user --mysql-db=test_db --tables=20 --table_size=1000000 oltp_update_non_index --db-ps-mode=disable run
```

测试数据库的插入性能，使用的是oltp_insert模式：

```
sysbench --db-driver=mysql --time=300 --threads=10 --report-interval=1 --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=test_user --mysql-password=test_user --mysql-db=test_db --tables=20 --table_size=1000000 oltp_insert --db-ps-mode=disable run
```

测试数据库的写入性能，使用的是oltp_write_only模式：

```
sysbench --db-driver=mysql --time=300 --threads=10 --report-interval=1 --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=test_user --mysql-password=test_user --mysql-db=test_db --tables=20 --table_size=1000000 oltp_write_only --db-ps-mode=disable run
```

使用上面的命令，sysbench工具会根据你的指令构造出各种各样的SQL语句去更新或者查询你的20张测试表里的数据，同时监测出你的数据库的压测性能指标，最后完成压测之后，可以执行下面的cleanup命令，清理数据。

sysbench --db-driver=mysql --time=300 --threads=10 --report-interval=1 --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=test_user --mysql-password=test_user --mysql-db=test_db --tables=20 --table_size=1000000 oltp_read_write --db-ps-mode=disable cleanup

7、压测结果分析

按照我们上面的命令，我们是让他每隔1秒都会输出一次压测报告的，此时他每隔一秒会输出类似下面的一段东西：

[22s] thds: 10 tps: 380.99 qps: 7312.66 (r/w/o: 5132.99/1155.86/1321.35) lat (ms, 95%): 21.33 err/s: 0.00 reconn/s: 0.00

我来给大家解释一下这是什么意思，首先他说的这是第22s输出的一段压测统计报告，然后是其他的一些统计字段：

thds: 10，这个意思就是有10个线程在压测
tps: 380.99，这个意思就是每秒执行了380.99个事务
qps: 7610.20，这个意思就是每秒可以执行7610.20个请求
(r/w/o: 5132.99/1155.86/1321.35)，这个意思就是说，在每秒7610.20个请求中，有5132.99个请求是读请求，1155.86个请求是写请求，1321.35个请求是其他的请求，就是对QPS进行了拆解
lat (ms, 95%): 21.33，这个意思就是说，95%的请求的延迟都在21.33毫秒以下
err/s: 0.00 reconn/s: 0.00，这两个的意思就是说，每秒有0个请求是失败的，发生了0次网络重连

这个压测结果会根据每个人的机器的性能不同有很大的差距，你要是机器性能特别高，那你可以开很多的并发线程去压测，比如100个线程，此时可能会发现数据库每秒的TPS有上千个，如果你的机器性能很低，可能压测出来你的TPS才二三十个，QPS才几百个，这都是有可能的。

另外在完成压测之后，最后会显示一个总的压测报告，我把解释写在下面了：

SQL statistics:

```
    queries performed:
        read: 1480084 // 这就是说在300s的压测期间执行了148万多次的读请求
        write: 298457 // 这是说在压测期间执行了29万多次的写请求
        other: 325436 // 这是说在压测期间执行了30万多次的其他请求
        total: 2103977 // 这是说一共执行了210万多次的请求
    // 这是说一共执行了10万多个事务，每秒执行350多个事务
    transactions: 105180( 350.6 per sec. )
    // 这是说一共执行了210万多次的请求，每秒执行7000+请求
    queries: 2103977 ( 7013.26 per sec. )
    ignored errors: 0 (0.00 per sec.)
    reconnects: 0 (0.00 per sec.)

// 下面就是说，一共执行了300s的压测，执行了10万+的事务
General staticstics:
    total time: 300.0052s
    total number of events: 105180

Latency (ms):
    min: 4.32 // 请求中延迟最小的是4.32ms
    avg: 13.42 // 所有请求平均延迟是13.42ms
    max: 45.56 // 延迟最大的请求是45.56ms
    95th percentile: 21.33 // 95%的请求延迟都在21.33ms以内
```

8、今日作业

今天希望大家可以完成一个作业，自己准备一台linux机器或者虚拟机，然后装一个mysql数据库，接着使用sysbench工具尝试一下数据库的压测，自己分析一下压测的报告和结果，感受一下你的数据库到底能抗多高的并发。

同时接下来我们还会给大家讲解在压测的过程中，如何去观察机器的其他重要的性能指标，比如说CPU、网络、内存、磁盘IO，等等。

End

狸猫技术窝精品专栏及课程推荐：

- 《从零开始带你成为消息中间件实战高手》
- 《21天互联网Java进阶面试训练营》（分布式篇）
- 《互联网Java工程师面试突击》（第1季）
- 《互联网Java工程师面试突击》（第3季）
- 《从零开始带你成为JVM实战高手》

图文 08 生产经验：在数据库的压测过程中，如何360度无死角观察机器性能？ 手机观看

410 人次阅读 2020-02-04 08:08:55

详情 评论

生产经验：在数据库的压测过程中，如何360度无死角观察机器性能？

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

1、除了QPS和TPS以外，我们还需要观察机器的性能

上一篇文章我们给大家讲解了如何使用sysbench这个工具非常方便的去对数据库进行压测，压测过后其实大家就会看到自己的数据库大概能抗下多少QPS和TPS了。

但是这里还得给大家说另外一个压测时的技巧，就是上一篇文章里我们是使用了10个线程去压测数据库，如果你的机器性能很高，然后你觉得10个线程没法压测出来数据库真实的最高负载能力，你其实可以在sysbench中不停的增加线程的数量，比如使用20个线程，甚至100个线程去并发的访问数据库，直到发现数据库的QPS和TPS上不去了。

当然，这个不停的提高线程数量，不停的让数据库承载更高的QPS的过程，还需要配合我们对机器性能表现的观察来做，不能盲目的不停的增加线程去压测数据库。

这篇文章，我们就是要讲解在压测过程中，如何同时观察机器的性能表现，从而来决定是否要继续增加线程数量去压测数据库。

2、为什么在不停增加线程数量的时候，要密切关注机器性能?

我们先来解答一个问题，就是在压测的时候我们需要不停的增加线程的数量去让数据库承载更高的QPS，一直到最后看看数据库到底最高可以承载多高的QPS。

那么在这个过程中，为什么必须要密切的关注机器的性能呢?

我们来给大家举两个例子，相信大家看完就明白这个问题了。

首先，假设数据库当前抗下了每秒2000的QPS，同时这个时候机器的CPU负载、内存负载、网络负载、磁盘IO负载，都在正常的范围内，负载相对较高一些，但是还没有达到这些硬件的极限，那么我们可以认为这台数据库在高峰期抗到每秒2000的QPS，是没有问题的。

但是如果你一直不停的在压测过程中增加sysbench的线程数量，然后数据库此时勉强抗到了每秒5000的QPS了，但是这个时候你发现机器的CPU已经满负荷运行了，内存使用率特别高，内存都快要不够了，然后网络带宽几乎被打满了，磁盘IO的等待时间特别长，这个时候说明机器已经到了极致了，再搞下去，机器都快挂了。

所以这个时候你压测出来的5000 QPS是没什么代表性的，因为在生产环境根本不可能让数据库抗下这么高的QPS，因为到这么高的QPS就说明你的数据库几乎已经快要挂掉了，这是不现实的。

所以说，在压测的过程中，必须是不停的增加sysbench的线程数量，持续的让数据库承载更高的QPS，同时密切关注机器的CPU、内存、磁盘和网络的负载情况，在硬件负载情况比较正常的范围内，哪怕负载相对较高一些，也还是可以继续增加线程数量和提高数据库的QPS的。

然后当你不停的增加线程数量，发现在数据库抗下一个QPS的数值的同时，机器的CPU、内存、网络和磁盘的负载已经比较高了，到了一个有一定风险的临界值的了，此时就不能继续增加线程数量和提高数据库抗下的QPS了。

接着我们今天就来给大家分析一下，在压测的过程中，需要使用哪些linux命令去观察机器的性能情况，以及机器的CPU、内存、磁盘和网络在什么样的负载下是正常的，在什么样的负载下就是比较危险的了。

3、压测时如何观察机器的CPU负载情况?

先来看一个最最常用的监测linux机器性能的命令，就是top命令，直接在linux命令行只能够输入top指令就可以了，然后我们这里来给大家解释一下，top指令展示出来的各种信息都是什么意思。

首先我们会看到如下一行信息：

top - 15:52:00 up 42:35, 1 user, load average: 0.15, 0.05, 0.01

先来解释一下这行信息，这行信息是最直观可以看到机器的cpu负载情况的，首先15:52:00指的是当前时间，up 42:35指的是机器已经运行了多长时间，1 user就是说当前机器有1个用户在使用。

最重要的是load average: 0.15, 0.05, 0.01这行信息，他说的是CPU在1分钟、5分钟、15分钟内的负载情况。

这里要给大家着重解释一下这个CPU负载是什么意思，假设我们是一个4核的CPU，此时如果你的CPU负载是0.15，这就说明，4核CPU中连一个核都没用满，4核CPU基本都很空闲，没啥人在用。

如果你的CPU负载是1，那说明4核CPU中有一个核已经被使用的比较繁忙了，另外3个核还是比较空闲一些。要是CPU负载是1.5，说明有一个核被使用繁忙，另外一个核也在使用，但是没那么繁忙，还有2个核可能还是空闲的。

如果你的CPU负载是4，那说明4核CPU都被跑满了，如果你的CPU负载是6，那说明4核CPU被繁忙的使用还不够处理当前的任务，很多进程可能一直在等待CPU去执行自己的任务。

这个就是CPU负载的概念和含义。

所以大家现在知道了，上面看到的load average实际上就是CPU在最近1分钟，5分钟，15分钟内的平均负载数值，上面都是0.15之类的，说明CPU根本就没怎么用。

但是如果你在压测的过程中，发现4核CPU的load average已经基本达到3.5，4了，那么说明几个CPU基本都跑满了，在满负荷运转，那么此时你就不要再继续提高线程的数量和增加数据库的QPS了，否则CPU负载太高是不合理的。

4、压测时如何观察机器的内存负载情况?

在你执行top命令之后，中间我们跳过几行内容，可以看到如下一行内容：

Mem: 33554432k total, 20971520k used, 12268339 free, 307200k buffers

这里说的就是当前机器的内存使用情况，这个其实很简单，明显可以看出来就是总内存大概有32GB，已经使用了20GB左右的内存，还有10多G的内存是空闲的，然后有大概300MB左右的内存用作OS内核的缓冲区了。

对于内存而言，同样是要在压测的过程中紧密的观察，一般来说，如果内存的使用率在80%以内，基本都还能接受，在正常范围内，但是如果你的机器的内存使用率到了70%~80%了，就说明有点危险了，此时就不要继续增加压测的线程数量和QPS了，差不多就可以了。

5、压测时如何观察机器的磁盘IO情况?

接着我们说说如何在压测的时候观察机器的磁盘IO的情况?

这里会使用dstat命令，我们之前给大家讲过几个磁盘IO相关的指标，包括存储的IO吞吐量、IOPS这些，我们下面就看看这里是如何查看的。

使用dstat -d命令，会看到如下的东西：

-dsk/total -
read writ
103k 211k
  0   11k

在上面可以清晰看到，存储的IO吞吐量是每秒钟读取103kb的数据，每秒写入211kb的数据，像这个存储IO吞吐量基本上都不算多的，因为普通的机械硬盘都可以做到每秒钟上百MB的读写数据量。

使用命令：dstat -r，可以看到如下的信息

--io/total-
read writ
0.25 31.9
  0   253
  0   39.0

他的这个意思就是读IOPS和写IOPS分别是多少，也就是说随机磁盘读取每秒钟多少次，随机磁盘写入每秒钟执行多少次，大概就是这个意思，一般来说，随机磁盘读写每秒在两三百次都是可以承受的。

所以在这里，我们就需要在压测的时候密切观察机器的磁盘IO情况，如果磁盘IO吞吐量已经太高了，都达到极限的每秒上百MB了，或者随机磁盘读写每秒都到极限的两三百次了，此时就不要继续增加线程数量了，否则磁盘IO负载就太高了。

6、压测时观察网卡的流量情况

接着我们可以使用dstat -n命令，可以看到如下的信息：

-net/total-
recv send
16k 17k

这个说的就是每秒钟网卡接收到流量有多少kb，每秒钟通过网卡发送出去的流量有多少kb，通常来说，如果你的机器使用的是千兆网卡，那么每秒钟网卡的总流量也就在100MB左右，甚至更低一些。

所以我们在压测的时候也得观察好网卡的流量情况，如果网卡传输流量已经到了极限值了，那么此时你再怎么提高sysbench线程数量，数据库的QPS也上不去了，因为这台机器每秒钟无法通过网卡传输更多的数据了。

7、今天的一点总结和作业

今天的文章我们来做一点总结，今天给大家介绍了在数据库压测的过程中，你必须不停的增加sysbench的线程数量，增加数据库抗下的QPS，同时通过各种命令观察机器的CPU、内存、磁盘和网络的负载情况，如果你发现某个硬件负载已经很高了，此时就可以不再提高数据库的QPS了。

在硬件的一定合理的负载范围内，把数据库的QPS提高到最大，这就是数据库压测的时候最合理的一个极限QPS值，而不是不管机器的各个硬件的负载，盲目的不停的增加sysbench的线程数量，不停的让数据库增加可以抗下的QPS的数值。

今天同样给大家布置一个小作业，大家可以自己本地笔记本电脑装一个数据库，然后基于sysbench去压测，不停的提高线程数量，不停的提高数据库抗下的QPS，同时观察你的机器的CPU、内存、磁盘和网络的各项负载

然后希望大家对自己压测的情况做一个总结，发布到评论区跟其他同学进行交流。

End

图文 09 生产经验：如何为生产环境中的数据库部署监控系统？

手机观看

443 人次阅读 2020-02-05 08:01:17

详情 评论

生产经验：如何为生产环境中的数据库部署监控系统？

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

1、生产环境的数据库可不能裸奔啊！

之前我们已经给大家讲解完了数据库的压测相关的知识，想必大家只要利用我们讲解的知识，在自己的公司里，哪怕DBA团队、QA团队都没法给你提供专业的数据库压测技术支持，但是大家手上拿到一个数据库之后，其实自己也可以通过各种工具和命令，非常好的完成一台机器上的数据库的压测了。

你应该可以心里非常有数，一台什么样配置的机器，部署了一个数据库之后，利用sysbench构造了多少个表和数据量，然后模拟了多少个线程压测的时候，机器的各项硬件负载在可以接受的范围内时，数据库的QPS和TPS可以压测到多高。

这个时候你大致就明白你的数据库在高峰时期最多可以让他去承受多少QPS和TPS了。

但是搞定压测之后，难道大家就想直接开始开发你的Java系统？直接让你的系统连接到MySQL上去执行各种CRUD的SQL语句？然后接着就开始拼命写各种Java代码和SQL语句，写好之后就找QA进行测试，然后部署到线上生产环境，接着就万事大吉了，不管数据库了？

这种做法可能目前很多公司和团队都是这样做的，但是如果你仅仅是这么搞是绝对不行的。因为实际上我们需要对线上系统进行完善的监控，不光是对你开发的Java系统进行监控，还得对你的数据库进行监控，包括对CPU、内存、网络、磁盘IO、慢查询、QPS、TPS的监控。

因为如果你不对你的数据库做任何监控，那么有可能你的数据库CPU负载已经很高了，或者磁盘IO已经达到极限了，你都不知道，结果你还是一如既往的运行你的Java系统，有一天可能你的数据库突然挂了你都没反应过来！

所以今天我们就带着大家来一步步搭建一下生产环境数据库的可视化监控平台，我们会基于Prometheus+Grafana来搭建。

当然在公司里，如果要针对数据库搭建一个统一的可视化监控平台，这个活儿往往是DBA团队负责的，但是不管如何，我们这里也要对这个数据库可视化监控的技术有一定的了解。

2、简单介绍一下Prometheus和Grafana是什么

我们先给大家简单介绍一下Prometheus和Grafana两个系统分别是什么。

简单来说，Prometheus其实就是一个监控数据采集和存储系统，他可以利用监控数据采集组件（比如mysql_exporter）从你指定的MySQL数据库中采集他需要的监控数据，然后他自己有一个时序数据库，他会把采集到的监控数据放入自己的时序数据库中，其实本质就是存储在磁盘文件里。

我们采集到了MySQL的监控数据还不够，现在我们还要去看这些数据组成的一些报表，所以此时就需要使用Grafana了，Grafana就是一个可视化的监控数据展示系统，他可以把Prometheus采集到的大量的MySQL监控数据展示成各种精美的报表，让我们可以直观的看到MySQL的监控情况。

其实不光是对数据库监控可以采用Prometheus+Grafana的组合，对你开发出来的各种Java系统、中间件系统，都可以使用这套组合去进行可视化的监控，无非就是让Prometheus去采集你的监控数据，然后用Grafana展示成报表而已。

3、安装和启动Prometheus

之前给大家说过，让大家自己准备一个linux机器，如果你是windows笔记本电脑，可以自己装一个linux虚拟机。我们就基于一台linux机器来部署Prometheus和Grafana，至于MySQL的安装，这个非常的简单，大家在网上很容易搜索到。

首先大家需要下载3个压缩包，在下面链接：

http://cactifans.hi-www.com/prometheus/

大家可以下载到下面两个压缩包，这里prometheus就是用来部署监控系统自己的，然后node_exporter是用来采集MySQL数据库所在机器的CPU、内存、网络、磁盘之类的监控数据的：

prometheus-2.1.0.linux-amd64.tar.gz
node_exporter-0.15.2.linux-amd64.tar.gz

接着大家可以通过下面的链接下载第三个压缩包：mysqld_exporter-0.10.0.linux-amd64.tar.gz，这个mysqld_exporter就是用来采集MySQL数据库自己的一些监控数据的，比如SQL性能、连接数量之类的：

https://github.com/prometheus/mysqld_exporter/releases/download/v0.10.0/mysqld_exporter-0.10.0.linux-amd64.tar.gz

接着需要解压缩上面的几个包，参照我如下的命令来做就可以了：

```
mkdir /data
mkdir /root

tar xvf prometheus-2.1.0.linux-amd64.tar -C /data
tar xf node_exporter-0.15.2.linux-amd64.tar -C /root
tar xf mysqld_exporter-0.10.0.linux-amd64.tar.gz -C /root

cd /data
mv prometheus-2.1.0.linux-amd64/ prometheus
```

cd /prometheus

vi prometheus.yml，接下来修改prometheus的配置文件，其实主要是在scrape_configs下面加入一大段自定义的配置，因为他需要去采集MySQL数据库本身和MySQL所在机器的监控数据：

```
scrape_configs:
 - file_sd_configs:
   -files:
     - host.yml
   job_name: Host
   metrics_path: /metrics
   relabel_configs:
   - source_labels: [__address__]
     regex: (.*)
     target_label: instance
     replacement: $1
   - source_labels: [__address__]
     regex: (.*)
     target_label: __address__
     replacement: $1:9100
 - file_sd_configs:
   - files:
     - mysql.yml
   job_name: MySQL
   metrics_path: /metrics
   relabel_configs:
   - source_labels: [__address__]
     regex: (.*)
     target_label: instance
     replacement: $1
```

```
  - source_labels: [__address__]
    regex: (.*)
    target_label: __address__
    replacement: $1:9104

- job_name: prometheus
  static_configs:
  - targets:
    - localhost: 9090
```

上面的配置文件写好之后，就可以启动Prometheus了，不过大家仔细看几遍上面的配置信息，因为我在写文章的时候不太方便，都是直接手敲出来的，可能会有少数配置错误，如果大家有发现配置文件错误的地方，及时在后台评论区告诉我。

接着必须要在/data/prometheus目录中，去执行启动命令：

/data/prometheus/prometheus --storage.tsdb.retention=30d &，这里的30d是说你的监控数据保留30天的。启动之后，就可以在浏览器中访问9090端口号去查看prometheus的主页了。

因为我们部署和安装Prometheus和Grafana的过程比较多，所以拆分为两篇文章，今天同步更新的第二篇文章里，会把剩余的Grafana的安装部署过程，以及监控配置和采集的过程都讲完。

End

图文 10 生产经验：如何为数据库的监控系统部署可视化报表系统？ 手机观看

375 人次阅读 2020-02-05 08:08:37

详情 评论

生产经验：如何为数据库的监控系统部署可视化报表系统？

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

1、部署Grafana

上一篇文章我们讲解到安装好了Prometheus，接着我们来继续 讲解如何安装Grafana，首先要从下面的地址下载grafana-4.6.3.linux-x64.tar.gz，然后一步一步的执行下面的命令，完成他的启动。

https://s3-us-west-2.amazonaws.com/grafana-releases/release/grafana-4.6.3.linux-x64.tar.gz

tar xf grafana-4.6.3.linux-x64.tar.gz -C /data/prometheus
cd /data/prometheus
mv grafana-4.6.3 grafana

cd /data/prometheus/grafana
./bin/grafana-server &

接着就完成了grafana的启动，然后可以通过浏览器访问3000端口，默认的用户名和密码是admin/admin。接着在Grafana左侧菜单栏里有一个Data Sources，点击里面的一个按钮是Add data source，就是添加一个数据源。

然后在界面里输入你的数据源的名字是Prometheus，类型是Prometheus，HTTP URL地址是http://127.0.0.1:9090，其他的都用默认的配置就行了，接下来Grafana就会自动从Prometheus里获取监控数据和展示了。

接着需要安装Grafana的仪表盘组件，首先需要下载grafana-dashboards-1.6.1.tar.gz，用如下的链接即可：
https://github.com/percona/grafana-dashboards/archive/v1.6.1.tar.gz。

接着执行一系列的命令去安装grafana-dashboard组件。

tar xvf grafana-dashboards-1.6.1.tar.gz
cd grafana-dashboards-1.6.1
updatedb
locate json |grep dashboards/

这个时候会看到一大堆的json文件，就是各种不同的仪表盘对应的json配置文件，你可以把这些json配置文件通过WinSCP之类的工具从linux机器上拖到你的windows电脑上来，因为需要通过浏览器上传他们。

接着在grafana页面中，可以看到最上面有一个Home按钮，点击一下进入一个界面，你会看到一个Import Dashboard的按钮，就是说可以导入一些仪表盘，这个时候就是要导入刚才看到的一大堆的json文件。

你点击Upload json file按钮，就会出现一个界面让你上传一个一个的json文件，然后你就依次上传，接着grafana中就会出现一大堆的仪表盘了，比如机器的CPU使用率的仪表盘，磁盘性能仪表盘，磁盘空间仪表盘，MySQL监控仪表盘，等等。

2、添加MySQL机器的监控

首先我们如果想要让Prometheus去采集MySQL机器的监控数据（CPU、内存、磁盘、网络，等等），然后让Grafana可以展示出来，那么就必须先添加Prometheus对MySQL机器的监控。

首先必须要在MySQL机器上解压缩和启动node_exporter，这启动之后是个linux进程，他会自动采集这台linux机器上的CPU、磁盘、内存、网络之类的各种监控数据，其实本质你可以理解为通过我们之前讲解的那些linux命令，就可以采集到一切你想要的linux机器的监控数据。

```
tar xf node_exporter-0.15.2.linux-amd64.tar
mv node_exporter-0.15.2.linux-amd64 node_exporter
cd node_exporter
nohup ./node_exporter &
```

到这一步为止，我们就在MySQL所在的机器上启动了一个node_exporter了，他就会自动采集这台机器的CPU、磁盘、内存、网络的监控数据，但是此时还不够，因为Prometheus上还没加入对这台机器的监控。

此时我们应该还记得，之前在Prometheus的yml配置文件中，我们已经定义了一个host监控项，他就是用来监控机器的，他的配置文件是host.yml，此时我们可以编辑一下这个host.yml配置文件，加入mysql所在机器的地址就可以了

```
vi host.yml
```

```
- labels:
   service: test
 targets:
 - 127.0.0.1
```

接着Prometheus就会跟MySQL机器上部署的node_exporter进行通信，源源不断的获取到这台机器的监控数据，写入自己的时序数据库中进行存储。接着我们就可以打开Grafana的页面，此时你就可以看到这台机器的相关性能监控了。

3、添加MySQL数据库的监控

接着我们同样需要在MySQL所在机器上再启动一个mysqld_exporter的组件，他负责去采集MySQL数据库自己的一些监控数据，我们看下面的指令就可以了。

tar xf mysqld_exporter-0.10.0.linux-amd64.tar
mv mysqld_exporter-0.10.0.linux-amd64 mysqld_exporter

接着需要配置一些环境变量，去设置mysqld_exporter要监控的数据库的地址信息，看下面配置了账号、密码以及地址和端口号

export DATA_SOURCE_NAME='root:root@(127.0.0.1:3306)/'
echo "export DATA_SOURCE_NAME='root:root@(127.0.0.1:3306)/'" >> /etc/profile

接着启动mysqld_exporter

cd mysqld_exporter

nohup ./mysqld_exporter --collect.info_schema.processlist --collect.info_schema.innodb_tablespaces --collect.info_schema.innodb_metrics --collect.perf_schema.tableiowaits --collect.perf_schema.indexiowaits --collect.perf_schema.tablelocks --collect.engine_innodb_status --collect.perf_schema.file_events --collect.info_schema.processlist --collect.binlog_size --collect.info_schema.clientstats --collect.perf_schema.eventswaits &

上面的启动命令指定了大量的选项去开启一些监控的采集，这些命令也都是我手敲的，因为目前写作环境的一些不便利的因素，所以只能是如此，如果大家发现有什么小的错误，可以评论区后台告诉我。

接着这个mysqld_exporter进程就会自动采集MySQL自己的监控数据了，然后我们还需要在Prometheus里配置一下他去跟mysqld_exporter通信获取数据以及存储，然后Grafana才能看到对应的报表。

vi /data/prometheus/mysql.yml

```
- labels:
   service: mysql_test
 targets:
 - 127.0.0.1
```

接着我们在Grafana中就可以看到MySQL的各种监控数据了。

4、一个作业

今天我想留给大家一个小作业，希望大家可以参考今天的两篇文章，动手搭建一下数据库的监控系统，然后可以用sysbench做一下压测，在压测过程中，可以直接看看Grafana上的机器以及MySQL的各项监控指标。

这个过程没什么难度，但是可能会遇到一些操作性的问题，如果大家搭建的过程中发现什么问题，不要直接把报错的截图贴在评论区，你可以先去上网查查，错误在哪里

如果真的有问题的话，我后续会想办法解决一下错误，然后更新出来的。

另外，我希望大家思考一个问题，大家可以去看看自己公司里的数据库有没有做过压测？可视化监控做了吗？是怎么做的？自己项目的数据库平时的一些机器负载和QPS、TPS都是多少？自己对数据库是否有一个较为全面的掌握？

希望每个人都去看看，然后发表在评论区里跟大家一起分享和交流。

End

小鹅通提供技术支持

图文 11 从数据的增删改开始讲起，回顾一下Buffer Pool在数据库里的地位

手机观看

340 人次阅读 2020-02-06 08:40:11

详情 评论

从数据的增删改开始讲起，回顾一下Buffer Pool在数据库里的地位

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

1、一切从数据的增删改开始讲起

好了，到这一讲为止，我们实际上已经初步的讲解了一下MySQL的整体架构设计原理，大家对于MySQL内部包含哪些组件，我们平时更新数据以及查询数据的时候，大致都是怎么做的，都已经有一个比较高层次的了解了。

另外现在我们初步的了解了MySQL的架构原理之后，还给大家介绍了一些我们的数据库相关的生产经验，就是对于任何一个项目，数据库都需要选择好合适的机器，同时做好压测，并且有一个完善的可视化监控系统。

现在可以理解为每个人手头都有了一个可用的数据库，而且对数据库的整体架构原理都有了一定的理解了。那么接下来，我们这个专栏一共有100多讲的内容，我们接着当然要细细的讲解数据库的方方面面了

那我们应该从哪个环节开始入手呢?

当然是从数据库的增删改开始了，因为当你手头有了一个数据库之后，你必然就会去开发一个系统，系统就直接基于数据库做各种增删改查的操作，实现各种各样的业务逻辑

而任何一个系统在使用数据库的时候，一定是从插入数据开始的，也就是首先先会对数据进行增删改的操作。

当你的数据库中有了数据之后，接着才会执行各种各样的查询操作。

所以我们专栏的讲解顺序，就按照你手头有了一个经过压测的、有完善监控的数据库之后，你开发的系统使用数据库的顺序来讲解，先讲解系统对数据库执行各种增删改操作时背后对应的内幕原理，以及事务的原理，包括锁的底层机制，然后讲解你有了数据之后，执行各种复杂的查询操作的时候，涉及到的索引底层原理，查询优化的底层原理。

当然这个中间我们会穿插各种各样的生产实践的案例，就跟我之前讲解的《从0开始带你成为JVM实战高手》专栏一样。

然后讲解完这些之后，我们再来讲解平时我们在开发系统的时候，如何进行数据库的建模，在数据库建模的时候，应该如何注意字段类型、索引类型的一些问题，如何保证数据库避免死锁、高性能的运行。

接着我们再讲解一些高阶的数据库架构设计，比如说主从架构设计以及分库分表架构设计，包括一些生产实践的案例。

所以上面的这些就是我们专栏接下来将要讲解的顺序，这里要给大家提前通知一下，我在实际讲解的过程中，会增加很多内容，比如接下来好几讲都是深度分析Buffer Pool的内容，实际上在原来的大纲中都是没有的。

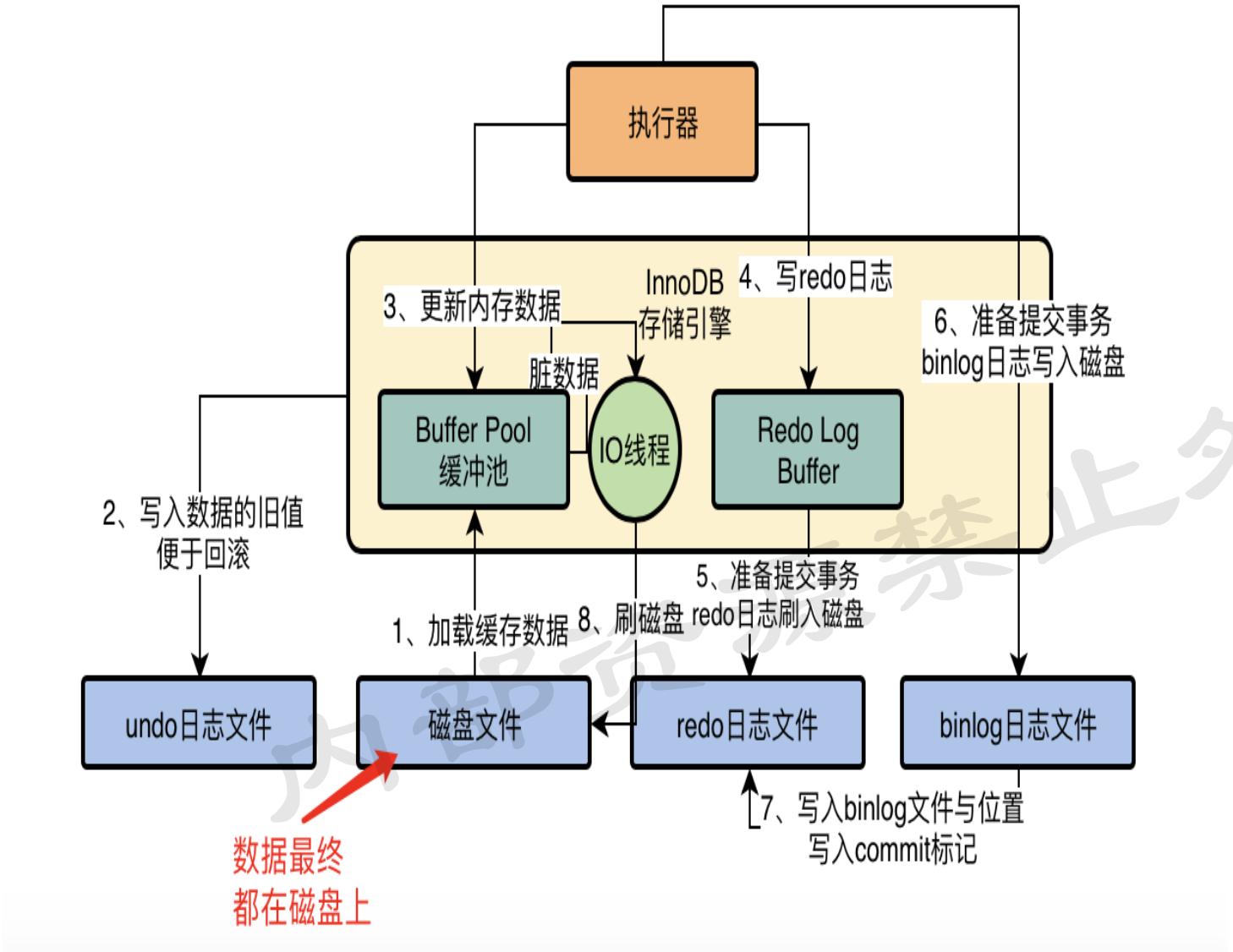
另外我接下来讲解的过程中，还可能随时会对大纲中原有内容的顺序做出调整，比如说我在讲解完Buffer Pool之后，接着可能直接会深入讲解redo log、undo log、binlog这些机制，同时接着讲解事务机制，锁机制，底层数据存储机制。

然后这些都讲完之后，才是讲解索引和查询优化的内容，所以希望大家能明白我们随时会对大纲内容做出额外的扩充，以及我们随时会调整大纲内容的顺序。

好，那么从这篇文章开始，让我们一起来探索数据库的各种底层机制和生产实践案例吧!

2、回顾一下Buffer Pool是个什么东西?

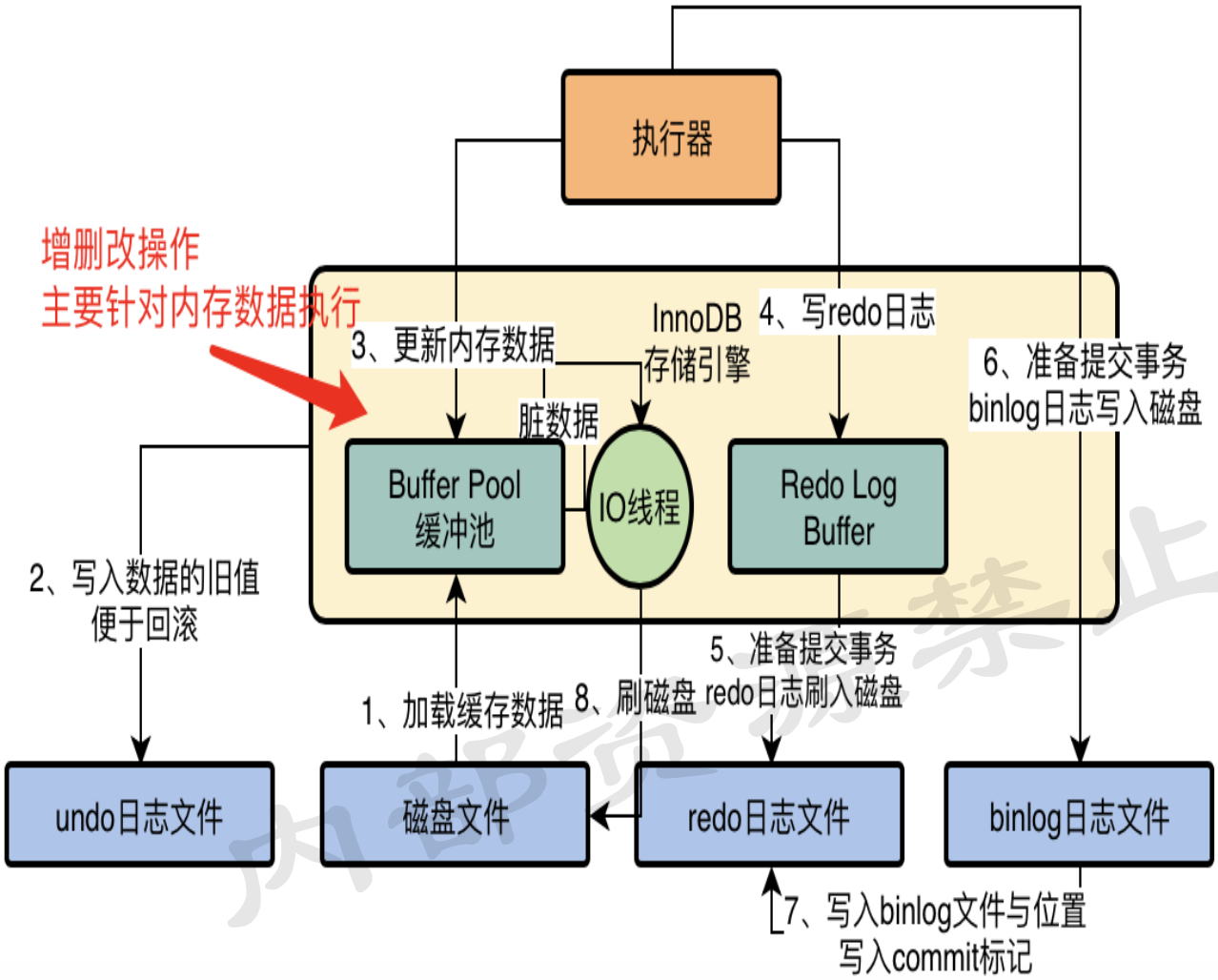
现在我们先来回顾一下数据库中的Buffer Pool是个什么东西？其实他是一个非常关键的组件，因为我们通过之前的讲解都知道一点，那就是数据库中的数据实际上最终都是要存放在磁盘文件上的，如下图所示。



但是我们在对数据库执行增删改操作的时候，不可能直接更新磁盘上的数据的，因为如果你对磁盘进行随机读写操作，那速度是相当的慢，随便一个大磁盘文件的随机读写操作，可能都要几百毫秒。如果要是那么搞的话，可能你的数据库每秒也就只能

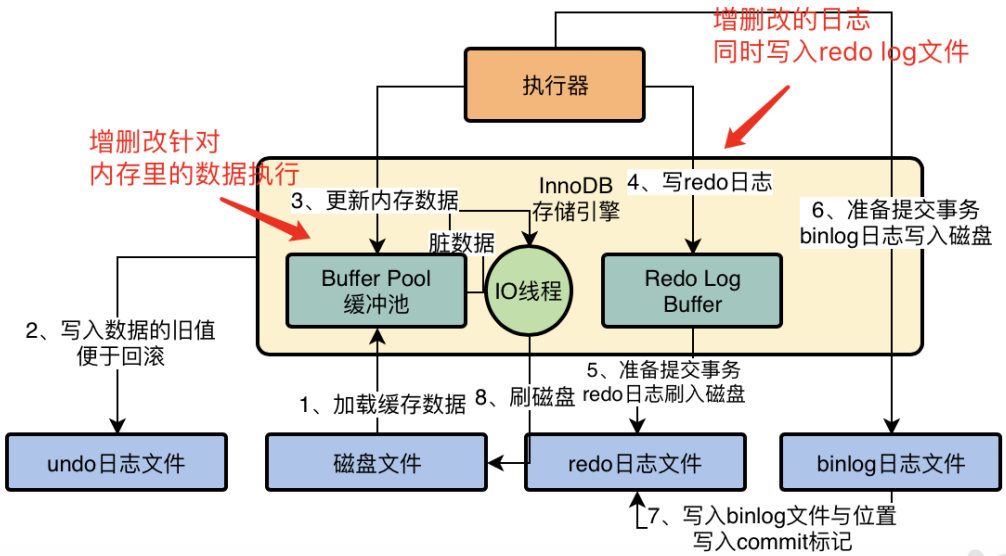
处理几百个请求了！

之前我们也都讲解过了，你在对数据库执行增删改操作的时候，实际上主要都是针对内存里的Buffer Pool中的数据进行的，也就是你实际上主要是对数据库的内存里的数据结构进行了增删改，如下图所示。



当然，我们之前都说过，其实每个人都担心一个事，就是你在数据库的内存里执行了一堆增删改的操作，内存数据是更新了，但是这个时候如果数据库突然崩溃了，那么内存里更新好的数据不是都没了吗？

所以其实之前我们开头就用了很多篇幅讲这个问题，MySQL就怕这个问题，所以引入了一个redo log机制，你在对内存里的数据进行增删改的时候，他同时会把增删改对应的日志写入redo log中，如下图。



万一你的数据库突然崩溃了，没关系，只要从redo log日志文件里读取出来你之前做过哪些增删改操作，瞬间就可以重新把这些增删改操作在你的内存里执行一遍，这就可以恢复出来你之前做过哪些增删改操作了。

当然对于数据更新的过程，他是有一套严密的步骤的，还涉及到undo log、binlog、提交事务、buffer pool脏数据刷回磁盘，等等。我们之前都讲过了，这里不再重复，仅仅是带着大家重新回顾一下数据库中的Buffer Pool这个东西。

3、Buffer Pool的一句话总结

所以这里我们简单对Buffer Pool这个东西做一下总结，他其实是数据库中我们第一个必须要搞清楚的核心组件，因为增删改操作首先就是针对这个内存中的Buffer Pool里的数据执行的，同时配合了后续的redo log、刷磁盘等机制和操作。

所以Buffer Pool就是数据库的一个内存组件，里面缓存了磁盘上的真实数据，然后我们的Java系统对数据库执行的增删改操作，其实主要就是对这个内存数据结构中的缓存数据执行的。

这一篇文章我们先对Buffer Pool这个东西的定位做一个简单的回顾，下一篇文章我们来分析一下Buffer Pool这个内存数据结构里到底包含了一些什么东西。

End

图文 12 Buffer Pool这个内存数据结构到底长个什么样子?

手机观看

328 人次阅读 2020-02-06 08:47:19

详情 评论

Buffer Pool这个内存数据结构到底长个什么样子?

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

1、如何配置你的Buffer Pool的大小?

首先我们来看看，我们应该如何配置你的Buffer Pool到底有多大呢?

因为Buffer Pool本质其实就是数据库的一个内存组件，你可以理解为他就是一片内存数据结构，所以这个内存数据结构肯定是有一定的大小的，不可能是无限大的。

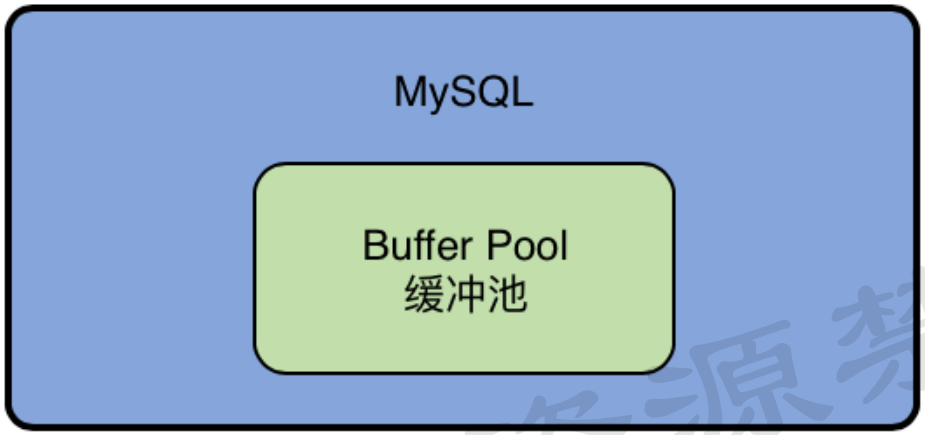
这个Buffer Pool默认情况下是128MB，还是有一点偏小了，我们实际生产环境下完全可以对Buffer Pool进行调整。

比如我们的数据库如果是16核32G的机器，那么你就可以给Buffer Pool分配个2GB的内存，使用下面的配置就可以了。

[server]
innodb_buffer_pool_size = 2147483648

如果有的朋友不知道数据库的配置文件在哪里以及如何修改其中的配置，那建议可以先在网上搜索一些MySQL入门的资料去看看，其实这都是最基础和简单的。

我们先来看一下下面的图，里面就画了数据库中的Buffer Pool内存组件。

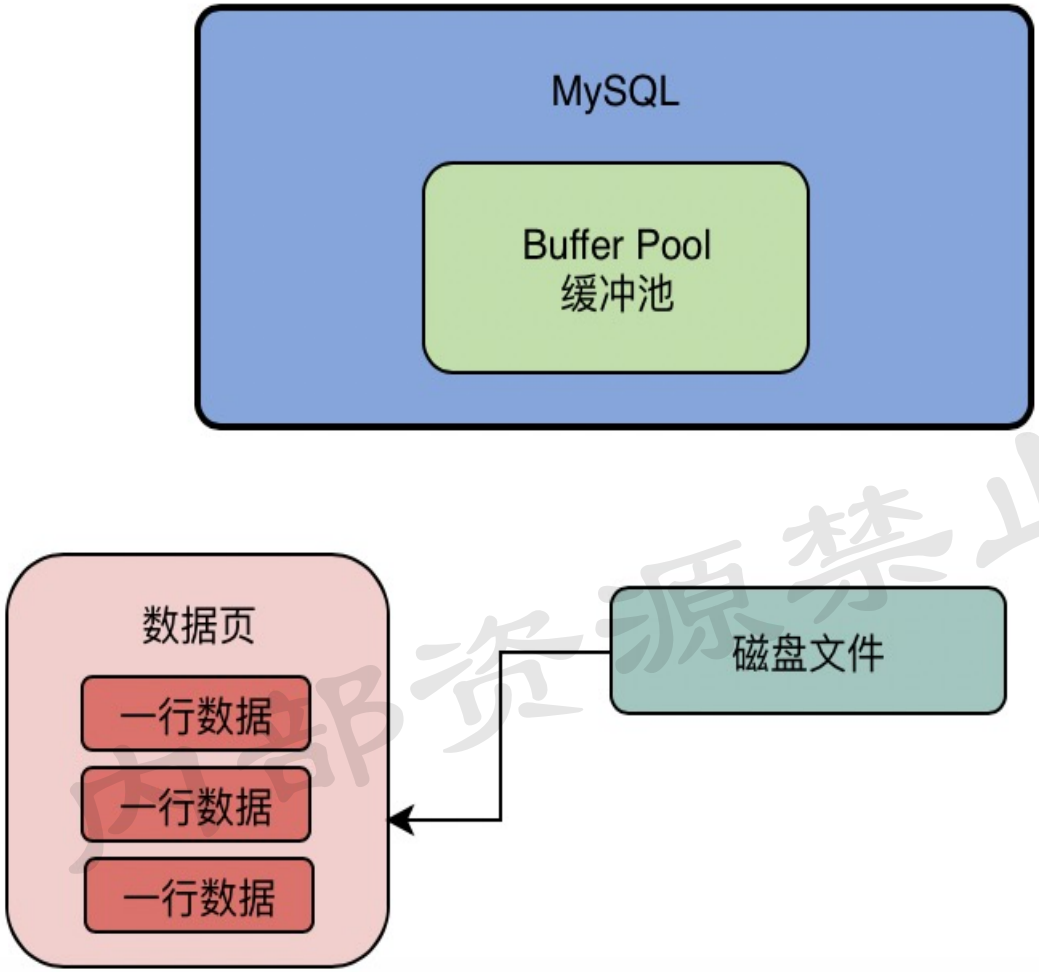


2、数据页：MySQL中抽象出来的数据单位

接着我们来看下一个问题，假设现在我们的数据库中一定有一片内存区域是Buffer Pool了，那么我们的数据是如何放在Buffer Pool中的?

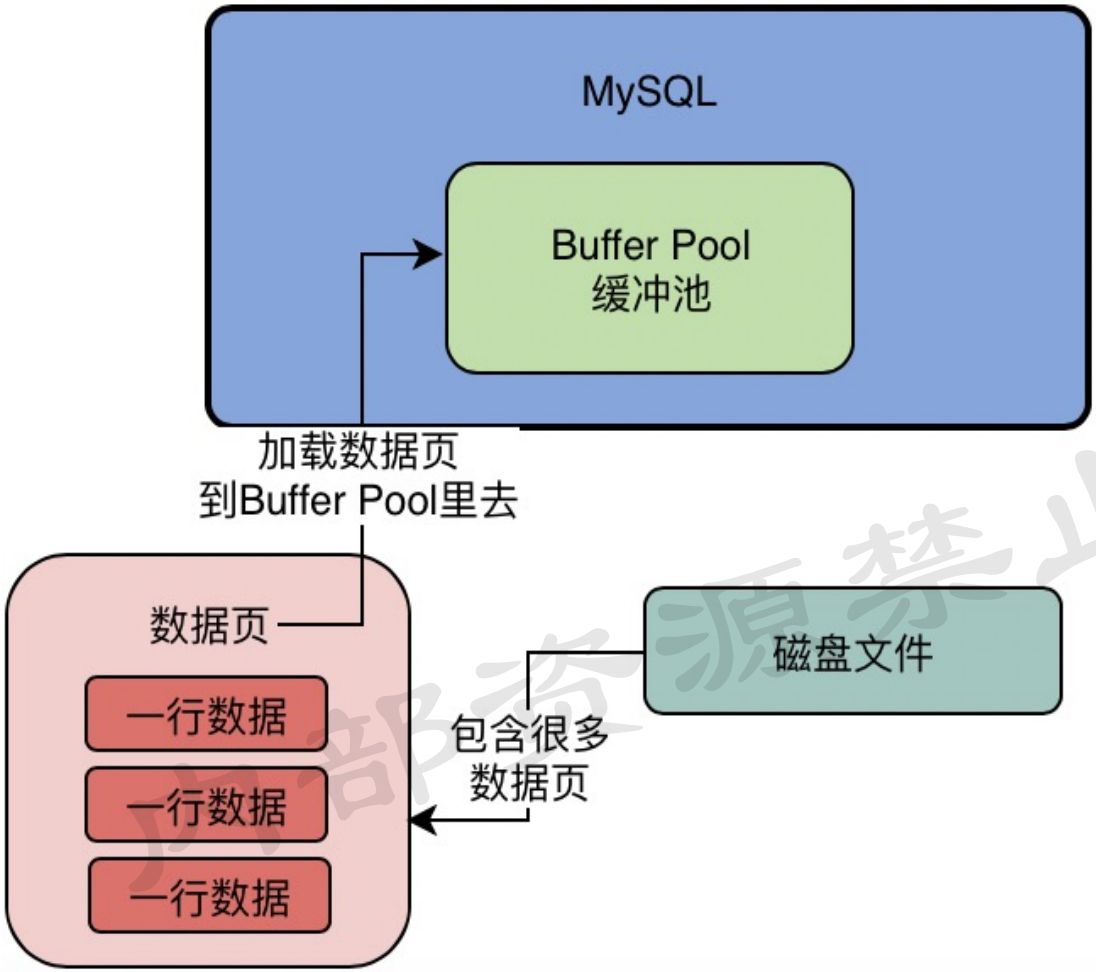
我们都知道数据库的核心数据模型就是表+字段+行的概念，也就是说我们都知道数据库里有一个一个的表，一个表有很多字段，然后一个表里有很多行数据，每行数据都有自己的字段值。所以大家觉得我们的数据是一行一行的放在Buffer Pool里面的吗?

这就明显不是了，实际上MySQL对数据抽象出来了一个数据页的概念，他是把很多行数据放在了一个数据页里，也就是说我们的磁盘文件中就是会有很多的数据页，每一页数据里放了很多行数据，如下图所示。



所以实际上假设我们要更新一行数据，此时数据库会找到这行数据所在的数据页，然后从磁盘文件里把这行数据所在的数据页直接给加载到Buffer Pool里去

也就是说，Buffer Pool中存放的是一个一个的数据页，如下图。



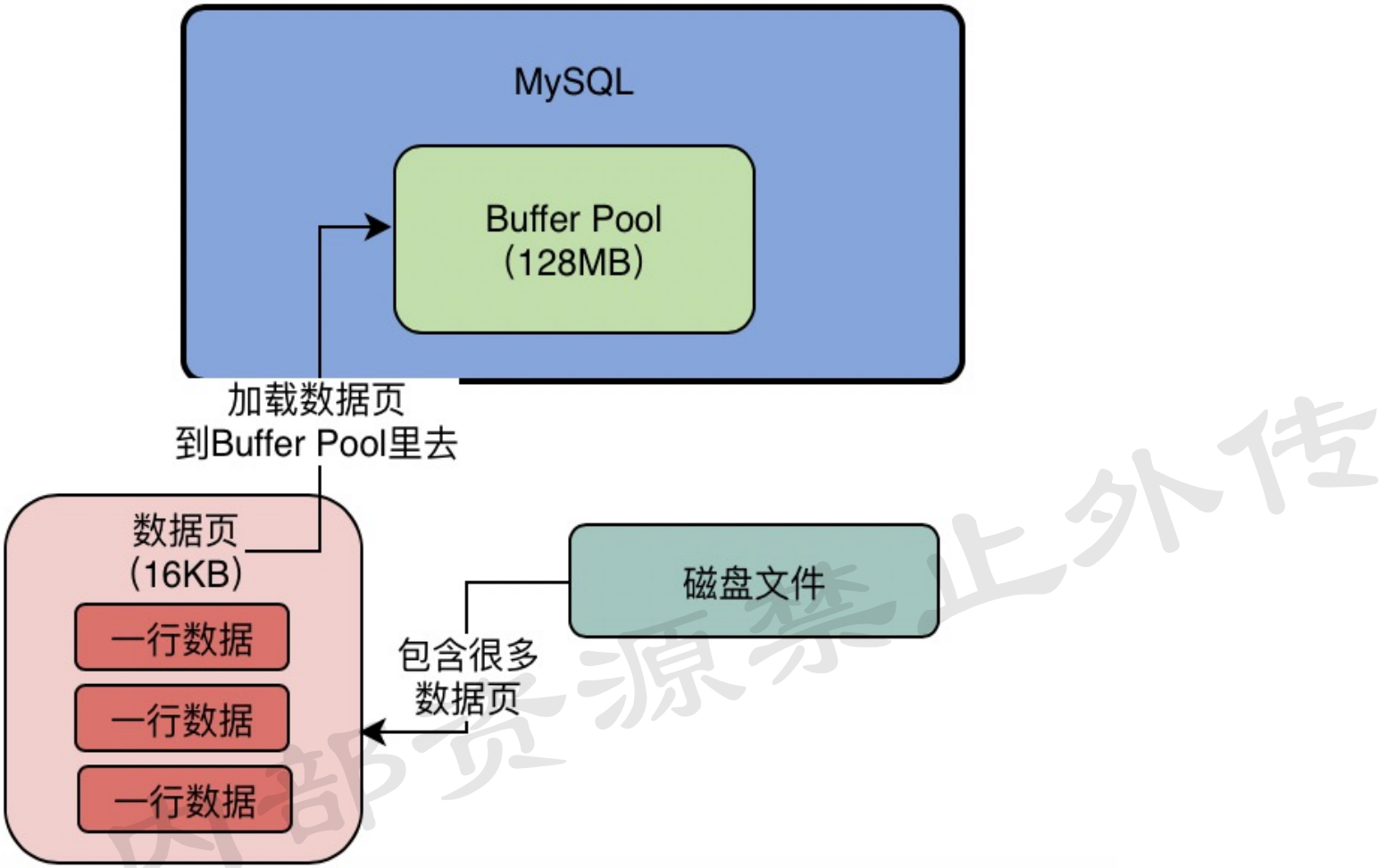
3、磁盘上的数据页和Buffer Pool中的缓存页是如何对应起来的?

实际上默认情况下，磁盘中存放的数据页的大小是16KB，也就是说，一页数据包含了16KB的内容。

而Buffer Pool中存放的一个一个的数据页，我们通常叫做缓存页，因为毕竟Buffer Pool是一个缓冲池，里面的数据都是从磁盘缓存到内存去的。

而Buffer Pool中默认情况下，一个缓存页的大小和磁盘上的一个数据页的大小是一一对应起来的，都是16KB。

所以我们看下图，我给图中的Buffer Pool标注出来了他的内存大小，假设他是128MB吧，然后数据页的大小是16KB。



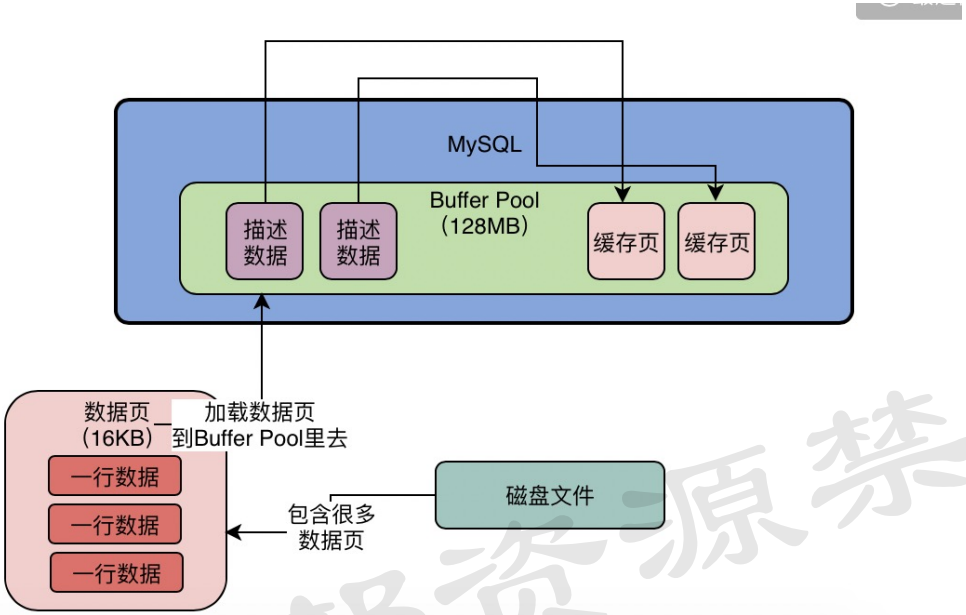
4、缓存页对应的描述信息是什么?

接着我们要了解下一个概念，对于每个缓存页，他实际上都会有一个描述信息，这个描述信息大体可以认为是用来描述这个缓存页的

比如包含如下的一些东西：这个数据页所属的表空间、数据页的编号、这个缓存页在Buffer Pool中的地址以及别的一些杂七杂八的东西。

每个缓存页都会对应一个描述信息，这个描述信息本身也是一块数据，在Buffer Pool中，每个缓存页的描述数据放在最前面，然后各个缓存页放在后面

所以此时我们看下面的图，Buffer Pool实际看起来大概长这个样子。



而且这里我们要注意一点，Buffer Pool中的描述数据大概相当于缓存页大小的5%左右，也就是每个描述数据大概是800个字节左右的大小，然后假设你设置的buffer pool大小是128MB，实际上Buffer Pool真正的最终大小会超出一些，可能有个130多MB的样子，因为他里面还要存放每个缓存页的描述数据。

5、今日思考题

今天想留给大家思考一个问题，就是内存碎片的问题

大家可以想象一下，对于Buffer Pool而言，他里面会存放很多的缓存页以及对应的描述数据，那么假设Buffer Pool里的内存都用尽了，已经没有足够的剩余内存来存放缓存页和描述数据了，此时Buffer Pool里就一点内存都没有了吗？还是说Buffer

Pool里会残留一些内存碎片呢?

如果你觉得Buffer Pool里会有内存碎片的话，那么你觉得应该怎么做才能尽可能减少Buffer Pool里的内存碎片呢?

请大家在评论区给出自己的思考，另外多跟其他人在评论区里交流。

End

狸猫技术窝精品专栏及课程推荐：

- 《从零开始带你成为消息中间件实战高手》
- 《21天互联网Java进阶面试训练营》（分布式篇）
- 《互联网Java工程师面试突击》（第1季）
- 《互联网Java工程师面试突击》（第3季）
- 《从零开始带你成为JVM实战高手》

图文 13 从磁盘读取数据页到Buffer Pool的时候，free链表有什么用？

手机观看

详情 评论

从磁盘读取数据页到Buffer Pool的时候，free链表有什么用？

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

1、数据库启动的时候，是如何初始化Buffer Pool的？

现在我们已经搞明白一件事儿了，那就是数据库的Buffer Pool到底长成个什么样，大家想必都是理解了

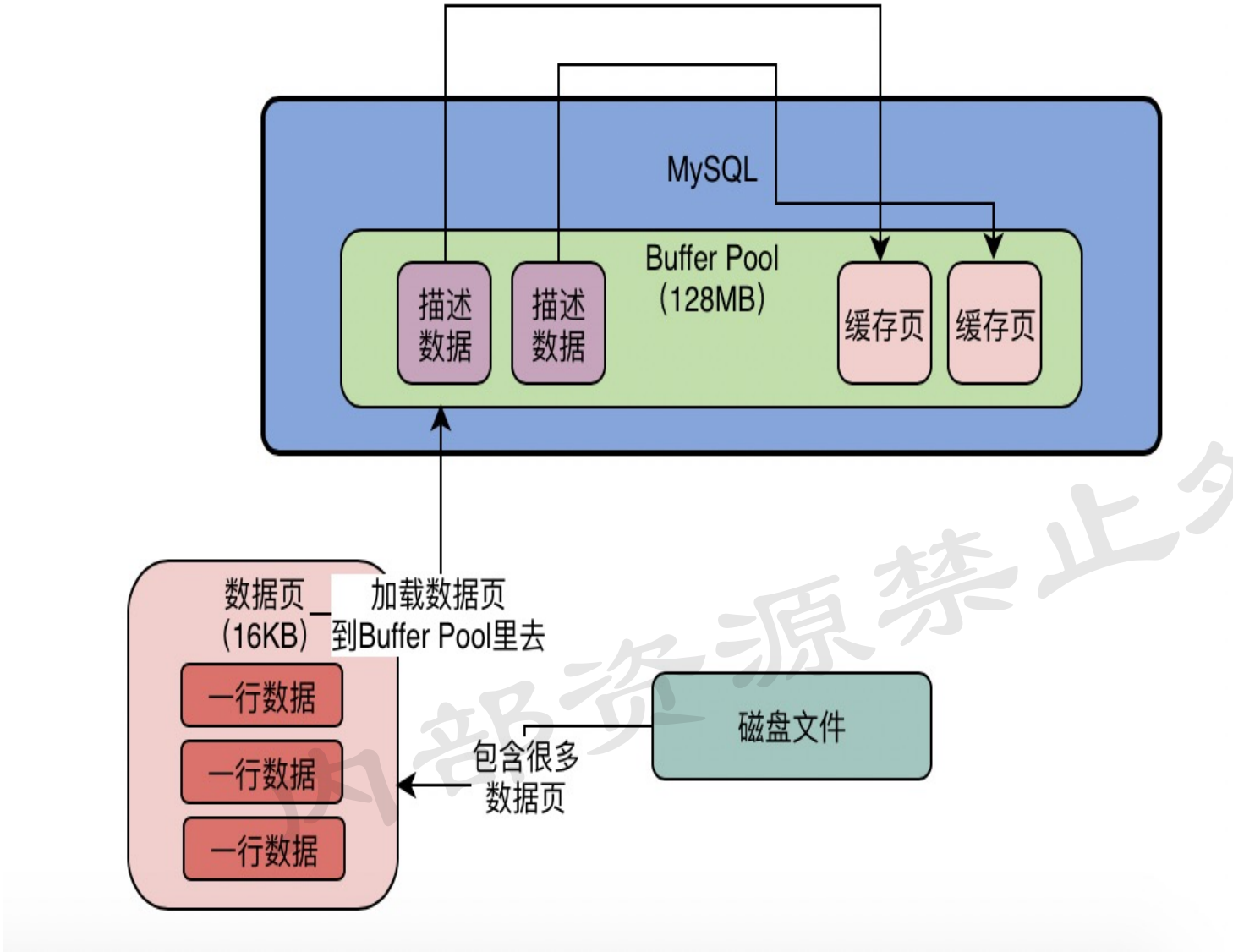
其实说白了，里面就是会包含很多个缓存页，同时每个缓存页还有一个描述数据，也可以叫做是控制数据，但是我个人是比较倾向于叫做描述数据，或者缓存页的元数据，都是可以的。

那么在数据库启动的时候，他是如何初始化Buffer Pool的呢？

其实这个也很简单，数据库只要一启动，就会按照你设置的Buffer Pool大小，稍微再加大一点，去找操作系统申请一块内存区域，作为Buffer Pool的内存区域。

然后当内存区域申请完毕之后，数据库就会按照默认的缓存页的16KB的大小以及对应的800个字节左右的描述数据的大小，在Buffer Pool中划分出来一个一个的缓存页和一个一个的他们对应的描述数据。

然后当数据库把Buffer Pool划分完毕之后，看起来就是之前我们看到的那张图了，如下图所示。



只不过这个时候，Buffer Pool中的一个一个的缓存页都是空的，里面什么都没有，要等数据库运行起来之后，当我们要对数据执行增删改查的操作的时候，才会把数据对应的页从磁盘文件里读取出来，放入Buffer Pool中的缓存页中。

2、我们怎么知道哪些缓存页是空闲的呢?

接着我们来看下一个问题，当你的数据库运行起来之后，你肯定会不停的执行增删改查的操作，此时就需要不停的从磁盘上读取一个一个的数据页放入Buffer Pool中的对应的缓存页里去，把数据缓存起来，那么以后就可以对这个数据在内存里执行增删改查了。

但是此时在从磁盘上读取数据页放入Buffer Pool中的缓存页的时候，必然涉及到一个问题，那就是哪些缓存页是空闲的?

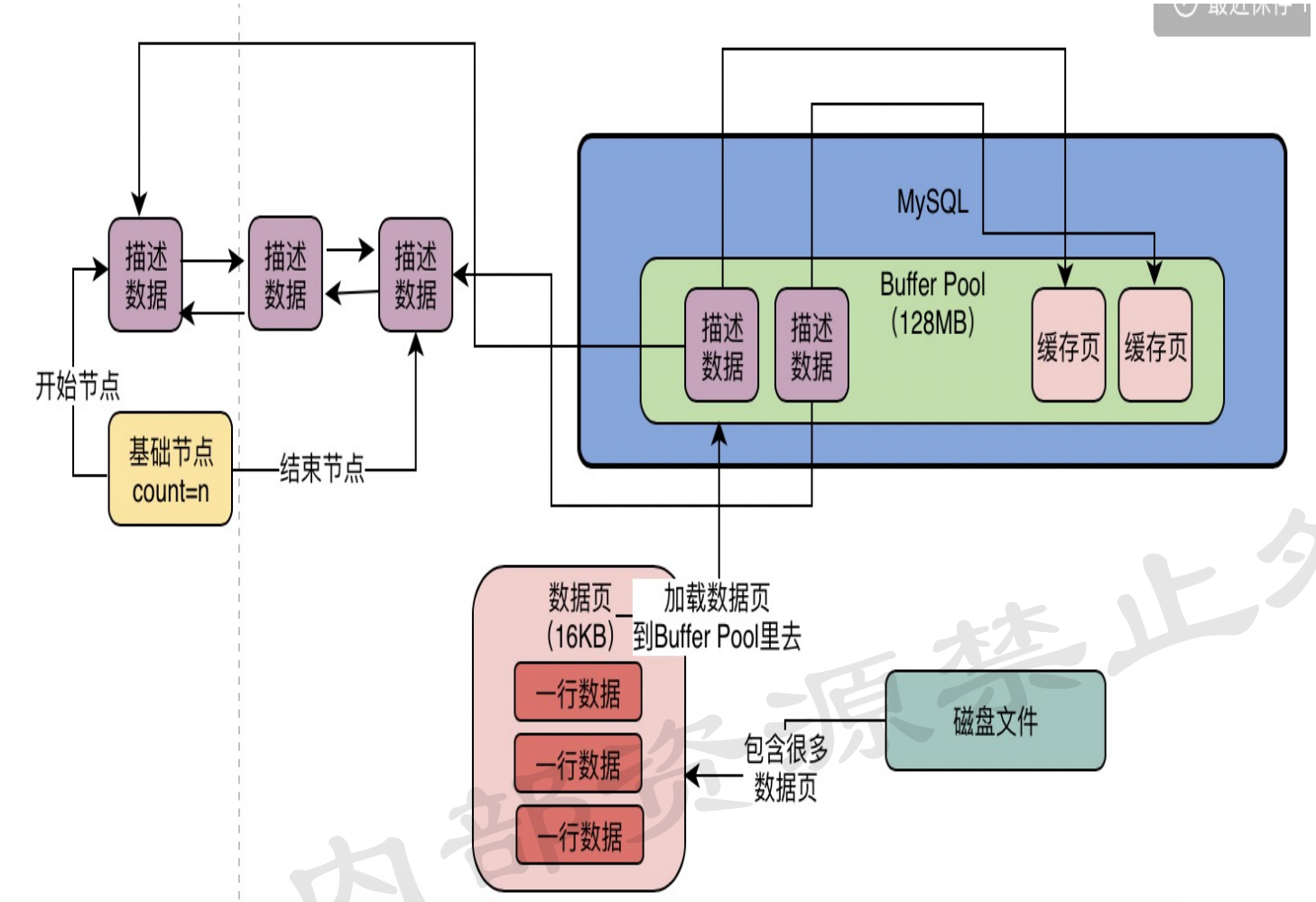
因为默认情况下磁盘上的数据页和缓存页是一 一对应起来的，都是16KB，一个数据页对应一个缓存页。

所以我们必须要知道Buffer Pool中哪些缓存页是空闲的状态。

所以数据库会为Buffer Pool设计一个**free链表**，他是一个双向链表数据结构，这个free链表里，每个节点就是一个空闲的缓存页的描述数据块的地址，也就是说，只要你一个缓存页是空闲的，那么他的描述数据块就会被放入这个free链表中。

刚开始数据库启动的时候，可能所有的缓存页都是空闲的，因为此时可能是一个空的数据库，一条数据都没有，所以此时所有缓存页的描述数据块，都会被放入这个free链表中

我们看下图所示



大家可以看到上面出现了一个free链表，这个free链表里面就是各个缓存页的描述数据块，只要缓存页是空闲的，那么他们对应的描述数据块就会加入到这个free链表中，每个节点都会双向链接自己的前后节点，组成一个双向链表。

除此之外，这个free链表有一个基础节点，他会引用链表的头节点和尾节点，里面还存储了链表中有多少个描述数据块的节点，也就是有多少个空闲的缓存页。

3、free链表占用多少内存空间?

可能有的人会以为这个描述数据块，在Buffer Pool里有一份，在free链表里也有一份，好像在内存里有两个一模一样的描述数据块，是么?

其实这么想就大错特错了。

这里要给大家讲明白一点，这个free链表，他本身其实就是由Buffer Pool里的描述数据块组成的，你可以认为是每个描述数据块里都有两个指针，一个是free_pre，一个是free_next，分别指向自己的上一个free链表的节点，以及下一个free链表的节点。

通过Buffer Pool中的描述数据块的free_pre和free_next两个指针，就可以把所有的描述数据块串成一个free链表，大家可以自己去思考一下这个问题。上面为了画图需要，所以把描述数据块单独画了一份出来，表示他们之间的指针引用关系。

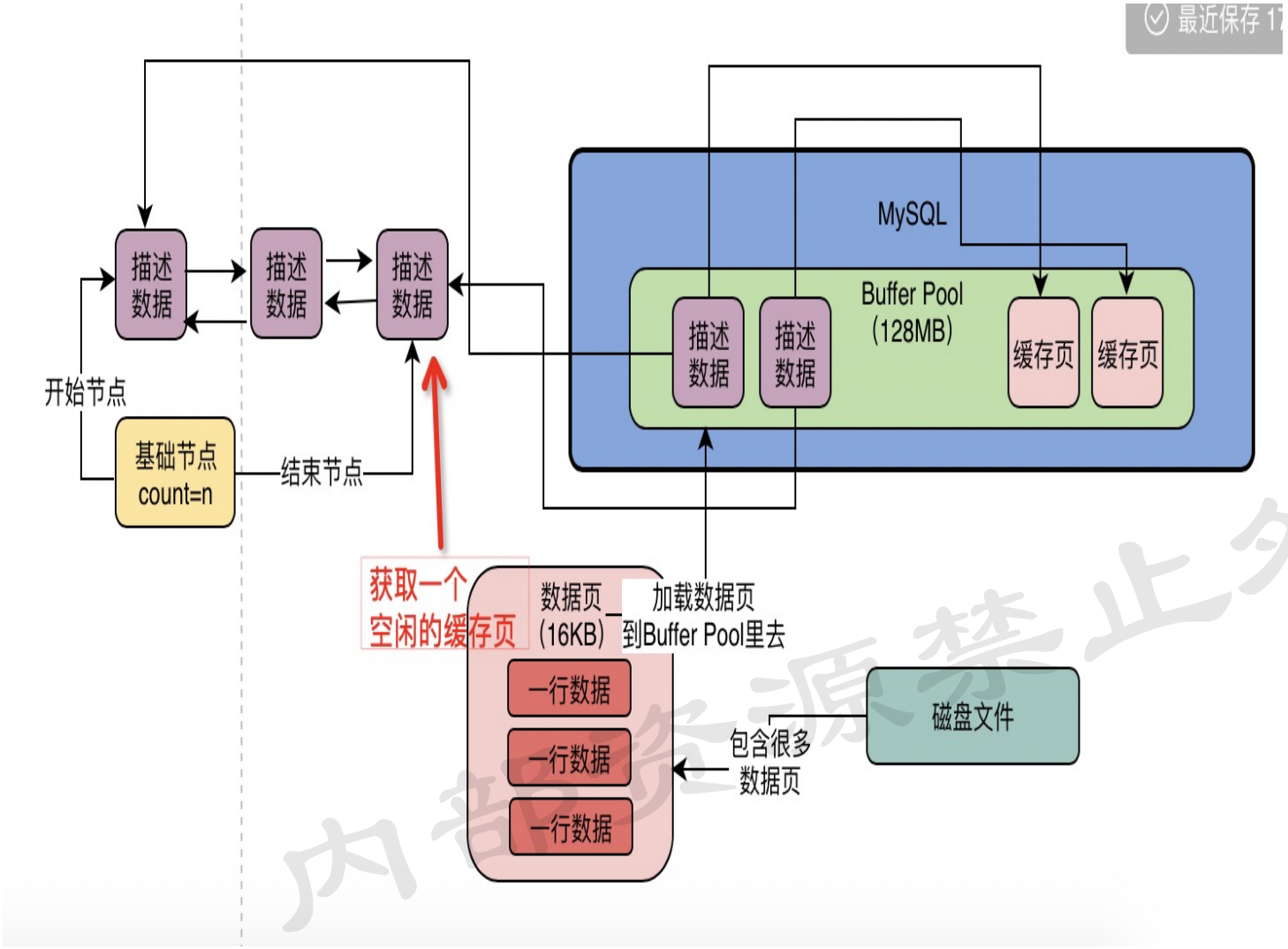
对于free链表而言，只有一个基础节点是不属于Buffer Pool的，他是40字节大小的一个节点，里面就存放了free链表的头节点的地址，尾节点的地址，还有free链表里当前有多少个节点。

4、如何将磁盘上的页读取到Buffer Pool的缓存页中去?

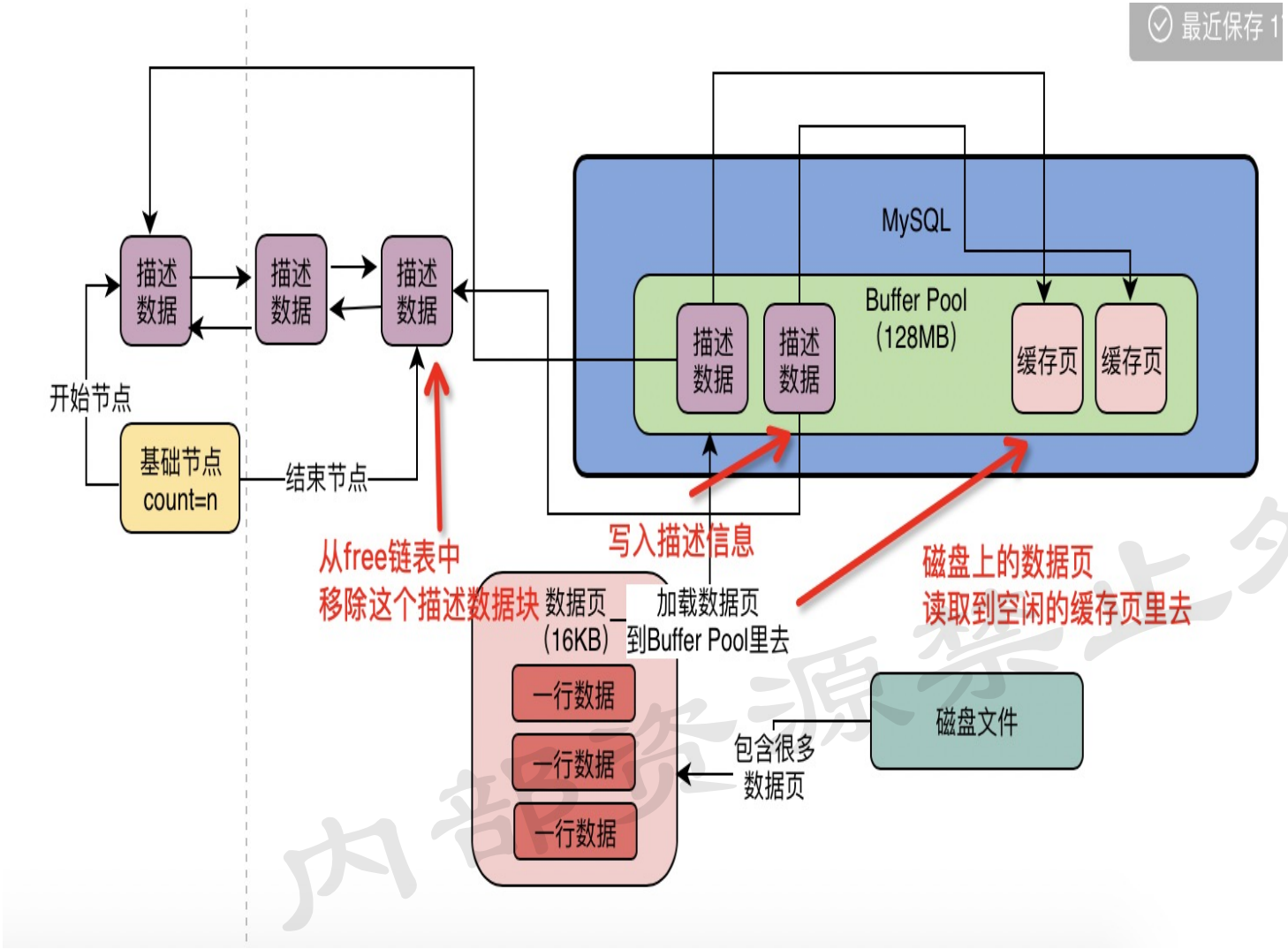
好了，现在我们可以来解答这一篇文章的最后一个问题了，当你需要把磁盘上的数据页读取到Buffer Pool中的缓存页里去的时候，是怎么做到的?

其实有了free链表之后，这个问题就很简单了。

首先，我们需要从free链表里获取一个描述数据块，然后就可以对应的获取到这个描述数据块对应的空闲缓存页，我们看下图所示。



接着我们就可以把磁盘上的数据页读取到对应的缓存页里去，同时把相关的一些描述数据写入缓存页的描述数据块里去，比如这个数据页所属的表空间之类的信息，最后把那个描述数据块从free链表里去除就可以了，如下图所示。



可能有朋友还是疑惑，这个描述数据块是怎么从free链表里移除的呢？

简单，我给你一段伪代码演示一下。

假设有一个描述数据块02，他的上一个节点是描述数据块01，下一个节点是描述数据块03，那么他在内存中的数据结构如下。

```
// 描述数据块
DescriptionDataBlock {
   // 这个块自己就是block02
   block_id = block02
   // 在free链表中的上一个节点是block01
   free_pre = block01;
   // 在free链表中的下一个节点是block03
   free_next = block03;
}
```

现在假设block03被使用了，要从free链表中移除，那么此时直接就可以把block02节点的free_next设置为null就可以了，block03就从free链表里失去引用关系了，如下所示。

```
// 描述数据块
DescriptionDataBlock {
   // 这个块自己就是block02
   block_id = block02
   // 在free链表中的上一个节点是block01
   free_pre = block01;
   // 在free链表中的下一个节点是空的
   free_next = null;
}
```

想必看到这里，大家就完全明白，磁盘中的数据页是如何读取到Buffer Pool中的缓存页里去的了，而且这个过程中free链表是用来干什么的。

5、你怎么知道数据页有没有被缓存?

接着我们来看下一个问题，那你怎么知道一个数据页有没有被缓存呢?

我们在执行增删改查的时候，肯定是先看看这个数据页有没有被缓存，如果没被缓存就走上面的逻辑，从free链表中找到一个空闲的缓存页，从磁盘上读取数据页写入缓存页，写入描述数据，从free链表中移除这个描述数据块。

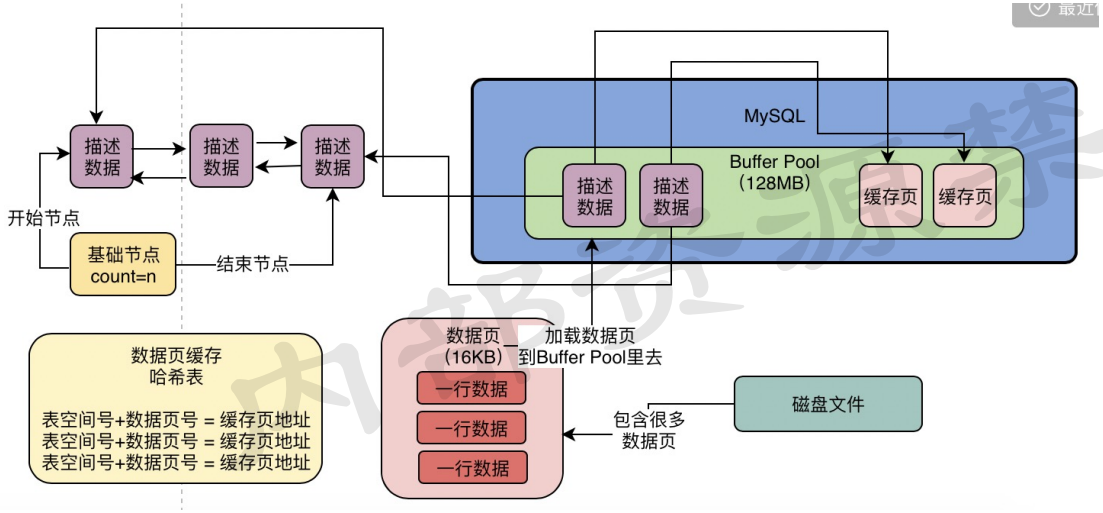
但是如果数据页已经被缓存了，那么就会直接使用了。

所以其实**数据库还会有一个哈希表数据结构，他会用表空间号+数据页号，作为一个key，然后缓存页的地址作为value。**

当你要使用一个数据页的时候，通过“表空间号+数据页号”作为key去这个哈希表里查一下，如果没有就读取数据页，如果已经有了，就说明数据页已经被缓存了。

我们看下图，又引入了一个数据页缓存哈希表的结构。

也就是说，每次你读取一个数据页到缓存之后，都会在这个哈希表中写入一个key-value对，key就是表空间号+数据页号，value就是缓存页的地址，那么下次如果你再使用这个数据页，就可以从哈希表里直接读取出来他已经被放入一个缓存页了。



6、今日思考题

今天我们给大家留一个思考题，大家去想一个问题，我们要取一个数据的时候，必然会取他所属的一个数据页，而且这个数据必然是属于一个表的，所以我们在上面初步引入了一个表空间的概念

也就是说我们写SQL的时候，只知道表+行的概念，但是在MySQL内部操作的时候，是表空间+数据页的概念。

那么大家觉得这两者之间的区别是什么？他们之间的联系是什么？

请大家积极在评论区写下你的思考，多跟其他人在评论区中交流。

End

狸猫技术窝精品专栏及课程推荐：

- 《从零开始带你成为消息中间件实战高手》
- 《21天互联网Java进阶面试训练营》（分布式篇）
- 《互联网Java工程师面试突击》（第1季）
- 《互联网Java工程师面试突击》（第3季）
- 《从零开始带你成为JVM实战高手》

图文 14 当我们更新Buffer Pool中的数据时，flush链表有什么用？

手机观看

详情 评论

当我们更新Buffer Pool中的数据时，flush链表有什么用？

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

1、昨日思考题解答

我们先解答一下昨日的思考题，昨天是问了大家一个问题，Buffer Pool中会不会有内存碎片？

答案是：当然有

因为Buffer Pool大小是你自己定的，很可能Buffer Pool划分完全部的缓存页和描述数据块之后，还剩一点点的内存，这一点点的内存放不下任何一个缓存页了，所以这点内存就只能放着不能用，这就是内存碎片。

那怎么减少内存碎片呢？

其实也很简单，数据库在Buffer Pool中划分缓存页的时候，会让所有的缓存页和描述数据块都紧密的挨在一起，这样尽可能减少内存浪费，就可以尽可能的减少内存碎片的产生了。

如果你的Buffer Pool里的缓存页是东一块西一块，那么必然导致缓存页的内存之间有很多内存空隙，这就会有大量的内存碎片了。

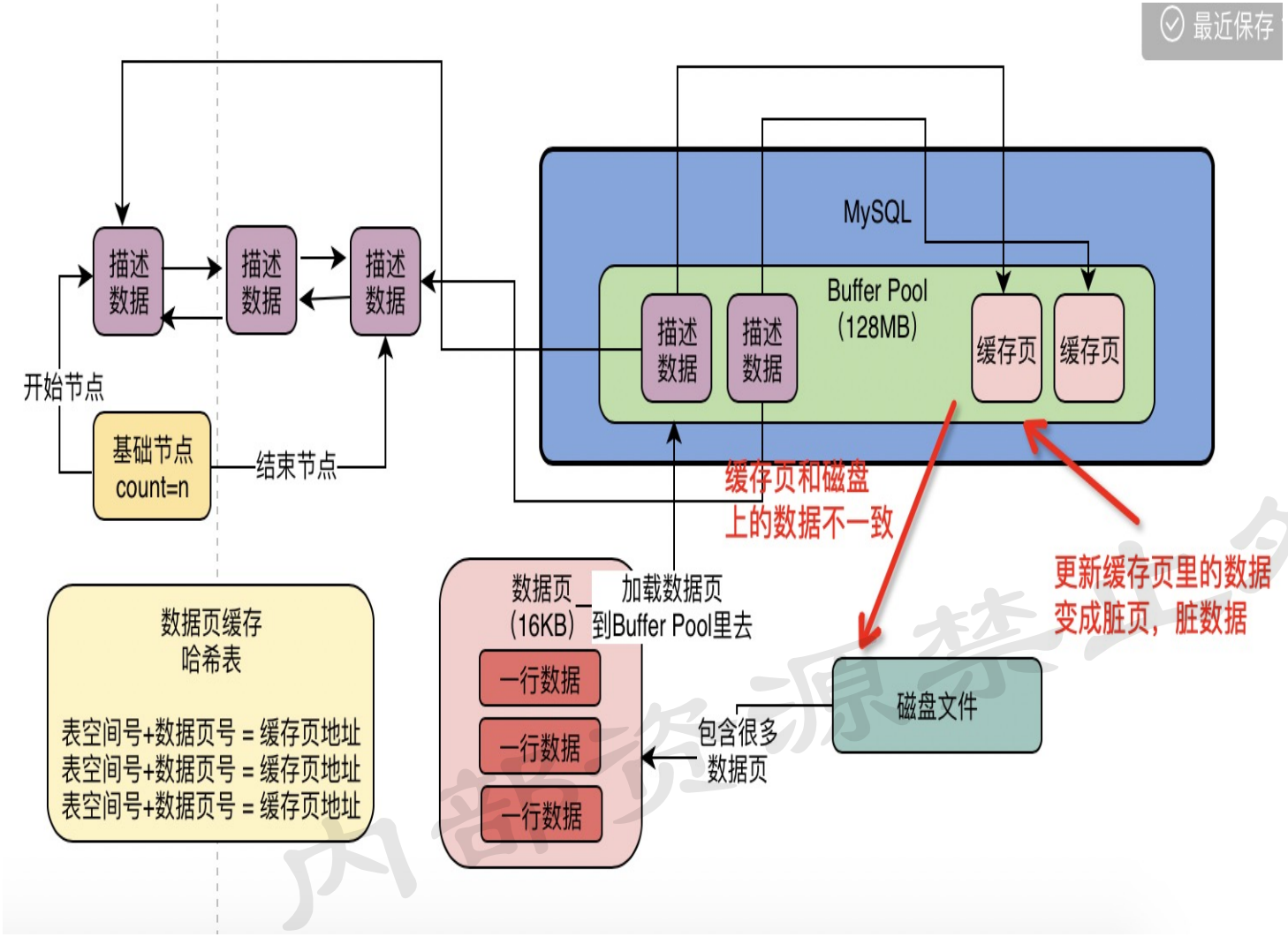
2、脏数据页到底为什么会脏?

接着我们看一个很关键的问题，你在执行增删改的时候，如果发现数据页没缓存，那么必然会基于free链表找到一个空闲的缓存页，然后读取到缓存页里去，但是如果已经缓存了，那么下一次就必然会直接使用缓存页。

反正不管怎么样，你要更新的数据页都会在Buffer Pool的缓存页里，供你在内存中直接执行增删改的操作。

接着你肯定会去更新Buffer Pool的缓存页中的数据，此时一旦你更新了缓存页中的数据，那么缓存页里的数据和磁盘上的数据页里的数据，是不是就不一致了?

这个时候，我们就说缓存页是脏数据，脏页



3、哪些缓存页是脏页呢?

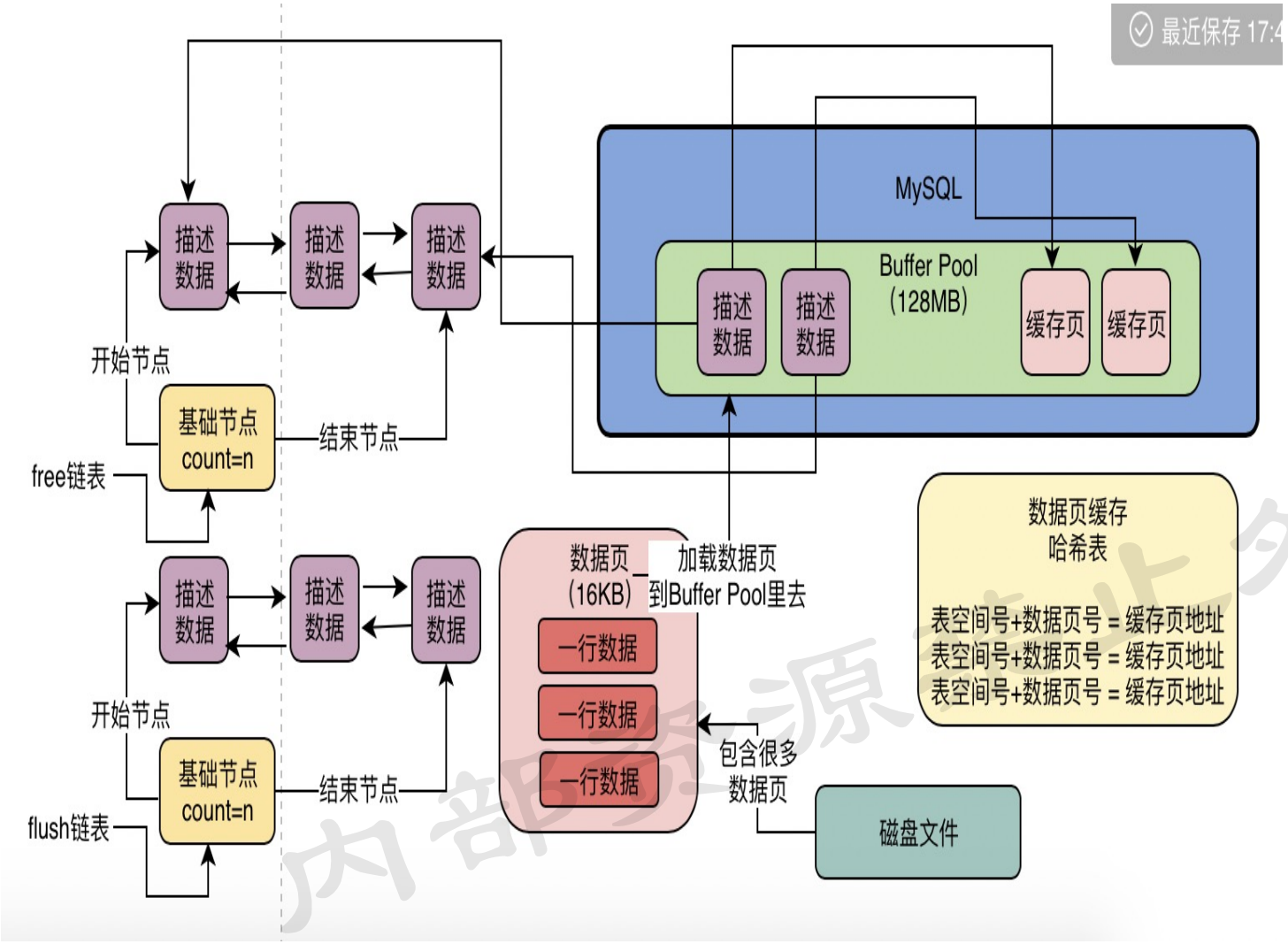
其实通过之前的学习，我们都是知道一点的，最终这些在内存里更新的脏页的数据，都是要被刷新回磁盘文件的。

但是这里就有一个问题了，不可能所有的缓存页都刷回磁盘的，因为有的缓存页可能是因为查询的时候被读取到Buffer Pool里去的，可能根本没修改过！

所以数据库在这里引入了另外一个跟free链表类似的**flush链表**，这个flush链表本质也是通过缓存页的描述数据块中的两个指针，让被修改过的缓存页的描述数据块，组成一个双向链表。

凡是被修改过的缓存页，都会把他的描述数据块加入到flush链表中去，flush的意思就是这些都是脏页，后续都是要flush刷新到磁盘上去的

所以flush链表的结构如下图所示，跟free链表几乎是一样的。



4、flush链表构造的伪代码演示

我们用一些伪代码来给大家展示一下这个flush链表的构造过程，比如现在缓存页01被修改了数据，那么他就是脏页了，此时就必须把他加入到flush链表中去

此时缓存页01的描述数据块假设如下所示

```
// 描述数据块
DescriptionDataBlock {
   // 这是缓存页01的数据块
   block_id = block01
   // 在free链表中的上一个节点和下一个节点
   // 因为这个缓存页已经被更新过了，肯定不在free链表里了
   // 所以他在free链表中的两个指针都是null
   free_pre = null
   free_next = null
   // 在flush链表中的上一个节点和下一个节点
   // 现在因为flush链表中就他一个节点，所以也都是null
   flush_pre = null
   flush_next = null
}
// flush链表的基础节点
FlushLinkListBaseNode {
    // 基础节点指向链表起始节点和结束节点的指针
    // flush链表中目前就一个缓存页01，所以指向他的描述数据块
    start = block01
    end = block01
    // flush链表中有几个节点
    count = 1
}
```

好了，我们可以看到，现在flush链表的基础节点就指向了一个block01的节点，接着比如缓存页02被更新了，他也是脏页了，
此时他的描述数据块也要被加入到flush链表中去

此时伪代码如下：

```
// 描述数据块
DescriptionDataBlock {
   // 这是缓存页01的数据块
   block_id = block01
   // 在free链表中的上一个节点和下一个节点
   // 因为这个缓存页已经被更新过了，肯定不在free链表里了
   // 所以他在free链表中的两个指针都是null
   free_pre = null
   free_next = null
   // 在flush链表中的上一个节点和下一个节点
   // 现在因为flush链表中他是起始节点，所以他的flush_pre指针是null
   flush_pre = null
   // 然后flush链表中他的下一个节点是block02，所以flush_next指向block02
   flush_next = block02
}
// 描述数据块
DescriptionDataBlock {
   // 这是缓存页02的数据块
   block_id = block02
   // 在free链表中的上一个节点和下一个节点
   // 因为这个缓存页已经被更新过了，肯定不在free链表里了
   // 所以他在free链表中的两个指针都是null
   free_pre = null
   free_next = null
   // 在flush链表中的上一个节点和下一个节点
   // 现在因为flush链表中，他是尾节点，他的上一个节点是block01
   // 他的下一个节点就是null
   flush_pre = block01
   flush_next = null
}
// flush链表的基础节点
FlushLinkListBaseNode {
    // 基础节点指向链表起始节点和结束节点的指针
    // flush链表中目前有缓存页01和缓存页02，所以指向他们的描述数据块
    start = block01 // 起始节点是block01
    end = block02 // 尾巴节点是block02
    // flush链表中有几个节点
    count = 2
}
```

大家可以看到，当你更新缓存页的时候，通过变换缓存页中的描述数据块的flush链表的指针，就可以把脏页的描述数据块组成一个双向链表，也就是flush链表，而且flush链表的基础节点会指向起始节点和尾巴节点。

通过这个flush链表，就可以记录下来哪些缓存页是脏页了！

End

狸猫技术窝精品专栏及课程推荐：

- 《从零开始带你成为消息中间件实战高手》
- 《21天互联网Java进阶面试训练营》（分布式篇）
- 《互联网Java工程师面试突击》（第1季）
- 《互联网Java工程师面试突击》（第3季）
- 《从零开始带你成为JVM实战高手》

图文 15 当Buffer Pool中的缓存页不够的时候，如何基于LRU算法淘汰部分缓存?

手机观看

607 人次阅读 2020-02-11 08:57:22

详情 评论

当Buffer Pool中的缓存页不够的时候，如何基于LRU算法淘汰部分缓存?

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

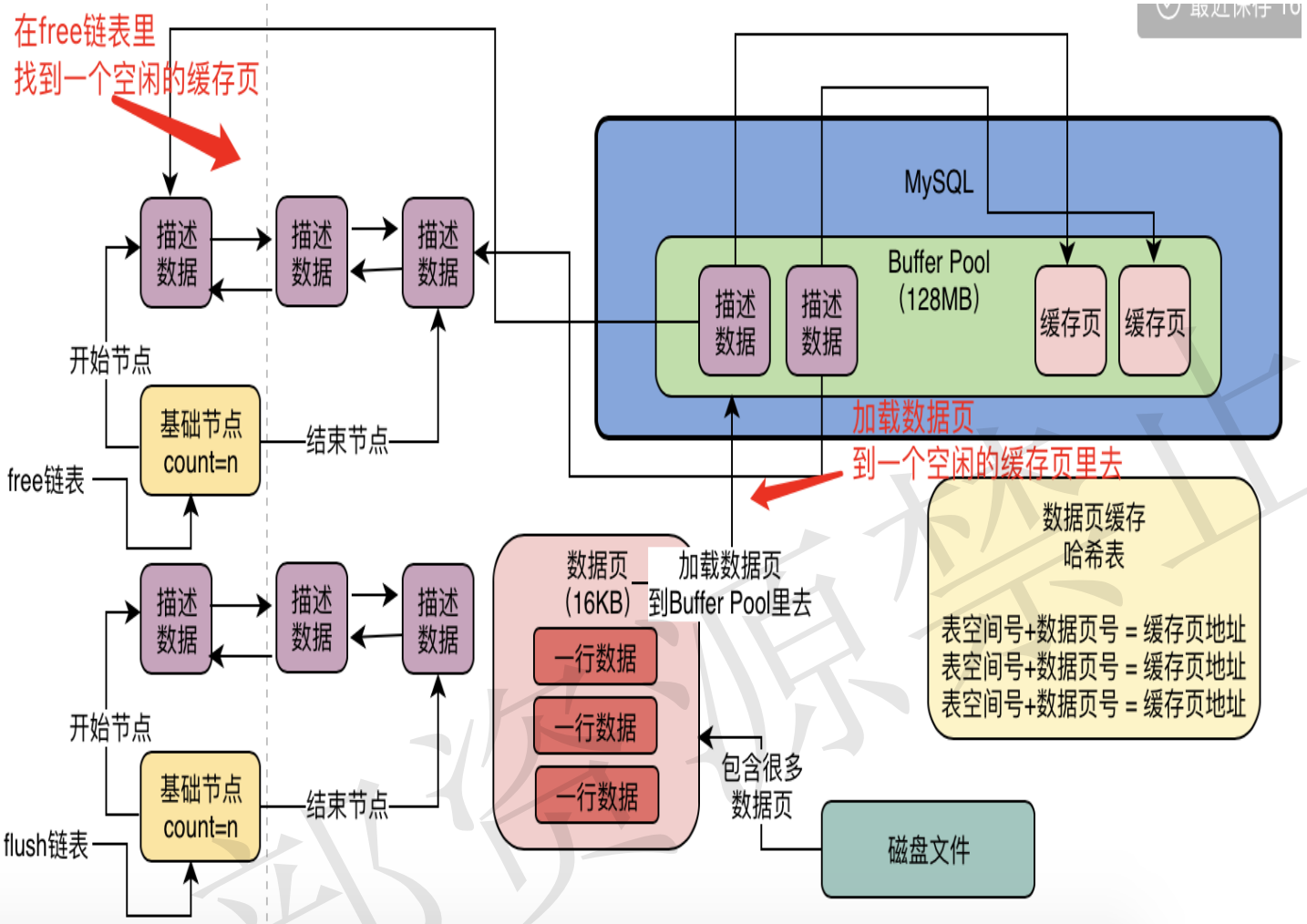
具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

1、如果Buffer Pool中的缓存页不够了怎么办?

之前我们已经给大家讲解了Buffer Pool中的缓存页的划分，包括free链表的使用，然后磁盘上的数据页是如何加载到缓存页里去的，包括对缓存页修改之后，flush链表是如何用来记载脏数据页的。

今天我们接着来分析Buffer Pool的工作原理，我们来思考一个问题，当你要执行CRUD操作的时候，无论是查询数据，还是修改数据，实际上都会把磁盘上的数据页加载到缓存页里来，这个大家都是没有问题的吧?

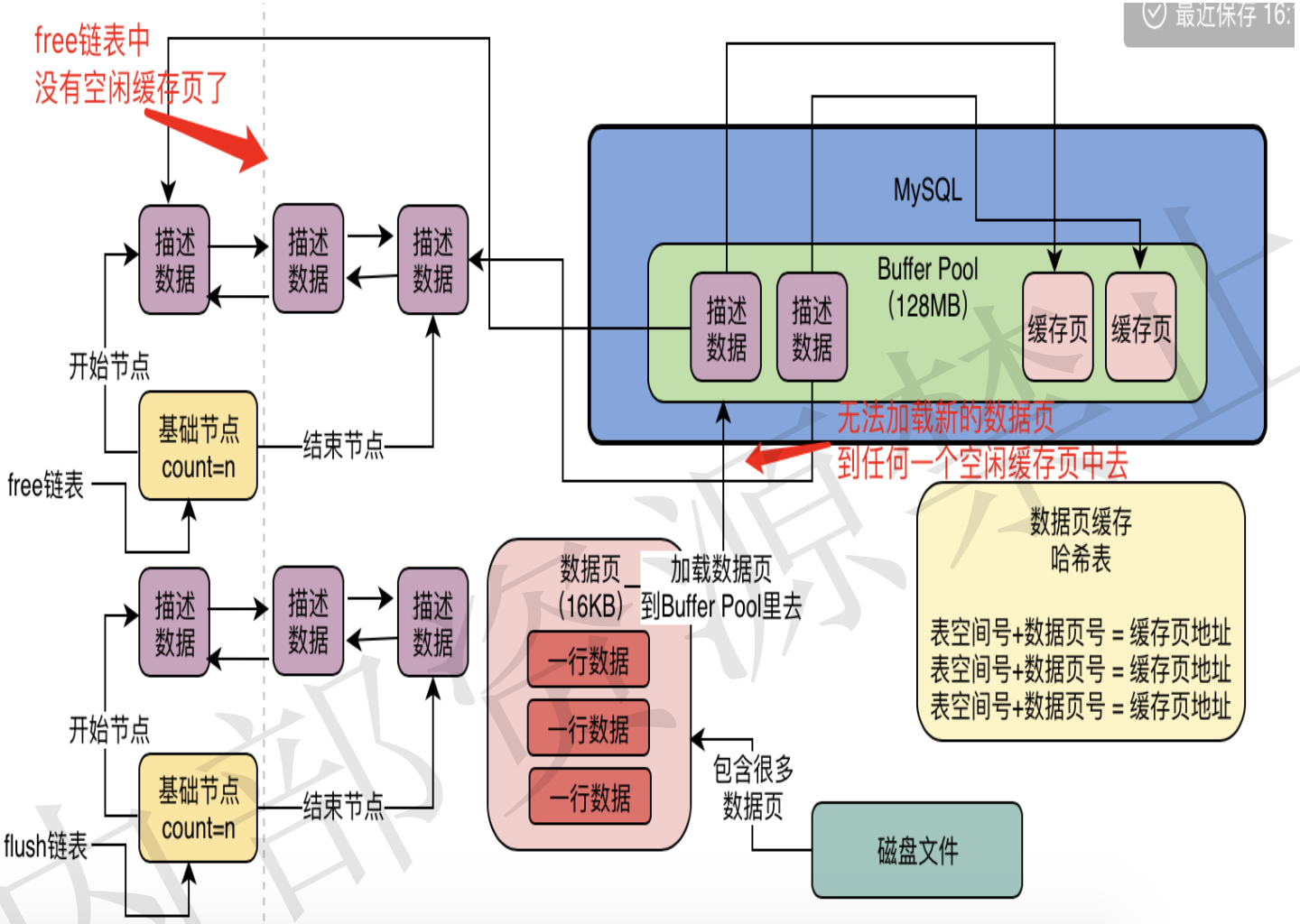
那么在加载数据到缓存页的时候，必然是要加载到空闲的缓存页里去的，所以必须要从free链表中找一个空闲的缓存页，然后把磁盘上的数据页加载到那个空闲的缓存页里去，我们看下图的红色箭头的示意。



那么大家通过之前的学习肯定都知道了，随着你不停的把磁盘上的数据页加载到空闲的缓存页里去，free链表中的空闲缓存页是不是会越来越少？因为只要你把一个数据页加载到一个空闲缓存页里去，free链表中就会减少一个空闲缓存页。

所以，当你不停的把磁盘上的数据页加载到空闲缓存页里去，free链表中不停的移除空闲缓存页，迟早有那么一瞬间，你会发现free链表中已经没有空闲缓存页了

这个时候，当你还要加载数据页到一个空闲缓存页的时候，怎么办呢？如下图。



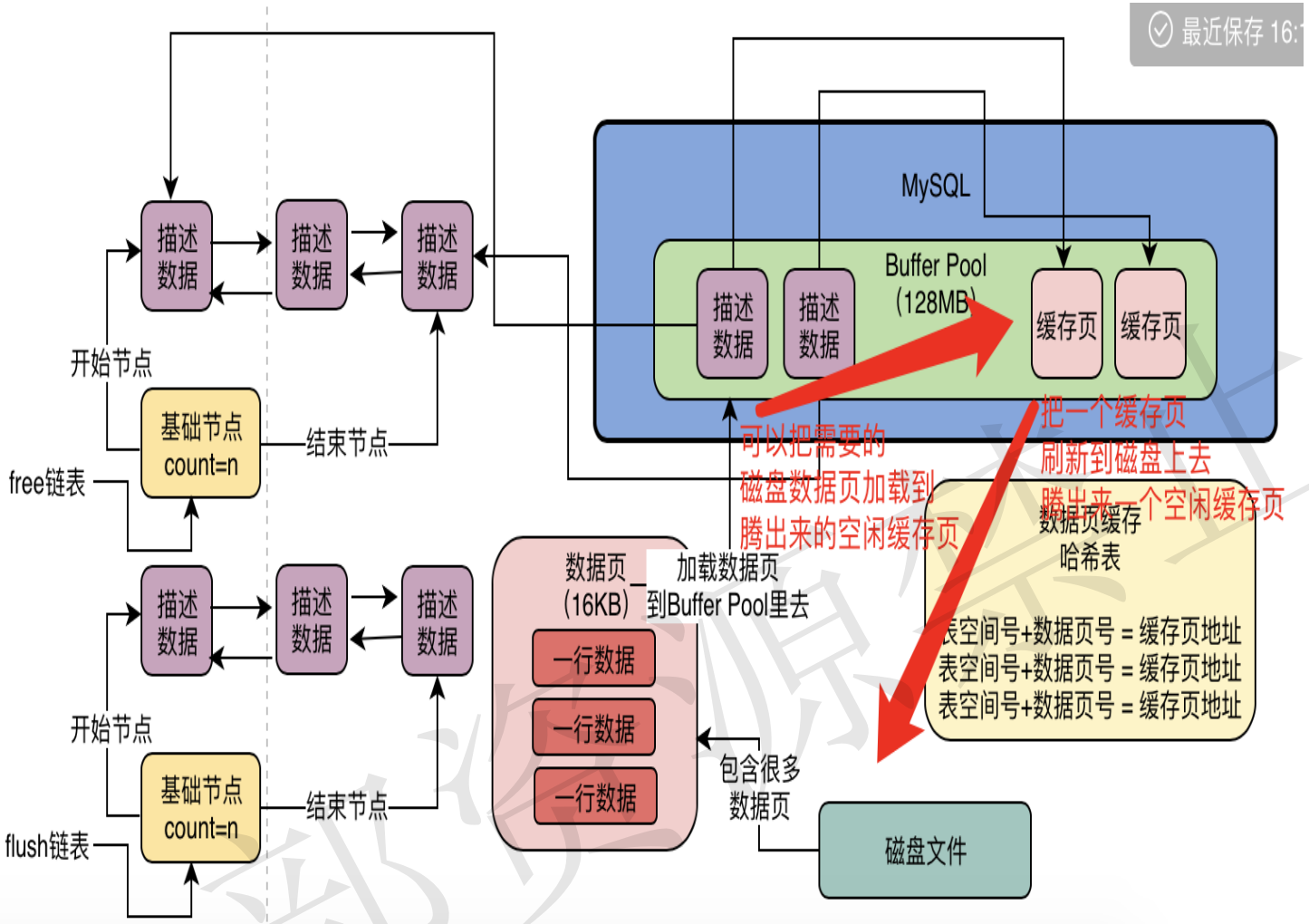
2、如果要淘汰掉一些缓存数据，淘汰谁？

针对上述的问题，大家来思考下一个问题，如果所有的缓存页都被塞了数据了，此时无法从磁盘上加载新的数据页到缓存页里去了，那么此时你只有一个办法，就是淘汰掉一些缓存页。

那什么叫淘汰缓存页呢？

顾名思义，你必须把一个缓存页里被修改过的数据，给他刷到磁盘上的数据页里去，然后这个缓存页就可以清空了，让他重新变成一个空闲的缓存页。

接着你再把磁盘上你需要的新的数据页加载到这个腾出来的空闲缓存页中去，如下图。



那么下一个问题来了，如果要把一个缓存页里的数据刷入磁盘，腾出来一个空闲缓存页，那么应该把哪个缓存页的数据给刷入磁盘呢?

3、缓存命中率概念的引入

要解答这个问题，我们就得引入一个缓存命中率的概念。

假设现在有两个缓存页，一个缓存页的数据，经常会被修改和查询，比如在100次请求中，有30次都是在查询和修改这个缓存页里的数据。那么此时我们可以说这种情况下，缓存命中率很高

为什么呢？因为100次请求中，30次都可以操作缓存，不需要从磁盘加载数据，这个缓存命中率就比较高了。

另外一个缓存页里的数据，就是刚从磁盘加载到缓存页之后，被修改和查询过1次，之后100次请求中没有一次是修改和查询这个缓存页的数据的，那么此时我们就说缓存命中率有点低，因为大部分请求可能还需要走磁盘查询数据，他们要操作的数据不在缓存中。

所以针对上述两个缓存页，假设此时让你做一个抉择，要把其中缓存页的数据刷入到磁盘去，腾出来一个空闲的缓存页，此时你会选择谁？

那还用想么，当然是选择第二个缓存页刷入磁盘中了！

因为第二个缓存页，压根儿就没什么人来使用他里面的数据，结果这些数据还空占据了一个缓存页，这不是占着茅坑不拉屎么？

4、引入LRU链表来判断哪些缓存页是不常用的

接着我们就要解决下一个问题了，就是你怎么知道哪些缓存页经常被访问，哪些缓存页很少被访问？

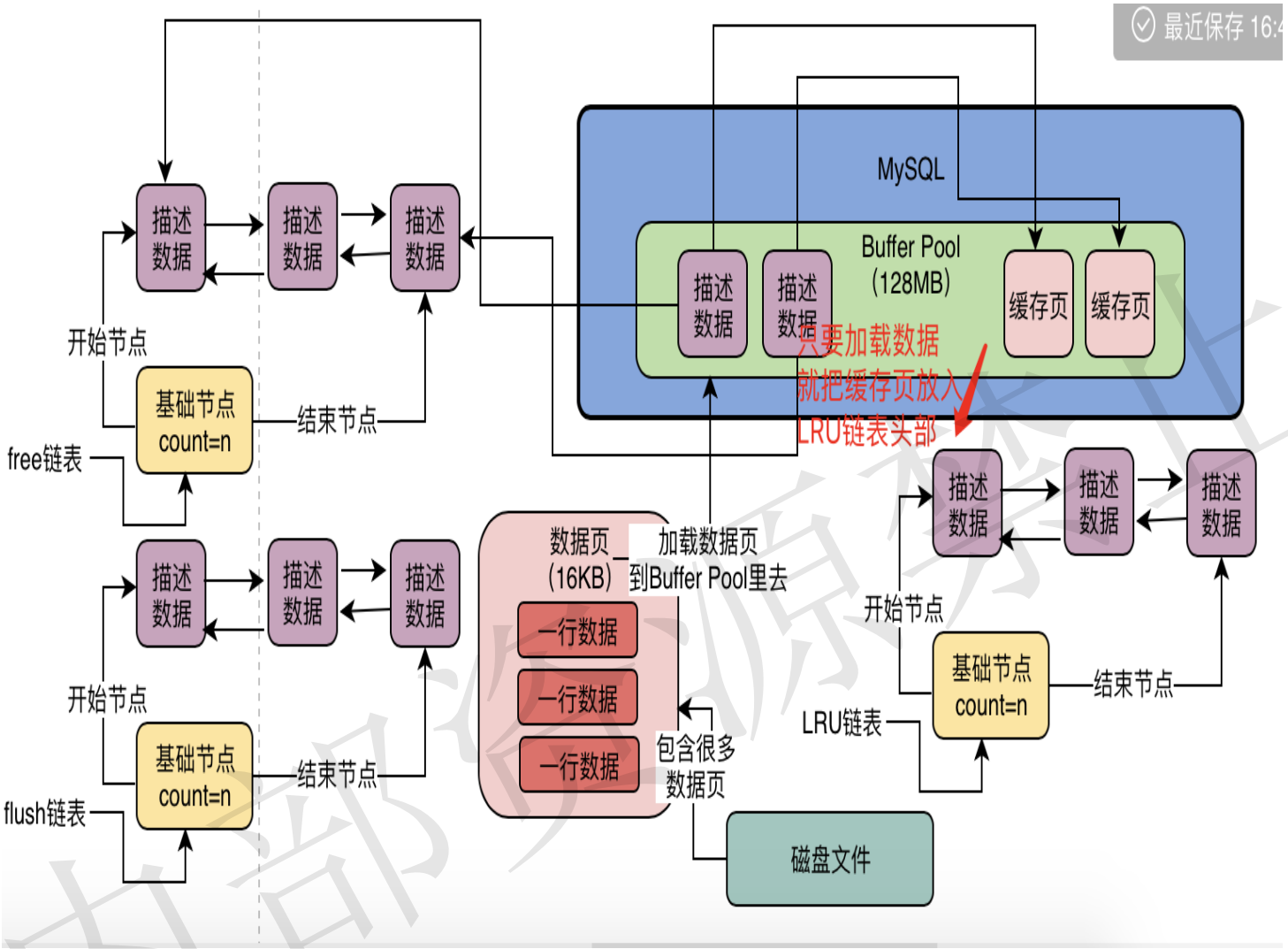
此时就要引入一个新的LRU链表了，这个所谓的LRU就是Least Recently Used，最近最少使用的意思。

通过这个LRU链表，我们可以知道哪些缓存页是最近最少被使用的，那么当你缓存页需要腾出来一个刷入磁盘的时候，不就可以选择那个LRU链表中最近最少被使用的缓存页了么？

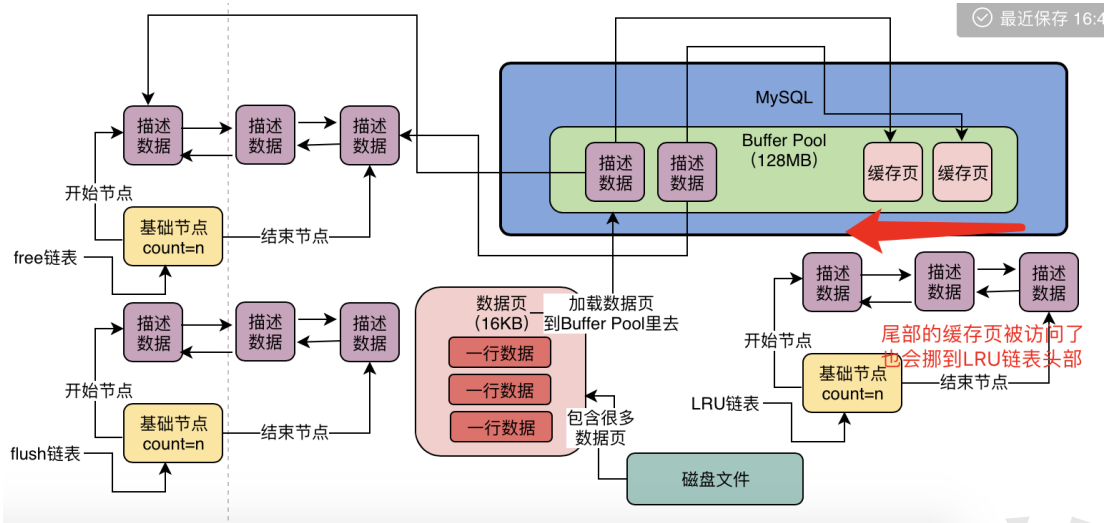
这个LRU链表大致是怎么个工作原理呢？

简单来说，我们看下图，假设我们从磁盘加载一个数据页到缓存页的时候，就把这个缓存页的描述数据块放到LRU链表头部去，那么只要有数据的缓存页，他都会在LRU里了，而且最近被加载数据的缓存页，都会放到LRU链表的头部

去。



然后假设某个缓存页的描述数据块本来在LRU链表的尾部，后续你只要查询或者修改了这个缓存页的数据，也要把这个缓存页挪动到LRU链表的头部去，也就是说最近被访问过的缓存页，一定在LRU链表的头部，如下图。



那么这样的话，当你的缓存页没有一个空闲的时候，你是不是要找出来那个最近最少被访问的缓存页去刷入磁盘？此时你就直接在LRU链表的尾部找到一个缓存页，他一定是最近最少被访问的那个缓存页！

然后你就把LRU链表尾部的那个缓存页刷入磁盘中，然后把你需要的磁盘数据页加载到腾出来的空闲缓存页中就可以了！

5、上次思考题解答

今天我们在末尾解答一下上次留的那个思考题，就是说，我们在SQL语句里都是用到的是表和行的概念，但是之前我们提到的表空间、数据页，他们之间的关系是什么呢？

其实简单来讲，**一个是逻辑概念，一个是物理概念。**

表、列和行，都是逻辑概念，我们只知道数据库里有一个表，表里有几个字段，有多少行，但是这些表里的数据，在数据库的磁盘上如何存储的，你知道吗？我们是不关注的，所以他们都是逻辑上的概念。

表空间、数据页，这些东西，都是物理上的概念，实际上在物理层面，你的表里的数据都放在一个表空间中，表空间是由一堆磁盘上的数据文件组成的，这些数据文件里都存放了你表里的数据，这些数据是由一个一个的数据页组织起来的，这些都是物理层面的概念，这就是他们之间的区别。

End

狸猫技术窝精品专栏及课程推荐：

- 《从零开始带你成为消息中间件实战高手》
- 《21天互联网Java进阶面试训练营》（分布式篇）
- 《互联网Java工程师面试突击》（第1季）
- 《互联网Java工程师面试突击》（第3季）
- 《从零开始带你成为JVM实战高手》

图文 16 简单的LRU链表在Buffer Pool实际运行中，可能导致哪些问题？ 手机观看

578 人次阅读 2020-02-11 09:03:06

详情 评论

简单的LRU链表在Buffer Pool实际运行中，可能导致哪些问题？

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

1、简单回顾一下

之前我们讲解了Buffer Pool在使用过程中如果缓存页都使用了，没有空闲的缓存页时，可以去LRU链表中的尾部找一个最近最少使用的缓存页，把他的数据刷入磁盘，腾出来一个空闲缓存页，然后加载需要的新的磁盘数据页到空闲缓存页里去。

而LRU链表的机制也很简单，只要是刚从磁盘上加载数据到缓存页里去，这个缓存页就放入LRU链表的头部，后续如果对任何一个缓存页访问了，也把缓存页从LRU链表中移动到头部去。

这样在LRU链表的尾部，一定是最近最少被访问的那个缓存页。

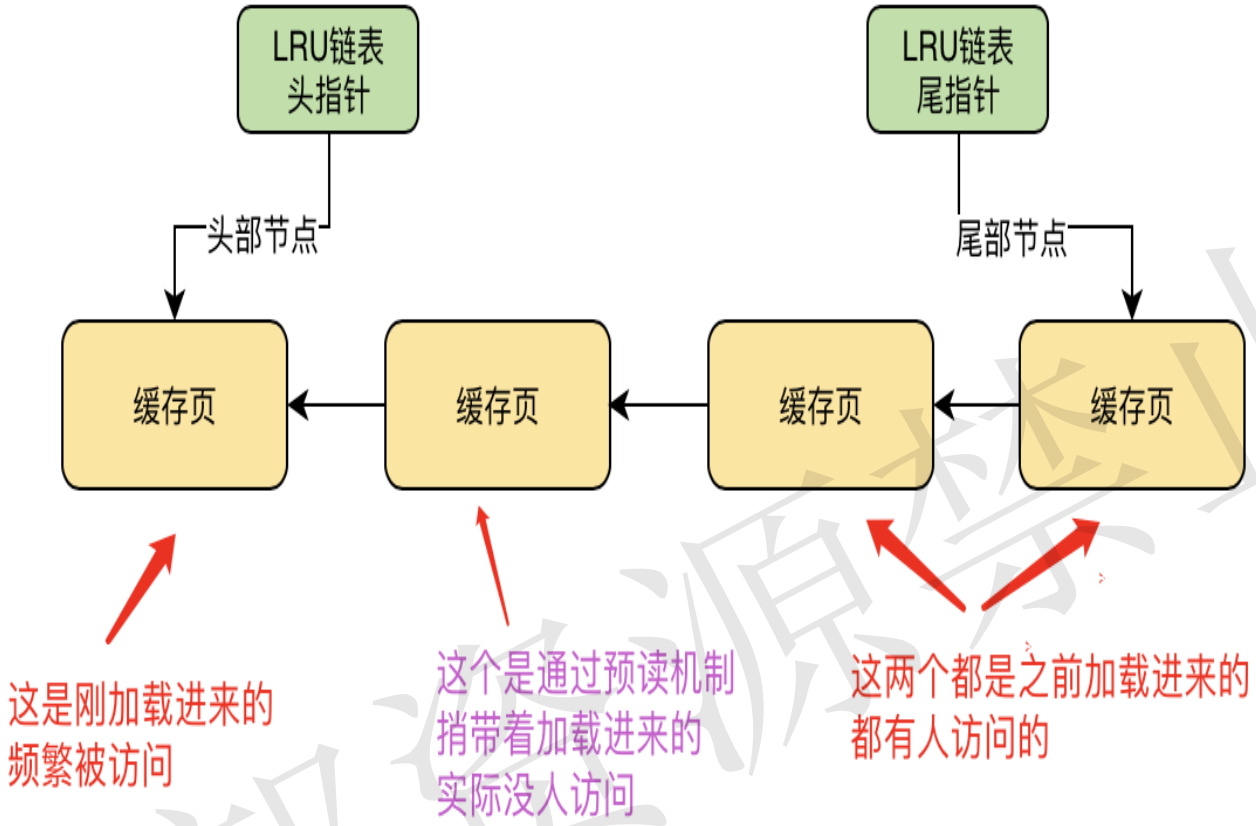
2、预读带来的一个巨大问题

但是这样的一个LRU机制在实际运行过程中，是会存在巨大的隐患的。

首先会带来隐患的就是MySQL的预读机制，这个所谓预读机制，说的就是当你从磁盘上加载一个数据页的时候，他可能会连带着把这个数据页相邻的其他数据页，也加载到缓存里去！

举个例子，假设现在有两个空闲缓存页，然后在加载一个数据页的时候，连带着把他的一个相邻的数据页也加载到缓存里去了，正好每个数据页放入一个空闲缓存页！

但是接下来呢，实际上只有一个缓存页是被访问了，另外一个通过预读机制加载的缓存页，其实并没有人访问，此时这两个缓存页可都在LRU链表的前面，如下图。



我们可以看到，这个图里很清晰的表明了，前两个缓存页都是刚加载进来的，但是此时第二个缓存页是通过预读机制捎带着加载进来的，他也放到了链表的前面，但是他实际没人访问他。

除了第二个缓存页之外，第一个缓存页，以及尾巴上两个缓存页，都是一直有人访问的那种缓存页，只不过上图代表的是刚刚把头部两个缓存页加载进来的时候的一个LRU链表当时的情况。

这个时候，假如没有空闲缓存页了，那么此时要加载新的数据页了，是不是就要从LRU链表的尾部把所谓的“最近最少使用的一个缓存页”给拿出来，刷入磁盘，然后腾出来一个空闲缓存页了？

这个时候，如果你把上图中LRU尾部的那个缓存页刷入磁盘然后清空，你觉得合理吗？他可是之前一直频繁被人访问的啊！只不过在这一个瞬间，被新加载进来的两个缓存页给占据了LRU链表前面的位置，尤其是第二个缓存页，居然还是通过预读机制加载进来的，根本就不会有人访问！

那么这个时候，你要是把LRU链表尾部的缓存页给刷入磁盘，这是绝对不合理的，最合理的反而是把上图中LRU链表的第二个通过预读机制加载进来的缓存页给刷入磁盘和清空，毕竟他几乎是没什么人会访问的！

3、哪些情况下会触发MySQL的预读机制?

现在我们已经理解了预读机制一下子把相邻的数据页加载进缓存，放入LRU链表前面的隐患了，预读机制加载进来的缓存页可能根本不会有人访问，结果他却放在了LRU链表的前面，此时可能会把LRU尾部的那些被频繁访问的缓存页刷入磁盘中！

所以我们来看看，到底哪些情况下会触发MySQL的预读机制呢?

（1）有一个参数是innodb_read_ahead_threshold，他的默认值是56，意思就是如果顺序的访问了一个区里的多个数据页，访问的数据页的数量超过了这个阈值，此时就会触发预读机制，把下一个相邻区中的所有数据页都加载到缓存里去

（2）如果Buffer Pool里缓存了一个区里的13个连续的数据页，而且这些数据页都是比较频繁会被访问的，此时就会直接触发预读机制，把这个区里的其他的数据页都加载到缓存里去

这个机制是通过参数innodb_random_read_ahead来控制的，他默认是OFF，也就是这个规则是关闭的

所以默认情况下，主要是第一个规则可能会触发预读机制，一下子把很多相邻区里的数据页加载到缓存里去，这些缓存页如果一下子都放在LRU链表的前面，而且他们其实并没什么人会访问的话，那就会如上图，导致本来就在缓存里的一些频繁被访问的缓存页在LRU链表的尾部。

这样的话，一旦要把一些缓存页淘汰掉，刷入磁盘，腾出来空闲缓存页，就会如上所述，把LRU链表尾部一些频繁被访问的缓存页给刷入磁盘和清空掉了！这是完全不合理的，并不应该这样！

4、另外一种可能导致频繁被访问的缓存页被淘汰的场景

接着我们讲另外一种可能导致频繁被访问的缓存页被淘汰的场景，那就是**全表扫描**

这个所谓的全表扫描，意思就是类似如下的SQL语句：SELECT * FROM USERS

此时他没加任何一个where条件，会导致他直接一下子把这个表里所有的数据页，都从磁盘加载到Buffer Pool里去。

这个时候他可能会一下子就把这个表的所有数据页都一一装入各个缓存页里去！此时可能LRU链表中排在前面的一大串缓存页，都是全表扫描加载进来的缓存页！那么如果这次全表扫描过后，后续几乎没用到这个表里的数据呢？

此时LRU链表的尾部，可能全部都是之前一直被频繁访问的那些缓存页！

然后当你要淘汰掉一些缓存页腾出空间的时候，就会把LRU链表尾部一直被频繁访问的缓存页给淘汰掉了，而留下了之前全表扫描加载进来的大量的不经常访问的缓存页！

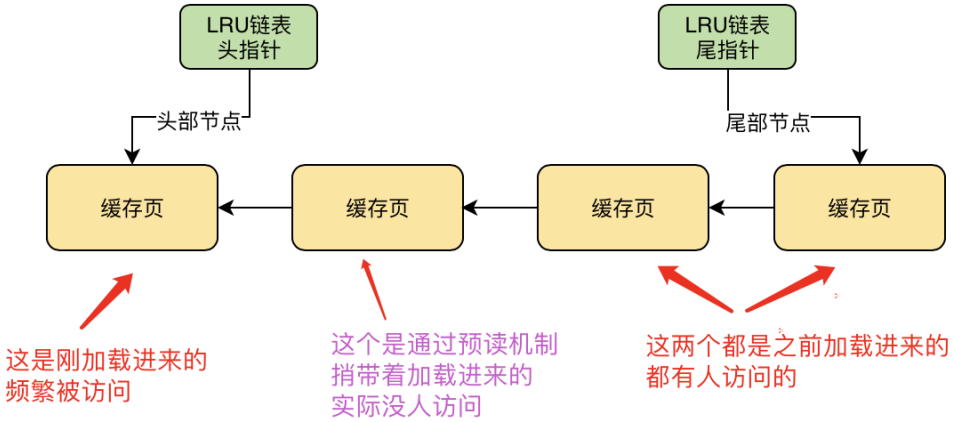
5、总结

所以我们来对今天讲到的内容做一点小小的总结，如果你使用简单的LRU链表的机制，其实是漏洞百出的，因为很可能预读机制，或者全表扫描的机制，都会一下子把大量未来可能不怎么访问的数据页加载到缓存页里去，然后LRU链表的前面全部是这些未来可能不怎么会被访问的缓存页！

而真正之前一直频繁被访问的缓存页可能此时都在LRU链表的尾部了！

如果此时此刻，需要把一些缓存页刷入磁盘，腾出空间来加载新的数据页，那么此时就只能把LRU链表尾部那些一直频繁被访问的缓存页给刷入磁盘了！

最后我们再看一下下面的图示，想必大家是很好理解的。



6、今日思考题

今天希望大家思考一下：

为什么MySQL要设计预读这个机制？

他加载一个数据页到缓存里去的时候，为什么要把一些相邻的数据页也加载到缓存里去呢？这么做的意义在哪里？

是为了应对什么样的一个场景？

希望大家积极思考，在评论区给出自己的答案！

End

狸猫技术窝精品专栏及课程推荐：

《从零开始带你成为消息中间件实战高手》
《21天互联网Java进阶面试训练营》（分布式篇）
《互联网Java工程师面试突击》（第1季）
《互联网Java工程师面试突击》（第3季）
《从零开始带你成为JVM实战高手》

小鹅通提供技术支持

图文 17 MySQL是如何基于冷热数据分离的方案，来优化LRU算法的？

手机观看

603 人次阅读 2020-02-12 07:00:00

详情 评论

MySQL是如何基于冷热数据分离的方案，来优化LRU算法的？

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

1、昨日思考题解答

先给大家解答一下上次给大家布置的思考题，上回我们给大家提了一个问题：为什么MySQL要设计一个预读机制，为什么有时候要把相邻的一些数据页一次性读入到Buffer Pool缓存里去？

道理很简单，说白了还不是为了提升性能么。假设你读取了数据页01到缓存页里去，那么好，接下来有可能会接着顺序读取数据页01相邻的数据页02到缓存页里去，这个时候，是不是可能在读取数据页02的时候要再次发起一次磁盘IO？

所以为了优化性能，MySQL才设计了预读机制，也就是说如果在一个区内，你顺序读取了好多数据页了，比如数据页01~数据页56都被你依次顺序读取了，MySQL会判断，你可能接着会继续顺序读取后面的数据页。

那么此时他就干脆提前把后续的一大堆数据页（比如数据页57~数据页72）都读取到Buffer Pool里去，那么后续你再读取数据页60的时候，是不是就可以直接从Buffer Pool里拿到数据了？

当然理想是上述那样，很丰满，但是现实可能很骨感。你预读的一大堆数据页要是占据了LRU链表的前面部分，可能这些预读的数据页压根儿后续没人会使用，那你这个预读机制就是在捣乱了。

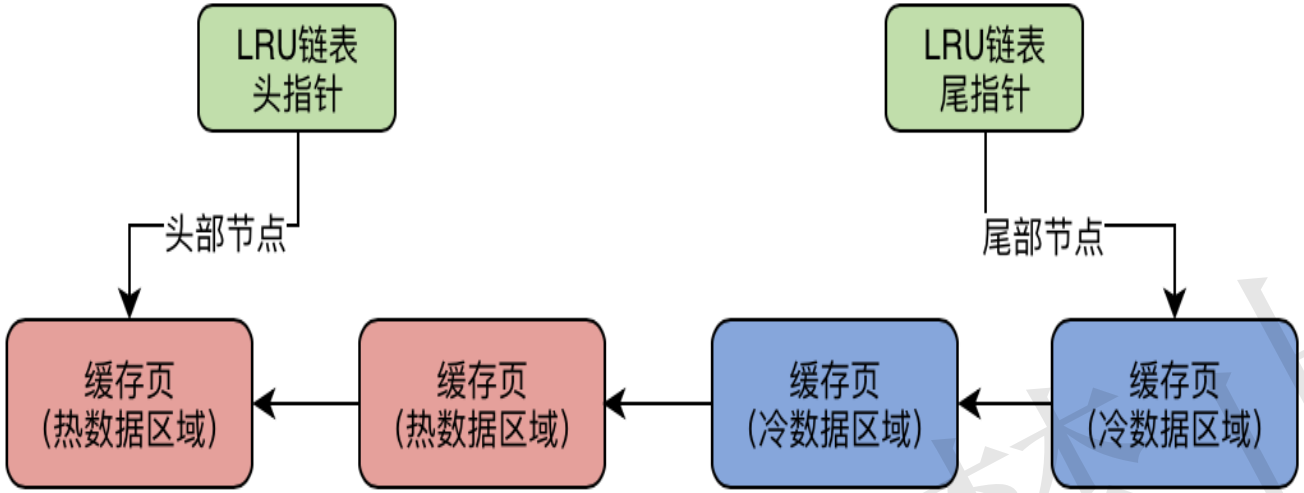
2、基于冷热数据分离的思想设计LRU链表

所以为了解决上一讲我们说的简单的LRU链表的问题，真正MySQL在设计LRU链表的时候，采取的实际上是冷热数据分离的思想。

之前一系列的问题，说白了，不都是因为所有缓存页都混在一个LRU链表里，才导致的么？

所以真正的LRU链表，会被拆分为两个部分，一部分是热数据，一部分是冷数据，这个冷热数据的比例是由innodb_old_blocks_pct参数控制的，他默认是37，也就是说冷数据占比37%。

这个时候，LRU链表实际上看起来是下面这样子的。

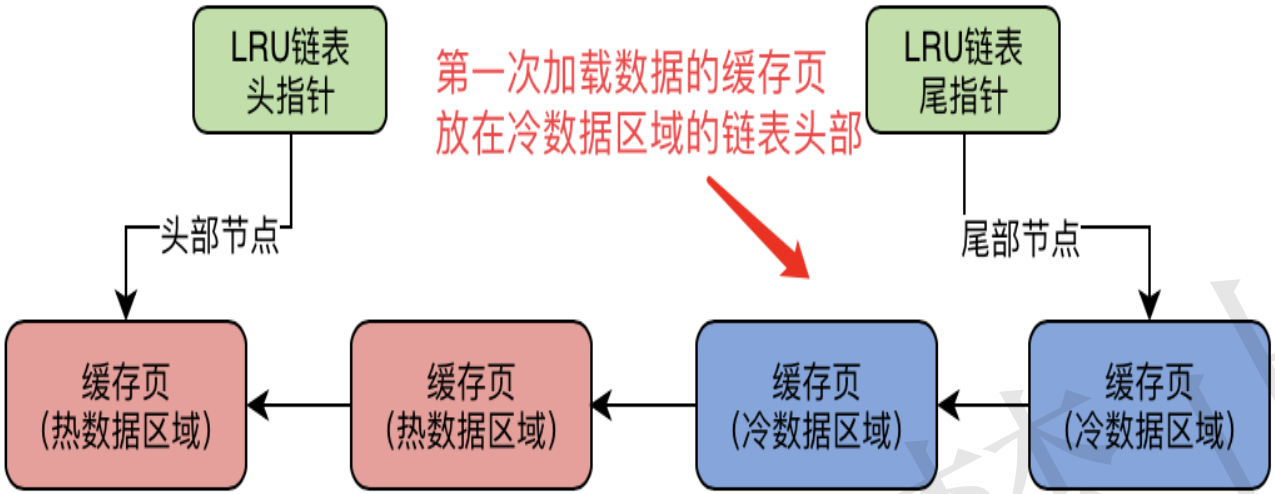


3、数据页第一次被加载到缓存的时候

好，既然我们知道LRU链表已经按照一定的比例被拆分为了冷热两块区域了，那么接下来就来看看在运行期间，冷热两个区域是如何使用的。

首先数据页第一次被加载到缓存的时候，这个时候缓存页会被放在LRU链表的哪个位置呢?

实际上这个时候，缓存页会被放在冷数据区域的链表头部，我们看下面的图，也就是第一次把一个数据页加载到缓存页之后，这个缓存页实际上是被放在下图箭头的位置，也就是冷数据区域的链表头部位置。

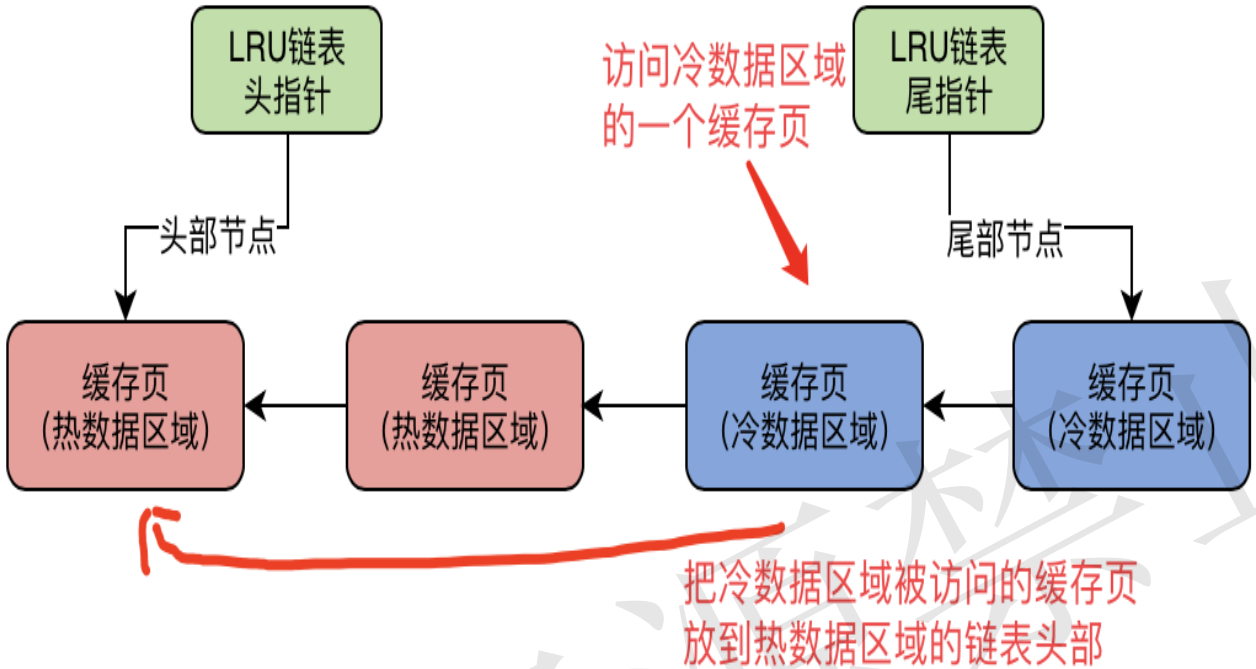


4、冷数据区域的缓存页什么时候会被放入到热数据区域?

接着我们来思考一个问题，第一次被加载了数据的缓存页，都会不停的移动到冷数据区域的链表头部，如上图所示

那么你要知道，冷数据区域的缓存页肯定是会被使用的，那么冷数据区域的缓存页什么时候会放到热数据区域呢?

实际上肯定很多人会想，只要对冷数据区域的缓存页进行了一次访问，就立马把这个缓存页放到热数据区域的头部行不行呢？如下图所示。



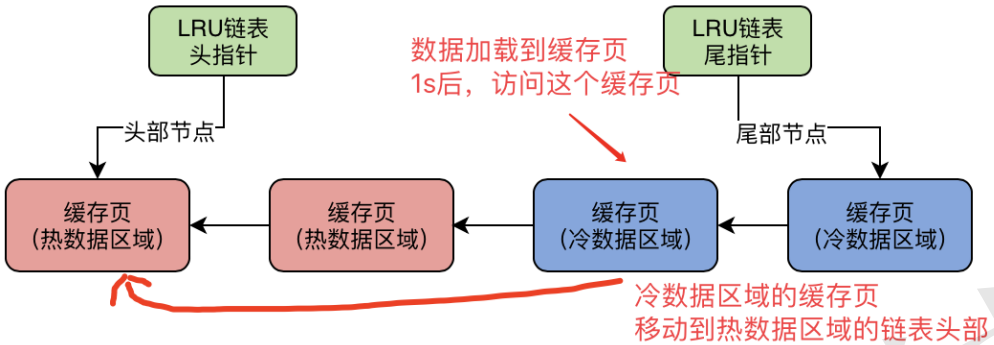
其实这也是不合理的，如果你刚加载了一个数据页到那个缓存页，他是在冷数据区域的链表头部，然后立马（在1ms以内）就访问了一下这个缓存页，之后就再也不访问他了呢？难道这种情况你也要把那个缓存页放到热数据区域的头部吗？

所以MySQL设定了一个规则，他设计了一个innodb_old_blocks_time参数，默认值1000，也就是1000毫秒

也就是说，必须是一个数据页被加载到缓存页之后，在1s之后，你访问这个缓存页，他才会被挪动到热数据区域的链表头部去。

因为假设你加载了一个数据页到缓存去，然后过了1s之后你还访问了这个缓存页，说明你后续很可能会经常要访问它，这个时间限制就是1s，因此只有1s后你访问了这个缓存页，他才会给你把缓存页放到热数据区域的链表头部去。

所以我们看下面的图，文字说明做了一点改动，是数据加载到缓存页之后过了1s，你再访问这个缓存页，他就会被放入热数据区域的链表头部，如果是你数据刚加载到缓存页，在1s内你就访问缓存页，此时他是不会把这个缓存页放入热数据区域的头部的。



5、思考题

今天给大家留一个思考题，大家思考一下，现在我们已经知道了，数据页第一次被加载到缓存页之后，这个缓存页是放在LRU链表的冷数据区域的头部的，然后必须是1s过后访问换个缓存页，他才会被移动到热数据区域的链表头部。

好，那么基于这套冷热数据隔离的方案，LRU链表的冷数据区域放的都是什么样的缓存页？这个问题有点像脑筋急转弯一样，大家脑子一转，就能思考出来了。

大家可以好好思考一下，把你的答案发到评论区里跟其他同学交流。

End

狸猫技术窝精品专栏及课程推荐：

[《从零开始带你成为消息中间件实战高手》]

《21天互联网Java进阶面试训练营》（分布式篇）
《互联网Java工程师面试突击》（第1季）
《互联网Java工程师面试突击》（第3季）
《从零开始带你成为JVM实战高手》

图文 18 基于冷热数据分离方案优化后的LRU链表，是如何解决之前的问题的？ 手机观看

547 人次阅读 2020-02-12 07:00:00

详情 评论

基于冷热数据分离方案优化后的LRU链表，是如何解决之前的问题的？

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

1、对于预读以及全表扫描加载进来的一大堆缓存页

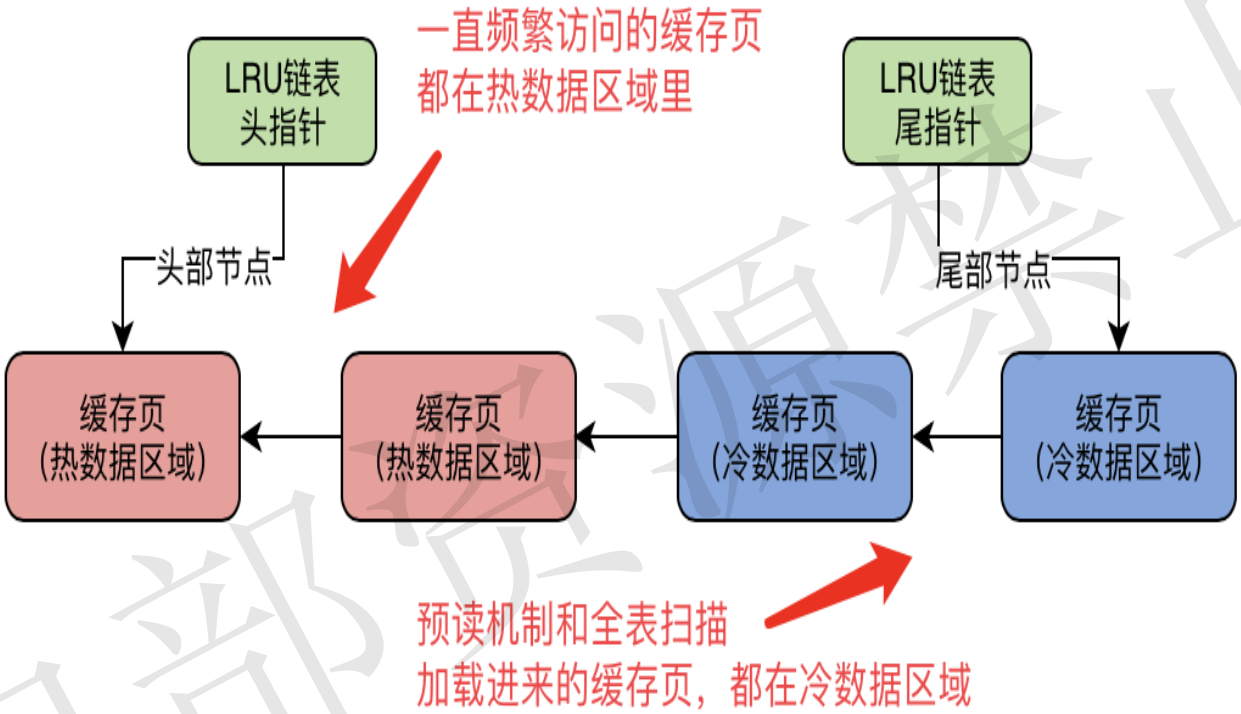
现在我们已经看完了LRU链表的冷热数据分离的方案，那么我们接着看这个冷热数据分离之后的LRU链表，他是如何解决之前遇到的一大堆问题的？

首先我们思考一下，在这样的一个LRU链表方案下，预读机制以及全表扫描加载进来的一大堆缓存页，他们会放在哪里？

明显是放在LRU链表的冷数据区域的前面啊！

假设这个时候热数据区域已经有很多被频繁访问的缓存页了，你会发现热数据区域还是存放被频繁访问的缓存页的，只要热数据区域有缓存页被访问，他还是会被移动到热数据区域的链表头部去。

所以此时你看下图，你会发现，预读机制和全表扫描加载进来的一大堆缓存页，此时都在冷数据区域里，跟热数据区域里的频繁访问的缓存页，是没关系的！



2、预读机制和全表扫描加载进来的缓存页，能进热数据区域吗？

接着我们看第二个问题，预读机制和全表扫描机制加载进来的缓存页，什么时候能进热数据区域呢？

如果你仅仅是一个全表扫描的查询，此时你肯定是在1s内就把一大堆缓存页加载进来，然后就访问了这些缓存页一下，通常这些操作1s内就结束了。

所以基于目前的一个机制，可以确定的是，这种情况下，那些缓存页是不会从冷数据区域转移到热数据区域的！

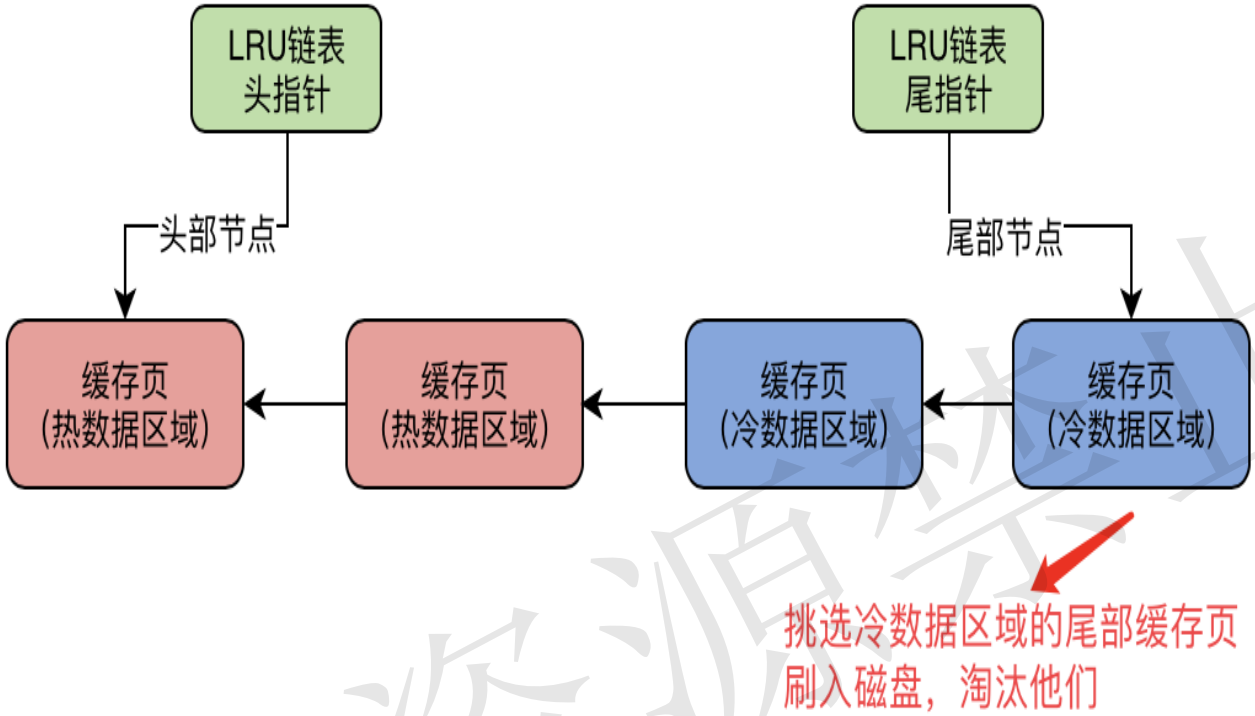
除非你在冷数据区域里的缓存页，在1s之后还被人访问了，那么此时他们就会判定为未来可能会被频繁访问的缓存页，然后移动到热数据区域的链表头部去！

3、如果此时缓存页不够了，需要淘汰一些缓存，会怎么样？

接着我们看，假设此时缓存页不够了，需要淘汰一些缓存页，此时会怎么做？

那就很简单了，直接就是可以找到LRU链表中的冷数据区域的尾部的缓存页，他们肯定是之前被加载进来的，而且加载进来1s过后都没人访问过，说明这个缓存页压根儿就没人愿意去访问他！他就是冷数据！

所以此时就直接淘汰冷数据区域的尾部的缓存页，刷入磁盘，就可以了，我们看下图。



4、之前的一大堆问题解决了吗?

在这样的一套缓存页分冷热数据的加载方案，以及冷数据转化为热数据的时间限制方案，还有就是淘汰缓存页的时候优先淘汰冷数据区域的方案，基于这套方案，大家会发现，之前发现的问题，完美的被解决了。

因为那种预读机制以及全表扫描机制加载进来的数据页，大部分都会在1s之内访问一下，之后可能就再也不访问了，所以这种缓存页基本上都会留在冷数据区域里。然后频繁访问的缓存页还是会留在热数据区域里。

当你要淘汰缓存的时候，优先就是会选择冷数据区域的尾部的缓存页，这就是非常合理的了！他不会让刚加载进来的缓存页占据LRU链表的头部，频繁访问的缓存页在LRU链表的尾部，淘汰的时候淘汰尾部的频繁访问的缓存页了！

问题完美的被解决了。

这就是LRU链表冷热数据分离的一套机制。

5、总结

通过这几篇文章的学习，我们已经彻底搞定了LRU链表的设计机制，刚加载数据的缓存页都是放冷数据区域的头部的，1s过后被访问了才会放热数据区域的头部，热数据区域的缓存页被访问了，就会自动放到头部去。

这样的话，实际上冷数据区域放的都是加载进来的缓存页，最多在1s内被访问过，之后就再也没访问过的冷数据缓存页！

而加载进来之后在1s过后还经常被访问的缓存页，都放在了热数据区域里，他们进行了冷热数据的隔离！

这样的话，在淘汰缓存的时候，一定是优先淘汰冷数据区域几乎不怎么被访问的缓存页的！也希望大家好好吸收这种冷热数据隔离的思想，尽可能让热数据和冷数据分开，避免冷数据影响热数据的访问！

6、一个发散思考问题

今天给大家留一个发散思考的问题，大家觉得对于这种缓存中同时包含冷热数据的场景，如果你是在Redis中放了你业务系统的很多缓存数据，其中也是冷热数据都有的，此时可能会有什么问题？

那么针对这样的一个问题，你是否可以考虑在你自己的缓存设计中，运用冷热隔离的思想来优化重构呢？

这是一个非常值得思考的问题，请大家积极思考，在评论区踊跃发言

End

狸猫技术窝精品专栏及课程推荐：

《从零开始带你成为消息中间件实战高手》
《21天互联网Java进阶面试训练营》（分布式篇）
《互联网Java工程师面试突击》（第1季）
《互联网Java工程师面试突击》（第3季）
《从零开始带你成为JVM实战高手》

图文 19 MySQL是如何将LRU链表的使用性能优化到极致的?

手机观看

535 人次阅读 2020-02-13 09:19:14

详情 评论

MySQL是如何将LRU链表的使用性能优化到极致的?

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

1、昨日第一个思考题的解答

昨天第一个思考题，我们是让大家思考一下，在LRU链表的冷数据区域中的都是什么样的数据呢?

其实大家脑筋一转就知道了，大部分应该都是预读加载进来的缓存页，加载进来1s之后都没人访问的，然后包括全表扫描或者一些大的查询语句，加载一堆数据到缓存页，结果都是1s之内访问了一下，后续就不再访问这些表的数据了。

类似这些数据，统统都会放在冷数据区域里。

2、昨日第二个思考题的解答

接着我们来说一下昨日第二个思考题的解答，昨天第二个思考题是让大家想了一下，对于我们开发的Java系统，如果在Redis里存放了很多缓存数据，那么此时会不会有类似冷热数据的问题？应该如何优化和解决呢？

答案是：那必然是存在一些问题的。

常见的一个场景就是电商系统里的商品缓存数据，假设你有1亿个商品，然后只要查询商品的时候发现商品不在缓存里，就给他放到缓存里去，你要这么搞的话，必然导致大量的不怎么经常访问的商品会被放在Redis缓存里！

经常被访问的商品其实就是热数据，不经常被访问的商品其实就是冷数据，我们应该尽量让Redis里放的都是经常访问的热数据，而不是大量的冷数据。因为你放一大堆不怎么经常访问的商品在Redis里，那么他占用了很多内存，而且后续还不怎么会访问到他们！

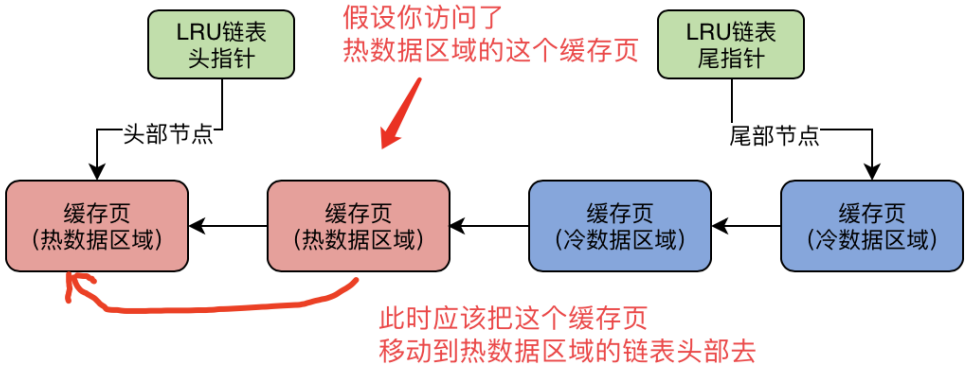
所以我们在设计缓存机制的时候，经常会考虑**热数据的缓存预加载**

也就是说，每天统计出来哪些商品被访问的次数最多，然后晚上的时候，系统启动一个定时作业，把这些热门商品的数据，预加载到Redis里。那么第二天是不是对热门商品的访问就自然会优先走Redis缓存了？

3、LRU链表的热数据区域是如何进行优化的？

接着我们来看看LRU链表的热数据区域的一个性能优化的点，就是说，在热数据区域中，如果你访问了一个缓存页，是不是应该要把他立马移动到热数据区域的链表头部去？

我们看下面的图示。



但是你要知道，热数据区域里的缓存页可能是经常被访问的，所以这么频繁的进行移动是不是性能也并不是太好？也没这个必要。

所以说，LRU链表的热数据区域的访问规则被优化了一下，即你只有在热数据区域的后3/4部分的缓存页被访问了，才会给你移动到链表头部去。

如果你是热数据区域的前面1/4的缓存页被访问，他是不会移动到链表头部去的。

举个例子，假设热数据区域的链表里有100个缓存页，那么排在前面的25个缓存页，他即使被访问了，也不会移动到链表头部去的。但是对于排在后面的75个缓存页，他只要被访问，就会移动到链表头部去。

这样的话，他就可以尽可能的减少链表中的节点移动了。

4、一个脑筋急转弯的思考题

今天给大家出一个脑筋急转弯的小思考题，大家看了以后都可以在评论区里回答一下，如果回答错误的同学，那真的得接受一点惩罚了！

这个问题就是：如果一个缓存页在冷数据区域的尾巴上，已经超过1s了，此时这个缓存页被访问了一下，那么他此时会移动到冷数据区域的链表头部吗？**注意，是冷数据区域的链表头部！**

End

狸猫技术窝精品专栏及课程推荐：

- 《从零开始带你成为消息中间件实战高手》
- 《21天互联网Java进阶面试训练营》（分布式篇）
- 《互联网Java工程师面试突击》（第1季）
- 《互联网Java工程师面试突击》（第3季）
- 《从零开始带你成为JVM实战高手》

图文 20 对于LRU链表中尾部的缓存页，是如何淘汰他们刷入磁盘的?

手机观看

详情 评论

对于LRU链表中尾部的缓存页，是如何淘汰他们刷入磁盘的?

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

1、Buffer Pool的缓存页以及几个链表的使用回顾

接着我们来讲讲，你的Buffer Pool在运行中被使用的时候，实际上会频繁的从磁盘上加载数据页到他的缓存页里去，然后free链表、flush链表、lru链表都会在使用的时候同时被使用

比如数据加载到一个缓存页，free链表里会移除这个缓存页，然后lru链表的冷数据区域的头部会放入这个缓存页。

然后如果你要是修改了一个缓存页，那么flush链表中会记录这个脏页，lru链表中还可能会把你从冷数据区域移动到热数据区域的头部去。

如果你是查询了一个缓存页，那么此时就会把这个缓存页在lru链表中移动到热数据区域去，或者在热数据区域中也有可能会移动到头部去。

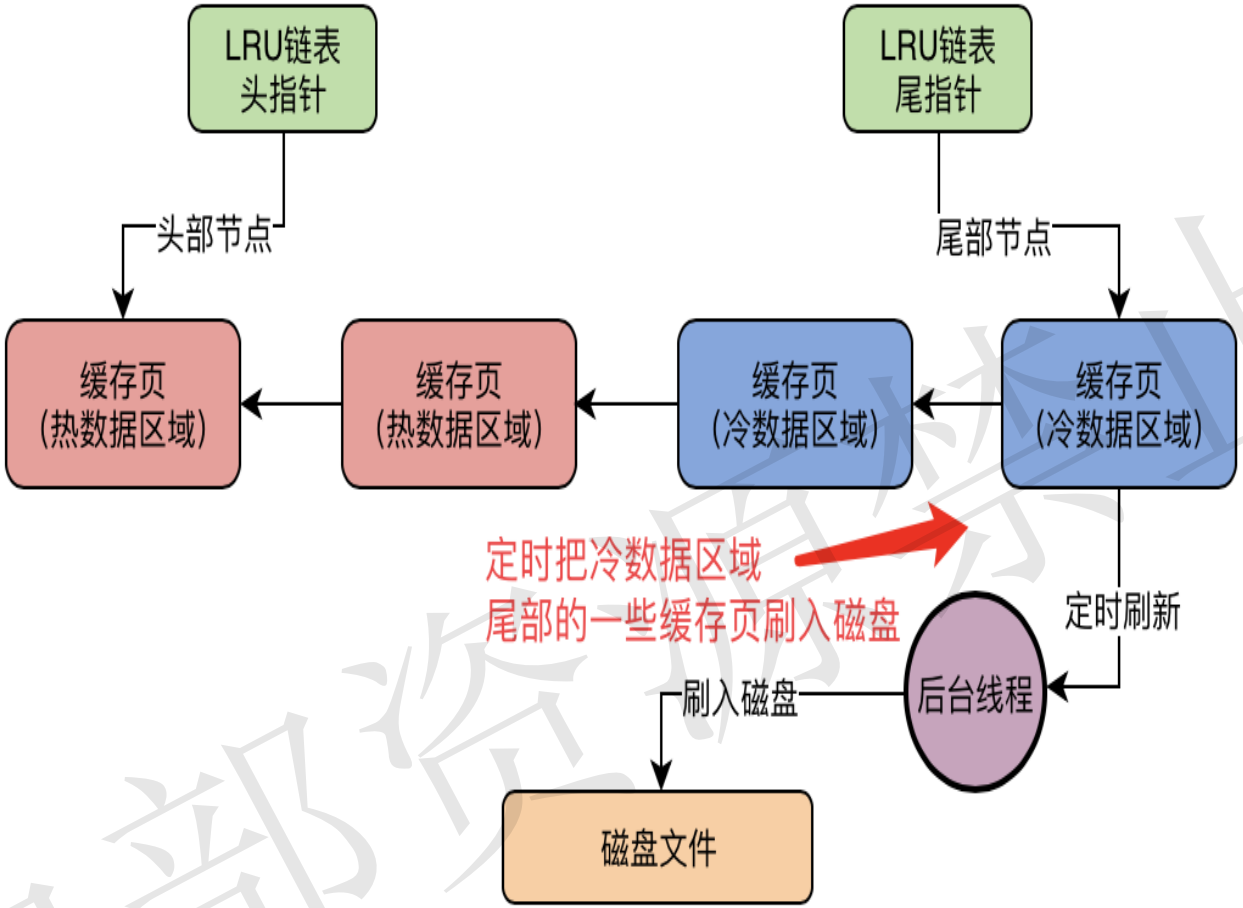
总之，MySQL在执行CRUD的时候，首先就是大量的操作缓存页以及对应的几个链表。然后在缓存页都满的时候，必然要想办法把一些缓存页给刷入磁盘，然后清空这几个缓存页，接着把需要的数据页加载到缓存页里去！

我们已经知道，他是根据LRU链表去淘汰缓存页的，那么他到底是什么时候把LRU链表的冷数据区域中的缓存页刷入磁盘的呢？实际上他有几个时机。

2、定时把LRU尾部的部分缓存页刷入磁盘

首先第一个时机，并不是在缓存页满的时候，才会挑选LRU冷数据区域尾部的几个缓存页刷入磁盘，而是有一个后台线程，他会运行一个定时任务，这个定时任务每隔一段时间就会把LRU链表的冷数据区域的尾部的一些缓存页，刷入磁盘里去，清空这几个缓存页，把他们加入回free链表去！

所以实际上在缓存页没用完的时候，可能就会清空一些缓存页了，我们看下面的图示。



所以大家会发现，只要有这个后台线程定时运行，可能你的缓存页都没用完呢，人家就给你把一批冷数据的缓存页刷入磁盘，清空出来一批缓存页，那么你就多了一批可以使用的空闲缓存页了！

所以如果在一个动态的运行效果中思考，大概就是你不停的加载数据到一些空闲的缓存页里去，然后这些缓存页可能被使用，会在lru链表中各种移动。然后同时有一个后台线程还不停的把冷数据区域的一些不用的缓存页刷入磁盘中，清空一些缓存页出来。

只要有缓存页被刷人磁盘，大家可以想象一下，那么这个缓存页必然会加入到free链表中，从flush链表中移除，从lru链表中移除。

3、把flush链表中的一些缓存页定时刷入磁盘

如果仅仅是把LRU链表中的冷数据区域的缓存页刷入磁盘，大家觉得够吗？

明显不够啊，因为在lru链表的热数据区域里的很多缓存页可能也会被频繁的修改，难道他们永远都不刷入磁盘中了吗？

所以这个后台线程同时也会在MySQL不怎么繁忙的时候，找个时间把flush链表中的缓存页都刷入磁盘中，这样被你修改过的数据，迟早都会刷入磁盘的！

只要flush链表中的一波缓存页被刷入了磁盘，那么这些缓存页也会从flush链表和lru链表中移除，然后加入到free链表中去！

所以你可以理解为，你一边不停的加载数据到缓存页里去，不停的查询和修改缓存数据，然后free链表中的缓存页不停的在减少，flush链表中的缓存页不停的在增加，lru链表中的缓存页不停的在增加和移动。

另外一边，你的后台线程不停的在把lru链表的冷数据区域的缓存页以及flush链表的缓存页，刷入磁盘中来清空缓存页，然后flush链表和lru链表中的缓存页在减少，free链表中的缓存页在增加。

这就是一个动态运行起来的效果！

4、实在没有空闲缓存页了怎么办？

那么实在没有空闲缓存页了怎么办呢？

此时可能所有的free链表都被使用了，然后flush链表中有一大堆被修改过的缓存页，lru链表中有一大堆的缓存页，根据冷热数据进行了分离，大致是如此的效果。

这个时候如果要从磁盘加载数据页到一个空闲缓存页中，此时就会从LRU链表的冷数据区域的尾部找到一个缓存页，他一定是最不经常使用的缓存页！然后把他刷入磁盘和清空，然后把数据页加载到这个腾出来的空闲缓存页里去！

这就是MySQL的Buffer Pool缓存机制的一整套运行原理！我们已经完整的讲完了缓存页的加载和使用，以及free链表、flush链表、lru链表是怎么使用的，包括缓存页是如何刷入磁盘腾出来空闲缓存页的，以及缓存页没有空闲的时候应该怎么处理。

大家首先理解了最近几篇文章之后，就应该完全理解了，MySQL在执行CRUD操作的时候，是如何尽可能基于内存中的缓存来处理的。

5、今日思考题

今天我们来给大家一个思考题，大家发现没有，如果你在执行CRUD的时候要从磁盘加载数据页到Buffer Pool的缓存页的时候，一旦此时没有空闲的缓存页，就必须从LRU链表的冷数据区域的尾部把一个缓存页刷入磁盘，然后腾出来一个空闲的缓存页，接着你才能基于缓存数据来执行这个CRUD的操作。

但是如果频繁的出现这样的一个情况，那你的很多CRUD执行的时候，难道都要先刷一个缓存页到磁盘上去？然后再从磁盘上读取一个数据页到空闲的缓存页里来？这样岂不是每次CRUD操作都要执行两次磁盘IO？那么性能岂不是会极差？

所以我们来思考一个问题：**你的MySQL的内核参数，应该如何优化，优化哪些地方的行为，才能够尽可能的避免在执行CRUD的时候，经常要先刷一个缓存页到磁盘上去，才能读取一个磁盘上的数据页到空闲缓存页里来？**

End

狸猫技术窝精品专栏及课程推荐：

《从零开始带你成为消息中间件实战高手》
《21天互联网Java进阶面试训练营》（分布式篇）
《互联网Java工程师面试突击》（第1季）
《互联网Java工程师面试突击》（第3季）
《从零开始带你成为JVM实战高手》

图文 21 生产经验：如何通过多个Buffer Pool来优化数据库的并发性能？ 手机观看

563 人次阅读 2020-02-14 07:44:37

详情 评论

生产经验：如何通过多个Buffer Pool来优化数据库的并发性能？

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

1、Buffer Pool在访问的时候需要加锁吗？

前面我们已经把Buffer Pool的整体工作原理和设计原理都已经给大家分析的比较清楚了，基本上目前大家都能够很好的理解，我们对MySQL执行CRUD操作时候的第一步，就是利用Buffer Pool里的缓存来更新或者查询。

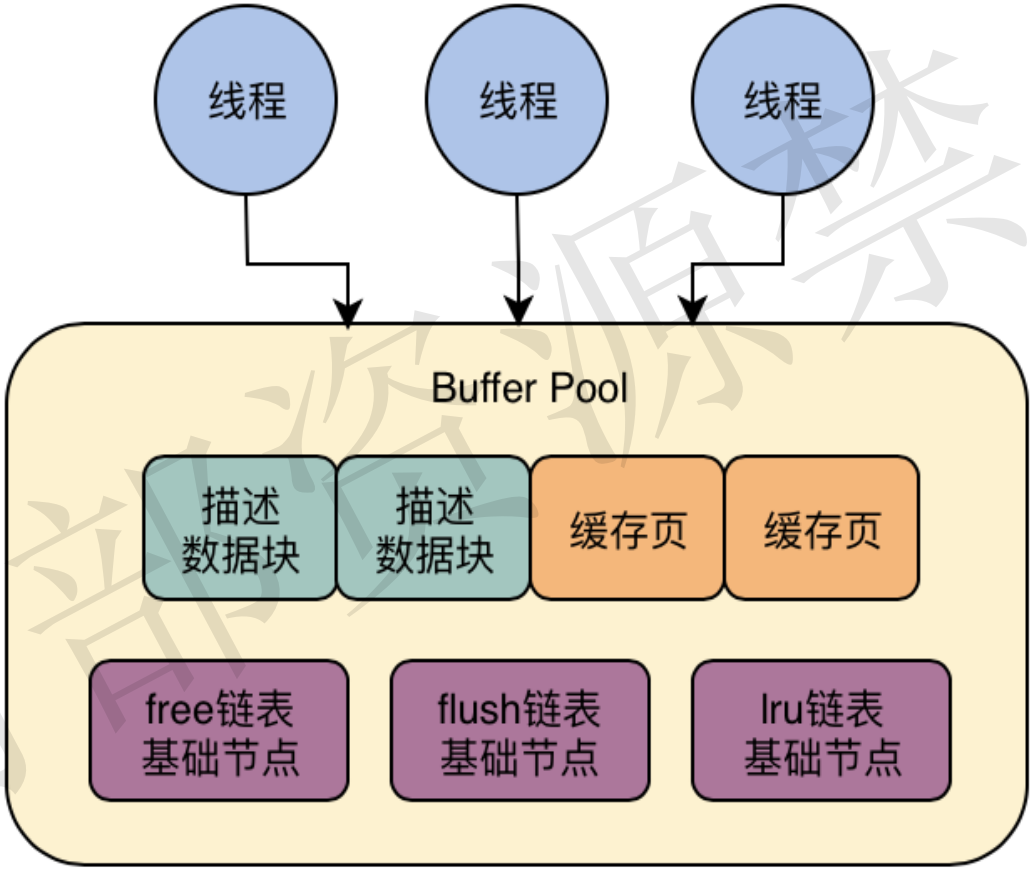
那么既然已经把Buffer Pool的原理都讲的差不多了，接着我们就可以来给大家说说Buffer Pool在实际生产环境运行中的一些经验，应该如何对Buffer Pool进行一些配置上的优化，来提升他的访问性能呢？

首先我们来看第一个问题，大家都知道，Buffer Pool其实本质就是一大块内存数据结构，由一大堆的缓存页和描述数据块组成的，然后加上了各种链表（free、flush、lru）来辅助他的运行。

好，那么这个时候假设MySQL同时接收到了多个请求，他自然会用多个线程来处理这多个请求，每个线程会负责处理一个请求，对吧？

然后这多个线程是不是应该会同时去访问Buffer Pool呢？就是同时去操作里面的缓存页，同时操作一个free链表、flush链表、lru链表，是吗？

我们看下图，就是一个多线程并发访问Buffer Pool的示意图。



那么大家思考一下，现在多个线程来并发的访问这个Buffer Pool了，此时他们都是在访问内存里的一些共享的数据结构，比如说缓存页、各种链表之类的，那么此时是不是必然要进行加锁？

对，多线程并发访问一个Buffer Pool，必然是要加锁的，然后让一个线程先完成一系列的操作，比如说加载数据页到缓存页，更新free链表，更新lru链表，然后释放锁，接着下一个线程再执行一系列的操作。

2、多线程并发访问加锁，数据库的性能还能好吗？

既然我们已经解决了第一个问题，就是多线程并发访问一个Buffer Pool的时候必然会加锁，然后很多线程可能要串行着排队，一个一个的依次执行自己要执行的操作，那么此时我问大家第二个问题，此时数据库的性能还能好吗？

应该这么说，即使就一个Buffer Pool，即使多个线程会加锁串行着排队执行，其实性能也差不到哪儿去。

因为大部分情况下，每个线程都是查询或者更新缓存页里的数据，这个操作是发生在内存里的，基本都是微秒级的，很快很快，包括更新free、flush、lru这些链表，他因为都是基于链表进行一些指针操作，性能也是极高的。

所以即使每个线程排队加锁，然后执行一系列操作，数据库的性能倒也是还可以的。

但是再怎么可以，你毕竟也是每个线程加锁然后排队一个一个操作，这也不是特别的好，特别是有的时候你的线程拿到锁之后，他可能要从磁盘里读取数据页加载到缓存页里去，这还发生了一次磁盘IO呢！所以他要是进行磁盘IO的话，也许耗时就会多一些，那么后面排队等他的线程自然就多等一会儿了！

3、MySQL的生产优化经验：多个Buffer Pool优化并发能力

因此这里我们给大家介绍一个MySQL的生产环境优化经验，就是可以给MySQL设置多个Buffer Pool来优化他的并发能力。

一般来说，MySQL默认的规则是，如果你给Buffer Pool分配的内存小于1GB，那么最多就只会给你一个Buffer Pool。

但是如果你的机器内存很大，那么你必然会给Buffer Pool分配较大的内存，比如给他个8G内存，那么此时你是同时可以设置多个Buffer Pool的，比如说下面的MySQL服务器端的配置。

[server]

innodb_buffer_pool_size = 8589934592

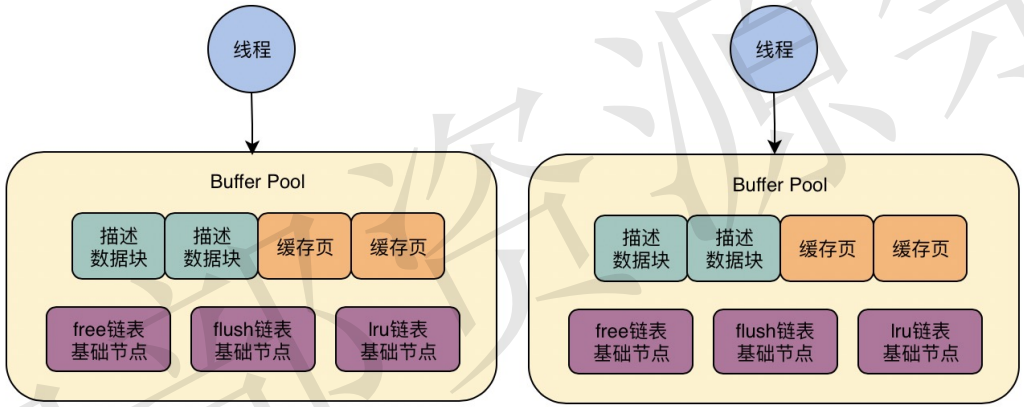
innodb_buffer_pool_instances = 4

我们给buffer pool设置了8GB的总内存，然后设置了他应该有4个Buffer Pool，此时就是说，每个buffer pool的大小就是2GB

这个时候，MySQL在运行的时候就会有4个Buffer Pool了！每个Buffer Pool负责管理一部分的缓存页和描述数据块，有自己独立的free、flush、lru等链表。

这个时候，假设多个线程并发过来访问，那么不就可以把压力分散开来了吗？有的线程访问这个buffer pool，有的线程访问那个buffer pool。

我们看下图：



所以这样的话，一旦你有了多个buffer pool之后，你的多线程并发访问的性能就会得到成倍的提升，因为多个线程可以在不同的buffer pool中加锁和执行自己的操作，大家可以并发来执行了！

所以这个在实际生产环境中，设置多个buffer pool来优化高并发访问性能，是mysql一个很重要的优化技巧。

End

狸猫技术窝精品专栏及课程推荐：

- 《从零开始带你成为消息中间件实战高手》
- 《21天互联网Java进阶面试训练营》（分布式篇）
- 《互联网Java工程师面试突击》（第1季）
- 《互联网Java工程师面试突击》（第3季）
- 《从零开始带你成为JVM实战高手》

图文 22 生产经验：如何通过chunk来支持数据库运行期间的Buffer Pool动态调整?

手机观看

详情 评论

生产经验：如何通过chunk来支持数据库运行期间的Buffer Pool动态调整?

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

1、buffer pool这种大块头，能在运行期间动态调整大小吗?

上一篇文章给大家分析了一下buffer pool在多线程并发访问的时候的一些问题，以及通过多个buffer pool是如何优化多线程并发访问性能的。

那么这一篇文章我们接着分析下一个问题，那就是buffer pool这种大块头数据结构，在数据库运行期间，可以动态的调整他的大小吗?

其实如果就我们讲的这套原理的话，buffer pool在运行期间是不能动态的调整自己的大小的

为什么呢？因为动态调整buffer pool大小，比如buffer pool本来是8G，运行期间你给调整为16G了，此时是怎么实现的呢？

就是需要这个时候向操作系统申请一块新的16GB的连续内存，然后把现在的buffer pool中的所有缓存页、描述数据块、各种链表，都拷贝到新的16GB的内存中去。这个过程是极为耗时的，性能很低下，是不可以接受的！

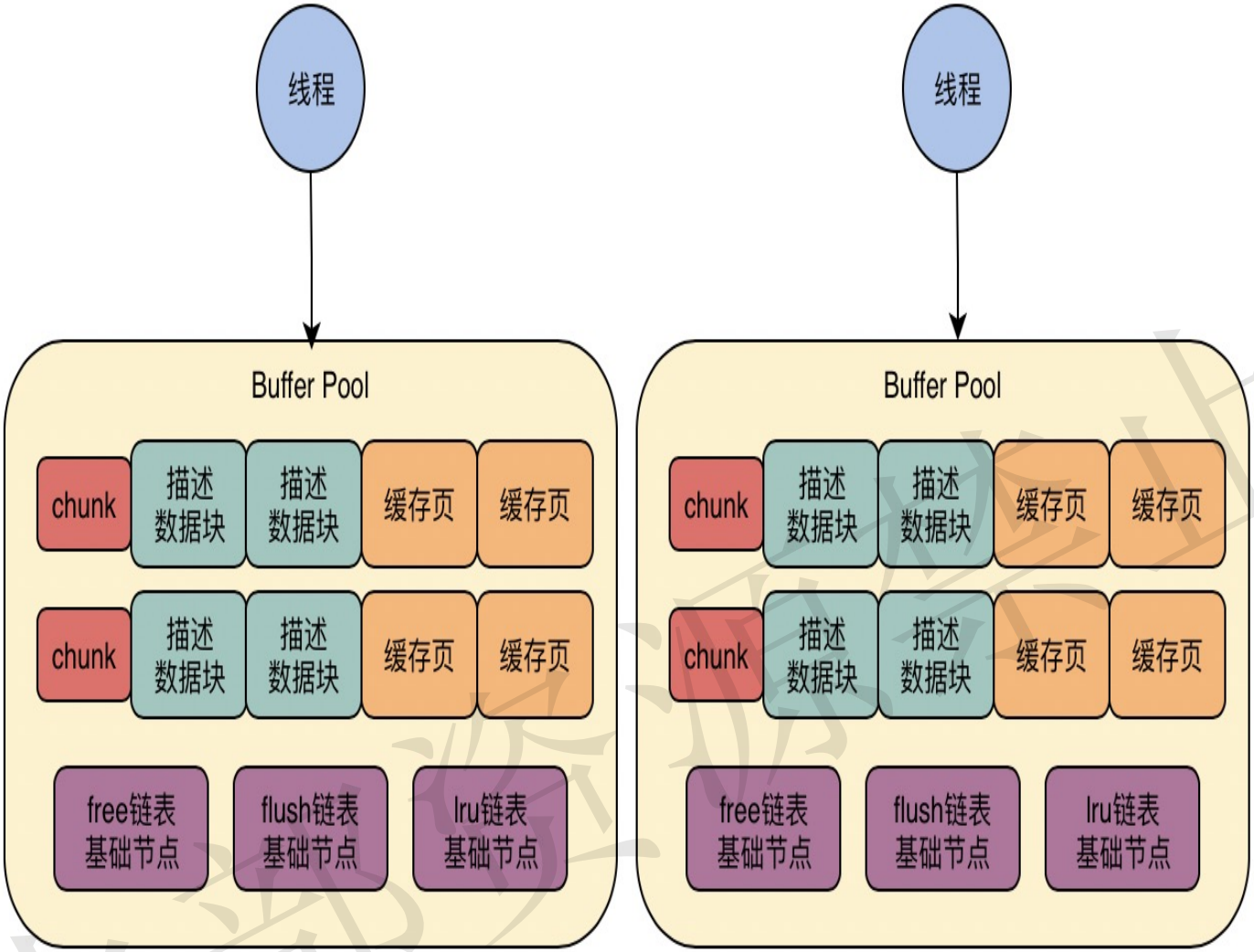
所以就目前讲解的这套原理，buffer pool是绝对不能支持运行期间动态调整大小的。

2、如何基于chunk机制把buffer pool给拆小呢？

但是MySQL自然会想办法去做一些优化的，他实际上设计了一个chunk机制，也就是说buffer pool是由很多chunk组成的，他的大小是innodb_buffer_pool_chunk_size参数控制的，默认值就是128MB。

所以实际上我们可以来做一个假设，比如现在我们给buffer pool设置一个总大小是8GB，然后有4个buffer pool，那么每个buffer pool就是2GB，此时每个buffer pool是由一系列的128MB的chunk组成的，也就是说每个buffer pool会有16个chunk。

然后每个buffer pool里的每个chunk里就是一系列的描述数据块和缓存页，每个buffer pool里的多个chunk共享一套free、flush、lru这些链表，此时的话，看起来可能大致如下图所示。



在上面的图里，可以清晰的看到，每个buffer pool里已经有了多个chunk，每个chunk就是一系列的描述数据块和缓存页，这样的话，就是把buffer pool按照chunk为单位，拆分为了一系列的小数据块，但是每个buffer pool是共用一套free、flush、lru的链表的。

3、基于chunk机制是如何支持运行期间，动态调整buffer pool大小的?

那么现在有了上面讲的这套chunk机制，就可以支持动态调整buffer pool大小了。

比如我们buffer pool现在总大小是8GB，现在要动态加到16GB，那么此时只要申请一系列的128MB大小的chunk就可以了，只要每个chunk是连续的128MB内存就行了。然后把这些申请到的chunk内存分配给buffer pool就行了。

有个这个chunk机制，此时并不需要额外申请16GB的连续内存空间，然后还要把已有的数据进行拷贝。

给大家讲解这个chunk机制，倒不是让大家在数据库运行的时候动态调整buffer pool大小，其实这不是重点，重点是大家要了解数据库的buffer pool的真实的数据结构，是可以由多个buffer pool组成的，每个buffer pool是多个chunk组成的，然后你只要知道他运行期间可以支持动态调整大小就可以了。

4、昨日思考题解答

现在我们来解答一下昨天的思考题，昨天让大家思考了一下，到底如何避免你执行crud的时候，频繁的发现缓存页都用完了，完了还得先把一个缓存页刷入磁盘腾出一个空闲缓存页，然后才能从磁盘读取一个自己需要的数据页到缓存页里来。

如果频繁这么搞，那么很多crud操作，每次都要执行两次磁盘IO，一次是缓存页刷入磁盘，一次是数据页从磁盘里读取出来，性能是很不高的。

其实结合我们了解到的buffer pool的运行原理就可以知道，如果要避免上述问题，说白了就是避免缓存页频繁的被使用完毕。那么我们知道实际上你在使用缓存页的过程中，有一个后台线程会定时把LRU链表冷数据区域的一些缓存页刷入磁盘中。

所以本质上缓存页一边会被你使用，一边会被后台线程定时的释放掉一批。

所以如果你的缓存页使用的很快，然后后台线程释放缓存页的速度很慢，那么必然导致你频繁发现缓存页被使用完了。但是缓存页被使用的速度你是没法控制的，因为那是由你的Java系统访问数据库的并发程度来决定的，你高并发访问数据库，缓存页必然使用的很快了!

然后你后台线程定时释放一批缓存页，这个过程也很难去优化，因为你要是释放的过于频繁了，那么后台线程执行磁盘IO过于频繁，也会影响数据库的性能。

所以这里的关键点就在于，你的buffer pool有多大！

如果你的数据库要抗高并发的访问，那么你的机器必然要配置很大的内存空间，起码是32GB以上的，甚至64GB或者128GB。此时你就可以给你的buffer pool设置很大的内存空间，比如20GB，48GB，甚至80GB。

这样的话，你会发现高并发场景下，数据库的buffer pool缓存页频繁的被使用，但是你后台线程也在定时释放一些缓存页，那么综合下来，空闲的缓存页还是会以一定的速率逐步逐步的减少。

因为你的buffer pool内存很大，所以空闲缓存页是很多很多的，即使你的空闲缓存页逐步的减少，也可能需要较长时间才会发现缓存页用完了，此时才会出现一次crud操作执行的时候，先刷缓存页到磁盘，再读取数据页到缓存页来，这种情况是不会出现的太频繁的！

而一旦你的数据库高峰过去，此时缓存页被使用的速率下降了很多很多，然后后台线程会定是基于flush链表和lru链表不停的释放缓存页，那么你的空闲缓存页的数量又会在数据库低峰的时候慢慢的增加了。

所以线上的MySQL在生产环境中，buffer pool的大小、buffer pool的数量，这都是要用心设置和优化的，因为多MySQL的性能和并发能力，都会有较大的影响。

5、实践思考题

请每位同学，去看看自己负责的系统的buffer pool大小、buffer pool数量、chunk大小，然后看看自己的数据库的机器配置，思考一下，当前设置是否合理？为什么要这样设置？

大家可以把自己的思考发在评论区一起交流。

End

狸猫技术窝精品专栏及课程推荐：

- 《从零开始带你成为消息中间件实战高手》
- 《21天互联网Java进阶面试训练营》（分布式篇）
- 《互联网Java工程师面试突击》（第1季）
- 《互联网Java工程师面试突击》（第3季）
- 《从零开始带你成为JVM实战高手》

图文 23 生产经验：在生产环境中，如何基于机器配置来合理设置Buffer Pool?

手机观看

详情 评论

生产经验：在生产环境中，如何基于机器配置来合理设置Buffer Pool?

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

1、生产环境中应该给buffer pool设置多少内存?

今天这篇文章我们接着上一次讲解的Buffer Pool的一些内存划分的原理，来给大家最后总结一下，在生产环境中到底应该如何设置Buffer Pool的大小呢。

首先考虑第一个问题，我们现在数据库部署在一台机器上，这台机器可能有个8G、16G、32G、64G、128G的内存大小，那么此时buffer pool应该设置多大呢?

有的人可能会想，假设我有32G内存，那么给buffer pool设置个30GB得了，这样的话，MySQL大量的crud操作都是基于内存来执行的，性能那是绝对高！

但是这么想就大错特错了，你要知道，虽然你的机器有32GB的内存，但是你的操作系统内核就要用掉起码几个GB的内存！

然后你的机器上可能还有别的东西在运行，是不是也要内存？然后你的数据库里除了buffer pool是不是还有别的内存数据结构，是不是也要内存？所以上面那种想法是绝对不可取的！

如果你胡乱设置一个特别大的内存给buffer，会导致你的mysql启动失败的，他启动的时候就发现操作系统的内存根本不够用了！

所以通常来说，我们建议一个比较合理的、健康的比例，是给buffer pool设置你的机器内存的50%~60%左右

比如你有32GB的机器，那么给buffer设置个20GB的内存，剩下的留给OS和其他人来用，这样比较合理一些。

假设你的机器是128GB的内存，那么buffer pool可以设置个80GB左右，大概就是这样的一个规则。

2、buffer pool总大小=(chunk大小 * buffer pool数量)的2倍数

接着确定了buffer pool的总大小之后，就得考虑一下设置多少个buffer pool，以及chunk的大小了

此时要记住，有一个很关键的公式就是：buffer pool总大小=(chunk大小 * buffer pool数量)的倍数

比如默认的chunk大小是128MB，那么此时如果你的机器的内存是32GB，你打算给buffer pool总大小在20GB左右，那么你得算一下，此时你的buffer pool的数量应该是多少个呢？

假设你的buffer pool的数量是16个，这是没问题的，那么此时chunk大小 * buffer pool的数量 = 16 * 128MB = 2048MB，然后buffer pool总大小如果是20GB，此时buffer pool总大小就是2048MB的10倍，这就符合规则了。

当然，此时你可以设置多一些buffer pool数量，比如设置32个buffer pool，那么此时buffer pool总大小（20GB）就是（chunk大小128MB * 32个buffer pool）的5倍，也是可以的。

那么此时你的buffer pool大小就是20GB，然后buffer pool数量是32个，每个buffer pool的大小是640MB，然后每个buffer pool包含5个128MB的chunk，算下来就是这么一个结果了。

3、一点总结

我们再来做一点总结，就是说你的数据库在生产环境运行的时候，你必须根据机器的内存设置合理的buffer pool的大小，然后设置buffer pool的数量，这样的话，可以尽可能的保证你的数据库的高性能和高并发能力。

然后在线上运行的时候，buffer pool是有多个的，每个buffer pool里多个chunk但是共用一套链表数据结构，然后执行crud的时候，就会不停的加载磁盘上的数据页到缓存页里来，然后会查询和更新缓存页里的数据，同时维护一系列的链表结构。

然后后台线程定时根据lru链表和flush链表，去把一批缓存页刷入磁盘释放掉这些缓存页，同时更新free链表。

如果执行crud的时候发现缓存页都满了，没法加载自己需要的数据页进缓存，此时就会把lru链表冷数据区域的缓存页刷入磁盘，然后加载自己需要的数据页进来。

整个buffer pool的结构设计以及工作原理，就是上面我们总结的这套东西了，大家只要理解了这个，首先你对MySQL执行crud的时候，是如何在内存里查询和更新数据的，你就彻底明白了。

接着我们后面继续探索undo log、redo log、事务机制、事务隔离、锁机制，这些东西，一点点就把MySQL他的数据更新、事务、锁这些原理，全部搞清楚了，同时中间再配合穿插一些生产经验、实战案例。

4、SHOW ENGINE INNODB STATUS

当你的数据库启动之后，你随时可以通过上述命令，去查看当前innodb里的一些具体情况，执行SHOW ENGINE INNODB STATUS就可以了。此时你可能会看到如下一系列的东西：

```
Total memory allocated xxxx;
Dictionary memory allocated xxx
Buffer pool size   xxxx
```

Free buffers xxx

Database pages xxx

Old database pages xxxx

Modified db pages xx

Pending reads 0

Pending writes: LRU 0, flush list 0, single page 0

Pages made young xxxx, not young xxx

xx youngs/s, xx non-youngs/s

Pages read xxxx, created xxx, written xxx

xx reads/s, xx creates/s, 1xx writes/s

Buffer pool hit rate xxx / 1000, young-making rate xxx / 1000 not xx / 1000

Pages read ahead 0.00/s, evicted without access 0.00/s, Random read ahead 0.00/s

LRU len: xxxx, unzip_LRU len: xxx

I/O sum[xxx]:cur[xx], unzip sum[16xx:cur[0]

下面我们给大家解释一下这里的东西，主要讲解这里跟buffer pool相关的一些东西。

（1）Total memory allocated，这就是说buffer pool最终的总大小是多少

（2）Buffer pool size，这就是说buffer pool一共能容纳多少个缓存页

（3）Free buffers，这就是说free链表中一共有多少个空闲的缓存页是可用的

（4）Database pages和Old database pages，就是说lru链表中一共有多少个缓存页，以及冷数据区域里的缓存页数量

（5）Modified db pages，这就是flush链表中的缓存页数量

（6）Pending reads和Pending writes，等待从磁盘上加载进缓存页的数量，还有就是即将从lru链表中刷入磁盘的数量、即将从flush链表中刷入磁盘的数量

（7）Pages made young和not young，这就是说已经lru冷数据区域里访问之后转移到热数据区域的缓存页的数量，以及在lru冷数据区域里1s内被访问了没进入热数据区域的缓存页的数量

（8）youngs/s和not youngs/s，这就是说每秒从冷数据区域进入热数据区域的缓存页的数量，以及每秒在冷数据区域里被访问了但是不能进入热数据区域的缓存页的数量

（9）Pages read xxxx, created xxx, written xxx，xx reads/s, xx creates/s, 1xx writes/s，这里就是说已经读取、创建和写入了多少个缓存页，以及每秒钟读取、创建和写入的缓存页数量

（10）Buffer pool hit rate xxx / 1000，这就是说每1000次访问，有多少次是直接命中了buffer pool里的缓存的

（11）young-making rate xxx / 1000 not xx / 1000，每1000次访问，有多少次访问让缓存页从冷数据区域移动到了热数据区域，以及没移动的缓存页数量

（12）LRU len：这就是lru链表里的缓存页的数量

（13）I/O sum：最近50s读取磁盘页的总数

（14）I/O cur：现在正在读取磁盘页的数量

5、今日实践思考题

今天留给大家的作业，就是每个人都对自己线上在运行的数据库执行上述命令，然后分析一下数据库的buffer pool的使用情况

这里要尤为关注的是free、lru、flush几个链表的数量的情况，然后就是lru链表的冷热数据转移的情况，然后你的缓存页的读写情况，这些代表了你当前buffer pool的使用情况。

最关键的是两个东西，一个是你的buffer pool的千次访问缓存命中率，这个命中率越高，说明你大量的操作都是直接基于缓存来执行的，性能越高。

第二个是你的磁盘IO的情况，这个磁盘IO越多，说明你数据库性能越差。

大家可以去观察一下，把自己的分析和思考发布在评论区里一起交流

End

狸猫技术窝精品专栏及课程推荐：

《从零开始带你成为消息中间件实战高手》
《21天互联网Java进阶面试训练营》（分布式篇）
《互联网Java工程师面试突击》（第1季）
《互联网Java工程师面试突击》（第3季）
《从零开始带你成为JVM实战高手》

图文 24 我们写入数据库的一行数据，在磁盘上是怎么存储的？

手机观看

601 人次阅读 2020-02-18 07:00:00

详情 评论

我们写入数据库的一行数据，在磁盘上是怎么存储的？

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

1、承上启下：在Buffer Pool之后，为什么要学习MySQL物理数据模型？

之前的一些文章我们已经深入的给大家分析了当你执行crud操作的时候，MySQL是如何把磁盘上的数据页加载到内存中的Buffer Pool的缓存页里去的，以及对Buffer Pool是如何进行一整套复杂的管理机制的。

相信现在每个人都对缓存页加载到Buffer Pool中，更新和读取缓存页里的数据，这个过程中的free链表、flush链表以及lru链表的维护，都有了一个深刻的理解，包括后台线程是如何定时根据flush链表以及lru链表将部分被更新的缓存页刷入磁盘的，以及缓存页都用完了以后是如何根据lru链表将一些冷数据缓存页刷入磁盘的。

按理来说，在讲解完上述内容之后，我们下一步就应该是要给大家讲解undo log和redo log以及事务机制了，但是现在还不行，实际上我们在理解了MySQL中的数据缓存机制以及内存数据更新机制，包括缓存到磁盘的数据刷新机制之后，我们还得来理解一下MySQL中的物理数据结构。

之前很多朋友可能会发现我在文章里提到了表空间、区、数据页、一个区中的连续数据页、表空间号以及数据页号，这些概念，这些概念，我们之前即使不理解，其实也不妨碍我们去理解MySQL的Buffer Pool缓存机制。

因为大家之前可能脑子里大致都有一个概念，就是数据页这个概念，起码知道每一行数据都是放在数据页里的，我们是按照数据页为单位把磁盘上的数据加载到内存的缓存页里来，也是按照页为单位，把缓存页的数据刷入磁盘上的数据页中。

但是我之前也问过大家，我们平时写SQL语句的时候脑子里都有一个表、行和字段的概念，但是为什么跑到MySQL内部，就出现了一堆表空间、数据区、数据页这些概念呢？

其实很多人都说了，表、行和字段是逻辑上的概念，而表空间、数据区和数据页其实已经落实到物理上的概念了。

实际上表空间、数据页这些东西，都对应到了MySQL在磁盘上的一些物理文件了。

所以接下来，我们要用一些文章来逐步逐步的讲解MySQL的表空间、数据区、数据页、磁盘上的物理文件这些概念。当大家看明白这些东西之后，你就会理解，当我们执行SQL语句的时候，是从MySQL机器上的哪些磁盘文件里加载数据页到缓存页里来的，数据页是如何由数据区这个概念来组织起来的，表空间这个概念是怎么回事

很多朋友之前还在评论区提问，你一个SQL语句仅仅指定了你要查询或者更新哪个表的哪些数据，那你怎么知道这些数据在哪个表空间里？在哪个数据区里？在哪些数据页里？对应是在MySQL机器上的哪些磁盘文件里呢？

当我们学习完接下来的一部分内容后，上述问题的答案都会迎刃而解。

2、之前遗留思考题解答：为什么不能直接更新磁盘上的数据？

之前我们留过一个思考题，让大家思考一下，为什么MySQL要设计这么一套复杂的数据存取机制，要基于内存、日志、磁盘上的数据文件来完成数据的读写呢？为什么对insert、update请求，不直接更新磁盘文件里的数据呢？

很多人都在评论区给出了自己的思考和回答，我觉得每个人的思考都特别的好，大家可以多去评论区里看看别人的思考以及进行交流。

这里我简单一句话总结，为什么不能直接更新磁盘上的数据，因为来一个请求就直接对磁盘文件进行随机读写，然后更新磁盘文件里的数据，虽然技术上是可以做到的，但是那必然导致执行请求的性能极差。

因为磁盘随机读写的性能是最差的，所以直接更新磁盘文件，必然导致我们的数据库完全无法抗下任何一点点稍微高并发一点的场景。

所以MySQL才设计了如此复杂的一套机制，通过内存里更新数据，然后写redo log以及事务提交，后台线程不定时刷新内存里的数据到磁盘文件里

通过这种方式保证，你每个更新请求，尽量就是更新内存，然后顺序写日志文件。

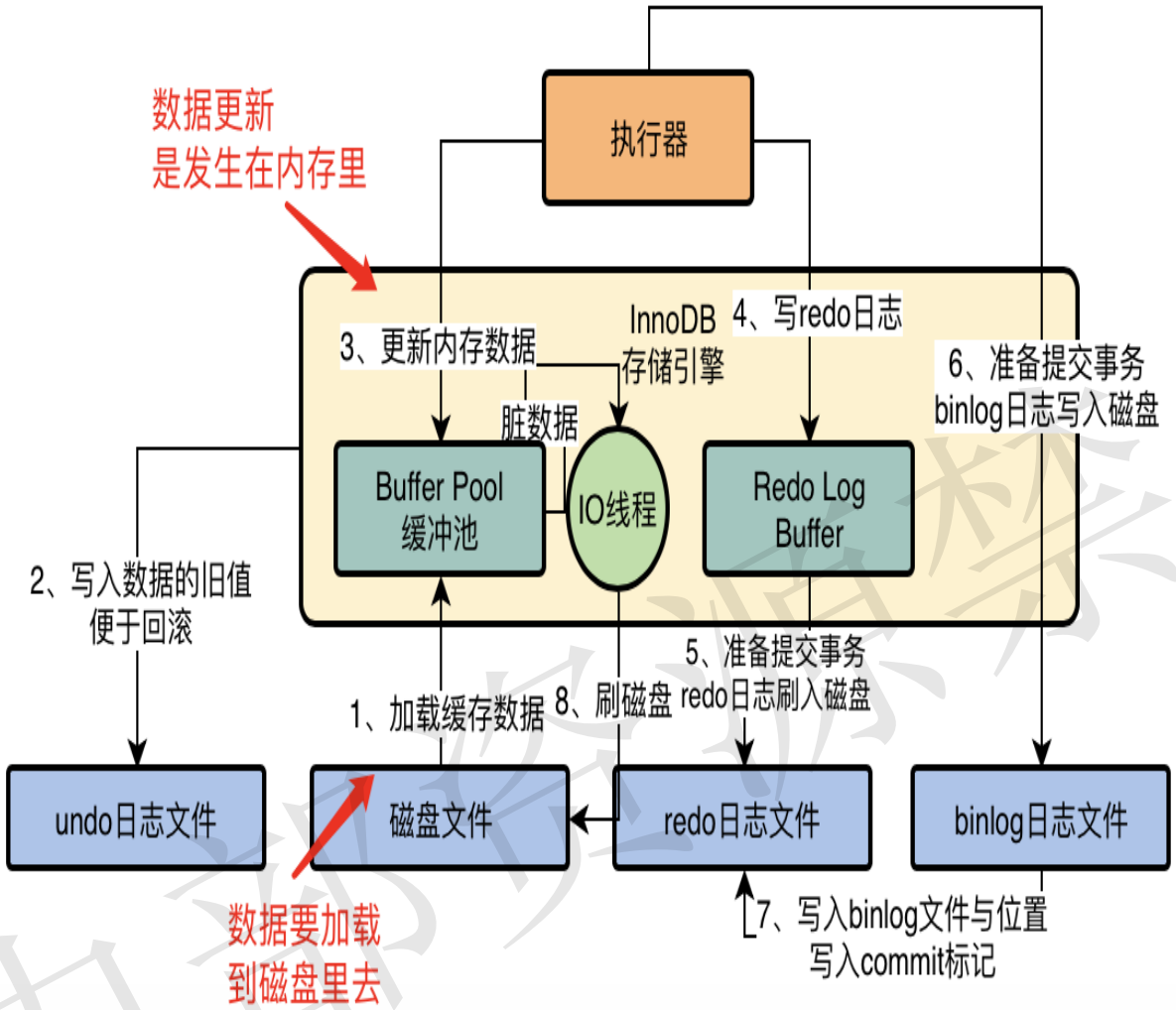
更新内存的性能是极高的，然后顺序写磁盘上的日志文件的性能也是比较高的，因为顺序写磁盘文件，他的性能要远高于随机读写磁盘文件。

也正是通过这套机制，才能让我们的MySQL数据库在较高配置的机器上，每秒可以抗下几千的读写请求。

3、复习巩固：MySQL为什么要引入数据页这个概念?

首先我们先考虑一下，刚才是不是已经给大家讲过，当我们要执行update之类的SQL语句的时候，必然涉及到对数据的更新操作？那么此时对数据是在哪里更新的?

此时并不是直接去更新磁盘文件，而是要把磁盘上的一些数据加载到内存里来，然后对内存里的数据进行更新，同时写redo log到磁盘上去，我们看下图回忆一下。

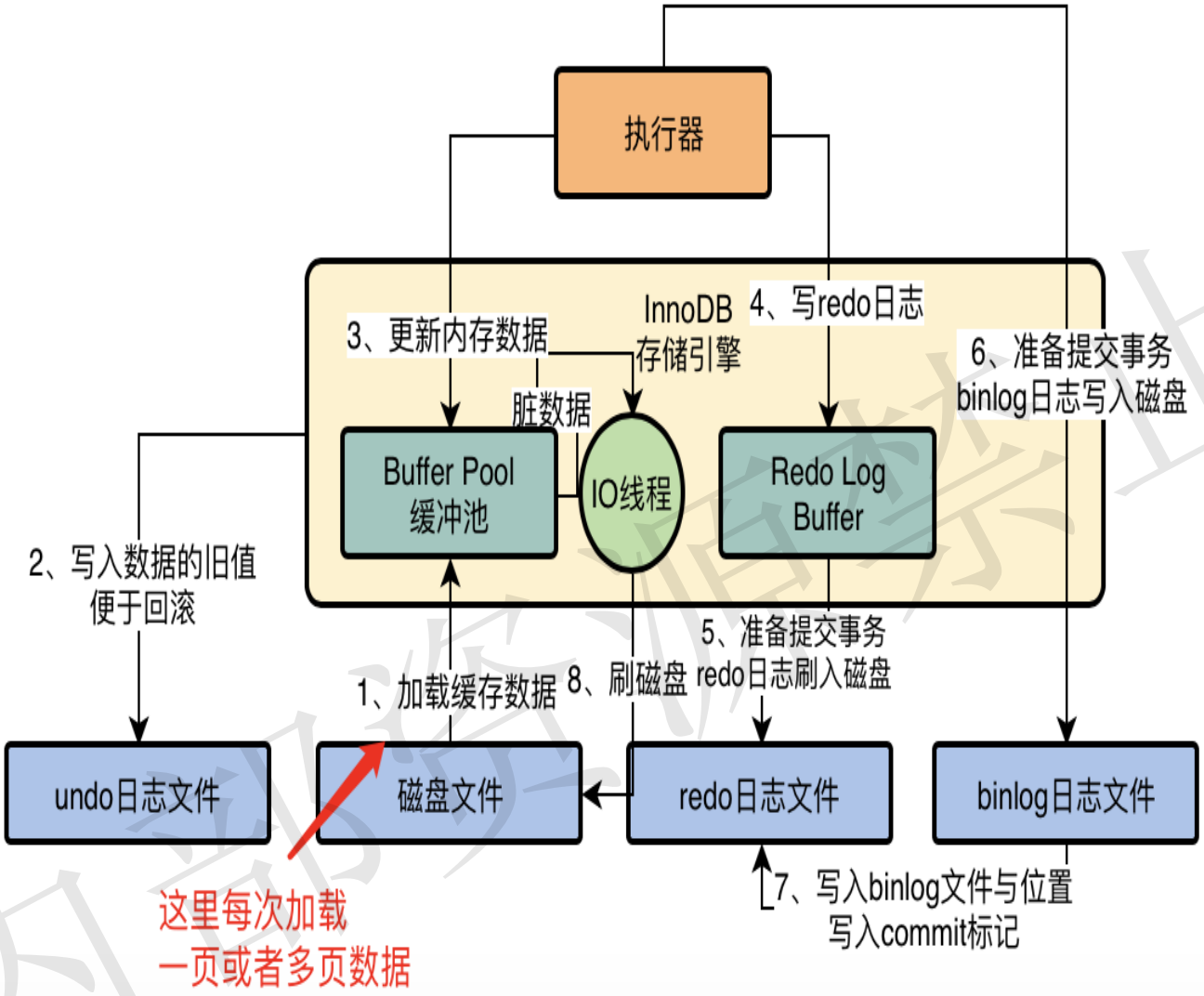


但是这里就有一个问题了，难道我们每次都是把磁盘里的一条数据加载到内存里去进行更新，然后下次要更新别的数据的时候，再从磁盘里加载另外一条数据到内存里去？

这样每次都是一条数据一条数据的加载到内存里去更新，大家觉得效率高吗？

很明显是不高的

所以innodb存储引擎在这里引入了一个**数据页**的概念，也就是把数据组织成一页一页的概念，每一页有16kb，然后每次加载磁盘的数据到内存里的时候，是至少加载一页数据进去，甚至是多页数据进去，我们看下图



假设我们有一次要更新一条id=1的数据：

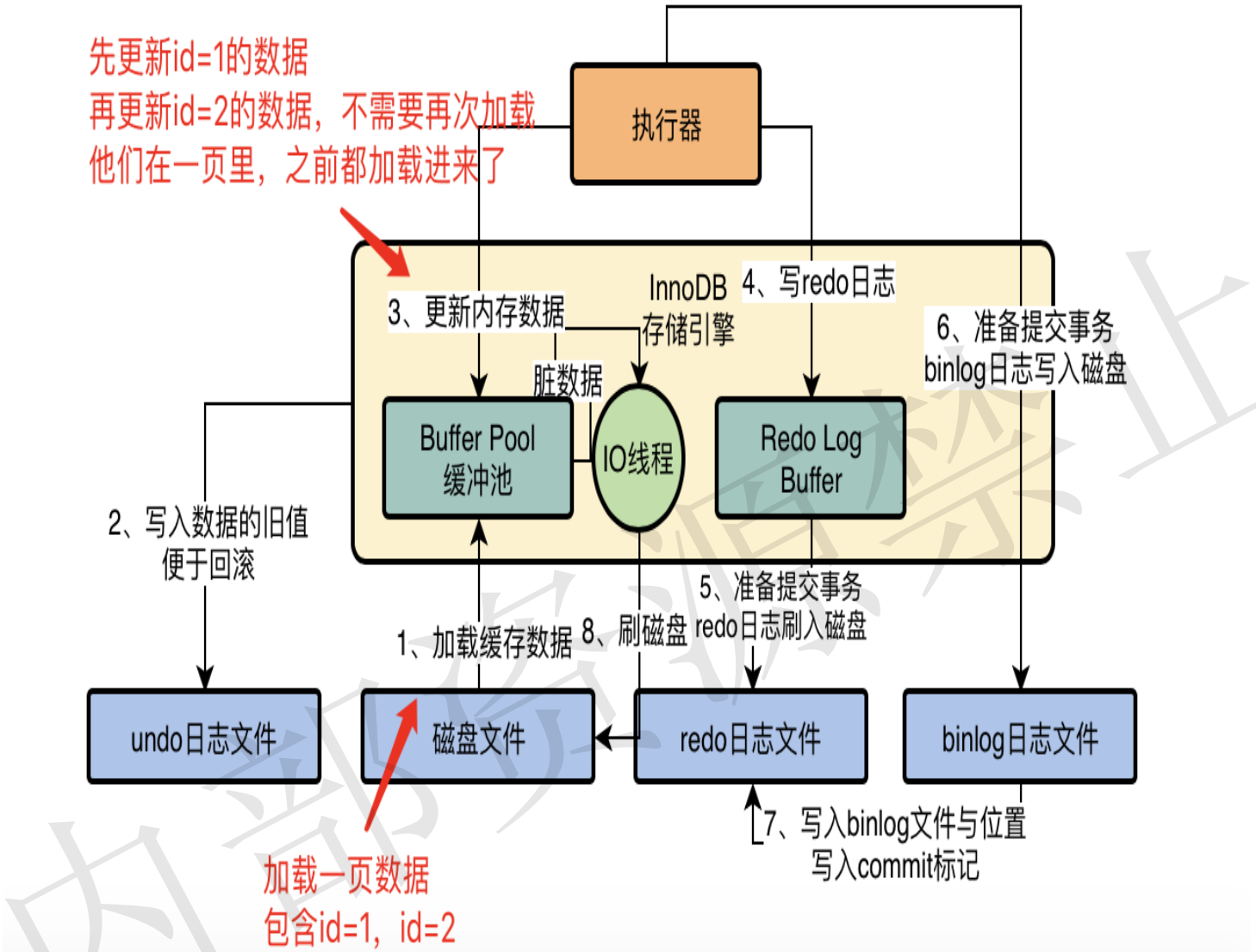
update xxx set xxx=xxx where id=1

那么此时他会把id=1这条数据所在的一页数据都加载到内存里去，这一页数据里，可能还包含了id=2，id=3等其他数据。

然后我们更新完id=1的数据之后，接着更新id=2的数据，那么此时是不是就不用再次读取磁盘里的数据了？

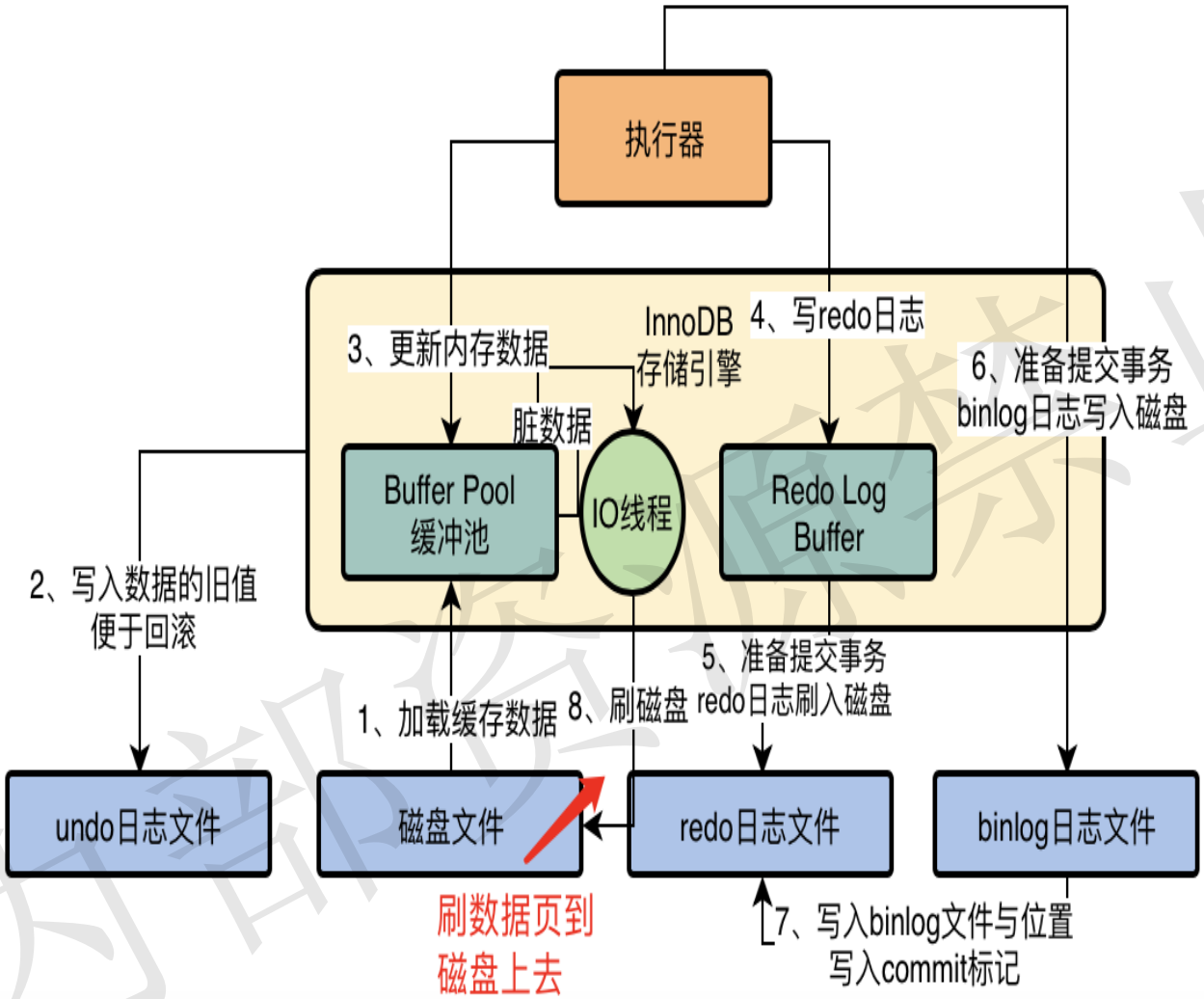
因为id=2本身就跟id=1在一页里，之前这一页数据就加载到内存里去了，你直接更新内存里的数据页中的id=2这条数据就可以了。

我们看下图，这就是数据页的意义，磁盘和内存之间的数据交换通过数据页来执行，包括内存里更新后的脏数据，刷回磁盘的时候，也是至少一个数据页刷回去。



我们再看下图，要明白的一点是，我们不是一直在内存里更新各种数据吗？当IO线程把内存里的脏数据刷到磁盘上去的时候，也是以数据页为单位来刷回去的

下图中有这个刷数据的图示：



4、初涉MySQL物理数据存储格式：一行数据在磁盘上是如何存储的？

那么接着我们可以来思考一下，对数据页中的每一行数据，他在磁盘上是怎么存储的？

其实这里涉及到一个概念，就是行格式。我们可以对一个表指定他的行存储的格式是什么样的，比如我们这里用一个COMPACT格式。

CREATE TABLE table_name (columns) ROW_FORMAT=COMPACT

ALTER TABLE table_name ROW_FORMAT=COMPACT

你可以在建表的时候，就指定一个行存储的格式，也可以后续修改行存储的格式。这里指定了一个COMPACT行存储格式，在这种格式下，每一行数据他实际存储的时候，大概格式类似下面这样：

变长字段的长度列表，null值列表，数据头，column01的值，column02的值，column0n的值.....

对于每一行数据，他其实存储的时候都会有一些头字段对这行数据进行一定的描述，然后再放上他这一行数据每一列的具体的值，这就是所谓的行格式。除了COMPACT以外，还有其他几种行存储格式，基本都大同小异。

大家可能对一行数据实际存储的时候，他里面的一些东西到底都是什么含义，感觉都很好奇，大可不必着急，我们明天的文章会继续讲解的。

5、今天学习的要点总结

今天我们主要是做了一些承上启下的复习和巩固，告诉了大家innodb存储引擎在存储数据的时候，是通过数据页的方式来组织数据的

然后我们初步的开始尝试切入了MySQL的物理数据存储格式，讲解了对于数据页中的每一行数据，其实他都有对应的行格式

接下来，我们将会深入探索这个每一行数据到底是怎么存储的。

End

狸猫技术窝精品专栏及课程推荐：

- 《从零开始带你成为消息中间件实战高手》
- 《21天互联网Java进阶面试训练营》（分布式篇）
- 《互联网Java工程师面试突击》（第1季）
- 《互联网Java工程师面试突击》（第3季）
- 《从零开始带你成为JVM实战高手》

图文 25 对于VARCHAR这种变长字段，在磁盘上到底是如何存储的？

手机观看

529 人次阅读 2020-02-19 07:00:00

详情 评论

对于VARCHAR这种变长字段，在磁盘上到底是如何存储的？

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

1、一行数据在磁盘上存储的时候，包含哪些东西？

上一讲我们已经告诉了大家，一行数据在磁盘上存储的时候，其实不仅仅是包含我们想象的那一点数据，他还包含了很多其他的信息，之前告诉大家，一行数据的存储格式大致如下所示。

变长字段的长度列表，null值列表，数据头，column01的值，column02的值，column0n的值.....

说白了，就是除了每一个字段的值以外，他还包含了一些额外的信息，这些额外的信息就是用来描述这一行数据的。今天我们就详细给大家说说这些额外的信息里都是放了什么东西。

2、变长字段在磁盘中是怎么存储的?

大家都知道，在MySQL里有一些字段的长度是变长的，是不固定的，比如VARCHAR(10)之类的这种类型的字段，实际上他里面存放的字符串的长度是不固定的，有可能是“hello”这么一个字符串，也可能是“a”这么一个字符串。

好，那么现在我们来假设一下，现在有一行数据，他的几个字段的类型为VRACHAR(10)，CHAR(1)，CHAR(1)，那么他第一个字段是VARCHAR(10)，这个长度是可能变化的，所以这一行数据可能就是类似于：hello a a，这样子，第一个字段的值是“hello”，后面两个字段的值都是一个字符，就是一个a

然后另外一行数据，同样也是这几个字段，他的第一个字段的值可能是“hi”，后面两个字段也是“a”，所以这一行数据可能是类似于：hi a a。一共三个字段，第一个字段的长度是是不固定的，后面两个字段的长度都是固定的1个字符。

想必这个道理大家都能理解吧?

那么现在，我们来假设你把上述两条数据写入了一个磁盘文件里，两行数据是挨在一起的，那么这个时候在一个磁盘文件里可能有下面的两行数据:

hello a a hi a a

大家可以看到，两行数据在底层磁盘文件里是不是挨着存储的?

没错！其实平时你看到的表里的很多行数据，最终落地到磁盘里的时候，都是上面那种样子的，一大坨数据放在一个磁盘文件里都挨着存储的。

3、存储在磁盘文件里的变长字段，为什么难以读取?

现在我们来继续思考一个问题，假设现在我们要读取上面的磁盘文件里的数据，要读取出来hello a a这一行数据。那你觉得是那么容易的吗?

当然不是了！这个过程比你想象的可能要困难一些。

假如现在你要读取hello a a这行数据，第一个问题就是，从这个磁盘文件里读取的时候，到底哪些内容是一行数据？我不知道啊！

因为这个表里的第一个字段是VARCHAR(10)类型的，第一个字段的长度是多少我们是不知道的！

所以有可能你读取出来“hello a a hi”是一行数据，也可能是你读取出来“hello a”是一行数据，你在不知道一行数据的每个字段到底是多少长度的情况下，胡乱的去读取是不现实的，根本不知道磁盘文件里混成一坨的数据里，哪些数据是你要读取的一行？

4、引入变长字段的长度列表，解决一行数据的读取问题

所以说才要在存储每一行数据的时候，都保存一下他的变长字段的长度列表，这样才能解决一行数据的读取问题。

也就是说，你在存储“hello a a”这行数据的时候，要带上一些额外的附加信息，比如第一块就是他里面的变长字段的长度列表

也就是说，这个hello是VARCHAR(10)类型的变长字段的值，那么这个“hello”字段值的长度到底是多少？

我们看到“hello”的长度是5，十六进制就是0x05，所以此时会在“hello a a”前面补充一些额外信息，首先就是变长字段的长度列表，你会看到这行数据在磁盘文件里存储的时候，其实是类似如下的格式：0x05 null值列表 数据头 hello a a。

你这行数据存储的时候应该是如上所示的！

这个时候假设你有两行数据，还有一行数据可能就是：0x02 null值列表 数据头 hi a a，两行数据放在一起存储在磁盘文件里，看起来是如下所示的：

0x05 null值列表 数据头 hello a a 0x02 null值列表 数据头 hi a a

5、引入变长字段长度列表后，如何解决变长字段的读取问题？

所以假设此时你要读取“hello a a”这行数据，你首先会知道这个表里的三个字段的类型是VARCHAR(10) CHAR(1) CHAR(1)，那么此时你先要读取第一个字段的值，那么第一个字段是变长的，到底他的实际长度是多少呢?

此时你会发现第一行数据的开头有一个变长字段的长度列表，里面会读取到一个0x05这个十六进制的数字，发现第一个变长字段的长度是5，于是按照长度为5，读取出来第一个字段的值，就是“hello”

接着你知道后续两个字段都是CHAR(1)，长度都是固定的1个字符，于是此时就依次按照长度为1读取出来后续两个字段的值，分别是“a” “a”，于是最终你会读取出来“hello a a”这一行数据!

接着假设你要读取第二行数据，你先看一下第二行数据后的变长字段长度列表，发现他第一个变长字段的长度是0x02，于是就读取长度为2的字段值，就是“hi”，再读取两个长度固定为1的字符值，都是“a”，此时读取出来“hi a a”这行数据。

6、如果有多个变长字段，如何存放他们的长度?

接着我们假设，如果说有多个变长字段，如何存放他们的长度?

比如一行数据有VARCHAR(10) VARCHAR(5) VARCHAR(20) CHAR(1) CHAR(1)，一共5个字段，其中三个是变长字段，此时假设一行数据是这样的：hello hi hao a a

此时在磁盘中存储的，必须在他开头的变长字段长度列表中存储几个变长字段的长度，一定要注意一点，他这里是逆序存储的!

也就是说先存放VARCHAR(20)这个字段的长度，然后存放VARCHAR(5)这个字段的长度，最后存放VARCHAR(10)这个字段的长度。

现在hello hi hao三个字段的长度分别是0x05 0x02 0x03，但是实际存放在变长字段长度列表的时候，是逆序放的，所以一行数据实际存储可能是下面这样的：

0x03 0x02 0x05 null值列表 头字段 hello hi hao a a

7、今日思考题

今天让大家思考一个问题，为什么MySQL在把一行一行的数据存储在磁盘上的时候，要采取这种“0x05 null值列表 数据头 hello a a 0x02 null值列表 数据头 hi a a”很多行数据都仅仅挨在一起的方式？

为什么MySQL不能用Java里面的序列化的那种方式？把很多行的数据做成一个大的对象，然后给他序列化一下写入到磁盘文件里，从磁盘里读取的时候压根儿不用care什么行存储格式，直接反序列化一下，把数据就可以从磁盘文件里拿回来了。

请大家思考一下，MySQL用这种数据紧凑挨在一起的方式来存储数据，到底有什么好处？可以在评论区里发表你的想法跟大家一起交流。

End

狸猫技术窝精品专栏及课程推荐：

- 《从零开始带你成为消息中间件实战高手》
- 《21天互联网Java进阶面试训练营》（分布式篇）
- 《互联网Java工程师面试突击》（第1季）
- 《互联网Java工程师面试突击》（第3季）
- 《从零开始带你成为JVM实战高手》

图文 26 一行数据中的多个NULL字段值在磁盘上怎么存储?

手机观看

详情 评论

一行数据中的多个NULL字段值在磁盘上怎么存储?

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

1、为什么一行数据里的NULL值不能直接存储?

之前我们已经给大家讲了在数据库里一行数据中如果有VARCHAR(10)之类的变长字段，那么他的存储和读取会有什么问题，以及为了解决这个问题，为什么要给磁盘上存储的每一行数据都加入变长字段长度列表。

今天我们继续给大家讲解在磁盘上存储的一行数据里另外一块特殊的数据区域，就是NULL值列表。

这个所谓的NULL值列表，顾名思义，说的就是你一行数据里可能有的字段值是NULL，比如你有一个name字段，他是允许为NULL的，那么实际上在存储的时候，如果你没给他赋值，他这个字段的值就是NULL。

好，那么假设这个字段的NULL值我们在磁盘上存储的时候，就是按照“NULL”这么个字符串来存储，是不是很浪费存储空间?

本来他就是个NULL，说明什么值都没有，你还给他存个“NULL”字符串，你说你这是干什么呢?

所以实际在磁盘上存储数据的时候，一行数据里的NULL值是肯定不会直接按照字符串的方式存放在磁盘上浪费空间的。

2、NULL值是以二进制bit位来存储的

我们接着看，那么NULL值列表在磁盘上到底应该如何存储呢?

很简单，对所有的NULL值，不通过字符串在磁盘上存储，而是通过二进制的bit位来存储，一行数据里假设有多个字段的值都是NULL，那么这多个字段的NULL，就会以bit位的形式存放在NULL值列表中。

现在我们来给大家举个例子，假设你有一个表的额，他的建表语句如下所示:

```
CREATE TABLE customer (
    name VARCHAR(10) NOT NULL,
    address VARCHAR(20),
    gender CHAR(1),
    job VARCHAR(30),
    school VARCHAR(50)
) ROW_FORMAT=COMPACT;
```

上面那个表就是一个假想出来的客户表，里面有5个字段，分别为name、address、genderjob、school，就代表了客户的姓名、地址、性别、工作以及学习小。

其中有4个变长字段，还有一个定长字段，然后第一个name字段是声明了NOT NULL的，就是不能为NULL，其他4个字段都可能是NULL的。

那么现在我们来假设这个表里有如下一行数据，现在来看看，他在磁盘上是怎么来存储的：“jack NULL m NULL xx_school”，他的5个字段里有两个字段都是NULL

3、结合小小案例来思考一行数据的磁盘存储格式

接着我们来思考上面那个表里的那行案例数据，在磁盘上应该如何存储呢，因为他有多个变长字段，还有多个字段允许为NULL。首先我们先回顾一下，一行数据在磁盘上的存储格式应该是下面这样的：

变长字段长度列表 NULL值列表 头信息 column1=value1 column2=value2 ... columnN=valueN

所以先看变长字段长度列表应该放什么东西，他一共有4个变长字段，那么按照我们上次说的，是不是应该按照逆序的顺序，先放school字段的长度，再放job、address、name几个字段的值长度？

说起来是这样，但是其实这里要区分一个问题，那就是如果这个变长字段的值是NULL，就不用在变长字段长度列表里存放他的值长度了，所以在上面那行数据中，只有name和school两个变长字段是有值的，把他们的长度按照逆序放在变长字段长度列表中就可以了，如下所示：

0x09 0x04 NULL值列表 头信息 column1=value1 column2=value2 ... columnN=valueN

接着来看NULL值列表，这个NULL值列表是这样存放的，你所有允许值为NULL的字段，注意，是允许值为NULL，不是说一定值就是NULL了，只要是允许你为NULL的字段，在这里每个字段都有一个二进制bit位的值，如果bit值是1说明是NULL，如果bit值是0说明不是NULL。

比如上面4个字段都允许为NULL，每个人都会有一个bit位，这一行数据的值是“jack NULL m NULL xx_school”，然后其中2个字段是null，2个字段不是null，所以4个bit位应该是：1010

但是实际放在NULL值列表的时候，他是按逆序放的，所以在NULL值列表里，放的是：0101，整体这一行数据看着是下面这样的

0x09 0x04 0101 头信息 column1=value1 column2=value2 ... columnN=valueN

另外就是他实际NULL值列表存放的时候，不会说仅仅是4个bit位，他一般起码是8个bit位的倍数，如果不足8个bit位就高位补0，所以实际存放看起来是如下的：

0x09 0x04 00000101 头信息 column1=value1 column2=value2 ... columnN=valueN

4、磁盘上的一行数据到底如何读取出来的?

我们结合上面的磁盘上的数据存储格式来思考一下，一行数据到底是如何读取出来的呢?

再看上面的磁盘数据存储格式：

0x09 0x04 00000101 头信息 column1=value1 column2=value2 ... columnN=valueN

首先他必然要把变长字段长度列表和NULL值列表读取出来，通过综合分析一下，就知道有几个变长字段，哪几个变长字段是NULL，因为NULL值列表里谁是NULL谁不是NULL都一清二楚。

此时就可以从变长字段长度列表中解析出来不为NULL的变长字段的值长度，然后也知道哪几个字段是NULL的，此时根据这些信息，就可以从实际的列值存储区域里，把你每个字段的值读取出来了。

如果是变长字段的值，就按照他的值长度来读取，如果是NULL，就知道他是个NULL，没有值存储，如果是定长字段，就按照定长长度来读取，这样就可以完美的把你一行数据的值都读取出来了!

5、今日思考题

昨天让大家思考，为什么数据要按照这种紧凑的格式来在磁盘里存储，今天我们不公布答案，再次问大家一个问题，为什么NULL值列表要按照二进制bit位的方式来存储?

他跟直接用NULL字符串的方式来存储，会有多少存储空间的差距呢?

希望大家思考一下这个问题，然后在评论区发表你的思考，和大家一起交流，答案我们后面会公布。

End

狸猫技术窝精品专栏及课程推荐：

- 《从零开始带你成为消息中间件实战高手》
- 《21天互联网Java进阶面试训练营》（分布式篇）
- 《互联网Java工程师面试突击》（第1季）
- 《互联网Java工程师面试突击》（第3季）
- 《从零开始带你成为JVM实战高手》

图文 27 磁盘文件中， 40个bit位的数据头以及真实数据是如何存储的?

手机观看

371 人次阅读 2020-02-21 07:00:00

详情 评论

磁盘文件中， 40个bit位的数据头以及真实数据是如何存储的?

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

之前我们已经给大家讲到了在磁盘上存储数据的时候，每一行数据都会有变长字段长度列表，逆序存放这行数据里的变长字段的长度，然后会有NULL值列表，对于允许为NULL的字段都会有一个bit位标识那个字段是否为NULL，也是逆序排列的。

今天我们接着给大家讲每一行数据存储的时候，还得有40个bit位的数据头，这个数据头是用来描述这行数据的。

这40个bit位里，第一个bit位和第二个bit位，都是预留位，是没任何含义的。

然后接下来有一个bit位是**delete_mask**，他标识的是这行数据是否被删除了，其实看到这个bit位，很多人可能已经反映过来了，这么说在MySQL里删除一行数据的时候，未必是立马把他从磁盘上清理掉，而是给他在数据头里搞1个bit

标记他已经被删了？

没错，其实大家现在看这些数据头，只要先留有一个印象就可以了，知道每一行数据都有一些数据头，不同的数据头都是用来描述这行数据的一些状态和附加信息的。

然后下一个bit位是**min_rec_mask**，这个bit位大家现在先不用去关注，他的含义以后我们讲到对应的内容的时候再说，他其实就是说在B+树里每一层的非叶子节点里的最小值都有这个标记。

接下来有4个bit位是**n_owned**，这个暂时我们也先不用去管他，他其实就是记录了一个记录数，这个记录数的作用，后续我们讲到对应的概念时会告诉大家的。

接着有13个bit位是**heap_no**，他代表的是当前这行数据在记录堆里的位置，现在大家可能也很难去理解他，这些概念都要结合后续的一些内容才能理解的，这里只能是初步的给大家介绍下。

然后是3个bit位的record_type，这就是说这行数据的类型

0代表的是普通类型，1代表的是B+树非叶子节点，2代表的是最小值数据，3代表的是最大值数据

很多朋友可能也不理解这些什么意思，其实我们也现在不用在乎他，因为很多这些概念都是往后在讲解索引之类的技术的时候才会涉及到的。

最后是16个bit的next_record，这个是指向他下一条数据的指针。

今天是周五，我们就讲这些，大家利用周末复习一下我们本周讲的这些内容，务必消化吸收，下周咱们继续。

End

狸猫技术窝精品专栏及课程推荐：

《从零开始带你成为消息中间件实战高手》
《21天互联网Java进阶面试训练营》（分布式篇）
《互联网Java工程师面试突击》（第1季）
《互联网Java工程师面试突击》（第3季）
《从零开始带你成为JVM实战高手》

图文 28 我们每一行的实际数据在磁盘上是如何存储的?

手机观看

详情 评论

我们每一行的实际数据在磁盘上是如何存储的?

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

之前我们已经给大家讲过了，一行数据在磁盘文件里存储的时候，实际上首先会包含自己的变长字段的长度列表，然后是NULL值列表，接着是数据头，然后接着才是真实数据，所以这一次我们就讲讲真实数据是如何存储的。

首先我们在存储真实数据的时候，并没什么特别的，无非就是按照我们那个字段里的数据值去存储就行了

比如我们之前说了一个例子，有一行数据是“jack NULL m NULL xx_school”，那么他真实存储大致如下所示：

0x09 0x04 00000101 00000000000000000000010000000000000011001 jack m xx_school

刚开始先是他的变长字段的长度，用十六进制来存储，然后是NULL值列表，指出了谁是NULL，接着是40个bit位的数据头，然后是真实的数据值，就放在后面。

在读取这个数据的时候，他会根据变长字段的长度，先读取出来jack这个值，因为他的长度是4，就读取4个长度的数据，jack就出来了；

然后发现第二个字段是NULL，就不用读取了；

第三个字段是定长字段，直接读取1个字符就可以了，就是m这个值；

第四个字段是NULL，不用读取了；

第五个字段是变长字段长度是9，读取出来xx_school就可以了。

但是等等，大家觉得真正在磁盘上存储的时候，我们那些字符串就是直接这么存储在磁盘上吗？

显然不是的！

实际上字符串这些东西都是根据我们数据库指定的字符集编码，进行编码之后再存储的，所以大致看起来一行数据是如下所示的：

0x09 0x04 00000101 00000000000000000000010000000000000011001 616161 636320 6262626262

大家会看到上面，我们的字符串和其他类型的数值最终都会根据字符集编码，搞成一些数字和符号存储在磁盘上

所以其实一行数据是如何存储的，我相信大家就都已经了解的很清晰了，那么我们今天来给大家简单提一下，在实际存储一行数据的时候，会在他的真实数据部分，加入一些隐藏字段，这个隐藏字段跟后续的一些内容是有关联的，大家先了解一下。

首先有一个DB_ROW_ID字段，这就是一个行的唯一标识，是他数据库内部给你搞的一个标识，不是你的主键ID字段。如果我们没有指定主键和unique key唯一索引的时候，他就内部自动加一个ROW_ID作为主键。

接着是一个DB_TRX_ID字段，这是跟事务相关的，他是说这是哪个事务更新的数据，这是事务ID，这个后续我们讲解到事务的时候会跟大家说的。

最后是DB_ROLL_PTR字段，这是回滚指针，是用来进行事务回滚的，也是我们后续在讲解事务的时候再详细说。

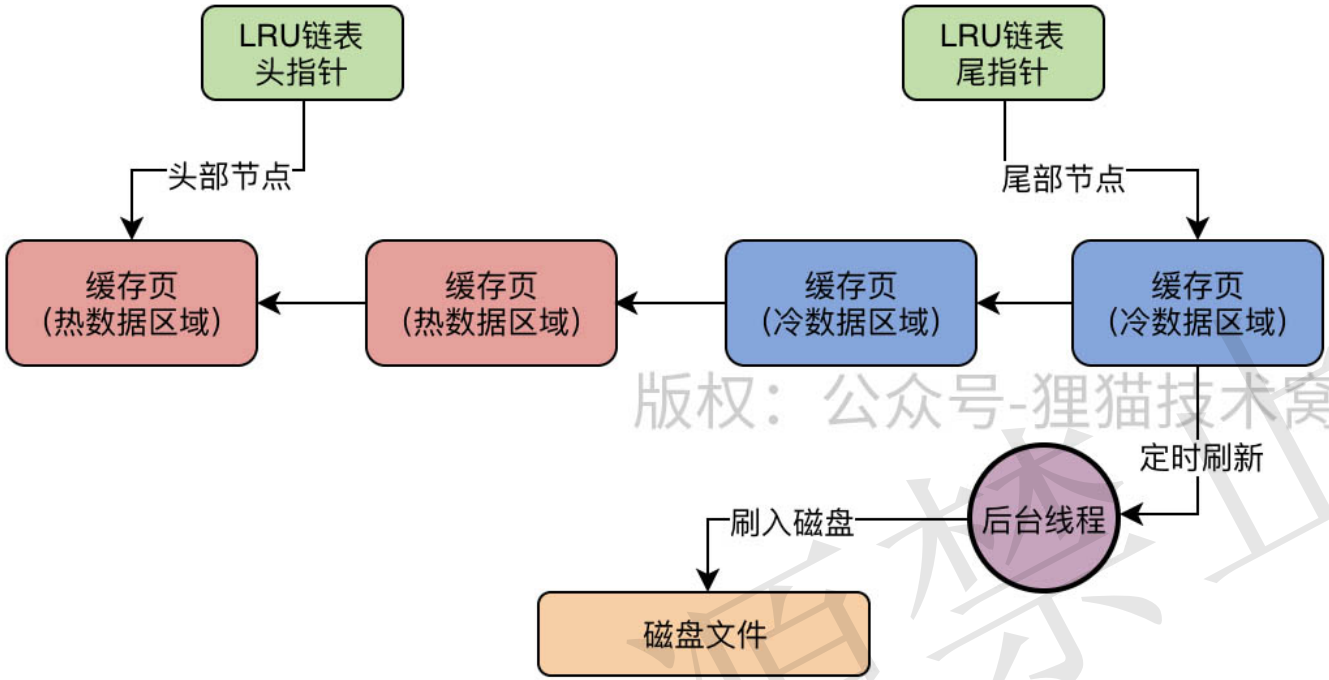
所以如果你加上这几个隐藏字段之后，实际一行数据可能看起来如下所示：

0x09 0x04 00000101 00000000000000000000100000000000000011001 00000000094C（DB_ROW_ID）
00000000032D（DB_TRX_ID） EA000010078E（DB_ROL_PTR） 616161 636320 6262626262

我给上面几个隐藏字段都加了括号说明了，上面那基本就是最终在磁盘上一行数据是长成什么样的了

我们再看看下面的图，大家回忆一下之前我们给大家讲解的，当你执行crud的时候，先会把磁盘上的数据加载到Buffer Pool里缓存，然后更新的时候也是更新Buffer Pool的缓存，同时维护一堆链表。

然后定时或者不定时的，根据flush链表和lru链表，Buffer Pool里的更新过的脏数据就会刷新到磁盘上去。



好，现在我们再结合最近讲解的一些内容思考一下，那么在磁盘上的数据，每一行数据是不是就是类似“0x09 0x04 00000101 00000000000000000000010000000000000011001 00000000094C（DB_ROW_ID）00000000032D（DB_TRX_ID） EA000010078E（DB_ROL_PTR） 616161 636320 6262626262”这样的东西?

所以现在我们就初步的把磁盘上的数据和内存里的数据给关联起来了，他每一行数据的真实存储结构我们就了解了，希望大家能在头脑里屡清楚这个他们之间的关系，其实这些都是有机一体的。

End

狸猫技术窝精品专栏及课程推荐：

《从零开始带你成为消息中间件实战高手》
《21天互联网Java进阶面试训练营》（分布式篇）
《互联网Java工程师面试突击》（第1季）
《互联网Java工程师面试突击》（第3季）
《从零开始带你成为JVM实战高手》

图文 29 理解数据在磁盘上的物理存储之后，聊聊行溢出是什么东西？

手机观看

271 人次阅读 2020-02-25 07:00:00

详情 评论

理解数据在磁盘上的物理存储之后，聊聊行溢出是什么东西?

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

上一篇文章我们已经理解清楚了一行数据在磁盘上的物理存储结构了，其实理解了这个，你也就理解了每一行数据在磁盘上是如何存储的，以及他被加载到缓存里来的时候，一行数据都包含哪些东西了。

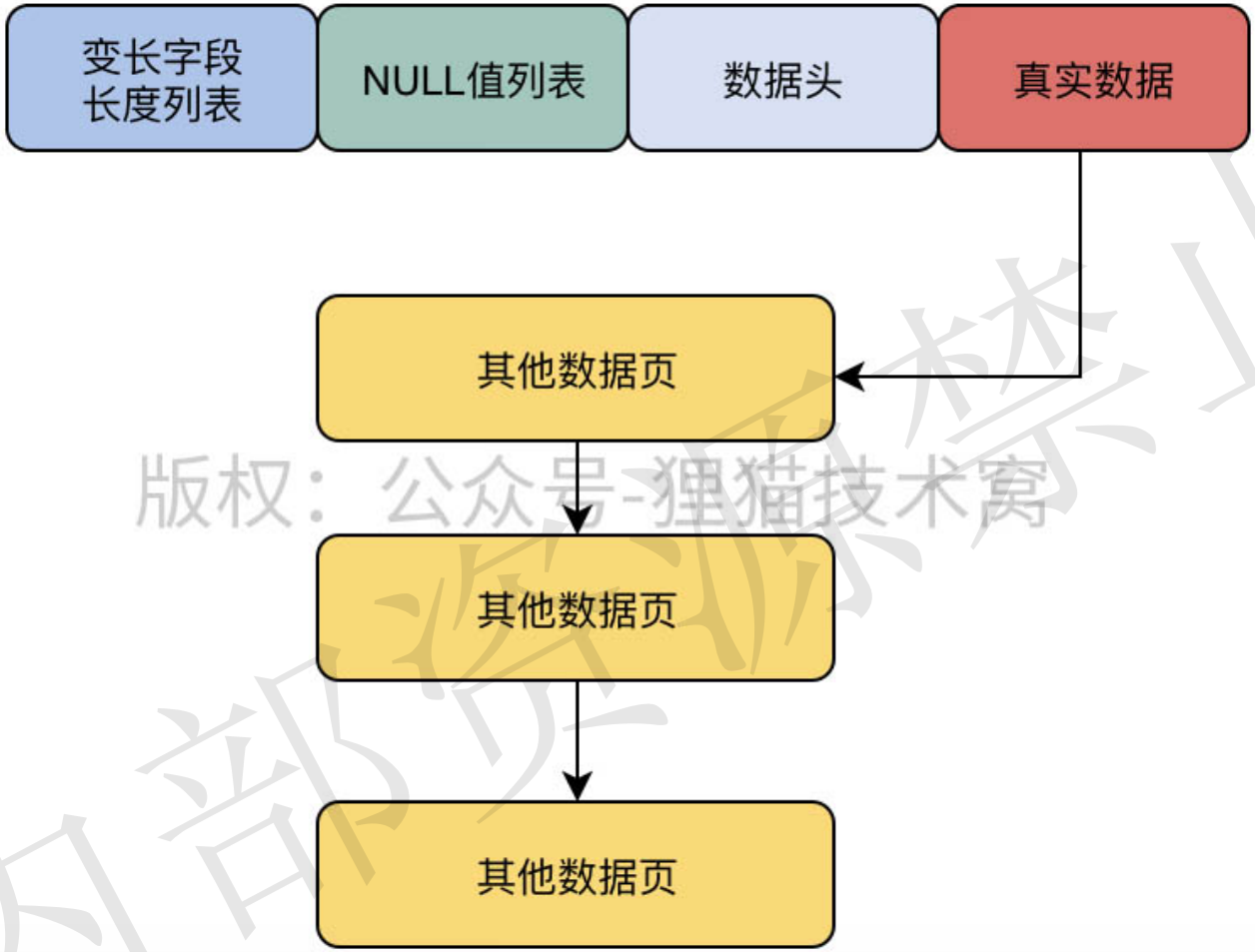
今天我们来聊聊行数据的物理存储的一个高阶的话题，就是行溢出到底是个什么东西?

我们之前已经初步了解到，实际上我们每一行数据都是放在一个数据页里的，这个数据页默认的大小是16KB，那么之前就有人在后台提过一个问题：万一 一行数据的大小超过了页的大小怎么办呢?

比如有一个表的字段类型是VARCHAR(65532)，意思就是最大可以包含65532个字符，那也就是65532个字节，这就远大于16kb的大小了，也就是说这一行数据的这个字段都远超一个数据页的大小了!

这个时候实际上会在那一页里存储你这行数据，然后在那个字段中，仅仅包含他一部分数据，同时包含一个20个字节的指针，指向了其他的一些数据页，那些数据页用链表串联起来，存放这个VARCHAR(65532)超大字段里的数据。

我们看下图，就给出了这个示意。



上面说的这个过程，其实就叫做**行溢出**，就是说一行数据存储的内容太多了，一个数据页都放不下了，此时只能溢出这个数据页，把数据溢出存放到其他数据页里去，那些数据页就叫做溢出页。

包括其他的一些字段类型都是一样的，比如TEXT、BLOB这种类型的字段，都有可能出现溢出，然后一行数据就会存储在多个数据页里。

讲到这里，其实就已经把我们的行数据的物理存储相关的内容都已经讲完了，很多琐碎和细节的东西，其实不需要我们在这里来死扣他，大家其实要理解的，就是一行数据的物理存储结构，然后这个数据其实是在一个数据页里的，如果一个数据页里放不下一行数据，就会有行溢出问题，存放到多个数据页里去。

讲到这里，我们可以做一点总结，当我们在数据库里插入一行数据的时候，实际上是在内存里插入一个有复杂存储结构的一行数据，然后随着一些条件的发生，这行数据会被刷到磁盘文件里去。

在磁盘文件里存储的时候，这行数据也是按照复杂的存储结构去存放的。

而且每一行数据都是放在数据页里的，如果一行数据太大了，就会产生行溢出问题，导致一行数据溢出到多个数据页里去，那么这行数据在Buffer Pool可能就是存在于多个缓存页里的，刷入到磁盘的时候，也是用磁盘上的多个数据页来存放这行数据的。

希望大家能够把最近几天学到的行数据物理存储结构，与之前学到的Buffer Pool缓存机制结合起来去理解，把他们有机的融合为一体。

接下来，我们就会开始讲解数据页的物理存储结构，然后是表空间的物理存储结构，最后是讲解这些数据以物理存储结构的方式，在磁盘上存储的时候，是放在哪些磁盘文件里的。

只要把后续那些内容讲完，那么大家就对数据库的Buffer Pool缓冲读写机制，以及磁盘上的物理存储机制，就完全理解了，而且这两个机制都是有机结合在一起的，Buffer Pool的数据是从磁盘上读取出来的，Buffer Pool里更新的数据又会刷新到磁盘上去。

在这个过程中，整个数据的物理存储机制，包括行数据、数据页、表空间、磁盘文件，这些概念，大家也都会理解了，到时候自然理解了数据在磁盘上如何存储的，加载到Buffer Pool缓存页之后如何存储的。

End

狸猫技术窝精品专栏及课程推荐：

- 《从零开始带你成为消息中间件实战高手》
- 《21天互联网Java进阶面试训练营》（分布式篇）
- 《互联网Java工程师面试突击》（第1季）
- 《互联网Java工程师面试突击》（第3季）
- 《从零开始带你成为JVM实战高手》

图文 30 用于存放磁盘上的多行数据的数据页到底长个什么样子?

手机观看

278 人次阅读 2020-02-26 07:00:00

详情 评论

用于存放磁盘上的多行数据的数据页到底长个什么样子?

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

之前我们老是给大家提到一个概念，就是数据页，大家都知道平时我们执行crud的时候，都会从磁盘上加载数据页到Buffer Pool的缓存页里去，然后更新了缓存页后，又会刷新回磁盘上的数据页里去。

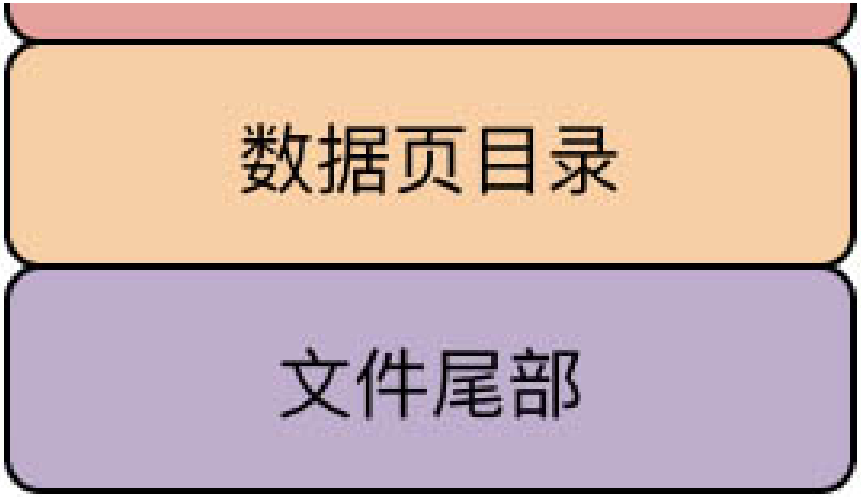
所以其实MySQL中进行数据操作的最小单位应该是数据页，那么我们之前已经给大家分析过了一行一行的数据在磁盘和缓存中存储的时候，他真正的格式是什么样子的

现在我们都知道，一行一行的数据是放在数据页里的，所以接下来就该分析分析，数据页到底是长什么样子的了。

之前介绍过，每个数据页，实际上是默认有16kb的大小，那么这16kb的大小就是存放大量的数据行吗?

明显不是的，其实一个数据页拆分成了很多个部分，大体上来说包含了文件头、数据页头、最小记录和最大记录、多个数据行、空闲空间、数据页目录、文件尾部。

我下面有一个图，在图里包含了一个数据页的各个部分，大家可以看一下



其中文件头占据了38个字节，数据页头占据了56个字节，最大记录和最小记录占据了26个字节，数据行区域的大小是不固定的，空闲区域的大小也是不固定的，数据页目录的大小也是不固定的，然后文件尾部占据8个字节。

看完了这个数据页的结构，是不是觉得很神奇？居然冒出了这么多稀奇古怪的概念出来。

其实也没什么好奇怪的，说白了，这个数据页就跟每一行数据一样，都是由MySQL开发人员设计出来的一个特殊的存储格式。

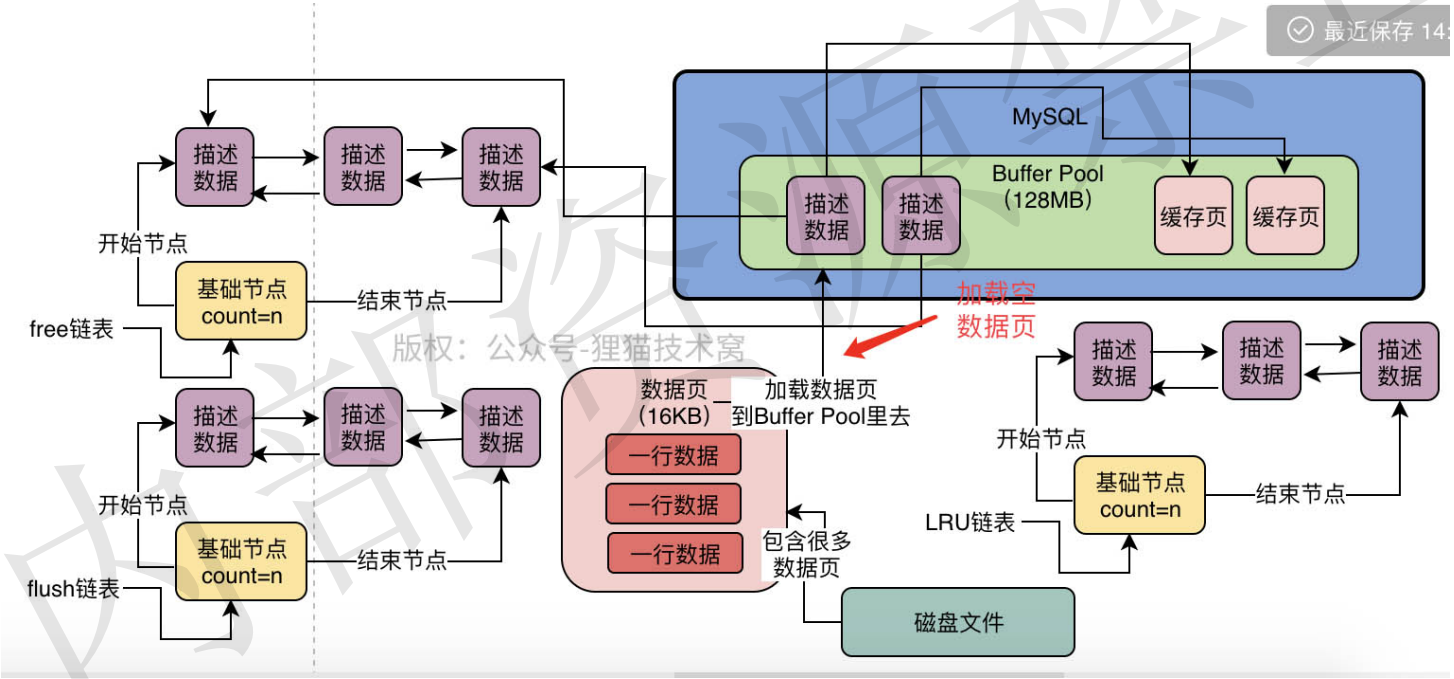
也就是说通过这种特殊的存储格式在磁盘文件里去存放一个又一个的数据页，每个数据页在磁盘里实际存储的时候，就是包含了上述一些特殊的数据，然后每个数据页里还有专门的区域包含了多个数据行，至于每个数据行，那就是用我们之前讲解的那套存储格式来存储的了。

接着我们给大家讲一下这个把数据插入数据页的一个过程，因为大家都知道，刚开始一个数据页可能是空的，没有一行数据的，此时这个数据页实际上是没有数据行那个区域的

也就是说，此时看起来一个空的数据页就是下面图里那样的。

然后我们来思考一下，假设我们现在要插入一行数据，此时数据库里可是一行数据都没有的，那么此时是不是应该先是从磁盘上加载一个空的数据页到缓存页里去？

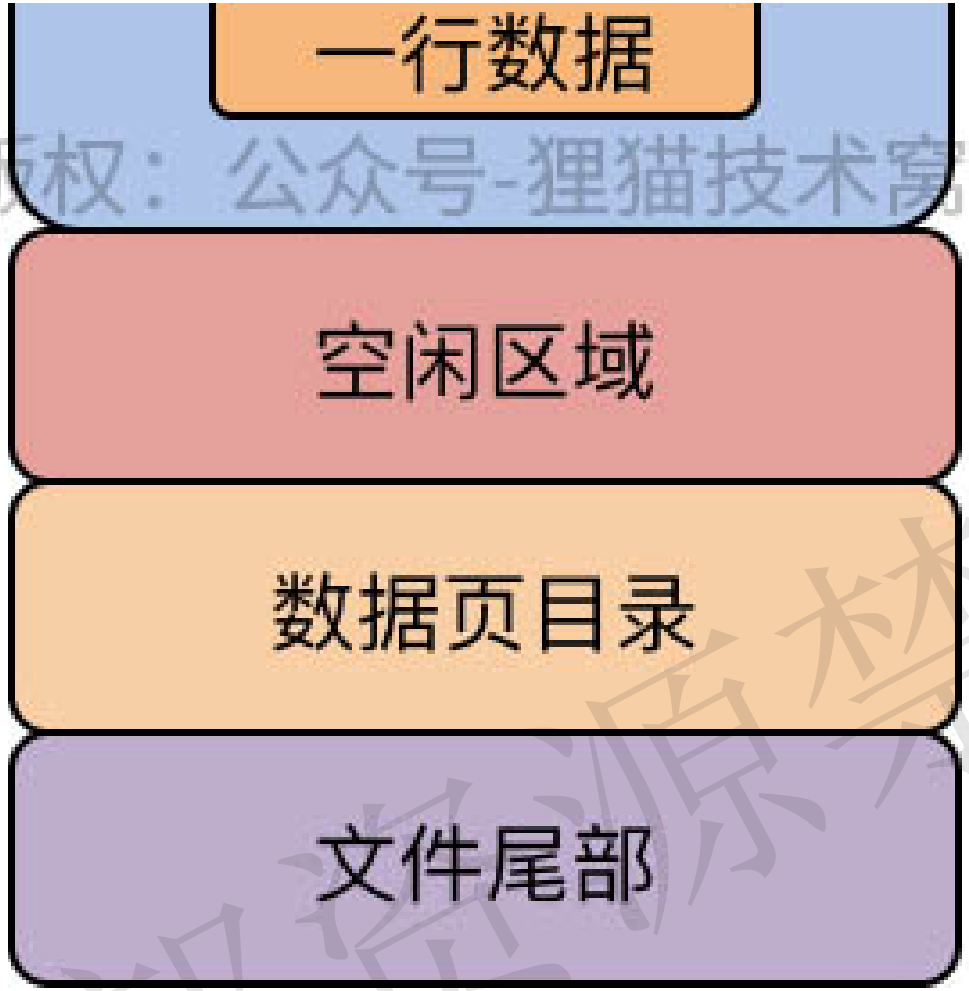
此时空的数据页就是如上图所示，至于加载的过程，则如下图所示。



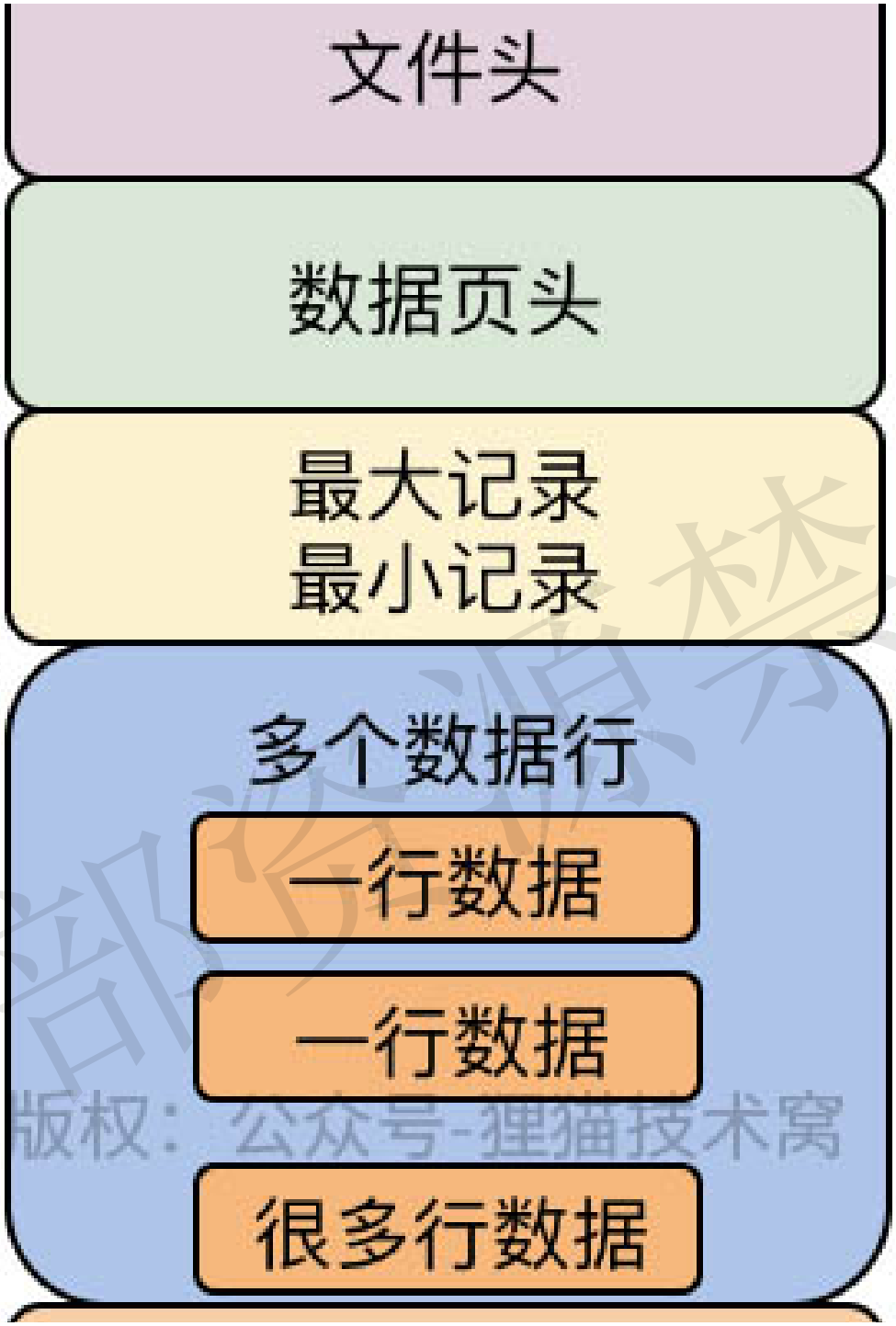
接着我们是不是应该在Buffer Pool中的一个空的缓存页里插入一条数据？

记住，缓存页跟数据页是一 一对应的，他在磁盘上的时候就是数据页，数据页加载到缓存页里了，我们就叫他缓存页了！

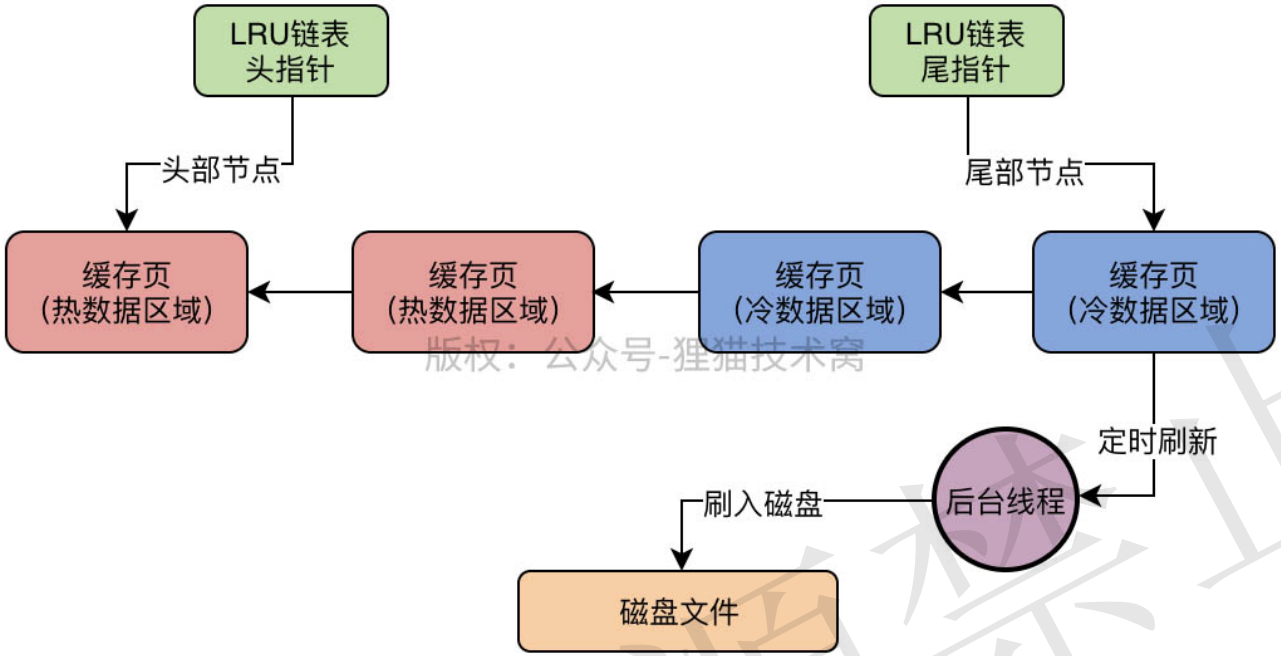
所以此时在缓存页里插入一条数据，实际上就是在数据行那个区域里插入一行数据，然后空闲区域的空间会减少一些，此时当缓存页里插入了一行数据之后，其实缓存页此时看起来如下图所示。



接着你就可以不停的插入数据到这个缓存页里去，直到他的空闲区域都耗尽了，就是这个页满了，此时数据行区域内可能有很多行数据，如下图所示，空闲区域就没了。



而且大家都知道，在更新缓存页的同时，其实他在lru链表里的位置会不停的变动，而且肯定会在flush链表里，所以最终他一定会通过后台IO线程根据lru链表和flush链表，把这个脏的缓存页刷到磁盘上去，如下图所示。



因此对于数据页的整体存储结构的初步介绍，以及MySQL实际运行过程中，数据页的使用，我们今天就介绍完了，接下来我们会继续去了解表空间、数据区、数据段这些物理存储中的概念。

End

狸猫技术窝精品专栏及课程推荐：

- 《从零开始带你成为消息中间件实战高手》
- 《21天互联网Java进阶面试训练营》（分布式篇）
- 《互联网Java工程师面试突击》（第1季）
- 《互联网Java工程师面试突击》（第3季）
- 《从零开始带你成为JVM实战高手》

小鹅通提供技术支持

图文 31 表空间以及划分多个数据页的数据区，又是什么概念？

手机观看

详情 评论

表空间以及划分多个数据页的数据区，又是什么概念？

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

上一次我们讲完了数据页的具体存储结构，当然里面有很多的细节我们还没讲，实际上现在也确实没必要去说那些细节，因为很多数据页的一些细节性的东西，都是要在后续讲解的内容中涉及到的，比如说数据的删除，查询的一些原理。

现在我们在大致了解了数据页的结构和使用之后，我们可以继续来了解下一个概念，就是表空间和数据区的概念

首先我们先说一下，什么是表空间？

简单来说，就是我们平时创建的那些表，其实都是有一个表空间的概念，在磁盘上都会对应着“表名.ibd”这样的一个磁盘数据文件

所以其实在物理层面，表空间就是对应一些磁盘上的数据文件。

有的表空间，比如系统表空间可能对应的是多个磁盘文件，有的我们自己创建的表对应的表空间可能就是对应了一个“表名.ibd”数据文件。

然后在表空间的磁盘文件里，其实会有很多很多的数据页，因为大家都知道一个数据页不过就是16kb而已，总不可能一个数据页就是一个磁盘文件吧。

所以一个表空间的磁盘文件里，其实是有很多的数据页的。

但是现在有一个问题，就是一个表空间里包含的数据页实在是太多了，不便于管理，所以在表空间里又引入了一个**数据区**的概念，英文就是**extent**

一个数据区对应着连续的64个数据页，每个数据页是16kb，所以一个数据区是1mb，然后256个数据区被划分为了一组。

对于表空间而言，他的第一组数据区的第一个数据区的前3个数据页，都是固定的，里面存放了一些描述性的数据。比如FSP_HDR这个数据页，他里面就存放了表空间和这一组数据区的一些属性。

IBUF_BITMAP数据页，里面存放的是这一组数据页的所有insert buffer的一些信息。

INODE数据页，这里也是存放了一些特殊的信息

大家暂时先不用了解这些东西具体是干什么的，你只要知道每一个组数据区的第一个数据区的前3个数据页，都是存放一些特殊的信息的。

然后这个表空间里的其他各组数据区，每一组数据区的第一个数据区的头两个数据页，都是存放特殊信息的，比如XDES数据页就是用来存放这一组数据区的一些相关属性的，其实就是很多描述这组数据区的东西，现在大家也不用去

知道是什么。

其实今天的内容讲到这里就差不多了，讲太多大家可能就被绕晕了，大家只要知道，**我们平时创建的那些表都是有对应的表空间的，每个表空间就是对应了磁盘上的数据文件，在表空间里有很多组数据区，一组数据区是256个数据区，每个数据区包含了64个数据页，是1mb**

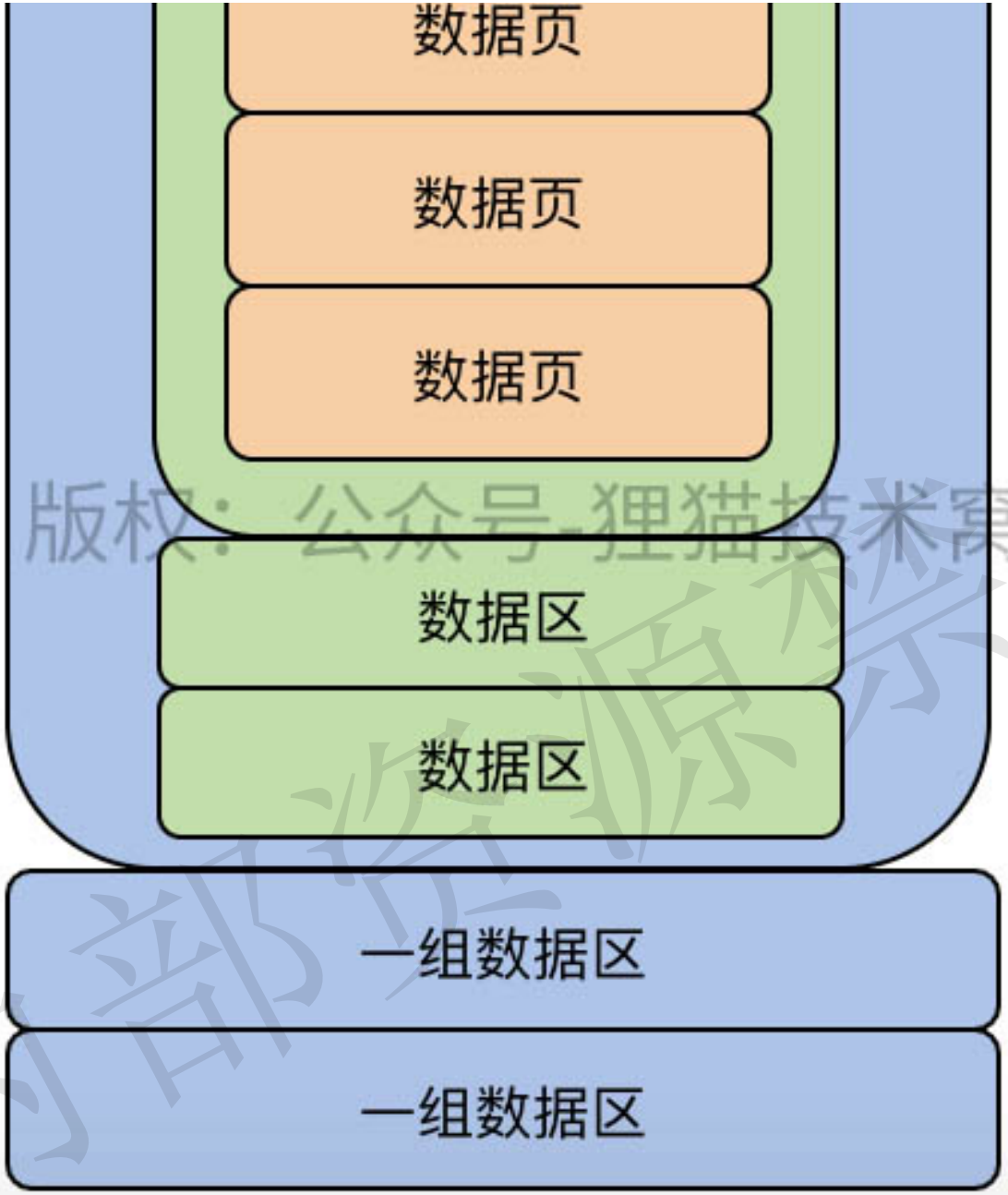
然后表空间的第一组数据区的第一个数据区的头三个数据页，都是存放特殊信息的；

表空间的其他组数据区的第一个数据区的头两个数据页，也都是存放特殊信息的。大家今天只要了解到这个程度就可以了。

所以磁盘上的各个表空间的数据文件里是通过数据区的概念，划分了很多很多的数据页的，因此**当我们需要执行crud操作的时候，说白了，就是从磁盘上的表空间的数据文件里，去加载一些数据页出来到Buffer Pool的缓存页里去使用。**

我下面给出了一张图，图里就给出了一个表空间内部的存储结构，包括一组一组的数据区，每一组数据区是256个数据区，然后一个数据区是64个数据页。

请大家牢记下图：



End

狸猫技术窝精品专栏及课程推荐：

- 《从零开始带你成为消息中间件实战高手》
- 《21天互联网Java进阶面试训练营》（分布式篇）
- 《互联网Java工程师面试突击》（第1季）
- 《互联网Java工程师面试突击》（第3季）
- 《从零开始带你成为JVM实战高手》

图文 32 一文总结初步了解到的MySQL存储模型以及数据读写机制

手机观看

156 人次阅读 2020-02-28 10:06:56

详情 评论

一文总结初步了解到的MySQL存储模型以及数据读写机制

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

今天我们来用一篇文章初步总结一下我们近期学习到的MySQL存储模型以及对应的读写机制，其实大家通过近期的学习也仅仅是初步了解了MySQL底层数据的存储模型而已，因为后续我们还要讲解MySQL的增删改查执行背后的深入底层的各种存储数据读写细节，现在仅仅是初步把存储模型的结构给建立起来罢了。

好，那么我们现在应该都知道了，最终MySQL的数据都是放在磁盘文件里的，这个大家应该都没什么问题吧

那么数据在磁盘文件里是怎么存放的呢？我们都知道我们平时数据都是插入一个一个的表中的，而表是个逻辑概念，其实在物理层面，他对应的是**表空间**这个概念。

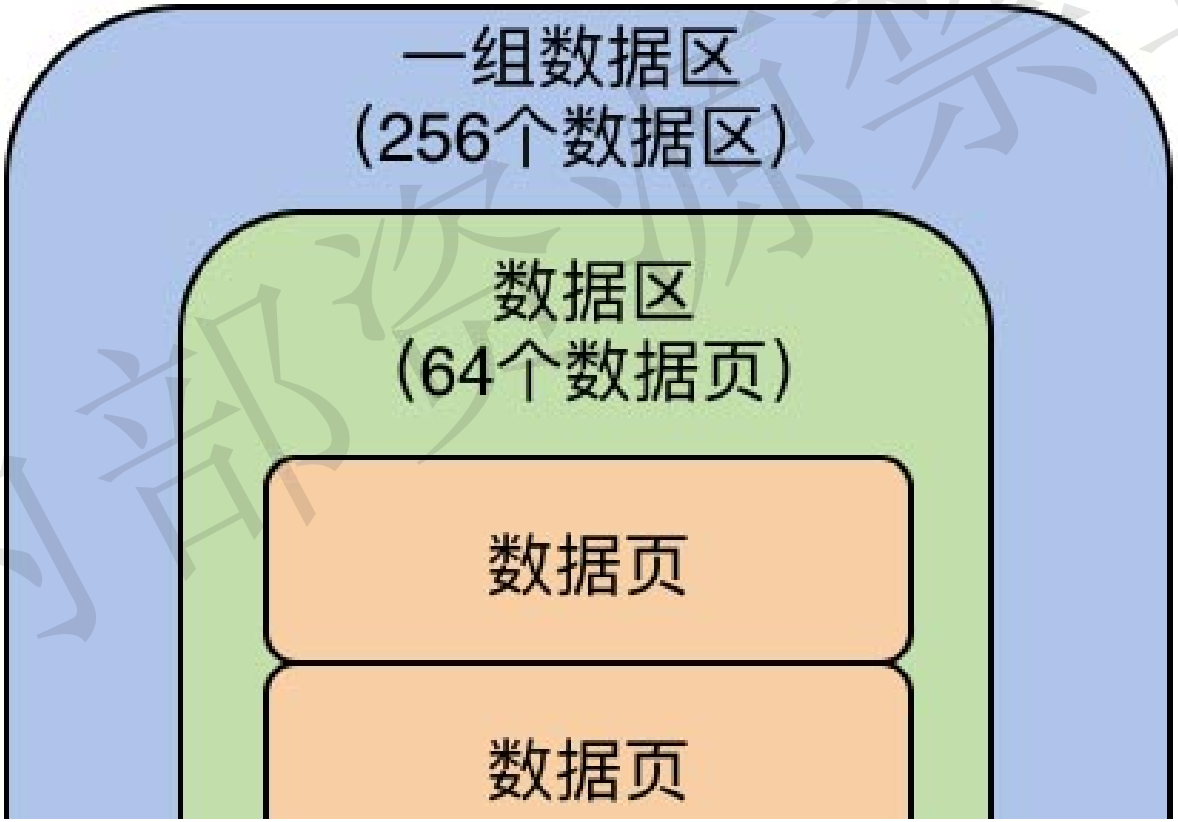
所以其实在MySQL的磁盘上，表空间就对应着磁盘文件，在磁盘文件里就存放着数据！

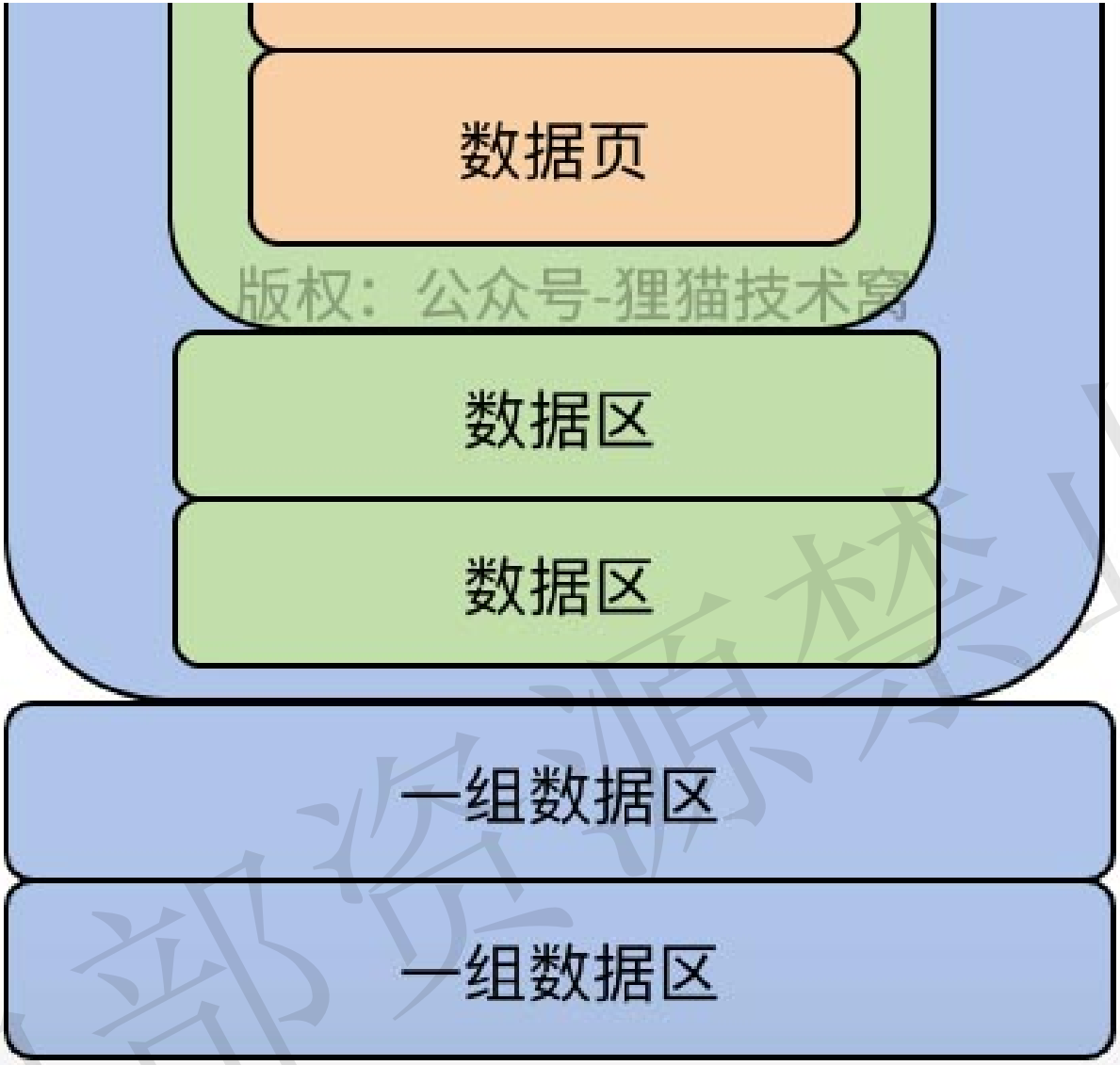
那么这个表空间的磁盘文件里，数据是如何组织的呢？

这个就非常的复杂了！因为你可以想象一下，假如让你把数据直接一行一行的写入一个磁盘文件，当然很简单了！

但是问题是你现在要存储的是数据库里的如此复杂的数据！他里面是有各种字段类型的，还有索引这个概念，当后面我们讲到索引的时候，就会详细分析这个索引在磁盘里的数据组织结构，这也是相当的复杂。

所以其实在磁盘文件里存放的数据，他从最基本的角度来看的话，就是被拆分为一个一个的数据区（extent）分组，以后我们干脆就用他的英文名叫做extent组好了，每个extent组中包含256个extent，然后每个extent里包含64个数据页！然后每个数据页里都包含了一行一行的数据！如下图所示。





大家听到这里，是不是觉得特别的简单，其实绝对不是的

在实际存储的时候，我们之前稍微给大家介绍过一点点，在数据行里都有很多附加的信息，在数据页、数据区里，都有很多附加的特殊信息。各种各样的特殊信息，就可以让我们在简简单单的磁盘文件里实现B+树索引、事务之类的非

常复杂的机制。

关于这些存储结构中的特殊信息是如何用于支撑实现很多高级的复杂功能的，我们后续会给大家讲解的！

那么现在问题来了，我们都知道，当我们在数据库中执行crud的时候，你必须先把磁盘文件里的一个数据页加载到内存的Buffer Pool的一个缓存页里去，然后我们增删改查都是针对缓存页里的数据来执行的！

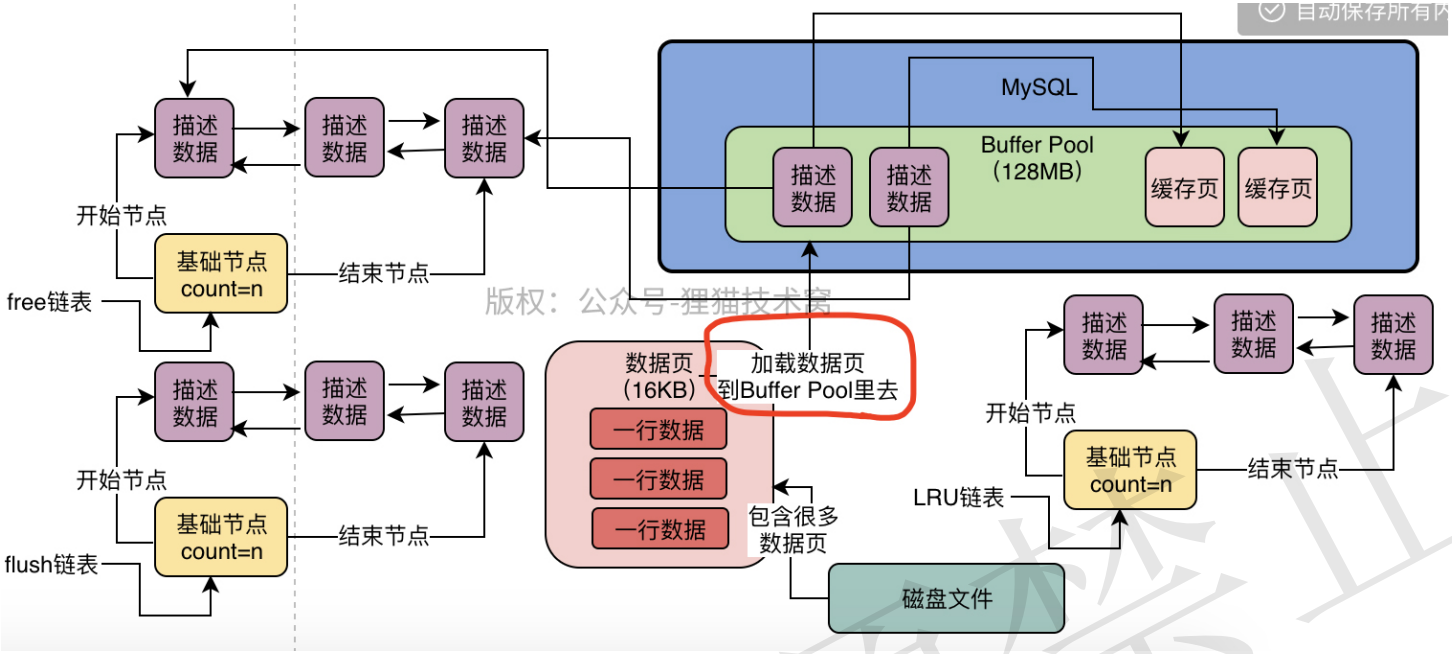
所以假设此时我们要插入一条数据，那么是选择磁盘文件里的哪个数据页加载到缓存页里去呢？

大家注意，这里要划重点了，其实这个时候会看看你往哪个表里插入数据？然后肯定得根据表找到一个表空间啊！

找到表空间之后，就可以定位到对应的磁盘文件啊！有了磁盘文件之后，就可以从里面找一个extent组，找一个extent，接着从里面找一个数据页出来！这个数据也可能是空的，也可能已经放了一些数据行了！

然后就可以把这个数据页从磁盘里完整加载出来，放入Buffer Pool的缓存页里了！

我们看下图的示意



当然，这个时候有人会问了，这个从磁盘文件里读取一个数据页，是怎么读取的啊？

其实这个很简单了，你可以想一下，磁盘文件里放的数据都是紧挨在一起的，类似于下面的那种样子。

0xdfs3439399abc0sfsdkslf9sdfpsfds0xdfs3439399abc0sfsdkslf9sdfpsfds

0xdfs3439399abc0sfsdkslf9sdfpsfds0xdfs3439399abc0sfsdkslf9sdfpsfds

其实上述字符完全无任何意义，就是我为了演示随便搞出来的一段东西而已，但是大致来说磁盘里存放的数据看起来就是那样的，可能先是有一个extent组开始的一些东西，然后里面是一个一个的extent，每个extent开始的时候会写一些特殊的信息，然后再是一个一个的数据页，里面是一个一个的数据行。

那么在读取一个数据页的时候，你就可以通过随机读写的方式来了，举个例子，我们下面有一个伪代码，大家看看。就是设置一下要从一个数据文件的哪个位置开始读取，一直到哪个位置就结束。

dataFile.setStartPosition(25347)

```
dataFile.setEndPosition(28890)
dataPage = dataFile.read()
```

通过上面伪代码那种方式，你指定磁盘文件里的开始和截止的位置，就能读取出来指定位置的一段数据，比如读取出来一大坨东西：psfds0xdfs343939。也许这坨东西就是一个数据页包含的内容了。

然后把数据页放到内存的缓存页里即可。

接着crud操作都可以直接针对缓存页去执行了，会自动把更新的缓存页加入flush链表，然后更新他在lru链表里的位置，包括更新过的缓存页会从free链表里拿出来，等等，后续一系列操作，都是之前我们分析过的了。

此时对于那些被更新过的缓存页来说，都会由后台线程刷入磁盘的，那么刷磁盘的时候是怎么刷呢？我们也是写一段伪代码给大家看看。

```
dataFile.setStartPosition(25347)
dataFile.setEndPosition(28890)
dataFile.write(cachePage)
```

因为一个数据页的大小其实是固定的，所以一个数据页固定就是可能在一个磁盘文件里占据了某个开始位置到结束位置的一段数据，此时你写回去的时候也是一样的，选择好固定的一段位置的数据，直接把缓存页的数据写回去，就覆盖掉了原来的那个数据页了，就如上面的伪代码示意。

今天通过一篇总结文章，就给大家讲清楚了我们初步了解到的存储模型是如何跟Buffer Pool缓存机制配合起来实现crud的，接着我们会给大家讲解几个数据库优化的案例了！

我们的风格就是讲一些理论的知识，就配合一些实战案例，让大家理论和实战相结合起来。

End

狸猫技术窝精品专栏及课程推荐：

《从零开始带你成为消息中间件实战高手》
《21天互联网Java进阶面试训练营》（分布式篇）
《互联网Java工程师面试突击》（第1季）
《互联网Java工程师面试突击》（第3季）
《从零开始带你成为JVM实战高手》

图文 33 MySQL数据库的日志顺序读写以及数据文件随机读写的原理

手机观看

565 人次阅读 2020-03-02 11:28:53

详情 评论

MySQL数据库的日志顺序读写以及数据文件随机读写的原理

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

之前我们花了很多篇幅去讲解MySQL的底层数据存储结构，其实那些知识是极为枯燥的，因为大部分时候，MySQL在底层如何存储数据的一些细节，比如什么数据头、附加信息之类的极为复杂，大家直接那么研究是很痛苦的。

所以我之前也就初步的给大家介绍了一下数据行、数据页、extent、extent分组、表空间、磁盘文件这些概念，主要是让大家把物理数据结构与Buffer Pool缓存的结合使用，有一个理解就行了。

掌握到之前的一些知识，基本上MySQL稍微进一步的原理，大家也就有一定的了解了。其实暂时来说这就足够了，更加细节的一些知识，比如表空间的存储结构细节，extent的存储结构细节，都要结合未来的索引优化原理、数据删除原理，结合这些东西去分析，大家从自己日常都接触的一些场景出发，去看一些技术细节，才能真正很好理解。

那么今天开始，我们将要用连续几天的时间，给大家介绍一个**真实的生产优化案例**，这个案例主要用到的知识，其实大家之前都学过了

所以这也是我一如既往的专栏风格，讲一些理论，同时插入一些我们生产环境的真实案例分析，让大家理论和实战结合起来。

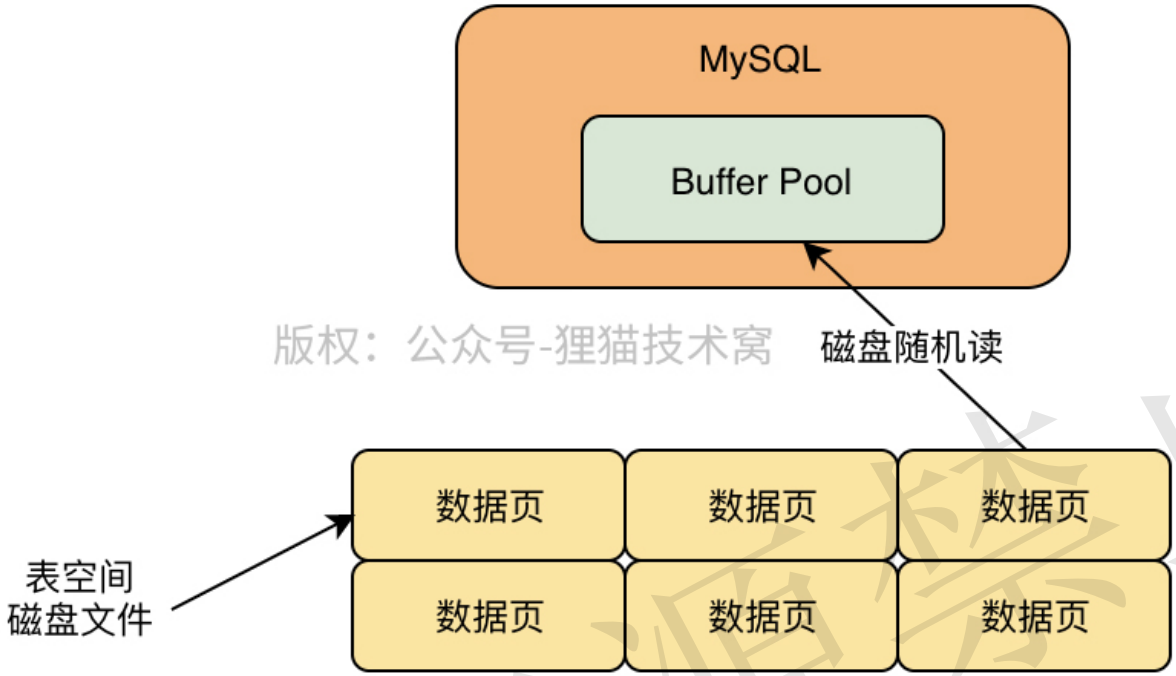
在讲解这个真实的生产案例之前，有一些前置的知识要给大家介绍一下

首先今天要讲解的就是MySQL数据库和底层的操作系统之间的交互原理，理解了这个原理后，我们再一步步剖析一个生产环境的MySQL数据库每隔一两个月性能就会出现急剧抖动的案例。

先给大家剖析一下MySQL在实际工作时候的两种数据读写机制，一种是对redo log、binlog这种日志进行的磁盘顺序读写，一种是对表空间的磁盘文件里的数据页进行的磁盘随机读写。

简单来说，MySQL在工作的时候，尤其是执行增删改操作的时候，肯定会先从表空间的磁盘文件里读取数据页出来，这个过程其实就是典型的磁盘随机读操作

我们先看下面的图，图里有一个磁盘文件的示意，里面有很多数据页，然后你可能需要在一个随机的位置读取一个数据页到缓存，这就是**磁盘随机读**



因为你要读取的这个数据页可能在磁盘的任意一个位置，所以你在读取磁盘里的数据页的时候只能是用随机读的这种方式。

磁盘随机读的性能是比较差的，所以不可能每次更新数据都进行磁盘随机读，必须是读取一个数据页之后放到Buffer Pool的缓存里去，下次要更新的时候直接更新Buffer Pool里的缓存页。

对于磁盘随机读来说，主要关注的性能指标是IOPS和响应延迟

IOPS之前给大家介绍过，就是说底层的存储系统每秒可以执行多少次磁盘读写操作，比如你底层磁盘支持每秒执行1000个磁盘随机读写操作和每秒执行200个磁盘随机读写操作，对你的数据库的性能影响其实是非常大的。

这个IOPS指标如何观察，之前也讲过了，大家在压测的时候可以观察一下。这个指标实际上对数据库的crud操作的QPS影响是非常大的，因为他在某种程度上几乎决定了你每秒能执行多少个SQL语句，底层存储的IOPS越高，你的数据库的并发能力就越高。

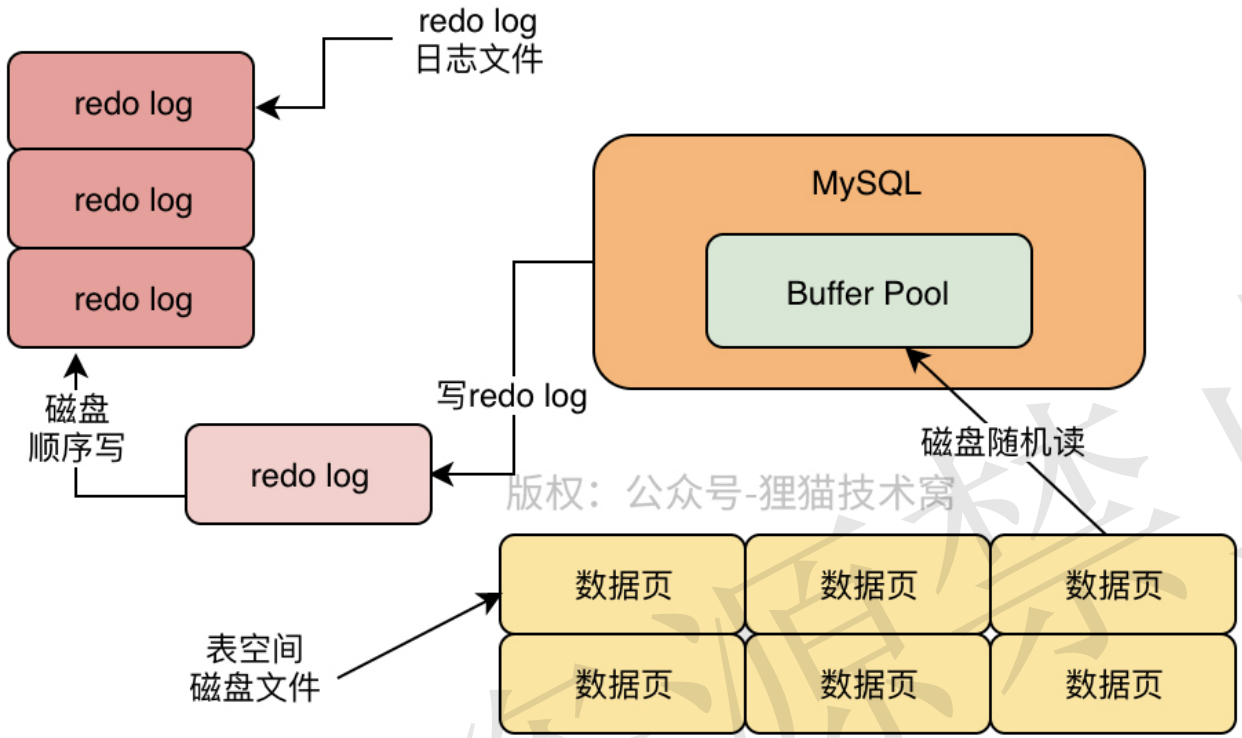
另外一个就是磁盘随机读写操作的响应延迟，也是对数据库的性能有很大的影响。因为假设你的底层磁盘支持你每秒执行200个随机读写操作，但是每个操作是耗费10ms完成呢，还是耗费1ms完成呢，这个其实也是有很大的影响的，决定了你对数据库执行的单个crud SQL语句的性能。

比如你一个SQL语句发送过去，他磁盘要执行随机读操作加载多个数据页，此时每个磁盘随机读响应时间是50ms，那么此时可能你的SQL语句要执行几百ms，但是如果每个磁盘随机读仅仅耗费10ms，可能你的SQL就执行100ms就行了。

所以其实一般对于核心业务的数据库的生产环境机器规划，我们都是推荐用SSD固态硬盘的，而不是机械硬盘，因为SSD固态硬盘的随机读写并发能力和响应延迟要比机械硬盘好的多，可以大幅度提升数据库的QPS和性能。

接着我们来看磁盘顺序读写，之前我们都知道，当你在Buffer Pool的缓存页里更新了数据之后，必须要写一条redo log日志，这个redo log日志，其实就是就是走的顺序写

所谓顺序写，就是说在一个磁盘日志文件里，一直在末尾追加日志，我们看下图。



所以上图可以清晰看到，写redo log日志的时候，其实是不停的在一个日志文件末尾追加日志的，这就是磁盘顺序写。

磁盘顺序写的性能其实是很高的，某种程度上来说，几乎可以跟内存随机读写的性能差不多，尤其是在数据库里其实也用了os cache机制，就是redo log顺序写入磁盘之前，先是进入os cache，就是操作系统管理的内存缓存里。

所以对于这个写磁盘日志文件而言，最核心关注的是磁盘每秒读写多少数据量的吞吐量指标，就是说每秒可以写入磁盘100MB数据和每秒可以写入磁盘200MB数据，对数据库的并发能力影响也是极大的。

因为数据库的每一次更新SQL语句，都必然涉及到多个磁盘随机读取数据页的操作，也会涉及到一条redo log日志文件顺序写的操作。所以磁盘读写的IOPS指标，就是每秒可以执行多少个随机读写操作，以及每秒可以读写磁盘的数据量的吞吐量指标，就是每秒可以写入多少redo log日志，整体决定了数据库的并发能力和性能。

包括你磁盘日志文件的顺序读写的响应延迟，也决定了数据库的性能，因为你写redo log日志文件越快，那么你的SQL语句性能就越高。

所以今天就先给大家在之前知识的基础之上，讲解一下数据库运行过程中，磁盘随机读写和磁盘顺序读写的两个机制的原理。

End

图文 34 生产经验：Linux操作系统的存储系统软件层原理剖析以及IO调度优化原理

手机观看

590 人次阅读 2020-03-03 07:00:00

详情 评论

生产经验：Linux操作系统的存储系统软件层原理剖析以及IO调度优化原理

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

接着上一篇文章的讲解，我们继续来讲解MySQL数据库在执行底层磁盘读写IO操作的原理，这其实就涉及到了Linux操作系统的磁盘IO原理了，不管是MySQL执行磁盘随机读写，还是磁盘顺序读写，其实在底层的Linux层面，原理几乎都是一致的。

同时我们还会针对这块内容，连带讲解一下生产环境中，针对MySQL数据库的IO调度优化的建议。

大家都知道，所谓的操作系统，无论是Linux也好，还是Windows也好，说白了他们自己本身就是软件系统，之所以需要操作系统，是因为我们不可能直接去操作CPU、内存、磁盘这些硬件，所以必须要用操作系统来管理CPU、内存、磁盘、网卡这些硬件设备。

操作系统除了管理硬件设备以外，还会提供一个操作界面给我们，比如Windows之所以在全世界大获成功，其实就是他提供了一个比较简便易用的可视化的界面，让我们可以普通人都能操作台式电脑或者笔记本电脑内部的内存、CPU、磁盘和网卡。

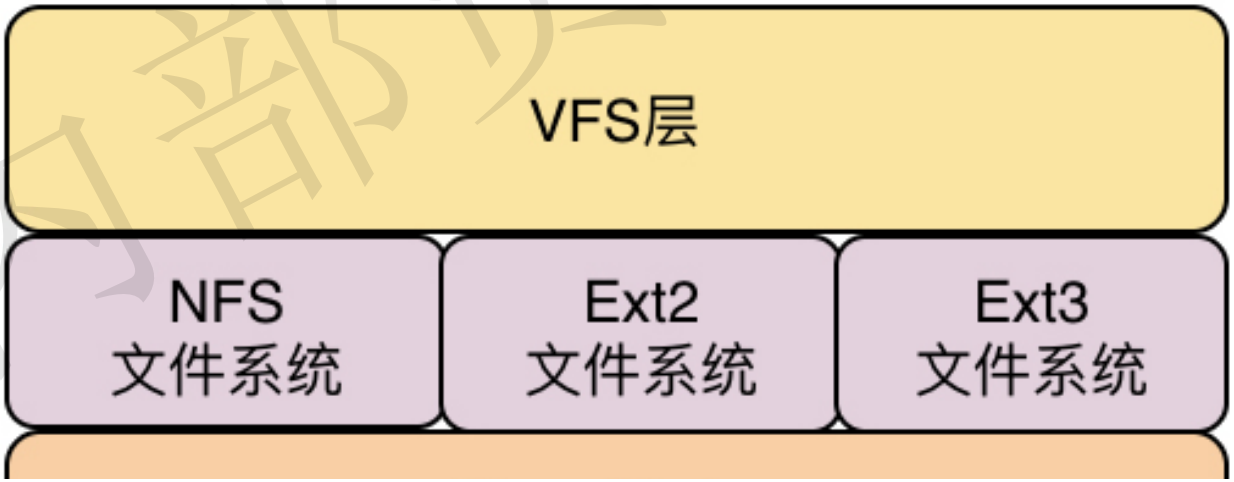
我们只要打开windows操作系统的电脑，就可以随意编辑文件，上网，聊天，使用各种软件，这些软件运行的时候本质底层都是在使用计算机的CPU、内存、磁盘和网卡，比如基于CPU执行你的文件编辑的操作，基于内存缓冲你对文件的编辑，基于磁盘存储你在文件里输入的内容，基于网卡去进行网络通信，让你进行QQ聊天什么的。

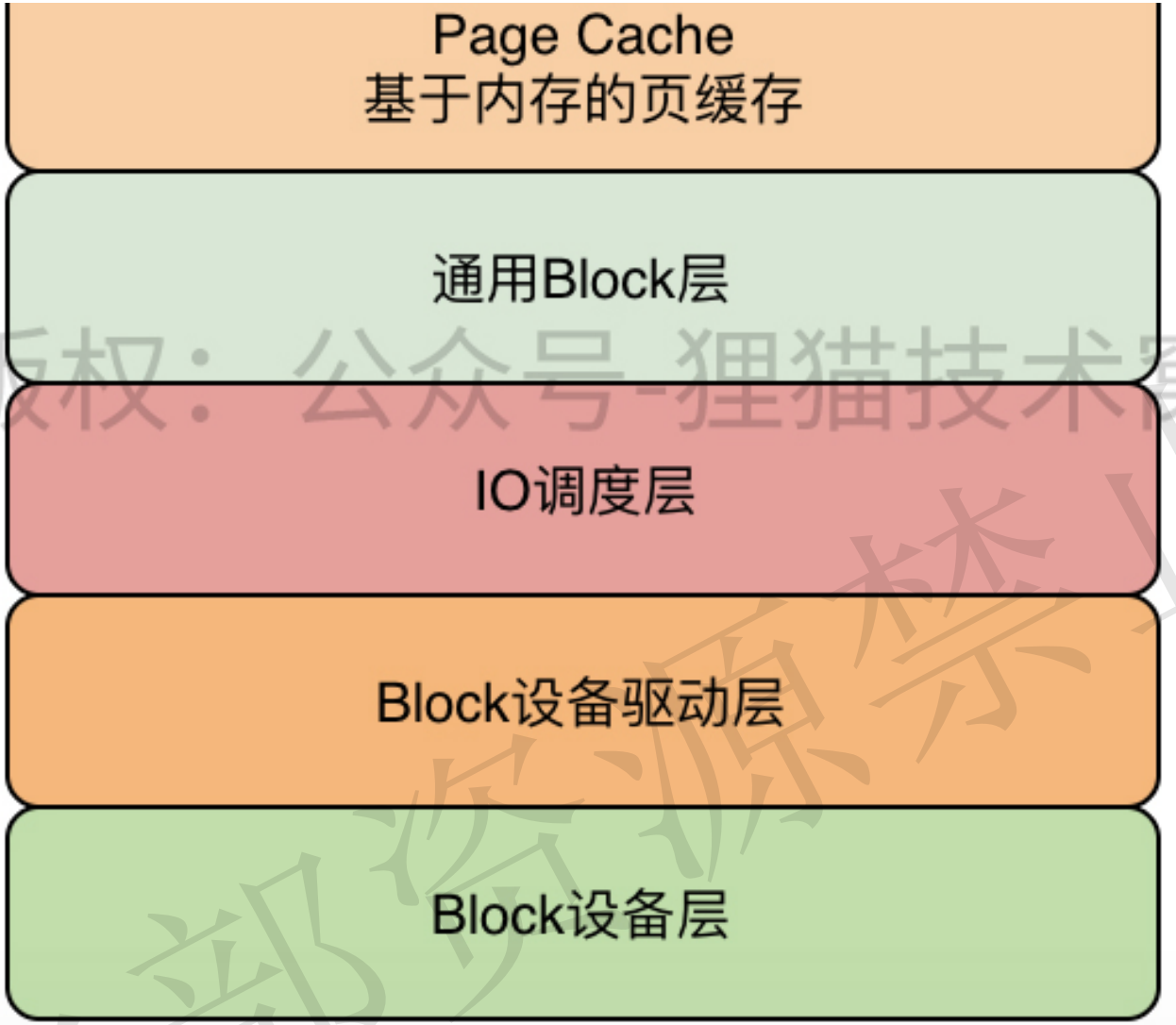
至于说linux操作系统，其实也是类似的，只不过一般我们用linux操作系统，他是不给我们提供可视化界面的，只有命令行的界面，我们需要输入各种各样的命令去执行文件编辑、系统部署和运行，本质linux操作系统在底层其实也是利用CPU、内存、磁盘和网卡这些硬件在工作。

所以，简单来说，我们今天要讲解的就是Linux操作系统的存储系统，Linux利用这套存储系统去管理我们的机器上的机械硬盘、□SSD固态硬盘，这些存储设备，可以在里面读取数据，或者是写入数据。

理解了这个，你就理解了MySQL执行的数据页随机读写，redo log日志文件顺序读写的磁盘IO操作，在Linux的存储系统中是如何执行的。

简单来说，Linux的存储系统分为VFS层、文件系统层、Page Cache缓存层、通用Block层、IO调度层、Block设备驱动层、Block设备层，如下图：

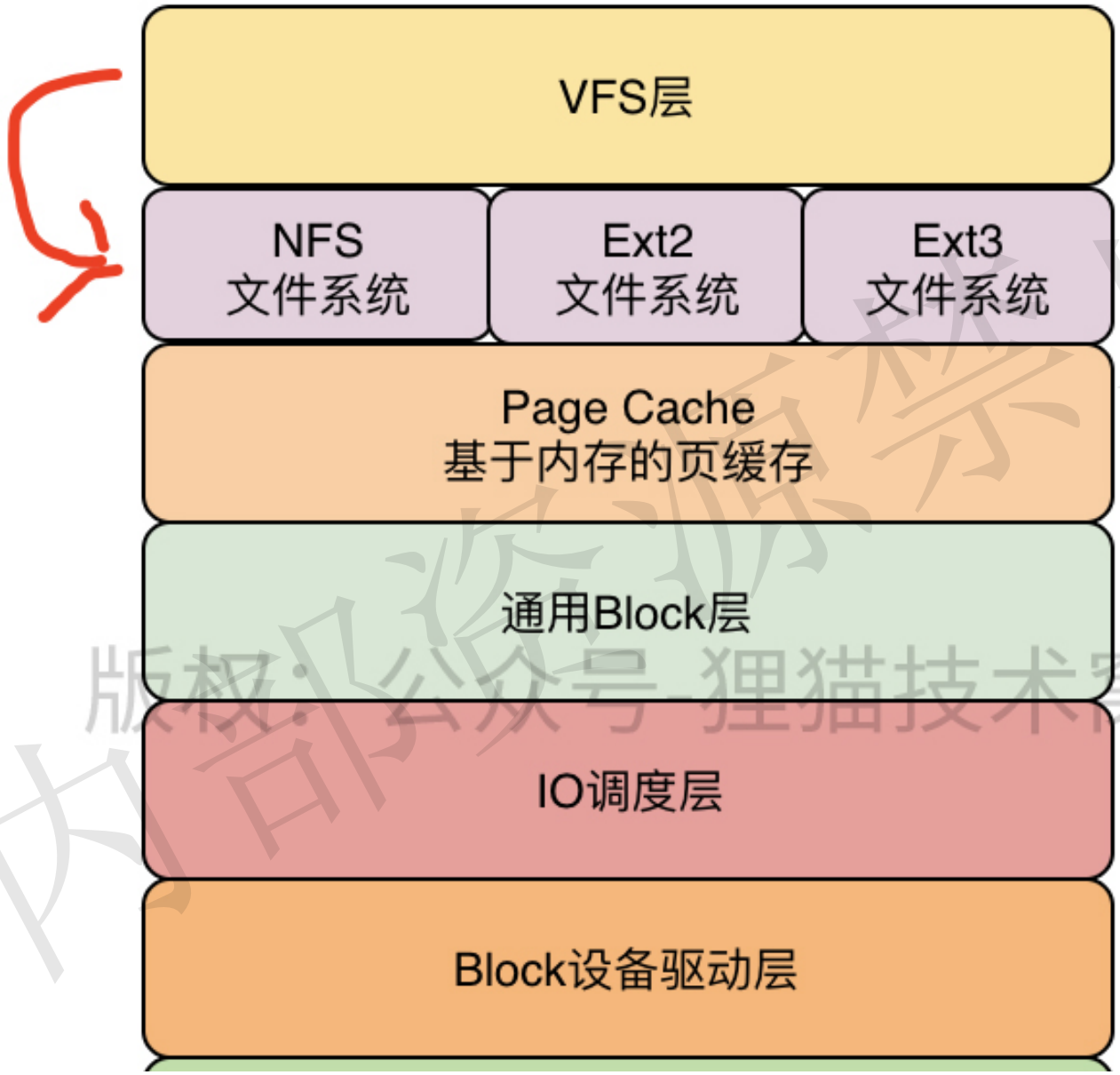




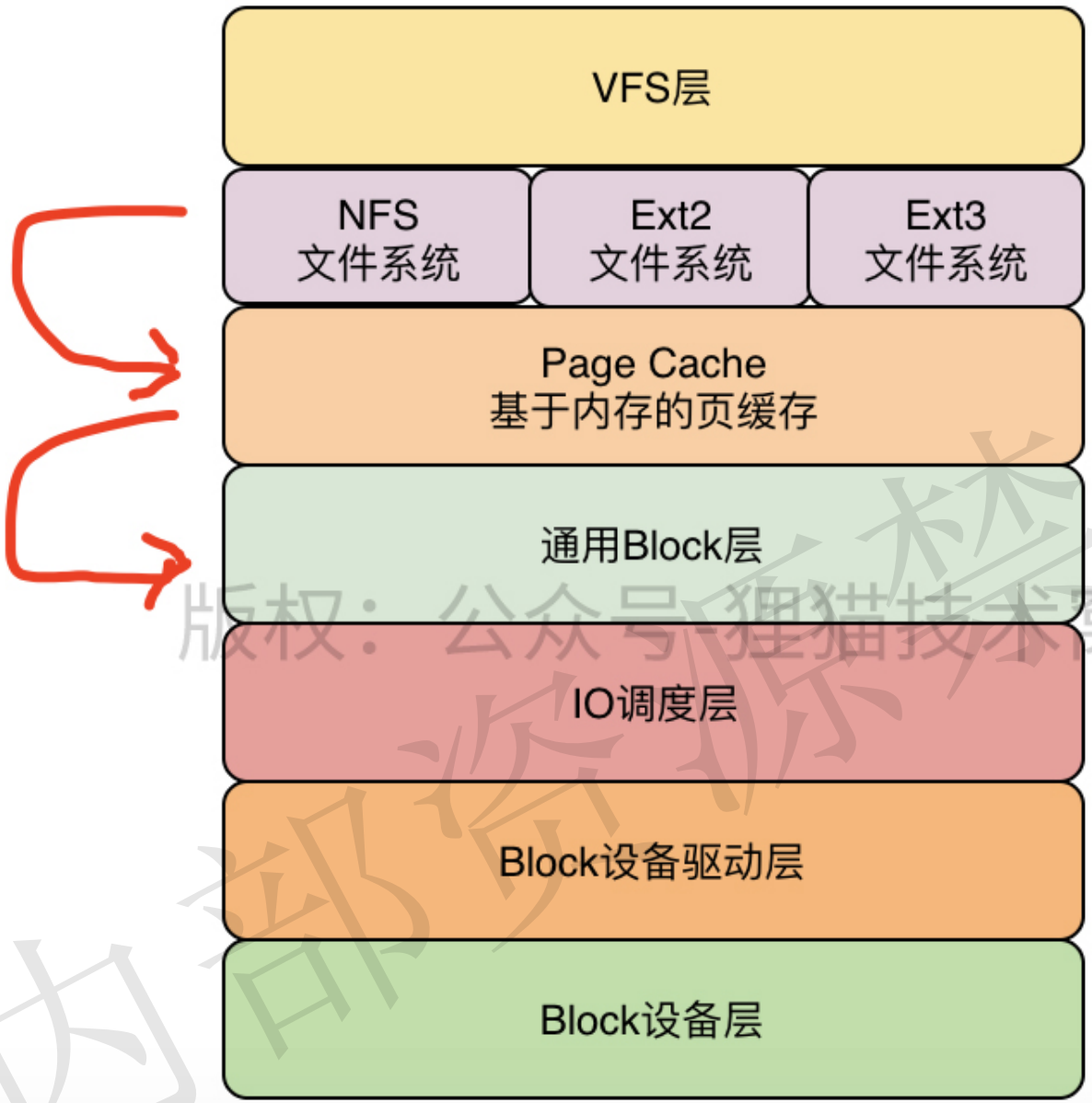
当MySQL发起一次数据页的随机读写，或者是一次redo log日志文件的顺序读写的时候，实际上会把磁盘IO请求交给Linux操作系统的VFS层

这一层的作用，就是根据你是对哪个目录中的文件执行的磁盘IO操作，把IO请求交给具体的文件系统。

举个例子，在linux中，有的目录比如/xx1/xx2里的文件其实是由NFS文件系统管理的，有的目录比如/xx3/xx4里的文件其实是由Ext3文件系统管理的，那么这个时候VFS层需要根据你是对哪个目录下的文件发起的读写IO请求，把请求转交给对应的文件系统，如下图所示。



接着文件系统会先在Page Cache这个基于内存的缓存里找你要的数据在不在里面，如果有就基于内存缓存来执行读写，如果没有就继续往下一层走，此时这个请求会交给通用Block层，在这一层会把你对文件的IO请求转换为Block IO请求，如下图所示。



接着IO请求转换为Block IO请求之后，会把这个Block IO请求交给IO调度层，在这一层里默认是用CFQ公平调度算法的

也就是说，可能假设此时你数据库发起了多个SQL语句同时在执行IO操作。

有一个SQL语句可能非常简单，比如update xxx set xx1=xx2 where id=1，他其实可能就只要更新磁盘上的一个block里的数据就可以了

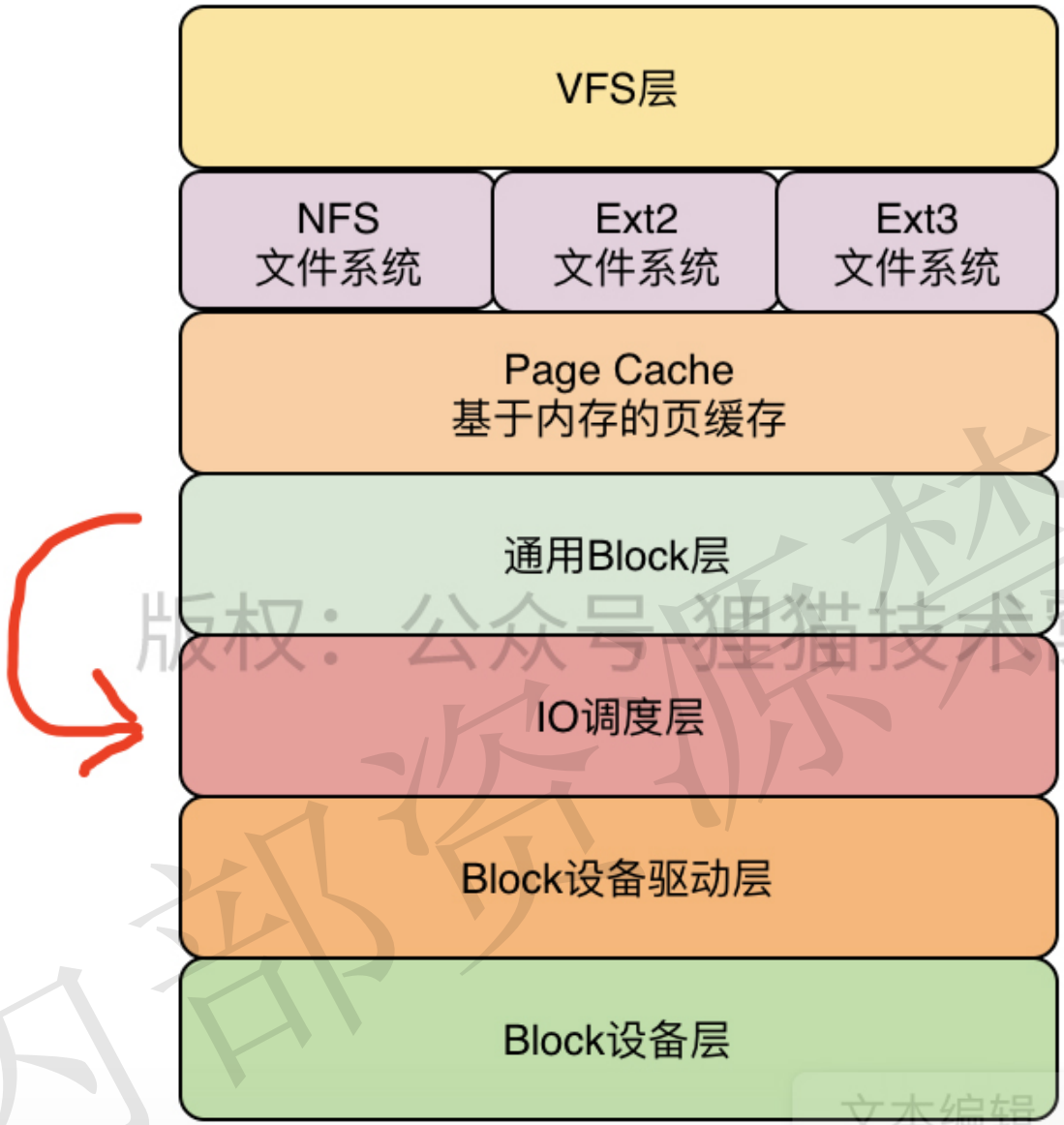
但是有的SQL语句，比如说select * from xx where xx1 like "%xx%"可能需要IO读取磁盘上的大量数据。

那么此时如果基于公平调度算法，就会导致他先执行第二个SQL语句的读取大量数据的IO操作，耗时很久，然后第一个仅仅更新少量数据的SQL语句的IO操作，就一直在等待他，得不到执行的机会。

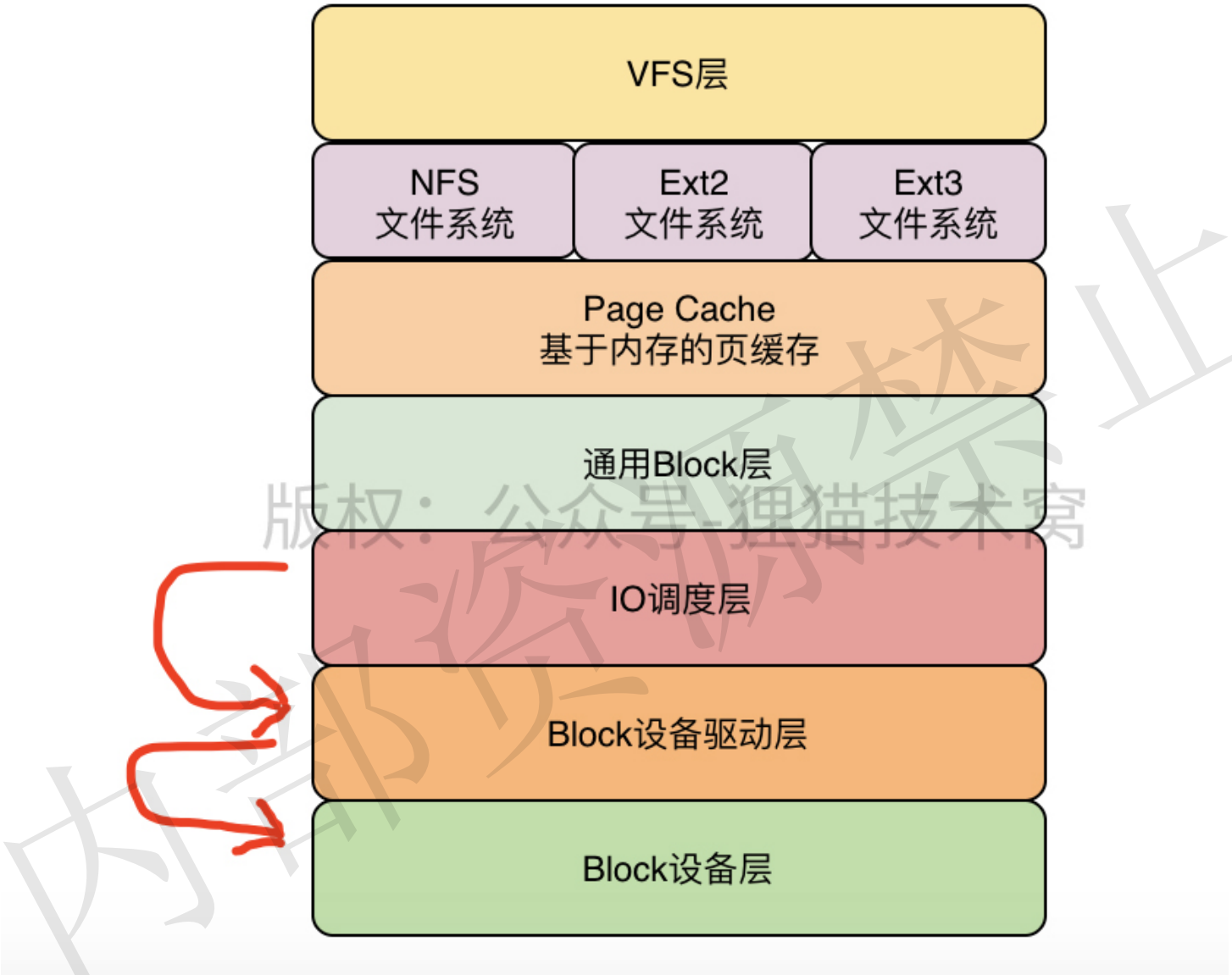
所以在这里，其实一般建议MySQL的生产环境，需要调整为deadline IO调度算法，他的核心思想就是，任何一个IO操作都不能一直不停的等待，在指定时间范围内，都必须让他去执行。

所以基于deadline算法，上面第一个SQL语句的更新少量数据的IO操作可能在等待一会儿之后，就会得到执行的机会，这也是一个生产环境的IO调度优化经验。

我们看下图，此时IO请求被转交给了IO调度层。



最后IO完成调度之后，就会决定哪个IO请求先执行，哪个IO请求后执行，此时可以执行的IO请求就会交给Block设备驱动层，然后最后经过驱动把IO请求发送给真正的存储硬件，也就是Block设备层，如下图所示。



然后硬件设备完成了IO读写操作之后，要不然是写，要不然是读，最后就把响应经过上面的层级反向依次返回，最终MySQL可以得到本次IO读写操作的结果

这就是MySQL跟Linux存储系统交互的的一个原理剖析，包括里面的IO调度算法那块的一个优化的点，大家可以仔细理解一下今天的内容。

End

狸猫技术窝精品专栏及课程推荐：

- 《从零开始带你成为消息中间件实战高手》
- 《21天互联网Java进阶面试训练营》（分布式篇）
- 《互联网Java工程师面试突击》（第1季）
- 《互联网Java工程师面试突击》（第3季）
- 《从零开始带你成为JVM实战高手》

图文 35 生产经验：数据库服务器使用的RAID存储架构初步介绍

手机观看

详情 评论

生产经验：数据库服务器使用的RAID存储架构初步介绍

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

今天我们继续给大家讲解生产环境下的MySQL数据库的一些存储技术的原理，之前已经给大家解释了MySQL的磁盘随机读写和顺序读写的场景和原理，包括Linux操作系统的存储系统的原理，那么我们接着就要继续讲解Linux操作系统再底层的存储硬件层面的一些原理

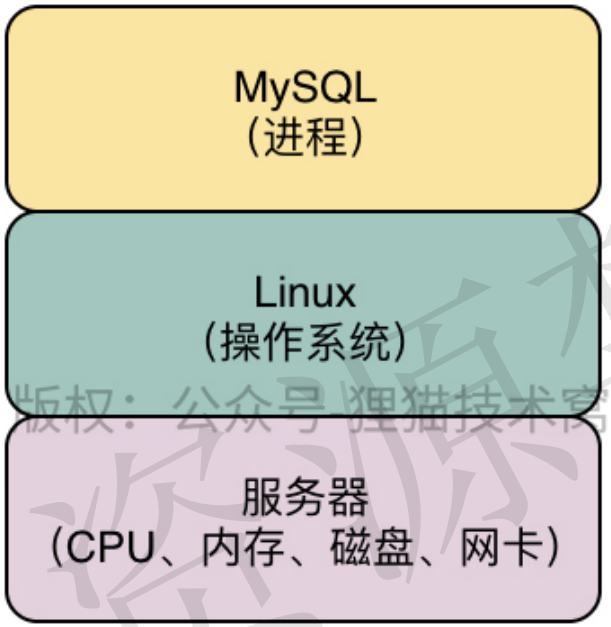
只有把这些都理解透了，才能给大家最终讲清楚一次线上生产环境的MySQL数据库的性能抖动的故障原因。

实际上MySQL数据库就是个软件，大家都知道，他其实就是用编程语言写的一套数据库管理软件而已，底层就是磁盘来存储数据，基于内存来提升数据读写性能，然后设计了复杂的数据模型，帮助我们高校的存储和管理数据。

所以MySQL数据库软件都是安装在一台linux服务器上的，然后启动MySQL的进程，就是启动了一个MySQL数据库

MySQL运行过程中，他需要使用CPU、内存、磁盘和网卡这些硬件，但是不能直接使用，都是通过调用操作系统提供的接口，依托于操作系统来使用和运行的，然后linux操作系统负责操作底层的硬件。

我们下面画了一个图，来给大家表示一下，他们之间的一个关系。



所以之前我们已经把MySQL层面的磁盘读写操作讲完了，同时也把Linux操作系统层面的存储系统的原理讲完了，上一讲不就讲到Linux操作系统的存储系统把IO请求交给机器上的存储硬件了么？

那么今天我们接着来讲讲存储硬件这块的东西。

一般来说，很多数据库部署在机器上的时候，存储都是搭建的RAID存储架构，其实这个RAID很多人以为非常的深奥，确实这个概念比较难以理解，而且说深了其实里面的技术含量很高，但是如果简单说一下，也是每个人都能理解的。

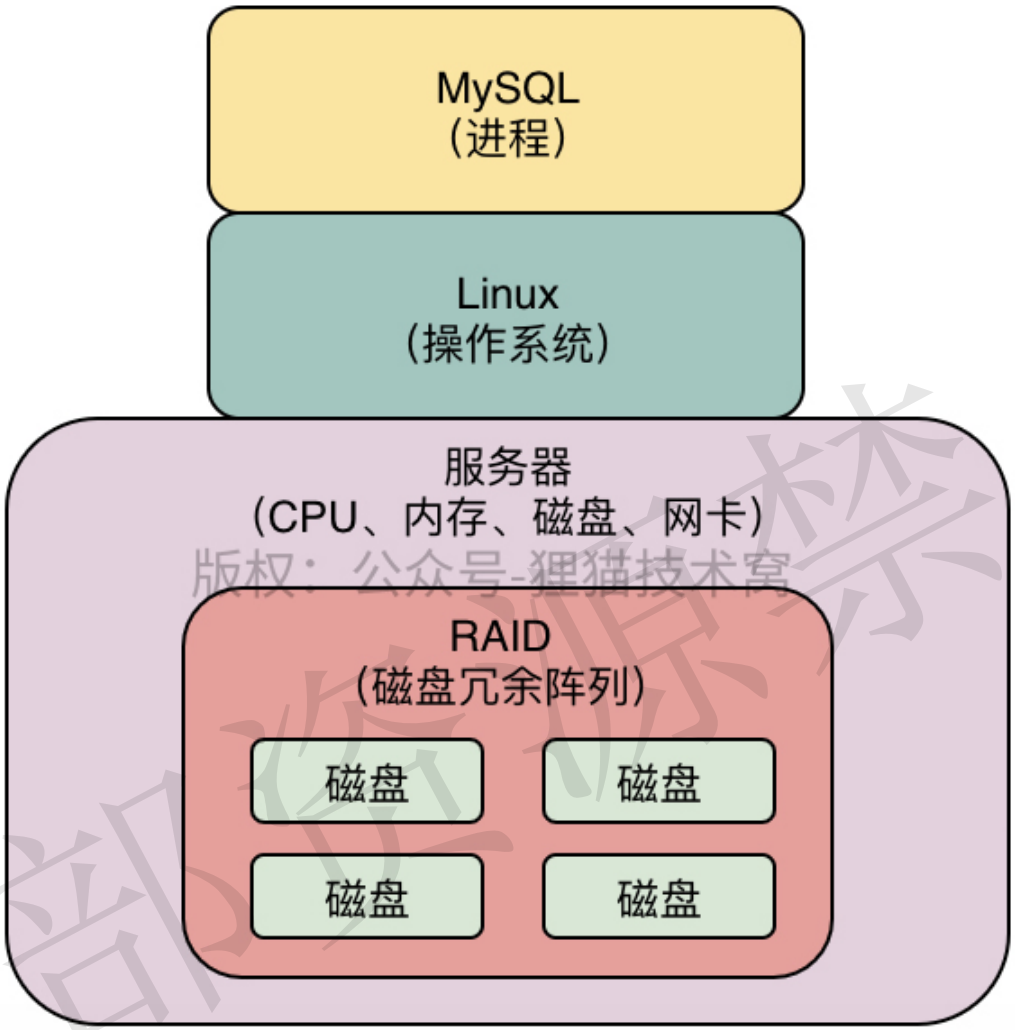
说白了，RAID就是一个磁盘冗余阵列，什么意思呢？

假设我们的服务器里的磁盘就一块，那万一 一块磁盘的容量不够怎么办？此时是不是就可以再搞几块磁盘出来放在服务器里

现在多搞了几块磁盘，机器里有很多块磁盘了，不好管理啊，怎么在多块磁盘上存放数据呢？

所以就是针对这个问题，在存储层面往往会在机器里搞多块磁盘，然后引入RAID这个技术，大致理解为用来管理机器里的多块磁盘的一种磁盘阵列技术！

有了他以后，你在往磁盘里读写数据的时候，他会告诉你应该在哪块磁盘上读写数据，如下图。

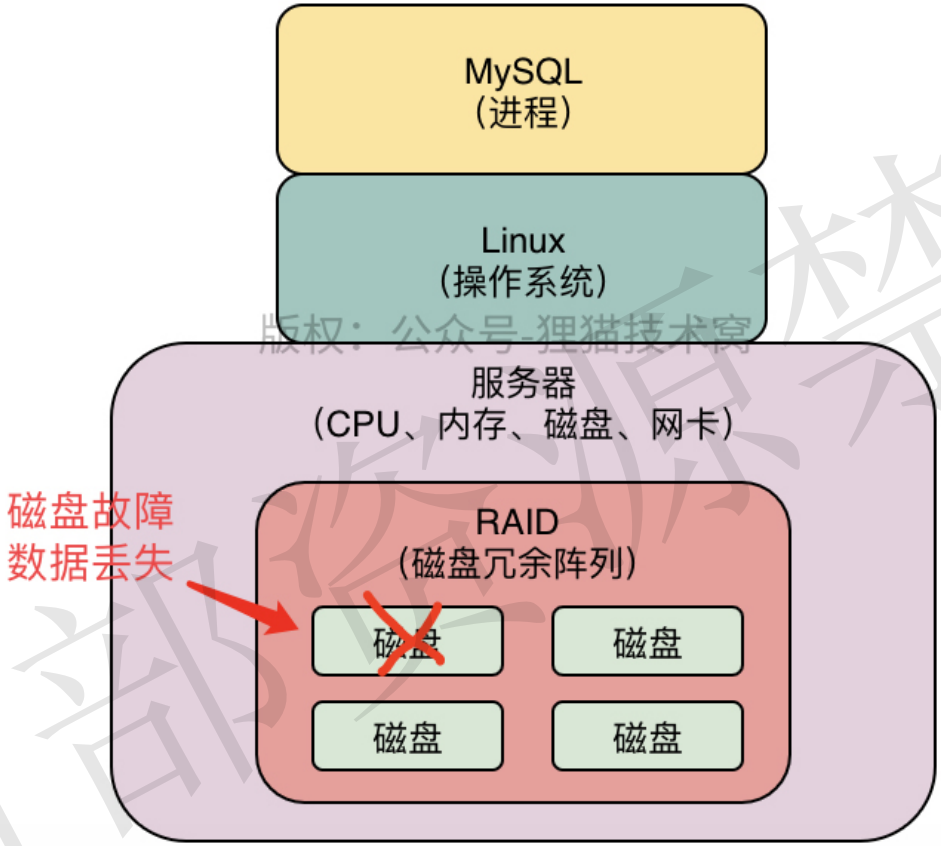


有了RAID这种多磁盘阵列技术之后，我们是不是就可以在一台服务器里加多块磁盘，扩大我们的磁盘存储空间了？

当我们往磁盘里写数据的时候，通过RAID技术可以帮助我们选择一块磁盘写入，在读取数据的时候，我们也知道从哪块磁盘去读取。

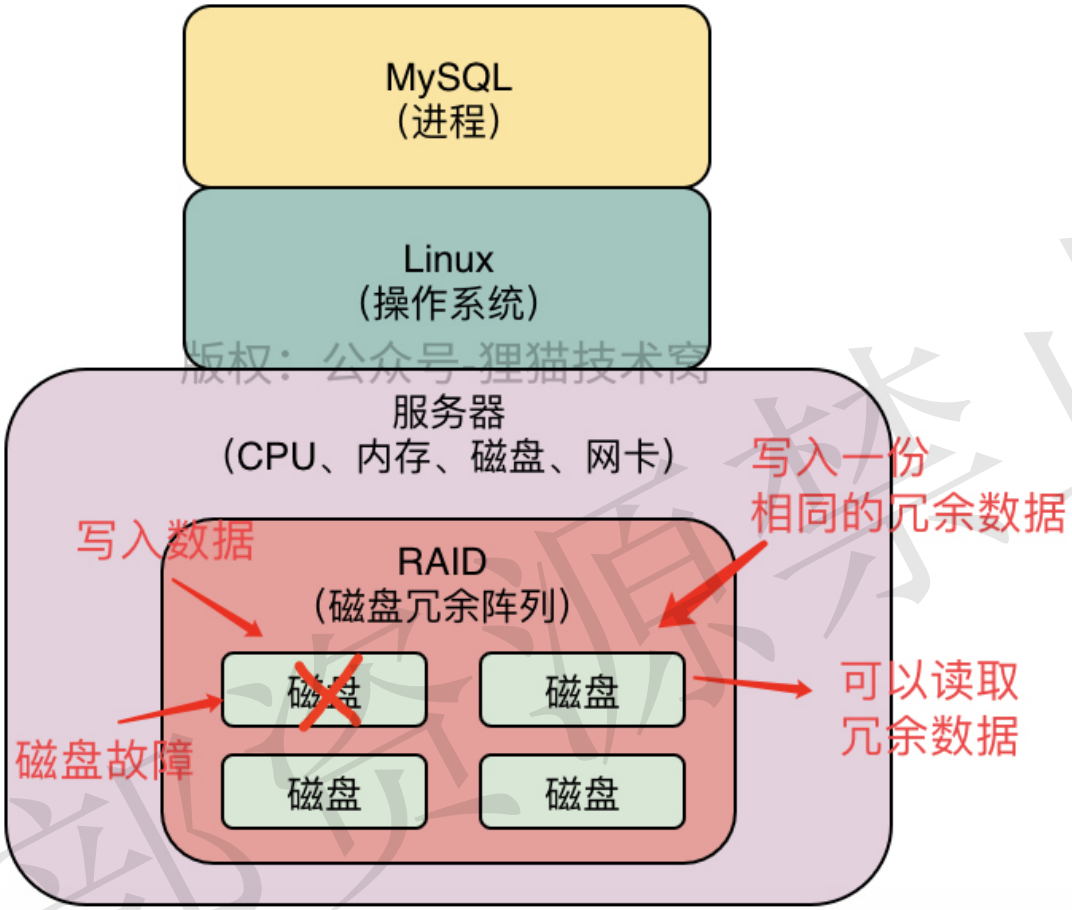
除此之外，RAID技术很重要的一个作用，就是他还可以实现**数据冗余机制**

所谓的数据冗余机制，就是如果你现在写入了一批数据在RAID中的一块磁盘上，然后这块磁盘现在坏了，无法读取了，那么岂不是你就丢失了一波数据？如下图所示



所以其实有的RAID磁盘冗余阵列技术里，是可以把你写入的同样一份数据，在两块磁盘上都写入的，这样可以让两块磁盘上的数据一样，作为冗余备份，然后当你一块磁盘坏掉的时候，可以从另外一块磁盘读取冗余数据出来，这一切

都是RAID技术自动帮你管理的，不需要你操心，如下图。



所以RAID技术实际上就是管理多块磁盘的一种磁盘阵列技术，他有软件层面的东西，也有硬件层买的东西，比如有RAID卡这种硬件设备。

具体来说，RAID还可以分成不同的技术方案，比如RAID 0、RAID 1、RAID 0+1、RAID2，等等，一直到RAID 10，很多种不同的多磁盘管理技术方案。

大家如果有兴趣的，可以自行去搜索对应的资料学习里面的技术细节，但是对于我们来说，这篇文章点到为止

大家只要了解一下RAID这种多磁盘冗余阵列技术的基本思想就可以了，我们毕竟不是专门讲解存储这块的。

对于存储的深入学习，主要也是一些运维工程师会去做的。

End

狸猫技术窝精品专栏及课程推荐：

- 《从零开始带你成为消息中间件实战高手》
- 《21天互联网Java进阶面试训练营》（分布式篇）
- 《互联网Java工程师面试突击》（第1季）
- 《互联网Java工程师面试突击》（第3季）
- 《从零开始带你成为JVM实战高手》

图文 36 生产经验：数据库服务器上的RAID存储架构的电池充放电原理

手机观看

409 人次阅读 2020-03-05 14:28:49

详情 评论

生产经验：数据库服务器上的RAID存储架构的电池充放电原理

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

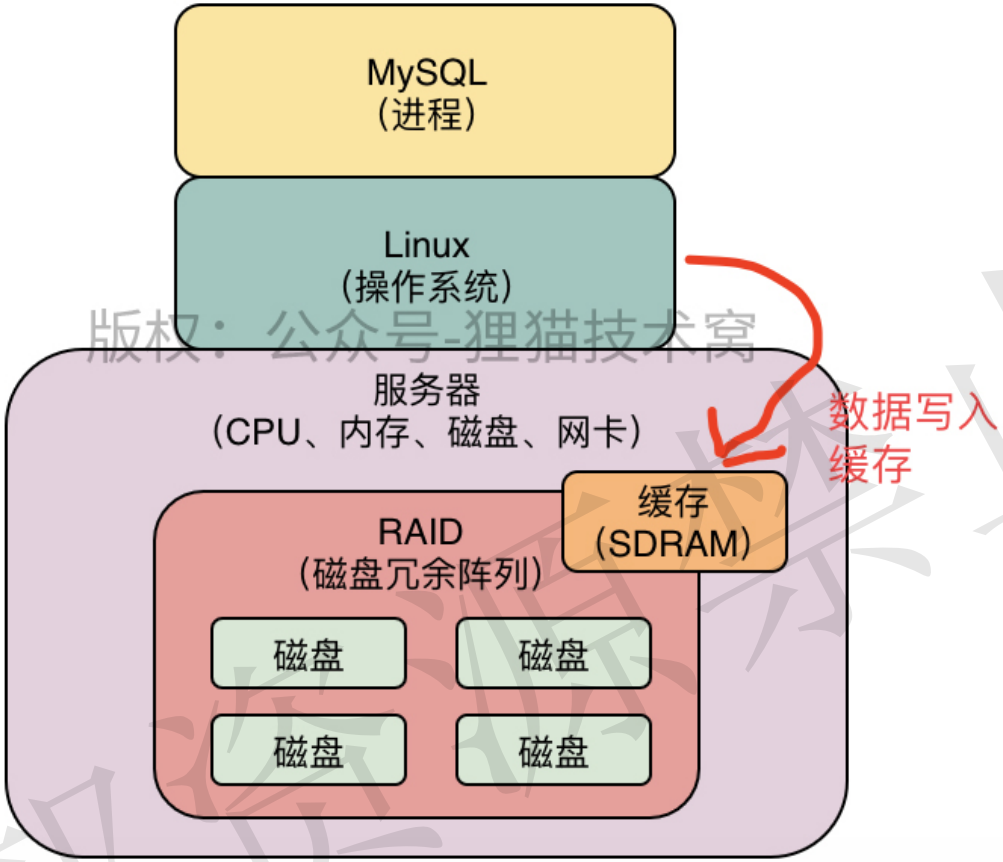
具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

上一篇文章我们初步给大家介绍了一下RAID多磁盘冗余阵列技术是什么东西，这一篇文章我们继续给大家讲解RAID存储架构的电池充放电原理，把这个理解了之后，我们下一篇文章就可以给大家讲一个真实的生产案例了。

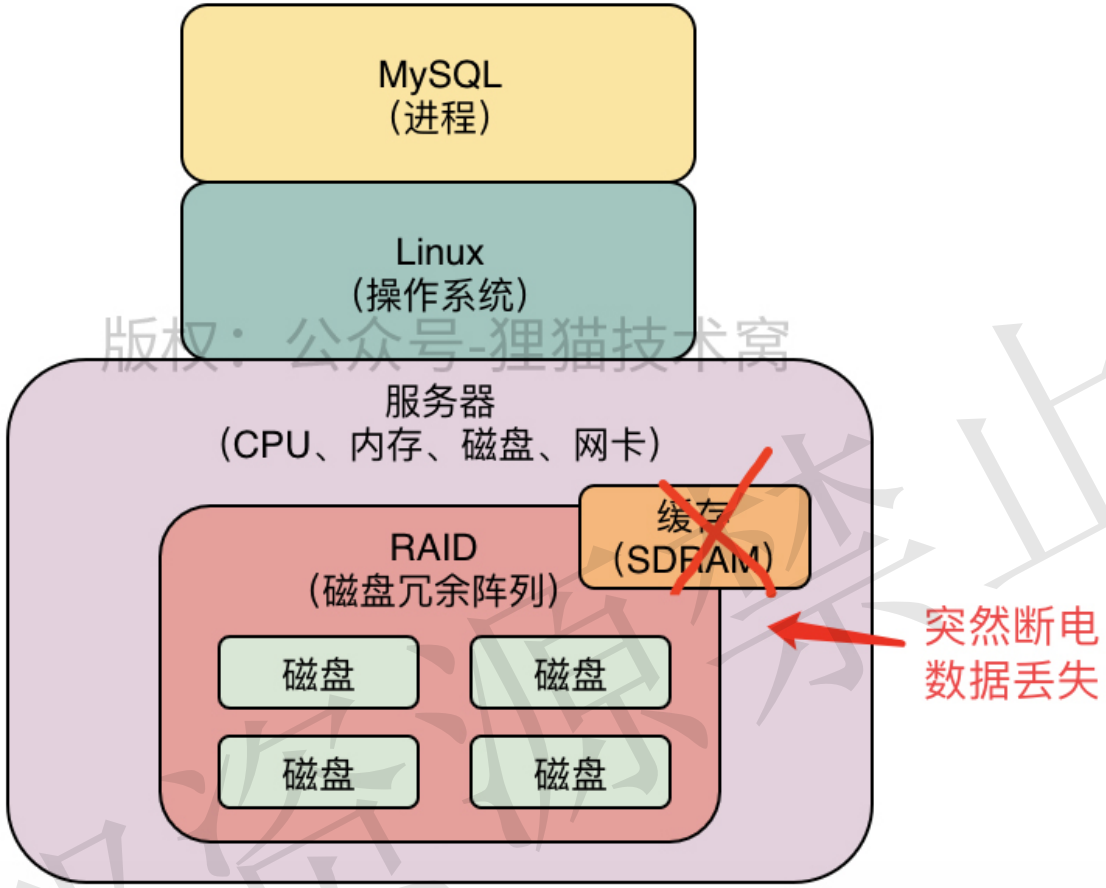
服务器使用多块磁盘组成的RAID阵列的时候，一般会有一个RAID卡，这个RAID卡是带有一个缓存的，这个缓存不是直接用我们的服务器的主内存的那种模式，他是一种跟内存类似的SDRAM，当然，你大致就认为他也是基于内存来存储的吧！

然后我们可以把RAID的缓存模式设置为write back，这样的话，所有写入到磁盘阵列的数据，先会缓存在RAID卡的缓存里，后续慢慢再写入到磁盘阵列里去，这种写缓冲机制，可以大幅度提升我们的数据库磁盘写的性能。

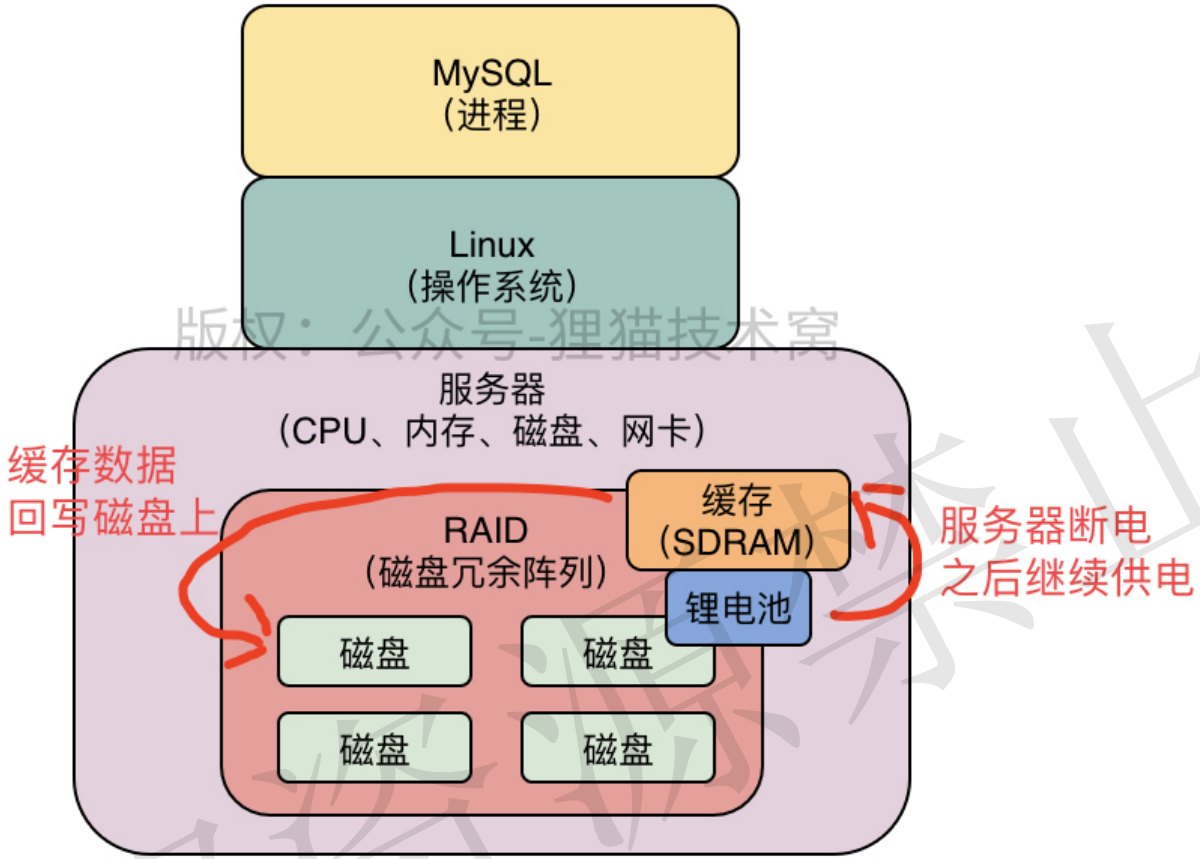
我们看下图，他说的就是这个RAID卡的缓存机制。



那么现在有一个问题来了，假设突然断电了，或者是服务器自己故障关闭了，那么是不是这个RAID卡的缓存里的数据会突然丢失？那你MySQL写入磁盘的数据不就没了吗？我们看下图。



所以正是因为如此，为了解决这个问题，RAID卡一般都配置有自己独立的锂电池或者是电容，如果服务器突然掉电了，无法接通电源了，RAID卡自己是基于锂电池来供电运行的，然后他会赶紧把缓存里的数据写入到阵列中的磁盘上去，如下图所示。



但是锂电池是存在性能衰减问题的，所以一般来说锂电池都是要配置定时充放电的，也就是说每隔30天~90天（不同的锂电池厂商是不一样的），就会自动对锂电池充放电一次，这可以延长锂电池的寿命和校准电池容量。

如果你要是不这么做的话，那么可能锂电池用着用着就会发现容量不够了，可能容纳的电量在你服务器掉电之后，都没法一次性把缓存里的数据写回磁盘上去，那就会导致数据丢失了！

所以在锂电池充放电的过程中，RAID的缓存级别会从write back变成write through，我们通过RAID写数据的时候，IO就直接写磁盘了，如果写内存的话，性能也就是0.1ms这个级别，但是直接写磁盘，就性能退化10倍到毫秒级了！

所以说，对于那些在生产环境的数据库部署使用了RAID多磁盘阵列存储技术的公司来说，通常都会开启RAID卡的缓存机制，但是此时就一定要注意这个RAID的锂电池自动充放电的问题，因为只要你用了RAID缓存机制，那么锂电池就必然会定时进行充放电去延长寿命，保证服务器掉电的时候可以把缓存数据写回磁盘，数据不会丢失。

所以这个时候一旦RAID锂电池自动充放电，往往会导致你的数据库服务器的RAID存储定期的性能出现几十倍的抖动，间接导致你的数据库每隔一段时间就会出现性能几十倍的抖动！

End

图文 37 案例实战：RAID锂电池充放电导致的MySQL数据库性能抖动的优化

手机观看

详情 评论

案例实战：RAID锂电池充放电导致的MySQL数据库性能抖动的优化

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

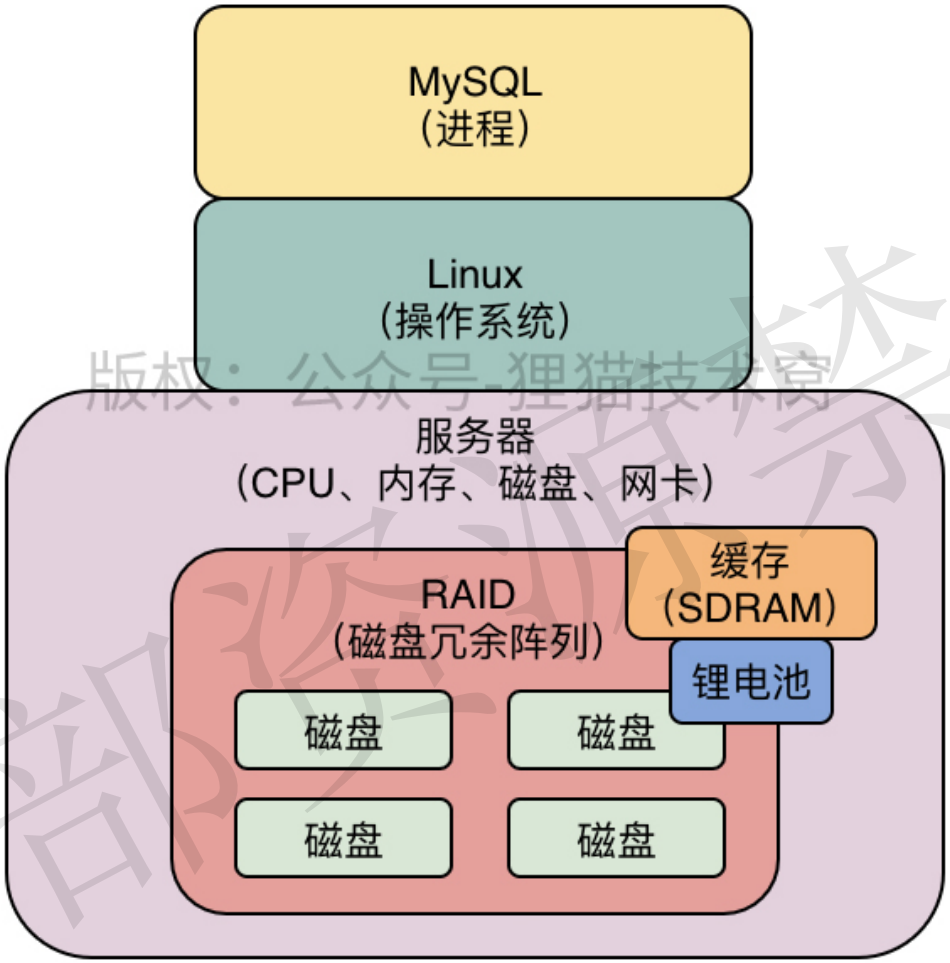
具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

前面经过了几天的生产经验的一些铺垫，包括MySQL磁盘读写的机制，Linux存储系统的原理，RAID磁盘阵列的介绍，RAID锂电池定时充放电的原理，今天终于可以切入真正的生产案例的讲解了！

其实只要大家理解了之前的内容，今天真正讲解案例的时候，你会发现理解起来是非常轻松的，几乎都不用画什么图，而且讲解起来也会非常的顺畅。

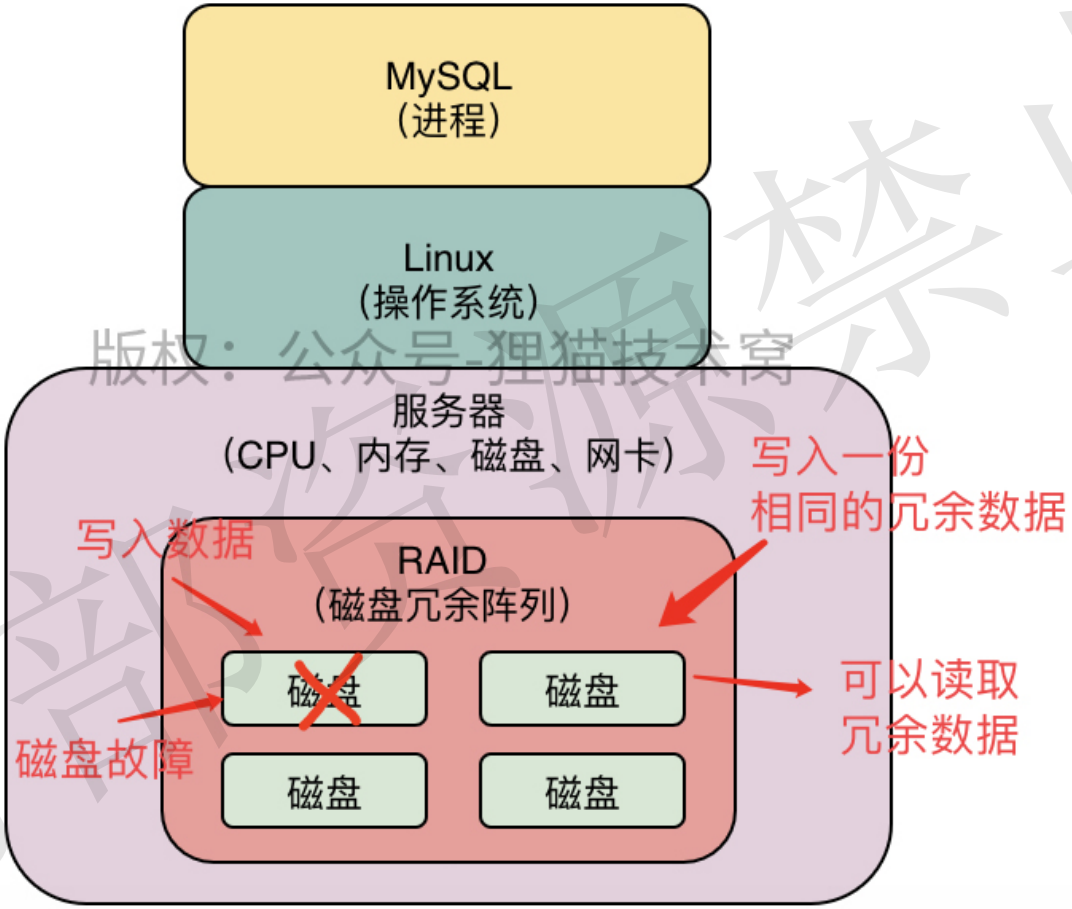
其实简单来说，就是曾经我们有一个非常核心的业务，他的数据库是部署在高配置服务器上的，磁盘就是用的RAID 10的阵列技术，用了6块磁盘组成了RAID 10磁盘阵列架构，具体RAID 10是什么，我们也简单的介绍一下。

其实RAID 0的意思，就是我们之前一幅图里画的，你有很多磁盘组成了一个阵列，然后你所有的数据是分散写入不同磁盘的，因为有多块磁盘，所以你的磁盘阵列的整体容量就很大，而且同时写入多块磁盘，让你的磁盘读写并发能力很强，如下图。

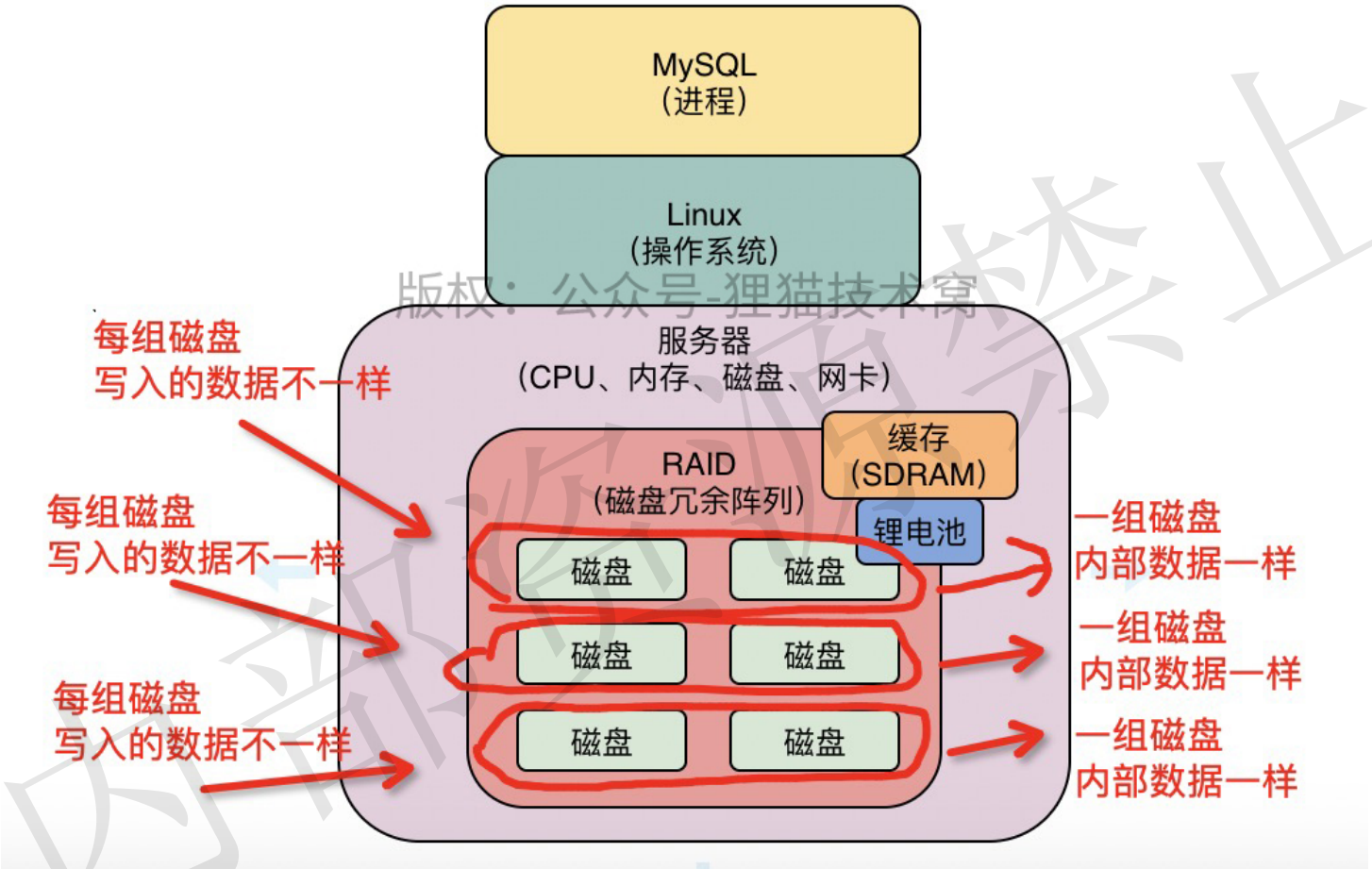


然后但是这种模式下，最大的问题就是万一你磁盘坏了一块，那么就会丢失一部分数据了！所以一般如果你要严格保证磁盘数据不丢失的话，就得用RAID 1，这个RAID 1的意思，就是两块磁盘为镜像关系，你写的所有数据，在两块磁盘上都有，形成了数据冗余，一块磁盘坏了，另外一块磁盘上还有数据。

一块磁盘如果压力很大，可以让读请求路由到另外一块磁盘上去，分担压力，反正他俩的数据都是冗余的，是一样的，如下图所示。



然后所谓的RAID 10，就是RAID 0 + RAID 1组合起来，就是说当时生产环境的服务器部署，我们有6块磁盘组成了一个RAID 10的阵列，那么其实就是每2块磁盘组成一个RAID 1互为镜像的架构，存放的数据是冗余一样的，一共有3组RAID 1，然后对于每一组RAID 1写入数据的时候，是用RAID 0的思路，就是不同组的磁盘的数据是不一样的，但是同一组内的两块磁盘的数据是冗余一致的，如下图。



所以对于这样的一个使用了RAID 10架构的服务器，他必然内部是有一个锂电池的，然后这个锂电池的厂商设定的默认是30天进行一次充放电，每次锂电池充放电就会导致RAID写入时不经过缓存，性能会急剧下降，所以我们发现线上

数据库每隔30天就会有一次剧烈性能抖动，数据库性能下降了10倍。

当时为了排查这个问题，我们使用linux命令查看了RAID硬件设备的日志，这个具体什么命令不说了，因为你用不同的厂商的RAID设备，这个命令实际上是不一样的，发现RAID就是每隔30天有一次充放电的日志，所以就是由于这个定期的充放电导致了线上数据库的性能定期抖动！

那么后续如何解决这个问题呢？对于RAID锂电池充放电问题导致的存储性能抖动，一般有**三种解决方案：**

给RAID卡把锂电池换成电容，电容是不用频繁充放电的，不会导致充放电的性能抖动，还有就是电容可以支持透明充放电，就是自动检查电量，自动进行充电，不会说在充放电的时候让写IO直接走磁盘，但是更换电容很麻烦，而且电容比较容易老化，这个其实一般不常用

手动充放电，这个比较常用，包括一些大家知道的顶尖互联网大厂的数据库服务器的RAID就是用了这个方案避免性能抖动，就是关闭RAID自动充放电，然后写一个脚本，脚本每隔一段时间自动在晚上凌晨的业务低峰时期，脚本手动触发充放电，这样可以避免业务高峰期的时候RAID自动充放电引起性能抖动

充放电的时候不要关闭write back，就是设置一下，锂电池充放电的时候不要把缓存级别从write back修改为write through，这个也是可以做到的，可以和第二个策略配合起来使用

这周通过一些生产经验的讲解以及一些底层技术的分析，同时结合一个真实的我们在大厂里的时候发现的一个数据库的性能抖动的案例，让大家能够把之前学到的数据库的理论知识以及底层技术，包括生产优化，都结合起来，希望大家能认真体会。

End

图文 38 案例实战：数据库无法连接故障的定位，Too many connections

手机观看

详情 评论

案例实战：数据库无法连接故障的定位，Too many connections

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

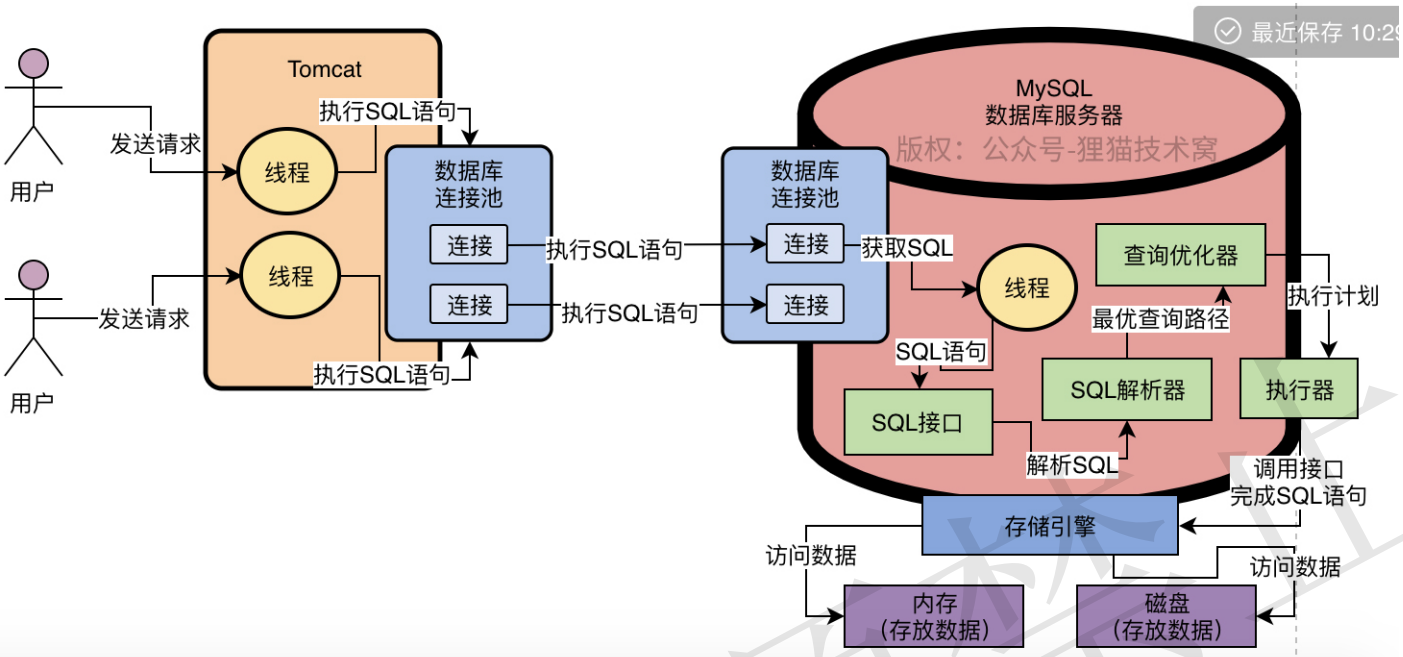
如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

今天要给大家分析另外一个真实的大家都经常会碰到的数据库生产故障，就是数据库无法连接的问题

大家会看到的异常信息往往是“ERROR 1040(HY000): Too many connections”，这个时候就是说数据库的连接池里已经有太多的连接了，不能再跟你建立新的连接了！

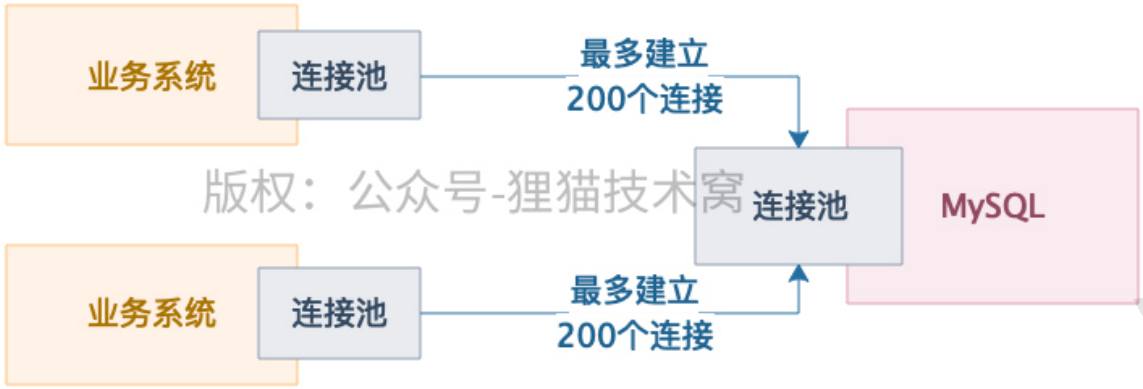
不知道大家是否还记得我们最早讲过的数据库的整体架构原理，数据库自己其实是有一个连接池的，你的每个系统部署在一台机器上的时候，你那台机器上部署的系统实例/服务实例自己也是有一个连接池的，你的系统每个连接Socket都会对应着数据库连接池里的一个连接Socket，这就是TCP网络连接，如下图所示。



所以当数据库告诉你Too many connections的时候，就是说他的连接池的连接已经满了，你业务系统不能跟他建立更多的连接了！

曾经在我们的一个生产案例中，数据库部署在64GB的大内存物理机上，机器配置各方面都很高，然后连接这台物理机的Java系统部署在2台机器上，Java系统设置的连接池的最大大小是200，也就是说每台机器上部署的Java系统，最多跟MySQL数据库建立200个连接，一共最多建立400个连接，我们看下图示意。

（另外给大家说明一点，最近狸猫技术窝的所有专栏都统一调整了画图工具，现在我们都使用下图的这种风格来进行绘图了）



但是这个时候如果MySQL报异常说Too many Connections，就说明目前MySQL甚至都无法建立400个网络连接？这也太少了吧！毕竟是高配置的数据库机器！

于是我们检查了一下MySQL的配置文件，my.cnf，里面有一个关键的参数是max_connections，就是MySQL能建立的最大连接数，设置的是800。

那奇怪了，明明设置了MySQL最多可以建立800个连接，为什么居然两台机器要建立400个连接都不行呢？

这个时候我们可以用命令行或者一些管理工具登录到MySQL去，可以执行下面的命令看一下：

show variables like 'max_connections'

此时你可以看到，当前MySQL仅仅只是建立了214个连接而已！

所以我们此时就可以想到，是不是MySQL根本不管我们设置的那个mac_connections，就是直接强行把最大连接数设置为214了？于是我们可以去检查一下MySQL的启动日志，可以看到如下的字样：

Could not increase number of max_open_files to more than mysqld (request: 65535)

Changed limits: max_connections: 214 (requested 2000)

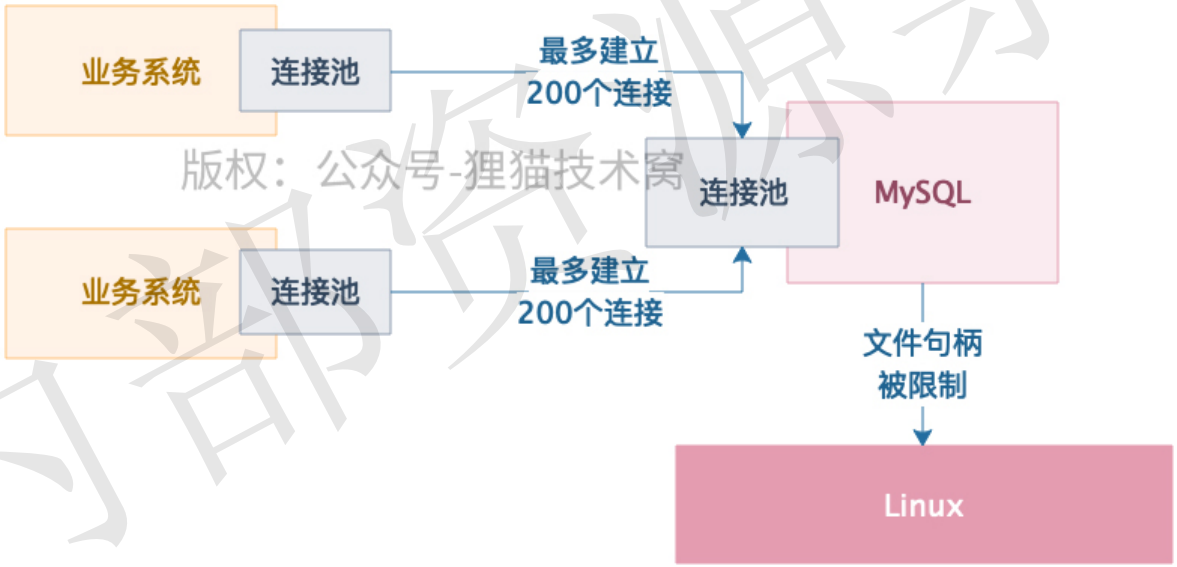
Changed limits: table_open_cache: 400 (requested 4096)

所以说，看看日志就很清楚了，MySQL发现自己无法设置max_connections为我们期望的800，只能强行限制为214了！

这是为什么呢？简单来说，就是因为底层的linux操作系统把进程可以打开的文件句柄数限制为了1024了，导致MySQL最大连接数是214！

可能有的人会疑惑说，为什么linux的文件句柄数量被限制了，MySQL最大连接数就被限制了呢？

其实这个问题你先不用操心了，因为这都是linux的知识，你现在只要知道有这么一件事儿就可以了，看下图的示意。



End

狸猫技术窝精品专栏及课程推荐：

- 《从零开始带你成为消息中间件实战高手》
- 《21天互联网Java进阶面试训练营》（分布式篇）
- 《互联网Java工程师面试突击》（第1季）
- 《互联网Java工程师面试突击》（第3季）
- 《从零开始带你成为JVM实战高手》

图文 # 39 案例实战：如何解决经典的Too many connections故障？背后原理是什么

手机观看

详情 评论

案例实战：如何解决经典的Too many connections故障？背后原理是什么

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

今天我们继续讲解昨天的那个案例背景，其实就是经典的Too many connections故障，他的核心就是linux的文件句柄限制，导致了MySQL的最大连接数被限制，那么今天来讲讲怎么解决这个问题。

其实核心就是一行命令：

ulimit -HSn 65535

然后就可以用如下命令检查最大文件句柄数是否被修改了

cat /etc/security/limits.conf

cat /etc/rc.local

如果都修改好之后，可以在MySQL的my.cnf里确保mac_connections参数也调整好了，然后可以重启服务器，然后重启MySQL，这样的话，linux的最大文件句柄就会生效了，MySQL的最大连接数也会生效了。

然后此时你再尝试业务系统去连接数据库，就没问题了！

另外再给大家解释一个问题，有人还是疑惑说，为什么linux的最大文件句柄限制为1024的时候，MySQL的最大连接数是214呢？

这个其实是MySQL源码内部写死的，他在源码中就是有一个计算公式，算下来就是如此罢了！

然后再给大家说一下，这个linux的ulimit命令是干嘛用的，其实说白了，linux的话是默认会限制你每个进程对机器资源的使用的，包括可以打开的文件句柄的限制，可以打开的子进程数的限制，网络缓存的限制，最大可以锁定的内存大小。

因为linux操作系统设计的初衷，就是要尽量避免你某个进程一下子耗尽机器上的所有资源，所以他默认都是会做限制的。

那么对于我们来说，常见的一个问题，其实就是文件句柄的限制。

因为如果linux限制你一个进程的文件句柄太少的话，那么就会导致我们没办法创建大量的网络连接，此时我们的系统进程就没法正常工作了

举个例子，比如MySQL运行的时候，其实就是linux上的一个进程，那么他其实是需要跟很多业务系统建立大量的连接的，结果你限制了他的最大文件句柄数量，那么他就不能建立太多连接了！

所以说，往往你在生产环境部署了一个系统，比如数据库系统、消息中间件系统、存储系统、缓存系统之后，都需要调整一下linux的一些内核参数，这个文件句柄的数量是一定要调整的，通常都得设置为65535

还有比如Kafka之类的消息中间件，在生产环境部署的时候，如果你不优化一些linux内核参数，会导致Kafka可能无法创建足够的线程，此时也是无法运行的。

所以我们平时可以用ulimit命令来设置每个进程被限制使用的资源量，用ulimit -a就可以看到进程被限制使用的各种资源的量

比如 core file size 代表的进程崩溃时候的转储文件的大小限制，max locked memory就是最大锁定内存大小，open files就是最大可以打开的文件句柄数量，max user processes就是最多可以拥有的子进程数量。

设置之后，我们要确保变更落地到/etc/security/limits.conf文件里，永久性的设置进程的资源限制

所以执行ulimit -HSn 65535命令后，要用如下命令检查一下是否落地到配置文件里去了。

cat /etc/security/limits.conf
cat /etc/rc.local

End

图文 40 重新回顾redo日志对于事务提交后，数据绝对不会丢失的意义

手机观看

详情 评论

重新回顾redo日志对于事务提交后，数据绝对不会丢失的意义

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

之前我们在给大家介绍了大量的MySQL底层原理知识之后，理论结合实践，给大家讲解了两个真实的生产环境的数据库优化案例，一个是数据库所在服务器的RAID存储系统的锂电池充放电导致的性能抖动问题，一个是数据库底层的linux操作系统的文件句柄限制导致的无法连接问题，相信大家在学习了理论知识之后，再来看这些真实的实战案例，会有不错的感觉。

那么接着在学习了两个实战案例之后，我们就要继续进行深入底层的MySQL原理剖析了，我们之前都知道，在我们执行增删改操作的时候，首先会在Buffer Pool中更新缓存页，那么缓存页和底层的物理磁盘上的数据页的原理，之前都已经详细讲解过了

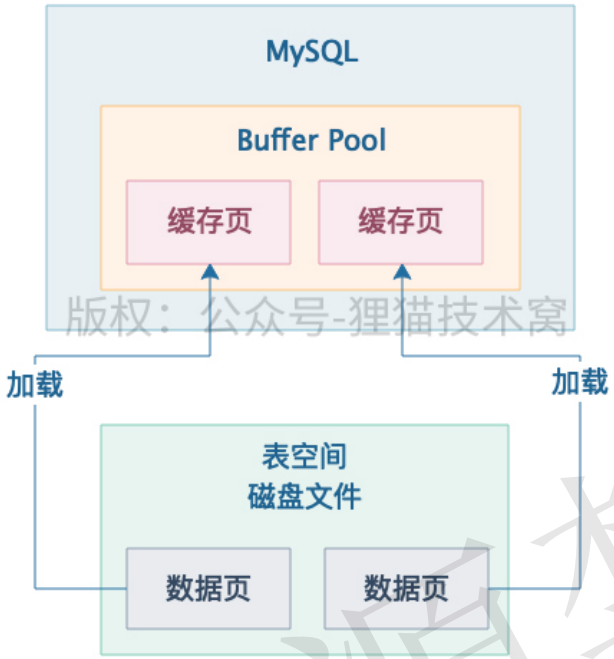
接着我们都知道，在更新完Buffer Pool中的缓存页之后，必须要写一条redo log，这样才能记录下来我们对数据库做的修改。

redo log可以保证我们事务提交之后，如果事务中的增删改SQL语句更新的缓存页还没刷到磁盘上去，此时MySQL宕机了，那么MySQL重启过后，就可以把redo log重做一遍，恢复出来事务当时更新的缓存页，然后再把缓存页刷到磁盘就可以了

redo log本质是保证事务提交之后，修改的数据绝对不会丢失的。

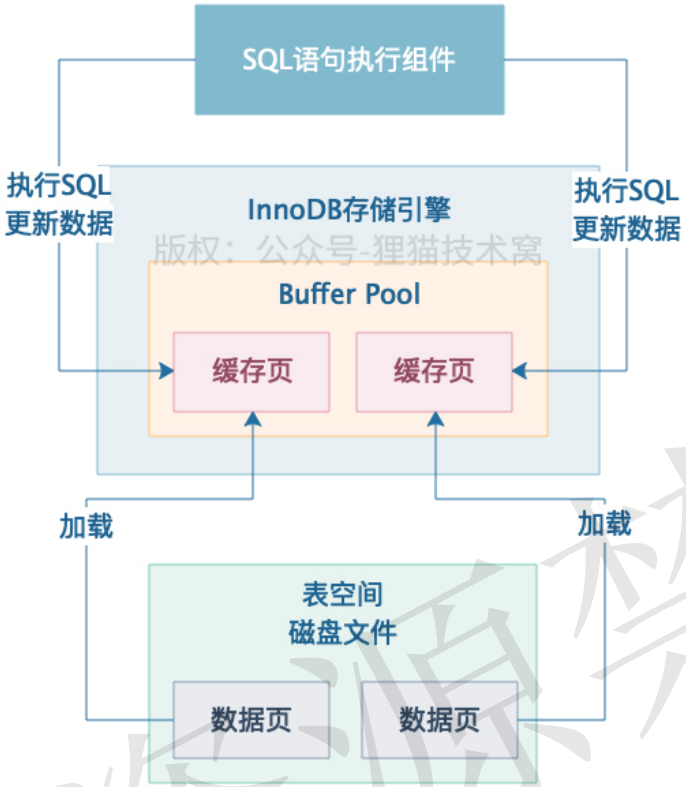
所以接下来一段时间我们会深入研究redo log的底层实现原理，今天我们就承上启下，给大家简单回顾一下redo log这个机制存在的意义。

首先我们都知道，执行增删改SQL语句的时候，都是针对一个表中的某些数据去执行的，此时的话，首先必须找到这个表对应的表空间，然后找到表空间对应的磁盘文件，接着从磁盘文件里把你要更新的那批数据所在的数据页从磁盘读取出来，放到Buffer Pool的缓存页里去，如下图所示。

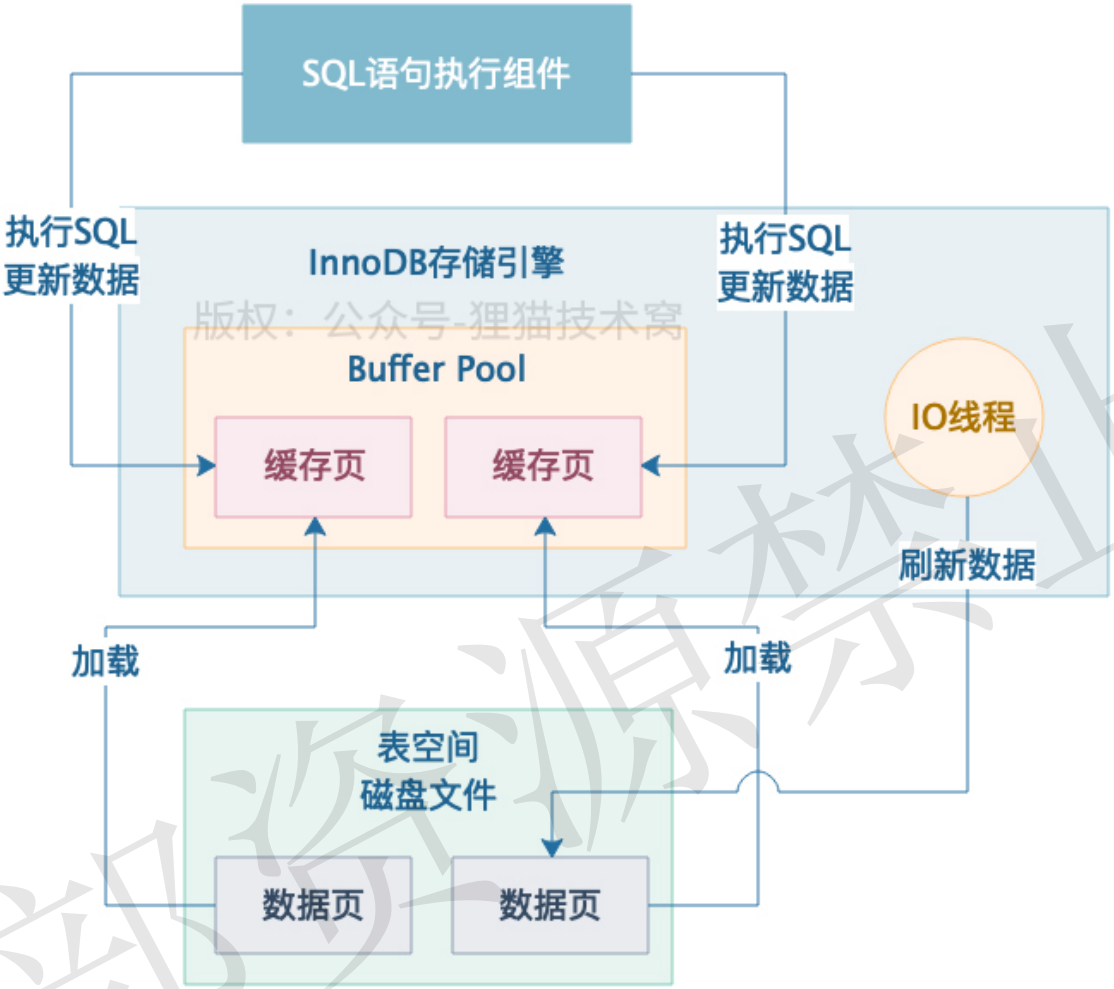


接着实际上你的增删改SQL语句就会针对Buffer Pool中的缓存页去执行你的更新逻辑，比如插入一行数据，或者更新一行数据，或者是删除一行数据。

当然，删除数据的逻辑我们还没讲，后续很快就要讲到了。至于说数据页和数据行的格式，就不用我多说了，其实都是MySQL自己定义的，之前都讲过了，大家现在对这些都知道了，如下图。



那么学习过之前的Buffer Pool底层原理之后都知道，其实你更新缓存页的时候，会更新free链表、flush链表、lru链表，然后有专门的后台IO线程，不定时的根据flush链表、lru链表，会把你更新过的缓存页刷新回磁盘文件的数据页里去，如下图所示。

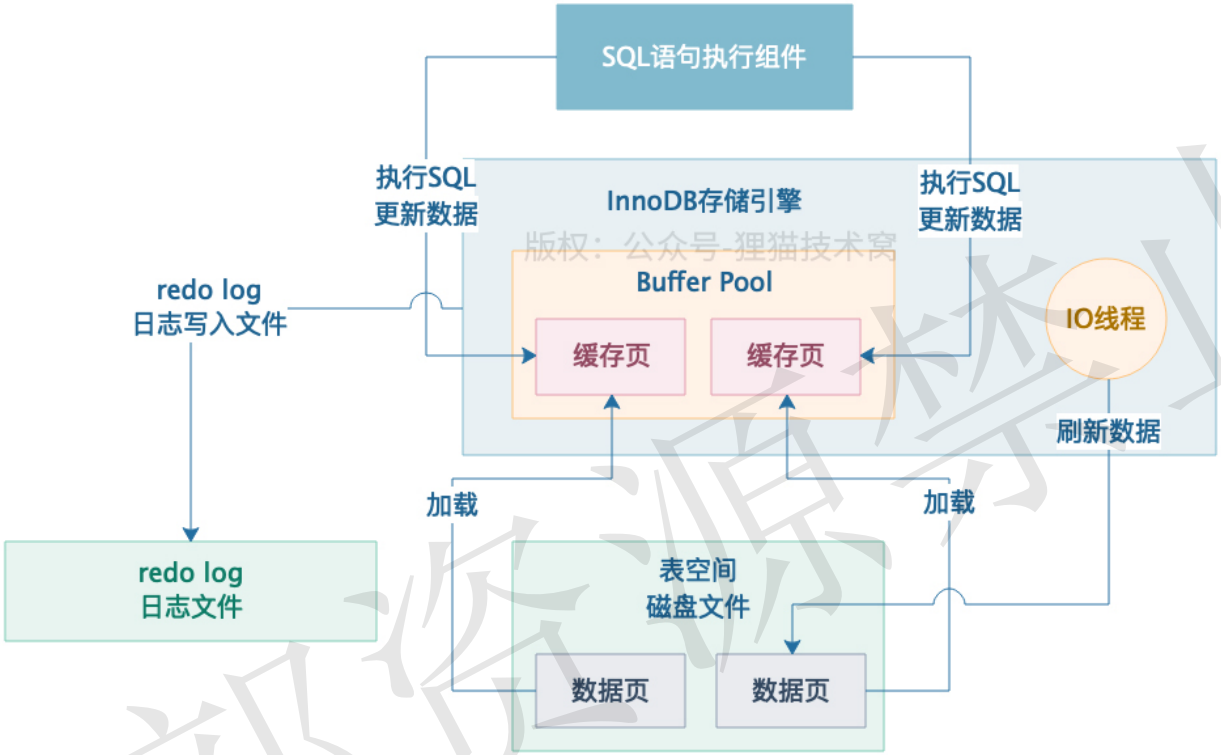


所以大家都知道这个机制里最大的漏洞就在于，万一你一个事务里有增删改SQL更新了缓存页，然后事务提交了，结果万一你还没来得及让IO线程把缓存页刷新到磁盘文件里，此时MySQL宕机了，然后内存数据丢失，你事务更新的数据就丢失了！

但是也不可能每次你事务一提交，就把你事务更新的缓存页都刷新回磁盘文件里去，因为大家之前也都知道，缓存页刷新到磁盘文件里，是随机磁盘读写，性能是相当的差！这会导致你数据库性能和并发能力都很弱的！

所以此时才会引入一个redo log机制，这个机制就是说，你提交事务的时候，绝对是保证把你对缓存页做的修改以日志的形式，写入到redo log日志文件里去的

这种日志大致的格式如下：对表空间XX中的数据页XX中的偏移量为XXXX的地方更新了数据XXX。如下图所示



只要你事务提交的时候保证你做的修改以日志形式写入redo log日志，那么哪怕你此时突然宕机了，也没关系！

因为你MySQL重启之后，把你之前事务更新过做的修改根据redo log在Buffer Pool里重做一遍就可以了，就可以恢复出来当时你事务对缓存页做的修改，然后找时机再把缓存页刷入磁盘文件里去。

那么有人会问了，你事务提交的时候把修改过的缓存页都刷入磁盘，跟你事务提交的时候把你做的修改的redo log都写入日志文件，他们不都是写磁盘么？差别在哪里？

这是本文一个关键的问题。

实际上，如果你把修改过的缓存页都刷入磁盘，这首先缓存页一个就是16kb，数据比较大，刷入磁盘比较耗时，而且你可能就修改了缓存页里的几个字节的数据，难道也把完整的缓存页刷入磁盘吗？

而且你缓存页刷入磁盘是随机写磁盘，性能是很差的，因为他一个缓存页对应的位置可能在磁盘文件的一个随机位置，比如偏移量为45336这个地方。

但是如果是写redo log，第一个一行redo log可能就占据几十个字节，就包含表空间好、数据页号、磁盘文件偏移量、更新值，这个写入磁盘速度很快。

此外，redo log写日志，是顺序写入磁盘文件，每次都是追加到磁盘文件末尾去，速度也是很快的。

所以你提交事务的时候，用redo log的形式记录下来你做的修改，性能会远远超过刷缓存页的方式，这也可以让你的数据库的并发能力更强。

End

小鹅通提供技术支持

图文 41 在Buffer Pool执行完增删改之后，写入日志文件的redo log长什么样？ 手机观看

376 人次阅读 2020-03-16 10:32:51

详情 评论

在Buffer Pool执行完增删改之后，写入日志文件的redo log长什么样？

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

昨天我们简单给大家回顾了一下在数据库里执行增删改操作的时候，redo log是用来干什么的，为什么需要这个东西，如果没有他会怎么样，有了他之后又能有什么样的效果，想必大家现在都对redo log这个东西有一定的理解了。

那么接下来我们就要深入研究一下redo log的一些技术细节了，今天就来看看写入磁盘上的日志文件的redo log，大致长个什么样，里面都包含一些什么东西。

之前略微给大家提到过，就是redo log里本质上记录的就是在对某个表空间的某个数据页的某个偏移量的地方修改了几个字节的值，具体修改的值是什么，他里面需要记录的就是**表空间号+数据页号+偏移量+修改几个字节的值+具体的值**

所以根据你修改了数据页里的几个字节的值，redo log就划分为了不同的类型，MLOG_1BYTE类型的日志指的就是修改了1个字节的值，MLOG_2BYTE类型的日志指的就是修改了2个字节的值，以此类推，还有修改了4个字节的值的日志类型，修改了8个字节的值的日志类型。

当然，如果你要是一下子修改了一大串的值，类型就是MLOG_WRITE_STRING，就是代表你一下子在那个数据页的某个偏移量的位置插入或者修改了一大串的值。

所以其实一条redo log看起来大致的结构如下所示：

日志类型（就是类似MLOG_1BYTE之类的），表空间ID，数据页号，数据页中的偏移量，具体修改的数据

大致就是一条redo log中依次排列上述的一些东西，这条redo log表达的语义就很明确了，他的类型是什么，类型就告诉了你他这次增删改操作修改了多少字节的数据；

然后在哪个表空间里操作的，这个就是跟你SQL在哪个表里执行的是对应的；接着就是在这个表空间的哪个数据页里执行的，在数据页的哪个偏移量开始执行的，具体更新的数据是哪些呢。

有了上述信息，就可以精准完美的还原出来一次数据增删改操作做的变动了。

只不过如果是MLOG_WRITE_STRING类型的日志，因为不知道具体修改了多少字节的数据，所以其实会多一个修改数据长度，就告诉你他这次修改了多少字节的数据，如下所示他的格式：

日志类型（就是类似MLOG_1BYTE之类的），表空间ID，数据页号，数据页中的偏移量，修改数据长度，具体修改的数据

因此今天就简单给大家讲解一下redo log的日志的格式，其实没大家想的那么复杂，当然如果往深了说，那可能也比你想的复杂很多，比如redo log日志里面可能会记录你更新了哪些索引之类的，那就复杂了去了，但是这些东西就等我们讲到索引那块的时候再说好了！

现在大家对redo log日志的格式了解到这个程度其实就可以了，你脑子里应该更加清晰了一些，就是执行增删改的时候，在Buffer Pool里通过复杂的缓存页机制完成更新，然后就会以今天讲解的这种格式写入一条redo log日志记录本次修改。

End

狸猫技术窝精品专栏及课程推荐：

- 《从零开始带你成为消息中间件实战高手》
- 《21天互联网Java进阶面试训练营》（分布式篇）
- 《互联网Java工程师面试突击》（第1季）
- 《互联网Java工程师面试突击》（第3季）
- 《从零开始带你成为JVM实战高手》

图文 42 redo log是直接一条一条写入文件的吗？非也，揭秘redo log block！ 手机观看

193 人次阅读 2020-03-19 10:48:24

详情 评论

redo log是直接一条一条写入文件的吗？非也，揭秘redo log block！

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

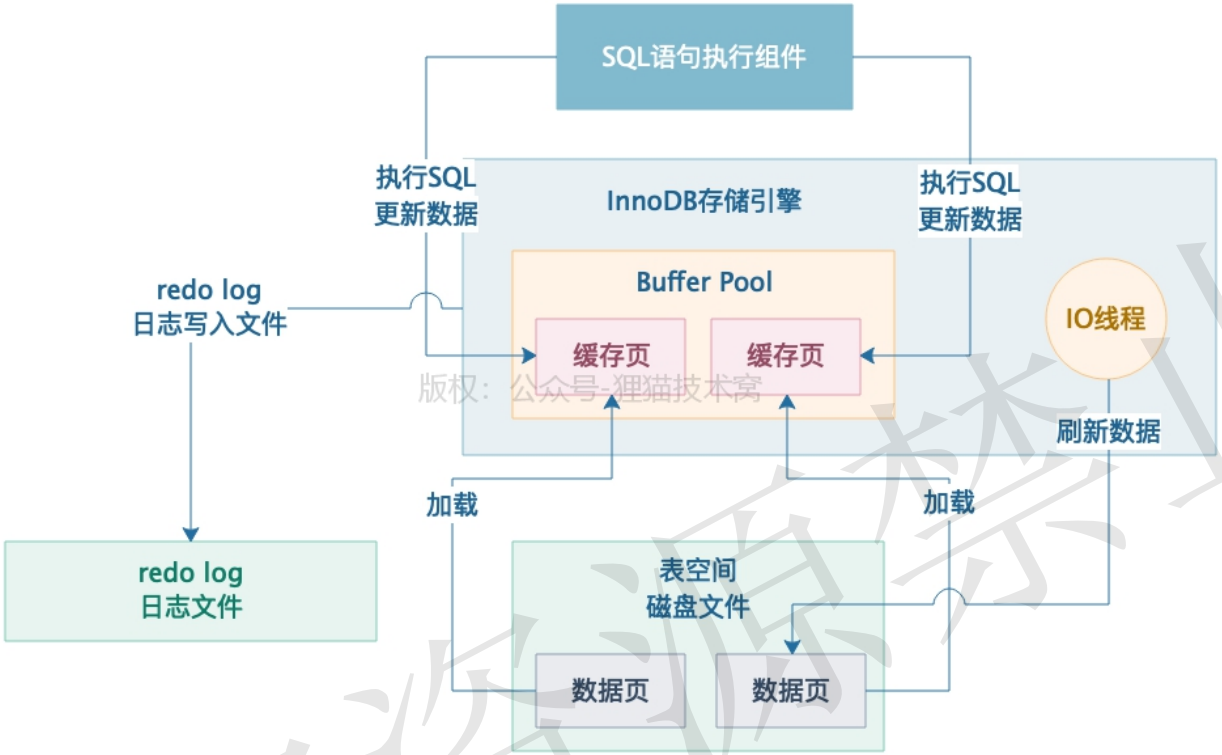
之前我们已经给大家讲解了redo log自己的一些基本的结构，今天我们就来讲解下一个问题，就是redo log是一条一条的直接就往磁盘文件里写入吗？

可能有一些朋友会认为就是如此简单粗暴的往磁盘文件里写，但其实并没那么简单！

接下来几天我们就会揭秘一下这个redo log写磁盘的过程。

首先大家看下面的图，学习到现在，我想任何一个朋友一看下面的图就知道是怎么回事了

平时我们执行CRUD的时候，从磁盘加载数据页到buffer pool的缓存页里去，然后对缓存页执行增删改，同时还会写redo log到日志文件里去，后续不定时把缓存页刷回磁盘文件里去，大概就是这个原理，如下图所示：



那么上次我们也介绍了一下每一条redo log长什么样子，说白了，他就是记录了：

表空间号+数据页号+数据页内偏移量+修改了几个字节的数据+实际修改数据

就是简简单单这么一条日志罢了

所以大家可以想一下，redo log就是按照上述格式，一条一条的直接就写入到磁盘上的日志文件里去了吗？

显然不是的!

其实MySQL内有另外一个数据结构，叫做**redo log block**，大概你可以理解为，平时我们的数据不是存放在数据页了的么，用一页一页的数据页来存放数据。

那么对于redo log也不是单行单行的写入日志文件的，他是用一个redo log block来存放多个单行日志的。

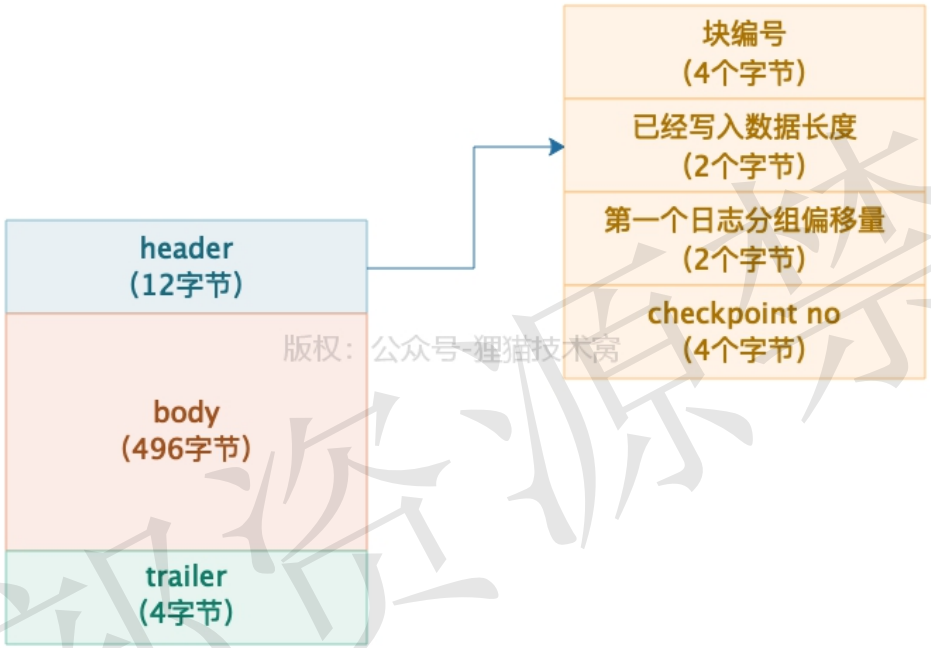
一个redo log block是512字节，这个redo log block的512字节分为3个部分，一个是12字节的header块头，一个是496字节的body块体，一个是4字节的trailer块尾

如下图所示

在这里面，12字节的header头又分为了4个部分。

包括4个字节的block no，就是块唯一编号；
2个字节的data length，就是block里写入了多少字节数据；
2个字节的first record group。这个是说每个事务都会有多个redo log，是一个redo log group，即一组redo log。那么在这个block里的第一组redo log的偏移量，就是这2个字节存储的；
4个字节的checkpoint on

我们看下图，这个header可以进行进一步的区分。

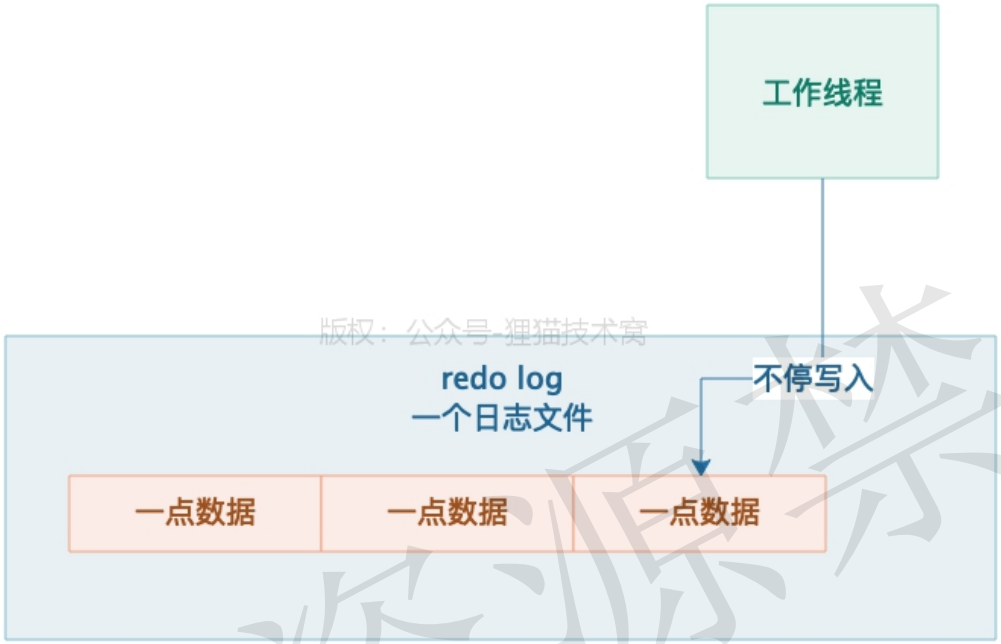


所以我们看到上图就知道，其实对于我们的redo log而言，他确实是不停的追加写入到redo log磁盘文件里去的，但是其实每一个redo log都是写入到文件里的一个redo log block里去的，一个block最多放496自己的redo log日志。

此时可能有人会有疑问了，到底一个一个的redo log block在日志文件里是怎么存在的？那么一条一条的redo log又是如何写入日志文件里的redo log block里去的呢？估计很多人都很奇怪这个问题。

所以我们接下来就给大家解答这个问题。

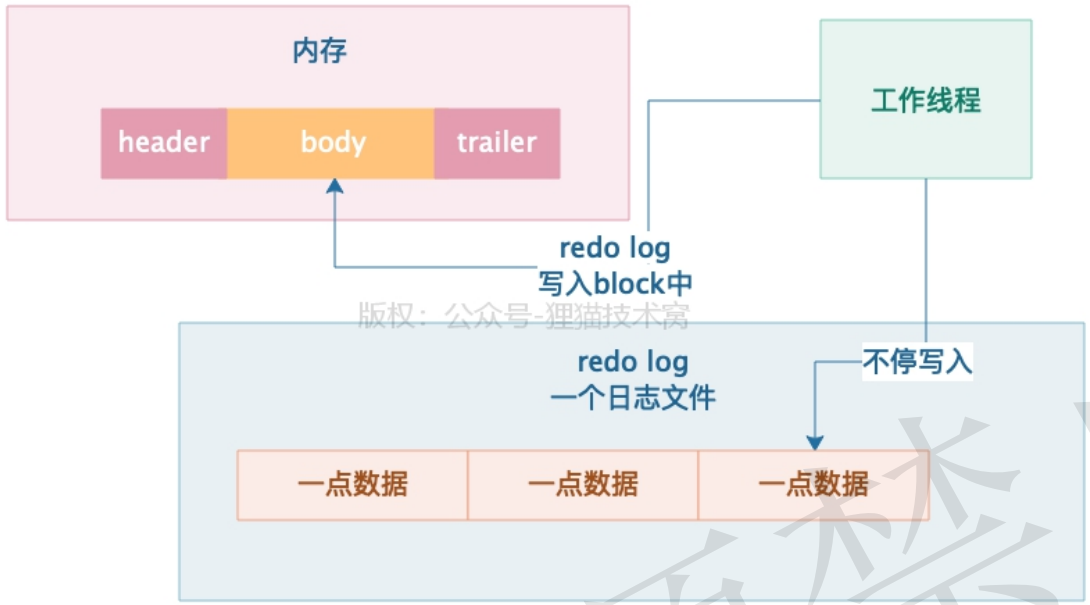
大家先想一下，假设你有一个redo log日志文件，平时我们往里面写数据，你大致可以认为是从第一行开始，从左往右写，可能会有很多行，比如下面那样子，你看看是不是你理解的那样？



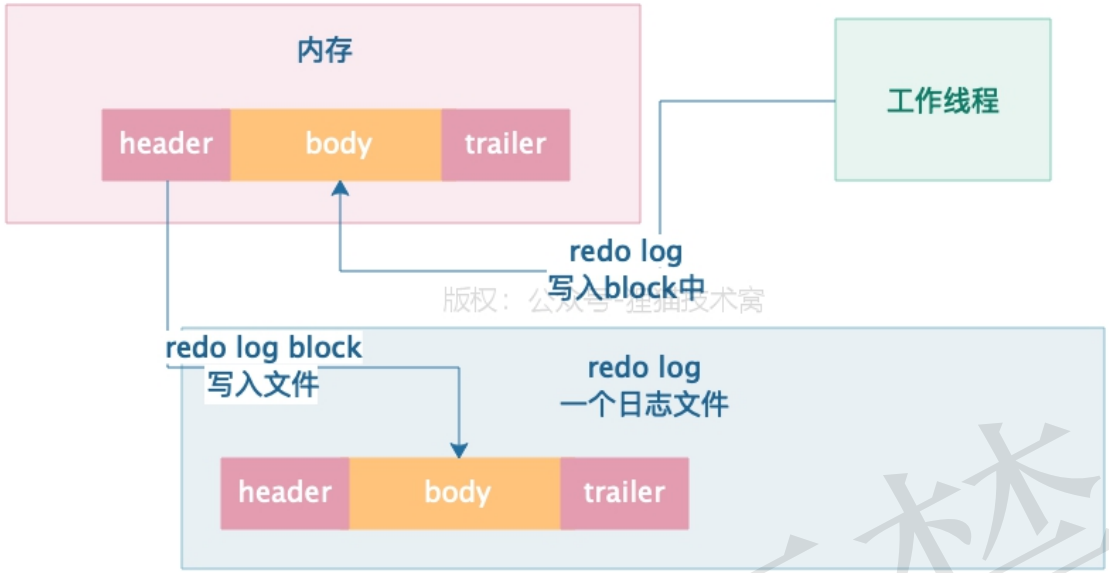
好，那么所以现在既然如此，假设你要写第一个redo log了，是不是应该起码是先在内存里把这个redo log给弄到一个redo log block数据结构里去？

然后似乎你应该是等内存里的一个redo log block的512字节都满了，再一次性把这个redo log block写入磁盘文件？

如下图所示



然后其实按照我们所说的，一个redo log block就是512字节，那么是不是真正写入的时候，把这个redo log block的512字节的数据，就写入到redo log文件里去就可以了？那么redo log文件里就多了一个block，如下图所示。

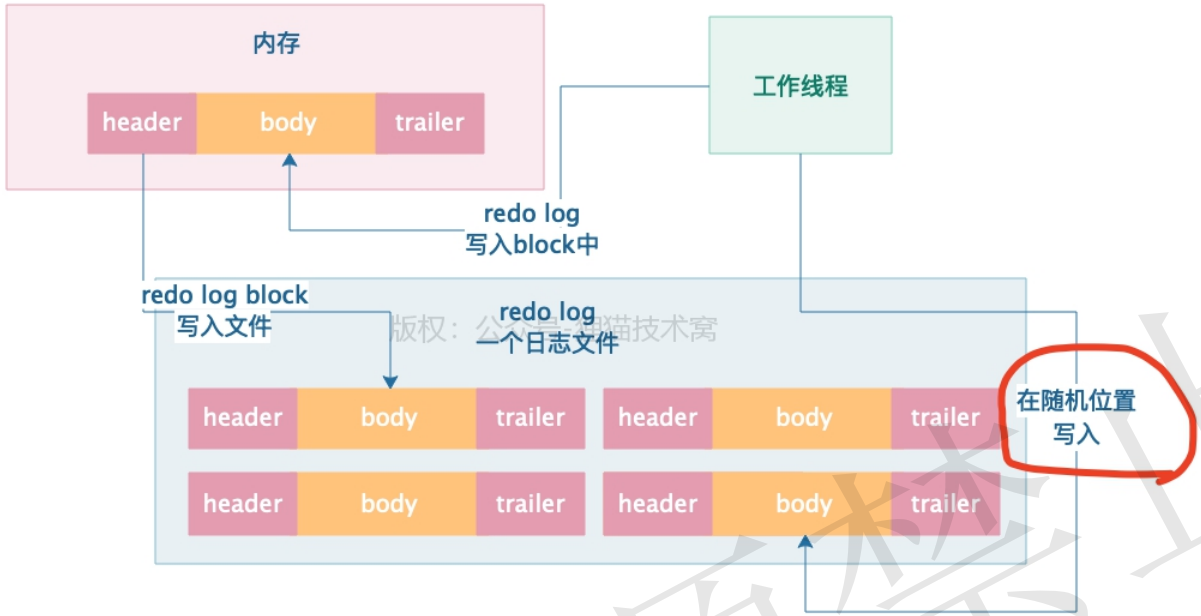


所以大家看到上图演示之后，对于这个所谓的redo log和redo log block的关系，以及redo log block如何进入日志文件，日志文件里是如何存放一个又一个的redo log block的，应该都很清楚了！

其实有一定开发经验的朋友都知道，写文件的时候，可以按照字节，一个字节一个字节的写入的，文件里存放的东西就是很多很多字节，依次排开，然后其中可能512个字节组合起来，就固定代表了一个redo log block。

这其实就是任何一个中间件系统，数据库系统，底层依赖磁盘文件存储数据的一个共同的原理，所以大家也不用把这个复杂数据写入磁盘文件想象的太复杂了。

那么如果依次在磁盘文件里的末尾追加不停的写字节数据，就是磁盘顺序写；但是假设现在磁盘文件里已经有很多很多的redo log block了，此时要在磁盘里某个随机位置找到一个redo log block去修改他里面几个字节的数据，这就是磁盘随机写，看下图。



好了，今天把redo log block的数据结构和他与磁盘文件的关系都讲的很清楚了，明天我们继续讲解redo log buffer，就是redo log是如何通过一个内存缓冲数据结构之后，再进入到磁盘文件的！

End

狸猫技术窝精品专栏及课程推荐：

- 《从零开始带你成为消息中间件实战高手》
- 《21天互联网Java进阶面试训练营》（分布式篇）
- 《互联网Java工程师面试突击》（第1季）

《互联网Java工程师面试突击》（第3季）
《从零开始带你成为JVM实战高手》

图文 43 直接强行把redo log写入磁盘？非也，揭秘redo log buffer！

手机观看

详情 评论

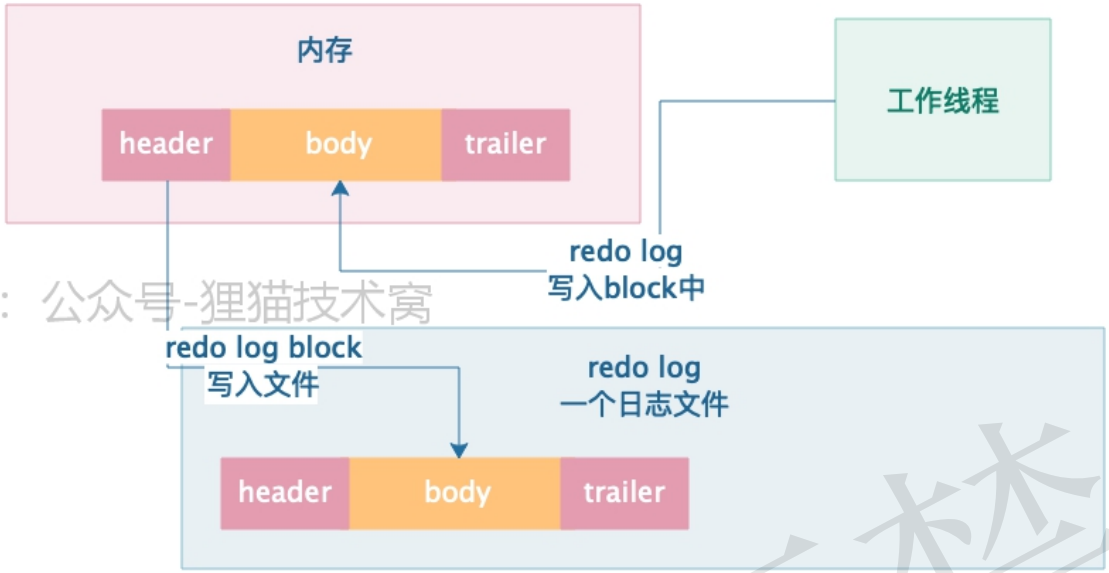
直接强行把redo log写入磁盘？非也，揭秘redo log buffer！

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：**《MySQL专栏付费用户如何加群》**（购买后可见）

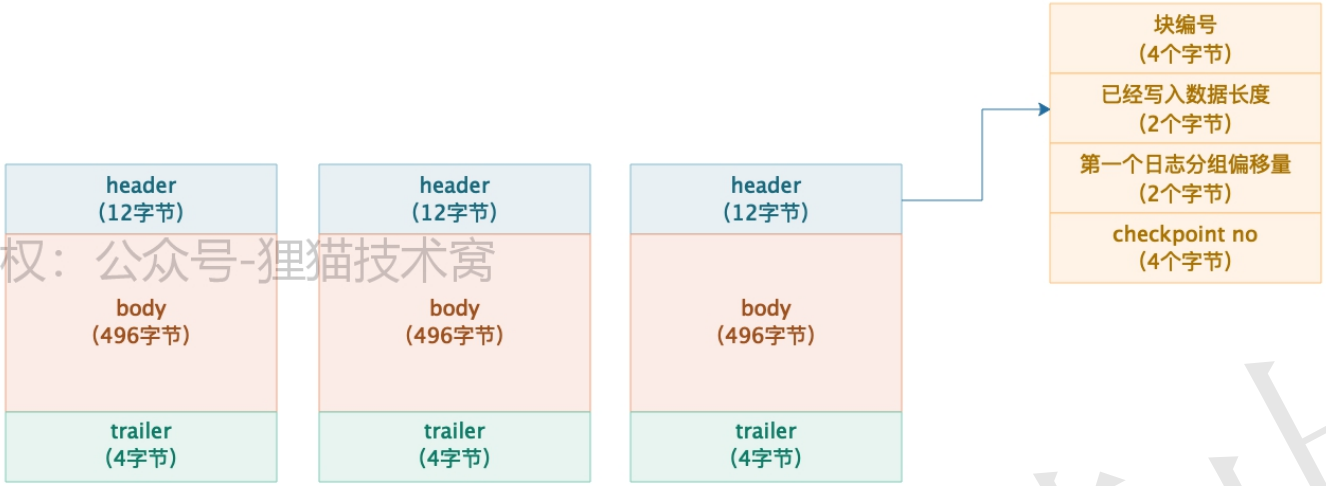
上一讲我们给大家说了一下redo log block这个概念，大家现在都知道平时我们执行完增删改之后，要写入磁盘的redo log，其实应该是先进入到redo log block这个数据结构里去的，然后再进入到磁盘文件里，如下图所示。



那么今天我们就来讲讲，这个redo log到底是如何通过内存缓冲之后，再进入磁盘文件里去的，这就涉及到了一个新的组件，redo log buffer，他就是MySQL专门设计了用来缓冲redo log写入的。

这个redo log buffer其实就是MySQL在启动的时候，就跟操作系统申请的一块连续内存空间，大概可以认为相当于是buffer pool吧。那个buffer pool是申请之后划分了N多个空的缓存页和一些链表结构，让你把磁盘上的数据页加载到内存里来的。

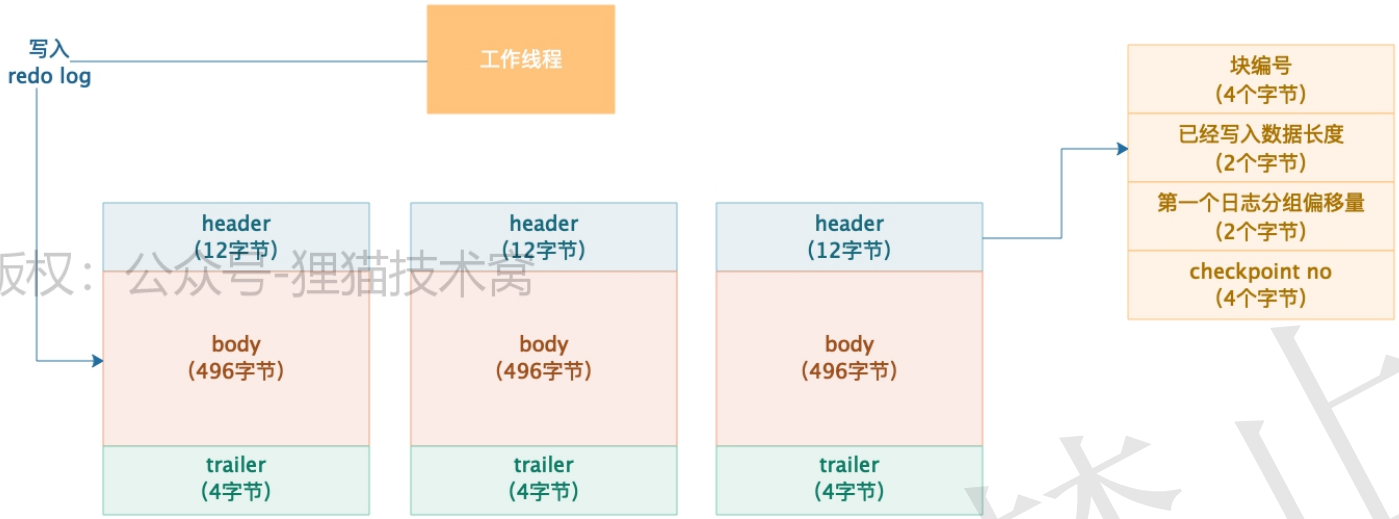
redo log buffer也是类似的，他是申请出来的一片连续内存，然后里面划分出了N多个空的redo log block，如下图所示。



通过设置mysql的innodb_log_buffer_size可以指定这个redo log buffer的大小，默认的值就是16MB，其实已经够大了，毕竟一个redo log block才512自己而已，每一条redo log其实也就几个字节到几十个字节罢了。

所以大家看到这里就明白了，上一讲我们就说了，其实redo log都是先写入内存里的redo log block数据结构里去的，然后完事儿了才会把redo log block写入到磁盘文件里去的

这里我们看到了redo log buffer的结构，就很清晰的知道，当你要写一条redo log的时候，就会先从第一个redo log block开始写入，如下图。



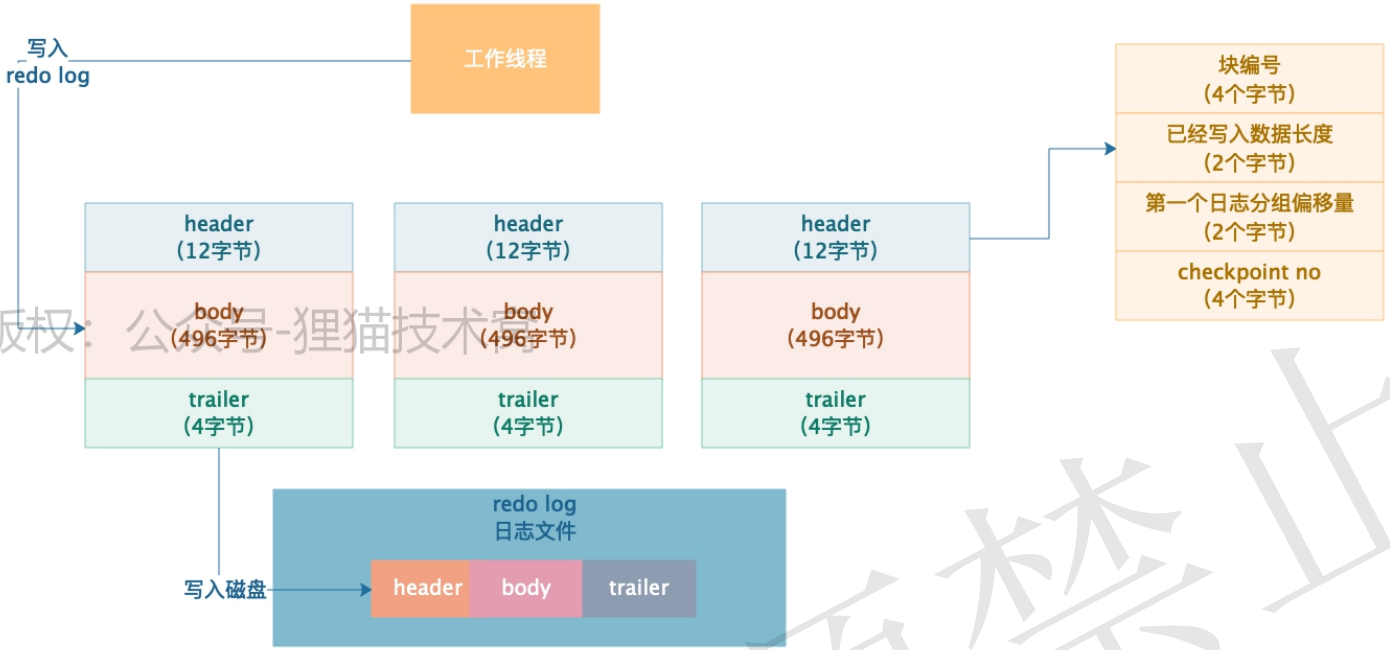
写满了一个redo log block，就会继续写下一个redo log block，以此类推，直到所有的redo log block都写满。

那么此时肯定有人会问了，万一要是redo log buffer里所有的redo log block都写满了呢?

那此时必然会强制把redo log block刷入到磁盘中去的!

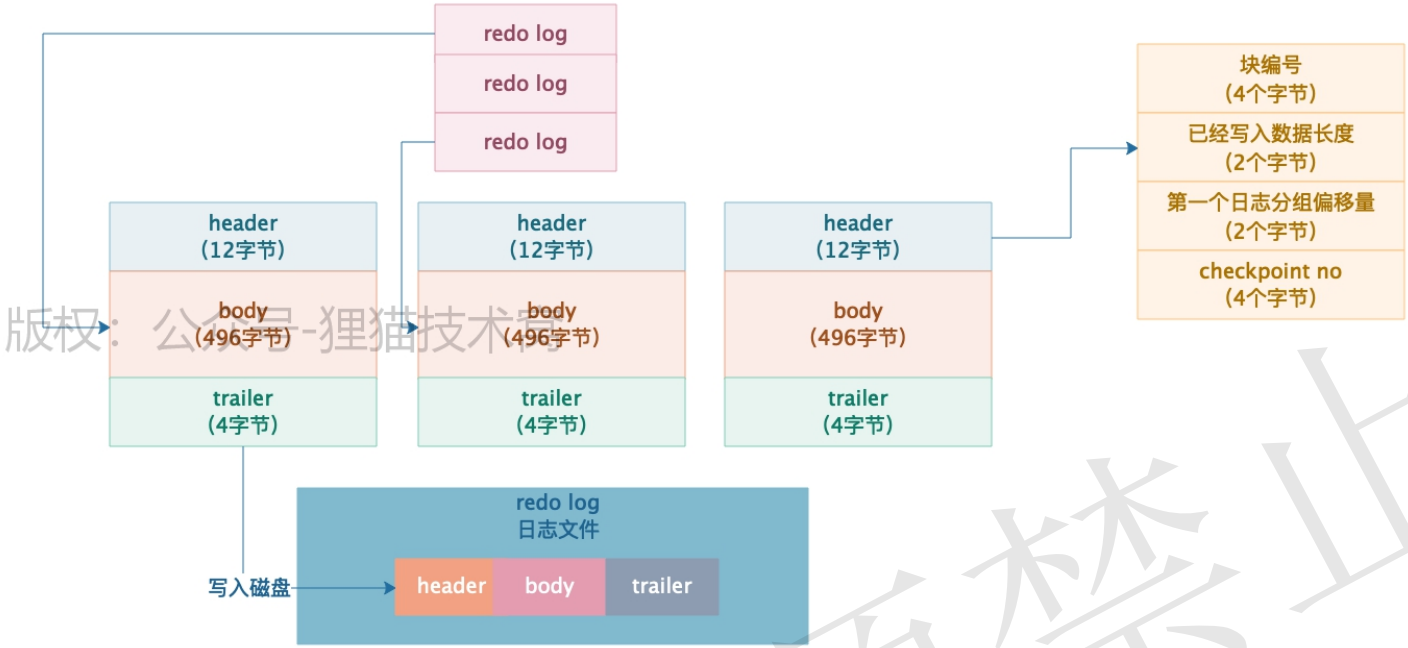
我们上一次讲到了redo log block刷入磁盘文件中的示意，其实就是把512字节的redo log block追加到redo log日志文件里去就可以了

看下面的图，里面就画的很清楚，在磁盘文件里不停的追加一个又一个的redo block。

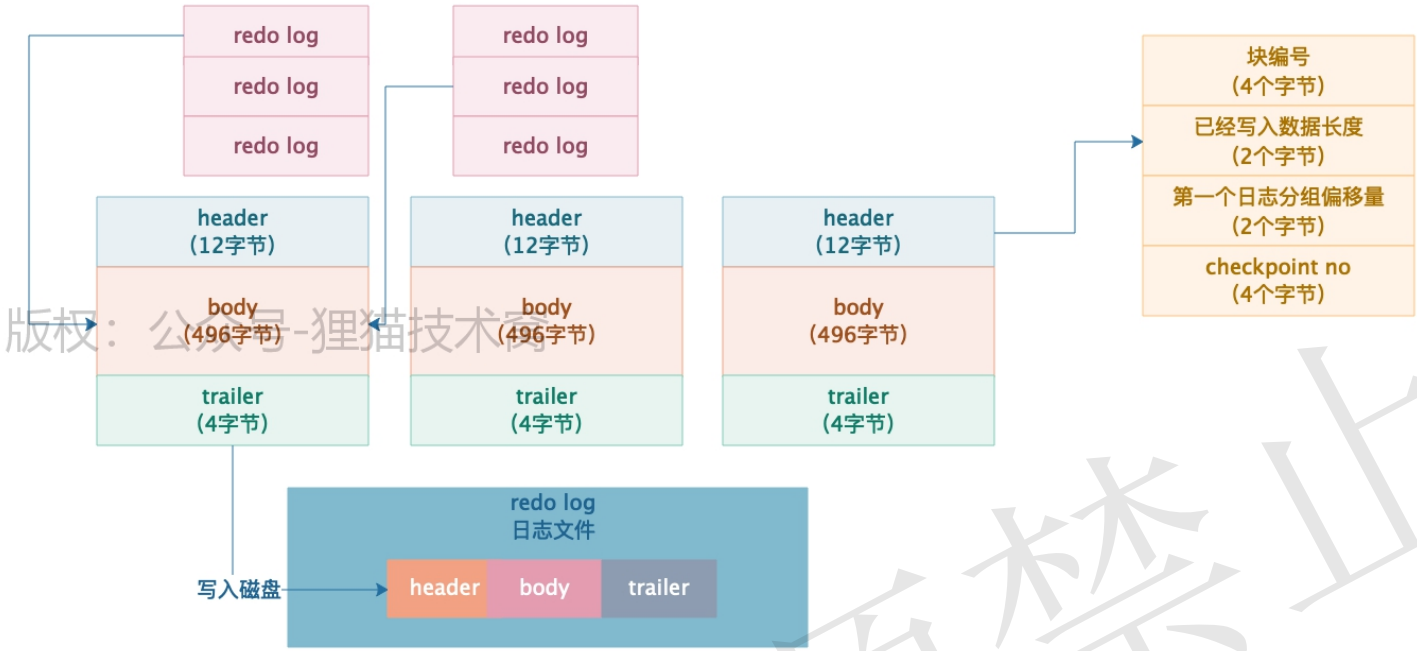


另外还要给大家讲一点的是，其实在我们平时执行一个事务的过程中，每个事务会有多个增删改操作，那么就会有多个redo log，这多个redo log就是一组redo log，其实每次一组redo log都是先在别的地方暂存，然后都执行完了，再把一组redo log给写入到redo log buffer的block里去的。

如果一组redo log实在是太多了，那么就可能会存放在两个redo log block中，我们看下图示意。



但是反之，如果说一个redo log group比较小，那么也可能多个redo log group是在一个redo log block里的，如下图所示。



想必今天的内容学习完，大家对于平时我们一个一个的事务里产生的多条redo log，是如何形成一个redo log组的，一组redo log是如何写入redo log buffer中的redo log block的，然后redo block是如何写入redo log磁盘文件的，这个全流程就有了一个清晰地理解和认识了！

下周我们要继续探索的，就是这个redo log buffer里的redo log block们到底是如何写入到磁盘文件里去的？

一定要等待redo log block全部写满了才会刷入磁盘吗？还有哪些其他的时机会把redo log block刷入磁盘吗？

这些问题，我们下周继续！

End

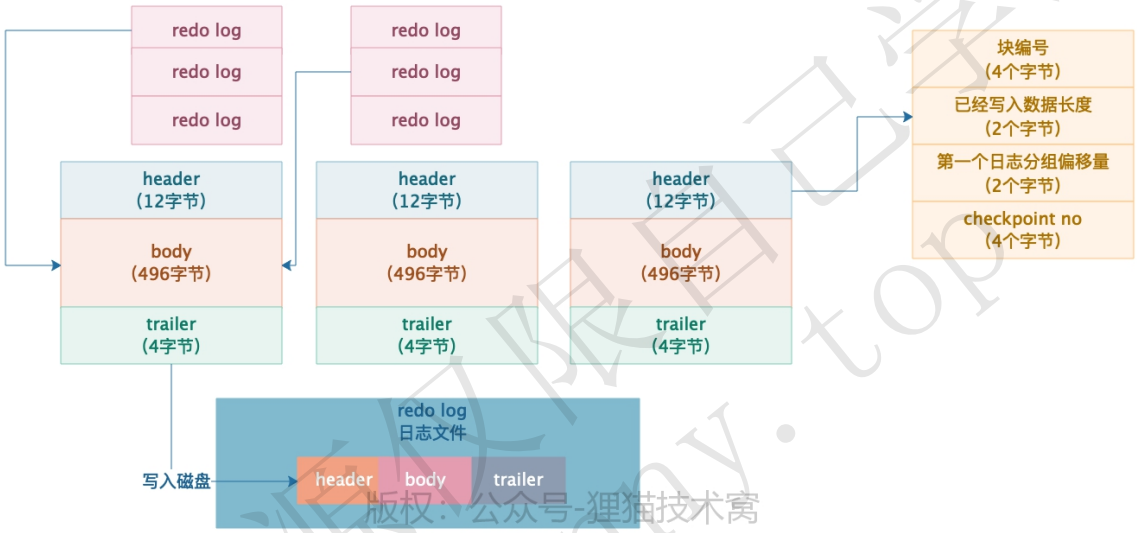
狸猫技术窝精品专栏及课程推荐：

- 《从零开始带你成为消息中间件实战高手》
- 《21天互联网Java进阶面试训练营》（分布式篇）
- 《互联网Java工程师面试突击》（第1季）
- 《互联网Java工程师面试突击》（第3季）
- 《从零开始带你成为JVM实战高手》

44 redo log buffer中的缓冲日志，到底什么时候可以写入磁盘?

之前我们给大家讲解了一下redo log buffer的缓冲机制，大家现在应该都知道了，redo log在写的时候，都是一个事务里的一组redo log，先暂存在一个地方，完事儿了以后把一组redo log写入redo log buffer。

写入redo log buffer的时候，是写入里面提前划分好的一个一个的redo log block的，选择有空闲空间的redo log block去写入，然后redo log block写满之后，其实会在某个时机刷入到磁盘里去，如下图。



那么今天我们就来研究一下，到底redo log buffer里的redo log block什么时候可以刷入到磁盘文件里去呢?

另外，磁盘上到底有几个redo log日志文件？不可能大量的redo log日志都放一个文件里吧？磁盘空间会占用的越来越多吗?

首先，我们先来看看redo log block是哪些时候会刷入到磁盘文件里去:

(1) 如果写入redo log buffer的日志已经占据了redo log buffer总容量的一半了，也就是超过了8MB的redo log在缓冲里了，此时就会把他们刷入到磁盘文件里去

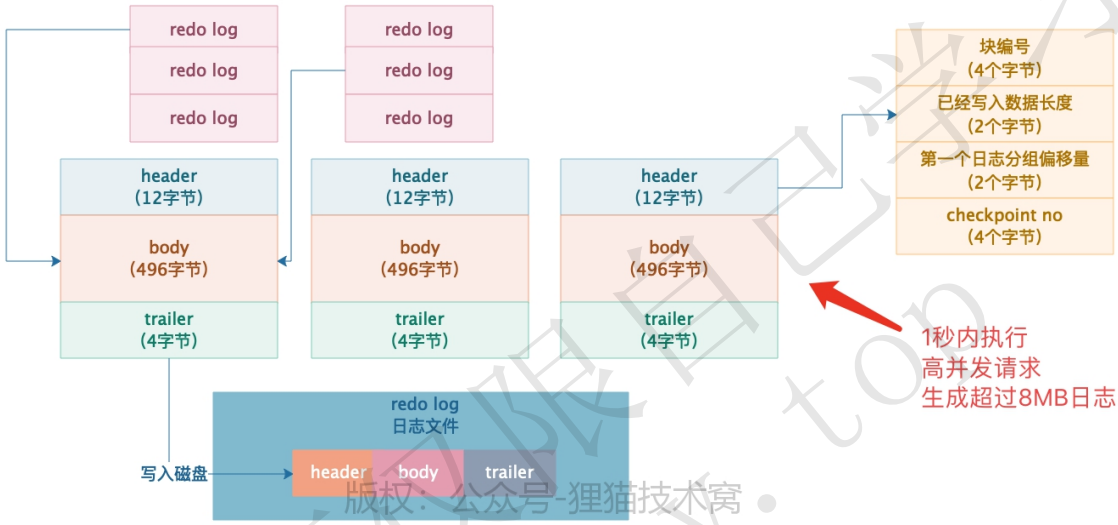
(2) 一个事务提交的时候，必须把他的那些redo log所在的redo log block都刷入到磁盘文件里去，只有这样，当事务提交之后，他修改的数据绝对不会丢失，因为redo log里有重做日志，随时可以恢复事务做的修改

（PS：当然，之前最早最早的时候，我们讲过，这个 redo log哪怕事务提交的时候写入磁盘文件，也是先进入 os cache的，进入 os的文件缓冲区里，所以是否提交事务就强行把 redo log刷入物理磁盘文件中，这个需要设置对应的参数，我们之前都讲过的 ，大家回过头去看看 ）

(3) 后台线程定时刷新，有一个后台线程每隔1秒就会把redo log buffer里的redo log block刷到磁盘文件里去

(4) MySQL关闭的时候，redo log block都会刷入到磁盘里去

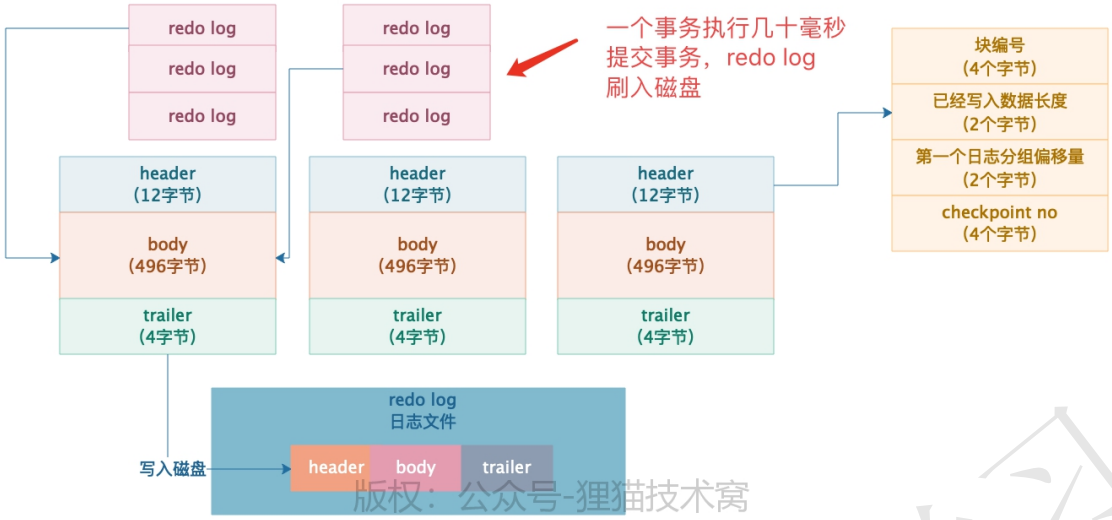
忽略上面的第四条不说，因为关闭MySQL的时候必然会刷redo log到磁盘，其他三条其实我们都看到了，也就是说，如果你瞬间执行大量的高并发的SQL语句，1秒内就产生了超过8MB的redo log，此时占据了redo log buffer一半的空间了，必然会直接把你的redo log刷入磁盘里去，如下图。



上面这种redo log刷盘的情况，在MySQL承载高并发请求的时候比较常见，比如每秒执行上万个增删改SQL语句，每个SQL产生的redo log假设有几百个字节，此时却是会在瞬间生成超过8MB的redo log日志，必然会触发立马刷新redo log到磁盘。

其次，第二种情况，其实就是平时执行一个事务，这个事务一般都是在几十毫秒到几百毫秒执行完毕的，说实在的，一般正常性能情况下，MySQL单事务性能一般不会超过1秒，否则数据库操作就太慢了。

那么如果在几十毫秒，或者几百毫秒的时候，执行完毕了一个事务，此时必然会立马把这个事务的redo log都刷入磁盘，如下图。



第一种情况其实是不常见的，第二种情况是比较常见的，往往redo log刷盘都是以一个短事务提交时候发生的，第三种情况就是后台线程每秒自动刷新redo log到磁盘去，这个就是说假设没有别的情况触发，后台线程自己都会不停的刷新redo log到磁盘。

但是不管怎么说，主要是保证一个事务执行的时候，redo log都进入redo log buffer，提交事务的时候，事务对应的redo log必须是刷入磁盘文件，接着才算是事务提交成功，否则事务提交就是失败，保证这一点，就能确保事务提交之后，数据不会丢，有redo log在磁盘里就行了。

当然，绝对保证数据不丢，还得配置一个参数，提交事务把redo log刷入磁盘文件的os cache之后，还得强行从os cache刷入物理磁盘。

最后给大家说一下redo log日志文件的问题，我们都知道平时不停的执行增删改，那么MySQL会不停的产生大量的redo log写入日志文件，那么日志文件就用一个写入全部的redo log？对磁盘占用空间越来越大怎么办？

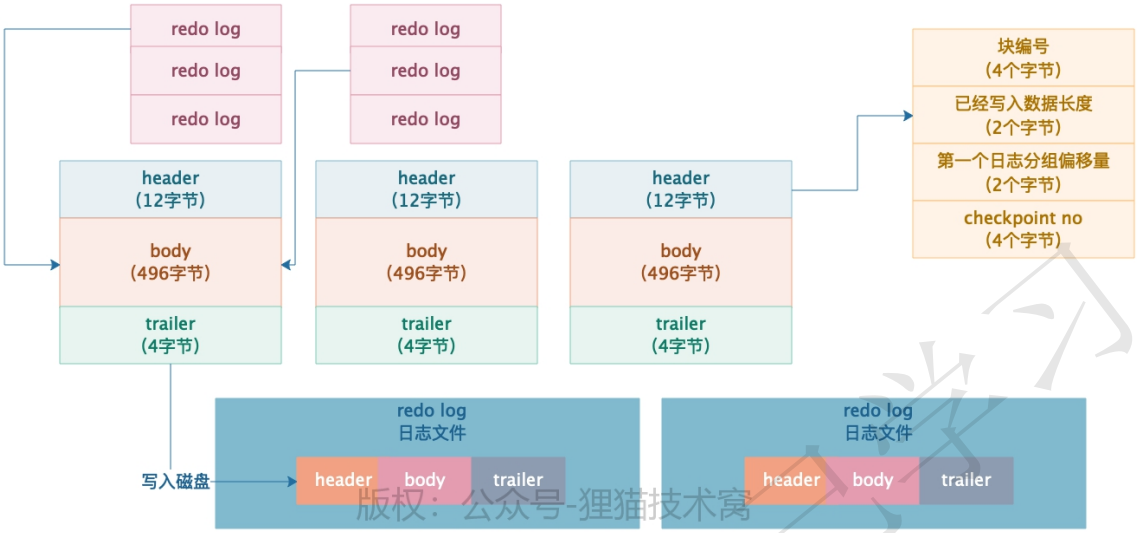
别担心，这些问题都可以解决，实际上默认情况下，redo log都会写入一个目录中的文件里，这个目录可以通过show variables like 'datadir'来查看，可以通过innodb_log_group_home_dir参数来设置这个目录的。

然后redo log是有多个的，写满了一个就会写下一个redo log，而且可以限制redo log文件的数量，通过innodb_log_file_size可以指定每个redo log文件的大小，默认是48MB，通过innodb_log_files_in_group可以指定日志文件的数量，默认就2个。

所以默认情况下，目录里就两个日志文件，分别为ib_logfile0和ib_logfile1，每个48MB，最多就这2个日志文件，就是先写第一个，写满了写第二个。那么如果第二个也写满了呢？别担心，继续写第一个，覆盖第一个日志文件里原来的redo log就可以了。

所以最多这个redo log，mysql就给你保留了最近的96MB的redo log而已，不过这其实已经很多了，毕竟redo log真的很小，一条通常就几个字节到几十个字节不等，96MB足够你存储上百万条redo log了！

如果你还想保留更多的redo log，其实调节上述两个参数就可以了，比如每个redo log文件是96MB，最多保留100个redo log文件。下面图里，给大家展示出来了多个redo log文件循环写入的示意。



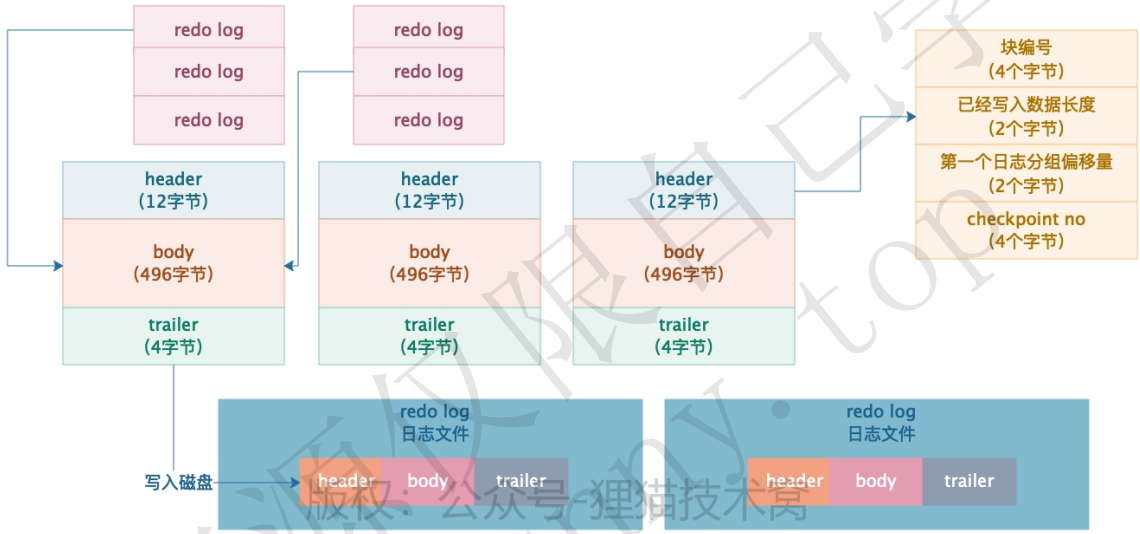
我想讲到这里，大家对redo log机制就理解更加深刻了，对于事务产生的redo log如何进入内存缓冲，如何进入block，什么时候刷入磁盘，磁盘上有几个redo log日志文件，这些机制都了解的很清晰了。

End

45 如果事务执行到一半要回滚怎么办？再探undo log回滚日志原理！

之前我们已经给大家深入讲解了在执行增删改操作时候的redo log的重做日志原理，其实说白了，就是你对buffer pool里的缓存页执行增删改操作的时候，必须要写对应的redo log记录下来你做了哪些修改

如下图所示，redo log都是先进入redo log buffer中的一个block，然后事务提交的时候就会刷入磁盘文件里去。

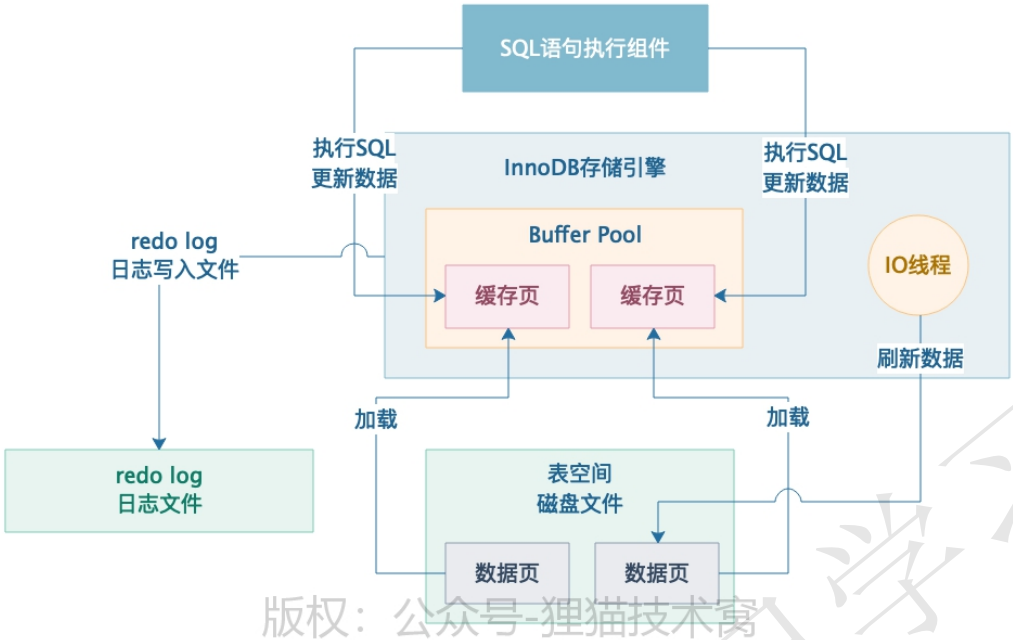


这样万一要是你提交事务了，结果事务修改的缓存页还没来得及刷入磁盘上的数据文件，此时你MySQL关闭了或者是宕机了，那么buffer pool里被事务修改过的数据就全部都丢失了！

但是只要有redo log，你重启MySQL之后完全是可以把那些修改了缓存页，但是缓存页还没来得及刷入磁盘的事务，他们所对应的redo log都加载出来，在buffer pool的缓存页里重做一遍，就可以保证事务提交之后，修改的数据绝对不会丢！

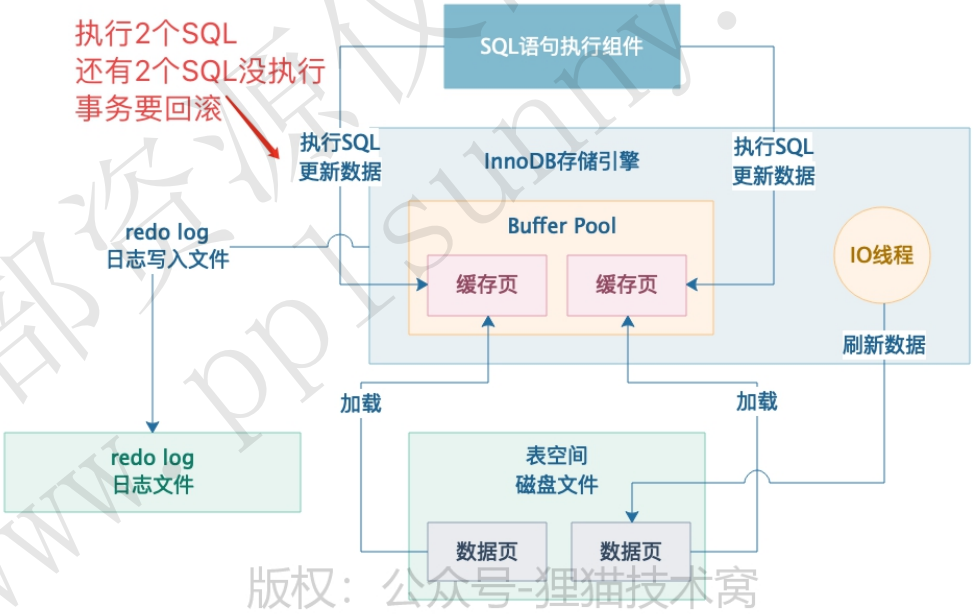
相信之前讲解了redo log日志之后，大家对这块都理解的更加深刻了，那么今天我们就带着大家来探索另外一种日志，就是undo log日志，也就是回滚日志，这种日志要应对的场景，就是事务回滚的场景！

那么首先大家先思考一个问题，假设现在我们一个事务里要执行一些增删改的操作，那么必然是先把对应的数据页从磁盘加载出来放buffer pool的缓存页里，然后在缓存页里执行一通增删改，同时记录redo log日志，对吧？如下图。



但是现在问题来了，万一要是一个事务里的一通增删改操作执行到了一半，结果就回滚事务了呢？

比如一个事务里有4个增删改操作，结果目前为止已经执行了2个增删改SQL了，已经更新了一些buffer pool里的数据了，但是还有2个增删改SQL的逻辑还没执行，此时事务要回滚了怎么办？看图



这个时候就很尴尬了，如果你要回滚事务的话，那么必须要把已经在buffer pool的缓存页里执行的增删改操作给回滚了

但是怎么回滚呢？毕竟无论是插入，还是更新，还是删除，该做的都已经做了啊！

所以在执行事务的时候，才必须引入另外一种日志，就是undo log回滚日志

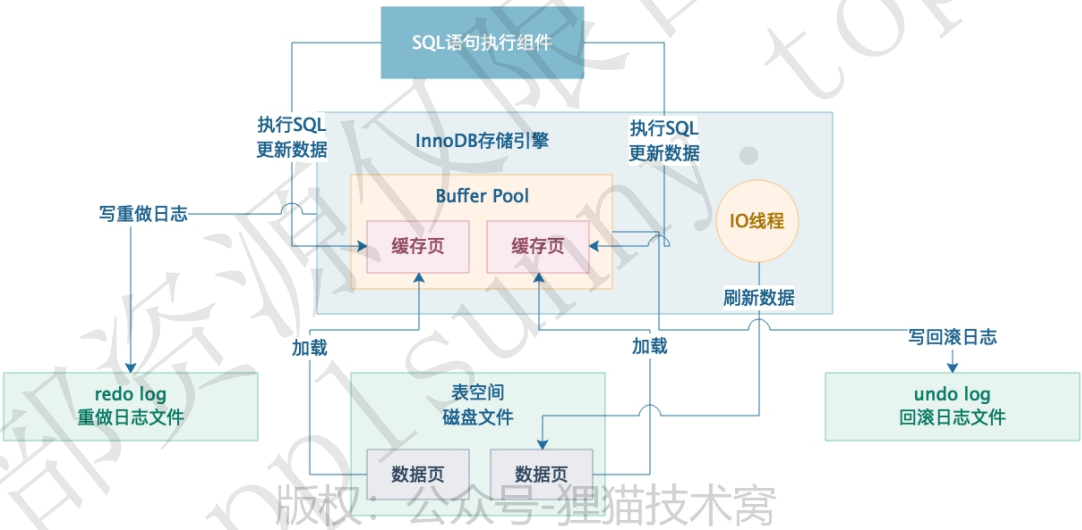
这个回滚日志，他记录的东西其实非常简单，比如你要是在缓存页里执行了一个insert语句，那么此时你在undo log日志里，对这个操作记录的回滚日志就必须是有一个主键和一个对应的delete操作，要能让你把这次insert操作给回退了。

那么比如说你要是执行的是delete语句，那么起码你要把你删除的那条数据记录下来，如果要回滚，就应该执行一个insert操作把那条数据插入回去。

如果你要是执行的是update语句，那么起码你要把你更新之前的那个值记录下来，回滚的时候重新update一下，把你之前更新前的旧值给他更新回去。

如果你要是执行的是select语句呢？不好意思，select语句压根儿没有在buffer pool里执行任何修改，所以根本不需要undo log！

好，所以我们来看下图，其实你在执行事务期间，之前我们最开始的几篇文章就讲过，你除了写redo log日志还必须要写undo log日志，这个undo log日志是至关重要的，没有他，你根本都没办法回滚事务！



明天我们继续来看看insert、delete和update几种操作的undo log到底长什么样，相信大家看完了，就会对undo log这块机制有一个更加深刻的理解了。

End

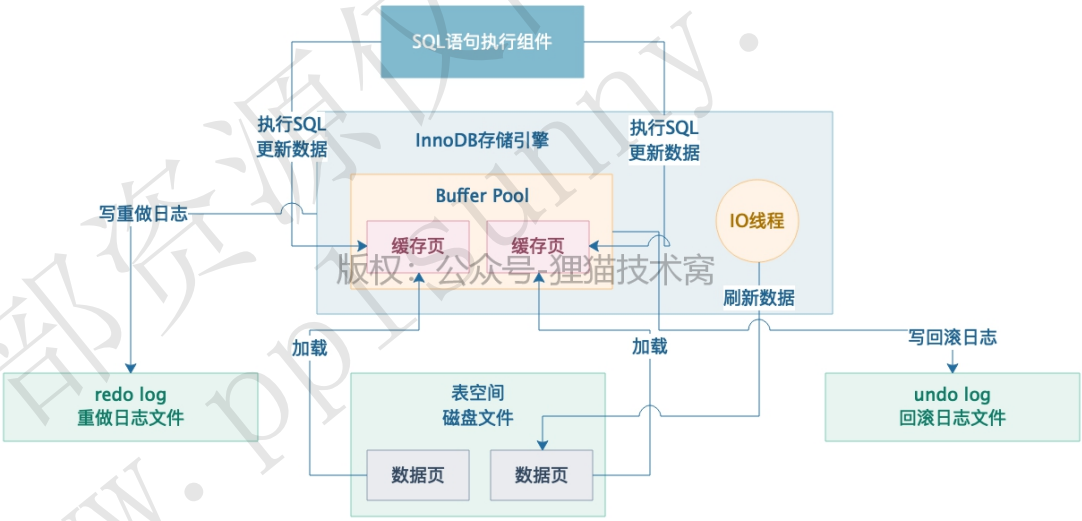
46 一起来看看INSRET语句的undo log回滚日志长什么样?

昨天我们讲解了undo log回滚日志的作用，说白了，就是你执行事务的时候，里面很多INSERT、UPDATE和DELETE语句都在更新缓存页里的数据，但是万一事务回滚，你必须有每条SQL语句对应的undo log回滚日志，根据回滚日志去恢复缓存页里被更新的数据。

比如你执行了INSERT语句，那么你的undo log必须告诉你插入数据的主键ID，让你在回滚的时候可以从缓存页里把这条数据给删除了；

如果你执行了DELETE语句，那么你的undo log必须记录下来被删除的数据，回滚的时候就得重新插入一条数据；

如果你执行了UPDATE语句，那么你必须记录下来修改之前的数据，回滚的时候就得把数据给更新回去，如下图所示。



那么今天我们就一起来看看这个INSERT语句的undo log日志到底长什么样子呢?

INSERT语句的undo log的类型是TRX_UNDO_INSERT_REC，这个undo log里包含了以下一些东西:

- 这条日志的开始位置
- 主键的各列长度和值
- 表id
- undo log日志编号
- undo log日志类型
- 这条日志的结束位置

接下来我们来给大家解释一下，首先，一条日志必须得有自己的一个开始位置，这个没什么好说的是吧?

那么主键的各列长度和值是什么意思？大家都知道，你插入一条数据，必然会有一个主键！

如果你自己指定了一个主键，那么可能这个主键就是一个列，比如id之类的，也可能是多个列组成的一个主键，比如“id+name+type”三个字段组成的一个联合主键，也是有可能的。

所以这个主键的各列长度和值，意思就是你插入的这条数据的主键的每个列，他的长度是多少，具体的值是多少。即使你没有设置主键，MySQL自己也会给你弄一个row_id作为隐藏字段，做你的主键。

接着是表id，这个就不用多说了，你插入一条数据必然是往一个表里插入数据的，那当然得有一个表id，记录下来是在哪个表里插入的数据了。

undo log日志编号，这个意思就是，每个undo log日志都是有自己的编号的。

而在一个事务里会有多个SQL语句，就会有多个undo log日志，在每个事务里的undo log日志的编号都是从0开始的，然后依次递增。

至于undo log日志类型，就是TRX_UNDO_INSERT_REC，insert语句的undo log日志类型就是这个东西。

最后一个undo log日志的结束位置，这个自然也不用多说了，他就是告诉你undo log日志结束的位置是什么。

那么接着我们用一个图画一下这个INSERT语句的undo log回滚日志的结构，大家来看一眼，感受一下。

大家可以想象一下，有了这条日志之后，剩下的事儿就好办了

万一要是你现在在buffer pool的一个缓存页里插入了一条数据了，执行了insert语句，然后你写了一条上面的那种undo log，现在事务要是回滚了，你直接就把这条insert语句的undo log拿出来。

然后在undo log里就知道在哪个表里插入的数据，主键是什么，直接定位到那个表和主键对应的缓存页，从里面删除掉之前insert语句插入进去的数据就可以了，这样就可以实现事务回滚的效果了！

好了，今天先初步的看一下insert语句的undo log回顾日志，delete语句和update语句的回滚日志我们暂时就不细讲了，其实大家应该都能想象到他们是如何实现的。

End

47 简单回顾一下， MySQL运行时多个事务同时执行是什么场景?

到目前为止，我们已经给大家深入讲解了MySQL的buffer pool机制、redo log机制和undo log机制，相信大家现在对我们平时执行一些增删改语句的实现原理，都有了一定较为深入的理解了!

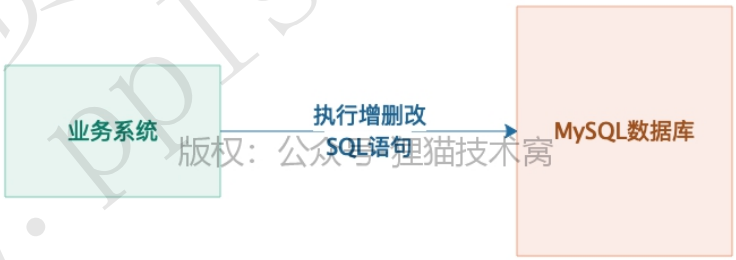
因为平时我们执行增删改的时候，无非就是从磁盘加载数据页到buffer pool的缓存页里去，对缓存页进行更新，同时记录下来undo log回滚日志和redo log重做日志，应对的是事务提交之后MySQL挂了恢复数据的场景，以及事务回滚的场景。

那么接下来其实我们要理解的东西，就是要提高一个层次了，我们要理解到事务这个层面了

所谓的事务呢，其实或多或少每个人都是知道一点的，我们今天只不过是站在MySQL内核原理的角度，拔高到事务的层面给大家回顾一下。

其实大家可以思考，平时我们是不是一般都是写一个业务系统，然后业务系统会去对数据库执行增删改查?

好，我们看下图



然后我们应该都知道一件事情，通常而言，我们都是在业务系统里会开启事务来执行增删改操作的，我随便给大家举个例子，下面的代码大家看看。

```java
@Transactional
public void doService() {
    // 增加一条数据
    addUser();
    // 修改一条数据
    updateUser();
    // 删除一条数据
    deleteUser();
}
```

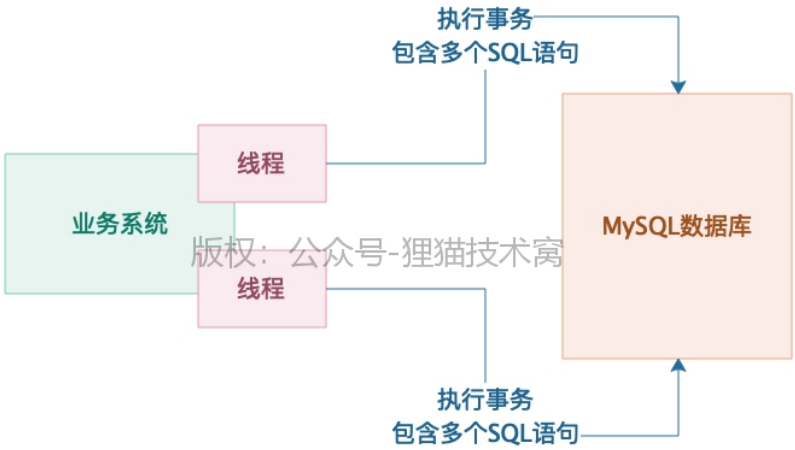
所以一般来说，业务系统是执行一个一个的事务，每个事务里可能是一个或者多个增删改查的SQL语句

这个事务的概念想必不用我多说了，其实就是一个事务里的SQL要不然一起成功就提交了，要不然有一个SQL失败，那么事务就回滚了，所有SQL做的修改都撤销了！我们看下图。

接着问题来了，这个业务系统他可不是一个单线程系统啊！他是有很多线程的！

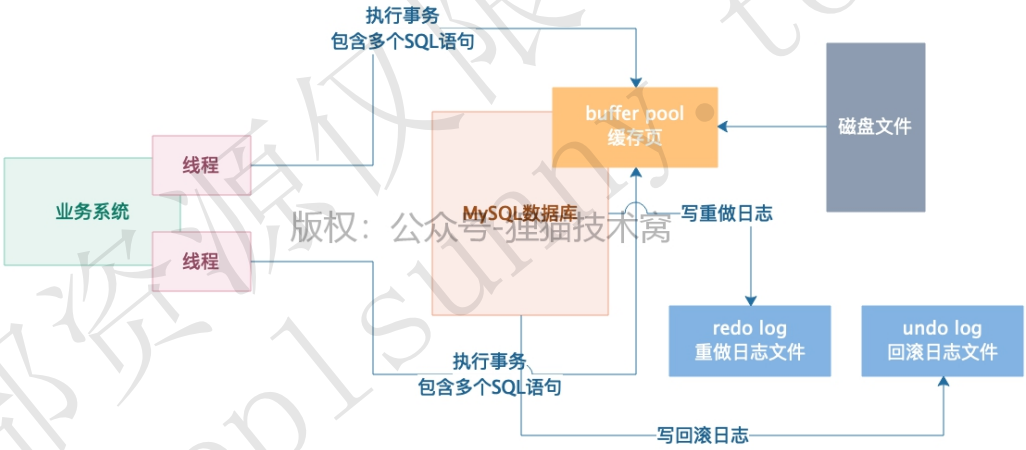
因为他面向的终端用户是有很多人的，可能会同时发起请求，所以他需要多个线程并发来处理多个请求的

于是，这个业务系统很可能是基于多线程并发的对MySQL数据库去执行多个事务的！看下图。



那么每个事务里面的多个SQL语句都是如何执行的呢?

其实就是我们之前给大家讲过的那一套原理了，包括从磁盘加载数据页到buffer pool的缓存页里去，然后更新buffer pool里的缓存页，同时记录redo log和undo log，如下图所示。



每个事务如果提交了，那么就皆大欢喜，这个提交的过程我之前最早就讲过了，他有一些步骤，包括在redo log里记录事务提交标识之类的。

如果事务提交之后，redo log刷入磁盘，结果MySQL宕机了，是可以根据redo log恢复事务修改过的缓存数据的。

如果要回滚事务，那么就基于undo log来回滚就可以了，把之前对缓存页做的修改都给回滚了就可以了。

这就是在MySQL内核层面，把多个事务和我们之前讲解的buffer pool、redo log、undo log几个机制都结合在一起的一个场景讲解。想必大家都是可以理解的。

但是这里就有很多问题了：

- 多个事务并发执行的时候，可能会同时对缓存页里的一行数据进行更新，这个冲突怎么处理？是否要加锁？
- 可能有的事务在对一行数据做更新，有的事务在查询这行数据，这里的冲突怎么处理？

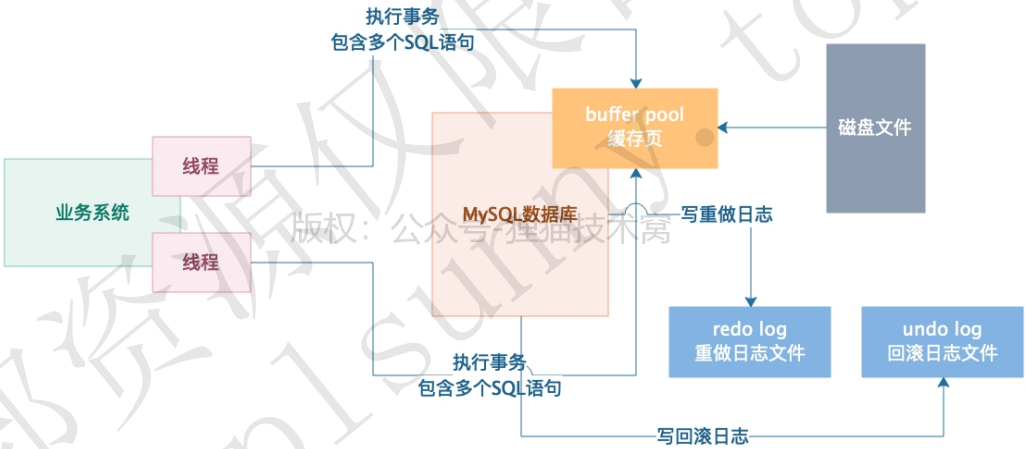
所以接下来，我们要给大家讲解的，**就是**解决多个事务并发运行的时候，同时写和同时读写的一些并发冲突的处理机制，包括了MySQL事务的隔离级别、MVCC多版本隔离、锁机制，等等。**

End

48 多个事务并发更新以及查询数据，为什么会有脏写和脏读的问题？

- **如何提问**：每篇文章都有评论区，大家在评论区留言提问
- **如何加入狸猫技术交流群**：
  - 添加微信号：Lvgu0715_ （微信名：绿小九），狸猫技术窝的管理员
  - 发送专栏购买截图
  - 2小时内管理员会拉群，人工操作请耐心等待

上一次我们已经讲到，其实对于我们的业务系统去访问数据库而言，他往往都是多个线程并发执行多个事务的，对于数据库而言，他会有多个事务同时执行，可能这多个事务还会同时更新和查询同一条数据，所以这里会有一些问题需要数据库来解决，如下图。



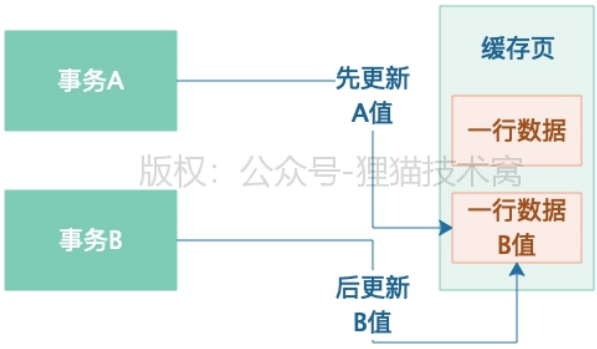
每个事务都会执行各种增删改查的语句，把磁盘上的数据页加载到buffer pool的缓存页里来，然后更新缓存页，记录redo log和undo log，最终提交事务或者是回滚事务，多个事务会并发干上述一系列事情。

所以今天我们就要来看看，如果多个事务要是对缓存页里的同一条数据同时进行更新或者查询，此时会产生哪些问题呢？

这里实际上会涉及到**脏写、脏读、不可重复读、幻读**，四种问题。

先看第一种问题，脏写

这个脏写的话，他的意思就是说有两个事务，事务A和事务B同时在更新一条数据，事务A先把他更新为A值，事务B紧接着就把他更新为B值，如下图所示



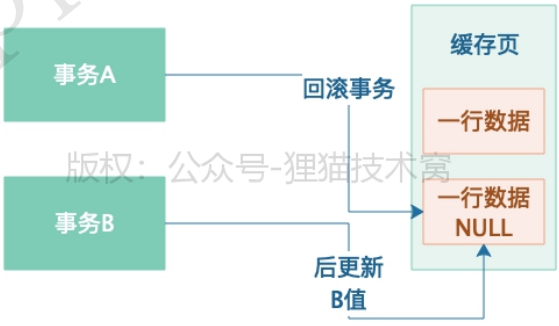
大家可以看到，此时事务B是后更新那行数据的值，所以此时那行数据的值是不是B值？

没错的。而且此时事务A更新之后会记录一条undo log日志，大家应该还记得吧。事务A是先更新的，他在更新之前，这行数据的值为NULL，对吧？

所以此时事务A的undo log日志大概就是：更新之前这行数据的值为NULL，主键为XX

好，那么此时事务B更新完了数据的值为B，结果此时事务A突然回滚了，那么就会用他的undo log日志去回滚。

此时事务A一回滚，直接就会把那行数据的值更新回之前的NULL值！所以此时事务A回滚了，可能看起来这行数据的值就是NULL了，如下图。



然后就尴尬了，事务B一看，我的妈呀，为什么我更新的B值没了？就因为你事务A反悔了就把数据值回滚成NULL了，搞的我更新的B值也没了，这也太坑爹了吧！

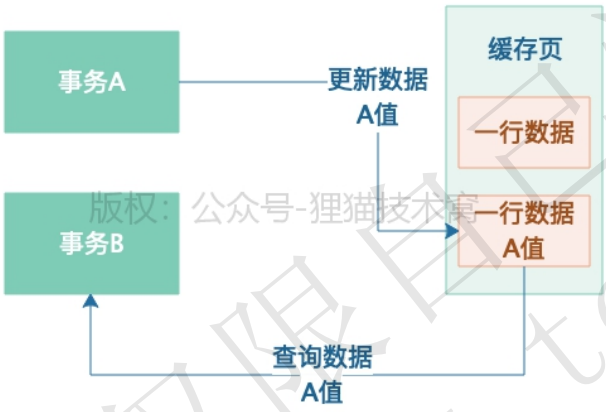
所以对于事务B看到的场景，就是自己明明更新了，结果值却没了，**这就是脏写**！

所谓脏写，就是我刚才明明写了一个数据值，结果过了一会儿却没了！真是莫名其妙。

而他的本质就是事务B去修改了事务A修改过的值，但是此时事务A还没提交，所以事务A随时会回滚，导致事务B修改的值也没了，这就是脏写的定义。

接着我们继续看坑爹的脏读问题

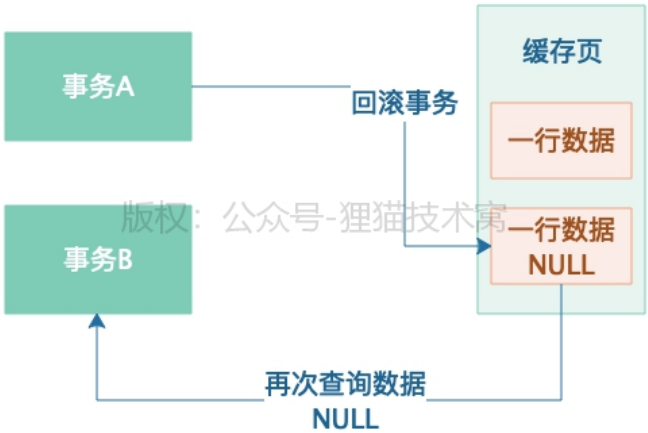
假设事务A更新了一行数据的值为A值，此时事务B去查询了一下这行数据的值，看到的值是不是A值？没错，此时如下图所示。



好，现在事务B可能还挺high的，拿着刚才查询到的A值做各种业务处理。大家知道，每个事务都是业务系统发出的，所以业务系统里的事务B此时肯定会拿到刚查出来的A值在做一些业务处理。

但是接着坑爹的事情发生了，事务A突然回滚了事务，导致他刚才更新的A值没了，此时那行数据的值回滚为NULL值！

然后事务B紧接着此时再次查询那行数据的值，看到的居然此时是NULL值？事务B此时简直欲哭无泪，看下图：



所以这就是坑爹的脏读，他的本质其实就是事务B去查询了事务A修改过的数据，但是此时事务A还没提交，所以事务A随时会回滚导致事务B再次查询就读不到刚才事务A修改的数据了！这就是脏读。

好了，今天我们先初步看一下脏写和脏读两种问题，相信大家有了之前的大量MySQL底层知识原理的铺垫，理解这两个问题，肯定是小case，轻松+愉快的

其实一句话总结，**无论是脏写还是脏读，都是因为一个事务去更新或者查询了另外一个还没提交的事务更新过的数据。**

因为另外一个事务还没提交，所以他随时可能会反悔会回滚，那么必然导致你更新的数据就没了，或者你之前查询到的数据就没了，这就是脏写和脏读两种坑爹场景。

End

49 一个事务多次查询一条数据读到的都是不同的值，这就是不可重复读?

上一讲我们说完了多个事务并发执行时候，对MySQL的缓存页里的同一行数据同时进行更新或者查询的时候，可能发生的脏写和脏读的问题

我们也都理解了，之所以会发生脏写和脏读，最关键的，其实是因为你一个事务写或者查的是人家事务还没提交的时候更新过的数据，所以人家事务随时会反悔回滚，导致你这里有问题。

那么今天我们继续看多个事务并发执行时候的另外两种问题：**一个是不可重复读，一个是幻读**

这两种问题都会奇葩一些，大家仔细看下面的图演示。

先来说说这个不可重复读的问题，这个问题是这样的：假设我们有一个事务A开启了，在这个事务A里会多次对一条数据进行查询

然后呢，另外有两个事务，一个是事务B，一个是事务C，他们俩都是对一条数据进行更新的。

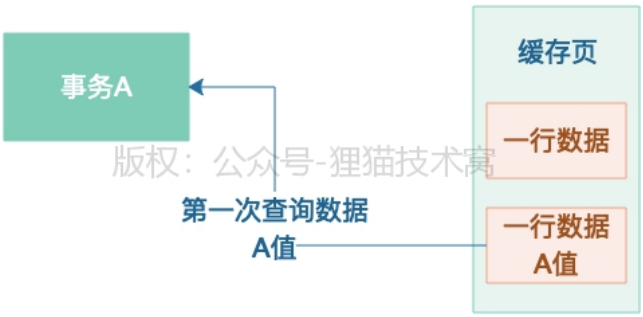
然后我们假设一个前提，就是比如说事务B更新数据之后，如果还没提交，那么事务A是读不到的，必须要事务B提交之后，他修改的值才能被事务A给读取到，其实这种情况下，就是我们首先避免了脏读的发生。

因为脏读的意思就是事务A可以读到事务B修改过还没提交的数据，此时事务B一旦回滚，事务A再次读就读不到了，那么此时就会发生脏读问题。

我们现在假设的前提是事务A只能在事务B提交之后读取到他修改的数据，所以此时必然是不会发生脏读的

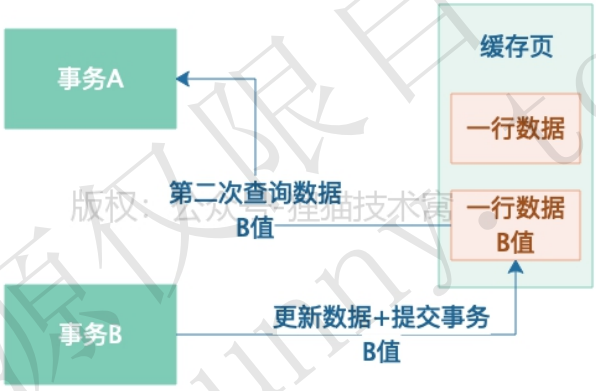
好了，但是你以为没有脏读就万事大吉了吗？绝对不是，此时会有另外一个问题，叫做**不可重复读**

假设缓存页里一条数据原来的值是A值，此时事务A开启之后，第一次查询这条数据，读取到的就是A值，如下图所示。

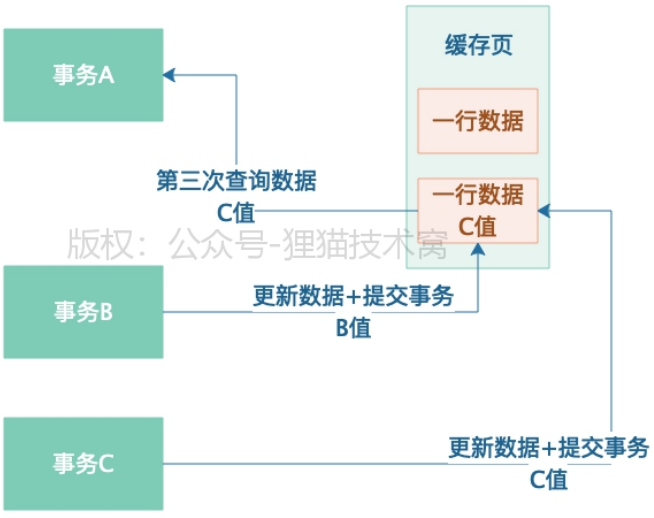


接着事务B更新了那行数据的值为B值，同时事务B立马提交了，然后事务A此时可是还没提交！

大家注意，此时事务A是没提交的，他在事务执行期间第二次查询数据，此时查到的是事务B修改过的值，B值，因为事务B已经提交了，所以事务A可以读到的了，此时如下图所示。



紧接着事务C再次更新数据为C值，并且提交事务了，此时事务A在没提交的情况下，第三次查询数据，查到的值为C值，如下图所示。



好，那么上面的场景有什么问题呢？

其实要说没问题也可以是没问题，毕竟事务B和事务C都提交之后，事务A多次查询查到他们修改的值，是ok的。

但是你要说有问题，也可以是有问题的，就是事务A可能第一次查询到的是A值，那么他可能希望的是在事务执行期间，如果多次查询数据，都是同样的一个A值，他希望这个A值是他重复读取的时候一直可以读到的！他希望这行数据的值是可重复读的！

但是此时，明显A值不是可重复读的，因为事务B和事务C一旦更新了值并且提交了，事务A会读到别的值，所以此时这行数据的值是不可重复读的！此时对于你来说，这个不可重复读的场景，就是一种问题了！

不知道大家看到这里理解了没？如果没理解，反复把这个例子看几遍，理解一下！

上面描述的，其实就是不可重复读的问题，其实这个问题你说是问题也不一定就是什么大问题，但是说他有问题，确实是有问题的。

因为这取决于你自己想要数据库是什么样子的，如果你希望看到的场景就是不可重复读，也就是事务A在执行期间多次查询一条数据，每次都可以查到其他已经提交的事务修改过的值，那么就是不可重复读的，如果你希望这样子，那也没问题。

但是如果你希望的是，假设你事务A刚开始执行，第一次查询读到的是值A，然后后续你希望事务执行期间，读到的一直都是这个值A，不管其他事务如何更新这个值，哪怕他们都提交了，你就希望你读到的一直是第一次查询到的值A，那么你就是希望可重复读的。

如果你期望的是可重复读，但是数据库表现的是不可重复读，让你事务A执行期间多次查到的值都不一样，都是别的提交过的事务修改过的值，那么此时你就可以认为，数据库有问题，这个问题就是“不可重复读”的问题！

不知道大家听懂这个不可重复读的问题了吗？稍微有点绕口，但是我觉得这篇文章已经解释的很清晰了，请大家反复看2遍，一定会理解可重复读和不可重复读的区别，以及为什么不可重复读会定义为一种数据库的问题呢！

End

50 听起来很恐怖的数据库幻读，到底是个什么奇葩问题？

上一讲我们给大家讲解了不可重复读这个问题，这个问题简单来说，就是一个事务多次查询一条数据，结果每次读到的值都不一样，这个过程中可能别的事务会修改这条数据的值，而且修改值之后事务都提交了，结果导致人家每次查到的值都不一样，都查到了提交事务修改过的值，这就是所谓的不可重复读。

不可重复读，就是一条数据的值没法满足多次重复读值都一样，别的事务修改了值提交之后，就不可重复读了，说着有点拗口，不过相信大家应该反复看两遍，应该都能理解

包括之前的脏写和脏读其实也都分别代表了不同的数据库的问题，脏写就是两个事务没提交的状况下，都修改同一条数据，结果一个事务回滚了，把另外一个事务修改的值也给撤销了，所谓脏写就是两个事务没提交状态下修改同一个值。

脏读就是一个事务修改了一条数据的值，结果还没提交呢，另外一个事务就读到了你修改的值，然后你回滚了，人家事务再次读，就读不到了，也就是说人家事务读到了你修改之后还没提交的值，这就是脏读了。

而不可重复读，针对的是已经提交的事务修改的值，被你事务给读到了，你事务内多次查询，多次读到的是别的已经提交的事务修改过的值，这就导致不可重复读了。

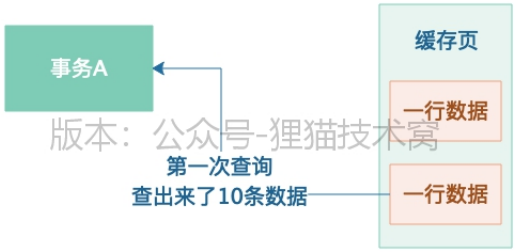
这个数据库的多种并发问题，确实很拗口，需要大家好好理解。

今天我们来最后讲一种数据库的并发问题，就是听着有点恐怖的幻读问题，幻读听起来很恐怖，搞的跟邪恶的巫师给你搞什么魔法一样，是不是？

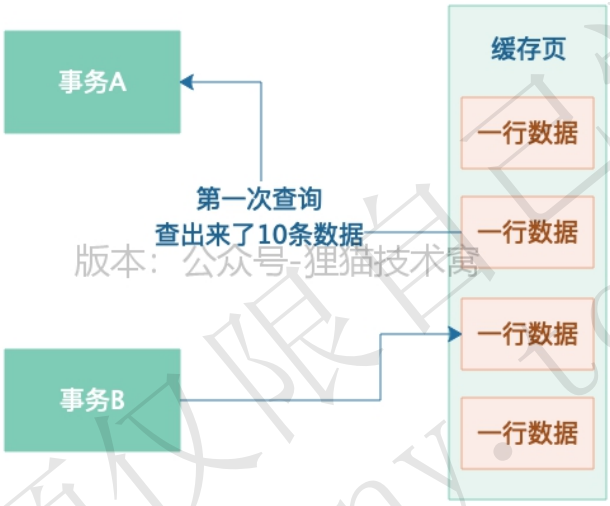
其实没那么恐怖的，我们今天给大家来讲讲。

简单来说，你一个事务A，先发送一条SQL语句，里面有一个条件，要查询一批数据出来，比如“select * from table where id>10”，类似这种SQL

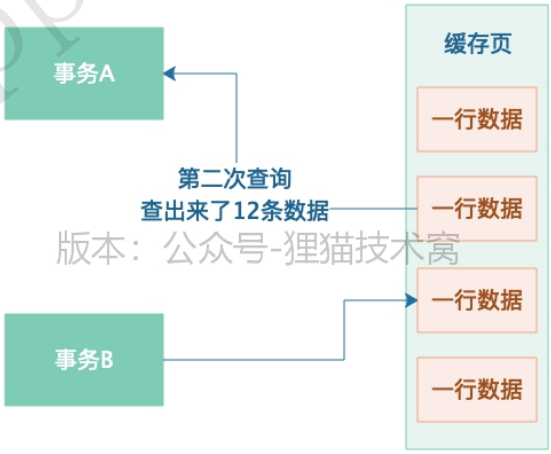
然后呢，他一开始查询出来了10条数据，如下图所示。



接着这个时候，别的事务B往表里插入了几条数据，而且事务B还提交了，如下图所示，此时多了几行数据出来。



接着事务A此时第三次查询，再次按照之前的一模一样的条件执行“select * from table where id>10”这条SQL语句，由于其他事务插入了几条数据，导致这次他查询出来了12条数据，如下图所示。



于是此时事务A开始怀疑自己的双眼了，为什么一模一样的SQL语句，第一次查询是10条数据，第二次查询是12条数据？难道刚才出现了幻觉？导致我刚才幻读了？这就是幻读这个名词的由来。

幻读指的就是你一个事务用一样的SQL多次查询，结果每次查询都会发现查到了一些之前没看到过的数据

注意，幻读特指的是你查询到了之前查询没看到过的数据！此时就说你是幻读了。

说实在的，大家看完最近几篇文章，应该都有一个感觉，就是脏写、脏读、不可重复读、幻读，都是因为业务系统会多线程并发执行，每个线程可能都会开启一个事务，每个事务都会执行增删改查操作。

然后数据库会并发执行多个事务，多个事务可能会并发的对缓存页里的同一批数据进行增删改查操作，于是这个并发增删改查同一批数据的问题，可能就会导致我们说的脏写、脏读、不可重复读、幻读，这些问题。

所以这些问题的本质，都是数据库的多事务并发问题，那么为了解决多事务并发问题，数据库才设计了事务隔离机制、MVCC多版本隔离机制、锁机制，用一整套机制来解决多事务并发问题，接下来，我们将要深入讲解这些机制，让大家彻底能够理解数据库内部的执行原理。

深刻理解了原理之后，后续我们讲解各种数据库优化实践案例，大家才能深刻理解，就跟我之前的《从0开始带你成为JVM实战高手》一样，先得透彻理解了JVM运行的工作原理，才能理解如何对JVM进行实战优化。

只不过JVM的内部运行原理比数据库而言相对内容少一些，所以当时很快就进入了大量的实践案例阶段，而数据库的话，内部原理会更加复杂一些，所以耗费的时间会多一些。

End

51 SQL标准中对事务的4个隔离级别，都是如何规定的呢？

之前我们给大家讲了数据库中多个事务并发时可能产生的几种问题，包括了脏写、脏读、不可重复读、幻读，几种问题

那么针对这些多事务并发的问题，实际上SQL标准中就规定了事务的几种隔离级别，用来解决这些问题。

注意一下，我们今天讲的这个SQL标准的事务隔离级别，并不是MySQL的事务隔离级别，MySQL在具体实现事务隔离级别的时候会有点差别，这个我们下一次再讲，今天先关注SQL标准是如何规定事务隔离级别的。

在SQL标准中规定了4种事务隔离级别，就是说多个事务并发运行的时候，互相是如何隔离的，从而避免一些事务并发问题

这4种级别包括了：**read uncommitted（读未提交），read committed（读已提交），repeatable read（可重复读），serializable（串行化）**

不同的隔离级别是可以避免不同的事务并发问题的，所以大家一定要对这个事务隔离级别有一个深刻的理解。

第一个read uncommitted隔离级别，是不允许发生脏写的

也就是说，不可能两个事务在没提交的情况下去更新同一行数据的值，但是在这种隔离级别下，可能发生脏读，不可重复读，幻读。

感觉如何？是不是感觉这种隔离级别让你整个人都感觉不好了！因为脏读的话，就是人家事务在没提交情况下修改的值，居然被你读到了，人家随时可能会回滚的！而且你执行期间多次查询一行数据，可能读到的值都不同，因为别的事务随时会修改值再提交，这个值是不可重复读的！幻读更不用说了，肯定会发生。

说实在的，一般来说，是没有人做系统开发的时候，傻到把事务隔离级别设置为读未提交这个级别的。

第二个是read committed隔离级别，这个级别下，不会发生脏写和脏读

也就是说，人家事务没提交的情况下修改的值，你是绝对读不到的！但是呢，可能会发生不可重复读和幻读问题，因为一旦人家事务修改了值然后提交了，你事务是会读到的，所以可能你多次读到的值是不同的！

这里教给大家一个稍微有点骚气的简写名词，就是RC，一般如果你在公司里做开发，有个其他团队的兄弟讨论技术方案的时候，跟你来了句，把事务隔离级别设置成RC！这个时候你不要目瞪口呆，知道是读已提交级别就行了。

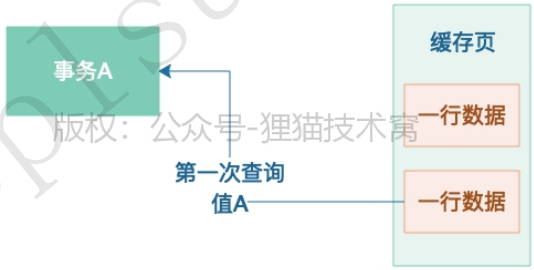
你只要记住，这个级别在别的事务已经提交之后读到他们修改过的值就可以了，但是别的事务没提交的时候，绝对不会读到人家修改的值。

第三个是REPEATABLE READ隔离级别，就是可重复读级别

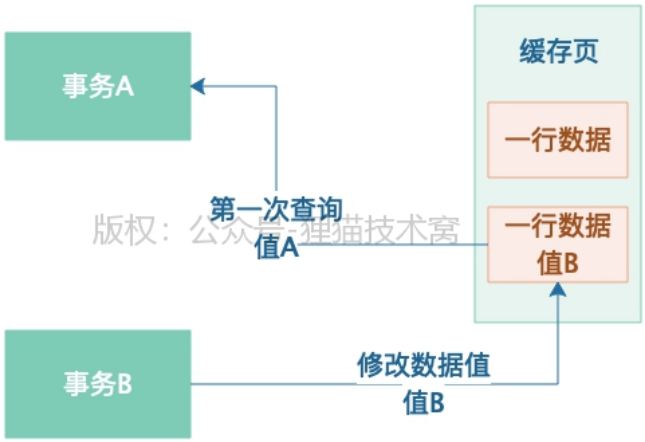
这个级别下，不会发生脏写、脏读和不可重复读的问题，因为你一个事务多次查询一个数据的值，哪怕别的事务修改了这个值还提交了，没用，你不会读到人家提交事务修改过的值，你事务一旦开始，多次查询一个值，会一直读到同一个值！

我们给大家一个图看看这个RR级别的效果，以后记得这个骚气的简写词，就是RR，公司里有兄弟让你把事务设置成RR的时候，你也不要一脸懵逼。

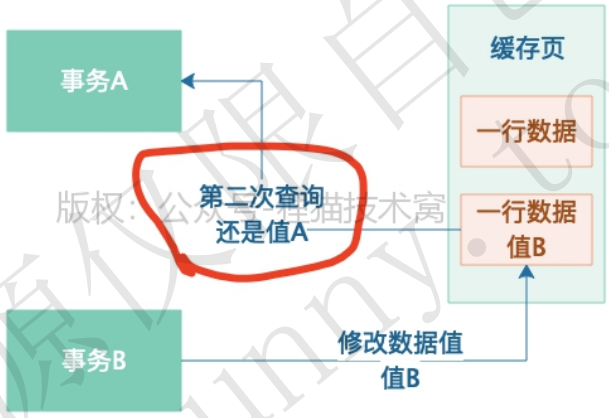
一个事务A，第一次查询一行数据的值是值A，如下图所示。



接着事务B修改了这行数据的值为值B，还提交了，如下图所示。



接着事务A再次查询这行数据的值，读到的还是值A，因为他在事务执行期间，多次读一行数据，绝对读到的都是一样的值，他是允许可重复读的！希望大家理解一下这个概念，可重复读！如下图。



这就是第三种隔离级别，给一个骚气的名字，RR级别，记住了RR级别保证你不会读到人家已经提交的事务修改过的值！但是他还是会发生幻读的

因为假设你一次SQL是根据条件查询，比如“select * from table where id>10”，第一次查出来10条数据，结果另外一个事务插入了一条数据，下次你可能会查出来11条数据，还是会有幻读问题的！

RR隔离级别，只不过保证对同一行数据的多次查询，你不会读到不一样的值，人家已提交事务修改了这行数据的值，对你也没影响！

最后一个隔离就别，就是serializable级别，这种级别，根本就不允许你多个事务并发执行，只能串行起来执行，先执行事务A提交，然后执行事务B提交，接着执行事务C提交，所以此时你根本不可能有幻读的问题，因为事务压根儿都不并发执行！

但是这种级别一般除非脑子坏了，否则更不可能设置了，因为多个事务串行，那数据库恨不能一秒并发就只有几十了，性能会极差的。

好了，今天就把SQL标准里的四种隔离级别讲完了，**大家一定要记住非常骚气的RC和RR级别**，因为平时比较常见的就是用RC和RR两种隔离级别。

下次我们继续讲MySQL里是如何支持事务隔离级别的，同时在Spring的事务注解里是如何设置事务隔离级别的。

End

52 MySQL是如何支持4种事务隔离级别的？Spring事务注解是如何设置的?

MySQL是如何支持4种事务隔离级别的？Spring事务注解是如何设置的?

上次我们讲完了SQL标准下的4种事务隔离级别，平时比较多用的就是RC和RR两种级别，那么在MySQL中也是支持那4种隔离级别的，基本的语义都是差不多的

但是要注意的一点是，MySQL默认设置的事务隔离级别，都是RR级别的，而且MySQL的RR级别是可以避免幻读发生的。

这点是MySQL的RR级别的语义跟SQL标准的RR级别不同的，毕竟SQL标准里规定RR级别是可以发生幻读的，但是MySQL的RR级别避免了！

也就是说，MySQL里执行的事务，默认情况下不会发生脏写、脏读、不可重复读和幻读的问题，事务的执行都是并行的，大家互相不会影响，我不会读到你没提交事务修改的值，即使你修改了值还提交了，我也不会读到的，即使你插入了一行值还提交了，我也不会读到的，总之，事务之间互相都完全不影响！

当然，要做到这么神奇和牛叉的效果，MySQL是下了苦功夫的，后续我们接着就要讲解MySQL里的MVCC机制，就是多版本并发控制隔离机制，依托这个MVCC机制，就能让RR级别避免不可重复读和幻读的问题。

然后给大家说一下，假设你要修改MySQL的默认事务隔离级别，是下面的命令，可以设置级别为不同的level，level的值可以是REPEATABLE READ，READ COMMITTED，READ UNCOMMITTED，SERIALIZABLE几种级别。

SET [GLOBAL|SESSION] TRANSACTION ISOLATION LEVEL level;

但是一般来说，真的其实不用修改这个级别，就用默认的RR其实就特别好，保证你每个事务跑的时候都没人干扰，何乐而不为呢?

另外，给大家说一下，假设你在开发业务系统的时候，比如用Spring里的@Transactional注解来做事务这块，假设某个事务你就是有点手痒痒，就想给弄成RC级别，你就想读到人家已经提交事务修改过的值，好，那么没问题。

在@Transactional注解里是有一个isolation参数的，里面是可以设置事务隔离级别的，具体的设置方式如下：

@Transactional(isolation=Isolation.DEFAULT)，然后默认的就是DEFAULT值，这个就是MySQL默认支持什么隔离级别就是什么隔离级别。

那MySQL默认是RR级别，自然你开发的业务系统的事务也都是RR级别的了。

但是你可以手动改成Isolation.READ_UNCOMMITTED级别，此时你就可以读到人家没提交事务修改的值了，够坑的！估计一般没人自己坑自己吧！

也可以改成Isolation.READ_COMMITTED，Isolation.REPEATABLE_READ，Isolation.SERIALIZABLE几个级别，都是可以的。

但是再次提醒，其实默认的RR隔离机制挺好的，真的没必要去修改，除非你一定要在你的事务执行期间多次查询的时候，必须要查到别的已提交事务修改过的最新值，那么此时你的业务有这个要求，你就把Spring的事务注解里的隔离级别设置为Isolation.READ_COMMITTED级别，偶尔可能也是有这种需求的。

好了，事务的并发问题以及事务隔离级别，我们迄今为止已经剖析的很透彻了，接下来就开始讲解MVCC机制，透彻剖析MySQL是怎么实现牛叉的RR级别的，怎么就能让事务互相之间彻底隔离开来呢？

End

53 理解MVCC机制的前奏：undo log版本链是个什么东西?

理解MVCC机制的前奏：undo log版本链是个什么东西?

今天我们正式开始切入讲解MySQL中多个事务并发执行时的隔离到底是怎么做的，因为我们知道默认是骚气的RR隔离级别，也就是说脏写、脏读、不可重复读、幻读，都不会发生，每个事务执行的时候，跟别的事务压根儿就没关系，甭管你别的事务怎么更新和插入，我查到的值都是不变的，是一致的!

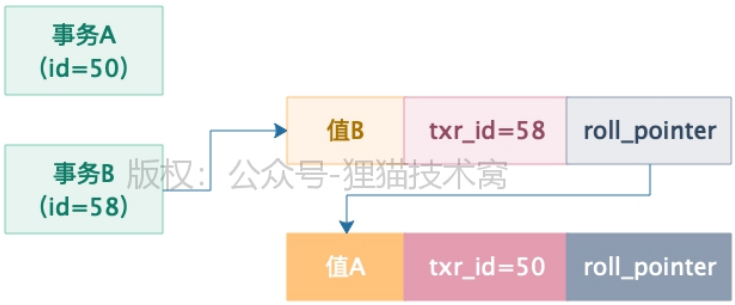
但是这到底是怎么做到的呢?

这就是由经典的**MVCC多版本并发控制机制**做到的，但是讲解这个MVCC机制之前，我们还得先讲讲undo log版本链的故事，这是一个前奏，了解了这个机制，大家才能更好的理解MVCC机制。

简单来说呢，我们每条数据其实都有两个隐藏字段，一个是trx_id，一个是roll_pointer，这个trx_id就是最近一次更新这条数据的事务id，roll_pointer就是指向你了你更新这个事务之前生成的undo log，关于undo log之前都讲过了，这里不用多说了。

我们给大家举个例子，现在假设有一个事务A（id=50），插入了一条数据，那么此时这条数据的隐藏字段以及指向的undo log如下图所示，插入的这条数据的值是值A，因为事务A的id是50，所以这条数据的txr_id就是50，roll_pointer指向一个空的undo log，因为之前这条数据是没有的。

接着假设有一个事务B跑来修改了一下这条数据，把值改成了值B，事务B的id是58，那么此时更新之前会生成一个undo log记录之前的值，然后会让roll_pointer指向这个实际的undo log回滚日志，如下图所示。

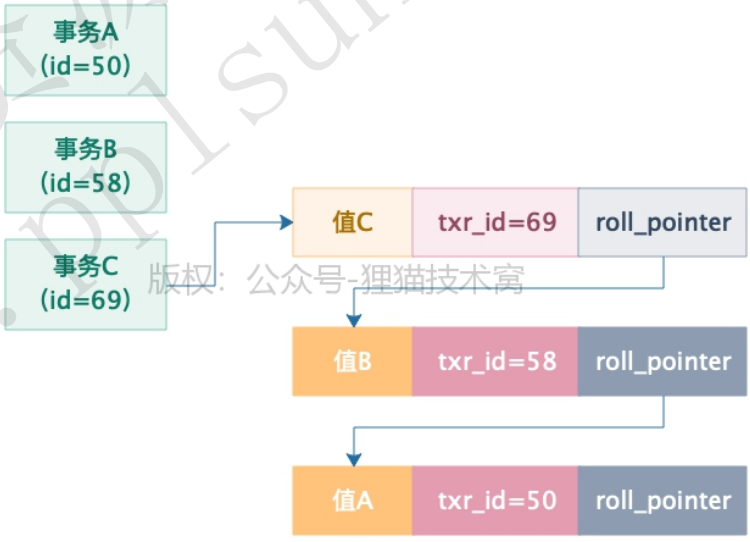


大家看上图是不是觉得很有意思？事务B修改了值为值B，此时表里的那行数据的值就是值B了，那行数据的txr_id就是事务B的id，也就是58，roll_pointer指向了undo log，这个undo log就记录你更新之前的那条数据的值。

所以大家看到roll_pointer指向的那个undo log，里面的值是值A，txr_id是50，因为undo log里记录的这个值是事务A插入的，所以这个undo log的txr_id就是50，我还特意把表里的那行数据和undo log的颜色弄成不一样的，以示区分。

接着假设事务C又来修改了一下这个值为值C，他的事务id是69，此时会把数据行里的txr_id改成69，然后生成一条undo log，记录之前事务B修改的那个值

此时如下图所示，看起来如下。



我们在上图可以清晰看到，数据行里的值变成了值C，txr_id是事务C的id，也就是69，然后roll_pointer指向了本次修改之前生成的undo log，也就是记录了事务B修改的那个值，包括事务B的id，同时事务B修改的那个undo log还串联了最早事务A插入的那个undo log，如图所示，过程很清晰明了。

所以这就是今天要给大家讲的一点，大家先不管多个事务并发执行是如何执行的，起码先搞清楚一点，就是多个事务串行执行的时候，每个人修改了一行数据，都会更新隐藏字段txr_id和roll_pointer，同时之前多个数据快照对应的undo log，会通过roll_pinter指针串联起来，形成一个重要的版本链！

今天要让大家明白的，就是这个多个事务串行更新一行数据的时候，txr_id和roll_pinter两个隐藏字段的概念，包括undo log串联起来的多版本链条的概念！

End

54 基于undo log多版本链条实现的ReadView机制，到底是什么?

基于undo log多版本链条实现的ReadView机制，到底是什么?

接着上次我们讲过的undo log多版本链条，我们来讲讲这个基于undo log多版本链条实现的ReadView机制

把这个机制讲明白了，下一次我们再正式讲解RC和RR隔离级别下的MVCC多版本并发控制机制，就很容易理解了。

这个ReadView呢，简单来说，就是你执行一个事务的时候，就给你生成一个ReadView，里面比较关键的东西有4个

- 一个是m_ids，这个就是说此时有哪些事务在MySQL里执行还没提交的;
- 一个是min_trx_id，就是m_ids里最小的值;
- 一个是max_trx_id，这是说mysql下一个要生成的事务id，就是最大事务id;
- 一个是creator_trx_id，就是你这个事务的id

那么现在我们来举个例子，让大家通过例子来理解这个ReadView是怎么用的

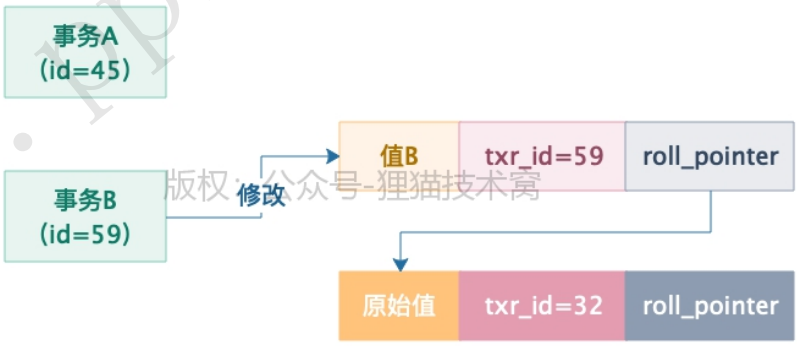
假设原来数据库里就有一行数据，很早以前就有事务插入过了，事务id是32，他的值就是初始值，如下图所示。

接着呢，此时两个事务并发过来执行了，一个是事务A（id=45），一个是事务B（id=59），事务B是要去更新这行数据的，事务A是要去读取这行数据的值的，此时两个事务如下图所示。

现在事务A直接开启一个ReadView，这个ReadView里的m_ids就包含了事务A和事务B的两个id，45和59，然后min_trx_id就是45，max_trx_id就是60，creator_trx_id就是45，是事务A自己。

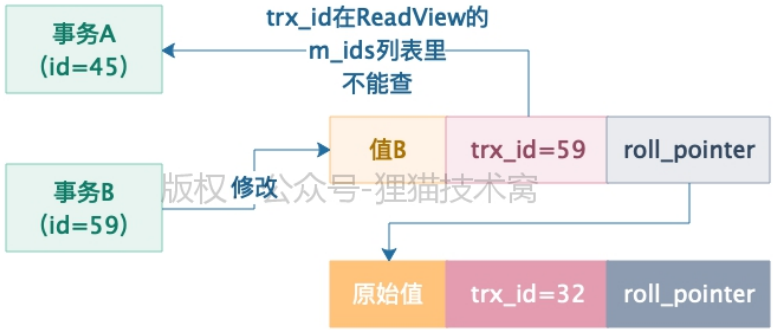
这个时候事务A第一次查询这行数据，会走一个判断，就是判断一下当前这行数据的txr_id是否小于ReadView中的min_trx_id，此时发现txr_id=32，是小于ReadView里的min_trx_id就是45的，说明你事务开启之前，修改这行数据的事务早就提交了，所以此时可以查到这行数据，如下图所示。

接着事务B开始动手了，他把这行数据的值修改为了值B，然后这行数据的txr_id设置为自己的id，也就是59，同时roll_pointer指向了修改之前生成的一个undo log，接着这个事务B就提交了，如下图所示。



这个时候事务A再次查询，此时查询的时候，会发现一个问题，那就是此时数据行里的txr_id=59，那么这个txr_id是大于ReadView里的min_txr_id(45)，同时小于ReadView里的max_trx_id（60）的，说明更新这条数据的事务，很可能就跟自己差不多同时开启的，于是会看一下这个txr_id=59，是否在ReadView的m_ids列表里?

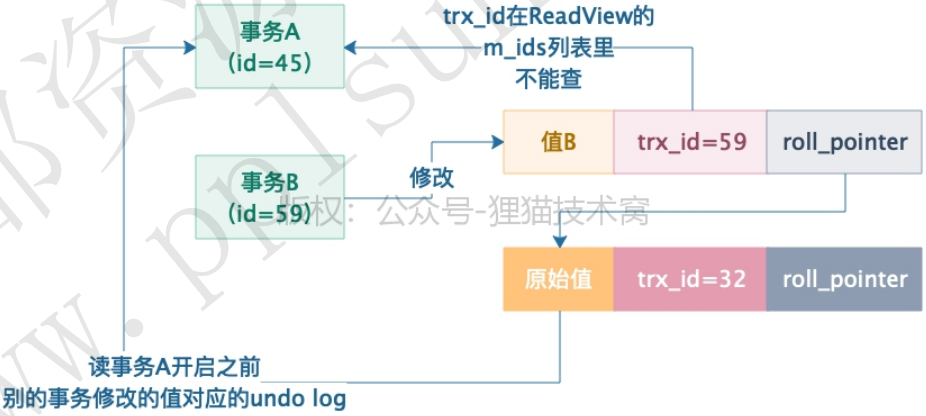
果然，在ReadView的m_ids列表里，有45和59两个事务id，直接证实了，这个修改数据的事务是跟自己同一时段并发执行然后提交的，所以对这行数据是不能查询的！如下图所示。



那么既然这行数据不能查询，那查什么呢？

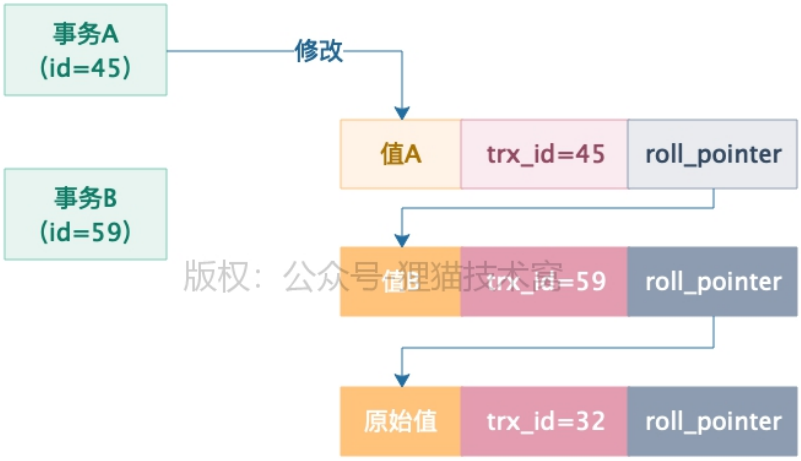
简单，顺着这条数据的roll_pointer顺着undo log日志链条往下找，就会找到最近的一条undo log，trx_id是32，此时发现trx_id=32，是小于ReadView里的min_trx_id（45）的，说明这个undo log版本必然是在事务A开启之前就执行且提交的。

好了，那么就查询最近的那个undo log里的值好了，这就是undo log多版本链条的作用，他可以保存一个快照链条，让你可以读到之前的快照值，如下图。



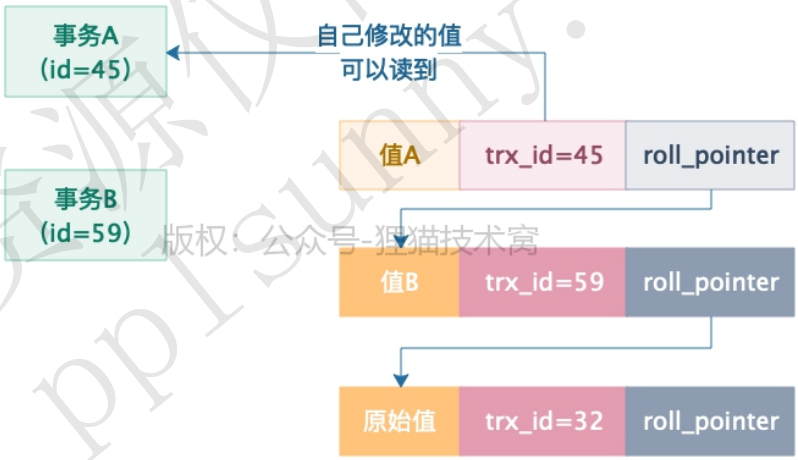
看到这里，大家有没有觉得很奇妙？多个事务并发执行的时候，事务B更新的值，通过这套**ReadView+undo log**日志链条的机制，就可以保证事务A不会读到并发执行的事务B更新的值，只会读到之前最早的值。

接着假设事务A自己更新了这行数据的值，改成值A，trx_id修改为45，同时保存之前事务B修改的值的快照，如下图所示。

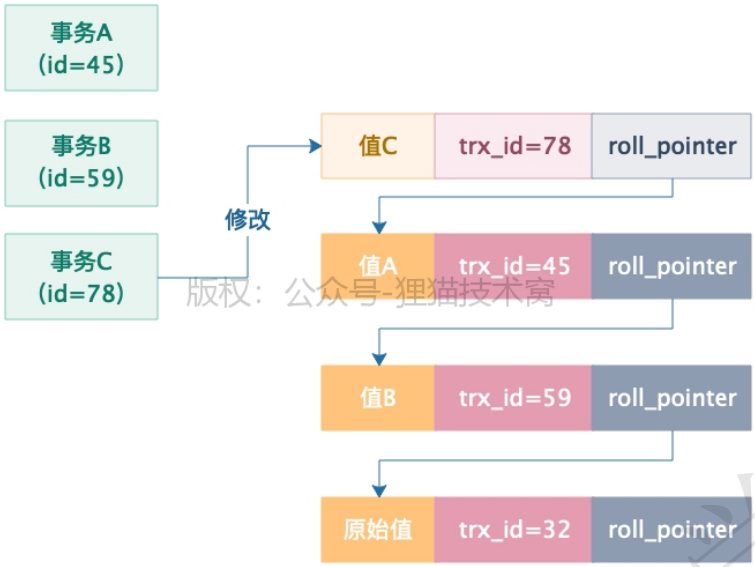


此时事务A来查询这条数据的值，会发现这个trx_id=45，居然跟自己的ReadView里的creator_trx_id（45）是一样的，说明什么?

说明这行数据就是自己修改的啊！自己修改的值当然是可以看到的了！如下图。

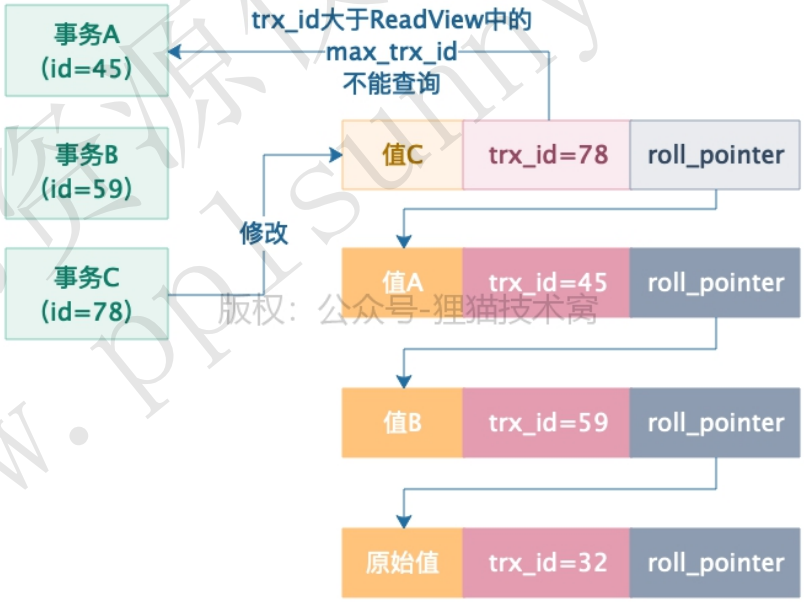


接着在事务A执行的过程中，突然开启了一个事务C，这个事务的id是78，然后他更新了那行数据的值为值C，还提交了，如下图所示。

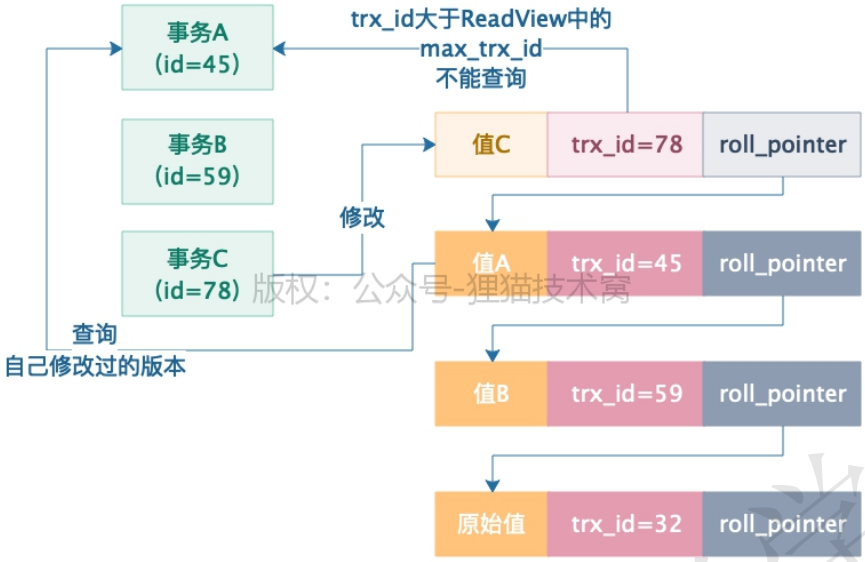


这个时候事务A再去查询，会发现当前数据的trx_id=78，大于了自己的ReadView中的max_trx_id（60），此时说明什么?

说明是这个事务A开启之后，然后有一个事务更新了数据，自己当然是不能看到的了！如下图。



此时就会顺着undo log多版本链条往下找，自然先找到值A自己之前修改的过的那个版本，因为那个trx_id=45跟自己的ReadView里的creator_trx_id是一样的，所以此时直接读取自己之前修改的那个版本，如下图。



不知道大家看到这里感觉如何？通过一系列的图，我相信每个人都能彻底理解这个ReadView的一套运行机制了

通过undo log多版本链条，加上你开启事务时候生产的一个ReadView，然后再有一个查询的时候，根据ReadView进行判断的机制，你就知道你应该读取哪个版本的数据。

而且他可以保证你只能读到你事务开启前，别的提交事务更新的值，还有就是你自己事务更新的值。假如说是你事务开启之前，就有别的事务正在运行，然后你事务开启之后 ，别的事务更新了值，你是绝对读不到的！或者是你事务开启之后，比你晚开启的事务更新了值，你也是读不到的！

通过这套机制就可以实现多个事务并发执行时候的数据隔离，下一次我们继续深入讲解RC和RR两个隔离级别下，这个ReadView是如何运用的。

End

55 Read Committed隔离级别是如何基于ReadView机制实现的?

Read Committed隔离级别是如何基于ReadView机制实现的?

今天我们来给大家讲一下，基于之前我们说的ReadView机制是如何实现Read Committed隔离级别的，那么当然了，首先就是要先做一些简单的回顾。所谓的Read Committed隔离级别，我们可以用骚气一点的名字，就是简称为RC个级别。

这个RC隔离级别，实际上意思就是说你事务运行期间，只要别的事务修改数据还提交了，你就是可以读到人家修改的数据的，所以是会发生不可重复读的问题，包括幻读的问题，都会有的。

那么所谓的ReadView机制，之前我们讲过，他是基于undo log版本链条实现的一套读视图机制，他意思就是说你事务生成一个ReadView，然后呢，如果是你事务自己更新的数据，自己是可以读到的，或者是在你生成ReadView之前提交的事务修改的值，也是可以读取到的。

但是如果是你生成ReadView的时候，就已经活跃的事务，在你生成ReadView之后修改了数据，接着提交了，此时你是读不到的，或者是你生成ReadView以后再开启的事务修改了数据，还提交了，此时也是读不到的。

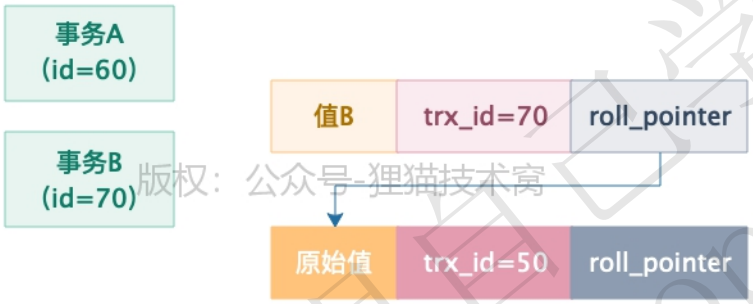
所以上面的那套机制，实际上就是ReadView机制的一个原理。好，那么既然都回顾完了，我们就来看看，如何基于ReadView机制来实现RC隔离级别呢?

其实这里的一个非常核心的要点在于，当你一个事务设置他处于RC隔离级别的时候，他是每次发起查询，都重新生成一个ReadView!

大家注意，这点是非常重要的，接着我们通过画图一步一步来给大家演示这个RC隔离级别是怎么做到的。

首先假设我们的数据库里有一行数据，是事务id=50的一个事务之前就插入进去的，然后现在呢，活跃着两个事务，一个是事务A（id=60），一个是事务B（id=70），此时如下图所示。

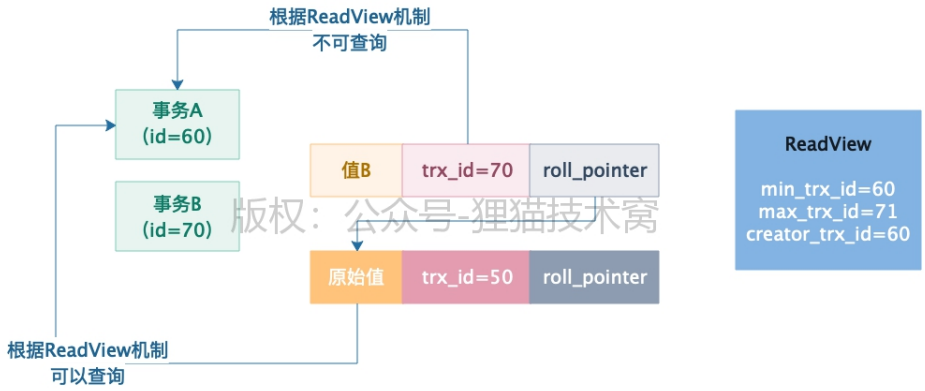
现在的情况就是，事务B发起了一次update操作，更新了这条数据，把这条数据的值修改为了值B，所以此时数据的trx_id会变为事务B的id=70，同时会生成一条undo log，由roll_pointer来指向，看下图：



这个时候，事务A要发起一次查询操作，此时他一发起查询操作，就会生成一个ReadView，此时ReadView里的min_trx_id=60，max_trx_id=71，creator_trx_id=60，此时如下图所示。

这个时候事务A发起查询，发现当前这条数据的trx_id是70。也就是说，属于ReadView的事务id范围之间，说明是他生成ReadView之前就有这个活跃的事务，是这个事务修改了这条数据的值，但是此时这个事务B还没提交，所以ReadView的m_ids活跃事务列表里，是有[60, 70]两个id的，所以此时根据ReadView的机制，此时事务A是无法查到事务B修改的值B的。

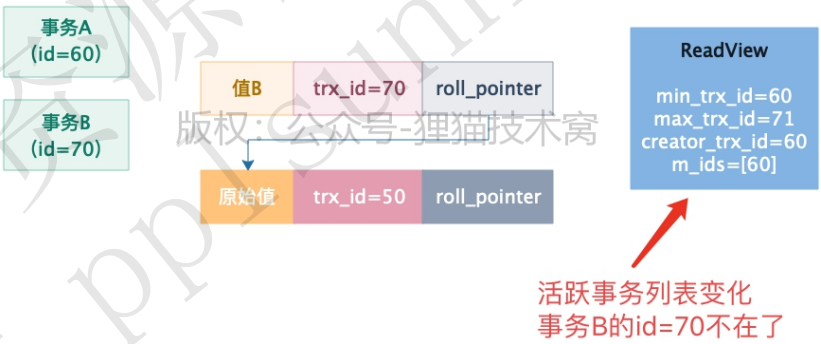
接着就顺着undo log版本链条往下查找，就会找到一个原始值，发现他的trx_id是50，小于当前ReadView里的min_trx_id，说明是他生成ReadView之前，就有一个事务插入了这个值并且早就提交了，因此可以查到这个原始值，如下图。



接着，咱们假设事务B此时就提交了，好了，那么提交了就说明事务B不会活跃于数据库里了，是不是？可以的，大家一定记住，事务B现在提交了。那么按照RC隔离级别的定义，事务B此时一旦提交了，说明事务A下次再查询，就可以读到事务B修改过的值了，因为事务B提交了。

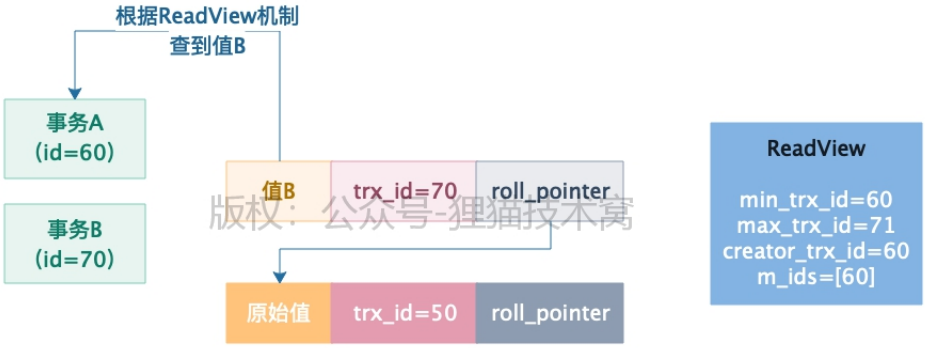
那么到底怎么让事务A能够独到提交的事务B修改过的值呢？

很简单，就是让事务A下次发起查询，再次生成一个ReadView。此时再次生成ReadView，数据库内活跃的事务只有事务A了，因此min_trx_id是60，mac_trx_id是71，但是m_ids这个活跃事务列表里，只会有一个60了，事务B的id=70不会出现在m_ids活跃事务列表里了，如下图。



此时事务A再次基于这个ReadView去查询，会发现这条数据的trx_id=70，虽然在ReadView的min_trx_id和max_trx_id范围之间，但是此时并不在m_ids列表内，说明事务B在生成本次ReadView之前就已经提交了。

那么既然在生成本次ReadView之前，事务B就已经提交了，就说明这次你查询就可以查到事务B修改过的这个值了，此时事务A就会查到值B，如下图所示。



到此为止，RC隔离级别如何实现的，大家应该就理解了，他的关键点在于每次查询都生成新的ReadView，那么如果在你这次查询之前，有事务修改了数据还提交了，你这次查询生成的ReadView里，那个m_ids列表当然不包含这个已经提交的事务了，既然不包含已经提交的事务了，那么当然可以读到人家修改过的值了。

这就是基于ReadView实现RC隔离级别的原理，希望大家好好仔细去体会，实际上，基于undo log多版本链条以及ReadView机制实现的多事务并发执行的RC隔离级别、RR隔离级别，就是数据库的MVCC多版本并发控制机制。

他本质是协调你多个事务并发运行的时候，并发的读写同一批数据，此时应该如何协调互相的可见性。

End

56 MySQL最牛的RR隔离级别，是如何基于ReadView机制实现的?

MySQL最牛的RR隔离级别，是如何基于ReadView机制实现的?

今天来接着给大家讲解，MySQL中最牛的RR隔离级别，是如何同时避免不可重复读问题和幻读问题的。

其实大家现在应该都知道，在MySQL中让多个事务并发运行的时候能够互相隔离，避免同时读写一条数据的时候有影响，是依托undo log版本链条和ReadView机制来实现的。

上次我们都讲过了，基于ReadView机制可以实现RC隔离级别，即你每次查询的时候都生成一个ReadView，这样的话，只要在你这次查询之前有别的事务提交了，那么别的事务更新的数据，你是可以看到的。

那么如果是RR级别呢？RR级别下，你这个事务读一条数据，无论读多少次，都是一个值，别的事务修改数据之后哪怕提交了，你也是看不到人家修改的值的，这就避免了不可重复读的问题。

同时如果别的事务插入了一些新的数据，你也是读不到的，这样你就可以避免幻读的问题。

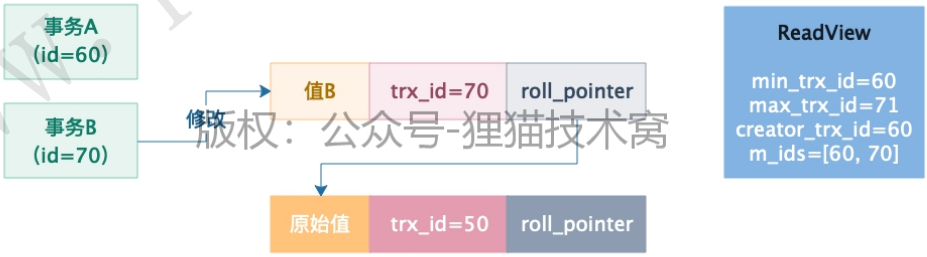
那么到底是如何实现的呢？我们今天来看看。

首先我们还是假设有一条数据是事务id=5的一个事务插入的，同时此时有事务A和事务B同时在运行，事务A的id是60，事务B的id是70，如下图所示。

这个时候，事务A发起了一个查询，他就是第一次查询就会生成一个ReadView，此时ReadView里的creator_trx_id是60，min_trx_id是60，max_trx_id是71，m_ids是[60, 70]，此时ReadView如下图所示。

这个时候事务A基于这个ReadView去查这条数据，会发现这条数据的trx_id为50，是小于ReadView里的min_trx_id的，说明他发起查询之前，早就有事务插入这条数据还提交了，所以此时可以查到这条原始值的，如下图。

接着就是事务B此时更新了这条数据的值为值B，此时会修改trx_id为70，同时生成一个undo log，而且关键是事务B此时他还提交了，也就是说此时事务B已经结束了，如下图所示。



这个时候大家思考一个问题，ReadView中的m_ids此时还会是60和70吗？

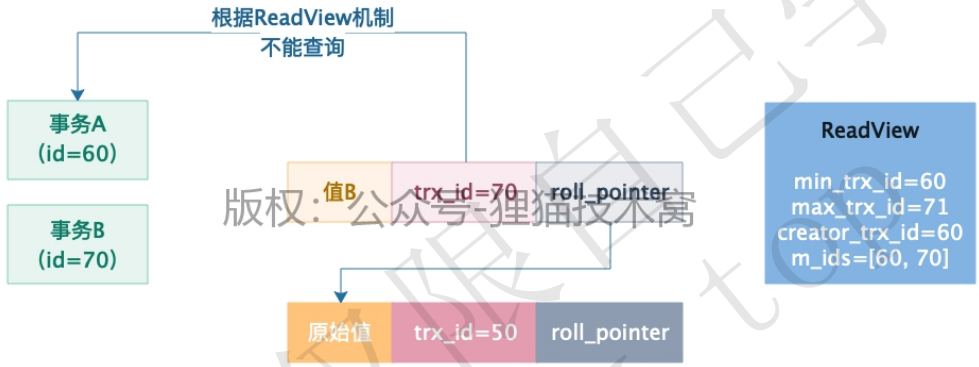
那必然是的，因为ReadView一旦生成了就不会改变了，这个时候虽然事务B已经结束了，但是事务A的ReadView里，还是会有60和70两个事务id。

他的意思其实就是，在你事务A开启查询的时候，事务B当时是在运行的，就是这个意思。

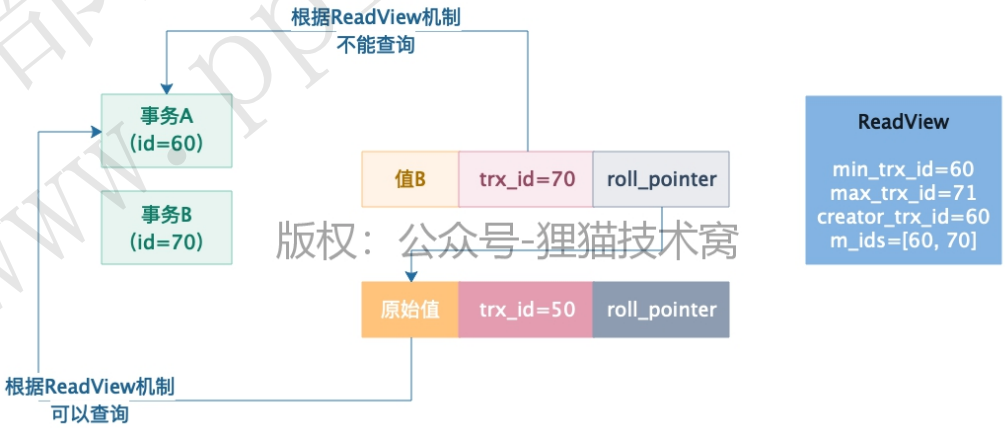
那么好，接着此时事务A去查询这条数据的值，他会惊讶的发现此时数据的trx_id是70了，70一方面是在ReadView的min_trx_id和max_trx_id的范围区间的，同时还在m_ids列表中

这说明什么？

说明起码是事务A开启查询的时候，id为70的这个事务B还是在运行的，然后由这个事务B更新了这条数据，所以此时事务A是不能查询到事务B更新的这个值的，因此这个时候继续顺着指针往历史版本链条上去找，如下图。



接着事务A顺着指针找到下面一条数据，trx_id为50，是小于ReadView的min_trx_id的，说明在他开启查询之前，就已经提交了这个事务了，所以事务A是可以查询到这个值的，此时事务A查到的是原始值，如下图。



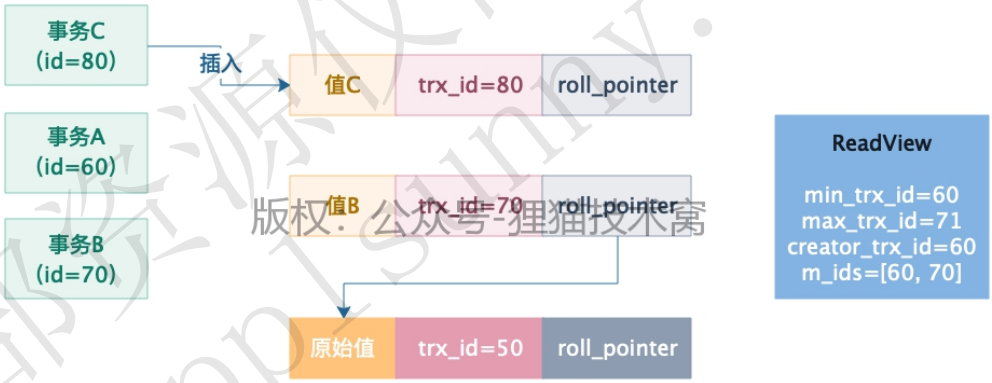
大家看到这里有什么感想？是不是感觉到这一下子就避免了不可重复读的问题？

你事务A多次读同一个数据，每次读到的都是一样的值，除非是他自己修改了值，否则读到的一直会一样的值。

不管别的事务如何修改数据，事务A的ReadView始终是不变的，他基于这个ReadView始终看到的值是一样的！

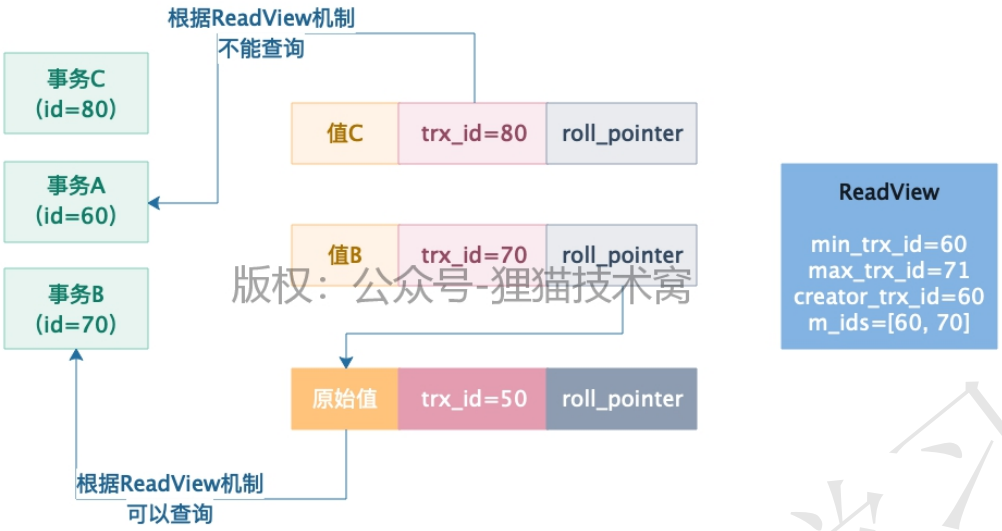
接着我们来看看幻读的问题他是如何解决的。假设现在事务A先用select * from x where id>10来查询，此时可能查到的就是一条数据，而且读到的是这条数据的原始值的那个版本，至于原因，上面都解释过了，如下图。

现在有一个事务C插入了一条数据，然后提交了，此时如下图所示。



接着，此时事务A再次查询，此时会发现符合条件的有2条数据，一条是原始值那个数据，一条是事务C插入的那条数据，但是事务C插入的那条数据的trx_id是80，这个80是大于自己的ReadView的max_trx_id的，说明是自己发起查询之后，这个事务才启动的，所以此时这条数据是不能查询的。

因此事务A本次查询，还是只能查到原始值一条数据，如下图。



所以大家可以看到，在这里，事务A根本不会发生幻读，他根据条件范围查询的时候，每次读到的数据都是一样的，不会读到人家插入进去的数据，这都是依托ReadView机制实现的！

好了，到此为止，如何基于ReadView机制实现RR隔离级别，避免不可重复读问题和幻读问题，就全部讲解清楚了，下次我们来做一个多事务并发隔离机制的总结。

End

57 停一停脚步：梳理一下数据库的多事务并发运行的隔离机制

停一停脚步，梳理一下数据库的多事务并发运行的隔离机制

今天给大家简单梳理一下MySQL中的多事务并发运行的隔离原理，其实这套隔离原理，说白了就是MVCC机制，也就是multi-version concurrent control，就是多版本并发控制机制，专门控制多个事务并发运行的时候，互相之间会如何影响。

首先我们先要明白，多个事务并发运行的时候，同时读写一个数据，可能会出现脏写、脏读、不可重复读、幻读几个问题

所谓的脏写，就是两个事务都更新一个数据，结果有一个人回滚了把另外一个人更新的数据也回滚没了。

脏读，就是一个事务读到了另外一个事务没提交的时候修改的数据，结果另外一个事务回滚了，下次读就读不到了。

不可重复读，就是多次读一条数据，别的事务老是修改数据值还提交了，多次读到的值不同。

幻读，就是范围查询，每次查到的数据不同，有时候别的事务插入了新的值，就会读到更多的数据。

针对这些问题，所以才有RU、RC、RR和串行四个隔离级别

RU隔离级别，就是可以读到人家没提交的事务修改的数据，只能避免脏写问题；

RC隔离级别，可以读到人家提交的事务修改过的数据，可以避免脏写和脏读问题。

RR是不会读到别的已经提交事务修改的数据，可以避免脏读、脏写和不可重复读的问题；

串行是让事务都串行执行，可以避免所有问题。

然后MySQL实现MVCC机制的时候，是基于**undo log多版本链条+ReadView机制**来做的，默认的RR隔离级别，就是基于这套机制来实现的，依托这套机制实现了RR级别，除了避免脏写、脏读、不可重复读，还能避免幻读问题。因此一般来说我们都用默认的RR隔离级别就好了

这就是数据库的隔离机制以及底层的原理，希望大家好好理解，可以复习一下之前的内容，把这套机制理解清楚了，接下来我们就要开始讲解锁机制了。

锁机制，解决的就是多个事务同时更新一行数据，此时必须要有一个加锁的机制

锁机制也是非常复杂的，我们接下来会用比较多的篇幅来讲清楚MySQL一套完整的锁机制，然后讲完了锁机制，就会来讲大量的实战案例了。

End

58 多个事务更新同一行数据时，是如何加锁避免脏写的?

多个事务更新同一行数据时，是如何加锁避免脏写的?

之前我们已经用很多篇幅给大家讲解了多个事务并发运行的时候，如果同时要读写一批数据，此时读和写时间的关系是如何协调的，毕竟要是你不协调好的话，可能就会有脏读、不可重复读、幻读等一系列的问题。

简单来说，脏读、不可重复读、幻读，都是别人在更新数据的时候，你怎么读的问题，读的不对，那就有问题，读的方法对了，那就不存在一系列问题了。

而你要解决一系列的数问题，其实就是依靠之前我们给大家讲的那套，基于undo log版本链条以及ReadView实现的mvcc机制。

现在开始我们接下来要用一系列的篇幅来研究另外一个问题了，那就是当有多个事务同时并发更新一行数据的时候，不就是会有脏写的问题吗?

我们之前讲过，脏写是绝对不允许的，那么这个脏写是靠什么防止的呢?

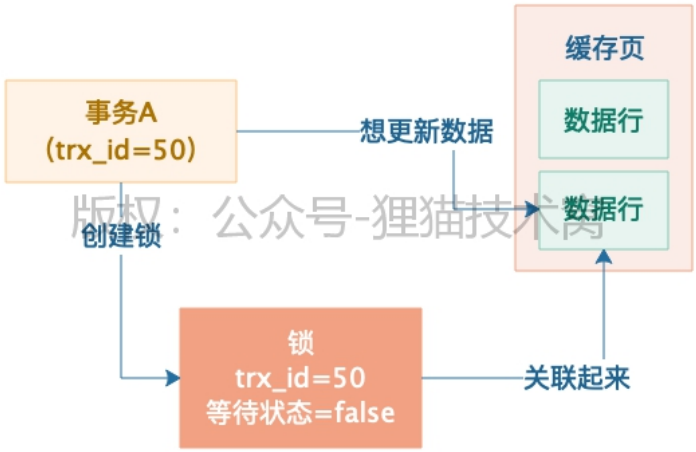
说白了，就是靠锁机制，依靠锁机制让多个事务更新一行数据的时候串行化，避免同时更新一行数据，今天我们就先对数据库的锁机制做一个初步的入门讲解。

在MySQL里，假设有一行数据摆在那儿不动，此时有一个事务来了要更新这行数据，这个时候他会先琢磨一下，看看这行数据此时有没有人加锁?

一看没人加锁，太好了，说明他是第一个人，捷足先登了。

此时这个事务就会创建一个锁，里面包含了自己的trx_id和等待状态，然后把锁跟这行数据关联在一起。

同时大家应该还记得，更新一行数据必须把他所在的数据页从磁盘文件里读取到缓存页里来才能更新的，所以说，此时这行数据和关联的锁数据结构，都是在内存里的，大家要明确这一点，如下图。



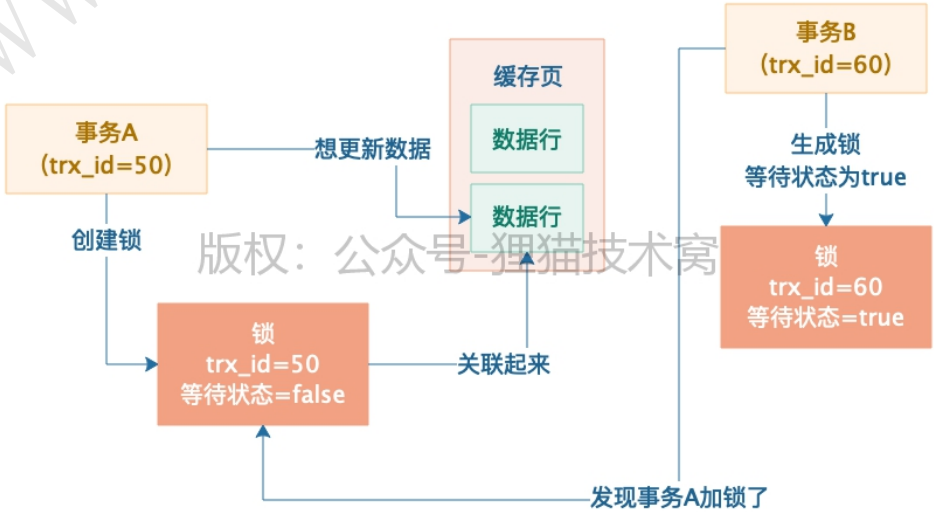
大家注意看上面的那个图，因为事务A给那行数据加了锁，所以此时就可以说那行数据已经被加锁了

那么既然被加锁了，此时就不能再让别人访问了！如果有朋友对加锁的概念不了解，可能是对编程语言不太了解，其实这个就跟Java里的加锁是一个概念。

现在呢，有另外一个事务B过来了，这个事务B就也想更新那行数据，此时就会检查一下，当前这行数据有没有别人加锁

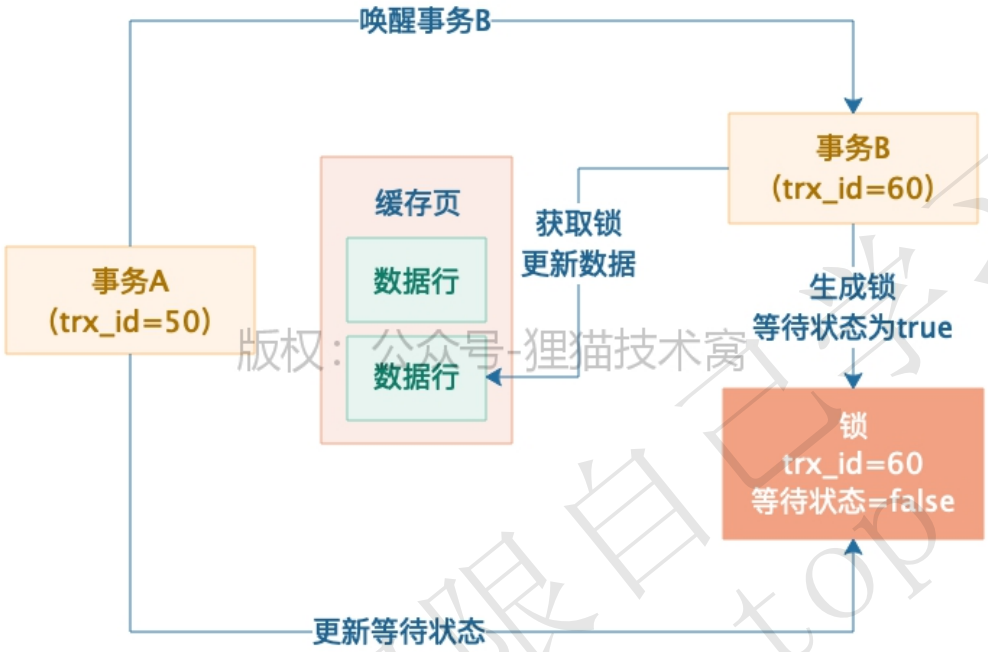
然而他一下子发现，真是糟糕啊，事务A这家伙太不地道了，居然抢先给这行数据加锁了，这怎么办呢？

事务B这个时候一想，那行，我也加个锁，然后等着排队不就得了，这个时候事务B也会生成一个锁数据结构，里面有他的trx_id，还有自己的等待状态，但是他因为是在排队等待，所以他的等待状态就是true了，意思是我在等着呢，如下图。



接着事务A这个时候更新完了数据，就会把自己的锁给释放掉了。锁一旦释放了，他就会去找，此时还有没有别人也对这行数据加锁了呢？他会发现事务B也加锁了

于是这个时候，就会把事务B的锁里的等待状态修改为false，然后唤醒事务B继续执行，此时事务B就获取到锁了，如下图。



上述就是MySQL中锁机制的一个最基本的原理，大家可以先好好理解一下，其实是跟Java里的锁机制，思路是完全类似的，从这种简单的锁里可以引申出很多其他的概念，比如读写锁，共享锁，独占锁，公平锁，非公平锁，等等。Java里的锁，也同样具备这些锁的概念。

End

59 对MySQL锁机制再深入一步，共享锁和独占锁到底是什么?

对 MySQL 锁机制再深入一步，共享锁和独占锁到底是什么?

今天我们来稍微深入的讲一下MySQL里的共享锁和独占锁这两个概念，上次我们都讲过了，其实多个事务同时更新一行数据，此时都会加锁，然后都会排队等待，必须一个事务执行完毕了，提交了，释放了锁，才能唤醒别的事务继续执行。

那么在这多个事务运行的时候，他们加的是什么锁呢?

其实是X锁，也就是Exclude独占锁，当有一个事务加了独占锁之后，此时其他事务再要更新这行数据，都是要加独占锁的，但是只能生成独占锁在后面等待。

那么这个时候我想问大家一个问题，当有人在更新数据的时候，其他的事务可以读取这行数据吗？默认情况下需要加锁吗?

答案是：不用

因为默认情况下，有人在更新数据的时候，然后你要去读取这行数据，直接默认就是开启mvcc机制的。

也就是说，此时对一行数据的读和写两个操作默认是不会加锁互斥的，因为MySQL设计mvcc机制就是为了解决这个问题，避免频繁加锁互斥。

此时你读取数据，完全可以根据你的ReadView，去在undo log版本链条里找一个你能读取的版本，完全不用去顾虑别人在不在更新。

就算你真的等他更新完毕了还提交了，基于mvcc机制你也读不到他更新的值啊！因为ReadView机制是不允许的，所以你默认情况下的读，完全不需要加锁，不需要去care其他事务的更新加锁问题，直接基于mvcc机制读某个快照就可以了。

那么假设万一要是你在执行查询操作的时候，就是想要加锁呢?

那也是ok的，MySQL首先支持一种共享锁，就是S锁，这个共享锁的语法如下：select * from table lock in share mode，你在一个查询语句后面加上lock in share mode，意思就是查询的时候对一行数据加共享锁。

如果此时有别的事务在更新这行数据，已经加了独占锁了，此时你的共享锁能加吗?

当然不行了，共享锁和独占锁是互斥的！此时你这个查询就只能等着了。

那么如果你先加了共享锁，然后别人来更新要加独占锁行吗？当然不行了，此时锁是互斥的，他只能等待。

那么如果你在加共享锁的时候，别人也加共享锁呢？此时是可以的，你们俩都是可以加共享锁的，共享锁和共享锁是不会互斥的。

所以这里可以先看出一个规律，就是更新数据的时候必然加独占锁，独占锁和独占锁是互斥的，此时别人不能更新；但是此时你要查询，默认是不加锁的，走mvcc机制读快照版本，但是你查询是可以手动加共享锁的，共享锁和独占锁是互斥的，但是共享锁和共享锁是不互斥的，如下规律。

锁类型	独占锁	共享锁
独占锁	互斥	互斥
共享锁	互斥	不互斥

不过说实话，一般开发业务系统的时候，其实你查询主动加共享锁，这种情况较为少见，数据库的行锁是实用功能，但是一般不会在数据库层面做复杂的手动加锁操作，反而会用基于redis/zookeeper的分布式锁来控制业务系统的锁逻辑。

另外就是，查询操作还能加互斥锁，他的方法是：select * from table for update。

这个意思就是，我查出来数据以后还要更新，此时我加独占锁了，其他闲杂人等，都不要更新这个数据了。

一旦你查询的时候加了独占锁，此时在你事务提交之前，任何人都不能更新数据了，只能你在本事务里更新数据，等你提交了，别人再更新数据。

这一讲内容，就是给大家讲了默认情况下更新数据的独占锁，默认情况下查询数据的mvcc机制读快照，然后通过查询加共享锁和独占锁的方式，共享锁和独占锁之间的互斥规则，大家都理解了就好。

End

60 在数据库里，哪些操作会导致在表级别加锁呢？

在数据库里，哪些操作会导致在表级别加锁呢？

之前我们已经给大家讲解了数据库里的行锁的概念，其实还是比较简单，容易理解的，因为在讲解锁这个概念之前，对于多事务并发以及隔离，我们已经深入讲解过了，所以大家应该很容易在脑子里有一个多事务并发执行的概念。

在多个事务并发更新数据的时候，都是要在行级别加独占锁的，这就是行锁，独占锁都是互斥的，所以不可能发生脏写问题，一个事务提交了才会释放自己的独占锁，唤醒下一个事务执行。

如果你此时去读取别的事务在更新的数据，有两种可能：

- 第一种可能是基于mvcc机制进行事务隔离，读取快照版本，这是比较常见的；
- 第二种可能是查询的同时基于特殊语法去加独占锁或者共享锁。

如果你查询的时候加独占锁，那么跟其他更新数据的事务加的独占锁都是互斥的；如果你查询的时候加共享锁，那么跟其他查询加的共享锁是不互斥的，但是跟其他事务更新数据就加的独占锁是互斥的，跟其他查询加的独占锁也是互斥的。

当然一般我个人从多年研发经验而言，不是太建议在数据库粒度去通过行锁实现复杂的业务锁机制，而更加建议通过redis、zookeeper来用分布式锁实现复杂业务下的锁机制，其实更为合适一些。

为什么呢？因为如果你把分布式系统里的复杂业务的一些锁机制依托数据库查询的时候，在SQL语句里加共享锁或者独占锁，会导致这个加锁逻辑隐藏在SQL语句里，在你的Java业务系统层面其实是非常的不好维护的，所以一般是不建议这么做的。

比较正常的情况而言，其实还是多个事务并发运行更新一条数据，默认加独占锁互斥，同时其他事务读取基于mvcc机制进行快照版本读，实现事务隔离。

今天我们要给大家在讲完行锁之后，继续讲一个新的概念，就是表级锁。

在数据库里，你不光可以通过查询中的特殊语法加行锁，比如lock in share mode、for update等等，还可以通过一些方式在表级别去加锁。

有些人可能会以为当你执行增删改的时候默认加行锁，然后执行DDL语句的时候，比如alter table之类的语句，会默认在表级别加表锁。这么说也不太正确，但是也有一定的道理，因为确实你执行DDL的时候，会阻塞所有增删改操作；执行增删改的时候，会阻塞DDL操作。

但这是通过MySQL通用的元数据锁实现的，也就是Metadata Locks，但这还不是表锁的概念。因为表锁其实是InnoDB存储引擎的概念，InnoDB存储引擎提供了自己的表级锁，跟这里DDL语句用的元数据锁还不是一个概念。

只不过DDL语句和增删改操作，确实是互斥的，大家要知道这一点。

今天讲到这里，其实就是先给大家提一下表级锁的概念，同时梳理清楚DDL之类的语句是跟增删改操作互斥的，大家先理解到这个点就好

下一讲我们继续聊表级锁这个概念，说一下具体如何加锁，表级锁之间是如何互斥的。

End

61 表锁和行锁互相之间的关系以及互斥规则是什么呢?

表锁和行锁互相之间的关系以及互斥规则是什么呢?

今天我们接着讲，MySQL里是如何加表锁的。这个MySQL的表锁，其实是极为鸡肋的一个东西，几乎一般很少会用到，表锁分为两种，一种就是表锁，一种是表级的意向锁，我们分别来看看。

首先说表锁，这个表锁，可以用如下语法来加：

LOCK TABLES xxx READ：这是加表级共享锁

LOCK TABLES xxx WRITE：这是加表级独占锁

其实一般来讲，几乎没人会用这两个语法去加表锁，这不是纯属没事儿找事儿么，所以才说表锁特别的鸡肋。

还有就是有另外两个情况会加表级锁。如果有事务在表里执行增删改操作，那在行级会加独占锁，此时其实同时会在表级加一个意向独占锁；如果有事务在表里执行查询操作，那么会在表级加一个意向共享锁。

其实平时我们操作数据库，比较常见的两种表锁，反而是更新和查询操作加的意向独占锁和意向共享锁，但是这个意向独占锁和意向共享锁，大家暂时可以当他是透明的就可以了，因为两种意向锁根本不会互斥。

为啥呢？因为假设有一个事务要在表里更新id=10的一行数据，在表上加了一个意向独占锁，此时另外一个事务要在表里更新id=20的一行数据，也会在表上加一个意向独占锁，你觉得这两把锁应该互斥吗?

明显是不应该互斥的啊，因为他们俩更新的都是表里不同的数据，你让他们俩在表上加的意向独占锁互斥干什么呢？所以意向锁之间是根本不会互斥的。

同理，假设一个事务要更新表里的数据，在表级加了一个意向独占锁，另外一个事务要在表里读取数据，在表级加了一个意向共享锁，此时你觉得表级的意向独占锁和意向共享锁应该互斥吗?

当然不应该了！一个人要更新数据，一个人要读取数据，俩人在表上加的意向锁，凭什么要互斥？没天理啊！

所以说，大家有没有发现一点，这个所谓的表级的意向独占锁和意向共享锁，似乎是跟脱了裤子放屁一样，多此一举？

但是我们接下来就要给大家讲讲，手动加表级共享锁和独占锁，以及更新和查询的时候自动在表级加的意向共享锁和意向独占锁，他们之间反而是有一定的互斥关系，关系如下表所示。

锁类型	独占锁	意向独占锁	共享锁	意向共享锁
独占锁	互斥	互斥	互斥	互斥
意向独占锁	互斥	不互斥	互斥	不互斥
共享锁	互斥	互斥	不互斥	不互斥
意向共享锁	互斥	不互斥	不互斥	不互斥

大家看看上面表格，仔细看一下，上面说的是在表上面手动加的独占锁和共享锁，以及更新数据和查询数据默认自动加的意向独占锁和意向共享锁，他们互相之间的互斥关系，大家一看就明白了。

其实更新数据自动加的表级意向独占锁，会跟你用 LOCK TABLES xxx WRITE 手动加的表级独占锁是互斥的，所以说，假设你手动加了表级独占锁，此时任何人都不能执行更新操作了！

或者你用LOCK TABLES xxx WRITE手动加了表级共享锁，此时任何人也不能执行更新操作了，因为更新就要加意向独占锁，此时是跟你手动加的表级共享锁，是互斥的！

具体其他实例就不举了，大家看看上面的表格就知道了，你如果手动加了表级的共享锁或者独占锁，此时是会阻塞掉其他事务的一些正常的读写操作的，因为跟他们自动加的意向锁都是互斥的。

但是说实话，这一讲也就是给你讲明白这个表级锁如何加的，如何互斥的，其实一般来说，根本就不会手动加表级锁，所以一般来说读写操作自动加的表级意向锁，互相之间绝对不会互斥。

一般来讲，都是对同一行数据的更新操作加的行级独占锁是互斥，跟读操作都是不互斥的，读操作默认都是走mvcc机制读快照版本的！

End

62 案例实战：线上数据库不确定性的性能抖动优化实践（上）

案例实战：线上数据库不确定性的性能抖动优化实践（上）

之前我们花费了很大篇幅来给大家深入和细致的讲解数据库在执行增删改这类更新语句时候的底层原理，这里涉及到了很多数据库内核级的概念，比如buffer pool、redo log buffer、lru/flush链表，等等，大家对数据库执行更新语句的原理都有了较为深入的理解。

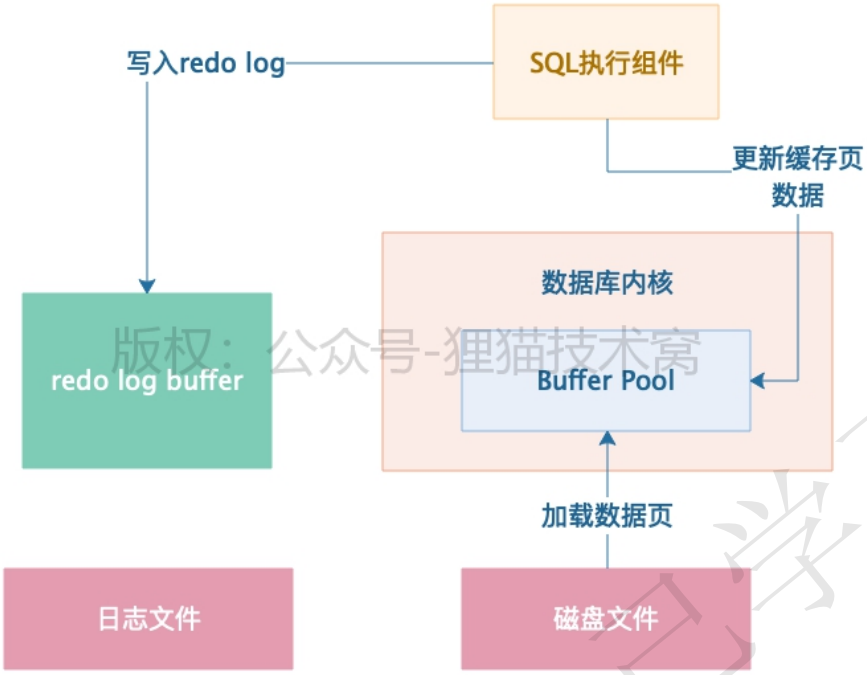
同时我们还在这个基础之上，给大家深入分析了数据库的事务的底层原理，包括事务隔离和mvcc机制的原理和概念，以及多事务并发运行时的锁机制，如何让多个事务合理的在数据库内部并发执行，同时读写共享的数据。

接着我们就要给大家讲解关于数据库更新这块的一些生产实践案例了，主要会讲解数据库更新时候的性能抖动优化、各种奇葩的锁导致性能降低的问题以及死锁问题，包括删库跑路、数据丢失等问题。

相信大家学习完接下来这部分内容之后，对自己日常工作中，线上数据库出现的一些数据更新导致的锁、数据丢失等生产故障，都能自己进行排查、定位和解决了，也就达到了我们希望的大家通过学习专栏掌握生产级优化能力的初衷。

今天我们要给大家讲解的第一个生产案例，就是线上数据库时不时莫名其妙的来一次性能抖动的问题，而且造成性能抖动的还不是之前我们讲过的数据库锂电池充放电的问题，而是另外一个新的问题，跟我们之前讲解的原理是息息相关的。

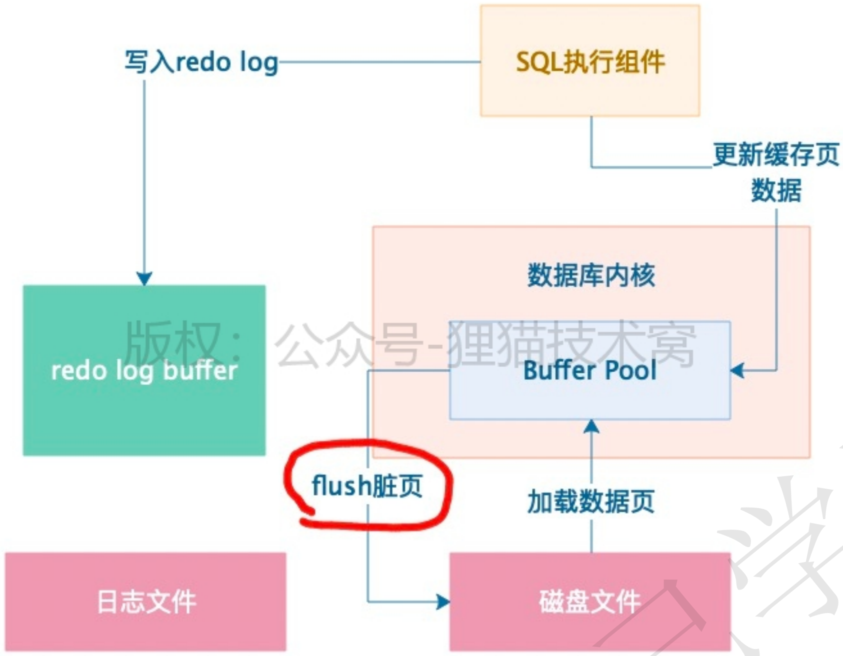
大家都知道一件事情，那就是我们平时在数据库里执行的更新语句，实际上都是从磁盘上加载数据页到数据库内存的缓存页里来，接着就直接更新内存里的缓存页，同时还更新对应的redo log写入一个buffer中，如下图所示。



那么大家都知道，既然我们更新了Buffer Pool里的缓存页，缓存页就会变成脏页，之所以说他是脏页，就是因为缓存页里的数据目前跟磁盘文件里的数据页的数据是不一样的，所以此时叫缓存页是脏页。

既然是脏页，那么就必然得有一个合适的时机要把那脏页给刷入到磁盘文件里去，之前我们其实就仔细分析过这个脏页刷入磁盘的机制，他是维护了一个lru链表来实现的，通过lru链表，他知道哪些缓存页是最近经常被使用的。

那么后续如果你要加载磁盘文件的数据页到buffer pool里去了，但是此时并没有空闲的缓存页了，此时就必须要把部分脏缓存页刷入到磁盘里去，此时就会根据lru链表找那些最近最少被访问的缓存页去刷入磁盘，如下图所示。



那么万一要是你要执行的是一个查询语句，需要查询大量的数据到缓存页里去，此时就可能导致内存里大量的脏页需要淘汰出去刷入磁盘上，才能腾出足够的内存空间来执行这条查询语句。

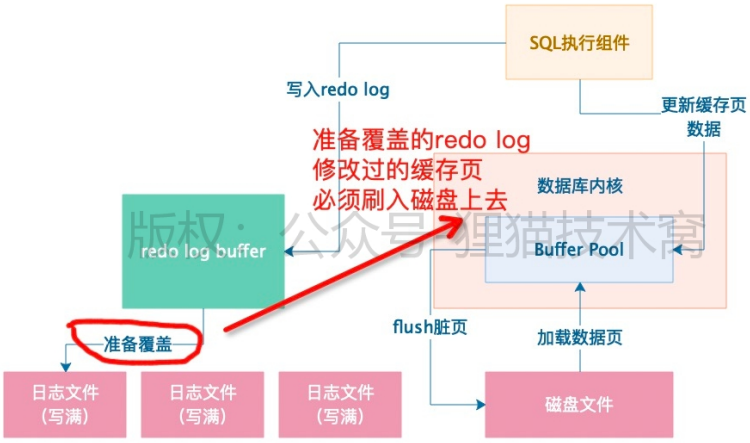
在这种情况下，可能你会发现突然莫名其妙的线上数据库执行某个查询语句就一下子性能出现抖动，平时只要几十毫秒的查询语句，这次一下子要几秒都有可能，毕竟你要等待大量脏页flush到磁盘，然后语句才能执行！

另外还有一种脏页刷磁盘的契机，之前我们并没有给大家提到，就是大家都知道redo log buffer里的redo log本身也是会随着各种条件刷入磁盘上的日志文件的，比如redo log buffer里的数据超过容量的一定比例了，或者是事务提交的时候，都会强制buffer里的redo log刷入磁盘上的日志文件。

然后我们也知道，磁盘上是有多个日志文件的，他会依次不停的写，如果所有日志文件都写满了，此时会重新回到第一个日志文件再次写入，这些日志文件是不停的循环写入的，所以其实在日志文件都被写满的情况下，也会触发一次脏页的刷新。

为什么呢？因为假设你的第一个日志文件的一些redo log对应的内存里的缓存页的数据都没被刷新到磁盘上的数据页里去，那么我问你，一旦你把第一个日志文件里的这部分redo log覆盖写了别的日志，那么此时万一你数据库崩溃，是不是有些你之前更新过的数据就彻底丢失了？

所以一旦你把所有日志文件写满了，此时重新从第一个日志文件开始写的时候，他会判断一下，如果要是你第一个日志文件里的一些redo log对应之前更新过的缓存页，迄今为止都没刷入磁盘，那么此时必然是要把 那些马上要被覆盖的redo log更新的缓存页都刷入磁盘的，如下图。



尤其是在这一种刷脏页的情况下，因为redo log所有日志文件都写满了，此时会导致数据库直接hang死，无法处理任何更新请求，因为执行任何一个更新请求都必须要写redo log，此时你需要刷新一些脏页到磁盘，然后才能继续执行更新语句，把更新语句的redo log从第一个日志文件开始覆盖写。

所以此时假设你在执行大量的更新语句，可能你突然发现线上数据库莫名其妙的很多更新语句短时间内性能都抖动了，可能很多更新语句平时就几毫秒就执行好了，这次要等待1秒才能执行完毕。

因此遇到这种情况，你必须要等待第一个日志文件里部分redo log对应的脏页都刷入磁盘了，才能继续执行更新语句，此时必然会导致更新语句的性能很差。

所以综上所述，导致线上数据库的查询和更新语句莫名其妙出现性能抖动，其实就很可能是上述两种情况导致的执行语句时大量脏缓存页刷入磁盘，你要等待他们刷完磁盘才能继续执行导致的。

下一次我们继续讲解，针对上述两种情况的数据库性能抖动，应该如何来优化数据库的参数配置，来解决上述的性能问题。

End

63 案例实战：线上数据库莫名其妙的随机性能抖动优化（下）

案例实战：线上数据库莫名其妙的随机性能抖动优化（下）

上一篇文章我们已经给大家详细分析了有时候我们在数据库里执行查询或者更新语句的时候，可能SQL语句的性能会出现不正常的莫名其妙的抖动，平时可能几十毫秒搞定的，现在居然要几秒钟。

其实这种莫名其妙的性能抖动，我们在分析过底层的原理之后，就理解的很清楚了，根本原因还是两个。

第一个，可能buffer pool的缓存页都满了，此时你执行一个SQL查询很多数据，一下子要把很多缓存页flush到磁盘上去，刷磁盘太慢了，就会导致你的查询语句执行的很慢。

因为你必须等很多缓存页都flush到磁盘了，你才能执行查询从磁盘把你需要的数据页加载到buffer pool的缓存页里来。

第二个，可能你执行更新语句的时候，redo log在磁盘上的所有文件都写满了，此时需要回到第一个redo log文件覆盖写，覆盖写的时候可能就涉及到第一个redo log文件里有很多redo log日志对应的更新操作改动了缓存页，那些缓存页还没flush到磁盘，此时就必须把那些缓存页flush到磁盘，才能执行后续的更新语句，那你这么一等待，必然会导致更新执行的很慢了。

所以上述两个场景导致的大量缓存页flush到磁盘，就会导致莫名其妙的SQL语句性能抖动了

那今天我们来说说怎么**尽可能优化MySQL的一些参数，**减少这种缓存页flush到磁盘带来的性能抖动问题。

其实大家可以想一下，如果要尽量避免缓存页flush到磁盘可能带来的性能抖动问题，那么核心的就两点

第一个是尽量减少缓存页flush到磁盘的频率，第二个是尽量提升缓存页flush到磁盘的速度。

那你想要减少缓存页flush到磁盘的频率，这个是很困难的，因为平时你的缓存页就是正常的在被使用，迟早会被填满，一旦填满，必然你执行下一个SQL会导致一批缓存页flush到磁盘，这个很难控制，除非你给你的数据库采用大内存机器，给buffer pool分配的内存空间大一些，那么他缓存页填满的速率低一些，flush磁盘的频率也会比较低。

所以今天我们主要是给讲解第二个问题的优化，就是如何尽量提升缓存页flush到磁盘的速度

给大家举个例子，假设你现在要执行一个SQL查询语句，此时需要等待flush一批缓存页到磁盘，接着才能加载查询出来的数据到缓存页。

那么如果flush那批缓存页到磁盘需要1s，然后SQL查询语句自己执行的时间是200ms，此时你这条SQL执行完毕的总时间就需要1.2s了。

但是如果你把那批缓存页flush到磁盘的时间优化到100ms，然后加上SQL查询自己执行的200ms，这条SQL的总执行时间就只要300ms了，性能就提升了很多。

所以这里一个关键之一，就是要尽可能减少flush缓存页到磁盘的时间开销到最小。

如果要做到这一点，通常给大家的一个建议就是对于数据库部署的机器，一定要采用SSD固态硬盘，而不要使用机械硬盘，因为SSD固态硬盘最强大的地方，就是他的随机IO性能非常高。

而flush缓存页到磁盘，就是典型的随机IO，需要在磁盘上找到各个缓存页所在的随机位置，把数据写入到磁盘里去。所以如果你采用的是SSD固态硬盘，那么你flush缓存页到磁盘的性能首先就会提高不少。

其次，光是用SSD还不够，因为你还得设置一个很关键的参数，就是数据库的innodb_io_capacity，这个参数是告诉数据库采用多大的IO速率把缓存页flush到磁盘里去的。

举个例子，假设你SSD能承载的每秒随机IO次数是600次，结果呢，你把数据库的innodb_io_capacity就设置为了300，也就是flush缓存页到磁盘的时候，每秒最多执行300次随机IO，那你不是速度很慢么，而且根本没把你的SSD固态硬盘的随机IO性能发挥出来！

所以通常都会建议大家对数据库部署机器的SSD固态硬盘能承载的最大随机IO速率做一个测试，这个可以使用fio工具来测试

fio工具是一种用于测试磁盘最大随机IO速率的linux上的工具，如何使用，大家可以网上搜一下，非常的简单。

查出来SSD固态硬盘的最大随机IO速率之后，就知道他每秒可以执行多少次随机IO，此时你把这个数值设置给数据库的innodb_io_capacity这个参数就可以了，尽可能的让数据库用最大速率去flush缓存页到磁盘。

但是实际flush的时候，其实他会按照innodb_io_capacity乘以一个百分比来进行刷磁盘，这个百分比就是脏页的比例，是innodb_max_dirty_pages_pct参数控制的，默认是75%，这个一般不用动，另外这个比例也有可能会变化，这个比例同时会参考你的redo log日志来计算，但是这个细节大家不用关注了。

其实比例不比例的，我们这里优化不用关注，核心就是把innodb_io_capacity调整为SSD固态硬盘的IOPS也就是随机IO速率就可以了。

另外还有一个参数，是innodb_flush_neighbors，他意思是说，在flush缓存页到磁盘的时候，可能会控制把缓存页临近的其他缓存页也刷到磁盘，但是这样有时候会导致flush的缓存页太多了。

实际上如果你用的是SSD固态硬盘，并没有必要让他同时刷邻近的缓存页，可以把innodb_flush_neighbors参数设置为0，禁止刷临近缓存页，这样就把每次刷新的缓存页数量降低到最少了。

所以呢，针对这次讲的这个案例，就是MySQL性能随机抖动的问题，**最核心**的就是把innodb_io_capacity设置为SSD固态硬盘的IOPS，让他刷缓存页尽量快，同时设置innodb_flush_neighbors为0，让他每次别刷临近缓存页，减少要刷缓存页的数量，这样就可以把刷缓存页的性能提升到最高。

同时也可以尽可能降低每次刷缓存页对执行SQL语句的影响。

End

64 深入研究索引之前，先来看看磁盘数据页的存储结构

深入研究索引之前，先来看看磁盘数据页的存储结构

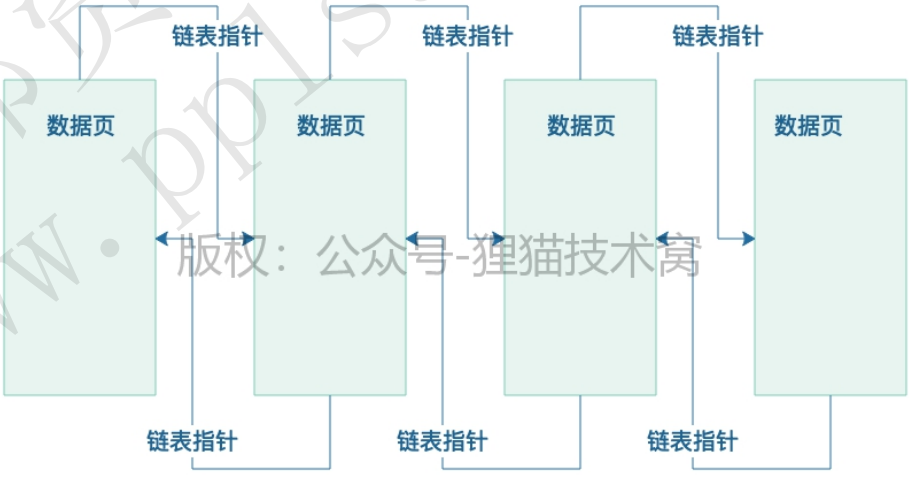
前面我们已经给大家把MySQL数据库的部分内核原理，更新语句的执行原理，事务原理以及锁原理，都初步的讲给大家听了，同时还穿插了一些相关的数据库性能优化的案例，相信现在大家已经对数据库执行增删改语句的原理有了较为深入的理解了。

接着我们就应该进入比较关键的一个环节，也是很多 人都很期盼的一个环节，就是**数据库的索引原理以及查询原理**，学完了这块，我们就可以学习大量的实战案例，包括索引设计案例，查询调优案例。

但是今天在深入研究索引之前，我们需要先来看看磁盘上的数据文件中的数据页的物理存储结构，因为后续研究索引的物理存储结构以及使用原理的时候，都是跟数据页的物理存储结构是有很大关联的。

其实之前大家都知道，数据库最终所有的数据（包括我们建的各种表以及表里的数据）都是要存放在磁盘上的文件里的，然后在文件里存放的物理格式就是数据页，那么大量的数据页在磁盘文件里是怎么存储的呢？

首先大家要明白的一点是，大量的数据页是按顺序一页一页存放的，然后两两相邻的数据页之间会采用双向链表的格式互相引用，大致看起来如下图所示。



但是可能有人看到上图就想问了，你画的这个图在磁盘文件里到底是怎么弄出来的啊？

其实一个数据页在磁盘文件里就是一段数据，可能是二进制或者别的特殊格式的数据，然后数据页里包含两个指针，一个指针指向自己上一个数据页的物理地址，一个指针指向自己下一个数据页的物理地址，大概可以认为类似下面这样。

DataPage: xx=xx, xx=xx, linked_list_pre_pointer=15367, linked_list_next_pointer=34126 ||
DataPage: xx=xx, xx=xx, linked_list_pre_pointer=23789, linked_list_next_pointer=46589 ||
DataPage: xx=xx, xx=xx, linked_list_pre_pointer=33198, linked_list_next_pointer=55681

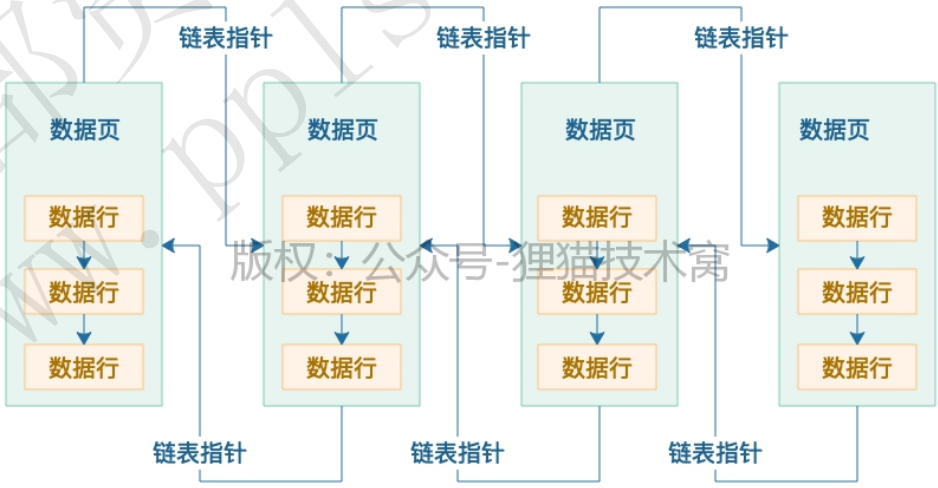
上面那段示例数据，当然不能完全认为是MySQL数据库的磁盘文件里的存储格式，但是我这里就是给你看一些类似的东西，其实MySQL实际存储大致也是类似这样的，就是每个数据页在磁盘文件里都是连续的一段数据。

然后每个数据页里，可以认为就是DataPage打头一直到 || 符号的一段磁盘里的连续的数据，你可以认为每一个数据页就是磁盘文件里这么一段连续的东西。

然后每个数据页，都有一个指针指向自己上一个数据页在磁盘文件里的起始物理位置，比如linked_list_pre_pointer=15367，就是指向了上一个数据页在磁盘文件里的起始物理位置，那个15367可以认为就是在磁盘文件里的position或者offset，同理，也有一个指针指向自己下一个数据页的物理位置。

现在你再回头看一下上面那个图，是不是就理解了一个磁盘文件里的多个数据页是如何通过指针组成一个双向链表的！

然后一个数据页内部会存储一行一行的数据，也就是平时我们在一个表里插入的一行一行的数据就会存储在数据页里，然后数据页里的每一行数据都会按照主键大小进行排序存储，同时每一行数据都有指针指向下一行数据的位置，组成单向链表，如下图。



好了，今天我们就把数据页在磁盘文件里的物理存储结构详细讲解了一下，包括数据页内部的物理存储结构，都讲了一下，接下来参照这个物理结构，就给大家说一下没有索引的时候，是如何查找数据的。

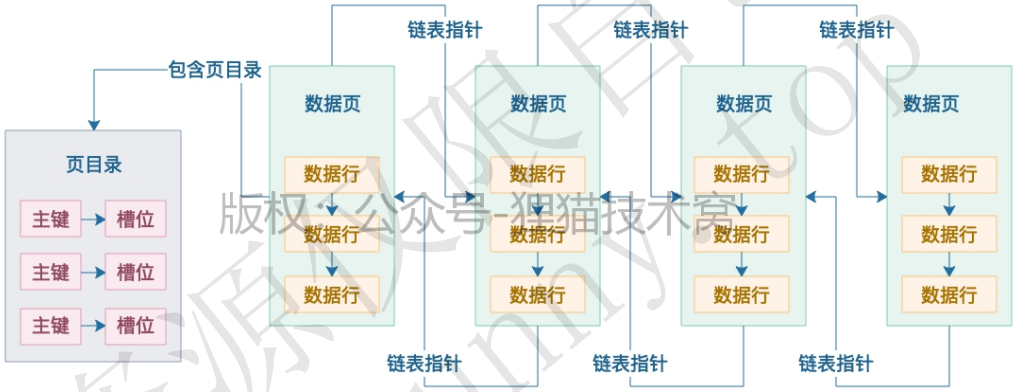
End

65 假设没有任何索引，数据库是如何根据查询语句搜索数据的?

假设没有任何索引，数据库是如何根据查询语句搜索数据的?

上一次我们给大家讲解了数据页在磁盘文件中的物理存储结构，大家应该目前都知道数据页之间是组成双向链表的，然后数据页内部的数据行是组成单向链表的，而且数据行是根据主键从小到大排序的。

然后每个数据页里都会有一个页目录，里面根据数据行的主键存放了一个目录，同时数据行是被分散存储到不同的槽位里去的，所以实际上每个数据页的目录里，就是这个页里每个主键跟所在槽位的映射关系，如下图所示。



所以假设你要根据主键查找一条数据，而且假设此时你数据库里那个表就没几条数据，那个表总共就一个数据页，那么就太简单了！首先就会先到数据页的页目录里根据主键进行二分查找（PS：不知道二分查找是什么的同学，建议去网上查一下，这是大学最基础算法）

然后通过二分查找在目录里迅速定位到主键对应的数据是在哪个槽位里，然后到那个槽位里去，遍历槽位里每一行数据，就能快速找到那个主键对应的数据了。每个槽位里都有一组数据行，你就是在里面遍历查找就可以了。

但是假设你要是根据非主键的其他字段查找数据呢?

那就尴尬了，此时你是没办法使用主键的那种页目录来二分查找的，只能进入到数据页里，根据单向链表依次遍历查找数据了，这就性能很差了。

好，那么现在假如我们有很多数据页呢?

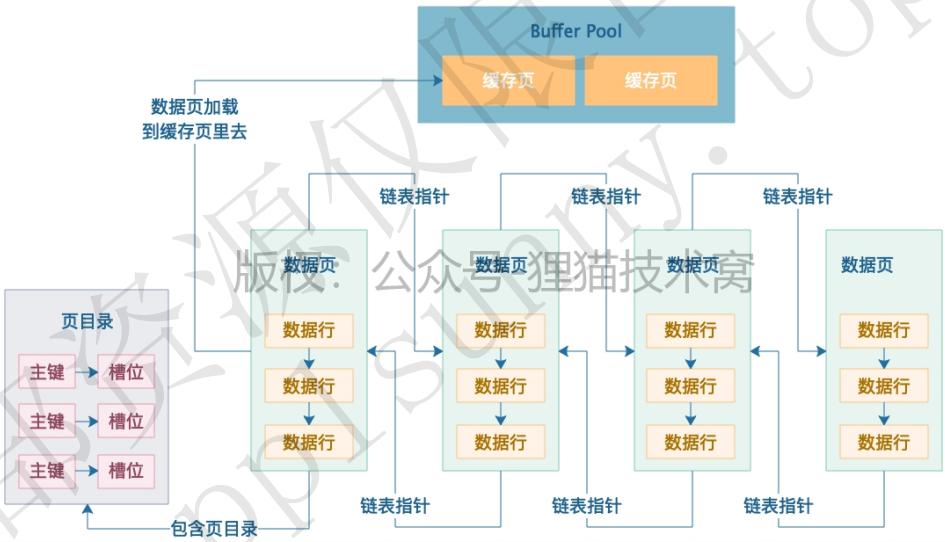
对了，一个表里往往都是有大量数据的，可能有多达成百上千个数据页，这些数据页就存放在物理磁盘文件里

所以此时是如何查询数据的呢？

之前就有不少同学在后台评论区提问过这个问题，这里我们可以先给大家解释一下，假设你要是没有建立任何索引，那么无论是根据主键查询，还是根据其他字段来条件查询，实际上都没有什么取巧的办法。

你一个表里所有数据页都是组成双向链表的吧？好，有链表就好办了，直接从第一个数据页开始遍历所有数据页，从第一个数据页开始，你得先把第一个数据页从磁盘上读取到内存buffer pool的缓存页里来。

然后你就在第一个数据页对应的缓存页里，按照上述办法查找，假设是根据主键查找的，你可以在数据页的页目录里二分查找，假设你要是根据其他字段查找的，只能是根据数据页内部的单向链表来遍历查找，如下图。



那么假设如上图所示，假设第一个数据页没找到你要的那条数据呢？

没办法，只能根据数据页的双向链表去找下一个数据页，然后读取到buffer pool的缓存页里去，然后按一样的方法在一个缓存页内部查找那条数据。

如果依然还是查找不到呢？

那只能根据双向链表继续加载下一个数据页到缓存页里来了，以此类推，循环往复。

不知道大家看到这个过程有什么感想没有？有没有觉得，你似乎是在做一个数据库里很尴尬的操作：全表扫描？

对了，其实上述操作过程，就是全表扫描，在你没有任何索引数据结构的时候，无论如何查找数据，说白了都是一个全表扫描的过程，就是根据双向链表依次把磁盘上的数据页加载到缓存页里去，然后在一个缓存页内部来查找那条数据。

最坏的情况下，你就得把所有数据页里的每条数据都得遍历一遍，才能找到你需要的那条数据，这就是全表扫描！

所以大家看完今天这篇文章，接下来我们才能正式进入索引的讲解，你才能体会到有了索引之后，是如何提升数据库的查询效率和性能的！

End

66 不断在表中插入数据时，物理存储是如何进行页分裂的?

不断在表中插入数据时，物理存储是如何进行页分裂的?

上回我们讲到了数据页的物理存储结构，数据页之间是组成双向链表的，数据页内部的数据行是组成单向链表的，每个数据页内根据主键做了一个页目录

然后一般来说，你没有索引的情况下，所有的数据查询，其实在物理层面都是全表扫描，依次扫描每个数据页内部的每个数据行。

上述描述，其实就是没有索引情况下在一个表中的数据查询情况，这个速度可以说是慢到惊人，所以一般肯定是不能让查询走全表扫描的。因此正常在数据库中的查询，必须要运用到索引来加速查询的执行。

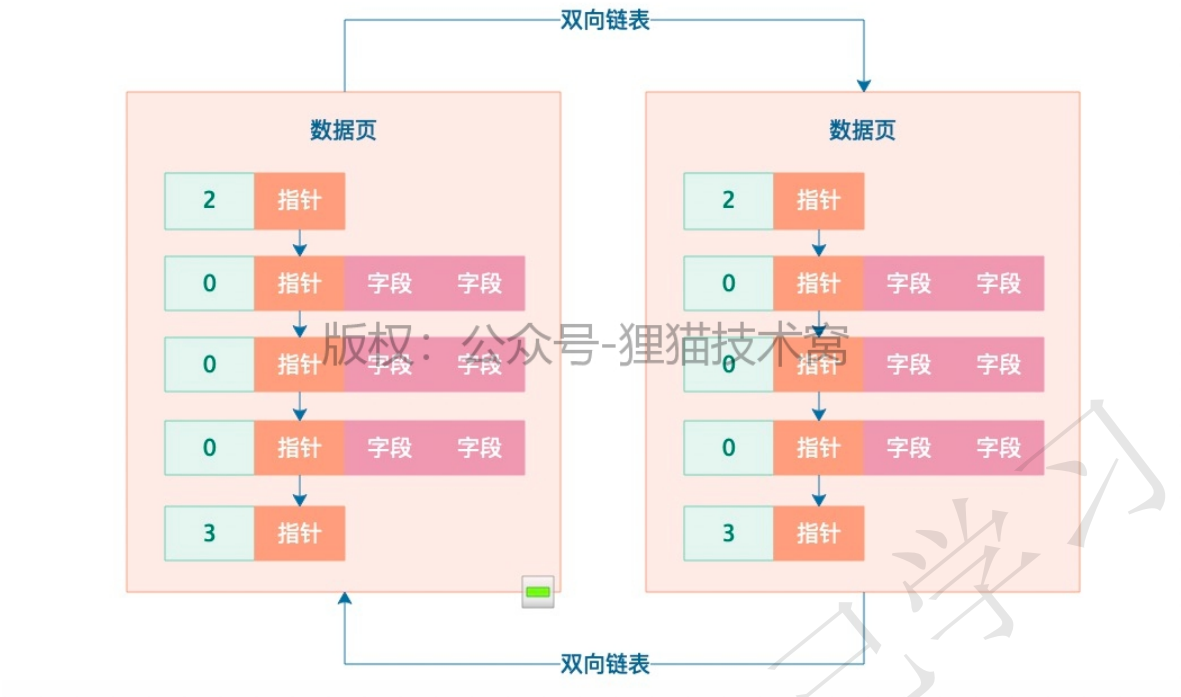
但是今天还是没法直接切入到索引这块内容，因为作为前置知识，今天还得给大家讲解另外一个知识点，就是我们在一个表里不停的插入数据的时候，会涉及到一个页分裂的过程，也就是说，这个表里是如何出现一个又一个的数据页的。

大家都知道，正常情况下我们在一个表里插入一些数据后，他们都会进入到一个数据页里去，在数据页内部，他们会组成一个单向链表，这个数据页内部的单向链表大致如下所示，我们看看

大家看上面的图，里面就是一行一行的数据，刚开始第一行是个起始行，他的行类型是2，就是最小的一行，然后他有一个指针指向了下一行数据，每一行数据都有自己每个字段的值，然后每一行通过一个指针不停的指向下一行数据，普通的数据行的类型都是0，最后一行是一个类型为3的，就是代表最大的一行。

上面就是一个典型的数据页内部的情况，那么今天要讲的页分裂是什么意思呢？

是这样的，假设你不停的在表里插入数据，那么刚开始是不是就是不停的在一个数据页插入数据？接着数据越来越多，越来越多，此时就要再搞一个数据页了，如下图。



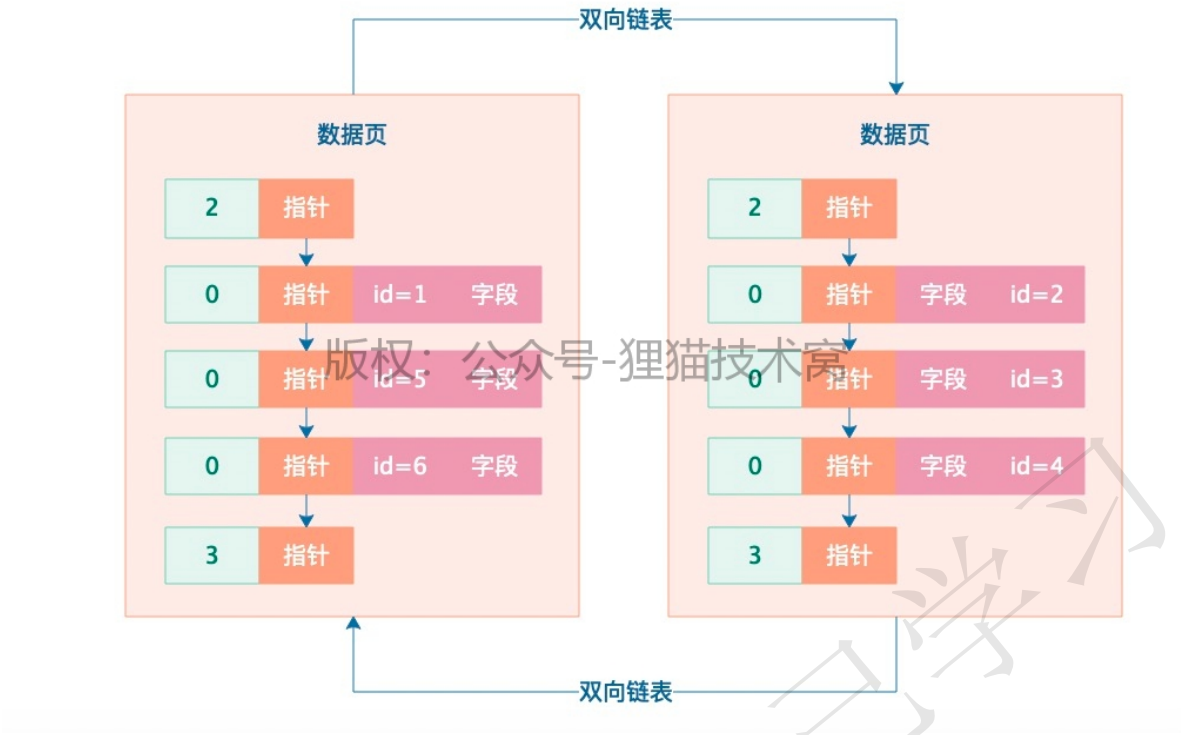
但是此时会遇到一个问题，后续我们会讲到索引这块机制，索引运作的一个核心基础就是要求你后一个数据页的主键值都大于前面一个数据页的主键值，但是如果你的主键是自增的，那还可以保证这一点，因为你新插入后一个数据页的主键值一定都大于前一个数据页的主键值。

但是有时候你的主键并不是自增长的，所以可能会出现你后一个数据页的主键值里，有的主键是小于前一个数据页的主键值的。

比如在第一个数据页里有一条数据的主键是10，第二个数据页里居然有一条数据的主键值是8，那此时肯定有问题了。

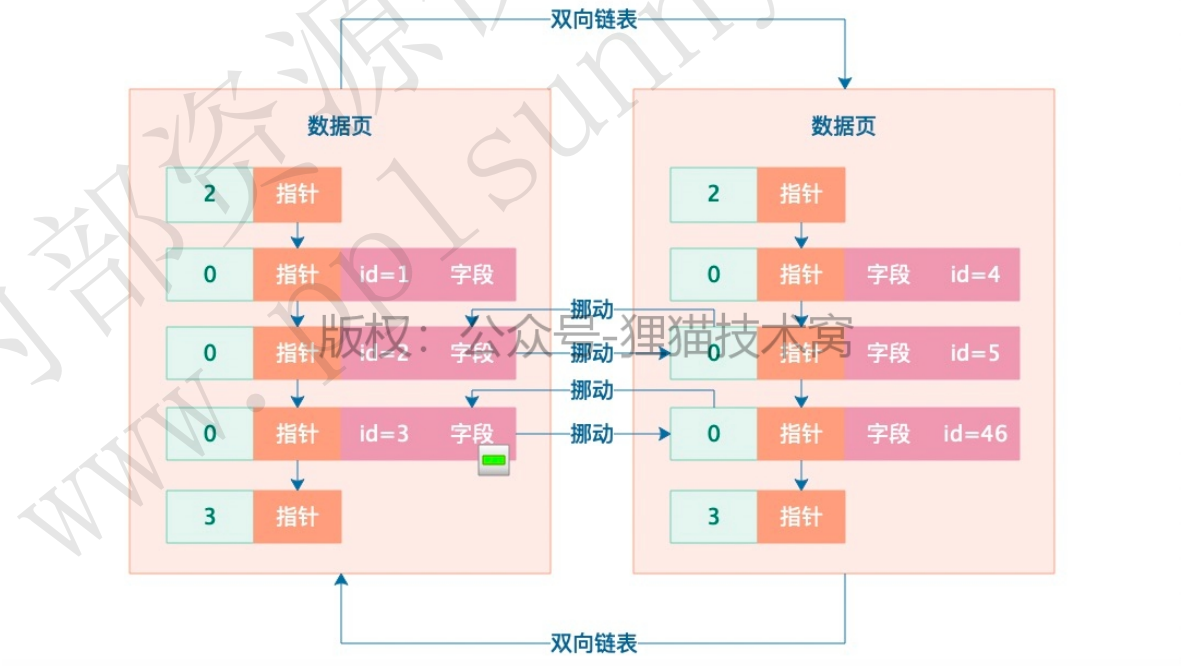
所以此时就会出现一个过程，叫做**页分裂**，就是万一你的主键值都是你自己设置的，那么在增加一个新的数据页的时候，实际上会把前一个数据页里主键值较大的，挪动到新的数据页里来，然后把你新插入的主键值较小的数据挪动到上一个数据页里去，保证新数据页里的主键值一定都比上一个数据页里的主键值大。

大家看下图，假设新数据页里，有两条数据的主键值明显是小于上一个数据页的主键值的，如图所示。



如上图所示，第一个数据页里有1、5、6三条数据，第二个数据页里有2、3、4三条数据，明显第二个数据页里的数据的主键值比第一个数据页里的5和6两个主键都小，所以这个是不行的。

此时就会出现页分裂的行为，把新数据页里的两条数据挪动到上一个数据页，上一个数据页里挪两条数据到新数据页里去，如下图所示。



所以上述就是一个页分裂的过程，核心目标就是保证下一个数据页里的主键值都比上一个数据页里的主键值要大。

这就是今天我们重点要讲的页分裂的过程，有了这个过程，保证了每个数据页的主键值，就能为后续的索引打下基础。

End

67 基于主键的索引是如何设计的，以及如何根据主键索引查询?

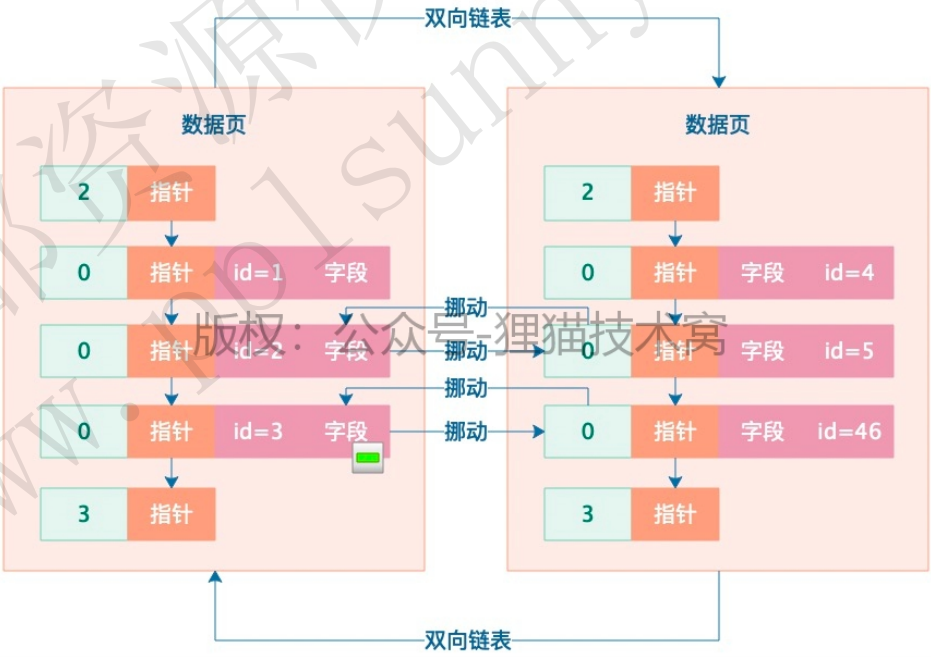
基于主键的索引是如何设计的，以及如何根据主键索引查询?

上回我们说了数据页分裂的过程，在你不停的往表里灌入数据的时候，会搞出来一个一个的数据页，如果你的主键不是自增的，他可能会有一个数据行的挪动过程，保证你下一个数据页的主键值都大于上一个数据页的主键值。

在这个基础之上，我们这一讲终于可以开始正式进入索引原理的分析了，我们就先拿最基础的主键索引来分析，一步一步的给大家把索引原理和查询原理，都讲清楚，接着就可以讲解索引设计案例和SQL调优案例了。

现在是这样的，假设我们有多个数据页，然后我们想要根据主键来查询数据，那么直接查询的话也是不行的，因为我们也不知道主键到底是在哪里，是不是?

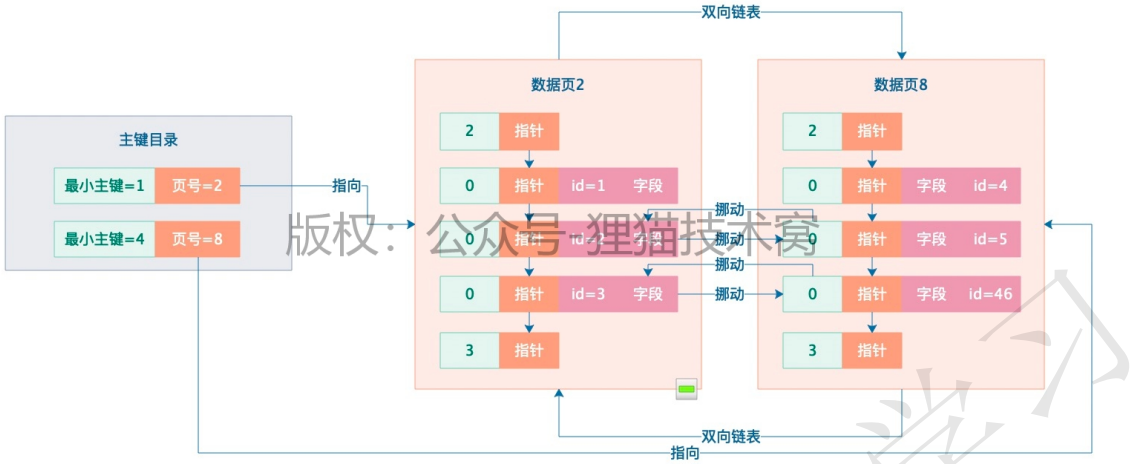
比如下图，大家回顾一下



现在假设你要搜id=4的数据，你怎么知道在哪个数据页里？没有任何证据可以告诉你他到底是在哪个数据页里啊!

所以假设还是这个样子的话，你也就只能全表扫描了，从第一个数据页开始，每个数据页都进入到页目录里查找主键，最坏情况下，所有数据页你都得扫描一遍，还是很坑的。

所以其实此时就需要针对主键设计一个索引了，针对主键的索引实际上就是主键目录，这个主键目录呢，就是把**每个数据页的页号，还有数据页里最小的主键值放在一起，组成一个索引的目录**，如下图所示。



现在我们有了上图的主键目录就方便了，直接就可以到主键目录里去搜索，比如你要找id=3的数据，此时就会跟每个数据页的最小主键来比，首先id=3大于了数据页2里的最小主键值1，接着小于了数据页8里的最小主键值4。

所以既然如此，你直接就可以定位到id=3的数据一定是在数据页2里的！

假设你有很多的数据页，在主键目录里就会有很多的数据页和最小主键值，此时你完全可以根据二分查找的方式来找你要找的id到底在哪个数据页里！

所以这个效率是非常之高的，而类似上图的主键目录，就可以认为是主键索引。

而大家都知道我们的数据页都是一坨一坨的连续数据放在很多磁盘文件里的，所以只要你能够根据主键索引定位到数据所在的数据页，此时假设我们有别的方式存储了数据页跟磁盘文件的对应关系，此时你就可以找到一个磁盘文件。

而且我们假设数据页在磁盘文件里的位置也就是offset偏移量，你也是可以知道的，此时就可以直接通过随机读的方式定位到磁盘文件的某个offset偏移量的位置，然后就可以读取连续的 一大坨数据页了！

大家看完了今天的文章，不知道有什么感想？

是不是觉得索引也不过尔尔罢了，但是其实我们今天讲的是最最简单和基础的一个索引的概念

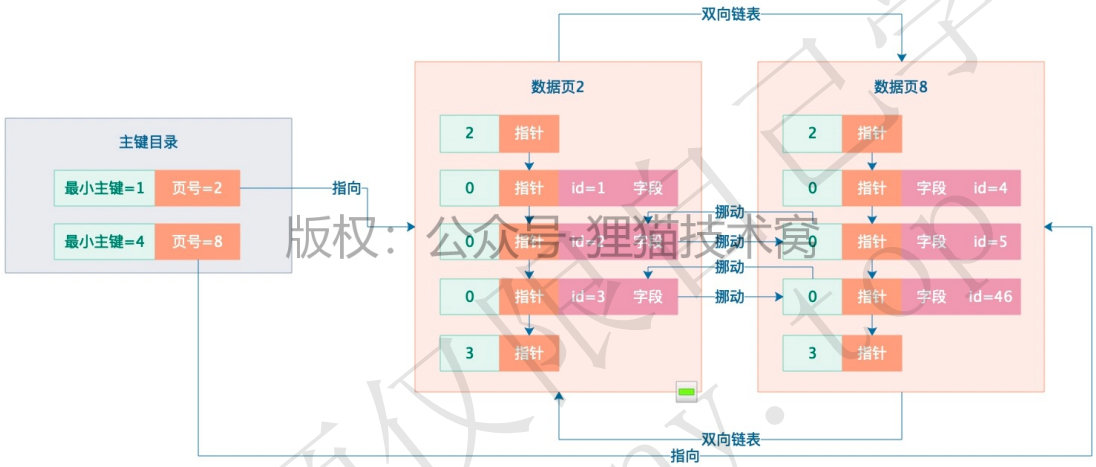
接下来下一次我们就要讲，到底为什么用B+树来组成一个索引的数据结构，那才是真正的索引！

End

68 索引的页存储物理结构，是如何用B+树来实现的?

索引的页存储物理结构，是如何用B+树来实现的?

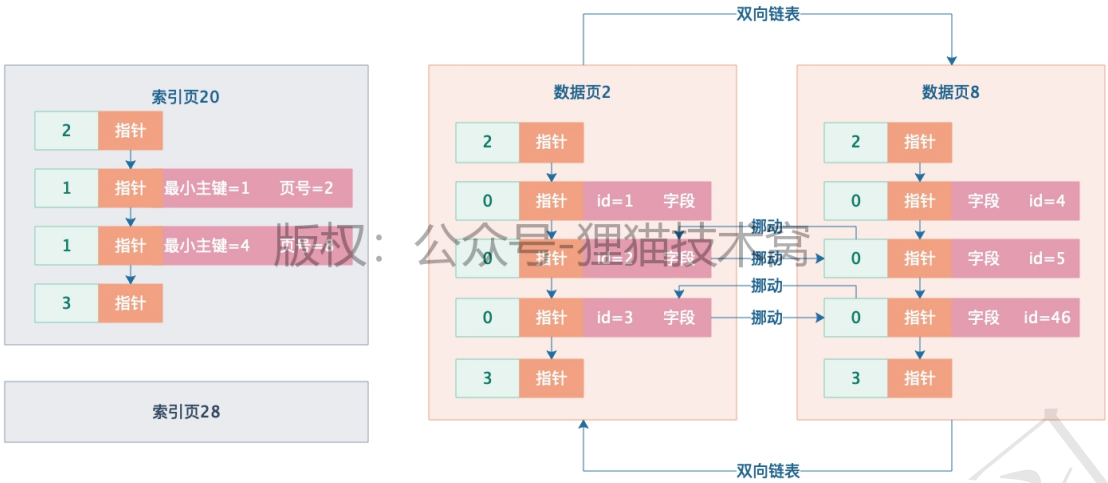
上一次我们给大家说了主键索引的目录结构，只要在一个主键索引里包含每个数据页跟他最小主键值，就可以组成一个索引目录，然后后续你查询主键值，就可以在目录里二分查找直接定位到那条数据所属的数据页，接着到数据页里二分查找定位那条数据就可以了，如下图所示。



但是现在问题来了，你的表里的数据可能很多很多，比如有几百万，几千万，甚至单表几亿条数据都是有可能的，所以此时你可能有大量的数据页，然后你的主键目录里就要存储大量的数据页和最小主键值，这怎么行呢?

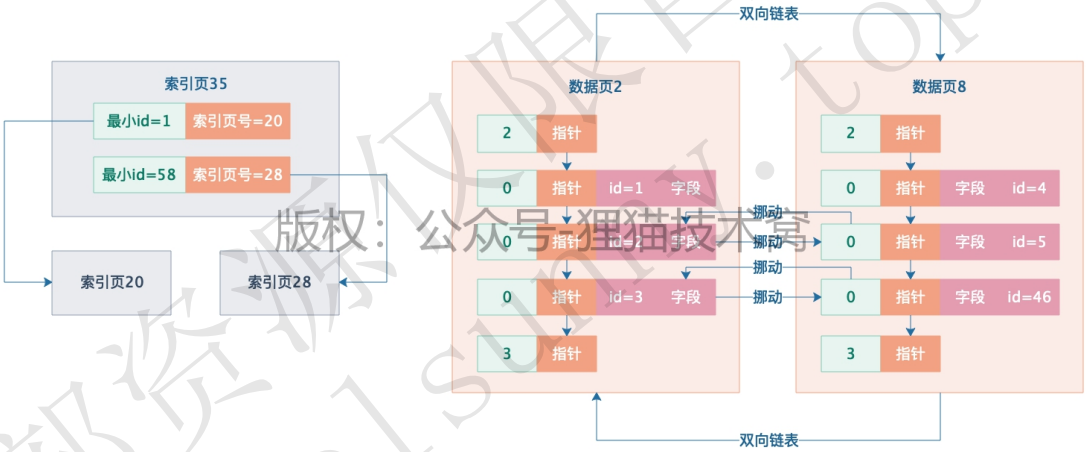
所以在考虑这个问题的时候，实际上是采取了一种把索引数据存储在数据页里的方式来做的

也就是说，你的表的实际数据是存放在数据页里的，然后你表的索引其实也是存放在页里的，此时索引放在页里之后，就会有索引页，假设你有很多很多的数据页，那么此时你就可以有很多的索引页，此时如下图所示。



但是现在又会存在一个问题了，你现在有很多索引页，但是此时你需要知道，你应该到哪个索引页里去找你的主键数据，是索引页20？还是索引页28？这也是个大问题

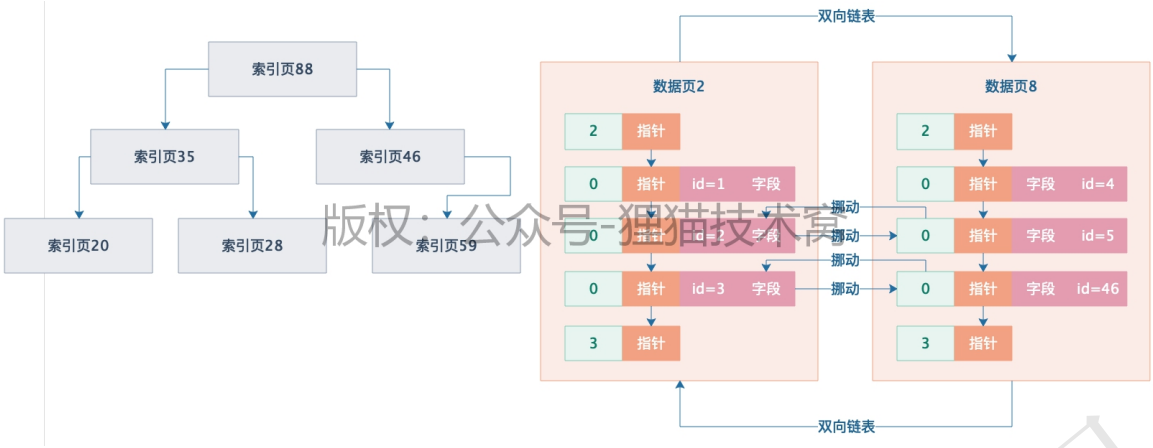
于是接下来我们又可以把索引页多加一个层级出来，在更高的索引层级里，保存了每个索引页和索引页里的最小主键值，如下图所示。



现在就好了，假设我们要查找id=46的，直接先到最顶层的索引页35里去找，直接通过二分查找可以定位到下一步应该到索引页20里去找，接下来到索引页20里通过二分查找定位，也很快可以定位到数据应该在数据页8里，再进入数据页8里，就可以找到id=46的那行数据了。

那么现在问题再次来了，假如你最顶层的那个索引页里存放的下层索引页的页号也太多了，怎么办呢？

此时可以再次分裂，再加一层索引页，比如下面图里那样子，大家看看下图。



不知道大家有没有发现索引页不知不觉中组成了多个层级，搞的是不是有点像一棵树？

没错了，**这就是一颗B+树**，属于数据结构里的一种树形数据结构，所以一直说MySQL的索引是用B+树来组成的，其实就是这个意思。

我们就以最简单最基础的主键索引来举例，当你为一个表的主键建立起来索引之后，其实这个主键的索引就是一颗B+树，然后当你要根据主键来查数据的时候，直接就是从B+树的顶层开始二分查找，一层一层往下定位，最终一直定位到一个数据页里，在数据页内部的目录里二分查找，找到那条数据。

这就是索引最真实的物理存储结构，采用跟数据页一样的页结构来存储，一个索引就是很多页组成的一颗B+树。

好了，今天讲完之后，基本上就初步让大家对索引这个东西有一个入门了，接下来我们就要比较深入的去分析各种索引的物理存储的原理

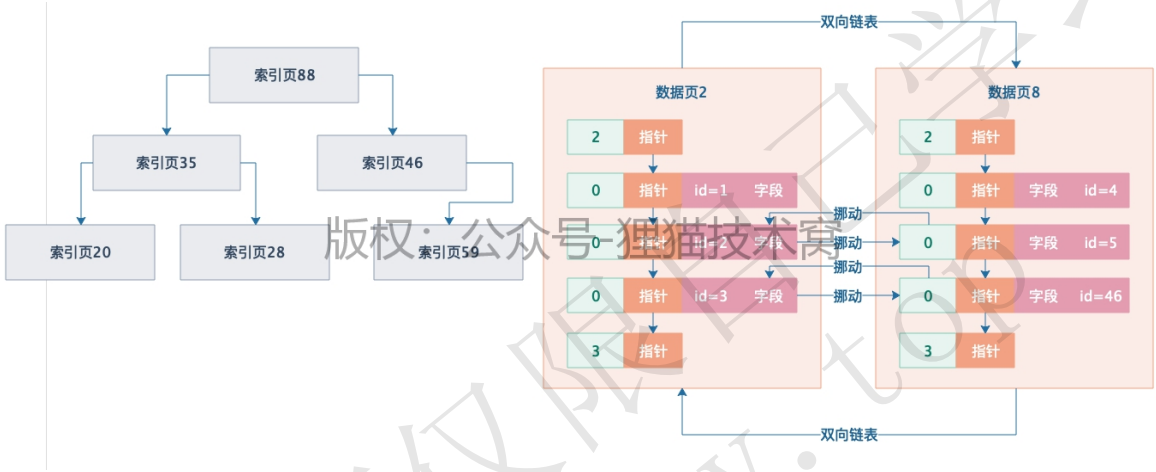
理解了索引，后续再讲查询原理和执行计划，你基本就很容易理解了。因为其实查询的过程，就是利用各种不同的索引去搜索数据的过程。

End

69 更新数据的时候，自动维护的聚簇索引到底是什么?

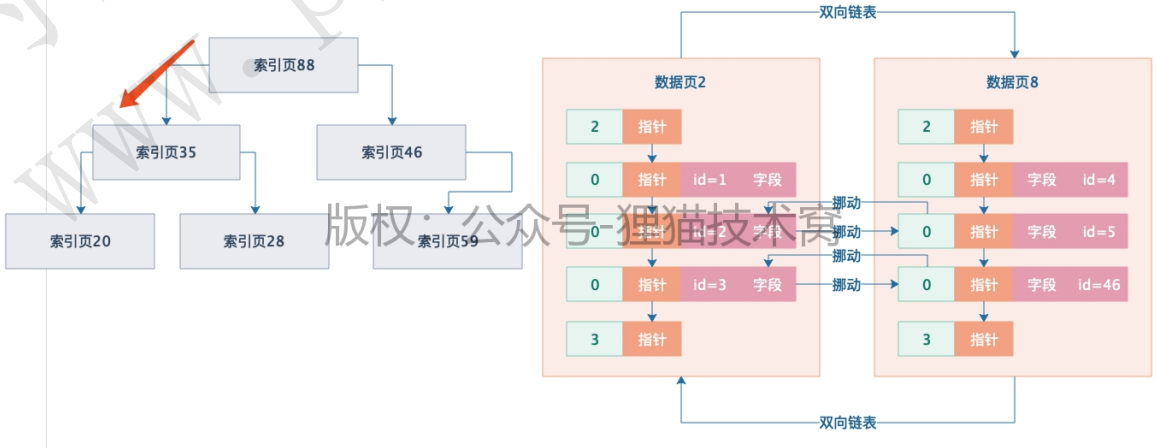
更新数据的时候，自动维护的聚簇索引到底是什么?

上一次我们给大家讲了一下基于主键如何组织一个索引，然后建立索引之后，如何基于主键在索引中快速定位到那行数据所在的数据页，再如何进入数据页快速到定位那行数据，大家看下面的图。

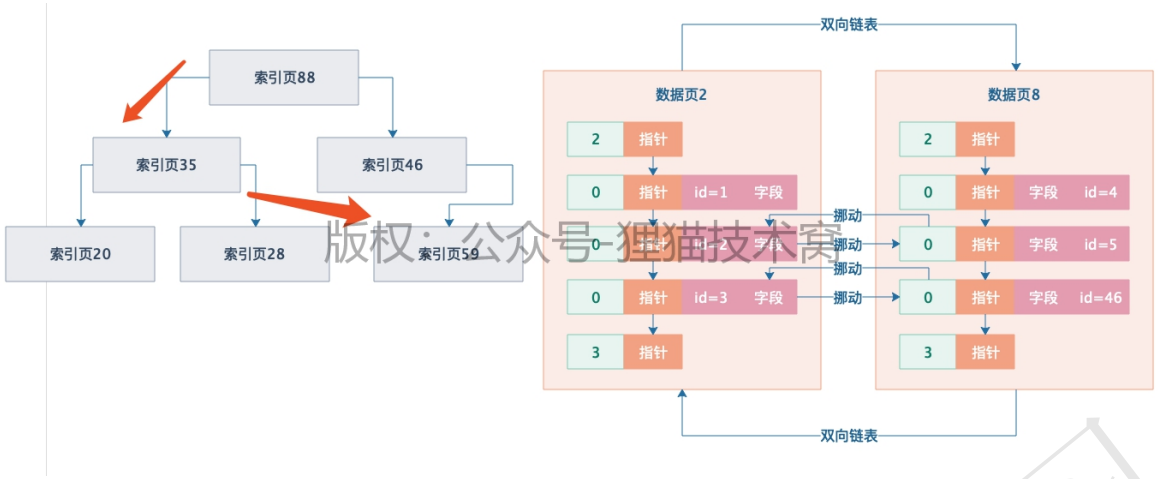


我们今天就先基于上面的图，把按照主键来搜索数据的过程重新再次给大家来梳理一遍，接着讲完了这个，其实大家也就理解今天的主题，**聚簇索引**了。

首先呢，现在假设我们要搜索一个主键id对应的行，此时你就应该先去顶层的索引页88里去找，通过二分查找的方式，很容易就定位到你应该去下层哪个索引页里继续找，如下图所示，我们给一个图示出来。

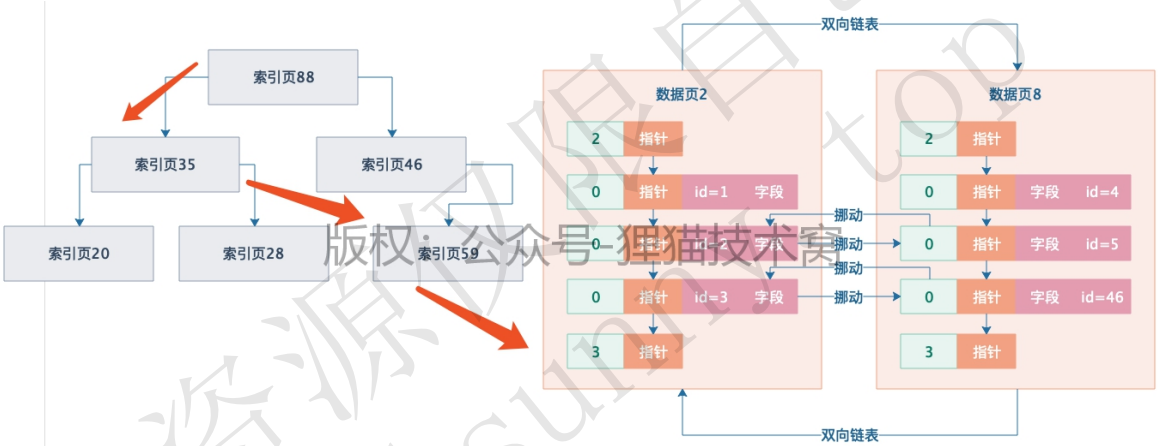


比如现在定位到了下层的索引页35里去继续找，此时在索引页35里也有一些索引条目的，分别都是下层各个索引页（20，28，59）和他们里面最小的主键值，此时在索引页35的索引条目里继续二分查找，很容易就定位到，应该再到下层的哪个索引页里去继续找，如下图所示。



我们这里看到，可能从索引页35接着就找到下层的索引页59里去了，此时索引页59里肯定也是有索引条目的，这里就存放了部分数据页页号（比如数据页2和数据页8）和每个数据页里最小的主键值

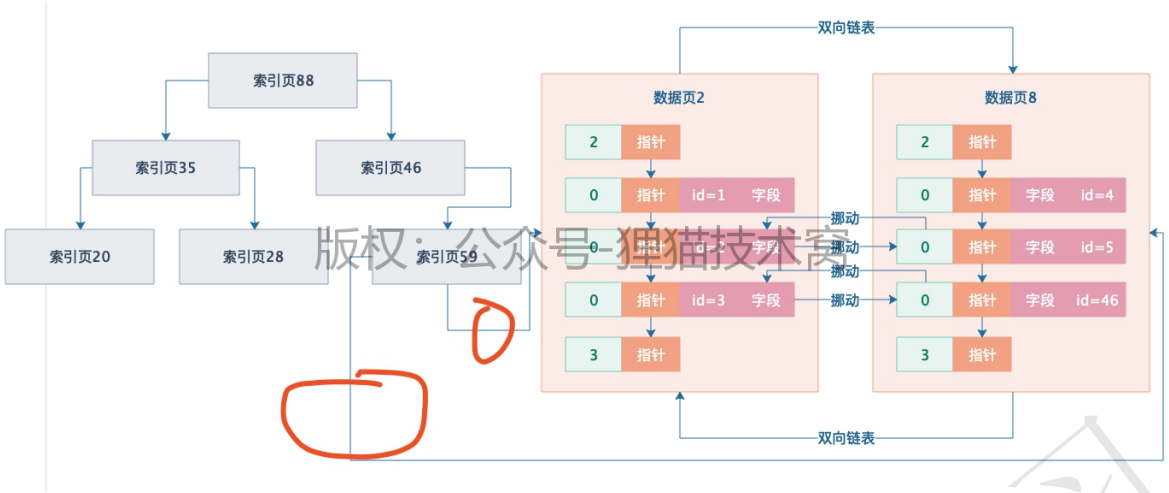
此时就在这里继续二分查找，就可以定位到应该到哪个数据页里去找，如下图所示。



接着比如进入了数据页2，里面就有一个页目录，都存放了各行数据的主键值和行的实际物理位置

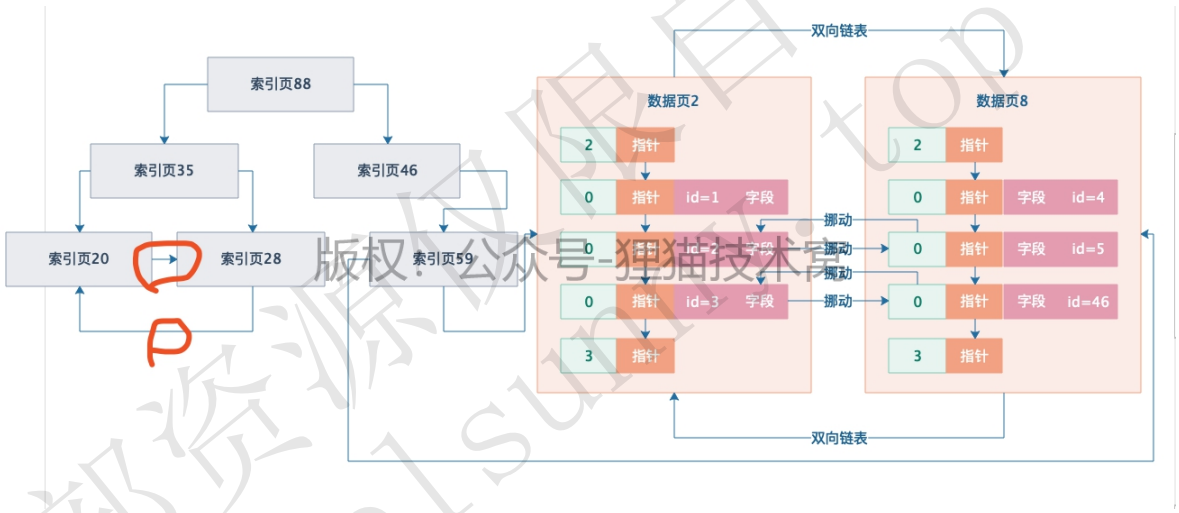
此时在这里直接二分查找，就可以快速定位到你要搜索的主键值对应行的物理位置，然后直接在数据页2里找到那条数据即可了。

这就是基于索引数据结构去查找主键的一个过程，那么大家有没有发现一件事情，其实最下层的索引页，都是会有指针引用数据页的，所以实际上索引页之间跟数据页之间是有指针连接起来的，如下图。



另外呢，其实索引页自己内部，对于一个层级内的索引页，互相之间都是基于指针组成双向链表的，如下面图示

大家可以看看，这个同一层级内的索引页组成双向链表，就跟数据页自己组成双向链表是一样的。



不知道大家把上面的图连起来看，有没有发现一些特点，就是说假设你把索引页和数据页综合起来看，他们都是连接在一起的，看起来就如同一颗完整的大的B+树一样，从根索引页88开始，一直到所有的数据页，其实组成了一颗巨大的B+树。

在这颗B+树里，最底层的一层就是数据页，数据页也就是B+树里的叶子节点了！

所以，如果一颗大的B+树索引数据结构里，叶子节点就是数据页自己本身，那么此时我们就可以称这颗B+树索引为聚簇索引！

也就是说，上图中所有的索引页+数据页组成的B+树就是聚簇索引！

其实在InnoDB存储引擎里，你在对数据增删改的时候，就是直接把你的数据页放在聚簇索引里的，数据就在聚簇索引里，聚簇索引就包含了数据！比如你插入数据，那么就是在数据页里插入数据。

如果你的数据页开始进行页分裂了，他此时会调整各个数据页内部的行数据，保证数据页内的主键值都是有顺序的，下一个数据页的所有主键值大于上一个数据页的所有主键值

同时在页分裂的时候，会维护你的上层索引数据结构，在上层索引页里维护你的索引条目，不同的数据页和最小主键值。

然后如果你的数据页越来越多，一个索引页放不下了，此时就会再拉出新的索引页，同时再搞一个上层的索引页，上层索引页里存放的索引条目就是下层索引页页号和最下主键值。

按照这个顺序，以此类推，如果你的数据量越大，此时可能就会多出更多的索引页层级来，不过说实话，一般索引页里可以放很多索引条目，所以通常而言，即使你是亿级的大表，基本上大表里建的索引的层级也就三四层而已。

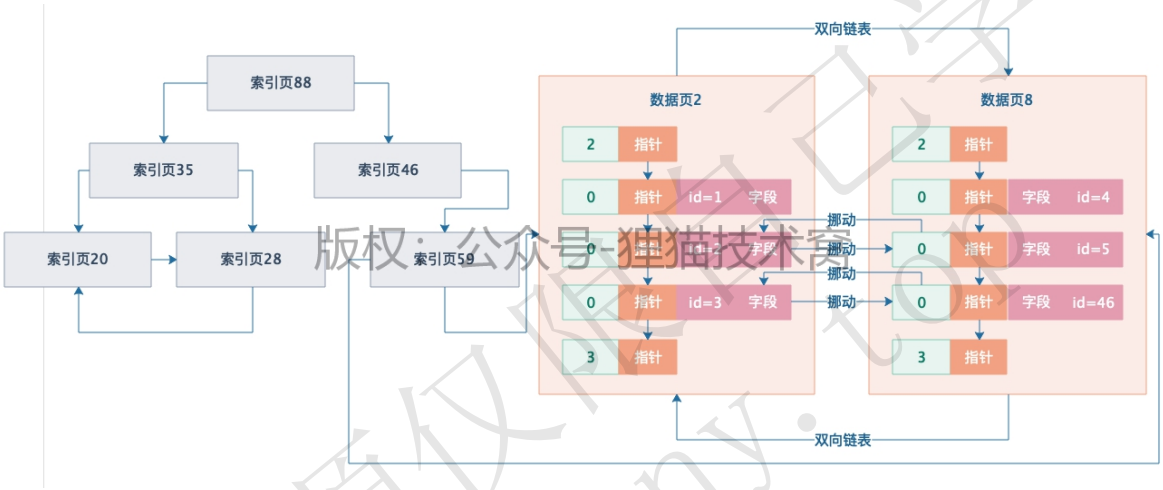
这个聚簇索引默认是按照主键来组织的，所以你在增删改数据的时候，一方面会更新数据页，一方面其实会给你自动维护B+树结构的聚簇索引，给新增和更新索引页，这个聚簇索引是默认就会给你建立的。

End

70 针对主键之外的字段建立的二级索引，又是如何运作的?

针对主键之外的字段建立的二级索引，又是如何运作的?

上一次我们已经给大家彻底讲透了聚簇索引这个东西，其实聚簇索引就是innodb存储引擎默认给我们创建的一套基于主键的索引结构，而且我们表里的数据就是直接放在聚簇索引里的，作为叶子节点的数据页，如下图。



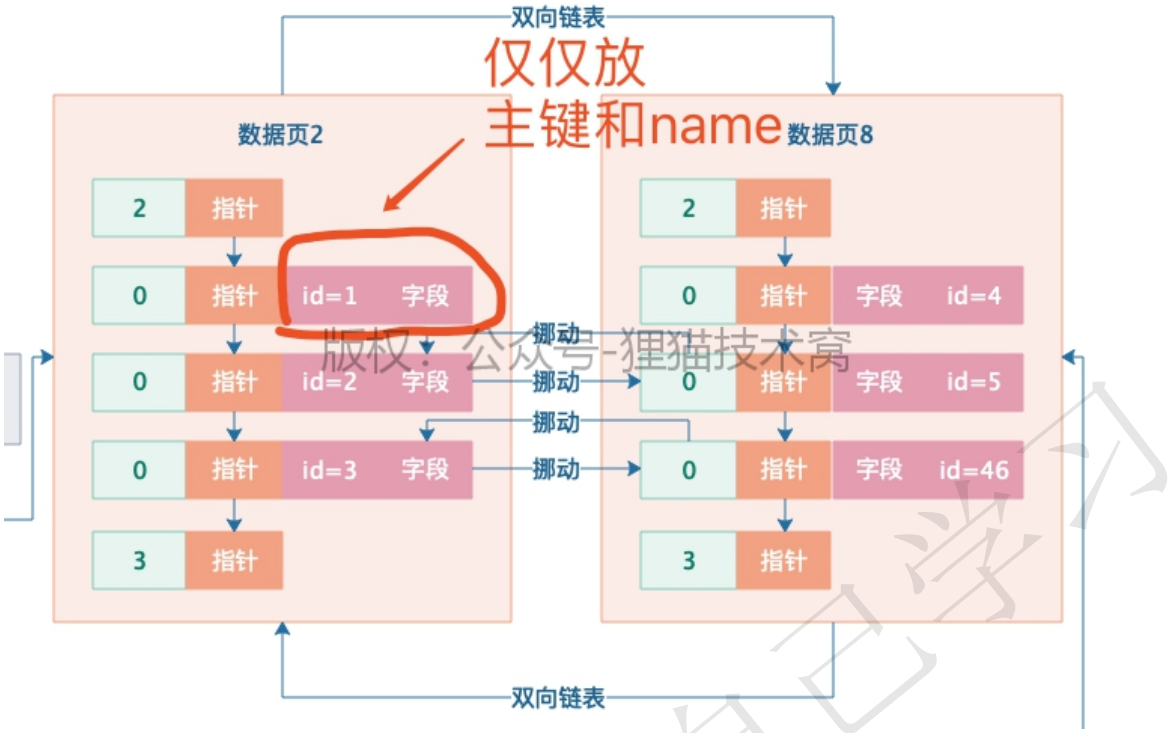
而且我们现在也对基于主键的数据搜索非常清晰了，其实就是从聚簇索引的根节点开始进行二分查找，一路找到对应的数据页里，基于页目录就直接定位到主键对应的数据就可以了，这个其实很好理解。

但是接着我们又会提另外一个疑惑了，那就是如果我们想要对其他的字段建立索引，甚至是基于多个字段建立联合索引，此时这个索引结构又是如何的呢?

今天就给大家讲讲**对主键外的其他字段建立索引的原理。**

其实假设你要是针对其他字段建立索引，比如name、age之类的字段，这都是一样的原理，简单来说，比如你插入数据的时候，一方面会把完整数据插入到聚簇索引的叶子节点的数据页里去，同时维护好聚簇索引，另一方面会为你其他字段建立的索引，重新再建立一颗B+树。

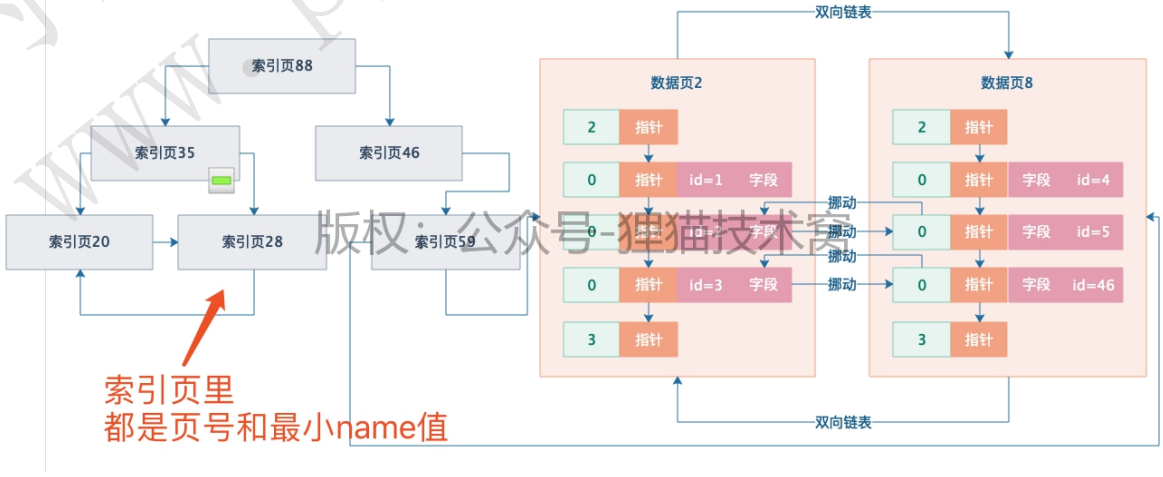
比如你基于name字段建立了一个索引，那么此时你插入数据的时候，就会重新搞一颗B+树，B+树的叶子节点也是数据页，但是这个数据页里仅仅放主键字段和name字段，大家看下面的示意图。



大家注意，这可是独立于聚簇索引之外的另外一个索引B+树了，严格来说是name字段的索引B+树，所以在name字段的索引B+树里，叶子节点的数据页里仅仅放主键和name字段的值，至于排序规则之类的，都是跟以前说的一样的。

也就是说，name字段的索引B+树里，叶子节点的数据页中的name值都是按大小排序的，同时下一个数据页里的name字段值都大于上一个数据页里的name字段值，这个整体的排序规则都跟聚簇索引按照主键的排序规则是一样的。

然后呢，name字段的索引B+树也会构建多层级的索引页，这个索引页里存放的就是下一层的页号和最小name字段值，整体规则都是一样的，只不过存放的都是name字段的值，根据name字段值排序罢了，看下图。



所以假设你要根据name字段来搜索数据，那搜索过程简直都一样了，不就是从name字段的索引B+树里的根节点开始找，一层一层往下找，一直找到叶子节点的数据页里，定位到name字段值对应的主键值。

然后呢？此时针对select * from table where name='xx'这样的语句，你先根据name字段值在name字段的索引B+树里找，找到叶子节点也仅仅可以找到对应的主键值，而找不到这行数据完整的所有字段。

所以此时还需要进行“**回表**”，这个回表，就是说还需要根据主键值，再到聚簇索引里从根节点开始，一路找到叶子节点的数据页，定位到主键对应的完整数据行，此时才能把select *要的全部字段值都拿出来。

因为我们根据name字段的索引B+树找到主键之后，还要根据主键去聚簇索引里找，所以一般把name字段这种普通字段的索引称之为二级索引，一级索引就是聚簇索引，这就是普通字段的索引的运行原理。

其实我们也可以把多个字段联合起来，建立联合索引，比如name+age

此时联合索引的运行原理也是一样的，只不过是建立一颗独立的B+树，叶子节点的数据页里放了id+name+age，然后默认按照name排序，name一样就按照age排序，不同数据页之间的name+age值的排序也如此。

然后这个name+age的联合索引的B+树的索引页里，放的就是下层节点的页号和最小的name+age的值，以此类推，所以当你根据name+age搜索的时候，就会走name+age联合索引的这颗B+树了，搜索到主键，再根据主键到聚簇索引里去搜索。

以上，就是innodb存储引擎的索引的完整实现原理了，其实大家一步一步看下来，会发现索引这块知识也没那么难，不过就是建立B+树，根据B+树一层一层二分查找罢了，然后不同的索引就是建立不同的B+树，然后你增删改的时候，一方面在数据页里更新数据，一方面就是维护你所有的索引。

后续查询，你就要尽量根据索引来查询。

End

71 插入数据时到底是如何维护好不同索引的B+树的?

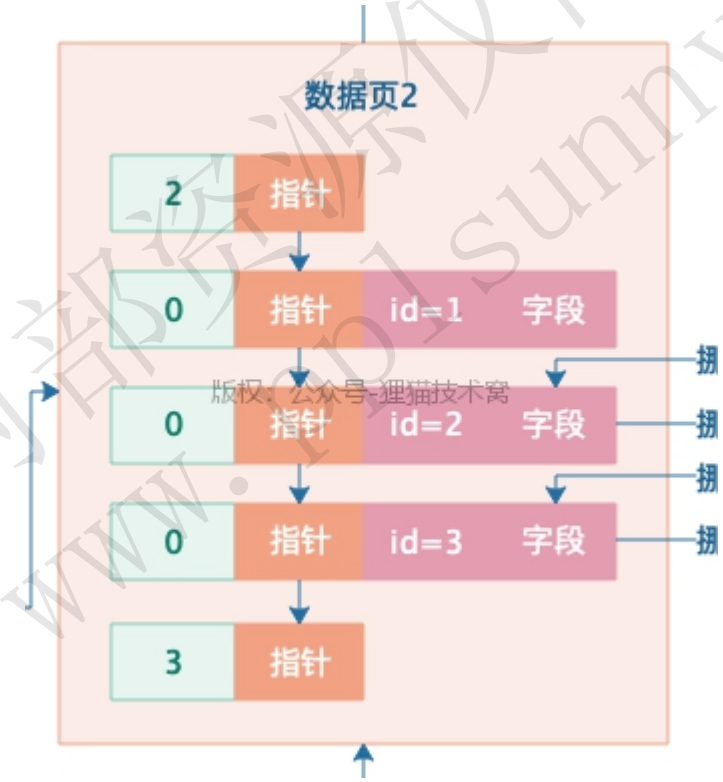
插入数据时到底是如何维护好不同索引的B+树的?

之前我们已经给大家彻底分析清楚了MySQL数据库的索引结构了，大家都知道不同索引的结构是如何的，大致是如何建立的，然后搜索的时候是如何根据不同的索引去查找数据的。

那么今天我们来给大家彻底讲清楚，你在插入数据的时候，是如何维护不同索引的B+树的。

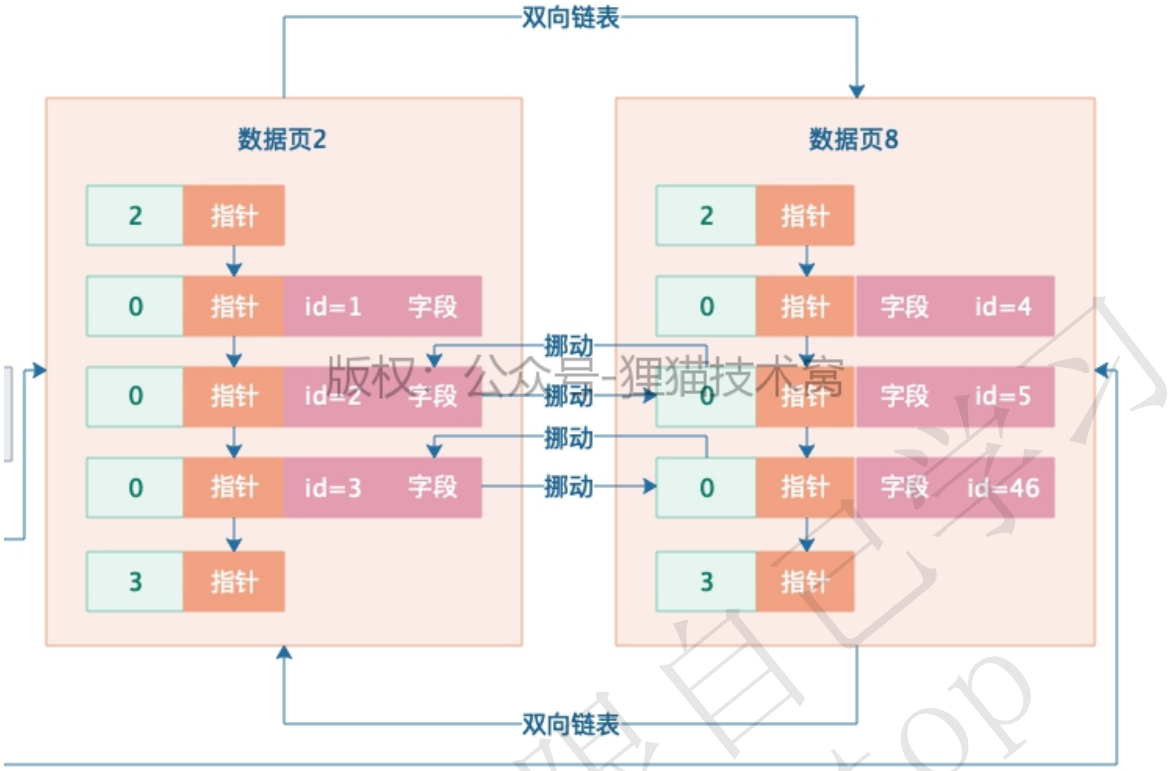
首先呢，其实刚开始你一个表搞出来以后，其实他就一个数据页，这个数据页就是属于聚簇索引的一部分，而且目前还是空的

此时如果你插入数据，就是直接在这个数据页里插入就可以了，也没必要给他弄什么索引页，如下图。



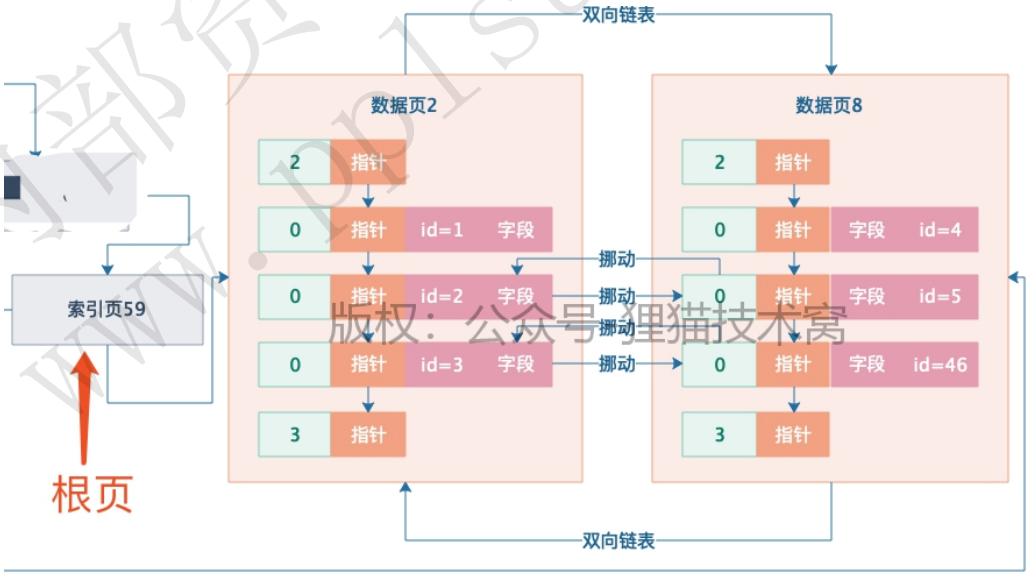
然后呢，这个初始的数据页其实就是一个根页，每个数据页内部默认就有一个基于主键的页目录，所以此时你根据主键来搜索都是ok没有问题的，直接在唯一 一个数据页里根据页目录找就行了。

然后你表里的数据越来越多了，此时你的数据页满了，那么就会搞一个新的数据页，然后把你根页面里的数据都拷贝过去，同时再搞一个新的数据页，根据你的主键值的大小进行挪动，让两个新的数据页根据主键值排序，第二个数据页的主键值都大于第一个数据页的主键值，如下图。



那么此时那个根页在哪儿呢？

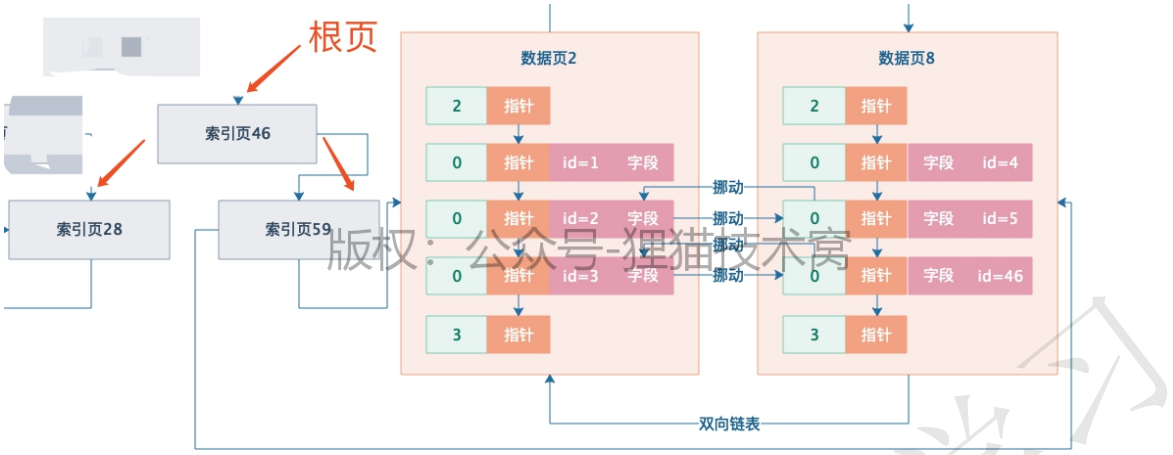
此时根页就升级为索引页了，这个根页里放的是两个数据页的页号和他们里面最小的主键值，所以此时看起来如下图，根页就成为了索引页，引用了两个数据页。



接着你肯定会不停的在表里灌入数据，然后数据页不停的页分裂，分裂出来越来越多的数据页

此时你的唯一一个索引页，也就是根页里存放的数据页索引条目越来越多，连你的索引页都放不下了，那你就让一个索引页分裂成两个索引页，然后根页继续往上走一个层级引用了两个索引页

如下图。



接着就是依次类推了，你的数据页越来越多，那么根页指向的索引页也会不停分裂，分裂出更多的索引页，当你下层的索引页数量太多的时候，会导致你的根页指向的索引页太多了，此时根页继续分裂成多个索引页，根页再次往上提上去去一个层级。

这其实就是你增删改的时候，整个聚簇索引维护的一个过程，其实其他的二级索引也是类似的一个原理

比如你name字段有一个索引，那么刚开始的时候你插入数据，一方面在聚簇索引的唯一的数据页里插入，一方面在name字段的索引B+树唯一的数据页里插入。

然后后续数据越来越多了，你的name字段的索引B+树里唯一的数据页也会分裂，整个分裂的过程跟上面说的是一样的，所以你插入数据的时候，本身就会自动去维护你的各个索引的B+树。

另外给大家补充一点，你的name字段的索引B+树里的索引页中，其实除了存放页号和最小name字段值以外，每个索引页里还会存放那个最小name字段值对应的主键值

这是因为有时候会出现多个索引页指向的下层页号的最小name字段值是一样的，此时就必须根据主键判断一下。

比如你插入了一个新的name字段值，此时他需要根据name字段的B+树索引的根页面开始，去逐层寻找和定位自己这个新的name字段值应该插入到叶子节点的哪个数据页里去

此时万一遇到一层里不同的索引页指向不同的下层页号，但是name字段值一样，此时就得根据主键值比较一下。

新的name字段值肯定是插入到主键值较大的那个数据页里去的。

好了，基本上讲到这里，大家应该对整个索引的数据结构，如何基于索引查询，插入的时候如何维护索引B+树，都有了一个比较清晰地理解了

接下来我们就要讲解MySQL中到底在查询语句里是如何使用索引的，然后单表查询语句的执行原理、多表join语句的执行原理、MySQL执行计划、SQL语句调优。

讲完这些之后，再给大家讲解一些查询优化的调优案例，索引设计案例，基本上大家就对MySQL的日常使用和优化，都有了一个系统性的知识体系了。

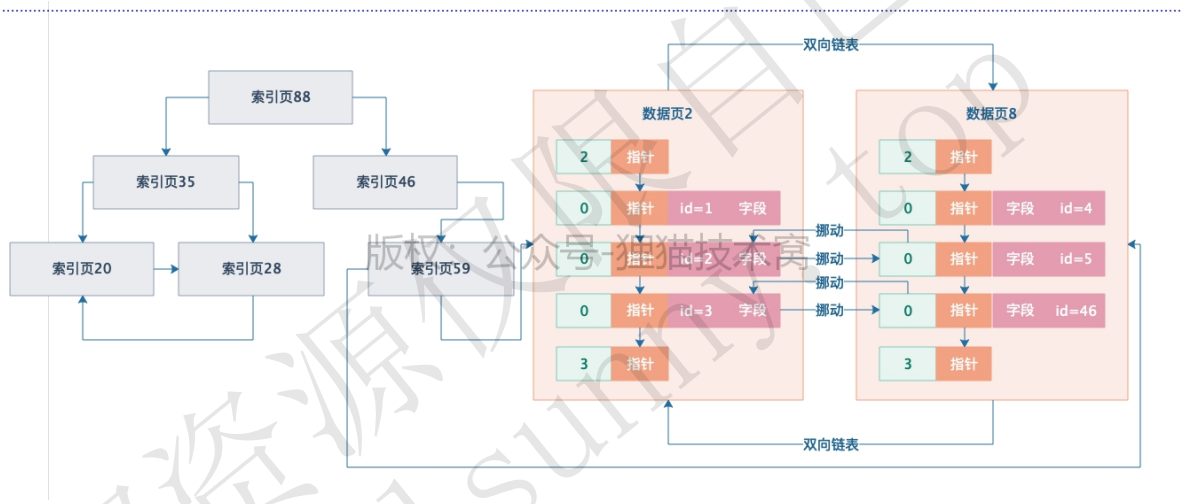
End

72 一个表里是不是索引搞的越多越好？那你就大错特错了！

一个表里是不是索引搞的越多越好？那你就大错特错了！

今天我们来稍微停一下脚步，做一个简单的关于索引知识的总结，然后再给大家分析一下索引的优点和缺点。

首先呢，我们都知道，正常我们在一个表里灌入数据的时候，都会基于主键给我们自动建立聚簇索引，这个聚簇索引大概看起来就是下面的样子。



随着我们不停的在表里插入数据，他就会不停的在数据页里插入数据，然后一个数据页放满了就会分裂成多个数据页，这个时候就需要索引页去指向各个数据页

然后如果数据页太多了，那么索引页里里的数据页指针也就会太多了，索引页也必然会放满的，此时索引页也会分裂成多个，再形成更上层的索引页。

最后这么逐步的演化下来，你就会看到上面那个图了！这个过程我们之前都详细分析过了，相信大家看一下文字说明就知道怎么回事！

默认情况下MySQL给我们建立的聚簇索引都是基于主键的值来组织索引的，聚簇索引的叶子节点都是数据页，里面放的就是我们插入的一行一行的完整的数据了！

在一个索引B+树中，他有一些特性，那就是数据页/索引页里面的记录都是组成一个单向链表的，而且是按照数据大小有序排列的；然后数据页/索引页互相之间都是组成双向链表的，而且也都是按照数据大小有序排列的，所以其实B+树索引是一个完全有序的数据结构，无论是页内还是页之间。

正是因为这个有序的B+树索引结构，才能让我们查找数据的时候，直接从根节点开始按照数据值大小一层一层往下找，这个效率是非常高的。

然后如果是针对主键之外的字段建立索引的话，实际上本质就是为那个字段的值重新建立另外一颗B+树索引，那个索引B+树的叶子节点，存放的都是数据页，里面放的都是你字段的值和主键值，然后每一层索引页里存放的都是下层页的引用，包括页内的排序规则，页之间的排序规则，B+树索引的搜索规则，都是一样的。

但是唯一要清晰记住的一点是，假设我们要根据其他字段的索引来搜索，那么只能基于其他字段的索引B+树快速查找到那个值所对应的主键，接着再次做回表查询，基于主键在聚簇索引的B+树里，重新从根节点开始查找那个主键值，找到主键值对应的完整数据。

以上就是我们之前给大家分析过的完整的MySQL的B+树索引原理了，包括B+树索引的数据结构，排序规则，以及你插入的时候他形成的过程，基于B+树查询的原理，以及不同字段的索引是有独立B+树的和回表查询过程，就给大家完整总结好了。

那么今天我们就站在这个总结的基础之上，给大家最后提一个结论，你在MySQL的表里建立一些字段对应的索引，好处是什么？

好处显而易见了，你可以直接根据某个字段的索引B+树来查找数据，不需要全表搜索，性能提升是很高的。

但是坏处呢？索引当然有缺点了，主要是两个缺点，一个是空间上的，一个是时间上的。

空间上而言，你要是给很多字段创建很多的索引，那你必须会有很多棵索引B+树，每一棵B+树都要占用很多的磁盘空间啊！所以你要是搞的索引太多了，是很耗费磁盘空间的。

其次，你要是搞了很多索引，那么你在进行增删改查的时候，每次都需要维护各个索引的数据有序性，因为每个索引B+树都要求页内是按照值大小排序的，页之间也是有序的，下一个页的所有值必须大于上一个页的所有值！

所以你不停的增删改查，必然会导致各个数据页之间的值大小可能会没有顺序，比如下一个数据页里插入了一个比较小的值，居然比上一个数据页的值要小！此时就没办法了，只能进行数据页的挪动，维护页之间的顺序。

或者是你不停的插入数据，各个索引的数据页就要不停的分裂，不停的增加新的索引页，这个过程都是耗费时间的。

所以你要是一个表里搞的索引太多了，很可能就会导致你的增删改的速度就比较差了，也许查询速度确实是可以提高，但是增删改就会受到影响，因此通常来说，我们是不建议一个表里搞的索引太多的！

那么怎么才能尽量用最少的索引满足最多的查询请求，还不至于让索引占用太多磁盘空间，影响增删改性能呢？这就需要我们深入理解索引的使用规则了，我们的SQL语句要怎么写，才能用上索引B+树来查询！

End

73 通过一步一图来深入理解联合索引查询原理以及全值匹配规则

通过一步一图来深入理解联合索引查询原理以及全值匹配规则

今天我们来通过一步一图的方式，深入理解一下多个字段组成的联合索引查询原理，以及使用索引的全职匹配的规则。

之所以讲解联合索引，那是因为平时我们设计系统的时候一般都是设计联合索引，很少用单个字段做索引，原因之前讲过，我们还是要尽可能的让索引数量少一些，避免磁盘占用太多，增删改性能太差。

另外，单个字段的索引组织结构和查询原理，之前其实我们都讲解的很清楚了，没必要在重复了。

现在我们来假设一下，咱们有一个表是存储学生成绩的，这个表当然有id了，这个id是一个自增主键，默认就会基于他做一个聚簇索引，这个就不用多说了。

然后呢，就是包含了学生班级、学生姓名、科目名称、成绩分数四个字段，平时查询，可能比较多的就是查找某个班的某个学生的某个科目的成绩。

所以，我们可以针对学生班级、学生姓名和科目名称建立一个联合索引。

接着我们画了下面的一个图，这个图就展示了这个三个字段组成的联合索引的部分内容，大家看一下。

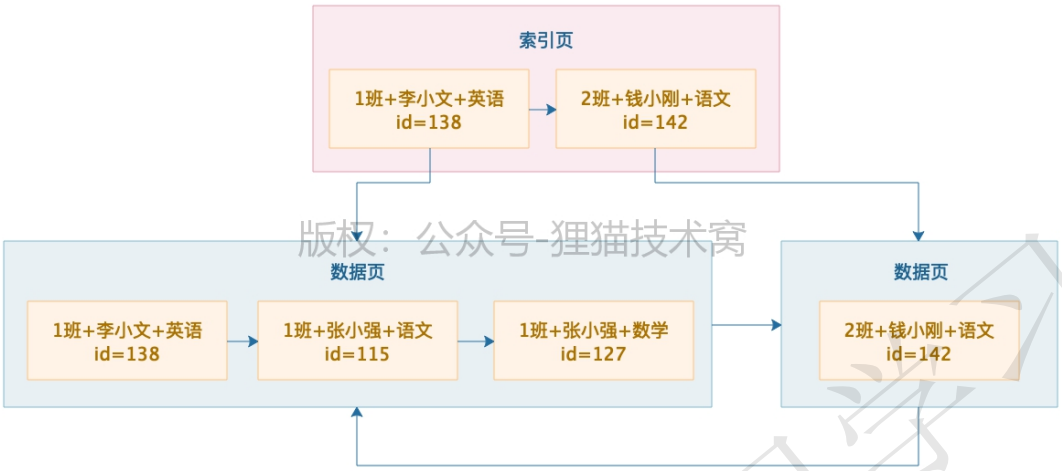
下面有两个数据页，第一个数据页里有三条数据，每条数据都包含了联合索引的三个字段的值和主键值，数据页内部是按照顺序排序的。

首先按照班级字段的值来排序，如果一样则按照学生姓名字段来排序，如果一样，则按照科目名称来排序，所以数据页内部都是按照三个字段的值来排序的，而且还组成了单向链表。

然后数据页之间也是有顺序的，第二个数据页里的三个字段的值一定都大于上一个数据页里三个字段的值，比较方法也是按照班级名称、学生姓名、科目名称依次来比较的，数据页之间组成双向链表。

索引页里就是两条数据，分别指向两个数据页，索引存放的是每个数据页里最小的那个数据的值，大家看到，索引页里指向两个数据页的索引项里都是存放了那个数据页里最小的值！

索引页内部的数据页是组成单向链表有序的，如果你有多个索引页，那么索引页之间也是有序的，组成了双向链表。



好了，那么现在假设我们想要搜索：1班+张小强+数学的成绩，此时你可能会写一个类似下面的SQL语句，select * from student_score where class_name='1班' and student_name='张小强' and subject_name='数学'。

此时就涉及到了一个索引使用的规则，那就是你发起的SQL语句里，where条件里的几个字段都是基于等值来查询，都是用的等于号！而且where条件里的几个字段的名称和顺序也跟你的联合索引一模一样！此时就是等值匹配规则，上面的SQL语句是百分百可以用联合索引来查询的。

那么查询的过程也很简单了，首先到索引页里去找，索引页里有多个数据页的最小值记录，此时直接在索引页里基于二分查找法来找就可以了，先是根据班级名称来找1班这个值对应的数据页，直接可以定位到他所在的数据页，如下图。

然后你就直接找到索引指向的那个数据页就可以了，在数据页内部本身也是一个单向链表，你也是直接就做二分查找就可以了，先按1班这个值来找，你会发现几条数据都是1班，此时就可以按照张小强这个姓名来二分查找，此时会发现多条数据都是张小强，接着就按照科目名称数学来二分查找。

很快就可以定位到下图中的一条数据，1班的张小强的数学科目，他对应的数据的id是127，如下图所示。

然后就根据主键id=127到聚簇索引里按照一样的思路，从索引根节点开始二分查找迅速定位下个层级的页，再不停的找，很快就可以找到id=127的那条数据，然后从里面提取所有字段，包括分数，就可以了。

上面整个过程就是联合索引的查找过程，以及全值匹配规则，假设你的SQL语句的where条件里用的几个字段的名称和顺序，都跟你的索引里的字段一样，同时你还是用等号在做等值匹配，那么直接就会按照上述过程来找。

对于联合索引而言，就是依次按照各个字段来进行二分查找，先定位到第一个字段对应的值在哪个页里，然后如果第一个字段有多条数据值都一样，就根据第二个字段来找，以此类推，一定可以定位到某条或者某几条数据！

End

74 再来看看几个最常见和最基本的索引使用规则

再来看看几个最常见和最基本的索引使用规则

今天我们来讲一下最常见和最基本的几个索引使用规则，也就是说，当我们建立好一个联合索引之后，我们的SQL语句要怎么写，才能让他的查询使用到我们建立好的索引呢？

下面就一起来看看，还是用之前的例子来说明。

上次我们讲的是等值匹配规则，就是你where语句中的几个字段名称和联合索引的字段完全一样，而且都是基于等号的等值匹配，那百分百会用上我们的索引，这个大家是没有问题的，即使你where语句里写的字段的顺序和联合索引里的字段顺序不一致，也没关系，MySQL会自动优化为按联合索引的字段顺序去找。

现在看第二个规则，就是**最左侧列匹配**，这个意思就是假设我们联合索引是KEY(class_name, student_name, subject_name)，那么不一定必须要在where语句里根据三个字段来查，其实只要根据最左侧的部分字段来查，也是可以的。

比如你可以写select * from student_score where class_name='' and student_name=''，就查某个学生所有科目的成绩，这都是没有问题的。

但是假设你写一个select * from student_score where subject_name=''，那就不行了，因为联合索引的B+树里，是必须先按class_name查，再按student_name查，不能跳过前面两个字段，直接按最后一个subject_name查的。

另外，假设你写一个select * from student_score where class_name='' and subject_name=''，那么只有class_name的值可以在索引里搜索，剩下的subject_name是没法在索引里找的，道理同上。

所以在建立索引的过程中，你必须考虑好联合索引字段的顺序，以及你平时写SQL的时候要按哪几个字段来查。

第三个规则，是**最左前缀匹配原则**，即如果你要用like语法来查，比如select * from student_score where class_name like '1%'，查找所有1打头的班级的分数，那么也是可以用到索引的。

因为你的联合索引的B+树里，都是按照class_name排序的，所以你要是给出class_name的确定的最左前缀就是1，然后后面的给一个模糊匹配符号，那也是可以基于索引来查找的，这是没问题的。

但是你如果写class_name like '%班'，在左侧用一个模糊匹配符，那他就没法用索引了，因为不知道你最左前缀是什么，怎么去索引里找啊？

第四个规则，就是**范围查找规则**，这个意思就是说，我们可以用select * from student_score where class_name>'1班' and class_name<'5班'这样的语句来范围查找某几个班级的分数。

这个时候也是会用到索引的，因为我们的索引的最下层的数据页都是按顺序组成双向链表的，所以完全可以先找到'1班'对应的数据页，再找到'5班'对应的数据页，两个数据页中间的那些数据页，就全都是在你范围内的数据了！

但是如果你要是写select * from student_score where class_name>'1班' and class_name<'5班' and student_name>''，这里只有class_name是可以基于索引来找的，student_name的范围查询是没法用到索引的！

这也是一条规则，就是你的where语句里如果有范围查询，那只有对联合索引里最左侧的列进行范围查询才能用到索引！

第五个规则，就是**等值匹配+范围匹配的规则**，如果你要是用select * from student_score where class_name='1班' and student_name>'' and subject_name<''，那么此时你首先可以用class_name在索引里精准定位到一波数据，接着这波数据里的student_name都是按照顺序排列的，所以student_name>''也会基于索引来查找，但是接下来的subject_name<''是不能用索引的。

所以综上所述，一般我们如果写SQL语句，都是用联合索引的最左侧的多个字段来进行等值匹配+范围搜索，或者是基于最左侧的部分字段来进行最左前缀模糊匹配，或者基于最左侧字段来进行范围搜索，这就要写符合规则的SQL语句，才能用上我们建立好的联合索引！

End

当我们在SQL里进行排序的时候，如何才能使用索引？

之前我们已经给大家讲解了在SQL里使用where语句进行数据过滤和筛选的时候，在where语句里要如何写才能用上我们建立好的索引，其实无论是哪条规则，总之，尽可能就是从联合索引最左侧的字段开始去使用，就能用上索引树！

那么今天我们来讲一下，当我们的SQL语句里使用order by语句进行排序的时候，如何才能用上索引呢？

通常而言，就我们自己想象一下，假设你有一个select * from table where xxx=xxx order by xxx这样的一个SQL语句，似乎应该是基于where语句通过索引快速筛选出来一波数据，接着放到内存里，或者放在一个临时磁盘文件里，然后通过排序算法按照某个字段走一个排序，最后把排序好的数据返回。

但是这么搞通常速度有点慢，尤其是万一你要排序的数据量比较大的话，还不能用内存来排序，如果基于磁盘文件来排序，那在MySQL里有一个术语，叫做filesort，这速度就比较慢了。

通常而言，咱们尽量是最好别这么搞，尤其是类似于select * from table order by xx1,xx2,xx3 limit 100这样的SQL语句，按照多个字段进行排序然后返回排名前100条数据，类似的语句其实常常见于分页SQL语句里，可能需要对表里的数据进行一定的排序，然后走一个limit拿出来指定部分的数据。

你要是纯粹把一坨数据放到一个临时磁盘文件里，然后直接硬上各种排序算法在磁盘文件里搞一通排序，接着按照你指定的要求走limit语句拿到指定分页的数据，这简直会让SQL的速度慢到家了！

所以通常而言，在这种情况下，假设我们建立了一个INDEX(xx1,xx2,xx3)这样的一个联合索引，这个时候默认情况下在索引树里本身就是依次按照xx1,xx2,xx3三个字段的值去排序的，那么此时你再运行select * from table order by xx1,xx2,xx3 limit 100这样的SQL语句，你觉得还需要在什么临时磁盘文件里排序吗？

显然是不用了啊！因为他要求也不过就是按照xx1,xx2,xx3三个字段来进行排序罢了，在联合索引的索引树里都排序好了，直接就按照索引树里的顺序，把xx1,xx2,xx3三个字段按照从小到大的值获取前面100条就可以了。

然后拿到100条数据的主键再去聚簇索引里回表查询剩余所有的字段。

所以说，在你的SQL语句里，应该尽量最好是按照联合索引的字段顺序去进行order by排序，这样就可以直接利用联合索引树里的数据有序性，到索引树里直接按照字段值的顺序去获取你需要的数据了。

但是这里有一些限定规则，因为联合索引里的字段值在索引树里都是从小到大依次排列的，所以你在order by里要不然就是每个字段后面什么都不加，直接就是order by xx1,xx2,xx3，要不然就都加DESC降序排列，就是order by xx1 DESC,xx2 DESC,xx3 DESC。

如果都是升序排列，直接就从索引树里最小的开始读取一定条数就可以了，要是都是降序排列，就是从索引树里最大的数据开始读取一定的条数就可以了，但是你不能order by语句里有的字段升序有的字段降序，那是不能用索引的。

另外，要是你order by语句里有的字段不在联合索引里，或者是你对order by语句里的字段用了复杂的函数，这些也不能使用索引去进行排序了。

所以说，今天的内容学完，那大家对于SQL语句的order by排序如何使用索引直接提取数据就心里有数了，其实这一讲内容是很实用的，因为我们平时写一些管理系统最常见的分页语句的时候，往往就是select * from table order by xxx limit xxx,xx这样的写法，按照某个字段自动排序，同时提取每一页的数据，所以如果你可以在排序用上索引，那么可以说你的性能就会很高。

End

当我们在SQL里进行分组的时候，如何才能使用索引？

今天我们接着上次的内容来谈谈在SQL语句里假设你要是用到了group by分组语句的话是否可以用上索引，因为大家都知道，有时候我们会想要做一个group by把数据分组接着用count sum之类的聚合函数做一个聚合统计。

那假设你要是走一个类似select count(*) from table group by xx的SQL语句，似乎看起来必须把你所有的数据放到一个临时磁盘文件里还有加上部分内存，去搞一个分组，按照指定字段的值分成一组一组的，接着对每一组都执行一个聚合函数，这个性能也是极差的，因为毕竟涉及大量的磁盘交互。

因为在我们的索引树里默认都是按照指定的一些字段都排序好的，其实字段值相同的数据都是在一起的，假设要是走索引去执行分组后再聚合，那性能一定是比临时磁盘文件去执行好多了。

所以通常而言，对于group by后的字段，最好也是按照联合索引里的最左侧的字段开始，按顺序排列开来，这样的话，其实就可以完美的运用上索引来直接提取一组一组的数据，然后针对每一组的数据执行聚合函数就可以了。

其实大家会发现，这个group by和order by用上索引的原理和条件都是差不多的，本质都是在group by和order by之后的字段顺序和联合索引中的从最左侧开始的字段顺序一致，然后就可以充分利用索引树里已经完成排序的特性，快速的根据排序好的数据执行后续操作了。

这样就不再需要针对杂乱无章的数据利用临时磁盘文件加上部分内存数据结构进行耗时耗力的现场排序和分组，那真是速度极慢，性能极差的。

所以学到这里，实际上大家应该已经理解了一点，那就是我们平时设计表里的索引的时候，必须充分考虑到后续你的SQL语句要怎么写，大概会根据哪些字段来进行where语句里的筛选和过滤？大概会根据哪些字段来进行排序和分组？

然后在考虑好之后，就可以为表设计两三个常用的索引，覆盖常见的where筛选、order by排序和group by分组的需求，保证常见的SQL语句都可以用上索引，这样你真正系统跑起来，起码是不会有太大的查询性能问题了。

毕竟只要你所有的查询语句都可以利用索引来执行，那么速度和性能通常都不会太慢。如果查询还是有问题，那就要深度理解查询的执行计划和执行原理了，然后基于执行计划来进行深度SQL调优。

然后对于更新语句而言，其实最核心的就是三大问题，一个是你索引别太多，索引太多了，更新的时候维护很多索引树肯定是不行的；一个是可能会涉及到一些锁等待和死锁的问题；一个就是可能会涉及到MySQL连接池、写redo log文件之类的问题。

所以接下来，我们会陆续讲解这些实战场景中最主要遇到的一些问题，先从查询这块的一些普通场景慢慢讲起，我们会下一讲说一下回表问题以及覆盖索引，接着就会基于电商的实际场景讲解一些案例，告诉大家如何设计索引保证查询性能别太差。

然后再讲解查询语句的执行计划以及深度SQL调优的原理以及一些实战案例，再接着讲解更新时候遇到的一些问题，包括索引、锁问题、写磁盘等等这些问题以及对应的实战案例，等大家把这些都学好之后，其实数据库日常的索引设计，查询和更新的优化，都能搞定了！

那么接着就可以进入下一步的数据库高阶场景的讲解了，包括数据库的备份和恢复，主从架构和读写分离，高可用架构，分库分表架构。

End

回表查询对性能的损害以及覆盖索引是什么?

通过之前的学习都知道，一般我们自己建的索引不管是单列索引还是联合索引，其实一个索引就对应着一颗独立的索引B+树，索引B+树的节点仅仅包含了索引里的几个字段的值以及主键值。

即使我们根据索引树按照条件找到了需要的数据，那也仅仅是索引里的几个字段的值和主键值，万一你搞了一个select *还需要很多其他的字段，那还得走一个回表操作，根据主键跑到主键的聚簇索引里去找，聚簇索引的叶子节点是数据页，找到数据页里才能把一行数据的所有字段值提取出来。

所以其实大家可以思考一下，假设你是类似select * from table order by xx1,xx2,xx3的语句，可能你就是得从联合索引的索引树里按照顺序取出来所有数据，接着对每一条数据都走一个主键的聚簇索引的查找，其实性能也是不高的。

有的时候MySQL的执行引擎甚至可能会认为，你要是类似select * from table order by xx1,xx2,xx3的语句，相当于是得把联合索引和聚簇索引，两个索引的所有数据都扫描一遍了，那还不如就不走联合索引了，直接全表扫描得了，这样还就扫描一个索引而已。

但是你如果要是select * from table order by xx1,xx2,xx3 limit 10这样的语句，那执行引擎就知道了，你先扫描联合索引的索引树拿到10条数据，接着对10条数据在聚簇索引里查找10次就可以了，那么就还是会走联合索引的。

所以说，上述原理大家首先得先知晓一下。

其次的话，就是给大家讲解一个覆盖索引的概念，其实覆盖索引不是一种索引，他就是一种基于索引查询的方式罢了。

他的意思就是针对类似select xx1,xx2,xx3 from table order by xx1,xx2,xx3这样的 语句，这种情况下，你仅仅需要联合索引里的几个字段的值，那么其实就只要扫描联合索引的索引树就可以了，不需要回表去聚簇索引里找其他字段了。

所以这个时候，需要的字段值直接在索引树里就能提取出来，不需要回表到聚簇索引，这种查询方式就是覆盖索引。

也正是这样，所以在写SQL语句的时候，一方面是你要注意一下也许你会用到联合索引，但是是否可能会导致大量的回表到聚簇索引，如果需要回表到聚簇索引的次数太多了，可能就直接给你做成全表扫描不走联合索引了；

一方面是尽可能还是在SQL里指定你仅仅需要的几个字段，不要搞一个select *把所有字段都拿出来，甚至最好是直接走覆盖索引的方式，不要去回表到聚簇索引。

即使真的要回表到聚簇索引，那你也尽可能用limit、where之类的语句限定一下回表到聚簇索引的次数，就从联合索引里筛选少数数据，然后再回表到聚簇索引里去，这样性能也会好一些。

好了，到这里为止，关于索引本身的工作原理以及SQL语句怎么写才能用上索引，就给大家都讲清楚了，下一讲我们给大家说说平时设计索引的时候，一些通用的原则，如何选择索引，如何设计索引。

End

设计索引的时候，我们一般要考虑哪些因素呢？（上）

本周我们将要讲解一下设计索引的时候，我们通常应该考虑哪些因素，给哪些字段建立索引，如何建立索引，建立好索引之后应该如何使用才是最合适的。

可能有的朋友会希望尽快更新后面的内容，但是因为工作的原因的确非常忙，也很少有周末时间，目前一周三更也是竭尽全力了，希望大家理解一下。

另外虽然更新频率下降了，但是质量绝对不会下降，这点还请大家放心

此外可以告诉大家的一个好消息是，下周开始将会开启为期两周的案例实战部分，也就是我们将会以一个电商平台的商品系统、交易系统以及营销系统的表结构设计以及索引设计作为案例背景，来告诉大家在实际的系统设计中，应该如何设计表结构以及索引。

接下来的这个案例将会包含商品表、商品详情表、订单表、物流表、退款表、购物车表、营销活动表，等多个表的设计，帮助大家在电商场景下去学习表结构的设计，以及针对具体的业务场景如何设计索引，这就跟我们最近学习的索引部分完全关联上了。

好了，那么接着就开始本周的索引设计一般原则的讲解吧。

首先，我们在针对业务需求建立好一张表的结构之后，就知道这个表有哪些字段，每个字段是什么类型的，会包含哪些数据

接着设计好表结构之后，接下来要做的，就是要设计表的索引，这个设计索引的时候，我们要考虑第一点，就是未来我们对表进行查询的时候，大概会如何来进行查询？

其实很多时候很多人可能说，你要让我刚设计完表结构就知道未来会怎么查询表，那我怎么可能知道呢，实在是想不出来！

好，那么没关系，此时我们完全可以在表结构设计完毕之后，先别急着设计索引，因为此时你根本不知道要怎么查询表。

接着我们就可以进入系统开发的环节，也就是说根据需求文档逐步逐步的把你的Java业务代码给写好，在写代码的过程中，现在一般我们都是用MyBatis作为数据持久层的框架的，你肯定会写很多的MyBatis的DAO和Mapper以及SQL吧？

那么当你系统差不多开发完毕了，功能都跑通了，此时你就可以来考虑如何建立索引了，因为你的系统里所有的MyBatis的SQL语句都已经写完了，你完全知道对每一张表会发起些什么样的查询语句，对吧？

那么这个时候，第一个索引设计原则就来了，针对你的SQL语句里的where条件、order by条件以及group by条件去设计索引

也就是说，你的where条件里要根据哪些字段来筛选数据？order by要根据哪些字段来排序？group by要根据哪些字段来分组聚合？

此时你就可以设计一个或者两三个联合索引，每一个联合索引都尽量去包含上你的where、order by、group by里的字段，接着你就要仔细审查每个SQL语句，是不是每个where、order by、group by后面跟的字段顺序，都是某个联合索引的最左侧字段开始的部分字段？

比如你有一个联合索引是INDEX(a,b,c)，此时你一看发现有三个SQL，包含了where a=? and b=?，order by a,b，group by a这些部分，那么此时where、order by、group by后续跟的字段都是联合索引的最左侧开始的部分字段，这就可以了，说明你的每个SQL语句都会用上你的索引了。

所以在设计索引的时候，首先第一条，就是要按照这个原则，去保证你的每个SQL语句的where、order by和group by都可以用上索引。

End

设计索引的时候，我们一般要考虑哪些因素呢？（中）

今天我们继续来说一下，在设计索引的时候要考虑哪些因素。之前已经说了，你设计的索引最好是让你的各个where、order by和group by后面跟的字段都是联合索引的最左侧开始的部分字段，这样他们都能用上索引。

但是在设计索引的时候还得考虑其他的一些问题，首先一个就是字段基数问题，举个例子，有一个字段他一共在10万行数据里有10万个值对吧？结果呢？这个10万值，要不然就是0，要不然就是1，那么他的基数就是2，为什么？因为这个字段的值就俩选择，0和1。

假设你要是针对上面说的这种字段建立索引的话，那就还不如全表扫描了，因为你的索引树里就仅仅包含0和1两种值，根本没法进行快速的二分查找，也根本就没有太大的意义了，所以这种时候，选用这种基数很低的字段放索引里意义就不大了。

一般建立索引，尽量使用那些基数比较大的字段，就是值比较多的字段，那么才能发挥出B+树快速二分查找的优势来。

其次的话，你尽量是对那些**字段的类型比较小的列来设计索引**，比如说什么tinyint之类的，因为他的字段类型比较小，说明这个字段自己本身的值占用磁盘空间小，此时你在搜索的时候性能也会比较好一点。

不过当然了，这个所谓的字段类型小一点的列，也不是绝对的，很多时候你就是要针对varchar(255)这种字段建立索引，哪怕多占用一些磁盘空间，那你也得去设计这样的索引，比较关键的其实还是尽量别把基数太低的字段包含在索引里，因为意义不是太大。

那当然了，万一要是你真的有那种varchar(255)的字段，可能里面的值太大了，你觉得都放索引树里太占据磁盘空间了，此时你仔细考虑了一下，发现完全可以换一种策略，也就是仅仅针对这个varchar(255)字段的前20个字符建立索引，就是说，对这个字段里的每个值的前20个字符放在索引树里而已。

此时你建立出来的索引其实类似于KEY my_index(name(20),age,course)，就这样的一个形式，假设name是varchar(255)类型的，但是在索引树里你对name的值仅仅提取前20个字符而已。

此时你在where条件里搜索的时候，如果是根据name字段来搜索，那么此时就会先到索引树里根据name字段的前20个字符去搜索，定位到之后前20个字符的前缀匹配的部分数据之后，再回到聚簇索引提取出来完整的name字段值进行比对就可以了。

但是假如你要是order by name，那么此时你的name因为在索引树里仅仅包含了前20个字符，所以这个排序是没法用上索引了！group by也是同理的。所以这里大家要对前缀索引有一个了解。

好了，同学们，今天给大家重点讲了**索引字段的基数和前缀索引**的知识，大家就记住两点，对于那种字段基数很低的列尽量别包含到索引里去，没多大用；

另外就是对于那种比较长的字符串类型的列，可以设计前缀索引，仅仅包含部分字符到索引树里去，where查询还是可以用的，但是order by和group by就用不上了。

End

设计索引的时候，我们一般要考虑哪些因素呢？（下）

今天我们最后来讲一下设计索引的时候，我们一般要考虑哪些因素。之前已经给大家讲解了索引设计时候如何根据你的查询语句来设计，让你的查询语句都能用上索引

另外还讲了字段基数的问题以及前缀索引的问题，其实就是你设计索引的时候尽量别把基数很低的字段包含进去，同时针对很长的字符串类型的字段，可以设计前缀索引来进行where查询

那么今天接着来讲剩下的一些索引设计的原则。

首先假设你设计好了一个索引，非常棒，接着你在SQL里这么写：where function(a) = xx，你给你的索引里的字段a套了一个函数，你觉得还能用上索引吗？

明显是不行了。所以尽量不要让你的查询语句里的字段搞什么函数，或者是搞个计算。

现在设计索引的时候需要注意的点都已经讲完了，其实就是好好设计索引，让你的查询语句都能用上索引，同时注意一下字段基数、前缀索引和索引列套函数的问题，尽量让你的查询都能用索引，别因为一些原因用不上索引了。

接着我们来看看索引设计好之后，接着你系统跑起来，有数据插入也有查询的情况，其实查询基本都能走索引一般问题都不会太大的，但是插入就有点讲究了，之前也跟大家说过，其实你插入数据的时候，他肯定会更新索引树。

你插入数据肯定有主键吧，那有主键就得更新聚簇索引树，你插入一条数据肯定会包含索引里各个字段的值吧，那你的联合索引的B+树是不是也要更新？

对了，你不停的增删改数据，就会不停的更新你的索引树。

所以因为你插入的数据值可能根本不是按照顺序来的，很可能会导致索引树里的某个页就会自动分裂，这个页分裂的过程就很耗费时间，**因此一般让大家设计索引别太多，建议两三个联合索引就应该覆盖掉你这个表的全部查询了。**

否则索引太多必然导致你增删改数据的时候性能很差，因为要更新多个索引树。

另外很关键一点，建议大家主键一定是自增的，别用UUID之类的，因为主键自增，那么起码你的聚簇索引不会频繁的分裂，主键值都是有序的，就会自然的新增一个页而已，但是如果你用的是UUID，那么也会导致聚簇索引频繁的页分裂。

所以说，以上就是我们本周要讲给大家听的索引设计的所有的原则，希望大家以后在索引设计的时候多想一想上述原则，接下来我们就给大家讲解电商平台的表设计以及索引设计的案例实战。

End

案例实战：陌生人社交APP的MySQL索引设计实战（一）

从今天开始，我们将会用4篇文章给大家介绍一些MySQL索引设计的实战案例，本来是想要用电商系统的场景给大家介绍索引设计的，但是在整理笔记的时候发现，电商场景的业务实在是太复杂了，可能要把电商的业务讲清楚都需要很大的篇幅，所以决定采取相对较为独立和简单一些的业务场景来讲解案例。

因此首先打算用我之前协助过一个朋友的公司的项目做过的MySQL索引设计案例来讲解，朋友的公司是做一个陌生人社交APP的，这个业务场景相对较为简单，大家一听就懂是怎么回事，而且这里设计索引也有很多讲究。

首先不知道大家玩过陌生人社交APP没有，市面上有很多，相信一些非单身的朋友可能玩儿的比较少，但是很多单身的年轻人可能都会去玩儿这类APP，他本身的核心主旨，其实就是你进入APP的时候，需要录入一系列的你的个人信息。

接着APP自己会通过一定的算法推荐一些可能适合你的人给你进行线上交友，当然也有可能是你自己通过一定的条件去搜索和筛选，查找APP上的哪些用户可能比较符合你的期望，你希望去去跟对方进行交友。

这里我们忽略掉APP基于算法自动推荐潜在感兴趣的好友给你的部分，就来看看你通过一系列的条件去筛选一些好友的过程。

我们来思考一下，在你筛选的时候，是针对社交APP的哪个表进行查询？

明显是用户信息表吧，我们可以叫做user_info这么一个表。

那这个表里往往会具备哪些用户的个人信息呢？

大致会包含你的地区（你在哪个省份、哪个城市，这个很关键，否则不在一个城市，可能线上聊的好，线下见面的机会都没有），性别，年龄，身高，体重，兴趣爱好，性格特点，还有照片，当然肯定还有最近一次在线时间（否则半年都不上线APP了，你把他搜出来干什么呢？）

另外如果支持交友过程中让其他人对他进行评价，那么可能还需要包含这个人的一个综合评分。

针对这个用户表进行搜索，可不仅仅是筛选那么简单的，因为你想一下，你除了select xx from user_info where xx=xx 有一系列的条件之外，APP肯定得支持分页展示吧？所以肯定还得跟上limit xx,xx的分页语句。

同时，很关键的一点是，你搜索的时候，肯定不是随便胡乱排序的吧，总得根据一定的规则对筛选出来的结果进行一个排序，把最符合你的条件和期望的用户排列在最上面才可以，各位想想是不是？

那么最终你的SQL语句可能是类似于：select xx from user_info where xx=xx order by xx limit xx,xx。

所以这里首先就给我们出了一个难题，之前学习索引使用规则的时候，我们都知道，你在where条件里必须是使用联合索引里最左侧开始的连续多个字段进行筛选，然后排序的时候也必须是用联合索引里的最左侧开始的多个连续字段进行排序。

那问题来了，假设你的SQL需要按照年龄进行范围筛选，同时需要按照用户的评分进行排序，类似下面的SQL：select xx from user_info where age between 20 and 25 order by score，那就有问题了。

假设你就一个联合索引，age在最左侧，那你的where是可以用上索引来筛选的，但是排序是基于score字段，那就不可以用索引了。那假设你针对age和score分别设计了两个索引，但是在你的SQL里假设基于age索引进行了筛选，是没法利用另外一个score索引进行排序的。

所以说，针对这个实际场景，**你要明白的第一个难题就是，往往在类似这种SQL里，你的where筛选和order by排序实际上大部分情况下是没法都用到索引的！**所谓鱼与熊掌不可兼得，就是这个意思！

那么除此之外，这个业务场景下的查询语句还有哪些索引设计上的难点呢？下次我们继续分析，这是一个综合性的案例，我们会用多篇文章来讲解。但是一旦大家把这个实际业务场景下的索引设计过程中的综合考虑的因素都理解了，那么自己也能很好的设计索引了。

End

案例实战：陌生人社交APP的MySQL索引设计实战（二）

今天我们继续分析这个社交APP的复杂用户搜索功能场景下的索引设计案例，上次我们讲到，在我们的这个场景里，SQL中会包含where、order by和limit几个语句，而且实际场景中，往往where和order by是没法都用到索引的，这是第一个我们要注意的问题。

今天我们来分析第二个问题，就是在where和order by出现索引设计冲突，鱼与熊掌不可兼得的时候，到底是针对where去设计索引，还是针对order by设计索引？到底是让where去用上索引，还是让order by用上索引？

其实这个问题的本质就是说，你是要让where语句先基于联合索引去进行一个筛选，筛选出来一部分用户指定的数据，接着再把数据加载到内存或者是基于临时磁盘文件去进行指定条件的排序，最后用limit语句拿到一页数据吗？

还是说要让order by语句按照你的索引的顺序去找，找的过程中基于where里的条件筛选出来指定的数据，然后再根据limit语句拿出来一页数据？

说实话，一般这种时候往往都是让where条件去使用索引来快速筛选出来一部分指定的数据，接着再进行排序，最后针对排序后的数据拿出来一页数据。

因为基于索引进行where筛选往往可以最快速度筛选出你要的少部分数据，如果筛选出来的数据量不是太大的话，那么后续排序和分页的成本往往不会太大！

好，那么假设我们打定主意要针对where条件去设计索引的话，此时又要犯难了，因为这个时候你要去考虑，用户在搜索潜在好友的时候，一般会用上哪些条件呢？我们到底要把哪些字段包含到索引里去？到底在联合索引里，字段的顺序要如何排列呢？

其实开门见山要告诉大家的一点就是，我们首先应该在联合索引里包含省份、城市、性别，这三个字段！

因为这三个字段都是在搜索里几乎必定包含的三个字段，假设你要搜索潜在好友，那么必定是会搜索跟你同一个地方的，然后搜索某个性别的的其他用户，这几个条件在APP里完全可以做成必选项，用户也几乎必定会指定。

但是此时有人就会说了，之前不是说过么，基数太低的字段最好别放到索引里去，那省份、城市和性别，都是基数非常小的几个字段，可选的值就那么几个，为什么要放到索引里去？

这是个好问题，但是规则是死的，人是活的。

假设你就因为省份、城市和性别几个字段的基数太小了，此时就不把他们几个包含到联合索引里去，那么你实际查询的时候都要基于这几个字段去搜索，此时你就只能把这几个字段放在where条件的最后，那么最后每次查询都必须要先用联合索引查询出来一部分数据，接着数据加载到内存里去，再根据where条件最后的省份、城市和性别几个字段进行过滤筛选，每次查询都得多这么一个步骤。

所以与其如此，还不如就把省份、城市和性别三个字段，放在联合索引的最左侧，这样跟其他字段组合联合索引后，让大部分的查询都可以直接通过索引树就可以把where条件指定的数据筛选出来了。

好，那么到今天为止，我们还是在分析这个案例，我们已经分析到了可以把基数较低但是频繁查询（几乎每次查询都会指定）的省份、城市和性别几个字段放到联合索引的最左侧去，此时就可以让每次查询时指定的省份、城市和性别，都直接从索引树里进行筛选。

那么联合索引中除了（province, city, sex）三个字段以外，还需要哪些其他的字段呢？其他字段该如何设计呢？是否还要设计其他的索引呢？针对这个问题，我们下一次继续分析。

End

案例实战：陌生人社交APP的MySQL索引设计实战（3）

上一次我们讲到我们的联合索引已经设计为了（province, city, sex）的样子，把省份、城市和性别三个几乎每次查询都会加的条件放入了联合索引的最左侧去，接着我们今天继续分析这个联合索引里还要放哪些字段。

分析这个问题之前，我们先来分析一个问题，那就是假设查询的时候，不指定性别，就指定了省份，城市，还有加了一个年龄，也就是说where province=xx and city=xx and age between xx and xx，那么此时怎么办呢？因为age不在索引里，所以就根本没法通过age去在索引里进行筛选了。

那如果把索引设计成（province, city, sex, age），此时你的语句写成where province=xx and city=xx and age>=xx and age<=xx，也是没法让age用上索引去筛选的，因为city和age中间差了一个sex，所以此时就不符合最左侧连续多个字段的原则了。

其实针对这个问题，大家完全没必要太担心，因为假设有上述场景，那么我们完全是可以把age放入联合索引的，设计成（province, city, sex, age）这样的索引，那么在搜索的时候就根据省份、城市和年龄来筛选，性别是不限的，此时就可以把where语句写成：where province=xx and city=xx and sex in ('female', 'male') and age >=xx and age<=xx。

如果我们把语句写成上面那样子，那么就可以让整个where语句里的条件全部都在索引树里进行筛选和搜索了！

另外，假设我们在查询语句里还有一些频繁使用的条件，通常都是兴趣爱好和性格特点，这个兴趣爱好和性格特点，往往都是有固定的一些枚举值的

比如兴趣爱好可以有下述的值可选：运动、电影、旅游、烹饪，性格特点可能包含下面的值：温柔、霸气、御姐、体贴、善良，等等。

那么针对这样的一些频繁使用的包含枚举值范围的一些字段，也完全可以加入到联合索引里去，可以设计成（province, city, sex, hobby, character, age）这样的一个联合索引，此时假设出现了这样一个查询，按照省份、城市、性格和年龄进行搜索，此时SQL怎么写？

还是用之前的那个策略和思路，就是写成where province=xx and city=xx and sex in(xx, xx) and hobby in (xx, xx, xx, xx) and character=xx and age>=xx and age<=xx

也就是说，即使你不需要按性别和爱好进行筛选，但是在SQL里你可以对这两个字段用in语句，把他们所有的枚举值都放进去。这样的话，就可以顺利的让province，city，character和age四个真正要筛选的字段用上索引，直接在索引里进行筛选都是没有问题的。

那么我们为什么一直强调，age字段必须要放在联合索引的最后一个呢？

很简单，因为之前我们讲索引使用规则的时候说过，假设你where语句里有等值匹配，还有范围匹配，此时必须是先让联合索引最左侧开始的多个字段使用等值匹配，接着最后一个字段是范围匹配。

就比如上面的语句where province=xx and city=xx and sex in(xx, xx) and hobby in (xx, xx, xx, xx) and character=xx and age>=xx and age<=xx，他们完全是按照联合索引最左侧开始的，province、city、sex、hobby、character都是联合索引最左侧开始的多个字段，他们都是等值匹配，然后最后一个age字段使用的是范围匹配，这种就是可以完全用上索引的。

但是假设你要是在联合索引里把age放在中间的位置，设计一个类似（province, city, sex, age, hobby, character）的联合索引，接着SQL写成where province=xx and city=xx and sex in(xx, xx) and age>=xx and age<=xx and hobby in (xx, xx, xx, xx) and character=xx的话，那么不好意思，只有province, city, sex, age几个字段可以用上索引。

因为在SQL里，一旦你的一个字段做范围查询用到了索引，那么这个字段接下来的条件都不能用索引了，这就是规则！

所以说，实际设计索引的时候，必须把经常用做范围查询的字段放在联合索引的最后一个，才能保证你SQL里每个字段都能基于索引去查询。

下次我们再针对这个场景下更多的一些特殊搜索场景去给大家讲，设计联合索引的时候还有哪些技巧！

End

案例实战：陌生人社交APP的MySQL索引设计实战（4）

今天是咱们的这个索引设计案例的最后一篇文章，之前通过三篇文章的分析，相信大家都已经理解了为什么我们要把索引设计成**（province, city, sex, hobby, character, age）**这样的一个形式。

这么做其实关键是要让最频繁查询的一些条件都放到索引里去，然后在查询的时候如果有些字段是不使用的，可以用in (所有枚举值)的方式去写，这样可以让所有查询条件都用上你的索引，同时对范围查询的age字段必须放在最后一个，这样保证范围查询也能用上索引。

那么今天我们来研究下一个问题，**假设在查询的时候还有一个条件**，是要根据用户最近登录时间在7天之内来进行筛选，筛选最近7天登录过APP的用户，那么实际上可能你的用户表里有这么一个字段，latest_login_time

你要是在where条件里加入这么一个latest_login_time <= 7天内语句，肯定这个是没法用上索引了。因为你这里必然会用一些计算或者是函数，才能进行一些时间的比对。

而且假设你的查询里还有age进行范围查询，那么我们之前说过，范围查询的时候，也就只有第一个范围查询是可以用上索引的，第一个范围查询之后的其他范围查询是用不上索引的。

也就是说，即使你索引设计成这样：（province, city, sex, hobby, character, age, latest_login_time），然后你的where语句写成这样：where xx xxx and age>=xx and age<=xxx and latest_login_time>=xx，虽然age和latest_login_time都在联合索引里，但是按照规则，只有age范围查询可以用到索引，latest_login_time始终是用不到索引的。

所以此时有一个技巧可以教给大家，你在设计表的时候，就必须考虑到这个问题，此时你完全可以设计一个字段为：does_login_in_latest_7_days，也就是说，这个人是否在最近7天内登录过APP。

假设在7天内登录了这个APP，那么这个字段就是1，否则超过7天没登录，这个字段就是0！这样就把一个时间字段转换为了一个枚举值的字段。

接下来的解决方案就简单化了，可以设计一个联合索引为：**（province, city, sex, hobby, character, does_login_in_latest_7_days, age）**，然后搜索的时候，一定会在where条件里带上一个does_login_in_latest_7_days=1，最后再跟上age范围查询，这样就可以让你的where条件里的字段都用索引来筛选。

实际上一般来说，假设你要是where语句里通过上述联合索引就可以过滤掉大部分的数据，就保留小部分数据下来基于磁盘文件进行order by语句的排序，最后基于limit进行分页，那么一般性能还是比较高的。

但有时候又怕一个问题，就是说万一你要是就仅仅使用联合索引里一些基数特别小的字段来筛选呢？

比如就基于性别来筛选，比如一下子筛选出所有的女性，可能有上百万用户数据，接着还要磁盘文件进行排序再分页？那这个性能可能就会极为的差劲了！

所以针对上述问题，可以针对那种基数很低的字段再加上排序字段单独额外设计一个辅助索引，专门用于解决where条件里都是基数低的字段，然后还要排序后分页的问题，比如说就可以设计一个联合索引为：（sex, score）。

此时万一你要是写出如下SQL：select xx from user_info where sex='female' order by score limit xx,xx，此时假设用之前设计的那个联合索引，那绝对是完蛋了，因为根本没法用索引

但是用我们设计的那个辅助的（sex, score）索引呢？

此时因为where条件里的字段是等值匹配，而且还是等于某个常量值，所以虽然order by后跟的score字段是（sex, score）索引里的第二个字段，order by没有从索引最左侧字段开始排列，但是他也可以使用到索引来排序。

因为具体到使用索引的层面，他会先对where条件里的sex='female'在索引树里筛选到这部分数据，接着在sex='female'的数据里，这些数据实际上都是排列在一起的，因为在索引里，会按照sex和score两个字段去进行排序，所以sex='female'的数据都是在一块儿的。

然后找到这部分数据之后，接着就可以确定，这部分数据肯定是按照score字段进行排序的，此时就可以按照score字段值的顺序，去读取你的limit语句指定的数据分页出来就可以了

所以此时你这种针对sex低基数的字段的筛选和基于评分排序的语句，整体运行的效率是非常高的，完全可以基于辅助索引来实现。

以此类推，完全可以通过对查询场景的分析，用**（province, city, sex, hobby, character, does_login_in_latest_7_days, age）**这样的联合索引去抗下复杂的where条件筛选的查询，此时走索引筛选速度很快，筛选出的数据量较少，接着进行排序和limit分页。

同时针对一些**低基数字段筛选+评分排序**的查询场景，可以设计类似（sex, score）的**辅助索引**来应对，让他快速定位到一大片低基数字段对应的数据，然后按照索引顺序去走limit语句获取指定分页的数据，速度同样会很快。

通过最近这个案例的分析，想必大家能够感悟到一些针对具体的查询场景来设计你的联合索引和辅助索引的技巧

核心重点就是，尽量利用一两个复杂的多字段联合索引，抗下你80%以上的 查询，然后用一两个辅助索引抗下剩余20%的非典型查询，保证你99%以上的查询都能充分利用索引，就能保证你的查询速度和性能！

End

提纲挈领的告诉你，SQL语句的执行计划和性能优化有什么关系?

之前我们已经彻底搞清楚了MySQL的索引结构，也知道了索引平时要怎么样写SQL才能用上，而且也是用一个案例给大家讲解了，平时我们做一个系统，写好代码之后，要如何去设计表的索引，让每个查询都可以用上索引，所以这里纠正了大家平时的一个观念，可能有些人认为，平时设计好表就必须同时设计好索引，其实完全不是这么回事。

一般开发一个系统，都是先设计表结构，表结构必须满足业务需求，然后写代码，代码都写好之后，再根据你的代码如何查询表的，来设计里面的索引，考虑设计几个索引，是不是联合索引，选择哪些字段，字段顺序如何排列，才能让查询语句都用上索引。

那么接着我们就要进入MySQL学习中极为重要的一个环节了，那就是MySQL的查询语句的执行计划分析以及SQL优化，这可以说是MySQL实践中对于开发人员最常见最需要掌握的一个技能了，但是很多人对MySQL内核级的原理的理解较为肤浅，同时对于索引结构和查询时使用索引的原理也不甚了解，更谈不上说能看懂MySQL的执行计划了。

如果是这样的话，你还怎么去说自己可以进行MySQL的SQL优化呢?

可能有人反问了 ，你不是都告诉我们索引结构和使用原理了么，使用规则我们也知道了，那SQL优化无非就是开发的时候让自己写的SQL都用上索引不就ok了?

这个话也对，也不对。应该这么说，根据查询语句设计良好的索引，让所有查询都尽可能用上索引，这本身就是一种SQL优化的技巧，但是他仅仅只是其一罢了，并不能说掌握这个，就掌握了所有的SQL优化技巧

反过来说，SQL优化技巧中包含了我们之前讲的设计索引以及让SQL用上索引，但是SQL优化还有很多其他的东西。

实际上有时候往往你会发现自己的数据库里有很多表，每个表的数据量也不小，然后写出来的SQL也比较复杂，各种关联和嵌套子查询，搞的人看的都眼晕，然后表面上看起来这个SQL部分用上了索引，结果性能还是差，差，差，这是为什么呢?

所以说，基础的以及日常的SQL优化就是设计好索引，让一般不太复杂的普通查询都用上索引，但是针对复杂表结构和大数据量的上百行复杂SQL的优化，必须得建立在你先懂这个复杂SQL是怎么执行的

你有那么多的数据表，每个表都有一个聚簇索引，聚簇索引的叶子就是那个表的真实数据，同时每个表还设计了一些二级索引，那么上百行的复杂SQL跑起来的时候到底是如何使用各个索引，如何读取数据的?

这个SQL语句（不管是简单还是复杂），在实际的MySQL底层，针对磁盘上的大量数据表、聚簇索引和二级索引，如何检索查询，如何筛选过滤，如何使用函数，如何进行排序，如何进行分组，到底怎么能把你想要的东西查出来，这个过程就是一个很重要的东西：**执行计划！**

也就是说，每次你提交一个SQL给MySQL，他内核里的查询优化器，都会针对这个SQL语句的语义去生成一个执行计划，这个执行计划就代表了，他会怎么查各个表，用哪些索引，如何做排序和分组，看懂这个执行计划，你就学会了真正的SQL优化的一半了！

当你看懂执行计划之后，还能根据他的实际情况去想各种办法改写你的SQL语句，改良你的索引设计，进而优化SQL语句的执行计划，最终让SQL语句的性能得到提升，这个就是所谓的SQL调优

好了，今天先提纲挈领的给大家讲一下执行计划和SQL优化的关系，下一次开始，我们就正式先学习如何读懂MySQL的SQL执行计划！

End

以MySQL单表查询来举例，看看执行计划包含哪些内容（1）？

今天咱们就以MySQL单表查询来举例，看看执行计划到底包含哪些内容

其实只要大家跟着专栏一步一步的学习下来，会很轻松的看懂执行计划，但是如果你之前对什么数据页，索引，索引使用规则，这些东西学的不扎实，那你可能会觉得现在看着吃力，很痛苦，如果你觉得痛苦，那就回过头去看看之前的内容，学扎实一些。

今天和下次要讲解的执行计划包含的内容，说白了，全是之前讲过的，只不过我们把之前讲过的一些东西跟MySQL的执行计划中的一些概念匹配起来，这样衔接上之后，你就知道MySQL的执行计划里出现一些专业术语和名词的时候，对应的是底层的什么行为。

我们之前都知道，假设你写一个select * from table where id=x，或者select * from table where name=x的语句，直接就可以通过聚簇索引或者二级索引+聚簇索引回源，轻松查到你要的数据，这种根据索引直接可以快速查找数据的过程，在执行计划里称之为const，意思就是性能超高的常量级的。

所以你以后在执行计划里看到const的时候，就知道他就是直接通过索引定位到数据，速度极快，这就是const的意思。

但是这里有一个要点，你的二级索引必须是唯一索引，才是属于const方式的，也就是说你必须建立unique key唯一索引，保证一个二级索引的每一个值都是唯一的，才可以。

那么如果你是一个普通的二级索引呢？就是个普通的KEY索引，这个时候如果你写一个select * from table where name=x的语句，name是个普通二级索引，不是唯一索引，那么此时这种查询速度也是很快的，他在执行计划里叫做ref。

如果你是包含多个列的普通索引的话，那么必须是从索引最左侧开始连续多个列都是等值比较才可以是属于ref方式，就是类似于select * from table where name=x and age=x and xx=xx，然后索引可能是个KEY(name,age,xx)。

然后一个例外，就是如果你用name IS NULL这种语法的话，即使name是主键或者唯一索引，还是只能走ref方式。但是如果你是针对一个二级索引同时比较了一个值还有限定了IS NULL，类似于select * from table where name=x and name IS NULL，那么此时在执行计划里就叫做ref_or_null

说白了，就是在二级索引里搜你要的值以及是NULL的值，然后再回源去聚簇索引里查罢了，因为同时有索引等值比较和NULL值查询，就叫做ref_or_null了，其实也没啥。

那这个ref就说完了，到这里大家先停顿一下，稍微来点深度思考，我们换个角度看，假设你以后在分析执行计划的时候看到了const，那是什么？对，肯定是通过主键或者唯一索引的访问，速度超高。

如果你看到了ref是什么意思？对，就是用了普通的索引，或者用主键/唯一索引搞了一个IS NULL/IS NOT NULL。

所以说，我们别急着去看后续的内容，先思考一下，以后你在执行计划里看到const和ref，是不是立马就知道他们底层都是基于什么方式来查询的，然后如果反映到之前画的很多图里，是如何查询那些索引的。

不管怎么说，只要你看到const或者ref，那恭喜你，说明起码这部分执行速度是很快的！而且相信大家结合之前的内容思考一下，立马就知道那部分查询是怎么做的，底层是通过哪些索引怎么查询的，这个之前都讲过了，下一次我们继续看执行计划里可能有的其他部分。

End

以MySQL单表查询来举例，看看执行计划包含哪些内容（2）？

今天我们继续来说执行计划里包含的数据访问方式，上次说了const和ref，以及ref_or_null，想必大家都理解了，今天来说说其他的数据访问方式

先说说range这个东西，这个东西顾名思义，其实就是你SQL里有范围查询的时候就会走这个方式。

比如写一个SQL是select * from table where age>=x and age <=x，假设age就是一个普通索引，此时就必然利用索引来进行范围筛选，一旦利用索引做了范围筛选，那么这种方式就是range。

接着停下脚步做个总结，假设你在执行计划里看到了const、ref和range，他们是什么意思?

别担心，他们都是说基于索引在查询，总之都是走索引，所以一般问题不是太大，除非你通过索引查出来的数据量太多了，比如上面那个范围筛选，一下子查出来10万条数据，那不是想搞死MySQL么！是不是!

下面我们来讲一种比较特殊的数据访问方式，就是index，可能有的人看到这个index，天真的认为，这不就是通过索引来获取数据么，从索引根节点开始一通二分查找，不停的往下层索引跳转，就可以了，速度超快，感觉上跟ref或者range是一回事。

那你就大错特错了!

假设我们有一个表，里面完整的字段联合索引是KEY(x1,x2,x3)，好，现在我们写一个SQL语句是select x1,x2,x3 from table where x2=xxx，相信大多数同学看到这里，都会觉得，完蛋了，x2不是联合索引的最左侧的那个字段啊!

对的，这个SQL是没办法直接从联合索引的索引树的根节点开始二分查找，快速一层一层跳转的，那么他会怎么执行呢？不知道大家是否发现这个SQL里要查的几个字段，就是联合索引里的几个字段，巧了!

所以针对这种SQL，在实际查询的时候，就会直接遍历KEY(x1,x2,x3)这个联合索引的索引树的叶子节点，大家还记得聚簇索引和普通索引的叶子节点分别存放了什么吗?

聚簇索引的叶子节点放的是完整的数据页，里面包含完整的一行一行的数据，联合索引的叶子节点放的也是页，但是页里每一行就x1、x2、x3和主键的值!

所以此时针对这个SQL，会直接遍历KEY(x1,x2,x3)索引树的叶子节点的那些页，一个接一个的遍历，然后找到 x2=xxx 的那个数据，就把里面的x1，x2，x3三个字段的值直接提取出来就可以了！这个遍历二级索引的过程，要比遍历聚簇索引快多了，毕竟二级索引叶子节点就包含几个字段的值，比聚簇索引叶子节点小多了，所以速度也快！

也就是说，此时只要遍历一个KEY(x1,x2,x3)索引就可以了，不需要回源到聚簇索引去！**针对这种只要遍历二级索引就可以拿到你想要的数据，而不需要回源到聚簇索引的访问方式，就叫做index访问方式！**

是不是跟大家一开始理解的很不一样？没错，所以理解执行计划的前提，是对索引结构和使用索引的原理有一个透彻的理解，在这个基础之上，很容易就可以理解各种各样的执行计划里的访问方式了，脑子里甚至直接可以知道不同的访问方式在图里的执行路径。

现在我们停一下脚步，思考一下，之前说的const、ref和range，本质都是基于索引树的二分查找和多层跳转来查询，所以性能一般都是很高的，然后接下来到index这块，速度就比上面三种要差一些了，因为他是走遍历二级索引树的叶子节点的方式来执行了，那肯定比基于索引树的二分查找要慢多了，但是还是比全表扫描好一些的。

End

88 再次重温写出各种SQL语句的时候，会用什么执行计划？（1）

今天开始，我们将用连续三篇文章给大家去重温平时我们写的SQL语句在执行的时候会用什么样的执行计划，因为我们讲完了SQL语句使用索引的规则和规律，也讲过了不同的使用索引的方法对应着执行计划里的什么访问方式，接下来就可以重温一下，直接把我们平时写的SQL语句和执行计划关联起来了。

首先，我们已经学习了const、ref、range、index几种执行计划里的访问方式，const、ref和range本质都是基于索引查询，只要你索引查出来的数据量不是特别大，一般性能都极为高效，index稍微次一点，需要遍历某个二级索引，但是因为二级索引比较小，所以遍历性能也还可以的。

另外最次的一种就是all了，all意思就是直接全表扫描，扫描你的聚簇索引的所有叶子节点，也就是一个表里一行一行数据去扫描，如果一个表就几百条数据那还好，如果是有几万条，或者几十万，几百万数据，全表扫描基本就得跪了。

那么大家对之前讲的一些特别简单的SQL语句，其实都知道会用什么样的执行计划和访问方式了，也知道不同的访问方式是如何使用索引的，今天开始我们来继续讲讲更多的SQL语句你写出来之后，会用什么样的执行计划。

首先大家看一个SQL语句：select * from table where x1=xx or x2>=xx，这个SQL语句要查一个表，用了x1和x2两个字段，此时有人可能会说了，要是你对x1和x2建了一个联合索引，那不就直接可以通过索引去扫描了？

但是万一要是你建的索引是两个呢？比如(x1,x3)，(x2,x4)，你建了两个联合索引，此时你这个SQL只能选择其中一个索引去用，此时会选择哪个呢？这里MySQL负责生成执行计划的查询优化器，一般会选择在索引里扫描行数比较少的那个条件。

比如说x1=xx，在索引里只要做等值比较，扫描数据比较少，那么可能就会挑选x1的索引，做一个索引树的查找，在执行计划里，其实就是一个ref的方式，找到几条数据之后，接着做一个回表，回到聚簇索引里去查出每条数据完整数据，接着加载到内存里，根据每条数据的x2字段的值，根据x2>=xx条件做一个筛选。

这就是面对两个字段都能用索引的时候如何选择，以及如何进行处理的方式。

接着我们再来考虑另外一种情况，就是：select * from table where x1=xx and c1=xx and c2>=xx and c3 IS NOT NULL

其实我们平时经常会写出来类似这样的SQL语句，就是在一个SQL的所有筛选条件里，就一个x1是有索引的，其他字段都是没有索引的。

这种情况其实也是非常常见的，一般我们在写好一个系统之后，针对所有的SQL分析时，当然不可能针对所有的SQL里的每一个where里的字段都加一个索引，那是不现实的，最终我们只能在所有的SQL语句里，抽取部分经常在where里用到的字段来设计两三个联合索引。

所以在这种情况下，必然很多SQL语句里，可能where后的条件有好几个，结果就一个字段可以用到索引的，此时查询优化器生成的执行计划，就会仅仅针对x1字段走一个ref访问，直接通过x1字段的索引树快速查找到指定的一波数据。

接着对这波数据都回表到聚簇索引里去，把每条数据完整的字段都查出来，然后都加载到内存里去。接着就可以针对这波数据的c1、c2、c3字段按照条件进行筛选和过滤，最后 拿到的就是符合条件的数据了。

所以你的x1索引的设计，必然尽可能是要让x1=xx这个条件在索引树里查找出来的数据量比较少，才能保证后续的性能比较高。

End

89 再次重温写出各种SQL语句的时候，会用什么执行计划？（2）

今天我们来看一个比较奇特的SQL语句以及特殊的执行计划，之前我们都是说，一般一个SQL语句只能用到一个二级索引，但是有一些特殊的情况下，可能会对一个SQL语句用到多个二级索引，这是怎么回事呢？

比如有这么一个SQL：select * from table where x1=xx and x2=xx，然后x1和x2两个字段分别都有一个索引，其实也有一定的可能会让查询优化器生成一个执行计划，执行计划里，就先对x1字段的索引树进行查找，查出一波数据，接着对x2的索引树查出一波数据，然后对两波数据，按照主键值做一个交集。

这个交集就是符合两个条件的数据了，接着回表到聚簇索引去查完整数据就可以了。

但是其实之前我们对这种情况一直说的是，选择x1或者x2其中一个字段的索引，就查一个字段的索引，找出一波数据，接着直接回表到聚簇索引查完整数据，然后根据另外一个字段的值进行过滤就可以了。

那么到底什么情况下，会直接对两个字段的两个索引一起查，然后取交集再回表到聚簇索引呢？也就是什么情况下可能会对一个SQL执行的时候，一下子查多个索引树呢？其实很简单，大家可以思考一下。

假设就上面那个SQL语句吧，比如你x1和x2两个字段，如果你先查x1字段的索引，一下子弄出来上万条数据，这上万条数据都回表到聚簇索引查完整数据，再根据x2来过滤，你有没有觉得效果不是太好？

那如果说同时从x2的索引树里也查一波数据出来，做一个交集，一下子就可以让交集的数据量变成几十条，再回表查询速度就很快了。一般来说，查索引树速度都比较快，但是到聚簇索引回表查询会慢一些。

所以如果同时查两个索引树取一个交集后，数据量很小，然后再回表到聚簇索引去查，此时会提升性能。

但是如果要在一个SQL里用多个索引，那有很多硬性条件的要求，比如说如果有联合索引，你必须把联合索引里每个字段都放SQL里，而且必须都是等值匹配；或者是通过主键查询+其他二级索引等值匹配，也有可能会做一个多索引查询和交集。

其实大家看这个可能看的很迷惑，但是不用迷惑，其实你只要记住，在执行SQL语句的时候，有可能是会同时查多个索引树取个交集，再回表到聚簇索引的，这个可能性是有的。大家只要记住这个结论就行了，后续在分析真实执行计划的时候，我们会再提到这个。

End

90 再次重温写出各种SQL语句的时候，会用什么执行计划？（3）

今天我们继续看看写出各种SQL语句的时候，会有什么样的执行计划？其实这些都是MySQL优化的一些基础知识。

如果大家不能把这些理论知识夯的很扎实的话，那么后续的多个MySQL SQL调优实战案例根本不可能会看懂，因为调优的前提，就是彻底搞明白执行计划，也就是彻底搞明白你的一个SQL，现在性能差，他是如何执行的，为什么性能会这么差，应该怎么改写或者设计索引，才能让他的性能变得更好。

之前讲了，有的时候可能会在一个SQL里同时用上多个索引，那么其实如果你在SQL里写了类似x1=xx or x2=xx的语句，也可能会用多个索引，只不过查多个大索引树之后，会取一个并集，而不是交集罢了。

那么现在为止，我们要做一个小小的停顿和总结，就是现在大家已经知道写出来的SQL有哪些执行的方式了。const、ref、range，都是性能最好的方式，说明在底层直接基于某个索引树快速查找了数据了，但有的时候可能你在用了索引之后，还会在回表到聚簇索引里查完整数据，接着根据其他条件来过滤。

然后index方式其实是扫描二级索引的意思，就是说不通过索引树的根节点开始快速查找，而是直接对二级索引的叶子节点遍历和扫描，这种速度还是比较慢的，大家尽量还是别出现这种情况。

当然index方式怎么也比all方式好一些，all就是直接全表扫描了，也就是直接扫描聚簇索引的叶子节点，那是相当的慢，index虽然扫描的是二级索引的叶子节点，但是起码二级索引的叶子节点数据量比较小，相对all要快一些。

然后之前给大家说的可能一个SQL里用多个索引，意思就是可能对多个索引树进行查找，接着用intersection交集、union并集的方式来进行合并，此时可能给你在执行计划里也会看到这些字样，那你起码这里要知道是怎么回事，其实他就是告诉你，他查找了多个索引，做了一些结果集的交集或者是并集，而且这种方式也不一定是会发生的。

好了，到这里为止，大家把一些基本的执行计划里的东西都了解差不多了，这其实都是一些单表查询的执行计划可能包含的内容，下周开始，正式讲解MySQL的多表关联的SQL语句会对应哪些执行计划，讲完多表关联的执行计划原理之后，还会讲解MySQL生成执行计划的原理，包括子查询之类的复杂SQL是如何生成执行计划的。

最后我们会讲多个案例，来给大家用真实复杂的SQL语句，来看MySQL生成的真实执行计划，彻底搞定SQL语句是如何执行的，然后再切入SQL调优实战案例，到时候大家一步一步的进行，就会觉得非常的自然了。

End

91 深入探索多表关联的SQL语句到底是如何执行的？（1）

之前我们已经用很大的篇幅讲完了针对单表的查询SQL语句，通常都会使用哪些执行计划，如何去使用索引去查找数据，想必大家都已经透彻的掌握这些知识了，比如以后在执行计划里看到const、ref、range、index、all以及多索引查询合并的一些字样，都知道具体在磁盘数据层面是如何执行的了

那么今天开始，我们来进入一块极为重要的知识领域，那就是MySQL的多表关联查询SQL语句是如何执行的？

大家都知道，平时一般如果我们仅仅是执行一下单表查询，那都是比较简单的，而且通常你把索引给建好了，让他尽可能走索引，性能都不是什么大问题。

但是往往我们平时基于MySQL做一些系统开发的时候，比较多的是写一些多表关联语句，因为有时候想要查找你需要的数据，不得不借助多表关联的语法去编写SQL语句，才能实现你想要的逻辑和语义，但是往往使用多表关联的时候，你的SQL性能就可能会遇到一些问题。

那么今天开始，我们就一起来看看，这个多表关联SQL语句到底是如何执行的吧。

今天先来给大家讲解一个超级简单，最最基础的多表关联查询的执行原理，假设我们有一个SQL语句是：select * from t1,t2 where t1.x1=xxx and t1.x2=t2.x2 and t2.x3=xxx

就这么一个SQL语句，大家知道他是什么意思吗？

首先，如果你在FROM字句后直接来了两个表名，这意思就是要针对两个表进行查询了，而且会把两个表的数据给关联起来，假设你要是没有限定什么多表连接条件，那么可能会搞出一个笛卡尔积的东西。

举个例子，假设t1表有10条数据，t2表有5条数据，那么此时select * from t1,t2，其实会查出来50条数据，因为t1表里的每条数据都会跟t2表里的每条数据连接起来返回给你，那么不就是会查出来10 * 5 = 50条数据吗？这就是笛卡尔积

不过通常一般没人会傻到写类似这样的SQL语句，因为查出来这种数据实在是没什么意义。所以通常都会在多表关联语句中的WHERE子句里引入一些关联条件，那么我们回头看看之前的SQL语句里的WHERE子句：where t1.x1=xxx and t1.x2=t2.x2 and t2.x3=xxx

首先呢，t1.x1=xxx，这个可以明确，绝对不是多表关联的连接条件，他是针对t1表的数据筛选条件，本质就是从t1表里筛选一波数据出来再跟t2表做关联的意思。然后t2.x3=xxx，也不是关联条件，他也是针对t2表的筛选条件。

其实真正的关联条件是t1.x2=t2.x2，这个条件，意思就是说，必须要让t1表里的每条数据根据自己的x2字段的值去关联上t2表里的某条记录，要求是t1表里这条数据的x2值和t2表里的那条数据的x2字段值是相等的。

举个例子，假设t1表里有1条数据的x2字段的值是265，然后t2表里有2条数据的x2字段的值也是265，那么此时就会把t1表里的那条数据和t2表的2条数据分别关联起来，最终会返回给你两条关联后的数据。

那么基本概念理解清楚了，具体到上面的SQL语句：select * from t1,t2 where t1.x1=xxx and t1.x2=t2.x2 and t2.x3=xxx

其实这个SQL执行的过程可能是这样的，首先根据t1.x1=xxx这个筛选条件，去t1表里查出来一批数据，此时可能是const、ref，也可能是index或者all，都有可能，具体看你的索引如何建的，他会挑一种执行计划访问方式。

然后假设从t1表里按照t1.x1=xxx条件筛选出2条数据，接着对这两条数据，根据每条数据的x2字段的值，以及t2.x3=xxx这个条件，去t2表里找x2字段值和x3字段值都匹配的数据，比如说t1表第一条数据的x2字段的值是265，此时就根据t2.x2=265和t2.x3=xxx这俩条件，找出来一波数据，比如找出来2条吧。

此时就把t1表里x2字段为265的那个数据跟t2表里t2.x2=265和t2.x3=xxx的两条数据，关联起来，就可以了，t1表里另外一条数据也是如法炮制而已，这就是多表关联最最基本的原理。

记住，他可能是先从一个表里查一波数据，这个表叫做“驱动表”，再根据这波数据去另外一个表里查一波数据进行关联，另外一个表叫做“被驱动表”

End

92 深入探索多表关联的SQL语句到底是如何执行的？（2）

今天我们来继续跟大家聊聊多表关联语句是如何执行的这个问题，上次讲了一个最最基础的两个表关联的语句和执行过程，其实今天我们稍微来复习一下，然后接着上次的内容，引入一个“内连接”的概念来。

假设我们有一个员工表，还有一个产品销售业绩表，员工表里包含了id（主键）、name（姓名）、department（部门），产品销售业绩表里包含了id（主键）、employee_id（员工id）、产品名称（product_name）、销售业绩（saled_amount）。

现在假设你想看看每个员工对每个产品的销售业绩，写个SQL：

select e.name,e.department,ps.product_name,ps.saled_amount from employee e,product_saled pa where e.id=pa.employee_id

此时看到的数据可能如下：

员工 部门 产品 业绩

张三 大客户部 产品A 30万

张三 大客户部 产品B 50万

张三 大客户部 产品C 80万

李四 零售部 产品A 10万

李四 零售部 产品B 12万

至于上述SQL的执行原理，相信大家应该都理解，其实就是从员工表里走全表扫描，找出每个员工，然后针对每个员工的id去业绩表里找 employee_id 跟员工id相等的数据，可能每个员工的id在业绩表里都会找到多条数据，因为他可能有多个产品的销售业绩。

然后就是把每个员工数据跟他在业绩表里找到的所有业绩数据都关联起来，比如张三这个员工就关联了业绩表里的三条数据，李四这个员工关联上了业绩表里的两条数据。

其实大家已经在不知不觉中学会了最基本的一个SQL关联语法，就是内连接，这个内连接，英语是inner join，意思就是要求两个表里的数据必须是完全能关联上的，才能返回回来，这就是内连接。

那么现在有这么一个问题，假设员工表里有一个人是新员工，入职到现在一个单子都没开过，也就没有任何的销售业绩，那么此时还是希望能够查出来这个员工的数据，只不过他的销售业绩那块可以给个NULL就行了，表示他没任何业绩。

但是如果仅仅是使用上述SQL语法，似乎是搞不定的，因为那种语法要求，必须要两个表能关联上的数据才会查出来，像你员工表里可能有个王五，根本在业绩表里关联不上任何数据，此时这个人是不会查出来的。

所以此时就要到外连接了，也就是outer join，这个outer join分为左外连接和右外连接，左外连接的意思就是，在左侧的表里的某条数据，如果在右侧的表里关联不到任何数据，也得把左侧表这个数据给返回出来，右外连接反之，在右侧的表里如果关联不到左侧表里的任何数据，得把右侧表的数据返回出来。

而且，这里还有一个语法限制，如果你是之前的那种内连接，那么连接条件是可以放在where语句里的，但是外连接一般是把连接条件放在ON字句里的，所以此时可以写出如下的SQL语句：

```
SELECT
e.name,
e.department,
ps.product_name,
ps.saled_amount
FROM employee e LEFT OUTER JOIN product_saled pa
ON e.id=pa.employee_id
```

此时返回的数据里，你可能会看到如下的结果：

```
员工 部门 产品 业绩
张三 大客户部 产品A 30万
张三 大客户部 产品B 50万
张三 大客户部 产品C 80万
李四 零售部 产品A 10万
李四 零售部 产品B 12万
王五 零售部 NULL NULL
```

所以说，到这里为止，想必大家都很清楚了，其实一般写多表关联，主要就是内连接和外连接，连接的基本语义和实现过程，大家应该也有一定的理解了。

End

93 深入探索多表关联的SQL语句到底是如何执行的？（3）

之前我们把连接的基本语义和基本原理讲了一下，今天开始正式来深入探索一下SQL关联语法的实现原理

首先，先给大家提出一个名词叫做：**嵌套循环关联（nested-loop join）**，这其实就是我们之前给大家提到的最基础的关联执行原理。

简单来说，假设有两个表要一起执行关联，此时会先在一个驱动表里根据他的where筛选条件找出一波数据，比如说找出10条数据吧

接着呢，就对这10条数据走一个循环，用每条数据都到另外一个被驱动表里去根据ON连接条件和WHERE里的被驱动表筛选条件去查找数据，找出来的数据就进行关联。

依次类推，假设驱动表里找出来10条数据，那么就要到被驱动表里去查询10次！

那么如果是三个表进行关联呢？那就更夸张了，你从表1里查出来10条数据，接着去表2里查10次，假设每次都查出来3条数据，然后关联起来，此时你会得到一个30条数据的结果集，接着再用这批数据去表3里去继续查询30次！

这种方法的伪代码有点类似下面这样：

```
t1Rows = queryFromt1()  // 根据筛选条件对t1标进行查询
for t1Row in t1Rows {   // 对t1里每一条符合条件的数据进行循环
    t2Rows = queryFromt2(t1Row) // 拿t1里的数据去t2表里查询以及做关联
    for t2Row in t2Rows {   // 对t1和t2关联后的数据进行循环
        t3Rows = queryFromt3(t2Row)     // 拿t1和t2关联后的数据去t3表里查询和关联
        for t3Row in t3Rows {  // 遍历最终t1和t2和t3关联好的数据

        }
    }
}
```

上面那伪代码其实就是3个表关联的伪代码，用的就是最笨的嵌套循环关联方法，大家可以好好理解上面的伪代码。

不知道大家有没有发现上面那种多表关联方法的问题在哪里？

没错，就是我们往往从驱动表里查出来一波数据之后，要对每一条数据都循环一次去被驱动表里查询数据，所以万一你要是被驱动表的索引都没建好，总不能每次都全表扫描吧？这就是一个很大的问题！

另外一个，刚开始对你的驱动表根据WHERE条件进行查询的时候，也总不能全表扫描吧？这也是一个问题！

所以说，为什么有的时候多表关联很慢呢？答案就在这里了，你两个表关联，先从驱动表里根据WHERE条件去筛选一波数据，这个过程如果你没给驱动表加索引，万一走一个all全表扫描，岂不是速度很慢？

其次，假设你好不容易从驱动表里扫出来一波数据，接着又来一个for循环一条一条去被驱动表里根据ON连接条件和WHERE筛选条件去查，万一你对被驱动表又没加索引，难道又来几十次或者几百次全表扫描？那速度岂不是慢的跟蜗牛一样了！

所以说，通常而言，针对多表查询的语句，我们要尽量给两个表都加上索引，索引要确保从驱动表里查询也是通过索引去查找，接着对被驱动表查询也通过索引去查找。如果能做到这一点，你的多表关联语句性能就会很高！

End

94 MySQL是如何根据成本优化选择执行计划的？（上）

之前已经给大家讲解清楚了 MySQL 在执行单表查询时候的一些执行计划，比如说const、ref、range、index、all之类的，也讲了多表关联的时候是如何执行的，本质其实就是先查一个驱动表，接着根据连接条件去被驱动表里循环查询，现在大家对MySQL执行查询的一些基本原理都有了一个了解了。

好，那么从今天开始，我们再更深入一步，因为其实大家之前或多或少也感觉到了一个问题，就是其实我们在执行单表查询也好，多表关联也好，似乎都有多种执行计划可以选择，比如有的表可以全表扫描，也可以用索引A，也可以用索引B，那么到底是用哪种执行计划呢？

所以今天开始，我们用为期两周的时间，彻底给大家讲解清楚MySQL是如何对一个查询语句的多个执行计划评估他的成本的？如何根据成本评估选择一个成本最低的执行计划，保证最佳的查询速度？

大家耐心学习，我们已经一点一点接近了MySQL查询原理的本质了，当大家透彻理解了这些内容，再去学习通过explain看真实的SQL语句的执行计划，就会完全明白是怎么回事了。当你能透彻理解了explain看SQL执行计划之后，那么任何SQL语句的调优都不在话下。

我们先了解一下MySQL里的成本是什么意思，简单来说，跑一个SQL语句，一般成本是两块，首先是那些数据如果在磁盘里，你要不要从磁盘里把数据读出来？这个从磁盘读数据到内存就是IO成本，而且MySQL里都是一页一页读的，读一页的成本的约定为1.0。

然后呢，还有一个成本，那就是说你拿到数据之后，是不是要对数据做一些运算？比如验证他是否符合搜索条件了，或者是搞一些排序分组之类的事，这些都是耗费CPU资源的，属于CPU成本，一般约定读取和检测一条数据是否符合条件的成本是0.2.

这个所谓1.0和0.2就是他自定义的一个成本值，代表的意思就是一个数据页IO成本就是1.0，一条数据检测的CPU成本就是0.2，就这个意思罢了。

然后呢，当你搞一个SQL语句给MySQL的时候，比如：

select * from t where x1=xx and x2=xx

此时你有两个索引，分别是针对x1和x2建立的，就会先看看这个SQL可以用到哪几个索引，此时发现x1和x2的索引都能用到，他们俩索引就是possible keys。

接着会针对这个SQL计算一下全表扫描的成本，这个全表扫描的话就比较坑了，因为他是需要先磁盘IO把聚簇索引里的叶子节点上的数据页一页一页都读到内存里，这有多少数据页就得耗费多少IO成本，接着对内存里的每一条数据都判断是否符合搜索条件的，这有多少条数据就要耗费多少CPU成本。

所以说，此时就得计算一下这块成本有多少，怎么算呢？简单，教大家一个命令：

show table status like "表名"

可以拿到你的表的统计信息，你在对表进行增删改的时候，MySQL会给你维护这个表的一些统计信息，比如这里可以看到rows和data_length两个信息，不过对于innodb来说，这个rows是估计值。

rows就是表里的记录数，data_length就是表的聚簇索引的字节数大小，此时用data_length除以1024就是kb为单位的大小，然后再除以16kb（默认一页的大小），就是有多少页，此时知道数据页的数量和rows记录数，就可以计算全表扫描的成本了。

IO成本就是：数据页数量 * 1.0 + 微调值，CPU成本就是：行记录数 * 0.2 + 微调值，他们俩相加，就是一个总的成本值，比如你有数据页100个，记录数有2万条，此时总成本值大致就是100 + 4000 = 4100，在这个左右。

好，今天先讲到这儿，大家先知道了一个全表扫描执行计划的成本计算方法，下次我们继续讲索引的成本计算方法。

End

95 MySQL是如何根据成本优化选择执行计划的？（中）

上次我们讲完了全表扫描的成本计算方法，相信大家应该都理解了，其实还是比较简单的，今天我们来讲一下索引的成本计算方法，因为除了全表扫描之外，还可能多个索引都可以使用，但是当然同时一般只能用一个索引，所以不同索引的使用成本都得计算一下。

这个使用索引访问数据的方式，大家应该都还记得，其实很简单，除非你直接根据主键查，那就直接走一个聚簇索引就ok了，否则普通索引，一般都是两步走，先从二级索引查询一波数据，再根据这波数据的主键去聚簇索引回表查询。

这个过程的成本计算方法稍微有点特别，首先，在二级索引里根据条件查一波数据的IO成本，一般是看你的查询条件涉及到几个范围，比如说name值在25~100，250~350两个区间，那么就是两个范围，否则name=xx就仅仅是一个范围区间。

一般一个范围区间就粗暴的认为等同于一个数据页，所以此时可能一般根据二级索引查询的时候，这个IO成本都会预估的很小，可能就是1 * 1.0 = 1，或者是n * 1.0 = n，基本就是个位数这个级别。

但是到此为止，还仅仅是通过IO读取了二级索引的数据页而已，这仅仅是二级索引读取的IO成本，但是二级索引数据页到内存里以后，还得根据搜索条件去拿出来一波数据，拿这波数据的过程就是根据搜索条件在二级索引里搜索的过程。

此时就要估算从二级索引里读取符合条件的数据的成本了，这需要估算一下在二级索引里会查出多少条数据，这个过程就稍微有点复杂了，不细讲了，总之呢，他会根据一个不是怎么太准确的算法去估算一下根据查询条件可能会在二级索引里查出多少条数据来。

估算出来之后，比如估算可能会查到100条数据，此时从二级索引里查询数据的CPU成本就是100 * 0.2 + 微调值，总之就是20左右而已。

接着你拿到100条数据之后，就得回表到聚簇索引里去查询完整数据，此时先估算回表到聚簇索引的IO成本，这里比较粗暴的直接默认1条数据就得回表到聚簇索引查询一个数据页，所以100条数据就是100个数据页的IO成本，也就是100 * 1.0 + 微调值，大致是100左右。

接着因为在二级索引里搜索到的数据是100条，然后通过IO成本最多回表到聚簇索引访问100个数据页之后，就可以拿到这100条数据的完整值了，此时就可以针对这100条数据去判断，他们是否符合其他查询条件了，这里耗费的CPU成本就是100 * 0.2 + 微调值，就是20左右。

把上面的所有成本都加起来，就是1 + 20 + 100 + 20 = 141，这就是使用一个索引进行查询的成本的计算方法，其实大家看明白这个过程了，那么每一个索引的成本计算过程就都明了了，假设你直接根据主键查询，那么也参考上述估算过程就可以了，那就不过是仅仅查询一个聚簇索引罢了。

总之，上次讲到全表扫描发现成本是4100左右，这次根据索引查找可能就141，所以，很多时候，使用索引和全表扫描，他的成本差距是非常之大的。所以一般就会针对全表扫描和各个索引的成本，都进行估算，然后比较一下，选择一个成本最低的执行计划。

End

96 MySQL是如何根据成本优化选择执行计划的？（下）

今天是我们讲解根据成本优化选择执行计划的最后一讲，下周就要给大家讲解基于规则的执行计划优化了，也就是MySQL是如何自动调整我们的SQL语句为性能比较优化的方式，好，那今天一起看看多表关联查询是如何选择执行计划的。

其实多表查询的执行计划选择思路，基本跟单表查询的执行计划选择思路是类似的，因为大家应该都记得，单表查询的时候，主要就是对这个表的多种访问方式（全表查询，各个索引查询）来根据一定的公式计算出来每种访问方式的成本，接着选择一个成本最低的访问方式，那么就可以确定下来这个表怎么访问了。

可能有的人看了之前的两讲，会觉得似乎这种成本计算的方式也不是太靠谱，因为里面有些过程感觉怪怪的，不过这个没办法，其实即使让你来设计，也很难设计出完全公平、完全精准的成本预估算法来

因为要在一个查询执行之前，就可以针对不同的访问方法精准计算他的成本，那是根本不现实的，最后只能是根据一些相对较为简单粗暴的办法，大致估算一下，估算结果可能不是太准确，但是也没办法了，反正算出来也就这么比较就是了。

那么接着如果我们要看看多表关联的成本计算访问和执行计划选择方式，那就很简单了，因为大家应该还记得，多表关联的语句，比如：

select * from t1 join t2 on t1.x1=t2.x1 where t1.x2=xxx and t1.x3=xxx and t2.x4=xxx and t2.x5=xxx

就这么一个语句，大家应该还记得他里面的访问过程

一般来说，都会先选择一个驱动表，比如t1作为驱动表，此时就需要根据t1.x2=xxx和t1.x3=xxx这个条件从表里查询一波符合条件的数据出来，此时就有一个问题了，这里用到了t1的两个字段来筛选数据，可能x2和x3字段都建了索引了，此时到底选择哪个索引呢？或者干脆直接就是全表扫描？

此时就会按照之前讲的那套方法来计算针对t1表查询的全表扫描和不同索引的成本，选择一个针对t1表的最佳访问方式，用最低成本从t1表里查出符合条件的数据来，接着就根据这波数据得去t2表里查数据，按照连接条件t1.x1=t2.x1去查，同时要符合t2.x4=xxx和t2.x5=xxx这两个条件。

此时一样会根据之前讲解的办法去估算，针对t2表的全表扫描以及基于x4、x5、x1几个字段不同索引的访问的成本，挑选一个成本最低的方法，然后从t2表里把数据给查找出来，就可以，这就完成了多表关联！

所以大家可以看到，其实多表关联的成本估算以及执行计划选择方式，跟单表关联基本上是差不多的，只不过多表关联要多查几个表罢了。

End

97 MySQL是如何基于各种规则去优化执行计划的？（上）

之前我们已经给大家讲解了单表查询语句和多表关联语句具体的执行原理，同时也给大家讲了在生成具体执行计划的时候，是如何根据成本计算去选择最优执行计划的，因为每个查询执行的时候实际都可能有多种执行计划可供选择，必须要选择成本最低的那种。

接着我们来给大家讲解一下MySQL在执行一些相对较为复杂的SQL语句的时候是如何对查询进行重写来优化具体的执行计划的，因为他有时候可能会觉得你写的SQL一点都不好，直接按你的SQL生成的执行计划效率还是不够高，需要自动帮你改改。

这里有很多很多的规则，可能比较琐碎，但还是需要让大家了解一下MySQL可能会改写我们的SQL语句这个事，所以请大家耐着性子看下去，大概理解一下就行了

首先呢，要是MySQL觉得你的SQL里有很多括号，那么无关紧要的括号他会给你删除了，其次比如你有类似于i = 5 and j > i这样的SQL，就会改写为i = 5 and j > 5，做一个常量替换。

还有比如x = y and y = k and k = 3这样的SQL，都会给你优化成x = 3 and y = 3 and k = 3，本质也是做个常量替换。或者是类似于什么b = b and a = a这种一看就是乱写的SQL，一看就是没意义的，就直接给你删了。

大家可能觉得很琐碎，其实不是的，这些SQL的改写，你会发现，他本质都是在优化SQL语句的清晰语义，方便后续在索引和数据页里进行查找。

还有一些是比较有意思的改写，比如下面的SQL语句：

select * from t1 join t2 on t1.x1=t2.x1 and t1.id=1

这个SQL明显是针对t1表的id主键进行了查询，同时还要跟t2表进行关联，其实这个SQL语句就可能在执行前就先查询t1表的id=1的数据，然后直接做一个替换，把SQL替换为：

select t1表中id=1的那行数据的各个字段的常量值, t2.* from t1 join t2 on t1表里x1字段的常量值=t2.x1

上面的SQL就是直接把t1相关的字段都替换成了提前查出来的id=1那行数据的字段常量值了。

今天就先给大家讲一点开胃菜，也就是大家能知道一下其实你写的SQL语句真正执行时，可能是会对SQL进行各种改动的，接下来我们还会继续分析。

End

98 MySQL是如何基于各种规则去优化执行计划的？（中）

今天我们来讲一下**子查询是如何执行的****，以及他的执行计划是如何优化的**。比如说类似于下面的SQL语句：

select * from t1 where x1 = (select x1 from t2 where id=xxx)

这就是一个典型的子查询

也就是说上面的SQL语句在执行的时候，其实会被拆分为两个步骤：第一个步骤先执行子查询，也就是：select x1 from t2 where id=xxx，直接根据主键定位出一条数据的x1字段的值。接着再执行select * from t1 where x1=子查询的结果值，这个SQL语句。

这个第二个SQL执行，其实也无非就是跟之前讲的单表查询的方式是一样的，其实大家看到最后会发现，这个SQL语句最核心的就是单表查询的几种执行方式，其他的多表关联，子查询，这些都是差不多这个意思。

最多就是在排序、分组聚合的时候，可能有的时候会直接用上索引，有的时候用不上索引就会基于内存或者临时磁盘文件执行。

另外还有一种子查询，就是：

select * from t1 where x1 = (select x1 from t2 where t1.x2=t2.x2)

这种时候，你会发现子查询里的where条件依赖于t1表的字段值，所以这种查询就会效率很低下，他需要遍历t1表里每一条数据，对每一条数据取出x2字段的值，放到子查询里去执行，找出t2表的某条数据的x1字段的值，再放到外层去判断，是否符合跟t1表的x1字段匹配。

其实大家只要理解透彻了前面的内容，现在看这些SQL语句的执行原理都是比较简单的，并没有什么新意，那么接着我们就重点来讲讲这个子查询执行的时候，执行计划上会有哪些优化的规则。

今天我们重点来讲一下IN语句结合子查询的一个优化手段，假设有如下的一个SQL语句：

select * from t1 where x1 in (select x2 from t2 where x3=xxx)

这个SQL语句就是典型的一个子查询运用，子查询查一波结果，然后判断t1表哪些数据的x1值在这个结果集里。

这个可能大家会想当然的认为先执行子查询，然后对t1表再进行全表扫描，判断每条数据是否在这个子查询的结果集里，但是这种方式其实效率是非常低下的。

所以其实对于上述的子查询，执行计划会被优化为，先执行子查询，也就是select x2 from t2 where x3=xxx这条SQL语句，把查出来的数据都写入一个临时表里，也可以叫做物化表，意思就是说，把这个中间结果集进行物化。

这个物化表可能会基于memory存储引擎来通过内存存放，如果结果集太大，则可能采用普通的b+树聚簇索引的方式放在磁盘里。但是无论如何，这个物化表都会建立索引，所以大家要清楚，这波中间结果数据写入物化表是有索引的。

接着大家可能会想，此时是不是全表扫描t1表，对每条数据的x1值都去物化表里根据索引快速查找一下是否在这个物化表里？如果是的话，那么就符合条件了。但是这里还有一个优化的点，那就是他可以反过来思考。

也就是说，假设t1表的数据量是10万条，而物化表的数据量只有500条，那么此时完全可以改成全表扫描物化表，对每个数据值都到t1表里根据x1这个字段的索引进行查找，查找物化表的这个值是否在t1表的x1索引树里，如果在的话，那么就符合条件了。

所以基于IN语句的子查询执行方式，实际上会在底层被优化成如上所述。

End

99 MySQL是如何基于各种规则去优化执行计划的？（下）

今天我们来给大家讲解MySQL里对子查询的执行计划进行优化的一种方式，就是semi join，也就是半连接

这个半连接是什么意思呢，其实就是假设你有一个子查询语句：select * from t1 where x1 in (select x2 from t2 where x3=xxx)，此时其实可能会在底层把他转化为一个半连接，有点类似于下面的样子：

select t1.* from t1 semi join t2 on t1.x1=t2.x2 and t2.x3=xxx

当然，其实并没有提供semi join这种语法，这是MySQL内核里面使用的一种方式，上面就是给大家说那么个意思，其实上面的semi join的语义，是和IN语句+子查询的语义完全一样的，他的意思就是说，对于t1表而言，只要在t2表里有符合t1.x1=t2.x2和t2.x3=xxx两个条件的数据就可以了，就可以把t1表的数据筛选出来了。

其实这个semi join我们这里就是简单提一下概念就行了，但是他还有很多适用场景和不适用场景，我们这里就不再说了，因为也没那个必要，这里先简单了解一下就可以

其实到今天为止，我们就已经把MySQL中各种SQL语句的大致执行原理以及执行计划，都有一个基本的了解了，无论是简单的单表查询，还是多表关联，或者是子查询，大家虽然很多细节还不够熟悉，但是大致的原理基本是都知道了。

而且不同的执行计划到底是如何生成的，可能是根据成本计算选择的，也可能是根据规则优化出来的，然后具体执行计划执行的时候，在底层是如何查询索引的，如何筛选数据，其实大家也都基本清楚了。

到此为止，有了这么多的铺垫，下周开始我们就可以正式进入**执行计划研究环节**了，这是后续能搞定SQL调优的最后一个难题攻关了，只要大家可以看明白各种SQL语句的执行计划以及真实SQL执行过程中各个环节的耗时，找出SQL语句执行慢的原因，那么后续就可以针对性的进行SQL调优。

当然，其实还是要给大家提醒一句，在互联网公司里，我们比较崇尚的是尽量写简单的SQL，复杂的逻辑用Java系统来实现就可以了，SQL能单表查询就不要多表关联，能多表关联就尽量别写子查询，能写几十行SQL就别写几百行的SQL，多考虑用Java代码在内存里实现一些数据就的复杂计算逻辑，而不是都放SQL里做。

其实一般的系统，只要你SQL语句尽量简单，然后建好必要的索引，每条SQL都可以走索引，数据库性能往往不是什么大问题，而接下来要讲的复杂SQL调优，主要是针对那种实在没办法必须写上百行的复杂SQL，然后你还必须得进行调优的情况，当然SQL高级调优也是程序员必须掌握的。

End

100 透彻研究通过explain命令得到的SQL执行计划（1）

今天我们正式进入研究explain命令得到的SQL执行计划的内容了，只要把explain分析得到的SQL执行计划都研究透彻，完全能看懂，知道每个执行计划在底层是怎么执行的，那么后面学习SQL语句的调优就非常容易了。

首先，我们现在应该都知道每条SQL语句，mysql都会经过成本和规则的优化，对这个SQL选择对应的一些访问方法和顺序，包括做一些特殊的改写确保执行效率是最优的，然后优化过后，就会得到一个执行计划。

这个执行计划其实真没那么神秘，如果你把之前的内容都学习的比较透彻的话就会知道，所谓的执行计划，落实到底层，无非就是先访问哪个表，用哪个索引还是全表扫描，拿到数据之后如何去聚簇索引回表，是否要基于临时磁盘文件做分组聚合或者排序，其实这个计划到最后就是这点东西。

平时我们只要用类似于：explain select * from table，这种SQL前面加一个explain命令，就可以轻松拿到这个SQL语句的执行计划。今天我们就先来看看，这个所谓的执行计划里会有哪些东西。

首先，当你执行explain命令之后，拿到的执行计划可能是类似下面这样的东西：

id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra

|+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------+|

1 | SIMPLE | NULL | NULL | NULL | NULL | NULL | NULL | NULL | NULL | NULL | No tables used |

大家看到那所谓的id、select_type、table、partitions、type之类的东西了吗，其实这些就是所谓的执行计划里包含的东西

大致来说，如果是一个简单的单表查询，可能这里就只有一条数据，也就是代表了他是打算如何访问这一个表而已。

但是如果你的SQL语句极为的复杂，可能这里会有很多条数据，因为一个复杂的SQL语句的执行是要拆分为很多步骤的，比如先访问表A，接着搞一个排序，然后来一个分组聚合，再访问表B，接着搞一个连接，类似这样子。

好，那么接下来我们就先来研究一下这个所谓的执行计划里包含的各个字段都是什么意思，首先是id这个东西

这个id呢，就是说每个SELECT都会对应一个id，其实说穿了，就是一个复杂的SQL里可能会有很多个SELECT，也可能会包含多条执行计划，每一条执行计划都会有一个唯一的id，这个没啥好说的。

select_type，顾名思义，说的就是这一条执行计划对应的查询是个什么查询类型，table就是表名，意思是要查询哪个表，partitions是表分区的概念，这个所谓的分区表我们会在后面给大家讲，这里先不用太关注他。

type，就是比较关键了，针对当前这个表的访问方法，这个之前我们都讲过很多，比如说const、ref、range、index、all之类的，分别代表了使用聚簇索引、二级索引、全表扫描之类的访问方式。

possible_keys，这也很关键，他是跟type结合起来的，意思就是说你type确定访问方式了，那么到底有哪些索引是可供选择，可以使用的呢，这都会放这里。key，就是在possible_keys里实际选择的那个索引，而key_len就是索引的长度。

ref，就是使用某个字段的索引进行等值匹配搜索的时候，跟索引列进行等值匹配的那个目标值的一些信息。rows，是预估通过索引或者别的方式访问这个表的时候，大概可能会读取多少条数据。filtered，就是经过搜索条件过滤之后的剩余数据的百分比。extra是一些额外的信息，不是太重要。

好了，今天就先看到这里，明天我们继续讲解对真实的SQL语句分析得到的执行计划会长什么样子，让大家彻底能看懂执行计划。

End

101 透彻研究通过explain命令得到的SQL执行计划（2）

今天我们就一步一步的来讲解不同的SQL语句的执行计划长什么样子，先来看第一条SQL语句，特别的简单，就是：

explain select * from t1

就这么一个简单的SQL语句，那么假设他这个里面有大概几千条数据，此时执行计划看起来是什么样的?

```
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
|  1 | SIMPLE      | t1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 3457 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
```

一起来分析一下上面的执行计划吧，学习当然得从最简单的地方开始一步一步得来，相信每个人都能成为分析SQL执行计划的高手。

首先呢，id是1，这个不用管他了，select_type是SIMPLE，这个先不说他什么意思，你要知道顾名思义，这个表的查询类型是很普通的、而且简单的就可以了。

table是t1，这还用说么？表名就是t1，所以意思就是这里要访问t1这个表。type是all，这就是我们之前提到的多种访问方式之一了，all就是全表扫描，这没办法，你完全没加任何where条件，那当然只能是全表扫描了！

而且如果大家记得我们之前讲解的底层访问方式，就会知道，这里直接会扫描表的聚簇索引的叶子节点，按顺序扫描过去拿到表里全部数据。

rows是3457，这说明全表扫描会扫描这个表的3457条数据，说明这个表里就有3457条数据，此时你全表扫描会全部扫描出来。filtered是100%，这个也很简单了，你没有任何where过滤条件，所以直接筛选出来的数据就是表里数据的100%占比。

怎么样，有没有觉得稍微对执行计划有点感觉了，似乎也没那么难是吧？因为有了之前内容的大量铺垫和积累，大家对SQL语句的底层执行原理本身已经有了一定的理解了，所以看执行计划就会很简单的。

接着再来看一个SQL语句的执行计划：

explain select * from t1 join t2

这是一个典型的多表关联语句，之前我们说过，这种关联语句，实际上会选择一个表先查询出来数据，接着遍历每一条数据去另外一个表里查询可以关联在一起的数据，然后关联起来，此时他的执行计划大概长下面这样子：

```
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                               |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------------------------------------------+
|  1 | SIMPLE      | t1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 3457 |   100.00 | NULL                                                |
|  1 | SIMPLE      | t2    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 4568 |   100.00 | Using join buffer (Block Nested Loop)               |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------------------------------------------+
```

这个执行计划就比较有意思了，因为是一个多表关联的执行计划。首先很明显，他的执行计划分为了两条，也就是会访问两个表，先看他如何访问第一个表的，针对第一个表就是t1，明显是先用ALL方式全表扫描他了，而且扫出了3457条数据。

接着对第二个表的访问，也就是t2表，同样是全表扫描，因为他这种多表关联方式，基本上是笛卡尔积的效果，t1表的每条数据都会去t2表全表扫描所有4568条数据，跟t2表的每一条数据都会做一个关联，而且extra里说了是Nested Loop，也就是嵌套循环的访问方式，跟我们之前讲解的关联语句的执行原理都是匹配的。

另外大家会发现上面两条执行计划的id都是1，是一样的，实际上一般来说，在执行计划里，一个SELECT会对应一个id，因为这两条执行计划对应的是一个SELECT语句，所以他们俩的id都是1，是一样。

如果你要是有一个子查询，有另外一个SELECT，那么另外一个SELECT子查询对应的执行计划的id就可能是2了。

好，那么今天我们讲解了一下单表查询和多表关联的执行计划长什么样子，接下来我们会讲解子查询之类的语句的执行计划，其实讲解执行计划的本质，就是用各种不同的SQL语句来给大家讲解他们的执行计划什么样子，大家看多了自然就知道了。

End

102 透彻研究通过explain命令得到的SQL执行计划（3）

今天我们继续来讲解不同SQL语句的执行计划长什么样子，来一起看一个包含子查询的SQL语句的执行计划：

EXPLAIN SELECT * FROM t1 WHERE x1 IN (SELECT x1 FROM t2) OR x3 = 'xxxx';

这个SQL就稍微有一点点的复杂了，因为主SELECT语句的WHERE筛选条件是依赖于一个子查询的，而且除此之外还有一个自己的WHERE筛选条件，那么他的执行计划长什么样子呢？我们看看。

```
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref  | rows | filtered | Extra       |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
|  1 | PRIMARY     | t1    | NULL       | ALL   | index_x3      | NULL     | NULL    | NULL | 3457 |   100.00 | Using where |
|  2 | SUBQUERY    | t2    | NULL       | index | index_x1      | index_x1 | 507     | NULL | 4687 |   100.00 | Using index |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
```

这个执行计划值得我们好好分析一下，首先，第一条执行计划的id是1，第二条执行计划的id是2，这是为什么？因为这个SQL里有两个SELECT，主查询SELECT的执行计划的id就是1，子查询SELECT的执行计划的id就是2

其次，第一条执行计划里，select_type是PRIMARY，不是SIMPLE了，说明第一个执行计划的查询类型是主查询的意思，对主查询而言，他有一个where条件是x3='xxx'，所以他的possible_keys里包含了index_x3，就是x3字段的索引，但是他的key实际是NULL，而且type是ALL，所以说他最后没选择用x3字段的索引，而是选择了全表扫描

这是为什么呢？其实很简单，可能他通过成本分析发现，使用x3字段的索引扫描xxx这个值，几乎就跟全表扫描差不多，可能x3这个字段的值几乎都是xxx，所以最后就选择还不如直接全表扫描呢。

接着第二条执行计划，他的select_type是SUBQUERY，也就是子查询，子查询针对的是t2这个表，当然子查询本身就是一个全表查询，但是对主查询而言，会使用x1 in 这个筛选条件，他这里type是index，说明使用了扫描index_x1这个x1字段的二级索引的方式，直接扫描x1字段的二级索引，来跟子查询的结果集做比对。

接着我们来看另外一个union的SQL语句：

```
EXPLAIN SELECT * FROM t1  UNION SELECT * FROM t2
```

这是一个典型的union语句，把两个表的查询结果合并起来，如果大家不理解union的意思，建议自己去网上查一下

那么他的执行计划是什么样的呢?

```
+----+--------------+------------+------------+------+---------------+------+---------+------+------+----------+-----------------+
| id | select_type  | table      | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra           |
+----+--------------+------------+------------+------+---------------+------+---------+------+------+----------+-----------------+
|  1 | PRIMARY      | t1         | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 3457 |   100.00 | NULL            |
|  2 | UNION        | t2         | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 4687 |   100.00 | NULL            |
| NULL | UNION RESULT | <union1,2> | NULL       | ALL  | NULL          | NULL | NULL    | NULL | NULL |     NULL | Using temporary |
+----+--------------+------------+------------+------+---------------+------+---------+------+------+----------+-----------------+
```

这个执行计划的第一条和第二条很好理解对吧？两个SELECT字句对应两个id，就是分别从t1表和t2表里进行全表扫描罢了

接着第三条执行计划是什么呢？其实union字句默认的作用是把两个结果集合并起来还会进行去重，所以第三条执行计划干的是个去重的活儿。

所以上面他的table是<union 1,2>，这就是一个临时表的表名，而且你看他的extra里，有一个using temporary，也就是使用临时表的意思，他就是把结果集放到临时表里进行去重的，就这么个意思。当然，如果你用的是union all，那么就不会进行去重了。

好了，今天的讲解就先到这里，后续我们会继续用很大篇幅分析MySQL的执行计划，实际上这个执行计划可能是程序员最最需要掌握的关于数据库的知识之一了。

End

103 透彻研究通过explain命令得到的SQL执行计划（4）

之前我们已经初步的对SQL执行计划有了一个了解了，现在开始，我们就来更加细致的探索一下执行计划的方方面面，把各种SQL语句的执行计划可能长什么样，都给大家分析出来，首先我们都知道，SQL执行计划里有一个id的概念。

这个id是什么意思呢？简单来说，有一个SELECT子句就会对应一个id，如果有多个SELECT那么就会对应多个id。但是往往有时候一个SELECT字句涉及到了多个表，所以会对应多条执行计划，此时可能多条执行计划的id是一样的。

接着我们来看看这个select_type，select_type之前我们似乎看到过几种，有什么SIMPLE的，还有primary和subquery的，那么这些select_type都是什么意思？除此之外，还有哪几种select_type呢？

首先要告诉大家的是，一般如果单表查询或者是多表连接查询，其实他们的select_type都是SIMPLE，这个之前大家也都看到过了，意思就是简单的查询罢了。

然后如果是union语句的话，就类似于select * from t1 union select * from t2，那么会对应两条执行计划，第一条执行计划是针对t1表的，select_type是PRIMARY，第二条执行计划是针对t2表的，select_type是UNION，这就是在出现union语句的时候，他们就不一样了。

我们之前给大家讲过，在使用union语句的时候，会有第三条执行计划，这个第三条执行计划意思是针对两个查询的结果依托一个临时表进行去重，这个第三条执行计划的select_type就是union_result。

另外，之前我们还看到过，如果是在SQL里有子查询，类似于select * from t1 where x1 in (select x1 ffrom t2) or x3='xxx'，此时其实会有两条执行计划，第一条执行计划的select_type是PRIMARY，第二条执行计划的select_type是SUBQUERY，这个我们之前也看到过了。

那么现在我们来看一个稍微复杂一点的SQL语句：

EXPLAIN SELECT * FROM t1 WHERE x1 IN (SELECT x1 FROM t2 WHERE x1 = 'xxx' UNION SELECT x1 FROM t1 WHERE x1 = 'xxx');

这个SQL语句就稍微有点复杂了，因为他有一个外层查询，还有一个内层子查询，子查询里还有两个SELECT语句进行union操作，那么我们来看看他的执行计划会是什么样的呢？

□+----+-------------------+------------+------------+------+---------------+----------+---------+-------+------+----------+-----------------------+

| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |

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

| 1 | PRIMARY | t1 | NULL | ALL | NULL | NULL | NULL | NULL | 3467 | 100.00 | Using where |

| 2 | DEPENDENT SUBQUERY | t2 | NULL | ref | index_x1 | index_x1 | 899 | const | 59 | 100.00 | Using where; Using index |

| 3 | DEPENDENT UNION | t1 | NULL | ref | index_x1 | index_x1 | 899 | const | 45 | 100.00 | Using where; Using index |

| NULL | UNION RESULT | <union2,3> | NULL | ALL | NULL | NULL | NULL | NULL | NULL | Using temporary |

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

第一个执行计划一看就是针对t1表查询的那个外层循环，select_type就是PRIMARY，因为这里涉及到了子查询，所以外层查询的select_type一定是PRIMARY了。

然后第二个执行计划是子查询里针对t2表的那个查询语句，他的select_type是DEPENDENT SUBQUERY，第三个执行计划是子查询里针对t1表的另外一个查询语句，select_type是DEPENDENT UNION，因为第三个执行计划是在执行union后的查询，第四个执行计划的select_type是UNION RESULT，因为在执行子查询里两个结果集的合并以及去重。

现在再来看一个更加复杂一点的SQL语句：

EXPLAIN SELECT * FROM (SELECT x1, count(*) as cnt FROM t1 GROUP BY x1) AS _t1 where cnt > 10;

这个SQL可有点麻烦了，他是FROM子句后跟了一个子查询，在子查询里是根据x1字段进行分组然后进行count聚合操作，也就是统计出来x1这个字段每个值的个数，然后在外层则是针对这个内层查询的结果集进行查询通过where条件来进行过滤，看看他的执行计划：

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

| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |

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

| 1 | PRIMARY | | NULL | ALL | NULL | NULL | NULL | NULL | 3468 | 33.33 | Using where |

| 2 | DERIVED | t1 | NULL | index | index_x1 | index_x1 | 899 | NULL | 3568 | 100.00 | Using index |

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

上面的执行计划里，我们其实应该先看第二条执行计划，他说的是子查询里的那个语句的执行计划，他的select_type是derived，意思就是说，针对子查询执行后的结果集会物化为一个内部临时表，然后外层查询是针对这个临时的物化表执行的。

大家可以看到，他这里执行分组聚合的时候，是使用的index_x1这个索引来进行的，type是index，意思就是直接扫描偶了index_x1这个索引树的所有叶子节点，把x1相同值的个数都统计出来就可以了。

然后外层查询是第一个执行计划，select_type是PRIMARY，针对的table是，就是一个子查询结果集物化形成的临时表，他是直接针对这个物化临时表进行了全表扫描根据where条件进行筛选的。

好，今天的执行计划就讲解到这里了，下次我们继续讲解。

End

104 透彻研究通过explain命令得到的SQL执行计划（5）

上回我们通过一些复杂的SQL语句给大家讲解了执行计划里的select_type一般都会有哪些取值，这次我们再来看看执行计划里的type有哪些取值，其实select_type并不是很关键，因为他主要是代表了大SQL里的不同的SELECT代表了一个什么角色，比如有的SELECT是PRIMARY查询，有的是UNION，有的是SUBQUERY。

但是这个type就非常关键了，因为他直接决定了对某个表是如何从里面查询数据的，关于这个查询方式我们之前早就讲过了，包括了const、ref、range、index、all这几种方式，分别是根据主键/唯一索引查询，根据二级索引查询，对二级索引进行全索引扫描，对聚簇索引进行全表扫描。

那今天我们就重点来通过几个SQL语句来看看在什么情况下会有什么样的type取值。

首先，假设是类似于select * fromt1 where id=110这样的SQL，直接根据主键进行等值匹配查询，那执行计划里的type就会是const，意思就是极为快速，性能几乎是线性的。

事实也确实是极为快速的，因为主键值是不会重复的，这个唯一值匹配，在一个索引树里跳转查询，基本上几次磁盘IO就可以定位到了。

接着我们来看一个SQL语句：

EXPLAIN SELECT * FROM t1 INNER JOIN t2 ON t1.id = t2.id

这里是通过两个表的id进行关联查询的，此时他的执行计划如下：

```
+----+-------------+-------+------------+--------+---------------+---------+---------+-----------------+------+----------+-------+
| id | select_type | table | partitions | type   | possible_keys | key     | key_len | ref       | rows | filtered | Extra |
+----+-------------+-------+------------+--------+---------------+---------+---------+-----------------+------+----------+-------+
|  1 | SIMPLE      | t1    | NULL       | ALL    | PRIMARY       | NULL    | NULL    | NULL      | 3467 |   100.00 | NULL  |
|  1 | SIMPLE      | t2    | NULL       | eq_ref | PRIMARY       | PRIMARY | 10      | test_db.t1.id |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+--------+---------------+---------+---------+-----------------+------+----------+-------+
```

在这个执行计划里，我们会发现针对t1表是一个全表扫描，这个是必然的，因为关联的时候会先查询一个驱动表，这里就是t1，他没什么where筛选条件，自然只能是全表扫描查出来所有的数据了。

接着针对t2表的查询type是eq_ref，而且使用了PRIMARY主键。这个意思就是说，针对t1表全表扫描获取到的每条数据，都会去t2表里基于主键进行等值匹配，此时会在t2表的聚簇索引里根据主键值进行快速查找，所以在连接查询时，针对被驱动表如果基于主键进行等值匹配，那么他的查询方式就是eq_ref了。

而如果要是正常基于某个二级索引进行等值匹配的时候，type就会是ref，而如果基于二级索引查询的时候允许值为null，那么查询方式就会是ref_or_null

另外之前讲过，有一些特殊场景下针对单表查询可能会基于多个索引提取数据后进行合并，此时查询方式会是index_merge这种。

而查询方式是range的话就是基于二级索引进行范围查询，查询方式是index的时候是直接扫描二级索引的叶子节点，也就是扫描二级索引里的每条数据，最后如果是all的话就是全表扫描，也就是对聚簇索引的叶子节点扫描每条数据。

基本上执行计划里的type就这么几种取值了，其实之前都讲过，这里主要是带着大家来复习一遍。今天我们就讲到这里，明天我们接着讲解执行计划。

End

105 透彻研究通过explain命令得到的SQL执行计划（6）

今天我们继续来讲解执行计划的一些细节，之前已经详细讲过了select_type和type，今天来先讲一下possible_keys

这个possible_keys，顾名思义，其实就是在针对一个表进行查询的时候有哪些潜在可以使用的索引。

比如你有两个索引，一个是KEY(x1, x2, x3)，一个是KEY(x1, x2, x4)，此时要是在where条件里要根据x1和x2两个字段进行查询，那么此时明显是上述两个索引都可以使用的，那么到底要使用哪个呢？

此时就需要通过我们之前讲解的成本优化方法，去估算使用两个索引进行查询的成本，看使用哪个索引的成本更低，那么就选择用那个索引，最终选择的索引，就是执行计划里的key这个字段的值了。

而key_len，其实就是当你在key里选择使用某个索引之后，那个索引里的最大值的长度是多少，这个就是给你一个参考，大概知道那个索引里的值最大能有多长，就这么个意思。

而执行计划里的 ref 也相对会关键一些，当你的查询方式是索引等值匹配的时候，比如const、ref、eq_ref、ref_or_null这些方式的时候，此时执行计划的ref字段告诉你的就是：你跟索引列等值匹配的是什么？是等值匹配一个常量值？还是等值匹配另外一个字段的值？

比如SQL语句：

EXPLAIN SELECT * FROM t1 WHERE x1 = 'xxx'

此时如果你看他的执行计划是下面这样的

```
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | t1    | NULL       | ref  | index_x1      | index_x1 | 589     | const |  468 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
```

大家在上面的查询计划里可以看到，针对t1表的查询，type是ref方式的，也就是说基于普通的二级索引进行等值匹配，然后possible_keys只有一个，就是index_x1，针对x1字段建立的一个索引，而实际使用的索引也是index_x1，毕竟就他一个是可以用的。

然后key_len是589，意思就是说index_x1这个索引里的x1字段最大值的长度也就是589个字节，其实这个不算是太大，不过基本可以肯定这个x1字段是存储字符串的，因为是一个不规律的长度。

比较关键的是ref字段，它的意思是说，既然你是针对某个二级索引进行等值匹配的，那么跟index_x1索引进行等值匹配的是什么？是一个常量或者是别的字段？这里的ref的值是const，意思就是说，是使用一个常量值跟index_x1索引里的值进行等值匹配的。

假设你要是用了类似如下的语句：

EXPLAIN SELECT * FROM t1 INNER JOIN t2 ON t1.id = t2.id;

此时执行计划里的ref肯定不是const，因为你跟t1表的id字段等值匹配的是另外一个表的id字段，此时ref的值就是那个字段的名称了，执行计划如下：

```
+----+-------------+-------+------------+--------+---------------+---------+---------+-----------------+------+----------+-------+
| id | select_type | table | partitions | type   | possible_keys | key     | key_len | ref             | rows | filtered | Extra |
+----+-------------+-------+------------+--------+---------------+---------+---------+-----------------+------+----------+-------+
|  1 | SIMPLE      | t1    | NULL       | ALL    | PRIMARY       | NULL    | NULL    | NULL            | 3457 |   100.00 | NULL  |
|  1 | SIMPLE      | t2    | NULL       | eq_ref | PRIMARY       | PRIMARY | 10      | test_db.t1.id |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+--------+---------------+---------+---------+-----------------+------+----------+-------+
```

大家看执行计划，针对t1表作为驱动表执行一个全表扫描，接着针对t1表里每条数据都会去t2表根据t2表的主键执行等值匹配，所以第二个执行计划的type是eq_ref，意思就是被驱动表基于主键进行等值匹配，而且使用的索引是PRIMARY就是使用了t2表的主键。

至于ref，意思就是说，到底是谁跟t2表的聚簇索引里的主键值进行等值匹配呢？是常量值吗？

不是，是test_db这个库下的t1表的id字段，这里跟t2表的主键进行 等值匹配的是t1表的主键id字段，所以ref这里显示的清清楚楚的。

最后简单说一下rows和filtered，这个rows顾名思义，就是说你使用指定的查询方式，会查出来多少条数据，而filtered意思就是说，在查询方式查出来的这波数据里再用上其他的不在索引范围里的查询条件，又会过滤出来百分之几的数据。

比如SQL语句：

EXPLAIN SELECT * FROM t1 WHERE x1 > 'xxx' AND x2 = 'xxx'

他只有一个x1字段建了索引，x2字段是没有索引的，此时执行计划如下：

```
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------------------
----------------+
```

| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |

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

| 1 | SIMPLE | t1 | NULL | range | index_x1 | index_x1 | 458 | NULL | 1987 | 13.00 | Using index condition; Using where |

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

上面的执行计划清晰的表明了，针对t1表的查询方式是range，也就是基于索引进行范围查询，用的索引是index_x1，也就是x1字段的索引，然后基于x1>'xxx'这个条件通过index_x1索引查询出来的数据大概是1987条，接着会针对这1987条数据再基于where条件里的其他条件，也就是x2='xxx'进行过滤。

这个filtered是13.00，意思是估算基于x2='xxx'条件过滤后的数据大概是13%，也就是说最终查出来的数据大概是1987 * 13% = 258条左右。

好，今天执行计划分析就到这里，其实大家看到这里为止，基本上对于执行计划已经了解的很清楚了，接下来下次就是执行计划分析的最后一讲，也就是分析extra这个字段，这里会包含了各种查询的附加条件，也是非常重要的

看懂了extra之后，我们以后对任何SQL语句的执行计划都能逐步分析得到结论，知道这个SQL语句是如何一步一步执行的了，过程中每个步骤查询出来多少条数据。

End

106 透彻研究通过explain命令得到的SQL执行计划（7）

这周我们继续来学习SQL语句的执行计划，通过之前的学习，大家基本上应该已经对执行计划是什么意思，代表的是你SQL语句怎么执行，有一个整体的了解了

这周我们最后三讲把SQL执行计划剩余的一些内容讲完，下周我们就可以正式进入本专栏最为核心和实用的环节了，就是深度进行SQL语句调优。

这周其实我们主要就是研究一下执行计划里的**extra**这个字段里的内容都是代表什么的，其实很多人可能以为extra字段是无关紧要的，其实并不是，因为除了extra字段以外的其他内容，最多就是告诉你针对你SQL里的每个表是如何查询的，用了哪个索引，查出来了多少数据，但是很多时候，往往针对一个表可不是那么简单的。

因为除了基于索引查询数据，可能同时还得基于where条件里的其他过滤条件去筛选数据，此时还会筛选出来一些数据。

这个extra里的信息可能会非常非常的多，我们不可能给大家都讲一遍，很多其实也偶尔出现，也没多大意义，大家看到了自然也明白。我们主要是给大家讲一些平时常见的，比较有用的extra信息。

比如下面的SQL语句：

EXPLAIN SELECT x1 FROM t1 WHERE x1 = 'xxx'

可以看看他的执行计划是什么样的

```
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------------+
|  1 | SIMPLE      | t1    | NULL       | ref  | index_x1      | index_x1 | 456     | const |   25 |   100.00 | Using index |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------------+
```

这里我们可以看一下，这个执行计划现在什么意思，可以说是一清二楚。首先他是访问了t1表，使用的是ref访问方法，也就是基于二级索引去查找，找的是index_x1这个索引，这个索引的最大数据长度是456字节，查找的目标是一个const代表的常量值，通过索引可以查出来25条数据，经过其他条件筛选过后，最终剩下数据是100%。

好，那么我们看看extra的信息，是Using index，这是什么意思呢？其实就是说这次查询，仅仅涉及到了一个二级索引，不需要回表，因为他仅仅是查出来了x1这个字段，直接从index_x1索引里查就行了。

如果没有回表操作，仅仅在二级索引里执行，那么extra里会告诉in是Using index。

另外，如果有个SQL语句是：

SELECT * FROM t1 WHERE x1 > 'xxx' AND x1 LIKE '%xxx'

此时他会先在二级索引index_x1里查找，查找出来的结果还会额外的跟x1 LIKE '%xxx'条件做比对，如果满足条件的才会被筛选出来，这种情况下，extra显示的是Using index condition。

End

今天我们继续讲执行计划里的extra的信息，给大家讲一个平时最常见到的东西，就是**Using where**，这个恐怕是最最常见的了，其实这个一般是见于你直接针对一个表扫描，没用到索引，然后where里好几个条件，就会告诉你Using where，或者是你用了索引去查找，但是除了索引之外，还需要用其他的字段进行筛选，也会告诉你Using where。

比如说下面的SQL语句：

EXPLAIN SELECT * FROM t1 WHERE x2 = 'xxx'

这里的x2是没有建立索引的，所以此时他的执行计划就是下面这样的

```
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | t1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 4578 |    15.00 | Using where |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
```

大家注意看，这里说了，针对t1表进行查询，用的是全表扫描方式，没有使用任何索引，然后全表扫描，扫出来的是4578条数据，这个时候大家注意看extra里显示了Using where，意思就是说，他对每条数据都用了WHERE x2 = 'xxx'去进行筛选。

最终filtered告诉了你，过滤出来了15%的数据，大概就是说，从这个表里筛选出来了686条数据，就这个意思。

那么如果你的where条件里有一个条件是针对索引列查询的，有一个列是普通列的筛选，类似下面的SQL语句：

EXPLAIN SELECT * FROM t1 WHERE x1 = 'xxx' AND x2 = 'xxx'

此时执行计划如下

```
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------------+
```

| 1 | SIMPLE | t1 | NULL | ref | index_x1 | index_x1 | 458 | const | 250 | 18.00 | Using where |

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

这个执行计划也是非常的清晰明了，这里针对t1表去查询，先通过ref方式直接在index_x1索引里查找，是跟const代表的常量值去查找，然后查出来250条数据，接着再用Using where代表的方式，去使用AND x2 = 'xxx'条件进行筛选，筛选后的数据比例是18%，最终所以查出来的数据大概应该是45条。

另外要给大家说的是，在多表关联的时候，有的时候你的关联条件并不是索引，此时就会用一种叫做**join buffer**的内存技术来提升关联的性能，比如下面的SQL语句：

EXPLAIN SELECT * FROM t1 INNER JOIN t2 ON t1.x2 = t2.x2

他们的连接条件x2是没有索引的，此时一起看看他的执行计划

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

| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |

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

| 1 | SIMPLE | t1 | NULL | ALL | NULL | NULL | NULL | NULL | 4578 | 100.00 | NULL |

| 1 | SIMPLE | t2 | NULL | ALL | NULL | NULL | NULL | NULL | 3472 | 1.00 | Using where; Using join buffer (Block Nested Loop) |

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

这个执行计划其实也很有意思，因为要执行join，那么肯定是先得查询t1表的数据，此时是对t1表直接全表查询，查出来4578条数据，接着似乎很明确了，就是对每条数据的x2字段的值，跑到t2表里去查对应的数据，进行关联。

但是此时因为 t2 表也没法根据索引来查，也是属于全表扫描，所以每次都得对t2表全表扫描一下，根据extra提示的Using where，就是根据t1表每条数据的x2字段的值去t2表查找对应的数据了，然后此时会用join buffer技术，在内存里做一些特殊优化，减少t2表的全表扫描次数。

End

108 透彻研究通过explain命令得到的SQL执行计划（9）

今天是我们学习SQL执行计划的最后一讲，下周就要开始进入SQL调优实战案例环节了，我们会讲解大量的SQL调优实战案例，所以大家务必要把SQL执行计划都给掌握的扎实一些。

今天我们来看看执行计划里平时常见的最后两种，一个是Using filesort，一个是Using temprory。

先来看看**Using filesort**是什么意思，首先大家要知道，有的时候我们在SQL语句里进行排序的时候，如果排序字段是有索引的，那么其实是直接可以从索引里按照排序顺序去查找数据的，比如这个SQL：

EXPLAIN SELECT * FROM t1 ORDER BY x1 LIMIT 10

这就是典型的一个排序后再分页的语句，他的执行计划如下

```
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref  | rows | filtered | Extra |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------+
|  1 | SIMPLE      | t1    | NULL       | index | NULL          | index_x1 | 458     | NULL |   10 |   100.00 | NULL  |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------+
```

大家可以看到，这个SQL语句，他是用了index方式访问的，意思就是说直接扫描了二级索引，而且实际使用的索引也是index_x1，本质上来说，他就是在 index_x1索引里，按照顺序找你LIMIT 10要求的10条数据罢了。

所以大家看到返回的数据是10条，也没别的过滤条件了，所以filtered是100%，也就是10条数据都返回了。

但是如果我们排序的时候是没法用到索引的，此时就会基于内存或者磁盘文件来排序，大部分时候得都基于磁盘文件来排序，比如说这个SQL：

EXPLAIN SELECT * FROM t1 ORDER BY x2 LIMIT 10

x2字段是没有索引的，此时执行计划如下

```
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra          |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------+
```

| 1 | SIMPLE | t1 | NULL | ALL | NULL | NULL | NULL | NULL | 4578 | 100.00 | Using filesort |

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

这个SQL很明确了，他基于x2字段来排序，是没法直接根据有序的索引去找数据的，只能把所有数据写入一个临时的磁盘文件，基于排序算法在磁盘文件里按照x2字段的值完成排序，然后再按照LIMIT 10的要求取出来头10条数据。

所以大家以后要注意一下，这种把表全数据放磁盘文件排序的做法真的是相当的糟糕，性能其实会极差的。

最后给大家讲一下，如果我们用group by、union、distinct之类的语法的时候，万一你要是没法直接利用索引来进行分组聚合，那么他会直接基于临时表来完成，也会有大量的磁盘操作，性能其实也是极低的。

比如这个SQL：

EXPLAIN SELECT x2, COUNT(*) AS amount FROM t1 GROUP BY x2

这里的x2是没有索引的，所以此时的执行计划如下

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

| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |

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

| 1 | SIMPLE | t1 | NULL | ALL | NULL | NULL | NULL | NULL | 5788 | 100.00 | Using temporary |

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

这个SQL里只能对全表数据放到临时表里做大量的磁盘文件操作，然后才能完成对x2字段的不同的值去分组，分组完了以后对不同x2值的分组去做聚合操作，这个过程也是相当的耗时的，性能是极低的。

所以大家最后记住，其实未来在SQL调优的时候，核心就是分析执行计划里哪些地方出现了全表扫描，或者扫描数据过大，尽可能通过合理优化索引保证执行计划每个步骤都可以基于索引执行，避免扫描过多的数据。

End

109 案例实战：千万级用户场景下的运营系统SQL调优（1）

今天开始我们正式进入MySQL的SQL性能优化的案例实战部分，我们一共将会讲解4个SQL优化案例，每个案例都会放在一周内通过三次文章来讲解，每个案例都会分为业务场景引入、SQL性能问题分析、SQL性能调优三个部分。

今天我们就开始讲解咱们的第一个案例，也就是**千万级用户场景下的运营系统的复杂SQL调优实战案例**

先说下这个案例的背景，简单来说，这是一个互联网公司的系统，这个互联网公司的用户量是比较大的，有百万级日活用户的一个量级。

在这个互联网公司里，有一个系统是专门通过各种条件筛选出大量的用户，接着对那些用户去推送一些消息的，有的时候可能是一些促销活动的消息，有的时候可能是让你办会员卡的消息，有的时候可能是告诉你有一个特价商品的消息。

总而言之，其实通过一些条件筛选出大量的用户，接着针对这些用户做一些推送，是互联网公司的运营系统里常见的一种功能，在这个过程中，比较坑爹，也比较耗时的，其实是筛选用户的这个过程。

因为这种互联网公司，我们已经说过了，用户是日活百万级的，注册用户是千万级的，而且如果还没有进行分库分表的话，那么这个数据库里的用户表可能就一张，单表里是上千万的用户数据，大概是这么一个情况。

现在我们来对运营系统筛选用户的SQL做一个简化，写出来给大家看个热闹，这个SQL经过简化看起来可能是这样的：

SELECT id, name FROM users WHERE id IN (SELECT user_id FROM users_extent_info WHERE latest_login_time < xxxxx)

上面的SQL语句是啥意思？给大家解释一下，它的意思就是说一般存储用户数据的表会分为两张表，一个表用来存储用户的核心数据，比如id、name、昵称、手机号之类的信息，也就是上面SQL语句里的users表

另外一个表可能会存储用户的一些拓展信息，比如说家庭住址、兴趣爱好、最近一次登录时间之类的，就是上面的users_extent_info表

所以上面的SQL语句的意思就很明显了，有一个子查询，里面针对用户的拓展信息表，也就是users_extent_info查询了一下最近一次登录时间小于某个时间点的用户，这里其实可以是查询最近才登陆过的用户，也可以查询的是很长时间没登录过的用户 ，然后给他们发送一些push，无论哪种场景，这个SQL都是适用的。

然后在外层的查询里，直接就是用了id IN字句去查询 id 在子查询结果范围里的users表的所有数据，此时这个SQL往往一下子会查出来很多数据，可能几千、几万、几十万，都有可能，所以其实一般运行这类SQL之前，都会先跑一个count聚合函数，看看有多少条，比如下面这儿样。

SELECT COUNT(id) FROM users WHERE id IN (SELECT user_id FROM users_extent_info WHERE latest_login_time < xxxxx)

然后内存里做一个小批量多批次读取数据的操作，比如判断如果在1000条以内，那么就一下子读取出来，如果超过1000条，可以通过LIMIT语句，每次就从这个结果集里查1000条数据，查1000条就做一次批量PUSH，再查下一波1000条。

这就是这个案例的一个完整的业务背景和讲解，那么当时产生的问题是什么呢?

很简单，就是在千万级数据量的大表场景下，上面的SQL直接轻松跑出来耗时几十秒的速度，所以说，这个SQL不优化是绝对不行了!

下次我们就来针对这个SQL，分析一下他的执行计划以及他的性能之所以差的问题所在。

End

110 案例实战：千万级用户场景下的运营系统SQL调优（2）

今天咱们继续来看这个千万级用户场景下的运营系统SQL调优案例，上次已经给大家说了一下业务背景以及SQL，这个SQL就是如下的一个：

SELECT COUNT(id) FROM users WHERE id IN (SELECT user_id FROM users_extent_info WHERE latest_login_time < xxxxx)

之前说了，系统运行的时候，肯定会先跑一下COUNT聚合函数来查查这个结果集有多少数据，然后再分批查询。结果就是这个COUNT聚合函数的SQL，在千万级大表的场景下，都要花几十秒才能跑出来，简直是大跌眼镜，这种性能，系统基本就没法跑了！

所以我们今天一起来分析一下这个SQL的执行计划，不过这里要给大家提醒一点的是，因为不同的MySQL版本的执行计划可能都不一样，平时我们开发可能感觉不出来，但是实际上每个不同的MySQL版本都可能会调整生成执行计划的方式，所以同样的SQL在不同的MySQL版本下跑 ，可能执行计划都不太一样。

我们这里给出的执行计划是当时在我们的MySQL中得到的，可能大家自己拿同样的SQL去自己的MySQL里没法还原出来这里的执行计划，但是没关系，大家重点学的是执行计划分析的思路，以及如何从执行计划里看出性能问题所在，最后就是如何进行调优，重点是这个过程，没法还原出来执行计划，也是没关系的。

通过：

EXPLAIN SELECT COUNT(id) FROM users WHERE id IN (SELECT user_id FROM users_extent_info WHERE latest_login_time < xxxxx)

可以得到下面的执行计划，我们为了方便大家看，把执行计划简化了几个字段，保留了最关键的几个字段。

另外，给大家提示一点，下面的执行计划是当时我们为了调优，在测试环境的单表2万条数据场景下跑出来的执行计划，即使是5万条数据，当时这个SQL都跑了十多秒，所以足够复现当时的生产问题了，所以大家注意下执行计划里的数据量问题。

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

| id | select_type | table | type | key | rows | filtered | Extra |

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

| 1 | SIMPLE | | ALL | NULL | NULL | 100.00 | NULL |

| 1 | SIMPLE | users | ALL | NULL | 49651 | 10.00 | Using where; Using join buffer(Block Nested Loop) |

| 2 | MATERIALIZED | users_extent_info | range | idx_login_time | 4561 | 100.00 | NULL |

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

从上面的执行计划，我们可以清晰的看到这条SQL语句的一个执行过程

首先，针对子查询，是执行计划里的第三行实现的，他清晰的表明，针对users_extent_info，使用了idx_login_time这个索引，做了range类型的索引范围扫描，查出来了4561条数据，没有做其他的额外筛选，所以filtered是100%。

接着他这里的MATERIALIZED，表明了这里把子查询的4561条数据代表的结果集进行了物化，物化成了一个临时表，这个临时表物化，一定是会把4561条数据临时落到磁盘文件里去的，这个过程其实就挺慢的。

然后第二条执行计划表明，接着就是针对users表做了一个全表扫描，在全表扫描的时候扫出来了49651条数据，同时大家注意看Extra字段，显示了一个Using join buffer的信息，这个明确表示，此处居然在执行join操作？？？

接着看执行计划里的第一条，这里他是针对子查询产出的一个物化临时表，也就是，做了一个全表查询，把里面的数据都扫描了一遍，那么为什么要对这个临时表进行全表扫描呢?

原因就是在让users表的每一条数据，都要去跟物化临时表里的数据进行join，所以针对users表里的每一条数据，只能是去全表扫描一遍物化临时表，找找物化临时表里哪条数据是跟他匹配的，才能筛选出来一条结果。

第二条执行计划的全表扫描的结果表明是一共扫到了49651条数据，但是全表扫描的过程中，因为去跟物化临时表执行了一个join操作，而物化临时表里就4561条数据，所以最终第二条执行计划的filtered显示的是10%，也就是说，最终从users表里筛选出了也是4000多条数据。

这就是这条SQL语句的执行计划，不同MySQL版本可能是不一样的，甚至差别很大，所以大家没必要强求必须是要在自己本地可以还原出这个执行计划，但是大家重点是看明白我们这里对这个SQL语句的执行计划过程的一个分析。

End

111 案例实战：千万级用户场景下的运营系统SQL调优（3）

今天是我们第一个千万级用户场景下的运营系统SQL调优案例的最后一讲，也是最关键的一讲，我们要根据SQL语句的执行计划找出他速度慢的原因所在，然后还得想办法去优化他的速度。

上一次我们已经对SQL语句的执行计划做了一个分析，知道了那个SQL语句的执行过程，今天就得对SQL执行计划做一个透彻的分析，看看到底为什么他会慢

先来回看一下那个执行计划的内容：

```
+----+-------------+-------+------------+-------+---------------+----------+---------+------+
| id | select_type | table | type  | key | rows | filtered | Extra |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+
| 1 | SIMPLE |  | ALL | NULL | NULL | 100.00 | NULL  |
| 1 | SIMPLE | users | ALL | NULL | 49651 | 10.00 | Using where; Using join buffer(Block Nested Loop)  |
| 2 | MATERIALIZED | users_extent_info | range | idx_login_time | 4561 | 100.00 | NULL  |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+
```

之前说过，他执行的过程就是先执行了子查询查出来4561条数据，物化成了一个临时表，接着他对users主表做了一个全表扫描，扫描的过程中把每一条数据都放到物化临时表里去做全表扫描，本质在做join的事情。

那么这里为什么会跑的这么慢呢？其实很明显了，大家可以想一下，首先他对子查询的结果做了一次物化临时表，落地磁盘了，接着他还全表扫描了users表的所有数据，每一条数据居然跑到一个没有索引的物化临时表里再做一次全表扫描找匹配数据。

在这个过程里，对users表的全表扫描耗时不耗时？对users表的每一条数据跑到物化临时表里做全表扫描，耗时不耗时？所以这个过程必然是非常慢的，几乎就没怎么用到索引。

那么接着我们就很奇怪了，为什么会出现上述的一个全表扫描users表，然后跟物化临时表做join，join的时候还要全表扫描物化临时表的过程？

这里交大家一个技巧，就是在执行完上述SQL的EXPLAIN命令，看到执行计划之后，可以执行一下**show warnings**命令。

这个show warnings命令此时显示出来的内容如下：

/* select#1 */ select count(`d2`.`users`.`user_id`) AS `COUNT(users.user_id)`

from `d2`.`users` `users` semi join xxxxxx，下面省略一大段内容，因为可读性实在不高，大家关注的应该是这里的**semi join**这个关键字

这里就显而易见了！MySQL在这里，生成执行计划的时候，自动就把一个普通的IN子句，“优化”成了基于semi join来进行IN+子查询的操作，这个semi join是什么意思呢？

简单来说，对users表不是全表扫描了么？对users表里每一条数据，去对物化临时表全表扫描做semi join，不需要把users表里的数据真的跟物化临时表里的数据join上。只要users表里的一条数据，在物化临时表里可以找到匹配的数据，那么users表里的数据就会返回，这就叫做semi join，他是用来筛选的。

所以慢，也就慢在这里了，那既然知道了是semi join和物化临时表导致的问题，应该如何优化呢？

先别急，做个小实验，执行SET optimizer_switch='semijoin=off'，也就是关闭掉半连接优化，此时执行EXPLAIN命令看一下此时的执行计划，发现此时会恢复为一个正常的状态。

就是有一个SUBQUERY的子查询，基于range方式去扫描索引搜索出4561条数据，接着有一个PRIMARY类型的主查询，直接是基于id这个PRIMARY主键聚簇索引去执行的搜索，然后再把这个SQL语句真实跑一下看看，发现性能一下子提升了几十倍，变成了100多毫秒！

因此到此为止，这个SQL的性能问题，真相大白，其实反而是他自动执行的semi join半连接优化，给咱们导致了问题，一旦禁止掉semi join自动优化，用正常的方式让他基于索引去执行，性能那是嗖嗖的。

当然，在生产环境是不能随意更改这些设置的，所以后来我们想了一个办法，多种办法尝试去修改SQL语句的写法，在不影响他语义的情况下，尽可能的去改变SQL语句的结构和格式，最终被我们尝试出了一个写法，如下所示：

SELECT COUNT(id)

FROM users

WHERE (id IN (SELECT user_id FROM users_extent_info WHERE latest_login_time < xxxxx) OR id IN (SELECT user_id FROM users_extent_info WHERE latest_login_time < -1))

在上述写法下，WHERE语句的OR后面的第二个条件，根本是不可能成立的，因为没有数据的latest_login_time是小于-1的，所以那是不会影响SQL语义的，但是我们发现改变了SQL的写法之后，执行计划也随之改变。

他并没有再进行semi join优化了，而是正常的用了子查询，主查询也是基于索引去执行的，这样我们在线上上线了这个SQL语句，性能从几十秒一下子就变成几百毫秒了。

希望大家能认真体会这个SQL调优案例里的方法，其实最核心的，还是看懂SQL的执行计划，然后去分析到底他为什么会那么慢，接着你就是要想办法避免他全表扫描之类的操作，一定要让他去用索引，用索引是王道，是最重要的！

End

112 案例实战：亿级数据量商品系统的SQL调优实战（1）

今天开始进入我们的SQL语句调优的第二个案例实战，这个案例讲的是一个电商平台的亿级数据量的商品系统的SQL语句的性能调优。对于这个案例，我们同样会拆分为三讲来讲解。

上次的案例大家会看到，主要问题在于MySQL内部自动使用了半连接优化，结果半连接的时候导致大量无索引的全表扫描，引发了性能的急剧下降；

而这次的这个案例，其实也是类似的，是我们的MySQL数据库在选择索引的时候，选择了一个不太合适的索引，导致了性能极差，引发了慢查询。

先从当时线上的商品系统出现的一个慢查询告警开始讲起，某一天晚上，我们突然收到了线上数据库的频繁报警，这个报警的意思大致就是说，数据库突然涌现出了大量的慢查询，而且因为大量的慢查询，导致每一个数据库连接执行一个慢查询都要耗费很久。

那这样的话，必然会导致突然过来的很多查询需要让数据库开辟出来更多的连接，因此这个时候报警也告诉我们，数据库的连接突然也暴增了，而且每个连接都打满，每个连接都要执行一个慢查询，慢查询还跑的特别慢。

接着引发的问题，就是数据库的连接全部打满，没法开辟新的连接了，但是还持续的有新的查询发送过来，导致数据库没法处理新的查询，很多查询发到数据库直接就阻塞然后超时了，这也直接导致线上的商品系统频繁的报警，出现了大量的数据库查询超时报错的异常！

当时看到这一幕报警，让人是非常揪心的，因为这种情况，基本意味着你的商品数据库以及商品系统濒临于崩溃了，大量慢查询耗尽了数据库的连接资源，而且一直阻塞在数据库里执行，数据库没法执行新的查询，商品数据库没法执行查询，用户没法使用商品系统，也就没法查询和筛选电商网站里的商品了！

而且大家要知道，当时正好是晚上晚高峰的时候！也就是一个电商网站比较繁忙的时候，虽说商品数据是有多级缓存架构的，但是实际上在下单等过程中，还是会大量的请求商品系统的，所以晚高峰的时候，商品系统本身TPS大致是在每秒几千的。

因此这个时候，发现数据库的监控里显示，每分钟的慢查询超过了10w+！！！也就是说商品系统大量的查询都变成了慢查询！！！

那么慢查询的都是一些什么语句呢？其实主要就是下面这条语句，大家可以看一下，我们做了一个简化：

```
select * from products where category='xx' and sub_category='xx' order by id desc limit xx,xx
```

这其实是一个很稀松平常的SQL语句，他就是用户在电商网站上根据商品的品类以及子类在进行筛选，当然真实的SQL语句里，可能还包含其他的一些字段的筛选，比如什么品牌以及销售属性之类的，我们这里是做了一个简化，然后按id倒序排序，最后是分页，就这么一个语句。

这个语句执行的商品表里大致是1亿左右的数据量，这个量级已经稳定了很长时间了，主要也就是这么多商品，但是上面的那个语句居然一执行就是几十秒！

几十秒，这还得了？基本上数据库的连接全部被慢查询打满，一个连接要执行几十秒的SQL，然后才能执行下一个SQL，此时数据库基本就废了，没法执行什么查询了！！！

所以难怪商品系统本身也大量的报警说查询数据库超时异常了！

End

113 案例实战：亿级数据量商品系统的SQL调优实战（2）

今天我们继续来分析这个案例，上次已经讲到，下面的这个商品系统让用户根据品类筛选商品的SQL语句

select * from products where category='xx' and sub_category='xx' order by id desc limit xx,xx

在一个亿级数据量的商品表里执行，需要耗时几十秒，结果导致了数据库的连接资源全部打满，商品系统无法运行，处于崩溃状态。

现在就得来分析一下，到底为什么会出现这样的一个情况，首先要给大家解释一下，这个表当时肯定是对经常用到的查询字段都建立好了索引的，那么针对这里简化后的SQL语句，你可以认为如下的一个索引，KEY index_category(catetory,sub_category)肯定是存在的，所以基本可以确认上面的SQL绝对是可以用上索引的。

因为如果你一旦用上了品类的那个索引，那么按品类和子类去在索引里筛选，其实第一，筛选很快速，第二，筛出来的数据是不多的，按说这个语句应该执行的速度是很快的，即使表有亿级数据，但是执行时间也最多不应该超过1s。

但是现在这个SQL语句跑了几十秒，那说明他肯定就没用我们建立的那个索引，所以才会这么慢，那么他到底是怎么执行的呢？我们来看一下他的执行计划：

explain select * from products where category='xx' and sub_category='xx' order by id desc limit xx,xx

此时执行计划具体内容就不写了，因为大家之前看了那么多执行计划，基本都很熟悉了，我就说这里最核心的信息，他的possible_keys里是有我们的index_category的，结果实际用的key不是这个索引，而是PRIMARY！！而且Extra里清晰写了Using where

到此为止，这个SQL语句为什么性能这么差，就真相大白了，他其实本质上就是在主键的聚簇索引上进行扫描，一边扫描，一边还用了where条件里的两个字段去进行筛选，所以这么扫描的话，那必然就是会耗费几十秒了！

因此此时为了快速解决这个问题，就需要强制性的改变MySQL自动选择这个不合适的聚簇索引进行扫描的行为

那么怎么改变呢？交给大家一个办法，就是使用force index语法，如下：

select * from products force index(index_category) where category='xx' and sub_category='xx' order by id desc limit xx,xx

使用上述语法过后，强制让SQL语句使用了你指定的索引，此时再次执行这个SQL语句，会发现他仅仅耗费100多毫秒而已！性能瞬间就提升上来了！

因此当时在紧急关头中，一下子就把这个问题给解决了，这里也是告诉大家这样的一个实战技巧，就是你如何去强制改变MySQL的执行计划，之前就有一个朋友来问我们说，面试官问我，如果MySQL使用了错误的执行计划，应该怎么办？

其实答案很简单，就是这个案例里的情况，方法就是force index语法就可以了。

但是这个案例还没完，这里还遗留了很多的问题，比如：

- 为什么在这个案例中MySQL默认会选择对主键的聚簇索引进行扫描？
- 为什么没使用index_category这个二级索引进行扫描？
- 即使用了聚簇索引，为什么这个SQL以前没有问题，现在突然就有问题了？

这都是一系列奇怪的问题，让我们对这个案例进行了深入的探究，下次，我们就来给大家分析这个案例背后的这些故事。

End

114 案例实战：亿级数据量商品系统的SQL调优实战（3）

今天我们来分析一下这个案例背后的一些事情，上次我们提到了一系列的问题，包括：

- 为什么在这个案例中MySQL默认会选择对主键的聚簇索引进行扫描？
- 为什么没使用index_category这个二级索引进行扫描？
- 即使用了聚簇索引，为什么这个SQL以前没有问题，现在突然就有问题了？

关于这些问题，咱们得一步一步的来解决。首先，第一个问题，为什么针对：

select * from products where category='xx' and sub_category='xx' order by id desc limit xx,xx

这样一个SQL语句，MySQL要选择对聚簇索引进行扫描呢？

其实关于这个逻辑，说起来也并不是太复杂，因为大家都知道，这个表是一个亿级数据量的大表，那么对于他来说，index_category这个二级索引也是比较大的

所以此时对于MySQL来说，他有这么一个判断，他觉得如果要是从index_category二级索引里来查找到符合where条件的一波数据，接着还得回表，回到聚簇索引里去。

因为SQL语句是要select *的，所以这里必然涉及到一次回表操作，回到聚簇索引里去把所有字段的数据都查出来，但是在回表之前，他必然要做完order by id desc limit xx,xx这个操作

举个例子吧，比如他根据where category='xx' and sub_category='xx'，从index_category二级索引里查找出了一大波数据。

比如从二级索引里假设搂出来了几万条数据，接着因为二级索引里是包含主键id值的，所以此时他就得按照order by id desc这个排序语法，对这几万条数据基于临时磁盘文件进行filesort磁盘排序，排序完了之后，再按照limit xx,xx语法，把指定位置的几条数据拿出来，假设就是limit 0,10，那么就是把10条数据拿出来。

拿出来10条数据之后，再回到聚簇索引里去根据id查找，把这10条数据的完整字段都查出来，这就是MySQL认为如果你使用index_category的话，可能会发生的一个情况。

所以他担心的是，你根据where category='xx' and sub_category='xx'，从index_category二级索引里查出来的数据太多了，还得在临时磁盘里排序，可能性能会很差，因此MySQL就把这种方式判定为一种不太好的方式。

因此他才会选择换一种方式，也就是说，直接扫描主键的聚簇索引，因为聚簇索引都是按照id值有序的，所以扫描的时候，直接按order by id desc这个倒序顺序扫描过去就可以了，然后因为他知道你是limit 0,10的，也就知道你仅仅只要拿到10条数据就行了。

所以他在按顺序扫描聚簇索引的时候，就会对每一条数据都采用Using where的方式，跟where category='xx' and sub_category='xx'条件进行比对，符合条件的就直接放入结果集里去，最多就是放10条数据进去就可以返回了。

此时MySQL认为，按顺序扫描聚簇索引，拿到10条符合where条件的数据，应该速度是很快的，很可能比使用index_category二级索引那个方案更快，因此此时他就采用了扫描聚簇索引的这种方式！

那接下来我们又要考虑一个问题了，那就是这个SQL语句，实际上之前在线上系统运行一直没什么问题，也就是说，之前在线上系统而言，即使采用扫描聚簇索引的方案，其实这个SQL语句也确实一般都运行不慢，最起码是不会超过1s的。

那么为什么会在某一天晚上突然的就大量报慢查询，耗时几十秒了呢？

原因也很简单，其实就是因为之前的时候，where category='xx' and sub_category='xx'这个条件通常都是有返回值的，就是说根据条件里的取值，扫描聚簇索引的时候，通常都是很快就能找到符合条件的值以及返回的，所以之前其实性能也没什么问题。

但是后来可能是商品系统里的运营人员，在商品管理的时候加了几种商品分类和子类，但是这几种分类和子类的组合其实没有对应的商品

也就是说，那一天晚上，很多用户使用这种分类和子类去筛选商品，where category='新分类' and sub_category='新子类'这个条件实际上是查不到任何数据的！

所以说，底层在扫描聚簇索引的时候，扫来扫去都扫不到符合where条件的结果，一下子就把聚簇索引全部扫了一遍，等于是上亿数据全表扫描了一遍，都没找到符合where category='新分类' and sub_category='新子类'这个条件的数据。

也正是因为如此，才导致这个SQL语句频繁的出现几十秒的慢查询，进而导致MySQL连接资源打满，商品系统崩溃！

因此到此为止，这个案例就彻底分析清楚了，包括案例背后的故事也给大家讲明白了，其实SQL调优并没那么难，核心在于你一定要看懂SQL的执行计划，理解他为什么会慢，只要你理解了，就是想各种办法去解决，这个解决办法可能不是专栏可以讲完的，我们会提供几种经典的方案，但是往往需要你在发现问题的时候自己想办法，或者网上搜索。

比如我们上次讲到的第一个案例，就是通过禁用MySQL的半连接优化或者是改写SQL语句结构来避免自动半连接优化，第二个案例就得通过force index语法来强制某个SQL用我们指定的索引，这些都是属于比较经典的解决方案。

End

115 案例实战：数十亿数量级评论系统的SQL调优实战（1）

今天来给大家讲一个新的SQL调优案例，就是针对我们电商场景下非常普遍的商品评论系统的一个SQL优化，这个商品评论系统的数据量非常大，拥有多达十亿量级的评论数据，所以当时对这个评论数据库，我们是做了分库分表的，基本上分完库和表过后，单表的评论数据在百万级别。

每一个商品的所有评论都是放在一个库的一张表里的，这样可以确保你作为用户在分页查询一个商品的评论时，一般都是直接从一个库的一张表里执行分页查询语句就可以了

好，那么既然提到了商品评论分页查询的问题，我们就可以从这里开始讲我们的案例了。

大家都知道，在电商网站里，有一些热门的商品，可能销量多达上百万，商品的评论可能多达几十万条。然后呢，有一些用户，可能就喜欢看商品评论，他就喜欢不停的对某个热门商品的评论不断的进行分页，一页一页翻，有时候还会用上分页跳转功能，就是直接输入自己要跳到第几页去。

所以这个时候，就会涉及到一个问题，**针对一个商品几十万评论的深分页问题。**

先来看看一个经过我们简化后的对评论表进行分页查询的SQL语句：

SELECT * FROM comments WHERE product_id ='xx' and is_good_comment='1' ORDER BY id desc LIMIT 100000,20

这个SQL语句想必大家都知道是怎么回事。

其实他的意思就是，比如用户选择了查看某个商品的评论，因此必须限定Product_id，同时还选了只看好评，所以is_good_commit也要限定一下

接着他要看第5001页评论，那么此时limit的offset就会是(5001 - 1) * 20，其中20就是每一页的数量，此时起始offset就是100000，所以limit后100000,20

对这个评论表呢，最核心的索引就是一个，那就是index_product_id，所以对上述SQL语句，正常情况下，肯定是会走这个索引的，也就是说，会通过index_product_id索引，根据product_id ='xx'这个条件从表里先删选出来这个表里指定商品的评论数据。

那么接下来第二步呢？当然是得按照 is_good_comment='1' 条件，筛选出这个商品评论数据里的所有好评了！但是问题来了，这个index_product_id的索引数据里，并没有is_good_commet字段的值，所以此时只能很尴尬的进行回表了。

也就是说，对这个商品的每一条评论，都要进行一次回表操作，回到聚簇索引里，根据id找到那条数据，取出来is_good_comment字段的值，接着对is_good_comment='1'条件做一个比对，筛选符合条件的数据。

那么假设这个商品的评论有几十万条，岂不是要做几十万次回表操作？虽然每次回表都是根据id在聚簇索引里快速查找的，但还是架不住你每条数据都回表啊！！！

接着对于筛选完毕的所有符合WHERE product_id ='xx' and is_good_comment='1'条件的数据，假设有十多万条吧，接着就是按照id做一个倒序排序，此时还得基于临时磁盘文件进行倒序排序，又得耗时很久。

排序完毕了，就得基于limit 100000,20获取第5001页的20条数据，最后返回。

这个过程，因为有几十万次回表查询，还有十多万条数据的磁盘文件排序，所以当时发现，这条SQL语句基本要跑个1秒~2秒。

那么如何对他进行优化呢？其实这个思路，反而就跟我们讲的第二个案例反过来了，第二个案例中基于商品品类去查商品表，是尽量避免对聚簇索引进行扫描，因为有可能找不到你指定的品类下的商品，出现聚簇索引全表扫描的问题。

所以当时第二个案例里，反而就是选择强制使用一个联合索引，快速定位到数据，这个过程中因为不需要进行回表，所以效率还是比较高的

大家如果有印象的话，应该还记得第二个案例里，就是根据category和sub_category组成的联合索引进行查找，所以不需要回表，这就节省下了大量回表操作的耗时，所以当时我们选择了这个方案。

然后第二个案例中，接着直接根据id临时磁盘文件排序后找到20条分页数据，再回表查询20次，找到20条商品的完整数据。因此当时对第二个案例而言，因为不涉及到大量回表的问题，所以这么做基本是合适的，性能通常在1s以内。

但是我们这个案例里，就不是这么回事了，因为WHERE product_id ='xx' and is_good_comment='1'这两个条件，不是一个联合索引，所以必须会出现大量的回表操作，这个耗时是极高的。

因此对于这个案例，我们通常会采取如下方式改造分页查询语句：SELECT * from comments a, (SELECT id FROM comments WHERE product_id ='xx' and is_good_comment='1' ORDER BY id desc LIMIT 100000,20) b WHERE a.id=b.id

上面那个SQL语句的执行计划就会彻底改变他的执行方式，他通常会先执行括号里的子查询，子查询反而会使用PRIMARY聚簇索引，按照聚簇索引的id值的倒序方向进行扫描，扫描过程中就把符合WHERE product_id ='xx' and is_good_comment='1'条件的数据给筛选出来。

比如这里就筛选出了十万多条的数据，并不需要把符合条件的数据都找到，因为limit后跟的是100000,20，理论上，只要有100000+20条符合条件的数据，而且是按照id有序的，此时就可以执行根据limit 100000,20提取到5001页的这20条数据了。

接着你会看到执行计划里会针对这个子查询的结果集，一个临时表，进行全表扫描，拿到20条数据，接着对20条数据遍历，每一条数据都按照id去聚簇索引里查找一下完整数据，就可以了。

所以针对我们的这个场景，反而是优化成这种方式来执行分页，他会更加合适一些，他只有一个扫描聚簇索引筛选符合你分页所有数据的成本，你的分页深度越深，扫描数据越多，分页深度越浅，那扫描数据就越少，然后再做一页20条数据的20次回表查询就可以了。

当时我们做了这个分页优化之后，发现这个分页语句一下子执行时间降低到了几百毫秒了，此时就达到了我们优化的目的。

但是这里还是要给大家提醒一点，大家会发现，SQL调优实际上是没有银弹的，比如对于第二个案例来说，按顺序扫描聚簇索引方案可能会因为找不到数据导致亿级数据量的全表扫描，所以对第二个案例而言，必须得根据二级索引去查找。

但是对于我们这第三个案例而言，因为前提是做了分库分表，评论表单表数据一般在一百万左右，所以首先，他即使一个商品没有评论，有全表扫描，也绝对不会像扫描上亿数据表那么慢

其次，如果你根据product_id的二级索引查找，反而可能出现几十万次回表查询，所以二级索引查找方式反而不适合，而按照聚簇索引顺序扫描的方式更加适合。

简而言之，针对不同的案例，要具体情况具体分析，他慢，慢的原因在哪儿，为什么慢，然后再用针对性的方式去优化他。

End

116 案例实战：千万级数据删除导致的慢查询优化实践（1）

今天给大家讲解一个新的案例，这个案例是我们之前线上系统遇到过的一个慢查询调优实战案例，案例的背景是，当时有人删除了千万级的数据，结果导致了频繁的慢查询，接下来给大家讲一下这个案例整个排查、定位以及解决的一个过程。

这个案例的开始，当时是从线上收到大量的慢查询告警开始的，当我们收到大量的慢查询告警之后，就去检查慢查询的SQL，结果发现不是什么特别的SQL，这些SQL语句主要都是针对一个表的，同时也比较简单，而且基本都是单行查询，看起来似乎不应该会慢查询。

所以这个时候我们是感觉极为奇怪的，因为SQL本身完全不应该有慢查询，按说那种SQL语句，基本上都是直接根据索引查找出来的，性能应该是极高的。

那么有没有另外一种可能，慢查询不是SQL的问题，而是MySQL生产服务器的问题呢?

这里给大家解释一下，实际上个别特殊情况下，MySQL出现慢查询并不是SQL语句的问题，而是他自己生产服务器的负载太高了，导致SQL语句执行很慢。

给大家举个例子，比如现在MySQL服务器的磁盘IO负载特别高，也就是每秒执行大量的高负载的随机IO，但是磁盘本身每秒能执行的随机IO是有限的。

结果呢，就导致你正常的SQL语句去磁盘上执行的时候，如果要跑一些随机IO，你的磁盘太繁忙了，顾不上你了，导致你本来很快的一个SQL，要等很久才能执行完毕，这个时候就可能导致正常SQL语句也会变成慢查询!

所以同理，除了磁盘之外，还有一个例子就是网络，也许网络负载很高，就可能会导致你一个SQL语句要发送到MySQL上去，光是等待获取一个跟MySQL的连接，都很难，要等很久，或者MySQL自己网络负载太高了，带宽打满，带宽打满了之后，你一个SQL也许执行很快，但是他查出来的数据返回给你，网络都送不出去，此时也会变成慢查询。

另外一个关键的点就是CPU负载，如果说CPU负载过高的话，也会导致CPU过于繁忙去执行别的任务了，没时间执行你这个SQL语句，此时也有可能会导致你的SQL语句出现问题的，所以这个大家也得注意。

所以说慢查询本身不一定是SQL导致的，如果你觉得SQL不应该慢查询，结果他那个时间段跑这个SQL就是慢，**此时你应该排查一下当时MySQL服务器的负载，尤其看看磁盘、网络以及CPU的负载，是否正常**

如果你发现那个时间段MySQL生产服务器的磁盘、网络或者CPU负载特别高，那么可能是服务器负载导致的问题

举个例子，我们之前解决过一个典型的问题，就是当某个离线作业瞬间大批量把数据往MySQL里灌入的时候，他一瞬间服务器磁盘、网络以及CPU的负载会超高。

此时你一个正常SQL执行下去，短时间内一定会慢查询的，针对类似的问题，优化手段更多的是控制你导致MySQL负载过高的那些行为，比如灌入大量数据，最好在凌晨低峰期灌入，别影响线上系统运行。

结果奇怪的是，当时我们看了下MySQL服务器的磁盘、网络以及CPU负载，一切正常，似乎也不是这个问题导致的。

这个时候，似乎看起来有点无解了是不是？别着急，这个案例的排查过程是极为漫长的，涉及到MySQL大量的调优知识，最终解决这个问题，甚至要深入我们之前讲过的MySQL内核级原理，才能分析清楚以及解决问题。

今天我们先站在当时的角度，给大家分析我们的头两步排查手段，一个是检查SQL是否有问题，主要就是看他的执行计划，这个我们之前都讲过了，另外一个是检查MySQL服务器的负载，今天我们也说明了背后的一些知识

那么在这两种办法都不奏效之后，下一次我们就要给大家讲当时我们排查问题的第三步，就是用MySQL profilling工具去细致的分析SQL语句的执行过程和耗时。

End

117 案例实战：千万级数据删除导致的慢查询优化实践（2）

案例实战：千万级数据删除导致的慢查询优化实践（2）

好，今天我们继续讲解这个案例，在当时这个案例的场景发生之后，也就是针对某个表的大量简单的单行数据查询SQL变成慢查询，我们先排查了SQL执行计划以及MySQL服务器负载，发现都没有问题。

此时就必须用上一个SQL调优的利器了，也就是**profiling**工具，这个工具可以对SQL语句的执行耗时进行非常深入和细致的分析，使用这个工具的过程，大致如下所示

首先要打开这个profiling，使用set profiling=1这个命令，接着MySQL就会自动记录查询语句的profiling信息了。

此时如果执行show profiles命令，就会给你列出各种查询语句的profiling信息，这里很关键的一点，就是他会记录下来每个查询语句的query id，所以你要针对你需要分析的query找到对他的query id，我们当时就是针对慢查询的那个SQL语句找到了query id。

然后就可以针对单个查询语句，看一下他的profiling具体信息，使用show profile cpu, block io for query xx，这里的xx是数字，此时就可以看到具体的profile信息了

除了cpu以及block io以外，你还可以指定去看这个SQL语句执行时候的其他各项负载和耗时，具体使用方法，大家自行网上搜索就行了，并不难。

他这里会给你展示出来SQL语句执行时候的各种耗时，比如磁盘IO的耗时，CPU等待耗时，发送数据耗时，拷贝数据到临时表的耗时，等等吧，反正SQL执行过程中的各种耗时都会展示出来的。

这里我们当时仔细检查了一下这个SQL语句的profiling信息，重点发现了一个问题，他的Sending Data的耗时是最高的，几乎使用了1s的时间，占据了SQL执行耗时的99%，这就很坑爹了。

因为其他环节耗时低是可以理解的，毕竟这种简单SQL执行速度真的很快，基本就是10ms级别的，结果跑成了1s，那肯定Sending Data就是罪魁祸首了！

这个Sending Data是在干什么呢?

MySQL的官方释义如下：为一个SELECT语句读取和处理数据行，同时发送数据给客户端的过程，简单来说就是为你的SELECT语句把数据读出来，同时发送给客户端。

可是为什么这个过程会这么慢呢？profiling确实是提供给我们更多的线索了，但是似乎还是没法解决掉问题。但是毕竟我们已经捕获到了第一个比较异常的点了，就是Sending Data的耗时很高！请大家记住这个线索。

有时候针对MySQL这种复杂数据库软件的调优过程，就跟福尔摩斯破案一样，你要通过各种手段和工具去检查MySQL的各种状态，然后把有异常的一些指标记下来，作为一个线索，当你线索足够多的时候，往往就能够汇总大量的线索整理出一个思路了，那也就是一个破案的时刻了！

接着我们又用了一个命令：**show engine innodb status**，看一下innodb存储引擎的一些状态，此时发现了一个奇怪的指标，就是history list length这个指标，他的值特别高，达到了上万这个级别。

这里我们给大家解释一下这个指标，当然如果大家自己在调优的时候发现了类似的情况，不知道一个指标什么意思，直接google一下就可以了，很快就会查到，这里我们直接给大家一个结论了。

大家应该还记得之前我们讲解过的MVCC机制吧？MVCC机制，说穿了就是多个事务在对同一个数据，有人写，有人读，此时可以有多种隔离级别，这个大家应该还记得吧。

至于这个MVCC和隔离级别的实现原理，跟一个Read View机制是有关系的，同时还有一个至关重要的机制，就是数据的undo多版本快照链条。

你必须对一个数据得有一个多版本快照链条，才能实现MVCC和各种隔离级别，这个具体的原理，我们这里不多说了，大家有遗忘的，建议回看之前的文章。

所以当你有大量事务执行的时候，就会构建这种undo多版本快照链条，此时history list length的值就会很高。然后在事务提交之后，会有一个多版本快照链条的自动purge清理机制，只要有清理，那么这个值就会降低。

一般来说，这个值是不应该过于高的，所以我们在这里注意到了第二个线索，history list length值过高！大量的undo多版本链条数据没被清理！推测很可能就是有的事务长时间运行，所以他的多版本快照不能被purge清理，进而导致了这个history list length的值过高！

第二个线索Get！基本可以肯定的一点是，经过两个线索的推测，在大量简单SQL语句变成慢查询的时候，SQL是因为Sending Data环节异常耗时过高，同时此时出现了一些长事务长时间运行，大量的频繁更新数据，导致有大量的undo多版本快照链条，还无法purge清理。

但是这两个线索之间的关系是什么呢？是第二个线索推导出的事务长时间运行现象的发生，进而导致了第一个线索发现的Sending Data耗时过高的问题吗？可是二者之间的关系是什么呢？是不是还得找到更多的线索还行呢？

大家别着急，到此为止，大家就跟看侦探小说一样，福尔摩斯已经找到了一些线索，但是似乎还缺少一些关键线索，把所有线索都串起来，进而去让他形成一个完善的破案推理，真相即将大白了，咱们下次继续讲。

End

118 我们为什么要搭建一套MySQL的主从复制架构？（1）

我们为什么要搭建一套MySQL的主从复制架构？（1）

之前的很长时间里，我们经过了大量内容的讲解，想必大家对于MySQL的内核级工作原理已经有了一个了解了，包括我们的数据是如何写入MySQL服务器的内存以及磁盘的，过程中的事务、锁分别是怎么实现的，多事务并发的时候，隔离机制是如何运作的，MVCC的原理是什么，想必大家都有一个较为透彻的理解了。

同时大家现在对MySQL的索引数据结构以及工作原理，包括SQL查询语句的执行原理以及执行计划的分析，以及SQL语句调优的一些技巧和方法，应该也都有了一个较为透彻的理解了

因此简单来说，现在各位同学假设面对一个单机版的MySQL数据库，对于你的数据是如何执行增删改操作写入数据库的，以及你的索引是如何设计的，如何组织的，你的查询是如何执行的，你的查询应该如何优化，都有了一个较为系统全面的理解，而且这个理解是基于MySQL的内核级的原理的，有一定的深度，是不是?

好，那么如果大家感觉自己对上述提问都有一个肯定的回答，说明大家之前的内容肯定都好好学，而且认真复习过，学习的都较为透彻，其实掌握上述内容，就意味着大家对MySQL的原理以及使用掌握的就比较好了。

那么从今天开始，我们将要进入一个全新的阶段，那就是在MySQL真正的生产环境中，他一定不是一个单机版的架构，因为单机版的MySQL一般仅能用于本地开发环境和测试环境，是绝对不可能运用于生产环境的。

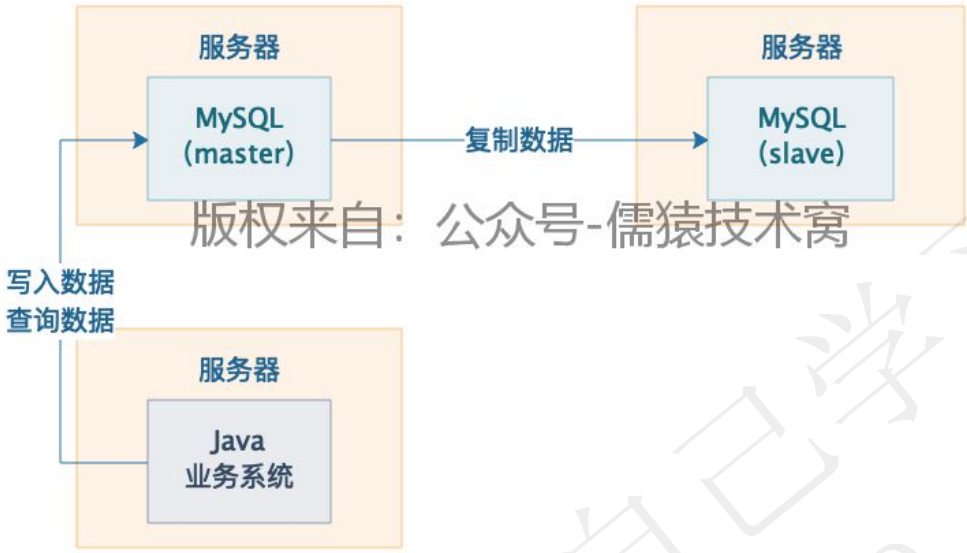
那么生产环境的MySQL架构应该是什么样子的呢？简单来说，MySQL在生产环境中，必须要搭建一套主从复制的架构，同时可以基于一些工具实现高可用架构

另外如果有需求，还需要基于一些中间件实现读写分离架构，最后就是如果数据量很大，还必须可以实现分库分表的架构。

所以当大家把MySQL单机版的内容学完之后，再把后续的这些架构学完，才能说作为一个合格以及优秀的Java工程师/后端工程师，对生产环境下的MySQL架构有了一个全面的理解，能够在自己的生产项目中运用上MySQL的生产级的架构。

那么今天我们就先来给大家讲讲MySQL的主从复制架构，这个主从复制架构，顾名思义，就是部署两台服务器，每台服务器上都得有一个MySQL，其中一个MySQL是master（主节点），另外一个MySQL是slave（从节点）。

然后我们的系统平时连接到master节点写入数据，当然也可以从里面查询了，就跟你用一个单机版的MySQL是一样的，但是master节点会把写入的数据自动复制到slave节点去，让slave节点可以跟master节点有一模一样的数据，如下图所示。

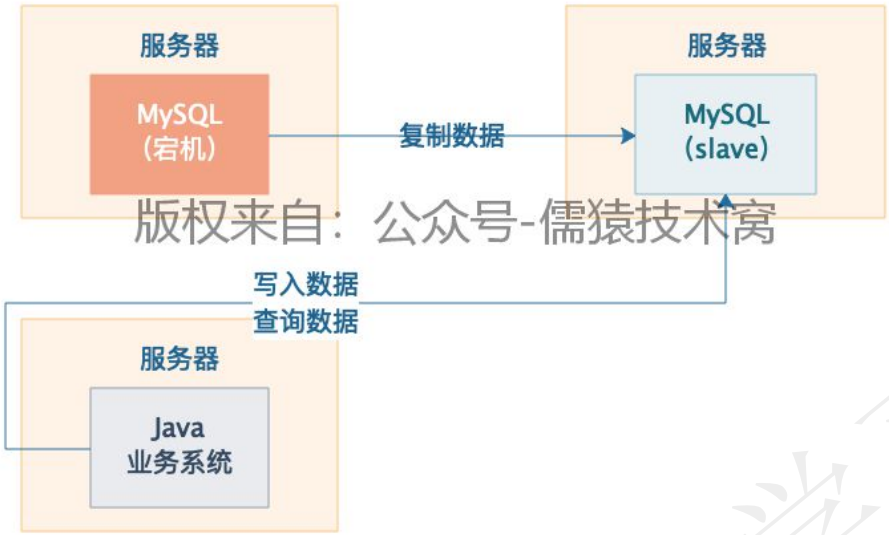


上图其实就是一个典型的MySQL的主从复制架构，那么做这个主从复制架构，意义在哪儿呢？

其实这个架构用处是极为多的，我们来给大家一 一举例，首当其冲的一个需求就是高可用架构。

大家可以想想，如果你的MySQL就单机部署，那么一旦他宕机了，岂不是你的数据库就完蛋了？数据库完蛋了，你的Java业务系统是不是也就完蛋了？所以说，真正生产架构里，MySQL必须得做高可用架构。

那么高可用架构怎么做呢？他的一个先决条件就是**主从复制架构。**你必须得让主节点可以复制数据到从节点，保证主从数据是一致的，接着万一你的主节点宕机了，此时可以让你的Java业务系统连接到从节点上去执行SQL语句，写入数据和查询数据，因为主从数据是一致的，所以这是没问题的，如下图所示。



如果实现这样的一个效果，自然就实现了MySQL的高可用了，他单机宕机不影响你的Java业务系统的运行。但是大家也得注意，这里其实是没这么简单的，因为实际哪怕这套架构运用到生产环境，也是有大量的问题要解决的。

比如主从进行数据复制的时候，其实从节点通常都会落后一些，所以数据不完全一致。另外，主节点宕机后，要能自动切换从节点对外提供服务，这个也需要一些中间件的支持，也没那么容易，这些问题，后续我们都会讲到的。

那么搭建了主从复制架构之后，还有其他什么用处呢？下回我们再继续讲解。

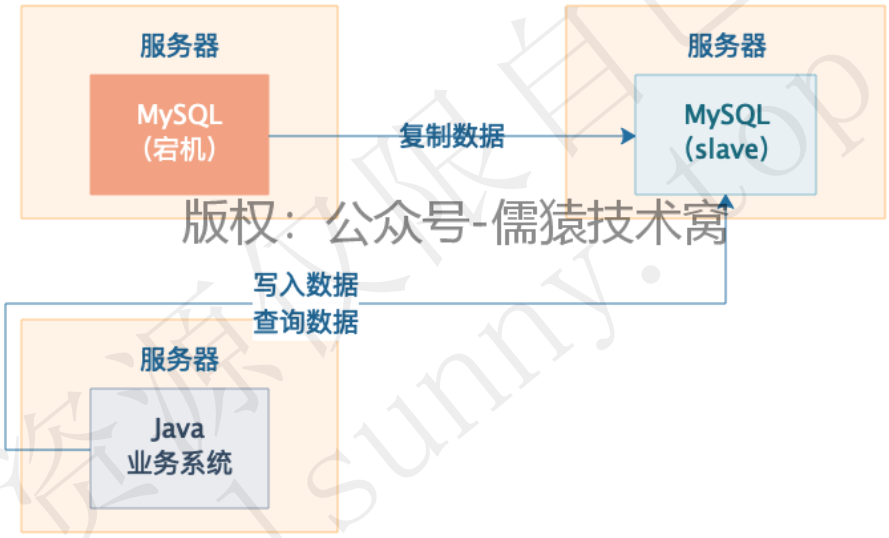
End

119 我们为什么要搭建一套MySQL的主从复制架构？（2）

我们为什么要搭建一套MySQL的主从复制架构？（2）

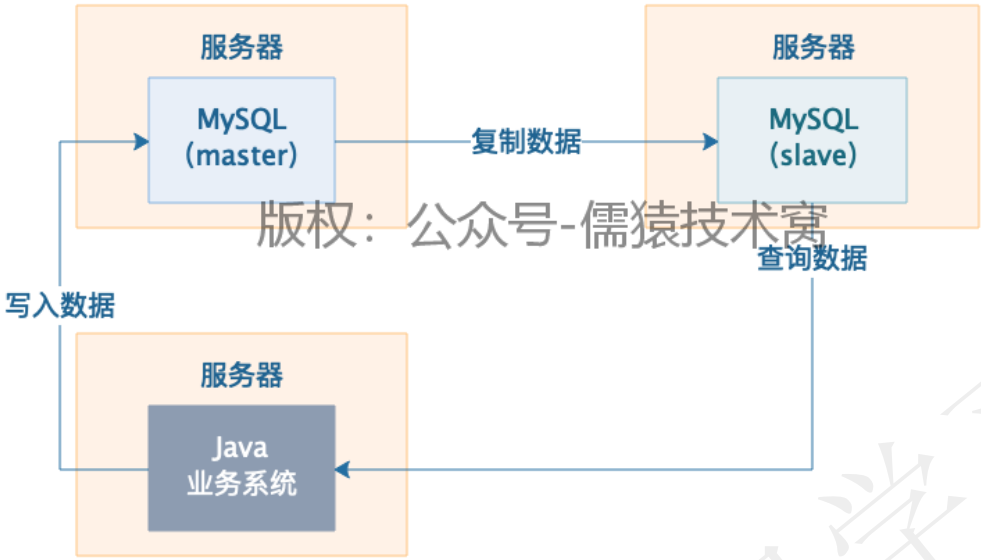
上一次已经讲到，我们搭建一套MySQL主从复制架构之后，可以实现一个高可用的效果，也就就是说主节点宕机，可以切换去读写从节点，因为主从节点数据基本是一致的

当然，暂时也就只能说是基本一致的，因为后续大家学习了他主从复制的原理之后就知道为什么说是基本了，如下图。



那么我们如果做了这个MySQL主从复制架构之后，除了这个高可用之外，还有什么作用呢？其实这就得说到大名鼎鼎的**读写分离架构**了！这个读写分离架构，也是依赖于MySQL的主从复制架构的。

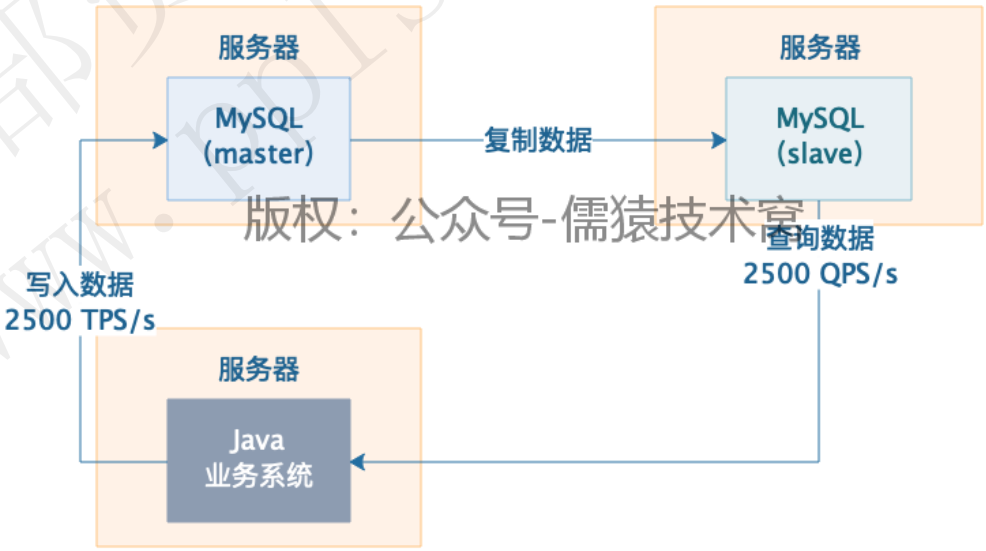
读写分离架构的意思就是，你的Java业务系统可以往主节点写入数据，但是从从节点去查询数据，把读写操作做一个分离，分离到两台MySQL服务器上去，一台服务器专门让你写入数据，然后复制数据到从节点，另外一台服务器专门让你查询数据，如下图所示。



可是好端端的，我们吃饱了没事儿，为什么要做读写分离呢？难道就为了好玩儿吗？

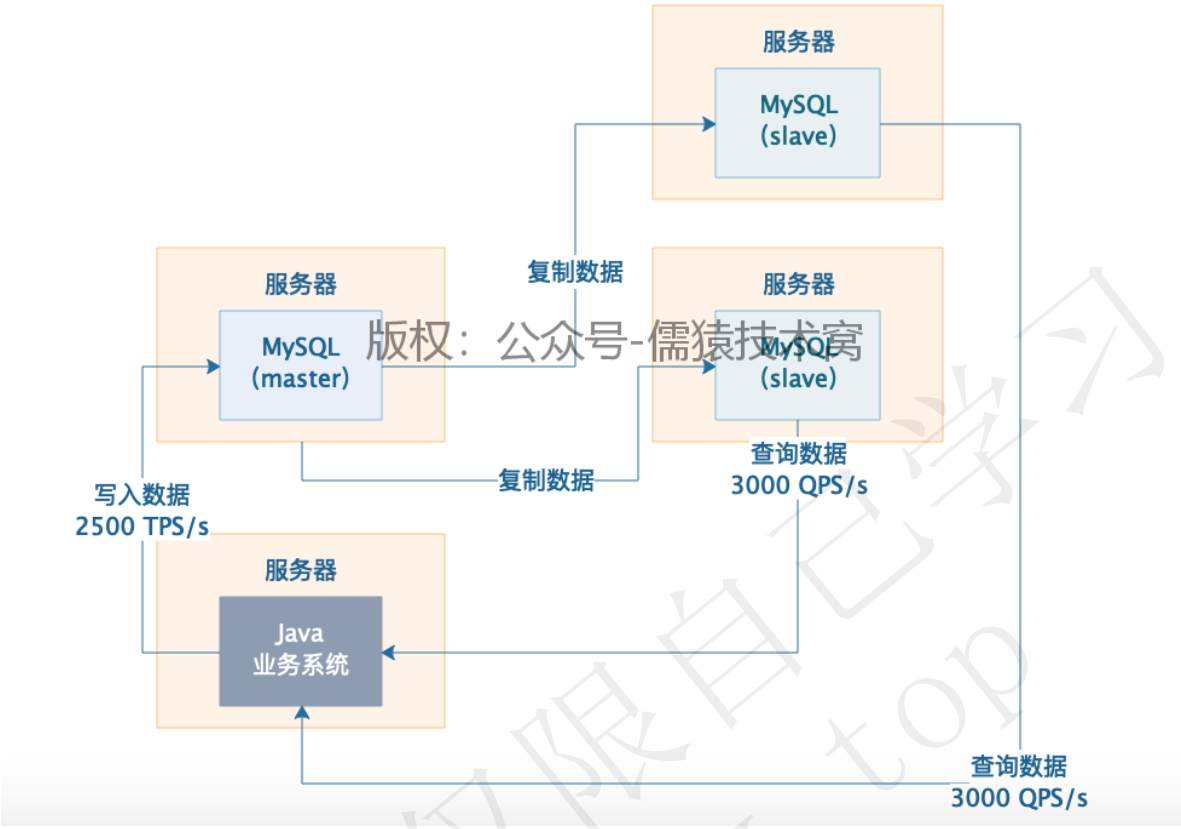
当然不是了！因为假设我们的MySQL单机服务器配置是8核16GB，然后每秒最多能抗4000读写请求，现在假设你真实的业务负载已经达到了，每秒有2500写请求+2500读请求，也就是每秒5000读写请求了，那么你觉得如果都放一台MySQL服务器，能抗的住吗？

必然不行啊！所以此时如果你可以利用主从复制架构，搭建起来读写分离架构，就可以让每秒2500写请求落到主节点那台服务器，2500读请求落到从节点那台服务器，用2台服务器来抗下你每秒5000的读写请求，如下图所示。



接着现在问题来了，大家都知道，其他大部分Java业务系统都是读多写少，读请求远远多于写请求，那么接着发现随着系统日益发展，读请求越来越多，每秒可能有6000读请求了，此时一个从节点服务器也抗不下来啊，那怎么办呢？

简单！因为MySQL的主从复制架构，是支持一主多从的，所以此时你可以再在一台服务器上部署一个从节点，去从主节点复制数据过来，此时你就有2个从节点了，然后你每秒6000读请求不就可以落到2个从节点上去了，每台服务器主要接受每秒3000的读请求，如下图。



如上图，Java业务系统每秒以2500的TPS写入主库，然后主库会复制数据到两个从库，接着你每秒6000QPS的读请求分散在两个从库上，一切似乎很完美，这就是主从复制架构的另外一个经典的应用场景，就是读写分离，通过读写分离，可以让你抗下很高的读请求。

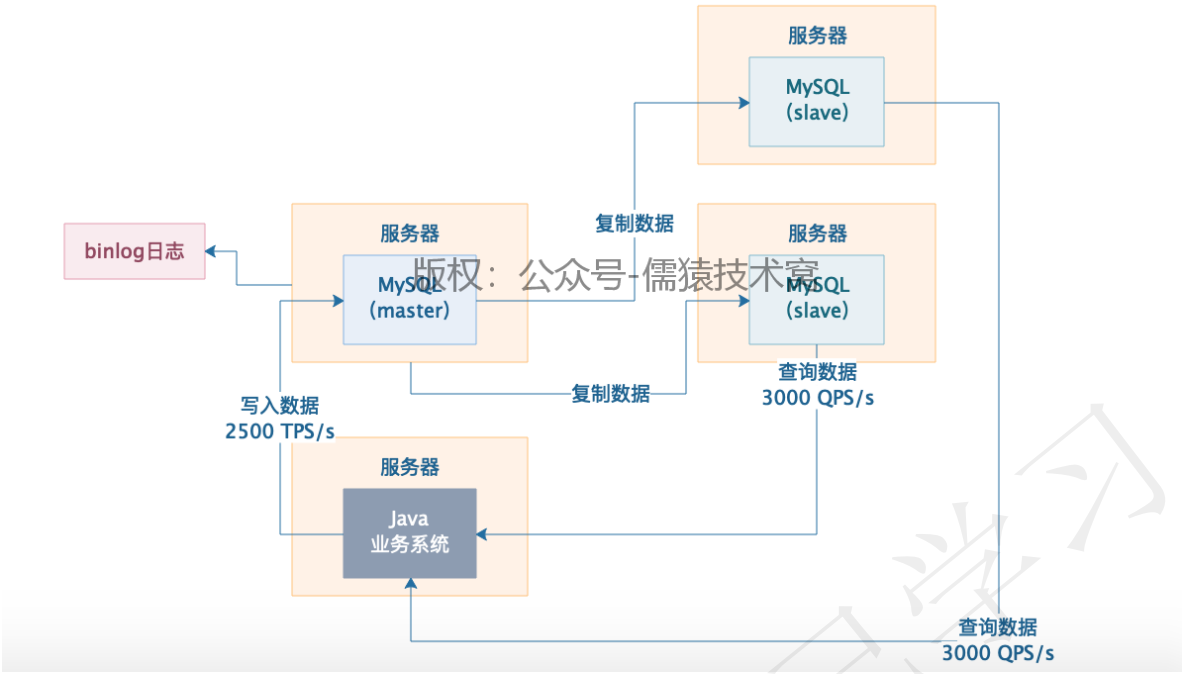
而且在上述架构之下，还可以融合高可用架构进去，因为你有多个从库，所以当你主库宕机的时候，可以通过中间件把一个从库切换为主库，此时你的Java业务系统可以继续运行，在实现读写分离的场景下，还可以同时实现高可用。

不过其实一般在项目中，高可用架构是必须做的，但是读写分离架构并不是必须的，因为对于大多数公司来说，读请求QPS并没那么高，远远达不到每秒几千那么夸张，但是高可用你是必须得做的，因为你必须保证主库宕机后，有另外一个从库可以接管提供服务，避免Java业务系统中断运行。

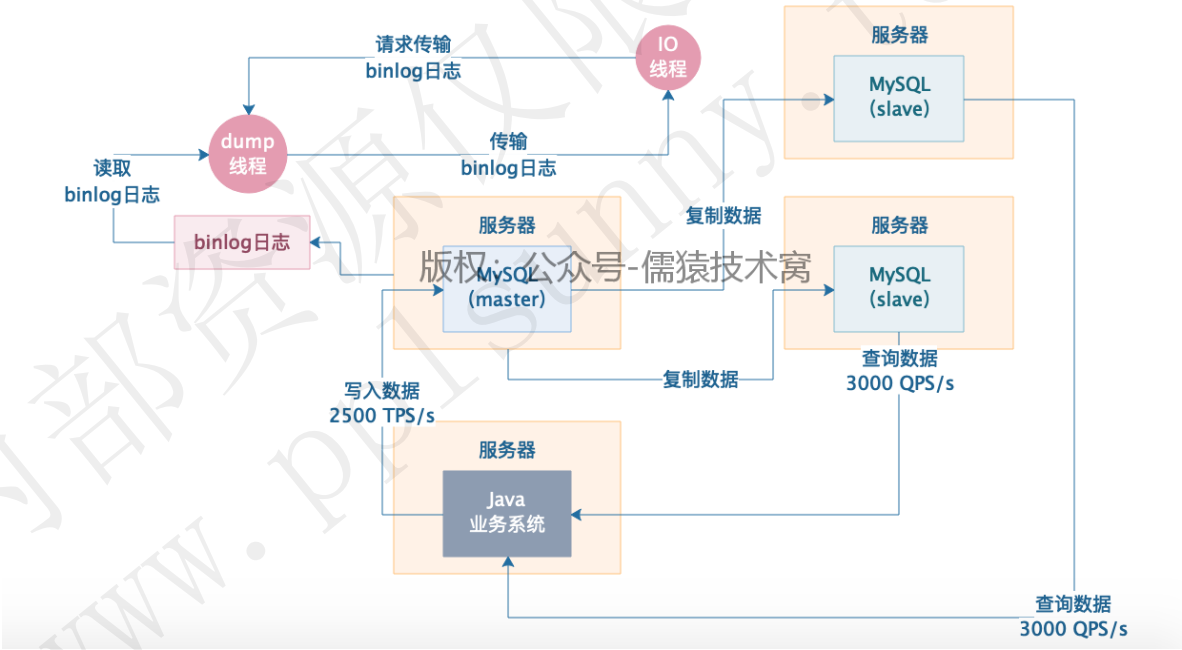
除此之外，这个从库其实还有很多其他的应用场景，比如你可以挂一个从库，专门用来跑一些报表SQL语句，那种SQL语句往往是上百行之多，运行要好几秒，所以可以专门给他一个从库来跑。也可以是专门部署一个从库，让你去进行数据同步之类的操作。

接着我们来说一下MySQL实现主从复制的一个基本的工作原理。

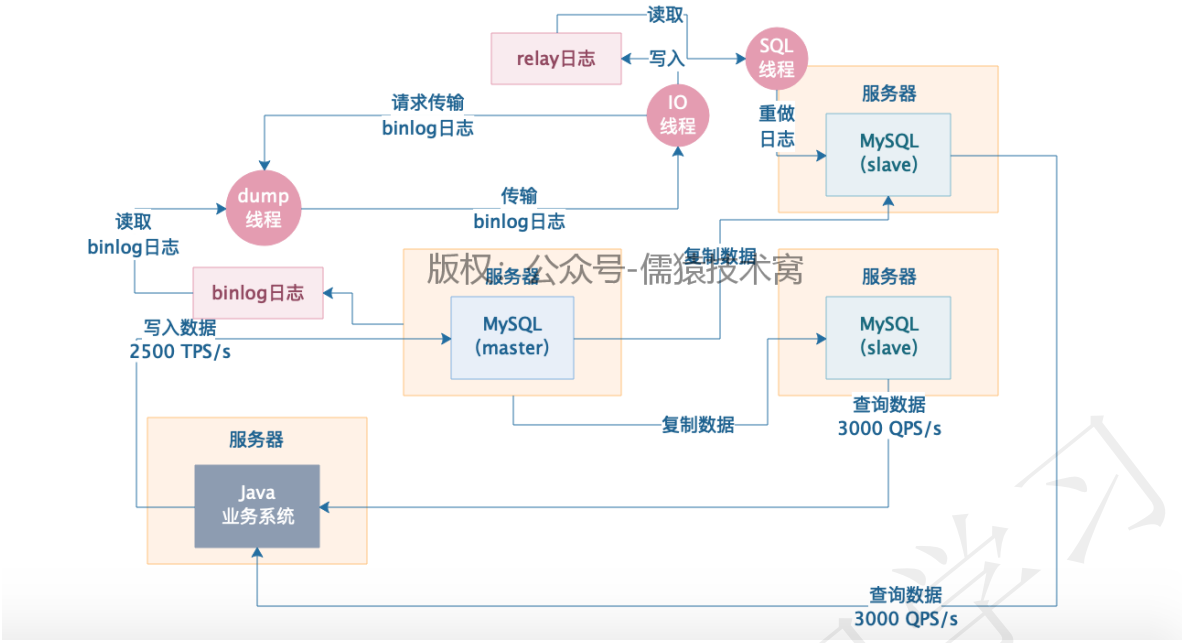
首先呢，大家通过之前的学习，应该都知道，MySQL自己在执行增删改的时候会记录binlog日志，这个大家没问题吧？忘记的同学可以回过头去看看，所以这个binlog日志里就记录了所有数据增删改的操作，如下图。



然后从库上有一个IO线程，这个IO线程会负责跟主库建立一个TCP连接，接着请求主库传输binlog日志给自己，这个时候主库上有一个IO dump线程，就会负责通过这个TCP连接把binlog日志传输给从库的IO线程，如下图所示。



接着从库的IO线程会把读取到的binlog日志数据写入到自己本地的relay日志文件中去，然后从库上另外有一个SQL线程会读取relay日志里的内容，进行日志重做，把所有在主库执行过的增删改操作，在从库上做一遍，达到一个还原数据的过程，如下图。



到此为止，想必大家对MySQL主从复制的原理也就有一个基本的了解了，简单来说，你只要给主节点挂上一个从节点，从节点的IO线程就会跟主节点建立网络连接，然后请求主节点传输binlog日志，主节点的IO dump线程就负责传输binlog日志给从节点，从节点收到日志后就可以回放增删改操作恢复数据。

在这个基础之上，就可以实现MySQL主从节点的数据复制以及基本一致，进而可以实现高可用架构以及读写分离架构

好了，今天就先讲到这里，下次我们会继续讲解MySQL的各种主从复制模式的搭建方式。

End

120 案例实战：千万级数据删除导致的慢查询优化实践（3）

案例实战：千万级数据删除导致的慢查询优化实践（3）

接着119讲的内容往下，我们就得开始要讲解如何为MySQL搭建一套主从复制架构了，会涉及到一些数据库配置的实操步骤

不过在这个之前，其实我们得先解决之前的一个案例的遗留问题，之前在117讲里，我们当时对一个SQL性能优化案例的讲解才到第二讲，最后还留了一个尾巴没解决，就是当时那个慢查询到底是什么原因导致的。

其实说穿了也并不难，这里我们就提一下吧，大家就可以跟之前的117讲衔接上去了。

简单来说，当时经过排查，一直排查到117讲末尾的时候，发现有大量的更新语句在活跃，而且有那种长期活跃的超长事务一直在跑没有结束，结果一问系统负责人，发现他在后台跑了一个定时任务，定时清理数据，结果清理的时候一下子清理了上千万的数据。

这个清理是怎么做的呢？他居然开了一个事务，然后在一个事务里删除上千万数据，导致这个事务一直在运行，所以才看到117讲末尾发现的一些现象。

然后呢，这种长事务的运行会导致一个问题，那就是你删除的时候仅仅只是对数据加了一个删除标记，事实上并没有彻底删除掉。此时你如果跟长事务同时运行的其他事务里在查询，他在查询的时候是可能会把那上千万被标记为删除的数据都扫描一遍的。

因为每次扫描到一批数据，都发现标记为删除了，接着就会再继续往下扫描，所以才导致一些查询语句会那么的慢。

那么可能有人会问了，为什么你启动一个事务，在事务里查询，凭什么就要去扫描之前那个长事务标记为删除状态的上千万的垃圾数据呢？按说那些数据都被删除了，跟你没关系了，你可以不用去扫描他们啊！

这个问题的关键点就在于，那个删除千万级数据的事务是个长事务！

也就是说，当你启动新事务查询的时候，那个删除千万级数据的长事务一直在运行，是活跃的！所以大家还记得我们之前讲解MVCC的时候，提到的一个Read View的概念么？MVCC是如何实现的？不就是基于一个Read View机制来实现的么？

当你启动一个新事务查询的时候，会生成一个Read View，里面包含了当前活跃事务的最大id、最小id和事务id集合，然后他有一个判定规则，具体判定规则大家不记得可以回顾一下当时我们讲过的内容。

总之就是，你的新事务查询的时候，会根据ReadView去判断哪些数据是你可见的，以及你可见的数据版本是哪个版本，因为一个数据有一个版本链条，有的时候你可能可见的仅仅是这个数据的一个历史版本而已。

所以正是因为这个长事务一直在运行还在删除大量的数据，而且这些数据仅仅是标记为删除，实际还没删除，所以此时你新开事务的查询是会读到所有被标记为删除的数据的，就会出现千万级的数据扫描，才会造成慢查询！

针对这个问题，其实大家要知道的一点是，永远不要在业务高峰期去运行那种删除大量数据的语句，因为这可能导致一些正常的SQL都变慢查询，因为那些SQL也许会不断扫描你标记为删除的大量数据，好不容易扫描到一批数据，结果发现是标记为删除的，于是继续扫描下去，导致了慢查询！

所以当时的解决方案也很简单，直接kill那个正在删除千万级数据的长事务，所有SQL很快会恢复正常，从此以后，对于大量数据清理全部放在凌晨去执行，那个时候就没什么人使用系统了，所以查询也很少。

End

121 如何为MySQL搭建一套主从复制架构？（1）

如何为MySQL搭建一套主从复制架构？（1）

今天我们来讲解一下如何为MySQL搭建一套主从复制架构，其实这个MySQL主从复制的原理之前也都讲过了，大致来说，就是主库接受增删改操作，把增删改操作binlog写入本地文件，然后从库发送请求来拉取binlog，接着在从库上重新执行一遍binlog的操作，就可以还原出一样的数据了。

那么搭建的时候肯定是需要两台机器的，一台机器放主库，一台机器放从库，至于主库和从库如何安装和启动？这个就不在我们讲的范围了，随便网上一搜就大把的MySQL安装步骤，我们这里就讲解搭建主从复制架构要做的一些配置。

首先呢，要确保主库和从库的server-id是不同的，这个是必然的，其次就是主库必须打开binlog功能，你必须打开binlog功能主库才会写binlog到本地磁盘，接着就可以按如下步骤在主库上执行一通操作了。

首先在主库上要创建一个用于主从复制的账号：

```
create user 'backup_user'@'192.168.31.%' identified by 'backup_123';

grant replication slave on . to 'backup_user'@'192.168.31.%';

flush privileges;
```

接着你要考虑一个问题，假设你主库都跑了一段时间了，现在要挂一个从库，那从库总不能把你主库从0开始的所有binlog都拉一遍吧！这是不对的，此时你就应该在凌晨的时候，在公司里直接让系统对外不可用，说是维护状态，然后对主库和从库做一个数据备份和导入。

可以使用如下的mysqldump工具把主库在这个时刻的数据做一个全量备份，但是此时一定是不能允许系统操作主库了，主库的数据此时是不能有变动的。

```
/usr/local/mysql/bin/mysqldump --single-transaction -uroot -proot --master-data=2 -A > backup.sql
```

注意，mysqldump工具就在你的MySQL安装目录的bin目录下，然后用上述命令就可以对你主库所有的数据都做一个备份，备份会以SQL语句的方式进入指定的backup.sql文件，只要执行这个backup.sql文件，就可以恢复出来跟主库一样的数据。

至于上面命令里的--master-data=2，意思就是说备份SQL文件里，要记录一下此时主库的binlog文件和position号，这是为主从复制做准备的。

接着你可以通过scp之类的命令把这个backup.sql文件拷贝到你的从库服务器上去就行了，这个scp命令怎么用就不用我们来说了，大家随便网上查一下就知道这个命令是怎么用的了，这个是很简单的。

接着操作步骤转移到从库上去执行，在从库上执行如下命令，把backup.sql文件里的语句都执行一遍，这就相当于把主库所有的数据都还原到从库上去了，主库上的所有database、table以及数据，在从库里全部都有了。

接着在从库上执行下面的命令去指定从主库进行复制。

CHANGE MASTER TO MASTER_HOST='192.168.31.229',
MASTER_USER='backup_user',MASTER_PASSWORD='backup_123',MASTER_LOG_FILE='mysql-bin.000015',MASTER_LOG_POS=1689;

可能有人会疑惑，上面的master机器的ip地址我们是知道的，master上用于执行复制的用户名和密码是我们自己创建的，也没问题，但是master的binlog文件和position是怎么知道的？这不就是之前我们mysqldump导出的backup.sql里就有，大家在执行上述命令前，打开那个backup.sql就可以看到如下内容：

MASTER_LOG_FILE='mysql-bin.000015',MASTER_LOG_POS=1689

然后你就把上述内容写入到主从复制的命令里去了。

接着执行一个开始进行主从复制的命令：start slave，再用show slave status查看一下主从复制的状态，主要看到Slave_IO_Running和Slave_SQL_Running都是Yes就说明一切正常了，主从开始复制了。

接着就可以在主库插入一条数据，然后在从库查询这条数据，只要能够在从库查到这条数据，就说明主从复制已经成功了。

这仅仅是最简单的一种主从复制，就是异步复制，就是之前讲过的那种原理，从库是异步拉取binlog来同步的，所以肯定会出现短暂的主从不一致的问题的，比如你在主库刚插入数据，结果在从库立马查询，可能是查不到的。

后续我们会再继续讲解MySQL主从同步的其他几种方式。

End

122 如何为MySQL搭建一套主从复制架构？（2）

上回已经给大家讲解了如何为MySQL搭建一套主从复制架构，其实搭建一点都不难，相信大家自己照着之前讲解的步骤做，基本都能搭建出来一套主从复制的架构，只要你搭建出来主从复制架构，就可以实现读写分离了。

比如可以用mycat或者sharding-sphere之类的中间件，就可以实现你的系统写入主库，从从库去读取了。

但是现在搭建出来的主从复制架构有一个问题，那就是之前那种搭建方式他默认是一种异步的复制方式，也就是说，主库把日志写入binlog文件，接着自己就提交事务返回了，他也不管从库到底收到日志没有。

那万一此时要是主库的binlog还没同步到从库，结果主库宕机了，此时数据不就丢失了么？即使你做了高可用自动切换，一下子把从库切换为主库，但是里面是没有刚才写入的数据的，所以这种方式是有问题的。

因此一般来说搭建主从复制，都是采取半同步的复制方式的，这个半同步的意思，就是说，你主库写入数据，日志进入binlog之后，起码得确保 binlog日志复制到从库了，你再告诉客户端说本次写入事务成本了是不是？

这样起码你主库突然崩了，他之前写入成功的数据的binlog日志都是到从库了，从库切换为主库，数据也不会丢的，这就是所谓的半同步的意思。

这个半同步复制，有两种方式，第一种叫做AFTER_COMMIT方式，他不是默认的，他的意思是说，主库写入日志到binlog，等待binlog复制到从库了，主库就提交自己的本地事务，接着等待从库返回给自己一个成功的响应，然后主库返回提交事务成功的响应给客户端。

另外一种是现在MySQL 5.7默认的方式，主库把日志写入binlog，并且复制给从库，然后开始等待从库的响应，从库返回说成功给主库了，主库再提交事务，接着返回提交事务成功的响应给客户端。

总而言之，这种方式可以保证你每个事务提交成功之前，binlog日志一定都复制到从库了，所以只要事务提交成功，就可以认为数据在从库也有一份了，那么主库崩溃，已经提交的事务的数据绝对不会丢失的。

搭建半同步复制也很简单，在之前搭建好异步复制的基础之上，安装一下半同步复制插件就可以了，先在主库中安装半同步复制插件，同时还得开启半同步复制功能：

install plugin rpl_semi_sync_master soname 'semisync_master.so';

set global rpl_semi_sync_master_enabled=on;

show plugins;

可以看到你安装了这个插件，那就ok了。

接着在从库也是安装这个插件以及开启半同步复制功能：

install plugin rpl_semi_sync_slave soname 'semisync_slave.so';

set global rpl_semi_sync_slave_enabled=on;

show plugins;

接着要重启从库的IO线程：stop slave io_thread; start slave io_thread;

然后在主库上检查一下半同步复制是否正常运行：show global status like '%semi%';，如果看到了Rpl_semi_sync_master_status的状态是ON，那么就可以了。

到此半同步复制就开启成功了，其实一般来说主从复制都建议做成半同步复制，因为这样配合高可用切换机制，就可以保证数据库有一个在线的从库热备份主库的数据了，而且主要主库宕机，从库立马切换为主库，数据不丢失，数据库还高可用。

End

123 如何为MySQL搭建一套主从复制架构？（3）

之前给大家讲完了MySQL传统的主从复制搭建方式，其实一般大家在生产中都会采用半同步的复制模式，但是其实除了那种传统搭建方式之外，还有一种更加简便一些的搭建方式，就是GTID搭建方式，今天就给大家讲讲GTID的搭建方式。

首先在主库进行配置：

gtid_mode=on

enforce_gtid_consistency=on

log_bin=on

server_id=单独设置一个

binlog_format=row

接着在从库进行配置：

gtid_mode=on

enforce_gtid_consistency=on

log_slave_updates=1

server_id=单独设置一个

接着按照之前讲解的步骤在主库创建好用于复制的账号之后，就可以跟之前一样进行操作了，比如在主库dump出来一份数据，在从库里导入这份数据，利用mysqldump备份工具做的导出，备份文件里会有SET @@GLOBAL.GTID_PURGED=***一类的字样，可以照着执行一下就可以了。

接着其余步骤都是跟之前类似的，最后执行一下show master status，可以看到executed_gtid_set，里面记录的是执行过的GTID，接着执行一下SQL：select * from gtid_executed，可以查询到，对比一下，就会发现对应上了。

那么此时就说明开始GTID复制了。

其实大家会发现无论是GTID复制，还是传统复制，都不难，很简单，往往这就是比较典型的MySQL主从复制的搭建方式了，然后大家可以自行搜索一下MyCat中间件或者是Sharding-Sphere的官方文档，其实也都不难，大家照着文档做，整合到ava代码里去，就可以做出来基于主从复制的读写分离的效果了。

那些中间件都是支持读写分离模式的，可以仅仅往主库去写，从从库去读，这都没问题的。

如果落地到项目里，那么就完成了一个主从架构以及读写分离的架构了，此时按照我们之前所说的，如果说你的数据库之前对一个库的读写请求每秒总共是2000，此时读写分离后，也许就对主库每秒写TPS才几百，从库的读QPS是1000多。

那么万一你要是从库的读QPS越来越大，达到了每秒几千，此时你是不是会压力很大？没关系，这个时候你可以给主库做更多的从库，搭建从库，给他挂到主库上去，每次都在凌晨搞，先让系统停机，对外不使用，数据不更新。

接着对主库做个dump，导出数据，到从库导入数据，做一堆配置，然后让从库开始接着某个时间点开始继续从主库复制就可以了，一旦搭建完毕，就等于给主库挂了一个新的从库上去，此时继续放开系统的对外限制，继续使用就可以了，整个过程基本在1小时以内。

如果在凌晨比如2点停机1小时，基本对业务是没有影响的。

好，那么到此为止，主从复制这块就初步的算讲完了，下讲给大家介绍一下主从复制的延迟问题如何解决。

End

124 主从复制架构中的数据延迟问题，应该如何解决？

之前大家都已经了解过主从复制架构是如何搭建的了，其实他并不难，但是这里比较关键的是，主从复制可能会有较大的延迟。这个延迟是什么意思呢？就是说主库可能你都写入了100条数据了，结果从库才复制过去了50条数据，那么从库就比主库落后了50条数据。

这就是所谓的主从延迟的问题。

可是为什么会产生这个主从延迟的问题呢？也很简单，其实你主库是多线程并发写入的，这个大家都知道的，所以主库写入数据的速度可能是很快的，但是从库是单个线程缓慢拉取数据的，所以才会导致从库复制数据的速度是比较慢的。

那自然会导致主从之间的延迟问题了，大家想，是不是？

那么这个主从之间到底延迟了多少时间呢？这个可以用一个工具来进行监控，比较推荐的是percona-toolkit工具集里的pt-heartbeat工具，他会在主库里创建一个heartbeat表，然后会有一个线程定时更新这个表里的时间戳字段，从库上就有一个monitor线程会负责检查从库同步过来的heartbeat表里的时间戳。

把时间戳跟当前时间戳比较一下，其实就知道主从之间同步落后了多长时间了，关于这个工具的使用，大家可以自行搜索一下，我们这里就不展开了，总之，主从之间延迟了多长时间，我们这里实际上是可以看到的。

那么这个主从同步延迟的问题，会导致一些什么样的不良情况呢？

其实大家可以思考一下，如果你做了读写分离架构，写都往主库写，读都从从库读，那么会不会你的系统刚写入一条数据到主库，接着代码里立即就在从库里读取，可能此时从库复制有延迟，你会读不到刚写入进去的数据！

没错，就是这个问题，这是我们之前也经常会遇到的一个问题。另外就是有可能你的从库同步数据太慢了，导致你从库读取的数据都是落后和过期的，也可能会导致你的系统产生一定的业务上的bug。

所以针对这个问题，首先你应该做的，是尽可能缩小主从同步的延迟时间，那么怎么做呢？其实就是让从库也用多线程并行复制数据就可以了，这样从库复制数据的速度快了，延迟就会很低了。

MySQL 5.7就已经支持并行复制了，可以在从库里设置slave_parallel_workers>0，然后把slave_parallel_type设置为LOGICAL_CLOCK，就ok了。

另外，如果你觉得还是要求刚写入的数据你立马强制必须一定可以读到，那么此时你可以使用一个办法，就是在类似MyCat或者Sharding-Sphere之类的中间件里设置强制读写都从主库走，这样你写入主库的数据，强制从主库里读取，一定立即可以读到的。

总体而言就是这样了，大家在落实读写分离架构的时候，要注意一下复制方式，是异步还是半同步？如果说你对数据丢失并不是强要求不能丢失的话，可以用异步模式来复制，再配合一下从库的并行复制机制。

如果说你要对MySQL做高可用保证数据绝对不丢失的话，建议还是用半同步机制比较好一些，同理最好是配合从库的并行复制机制。

接下来配合这个主从复制架构，我们可以来讲解一下数据库的高可用架构了。

End

125 数据库高可用：基于主从复制实现故障转移（1）

这周我们来说说MySQL数据库的高可用架构如何搭建，上一周我们已经聊了MySQL的主从复制架构如何搭建了，说白了，就是允许主库把数据复制到从库上去，然后允许我们的系统往主库写入数据，从从库读取数据，实现一个读写分离的模式。

那么读写分离的模式确定了，接着就可以来考虑一下数据库的高可用架构了，所谓的高可用就是说，如果数据库突然宕机了一台机器，比如说主库或者从库宕机了，那么数据库还能正常使用吗？

其实如果从库宕机了影响并不是很大，因为大不了就是让所有的读流量都从主库去读就可以了，但是如果主库宕机了呢？那就真的麻烦了，因为主库一旦宕机，你就没法写入数据了，从库毕竟是不允许写入的，只允许读取。

所以有没有一种办法，可以在主库宕机之后，就立马把从库切换为主库呢，然后所有人都对从库切换为的主库去写入和读取呢？如果能实现这样的一个效果，那数据库不就实现高可用了吗？没错，就这么简单，这就是数据库的高可用架构。

一般生产环境里用于进行数据库高可用架构管理的工具是MHA，也就是Master High Availability Manager and Tools for MySQL，是日本人写的，用perl脚本写的一个工具，这个工具就是专门用于监控主库的状态，如果感觉不对劲，可以把从库切换为主库。

这个MHA自己也是需要单独部署的，分为两种节点，一个是Manager节点，一个是Node节点，Manager节点一般是单独部署一台机器的，Node节点一般是部署在每台MySQL机器上的，因为Node节点得通过解析各个MySQL的日志来进行一些操作。

Manager节点会通过探测集群里的Node节点去判断各个Node所在机器上的MySQL运行是否正常，如果发现某个Master故障了，就直接把他的一个Slave提升为Master，然后让其他Slave都挂到新的Master上去，完全透明。

其实这个原理是非常简单的，不过搭建的过程非常的复杂，因此我们可能要用本周一周的时间来一步一步讲解搭建过程，大家做好心理准备。

首先，大家最好是准备4台机器，其中一台机器装一个mysql作为master，另外两台机器都装mysql作为slave，然后在每个机器上都得部署一个MHA的node节点，然后用单独的最后一台机器装MHA的master节点，整体就这么一个结构。

首先，大家得确保4台机器之间都是免密码互相通信的，这个大家可以自行搜索一下，大量的方法可以做到，就是4台机器之间要不依靠密码可以直接ssh登录上去，因为这是MHA的perl脚本要用的。

接着大家就应该部署一个MySQL master和两个MySQL slave，搭建的过程就按照之前讲解的就行了，先装好MySQL，接着进行主从复制的搭建，全部按照之前的步骤走就行了，可以选择异步复制，当然也可以是半同步复制的。

这就是今天的内容，大家可以先自动准备好上述实验环境，然后本周接下来两天就会把剩余的MHA搭建步骤和实验步骤讲解完毕。

End

126 数据库高可用：基于主从复制实现故障转移（2）

今天我们正式来讲解MHA数据库高可用架构的搭建，先来讲解一下在三个数据库所在机器上安装MHA node节点的步骤，首先那必须要先安装Perl语言环境了，这就跟我们平时用Java开发，那你必须得先装个JDK吧！

所以先可以用yum装一下Perl语言环境：yum install perl-DBD-MySQL

然后从下述地址下载MHA node代码：https://github.com/yoshinorim/mha4mysql-node，接着就可以把node的压缩包用WinSCP之类的工具上传到机器上去，接着解压缩node包就可以了，tar -zxvf mha4mysql-node-0.57.tar.gz。

然后可以安装perl-cpan软件包：

cd mha4mysql-node-0.57

yum -y install perl-CPAN*

perl Makefile.PL

make && make install

到此为止，暂时node的安装就可以了，记得3个部署MySQL的机器都要安装node，接着就是安装MHA的manager节点，先安装需要的一些依赖包：

yum install -y perl-DBD-MySQL*

rpm -ivh perl-Params-Validate-0.92-3.el6.x86_64.rpm

rpm -ivh perl-Config-Tiny-2.12-1.el6.rfx.noarch.rpm

rpm -ivh perl-Log-Dispatch-2.26-1.el6.rf.noarch.rpm

rpm -ivh perl-Parallel-ForkManager-0.7.5-2.2.el6.rf.noarch.rpm

接着就可以安装manager节点了，先在下面的地址下载manager的压缩包：https://github.com/yoshinorim/mha4mysql-manager，然后上传到机器上去，按照下述步骤安装就可以了：

tar -zxvf mha4mysql-manager-0.57.tar.gz

perl Makefile.PL

make

make install

接着为MHA manager创建几个目录：/usr/local/mha，/etc/mha，然后进入到/etc/mha目录下，vi mha.conf一下，编辑他的配合文件

[server default]

user=zhss

password=12345678

manager_workdir=/usr/local/mha

manager_log=/usr/local/mha/manager.log

remote_workdir=/usr/local/mha

ssh_user=root

repl_user=repl

repl_password=repl

ping_interval=1

master_ip_failover_script=/usr/local/scripts/master_ip_failover

master_ip_online_change_script=/usr/local/scripts/master_ip_online_change

[server1]

hostname=xx.xx.xx.xx

ssh_port=22

master_binlog_dir=/data/mysqll

condidate_master=1

port=3306

[server1]

hostname=xx.xx.xx.xx

ssh_port=22

master_binlog_dir=/data/mysqll

condidate_master=1

port=3306

[server1]

hostname=xx.xx.xx.xx

ssh_port=22

master_binlog_dir=/data/mysqll

condidate_master=1

port=3306

上面那份配置文件就可以指导MHA manager节点去跟其他节点的node通信了，大家可以观察到，上面说白了都是配置一些工作目录，日志目录，用户密码之类的东西，还有一些脚本，另外比较关键的是，你有几个node节点，就配置一个server，把每个server的ip地址配置进去就可以了

接着创建存放脚本的目录：/usr/local/scripts，在里面需要放一个master_ip_failover脚本，vi master_ip_failover就可以了，输入下面的内容：

```
#!/usr/bin/env perl

use strict;
use warnings FATAL => 'all';

use Getopt::Long;
my (
    $command, $ssh_user, $orig_master_host, $orig_master_ip, $orig_master_port, $new_master_host, $new_master_ip, $new_master_port
);

my $vip = '192.168.56.123/24';
my $key = '0';
my $ssh_start_vip = "/sbin/ifconfig eth0:$key $vip";
my $ssh_stop_vip = "/sbin/ifconfig eth0:$key down";

GetOptions(
   'command=s' => \$command,
   'ssh_user=s' => \$ssh_user,
   'orig_master_host=s' => \$orig_master_host,
   'orig_master_ip=s' => \$orig_master_ip,
   'orig_master_port=i' => \$orig_master_port,
   'new_master_host=s' => \$new_master_host,
   'new_master_ip=s' => \$new_master_ip,
   'new_master_port=i' => \$new_master_port,
);

exit &main();

sub main {
    print "\n\nIN SCRIPT TEST----$ssh_stop_vip--$ssh_start_vip--\n\n";

    if($command eq "stop" || $command eq "stopssh") {
    my $exit_code=1;
    eval {
         print "Disabling the VIP on old master: $orig_master_host \n";
         &stop_vip();
         $exit_code=0;
    };
    if($@) {
        warn "Got Error: $@\n";
        exit $exit_code;
    }
    exit $exit_code;
}
elsif($command eq "start") {
     my $exit_code = 10;
     eval {
          print "Enabling the VIP - $vip on the new master - $new_master_host \n";
          &start_vip();
          $exit_code = 0;
     };
     if ($@) {
          warn $@;
          exit $exit_code;
     }
     elsif($command eq "status") {
           print "Checking the Status of the script.. OK \n";
           exit 0;
     }
     else {
          &usage();
          exit 1;
     }
}

sub start_vip() {
    `ssh $ssh_user\@$new_master_host \" $ssh_start_vip \"`;
}

sub stop_vip() {
    return 0 unless ($ssh_user);
    `ssh $ssh_user\@$orig_master_post \" $ssh_stop_vip \"`;
}

sub usage {
    print "Usage: master_ip_failover --command=start|stop|stopssh|status --orig_master_host=host --orig_master_ip=ip --orig_master_port=port --new_master_host=host --new_master_ip=ip --new_master_port=port\n";
}
```

接着在编辑一下online_change这个脚本，如下：

```
cd /usr/local/scripts/
vim master_ip_online_change
#!/usr/bin/env perl
use strict;
use warnings FATAL => 'all';

use Getopt::Long;
use MHA::DBHelper;
use MHA::NodeUtil;
use Time::HiRes qw( sleep gettimeofday tv_interval );
use Data::Dumper;

my $_tstart;
my $_running_interval = 0.1;

my $vip = "192.168.56.123";
my $if = "eth0";
```

```
my (
 $command,               $orig_master_is_new_slave,  $orig_master_host,
 $orig_master_ip,       $orig_master_port,           $orig_master_user,
 $orig_master_password, $orig_master_ssh_user,      $new_master_host,
 $new_master_ip,        $new_master_port,           $new_master_user,
 $new_master_password, $new_master_ssh_user,
);
GetOptions(
 'command=s'                    => \$command,
 'orig_master_is_new_slave'    =>\$orig_master_is_new_slave,
 'orig_master_host=s'           => \$orig_master_host,
 'orig_master_ip=s'             => \$orig_master_ip,
 'orig_master_port=i'           =>\$orig_master_port,
 'orig_master_user=s'           => \$orig_master_user,
 'orig_master_password=s'      => \$orig_master_password,
 'orig_master_ssh_user=s'      => \$orig_master_ssh_user,
 'new_master_host=s'            => \$new_master_host,
 'new_master_ip=s'              =>\$new_master_ip,
 'new_master_port=i'            => \$new_master_port,
 'new_master_user=s'            => \$new_master_user,
 'new_master_password=s'       => \$new_master_password,
 'new_master_ssh_user=s'       => \$new_master_ssh_user,
);
```

```
exit &main();
sub drop_vip {
    my $output = `ssh -oConnectTimeout=15  -oConnectionAttempts=3 $orig_master_host /sbin/ip addr del $vip/32 dev $if`;

}
sub add_vip {
    my$output = `ssh -o ConnectTimeout=15  -oConnectionAttempts=3 $new_master_host /sbin/ip addr add $vip/32 dev $if`;

}
```

```
sub current_time_us {
 my ( $sec,$microsec ) = gettimeofday();
 my $curdate =localtime($sec);
 return$curdate . " " . sprintf( "%06d", $microsec );
}

sub sleep_until {
 my $elapsed =tv_interval($_tstart);
 if ($_running_interval > $elapsed ) {
    sleep($_running_interval - $elapsed );
 }
}
```

```
sub get_threads_util {
 my $dbh                    = shift;
 my$my_connection_id       = shift;
 my$running_time_threshold = shift;
 my $type                   = shift;
 $running_time_threshold = 0 unless ($running_time_threshold);
  $type                  = 0 unless ($type);
  my @threads;

  my $sth =$dbh->prepare("SHOW PROCESSLIST");
 $sth->execute();

  while ( my$ref = $sth->fetchrow_hashref() ) {
   my $id         = $ref->{Id};
   my$user       = $ref->{User};
   my$host       = $ref->{Host};
   my$command    = $ref->{Command};
   my$state      = $ref->{State};
   my$query_time = $ref->{Time};
   my$info       = $ref->{Info};
   $info =~s/^\s*(.*?)\s*$/$1/ if defined($info);
   next if ($my_connection_id == $id );
   next if ( defined($query_time) &&$query_time < $running_time_threshold );
   next if (defined($command)    && $commandeq "Binlog Dump" );
   next if ( defined($user)       && $user eq "systemuser" );
   next
    if (defined($command)
    && $command eq "Sleep"
    && defined($query_time)
    && $query_time >= 1 );
```

```
    if ( $type>= 1 ) {
        next if (defined($command) && $command eq "Sleep" );
        next if (defined($command) && $command eq "Connect" );
      }

      if ( $type>= 2 ) {
        next if (defined($info) && $info =~ m/^select/i );
        next if (defined($info) && $info =~ m/^show/i );
      }

      push@threads, $ref;
    }
    return@threads;
}
```

```
sub main {
  if ( $commandeq "stop" ) {
    ##Gracefully killing connections on the current master
    # 1. Setread_only= 1 on the new master
    # 2. DROPUSER so that no app user can establish new connections
    # 3. Setread_only= 1 on the current master
    # 4. Killcurrent queries
    # * Anydatabase access failure will result in script die.
    my$exit_code = 1;
    eval {
      ##Setting read_only=1 on the new master (to avoid accident)
      my$new_master_handler = new MHA::DBHelper();

      # args:hostname, port, user, password, raise_error(die_on_error)_ or_not
     $new_master_handler->connect( $new_master_ip, $new_master_port,
       $new_master_user, $new_master_password, 1 );
      printcurrent_time_us() . " Set read_only on the new master.. ";
     $new_master_handler->enable_read_only();
      if ( $new_master_handler->is_read_only()) {
        print"ok.\n";
      }
      else {
        die"Failed!\n";
      }
     $new_master_handler->disconnect();
```

```
#Connecting to the orig master, die if any database error happens
my$orig_master_handler = new MHA::DBHelper();
$orig_master_handler->connect( $orig_master_ip, $orig_master_port,
 $orig_master_user, $orig_master_password, 1 );

 ## Dropapplication user so that nobody can connect. Disabling per-session binlogbeforehand
 $orig_master_handler->disable_log_bin_local();
 # printcurrent_time_us() . " Drpping app user on the orig master..\n";
  printcurrent_time_us() . " drop vip $vip..\n";
 #drop_app_user($orig_master_handler);
 &drop_vip();
```

```
 ## Waitingfor N * 100 milliseconds so that current connections can exit
      my$time_until_read_only = 15;
      $_tstart= [gettimeofday];
      my@threads = get_threads_util( $orig_master_handler->{dbh},
       $orig_master_handler->{connection_id} );
      while ($time_until_read_only > 0 && $#threads >= 0 ) {
        if ($time_until_read_only % 5 == 0 ) {
         printf
"%s Waiting all running %d threads aredisconnected.. (max %d milliseconds)\n",
           current_time_us(), $#threads + 1, $time_until_read_only * 100;
          if ($#threads < 5 ) {
           print Data::Dumper->new( [$_] )->Indent(0)->Terse(1)->Dump ."\n"
              foreach (@threads);
          }
        }
```

```
sleep_until();
      $_tstart = [gettimeofday];
      $time_until_read_only--;
      @threads = get_threads_util( $orig_master_handler->{dbh},
        $orig_master_handler->{connection_id} );
    }

    ##Setting read_only=1 on the current master so that nobody(except SUPER) canwrite
    printcurrent_time_us() . " Set read_only=1 on the orig master.. ";
   $orig_master_handler->enable_read_only();
    if ($orig_master_handler->is_read_only() ) {
     print"ok.\n";
    }
    else {
      die"Failed!\n";
    }
```

```
##Waiting for M * 100 milliseconds so that current update queries can complete
      my$time_until_kill_threads = 5;
      @threads= get_threads_util( $orig_master_handler->{dbh},
       $orig_master_handler->{connection_id} );
      while ($time_until_kill_threads > 0 && $#threads >= 0 ) {
        if ($time_until_kill_threads % 5 == 0 ) {
         printf
"%sWaiting all running %d queries are disconnected.. (max %dmilliseconds)\n",
           current_time_us(), $#threads + 1, $time_until_kill_threads * 100;
          if ($#threads < 5 ) {
           print Data::Dumper->new( [$_] )->Indent(0)->Terse(1)->Dump ."\n"
             foreach (@threads);
          }
        }
       sleep_until();
       $_tstart = [gettimeofday];
        $time_until_kill_threads--;
       @threads = get_threads_util( $orig_master_handler->{dbh},
         $orig_master_handler->{connection_id} );
      }
```

```
 ##Terminating all threads
      printcurrent_time_us() . " Killing all application threads..\n";
      $orig_master_handler->kill_threads(@threads)if ( $#threads >= 0 );
      printcurrent_time_us() . " done.\n";
     $orig_master_handler->enable_log_bin_local();
     $orig_master_handler->disconnect();

      ## Afterfinishing the script, MHA executes FLUSH TABLES WITH READ LOCK
     $exit_code = 0;
    };
    if ($@) {
      warn"Got Error: $@\n";
      exit$exit_code;
    }
    exit$exit_code;
  }
  elsif ($command eq "start" ) {
    ##Activating master ip on the new master
    # 1. Createapp user with write privileges
    # 2. Movingbackup script if needed
    # 3.Register new master's ip to the catalog database
```

```
# We don't return error even though activatingupdatable accounts/ip failed so that we don't interrupt slaves' recovery.
# If exit code is 0 or 10, MHA does not abort
    my$exit_code = 10;
    eval {
      my$new_master_handler = new MHA::DBHelper();

      # args:hostname, port, user, password, raise_error_or_not
     $new_master_handler->connect( $new_master_ip, $new_master_port,
       $new_master_user, $new_master_password, 1 );

      ## Setread_only=0 on the new master
     $new_master_handler->disable_log_bin_local();
      printcurrent_time_us() . " Set read_only=0 on the new master.\n";
     $new_master_handler->disable_read_only();

      ##Creating an app user on the new master
      #printcurrent_time_us() . " Creating app user on the new master..\n";
      printcurrent_time_us() . "Add vip $vip on $if..\n";
     #create_app_user($new_master_handler);
     &add_vip();
     $new_master_handler->enable_log_bin_local();
     $new_master_handler->disconnect();
```

```
## Updatemaster ip on the catalog database, etc
     $exit_code = 0;
    };
    if ($@) {
      warn"Got Error: $@\n";
      exit$exit_code;
    }
    exit$exit_code;
  }
  elsif ($command eq "status" ) {

    # donothing
    exit 0;
  }
  else {
    &usage();
    exit 1;
  }
}
```

```
sub usage {
  print
"Usage:master_ip_online_change --command=start|stop|status --orig_master_host=host--orig_master_ip=ip --orig_master_port=port --new_master_host=host--new_master_ip=ip --new_master_port=port\n";
  die;
}
```

完事儿过后，就可以给两个脚本增加权限：

chmod +x master_ip_failover

chmod +x master_ip_online_change

接着安装需要的软件包：yum -y install perl-Time-HiRes

执行SSH检测命令：/usr/local/bin/masterha_check_ssh --conf=/etc/mha/mha.conf

如果检测结果全部显示为OK，那么就代表你安装完毕了

然后检测主从架构：/usr/local/bin/masterha_check_repl --conf=/etc/mha/mha.conf

如果检测结果全部正常，那么就代表没问题了

好，今天我们就讲解到这里，下次继续讲解

End

127 数据库高可用：基于主从复制实现故障转移（3）

今天我们继续讲解数据库高可用架构的搭建，在做好了之前的准备工作之后，咱们今天就可以继续来做了，首先，要在MySQL主库所在的机器上去添加VIP，所谓VIP就是虚拟VIP地址，这个大家可以关注一下，不懂的自行搜索，是一个重要的网络概念。

ip addr add xx.xx.xx.xx dev eth0，这里的xx.xx.xx.xx，就是你自定义的一个VIP地址

接着就可以启动MHA manager节点了，在MHA manager所在机器上执行下述命令：nohup masterha_manager --conf=/etc/mha/mha.conf > /tmp/mha_manager.log < /dev/null 2>&1 &，这就可以启动MHA的manager节点了

接着验证一下启动是否成功：masterha_check_status --conf=/etc/mha/mha.conf，此时只要看到MHA manager正常工作就行了，接着就可以测试一下数据库高可用了，比如你可以先把主库停了：mysqladmin -uroot -proot shutdown

然后从库会自动获取到主库机器上的VIP的，同时从库会被转换为新的主库，其他从库也会指向新的主库，这些都是MHA自动给你完成的，然后你可以把宕机的主库重新启动，然后把他配置为从库，指向新的主库就可以了

整体来说，数据库的高可用架构就是这么个意思，其实搭建虽然很繁琐，但是只要搭建好了，基本就是比较自动化的了，相信大家结合之前的一些内容，应该都能理解，只不过在搭建的过程中可能会遇到一些小问题，可以自己尝试去解决一下。

今天的内容就讲到这里了，这也是本周的内容，多给大家留一些时间去自己尝试一下做这个实验，下周和下下周我们会讲解最后两周内容，是有关分库分表这块的一些实践案例的，大家有时间可以看一下儒猿技术窝的《互联网Java工程师面试突击第一季》里的分库分表的部分，里面有一些前置的分库分表的基础知识，大家可以先预习一下，后续我们会直接讲解分库分表这块的案例了。

End

图文 128 案例实战：大型电商网站的上亿数据量的用户表如何进行水平拆分？

手机观看

276 人次阅读 2020-11-11 07:00:00

详情 目录 评论

案例实战：大型电商网站的上亿数据量的用户表如何进行水平拆分？

石杉老师最新出品：《C2C 电商系统微服务架构 120 天实战训练营》，点击了解详情

儒猿学院官网上线，内有石杉老师架构课最新大纲，儒猿云平台详细介绍，敬请浏览

官网：www.ruyuan2020.com（**建议PC端访问**）

今天开始会用最后两周的时间给大家讲解一下数据库分库分表的实战案例，首先，关于分库分表需要大家有一些基础知识的掌握和积累，关于分库分表的基础知识，请大家去看一下儒猿技术窝里的《互联网Java工程师面试突击第一季》，那是完全免费的一个课程，里面有关于分库分表的一些基础知识。

儒猿技术窝

进店逛逛

相关频道

从零开始带你成为MySQL实战优化高手

已更新139期

其次，我们专栏里仅仅是讲解分库分表的整体方案的设计，但是在进行具体的方案落地的时候，是需要数据库中间件技术的支持的，业内常用的一般有Sharding-Sphere以及MyCat两种，各自用的公司都很多，都比较成熟，大家可以自行选择一个数据库中间件技术，去关注一下他们的官方文档，熟悉一下他们的用法。

因为上面介绍的两个数据库中间件技术都是国内开源的，所以文档都是有中文版的，大家阅读起来也是完全不费劲的。当大家有了上述的分库分表技术积累之后，那么就可以直接来阅读我们专栏里的分库分表案例实践了。

今天要给大家介绍的是平时我们见到的互联网公司里的海量用户数据的分库分表的方案，其实任何一个互联网公司都会有用户中心，这个用户中心就是负责这家公司的所有用户的数据管理，包括了用户的数据存储，用户信息的增删改查，用户的注册登录之类的。

而且一般互联网公司因为是直接面向终端用户的，所以用户数据一般都很多，哪怕是一个小互联网公司都能轻松拥有几百万级别的用户，如果是中型的互联网公司，一般达到几千万级别的用户都没问题，如果是大型互联网公司呢？那就是上亿，甚至几个亿级别的用户，甚至如果是BAT一类的巨头，用户体量可能是十亿级的。

所以说互联网公司的用户数据一般都是极为庞大的，那么我们今天的案例就选择一个背景是一个中型的电商公司，不是那种顶级电商巨头，就算是一个垂直领域的中型电商公司吧，那么他覆盖的用户大概算他有1亿以内，大概几千万的样子，就以这么个公司背景和用户量级来开展我们的案例。

首先，大家要明确一个要点，就是一般面对这么一个几千万级的数据，刚开始可能都是把数据放在MySQL的一个单库单表里的，但是往往这么大量级的数据到了后期，会搞的数据库查询速度很慢，因为结合之前的知识大家知道，数据量级太大了，会导致表 的索引很大，树的层级很高，进而导致搜索性能下降，而且能放内存缓存的数据页是比较少的。

所以说，往往我们都建议MySQL单表数据量不要超过1000万，最好是在500万以内，如果能控制在100万以内，那是最佳的选择了，基本单表100万以内的数据，性能上不会有太大的问题，前提是，只要你建好索引就行，其实保证MySQL高性能通常没什么特别高深的技巧，就是控制数据量不要太大，另外就是保证你的查询用上了索引，一般就没问题。

好，所以针对这个问题，我们就可以进行分库分表了，可以选择把这个用户大表拆分为比如100张表，那么此时几千万数据瞬间分散到100个表里去，类似user_001、user_002、user_100这样的100个表，每个表也就几十万数据而已。

其次，可以把这100个表分散到多台数据库服务器上去，此时要分散到几台服务器呢？你要考虑两个点，一个是数据量有多少个GB/TB，一个是针对用户中心的并发压力有多高。实际上一般互联网公司对用户中心的压力不会高的太离谱，因为一般不会有很多人同时注册/登录，或者是同时修改自己的个人信息，所以并发这块不是太大问题。

至于数据量层面的话，我可以给大家一个经验值，一般1亿行数据，大致在1GB到几个GB之间的范围，这个跟具体你

一行数据有多少字段也有关系，大致大致就是这么个范围，所以说你几千万的用户数据，往多了说也就几个GB而已。

这点数据量，对于服务器的存储空间来说，完全没压力，不是问题。

所以综上所述，此时你完全可以给他分配两台数据库服务器，放两个库，然后100张表均匀分散在2台服务器上就可以了，分的时候需要指定一个字段来分，一般来说会指定userid，根据用户id进行hash后，对表进行取模，路由到一个表里去，这样可以让数据均匀分散。

到此就搞定了用户表的分库分表，你只要给系统加上数据库中间件技术，设置好路由规则，就可以轻松的对2个分库上的100张表进行增删改查的操作了。平时针对某个用户增删改查，直接对他的userid进行hash，然后对表取模，做一个路由，就知道到哪个表里去找这个用户的数据了。

但是这里可能会出现一些问题，一个是说，用户在登录的时候，可能不是根据userid登陆的，可能是根据username之类的用户名，手机号之类的来登录的，此时你又没有userid，怎么知道去哪个表里找这个用户的数据判断是否能登录呢?

关于这个问题，一般来说常规方案是建立一个索引映射表，就是说搞一个表结构为（username, userid）的索引映射表，把username和userid一一映射，然后针对username再做一次分库分表，把这个索引映射表可以拆分为比如100个表分散在两台服务器里。

然后用户登录的时候，就可以根据username先去索引映射表里查找对应的userid，比如对username进行hash然后取模路由到一个表里去，找到username对应的userid，接着根据userid进行hash再取模，然后路由到按照userid分库分表的一个表里去，找到用户的完整数据即可。

但是这种方式会把一次查询转化为两个表的两次查询，先查索引映射表，再根据userid去查具体的数据，性能上是有一定的损耗的，不过有时候为了解决分库分表的问题，也只能用这种类似的办法。

另外就是如果在公司运营团队里，有一个用户管理模块，需要对公司的用户按照手机号、住址、年龄、性别、职业等各种条件进行极为复杂的搜索，这怎么办呢？其实没太多的好办法，基本上就是要对你的用户数据表进行binlog监听，把你要搜索的所有字段同步到Elasticsearch里去，建立好搜索的索引。

然后你的运营系统就可以通过Elasticsearch去进行复杂的多条件搜索，ES是适合干这个事儿的，然后定位到一批userid，通过userid回到分库分表环境里去找出具体的用户数据，在页面上展示出来即可。

这就是一套比较常规和完整的分库分表的方案。

End

儒猿技术窝精品专栏及课程推荐：

- 《从零开始带你成为消息中间件实战高手》
- 《互联网Java工程师面试突击》（第2季）
- 《互联网Java工程师面试突击》（第1季）
- 《互联网Java工程师面试突击》（第3季）
- 《从零开始带你成为JVM实战高手》
- 《C2C电商系统微服务架构120天实战训练营》

图文 129 案例实战：一线电商公司的订单系统是如何进行数据库设计的?

手机观看

210 人次阅读 2020-11-13 07:00:00

详情 目录 评论

案例实战：一线电商公司的订单系统是如何进行数据库设计的?

欢迎大家加入我们的**儒猿技术交流群**，一个纯粹的**交流技术、分享面经**的地方。

群里有一线大厂助教答疑、专栏优秀作业交流、互联网大厂面经分享、知名互联网公司内推，**点击下方链接了解**：

- 儒猿技术交流群学员面经分享
- 儒猿技术窝优秀学员作业展示
- 儒猿技术交流群日常技术交流及助教答疑

如何加群？点击了解

儒猿学院官网上线，内有石杉老师架构课最新大纲，儒猿云平台详细介绍，敬请浏览

官网：www.ruyuan2020.com（**建议PC端访问**）

今天我们来给大家讲讲第二个案例拓展，也就是一般互联网公司的订单系统是如何做分库分表的，既然要聊订单系统的分库分表，那么就得先说说为什么订单需要分库分表，其实最关键的一点就是要分析一下订单系统的数据量，那么订单系统的数据量有多大？这个就得看具体公司的情况了。

儒猿技术窝

进店逛逛

相关频道

从零开始带你成为MySQL实战优化高手

已更新139期

比如说一个小型互联网公司，如果是涉及到电商交易的，那么肯定每天都会有一些订单进来的，那么比如小型互联网公司假设有500万的注册用户，每天日活的用户会有多少人？意思就是说，你500万的注册用户，并不是每个人每天都来光顾你这里的！

我们往多了说，即使按照28法则，你500万的注册用户，每天最多是20%的用户会过来光顾你这里，也就是会来访问你的APP/小程序/网站，也就是100万的日活用户，但是这个日活比例恐怕很多公司都达不到，所以一般靠谱点就算他是10%的用户每天会来光顾你，算下来就是平均每个注册用户10天会来光顾你一次，这就是50万的日活用户。

但是这50万的日活用户仅仅是来看看而已，那么有多少人会来买你的东西呢？这个购买比例可就更低了，基本上很可能这种小型互联网公司每天就做个1w订单，或者几万订单，这就已经相当的不错了，咱们就以保守点按1w订单来算吧。

那么也就是说，这个互联网公司的订单表每天新增数据大概是1w左右，每个月是新增30w数据，每年是新增360w数据。大家对这个数据量感觉如何？看着不大是吧，但是按照我们上次说的，一般建议单表控制在千万以内，尽量是100w到500w之间，如果控制在几十万是最好了！

所以说，分析下来，大家会发现，哪怕是个小互联网公司，居然订单数据量也不少！因为订单这种数据和用户数据是不同的，你用户数据一般不会增长过快，而且很快会达到一个天花板，就不会怎么再涨了，但是订单数据是每天都有增量的，他们的特点是不同的。

所以说这个订单表，即使你按一年360w数据增长来计算，最多3年就到千万级大表了，这个就绝对会导致你涉及订单的操作，速度挺慢的。我这里可以给大家分享两个我亲身体验过的订单这块的案例。

一个是我使用过的某社保类的APP，这个APP可以让你在上面下单自助缴纳五险一金，你每次自助缴纳，说白了就是下一个订单，把钱给他，他帮你缴纳五险一金，这个东西对于很多自由职业者是很有用的。

这个APP，很明显就是订单日积月累很多，而且一定是没有做任何的分表，导致每次对自己的订单进行查询的时候，基本都是秒级，每次打开订单页面都很慢，有时候甚至会达到两三秒的样子，这个体验就很差。

另外一个是我使用过的一个企业银行的APP，大家都知道，企业银行是可以允许财务提交打款申请，然后有人可以去审批的，但是有一个银行APP，很明显也是对这类申请和审批的数据表，没有做分库分表的处理，导致数据日积月累的增加，每次在申请和审批的查询界面都很慢，起码要卡1s以上的时间，这个体验也很不好。

所以说，基本上个这类订单表，哪怕是个小互联网公司，按分库分表几乎是必须得做的，那么怎么做呢？订单表，一般在拆分的时候，往往要考虑到三个维度，一个是必然要按照订单id为粒度去分库分表，也就是把订单id进行hash后，对表数量进行取模然后把订单数据均匀分散到100~1000个表里去，再把这些表分散在多台服务器上。

但是这里有个问题，另外两个维度是用户端和运营端，用户端，就是用户可能要查自己的订单，运营端就是公司可能要查所有

订单，那么怎么解决这类问题呢？其实就跟上次的差不多，基本上针对用户端，你就需要按照（userid, orderid）这个表结构，去做一个索引映射表。

userid和orderid的一一对应映射关系要放在这个表里，然后针对userid为粒度去进行分库分表，也就是对userid进行hash后取模，然后把数据均匀分散在很多索引映射表里，再把表放在很多数据库里。

然后每次用户端拿出APP查询自己的订单，直接根据userid去hash然后取模路由到一个索引映射表，找到这个用户的orderid，这里当然可以做一个分页了，因为一般订单都是支持分页的，此时可以允许用于户分页查询orderid，然后拿到一堆orderid了，再根据orderid去按照orderid粒度分库分表的表里提取订单完整数据。

至于运营端，一般都是要根据N多条件对订单进行搜索的，此时跟上次讲的一样，可以把订单数据的搜索条件都同步到ES里，然后用ES来进行复杂搜索，找出来一波orderid，再根据orderid去分库分表里找订单完整数据。

其实大家到最后会发现，分库分表的玩法基本都是这套思路，按业务id分库分表，建立索引映射表同时进行分库分表，数据同步到ES做复杂搜索，基本这套玩法就可以保证你的分库分表场景下，各种业务功能都可以支撑了。

End

儒猿技术窝精品专栏及课程推荐：

- 《从零开始带你成为消息中间件实战高手》
- 《互联网Java工程师面试突击》（第2季）
- 《互联网Java工程师面试突击》（第1季）
- 《互联网Java工程师面试突击》（第3季）
- 《从零开始带你成为JVM实战高手》
- 《C2C电商系统微服务架构120天实战训练营》

图文 130 案例实战：下一个难题，如果需要进行垮库的分页操作，应该怎么来做?

手机观看

105 人次阅读 2020-11-16 07:00:00

详情 目录 评论

案例实战：下一个难题，如果需要进行垮库的分页操作，应该怎么来做?

石杉老师最新出品：《C2C 电商系统微服务架构 120 天实战训练营》，点击了解详情

儒猿学院官网上线，内有石杉老师架构课最新大纲，儒猿云平台详细介绍，敬请浏览

官网：www.ruyuan2020.com（**建议PC端访问**）

今天我们要来给大家分享的一个案例拓展，是关于分库分表后的跨库/跨表的分页问题，首先我们先来聊聊这个所谓的分页是个什么场景。那比如说还是说之前的那个订单的场景，假设用户现在要查询自己的订单，同时订单要求要支持分页，该怎么做?

儒猿技术窝

进店逛逛

相关频道

从零开始带你成为MySQL实战优化高手

已更新139期

其实按我们之前所说的，基本上你只要按照userid先去分库分表的（userid, orderid）索引映射表里查找到你的那些orderid，然后搞一个分页就可以了，对分页内的orderid，每个orderid都得去按orderid分库分表的数据里查找完整的订单数据，这就可以搞定分库分表环境的下分页问题了。

这仅仅是一个例子，告诉你的是，如果要在分库分表环境下搞分页，最好是保证你的一个主数据粒度（比如userid）是你的分库分表的粒度，你可以根据一个业务id路由到一个表找到他的全部数据，这就可以做分页了。

但是此时可能有人会提出一个疑问了，那如果说现在我想要对用户下的订单做分页，但是同时还能支持指定一些查询条件呢？对了，这其实也是很多APP里都支持的，就是对自己的订单查询，有的APP是支持指定一些条件的，甚至是排序规则，比如订单名称模糊搜索，或者是别的条件，比如说订单状态。

举个例子吧，比如说最经典的某个电商APP，大家平时都玩儿的一个，在我的订单界面，可以按照订单状态来搜索，分别是全部、待付款、待收货、已完成、已取消几个状态，同时就是对订单购买的商品标题进行模糊搜索。

那么此时你怎么玩儿分页呢？因为毕竟你的索引映射表里，只有（userid, orderid）啊！可是这又如何呢？你完全可以在这个索引映射表里加入更多的数据，比如（userid, orderid, order_status, product_description），加上订单所处的状态，以及商品的标题、副标题等文本。

然后你在对我的订单进行分页的 时候，直接就可以根据userid去索引映射表里找到用户的所有订单，然后按照订单状态、商品描述文本模糊匹配去搜索，完了再分页，分页拿到的orderid，再去获取订单需要展示的数据，比如说订单里包含的商品列表，每个商品的缩略图、名称、价格以及所属店铺。

那如果是针对运营端的分页查询需求呢？这还用说？上次都提过了，数据直接进入ES里，通过ES就可以对多条件进行搜索同时再进行分页了，这很好搞定！

当然，网上是有人说过一些所谓的跨库的分页方案，比如说一定要针对跨多个库和多个表的数据搞查询和分页，那这种如果你一定要做，基本上只能是自己从各个库表拉数据到内存，自己内存里做筛选和分页了，或者是基于数据库中间件去做，那数据库中间件本质也是干这个，把各个库表的数据拉到内存做筛选和分页。

实际上我是绝对反对这种方案的，因为效率和性能极差，基本都是几秒级别的速度。

所以当你觉得似乎必须要跨库和表查询和分页的时候，我建议你，第一，你考虑一下是不是可以把你查询里按照某个主要的业务id进行分库分表建立一个索引映射表，第二是不是可以可以把这个查询里要的条件都放到索引映射表里去，第三，是不是可以通过ES来搞定这个需求。

尽可能还是按照上述思路去做分库分表下的分页，而不要去搞什么跨库/表的分页查询。

End

儒猿技术窝精品专栏及课程推荐：

- 《从零开始带你成为消息中间件实战高手》
- 《互联网Java工程师面试突击》（第2季）
- 《互联网Java工程师面试突击》（第1季）
- 《互联网Java工程师面试突击》（第3季）
- 《从零开始带你成为JVM实战高手》
- 《C2C电商系统微服务架构120天实战训练营》