消费者

一个正常的消费者逻辑需要具备以下几个步骤

①配置消费者客户端参数及创建相应的消费者实例

②订阅主题

③拉取消息并消费

④提交消费位移

⑤关闭消费者实例

1. 必要的参数配置

bootstrap.servers:用来指定连接Kafka集群所需的broker地址清单，多个地址用逗号隔开，此参数默认为””

group.id:消费者隶属的消息组的名称，默认值为””。如果设置为空，则会报出异常。

1. 订阅主体与分区

一个消费者可以订阅多个主题，subscribe()

订阅方式

①集合订阅subscribe(Collection) AUTO\_TOPICS

②正则表达式订阅subscribe(Pattern) AUTO\_PATTERN

③指定分区的订阅方式assign(Collection) USER\_ASSIGN

如果都没有订阅，那么订阅状态为NONE

这三个状态是互斥的，只能使用一种，否则会抛出异常

assign()方法不具备消费者自动均衡的功能

1. 反序列化

自定义反序列化实现Deserializer接口

一般不建议使用自定义的序列化和反序列化器，这样会增加生产者与消费者之间的耦合度。如果上游类型改变，下游也得修改。

1. 消息消费

Kafaka中的消费是基于拉模式的。消息消费一般是两种模式：推模式和拉模式。

public ConsumerRecords<K,V> poll(final Duration timeout)

包含：消费位移、消费者协调器、组协调器、消费者的选举、分区分配的分发、再均衡的逻辑、心跳等内容

1. 位移提交

Rabbitmq

<http://svip.iocoder.cn/RabbitMQ/Interview/>

1、自我介绍，项目介绍，遇到的难点？产生原因？如何解决？

2、HashMap1.8与1.7区别？ConcurrentHashMap实现原理 ？

3、jvm类加载器，自定义类加载器，双亲委派机制，优缺点，tomcat类加载机制?

4、tomcat热部署，怎么做到的热加载？

5、cms收集器过程，g1收集器原理，怎么实现可预测停顿的，region的大小结构？

6、内存溢出，内存泄漏遇到过吗？什么场景产生的，怎么解决的？

7、volatile的原理？synchronized和重入锁实现原理以及区别？

8、redis字符串实现，sds和c区别？

9、redis集群，为什么是16384个slot？选举过程，会有脑裂问题么，raft算法，优缺点？

10、redis有序集合怎么实现的，跳表是什么？往跳表添加一个元素的过程获取分数的时间复杂度，为什么不用红黑树，红黑树有什么特点，左旋右旋操作？

11、锁升级过程，轻量锁可以变成偏向锁么？偏向锁可以变成无锁么？对象头结构，锁状态变化过程？

12、Innodb的结构了解么？磁盘页和缓存区是怎么配合的？缓冲区和磁盘数据不一致怎么办，服务器突然宕机了数据会丢失么？

13、InnoDB 索引为什使用B+树而不是用B树？

14、MySQL 分表是怎么实现的？跨库join如何解决？数据量突增怎么解决？

15、数据库的隔离级别，怎么实现的？当前读，快照读？MVCC？

16、mysql优化的实践经验

17、分布式事务出现过不一致吗？为什么？怎么解决？有什么方法避免？怎么监控？监控到怎么处理？什么时候需要人工接入？

18、io模型了解么？多路复用？selete，poll，epoll，epoll的结构？怎么注册事件？

19、你们用的什么消息中间件，kafka，为什么用kafka？kafka是怎么保证高吞吐量的？

20、kafka重平衡，重启服务怎么保证kafka不发生重平衡，有什么方案？

21、netty的原理和使用？tcp的连接过程？一台服务器能支持多少连接，为什么 ？tcp各个参数怎么设置？

22、Sping的AOP实现原理，以及对象生成方式的种类，单例的还是原型的？

23、讲讲调度接口是怎么实现的

24、分布式唯一ID是怎么实现的

25、设计模式，以及自己使用的场景

26、有没有用过分布式锁，怎么实现的，讲讲原理

27、如何解决线上问题？cpu狂飙怎么办？频繁minor gc怎么办？可能造成的原因是什么？如何避免？

28、怎么理解分布式和微服务，为什么要拆分服务，会产生什么问题，怎么解决这些问题 ？

29、怎么理解高可用，如何保证高可用，有什么弊端，熔断机制，怎么实现 ？

30、对于高并发怎么看，怎么算高并发，你们项目有么，如果有会产生什么问题，怎么解决

31、有没有做过压测的项目？首页接口优化是怎么做的？

32、如何优雅的写代码？什么代码算做优雅？什么代码是规范？你们代码规范是什么样的？如何进行code review？

33、算法：给定一个长度为N的整形数组arr，其中有N个互不相等的自然数1-N，请实现arr的排序，但是不要把下标0∼N−1位置上的数通过直接赋值的方式替换成1∼N

34、算法：判断一个树是否是平衡二叉树

35、算法：给定一个二叉树，请计算节点值之和最大的路径的节点值之和是多少，这个路径的开始节点和结束节点可以是二叉树中的任意节点

36、算法：LRU 缓存

37、算法：实现带有getMin功能的栈，要求push，pop，getMin的时间复杂度都是O(1)

38、算法：两数之和

39、算法：实现二叉树先序，中序和后序遍历

40、你对于自己的规划是怎么样的？

41、什么时候能入职，你对岗位的期望是什么

42、你还在面其他公司么，目前是一个什么流程

一、LinkedHashMap

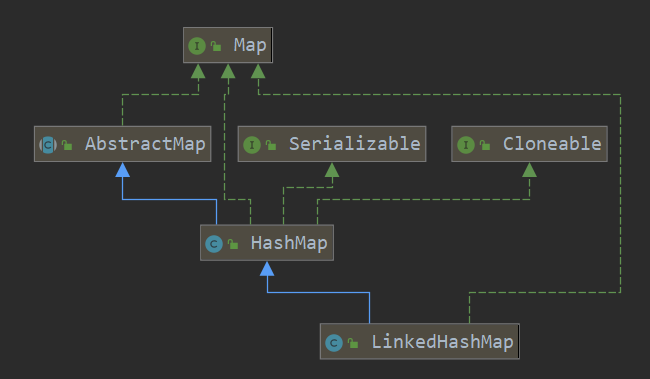
1. 概述：

HashMap是无序的，有时候需要有序访问的HashMap.

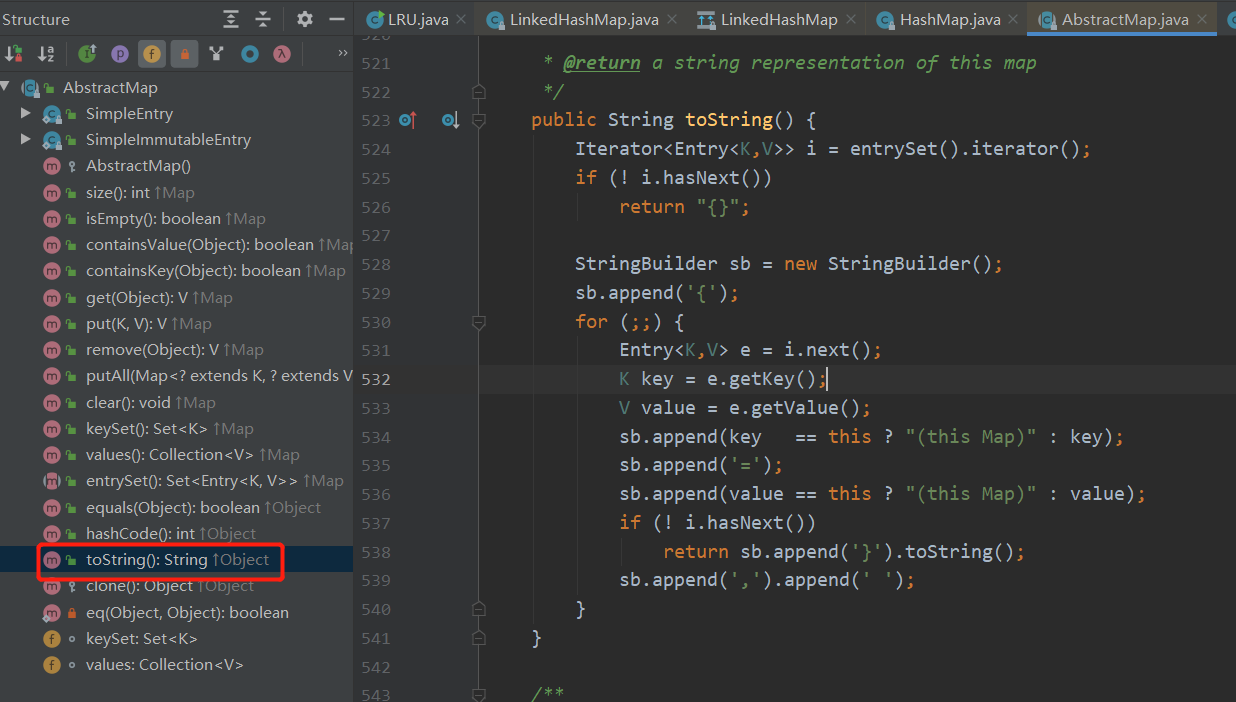
1. TreeMap:按照key的顺序
2. LinkedHashMap：按照key的插入和访问的顺序
3. 类图：

LinkedHashMap继承于HashMap，实现Map接口。

LinkedHashMap是HashMap的子类，增加了顺序访问的属性。



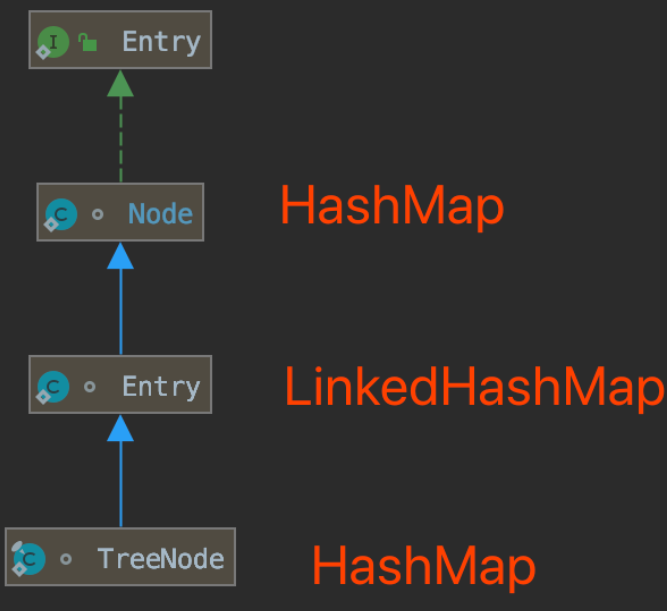
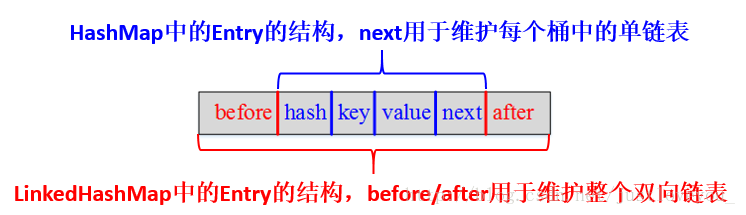
AbstractMap重写了toString()方法。所以可以直接输出map打印的不是物理地址。



1. 属性

LinkedHashMap实现了自定义的节点Entry，一个支持指向前后节点的Node子类。

*/\*\*  
 \* HashMap.Node subclass for normal LinkedHashMap entries.  
 \*/*static class Entry<K,V> extends HashMap.Node<K,V> {  
 Entry<K,V> before, after;  
 Entry(int hash, K key, V value, Node<K,V> next) {  
 super(hash, key, value, next);  
 }  
}



next用于维护HashMap各个桶中Entry的连接顺序，before、after用于维护Entry插入的先后顺序

*/\*\**

*\* 头结点*

*\* 越老的节点，放在越前面。所以头结点，指向链表的开头  
 \* The head (eldest) of the doubly linked list.  
 \*/*transient LinkedHashMap.Entry<K,V> head;  
  
*/\*\**

*\*尾结点*

*\* 越新的节点，放在越后面。所以尾结点。指向链表的结尾  
 \* The tail (youngest) of the doubly linked list.  
 \*/*transient LinkedHashMap.Entry<K,V> tail;  
  
*/\*\**

*是否按照访问的顺序*

*true 按照访问顺序*

*false 按照插入顺序，默认为false  
 \* The iteration ordering method for this linked hash map: <tt>true</tt>  
 \* for access-order, <tt>false</tt> for insertion-order.  
 \*  
 \** ***@serial*** *\*/*final boolean accessOrder;

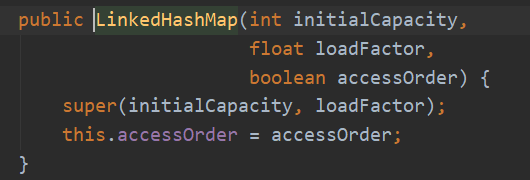
本质上，LinkedHashMap=HashMap+双向链表。在不对HashMap做任何改变的基础上，给HashMap的任意两个节点间加了两条连线(before指针和after指针)，使这些节点形成一个双向链表。每次put进来的Entry还会将其插入到双向链表的尾部。访问的顺序为head=>tail的过程。

1. 构造方法

一共5个构造方法，其中四个和HashMap相同，只是多初始化accessOrder=false。



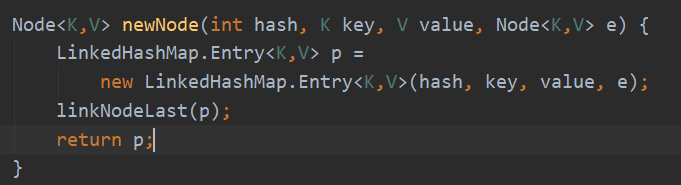
还有一个允许自定义accessOrder属性。

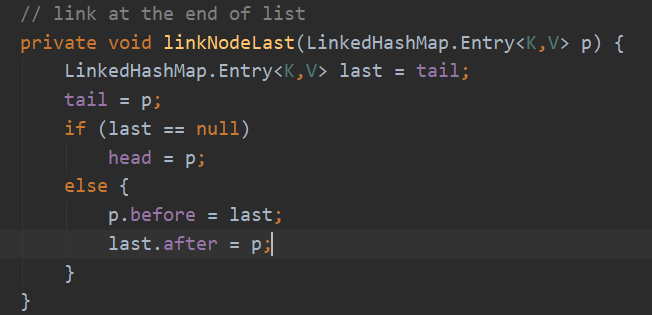


5.创建节点、节点操作回调

put()方法用的HashMap.put()方法

创建新节点时，重写了newNode()方法。 ①创建Entry节点②添加到结尾





节点操作回调

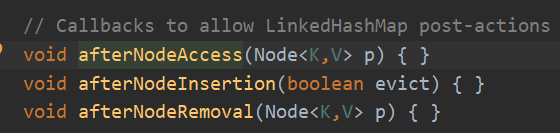
在 HashMap 的读取、添加、删除时，分别提供了

#afterNodeAccess(Node<K,V> e)

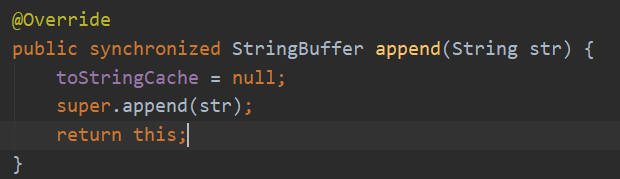
#afterNodeInsertion(boolean evict)

#afterNodeRemoval(Node<K,V> e)回调方法。

HashMap都是空方法，LinkedHashMap自定义重写了。



二、StringBuffer线程安全，StringBuilder线程不安全



StringBuffer加了synchronized，所以效率上会慢一些。

1. Mysql相关

ACID:原子性、一致性、隔离性、持久性

A：undolog，记录回滚信息

I：锁和MVCC

D：通过redolog记录物理变化，binlog记录逻辑变化

行锁通过给索引上的索引项加锁来实现。

binlog 和redo log记录过程：

对于语句 update T set c=c+1 where ID=2;

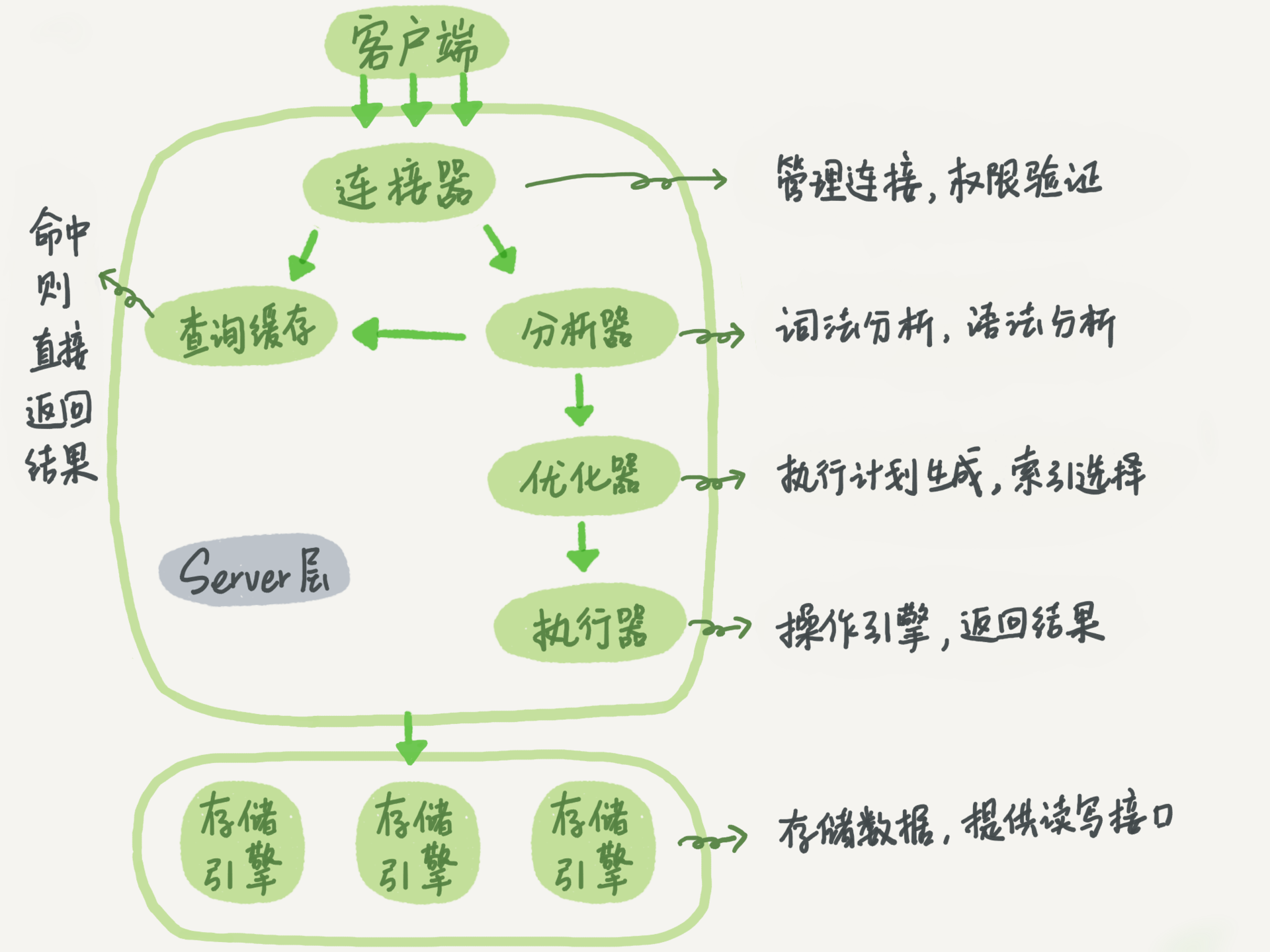
①执行器先找引擎取 ID=2 这一行。ID 是主键，直接用树搜索找到。如果 ID = 2 这一行所在数据页就在内存中，就直接返回给执行器；否则，需要先从磁盘读入内存，再返回。

②执行器拿到引擎给的行数据，把这个值加上 1，N+1，得到新的一行数据，再调用引擎接口写入这行新数据。

③引擎将这行新数据更新到内存中，同时将这个更新操作记录到 redo log 里面，此时 redo log 处于 prepare 状态。然后告知执行器执行完成了，随时可以提交事务。

④执行器生成这个操作的 binlog，并把 binlog 写入磁盘。

⑤执行器调用引擎的提交事务接口，引擎把刚刚写入的 redo log 改成提交（commit）状态，更新完成



3.1 隔离级别

事务支持是在引擎层实现的。

Mysql中InnoDB支持事务。MyISAM不支持事务。

多个事务同时执行时，可能出现脏读、不可重复读、幻读

Mysql事务隔离级别：读未提交、读已提交、可重复读、串行化

可重复读：一个事务执行过程中看到的数据，总是跟这个事务在启动时看到的数据是一致的。

读已提交和可重复读的实现：

读已提交：视图是在每个SQL语句开始执行时的创建的

可重复读：视图是在事务启动时创建的，整个事务存在期间都用这个视图。

串行化是通过直接加锁的方式来避免并行访问。

Oracle数据库默认隔离界别其实就是“读提交”

3.2 隔离级别的配置

show variables like 'transaction\_isolation';

MVCC多版本并发控制，同一条记录在系统中存在多个版本

事务启动的方式：

①显式启动事务语句，begin或者start transaction,提交commit，回滚rollback；

②set autocommit=0，该命令会把这个线程的自动提交关掉。

建议使用方法一，如果考虑多一次交互问题，可以使用commit work and chain语法

为什么建议你尽量不要使用长事务？

①回滚段的影响②长事务还占用锁资源

长事务意味着系统里面会存在很老的事务视图。由于这些事务随时可能访问数据库里面的任何数据，所以这个事务提交之前，数据库里面它可能用到的回滚记录都必须保留，这就会导致大量占用存储空间。

如何查询长事务

select \* from information\_schema.innodb\_trx where TIME\_TO\_SEC(timediff(now(),trx\_started))>60;

3.3 隔离性实现-读已提交和可重复读

InnoDB 的默认事务隔离级别是可重复读

3.3.1 当前读-current read：总是读取已经提交完成的最新版本

除了update以外，select加锁也是当前读，例如：

mysql> select k from t where id=1 lock in share mode;//加了读锁

mysql> select k from t where id=1 for update;//加了写锁

PS:共享锁S锁都是读锁，排他锁X锁都是写锁，说法不同

3.3.2 而读提交的逻辑和可重复读的逻辑类似，它们最主要的区别是：

在可重复读隔离级别下，只需要在事务开始的时候创建一致性视图，之后事务里的其他查询都共用这个一致性视图；

在读提交隔离级别下，每一个语句执行前都会重新算出一个新的视图。

start transaction with consistent snapshot; 执行这个语句的时候，就开启事务

start transaction; 从执行第一个语句才会开启事务

一个数据版本，对于一个事务视图来说，除了自己的更新总是可见以外，有三种情况：

版本未提交，不可见；

版本已提交，但是是在视图创建后提交的，不可见；

版本已提交，而且是在视图创建前提交的，可见。

InnoDB 的行数据有多个版本，每个数据版本有自己的 row trx\_id，每个事务或者语句有自己的一致性视图。普通查询语句是一致性读，一致性读会根据 row trx\_id 和一致性视图确定数据版本的可见性。

对于可重复读，查询只承认在事务启动前就已经提交完成的数据；

对于读提交，查询只承认在语句启动前就已经提交完成的数据；而当前读，总是读取已经提交完成的最新版本。

3.3.3 为什么表结构不支持“可重复读”?

这是因为表结构没有对应的行数据，也没有 row trx\_id，因此只能遵循当前读的逻辑。

MySQL 8.0 已经可以把表结构放在 InnoDB 字典里了，也许以后会支持表结构的可重复读???

大家需要区分行锁、MDL 锁和表锁的区别。对 InnoDB 表更新一行，可能过了 MDL 关，却被挡在行锁阶段。

3.4 幻读

两阶段锁协议：两段锁协议是指所有事务必须分两个阶段对数据项加锁和解锁

两段锁协议不要求事务必须一次将所有要使用的数据全部加锁。

为了解决交叉调度的不可串行化问题。

3.4.1 幻读的定义和幻读有什么问题：

幻读指的是一个事务在前后两次查询同一个范围的时候，后一次查询看到了前一次查询没有看到的行。

幻读是针对insert导致的数据不一致。专指新插入的行

不可重复读是针对 delete、update导致的数据不一致。

a.读未提交：可能出现脏读，不可重复读，幻读；

b.读已提交：可能出现不可重复读，幻读；

c.可重复读：可能出现幻读；

d.串行化：都没问题

幻读是当前读才会出现的，快照读不会出现。

幻读出现的问题：

①语义上不正确，本来说是把某一条件查询的数据锁住，但是看起来并没有锁住。

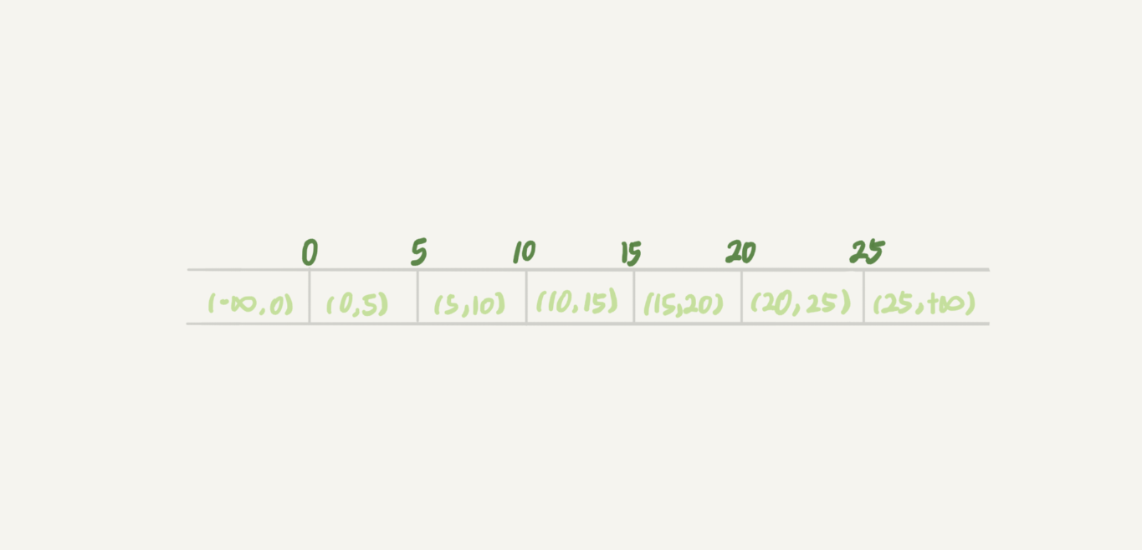
②保证不了数据和日志在逻辑上的一致性。

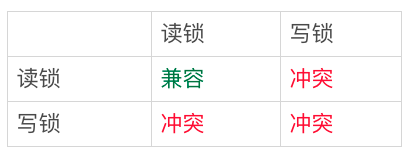
即使把所有的记录都加上锁，还是阻止不了新插入的记录，这也是为什么“幻读”会被单独拿出来解决的原因。

3.4.2 如何解决幻读

为了解决幻读问题，InnoDB 只好引入新的锁，也就是间隙锁 (Gap Lock)

6条数据产生了7个间隙



行锁：

间隙锁和行锁不一样，间隙锁之间都不存在冲突关系。

跟间隙锁存在冲突关系的，是“往这个间隙中插入一个记录”这个操作。

间隙锁和行锁合称 next-key lock

每个next-key lock是前开后闭区间(5,10]

间隙锁是开区间，间隙锁+行锁=next-key lock

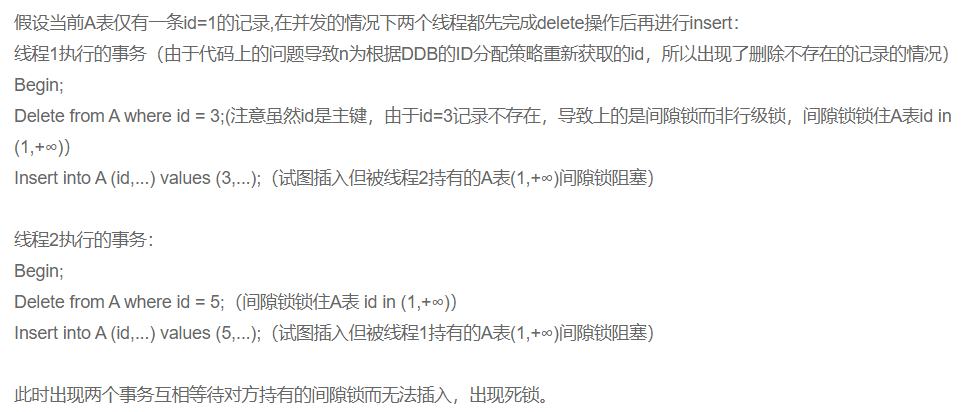
读已提交是没有间隙锁的概念，这时候需要解决数据和日志不一致的问题，把binlog格式设置为row。

可重复读有间隙锁的概念。

3.4.3 说一个间隙锁引发的死锁：

删除或者更新表中不存在的记录，容易产生间隙锁死锁。

网上摘抄：

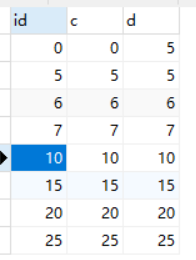


自我模拟：

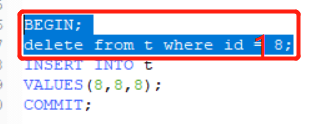
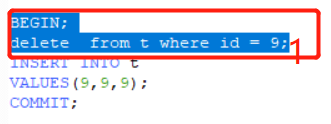
先关闭当前会话自动提交：

select @@autocommit;

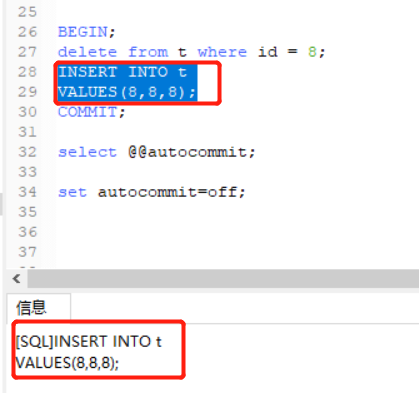
set autocommit=off;



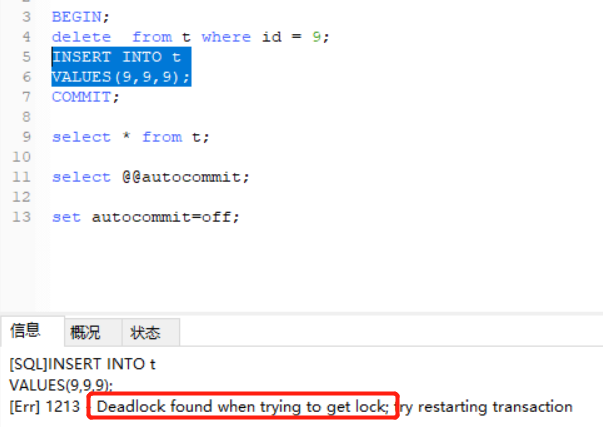
表里面并没有id=8和id=9这两条记录

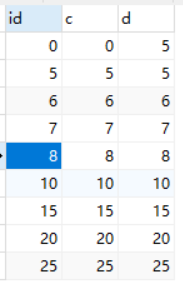
①先分别执行如图两个会话的1

②再执行第一个会话的Insert语句，此时出现阻塞



③再执行第二个会话的insert语句，出现死锁。此时检测到死锁，该事务就回滚了，释放了间隙锁。第一个会话执行成功，成功插入id=8。



select \* from t;

3.5 索引

3.5.1 索引常见模型

哈希表：适用于只有等值查询的场景

有序数组：在等值查询和范围查询场景中的性能就都非常优秀，但是对于更新或者插入数据就比较困难。只适用于静态存储引擎

搜索树：时间复杂度O(log(N))

3.5.2 InnoDB的索引模型

InnoDB 使用了 B+ 树索引模型，所以数据都是存储在 B+ 树中的。

为什么这么选择？

B+ 树能够很好地配合磁盘的读写特性，减少单次查询的磁盘访问次数

根据叶子节点的内容，索引类型分为主键索引和非主键索引

主键索引也称为聚簇索引，非主键索引也称为二级索引

基于主键索引和普通索引的查询有什么区别？

如果语句是 select \* from T where ID=500，即主键查询方式，则只需要搜索 ID 这棵 B+ 树；如果语句是 select \* from T where k=5，即普通索引查询方式，则需要先搜索 k 索引树，得到 ID 的值为 500，再到 ID 索引树搜索一次。这个过程称为回表。也就是说，基于非主键索引的查询需要多扫描一棵索引树。因此，我们在应用中应该尽量使用主键查询。

尽量使用自增主键，①性能上，自增主键有序，避免索引树数据页的分裂和合并，提高性能

②存储空间上，整型做主键，则只要4个字节，如果长整型只要8个字节。并且二级索引存放的叶子节点及主键字段占用空间也就越小。

如果用业务字段作为主键，则往往不容易保证有序插入，这样写数据成本会比较高。

哪些场景适合用业务字段直接做主键呢？

①只有一个索引 ②该索引必须是唯一索引 典型的KV场景

general\_log：会记录所有sql，短时间内开启，对排查和分析mysql的性能问题，还是有很大的帮助。

show global variables like '%general%';

set global general\_log=off;

-- 慢查询

show variables like 'slow\_query%';

-- 查询超过多少秒才记录

show variables like 'long\_query\_time';

Percona 的 pt-kill 这个工具不错，推荐使用

如何避免长事务对业务的影响?

首先，从应用开发端：

①确认是否使用了 set autocommit=0

②确认是否有不必要的只读事务。把好几个 select 语句放到了事务中。这种只读事务可以去掉。

③通过 SET MAX\_EXECUTION\_TIME 命令，来控制每个语句执行的最长时间

其次，数据库端来看：

①监控 information\_schema.Innodb\_trx 表，设置长事务阈值，超过就报警 / 或者 kill

②在业务功能测试阶段要求输出所有的 general\_log，分析日志行为提前发现问题；

③如果使用的是 MySQL 5.6 或者更新版本，把 innodb\_undo\_tablespaces 设置成 2（或更大的值）。如果真的出现大事务导致回滚段过大，这样设置后清理起来更方便。

innodb\_undo\_tablespaces：默认值为0，表示不独立设置undo的tablespace，默认记录到ibdata中。不支持后期修改，修改会导致MySQL无法启动。

3.5.3 索引类型

覆盖索引

最左前缀原则

索引下推优化：MySQL 5.6 引入

在索引遍历过程中，对索引中包含的字段先做判断，直接过滤掉不满足条件的记录，减少回表次数。

3.5.4 B树和B+树

http://www.liuzk.com/410.html

1. JVM虚拟机

五、深入理解Java线程池

<http://www.ideabuffer.cn/2017/04/04/%E6%B7%B1%E5%85%A5%E7%90%86%E8%A7%A3Java%E7%BA%BF%E7%A8%8B%E6%B1%A0%EF%BC%9AThreadPoolExecutor/>

六、设计模式