B树和B+树



|  |  |
| --- | --- |
| **B树** | **B+树** |
| **每个节点既有键值(k) 又有数据(v)，指针** | **数据都在叶子节点** |
| **叶子节点没有关系** | **叶子节点相互之间有引用链路，**  **mysql里更好的全表扫描** |
| **查找数据不一定到叶子节点** | **查找数据一定到叶子节点** |

**可视化工具 https://visualgo.net**

**红黑树又叫平衡二叉树**

**通常大家以为point即为所创建的类，其实并不是这样，实际上point是我们创建的对象的引用。这个语句可以分解为两部分来看：**

**Point point;//①**

**point=new Point();//②**

**Point是这里自定义的一个类型，即类类型。**

**① 类类型的变量point，是Point类引用变量，它存放在栈空间中。也就是用来指向Point的对象的对象引用。但是此时point不是任何对象的引用，其指向为null，因为对象要在语句②中才创建。**

**在语句②中由关键字new创建了一个Point的对象，并将之存放在堆中，然后利用操作符“="使得point引用变量指向刚刚创建的在堆中的对象。**

**对象本身看不见摸不着，它是堆中的一块内存实体，甚至说对象本身没有名字。Point不是对象，因为一个类可以创建n个对象不可能每一个都叫做Point，point也不是对象它是Point定义的一个变量，经由语句②它成为了对象的引用。有了对象的引用，就可以方便的调用方法等操作。**

HashMap **数组+链表+红黑树（jdk1.8）**

**链表长度>8 链表转换成红黑树 链表长度<6 换回链表**

**O(n) O(logn)**

**用来处理有键值对的数据**

**new HashMap<>().put(null,null) 允许使用空值和空键**

**非线程安全，多个数据写入时，可能导致数据不一致**

**transient int size 实际键值对数量**

**transient int modCount 内部结构发生变化的次数**

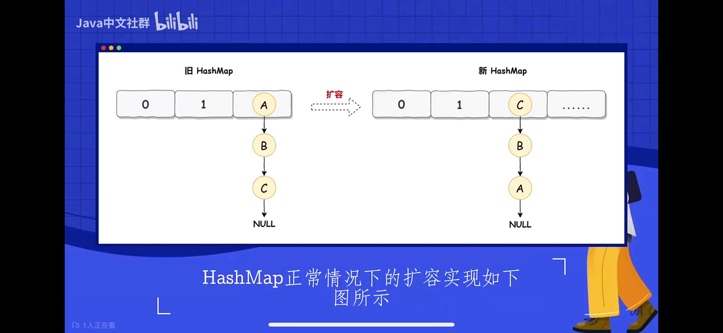
**int threshold(思锐思修的) 容纳键值对的临界值=数组长度\*负载因子**

**Final float loadFactor(楼的飞科特) 负载因子 0.75**

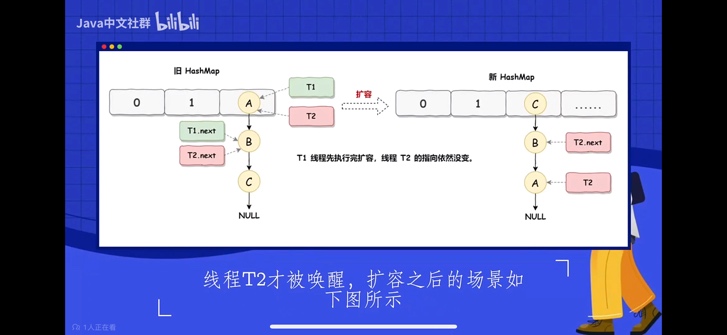
**默认容量16 Hashmap调用hashcode（）方法获取索引下标。**

死循环原因 **（发生在jdk1.7） 运行机制和并发扩容操作**

**T1 T2线程进行扩容操作**

 **因为插入方式头插法**

**T1扩容完唤醒T2 T2对刚才操作不可知，指向没变**



解决方法

**用ConcurrentHashMap （推荐）**

**用HashTable （性能低）**

**用Synchronized或Lock加锁HashMap­­­­­­**

数据覆盖：**并发添加元素**

**T1添加数据时，判断位置可以插入，在真正插入之前，时间片用完了，T2开始执行，在一个位置插入，T1恢复时，T1值也插入**

解决方法 **用ConcurrentHashMap（推荐）**

无序性：**添加和查询顺序不一致**

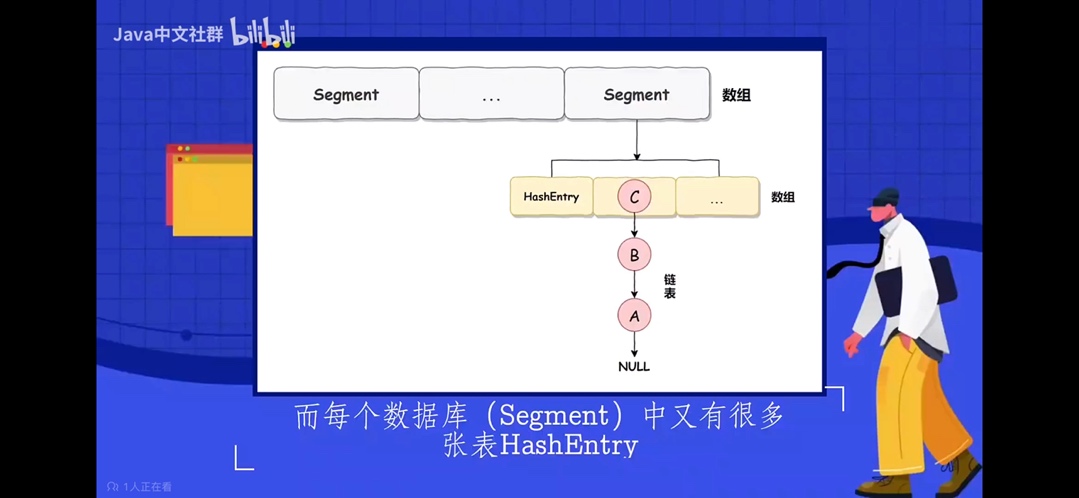
解决方法 **用LinkedHashMap**

ConcurrentHashMap**（肯康润特）**

**线程安全 不能插入空值（源码如果空值抛出空指针异常）防止并发场景下的歧义**

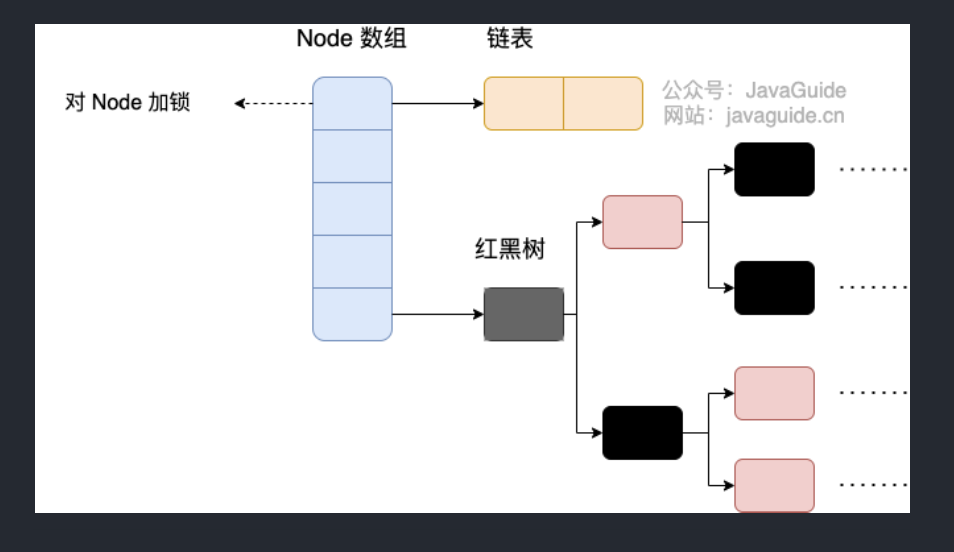
为什么保证线程安全

在Jdk1.7 **数组+链表**



**如何保证线程安全：Segment（固定16，不能扩容）基于ReenTrantLock（可重入锁）实现的加锁和释放锁的操作，访问ConcurrentHashMap时，同一时间只有一个线程能操作相应的节点 HashEntry可扩容**

在jdk1.8 node数组 单向链表 红黑树



**在头节点中加锁CAS或synchronized，就锁单个元素。**

接口和抽象类有什么共同点和区别？

共同点 ： 2

**1.都不能被实例化**

**2.都可以包含抽象方法 public abstract/public static final**

**3.都可以有默认实现的方法(Java8可以用default关键字在接口中定义默认方法)**

区别 ：3

**1.接口主要用于对类的行为进行约束，你实现了某个接口就具有了对应的行为。抽象类主要用于代码复用，强调的是所属关系。**

**2.一个类只能继承一个类，但是可以实现多个接口。**

**3.接口中的成员变量只能是public static final类型的，不能被修改且必须有初始值，而抽象类的成员变量默认 default，可在子类中被重新定义，也可被重新赋值。**

**为什么重写equals要重写hashcode：在hash表中，一个完全相同的对象会存储在hash表的两个位置。** 保证在不同场景下使用不会出错

进程与线程

**进程是程序的一次执行过程，是系统运行程序的基本单位**

**线程是程序中执行的基本单元，进程的基本执行单位**

**线程的生命周期和状态**

**NEW: 初始状态**

**RUNNABLE: 运行状态**

**BLOCKED ：阻塞状态**

**WAITING：等待状态**

**TIME\_WAITING：超时等待状态**

**TERMINATED：终止状态**

**创建线程三种方式：继承Thread，实现Runnable接口，实现Callable接口**

**三种方式的区别：**

1. **Thread是类只能单一继承，runnable是接口支持多继承，在已存在继承关系类中，只能实现runnable**
2. **Runnable表示线程的顶级接口，Thread类其实是实现Runnable接口，两者都要实现run方法**
3. **在面向对象思想来说，Runnable是一个任务，Thread才是处理线程，用Runnable定义任务交给Thread处理。**

**什么是线程安全**

**当多个线程访问某个类时，不管运行时环境采用何种调度或者这些线程将如何交替执行，并且在调用代码中不需要任何额外的同步或者协同，这个类都能表现出正确的行为，那么这个类就是线程安全的**

**锁：多线程变成单线程**

Synchronized（森困niced） 同步锁

同一时刻最多只能有一个线程执行这段代码

同步方法 多个线程同时执行该方法时，也不会有线程安全问题

不同锁线程之间不会相互争夺

|  |  |
| --- | --- |
| sleep | wait |
| 线程方法 在Thread类里 | 锁方法 在Object类 |
| 不需要线程有锁 | 需要线程有锁 |
| 不支持手动唤醒 | 用notify,notifyAll唤醒 |
| 支持自动唤醒 | 支持自动唤醒 |
| 支持中断interrupt 发生中断异常 | 支持中断interrupt 发生中断异常 |
| 不释放锁 | 立即释放锁 |
| 一种线程状态TIME\_WAITING | 两种线程状态TIME\_WAITING，WAITING，因为有三个方法，wait无参数是WAITING，有参数的两个是TIME\_WAITING |

volatile（vo冷胎儿） 仅限于一写多读

保证变量的可见性，不保证原子性

插入特定的内存屏障来禁止指令重排序

**构造方法不能使用 synchronized 关键字修饰。**

Lock 显式锁 是一个接口 实现线程同步

方法：lock（）获取锁 unlock（）释放锁

synchronized时自动获取锁和释放锁（隐式锁） 弥补其不足

常用子类 ReenTrantLock (润恩去特) 可重入锁

非阻塞式获取锁（不排队，做别的事） trylock（）返回boolean

重量级锁：依赖操作系统

Java不能直接操作线程，依赖操作系统申请，操作系统内核很多命令：ring0，ring1，ring2，ring3，Java（应用层/用户态）操作ring3及以上，阻塞是ring0，Synchronized依赖操作系统进行阻塞所以是重量级锁

内核态ring0到用户态ring3及以上叫上下文切换，效率不高

Synchronized和volatile是两个互补的存在，而不是对立的存在

1. volatile关键字是线程同步的轻量级实现，所以volatile性能肯定比Synchronized关键字要好。但是volatile关键字只能用于变量而Synchronized 关键字可以修饰方法以及代码块
2. volatile关键字能保证数据的可见性，但不能保证数据的原子性。Synchronized关键字两者都能保证

三．volatile关键字主要用于解决变量在多个线程之间的可见性，而Synchronized关键字解决的是多个线程之间访问资源的同步性

原子操作类

AtomicLong（啊套米壳 ） 加了Synchronized的long

private static AtomicLong counter = new AtomicLong(0);

结果实时 底层原理： CAS+volatile和native方法来保证原子操作

CAS compare and swap（斯瓦pe） 比较并交换

自旋锁就是无锁。因为没有加锁。锁是一种很重量级的东西，例如sy.....加上之后程序运行速度奇慢无比。而此时出现了代替他的自旋锁。

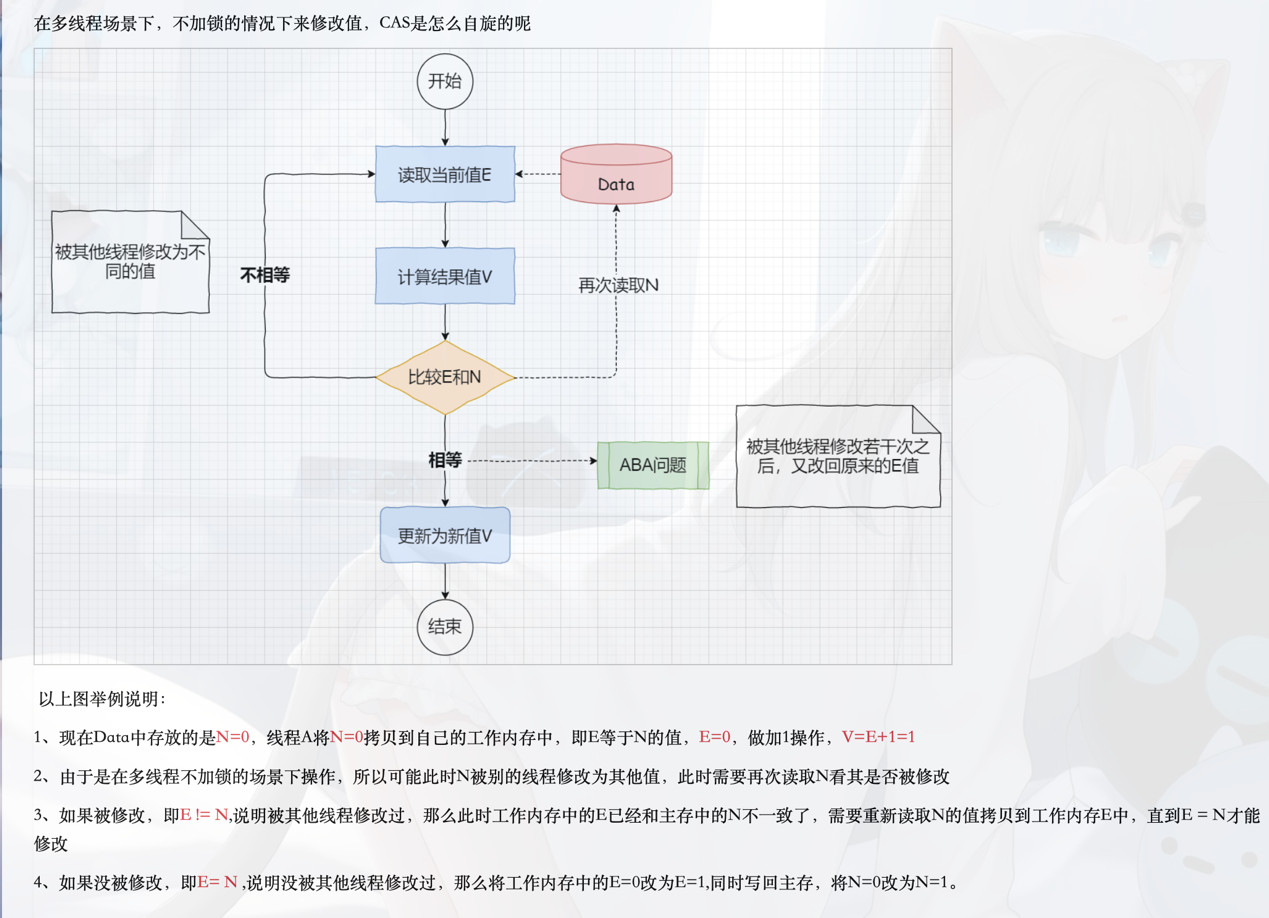
在多线程没有锁的状态下，可以保证多个线程对同一个值的更新

一种乐观锁 不涉及上下文切换

基本数据类型是交换compare and swap 对象是设置compare and set

public final boolean compareAndSet(int expect ,int update)

如比较当前工作内存中的值等于期望值，就更新，否则就不更新。



CAS为什么这么厉害（CAS源码分析：在unsafe类下）

是一个cpu指令，速度极快

调用CAS调用的是native方法：native说明是通过JNI调用C++的代码了,实现CAS跟操作系统,CPU有关，通过hotspot(jvm) cpu的cmpxchg指令，在主板加锁，保证比较和交换的原子性。

JNI(Java Native Interface)

Java本地接口书写程序,在 Java 虚拟机内运行的 Java 代码能够与其它编程语言互相操作，包括创建本地方法、更新Java对象、调用Java方法，引用 Java类，捕捉和抛出异常等。

存在三个问题

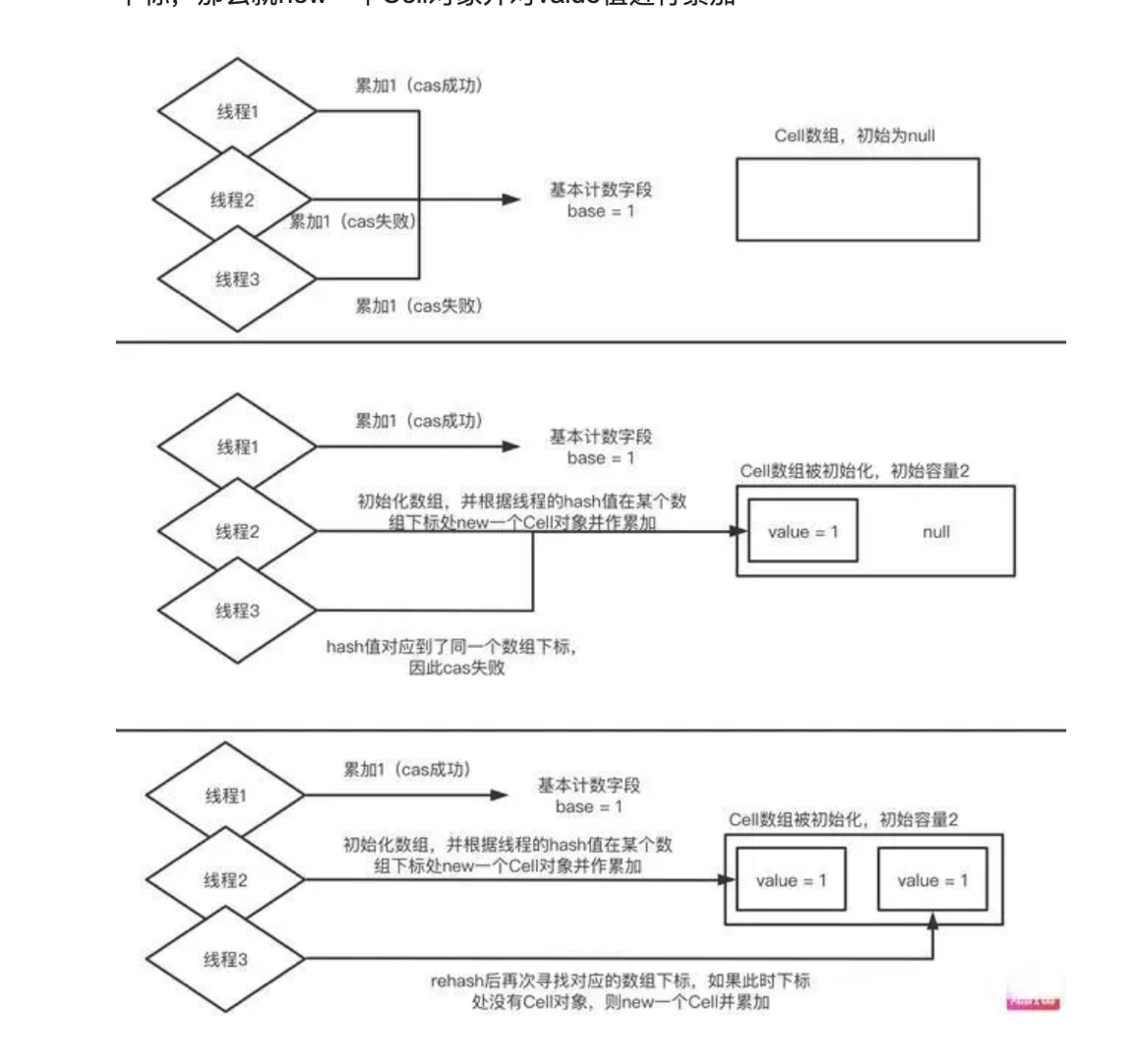
1.只能一个变量 用AtomicReference

2.ABA 问题 用AtomicStampedReference定义版本

3.cpu开销 极端情况下自旋多次

解决自旋：jdk1.8 推荐使用LongAdder 多线程计数的更优解 结果延迟+base

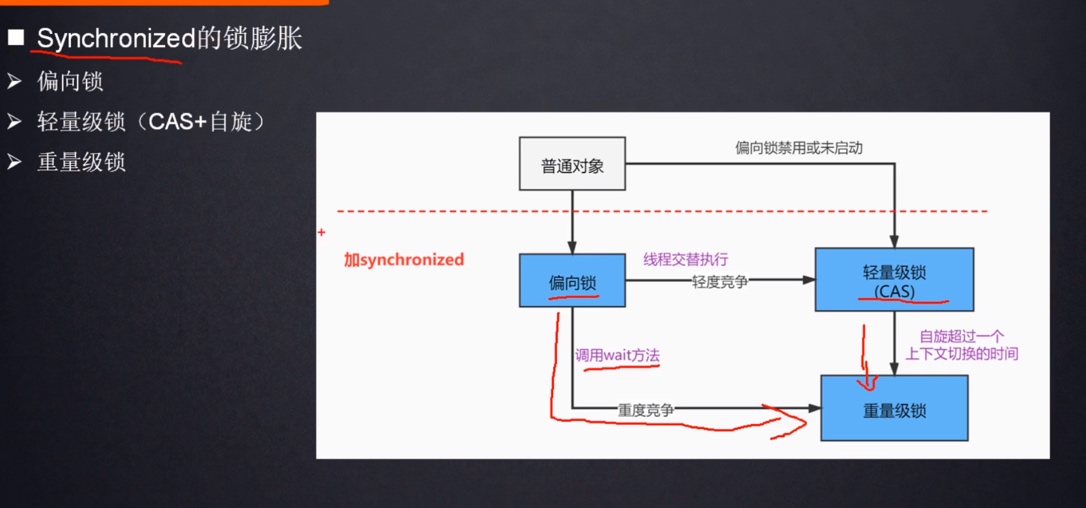
https://baijiahao.baidu.com/s?id=1681836058712094026&wfr=spider&for=pc



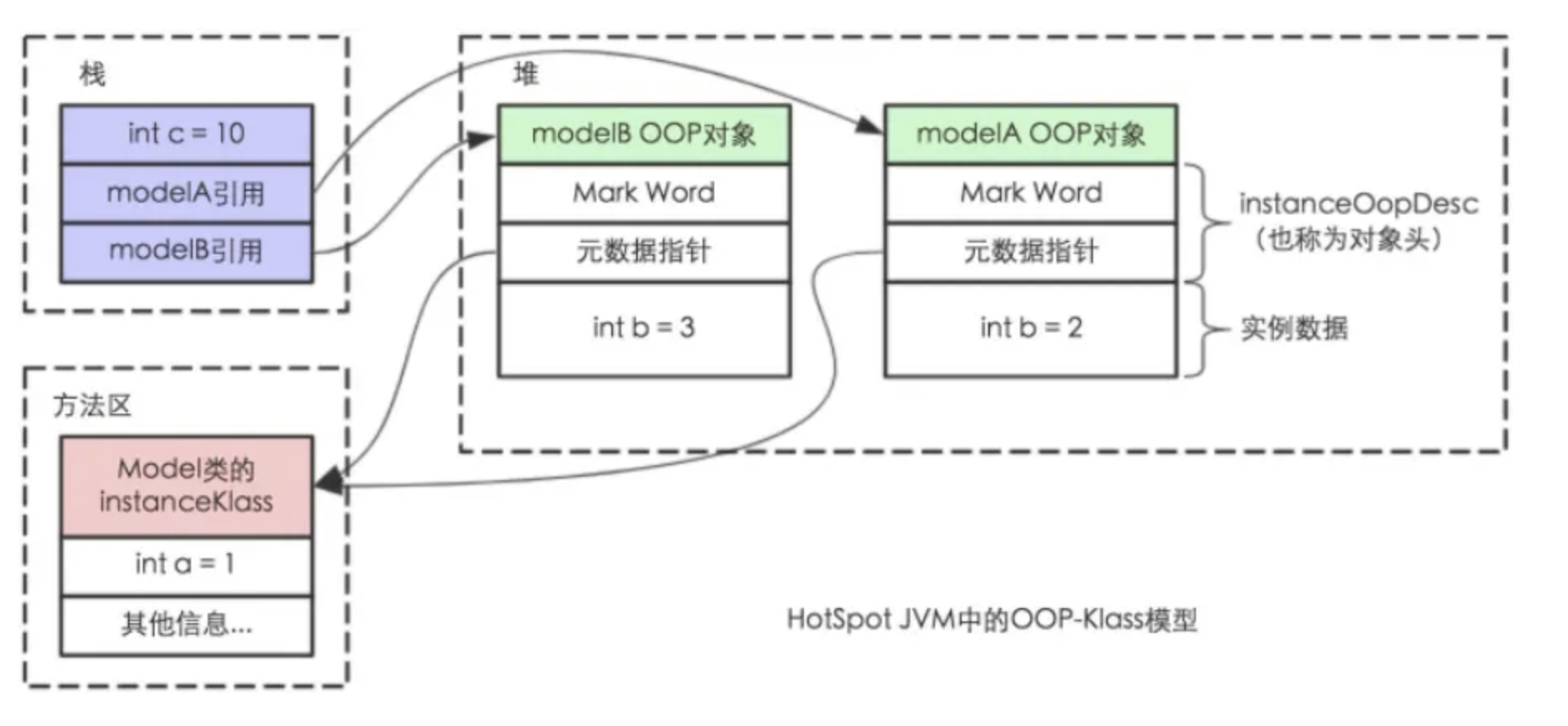
单线程加普通对象+synchronized就是偏向锁，有线程竞争，撤销偏向锁。synchronized本质是在对象头markword写锁标识。

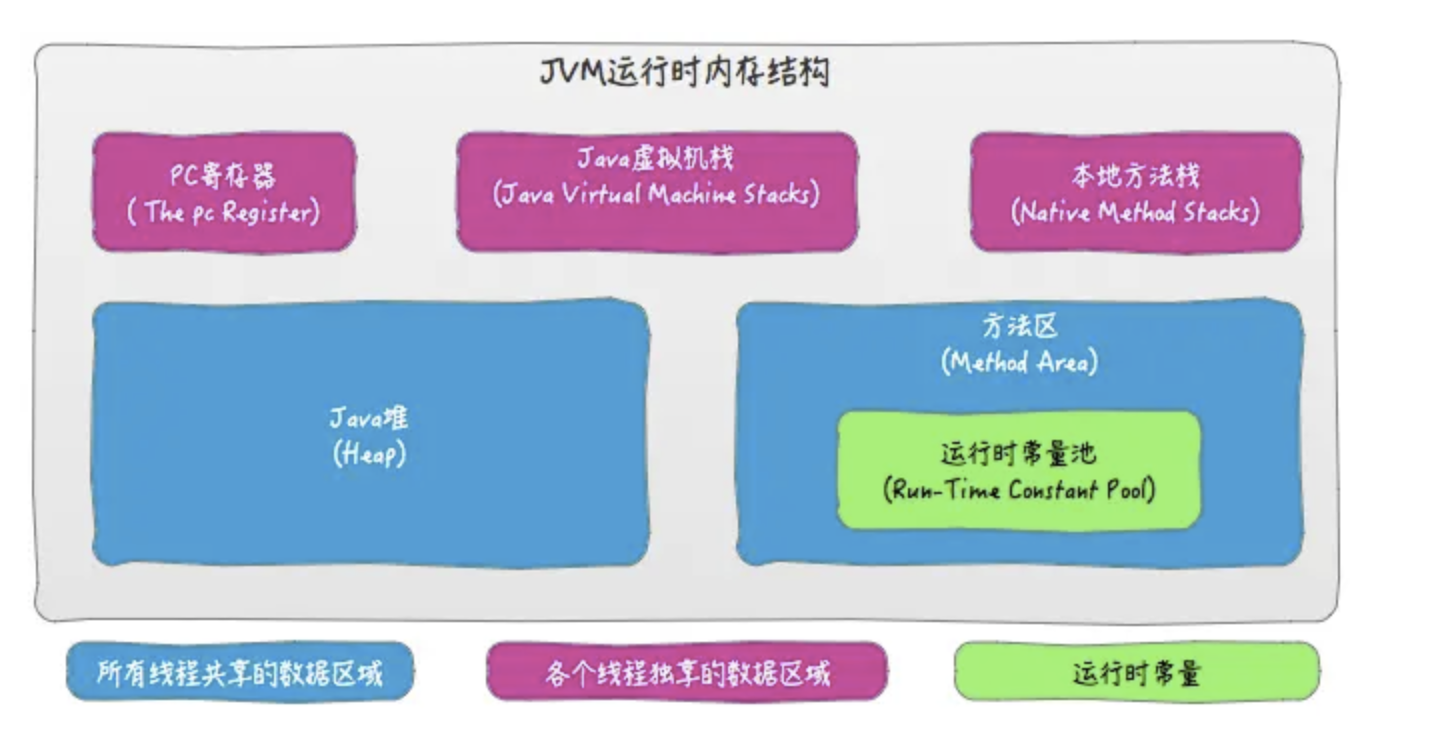
各种锁：https://zhuanlan.zhihu.com/p/139021371

根据运行状态进行升级

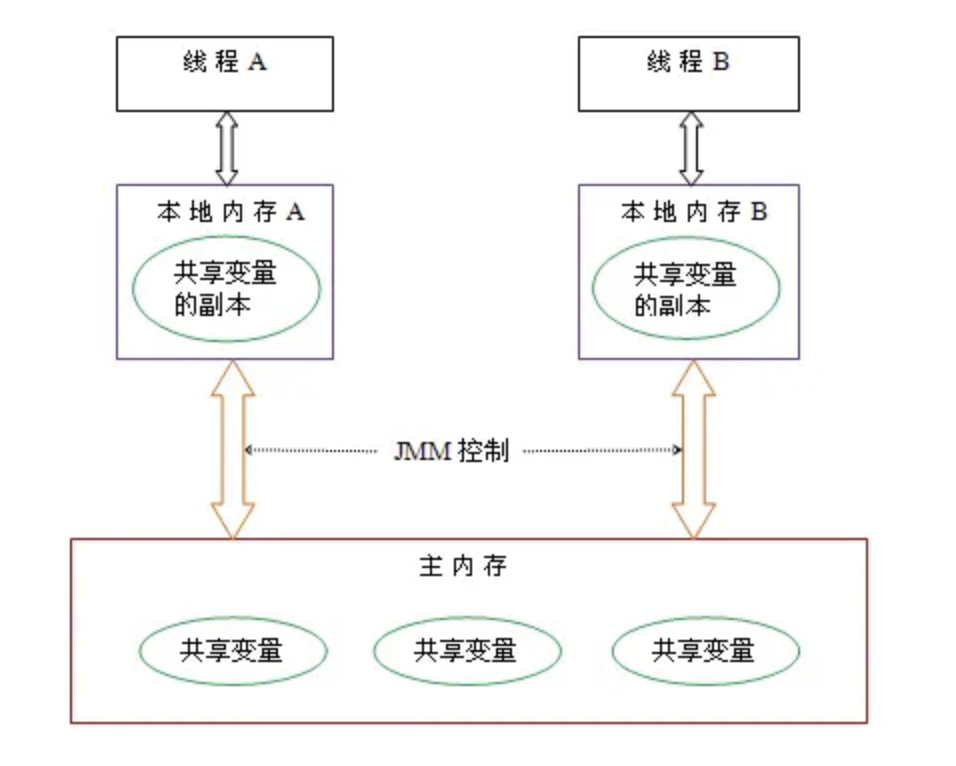


Java 对象模型



Jvm 内存模型

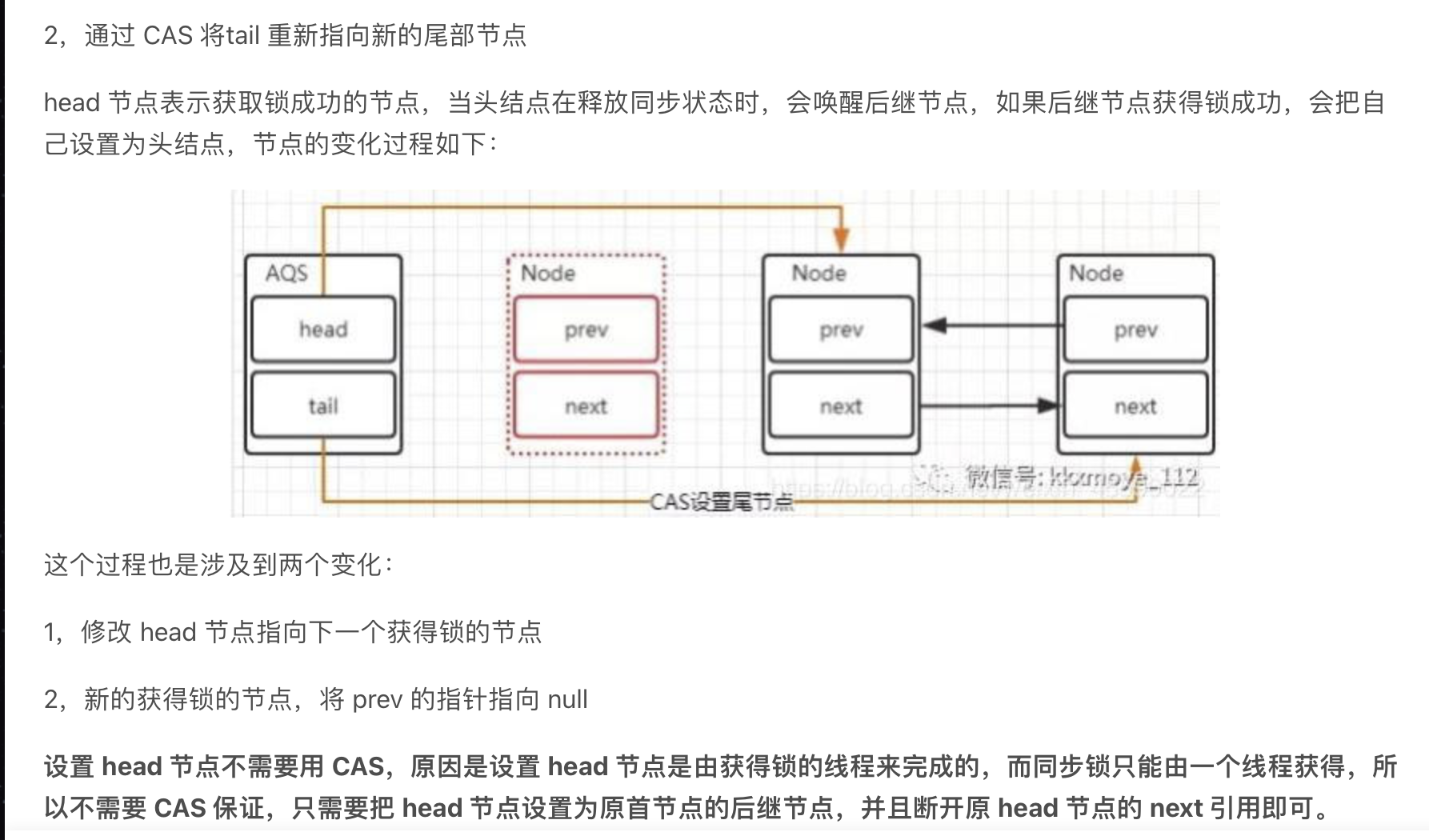
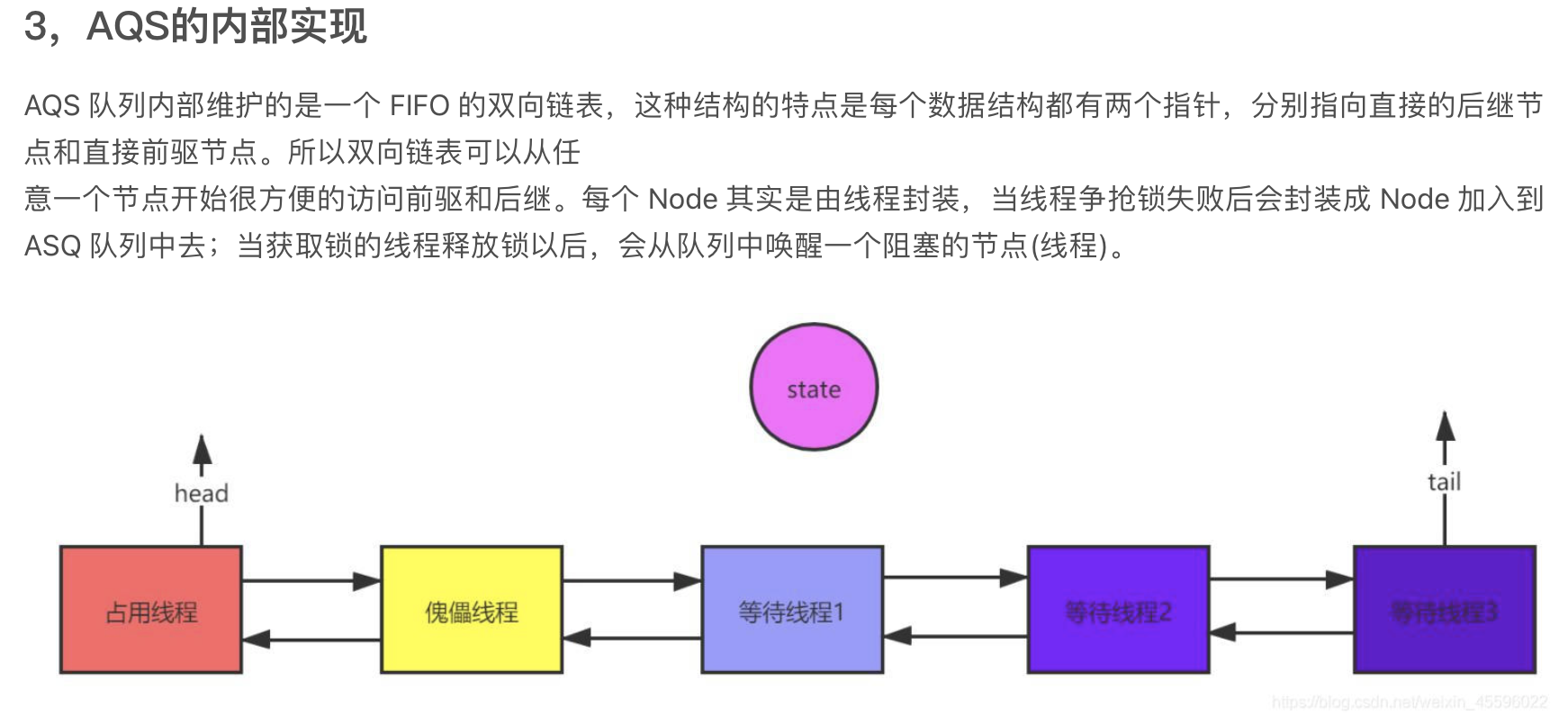
Java 内存模型JMM



AQS（AbstractQueuedSynchronizer） **ReenTrantLock实现原理**

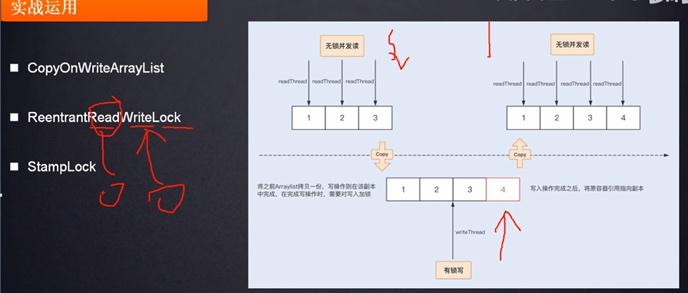
**https://blog.csdn.net/weixin\_45596022/article/details/113817683**

ReentrantLock：表示[重入锁](https://so.csdn.net/so/search?q=%E9%87%8D%E5%85%A5%E9%94%81&spm=1001.2101.3001.7020)，是一个轻量级锁，重入锁指的是线程在获得锁之后，再次获取该锁不需要阻塞，而是直接关联一次计数器增加重入次数。



ReentrantReadWriteLock：重入读写锁，可以保证多个线程可以同时读，所以在读操作远大于写操作的时候，读写锁就非常有用了。

CopyOnWriteArrayList 高并发读



**串行在时间上不可以发生重叠，前一个任务没有搞定，下一个任务只能等待。**

**并行在时间上是重叠的，两个任务在同一时刻互不干扰的同时进行。**

[**并发**](https://so.csdn.net/so/search?q=%E5%B9%B6%E5%8F%91&spm=1001.2101.3001.7020)**允许两个任务之间是彼此干扰的。统一时间点，只有一个任务运行，交替执行。**

常用包：java.lang系统的基础类

java.io输入输出有关的类

java.util系统辅助类，特别是集合类

java.sql数据库操作的类

ThreadLocal

一种线程隔离机制，多线程环境下，对共享变量访问的安全性，一般情况下是加锁，但性能会下降，ThreadLocal用一种空间换时间，在每个线程里面有一个容器来存储共享变量的一个副本，每个线程只对自己的共享变量副本来更新操作。

原理：Thread类里有内部类ThreadLocalMap（键值对）用来存储共享变量 的副本，后续变更也是在这里面。

内存泄露：不再会被使用的对象和变量占用的内存不能被回收。

原因：key不是字符串，是ThreadLocal类型（封装），而且是弱引用

所以垃圾回收，key清除，只剩下null，但value不会回收

解决方法：不用时用remove方法移除。

强引用：默认就是强引用，任何一个对象的赋值操作就产生了对这个对象的强引用。 只要有强引用存在，被引用的对象就不会被垃圾回收。

软引用： 只有在内存不足的情况下，被引用的对象才会被回收。

弱引用： 只要垃圾回收执行，就会被回收，而不管是否内存不足。

虚引用： 跟踪垃圾回收器收集对象的活动，在GC的过程中，如果发现有PhantomReference（虚引用），GC则会将引用放到ReferenceQueue中。

线程复用原理：线程池可以把线程和任务进行解耦，线程归线程，任务归任务，摆脱了之前通过 Thread 创建线程时的一个线程必须对应一个任务的限制。在线程池中，同一个线程可以从BlockingQueue 中不断提取新任务来执行，其核心原理在于线程池对 Thread 进行了封装，并不是每次执行任务都会调用 Thread.start() 来创建新线程，而是让每个线程去执行一个“循环任务”，在这个“循环任务”中，不停地检查是否还有任务等待被执行，如果有则直接去执行这个任务，也就是调用任务的 run 方法，把 run 方法当作和普通方法一样的地位去调用，相当于把每个任务的 run() 方法串联了起来，所以线程数量并不增加。

MySQL（**关系型数据库）**

**大部分关系型数据库支持事务四大特性ACID**

**默认端口号：3306**

**Mysql SQL server oracle哦若可 SQLite死磕来特**

**create table t2(**

**a int primary key,**

**b int,**

**c int,**

**d int,**

**e varchar(20),**

**) engine=MyISAM; InnoDB**

**insert into t2 values(4,3,2,1,'d')**

**create index ………on t2(b,c,d);**

**存储引擎：**

**MyISAM: 5.5之前默认 基于b+树**

**读写效率高，占用数据空间小，有单独的物理索引文件，适合读多写少**

**不支持事务，不支持行级锁，所以添加修改时，会执行锁表结构，不支持外键**

**InnoDB: 5.5之后默认 页结构16kb**

基于b+树 **（插入时按主键自动排序）**

支持事务，外键，崩溃修复，自增列，控制事务提交和回 滚

读写效率差，占用数据空间大

MEMORY（慢么瑞）：

不支持事务不支持外键 支持hash索引和b树索引

Show engines 查数据库所有存储引擎

表锁定只用于防止其它客户端进行不正当地读取和写入

**事务**

**逻辑上的一组操作，要么都执行，要么都不执行。保证多个对数据库的操作（也就是 SQL 语句）构成一个逻辑上的整体。**

**ACID**

**原子性:** **一个事务中的所有操作，要么全部完成，要么全部不完成，不会结**

**束在中间某个环节。**

**一致性:** **在事务开始之前和事务结束以后，数据库的完整性没有被破坏。**

**隔离性: 数据库允许多个并发事务同时对其数据进行读写和修改的能力，隔离**

**性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。**

**持久性: 事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢**

**失。**

**数据**

**增加insert 查看select 删除delete 改update**

**数据库和表**

**创建create 修改alter 删除drop**

索引

一种用于快速查询和检索数据的数据结构。常见的索引结构有: B 树， B+树和 Hash。

主键索引：索引的key是主键，叶节点 data 域保存了完整的数据记录。

二级索引（辅助索引）: **叶子节点存储的数据是主键**

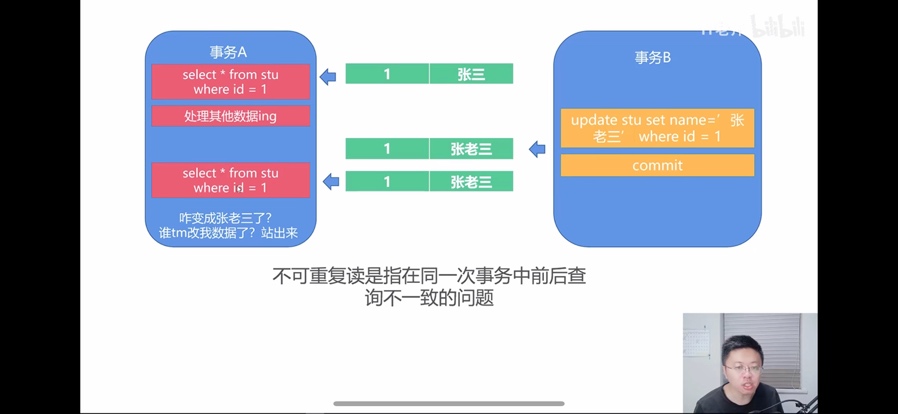
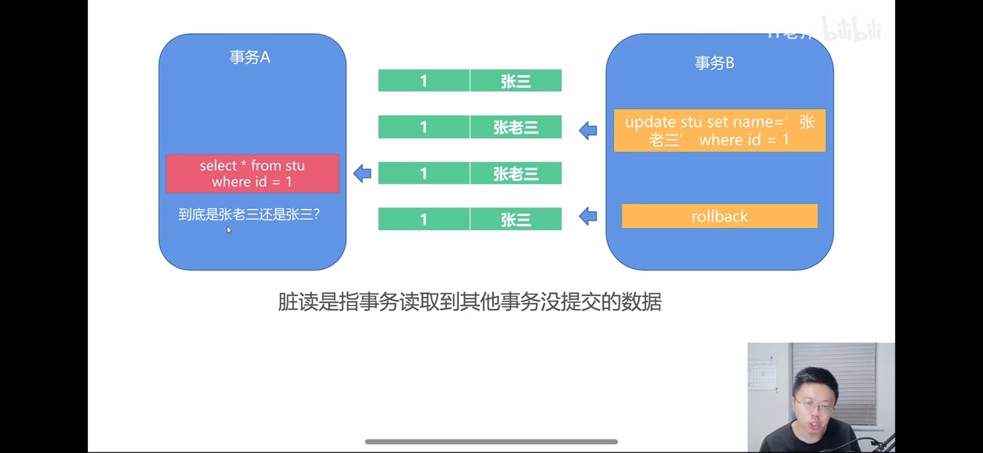
聚集索引：索引结构和数据一起存放的索引。

非聚集索引：索引结构和数据分开存放的索引。

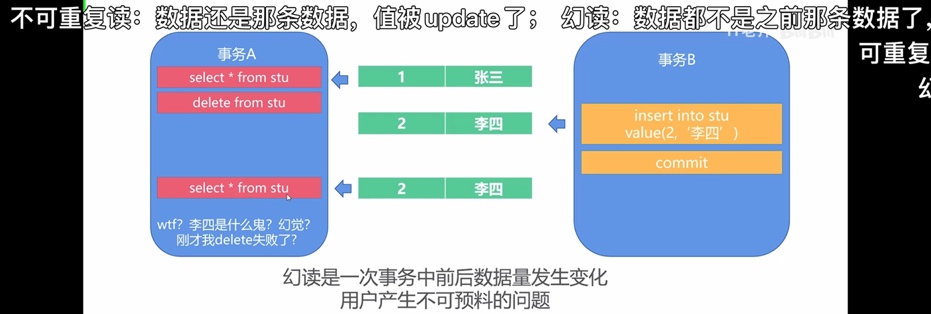
MyISAM，B+树叶子节点的data域存放的是数据记录的地址，key不一定是主键，这被称为非聚簇索引。

InnoDB，B+树的叶节点的data域保存了完整的数据记录，key不一定是主键（在 MySQL 的 InnoDB 的表中，当没有显示的指定表的主键时，InnoDB 会自动先检查表中是否有唯一索引且不允许存在 null 值的字段，如果有，则选择该字段为默认的主键，否则 InnoDB 将会自动创建一个 6Byte 的自增主键），这被称为聚簇索引。

并发事务带来的问题

****

一般是更新

****一般是新增和修改数据

事务隔离级别

READ-UNCOMMITTED(读取未提交) READ-COMMITTED(读取已提交)

REPEATABLE-READ(可重复读) SERIALIZABLE(可串行化)

MVCC 多版本并发控制

MySQL 的隔离级别基于锁和 MVCC 机制共同实现的。

SERIALIZABLE 隔离级别，是通过锁来实现的

多版本并发控制 基于RC，RR隔离级别下工作

为了在读取数据时不加锁来提高读取效率和并发性的一种手段。

在读已提交和可重复读隔离级别下的快照读，都是基于MVCC实现的

快照读（普通读）不会对记录进行加锁:普通select查询sql语句

当前读（锁定读）对当前读取的数据进行加锁:

insert update delete

select … lock in share mode select … for update

实现依赖于隐藏字段、Read View、undo log版本链。

隐藏字段：trx\_id事务id

Roll\_ptr指针信息，指向上一版本变化时的数据。

Read View数据结构包含四个字段

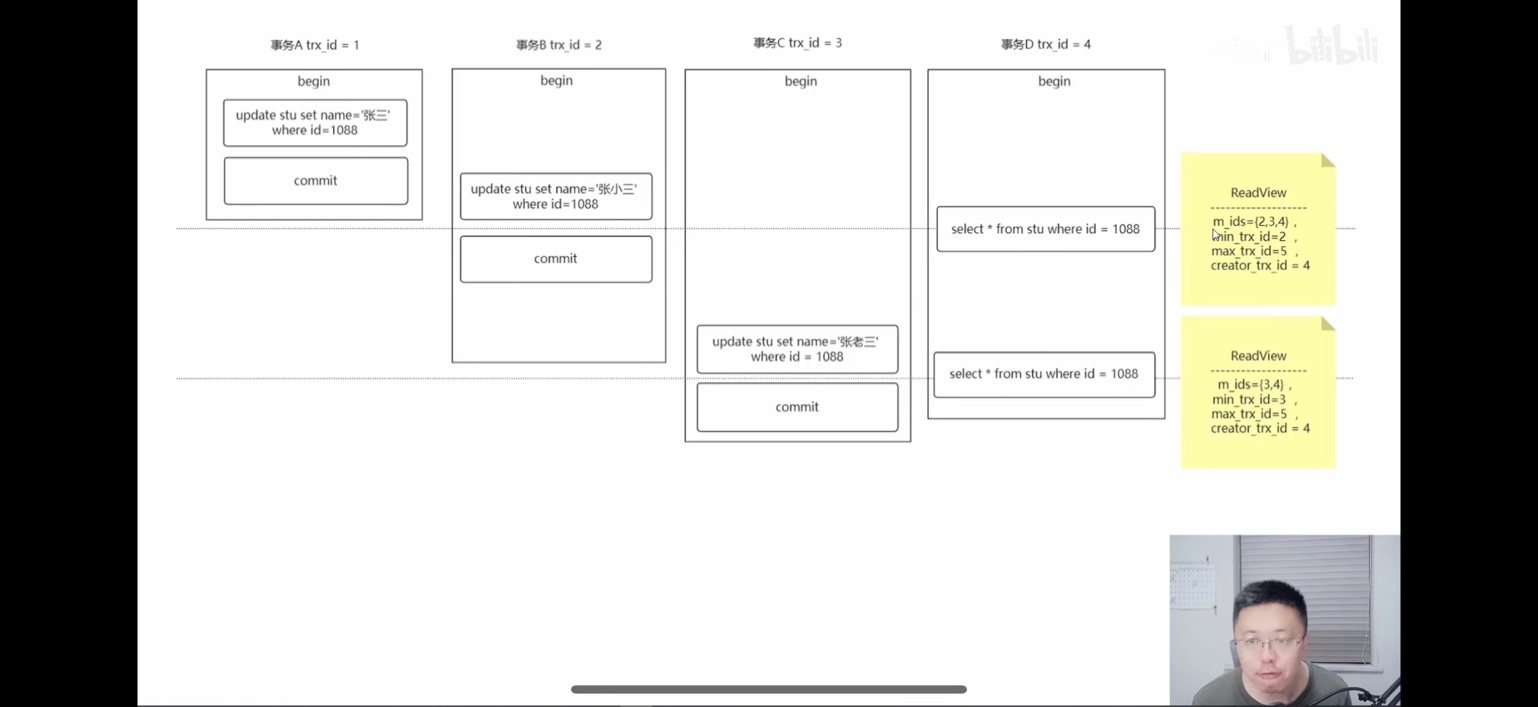
m\_ids：未提交的活跃事务 ID 列表

min\_trx\_id：最小活跃事务编号

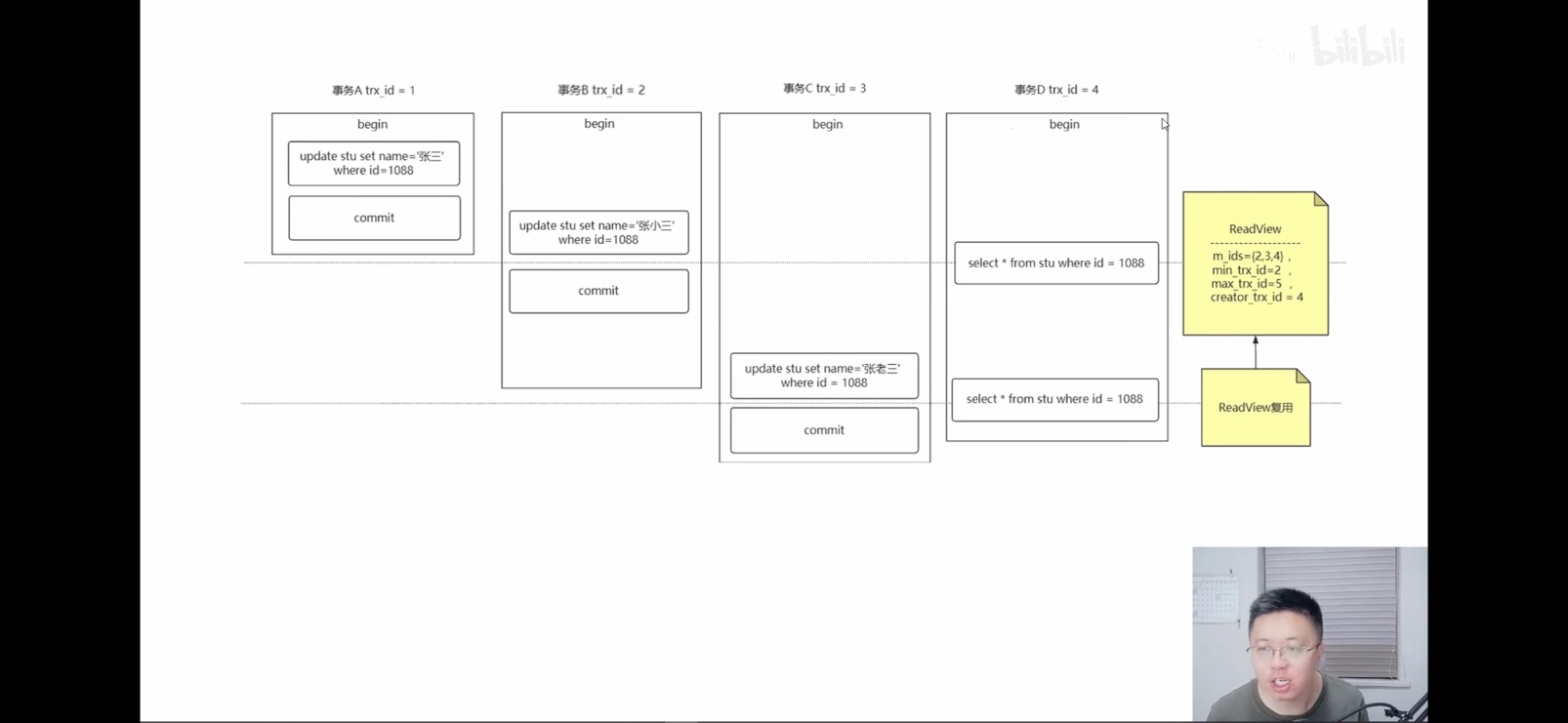
max\_trx\_id：最大活跃事务编号+1

creator\_trx\_id：创建该 Read View 的事务 ID

RC 每次执行快照读时生成Read View



RR 仅在第一次执行快照读时生成Read View，后续复用



RR下 连续多次快照读Read View产生复用，无幻读

两次快照读之间生成当前读，Read View重新产生，发生幻读。

java架构师路线图：

<https://www.processon.com/view/link/617be37ee0b34d7894fcf545>

springBoot和springCloud：

1. spring boot使用了默认大于配置的理念，集成了快速开发的spring多个插件，同时自动过滤不需要配置的多余的插件，简化了项目的开发配置流程，一定程度上取消xml配置，是一套快速配置开发的脚手架，能快速开发单个微服务。

2. spring cloud大部分的功能插件都是基于springBoot去实现的，springCloud关注于全局的微服务整合和管理，将多个springBoot单体微服务进行整合以及管理；springCloud依赖于springBoot开发，而springBoot可以独立开发。

mac上安装&配置&使用git

https://blog.csdn.net/ZeroRm/article/details/109101918?spm=1001.2101.3001.6661.1&utm\_medium=distribute.pc\_relevant\_t0.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-1-109101918-blog-128067901.pc\_relevant\_default&depth\_1-utm\_source=distribute.pc\_relevant\_t0.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-1-109101918-blog-128067901.pc\_relevant\_default&utm\_relevant\_index=1