

前言

关于线程安全一提到可能就是加锁，在面试中也是面试官百问不厌的考察点，往往能看出面试者的基本功和是否对线程安全有自己的思考。

那锁本身是怎么去实现的呢？又有哪些加锁的方式呢？

我今天就简单聊一下乐观锁和悲观锁，他们对应的实现 CAS，Synchronized，ReentrantLock

正文

一个120斤一身黑的小伙子走了进来，看到他微微发福的面容，看来是最近疫情伙食好运动少的结果，他难道就是今天的面试官渣渣丙？



等等难道是他？前几天刷B站看到的不会是他吧！！

[我的粉丝会看到](#)[新访客会看到](#)



今天帮公司面试了个要25K的Java程序员，看看我都问他些什么问题

12.1万 345 3-2

别问，问就是胖了，没刮胡子，长痘痘了哈哈，播放能上5000就录制下一个。

是的我已经开始把面试系列做成视频了，以后会有各种级别的面试，从大学生到阿里P7+的面试，还有阿里，拼多多，美团，字节风格的面试我也都约好人了，就差时间了，大家可以去B站搜：三太子敖丙观看

我也不多跟你BB了，我们直接开始好不好，你能跟我聊一下CAS么？

CAS（Compare And Swap 比较并且替换）是乐观锁的一种实现方式，是一种轻量级锁，JUC 中很多工具类的实现就是基于 CAS 的。

CAS 是怎么实现线程安全的？

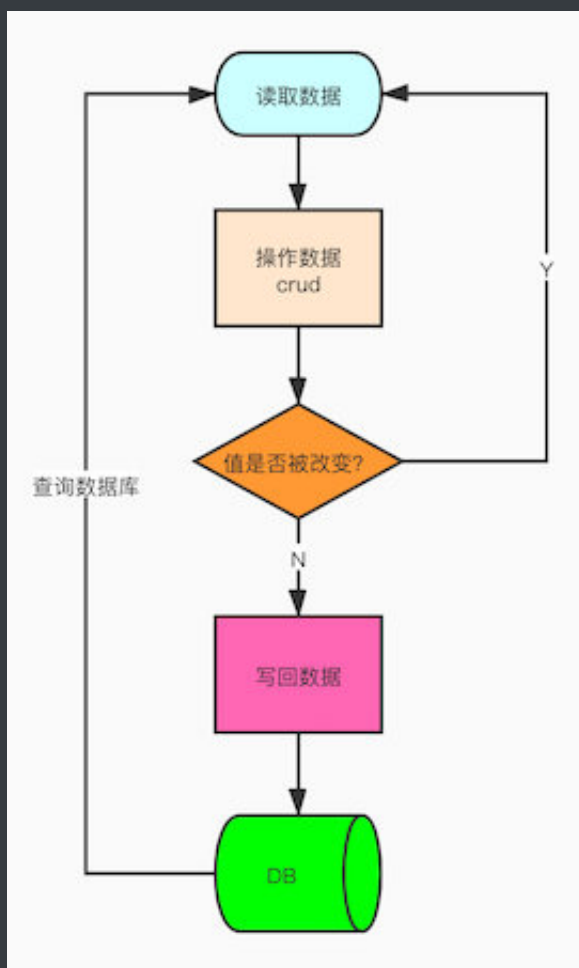
线程在读取数据时不进行加锁，在准备写回数据时，先去查询原值，操作的时候比较原值是否修改，若未被其他线程修改则写回，若已被修改，则重新执行读取流程。

举个栗子：现在一个线程要修改数据库的name，修改前我会先去数据库查name的值，发现name=“帅丙”，拿到值了，我们准备修改成name=“三歪”，在修改之前我们判断一下，原来的name是不是等于“帅丙”，如果被其他线程修改就会发现name不等于“帅丙”，我们就不进行操作，如果原来的值还是帅丙，我们就把name修改为“三歪”，至此，一个流程就结束了。

有点懵？理一下停下来理一下思路。

Tip：比较+更新 整体是一个原子操作，当然这个流程还是有问题的，我下面会提到。

他是乐观锁的一种实现，就是说认为数据总是不会被更改，我是乐观的仔，每次我都觉得你不会渣我，差不多是这个意思。



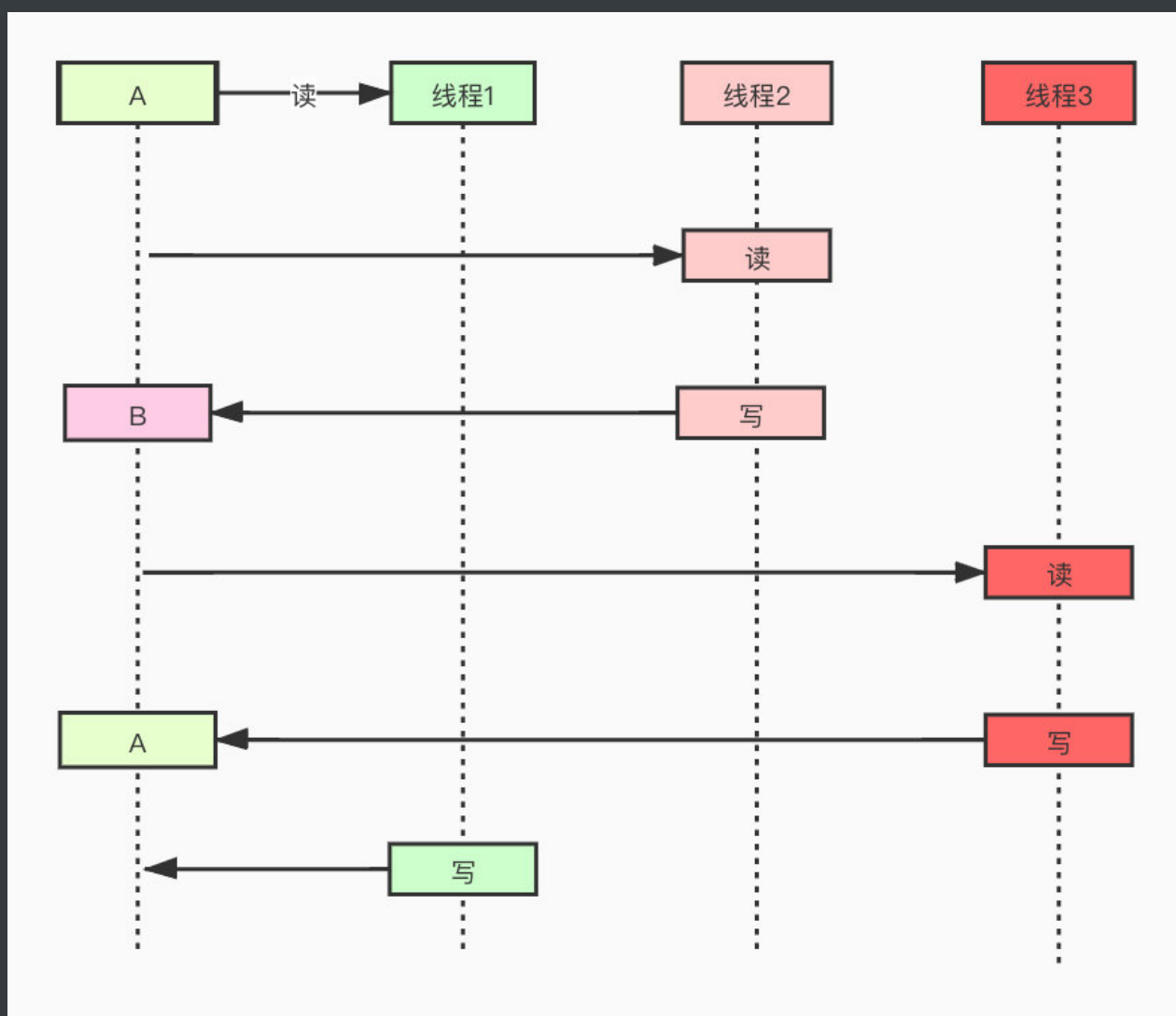
你这个栗子不错，他存在什么问题呢？

有，当然是有问题的，我也刚好想提到。

你们看图发现没，要是结果一直就一直循环了，CUP开销是个问题，还有ABA问题和只能保证一个共享变量原子操作的问题。

你能分别介绍一下么？

好的，我先介绍一下ABA这个问题，直接口述可能有点抽象，我画图解释一下：



看到问题所在没，我说一下顺序：

1. 线程1读取了数据A
2. 线程2读取了数据A
3. 线程2通过CAS比较，发现值是A没错，可以把数据A改成数据B
4. 线程3读取了数据B
5. 线程3通过CAS比较，发现数据是B没错，可以把数据B改成了数据A
6. 线程1通过CAS比较，发现数据还是A没变，就写成了自己要改的值

懂了么，我尽可能的幼儿园化了，在这个过程中任何线程都没做错什么，但是值被改变了，线程1却没有办法发现，其实这样的情况出现对结果本身是没有什么影响的，但是我们还是要防范，怎么防范我下面会提到。

循环时间长开销大的问题：

是因为CAS操作长时间不成功的话，会导致一直自旋，相当于死循环了，CPU的压力会很大。

只能保证一个共享变量的原子操作：

CAS操作单个共享变量的时候可以保证原子的操作，多个变量就不行了，JDK 5之后 AtomicReference 可以用来保证对象之间的原子性，就可以把多个对象放入CAS中操作。

我还记得你之前说在JUC包下的原子类也是通过这个实现的，能举个栗子么？

那我就拿AtomicInteger举例，他的自增函数incrementAndGet () 就是这样实现的，其中就有大量循环判断的过程，直到符合条件才成功。

```
public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5, var5: var5 + var4));

    return var5;
}
```

大概意思就是循环判断给定偏移量是否等于内存中的偏移量，直到成功才退出，看到do while的循环没。

乐观锁在项目开发中的实践，有么？

有的就比如我们在很多订单表，流水表，为了防止并发问题，就会加入CAS的校验过程，保证了线程的安全，但是看场景使用，并不是适用所有场景，他的优点缺点都很明显。

那开发过程中ABA你们是怎么保证的？

加标志位，例如搞个自增的字段，操作一次就自增加一，或者搞个时间戳，比较时间戳的值。

举个栗子：现在我们去要求操作数据库，根据CAS的原则我们本来只需要查询原本的值就好了，现在我们一同查出他的标志位版本字段version。

之前不能防止ABA的正常修改：

```
update table set value = newValue where value = #{oldValue}
//oldValue就是我们执行前查询出来的值
```

带版本号能防止ABA的修改：

```
update table set value = newValue , vision = vision + 1 where value = #
{oldValue} and vision = #{vision}
// 判断原来的值和版本号是否匹配，中间有别的线程修改，值可能相等，但是版本号100%不一样
```

除了版本号，像什么时间戳，还有JUC工具包里面也提供了这样的类，想要扩展的小伙伴可以去了解一下。

聊一下悲观锁？

悲观锁从宏观的角度讲就是，他是个渣男，你认为他每次都会渣你，所以你每次都提防着他。

我们先聊下JVM层面的synchronized：

synchronized加锁，synchronized 是最常用的线程同步手段之一，上面提到的CAS是乐观锁的实现，synchronized就是悲观锁了。

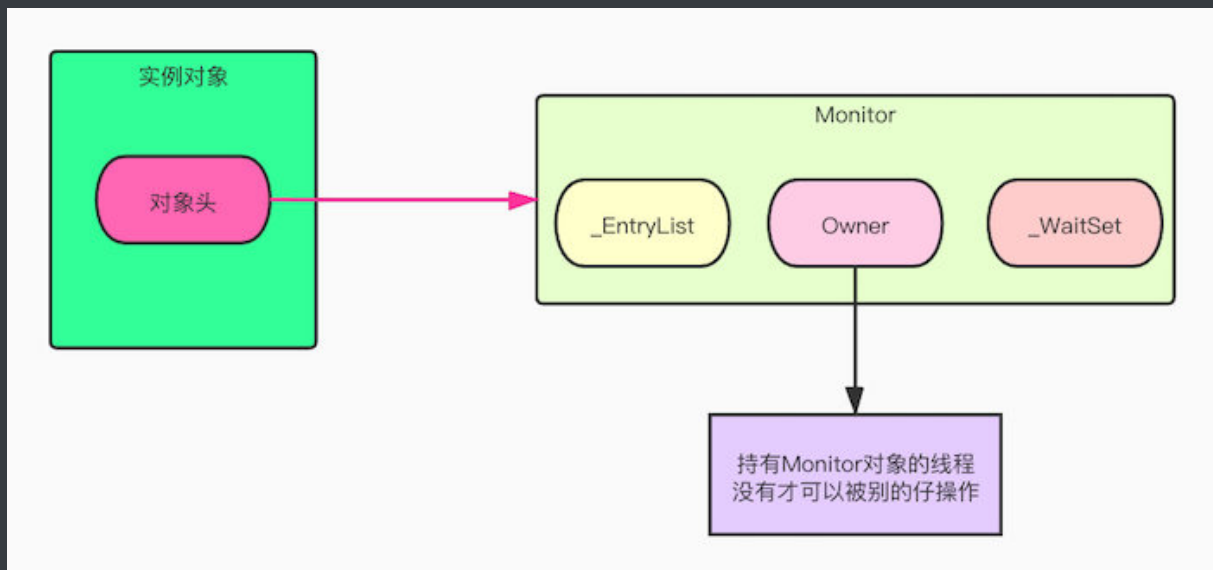
它是如何保证同一时刻只有一个线程可以进入临界区呢？

synchronized，代表这个方法加锁，相当于不管哪一个线程（例如线程A），运行到这个方法时,都要检查有没有其它线程B（或者C、D等）正在用这个方法(或者该类的其他同步方法)，有的话要等正在使用synchronized方法的线程B（或者C、D）运行完这个方法后再运行此线程A，没有的话，锁定调用者，然后直接运行。

我分别从他对对象、方法和代码块三方面加锁，去介绍他怎么保证线程安全的：

- synchronized 对对象进行加锁，在 JVM 中，对象在内存中分为三块区域：**对象头**（Header）、实例数据（Instance Data）和**对齐填充**（Padding）。
 - **对象头**：我们以Hotspot虚拟机为例，Hotspot的对象头主要包括两部分数据：Mark Word（标记字段）、Klass Pointer（类型指针）。
 - **Mark Word**：默认存储对象的HashCode，分代年龄和锁标志位信息。它会根据对象的状态复用自己的存储空间，也就是说在运行期间Mark Word里存储的数据会随着锁标志位的变化而变化。
 - **Klass Point**：对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。

你可以看到在对象头中保存了锁标志位和指向 monitor 对象的起始地址，如下图所示，右侧就是对象对应的 Monitor 对象。



当 Monitor 被某个线程持有后，就会处于锁定状态，如图中的 Owner 部分，会指向持有 Monitor 对象的线程。

另外 Monitor 中还有两个队列分别是EntryList和WaitList，主要是用来存放进入及等待获取锁的线程。

如果线程进入，则得到当前对象锁，那么别的线程在该类所有对象上的任何操作都不能进行。

在对象级使用锁通常是一种比较粗糙的方法，为什么要将整个对象都上锁，而不允许其他线程短暂地使用对象中其他同步方法来访问共享资源？

如果一个对象拥有多个资源，就不需要只为了让一个线程使用其中一部分资源，就将所有线程都锁在外面。

由于每个对象都有锁，可以如下所示使用虚拟对象来上锁：

```
class FineGrainLock{
    MyMemberClass x,y;
    Object xlock = new Object(), ylock = new Object();
    public void foo(){
        synchronized(xlock){
            //accessxhere
        }
        //dosomethinghere-butdon't use shared resources
        synchronized(ylock){
            //accessyhere
        }
    }
    public void bar(){
        synchronized(this){
            //accessbothxandyhere
        }
    }
}
```

```

    }
    //dosomethinghere-butdon'tusessharedresources
}
}

```

- synchronized 应用在方法上时，在字节码中是通过方法的 ACC_SYNCHRONIZED 标志来实现的。我反编译了一小段代码，我们可以看一下我加锁了一个方法，在字节码长啥样，**flags**字段瞩目：

```

synchronized void test();
  descriptor: ()V
  flags: ACC_SYNCHRONIZED
  Code:
    stack=0, locals=1, args_size=1
      0: return
  LineNumberTable:
    line 7: 0
  LocalVariableTable:
    Start   Length  Slot  Name   Signature
      0       1      0   this   Ljvm/ClassCompile;

```

反正其他线程进这个方法就看看是否有这个标志位，有就代表有别的仔拥有了他，你就别碰了。

- synchronized 应用在同步块上时，在字节码中是通过 monitorenter 和 monitorexit 实现的。每个对象都会与一个monitor相关联，当某个monitor被拥有之后就会被锁住，当线程执行到 monitorenter指令时，就会去尝试获得对应的monitor。

步骤如下：

1. 每个monitor维护着一个记录着拥有次数的计数器。未被拥有的monitor的该计数器为0，当一个线程获得monitor（执行monitorenter）后，该计数器自增变为 1 。
 - 当同一个线程再次获得该monitor的时候，计数器再次自增；
 - 当不同线程想要获得该monitor的时候，就会被阻塞。
2. 当同一个线程释放 monitor（执行monitorexit指令）的时候，计数器再自减。
当计数器为0的时候，monitor将被释放，其他线程便可以获得monitor。

同样看一下反编译后的一段锁定代码块的结果：

```

public void syncTask();
  descriptor: ()V
  flags: ACC_PUBLIC
  Code:
    stack=3, locals=3, args_size=1

```



```

0: aload_0
1: dup
2: astore_1
3: monitorenter //注意此处，进入同步方法
4: aload_0
5: dup
6: getfield      #2           // Field i:I
9: iconst_1
10: iadd
11: putfield      #2           // Field i:I
14: aload_1
15: monitorexit //注意此处，退出同步方法
16: goto          24
19: astore_2
20: aload_1
21: monitorexit //注意此处，退出同步方法
22: aload_2
23: athrow
24: return
Exception table:
//省略其他字节码.....


```

小结：

同步方法和同步代码块底层都是通过monitor来实现同步的。

两者的区别：同步方式是通过方法中的access_flags中设置ACC_SYNCHRONIZED标志来实现，同步代码块是通过monitorenter和monitorexit来实现。

我们知道了每个对象都与一个monitor相关联，而monitor可以被线程拥有或释放。

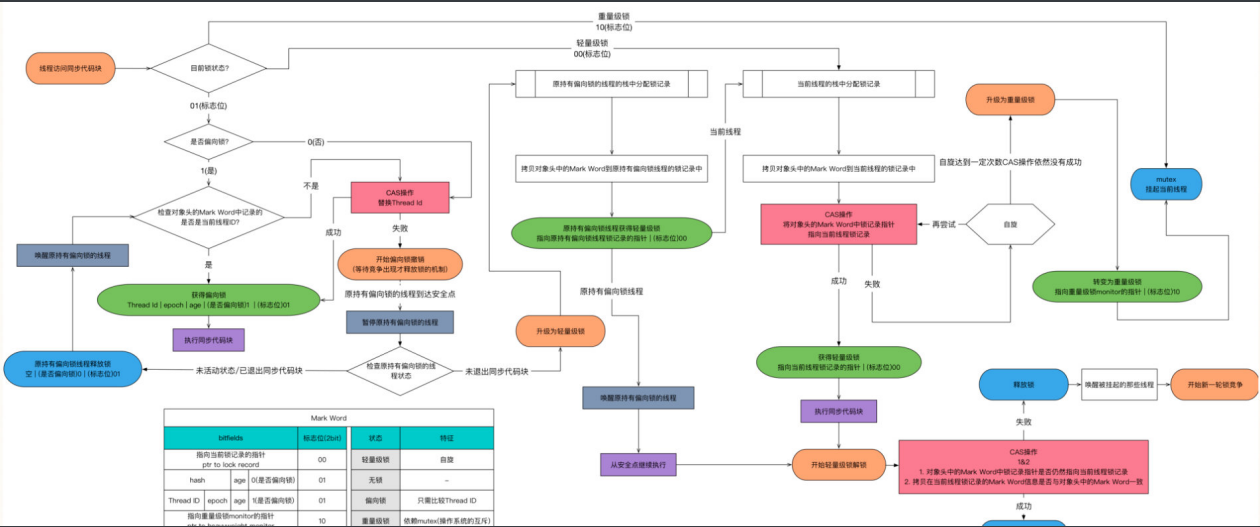
，小伙子我只能说，你确实有点东西，以前我们一直锁synchronized是重量级的锁，为啥现在都不提了？

在多线程并发编程中 synchronized 一直是元老级角色，很多人都会称呼它为重量级锁。

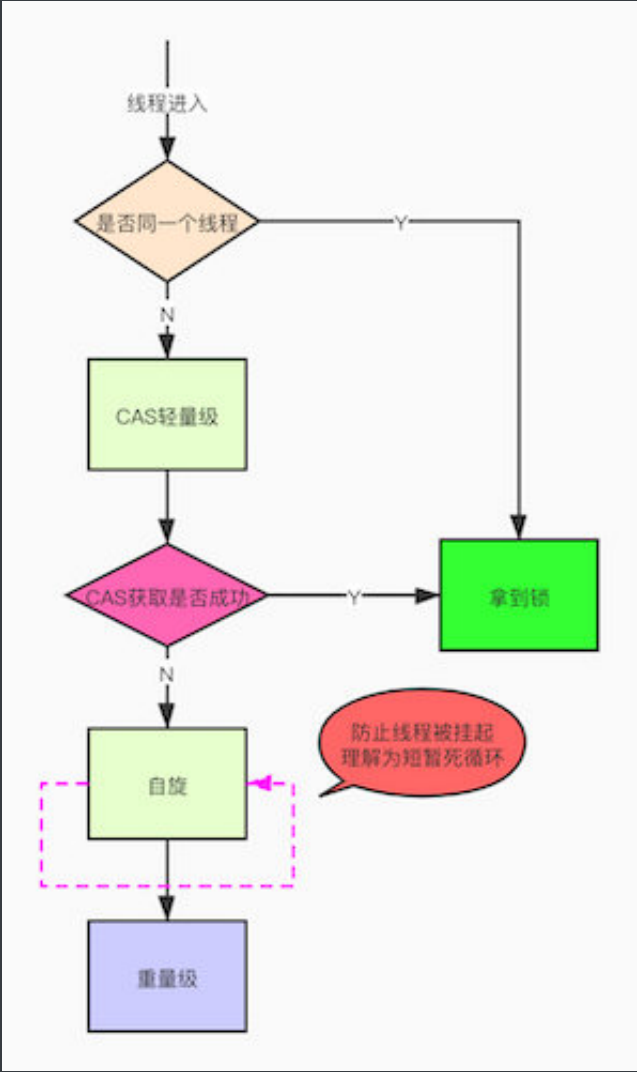
但是，随着 Java SE 1.6 对 synchronized 进行了各种优化之后，有些情况下它就并不那么重，Java SE 1.6 中为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁。

针对 synchronized 获取锁的方式，JVM 使用了锁升级的优化方式，就是先使用偏向锁优先同一线程然后再再次获取锁，如果失败，就升级为 CAS 轻量级锁，如果失败就会短暂自旋，防止线程被系统挂起。最后如果以上都失败就升级为重量级锁。

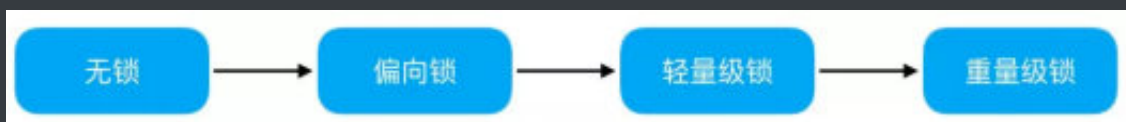
Tip: 本来锁升级的过程我是搞了个贼详细贼复杂的图，但是我发现不便于理解，我就幼儿园化了，所以就有了个简单版本的，先看下复杂版本的：



幼儿园版本：



看到这你如果还想白嫖，我劝你善良，万水千山总是情，不要白嫖行不行？点个赞再走哈哈。



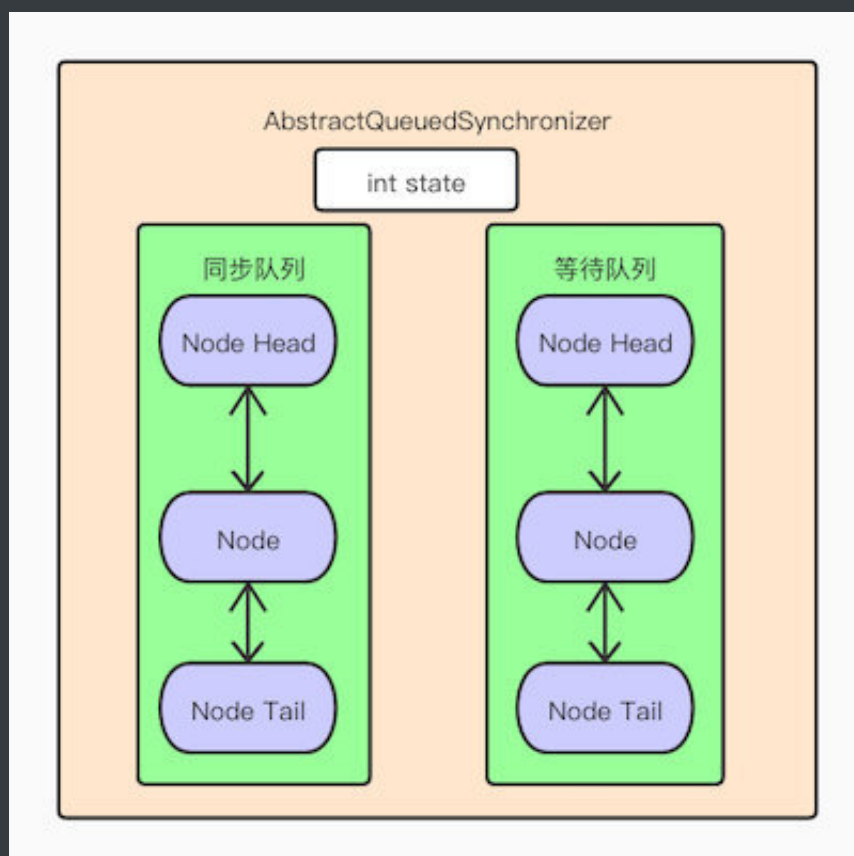
对于锁只能升级，不能降级。

还有其他的同步手段么？

ReentrantLock但是在介绍这玩意之前，我觉得我有必要先介绍 AQS (AbstractQueuedSynchronizer) 。

AQS：也就是队列同步器，这是实现 ReentrantLock 的基础。

AQS 有一个 state 标记位，值为1 时表示有线程占用，其他线程需要进入到同步队列等待，同步队列是一个双向链表。

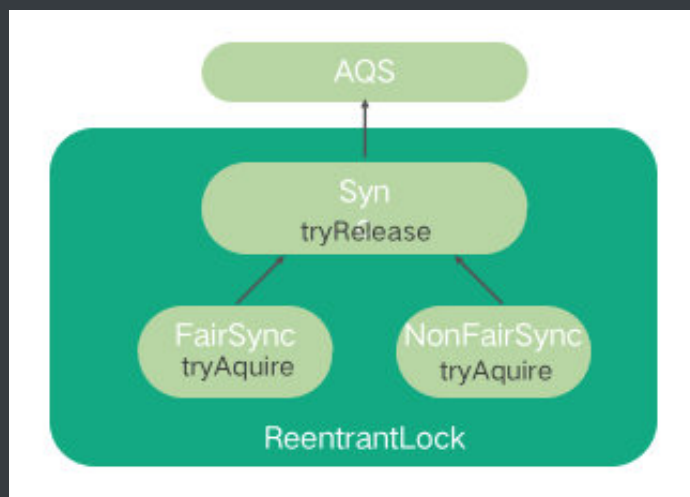


当获得锁的线程需要等待某个条件时，会进入 condition 的等待队列，等待队列可以有多个。

当 condition 条件满足时，线程会从等待队列重新进入同步队列进行获取锁的竞争。

ReentrantLock 就是基于 AQS 实现的，如下图所示，ReentrantLock 内部有公平锁和非公平锁两种实现，差别就在于新来的线程是否比已经在同步队列中的等待线程更早获得锁。

和 ReentrantLock 实现方式类似，Semaphore 也是基于 AQS 的，差别在于 ReentrantLock 是独占锁，Semaphore 是共享锁。



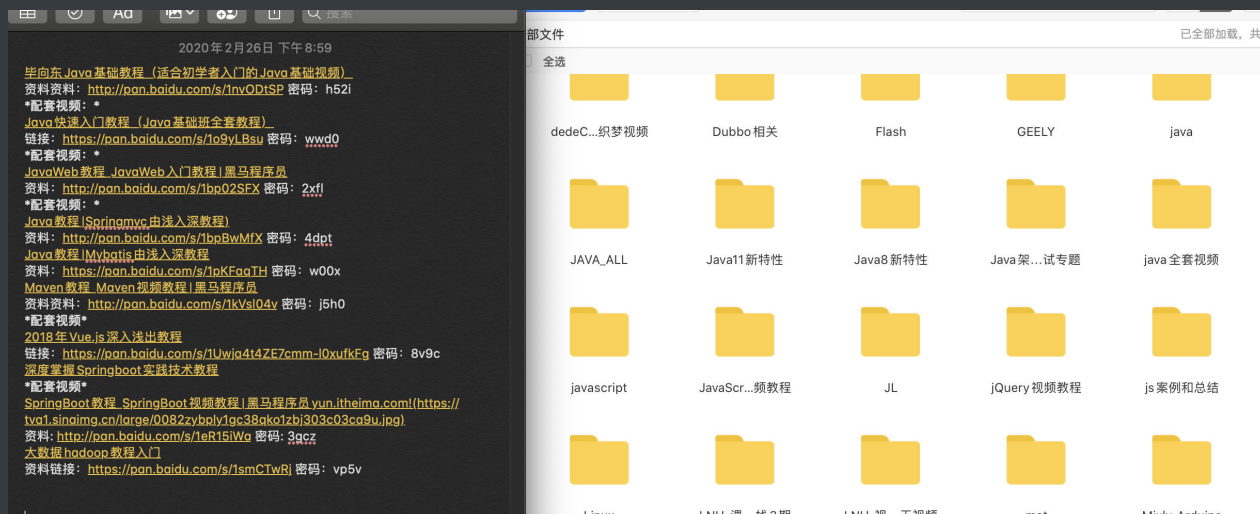
从图中可以看到，ReentrantLock里面有一个内部类Sync，Sync继承AQS（AbstractQueuedSynchronizer），添加锁和释放锁的大部分操作实际上都是在Sync中实现的。

它有公平锁FairSync和非公平锁NonfairSync两个子类。

ReentrantLock默认使用非公平锁，也可以通过构造器来显示的指定使用公平锁。

相关资料

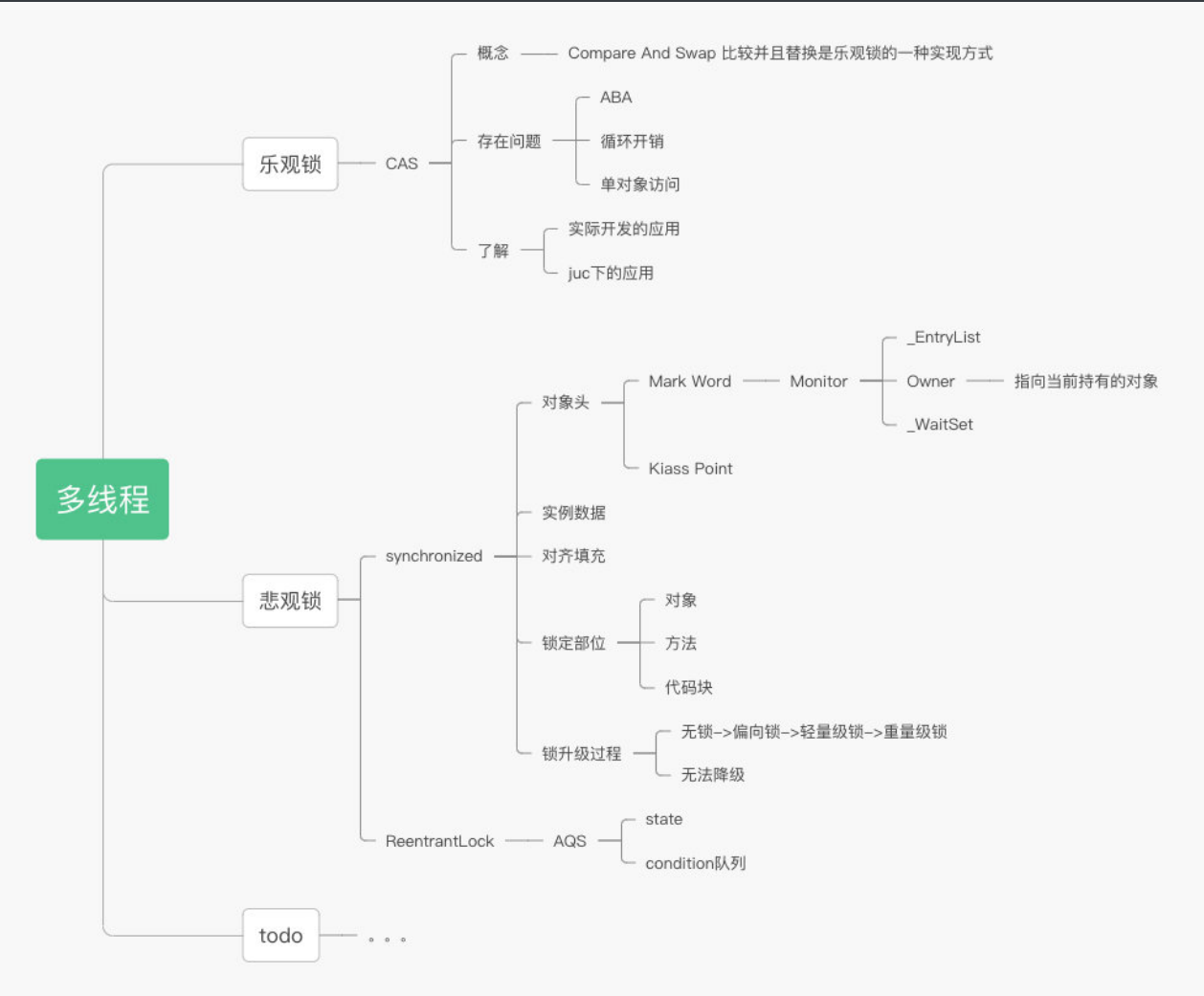
Tip：本来这一栏有很多我准备的资料的，但是都是外链，或者不合适的分享方式，博客的运营小姐姐提醒了我，所以大家去公众号回复【资料】好了。



技术总结

锁其实有很多，我这里只是简单的介绍了一下乐观锁和悲观锁，后面还会有，自旋锁，自适应自旋，公平锁，非公平锁，可重入（文中提到的都是可重入），不可重入锁，共享锁，排他锁等。

多去了解他们的用法，多去深究他们的原理以及实现，我后面会持续更新多线程方面的知识点。



Tip：这是我新的技术总结方式，以后会慢慢完善，如果被博客平台二压公众号回复【多线程】获取。

参考：《Java高并发编程》、《美团技术团队锁的思考》、《拉钩张雷java32个考点》

絮叨

我不断的尝试新的文章风格，我也把絮叨环节放到了最后就是给大家一个好的阅读体验，有建议随时提哟，新的技术总结方式如何？

还记得我帮公司内推么，我收到300封简历，但是我放到系统的只有**13**份，不是我想吐槽大家，是真的得用心点啊，叫我内推发个邮件，简历都不发什么鬼，简历总共才100个字又是什么鬼。。。

周末出一期视频说一下简历的问题，真的为你们春招担心啊仔。。。

我是敖丙，一个在互联网苟且偷生的工具人。

创作不易，不想被白嫖，各位的「三连」就是丙丙创作的最大动力，我们下次见！

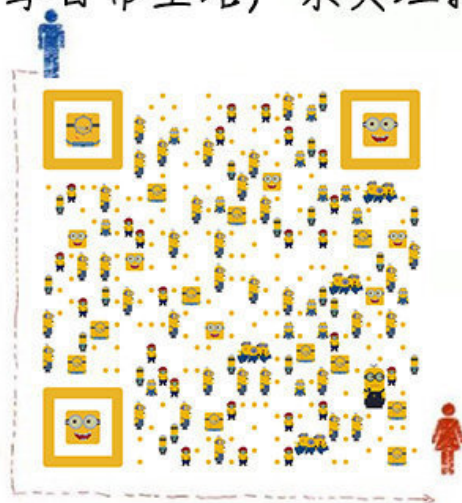
资料/面试/联系我/公众号



关注后回复【资料】领整理的互联网大厂技术栈资料
关注后回复【面试】领整理好的面试线路相关文章
关注后回复【加群】有我个人联系方式和大佬云集的技术交流群，有问题一起交流。

如果觉得丙丙的文章确实有点东西，求个赞就好啦
对在博客苟且偷生敖丙来说真的 非常有用！！

文章第一时间我都会发在公众号，微信搜索【三太子敖丙】
我还会在公众号分享日常生活，来关注我吧。



微信搜一搜



三太子敖丙

别爱我，没结果