

原理篇

指令级并行原理

1. 名词

Clock Cycle Time

主频的概念大家接触的比较多，而 CPU 的 Clock Cycle Time（时钟周期时间），等于主频的倒数，意思是 CPU 能够识别的最小时间单位，比如说 4G 主频的 CPU 的 Clock Cycle Time 就是 0.25 ns，作为对比，我们墙上挂钟的 Cycle Time 是 1s

例如，运行一条加法指令一般需要一个时钟周期时间

CPI

有的指令需要更多的时钟周期时间，所以引出了 CPI（Cycles Per Instruction）指令平均时钟周期数

IPC

IPC（Instruction Per Clock Cycle）即 CPI 的倒数，表示每个时钟周期能够运行的指令数

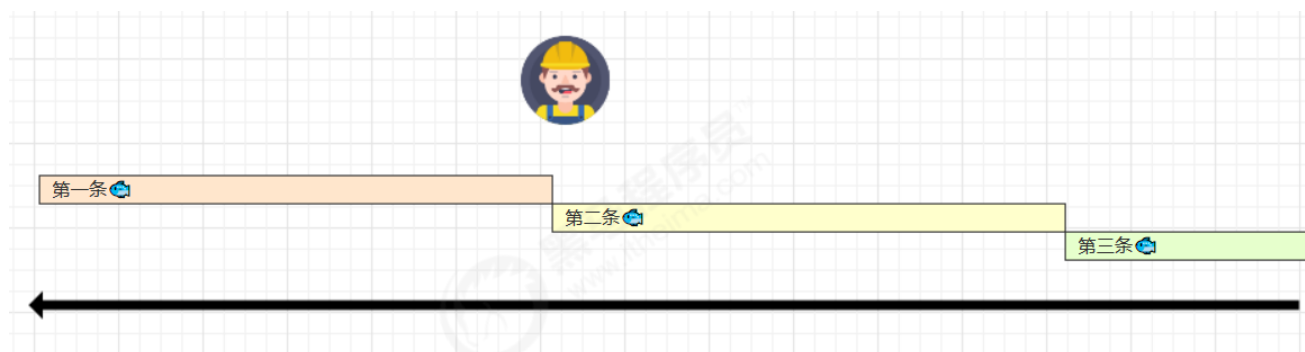
CPU 执行时间

程序的 CPU 执行时间，即我们前面提到的 user + system 时间，可以用下面的公式来表示

程序 CPU 执行时间 = 指令数 * CPI * Clock Cycle Time

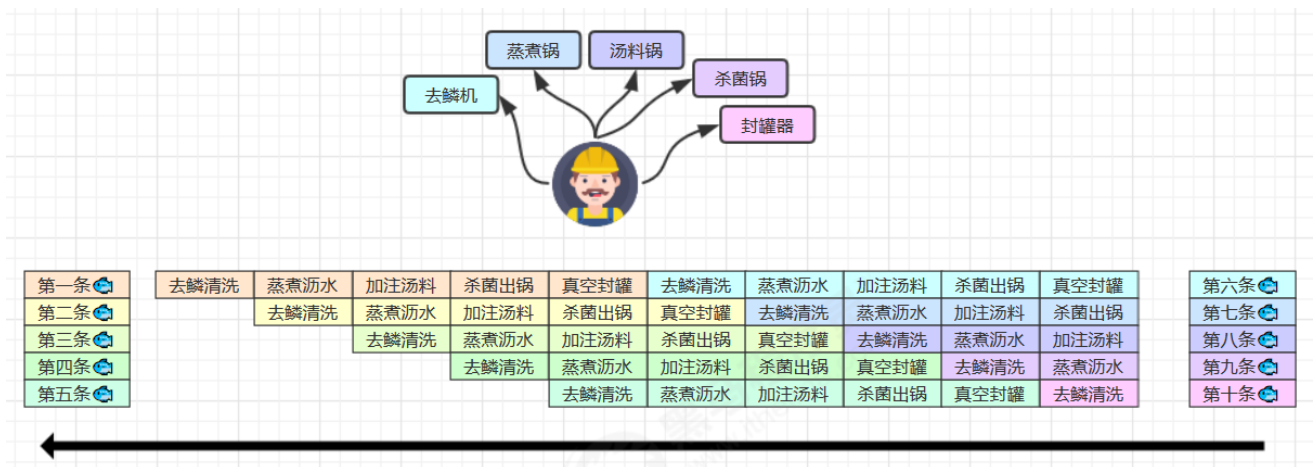
2. 鱼罐头的故事

加工一条鱼需要 50 分钟，只能一条鱼、一条鱼顺序加工...



可以将每个鱼罐头的加工流程细分为 5 个步骤：

- 去鳞清洗 10分钟
- 蒸煮沥水 10分钟
- 加注汤料 10分钟
- 杀菌出锅 10分钟
- 真空封罐 10分钟



即使只有一个工人，最理想的情况是：他能够在 10 分钟内同时做好这 5 件事，因为对第一条鱼的真空装罐，不会影响对第二条鱼的杀菌出锅...

3. 指令重排序优化

事实上，现代处理器会设计为一个时钟周期完成一条执行时间最长的 CPU 指令。为什么这么做呢？可以想到指令还可以再划分成一个个更小的阶段，例如，每条指令都可以分为：取指令 - 指令译码 - 执行指令 - 内存访问 - 数据写回 这 5 个阶段



术语参考：

- instruction fetch (IF)
- instruction decode (ID)
- execute (EX)
- memory access (MEM)
- register write back (WB)

在不改变程序结果的前提下，这些指令的各个阶段可以通过**重排序**和**组合**来实现**指令级并行**，这一技术在 80's 中叶到 90's 中叶占据了计算架构的重要地位。

提示：

分阶段，分工是提升效率的关键！

指令重排的前提是，重排指令不能影响结果，例如

```
// 可以重排的例子
int a = 10; // 指令1
int b = 20; // 指令2
System.out.println( a + b );

// 不能重排的例子
int a = 10; // 指令1
int b = a - 5; // 指令2
```

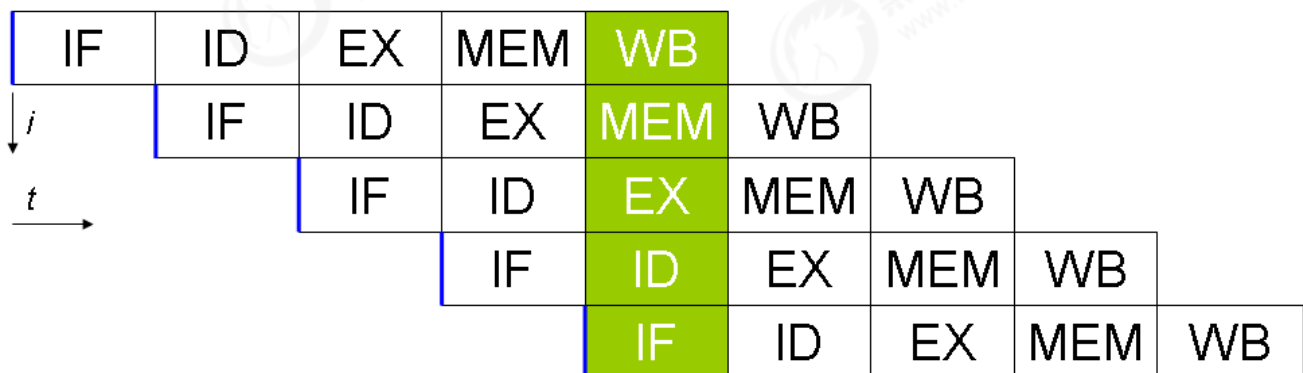
参考：[Scoreboarding](#) and the [Tomasulo algorithm](#) (which is similar to scoreboarding but makes use of [register renaming](#)) are two of the most common techniques for implementing out-of-order execution and instruction-level parallelism.

4. 支持流水线的处理器

现代 CPU 支持**多级指令流水线**，例如支持同时执行 **取指令 - 指令译码 - 执行指令 - 内存访问 - 数据写回** 的处理器，就可以称之为**五级指令流水线**。这时 CPU 可以在一个时钟周期内，同时运行五条指令的不同阶段（相当于一执行时间最长的复杂指令），IPC = 1，本质上，流水线技术并不能缩短单条指令的执行时间，但它变相地提高了指令地吞吐率。

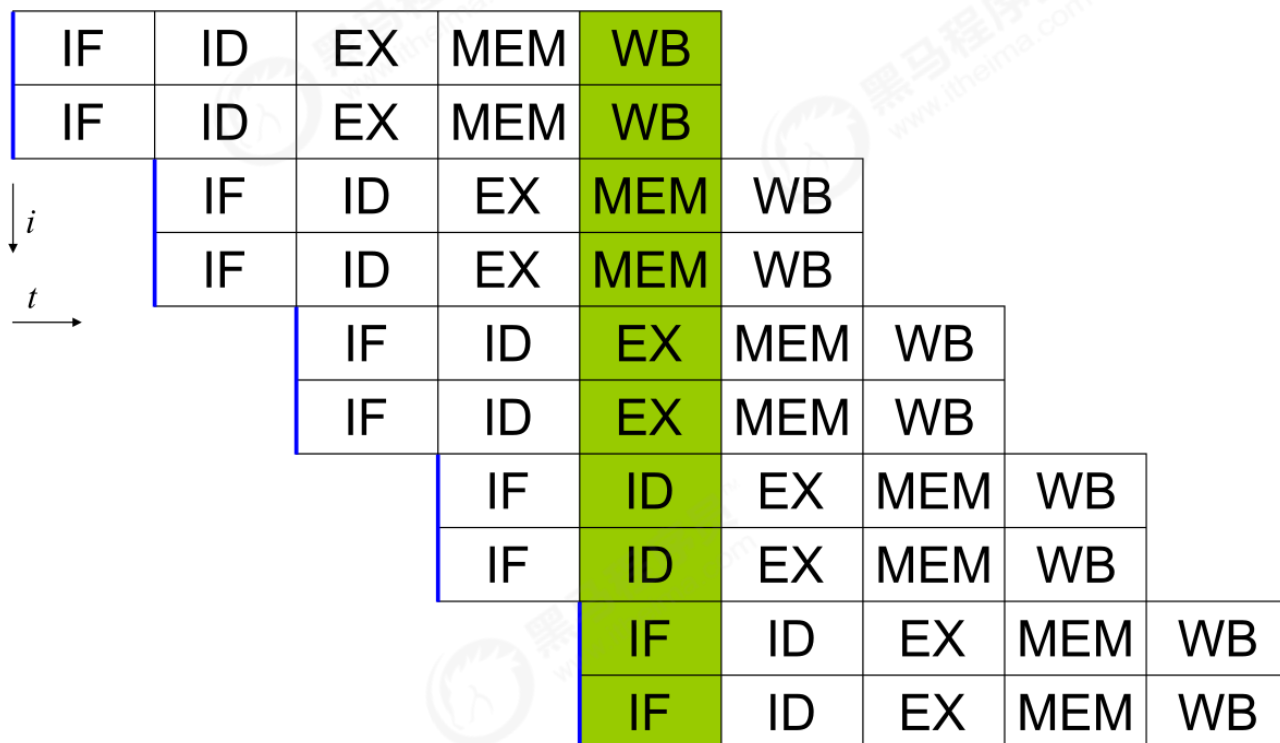
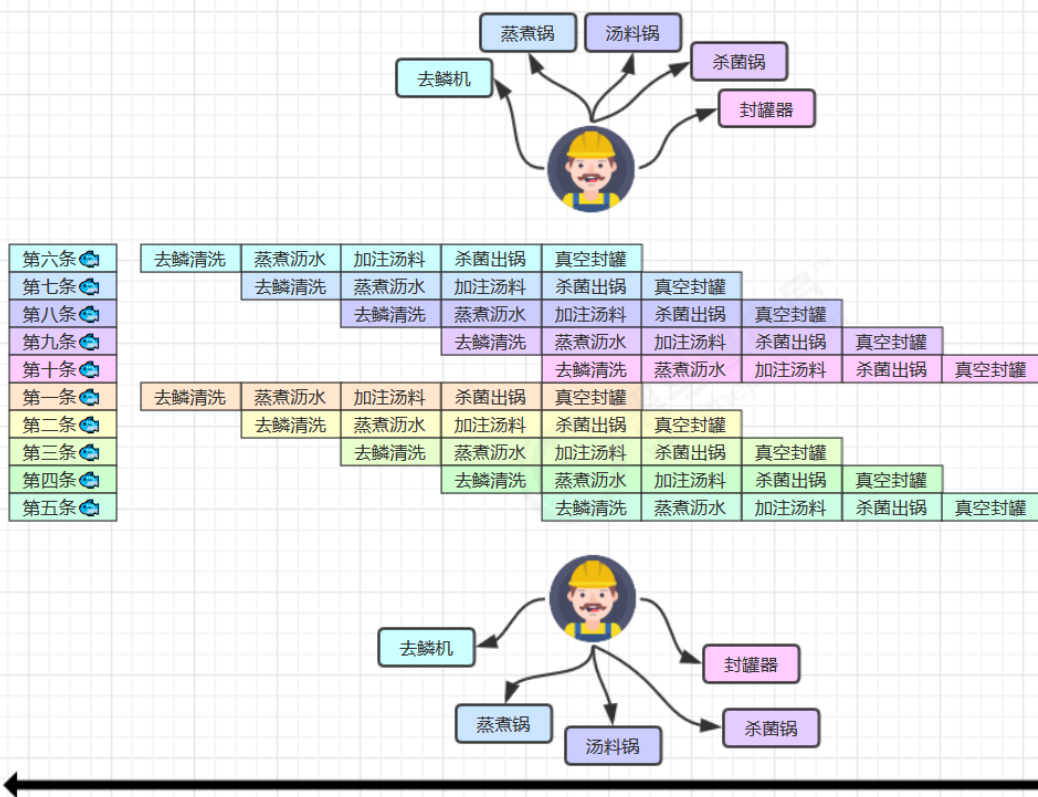
提示：

奔腾四（Pentium 4）支持高达 35 级流水线，但由于功耗太高被废弃



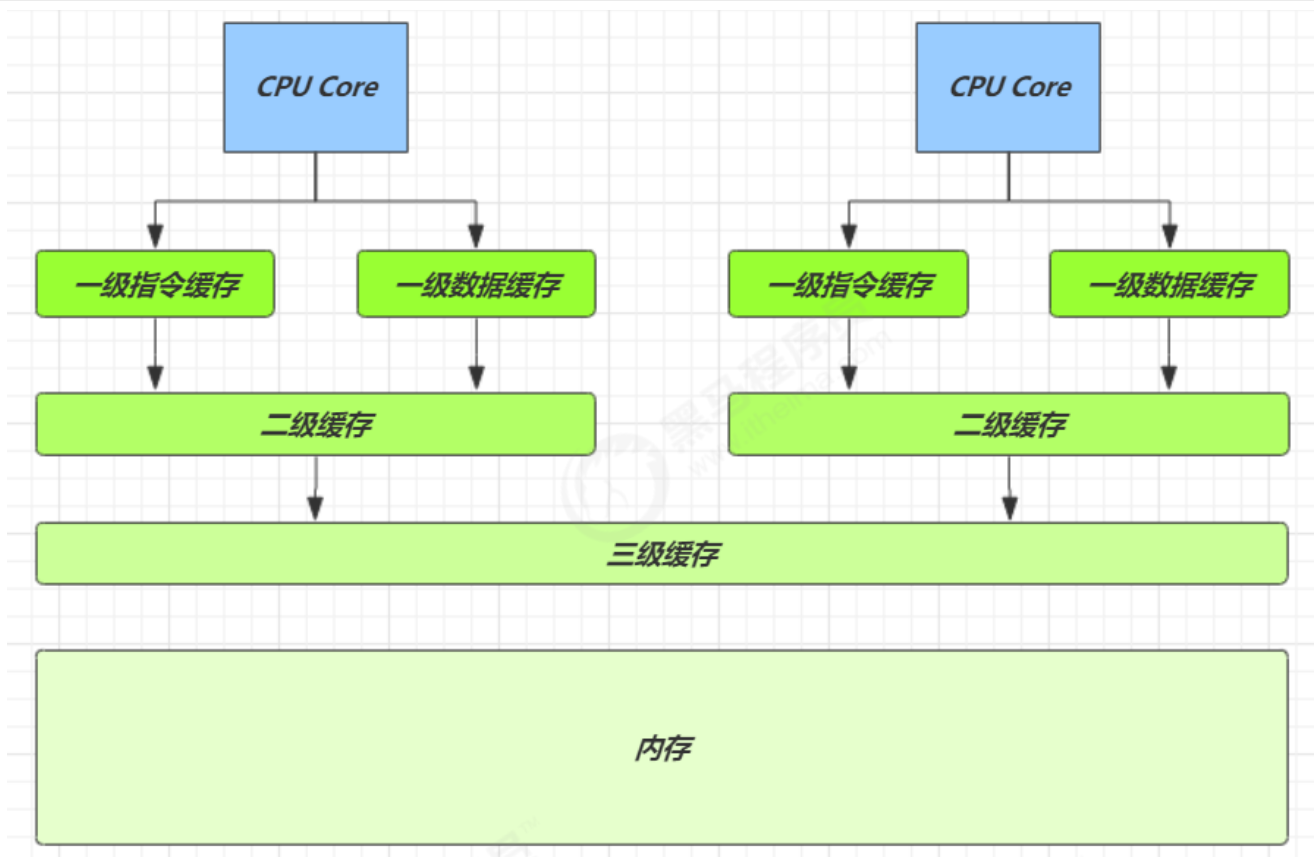
5. SuperScalar 处理器

大多数处理器包含多个执行单元，并不是所有计算功能都集中在一起，可以再细分为整数运算单元、浮点数运算单元等，这样可以把多条指令也可以做到并行获取、译码等，CPU 可以在一个时钟周期内，执行多于一条指令，IPC > 1



CPU 缓存结构原理

1. CPU 缓存结构



查看 cpu 缓存

```
✂ root@yihang01 ~ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                 1
On-line CPU(s) list:   0
Thread(s) per core:    1
Core(s) per socket:    1
Socket(s):              1
NUMA node(s):          1
Vendor ID:              GenuineIntel
CPU family:             6
Model:                 142
Model name:             Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz
Stepping:              11
CPU MHz:               1992.002
BogoMIPS:              3984.00
Hypervisor vendor:     VMware
Virtualization type:    full
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              8192K
NUMA node0 CPU(s):     0
```

速度比较

从 cpu 到	大约需要的时钟周期
寄存器	1 cycle
L1	3~4 cycle
L2	10~20 cycle
L3	40~45 cycle
内存	120~240 cycle

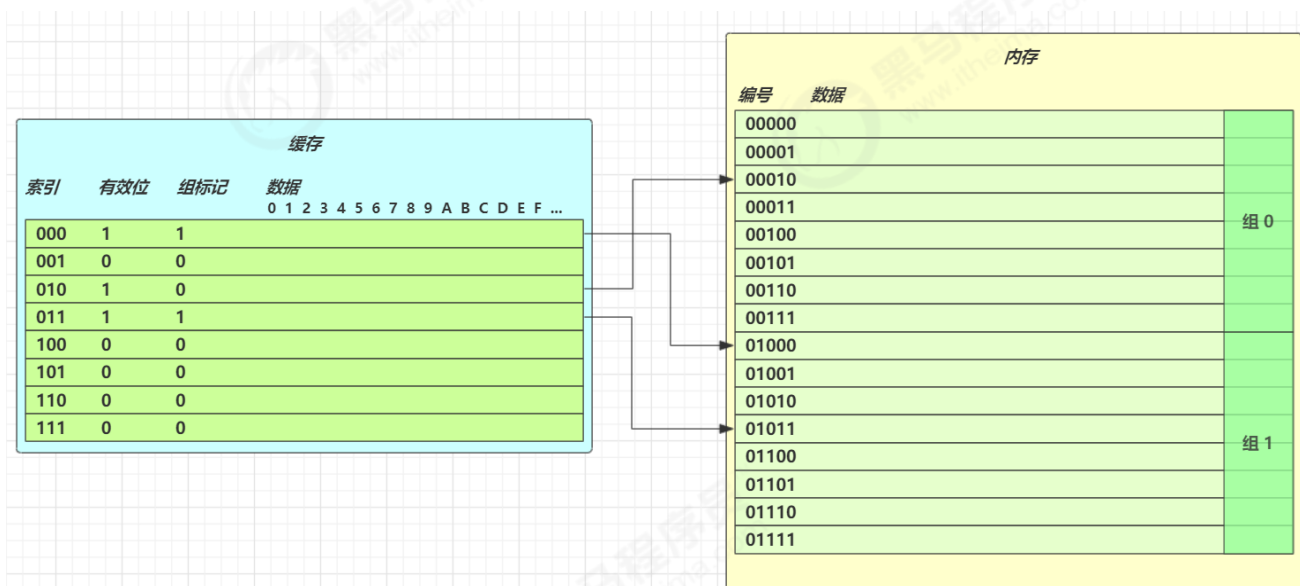
查看 cpu 缓存行

```

$ root@yihang01 ~ cat /sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size
64
    
```

cpu 拿到的内存地址格式是这样的

[高位组标记][低位索引][偏移量]



2. CPU 缓存读

读取数据流程如下

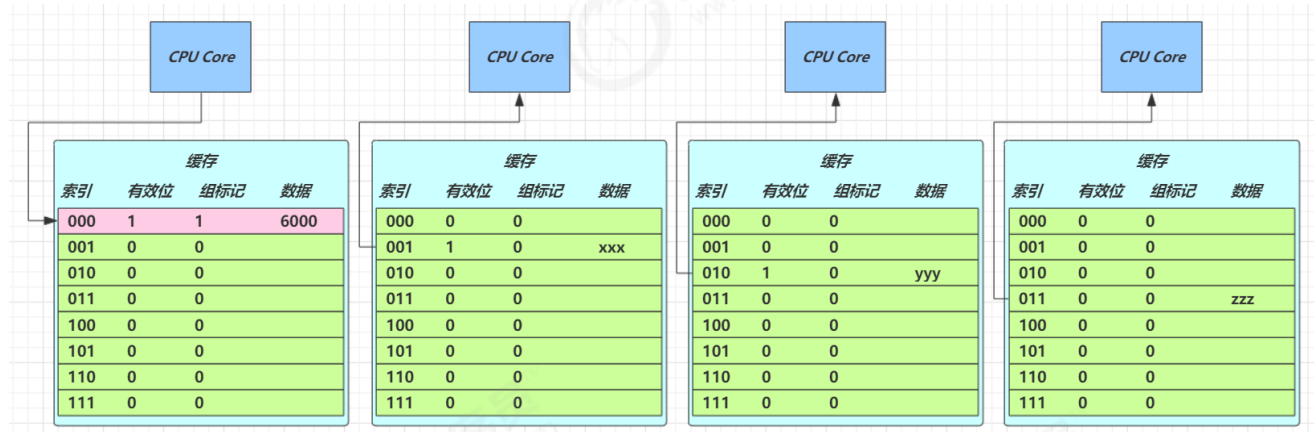
- 根据低位，计算在缓存中的索引
- 判断是否有效
 - 0 去内存读取新数据更新缓存行
 - 1 再对比高位组标记是否一致

- 一致，根据偏移量返回缓存数据
- 不一致，去内存读取新数据更新缓存行

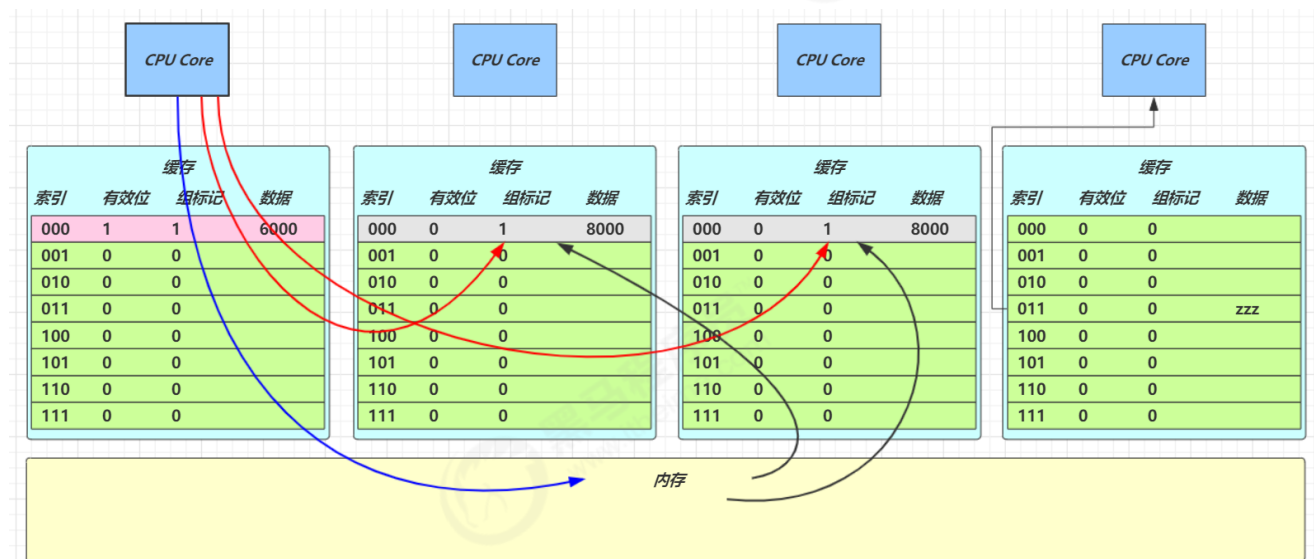
3. CPU 缓存一致性

MESI 协议

1. E、S、M 状态的缓存行都可以满足 CPU 的读请求
2. E 状态的缓存行，有写请求，会将状态改为 M，这时并不触发向主存的写
3. E 状态的缓存行，必须监听该缓存行的读操作，如果有，要变为 S 状态



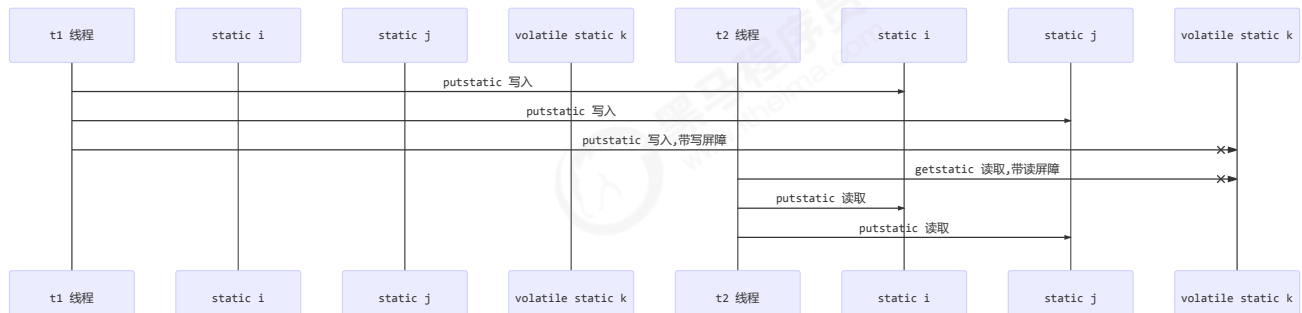
4. M 状态的缓存行，必须监听该缓存行的读操作，如果有，先将其它缓存 (S 状态) 中该缓存行变成 I 状态 (即 6. 的流程)，写入主存，自己变为 S 状态
5. S 状态的缓存行，有写请求，走 4. 的流程
6. S 状态的缓存行，必须监听该缓存行的失效操作，如果有，自己变为 I 状态
7. I 状态的缓存行，有读请求，必须从主存读取



4. 内存屏障

Memory Barrier (Memory Fence)

- 可见性
 - 写屏障 (sfence) 保证在该屏障之前的，对共享变量的改动，都同步到主存当中
 - 而读屏障 (lfence) 保证在该屏障之后，对共享变量的读取，加载的是主存中最新数据
- 有序性
 - 写屏障会确保指令重排序时，不会将写屏障之前的代码排在写屏障之后
 - 读屏障会确保指令重排序时，不会将读屏障之后的代码排在读屏障之前



volatile 原理

volatile 的底层实现原理是内存屏障，Memory Barrier (Memory Fence)

- 对 volatile 变量的写指令后会加入写屏障
- 对 volatile 变量的读指令前会加入读屏障

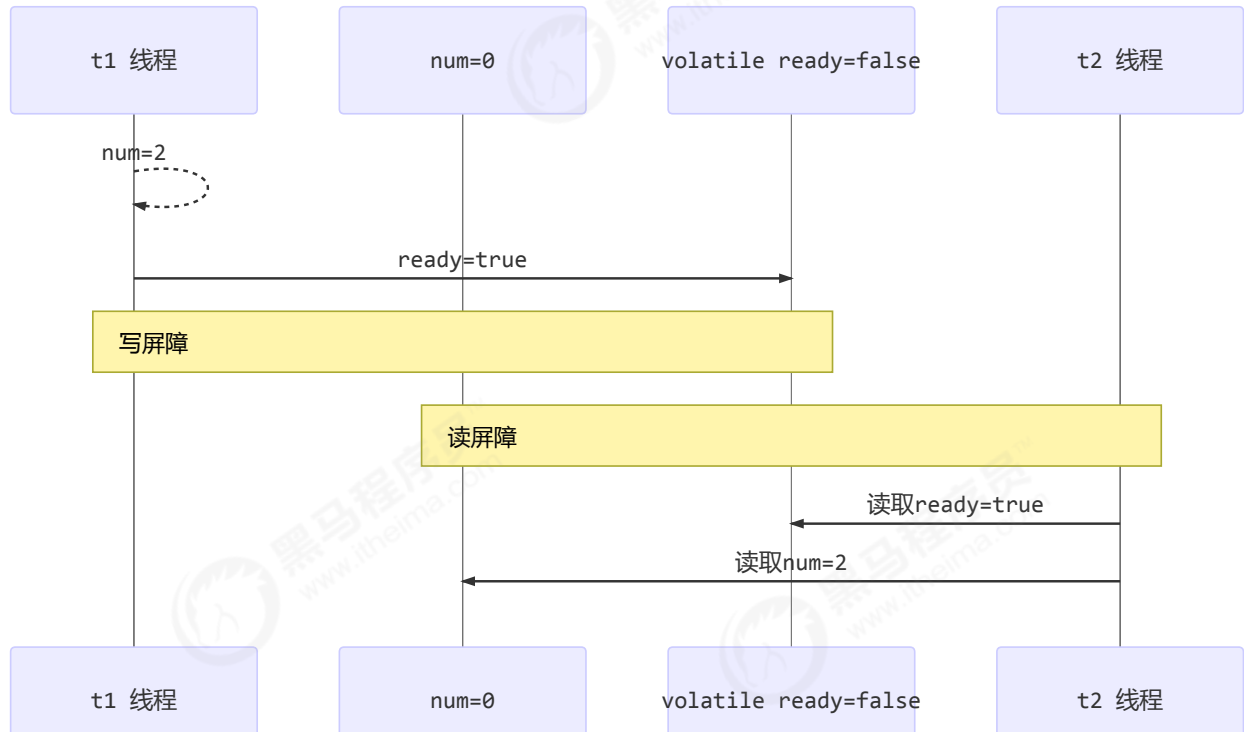
1. 如何保证可见性

- 写屏障 (sfence) 保证在该屏障之前的，对共享变量的改动，都同步到主存当中

```
public void actor2(I_Result r) {  
    num = 2;  
    ready = true; // ready 是 volatile 赋值带写屏障  
    // 写屏障  
}
```

- 而读屏障 (lfence) 保证在该屏障之后，对共享变量的读取，加载的是主存中最新数据

- ```
public void actor1(I_Result r) {
 // 读屏障
 // ready 是 volatile 读取值带读屏障
 if(ready) {
 r.r1 = num + num;
 } else {
 r.r1 = 1;
 }
}
```



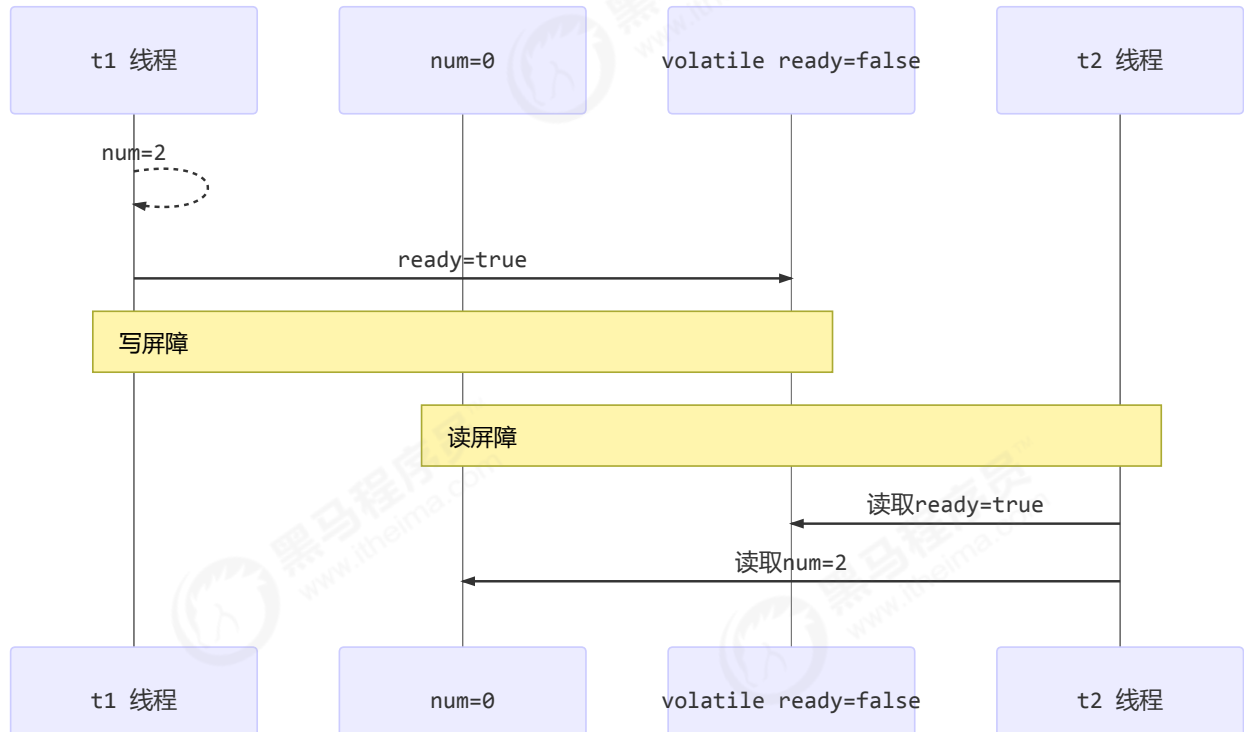
## 2. 如何保证有序性

- 写屏障会确保指令重排序时，不会将写屏障之前的代码排在写屏障之后

- ```
public void actor2(I_Result r) {  
    num = 2;  
    ready = true; // ready 是 volatile 赋值带写屏障  
    // 写屏障  
}
```

- 读屏障会确保指令重排序时，不会将读屏障之后的代码排在读屏障之前

```
public void actor1(I_Result r) {  
    // 读屏障  
    // ready 是 volatile 读取值带读屏障  
    if(ready) {  
        r.r1 = num + num;  
    } else {  
        r.r1 = 1;  
    }  
}
```



还是那句话，不能解决指令交错：

- 写屏障仅仅是保证之后的读能够读到最新的结果，但不能保证读跑到它前面去
- 而有序性的保证也只是保证了本线程内相关代码不被重排序



3. double-checked locking 问题

以著名的 double-checked locking 单例模式为例

```
public final class Singleton {
    private Singleton() { }
    private static Singleton INSTANCE = null;
    public static Singleton getInstance() {
        if(INSTANCE == null) { // t2
            // 首次访问会同步，而之后的使用没有 synchronized
            synchronized(Singleton.class) {
                if (INSTANCE == null) { // t1
                    INSTANCE = new Singleton();
                }
            }
        }
        return INSTANCE;
    }
}
```

以上的实现特点是：

- 懒惰实例化
- 首次使用 getInstance() 才使用 synchronized 加锁，后续使用时无需加锁
- 有隐含的，但很关键的一点：第一个 if 使用了 INSTANCE 变量，是在同步块之外

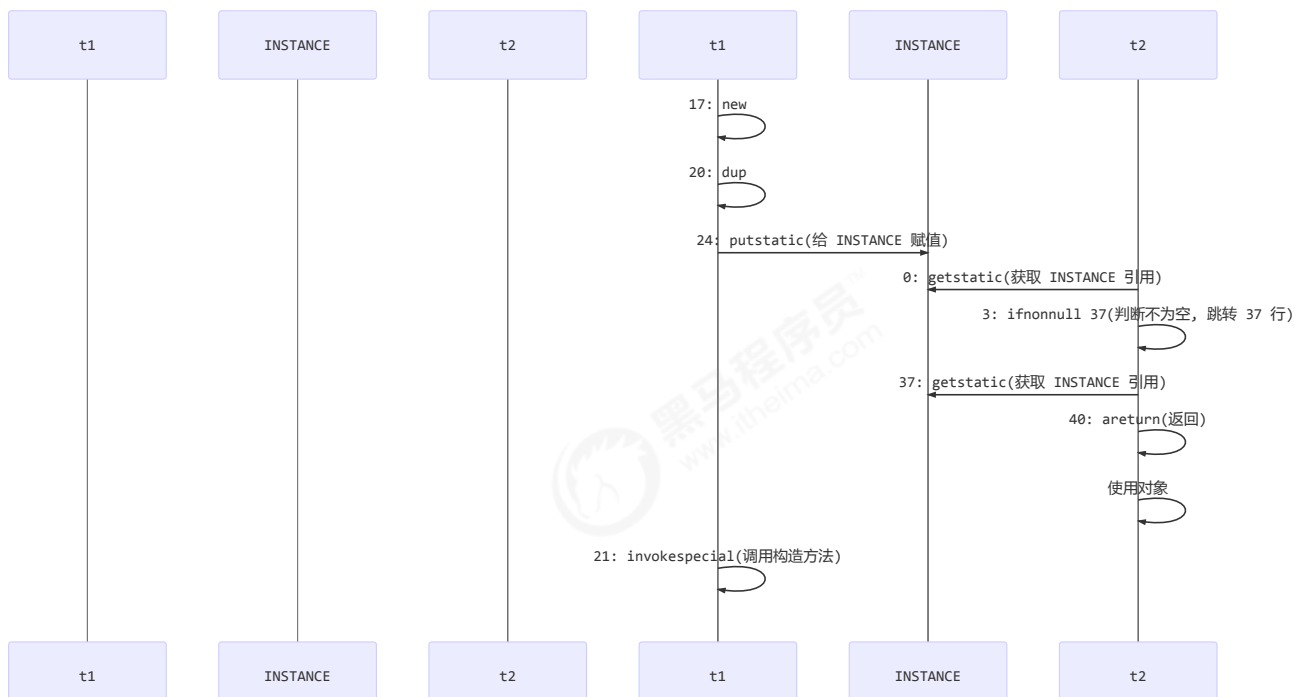
但在多线程环境下，上面的代码是有问题的，getInstance 方法对应的字节码为：

```
0: getstatic      #2          // Field INSTANCE:Lcn/itcast/n5/Singleton;
3: ifnonnull      37
6: ldc           #3          // class cn/itcast/n5/Singleton
8: dup
9: astore_0
10: monitorenter
11: getstatic      #2          // Field INSTANCE:Lcn/itcast/n5/Singleton;
14: ifnonnull      27
17: new           #3          // class cn/itcast/n5/Singleton
20: dup
21: invokespecial #4          // Method "<init>":()V
24: putstatic      #2          // Field INSTANCE:Lcn/itcast/n5/Singleton;
27: aload_0
28: monitorexit
29: goto          37
32: astore_1
33: aload_0
34: monitorexit
35: aload_1
36: athrow
37: getstatic      #2          // Field INSTANCE:Lcn/itcast/n5/Singleton;
40: areturn
```

其中

- 17 表示创建对象，将对象引用入栈 // new Singleton
- 20 表示复制一份对象引用 // 引用地址
- 21 表示利用一个对象引用，调用构造方法
- 24 表示利用一个对象引用，赋值给 static INSTANCE

也许 jvm 会优化为：先执行 24，再执行 21。如果两个线程 t1，t2 按如下时间序列执行：



关键在于 0: getstatic 这行代码在 monitor 控制之外，它就像之前举例中不守规则的人，可以越过 monitor 读取 INSTANCE 变量的值

这时 t1 还未完全将构造方法执行完毕，如果在构造方法中要执行很多初始化操作，那么 t2 拿到的是将是一个未初始化完毕的单例

对 INSTANCE 使用 volatile 修饰即可，可以禁用指令重排，但要注意在 JDK 5 以上的版本的 volatile 才会真正有效

4. double-checked locking 解决

```

public final class Singleton {
    private Singleton() { }
    private static volatile Singleton INSTANCE = null;
    public static Singleton getInstance() {
        // 实例没创建，才会进入内部的 synchronized代码块
        if (INSTANCE == null) {
            synchronized (Singleton.class) { // t2
                // 也许有其它线程已经创建实例，所以再判断一次
                if (INSTANCE == null) { // t1
                    INSTANCE = new Singleton();
                }
            }
        }
        return INSTANCE;
    }
}
    
```

字节码上看起来 volatile 指令的效果

```

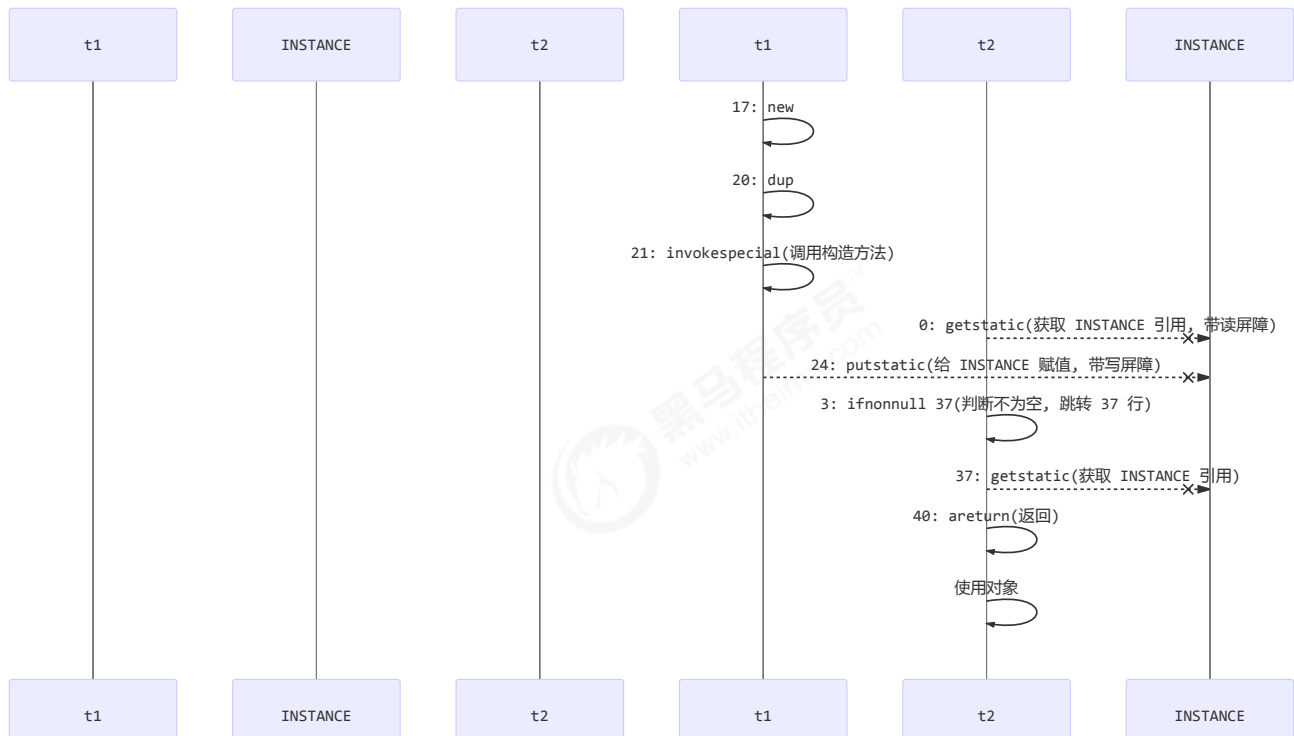
// -----> 加入对 INSTANCE 变量的读屏障
0: getstatic    #2                // Field INSTANCE:Lcn/itcast/n5/Singleton;
    
```



```
3: ifnonnull      37
6: ldc           #3                // class cn/itcast/n5/Singleton
8: dup
9: astore_0
10: monitorenter -----> 保证原子性、可见性
11: getstatic      #2                // Field INSTANCE:Lcn/itcast/n5/Singleton;
14: ifnonnull      27
17: new            #3                // class cn/itcast/n5/Singleton
20: dup
21: invokespecial  #4                // Method "<init>":()V
24: putstatic      #2                // Field INSTANCE:Lcn/itcast/n5/Singleton;
// -----> 加入对 INSTANCE 变量的写屏障
27: aload_0
28: monitorexit -----> 保证原子性、可见性
29: goto           37
32: astore_1
33: aload_0
34: monitorexit
35: aload_1
36: athrow
37: getstatic      #2                // Field INSTANCE:Lcn/itcast/n5/Singleton;
40: areturn
```

如上面的注释内容所示，读写 volatile 变量时会加入内存屏障（Memory Barrier（Memory Fence）），保证下面两点：

- 可见性
 - 写屏障（sfence）保证在该屏障之前的 t1 对共享变量的改动，都同步到主存当中
 - 而读屏障（lfence）保证在该屏障之后 t2 对共享变量的读取，加载的是主存中最新数据
- 有序性
 - 写屏障会确保指令重排序时，不会将写屏障之前的代码排在写屏障之后
 - 读屏障会确保指令重排序时，不会将读屏障之后的代码排在读屏障之前
- 更底层是读写变量时使用 lock 指令来多核 CPU 之间的可见性与有序性



final 原理

1. 设置 final 变量的原理

理解了 volatile 原理，再对比 final 的实现就比较简单了

```
public class TestFinal {  
    final int a = 20;  
}
```

字节码

```
0: aload_0  
1: invokespecial #1          // Method java/lang/Object."<init>":()V  
4: aload_0  
5: bipush        20  
7: putfield     #2          // Field a:I  
  <-- 写屏障  
10: return
```

发现 final 变量的赋值也会通过 putfield 指令来完成，同样在这条指令之后也会加入写屏障，保证在其它线程读到它的值时不会出现为 0 的情况

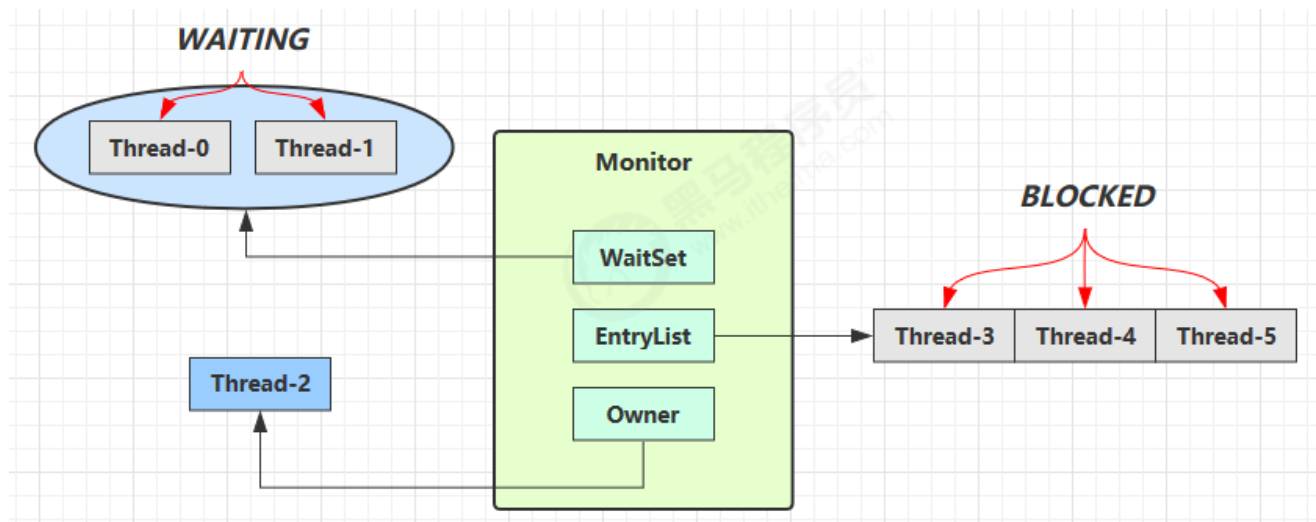
2. 获取 final 变量的原理

Monitor 原理

Monitor 被翻译为**监视器**或**管程**

每个 Java 对象都可以关联一个 Monitor 对象，如果使用 synchronized 给对象上锁（重量级）之后，该对象头的 Mark Word 中就被设置指向 Monitor 对象的指针

Monitor 结构如下



- 刚开始 Monitor 中 Owner 为 null
- 当 Thread-2 执行 synchronized(obj) 就会将 Monitor 的所有者 Owner 置为 Thread-2，Monitor 中只能有一个 Owner
- 在 Thread-2 上锁的过程中，如果 Thread-3，Thread-4，Thread-5 也来执行 synchronized(obj)，就会进入 EntryList BLOCKED
- Thread-2 执行完同步代码块的内容，然后唤醒 EntryList 中等待的线程来竞争锁，竞争的时是非公平的
- 图中 WaitSet 中的 Thread-0，Thread-1 是之前获得过锁，但条件不满足进入 WAITING 状态的线程，后面讲 wait-notify 时会分析

注意：

- synchronized 必须是进入同一个对象的 monitor 才有上述的效果
- 不加 synchronized 的对象不会关联监视器，不遵从以上规则

synchronized 原理

```
static final Object lock = new Object();
static int counter = 0;

public static void main(String[] args) {
    synchronized (lock) {
        counter++;
    }
}
```

对应的字节码为

```
public static void main([Ljava/lang/String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: ACC_PUBLIC, ACC_STATIC
```


Code:

```
stack=2, locals=3, args_size=1
  0: getstatic      #2                // <- lock引用 (synchronized开始)
  3: dup
  4: astore_1        // lock引用 -> slot 1
  5: monitorenter    // 将 lock对象 MarkWord 置为 Monitor 指针
  6: getstatic      #3                // <- i
  9: iconst_1        // 准备常数 1
 10: iadd            // +1
 11: putstatic      #3                // -> i
 14: aload_1        // <- lock引用
 15: monitorexit    // 将 lock对象 MarkWord 重置, 唤醒 EntryList
 16: goto          24
 19: astore_2        // e -> slot 2
 20: aload_1        // <- lock引用
 21: monitorexit    // 将 lock对象 MarkWord 重置, 唤醒 EntryList
 22: aload_2        // <- slot 2 (e)
 23: athrow         // throw e
 24: return

Exception table:
   from    to  target type
    6      16     19   any
   19      22     19   any

LineNumberTable:
 line 8: 0
 line 9: 6
 line 10: 14
 line 11: 24

LocalVariableTable:
 Start Length Slot Name   Signature
    0      25     0  args  [Ljava/lang/String;

StackMapTable: number_of_entries = 2
 frame_type = 255 /* full_frame */
   offset_delta = 19
   locals = [ class "[Ljava/lang/String;", class java/lang/Object ]
   stack = [ class java/lang/Throwable ]
 frame_type = 250 /* chop */
   offset_delta = 4
```

注意

方法级别的 synchronized 不会在字节码指令中有所体现

synchronized 原理进阶

1. 轻量级锁

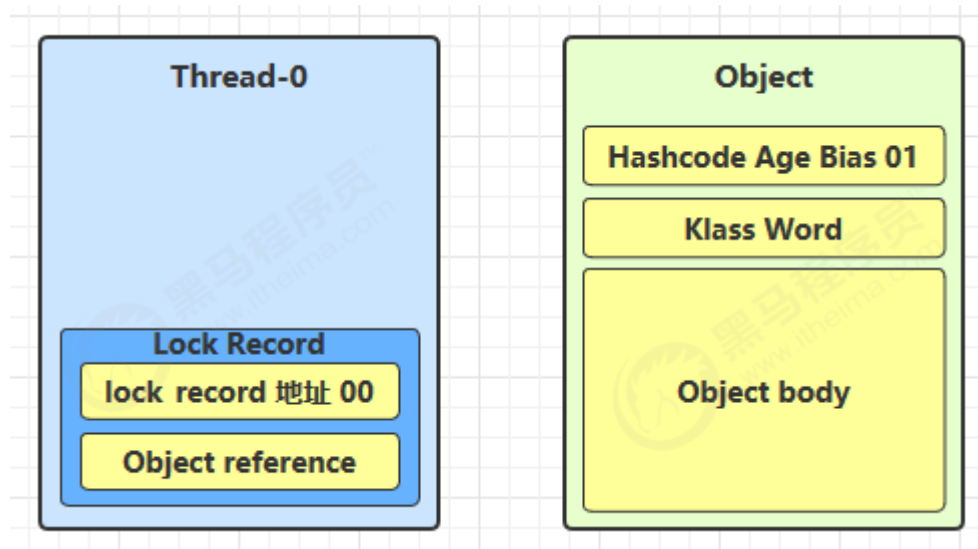
轻量级锁的使用场景：如果一个对象虽然有多线程要加锁，但加锁的时间是错开的（也就是没有竞争），那么可以使用轻量级锁来优化。

轻量级锁对使用者是透明的，即语法仍然是 `synchronized`

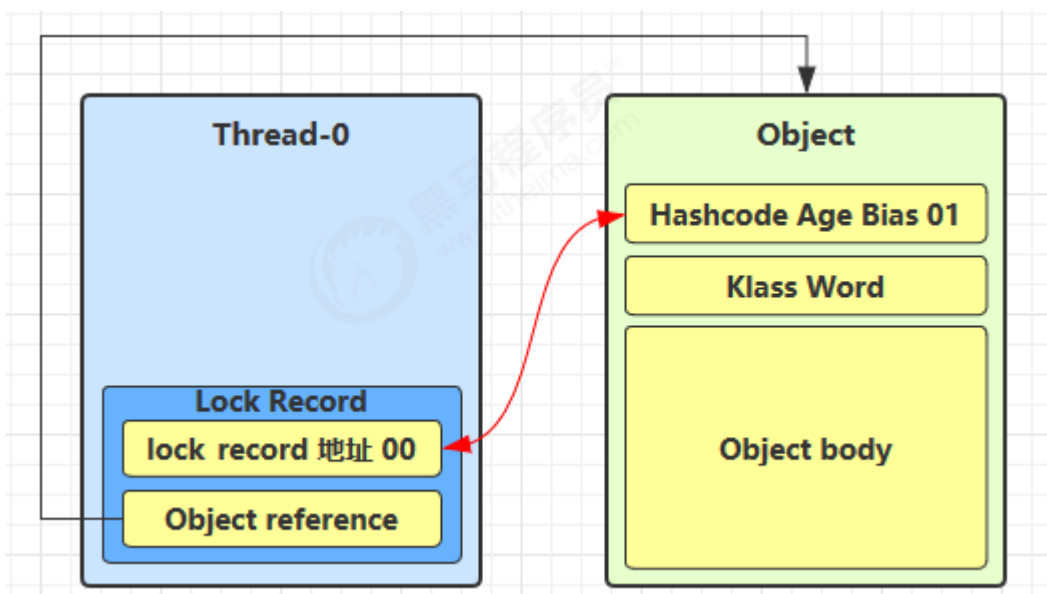
假设有两个方法同步块，利用同一个对象加锁

```
static final Object obj = new Object();
public static void method1() {
    synchronized( obj ) {
        // 同步块 A
        method2();
    }
}
public static void method2() {
    synchronized( obj ) {
        // 同步块 B
    }
}
```

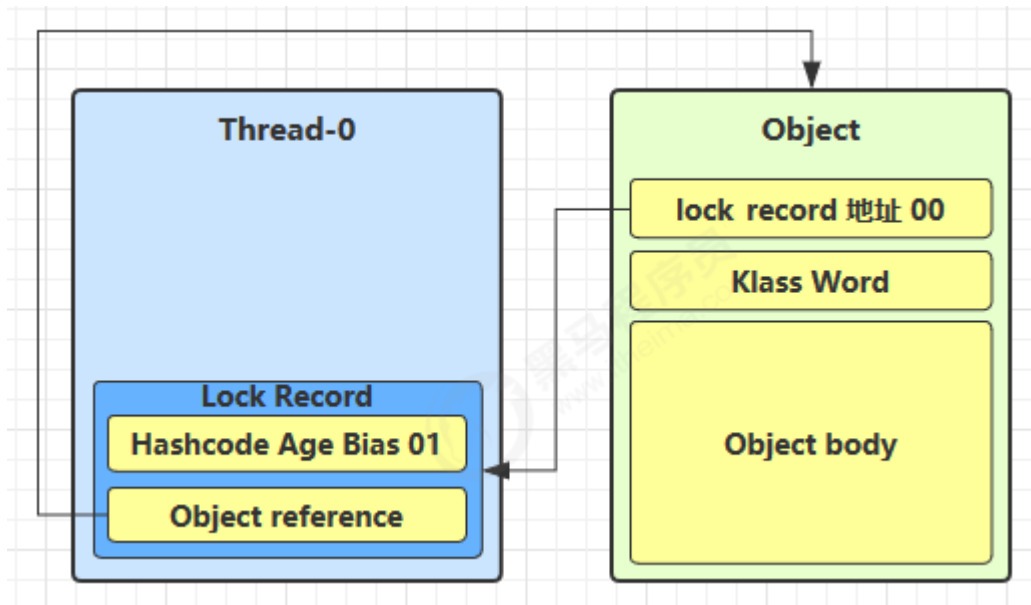
- 创建锁记录 (Lock Record) 对象，每个线程的栈帧都会包含一个锁记录的结构，内部可以存储锁定对象的 Mark Word



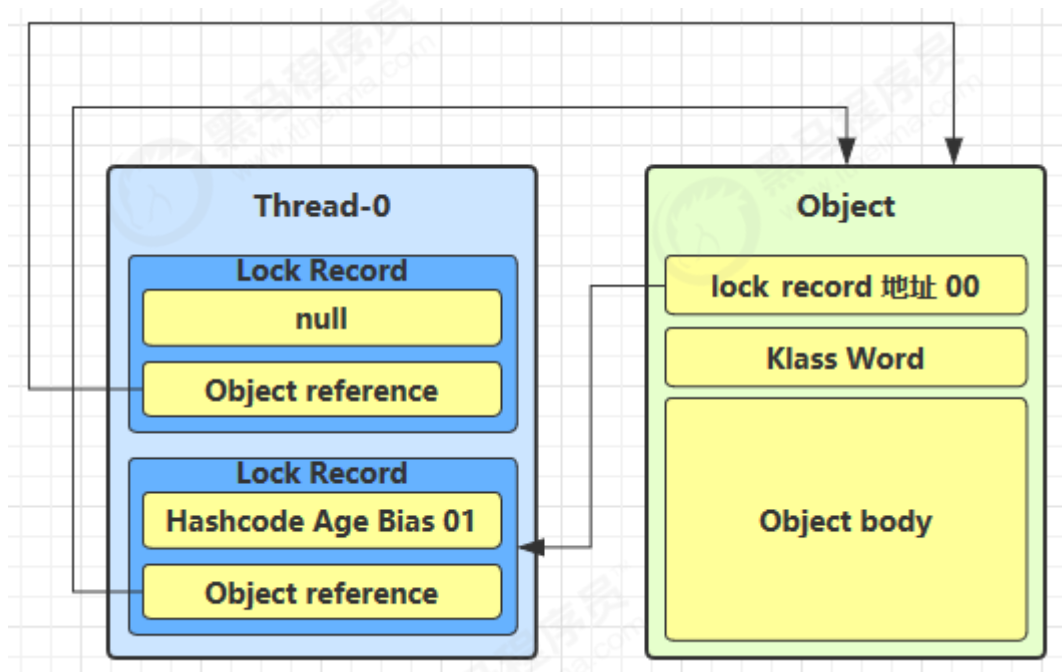
- 让锁记录中 Object reference 指向锁对象，并尝试用 cas 替换 Object 的 Mark Word，将 Mark Word 的值存入锁记录



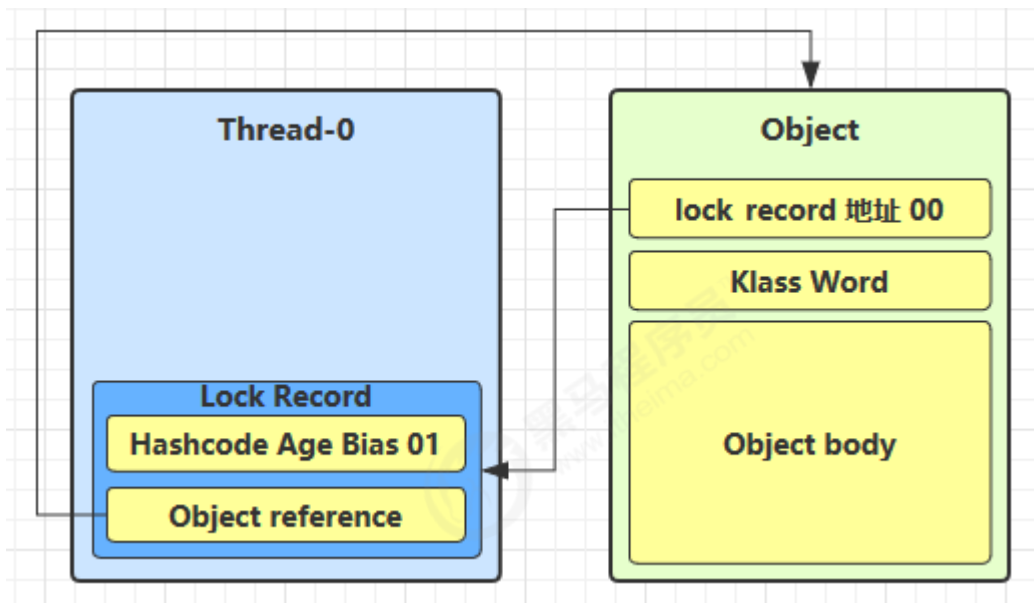
- 如果 cas 替换成功，对象头中存储了 锁记录地址和状态 00，表示由该线程给对象加锁，这时图示如下



- 如果 cas 失败，有两种情况
 - 如果是其它线程已经持有了该 Object 的轻量级锁，这时表明有竞争，进入锁膨胀过程
 - 如果是自己执行了 synchronized 锁重入，那么再添加一条 Lock Record 作为重入的计数



- 当退出 synchronized 代码块（解锁时）如果有取值为 null 的锁记录，表示有重入，这时重置锁记录，表示重入计数减一



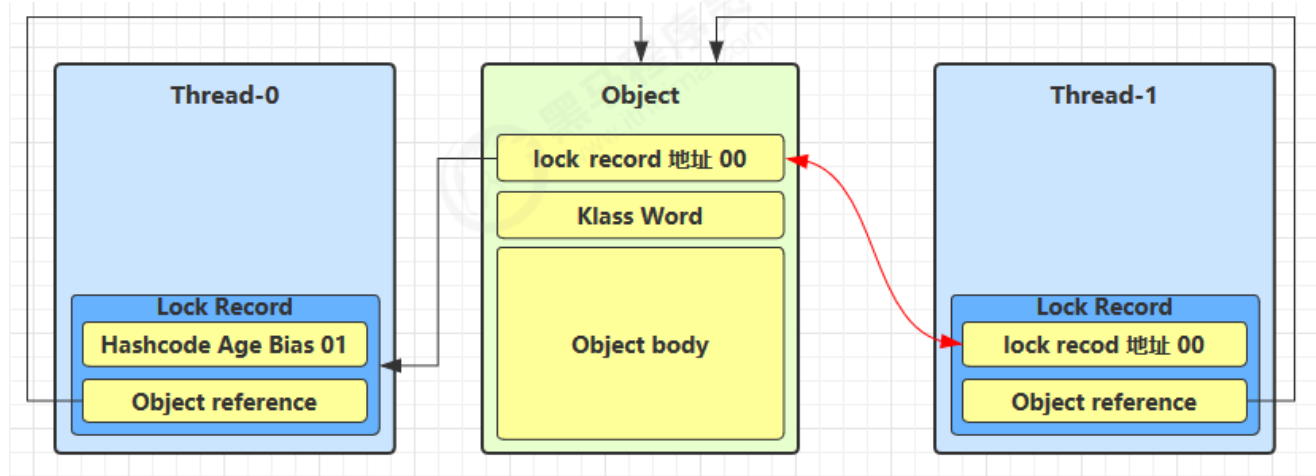
- 当退出 synchronized 代码块（解锁时）锁记录的值不为 null，这时使用 cas 将 Mark Word 的值恢复给对象头
 - 成功，则解锁成功
 - 失败，说明轻量级锁进行了锁膨胀或已经升级为重量级锁，进入重量级锁解锁流程

2. 锁膨胀

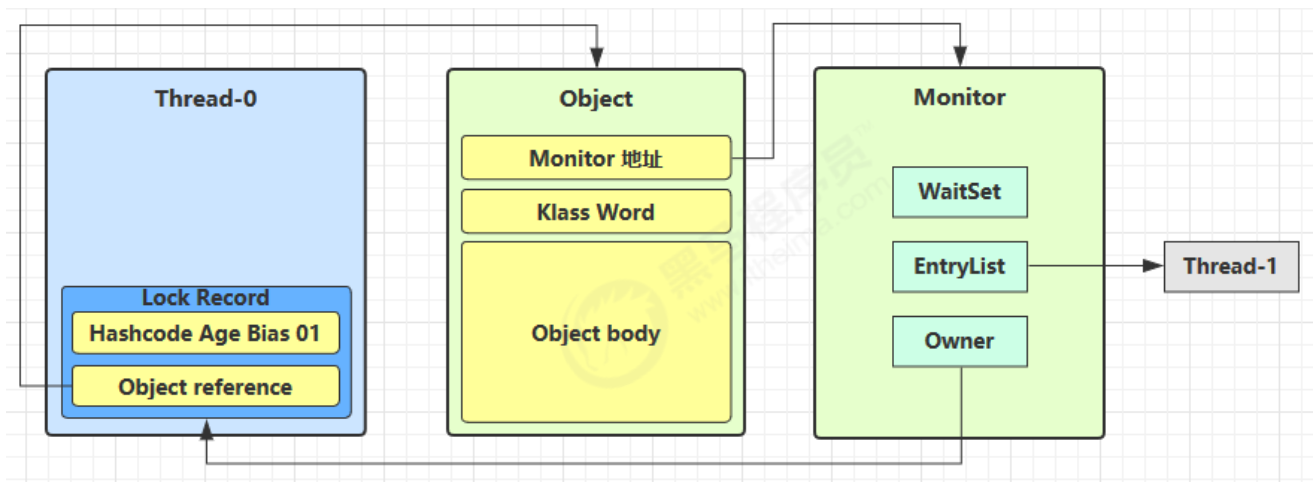
如果在尝试加轻量级锁的过程中，CAS 操作无法成功，这时一种情况就是有其它线程为此对象加上了轻量级锁（有竞争），这时需要进行锁膨胀，将轻量级锁变为重量级锁。

```
static Object obj = new Object();
public static void method1() {
    synchronized( obj ) {
        // 同步块
    }
}
```

- 当 Thread-1 进行轻量级加锁时，Thread-0 已经对该对象加了轻量级锁



- 这时 Thread-1 加轻量级锁失败，进入锁膨胀流程
 - 即为 Object 对象申请 Monitor 锁，让 Object 指向重量级锁地址
 - 然后自己进入 Monitor 的 EntryList BLOCKED



- 当 Thread-0 退出同步块解锁时，使用 cas 将 Mark Word 的值恢复给对象头，失败。这时会进入重量级解锁流程，即按照 Monitor 地址找到 Monitor 对象，设置 Owner 为 null，唤醒 EntryList 中 BLOCKED 线程

3. 自旋优化

重量级锁竞争的时候，还可以使用自旋来进行优化，如果当前线程自旋成功（即这时候持锁线程已经退出了同步块，释放了锁），这时当前线程就可以避免阻塞。

自旋重试成功的情况

线程 1（core 1 上）	对象 Mark	线程 2（core 2 上）
-	10（重量锁）	-
访问同步块，获取 monitor	10（重量锁）重量锁指针	-
成功（加锁）	10（重量锁）重量锁指针	-
执行同步块	10（重量锁）重量锁指针	-
执行同步块	10（重量锁）重量锁指针	访问同步块，获取 monitor
执行同步块	10（重量锁）重量锁指针	自旋重试
执行完毕	10（重量锁）重量锁指针	自旋重试
成功（解锁）	01（无锁）	自旋重试
-	10（重量锁）重量锁指针	成功（加锁）
-	10（重量锁）重量锁指针	执行同步块
-

自旋重试失败的情况

线程 1 (core 1 上)	对象 Mark	线程 2 (core 2 上)
-	10 (重量锁)	-
访问同步块，获取 monitor	10 (重量锁) 重量锁指针	-
成功 (加锁)	10 (重量锁) 重量锁指针	-
执行同步块	10 (重量锁) 重量锁指针	-
执行同步块	10 (重量锁) 重量锁指针	访问同步块，获取 monitor
执行同步块	10 (重量锁) 重量锁指针	自旋重试
执行同步块	10 (重量锁) 重量锁指针	自旋重试
执行同步块	10 (重量锁) 重量锁指针	自旋重试
执行同步块	10 (重量锁) 重量锁指针	阻塞
-

- 自旋会占用 CPU 时间，单核 CPU 自旋就是浪费，多核 CPU 自旋才能发挥优势。
- 在 Java 6 之后自旋锁是自适应的，比如对象刚刚的一次自旋操作成功过，那么认为这次自旋成功的可能性会高，就多自旋几次；反之，就少自旋甚至不自旋，总之，比较智能。
- Java 7 之后不能控制是否开启自旋功能

4. 偏向锁

轻量级锁在没有竞争时（就自己这个线程），每次重入仍然需要执行 CAS 操作。

Java 6 中引入了偏向锁来做进一步优化：只有第一次使用 CAS 将线程 ID 设置到对象的 Mark Word 头，之后发现这个线程 ID 是自己的就表示没有竞争，不用重新 CAS。以后只要不发生竞争，这个对象就归该线程所有

例如：

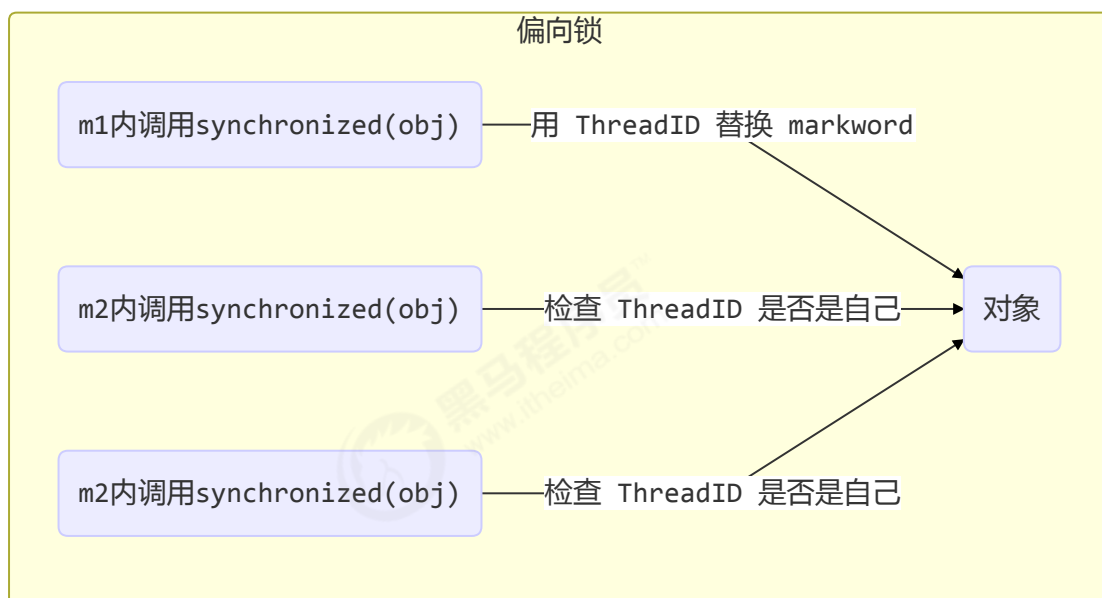
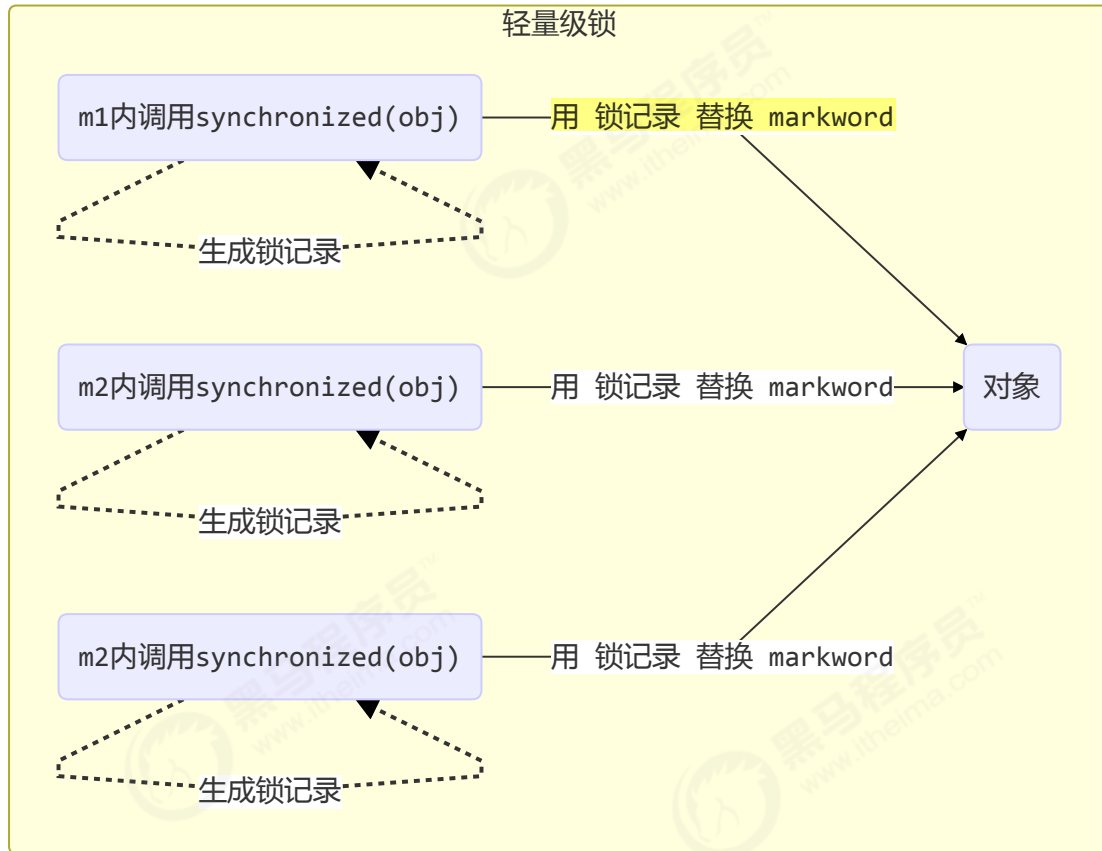
```

static final Object obj = new Object();
public static void m1() {
    synchronized( obj ) {
        // 同步块 A
        m2();
    }
}
public static void m2() {
    synchronized( obj ) {
        // 同步块 B
        m3();
    }
}
public static void m3() {
    synchronized( obj ) {

```

// 同步块 c

```
}  
}
```



偏向状态

回忆一下对象头格式

Mark Word (64 bits)						State
unused:25	hashCode:31	unused:1	age:4	biased_lock:0	01	Normal
thread:54	epoch:2	unused:1	age:4	biased_lock:1	01	Biased
ptr_to_lock_record:62					00	Lightweight Locked
ptr_to_heavyweight_monitor:62					10	Heavyweight Locked
					11	Marked for GC

一个对象创建时：

- 如果开启了偏向锁（默认开启），那么对象创建后，markword 值为 0x05 即最后 3 位为 101，这时它的 thread、epoch、age 都为 0
- 偏向锁是默认是延迟的，不会在程序启动时立即生效，如果想避免延迟，可以加 VM 参数 `-XX:BiasedLockingStartupDelay=0` 来禁用延迟
- 如果没有开启偏向锁，那么对象创建后，markword 值为 0x01 即最后 3 位为 001，这时它的 hashCode、age 都为 0，第一次用到 hashCode 时才会赋值

1) 测试延迟特性

2) 测试偏向锁

```
class Dog {}
```

利用 jol 第三方工具来查看对象头信息（注意这里我扩展了 jol 让它输出更为简洁）

```
// 添加虚拟机参数 -XX:BiasedLockingStartupDelay=0
public static void main(String[] args) throws IOException {
    Dog d = new Dog();
    ClassLayout classLayout = ClassLayout.parseInstance(d);

    new Thread(() -> {
        log.debug("synchronized 前");
        System.out.println(classLayout.toPrintableSimple(true));
        synchronized (d) {
            log.debug("synchronized 中");
            System.out.println(classLayout.toPrintableSimple(true));
        }
        log.debug("synchronized 后");
        System.out.println(classLayout.toPrintableSimple(true));
    }, "t1").start();
}
```



```
}
```

输出

```
11:08:58.117 c.TestBiased [t1] - synchronized 前
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000101
11:08:58.121 c.TestBiased [t1] - synchronized 中
00000000 00000000 00000000 00000000 00011111 11101011 11010000 00000101
11:08:58.121 c.TestBiased [t1] - synchronized 后
00000000 00000000 00000000 00000000 00011111 11101011 11010000 00000101
```

注意

处于偏向锁的对象解锁后，线程 id 仍存储于对象头中

3) 测试禁用

在上面测试代码运行时在添加 VM 参数 `-XX:-UseBiasedLocking` 禁用偏向锁

输出

```
11:13:10.018 c.TestBiased [t1] - synchronized 前
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001
11:13:10.021 c.TestBiased [t1] - synchronized 中
00000000 00000000 00000000 00000000 00100000 00010100 11110011 10001000
11:13:10.021 c.TestBiased [t1] - synchronized 后
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001
```

4) 测试 hashCode

- 正常状态对象一开始是没有 hashCode 的，第一次调用才生成

撤销 - 调用对象 hashCode

调用了对象的 hashCode，但偏向锁的对象 MarkWord 中存储的是线程 id，如果调用 hashCode 会导致偏向锁被撤销

- 轻量级锁会在锁记录中记录 hashCode
- 重量级锁会在 Monitor 中记录 hashCode

在调用 hashCode 后使用偏向锁，记得去掉 `-XX:-UseBiasedLocking`

输出

```
11:22:10.386 c.TestBiased [main] - 调用 hashCode:1778535015
11:22:10.391 c.TestBiased [t1] - synchronized 前
00000000 00000000 00000000 01101010 00000010 01001010 01100111 00000001
11:22:10.393 c.TestBiased [t1] - synchronized 中
00000000 00000000 00000000 00000000 00100000 11000011 11110011 01101000
11:22:10.393 c.TestBiased [t1] - synchronized 后
00000000 00000000 00000000 01101010 00000010 01001010 01100111 00000001
```

撤销 - 其它线程使用对象

当有其它线程使用偏向锁对象时，会将偏向锁升级为轻量级锁

```
private static void test2() throws InterruptedException {

    Dog d = new Dog();
    Thread t1 = new Thread(() -> {
        synchronized (d) {
            log.debug(ClassLayout.parseInstance(d).toPrintableSimple(true));
        }
        synchronized (TestBiased.class) {
            TestBiased.class.notify();
        }
        // 如果不用 wait/notify 使用 join 必须打开下面的注释
        // 因为：t1 线程不能结束，否则底层线程可能被 jvm 重用作为 t2 线程，底层线程 id 是一样的
        /*try {
            System.in.read();
        } catch (IOException e) {
            e.printStackTrace();
        }*/
    }, "t1");
    t1.start();

    Thread t2 = new Thread(() -> {
        synchronized (TestBiased.class) {
            try {
                TestBiased.class.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        log.debug(ClassLayout.parseInstance(d).toPrintableSimple(true));
        synchronized (d) {
            log.debug(ClassLayout.parseInstance(d).toPrintableSimple(true));
        }
        log.debug(ClassLayout.parseInstance(d).toPrintableSimple(true));
    }, "t2");
    t2.start();
}
```

输出

```
[t1] - 00000000 00000000 00000000 00000000 00011111 01000001 00010000 00000101
[t2] - 00000000 00000000 00000000 00000000 00011111 01000001 00010000 00000101
[t2] - 00000000 00000000 00000000 00000000 00011111 10110101 11110000 01000000
[t2] - 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001
```

撤销 - 调用 wait/notify

```
public static void main(String[] args) throws InterruptedException {
    Dog d = new Dog();

    Thread t1 = new Thread(() -> {
        log.debug(ClassLayout.parseInstance(d).toPrintableSimple(true));
        synchronized (d) {
            log.debug(ClassLayout.parseInstance(d).toPrintableSimple(true));
            try {
                d.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            log.debug(ClassLayout.parseInstance(d).toPrintableSimple(true));
        }
    }, "t1");
    t1.start();

    new Thread(() -> {
        try {
            Thread.sleep(6000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        synchronized (d) {
            log.debug("notify");
            d.notify();
        }
    }, "t2").start();
}
```

输出

```
[t1] - 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000101
[t1] - 00000000 00000000 00000000 00000000 00011111 10110011 11111000 00000101
[t2] - notify
[t1] - 00000000 00000000 00000000 00000000 00011100 11010100 00001101 11001010
```

批量重偏向

如果对象虽然被多个线程访问，但没有竞争，这时偏向了线程 T1 的对象仍有机会重新偏向 T2，重偏向会重置对象的 Thread ID

当撤销偏向锁阈值超过 20 次后，jvm 会这样觉得，我是不是偏向错了呢，于是会在给这些对象加锁时重新偏向至加锁线程

```
private static void test3() throws InterruptedException {  
  
    Vector<Dog> list = new Vector<>();  
    Thread t1 = new Thread(() -> {  
        for (int i = 0; i < 30; i++) {  
            Dog d = new Dog();  
            list.add(d);  
            synchronized (d) {  
                log.debug(i + "\t" + ClassLayout.parseInstance(d).toPrintableSimple(true));  
            }  
        }  
        synchronized (list) {  
            list.notify();  
        }  
    }, "t1");  
    t1.start();  
  
    Thread t2 = new Thread(() -> {  
        synchronized (list) {  
            try {  
                list.wait();  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
        log.debug("=====> ");  
        for (int i = 0; i < 30; i++) {  
            Dog d = list.get(i);  
            log.debug(i + "\t" + ClassLayout.parseInstance(d).toPrintableSimple(true));  
            synchronized (d) {  
                log.debug(i + "\t" + ClassLayout.parseInstance(d).toPrintableSimple(true));  
            }  
            log.debug(i + "\t" + ClassLayout.parseInstance(d).toPrintableSimple(true));  
        }  
    }, "t2");  
    t2.start();  
}
```

输出

[illegible]

[illegible]

```
[t2] - 25      00000000 00000000 00000000 00000000 00011111 11110011 11100000 00000101
[t2] - 25      00000000 00000000 00000000 00000000 00011111 11110011 11110001 00000101
[t2] - 25      00000000 00000000 00000000 00000000 00011111 11110011 11110001 00000101
[t2] - 26      00000000 00000000 00000000 00000000 00011111 11110011 11100000 00000101
[t2] - 26      00000000 00000000 00000000 00000000 00011111 11110011 11110001 00000101
[t2] - 26      00000000 00000000 00000000 00000000 00011111 11110011 11110001 00000101
[t2] - 27      00000000 00000000 00000000 00000000 00011111 11110011 11100000 00000101
[t2] - 27      00000000 00000000 00000000 00000000 00011111 11110011 11110001 00000101
[t2] - 27      00000000 00000000 00000000 00000000 00011111 11110011 11110001 00000101
[t2] - 28      00000000 00000000 00000000 00000000 00011111 11110011 11100000 00000101
[t2] - 28      00000000 00000000 00000000 00000000 00011111 11110011 11110001 00000101
[t2] - 28      00000000 00000000 00000000 00000000 00011111 11110011 11110001 00000101
[t2] - 29      00000000 00000000 00000000 00000000 00011111 11110011 11100000 00000101
[t2] - 29      00000000 00000000 00000000 00000000 00011111 11110011 11110001 00000101
[t2] - 29      00000000 00000000 00000000 00000000 00011111 11110011 11110001 00000101
```

批量撤销

当撤销偏向锁阈值超过 40 次后，jvm 会这样觉得，自己确实偏向错了，根本就不该偏向。于是整个类的所有对象都会变为不可偏向的，新建的对象也是不可偏向的

```
static Thread t1,t2,t3;
private static void test4() throws InterruptedException {
    Vector<Dog> list = new Vector<>();

    int loopNumber = 39;
    t1 = new Thread(() -> {
        for (int i = 0; i < loopNumber; i++) {
            Dog d = new Dog();
            list.add(d);
            synchronized (d) {
                log.debug(i + "\t" + ClassLayout.parseInstance(d).toPrintableSimple(true));
            }
        }
        LockSupport.unpark(t2);
    }, "t1");
    t1.start();

    t2 = new Thread(() -> {
        LockSupport.park();
        log.debug("=====> ");
        for (int i = 0; i < loopNumber; i++) {
            Dog d = list.get(i);
            log.debug(i + "\t" + ClassLayout.parseInstance(d).toPrintableSimple(true));
            synchronized (d) {
                log.debug(i + "\t" + ClassLayout.parseInstance(d).toPrintableSimple(true));
            }
            log.debug(i + "\t" + ClassLayout.parseInstance(d).toPrintableSimple(true));
        }
        LockSupport.unpark(t3);
    }, "t2");
```



```
t2.start();

t3 = new Thread(() -> {
    LockSupport.park();
    log.debug("=====> ");
    for (int i = 0; i < loopNumber; i++) {
        Dog d = list.get(i);
        log.debug(i + "\t" + ClassLayout.parseInstance(d).toPrintableSimple(true));
        synchronized (d) {
            log.debug(i + "\t" + ClassLayout.parseInstance(d).toPrintableSimple(true));
        }
        log.debug(i + "\t" + ClassLayout.parseInstance(d).toPrintableSimple(true));
    }
}, "t3");
t3.start();

t3.join();
log.debug(ClassLayout.parseInstance(new Dog()).toPrintableSimple(true));
}
```

参考资料

<https://github.com/farmerjohngit/myblog/issues/12>

<https://www.cnblogs.com/LemonFive/p/11246086.html>

<https://www.cnblogs.com/LemonFive/p/11248248.html>

[偏向锁论文](#)

5. 锁消除

锁消除

```
@Fork(1)
@BenchmarkMode(Mode.AverageTime)
@Warmup(iterations=3)
@Measurement(iterations=5)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
public class MyBenchmark {
    static int x = 0;
    @Benchmark
    public void a() throws Exception {
        x++;
    }
    @Benchmark
    public void b() throws Exception {
        Object o = new Object();
        synchronized (o) {
            x++;
        }
    }
}
```



```
}  
}  
}
```

```
java -jar benchmarks.jar
```

Benchmark	Mode	Samples	Score	Score error	Units
c.i.MyBenchmark.a	avgt	5	1.542	0.056	ns/op
c.i.MyBenchmark.b	avgt	5	1.518	0.091	ns/op

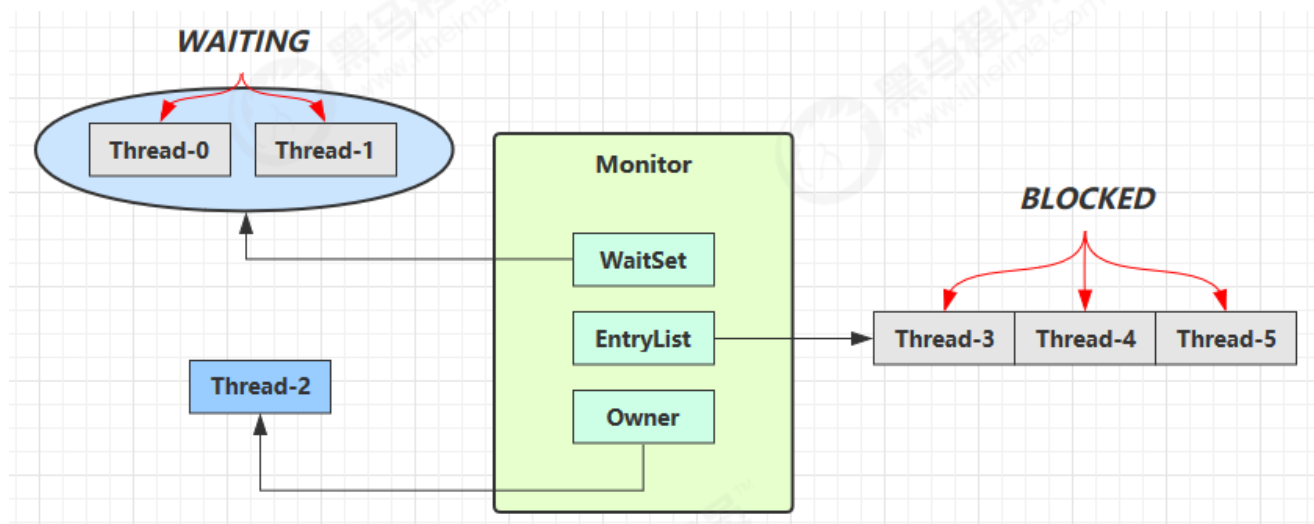
```
java -XX:-EliminateLocks -jar benchmarks.jar
```

Benchmark	Mode	Samples	Score	Score error	Units
c.i.MyBenchmark.a	avgt	5	1.507	0.108	ns/op
c.i.MyBenchmark.b	avgt	5	16.976	1.572	ns/op

锁粗化

对相同对象多次加锁，导致线程发生多次重入，可以使用锁粗化方式来优化，这不同于之前讲的细分锁的粒度。

wait notify 原理



- Owner 线程发现条件不满足，调用 wait 方法，即可进入 WaitSet 变为 WAITING 状态
- BLOCKED 和 WAITING 的线程都处于阻塞状态，不占用 CPU 时间片
- BLOCKED 线程会在 Owner 线程释放锁时唤醒
- WAITING 线程会在 Owner 线程调用 notify 或 notifyAll 时唤醒，但唤醒后并不意味着立刻获得锁，仍需进入 EntryList 重新竞争

join 原理

是调用者轮询检查线程 alive 状态

```
t1.join();
```

等价于下面的代码

```
synchronized (t1) {  
    // 调用者线程进入 t1 的 waitSet 等待, 直到 t1 运行结束  
    while (t1.isAlive()) {  
        t1.wait(0);  
    }  
}
```

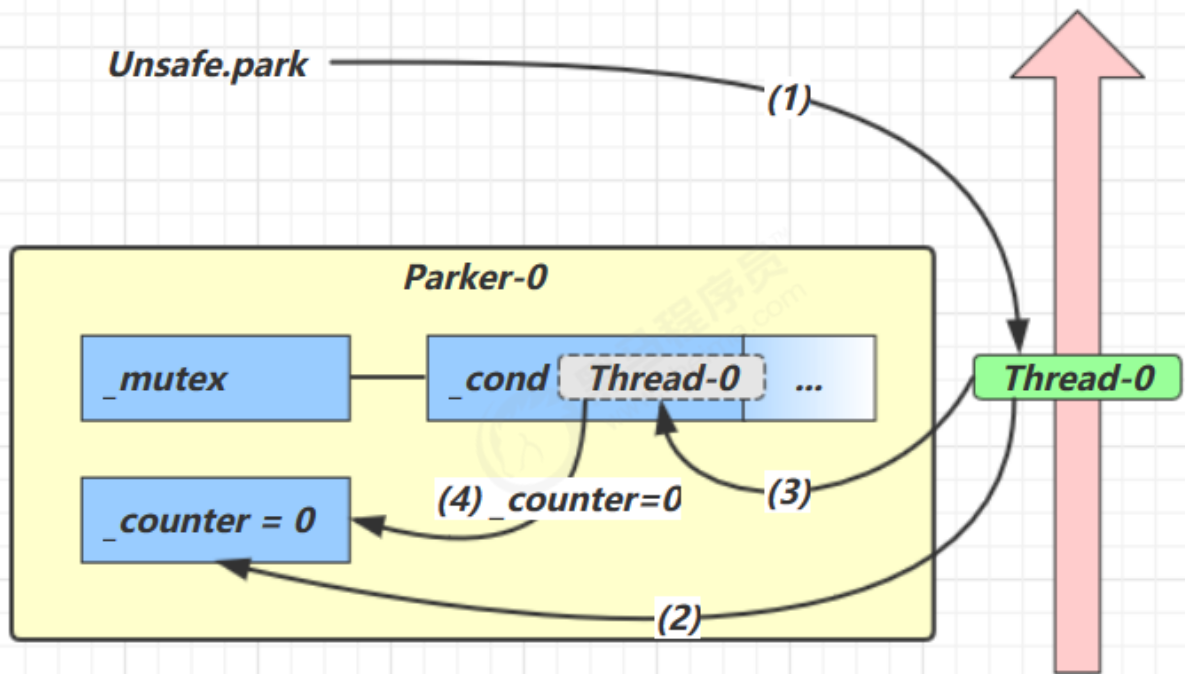
注意

join 体现的是【保护性暂停】模式，请参考之

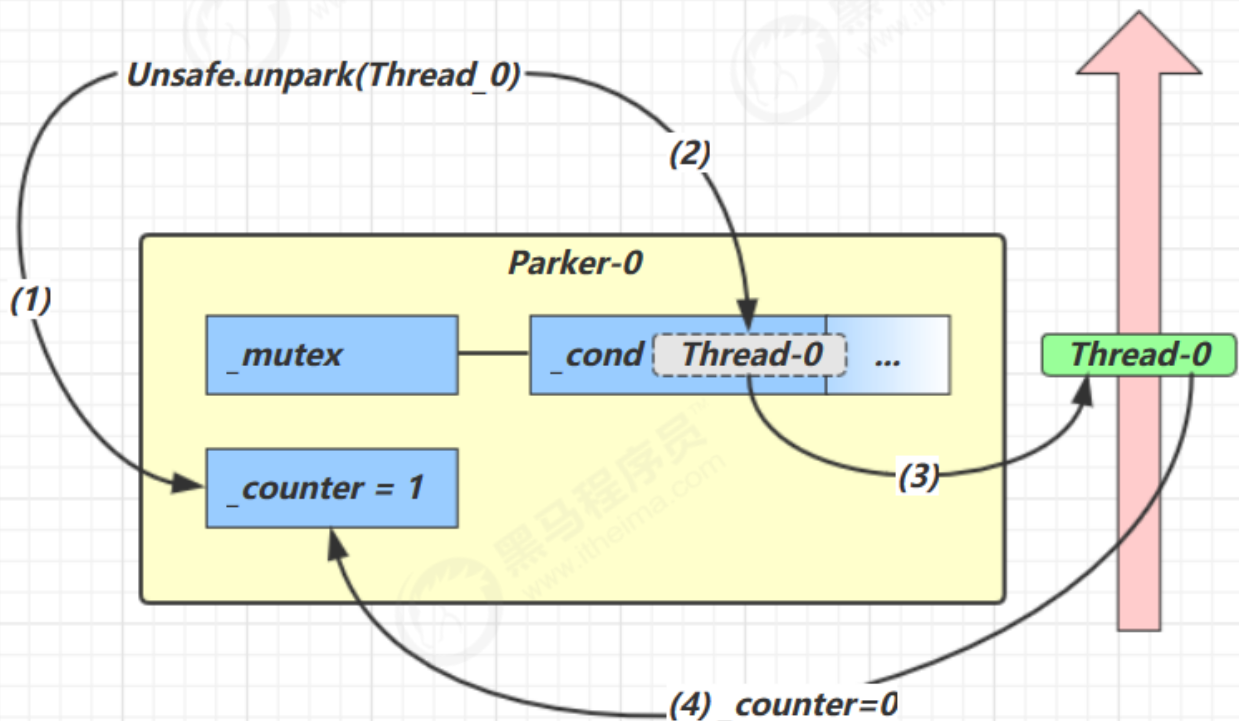
park unpark 原理

每个线程都有自己的一个 Parker 对象，由三部分组成 `_counter`，`_cond` 和 `_mutex` 打个比喻

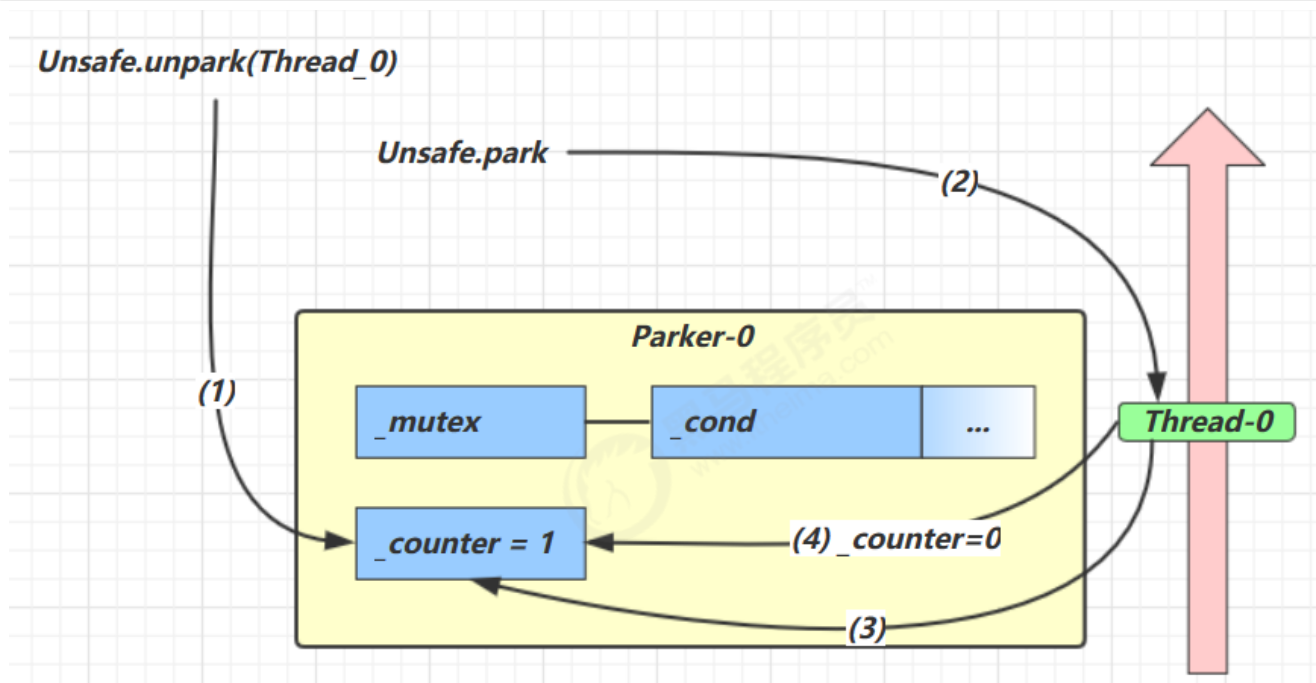
- 线程就像一个旅人，Parker 就像他随身携带的背包，条件变量就好比背包中的帐篷。`_counter` 就好比背包中的备用干粮（0 为耗尽，1 为充足）
- 调用 park 就是要看需不需要停下来歇息
 - 如果备用干粮耗尽，那么钻进帐篷歇息
 - 如果备用干粮充足，那么不需停留，继续前进
- 调用 unpark，就好比令干粮充足
 - 如果这时线程还在帐篷，就唤醒让他继续前进
 - 如果这时线程还在运行，那么下次他调用 park 时，仅是消耗掉备用干粮，不需停留继续前进
 - 因为背包空间有限，多次调用 unpark 仅会补充一份备用干粮



1. 当前线程调用 `Unsafe.park()` 方法
2. 检查 `_counter`，本情况为 0，这时，获得 `_mutex` 互斥锁
3. 线程进入 `_cond` 条件变量阻塞
4. 设置 `_counter = 0`



1. 调用 `Unsafe.unpark(Thread_0)` 方法，设置 `_counter` 为 1
2. 唤醒 `_cond` 条件变量中的 `Thread_0`
3. `Thread_0` 恢复运行
4. 设置 `_counter` 为 0



1. 调用 `Unsafe.unpark(Thread_0)` 方法，设置 `_counter` 为 1
2. 当前线程调用 `Unsafe.park()` 方法
3. 检查 `_counter`，本情况为 1，这时线程无需阻塞，继续运行
4. 设置 `_counter` 为 0

AQS 原理

1. 概述

全称是 `AbstractQueuedSynchronizer`，是阻塞式锁和相关的同步器工具的框架

特点：

- 用 `state` 属性来表示资源的状态（分独占模式和共享模式），子类需要定义如何维护这个状态，控制如何获取锁和释放锁
 - `getState` - 获取 `state` 状态
 - `setState` - 设置 `state` 状态
 - `compareAndSetState` - cas 机制设置 `state` 状态
 - 独占模式是只有一个线程能够访问资源，而共享模式可以允许多个线程访问资源
- 提供了基于 FIFO 的等待队列，类似于 Monitor 的 `EntryList`
- 条件变量来实现等待、唤醒机制，支持多个条件变量，类似于 Monitor 的 `WaitSet`

子类主要实现这样一些方法（默认抛出 `UnsupportedOperationException`）

- `tryAcquire`
- `tryRelease`
- `tryAcquireShared`
- `tryReleaseShared`
- `isHeldExclusively`

获取锁的姿势

```
// 如果获取锁失败
if (!tryAcquire(arg)) {
    // 入队，可以选择阻塞当前线程 park unpark
}
```

释放锁的姿势

```
// 如果释放锁成功
if (tryRelease(arg)) {
    // 让阻塞线程恢复运行
}
```

2. 实现不可重入锁

自定义同步器

```
final class MySync extends AbstractQueuedSynchronizer {
    @Override
    protected boolean tryAcquire(int acquires) {
        if (acquires == 1){
            if (compareAndSetState(0, 1)) {
                setExclusiveOwnerThread(Thread.currentThread());
                return true;
            }
        }
        return false;
    }

    @Override
    protected boolean tryRelease(int acquires) {
        if(acquires == 1) {
            if(getState() == 0) {
                throw new IllegalMonitorStateException();
            }
            setExclusiveOwnerThread(null);
            setState(0);
            return true;
        }
        return false;
    }

    protected Condition newCondition() {
        return new ConditionObject();
    }

    @Override
    protected boolean isHeldExclusively() {
        return getState() == 1;
    }
}
```

```
}  
}
```

自定义锁

有了自定义同步器，很容易复用 AQS，实现一个功能完备的自定义锁

```
class MyLock implements Lock {  
  
    static MySync sync = new MySync();  
  
    @Override  
    // 尝试，不成功，进入等待队列  
    public void lock() {  
        sync.acquire(1);  
    }  
  
    @Override  
    // 尝试，不成功，进入等待队列，可打断  
    public void lockInterruptibly() throws InterruptedException {  
        sync.acquireInterruptibly(1);  
    }  
  
    @Override  
    // 尝试一次，不成功返回，不进入队列  
    public boolean tryLock() {  
        return sync.tryAcquire(1);  
    }  
  
    @Override  
    // 尝试，不成功，进入等待队列，有时限  
    public boolean tryLock(long time, TimeUnit unit) throws InterruptedException {  
        return sync.tryAcquireNanos(1, unit.toNanos(time));  
    }  
  
    @Override  
    // 释放锁  
    public void unlock() {  
        sync.release(1);  
    }  
  
    @Override  
    // 生成条件变量  
    public Condition newCondition() {  
        return sync.newCondition();  
    }  
}
```

测试一下

```
MyLock lock = new MyLock();
```

```
new Thread(() -> {
    lock.lock();
    try {
        log.debug("locking...");
        sleep(1);
    } finally {
        log.debug("unlocking...");
        lock.unlock();
    }
}, "t1").start();

new Thread(() -> {
    lock.lock();
    try {
        log.debug("locking...");
    } finally {
        log.debug("unlocking...");
        lock.unlock();
    }
}, "t2").start();
```

输出

```
22:29:28.727 c.TestAqs [t1] - locking...
22:29:29.732 c.TestAqs [t1] - unlocking...
22:29:29.732 c.TestAqs [t2] - locking...
22:29:29.732 c.TestAqs [t2] - unlocking...
```

不可重入测试

如果改为下面代码，会发现自己也会被挡住（只会打印一次 locking）

```
lock.lock();
log.debug("locking...");
lock.lock();
log.debug("locking...");
```

3. 心得

起源

早期程序员会自己通过一种同步器去实现另一种相近的同步器，例如用可重入锁去实现信号量，或反之。这显然不够优雅，于是在 JSR166（java 规范提案）中创建了 AQS，提供了这种通用的同步器机制。

目标

AQS 要实现的功能目标

- 阻塞版本获取锁 acquire 和非阻塞的版本尝试获取锁 tryAcquire
- 获取锁超时机制

- 通过打断取消机制
- 独占机制及共享机制
- 条件不满足时的等待机制

要实现的性能目标

Instead, the primary performance goal here is scalability: to predictably maintain efficiency even, or especially, when synchronizers are contended.

设计

AQS 的基本思想其实很简单

获取锁的逻辑

```
while(state 状态不允许获取) {  
    if(队列中还没有此线程) {  
        入队并阻塞  
    }  
}  
当前线程出队
```

释放锁的逻辑

```
if(state 状态允许了) {  
    恢复阻塞的线程(s)  
}
```

要点

- 原子维护 state 状态
- 阻塞及恢复线程
- 维护队列

1) state 设计

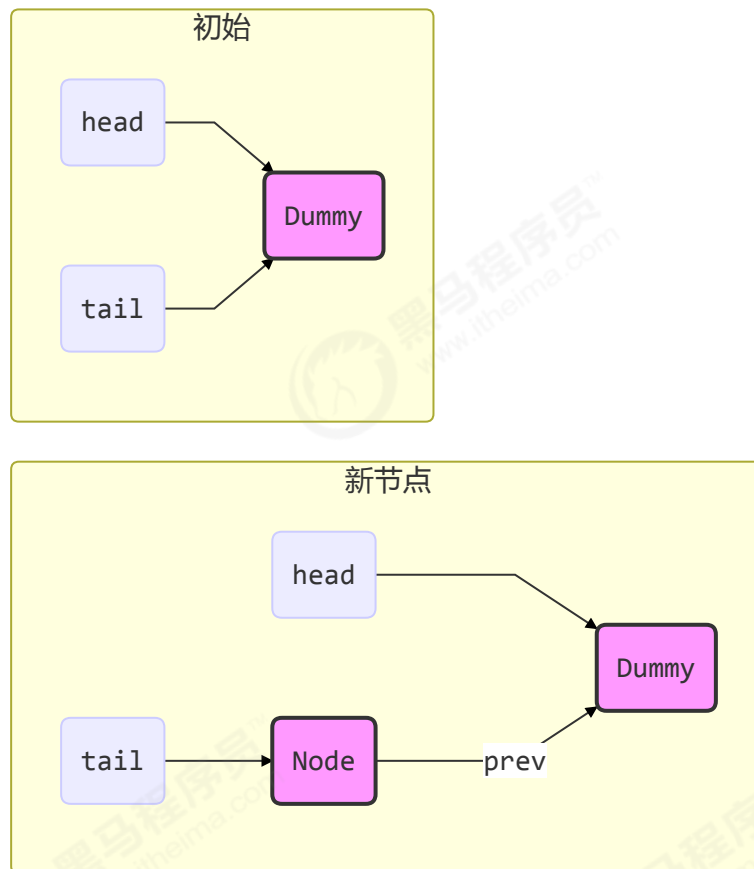
- state 使用 volatile 配合 cas 保证其修改时的原子性
- state 使用了 32bit int 来维护同步状态，因为当时使用 long 在很多平台下测试的结果并不理想

2) 阻塞恢复设计

- 早期的控制线程暂停和恢复的 api 有 suspend 和 resume，但它们是不可用的，因为如果先调用的 resume 那么 suspend 将感知不到
- 解决方法是使用 park & unpark 来实现线程的暂停和恢复，具体原理在之前讲过了，先 unpark 再 park 也没问题
- park & unpark 是针对线程的，而不是针对同步器的，因此控制粒度更为精细
- park 线程还可以通过 interrupt 打断

3) 队列设计

- 使用了 FIFO 先入先出队列，并不支持优先级队列
- 设计时借鉴了 CLH 队列，它是一种单向无锁队列



队列中有 head 和 tail 两个指针节点，都用 volatile 修饰配合 cas 使用，每个节点有 state 维护节点状态

入队伪代码，只需要考虑 tail 赋值的原子性

```
do {  
    // 原来的 tail  
    Node prev = tail;  
    // 用 cas 在原来 tail 的基础上改为 node  
} while(tail.compareAndSet(prev, node))
```

出队伪代码

```
// prev 是上一个节点  
while((Node prev=node.prev).state != 唤醒状态) {  
}  
// 设置头节点  
head = node;
```

CLH 好处：

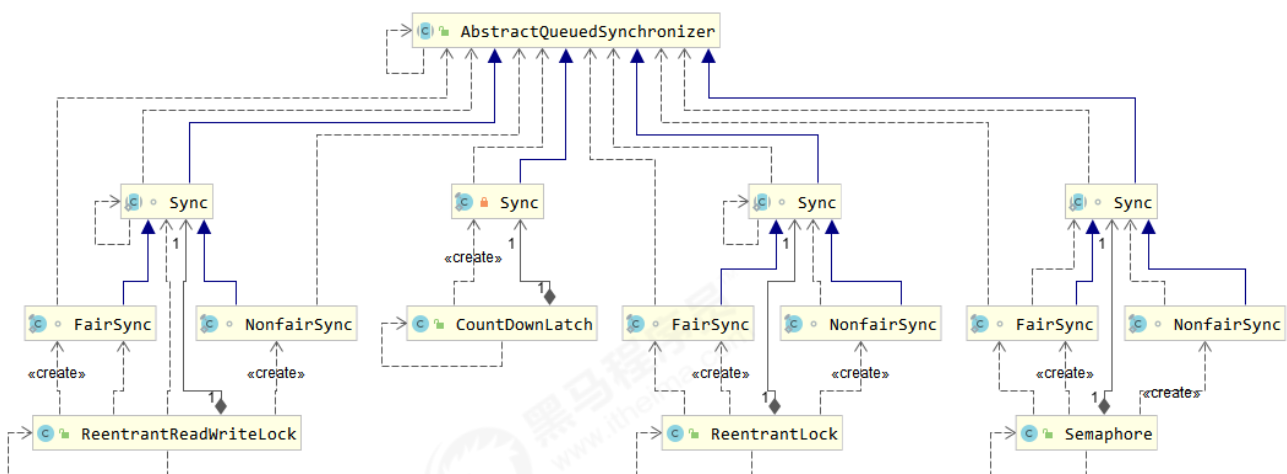
- 无锁，使用自旋

- 快速，无阻塞

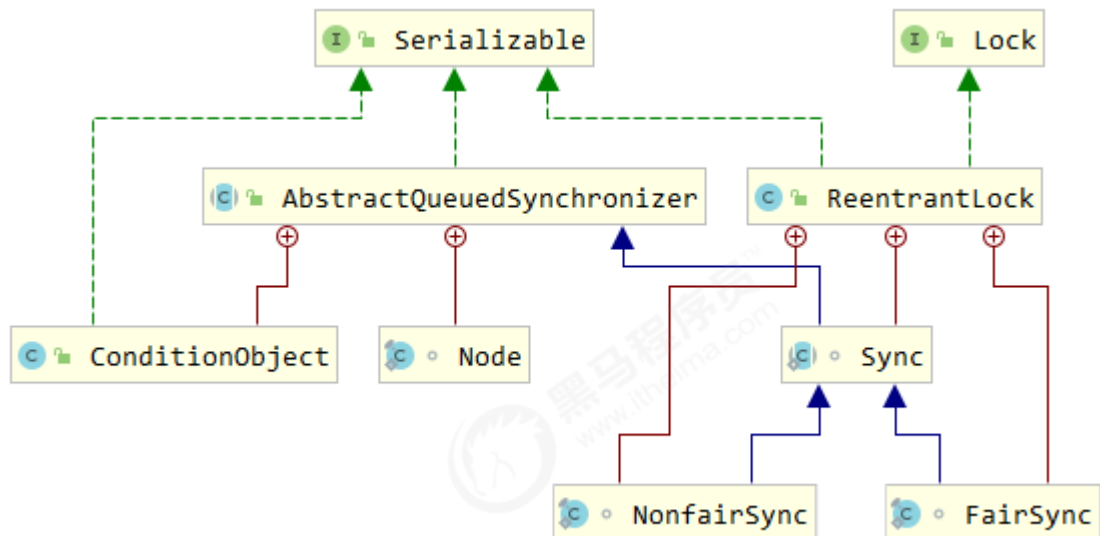
AQS 在一些方面改进了 CLH

```
private Node enq(final Node node) {
    for (;;) {
        Node t = tail;
        // 队列中还没有元素 tail 为 null
        if (t == null) {
            // 将 head 从 null -> dummy
            if (compareAndSetHead(new Node()))
                tail = head;
        } else {
            // 将 node 的 prev 设置为原来的 tail
            node.prev = t;
            // 将 tail 从原来的 tail 设置为 node
            if (compareAndSetTail(t, node)) {
                // 原来 tail 的 next 设置为 node
                t.next = node;
                return t;
            }
        }
    }
}
```

主要用到 AQS 的 并发工具类



ReentrantLock 原理



1. 非公平锁实现原理

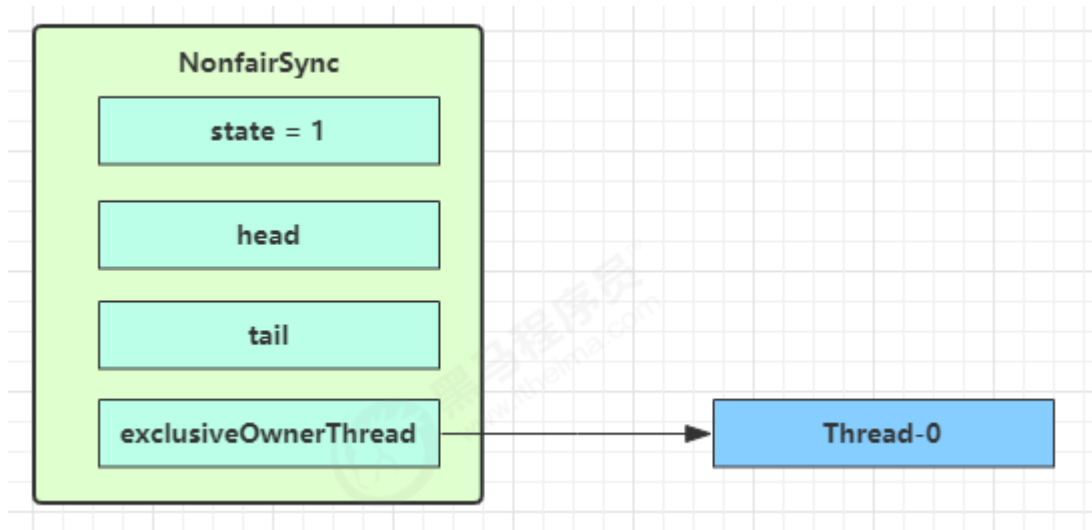
加锁解锁流程

先从构造器开始看，默认为非公平锁实现

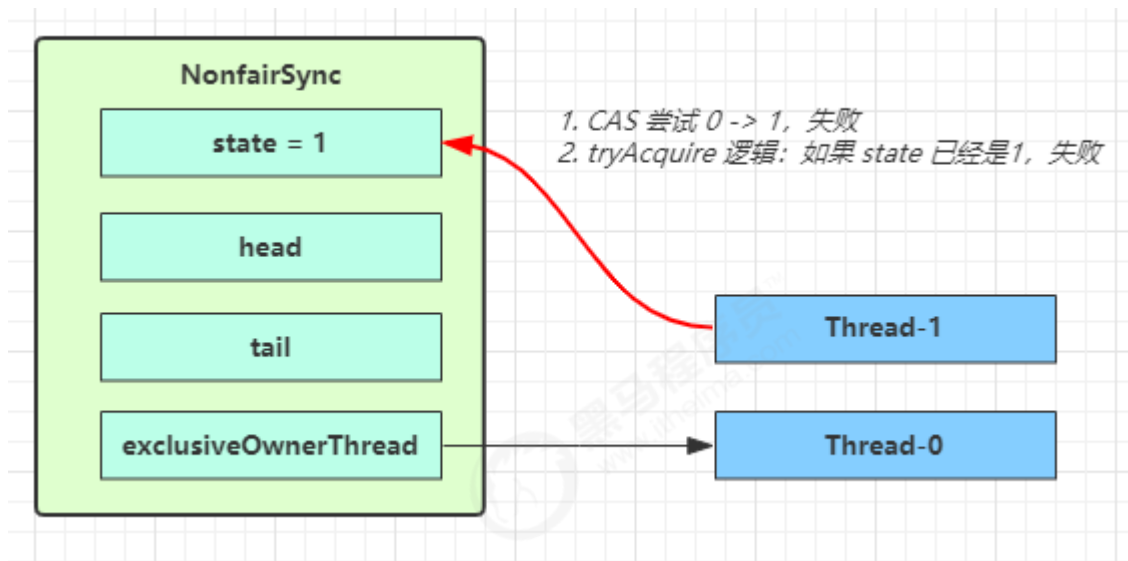
```
public ReentrantLock() {
    sync = new NonfairSync();
}
```

NonfairSync 继承自 AQS

没有竞争时

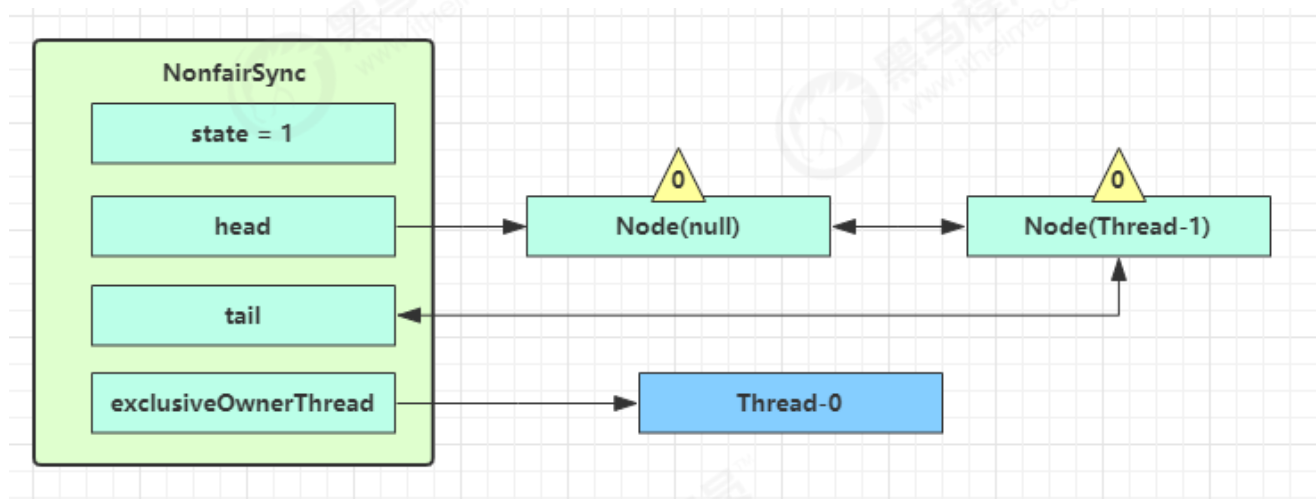


第一个竞争出现时



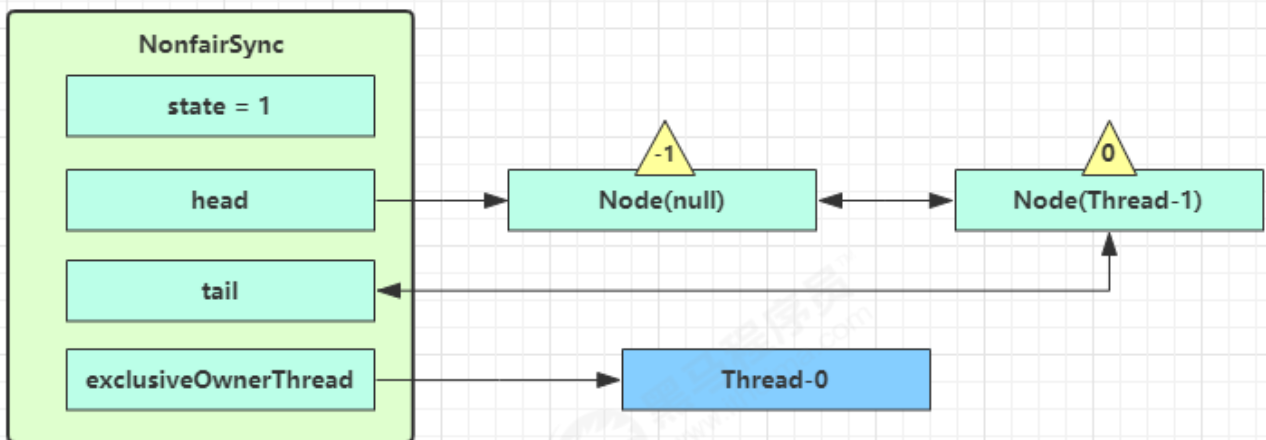
Thread-1 执行了

1. CAS 尝试将 state 由 0 改为 1，结果失败
2. 进入 tryAcquire 逻辑，这时 state 已经是1，结果仍然失败
3. 接下来进入 addWaiter 逻辑，构造 Node 队列
 - 图中黄色三角表示该 Node 的 waitStatus 状态，其中 0 为默认正常状态
 - Node 的创建是懒惰的
 - 其中第一个 Node 称为 Dummy（哑元）或哨兵，用来占位，并不关联线程

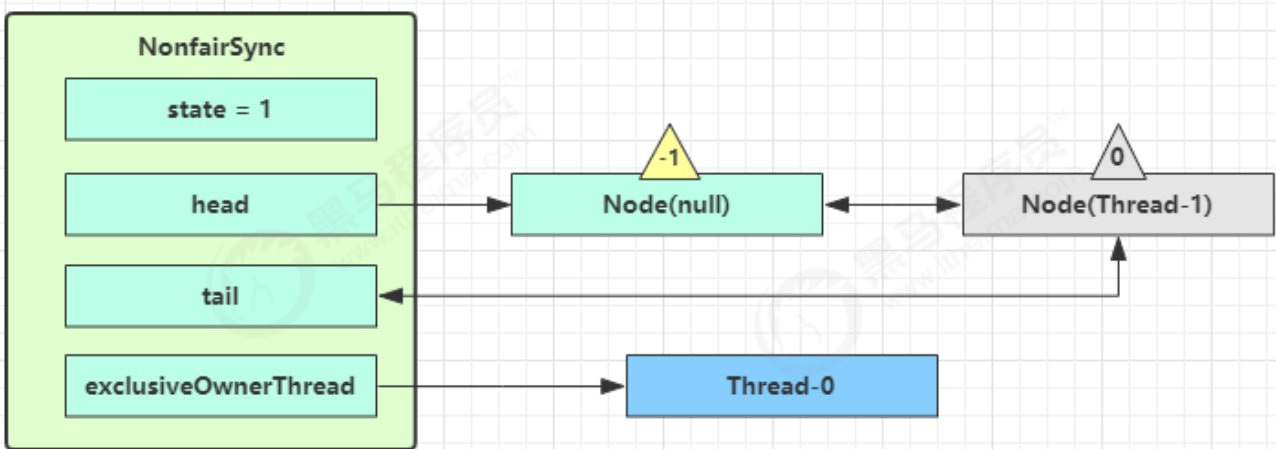


当前线程进入 acquireQueued 逻辑

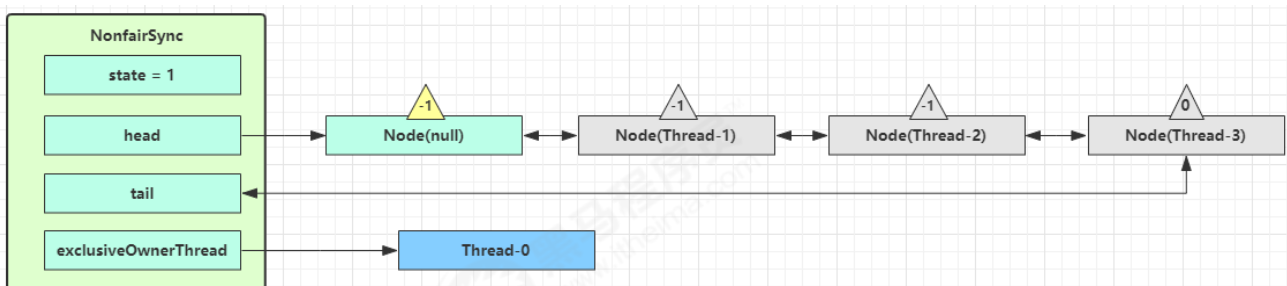
1. acquireQueued 会在一个死循环中不断尝试获得锁，失败后进入 park 阻塞
2. 如果自己是紧邻着 head（排第二位），那么再次 tryAcquire 尝试获取锁，当然这时 state 仍为 1，失败
3. 进入 shouldParkAfterFailedAcquire 逻辑，将前驱 node，即 head 的 waitStatus 改为 -1，这次返回 false



4. `shouldParkAfterFailedAcquire` 执行完毕回到 `acquireQueued`，再次 `tryAcquire` 尝试获取锁，当然这时 `state` 仍为 1，失败
5. 当再次进入 `shouldParkAfterFailedAcquire` 时，这时因为其前驱 `node` 的 `waitStatus` 已经是 -1，这次返回 `true`
6. 进入 `parkAndCheckInterrupt`，`Thread-1` `park`（灰色表示）

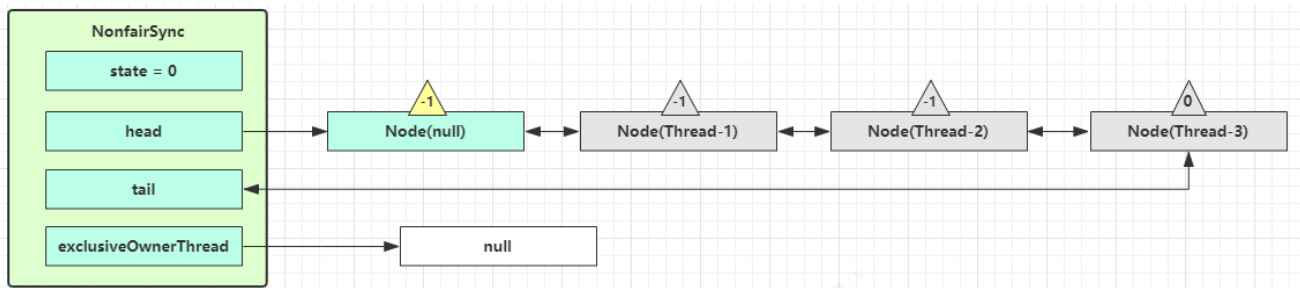


再次有多个线程经历上述过程竞争失败，变成这个样子



`Thread-0` 释放锁，进入 `tryRelease` 流程，如果成功

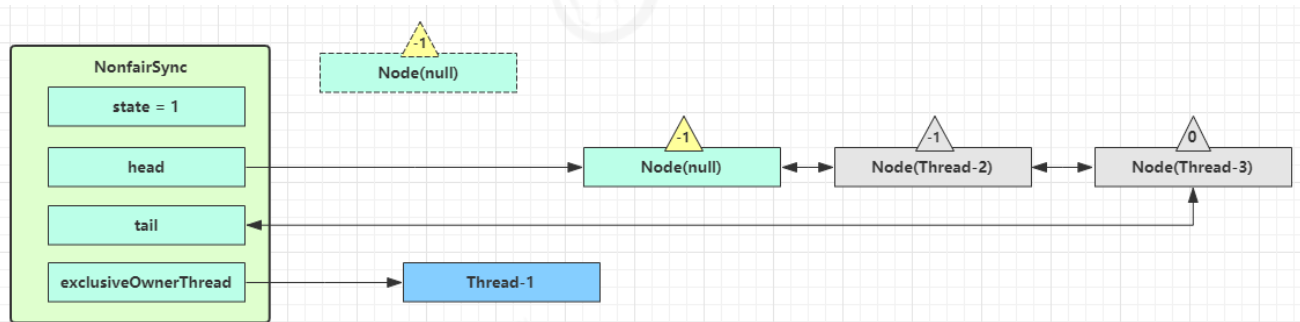
- 设置 `exclusiveOwnerThread` 为 `null`
- `state = 0`



当前队列不为 null，并且 head 的 waitStatus = -1，进入 unparkSuccessor 流程

找到队列中离 head 最近的一个 Node（没取消的），unpark 恢复其运行，本例中即为 Thread-1

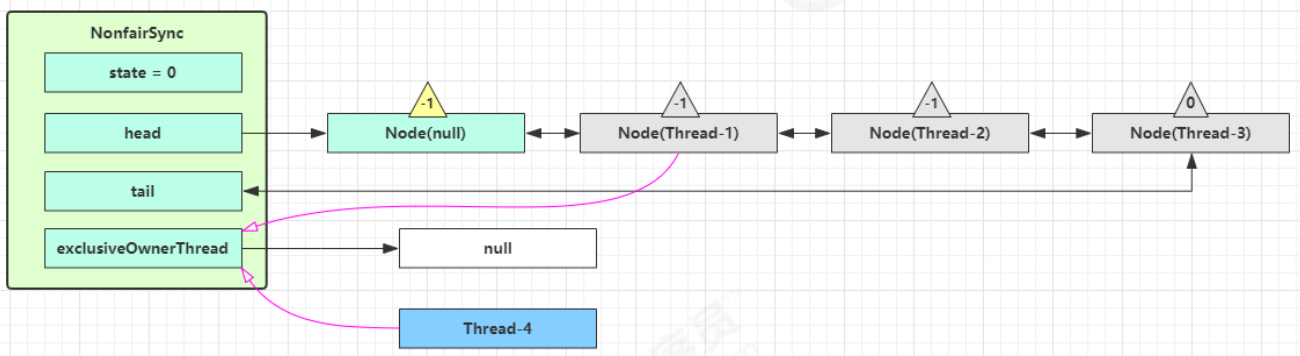
回到 Thread-1 的 acquireQueued 流程



如果加锁成功（没有竞争），会设置

- exclusiveOwnerThread 为 Thread-1，state = 1
- head 指向刚刚 Thread-1 所在的 Node，该 Node 清空 Thread
- 原本的 head 因为从链表断开，而可被垃圾回收

如果这时候有其它线程来竞争（不公平的体现），例如这时有 Thread-4 来了



如果不巧又被 Thread-4 占了先

- Thread-4 被设置为 exclusiveOwnerThread，state = 1
- Thread-1 再次进入 acquireQueued 流程，获取锁失败，重新进入 park 阻塞

加锁源码

```
// Sync 继承自 AQS
static final class NonfairSync extends Sync {
    private static final long serialVersionUID = 7316153563782823691L;

    // 加锁实现
```



```
final void lock() {
    // 首先用 cas 尝试 (仅尝试一次) 将 state 从 0 改为 1, 如果成功表示获得了独占锁
    if (compareAndSetState(0, 1))
        setExclusiveOwnerThread(Thread.currentThread());
    else
        // 如果尝试失败, 进入 (一)
        acquire(1);
}

// (一) AQS 继承过来的方法, 方便阅读, 放在此处
public final void acquire(int arg) {
    // (二) tryAcquire
    if (
        !tryAcquire(arg) &&
        // 当 tryAcquire 返回为 false 时, 先调用 addWaiter (四), 接着 acquireQueued (五)
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg)
    ) {
        selfInterrupt();
    }
}

// (三) 进入 (三)
protected final boolean tryAcquire(int acquires) {
    return nonfairTryAcquire(acquires);
}

// (四) Sync 继承过来的方法, 方便阅读, 放在此处
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    // 如果还没有获得锁
    if (c == 0) {
        // 尝试用 cas 获得, 这里体现了非公平性: 不去检查 AQS 队列
        if (compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    // 如果已经获得了锁, 线程还是当前线程, 表示发生了锁重入
    else if (current == getExclusiveOwnerThread()) {
        // state++
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    // 获取失败, 回到调用处
    return false;
}

// (五) AQS 继承过来的方法, 方便阅读, 放在此处
private Node addWaiter(Node mode) {
```



```
// 将当前线程关联到一个 Node 对象上，模式为独占模式
Node node = new Node(Thread.currentThread(), mode);
// 如果 tail 不为 null, cas 尝试将 Node 对象加入 AQS 队列尾部
Node pred = tail;
if (pred != null) {
    node.prev = pred;
    if (compareAndSetTail(pred, node)) {
        // 双向链表
        pred.next = node;
        return node;
    }
}
// 尝试将 Node 加入 AQS, 进入 (四)
enq(node);
return node;
}

// (四) AQS 继承过来的方法，方便阅读，放在此处
private Node enq(final Node node) {
    for (;;) {
        Node t = tail;
        if (t == null) {
            // 还没有，设置 head 为哨兵节点（不对应线程，状态为 0）
            if (compareAndSetHead(new Node())) {
                tail = head;
            }
        } else {
            // cas 尝试将 Node 对象加入 AQS 队列尾部
            node.prev = t;
            if (compareAndSetTail(t, node)) {
                t.next = node;
                return t;
            }
        }
    }
}

// (五) AQS 继承过来的方法，方便阅读，放在此处
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {
            final Node p = node.predecessor();
            // 上一个节点是 head，表示轮到自己（当前线程对应的 node）了，尝试获取
            if (p == head && tryAcquire(arg)) {
                // 获取成功，设置自己（当前线程对应的 node）为 head
                setHead(node);
                // 上一个节点 help GC
                p.next = null;
                failed = false;
                // 返回中断标记 false
                return interrupted;
            }
        }
    }
}
```




```
    }
    if (
        // 判断是否应当 park, 进入 (t)
        shouldParkAfterFailedAcquire(p, node) &&
        // park 等待, 此时 Node 的状态被置为 Node.SIGNAL (v)
        parkAndCheckInterrupt()
    ) {
        interrupted = true;
    }
}
} finally {
    if (failed)
        cancelAcquire(node);
}
}

// (t) AQS 继承过来的方法, 方便阅读, 放在此处
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
    // 获取上一个节点的状态
    int ws = pred.waitStatus;
    if (ws == Node.SIGNAL) {
        // 上一个节点都在阻塞, 那么自己也阻塞好了
        return true;
    }
    // > 0 表示取消状态
    if (ws > 0) {
        // 上一个节点取消, 那么重构删除前面所有取消的节点, 返回到外层循环重试
        do {
            node.prev = pred = pred.prev;
        } while (pred.waitStatus > 0);
        pred.next = node;
    } else {
        // 这次还没有阻塞
        // 但下次如果重试不成功, 则需要阻塞, 这时需要设置上一个节点状态为 Node.SIGNAL
        compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
    }
    return false;
}

// (v) 阻塞当前线程
private final boolean parkAndCheckInterrupt() {
    LockSupport.park(this);
    return Thread.interrupted();
}
}
```

注意

- 是否需要 unpark 是由当前节点的前驱节点的 waitStatus == Node.SIGNAL 来决定, 而不是本节点的 waitStatus 决定

解锁源码



```
// Sync 继承自 AQS
static final class NonfairSync extends Sync {
    // 解锁实现
    public void unlock() {
        sync.release(1);
    }

    // AQS 继承过来的方法，方便阅读，放在此处
    public final boolean release(int arg) {
        // 尝试释放锁，进入 (→)
        if (tryRelease(arg)) {
            // 队列头节点 unpark
            Node h = head;
            if (
                // 队列不为 null
                h != null &&
                // waitStatus == Node.SIGNAL 才需要 unpark
                h.waitStatus != 0
            ) {
                // unpark AQS 中等待的线程，进入 (→)
                unparkSuccessor(h);
            }
            return true;
        }
        return false;
    }
}

// (→) Sync 继承过来的方法，方便阅读，放在此处
protected final boolean tryRelease(int releases) {
    // state--
    int c = getState() - releases;
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    // 支持锁重入，只有 state 减为 0，才释放成功
    if (c == 0) {
        free = true;
        setExclusiveOwnerThread(null);
    }
    setState(c);
    return free;
}

// (→) AQS 继承过来的方法，方便阅读，放在此处
private void unparkSuccessor(Node node) {
    // 如果状态为 Node.SIGNAL 尝试重置状态为 0
    // 不成功也可以
    int ws = node.waitStatus;
    if (ws < 0) {
        compareAndSetWaitStatus(node, ws, 0);
    }

    // 找到需要 unpark 的节点，但本节点从 AQS 队列中脱离，是由唤醒节点完成的
}
```



```
Node s = node.next;
// 不考虑已取消的节点，从 AQS 队列从后至前找到队列最前面需要 unpark 的节点
if (s == null || s.waitStatus > 0) {
    s = null;
    for (Node t = tail; t != null && t != node; t = t.prev)
        if (t.waitStatus <= 0)
            s = t;
}
if (s != null)
    LockSupport.unpark(s.thread);
}
```

2. 可重入原理

```
static final class NonfairSync extends Sync {
    // ...

    // Sync 继承过来的方法，方便阅读，放在此处
    final boolean nonfairTryAcquire(int acquires) {
        final Thread current = Thread.currentThread();
        int c = getState();
        if (c == 0) {
            if (compareAndSetState(0, acquires)) {
                setExclusiveOwnerThread(current);
                return true;
            }
        }
        // 如果已经获得了锁，线程还是当前线程，表示发生了锁重入
        else if (current == getExclusiveOwnerThread()) {
            // state++
            int nextc = c + acquires;
            if (nextc < 0) // overflow
                throw new Error("Maximum lock count exceeded");
            setState(nextc);
            return true;
        }
        return false;
    }

    // Sync 继承过来的方法，方便阅读，放在此处
    protected final boolean tryRelease(int releases) {
        // state--
        int c = getState() - releases;
        if (Thread.currentThread() != getExclusiveOwnerThread())
            throw new IllegalMonitorStateException();
        boolean free = false;
        // 支持锁重入，只有 state 减为 0，才释放成功
        if (c == 0) {
            free = true;
            setExclusiveOwnerThread(null);
        }
    }
}
```

```
        setState(c);  
        return free;  
    }  
}
```

3. 可打断原理

不可打断模式

在此模式下，即使它被打断，仍会驻留在 AQS 队列中，一直要等到获得锁后方能得知自己被打断了

```
// Sync 继承自 AQS  
static final class NonfairSync extends Sync {  
    // ...  
  
    private final boolean parkAndCheckInterrupt() {  
        // 如果打断标记已经是 true, 则 park 会失效  
        LockSupport.park(this);  
        // interrupted 会清除打断标记  
        return Thread.interrupted();  
    }  
  
    final boolean acquireQueued(final Node node, int arg) {  
        boolean failed = true;  
        try {  
            boolean interrupted = false;  
            for (;;) {  
                final Node p = node.predecessor();  
                if (p == head && tryAcquire(arg)) {  
                    setHead(node);  
                    p.next = null;  
                    failed = false;  
                    // 还是需要获得锁后, 才能返回打断状态  
                    return interrupted;  
                }  
                if (  
                    shouldParkAfterFailedAcquire(p, node) &&  
                    parkAndCheckInterrupt()  
                ) {  
                    // 如果是因为 interrupt 被唤醒, 返回打断状态为 true  
                    interrupted = true;  
                }  
            }  
        } finally {  
            if (failed)  
                cancelAcquire(node);  
        }  
    }  
  
    public final void acquire(int arg) {
```



```
        if (
            !tryAcquire(arg) &&
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg)
        ) {
            // 如果打断状态为 true
            selfInterrupt();
        }
    }

    static void selfInterrupt() {
        // 重新产生一次中断
        Thread.currentThread().interrupt();
    }
}
```

可打断模式

```
static final class NonfairSync extends Sync {
    public final void acquireInterruptibly(int arg) throws InterruptedException {
        if (Thread.interrupted())
            throw new InterruptedException();
        // 如果没有获得锁，进入 (↩)
        if (!tryAcquire(arg))
            doAcquireInterruptibly(arg);
    }

    // (↩) 可打断的获取锁流程
    private void doAcquireInterruptibly(int arg) throws InterruptedException {
        final Node node = addWaiter(Node.EXCLUSIVE);
        boolean failed = true;
        try {
            for (;;) {
                final Node p = node.predecessor();
                if (p == head && tryAcquire(arg)) {
                    setHead(node);
                    p.next = null; // help GC
                    failed = false;
                    return;
                }
                if (shouldParkAfterFailedAcquire(p, node) &&
                    parkAndCheckInterrupt()) {
                    // 在 park 过程中如果被 interrupt 会进入此
                    // 这时候抛出异常，而不会再次进入 for (;;)
                    throw new InterruptedException();
                }
            }
        } finally {
            if (failed)
                cancelAcquire(node);
        }
    }
}
```



4. 公平锁实现原理

```
static final class FairSync extends Sync {  
    private static final long serialVersionUID = -3000897897090466540L;  
  
    final void lock() {  
        acquire(1);  
    }  
  
    // AQS 继承过来的方法，方便阅读，放在此处  
    public final void acquire(int arg) {  
        if (  
            !tryAcquire(arg) &&  
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg)  
        ) {  
            selfInterrupt();  
        }  
    }  
  
    // 与非公平锁主要区别在于 tryAcquire 方法的实现  
    protected final boolean tryAcquire(int acquires) {  
        final Thread current = Thread.currentThread();  
        int c = getState();  
        if (c == 0) {  
            // 先检查 AQS 队列中是否有前驱节点，没有才去竞争  
            if (!hasQueuedPredecessors() &&  
                compareAndSetState(0, acquires)) {  
                setExclusiveOwnerThread(current);  
                return true;  
            }  
        }  
        else if (current == getExclusiveOwnerThread()) {  
            int nextc = c + acquires;  
            if (nextc < 0)  
                throw new Error("Maximum lock count exceeded");  
            setState(nextc);  
            return true;  
        }  
        return false;  
    }  
  
    // (←) AQS 继承过来的方法，方便阅读，放在此处  
    public final boolean hasQueuedPredecessors() {  
        Node t = tail;  
        Node h = head;  
        Node s;  
        // h != t 时表示队列中有 Node  
        return h != t &&  
            (  
                // (s = h.next) == null 表示队列中还有没有老二  
                (s = h.next) == null ||
```

```
// 或者队列中老二线程不是此线程
s.thread != Thread.currentThread()

    );
}
}
```

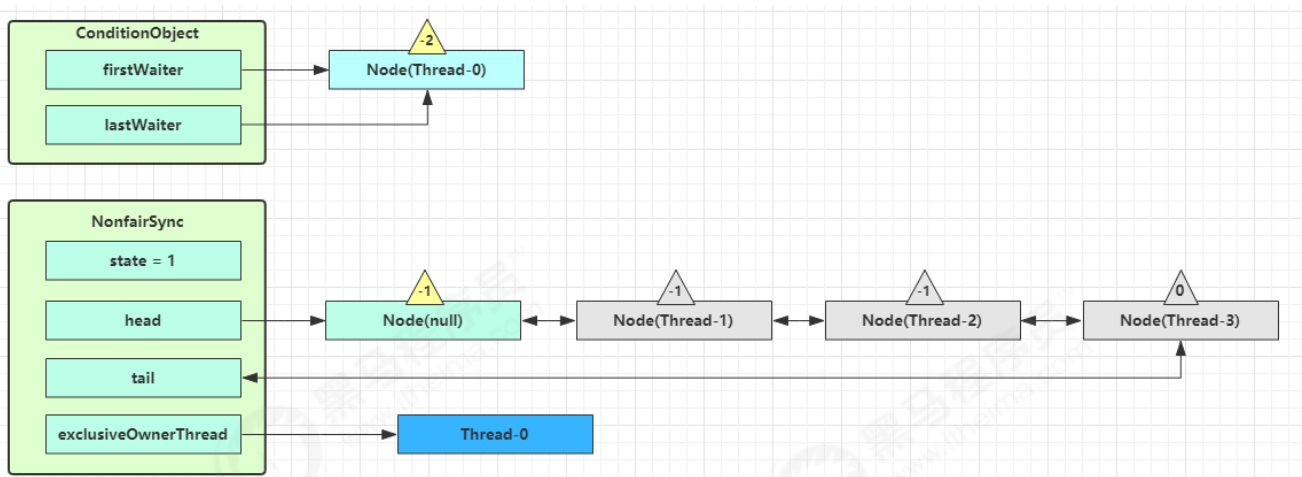
5. 条件变量实现原理

每个条件变量其实就对应着一个等待队列，其实现类是 ConditionObject

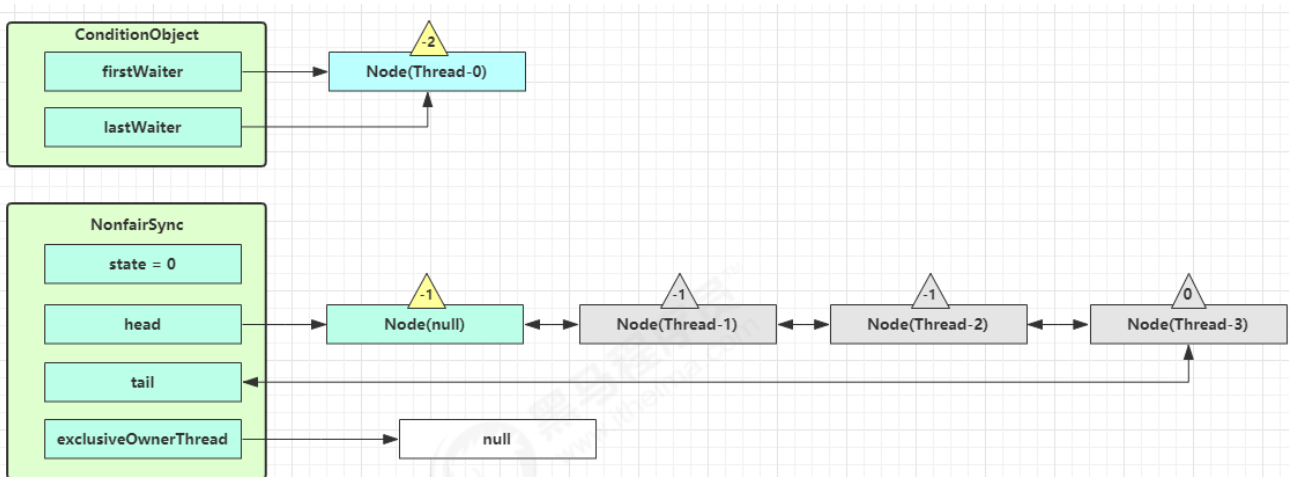
await 流程

开始 Thread-0 持有锁，调用 await，进入 ConditionObject 的 addConditionWaiter 流程

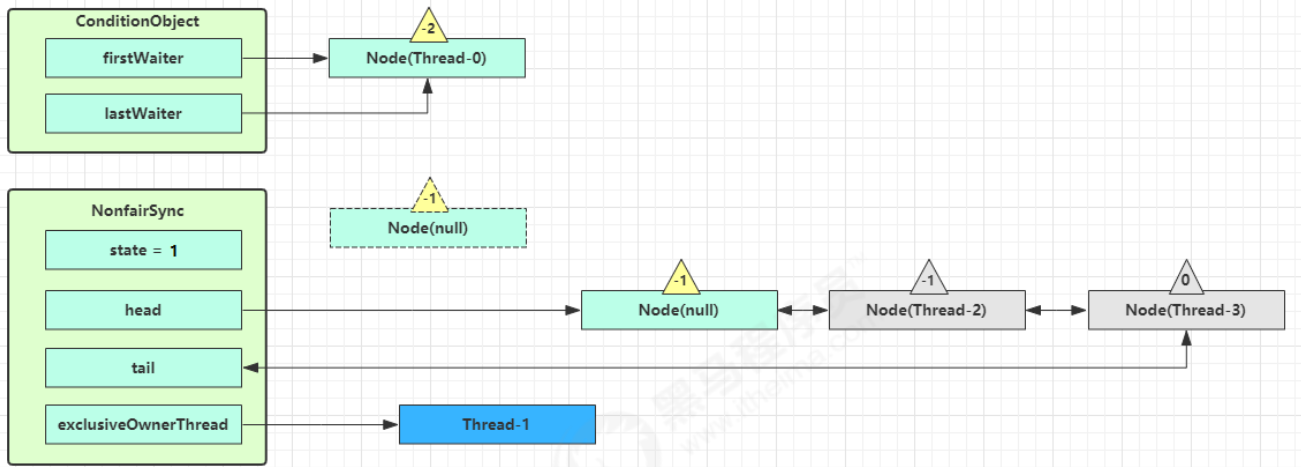
创建新的 Node 状态为 -2 (Node.CONDITION)，关联 Thread-0，加入等待队列尾部



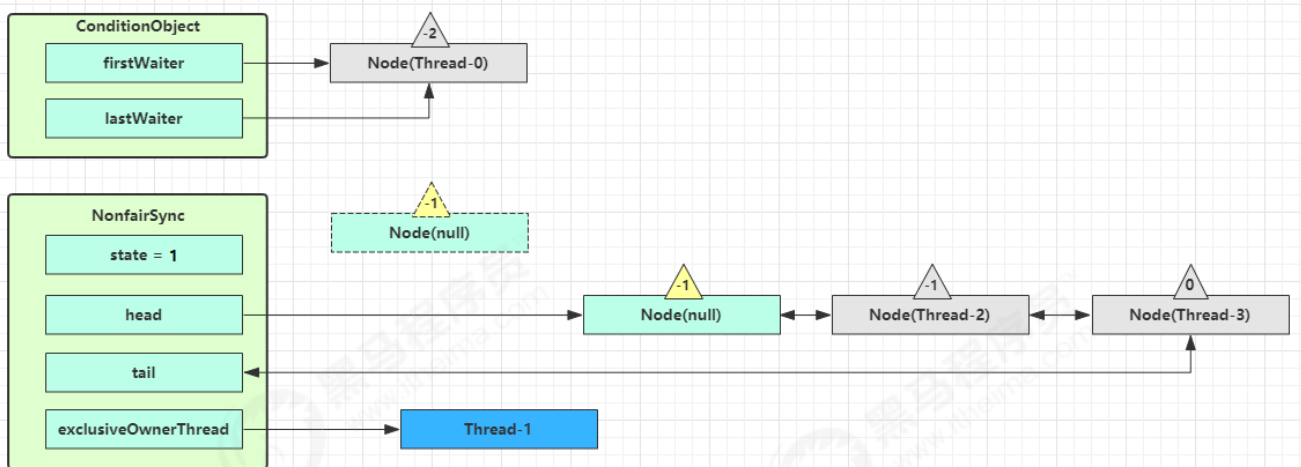
接下来进入 AQS 的 fullyRelease 流程，释放同步器上的锁



unpark AQS 队列中的下一个节点，竞争锁，假设没有其他竞争线程，那么 Thread-1 竞争成功

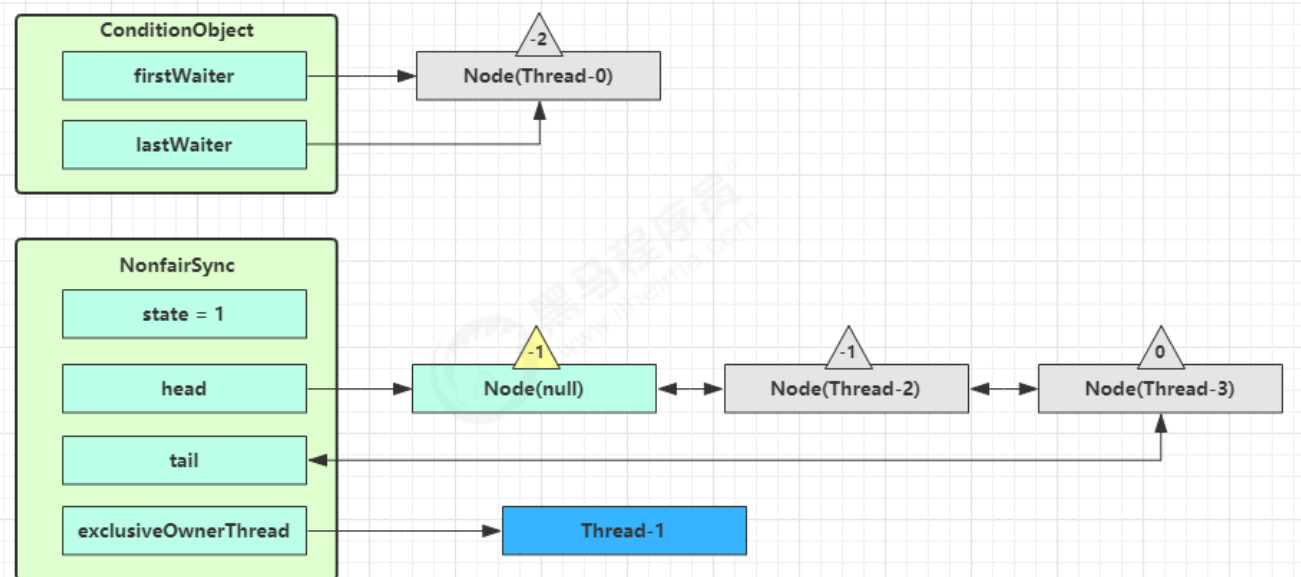


park 阻塞 Thread-0

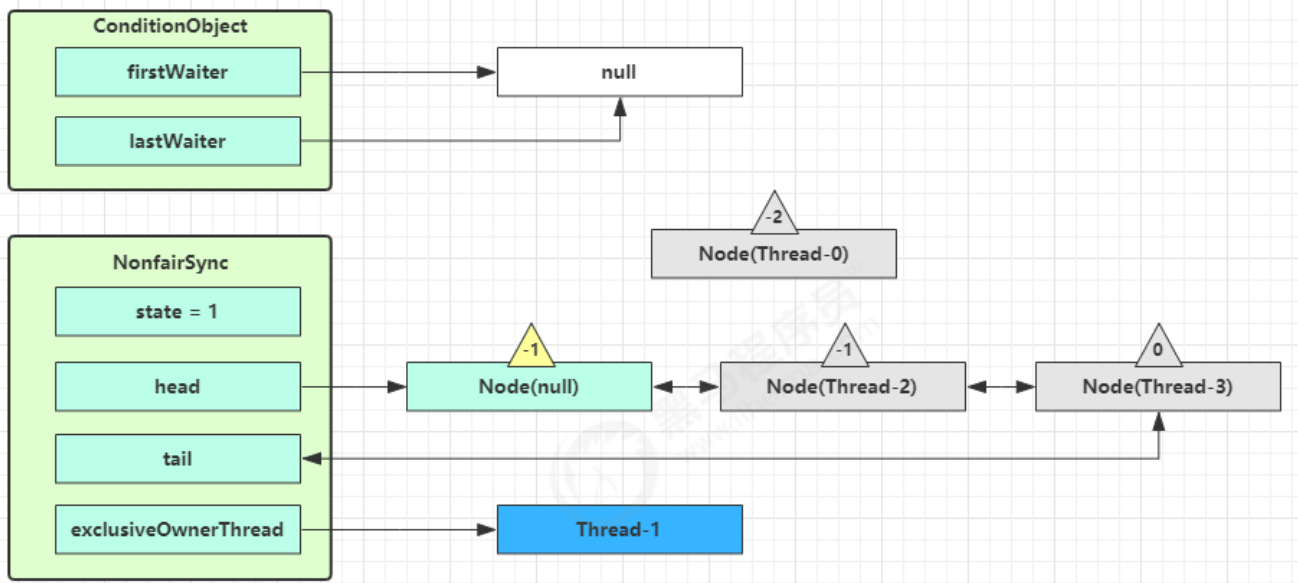


signal 流程

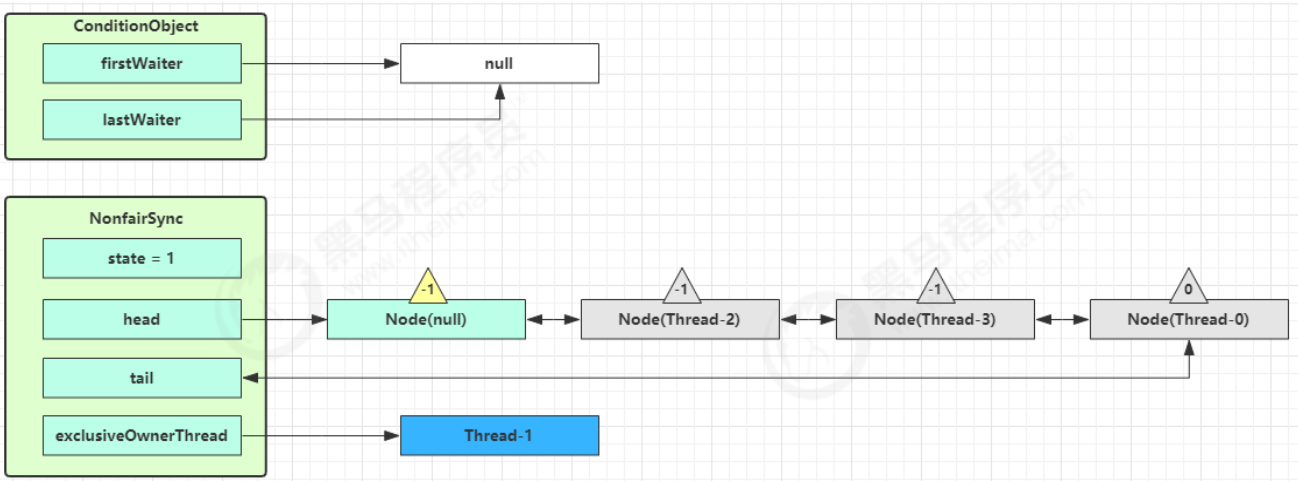
假设 Thread-1 要来唤醒 Thread-0



进入 ConditionObject 的 doSignal 流程，取得等待队列中第一个 Node，即 Thread-0 所在 Node



执行 transferForSignal 流程，将该 Node 加入 AQS 队列尾部，将 Thread-0 的 waitStatus 改为 0，Thread-3 的 waitStatus 改为 -1



Thread-1 释放锁，进入 unlock 流程，略

源码

```

public class ConditionObject implements Condition, java.io.Serializable {
    private static final long serialVersionUID = 1173984872572414699L;

    // 第一个等待节点
    private transient Node firstWaiter;

    // 最后一个等待节点
    private transient Node lastWaiter;

    public ConditionObject() { }

    // (⇐) 添加一个 Node 至等待队列
    private Node addConditionWaiter() {
        Node t = lastWaiter;

        // 所有已取消的 Node 从队列链表删除，见 (⇐)
    }
    
```



```
if (t != null && t.waitStatus != Node.CONDITION) {
    unlinkCancelledWaiters();
    t = lastWaiter;
}
// 创建一个关联当前线程的新 Node，添加至队列尾部
Node node = new Node(Thread.currentThread(), Node.CONDITION);
if (t == null)
    firstWaiter = node;
else
    t.nextWaiter = node;
lastWaiter = node;
return node;
}

// 唤醒 - 将没取消的第一个节点转移至 AQS 队列
private void doSignal(Node first) {
    do {
        // 已经是尾节点了
        if ( (firstWaiter = first.nextWaiter) == null) {
            lastWaiter = null;
        }
        first.nextWaiter = null;
    } while (
        // 将等待队列中的 Node 转移至 AQS 队列，不成功且还有节点则继续循环 (⇐)
        !transferForSignal(first) &&
        // 队列还有节点
        (first = firstWaiter) != null
    );
}

// 外部类方法，方便阅读，放在此处
// ⇐ 如果节点状态是取消，返回 false 表示转移失败，否则转移成功
final boolean transferForSignal(Node node) {
    // 如果状态已经不是 Node.CONDITION，说明被取消了
    if (!compareAndSetWaitStatus(node, Node.CONDITION, 0))
        return false;

    // 加入 AQS 队列尾部
    Node p = enq(node);
    int ws = p.waitStatus;
    if (
        // 上一个节点被取消
        ws > 0 ||
        // 上一个节点不能设置状态为 Node.SIGNAL
        !compareAndSetWaitStatus(p, ws, Node.SIGNAL)
    ) {
        // unpark 取消阻塞，让线程重新同步状态
        LockSupport.unpark(node.thread);
    }
    return true;
}

// 全部唤醒 - 等待队列的所有节点转移至 AQS 队列
```



```
private void doSignalAll(Node first) {
    lastWaiter = firstWaiter = null;
    do {
        Node next = first.nextWaiter;
        first.nextWaiter = null;
        transferForSignal(first);
        first = next;
    } while (first != null);
}

// ⇨
private void unlinkCancelledWaiters() {
    // ...
}

// 唤醒 - 必须持有锁才能唤醒，因此 doSignal 内无需考虑加锁
public final void signal() {
    if (!isHeldExclusively())
        throw new IllegalMonitorStateException();
    Node first = firstWaiter;
    if (first != null)
        doSignal(first);
}

// 全部唤醒 - 必须持有锁才能唤醒，因此 doSignalAll 内无需考虑加锁
public final void signalAll() {
    if (!isHeldExclusively())
        throw new IllegalMonitorStateException();
    Node first = firstWaiter;
    if (first != null)
        doSignalAll(first);
}

// 不可打断等待 - 直到被唤醒
public final void awaitUninterruptibly() {
    // 添加一个 Node 至等待队列，见 ⇨
    Node node = addConditionWaiter();
    // 释放节点持有的锁，见 ⇨
    int savedState = fullyRelease(node);
    boolean interrupted = false;
    // 如果该节点还没有转移至 AQS 队列，阻塞
    while (!isOnSyncQueue(node)) {
        // park 阻塞
        LockSupport.park(this);
        // 如果被打断，仅设置打断状态
        if (Thread.interrupted())
            interrupted = true;
    }
    // 唤醒后，尝试竞争锁，如果失败进入 AQS 队列
    if (acquireQueued(node, savedState) || interrupted)
        selfInterrupt();
}
```



```
// 外部类方法，方便阅读，放在此处
// ④ 因为某线程可能重入，需要将 state 全部释放
final int fullyRelease(Node node) {
    boolean failed = true;
    try {
        int savedState = getState();
        if (release(savedState)) {
            failed = false;
            return savedState;
        } else {
            throw new IllegalMonitorStateException();
        }
    } finally {
        if (failed)
            node.waitStatus = Node.CANCELLED;
    }
}

// 打断模式 - 在退出等待时重新设置打断状态
private static final int REINTERRUPT = 1;
// 打断模式 - 在退出等待时抛出异常
private static final int THROW_IE = -1;

// 判断打断模式
private int checkInterruptWhileWaiting(Node node) {
    return Thread.interrupted() ?
        (transferAfterCancelledWait(node) ? THROW_IE : REINTERRUPT) :
        0;
}

// ⑤ 应用打断模式
private void reportInterruptAfterWait(int interruptMode)
    throws InterruptedException {
    if (interruptMode == THROW_IE)
        throw new InterruptedException();
    else if (interruptMode == REINTERRUPT)
        selfInterrupt();
}

// 等待 - 直到被唤醒或打断
public final void await() throws InterruptedException {
    if (Thread.interrupted()) {
        throw new InterruptedException();
    }
    // 添加一个 Node 至等待队列，见 ③
    Node node = addConditionWaiter();
    // 释放节点持有的锁
    int savedState = fullyRelease(node);
    int interruptMode = 0;
    // 如果该节点还没有转移至 AQS 队列，阻塞
    while (!isOnSyncQueue(node)) {
        // park 阻塞

        LockSupport.park(this);
    }
}
```



```
// 如果被打断，退出等待队列
if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
    break;
}

// 退出等待队列后，还需要获得 AQS 队列的锁
if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
    interruptMode = REINTERRUPT;
// 所有已取消的 Node 从队列链表删除，见 (二)
if (node.nextWaiter != null)
    unlinkCancelledWaiters();
// 应用打断模式，见 (四)
if (interruptMode != 0)
    reportInterruptAfterWait(interruptMode);
}

// 等待 - 直到被唤醒或打断或超时
public final long awaitNanos(long nanosTimeout) throws InterruptedException {
    if (Thread.interrupted()) {
        throw new InterruptedException();
    }
    // 添加一个 Node 至等待队列，见 (一)
    Node node = addConditionWaiter();
    // 释放节点持有的锁
    int savedState = fullyRelease(node);
    // 获得最后期限
    final long deadline = System.nanoTime() + nanosTimeout;
    int interruptMode = 0;
    // 如果该节点还没有转移至 AQS 队列，阻塞
    while (!isOnSyncQueue(node)) {
        // 已超时，退出等待队列
        if (nanosTimeout <= 0L) {
            transferAfterCancelledWait(node);
            break;
        }
        // park 阻塞一定时间，spinForTimeoutThreshold 为 1000 ns
        if (nanosTimeout >= spinForTimeoutThreshold)
            LockSupport.parkNanos(this, nanosTimeout);
        // 如果被打断，退出等待队列
        if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
            break;
        nanosTimeout = deadline - System.nanoTime();
    }
    // 退出等待队列后，还需要获得 AQS 队列的锁
    if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
        interruptMode = REINTERRUPT;
    // 所有已取消的 Node 从队列链表删除，见 (二)
    if (node.nextWaiter != null)
        unlinkCancelledWaiters();
    // 应用打断模式，见 (四)
    if (interruptMode != 0)
        reportInterruptAfterWait(interruptMode);
    return deadline - System.nanoTime();
}
```

```
// 等待 - 直到被唤醒或打断或超时，逻辑类似于 awaitNanos
public final boolean awaitUntil(Date deadline) throws InterruptedException {
    // ...
}

// 等待 - 直到被唤醒或打断或超时，逻辑类似于 awaitNanos
public final boolean await(long time, TimeUnit unit) throws InterruptedException {
    // ...
}

// 工具方法 省略 ...
}
```

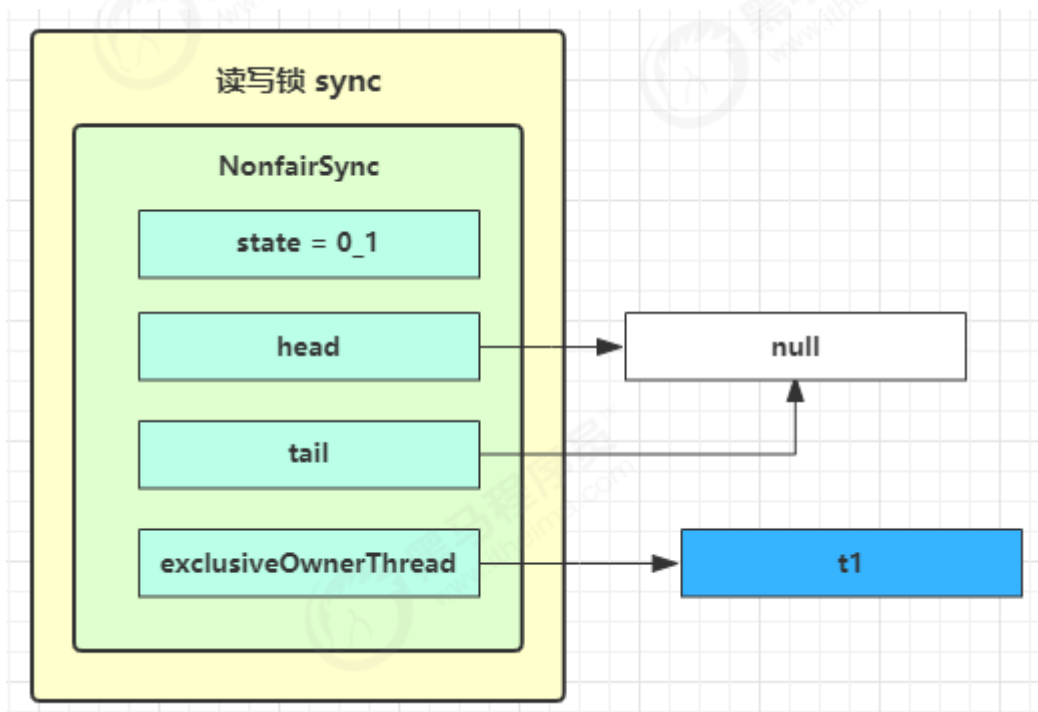
读写锁原理

1. 图解流程

读写锁用的是同一个 Sync 同步器，因此等待队列、state 等也是同一个

t1 w.lock , t2 r.lock

1) t1 成功上锁，流程与 ReentrantLock 加锁相比没有特殊之处，不同是写锁状态占了 state 的低 16 位，而读锁使用的是 state 的高 16 位

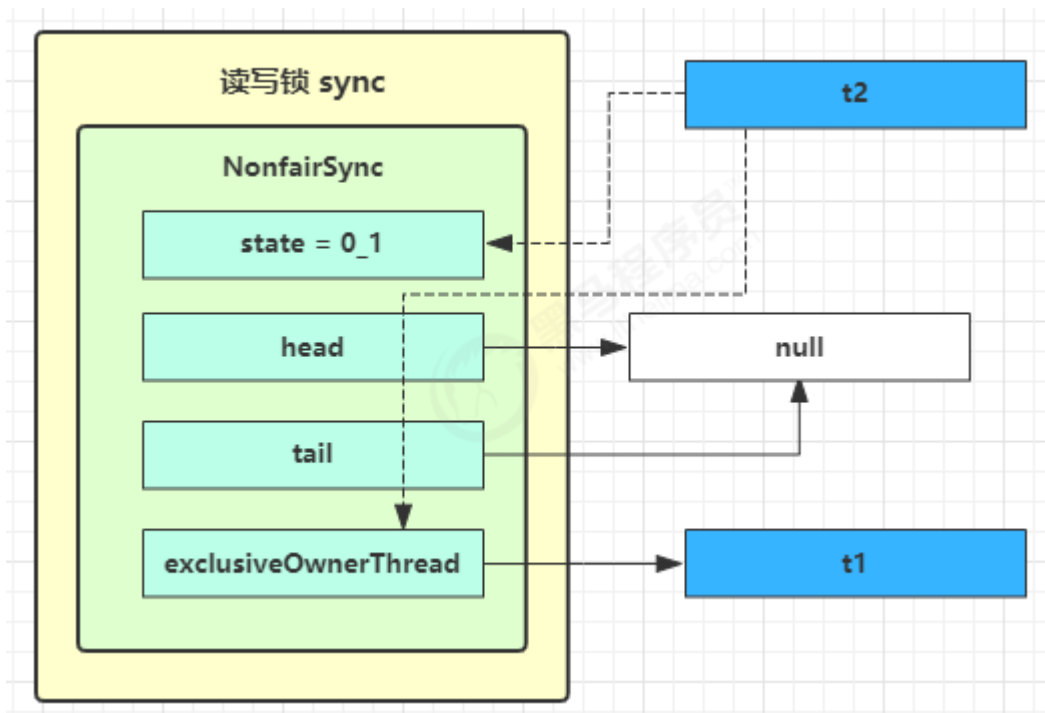


2) t2 执行 r.lock，这时进入读锁的 sync.acquireShared(1) 流程，首先会进入 tryAcquireShared 流程。如果有写锁占据，那么 tryAcquireShared 返回 -1 表示失败

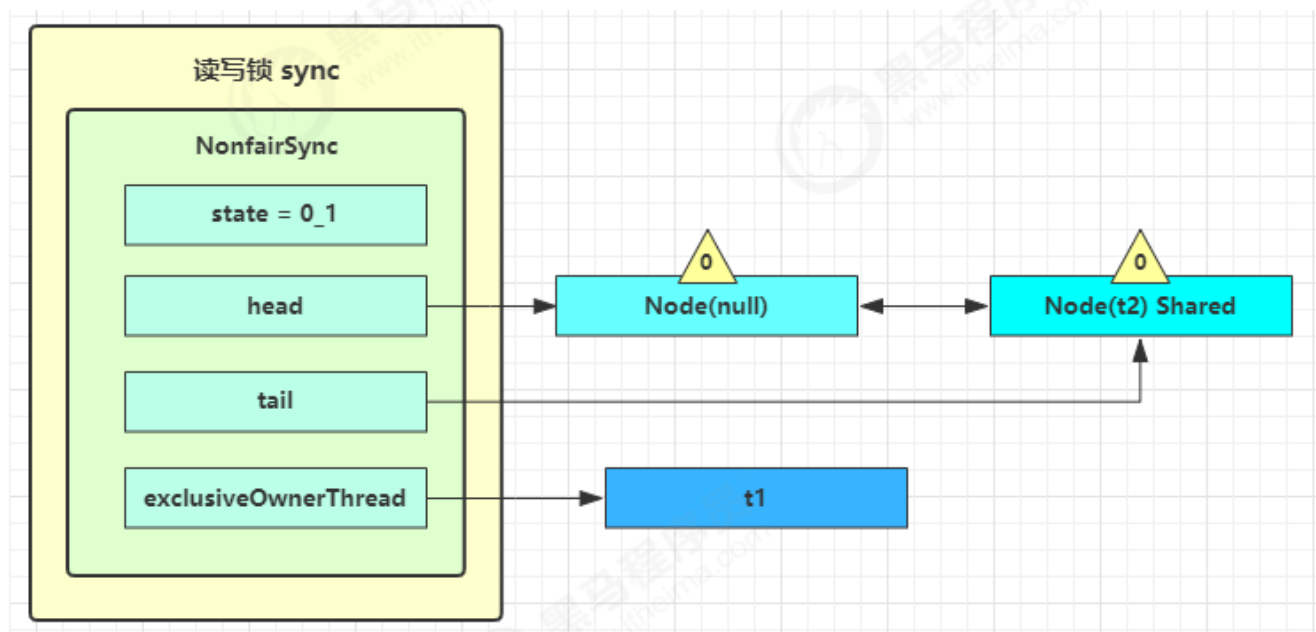
tryAcquireShared 返回值表示

- -1 表示失败

- 0 表示成功，但后继节点不会继续唤醒
- 正数表示成功，而且数值是还有几个后继节点需要唤醒，读写锁返回 1

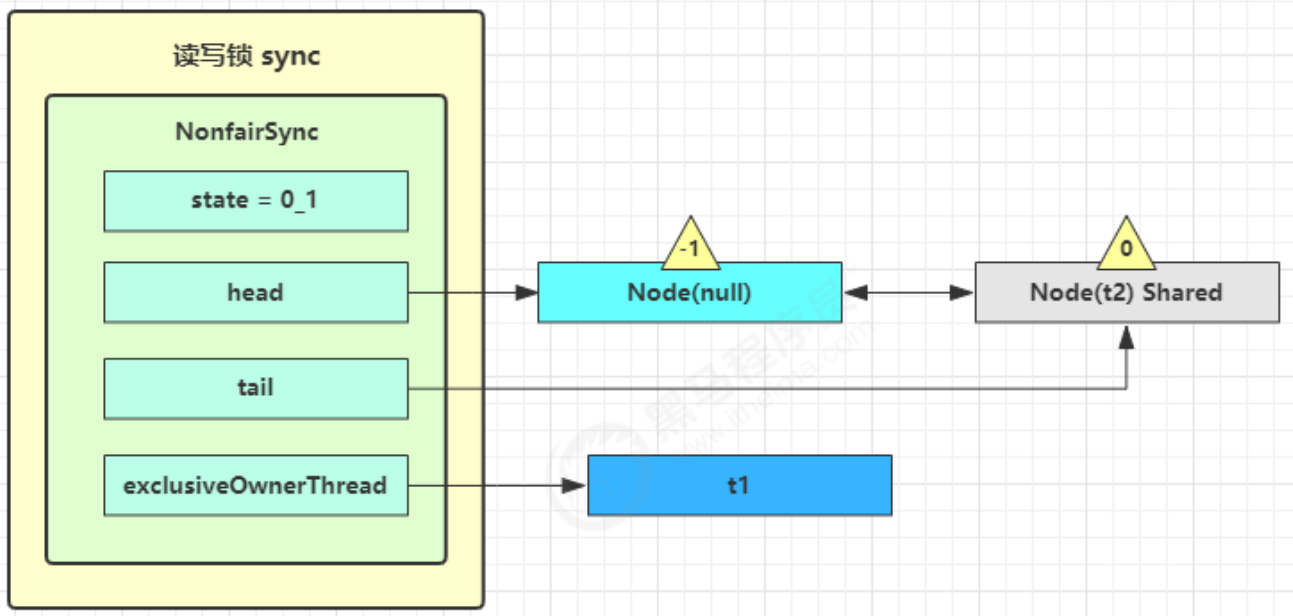


3) 这时会进入 sync.doAcquireShared(1) 流程，首先也是调用 addWaiter 添加节点，不同之处在于节点被设置为 Node.SHARED 模式而非 Node.EXCLUSIVE 模式，注意此时 t2 仍处于活跃状态



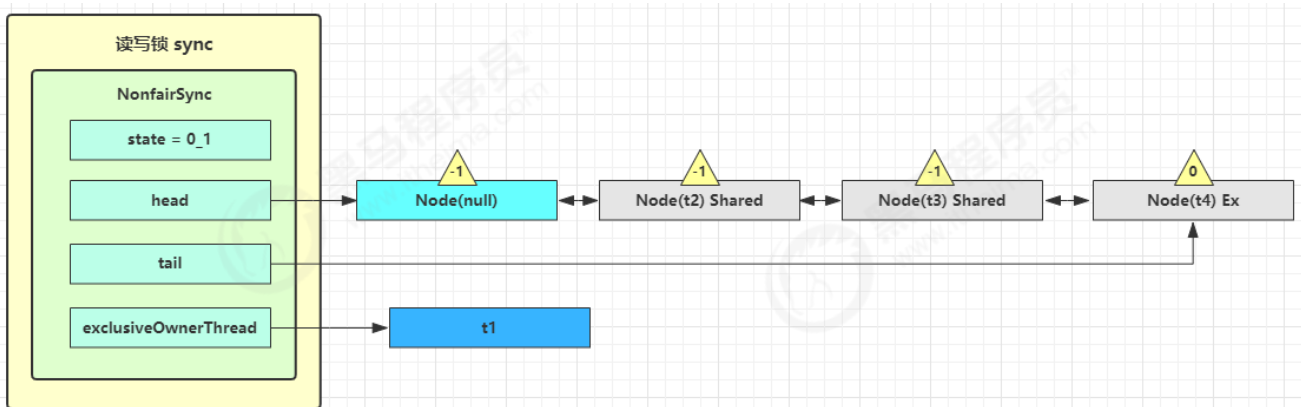
4) t2 会看看自己的节点是不是老二，如果是，还会再次调用 tryAcquireShared(1) 来尝试获取锁

5) 如果没有成功，在 doAcquireShared 内 for (;;) 循环一次，把前驱节点的 waitStatus 改为 -1，再 for (;;) 循环一次尝试 tryAcquireShared(1) 如果还不成功，那么在 parkAndCheckInterrupt() 处 park



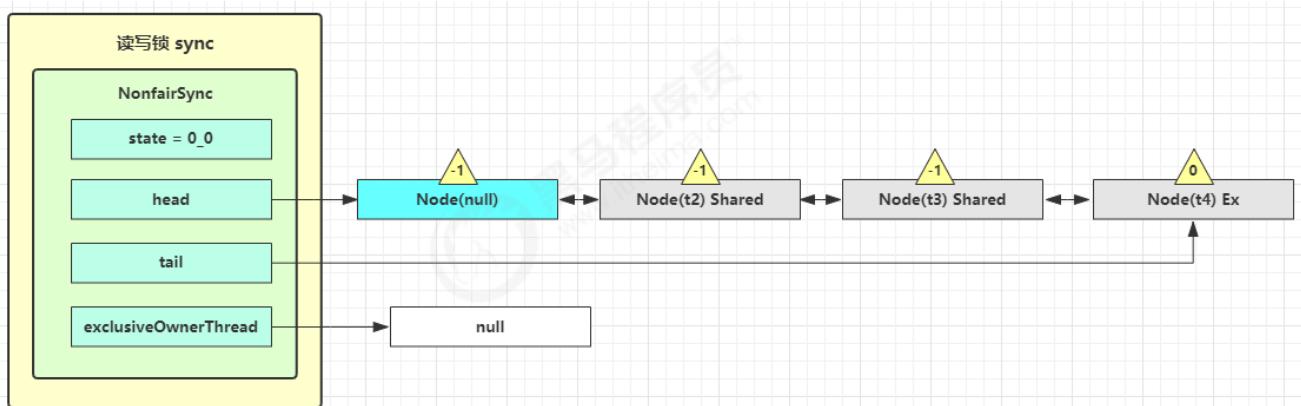
t3 r.lock , t4 w.lock

这种状态下，假设又有 t3 加读锁和 t4 加写锁，这期间 t1 仍然持有锁，就变成了下面的样子



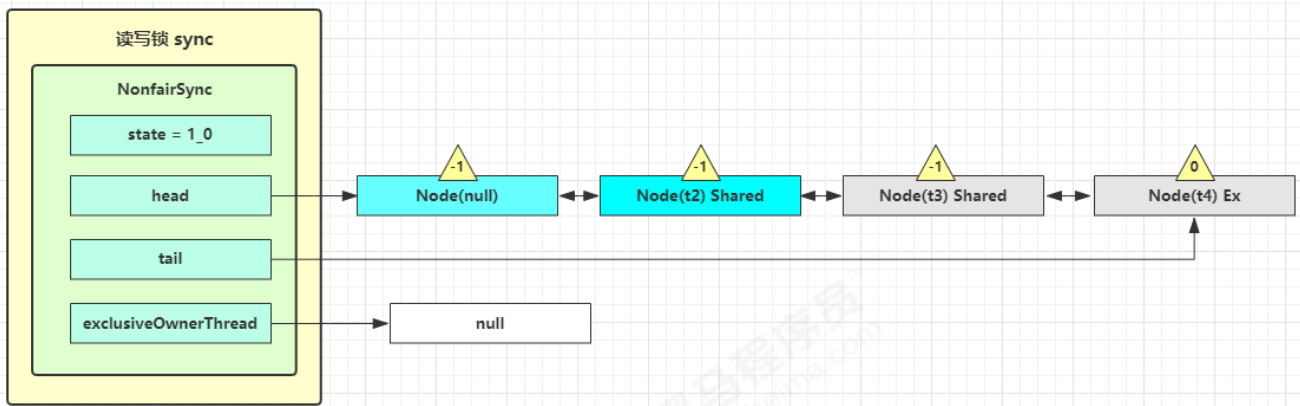
t1 w.unlock

这时会走到写锁的 sync.release(1) 流程，调用 sync.tryRelease(1) 成功，变成下面的样子

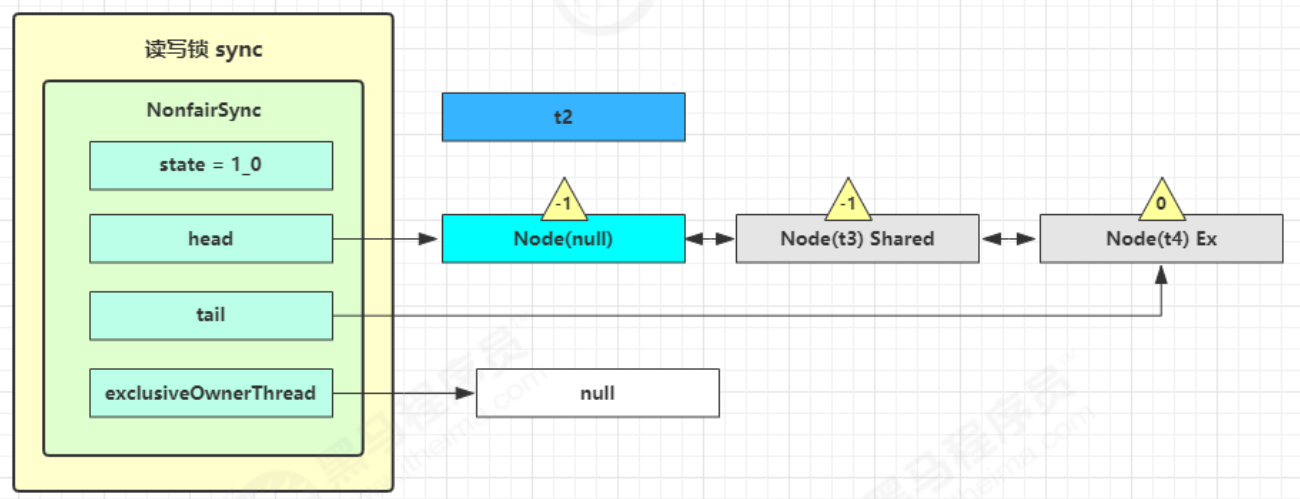


接下来执行唤醒流程 sync.unparkSuccessor，即让老二恢复运行，这时 t2 在 doAcquireShared 内 parkAndCheckInterrupt() 处恢复运行

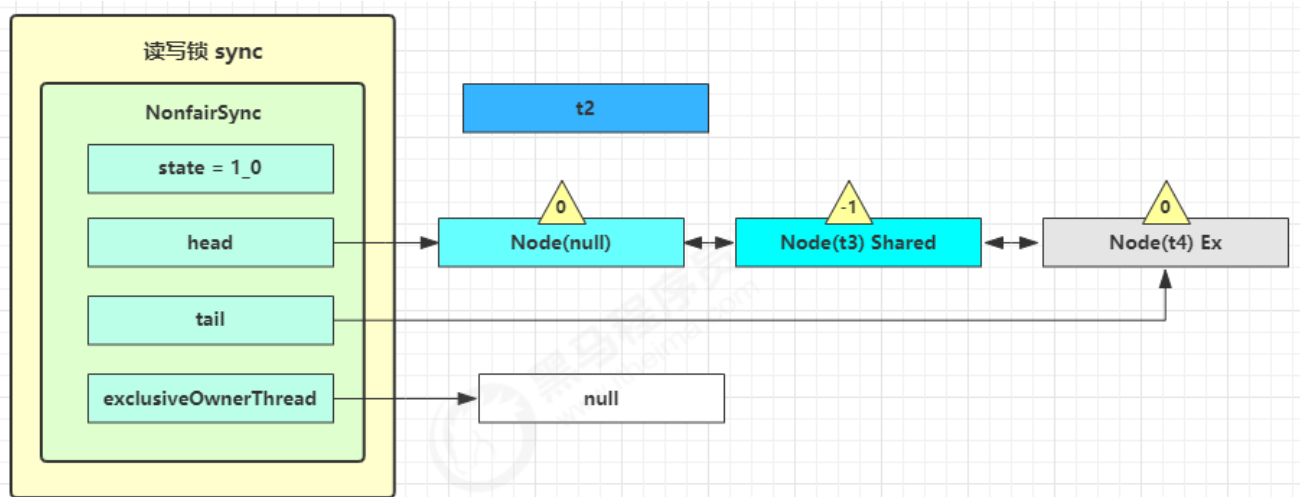
这回再来一次 for (;;) 执行 tryAcquireShared 成功则让读锁计数加一



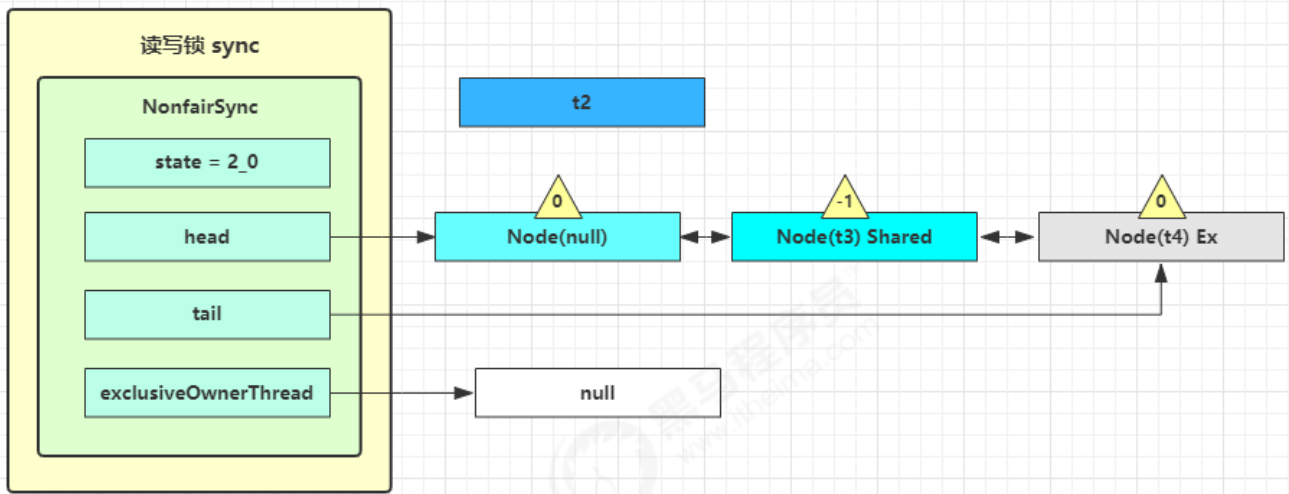
这时 t2 已经恢复运行，接下来 t2 调用 setHeadAndPropagate(node, 1)，它原本所在节点被置为头节点



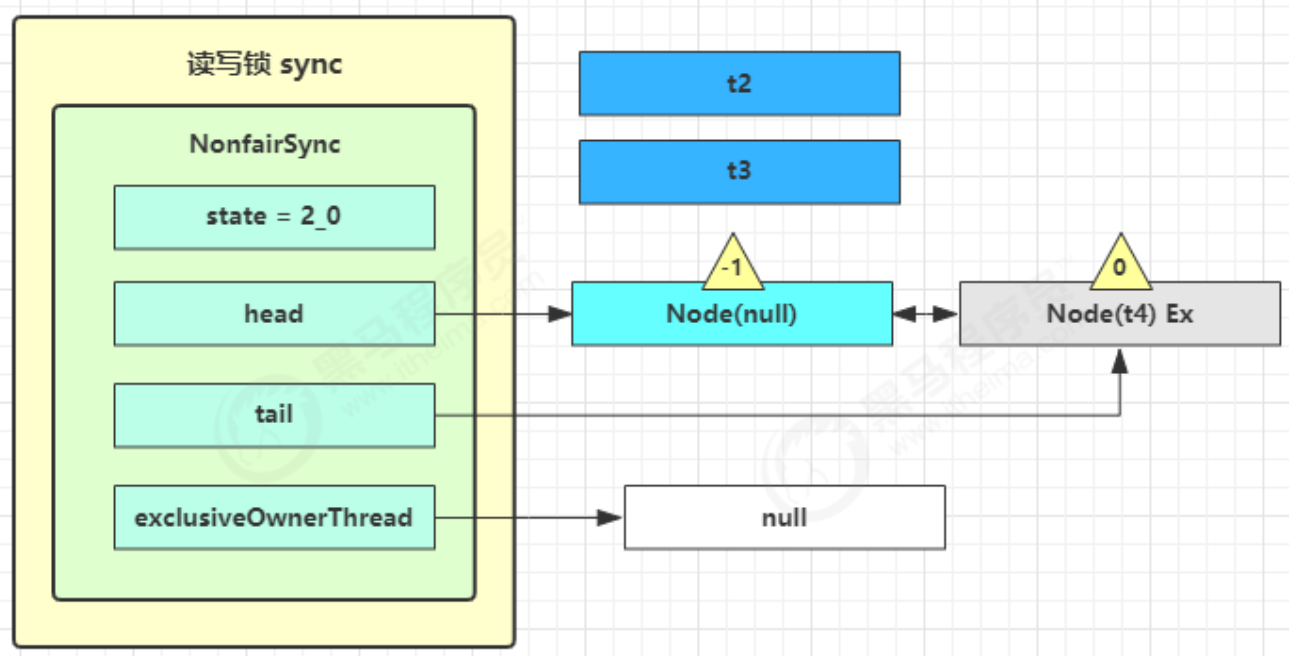
事情还没完，在 setHeadAndPropagate 方法内还会检查下一个节点是否是 shared，如果是则调用 doReleaseShared() 将 head 的状态从 -1 改为 0 并唤醒老二，这时 t3 在 doAcquireShared 内 parkAndCheckInterrupt() 处恢复运行



这回再来一次 for (;;) 执行 tryAcquireShared 成功则让读锁计数加一



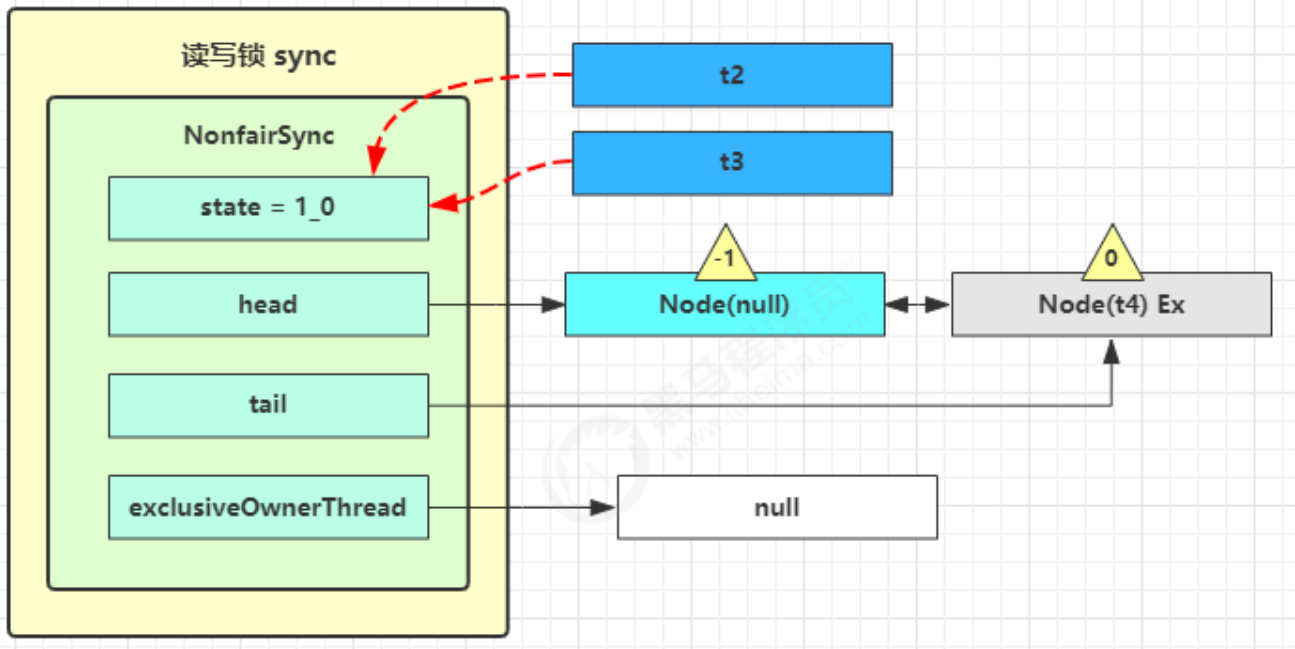
这时 t3 已经恢复运行，接下来 t3 调用 setHeadAndPropagate(node, 1)，它原本所在节点被置为头节点



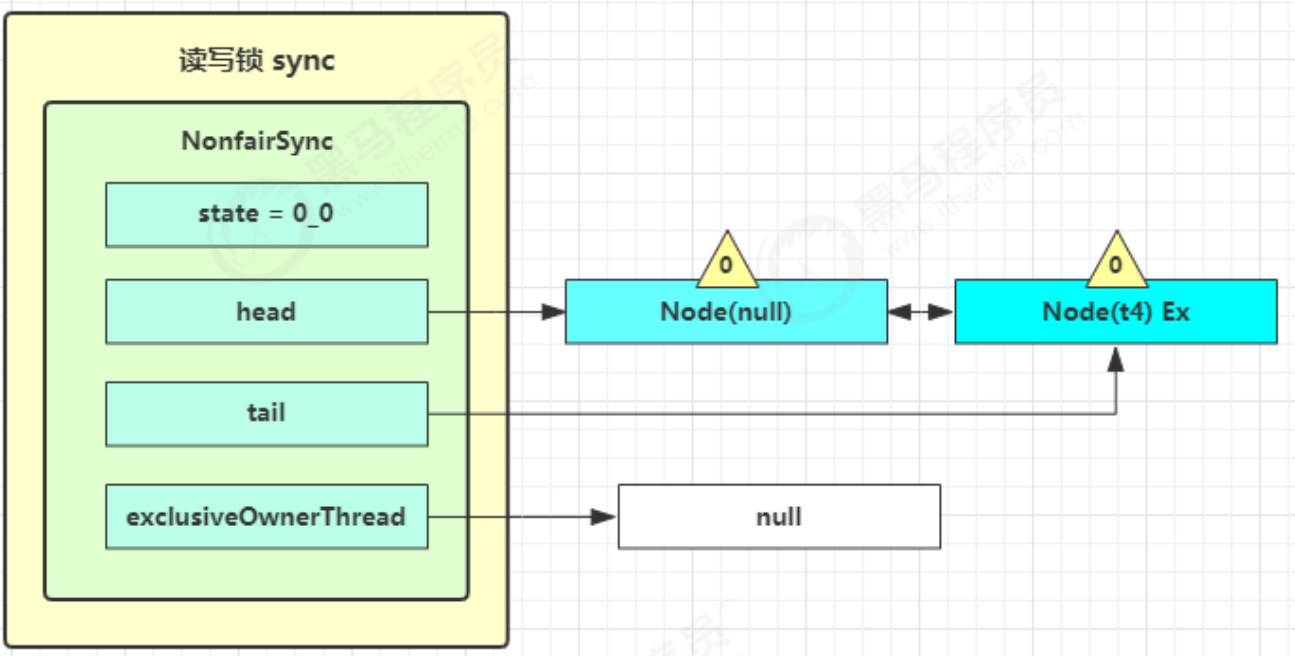
下一个节点不是 shared 了，因此不会继续唤醒 t4 所在节点

t2 r.unlock , t3 r.unlock

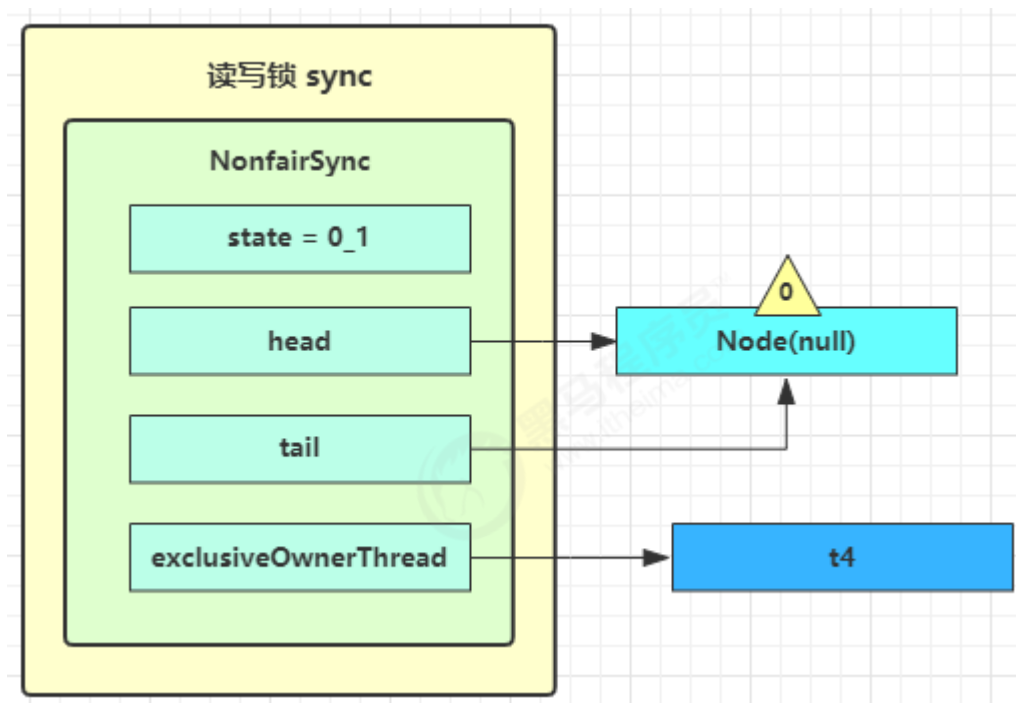
t2 进入 sync.releaseShared(1) 中，调用 tryReleaseShared(1) 让计数减一，但由于计数还不为零



t3 进入 `sync.releaseShared(1)` 中，调用 `tryReleaseShared(1)` 让计数减一，这回计数为零了，进入 `doReleaseShared()` 将头节点从 -1 改为 0 并唤醒老二，即



之后 t4 在 `acquireQueued` 中 `parkAndCheckInterrupt` 处恢复运行，再次 `for (;;) 这次自己是老二`，并且没有其他竞争，`tryAcquire(1)` 成功，修改头结点，流程结束



2. 源码分析

写锁上锁流程

```
static final class NonfairSync extends Sync {
    // ... 省略无关代码

    // 外部类 WriteLock 方法，方便阅读，放在此处
    public void lock() {
        sync.acquire(1);
    }

    // AQS 继承过来的方法，方便阅读，放在此处
    public final void acquire(int arg) {
        if (
            // 尝试获得写锁失败
            !tryAcquire(arg) &&
            // 将当前线程关联到一个 Node 对象上，模式为独占模式
            // 进入 AQS 队列阻塞
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg)
        ) {
            selfInterrupt();
        }
    }

    // Sync 继承过来的方法，方便阅读，放在此处
    protected final boolean tryAcquire(int acquires) {
        // 获得低 16 位，代表写锁的 state 计数
        Thread current = Thread.currentThread();

        int c = getState();
```



```
int w = exclusiveCount(c);

if (c != 0) {
    if (
        // c != 0 and w == 0 表示有读锁，或者
        w == 0 ||
        // 如果 exclusiveOwnerThread 不是自己
        current != getExclusiveOwnerThread()
    ) {
        // 获得锁失败
        return false;
    }
    // 写锁计数超过低 16 位，报异常
    if (w + exclusiveCount(acquires) > MAX_COUNT)
        throw new Error("Maximum lock count exceeded");
    // 写锁重入，获得锁成功
    setState(c + acquires);
    return true;
}
if (
    // 判断写锁是否该阻塞，或者
    writerShouldBlock() ||
    // 尝试更改计数失败
    !compareAndSetState(c, c + acquires)
) {
    // 获得锁失败
    return false;
}
// 获得锁成功
setExclusiveOwnerThread(current);
return true;
}

// 非公平锁 writerShouldBlock 总是返回 false，无需阻塞
final boolean writerShouldBlock() {
    return false;
}
}
```

写锁释放流程

```
static final class NonfairSync extends Sync {
    // ... 省略无关代码

    // WriteLock 方法，方便阅读，放在此处
    public void unlock() {
        sync.release(1);
    }

    // AQS 继承过来的方法，方便阅读，放在此处
    public final boolean release(int arg) {
        // 尝试释放写锁成功
    }
}
```



```
        if (tryRelease(arg)) {
            // unpark AQS 中等待的线程
            Node h = head;
            if (h != null && h.waitStatus != 0)
                unparkSuccessor(h);
            return true;
        }
        return false;
    }

    // Sync 继承过来的方法，方便阅读，放在此处
    protected final boolean tryRelease(int releases) {
        if (!isHeldExclusively())
            throw new IllegalMonitorStateException();
        int nextc = getState() - releases;
        // 因为可重入的原因，写锁计数为 0，才算释放成功
        boolean free = exclusiveCount(nextc) == 0;
        if (free) {
            setExclusiveOwnerThread(null);
        }
        setState(nextc);
        return free;
    }
}
```

读锁上锁流程

```
static final class NonfairSync extends Sync {

    // ReadLock 方法，方便阅读，放在此处
    public void lock() {
        sync.acquireShared(1);
    }

    // AQS 继承过来的方法，方便阅读，放在此处
    public final void acquireShared(int arg) {
        // tryAcquireShared 返回负数，表示获取读锁失败
        if (tryAcquireShared(arg) < 0) {
            doAcquireShared(arg);
        }
    }

    // Sync 继承过来的方法，方便阅读，放在此处
    protected final int tryAcquireShared(int unused) {
        Thread current = Thread.currentThread();
        int c = getState();
        // 如果是其它线程持有写锁，获取读锁失败
        if (
            exclusiveCount(c) != 0 &&
            getExclusiveOwnerThread() != current
        ) {
            return -1;
        }
    }
}
```



```
}
int r = sharedCount(c);
if (
    // 读锁不该阻塞(如果老二是写锁, 读锁该阻塞), 并且
    !readerShouldBlock() &&
    // 小于读锁计数, 并且
    r < MAX_COUNT &&
    // 尝试增加计数成功
    compareAndSetState(c, c + SHARED_UNIT)
) {
    // ... 省略不重要的代码
    return 1;
}
return fullTryAcquireShared(current);
}

// 非公平锁 readerShouldBlock 看 AQS 队列中第一个节点是否是写锁
// true 则该阻塞, false 则不阻塞
final boolean readerShouldBlock() {
    return apparentlyFirstQueuedIsExclusive();
}

// AQS 继承过来的方法, 方便阅读, 放在此处
// 与 tryAcquireShared 功能类似, 但会不断尝试 for (;;) 获取读锁, 执行过程中无阻塞
final int fullTryAcquireShared(Thread current) {
    HoldCounter rh = null;
    for (;;) {
        int c = getState();
        if (exclusiveCount(c) != 0) {
            if (getExclusiveOwnerThread() != current)
                return -1;
        } else if (readerShouldBlock()) {
            // ... 省略不重要的代码
        }
        if (sharedCount(c) == MAX_COUNT)
            throw new Error("Maximum lock count exceeded");
        if (compareAndSetState(c, c + SHARED_UNIT)) {
            // ... 省略不重要的代码
            return 1;
        }
    }
}

// AQS 继承过来的方法, 方便阅读, 放在此处
private void doAcquireShared(int arg) {
    // 将当前线程关联到一个 Node 对象上, 模式为共享模式
    final Node node = addWaiter(Node.SHARED);
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {
            final Node p = node.predecessor();

            if (p == head) {
```



```
// 再一次尝试获取读锁
int r = tryAcquireShared(arg);
// 成功
if (r >= 0) {
    // (→)
    // r 表示可用资源数，在这里总是 1 允许传播
    // (唤醒 AQS 中下一个 Share 节点)
    setHeadAndPropagate(node, r);
    p.next = null; // help GC
    if (interrupted)
        selfInterrupt();
    failed = false;
    return;
}
}
if (
    // 是否在获取读锁失败时阻塞 (前一个阶段 waitStatus == Node.SIGNAL)
    shouldParkAfterFailedAcquire(p, node) &&
    // park 当前线程
    parkAndCheckInterrupt()
) {
    interrupted = true;
}
}
} finally {
    if (failed)
        cancelAcquire(node);
}
}

// (→) AQS 继承过来的方法，方便阅读，放在此处
private void setHeadAndPropagate(Node node, int propagate) {
    Node h = head; // Record old head for check below
    // 设置自己为 head
    setHead(node);

    // propagate 表示有共享资源 (例如共享读锁或信号量)
    // 原 head waitStatus == Node.SIGNAL 或 Node.PROPAGATE
    // 现在 head waitStatus == Node.SIGNAL 或 Node.PROPAGATE
    if (propagate > 0 || h == null || h.waitStatus < 0 ||
        (h = head) == null || h.waitStatus < 0) {
        Node s = node.next;
        // 如果是最后一个节点或者是等待共享读锁的节点
        if (s == null || s.isShared()) {
            // 进入 (→)
            doReleaseShared();
        }
    }
}

// (→) AQS 继承过来的方法，方便阅读，放在此处
private void doReleaseShared() {

    // 如果 head.waitStatus == Node.SIGNAL ==> 0 成功，下一个节点 unpark
}
```




```
// 如果 head.waitStatus == 0 ==> Node.PROPAGATE, 为了解决 bug, 见后面分析
for (;;) {
    Node h = head;
    // 队列还有节点
    if (h != null && h != tail) {
        int ws = h.waitStatus;
        if (ws == Node.SIGNAL) {
            if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))
                continue;          // loop to recheck cases
            // 下一个节点 unpark 如果成功获取读锁
            // 并且下下个节点还是 shared, 继续 doReleaseShared
            unparkSuccessor(h);
        }
        else if (ws == 0 &&
            !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))
            continue;              // loop on failed CAS
    }
    if (h == head)                // loop if head changed
        break;
}
}
```

读锁释放流程

```
static final class NonfairSync extends Sync {

    // ReadLock 方法, 方便阅读, 放在此处
    public void unlock() {
        sync.releaseShared(1);
    }

    // AQS 继承过来的方法, 方便阅读, 放在此处
    public final boolean releaseShared(int arg) {
        if (tryReleaseShared(arg)) {
            doReleaseShared();
            return true;
        }
        return false;
    }

    // Sync 继承过来的方法, 方便阅读, 放在此处
    protected final boolean tryReleaseShared(int unused) {
        // ... 省略不重要的代码
        for (;;) {
            int c = getState();
            int nextc = c - SHARED_UNIT;
            if (compareAndSetState(c, nextc)) {
                // 读锁的计数不会影响其它获取读锁线程, 但会影响其它获取写锁线程
                // 计数为 0 才是真正释放
                return nextc == 0;
            }
        }
    }
}
```

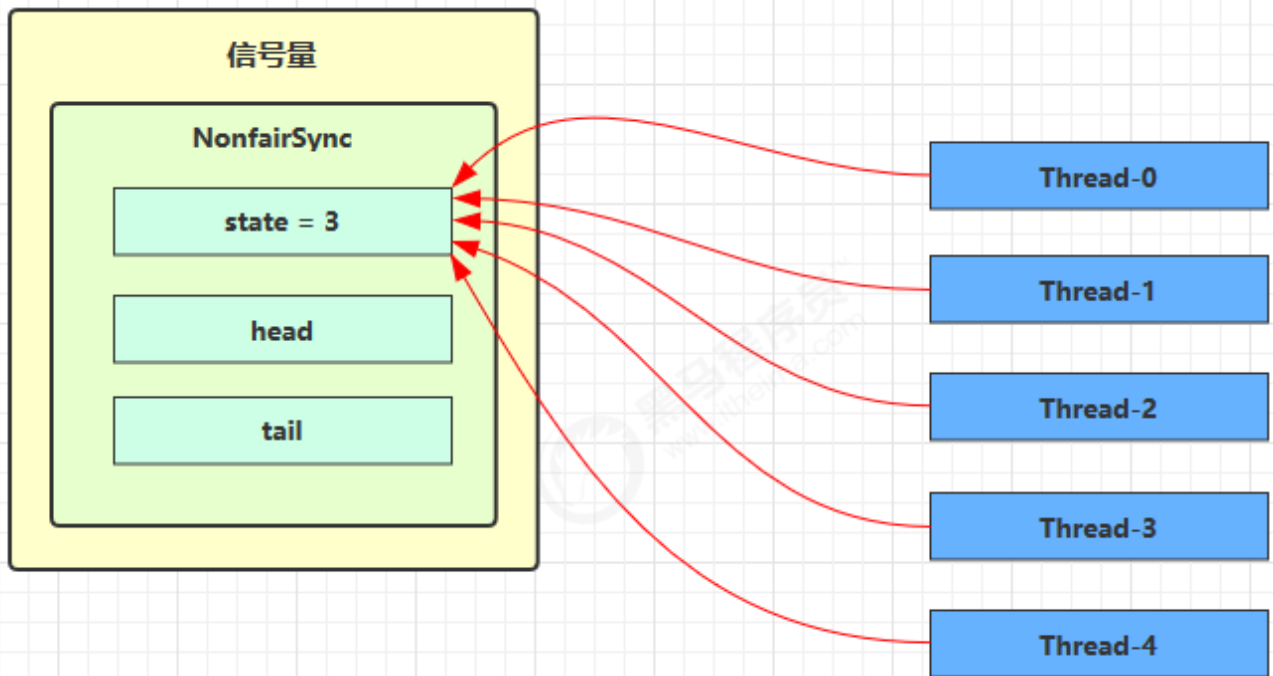
```
}  
}  
  
// AQS 继承过来的方法，方便阅读，放在此处  
private void doReleaseShared() {  
    // 如果 head.waitStatus == Node.SIGNAL ==> 0 成功，下一个节点 unpark  
    // 如果 head.waitStatus == 0 ==> Node.PROPAGATE  
    for (;;) {  
        Node h = head;  
        if (h != null && h != tail) {  
            int ws = h.waitStatus;  
            // 如果有其它线程也在释放读锁，那么需要将 waitStatus 先改为 0  
            // 防止 unparkSuccessor 被多次执行  
            if (ws == Node.SIGNAL) {  
                if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))  
                    continue; // loop to recheck cases  
                unparkSuccessor(h);  
            }  
            // 如果已经是 0 了，改为 -3，用来解决传播性，见后文信号量 bug 分析  
            else if (ws == 0 &&  
                !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))  
                continue; // loop on failed CAS  
        }  
        if (h == head) // loop if head changed  
            break;  
    }  
}
```

Semaphore 原理

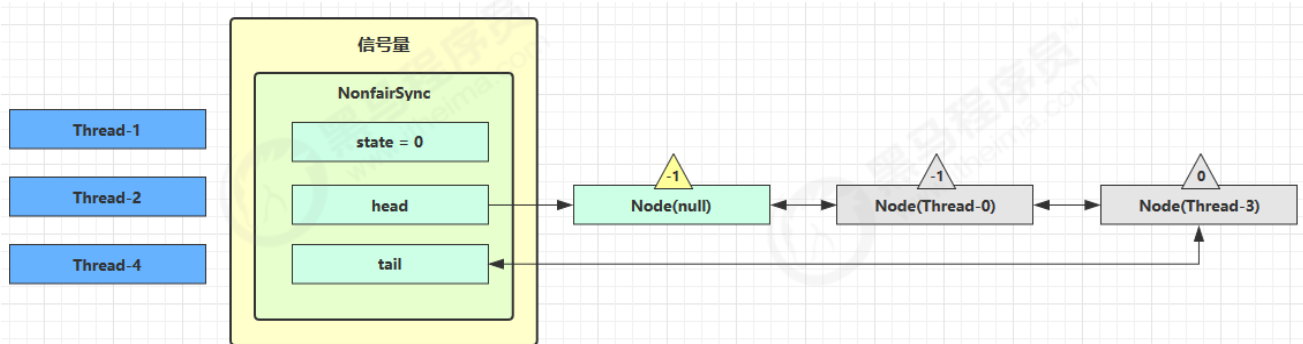
1. 加锁解锁流程

Semaphore 有点像一个停车场，permits 就好像停车位数量，当线程获得了 permits 就像是获得了停车位，然后停车场显示空余车位减一

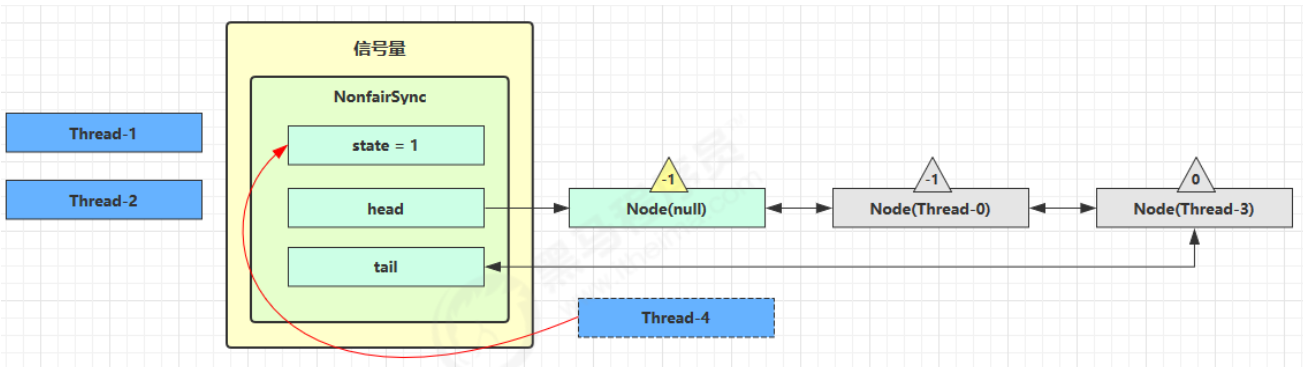
刚开始，permits (state) 为 3，这时 5 个线程来获取资源



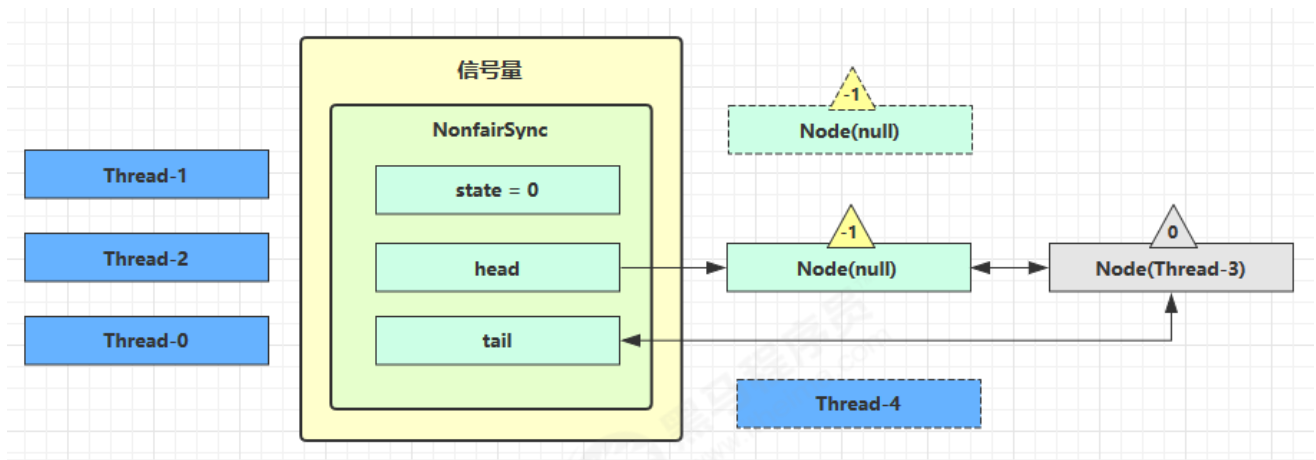
假设其中 Thread-1, Thread-2, Thread-4 竞争成功, 而 Thread-0 和 Thread-3 竞争失败, 进入 AQS 队列 park 阻塞



这时 Thread-4 释放了 permits, 状态如下



接下来 Thread-0 竞争成功, permits 再次设置为 0, 设置自己为 head 节点, 断开原来的 head 节点, unpark 接下来的 Thread-3 节点, 但由于 permits 是 0, 因此 Thread-3 在尝试不成功后再进入 park 状态



2. 源码分析

```
static final class NonfairSync extends Sync {
    private static final long serialVersionUID = -2694183684443567898L;

    NonfairSync(int permits) {
        // permits 即 state
        super(permits);
    }

    // Semaphore 方法，方便阅读，放在此处
    public void acquire() throws InterruptedException {
        sync.acquireSharedInterruptibly(1);
    }

    // AQS 继承过来的方法，方便阅读，放在此处
    public final void acquireSharedInterruptibly(int arg)
        throws InterruptedException {
        if (Thread.interrupted())
            throw new InterruptedException();
        if (tryAcquireShared(arg) < 0)
            doAcquireSharedInterruptibly(arg);
    }

    // 尝试获得共享锁
    protected int tryAcquireShared(int acquires) {
        return nonfairTryAcquireShared(acquires);
    }

    // Sync 继承过来的方法，方便阅读，放在此处
    final int nonfairTryAcquireShared(int acquires) {
        for (;;) {
            int available = getState();
            int remaining = available - acquires;
            if (
                // 如果许可已经用完，返回负数，表示获取失败，进入 doAcquireSharedInterruptibly
                remaining < 0 ||
                // 如果 cas 重试成功，返回正数，表示获取成功
                compareAndSetState(available, remaining)
            )
                return remaining < 0 ? -1 : 1;
        }
    }
}
```



```
    } {
        return remaining;
    }
}

// AQS 继承过来的方法，方便阅读，放在此处
private void doAcquireSharedInterruptibly(int arg) throws InterruptedException {
    final Node node = addWaiter(Node.SHARED);
    boolean failed = true;
    try {
        for (;;) {
            final Node p = node.predecessor();
            if (p == head) {
                // 再次尝试获取许可
                int r = tryAcquireShared(arg);
                if (r >= 0) {
                    // 成功后本线程出队 (AQS)，所在 Node 设置为 head
                    // 如果 head.waitStatus == Node.SIGNAL ==> 0 成功，下一个节点 unpark
                    // 如果 head.waitStatus == 0 ==> Node.PROPAGATE
                    // r 表示可用资源数，为 0 则不会继续传播
                    setHeadAndPropagate(node, r);
                    p.next = null; // help GC
                    failed = false;
                    return;
                }
            }
            // 不成功，设置上一个节点 waitStatus = Node.SIGNAL，下轮进入 park 阻塞
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                throw new InterruptedException();
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}

// Semaphore 方法，方便阅读，放在此处
public void release() {
    sync.releaseShared(1);
}

// AQS 继承过来的方法，方便阅读，放在此处
public final boolean releaseShared(int arg) {
    if (tryReleaseShared(arg)) {
        doReleaseShared();
        return true;
    }
    return false;
}

// Sync 继承过来的方法，方便阅读，放在此处
```



```
protected final boolean tryReleaseShared(int releases) {
    for (;;) {
        int current = getState();
        int next = current + releases;
        if (next < current) // overflow
            throw new Error("Maximum permit count exceeded");
        if (compareAndSetState(current, next))
            return true;
    }
}
```

3. 为什么要有 PROPAGATE

早期有 bug

- releaseShared 方法

```
public final boolean releaseShared(int arg) {
    if (tryReleaseShared(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}
```

- doAcquireShared 方法

```
private void doAcquireShared(int arg) {
    final Node node = addWaiter(Node.SHARED);
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {
            final Node p = node.predecessor();
            if (p == head) {
                int r = tryAcquireShared(arg);
                if (r >= 0) {
                    // 这里会有空档
                    setHeadAndPropagate(node, r);
                    p.next = null; // help GC
                    if (interrupted)
                        selfInterrupt();
                    failed = false;
                    return;
                }
            }
        }
    }
    if (shouldParkAfterFailedAcquire(p, node) &&
        parkAndCheckInterrupt())
        return;
}
```

```

        interrupted = true;
    }
} finally {
    if (failed)
        cancelAcquire(node);
}
}

```

- setHeadAndPropagate 方法

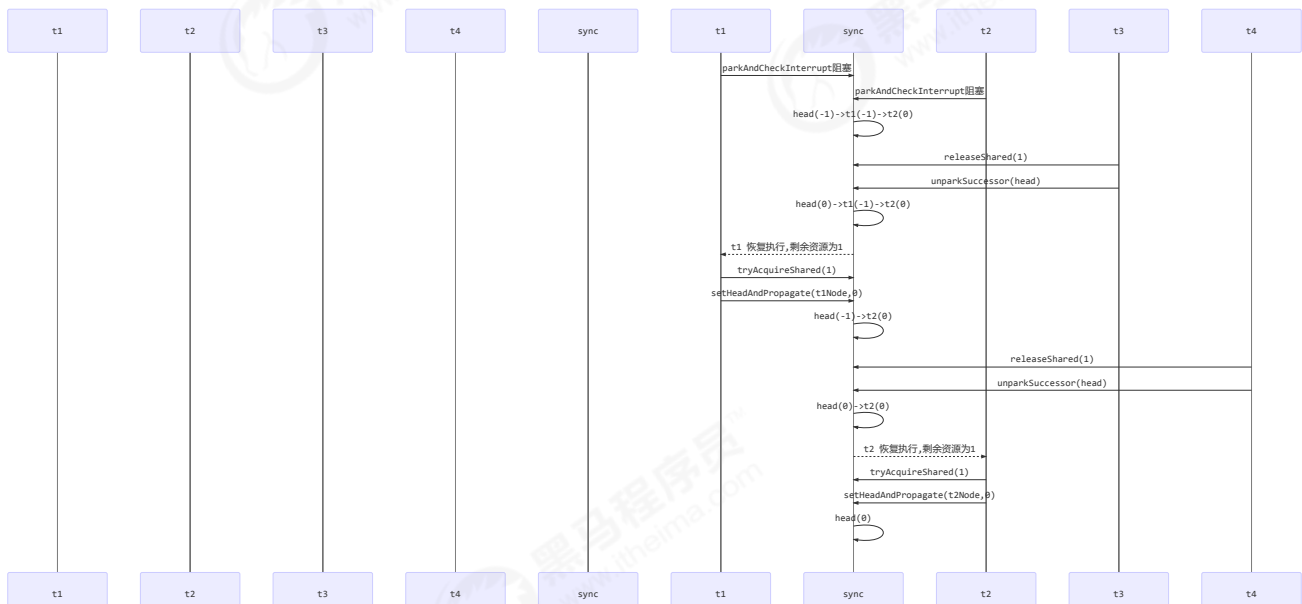
```

private void setHeadAndPropagate(Node node, int propagate) {
    setHead(node);
    // 有空闲资源
    if (propagate > 0 && node.waitStatus != 0) {
        Node s = node.next;
        // 下一个
        if (s == null || s.isShared())
            unparkSuccessor(node);
    }
}

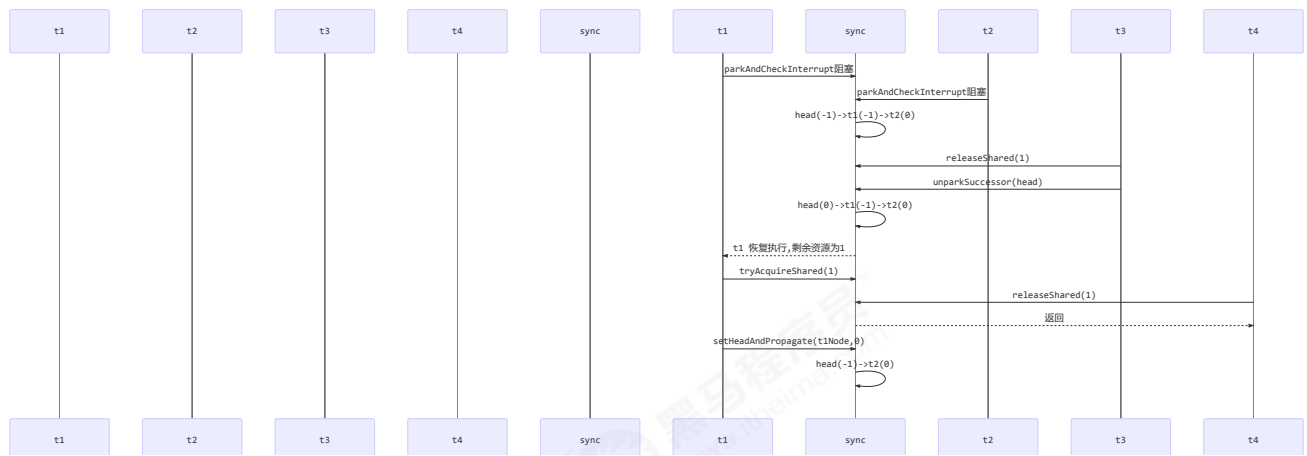
```

- 假设存在某次循环中队列里排队的结点情况为 head(-1)->t1(-1)->t2(-1)
- 假设存在将要信号量释放的 T3 和 T4，释放顺序为先 T3 后 T4

正常流程



产生 bug 的情况



修复前版本执行流程

1. T3 调用 `releaseShared(1)`，直接调用了 `unparkSuccessor(head)`，`head` 的等待状态从 -1 变为 0
2. T1 由于 T3 释放信号量被唤醒，调用 `tryAcquireShared`，假设返回值为 0（获取锁成功，但没有剩余资源量）
3. T4 调用 `releaseShared(1)`，此时 `head.waitStatus` 为 0（此时读到的 `head` 和 1 中为同一个 `head`），不满足条件，因此不调用 `unparkSuccessor(head)`
4. T1 获取信号量成功，调用 `setHeadAndPropagate` 时，因为不满足 `propagate > 0`（2 的返回值也就是 `propagate（剩余资源量）== 0`），从而不会唤醒后继结点，T2 线程得不到唤醒

bug 修复后

```

private void setHeadAndPropagate(Node node, int propagate) {
    Node h = head; // Record old head for check below
    // 设置自己为 head
    setHead(node);

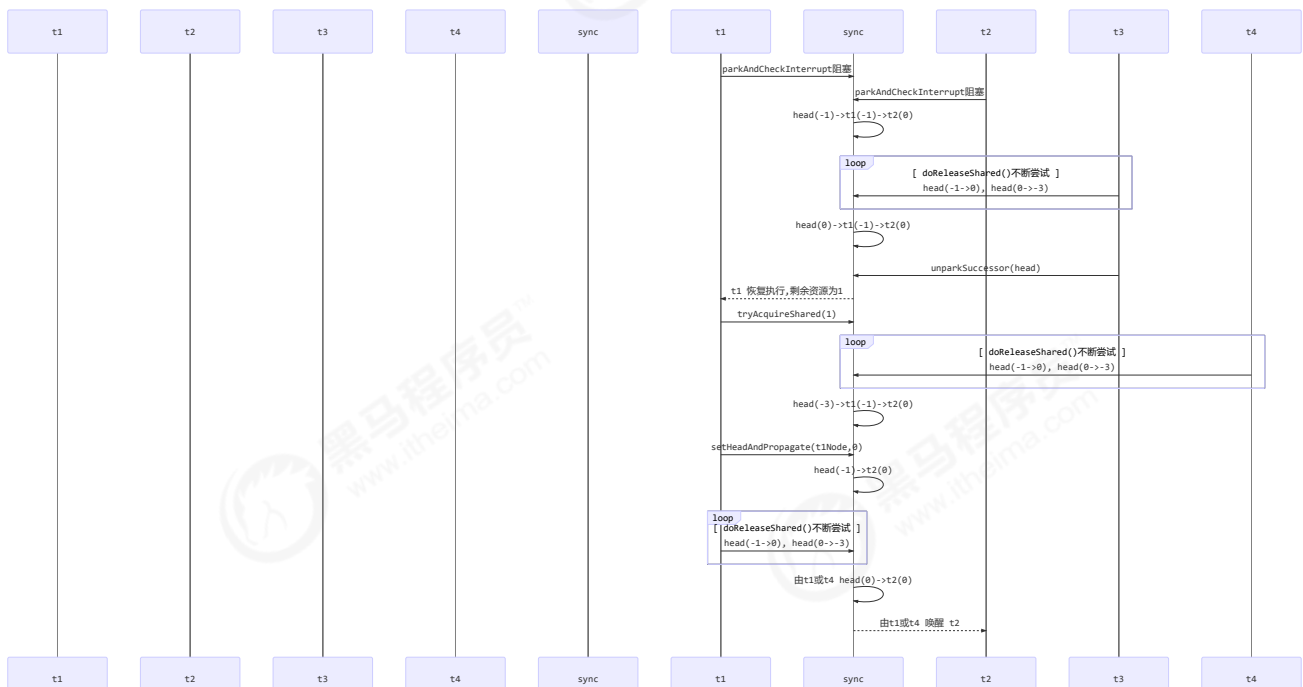
    // propagate 表示有共享资源（例如共享读锁或信号量）
    // 原 head waitStatus == Node.SIGNAL 或 Node.PROPAGATE
    // 现在 head waitStatus == Node.SIGNAL 或 Node.PROPAGATE
    if (propagate > 0 || h == null || h.waitStatus < 0 ||
        (h = head) == null || h.waitStatus < 0) {
        Node s = node.next;
        // 如果是最后一个节点或者是等待共享读锁的节点
        if (s == null || s.isShared()) {
            doReleaseShared();
        }
    }
}

private void doReleaseShared() {
    // 如果 head.waitStatus == Node.SIGNAL ==> 0 成功，下一个节点 unpark
    // 如果 head.waitStatus == 0 ==> Node.PROPAGATE
    for (;;) {
        Node h = head;
        if (h != null && h != tail) {
            int ws = h.waitStatus;
        }
    }
}
    
```



```

        if (ws == Node.SIGNAL) {
            if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))
                continue;            // loop to recheck cases
            unparkSuccessor(h);
        }
        else if (ws == 0 &&
                !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))
            continue;                // loop on failed CAS
    }
    if (h == head)                   // loop if head changed
        break;
}
    
```



1. T3 调用 `releaseShared()`，直接调用了 `unparkSuccessor(head)`，head 的等待状态从 -1 变为 0
2. T1 由于 T3 释放信号量被唤醒，调用 `tryAcquireShared`，假设返回值为 0（获取锁成功，但没有剩余资源量）
3. T4 调用 `releaseShared()`，此时 head.waitStatus 为 0（此时读到的 head 和 1 中为同一个 head），调用 `doReleaseShared()` 将等待状态置为 **PROPAGATE (-3)**
4. T1 获取信号量成功，调用 `setHeadAndPropagate` 时，读到 `h.waitStatus < 0`，从而调用 `doReleaseShared()` 唤醒 T2

ConcurrentHashMap 原理

1. JDK 7 HashMap 并发死链

测试代码

注意

- 要在 JDK 7 下运行，否则扩容机制和 hash 的计算方法都变了



- 以下测试代码是精心准备的，不要随便改动

```
public static void main(String[] args) {
    // 测试 java 7 中哪些数字的 hash 结果相等
    System.out.println("长度为16时，桶下标为1的key");
    for (int i = 0; i < 64; i++) {
        if (hash(i) % 16 == 1) {
            System.out.println(i);
        }
    }
    System.out.println("长度为32时，桶下标为1的key");
    for (int i = 0; i < 64; i++) {
        if (hash(i) % 32 == 1) {
            System.out.println(i);
        }
    }
    // 1, 35, 16, 50 当大小为16时，它们在一个桶内
    final HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
    // 放 12 个元素
    map.put(2, null);
    map.put(3, null);
    map.put(4, null);
    map.put(5, null);
    map.put(6, null);
    map.put(7, null);
    map.put(8, null);
    map.put(9, null);
    map.put(10, null);
    map.put(16, null);
    map.put(35, null);
    map.put(1, null);

    System.out.println("扩容前大小[main]:"+map.size());
    new Thread() {
        @Override
        public void run() {
            // 放第 13 个元素，发生扩容
            map.put(50, null);
            System.out.println("扩容后大小[Thread-0]:"+map.size());
        }
    }.start();
    new Thread() {
        @Override
        public void run() {
            // 放第 13 个元素，发生扩容
            map.put(50, null);
            System.out.println("扩容后大小[Thread-1]:"+map.size());
        }
    }.start();
}

final static int hash(Object k) {
    int h = 0;
```

```
if (0 != h && k instanceof String) {  
    return sun.misc.Hashing.stringHash32((String) k);  
}  
h ^= k.hashCode();  
h ^= (h >>> 20) ^ (h >>> 12);  
return h ^ (h >>> 7) ^ (h >>> 4);  
}
```

死链复现

调试工具使用 idea

在 HashMap 源码 590 行加断点

```
int newCapacity = newTable.length;
```

断点的条件如下，目的是让 HashMap 在扩容为 32 时，并且线程为 Thread-0 或 Thread-1 时停下来

```
newTable.length==32 &&  
(  
    Thread.currentThread().getName().equals("Thread-0") ||  
    Thread.currentThread().getName().equals("Thread-1")  
)
```

断点暂停方式选择 Thread，否则在调试 Thread-0 时，Thread-1 无法恢复运行

运行代码，程序在预料的断点位置停了下来，输出

```
长度为16时，桶下标为1的key  
1  
16  
35  
50  
长度为32时，桶下标为1的key  
1  
35  
扩容前大小[main]:12
```

接下来进入扩容流程调试

在 HashMap 源码 594 行加断点

```
Entry<K,V> next = e.next; // 593  
if (rehash) // 594  
// ...
```

这是为了观察 e 节点和 next 节点的状态，Thread-0 单步执行到 594 行，再 594 处再添加一个断点（条件 Thread.currentThread().getName().equals("Thread-0")）

这时可以在 Variables 面板观察到 e 和 next 变量，使用 `view as -> Object` 查看节点状态

```
e          (1)->(35)->(16)->null
next      (35)->(16)->null
```

在 Threads 面板选中 Thread-1 恢复运行，可以看到控制台输出新的内容如下，Thread-1 扩容已完成

```
newTable[1] (35)->(1)->null
```

扩容后大小:13

这时 Thread-0 还停在 594 处，Variables 面板变量的状态已经变化为

```
e          (1)->null
next      (35)->(1)->null
```

为什么呢，因为 Thread-1 扩容时链表也是后加入的元素放入链表头，因此链表就倒过来了，但 Thread-1 虽然结果正确，但它结束后 Thread-0 还要继续运行

接下来就可以单步调试（F8）观察死链的产生

下一轮循环到 594，将 e 搬迁到 newTable 链表头

```
newTable[1] (1)->null
e          (35)->(1)->null
next      (1)->null
```

下一轮循环到 594，将 e 搬迁到 newTable 链表头

```
newTable[1] (35)->(1)->null
e          (1)->null
next      null
```

再看看源码

```
e.next = newTable[1];
// 这时 e (1,35)
// 而 newTable[1] (35,1)->(1,35) 因为是同一个对象

newTable[1] = e;
// 再尝试将 e 作为链表头，死链已成

e = next;
// 虽然 next 是 null，会进入下一个链表的复制，但死链已经形成了
```

源码分析

HashMap 的并发死链发生在扩容时

```
// 将 table 迁移至 newTable
```

```
void transfer(Entry[] newTable, boolean rehash) {  
    int newCapacity = newTable.length;  
    for (Entry<K,V> e : table) {  
        while(null != e) {  
            Entry<K,V> next = e.next;  
            // 1 处  
            if (rehash) {  
                e.hash = null == e.key ? 0 : hash(e.key);  
            }  
            int i = indexFor(e.hash, newCapacity);  
            // 2 处  
            // 将新元素加入 newTable[i], 原 newTable[i] 作为新元素的 next  
            e.next = newTable[i];  
            newTable[i] = e;  
            e = next;  
        }  
    }  
}
```

假设 map 中初始元素是

原始链表，格式：[下标] (key,next)

[1] (1,35)->(35,16)->(16,null)

线程 a 执行到 1 处，此时局部变量 e 为 (1,35)，而局部变量 next 为 (35,16) 线程 a 挂起

线程 b 开始执行

第一次循环

[1] (1,null)

第二次循环

[1] (35,1)->(1,null)

第三次循环

[1] (35,1)->(1,null)

[17] (16,null)

切换回线程 a，此时局部变量 e 和 next 被恢复，引用没变但内容变了：e 的内容被改为 (1,null)，而 next 的内容被改为 (35,1) 并链向 (1,null)

第一次循环

[1] (1,null)

第二次循环，注意这时 e 是 (35,1) 并链向 (1,null) 所以 next 又是 (1,null)

[1] (35,1)->(1,null)

第三次循环，e 是 (1,null)，而 next 是 null，但 e 被放入链表头，这样 e.next 变成了 35 (2 处)

[1] (1,35)->(35,1)->(1,35)

已经是死链了

小结

- 究其原因，是因为在多线程环境下使用了非线程安全的 map 集合
- JDK 8 虽然将扩容算法做了调整，不再将元素加入链表头（而是保持与扩容前一样的顺序），但仍不意味着能够在多线程环境下能够安全扩容，还会出现其它问题（如扩容丢数据）

2. JDK 8 ConcurrentHashMap

重要属性和内部类

```
// 默认为 0
// 当初始化时，为 -1
// 当扩容时，为 -(1 + 扩容线程数)
// 当初始化或扩容完成后，为 下一轮的扩容的阈值大小
private transient volatile int sizeCtl;

// 整个 ConcurrentHashMap 就是一个 Node[]
static class Node<K,V> implements Map.Entry<K,V> {}

// hash 表
transient volatile Node<K,V>[] table;

// 扩容时的 新 hash 表
private transient volatile Node<K,V>[] nextTable;

// 扩容时如果某个 bin 迁移完毕，用 ForwardingNode 作为旧 table bin 的头结点
static final class ForwardingNode<K,V> extends Node<K,V> {}

// 用在 compute 以及 computeIfAbsent 时，用来占位，计算完成后替换为普通 Node
static final class ReservationNode<K,V> extends Node<K,V> {}

// 作为 treebin 的头节点，存储 root 和 first
static final class TreeBin<K,V> extends Node<K,V> {}

// 作为 treebin 的节点，存储 parent, left, right
static final class TreeNode<K,V> extends Node<K,V> {}
```

重要方法

```
// 获取 Node[] 中第 i 个 Node
static final <K,V> Node<K,V> tabAt(Node<K,V>[] tab, int i)

// cas 修改 Node[] 中第 i 个 Node 的值，c 为旧值，v 为新值
static final <K,V> boolean casTabAt(Node<K,V>[] tab, int i, Node<K,V> c, Node<K,V> v)

// 直接修改 Node[] 中第 i 个 Node 的值，v 为新值
static final <K,V> void setTabAt(Node<K,V>[] tab, int i, Node<K,V> v)
```

构造器分析

可以看到实现了懒惰初始化，在构造方法中仅仅计算了 table 的大小，以后在第一次使用时才会真正创建

```
public ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel) {
    if (!(loadFactor > 0.0f) || initialCapacity < 0 || concurrencyLevel <= 0)
        throw new IllegalArgumentException();
    if (initialCapacity < concurrencyLevel) // Use at least as many bins
        initialCapacity = concurrencyLevel; // as estimated threads
    long size = (long)(1.0 + (long)initialCapacity / loadFactor);
    // tableSizeFor 仍然是保证计算的大小是 2^n, 即 16,32,64 ...
    int cap = (size >= (long)MAXIMUM_CAPACITY) ?
        MAXIMUM_CAPACITY : tableSizeFor((int)size);
    this.sizeCtl = cap;
}
```

get 流程

```
public V get(Object key) {
    Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;
    // spread 方法能确保返回结果是正数
    int h = spread(key.hashCode());
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (e = tabAt(tab, (n - 1) & h)) != null) {
        // 如果头结点已经是要查找的 key
        if ((eh = e.hash) == h) {
            if ((ek = e.key) == key || (ek != null && key.equals(ek)))
                return e.val;
        }
        // hash 为负数表示该 bin 在扩容中或是 treebin, 这时调用 find 方法来查找
        else if (eh < 0)
            return (p = e.find(h, key)) != null ? p.val : null;
        // 正常遍历链表, 用 equals 比较
        while ((e = e.next) != null) {
            if (e.hash == h &&
                ((ek = e.key) == key || (ek != null && key.equals(ek))))
                return e.val;
        }
    }
    return null;
}
```

put 流程

以下数组简称 (table)，链表简称 (bin)

```
public V put(K key, V value) {
```



```
        return putVal(key, value, false);
    }

    final V putVal(K key, V value, boolean onlyIfAbsent) {
        if (key == null || value == null) throw new NullPointerException();
        // 其中 spread 方法会综合高位低位，具有更好的 hash 性
        int hash = spread(key.hashCode());
        int binCount = 0;
        for (Node<K,V>[] tab = table;;) {
            // f 是链表头节点
            // fh 是链表头结点的 hash
            // i 是链表在 table 中的下标
            Node<K,V> f; int n, i, fh;
            // 要创建 table
            if (tab == null || (n = tab.length) == 0)
                // 初始化 table 使用了 cas，无需 synchronized 创建成功，进入下一轮循环
                tab = initTable();
            // 要创建链表头节点
            else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
                // 添加链表头使用了 cas，无需 synchronized
                if (casTabAt(tab, i, null,
                    new Node<K,V>(hash, key, value, null)))
                    break;
            }
            // 帮忙扩容
            else if ((fh = f.hash) == MOVED)
                // 帮忙之后，进入下一轮循环
                tab = helpTransfer(tab, f);
            else {
                V oldVal = null;
                // 锁住链表头节点
                synchronized (f) {
                    // 再次确认链表头节点没有被移动
                    if (tabAt(tab, i) == f) {
                        // 链表
                        if (fh >= 0) {
                            binCount = 1;
                            // 遍历链表
                            for (Node<K,V> e = f;; ++binCount) {
                                K ek;
                                // 找到相同的 key
                                if (e.hash == hash &&
                                    ((ek = e.key) == key ||
                                    (ek != null && key.equals(ek)))) {
                                    oldVal = e.val;
                                    // 更新
                                    if (!onlyIfAbsent)
                                        e.val = value;
                                    break;
                                }
                                Node<K,V> pred = e;
                                // 已经是最后的节点了，新增 Node，追加至链表尾
                                if ((e = e.next) == null) {
```




```
        pred.next = new Node<K,V>(hash, key,
                                   value, null);
        break;
    }
}
// 红黑树
else if (f instanceof TreeBin) {
    Node<K,V> p;
    binCount = 2;
    // putTreeVal 会看 key 是否已经在树中，是，则返回对应的 TreeNode
    if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
                                           value)) != null) {
        oldVal = p.val;
        if (!onlyIfAbsent)
            p.val = value;
    }
}
// 释放链表头节点的锁
}

if (binCount != 0) {
    if (binCount >= TREEIFY_THRESHOLD)
        // 如果链表长度 >= 树化阈值(8)，进行链表转为红黑树
        treeifyBin(tab, i);
    if (oldVal != null)
        return oldVal;
    break;
}
}
// 增加 size 计数
addCount(1L, binCount);
return null;
}

private final Node<K,V>[] initTable() {
    Node<K,V>[] tab; int sc;
    while ((tab = table) == null || tab.length == 0) {
        if ((sc = sizeCtl) < 0)
            Thread.yield();
        // 尝试将 sizeCtl 设置为 -1 (表示初始化 table)
        else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
            // 获得锁，创建 table，这时其它线程会在 while() 循环中 yield 直至 table 创建
            try {
                if ((tab = table) == null || tab.length == 0) {
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                    Node<K,V>[] nt = (Node<K,V>[]) new Node<?,?>[n];
                    table = tab = nt;
                    sc = n - (n >>> 2);
                }
            }
        }
    }
    finally {
```



```
        sizeCtl = sc;
    }
    break;
}
}
return tab;
}

// check 是之前 binCount 的个数
private final void addCount(long x, int check) {
    CounterCell[] as; long b, s;
    if (
        // 已经有了 counterCells, 向 cell 累加
        (as = counterCells) != null ||
        // 还没有, 向 baseCount 累加
        !U.compareAndSwapLong(this, BASECOUNT, b = baseCount, s = b + x)
    ) {
        CounterCell a; long v; int m;
        boolean uncontended = true;
        if (
            // 还没有 counterCells
            as == null || (m = as.length - 1) < 0 ||
            // 还没有 cell
            (a = as[ThreadLocalRandom.getProbe() & m]) == null ||
            // cell cas 增加计数失败
            !(uncontended = U.compareAndSwapLong(a, CELLVALUE, v = a.value, v + x))
        ) {
            // 创建累加单元数组和cell, 累加重试
            fullAddCount(x, uncontended);
            return;
        }
        if (check <= 1)
            return;
        // 获取元素个数
        s = sumCount();
    }
    if (check >= 0) {
        Node<K,V>[] tab, nt; int n, sc;
        while (s >= (long)(sc = sizeCtl) && (tab = table) != null &&
            (n = tab.length) < MAXIMUM_CAPACITY) {
            int rs = resizeStamp(n);
            if (sc < 0) {
                if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                    sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
                    transferIndex <= 0)
                    break;
                // newtable 已经创建了, 帮忙扩容
                if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
                    transfer(tab, nt);
            }
            // 需要扩容, 这时 newtable 未创建
            else if (U.compareAndSwapInt(this, SIZECTL, sc,
                (rs << RESIZE_STAMP_SHIFT) + 2))
```

```
        transfer(tab, null);
        s = sumCount();
    }
}
```

size 计算流程

size 计算实际发生在 put，remove 改变集合元素的操作之中

- 没有竞争发生，向 baseCount 累加计数
- 有竞争发生，新建 counterCells，向其中的一个 cell 累加计数
 - counterCells 初始有两个 cell
 - 如果计数竞争比较激烈，会创建新的 cell 来累加计数

```
public int size() {
    long n = sumCount();
    return ((n < 0L) ? 0 :
            (n > (long)Integer.MAX_VALUE) ? Integer.MAX_VALUE :
            (int)n);
}

final long sumCount() {
    CounterCell[] as = counterCells; CounterCell a;
    // 将 baseCount 计数与所有 cell 计数累加
    long sum = baseCount;
    if (as != null) {
        for (int i = 0; i < as.length; ++i) {
            if ((a = as[i]) != null)
                sum += a.value;
        }
    }
    return sum;
}
```

Java 8 数组 (Node) + (链表 Node | 红黑树 TreeNode) 以下数组简称 (table)，链表简称 (bin)

- 初始化，使用 cas 来保证并发安全，懒惰初始化 table
- 树化，当 table.length < 64 时，先尝试扩容，超过 64 时，并且 bin.length > 8 时，会将链表树化，树化过程会用 synchronized 锁住链表头
- put，如果该 bin 尚未创建，只需要使用 cas 创建 bin；如果已经有了，锁住链表头进行后续 put 操作，元素添加至 bin 的尾部
- get，无锁操作仅需要保证可见性，扩容过程中 get 操作拿到的是 ForwardingNode 它会让 get 操作在新 table 进行搜索
- 扩容，扩容时以 bin 为单位进行，需要对 bin 进行 synchronized，但这时妙的是其它竞争线程也不是无事可做，它们会帮助把其它 bin 进行扩容，扩容时平均只有 1/6 的节点会把复制到新 table 中
- size，元素个数保存在 baseCount 中，并发时的个数变动保存在 CounterCell[] 当中。最后统计数量时累加即可

源码分析 <http://www.importnew.com/28263.html>

其它实现 [Cliff Click's high scale lib](#)

3. JDK 7 ConcurrentHashMap

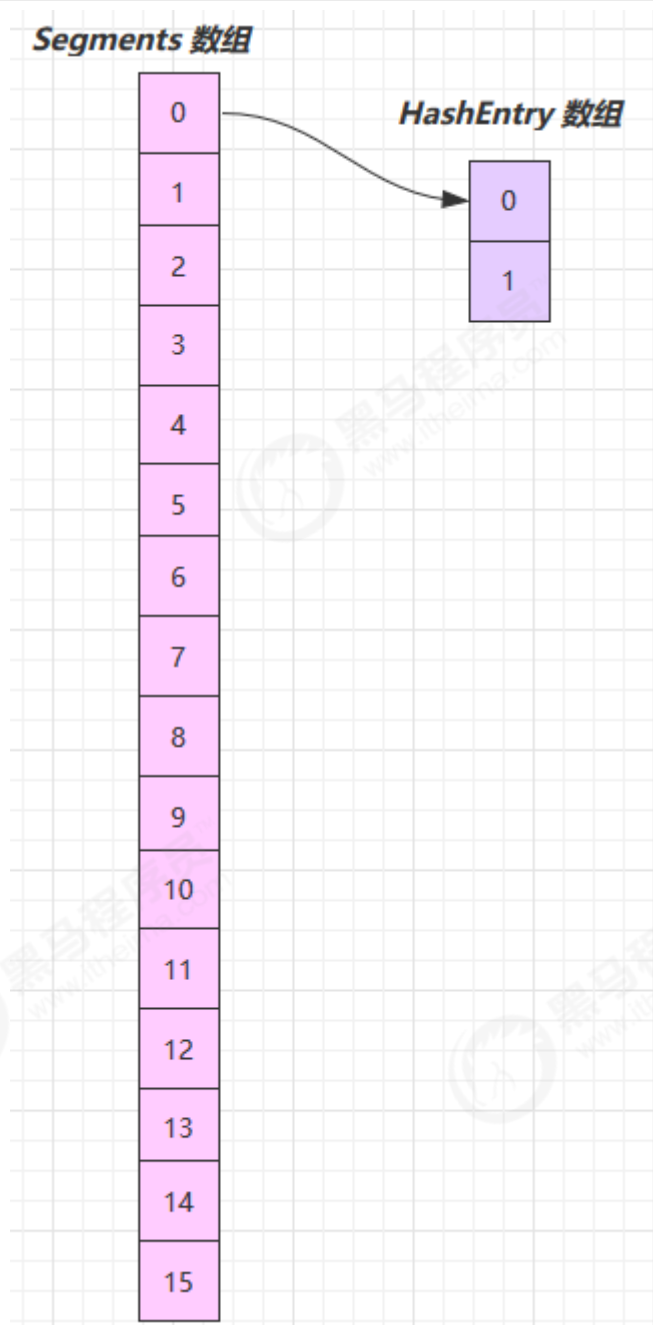
它维护了一个 segment 数组，每个 segment 对应一把锁

- 优点：如果多个线程访问不同的 segment，实际是没有冲突的，这与 jdk8 中是类似的
- 缺点：Segments 数组默认大小为16，这个容量初始化指定后就不能改变了，并且不是懒惰初始化

构造器分析

```
public ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel) {
    if (!(loadFactor > 0) || initialCapacity < 0 || concurrencyLevel <= 0)
        throw new IllegalArgumentException();
    if (concurrencyLevel > MAX_SEGMENTS)
        concurrencyLevel = MAX_SEGMENTS;
    // ssize 必须是 2^n, 即 2, 4, 8, 16 ... 表示了 segments 数组的大小
    int sshift = 0;
    int ssize = 1;
    while (ssize < concurrencyLevel) {
        ++sshift;
        ssize <<= 1;
    }
    // segmentShift 默认是 32 - 4 = 28
    this.segmentShift = 32 - sshift;
    // segmentMask 默认是 15 即 0000 0000 0000 1111
    this.segmentMask = ssize - 1;
    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
    int c = initialCapacity / ssize;
    if (c * ssize < initialCapacity)
        ++c;
    int cap = MIN_SEGMENT_TABLE_CAPACITY;
    while (cap < c)
        cap <<= 1;
    // 创建 segments and segments[0]
    Segment<K,V> s0 =
        new Segment<K,V>(loadFactor, (int)(cap * loadFactor),
            (HashEntry<K,V>[])new HashEntry[cap]);
    Segment<K,V>[] ss = (Segment<K,V>[])new Segment[ssize];
    UNSAFE.putOrderedObject(ss, SBASE, s0); // ordered write of segments[0]
    this.segments = ss;
}
```

构造完成，如下图所示



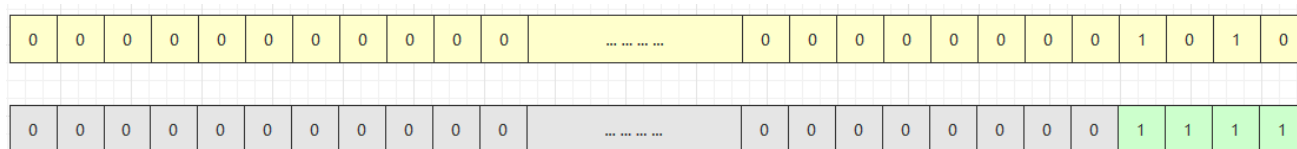
可以看到 `ConcurrentHashMap` 没有实现懒情初始化，空间占用不友好

其中 `this.segmentShift` 和 `this.segmentMask` 的作用是决定将 key 的 hash 结果匹配到哪个 segment

例如，根据某一 hash 值求 segment 位置，先将高位向低位移动 `this.segmentShift` 位



结果再与 `this.segmentMask` 做位于运算，最终得到 1010 即下标为 10 的 segment



put 流程

```
public V put(K key, V value) {
    Segment<K,V> s;
    if (value == null)
        throw new NullPointerException();
    int hash = hash(key);
    // 计算出 segment 下标
    int j = (hash >>> segmentShift) & segmentMask;

    // 获得 segment 对象，判断是否为 null，是则创建该 segment
    if ((s = (Segment<K,V>)UNSAFE.getObject(
        (segments, (j << SSHIFT) + SBASE))) == null) {
        // 这时不能确定是否真的为 null，因为其它线程也发现该 segment 为 null，
        // 因此在 ensureSegment 里用 cas 方式保证该 segment 安全性
        s = ensureSegment(j);
    }
    // 进入 segment 的 put 流程
    return s.put(key, hash, value, false);
}
```

segment 继承了可重入锁 (ReentrantLock) ，它的 put 方法为

```
final V put(K key, int hash, V value, boolean onlyIfAbsent) {
    // 尝试加锁
    HashEntry<K,V> node = tryLock() ? null :
        // 如果不成功，进入 scanAndLockForPut 流程
        // 如果是多核 cpu 最多 tryLock 64 次，进入 lock 流程
        // 在尝试期间，还可以顺便看该节点在链表中有无，如果没有顺便创建出来
        scanAndLockForPut(key, hash, value);

    // 执行到这里 segment 已经被成功加锁，可以安全执行
    V oldValue;
    try {
        HashEntry<K,V>[] tab = table;
        int index = (tab.length - 1) & hash;
        HashEntry<K,V> first = entryAt(tab, index);
        for (HashEntry<K,V> e = first;;) {
            if (e != null) {
                // 更新
                K k;
                if ((k = e.key) == key ||
                    (e.hash == hash && key.equals(k))) {
                    oldValue = e.value;
                    if (!onlyIfAbsent) {
                        e.value = value;
                        ++modCount;
                    }
                }
            }
        }
    }
```

```
        break;
    }
    e = e.next;
}
else {
    // 新增
    // 1) 之前等待锁时, node 已经被创建, next 指向链表头
    if (node != null)
        node.setNext(first);
    else
        // 2) 创建新 node
        node = new HashEntry<K,V>(hash, key, value, first);
    int c = count + 1;
    // 3) 扩容
    if (c > threshold && tab.length < MAXIMUM_CAPACITY)
        rehash(node);
    else
        // 将 node 作为链表头
        setEntryAt(tab, index, node);
    ++modCount;
    count = c;
    oldValue = null;
    break;
}
}
} finally {
    unlock();
}
return oldValue;
}
```

rehash 流程

发生在 put 中，因为此时已经获得了锁，因此 rehash 时不需要考虑线程安全

```
private void rehash(HashEntry<K,V> node) {
    HashEntry<K,V>[] oldTable = table;
    int oldCapacity = oldTable.length;
    int newCapacity = oldCapacity << 1;
    threshold = (int)(newCapacity * loadFactor);
    HashEntry<K,V>[] newTable =
        (HashEntry<K,V>[]) new HashEntry[newCapacity];
    int sizeMask = newCapacity - 1;
    for (int i = 0; i < oldCapacity; i++) {
        HashEntry<K,V> e = oldTable[i];
        if (e != null) {
            HashEntry<K,V> next = e.next;
            int idx = e.hash & sizeMask;
            if (next == null) // Single node on list
                newTable[idx] = e;
            else { // Reuse consecutive sequence at same slot
                HashEntry<K,V> lastRun = e;
```



```

        int lastIdx = idx;
        // 过一遍链表，尽可能把 rehash 后 idx 不变的节点重用
        for (HashEntry<K,V> last = next;
             last != null;
             last = last.next) {
            int k = last.hash & sizeMask;
            if (k != lastIdx) {
                lastIdx = k;
                lastRun = last;
            }
        }
        newTable[lastIdx] = lastRun;
        // 剩余节点需要新建
        for (HashEntry<K,V> p = e; p != lastRun; p = p.next) {
            V v = p.value;
            int h = p.hash;
            int k = h & sizeMask;
            HashEntry<K,V> n = newTable[k];
            newTable[k] = new HashEntry<K,V>(h, p.key, v, n);
        }
    }
}

// 扩容完成，才加入新的节点
int nodeIndex = node.hash & sizeMask; // add the new node
node.setNext(newTable[nodeIndex]);
newTable[nodeIndex] = node;

// 替换为新的 HashEntry table
table = newTable;
}

```

附，调试代码

```

public static void main(String[] args) {
    ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<>();
    for (int i = 0; i < 1000; i++) {
        int hash = hash(i);
        int segmentIndex = (hash >>> 28) & 15;
        if (segmentIndex == 4 && hash % 8 == 2) {
            System.out.println(i + "\t" + segmentIndex + "\t" + hash % 2 + "\t" + hash % 4 +
                               "\t" + hash % 8);
        }
    }
    map.put(1, "value");
    map.put(15, "value"); // 2 扩容为 4 15 的 hash%8 与其他不同
    map.put(169, "value");
    map.put(197, "value"); // 4 扩容为 8
    map.put(341, "value");
    map.put(484, "value");
    map.put(545, "value"); // 8 扩容为 16
    map.put(912, "value");
}

```




```
map.put(941, "value");
System.out.println("ok");
}

private static int hash(Object k) {
    int h = 0;

    if ((0 != h) && (k instanceof String)) {
        return sun.misc.Hashing.stringHash32((String) k);
    }

    h ^= k.hashCode();

    // Spread bits to regularize both segment and index locations,
    // using variant of single-word Wang/Jenkins hash.
    h += (h << 15) ^ 0xffffcd7d;
    h ^= (h >>> 10);
    h += (h << 3);
    h ^= (h >>> 6);
    h += (h << 2) + (h << 14);
    int v = h ^ (h >>> 16);
    return v;
}
```

get 流程

get 时并未加锁，用了 UNSAFE 方法保证了可见性，扩容过程中，get 先发生就从旧表取内容，get 后发生就从新表取内容

```
public V get(Object key) {
    Segment<K,V> s; // manually integrate access methods to reduce overhead
    HashEntry<K,V>[] tab;
    int h = hash(key);
    // u 为 segment 对象在数组中的偏移量
    long u = (((h >>> segmentShift) & segmentMask) << SSHIFT) + SBASE;
    // s 即为 segment
    if ((s = (Segment<K,V>)UNSAFE.getObjectVolatile(segments, u)) != null &&
        (tab = s.table) != null) {
        for (HashEntry<K,V> e = (HashEntry<K,V>) UNSAFE.getObjectVolatile
            (tab, (((long)((tab.length - 1) & h)) << TSHIFT) + TBASE);
            e != null; e = e.next) {
            K k;
            if ((k = e.key) == key || (e.hash == h && key.equals(k)))
                return e.value;
        }
    }
    return null;
}
```

size 计算流程



- 计算元素个数前，先不加锁计算两次，如果前后两次结果如一样，认为个数正确返回
- 如果不一样，进行重试，重试次数超过 3，将所有 segment 锁住，重新计算个数返回

```
public int size() {  
    // Try a few times to get accurate count. On failure due to  
    // continuous async changes in table, resort to locking.  
    final Segment<K,V>[] segments = this.segments;  
    int size;  
    boolean overflow; // true if size overflows 32 bits  
    long sum;          // sum of modCounts  
    long last = 0L;    // previous sum  
    int retries = -1; // first iteration isn't retry  
    try {  
        for (;;) {  
            if (retries++ == RETRIES_BEFORE_LOCK) {  
                // 超过重试次数，需要创建所有 segment 并加锁  
                for (int j = 0; j < segments.length; ++j)  
                    ensureSegment(j).lock(); // force creation  
            }  
            sum = 0L;  
            size = 0;  
            overflow = false;  
            for (int j = 0; j < segments.length; ++j) {  
                Segment<K,V> seg = segmentAt(segments, j);  
                if (seg != null) {  
                    sum += seg.modCount;  
                    int c = seg.count;  
                    if (c < 0 || (size += c) < 0)  
                        overflow = true;  
                }  
            }  
            if (sum == last)  
                break;  
            last = sum;  
        }  
    } finally {  
        if (retries > RETRIES_BEFORE_LOCK) {  
            for (int j = 0; j < segments.length; ++j)  
                segmentAt(segments, j).unlock();  
        }  
    }  
    return overflow ? Integer.MAX_VALUE : size;  
}
```

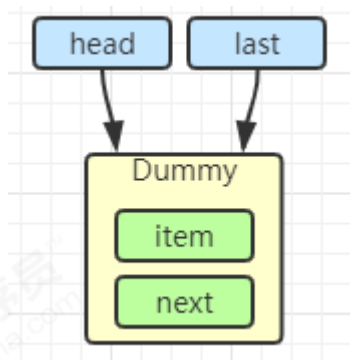
LinkedBlockingQueue 原理

1. 基本的入队出队

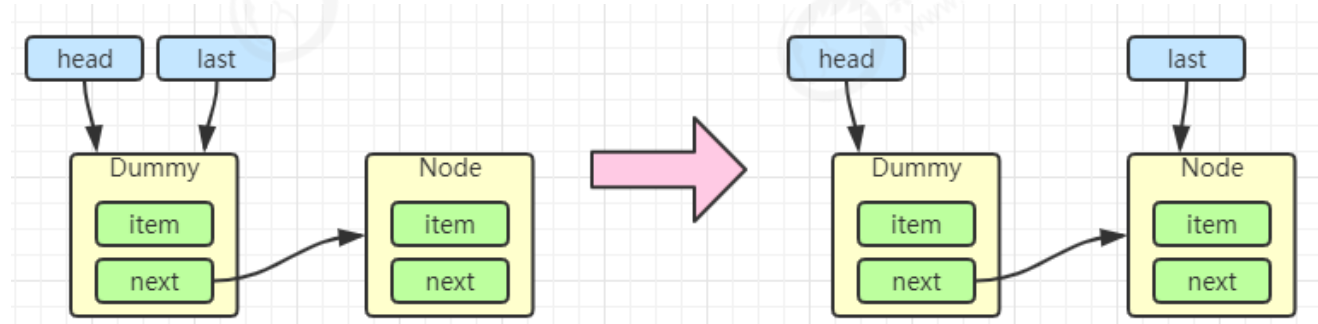
```
public class LinkedBlockingQueue<E> extends AbstractQueue<E>  
    implements BlockingQueue<E>, java.io.Serializable {  
    static class Node<E> {
```

```
E item;  
  
/**  
 * 下列三种情况之一  
 * - 真正的后继节点  
 * - 自己，发生在出队时  
 * - null，表示是没有后继节点，是最后了  
 */  
Node<E> next;  
  
Node(E x) { item = x; }  
}  
}
```

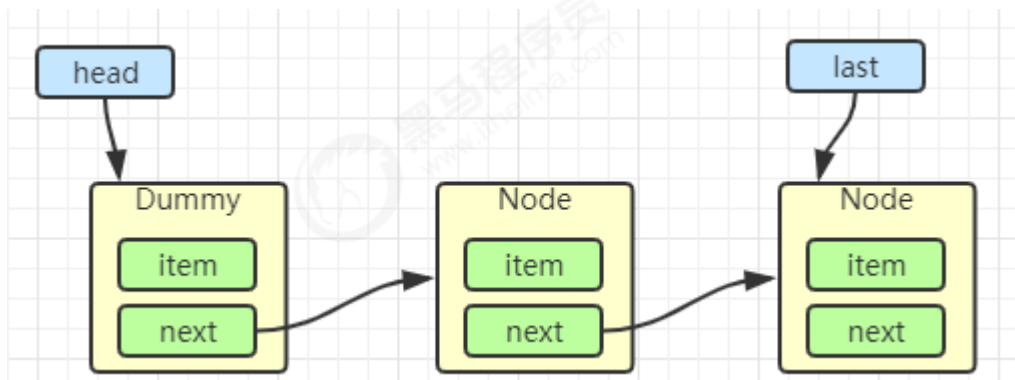
初始化链表 `last = head = new Node<E>(null);` Dummy 节点用来占位，item 为 null



当一个节点入队 `last = last.next = node;`



再来一个节点入队 `last = last.next = node;`



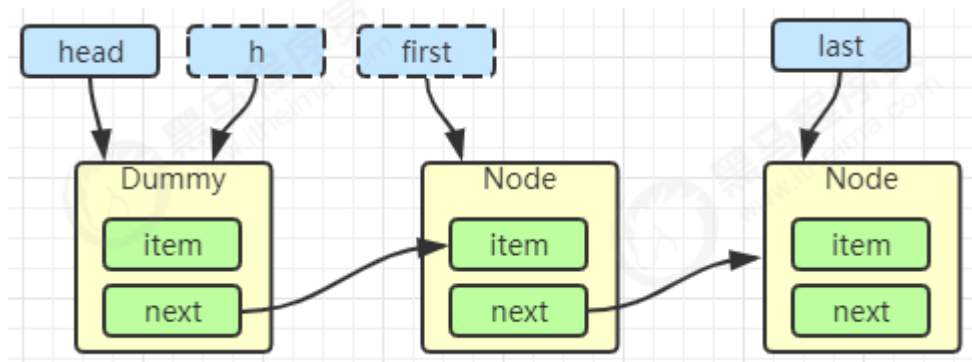
出队

```
Node<E> h = head;  
Node<E> first = h.next;  
h.next = h; // help GC  
head = first;  
E x = first.item;  
first.item = null;  
return x;
```

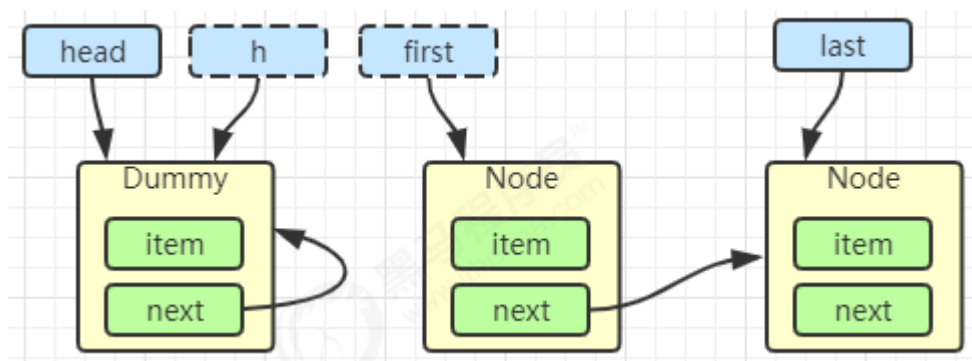
h = head



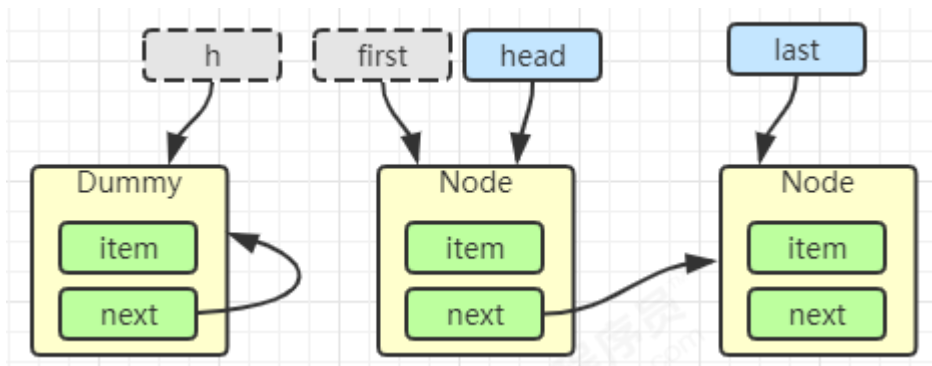
first = h.next



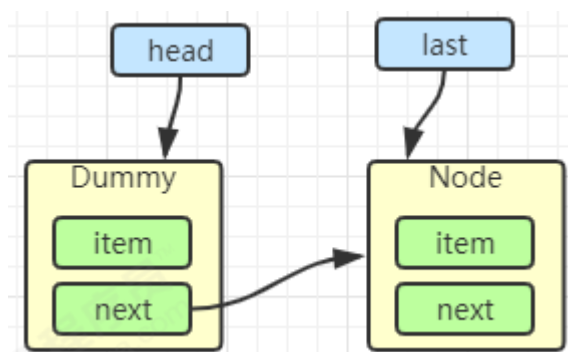
h.next = h



head = first



```
E x = first.item;  
first.item = null;  
return x;
```



2. 加锁分析

==高明之处==在于用了两把锁和 dummy 节点

- 用一把锁，同一时刻，最多只允许有一个线程（生产者或消费者，二选一）执行
- 用两把锁，同一时刻，可以允许两个线程同时（一个生产者与一个消费者）执行
 - 消费者与消费者线程仍然串行
 - 生产者与生产者线程仍然串行

线程安全分析

- 当节点总数大于 2 时（包括 dummy 节点），putLock 保证的是 last 节点的线程安全，takeLock 保证的是 head 节点的线程安全。两把锁保证了入队和出队没有竞争
- 当节点总数等于 2 时（即一个 dummy 节点，一个正常节点）这时候，仍然是两把锁锁两个对象，不会竞争
- 当节点总数等于 1 时（就一个 dummy 节点）这时 take 线程会被 notEmpty 条件阻塞，有竞争，会阻塞

```
// 用于 put(阻塞) offer(非阻塞)  
private final ReentrantLock putLock = new ReentrantLock();  
  
// 用户 take(阻塞) poll(非阻塞)  
private final ReentrantLock takeLock = new ReentrantLock();
```

put 操作

```
public void put(E e) throws InterruptedException {
```



```
if (e == null) throw new NullPointerException();
int c = -1;
Node<E> node = new Node<E>(e);
final ReentrantLock putLock = this.putLock;
// count 用来维护元素计数
final AtomicInteger count = this.count;
putLock.lockInterruptibly();
try {
    // 满了等待
    while (count.get() == capacity) {
        // 倒过来读就好：等待 notFull
        notFull.await();
    }
    // 有空位，入队且计数加一
    enqueue(node);
    c = count.getAndIncrement();
    // 除了自己 put 以外，队列还有空位，由自己叫醒其他 put 线程
    if (c + 1 < capacity)
        notFull.signal();
} finally {
    putLock.unlock();
}
// 如果队列中有一个元素，叫醒 take 线程
if (c == 0)
    // 这里调用的是 notEmpty.signal() 而不是 notEmpty.signalAll() 是为了减少竞争
    signalNotEmpty();
}
```

take 操作

```
public E take() throws InterruptedException {
    E x;
    int c = -1;
    final AtomicInteger count = this.count;
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lockInterruptibly();
    try {
        while (count.get() == 0) {
            notEmpty.await();
        }
        x = dequeue();
        c = count.getAndDecrement();
        if (c > 1)
            notEmpty.signal();
    } finally {
        takeLock.unlock();
    }
    // 如果队列中只有一个空位时，叫醒 put 线程
    // 如果有多个线程进行出队，第一个线程满足 c == capacity，但后续线程 c < capacity
    if (c == capacity)
        // 这里调用的是 notFull.signal() 而不是 notFull.signalAll() 是为了减少竞争
        signalNotFull();
}
```

```
        return x;  
    }
```

由 put 唤醒 put 是为了避免信号不足

3. 性能比较

主要列举 LinkedBlockingQueue 与 ArrayBlockingQueue 的性能比较

- Linked 支持有界，Array 强制有界
- Linked 实现是链表，Array 实现是数组
- Linked 是懒惰的，而 Array 需要提前初始化 Node 数组
- Linked 每次入队会生成新 Node，而 Array 的 Node 是提前创建好的
- Linked 两把锁，Array 一把锁

ConcurrentLinkedQueue 原理

1. 模仿 ConcurrentLinkedQueue

初始代码

```
package cn.itcast.concurrent.thirdpart.test;  
  
import java.util.Collection;  
import java.util.Iterator;  
import java.util.Queue;  
import java.util.concurrent.atomic.AtomicReference;  
  
public class Test3 {  
  
    public static void main(String[] args) {  
        MyQueue<String> queue = new MyQueue<>();  
        queue.offer("1");  
        queue.offer("2");  
        queue.offer("3");  
        System.out.println(queue);  
    }  
}
```



```
class MyQueue<E> implements Queue<E> {

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        for (Node<E> p = head; p != null; p = p.next.get()) {
            E item = p.item;
            if (item != null) {
                sb.append(item).append("->");
            }
        }
        sb.append("null");
        return sb.toString();
    }

    @Override
    public int size() {
        return 0;
    }

    @Override
    public boolean isEmpty() {
        return false;
    }

    @Override
    public boolean contains(Object o) {
        return false;
    }

    @Override
    public Iterator<E> iterator() {
        return null;
    }

    @Override
    public Object[] toArray() {
        return new Object[0];
    }

    @Override
    public <T> T[] toArray(T[] a) {
        return null;
    }

    @Override
    public boolean add(E e) {
        return false;
    }

    @Override
    public boolean remove(Object o) {
```




```
        return false;
    }

    @Override
    public boolean containsAll(Collection<?> c) {
        return false;
    }

    @Override
    public boolean addAll(Collection<? extends E> c) {
        return false;
    }

    @Override
    public boolean removeAll(Collection<?> c) {
        return false;
    }

    @Override
    public boolean retainAll(Collection<?> c) {
        return false;
    }

    @Override
    public void clear() {

    }

    @Override
    public E remove() {
        return null;
    }

    @Override
    public E element() {
        return null;
    }

    @Override
    public E peek() {
        return null;
    }

    public MyQueue() {
        head = last = new Node<>(null, null);
    }

    private volatile Node<E> last;
    private volatile Node<E> head;

    private E dequeue() {
```



```
        /*Node<E> h = head;
        Node<E> first = h.next;
        h.next = h;
        head = first;
        E x = first.item;
        first.item = null;
        return x;*/
        return null;
    }

    @Override
    public E poll() {
        return null;
    }

    @Override
    public boolean offer(E e) {
        return true;
    }

    static class Node<E> {
        volatile E item;

        public Node(E item, Node<E> next) {
            this.item = item;
            this.next = new AtomicReference<>(next);
        }

        AtomicReference<Node<E>> next;
    }
}
```

offer

```
public boolean offer(E e) {
    Node<E> n = new Node<>(e, null);
    while(true) {
        // 获取尾节点
        AtomicReference<Node<E>> next = last.next;
        // S1: 真正尾节点的 next 是 null, cas 从 null 到新节点
        if(next.compareAndSet(null, n)) {
            // 这时的 last 已经是倒数第二, next 不为空了, 其它线程的 cas 肯定失败
            // S2: 更新 last 为倒数第一的节点
            last = n;
            return true;
        }
    }
}
```