网上参考

参考了这个网上文档，但它的内容我没有读懂，觉得它的实现没有做到Lock Free（或者是文章太简略或则我太愚钝，没有领会到它的妙趣），anyway，我借鉴了它的一个思想，做了一个自己的设计，应该比较好理解，如下

Sorted Set Double Linked List

图示

我们假设针对的是一个Lock Free Set Double Linked List，即每个元素都是唯一的，而且是sorted，因为我们最终目的是为了SkipList，所以这个没毛病

我们dummy头结点head和尾tail，并且认为其虚拟key值代表-infinite和+infinite（因为对于这两个node，我们实际比较的是指针），所以一个范例如下

head ->(<-) Node(1) ->(<-) Node(2) ->(<-) Node(5) ->(<-) tail

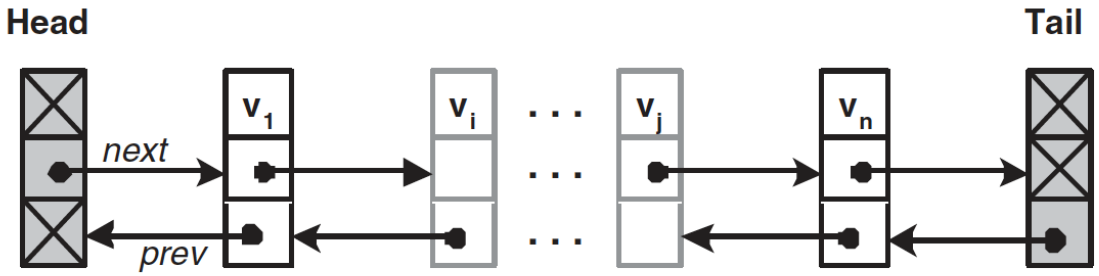


Fig. 1. The doubly linked list data structure

三个操作：find\insert\delete

我们支持三个基本操作（未来你可以加iterator），insert(key)和delete(key)和find(key)

find(search_key)

if return true, [pred, curr]

我们在list找到一个键值相同的element，就是curr，pred是curr其前一个node

else return false, [pred, curr]

其中pred node在curr node前面，并且pred_key < search_key < curr_key

注意：因为是thread并发，这个find()仅是当时的snapshot，随时可能被其他thread修改

insert(key)

通过find()定位，然后在curr前insert。如果失败，返回false，因为可能被并发的thread同时且抢先也insert一个同样的key。否则，它必须成功（可能需要再次find），即使这时，[pred,curr]中间又插入了其他的key或发生了其他的变化。即失败的唯一可能是：有相同的key出现在list里。

delete(key)

通过find()定位，然后删除curr。如果失败，返回false，因为可能被并发的thread同时且抢先delete一个同样的key。否则，它必须成功（可能需要再次find），即使这时，[pred,curr]发生了任何变化。即失败的唯一可能是：我们要删除的key消失了。

compound pointer

因为硬件只能支持64位cas原子操作，所以我们利用现代64位操作系统里内存地址指针实际只用到48位，或者内存分配保证是word对齐（因此分配的地址保证指针后4bit都是0）。这样，有两种编码方式可选择，去解决下面的问题（虽然我个人喜欢第二种）

为了好理解，我这里用48位，即我们在指针的前16位设置一些值，和内存物理地址实际用到48位组合在一起，形成一个compound pointer，如下

compound pointer = (16位flag值）+ （48位物理内存地址）

这样，当用到真实的物理内存地址时，如果我们访问的指针变量是compound pointer，我们必须取其后48位进行实际的内存操作。但同时，我们可以在compound pointer里添加一些额外的flag信息，在使用cas时，可以利用更多的信息来做一些有趣的功能，从而避免64位hardware atomic的CPU限制

FLAG值

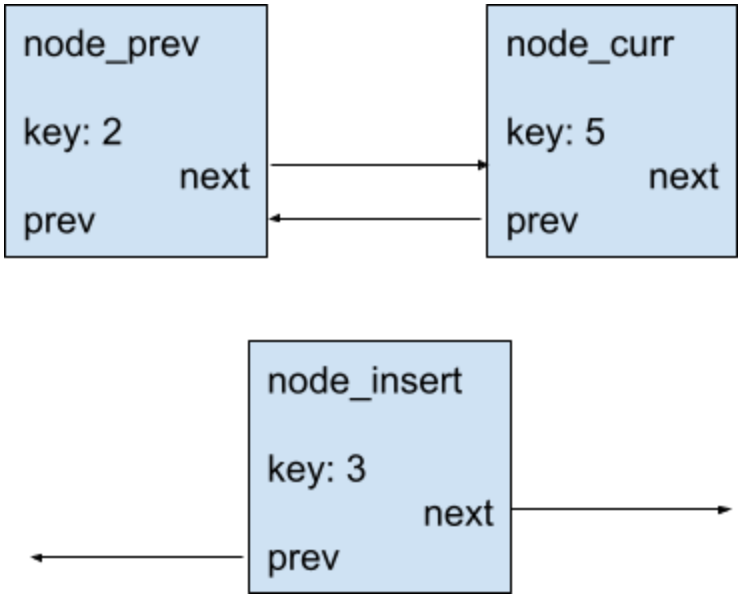
我们有下面一些flag，先列表，暂不解释，后面会一个接一个地解释

macro名	值	对应bit
FLAG_FREE	0	0000
FLAG_WILL_INSERT	1	0001
FLAG_NOW_INSERTING	2	0010
FLAG_WILL_DELETE	3	0011
FLAG_NOW_DELETING	4	0100

- NOTE 1: 当指针刚被new或其他内存分配器返回时（因为new实际违反了lock free，内部有锁），其前16位都是0，就是FLAG_FREE
- NOTE 2: FLAG信息，我们只存在node的next，这个compound pointer变量上

Insert初步分析

先看insert，假设我们已经通过find()定位了位置，如下图



在试图insert中，由于并发race的原因，有很多东西可能导致我们失败，比如：

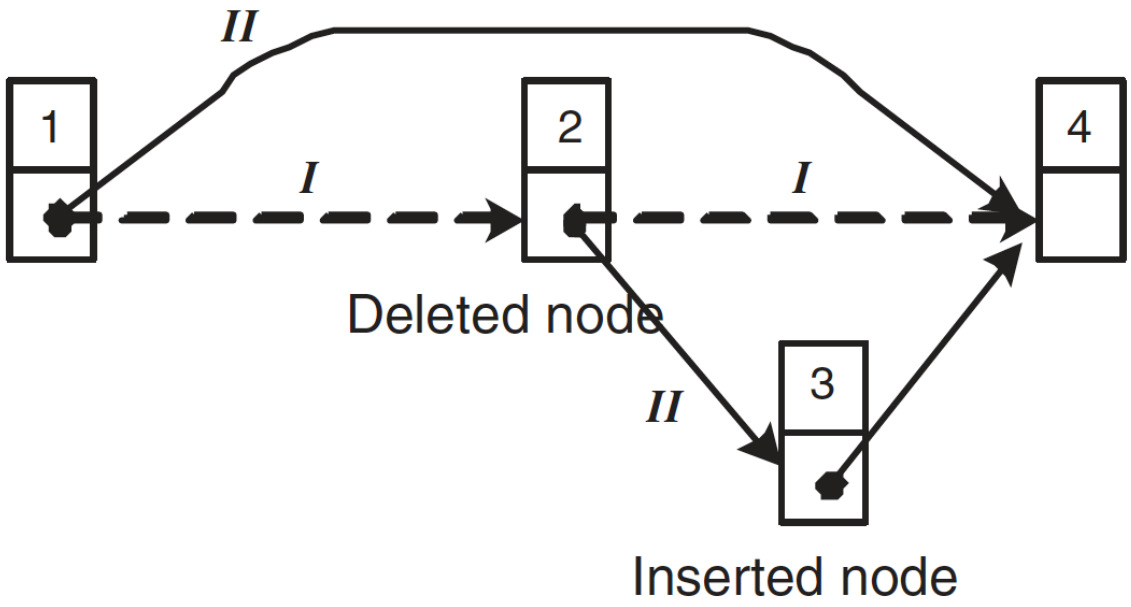
1. 另外一个线程也同时而且抢先插入了3

2. 我们准备插入5之前，突然前面又插入了4（因此，我们如果连接2->3->5就不对了）

3. 我们准备插入5之前，5被删除了（同理2也有这种可能）

4. 更糟糕的是，对于每一个需要变化的node，我们都需要设置其next和prev两个指针，而这不可能在一个硬件原子里完成，所以，可能出现和并发的另外一个thread，其指针被改成麻花，从而导致整个数据结构崩溃

5. 还有一种很可怕的corner case，就是我们看上去好像成功了，实际却是失败的，如下图（一个引用图：网上文档有描述）



copyright of the original author

这个可怕的情况是：我们成功地在2和4之间插入了3，但还没有visible，这时，另外一个删除线程做了一个快照，只看到了1->2->4，这个删除线程删除了2，让1 link了4, 所以，3虽然成功地加入了2和4之间，但实际却是失败的。

我们期望的伊甸园

综上而言，我们希望有这么一个完美的环境，在这个伊甸园里，有下面特征：

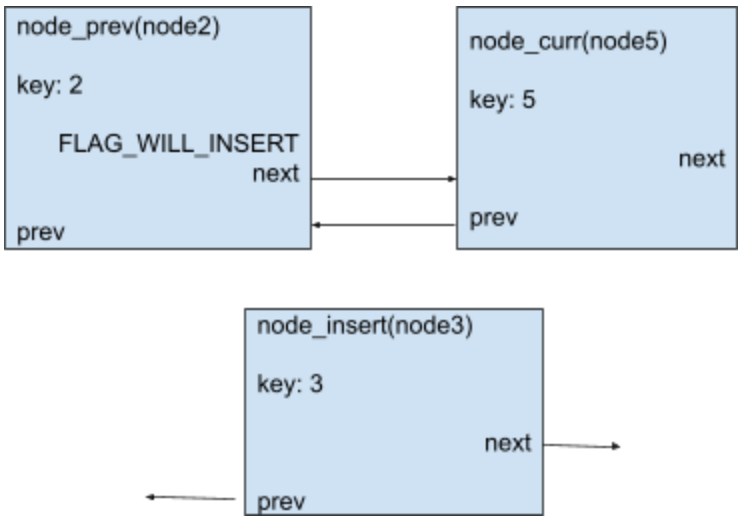
- 2永远指着4（或者5）。NOTE: 引用图(用的4)，我自己画的图(用的5)，都指同一object，2后面的那个node，即curr node，即准备插入之前的那个node
- 在3插入时，绝对不会有第三者同时也想插足
- 2和4(5)永远都不会被删除（至少在插入完成前）

简而言之，我们就是希望固定2和4(5)的位置在整个linked list里不变。这样，其他地方race热火朝天而且不停地变化就没有关系。只要存在这个伊甸园。

FLAG_WILL_INSERT

我们怎么做到这点？我们是通过下面这个标志FLAG_WILL_INSERT做到的

NOTE: 下面这个图，用node5替代了上面的node4，未来都以node5为范本



对于要insert node3的thread，它尝试通过cas去设置FLAG_WILL_INSERT到node2（根据上面的定义，全部都存在next变量里），如果成功，那么我们就保证上面那个伊甸园出现

首先看一下这个cas的definition

cas(设置node2的next = FALG_WILL_INSERT + node2.next
    if node2.next的前16位 == FLAG_FREE && node2.next的后48位 == node5)

也就是说，如果cas失败，意味以下几个可能：

- node2.next的FLAG不是FLAG_FREE，即有其他线程要么准备insert了，要么准备delete了，先抢了这个FLAG茅坑
- node2.next的后48位不是node5，这意味node2到node5的连接已经消失，可能中间已经插入了4，也可能node5已经被delete掉了

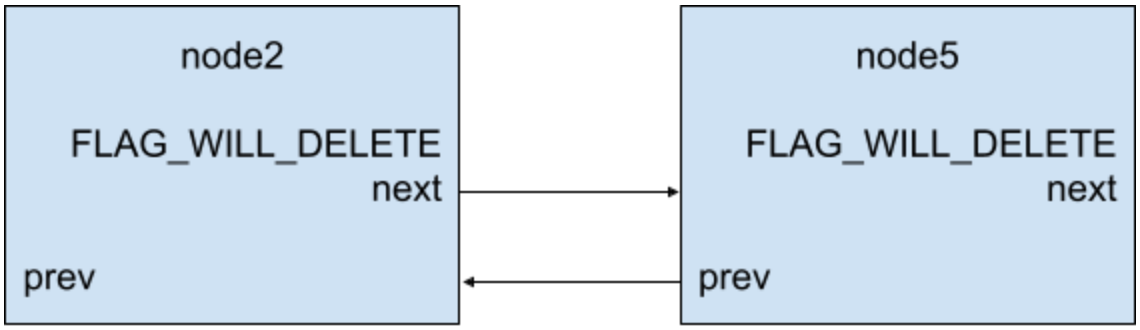
不管如何，如果调用cas失败，我们首先保证了其原来的值没有变（node2.next，里面可能已经存有某些其他线程设置的FLAG值），同时我们也发现了异常，即2到5的伊甸园假设不成立了，所以，很简单，此线程返回，重新find()，retry（因为最终肯定会找到另外一对新的nodes去插入，或者find发现有重复的key被其他线程提前插入，如果是这样，我们可以直接返回。anyway，eventually consistency）

下面我们假设这个insert线程，调用cas设置FLAG_WILL_INSERT成功

为什么伊甸园：delete前提条件

你可能会提一个很深刻（同时也很挑战）的问题，为什么能保证node2或者node5这时不会被删除呢?

很好，我们来看一下这个设定，即：如果一个node要被删除，那个删除线程在删除之前，必须做到下面两个cas



我们准备删除的是node5

一个delete线程，如果它想删除node5，它必须先在它前面的node2，通过cas设置FLAG_WILL_DELETE而且必须成功，然后再通过一个cas设置node5的FLAG_WILL_DELETE而且必须再次成功

cas语法如下：

cas(设置node.next = FLAG_WILL_DELETE + node.next
    if node.next的前16位，FLAG部分 == FLAG_FREE && node.next的后48位 == next_node)

用上面的公式，如果先设置node2，则是：
cas(node2->next = FLAG_WILL_DELETE + node2.next
  if node2.next FLAG部分 == FLAG_FREE && node2.next后48位 == node5)

再设置node5，则是
cas(node5->next = FLAG_WILL_DELETE + node5.next
    if node5.next FLAG部分 == FLAG_FREE && node5.next后48位 == node 6 or node 7 ...)

注意：node5可能是tail，即node 6 or node 7 == nullptr

对于delete线程，如果任何一个设置失败，它都需要把曾经设置成功的FLAG_WILL_DELETE的node还原成FLAG_FREE（这里只可能是node2），然后返回失败信息给上层，然后find() and retry

证明伊甸园

我们来证明一下伊甸园的成立：

如果一个insert线程，它成功通过一个cas在node2上设置了FLAG_WILL_INSERT，意味着：

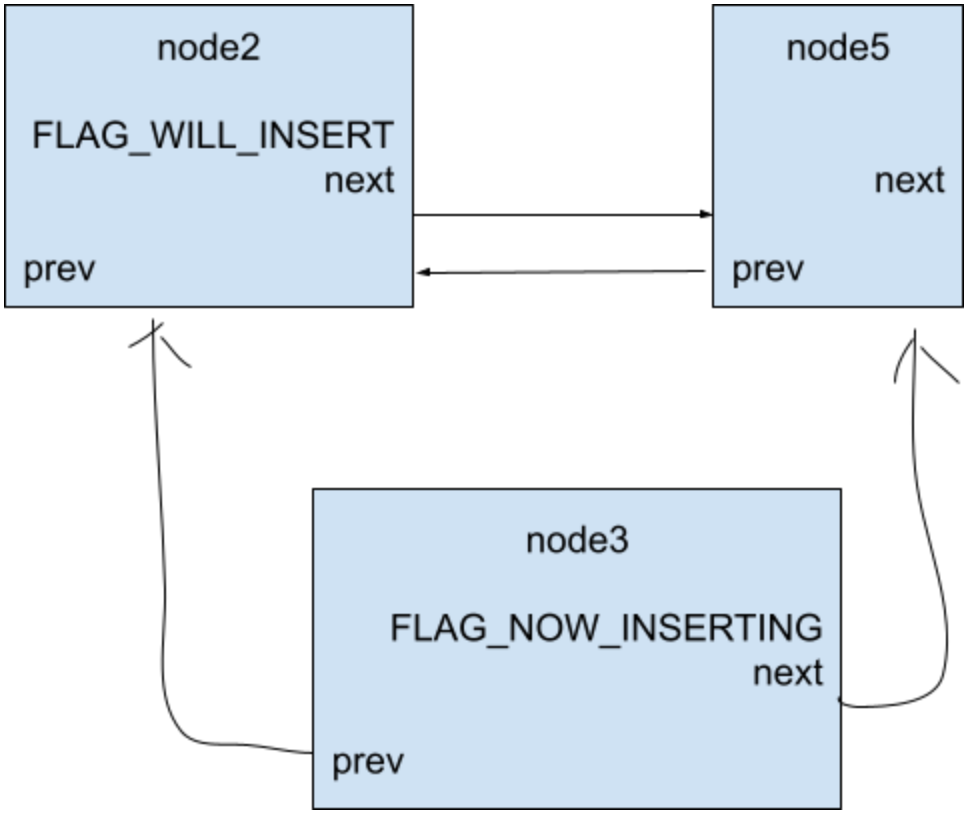
- node2此时此刻肯定指着node5（但未来暂且还没有保证）
- 肯定不会有其他线程先于我在node2和node5之间插入，因为其他线程必须先在node2上抢到FLAG_WILL_INSERT
- node5肯定不会被删除，因为要删除node5，必须在node2上设置成功FLAG_WILL_DELETE
- node2肯定不会被删除，因为要删除node2，那个线程必须设置node2前面那个node的FLAG_WILL_DELETE，同时也必须设置node2的FLAG_WILL_DELETE
- 所以，我现在肯定无疑地确认，只要我这个insert线程不做完事，只要被我这个insert线程设置的node2上的FLAG_WILL_INSERT存在，node2永远地指着node5（或即将新增的本insert线程所拥有的node，i.e., node3），node2永远不会被删除同时node5也永远不会删除（只要node2还指着node5）。所以第一条里面”但未来暂且还没有保证“可以拿掉了。

所以证毕：伊甸园存在

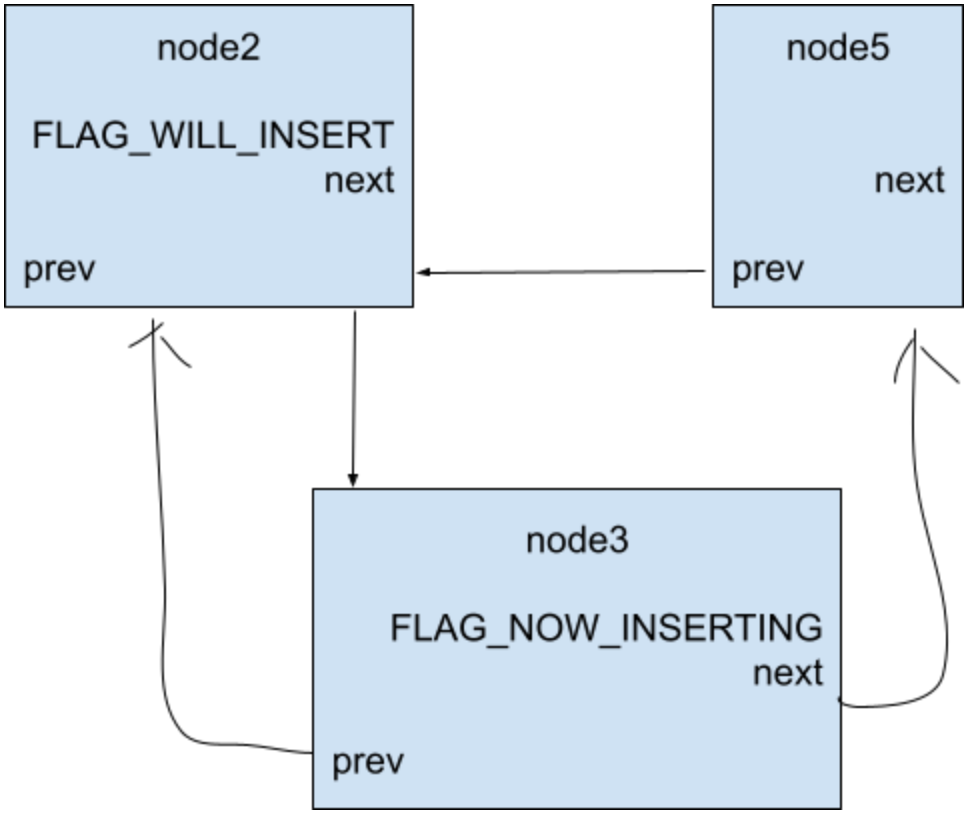
Insert的后续步骤

伊甸园的存在（由于设置成功了FLAG_WILL_INSERT），可以让我们的insert线程放心而大胆地做双link的修改，只要最后退出时，将FLAG_WILL_INSERT设置回FLAG_FREE即可

下面是图示：



你一定很奇怪怎么又冒出一个FLAG_NOW_INSERTING，先暂且不表，下图就会用到，你只需要知道，在node3.next里，放入了FLAG_NOW_INSERTING



这时，nod2.next的后48位指向node3，即我们link了node2 and node3（部分link）。

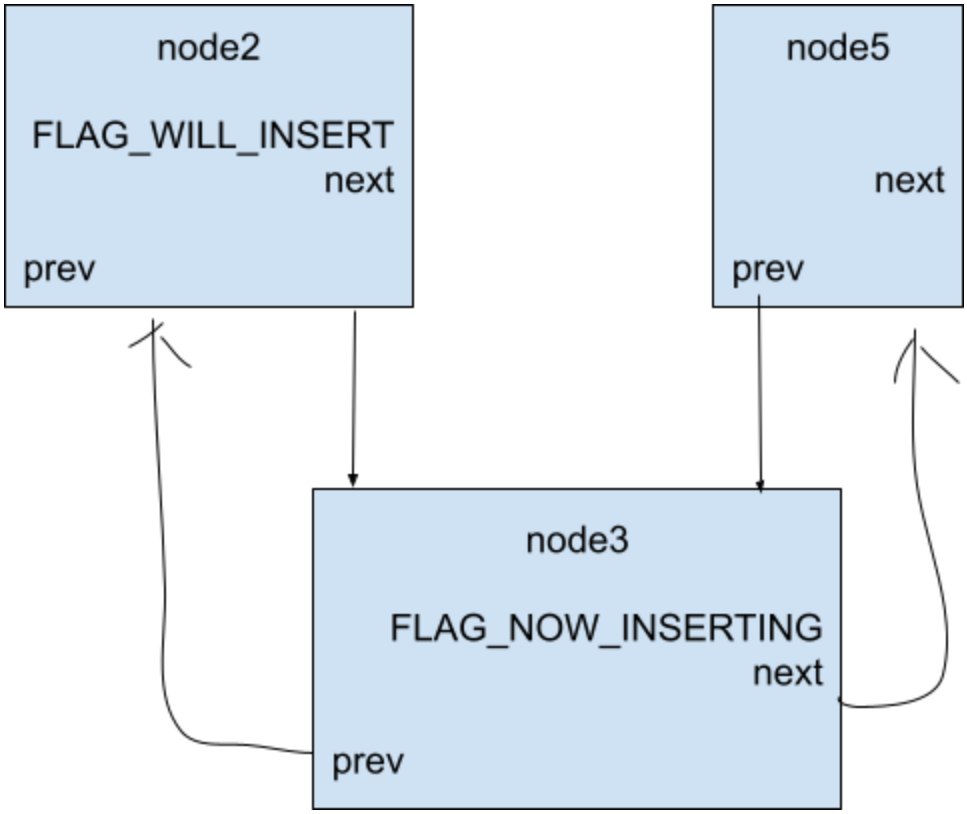
但大家发现一个小风险：如果这时，从左边iterator，我们可以看到node3，但如果从右边iterator，我们看不到node3。

一般情况下，这种不一致问题不大，因为很快node3将加入到整个list，那时就eventually consistency。

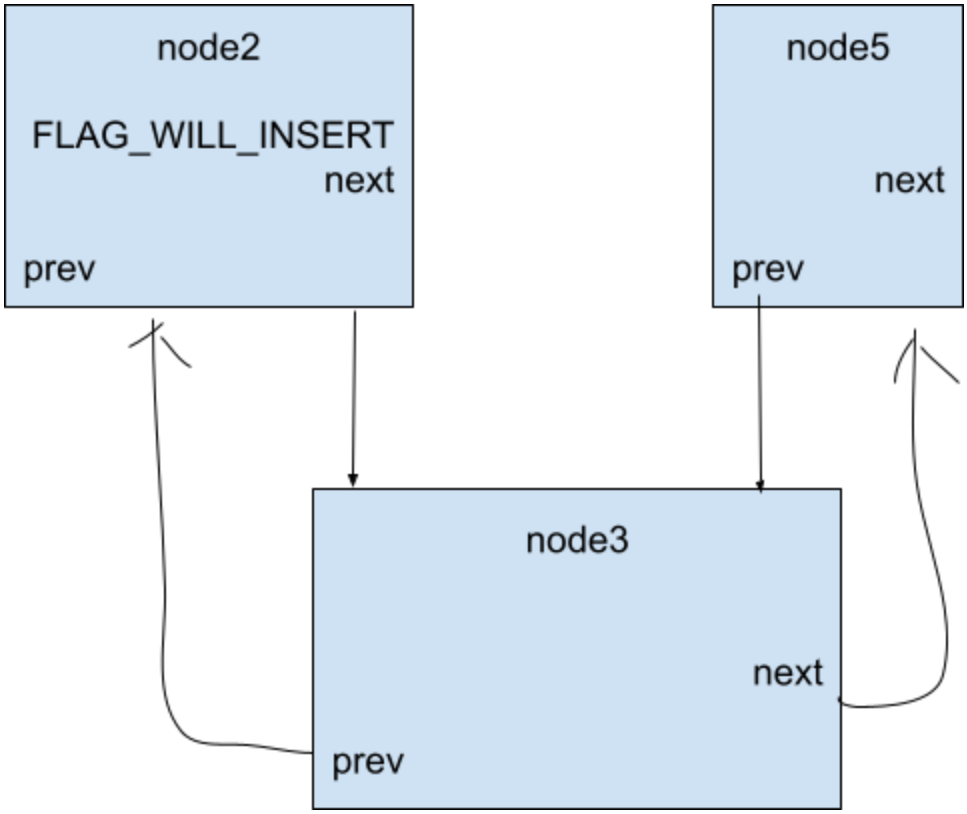
但苛求的我们还是希望instant consistency，所以我们设置了FLAG_NOW_INSERTING。

当iterator时（比如：上面那个find方法），我们遇到某个node的flag是FLAG_NOW_INSERTING，我们当它不存在，继续向左或向右前进一步

这样，就实现了instant consistency

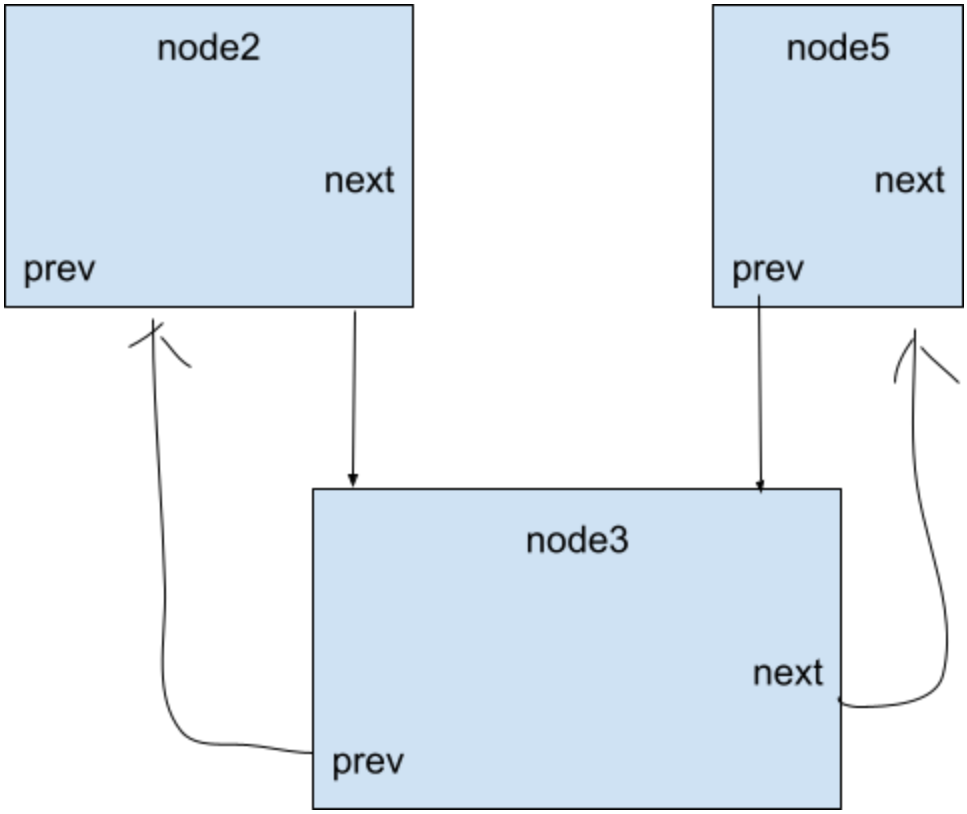


这时，node3全link了，从法理上讲它已经在完整的list里了，但由于FLAG_NOW_INSERTING的存在，它被视而不见。但没有关系，这个对node3不友好的时刻非常短暂，且看下图



insert线程清理了node3的FLAG_NOW_INSERTING，这时node3就正式公之于众。node5这时可以被删除，node3和node5之间也可以被其他线程insert了（但node3此时不可能被删除）。

这个insert线程继续：

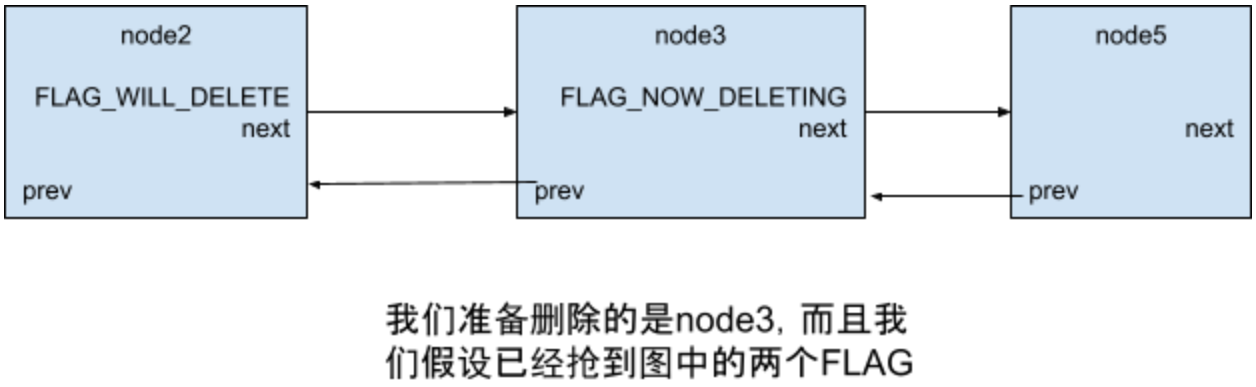


ok，一切都归于平静，insert线程完成了任务，可以成功返回了

Delete过程

参考上面的delete线程的前提条件，假设delete线程准备删除node3，而且抢到了这两个FLAG

同时，我们做个小的修正，再抢第二个FLAG时，即那个真正需要删除的node里的next变量，如果成功，我们设置为FLAG_NOW_DELETING，这个不影响上面的逻辑，稍微修改一下代码即可。

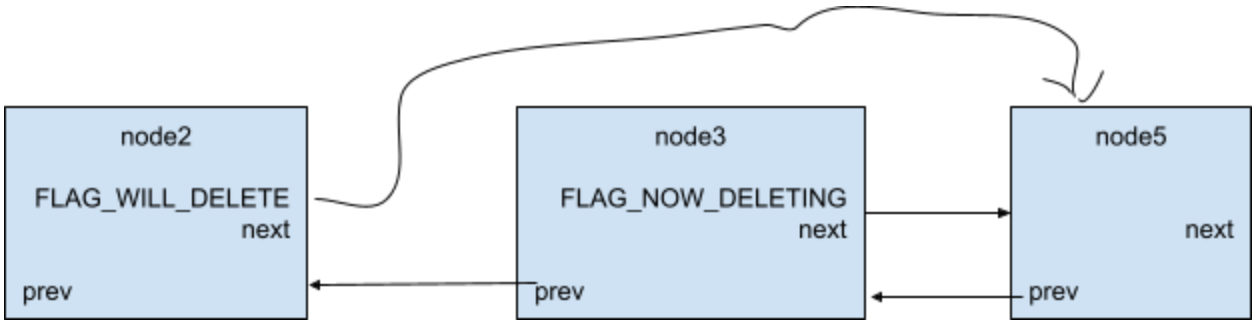


这也是一个伊甸园。因为：

- node2不可能被删除
- node3不可能被删除
- node5不可能被删除
- node2和node3之间不可能被插入（我们假设有node2.5的可能）
- node3和node5之间不可能被插入

用FLAG_NOW_DELETING的原因

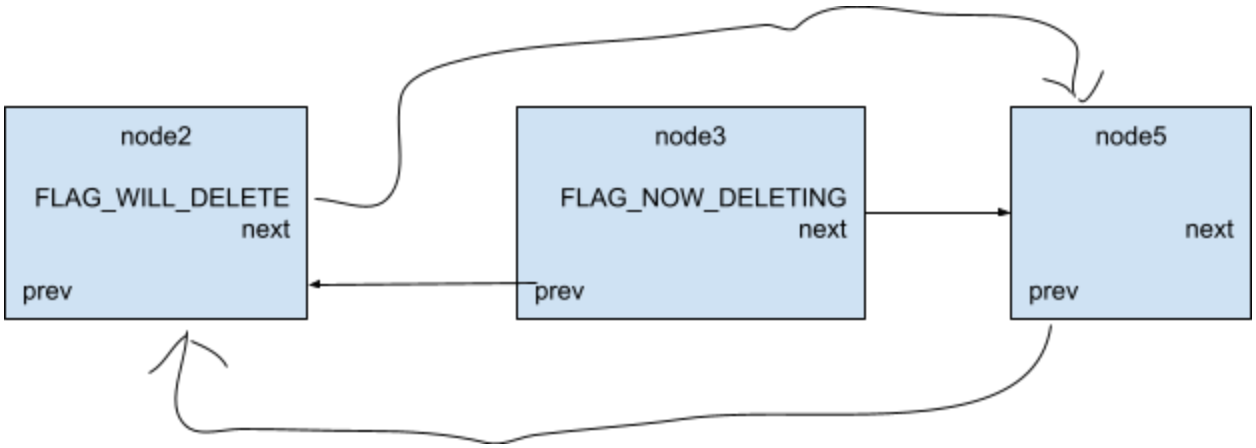
请看下图（画得真丑）



我们修改node2的指针的后48位，使其指向node5，这时，又存在左右iterator不一致问题，所以，我们用FLAG_NOW_DELETING来达到通用的目的（为了instant consistency）

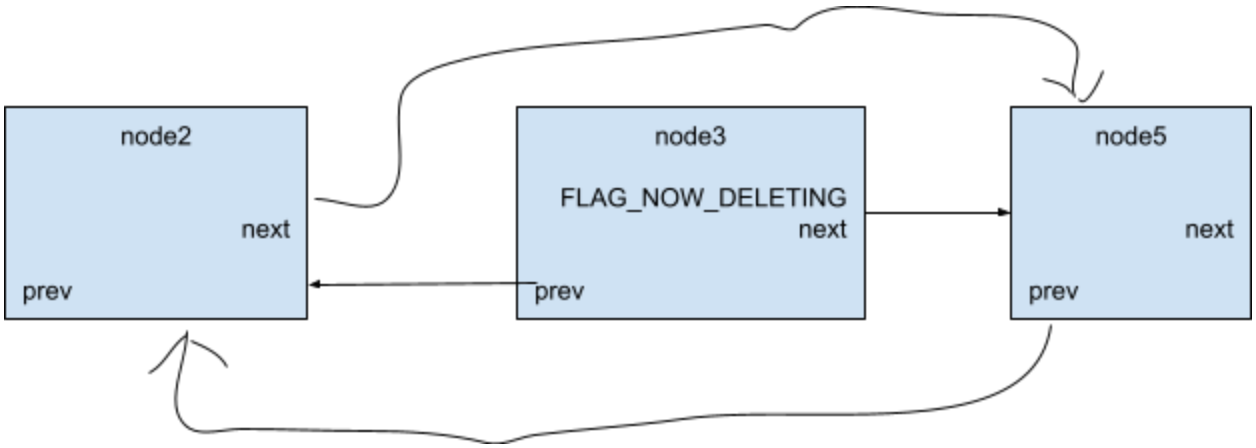
你可能说：上上面那个图，整个链表不是很完整的吗，如果node3被标志为FLAG_NOW_DELETEING，那么不就意味node3对其他线程已经不可见了？

是的，但这没有问题，因为马上就有下面这个图的操作，非常快



我们再改node5的prev，这样，就将node3从整个list删除掉了

最后，恢复正常的状态如下（代表delete线程成功完成任务）：



这时，可以对node3做GC。但对于C++而言，做GC要小心，因为node3可能还有其他线程在访问，所以并不能随意地delete。至于C++如何GC，非常麻烦，可以参考我的一篇文章。但如果你使用Java或Golang，则没有任何GC负担。

Lock Free的理论探讨

如果我们用程序实现上面的代码，我们很容易写出这样的代码

```
while (true) {
        find_res = find();
        if find_res 不对，return fail;

        bool success = try_set_flag();
        if success:
                break;
}
// do insert or delete which guarantees to be successful
```

我看的所有和lock free相关的书籍，以及我自己的代码，基本都是这个代码逻辑

但这实际上是违反lock free理论的。

假设：只有两个线程，一个线程执行上面的代码，另外一个线程提前设置了flag，但没有做完后面的insert or delete，然后就被OS永久休眠了。这种情况下，sucess永远是false。于是：死循环

Lock Free的理论是，假设一个线程永久休眠，那么其他线程必须能做到move progress。

但如果按这个定义，我们就会发现，这两个线程，假设一个永久休眠，另外一个肯定无法move progress（陷死在那个死循环里）。所以，这是违背Lock Free理论的。

但实际上，从工程意义角度，没有必要去改变代码：因为：

1. 假设那个休眠线程被永久冻结，会影响和它冲突的某个线程，但其他线程，只要不发生race，就不会有任何问题。即其他线程会move progress

2. 现代操作系统里，一个线程不会被长久的冷冻

如果你真要较真，还有一个方法可以解决，就是加time_out，在上面那个循环里，定时获得时间（注意：最好用一个单独的线程产生全局的时间tick tock(++milliseconds)，而不要用操作系统的返回时间，因为操作系统可能做page in/out或者本程序做GC，这个可能需要大量的时间，从而导致操作系统时间对于time_out而言并不准确），如果时间过长，当成time out错误对待，返回给caller。

但这不是真正意义或者工程角度的的move progress，而是return fault（相当于那个永久休眠的线程将某个node资源也永久锁死，从而导致其他需要这个node的线程肯定失败）。但我是不会这样做的，太复杂了。