第三讲　并发对象

一个对象封装了一个数据结构，提供一组方法用以对该数据结构进行操作。在顺序程序中，一个对象在任何时刻都具有一个确定的状态，即它所涉及的所有变量的取值。调用方法可以改变对象的状态。每个方法功能通常用如下方法描述：如果调用该方法前对象处于如此这般的状态，那么调用结束时该方法会返回怎样的值，并且对象将处于什么样的状态。完全不用考虑方法执行过程中对象状态是如何一步一步改变的。

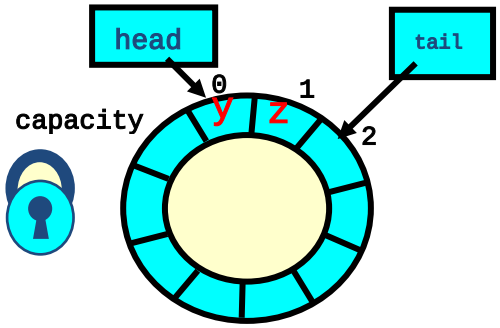
在并发系统中，在同一时刻可以有多个线程都在执行一个对象的方法，并且它们执行的相对速度是不确定的。这导致对象的状态不确定。在一个线程执行某个方法过程中的每一步，都可能受到其它线程的干扰。因此不能不考虑方法执行过程中对象状态的改变情况。

一、先进先出队列

考虑并发的先进先出队列。队列对象的元素存放在一个数组中。提供两个方法 enq( )（入队）和 deq()（出队）。

基于锁的队列代码如下：

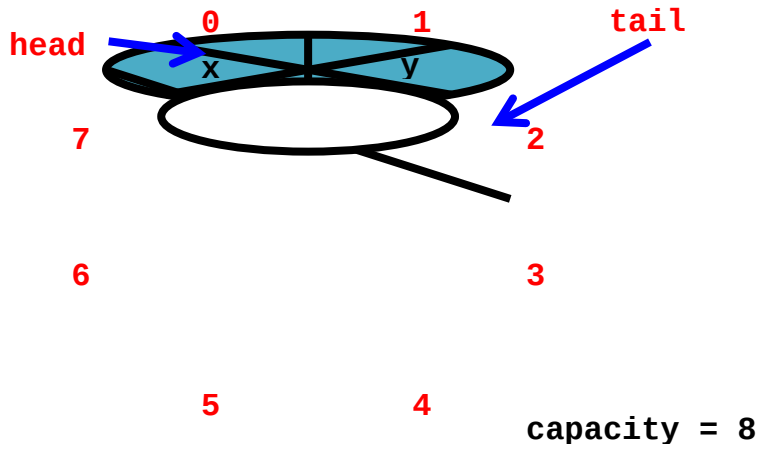
```
class LockBasedQueue<T> {
   int head, tail;
   T[] items;
   Lock lock;   //用锁实现互斥
   public LockBasedQueue(int capacity) {
      head = 0; tail = 0;
      lock = new ReentrantLock();
      items = (T[]) new Object[capacity];
}

public T deq() throws EmptyException {
   lock.lock();
   try {
      if (tail == head)
          throw new EmptyException();
      T x = items[head % items.length];
      head++;
      return x;
   } finally {
      lock.unlock();
   }
}
```



由于方法体的执行完全互斥，这个“并发”队列的实际行为与顺序队列无异，可以保证“先进先出”的性质，其正确性易见。但是完全互斥排除了并发性，在多核系统上，这意味着任何时刻只能有一个核在执行有用的操作，其它核都在“忙等待”。按 Amdahl’s Law，加速比是 0！

二、无等待队列（Wait-free Queue）

现在考虑完成不用锁的并发队列。为了简单起见，假定一共只有两个线程使用这个并发队列，并且一个线程只调用 enq，另一个线程只调用 deq。



代码：

```
public class WaitFreeQueue {
   int head = 0, tail = 0;
   items = (T[]) new Object[capacity];

   public void enq(Item x) {
      if (tail-head == capacity) throw
            new FullException();
      items[tail % capacity] = x; tail++;
   }
   public Item deq() {
       if (tail == head) throw
            new EmptyException();
       Item item = items[head % capacity]; head++;
       return item;
}}
```

由于没有用锁，两个线程对 **enq** 方法和 **deq** 方法的并发调用在时间上可以任意交叠，无法说清谁先谁后。在这种情况下，怎么能证明这个队列的实现保证了“先

进先出”？

我们的思路是：寻找一种方法，将并发执行的历史归结到顺序执行历史，然后论证所得到的顺序执行历史满足所要求的性质（如先进先出）。

三、顺序对象

一个顺序对象由若干变量和一组方法组成。对象的状态就是其变量的取值。改变对象状态的唯一途径是调用它的方法。

对顺序对象可以用不同的方式的方法来规约。比如可以对其方法用前置条件-后置条件来规约。

If (前置条件)
　在调用方法之前，对象处于如此这般的状态
Then (后置条件)
　方法将返回某个值，或者抛出一个异常
and (后置条件续)
　当方法返回时,对象将处于某个状态

例如对 deq 方法：

前置条件：队列非空
后置条件：返回队列中的第一个元素，且队头指针加 1

前置条件：队列为空
后置条件：抛出异常
后置条件：队列状态不变

对顺序对象，不同的方法调用之间在时间上没有交叠，对方法的规约只需要考虑方法执行前和执行后的状态，完全不用考虑方法执行过程中对象状态是如何一步一步改变的。

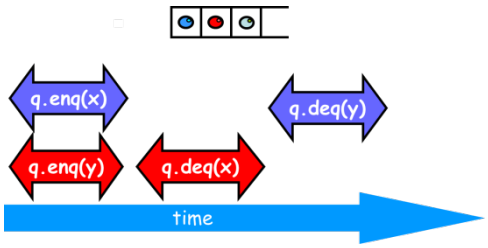
对并发对象，不同的方法调用在时间上可以交叠，相互间以不可预测的方式进行交互，互相干扰。整个系统的状态空间关于线程的数目呈指数增长。对于队列对象，要求先进先出，这是严格的时序关系；而办法意味着模糊的时序。因此，要说明并发的队列对象满足先进先出，不是显而易见的事情。

四、可线性化（linearizability）

直观地说，如果一个并发对象的每个方法的每次调用对系统状态改变的效果都可以看作是在该次调用的开始和结束之间的某个时间点发生的，这个并发对象就是可线性化的。这个时间点称为该次调用的可线性化点。

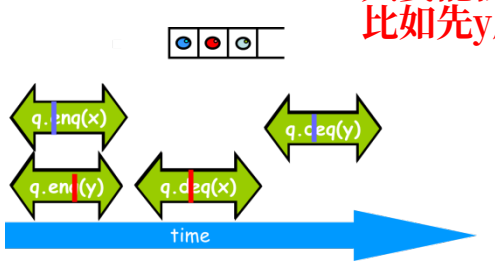
不同的方法调用可以在时间上交叠，但它们的可线性化点不可能交叠。如果一个并发对象是可线性化的，它的每次并发执行中的所有方法调用就可以按照它们的可线性化点排成线性序，该并发执行就等效于其中的方法调用按照这个线性序的顺序执行。如果所有这样得到的顺序执行都是正确的，这个并发对象就是正确的。这样，可线性化就将并发对象的正确性归结为顺序执行的正确性。

例 1：假设有两个线程 A、B。A 执行 enq(x) deq (y)，B 执行 enq (y) deq (x)。执行过程如下图所示。

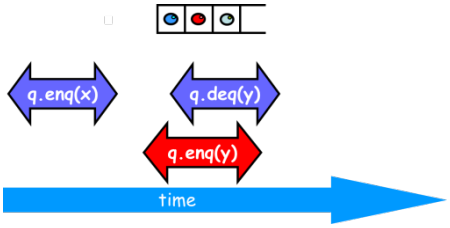


这个并发执行是可线性化的：

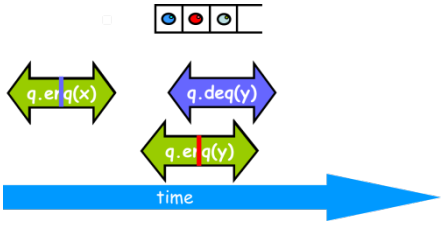
只要能找出2点来就行？
比如先y后x这个就不行



例 2：假设有两个线程 A、B。A 执行 enq(x) deq (y)，B 执行 enq (y)。执行过程如下图所示。

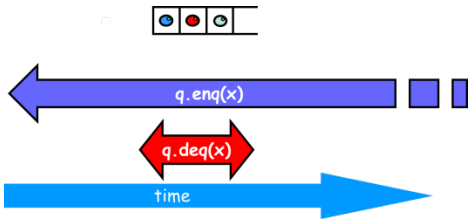


由于 enq(x) ➔ enq (y)，这个并发执行不是可线性化的：

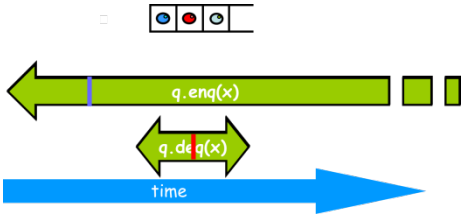


例 3：假设有两个线程 A、B。A 执行 enq(x) ，B 执行 deq (x)。执行过程如下图所示。

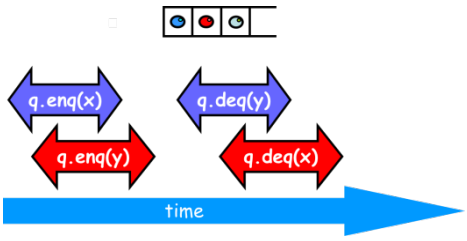
这里这个箭头是啥意思？？



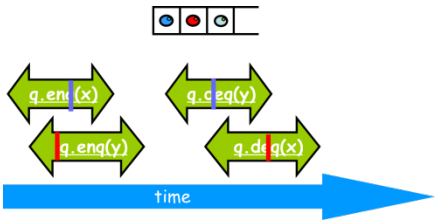
这个并发执行是可线性化的：



例 4：假设有两个线程 A、B。A 执行 enq(x) deq (y)，B 执行 enq (y) deq (x)。执行过程如下图所示。

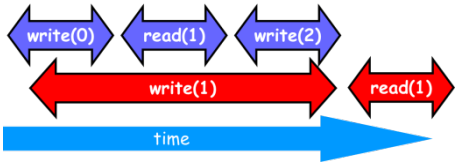


这个并发执行是可线性化的：

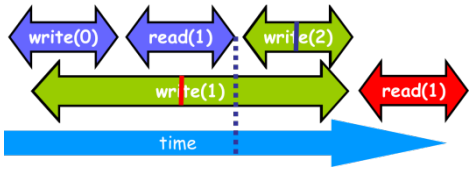


这里也说明了只要能找到2个可线性化点就行

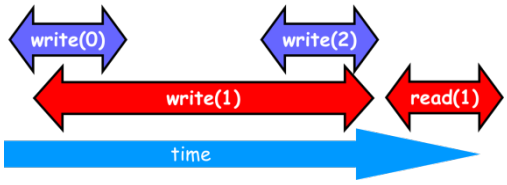
例 5：假设有两个线程 A、B。A 执行读写，B 执行读写。执行过程如下图所示。



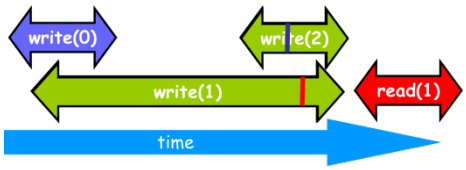
因 **write(1)➔read(1)➔ write(2)➔read(1)** ，这个并发执行不是可线性化的：



例 6：假设有两个线程 A、B。A 执行读写，B 执行读写。执行过程如下图所示。



这个并发执行是可线性化的：



前面可线性化的非形式定义中，可线性化点是相对于并发对象的一次执行中的一次方法调用定义的。在很多情况下，一个方法的所有调用的可线性化点都对应于该方法代码中的同一个位置。这种可线性化点称为固定可线性化点。但是也有不少并发对象同一个执行中，同一个方法的不同调用具有不同的可线性化点，有的甚至对应于其它方法代码中的位置。

五、形式化定义

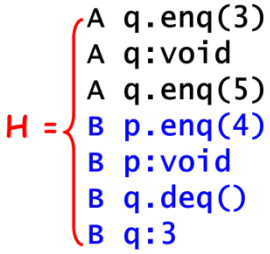
并发系统的执行可以用*历史(history)*来建模。一个历史是由方法的启用（invocation）和回应（response）两类事件组成的有穷序列。一个历史的*子历史*是它的一个子序列。

1. 方法调用的记号

（1）启用（Invocation）：方法名和参数
如 q.enq(x)
启用的记号（Invocation Notation） **thread object. method (arguments)**
如 A q.enq(x)
（2）回应（Response）：返回结果或抛出异常，方法名可以省略。
如：q.enq(x) returns void
q.deq() returns y
q.deq() throws EmptyException

响应的记号（Response Notation）**thread object：result**
如 A q: void （viod 表示没有返回值）

*历史*是由启用事件和回应事件构成的序列：

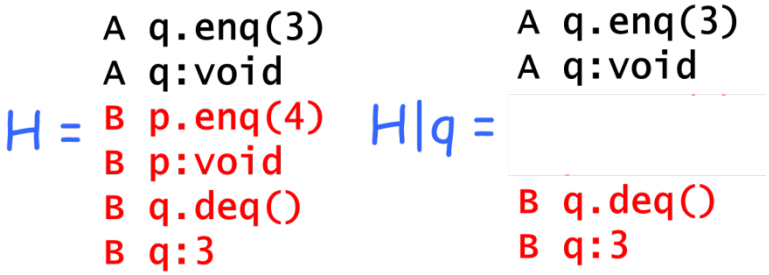


2. 定义

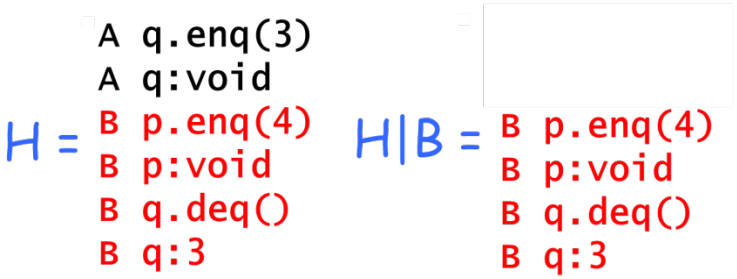
（1）匹配：线程名一样、对象名也一样的启用和回应称为是匹配的。

（2）顺序历史：如果历史 H 的第一个事件是启用，并且除最后一个之外的所有启用事件都随后紧接着与之匹配的回应，则称 H 是顺序历史。顺序历史中不同的方法调用在时间上不交叠。

（2）对象投影（Object Projections）：关于某一对象的子历史



（3）线程投影（Thread Projections）：关于某一线程的子历史



（4）历史的延拓（History extension）

历史中一个启动，如果没有与之匹配的回应，则称为是待结的（pending）。一个启动待结的方法调用，可能已经产生效果，也可能还没有产生效果。

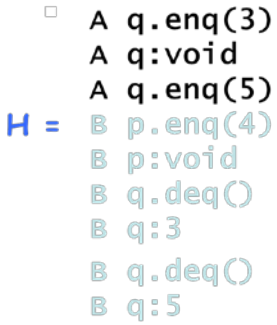
如果一个历史中没有待结的启动，就称它是完整的。

历史 H 的*延拓*是通过在 H 的末尾添加若干个（可以是零个）与 H 中（已经产生效果的）待结启动相匹配的回应而得到的历史。

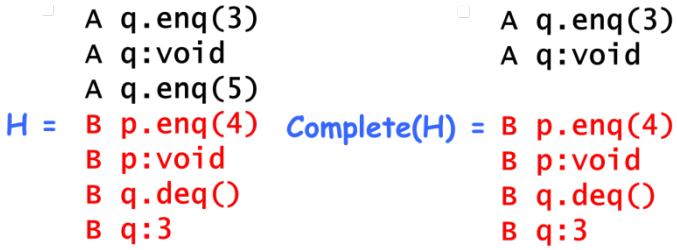
complete(H)是通过删去 H 中的（还没有产生效果的）待结启动得到的完整子历史。

下图中，A q.enq(5) 没有相匹配的回应，因此它不是一个完整的执行：

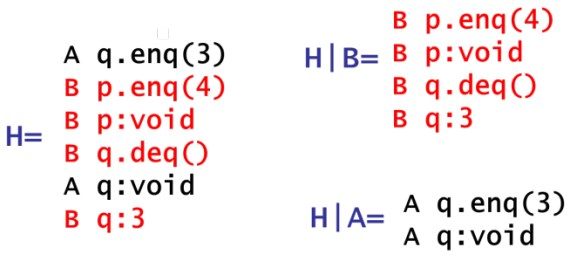
。



可以删去尚未产生效果的待结启动，得到一个完整的子历史：

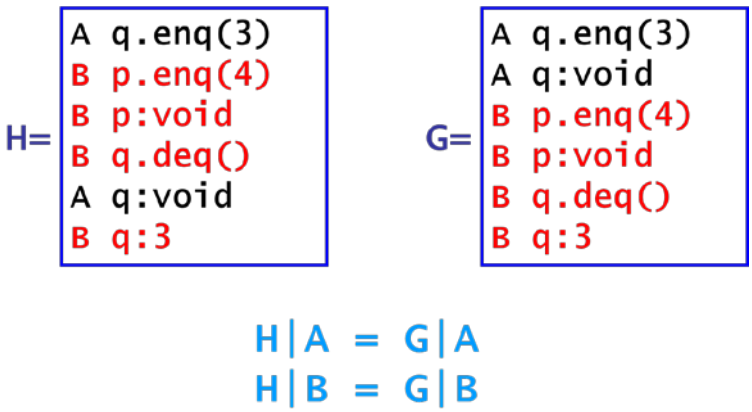


（5）良形历史（Well-Formed Histories）：在每一个线程上的投影都是顺序的历史。如：



以后只考虑良形的历史。

（6）等价历史（Equivalent Histories）：在所有线程上的投影都相同的良形历史称为是等价的。



（7）顺序规约（Sequential Specifications）

描述单线程、单对象的执行历史是否正确的、合法。
例如用前置条件和后置条件对每个方法分别加以规约，又如用一组代数公理对对象的整体行为加以规约。

（8）合法历史（Legal Histories）

一个顺序历史 H 是合法的，如果对于每个对象 x，H | x 满足对象的顺序规约。
例如：一个历史分别投影到栈和队列上，且分别满足先进后出和先进先出，那么这个历史就是合法的。目的是将并发行为归结为顺序行为来理解。

（9）先于关系（Precedence）
如果历史 H 中一个方法调用 p 的回应事件发生在另一个方法调用 q 的启动事件之前，就说 p *在 H 中先于* q，记作 p $\rightarrow_H$ q。
$\rightarrow_H$ 是偏序关系。如果 H 是顺序历史，那么 $\rightarrow_H$ 是全序关系。

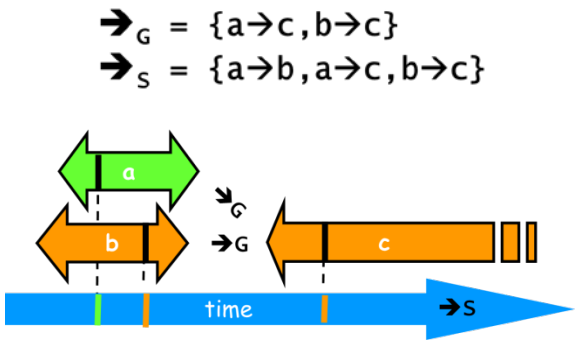
如果两个方法调用在时间上交叠，它们之间就没有先于关系。

（10）可线性化（Linearizability）

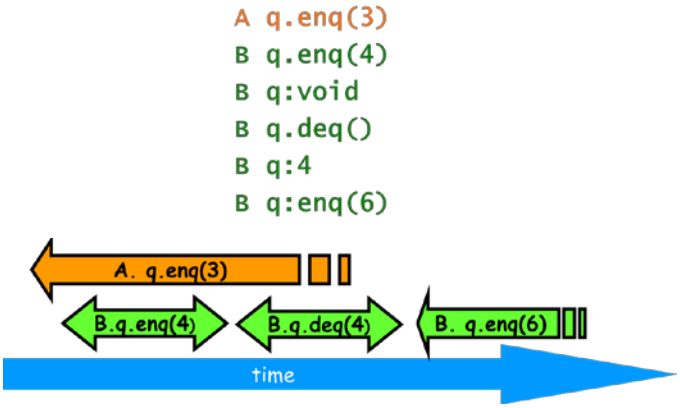
定义 历史 H 是可线性化的，如果存在 H 的一个延拓 G 和一个合法的顺序历史 S，满足：
（1）complete（G）与 S 等价，且
（2）$\rightarrow_G \subset \rightarrow_S$

这个定义的第一部分是说：保留 H 中已经产生效果的待结启动并在 H 的末尾添加与其相匹配的回应，得到 G，删去 H 中尚未产生效果的待结启动，得到 complete（G）。第二部分是说：complete（G）与顺序 S 等价，并且 S 中的先于关系 $\rightarrow_S$ “尊重”G 中的先于关系 $\rightarrow_G$。（p $\rightarrow_G$ q 蕴含 p $\rightarrow_S$ q）。

例： $\rightarrow_G \subset \rightarrow_S$

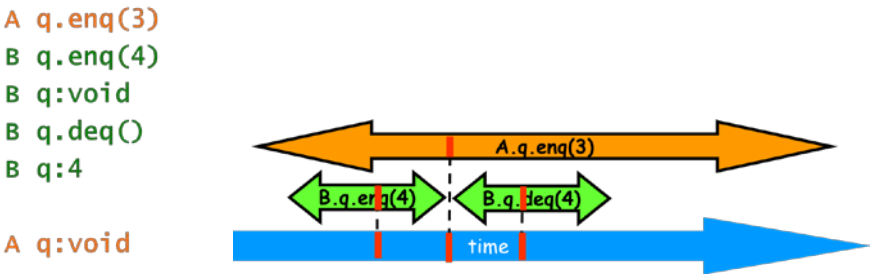


可线性化例：

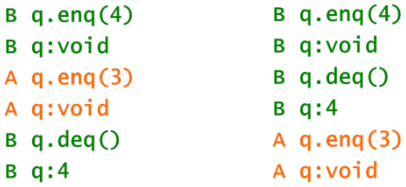


方案 1：删去 **A q.enq(3)　B q:enq(6)**

方案 2：完整 **A q.enq(3)**　删去 **B q:enq(6)**



所得到的历史与下面两个顺序历史都等价：



六、可线性化与并发性

并发系统中多个线程对同时共享对象进行操作，希望尽可能提高并发度。可

线性化性是否会在有些情况下要求方法调用阻塞（等待）？答案是：不会。可线性化性是*非阻塞*的。

非阻塞定理（Non-Blocking Theorem）如果启动 **A q.inv(…)** 在历史 H 中待结，那么一定存在一个回应 **A q:res(…)** 使得 H • <**A q:res(…)**> 是可线性化的。

证明：令 S 是 H 的线性化。如果 S 包含了 **A q.inv(…)**及其回应 **A q:res(…)**，那么 S 也是 H • <**A q:res(…)**> 的线性化，定理得证。否则，由于线性化历史中不含待结的启动，**A q.inv(…)** 不在 S 中出现。由于对象是完全的，我们总可以选择一个回应 A q.res(⋯) 使得 **S'** = S • <A q.**inv** (⋯) > • < A q.**res**(⋯)> 是合法的。而 **S'** 是 H • <A q.res(⋯)> 的一个线性化，因此也是 H 的一个线性化的。

这个定理说明可线性化本身不会要求一个含有（完全方法）待解启动的线程发生阻塞。这不排除可线性化的某些实现中出现阻塞（甚至死锁），但这不是可线性化自身固有的问题。

可组合性定理（Compositionality Theorem）历史 H 是可线性化的当且仅当对于每个对象 x H|x 是可线性化的。

证明 : (=>) 方向。易证。

(<=) 方向。对任意对象 x，任取一个 H|x 的线性化。令 R_x 是在构造这个线性化时添加在 H|x 末尾的回应的集合，$\rightarrow_x$ 是相应的线性化序。令 H'为将 R_x 中的回应添加在 H 的末尾得到的历史。

对 H'中所含的方法调用的个数施归纳。

归纳基始：k=1 时，结论显然成立。

归纳步：假定结论对含不到 k 个方法调用（k>1）的所有历史都成立，考虑含 k 个方法调用的历史 H。

对于任意对象 x，考虑 H'|x 的最后一个调用。在这些调用中一定存在一个关于$\rightarrow_H$ 极大的调用 m，即不存在 m'满足 m$\rightarrow_H$ m'。令 G'是从 H'中删去 m 得到的历史。因为 m 是极大的，H'等价于 G' •m。由归纳假设，G'可以线性化为一个顺序历史 S'。于是 H'和 H 都可以线性化为 S'•m。

由于可线性化性质是可组合的，在证明一个系统可线性化时，我们可以对每个对象分别加以证明，这给证明带来了极大的便利。另外，我们也可以将独立实现的可线性化对象组合起来，得到的系统也是可线性化的。

证明可线性化性的策略：试图确定方法调用的可线性化点，即方法调用产生效果的原子步骤。通常可以考虑代码中离开临界区的点。

例如对使用锁实现互斥的队列，通常释放锁的时间点可以看做是可线性化点。

对无锁队列（LockFreeQueue），可线性化点是修改队头指针或队尾指针的地方。

但在事情并不总是这么简单。有些情况下，在同一个执行中，同一个方法的

不同调用中产生效果的点不一样，有的调用中产生效果的点可能落在其它方法的代码中。对这样的方法需要确定多个可线性化点。

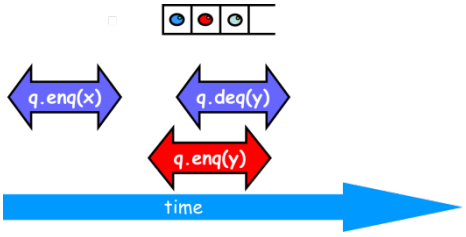
七、可线性化与顺序一致性

定义：历史 H 是顺序一致的，如果存在 H 的一个延拓 G 和一个合法的顺序历史 S，使得 complete（G）与 S 等价。

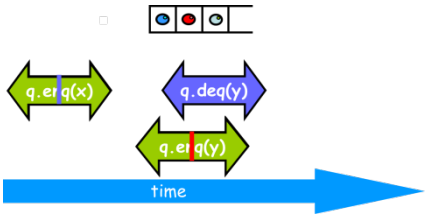
同一线程对方法进行调用的次序称为程序次序（程序次序对不同线程调用方法的次序不做要求）。顺序一致性要求方法调用产生效果的顺序与程序次序一致。

顺序一致性与可线性化性的不同在于，不要求$\rightarrow_G \subset \rightarrow_S$。顺序一致性不要求不同线程的方法调用保持它们的实时次序。

例：假设有两个线程 A、B。A 执行 enq(x) deq (y)，B 执行 enq (y)。执行过程如下图所示：

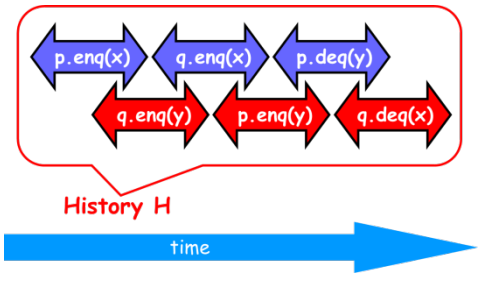


这个历史是不可线性化的，但是满足顺序一致性。



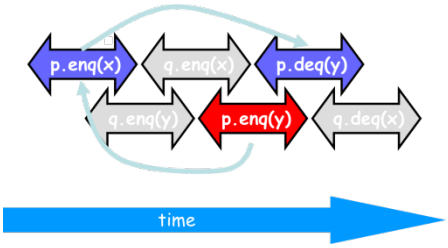
定理：顺序一致性不是可组合的。

证明：以先进先出队列为例，给出如下反例。

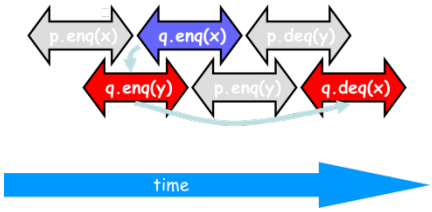


易见 H|p 和 H|q 分别都是顺序一致性的。

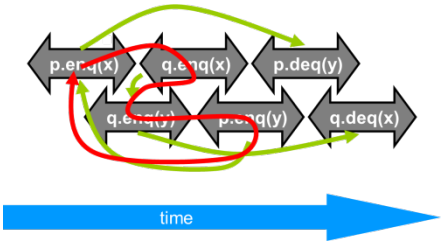
p 要求的序：



q 要求的序：



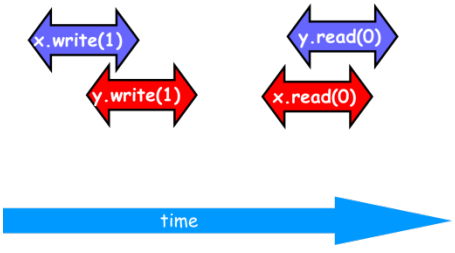
合并得到：



而偏序不可能出现环。矛盾。

顺序一致性适用于不需要组合的系统，如内存；可线性化性适用于由多个组件复合而成的系统，如软件。

大多数硬件体系结构都不支持顺序一致性。但这可能出现困难。如两个线程读写 flag：



从每个线程看是顺序一致的，因为它可以比另一线程先做完。但整个历史不是顺序一致的，因为不可能两个线程都先做完。

像 flag 这样，写自己的，读别人的，是实现互斥的核心手段。如果硬件不支持，就难以编写并发程序。

但是绝大多数的读写操作并不用于同步。如果都要求顺序一致性，则成本太高。实际内存模型一般都比顺序一致性弱，但允许在需要的时候显式地要求顺序

一致性（在性能上付出代价）。比如 Java 中的 volatile。

。

八、进展性

无死锁：某个申请锁的线程最终会获得锁。
无饥饿：每个申请锁的线程最终都会获得锁。
无锁：某个调用一个方法的线程最终会返回。
无等待：调用一个方法的线程最终都会返回。