

## 二

# 54 理解Disruptor（上）：带你体会CPU高速缓存的风驰电掣

坚持到底就是胜利，终于我们一起来到了专栏的最后一个主题。让我一起带你来看一看，CPU 到底能有多快。在接下来的两讲里，我会带你一起来看一个开源项目 Disruptor。看看我们怎么利用 CPU 和高速缓存的硬件特性，来设计一个对于性能有极限追求的系统。

不知道你还记不记得，在[第 37 讲](#)里，为了优化 4 毫秒专门铺设光纤的故事。实际上，最在意极限性能的并不是互联网公司，而是高频交易公司。我们今天讲解的 Disruptor 就是由一家专门做高频交易的公司 LMAX 开源出来的。

有意思的是，Disruptor 的开发语言，并不是很多人心目中最容易做到性能极限的 C/C++，而是性能受限于 JVM 的 Java。这到底是怎么一回事呢？那通过这一讲，你就能体会到，其实只要通晓硬件层面的原理，即使是像 Java 这样的高级语言，也能够把 CPU 的性能发挥到极限。

## Padding Cache Line，体验高速缓存的威力

我们先来看看 Disruptor 里面一段神奇的代码。这段代码里，Disruptor 在 RingBufferPad 这个类里面定义了 p1, p2 一直到 p7 这样 7 个 long 类型的变量。

```
abstract class RingBufferPad
{
    protected long p1, p2, p3, p4, p5, p6, p7;
}
```

我在看到这段代码的第一反应是，变量名取得不规范，p1-p7 这样的变量名没有明确的意义啊。不过，当我深入了解了 Disruptor 的设计和源代码，才发现这些变量名取得恰如其分。因为这些变量就是没有实际意义，只是帮助我们进行**缓存行填充**（Padding Cache Line），使得我们能够尽可能地用上 CPU 高速缓存（CPU Cache）。那么缓存行填充这个黑科技到底是什么样的呢？我们接着往下看。

不知道你还记不记得，我们在[35 讲](#)里面的这个表格。如果访问内置在 CPU 里的 L1 Cache 或者 L2 Cache，访问延时是内存的 1/15 乃至 1/100。而内存的访问速度，其实是远远慢于

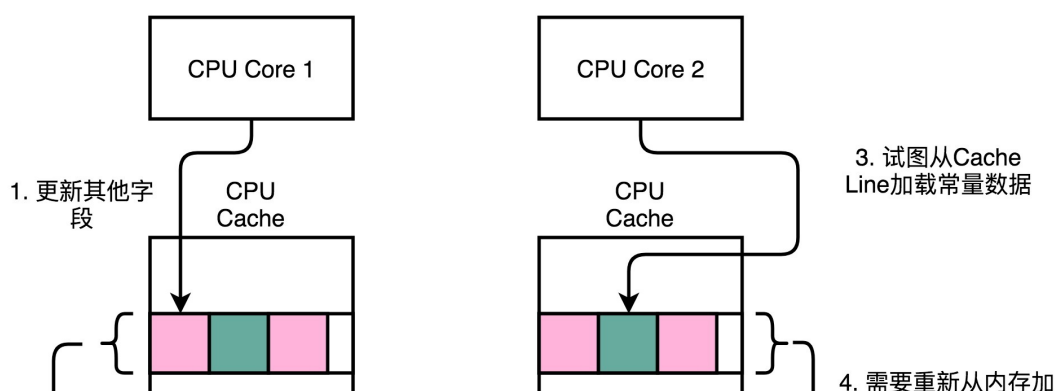
CPU 的。想要追求极限性能，需要我们尽可能地多从 CPU Cache 里面拿数据，而不是从内存里面拿数据。

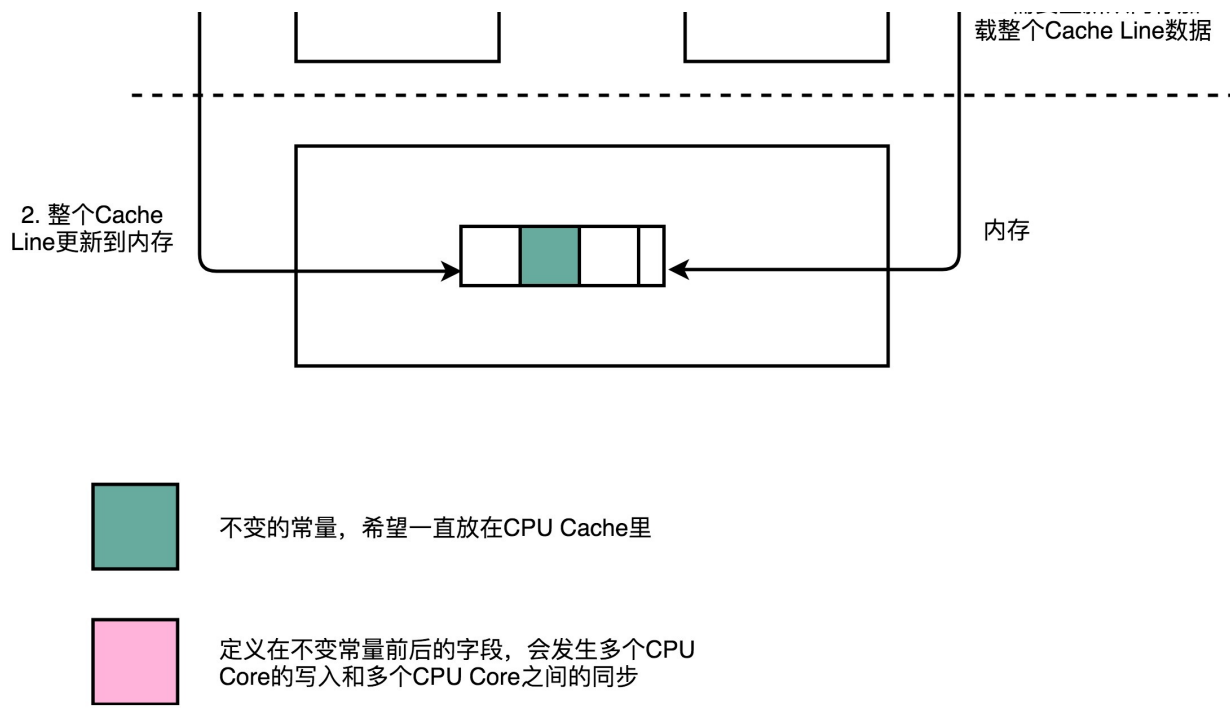
存储器	硬件介质	单位成本(美元/MB)	随机访问延时	说明
L1 Cache	SRAM	7	1ns	
L2 Cache	SRAM	7	4ns	访问延时15x L1 Cache
Memory	DRAM	0.015	100ns	访问延时15X SRAM，价格1/40 SRAM
Disk	SSD(NAND)	0.0004	150μs	访问延时 1500X DRAM，价格 1/40 DRAM
Disk	HDD	0.00004	10ms	访问延时 70X SSD，价格 1/10 SSD

CPU Cache 装载内存里面的数据，不是一个一个字段加载的，而是加载一整个缓存行。举个例子，如果我们定义了一个长度为 64 的 long 类型的数组。那么数据从内存加载到 CPU Cache 里面的时候，不是一个一个数组元素加载的，而是一次性加载固定长度的一个缓存行。

我们现在的 64 位 Intel CPU 的计算机，缓存行通常是 64 个字节（Bytes）。一个 long 类型的数据需要 8 个字节，所以我们一下子会加载 8 个 long 类型的数据。也就是说，一次加载数组里面连续的 8 个数值。这样的加载方式使得我们遍历数组元素的时候会很快。因为后面连续 7 次的数据访问都会命中缓存，不需要重新从内存里面去读取数据。这个性能层面的好处，我在第 37 讲的第一个例子里面为你演示过，印象不深的话，可以返回去看看。

但是，在我们不是使用数组，而是使用单独的变量的时候，这里就会出现问题了。在 Disruptor 的 RingBuffer（环形缓冲区）的代码里面，定义了一个单独的 long 类型的变量。这个变量叫作 INITIAL\_CURSOR\_VALUE，用来存放 RingBuffer 起始的元素位置。





CPU 在加载数据的时候，自然也会把这个数据从内存加载到高速缓存里面来。不过，这个时候，高速缓存里面除了这个数据，还会加载这个数据前后定义的其他变量。这个时候，问题就来了。Disruptor 是一个多线程的服务器框架，在这个数据前后定义的其他变量，可能会被多个不同的线程去更新数据、读取数据。这些写入以及读取的请求，会来自于不同的 CPU Core。于是，为了保证数据的同步更新，我们不得不把 CPU Cache 里面的数据，重新写回到内存里面去或者重新从内存里面加载数据。

而我们刚刚说过，这些 CPU Cache 的写回和加载，都不是以一个变量作为单位的。这些动作都是以整个 Cache Line 作为单位的。所以，当 INITIAL\_CURSOR\_VALUE 前后的那些变量被写回到内存的时候，这个字段自己也写回到了内存，这个常量的缓存也就失效了。当我们要再次读取这个值的时候，要再重新从内存读取。这也就意味着，读取速度大大变慢了。

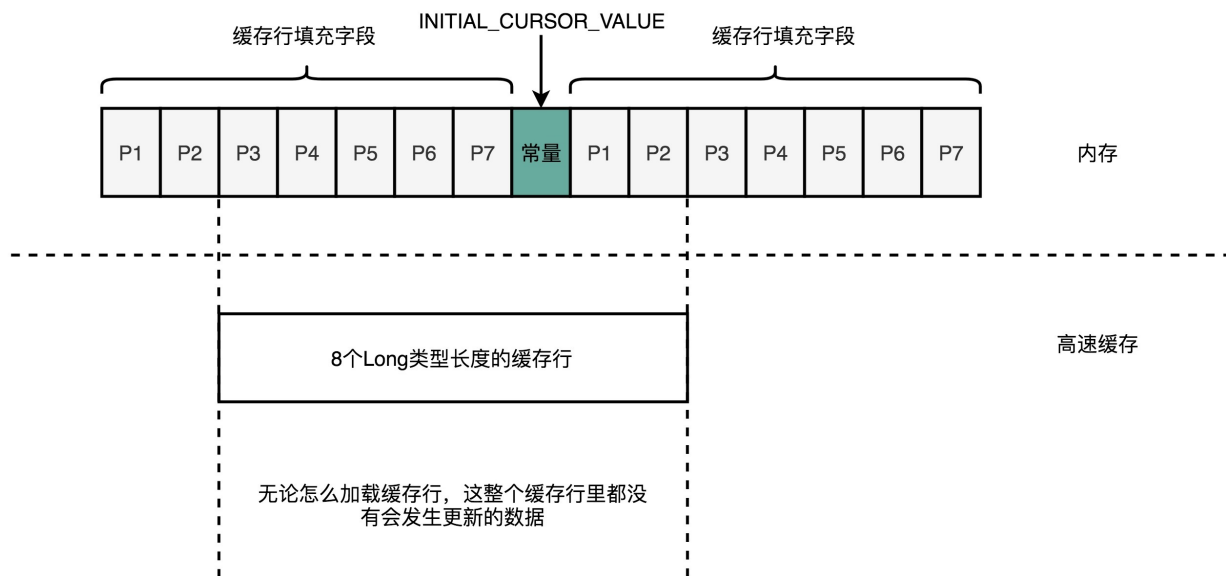
.....

```
abstract class RingBufferPad
{
    protected long p1, p2, p3, p4, p5, p6, p7;
}

abstract class RingBufferFields<E> extends RingBufferPad
{
    .....
}

public final class RingBuffer<E> extends RingBufferFields<E> implements Cursored, Eve
```

```
{  
    public static final long INITIAL_CURSOR_VALUE = Sequence.INITIAL_VALUE;  
    protected long p1, p2, p3, p4, p5, p6, p7;  
    .....  
}
```

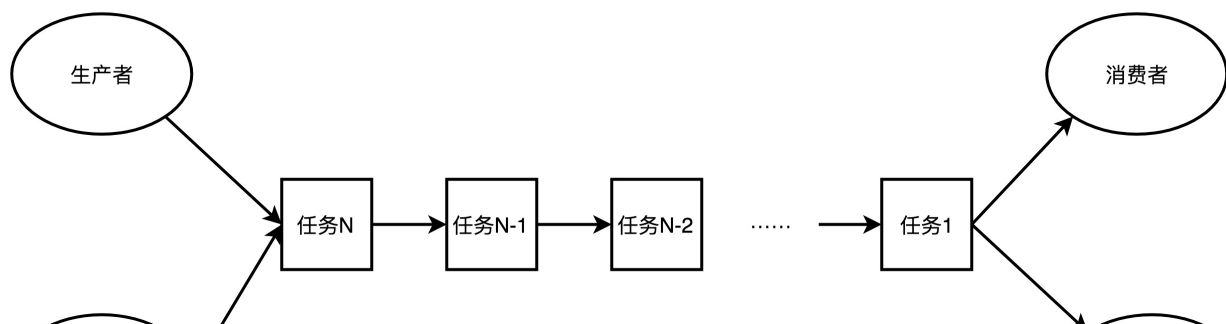


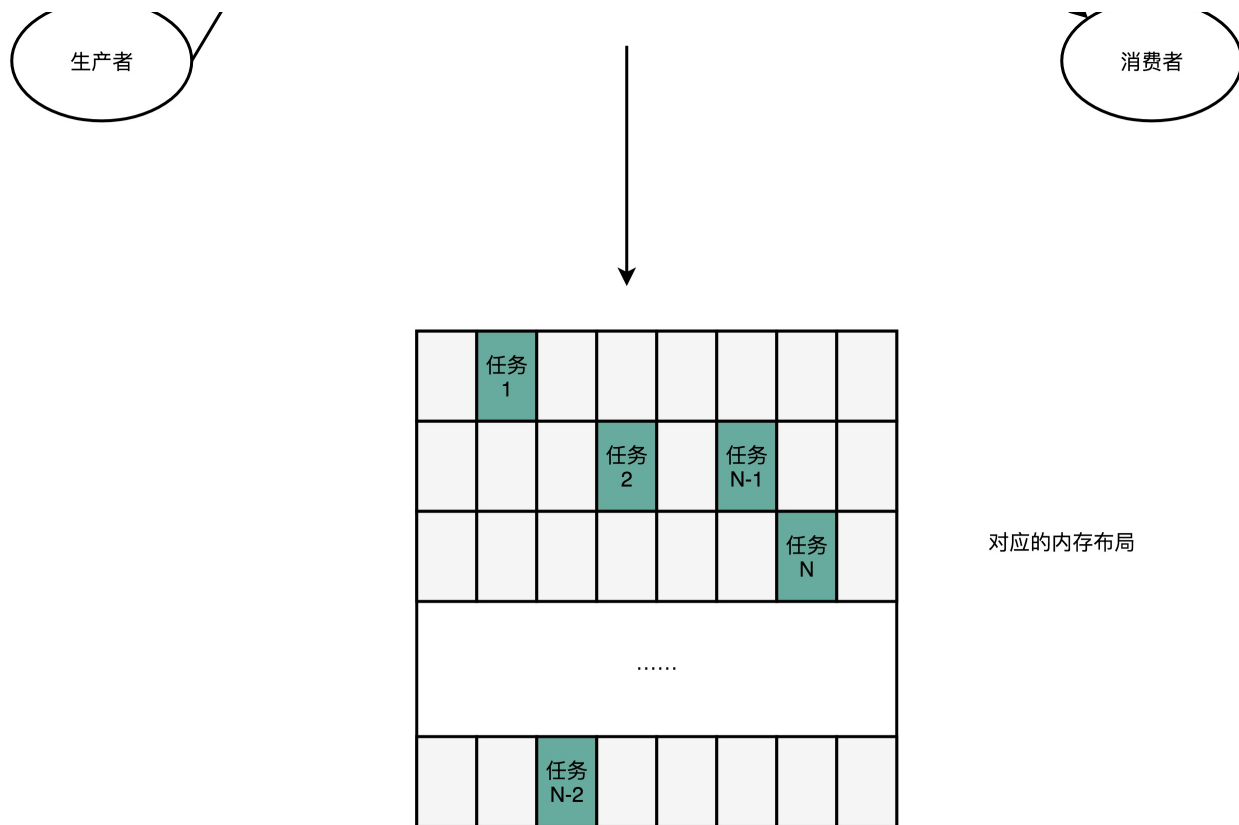
面临这样一个情况，Disruptor 里发明了一个神奇的代码技巧，这个技巧就是缓存行填充。Disruptor 在 `INITIAL_CURSOR_VALUE` 的前后，分别定义了 7 个 long 类型的变量。前面的 7 个来自继承的 `RingBufferPad` 类，后面的 7 个则是直接定义在 `RingBuffer` 类里面。这 14 个变量没有任何实际的用途。我们既不会去读他们，也不会去写他们。

而 `INITIAL_CURSOR_VALUE` 又是一个常量，也不会进行修改。所以，一旦它被加载到 CPU Cache 之后，只要被频繁地读取访问，就不会再被换出 Cache 了。这也就意味着，对于这个值的读取速度，会一直是 CPU Cache 的访问速度，而不是内存的访问速度。

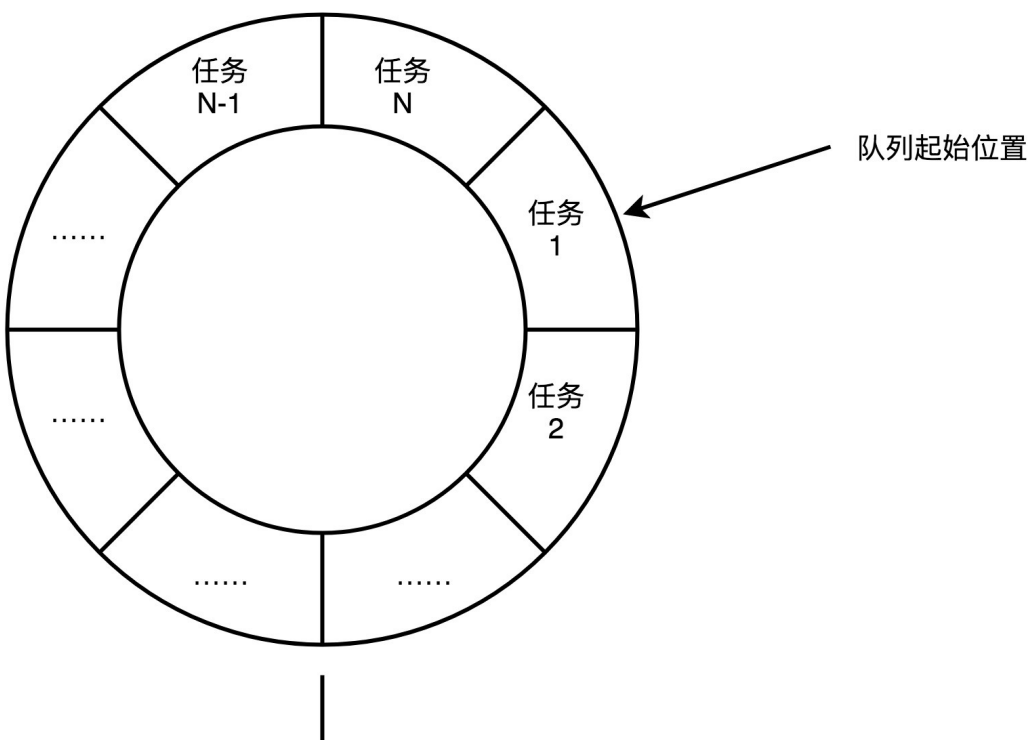
## 使用 RingBuffer，利用缓存和分支预测

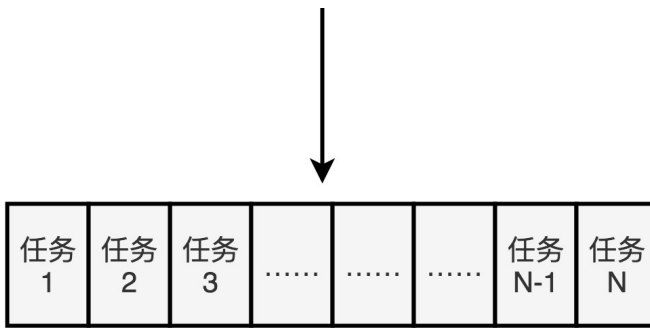
其实这个利用 CPU Cache 的性能的思路，贯穿了整个 Disruptor。Disruptor 整个框架，其实就是一个高速的**生产者 - 消费者模型** (Producer-Consumer) 下的队列。生产者不停地往队列里面生产新的需要处理的任务，而消费者不停地从队列里面处理掉这些任务。



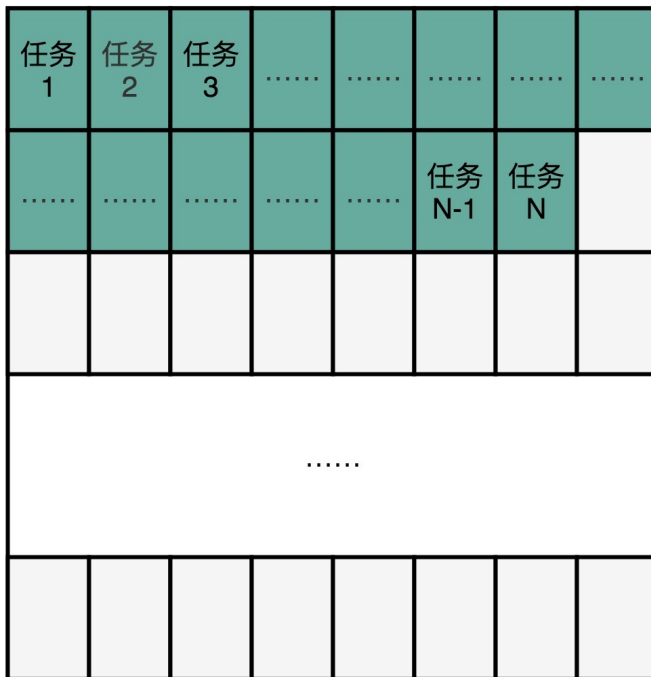


如果你熟悉算法和数据结构，那你应该非常清楚，如果要实现一个队列，最合适的数据结构应该是链表。我们只要维护好链表的头和尾，就能很容易实现一个队列。生产者只要不断地往链表的尾部不断插入新的节点，而消费者只需要不断从头部取出最老的节点进行处理就好了。我们可以很容易实现生产者 - 消费者模型。实际上，Java 自己的基础库里面就有 `LinkedBlockingQueue` 这样的队列库，可以直接用在生产者 - 消费者模式上。





实际代码里是一个数组



内存数据布局连续  
有空间局部性  
可以被CPU Cache利用好

不过，Disruptor 里面并没有用 `LinkedBlockingQueue`，而是使用了一个 `RingBuffer` 这样的数据结构，这个 `RingBuffer` 的底层实现则是一个固定长度的数组。比起链表形式的实现，数组的数据在内存里面会存在空间局部性。

就像上面我们看到的，数组的连续多个元素会一并加载到 CPU Cache 里面来，所以访问遍历的速度会更快。而链表里面各个节点的数据，多半不会出现在相邻的内存空间，自然也就享受不到整个 Cache Line 加载后数据连续从高速缓存里面被访问到的优势。

除此之外，数据的遍历访问还有一个很大的优势，就是 CPU 层面的分支预测会很准确。这可以使得我们更有效地利用了 CPU 里面的多级流水线，我们的程序就会跑得更快。这一部分的原理如果你已经不太记得了，可以回过头去复习一下[第 25 讲](#)关于分支预测的内容。

## 总结延伸

---

好了，不知道讲完这些，你有没有体会到 Disruptor 这个框架的神奇之处呢？

CPU 从内存加载数据到 CPU Cache 里面的时候，不是一个变量一个变量加载的，而是加载固定长度的 Cache Line。如果是加载数组里面的数据，那么 CPU 就会加载到数组里面连续的多个数据。所以，数组的遍历很容易享受到 CPU Cache 那风驰电掣的速度带来的红利。

对于类里面定义的单独的变量，就不容易享受到 CPU Cache 红利了。因为这些字段虽然在内存层面会分配到一起，但是实际应用的时候往往没有什么关联。于是，就会出现多个 CPU Core 访问的情况下，数据频繁在 CPU Cache 和内存里面来来回回的情况。而 Disruptor 很取巧地在需要频繁高速访问的常量 INITIAL\_CURSOR\_VALUE 前后，各定义了 7 个没有任何作用和读写请求的 long 类型的变量。

这样，无论在内存的什么位置上，这个 INITIAL\_CURSOR\_VALUE 所在的 Cache Line 都不会有任何写更新的需求。我们就可以始终在 Cache Line 里面读到它的值，而不需要从内存里面去读取数据，也就大大加速了 Disruptor 的性能。

这样的思路，其实渗透在 Disruptor 这个开源框架的方方面面。作为一个生产者 - 消费者模型，Disruptor 并没有选择使用链表来实现一个队列，而是使用了 RingBuffer。RingBuffer 底层的数据结构则是一个固定长度的数组。这个数组不仅让我们更容易用好 CPU Cache，对 CPU 执行过程中的分支预测也非常有利。更准确的分支预测，可以使得我们更好地利用好 CPU 的流水线，让代码跑得更快。

## 推荐阅读

---

今天讲的是 Disruptor，推荐的阅读内容自然是 Disruptor 的官方文档。作为一个开源项目，Disruptor 在自己[GitHub](#)上有很详细的设计文档，推荐你好好阅读一下。

这里面不仅包含了怎么用好 Disruptor，也包含了整个 Disruptor 框架的设计思路，是一份很好的阅读学习材料。另外，Disruptor 的官方文档里，还有很多文章、演讲，详细介绍了这个框架，很值得深入去看一看。Disruptor 的源代码其实并不复杂，很适合用来学习怎么阅读开源框架代码。

[上一页](#)

[下一页](#)