

29 | Copy-on-Write模式：不是延时策略的COW

2019-05-04 王宝令

Java并发编程实战

[进入课程 >](#)



讲述：王宝令

时长 08:20 大小 7.64M



在上一篇文章中我们讲到 Java 里 String 这个类在实现 `replace()` 方法的时候，并没有更改原字符串里面 `value[]` 数组的内容，而是创建了一个新字符串，这种方法在解决不可变对象的修改问题时经常用到。如果你深入地思考这个方法，你会发现它本质上是一种**Copy-on-Write 方法**。所谓 Copy-on-Write，经常被缩写为 COW 或者 CoW，顾名思义就是**写时复制**。

不可变对象的写操作往往都是使用 Copy-on-Write 方法解决的，当然 Copy-on-Write 的应用领域并不局限于 Immutability 模式。下面我们先简单介绍一下 Copy-on-Write 的应用领域，让你对它有个更全面的认识。

Copy-on-Write 模式的应用领域

我们前面在[《20 | 并发容器：都有哪些“坑”需要我们填？》](#)中介绍过

CopyOnWriteArrayList 和 CopyOnWriteArraySet 这两个 Copy-on-Write 容器，它们背后的设计思想就是 Copy-on-Write；通过 Copy-on-Write 这两个容器实现的读操作是无锁的，由于无锁，所以将读操作的性能发挥到了极致。

除了 Java 这个领域，Copy-on-Write 在操作系统领域也有广泛的应用。

我第一次接触 Copy-on-Write 其实就是在操作系统领域。类 Unix 的操作系统中创建进程的 API 是 fork()，传统的 fork() 函数会创建父进程的一个完整副本，例如父进程的地址空间现在用到了 1G 的内存，那么 fork() 子进程的时候要复制父进程整个进程的地址空间（占有 1G 内存）给予进程，这个过程是很耗时的。而 Linux 中的 fork() 函数就聪明得多了，fork() 子进程的时候，并不复制整个进程的地址空间，而是让父子进程共享同一个地址空间；只用在父进程或者子进程需要写入的时候才会复制地址空间，从而使父子进程拥有各自的地址空间。

本质上来讲，父子进程的地址空间以及数据都是要隔离的，使用 Copy-on-Write 更多地体现的是一种**延时策略，只有在真正需要复制的时候才复制，而不是提前复制好**，同时 Copy-on-Write 还支持按需复制，所以 Copy-on-Write 在操作系统领域是能够提升性能的。相比较而言，Java 提供的 Copy-on-Write 容器，由于在修改的同时会复制整个容器，所以在提升读操作性能的同时，是以内存复制为代价的。这里你会发现，同样是应用 Copy-on-Write，不同的场景，对性能的影响是不同的。

在操作系统领域，除了创建进程用到了 Copy-on-Write，很多文件系统也同样用到了，例如 Btrfs (B-Tree File System)、aufs (advanced multi-layered unification filesystem) 等。

除了上面我们说的 Java 领域、操作系统领域，很多其他领域也都能看到 Copy-on-Write 的身影：Docker 容器镜像的设计是 Copy-on-Write，甚至分布式源码管理系统 Git 背后的设计思想都有 Copy-on-Write.....

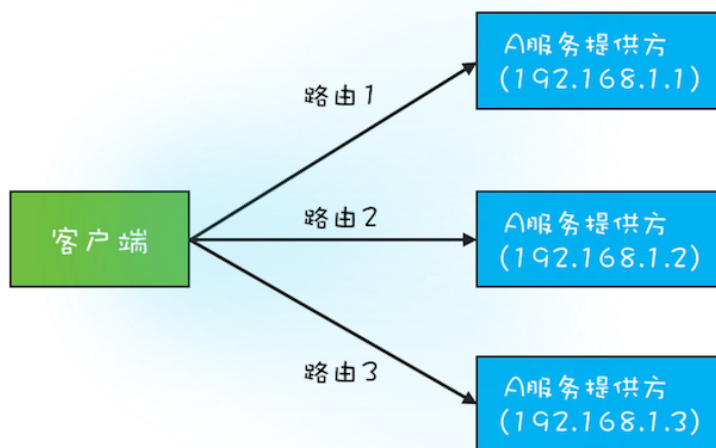
不过，**Copy-on-Write 最大的应用领域还是在函数式编程领域**。函数式编程的基础是不可变性 (Immutability)，所以函数式编程里面所有的修改操作都需要 Copy-on-Write 来解决。你或许会有疑问，“所有数据的修改都需要复制一份，性能是不是会成为瓶颈呢？”你的担忧是有道理的，之所以函数式编程早年间没有兴起，性能绝对拖了后腿。但是随着硬件性能的提升，性能问题已经慢慢变得可以接受了。而且，Copy-on-Write 也远不像 Java

里的 CopyOnWriteArrayList 那样笨：整个数组都复制一遍。Copy-on-Write 也是可以按需复制的，如果你感兴趣可以参考[Purely Functional Data Structures](#)这本书，里面描述了各种具备不变性的数据结构的实现。

CopyOnWriteArrayList 和 CopyOnWriteArraySet 这两个 Copy-on-Write 容器在修改的时候会复制整个数组，所以如果容器经常被修改或者这个数组本身就非常大的时候，是不建议使用的。反之，如果是修改非常少、数组数量也不大，并且对读性能要求苛刻的场景，使用 Copy-on-Write 容器效果就非常好了。下面我们结合一个真实的案例来讲解一下。

一个真实案例

我曾经写过一个 RPC 框架，有点类似 Dubbo，服务提供方是多实例分布式部署的，所以服务的客户端在调用 RPC 的时候，会选定一个服务实例来调用，这个选定的过程本质上就是在做负载均衡，而做负载均衡的前提是客户端要有全部的路由信息。例如在下图中，A 服务的提供方有 3 个实例，分别是 192.168.1.1、192.168.1.2 和 192.168.1.3，客户端在调用目标服务 A 前，首先需要做的是负载均衡，也就是从这 3 个实例中选出 1 个来，然后再通过 RPC 把请求发送选中的目标实例。



RPC 路由关系图

RPC 框架的一个核心任务就是维护服务的路由关系，我们可以把服务的路由关系简化成下图所示的路由表。当服务提供方上线或者下线的时候，就需要更新客户端的这张路由表。

接口	服务提供方IP	服务提供方端口	状态
io.service.helloworld	192.168.1.1	7890	ONLINE
io.service.helloworld	192.168.1.2	7890	ONLINE
io.service.hellojava	192.168.2.3	7890	OFFLINE

我们首先来分析一下如何用程序来实现。每次 RPC 调用都需要通过负载均衡器来计算目标服务的 IP 和端口号，而负载均衡器需要通过路由表获取接口的所有路由信息，也就是说，每次 RPC 调用都需要访问路由表，所以访问路由表这个操作的性能要求是很高的。不过路由表对数据的一致性要求并不高，一个服务提供方从上线到反馈到客户端的路由表里，即便有 5 秒钟，很多时候也都是能接受的（5 秒钟，对于以纳秒作为时钟周期的 CPU 来说，那何止是一万年，所以路由表对一致性的要求并不高）。而且路由表是典型的读多写少类问题，写操作的量相比于读操作，可谓是沧海一粟，少得可怜。

通过以上分析，你会发现一些关键词：对读的性能要求很高，读多写少，弱一致性。它们综合在一起，你会想到什么呢？CopyOnWriteArrayList 和 CopyOnWriteArraySet 天生就适用这种场景啊。所以下面的示例代码中，RouteTable 这个类内部我们通过 `ConcurrentHashMap<String, CopyOnWriteArraySet<Router>>` 这个数据结构来描述路由表，ConcurrentHashMap 的 Key 是接口名，Value 是路由集合，这个路由集合我们用的是 CopyOnWriteArraySet。

下面我们再来思考 Router 该如何设计，服务提供方的每一次上线、下线都会更新路由信息，这时候你有两种选择。一种是通过更新 Router 的一个状态位来标识，如果这样做，那么所有访问该状态位的地方都需要同步访问，这样很影响性能。另外一种就是采用 Immutability 模式，每次上线、下线都创建新的 Router 对象或者删除对应的 Router 对象。由于上线、下线的频率很低，所以后者是最好的选择。

Router 的实现代码如下所示，是一种典型 Immutability 模式的实现，需要你注意的是我们重写了 equals 方法，这样 CopyOnWriteArraySet 的 add() 和 remove() 方法才能正常工作。

```

1 // 路由信息
2 public final class Router{
3     private final String ip;
4     private final Integer port;
5     private final String iface;
6     // 构造函数
7     public Router(String ip,
8         Integer port, String iface){
9         this.ip = ip;
10        this.port = port;
11        this.iface = iface;
12    }
13    // 重写 equals 方法
14    public boolean equals(Object obj){
15        if (obj instanceof Router) {
16            Router r = (Router)obj;
17            return iface.equals(r.iface) &&
18                ip.equals(r.ip) &&
19                port.equals(r.port);
20        }
21        return false;
22    }
23    public int hashCode() {
24        // 省略 hashCode 相关代码
25    }
26 }
27 // 路由表信息
28 public class RouterTable {
29     //Key: 接口名
30     //Value: 路由集合
31     ConcurrentHashMap<String, CopyOnWriteArraySet<Router>>
32     rt = new ConcurrentHashMap<>();
33     // 根据接口名获取路由表
34     public Set<Router> get(String iface){
35         return rt.get(iface);
36     }
37     // 删除路由
38     public void remove(Router router) {
39         Set<Router> set=rt.get(router.iface);
40         if (set != null) {
41             set.remove(router);
42         }
43     }
44     // 增加路由
45     public void add(Router router) {
46         Set<Router> set = rt.computeIfAbsent(
47             route.iface, r ->
48                 new CopyOnWriteArraySet<>());
49         set.add(router);
50     }

```


总结

目前 Copy-on-Write 在 Java 并发编程领域知名度不是很高，很多人都在无意中把它忽视了，但其实 Copy-on-Write 才是最简单的并发解决方案。它是如此简单，以至于 Java 中的基本数据类型 String、Integer、Long 等都是基于 Copy-on-Write 方案实现的。

Copy-on-Write 是一项非常通用的技术方案，在很多领域都有着广泛的应用。不过，它也有缺点的，那就是消耗内存，每次修改都需要复制一个新的对象出来，好在随着自动垃圾回收（GC）算法的成熟以及硬件的发展，这种内存消耗已经渐渐可以接受了。所以在实际工作中，如果写操作非常少，那你就可以尝试用一下 Copy-on-Write，效果还是不错的。

课后思考

Java 提供了 CopyOnWriteArrayList，为什么没有提供 CopyOnWriteLinkedList 呢？

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。



Java 并发编程实战

全面系统提升你的并发编程能力

王宝令

资深架构师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

上一篇 28 | Immutability模式：如何利用不变性解决并发问题？

下一篇 30 | 线程本地存储模式：没有共享，就没有伤害

精选留言 (25)

写留言



GeekAml

2019-05-04

24

CopyOnWriteLinkedList的链表结构读取效率比较低，就违背了读多写少的设计初衷。



Knight20...

2019-05-06

7

很多童鞋提到了链表copy的代价，个人觉得这并不是最根本的原因。首先数组无论的新增还是删除copy是避免不了的，因此我们采用copy on write的方式在保证代价相当的前提下保证了并发的安全问题，何乐而不为呢。其次是链表的新增删除压根就不需要复制，就算是在并发场景下采用锁的方式性能损耗都不大，因此也就没必要采用copy的方式了，更何况链表的操作可以采用分段锁、节点锁。所以没有CopyOnWriteLinkedList的主要原...

展开



Corner

2019-05-04

6

数组的拷贝效率应该比链表高，一维数组是连续分配内存的，所以可以直接复制内存块就能完成拷贝。但是链表元素之间是通过引用建立连接的，所以要遍历整个链表才能完成拷贝。

展开



假行僧

2019-05-05

4

没有提供CopyOnWriteLinkedList是因为linkedlist的数据结构关系分散到每一个节点里面，对每一个节点的修改都存在竞态条件，需要同步才能保证一致性。arraylist就不一样，数组天然的拥有前驱后继的结构关系，对列表的增删，因为是copy on write，所以只需要cas操作数组对象就能够保证线程安全，效率上也能接受，更重要的是避免锁竞争带来的上

下文切换消耗。有一点需要注意的是CopyOnWriteArrayList在使用上有数据不完整的时...
展开 ▾

作者回复: 📖



ban

2019-05-04

👍 2

一种是通过更新 Router 的一个状态位来标识，如果这样做，那么所有访问该状态位的地方都需要同步访问，这样很影响性能。

老师好，这句话的意思没怎么看懂，我理解的是route如果下线后更新状态标识，所以每次调用的时候都需要遍历所以route节点，判断每个节点的状态来判断是否下线，所以比较...
展开 ▾



nonohony

2019-05-08

👍 1

- 1.链表本身适合于顺序读和写多的场景，和cop读多写少是违背的。
- 2.链表可以锁节点，力度已经很小了。
- 3.链表整体复制的性能比数组差太多。

展开 ▾



夏天

2019-05-06

👍 1

王老师，问一个单例模式的问题：在双重检查加锁的单例模式中 需不需要加 volatile 关键字修饰？自己的理解：是需要。但是我在考虑其中的锁是不是存在happen before规则，不用加volatile也能保证可见性？

展开 ▾

作者回复: 必须加，还有指令重排问题



周治慧

2019-05-05

👍 1

本质就是数组查询块增删慢，链表增删块查询慢。copyandwrite本质就是读多写少即查询

多增删少的一个过程所以数组更加合适



晓杰

2019-05-04

👍 1

LinkedList本身适用于写多读少的场景，而copy-on-write模式适用于读多写少的场景，两者适用场景相反。



Darren

2019-05-04

👍 1

LinkedList 在复制时，因为其包含前后节点地址，每个节点需要去创建，成本比较高，所以很少或者没有写时复制的Linked 结构吧



污名侦探

2019-05-17

👍

首先CopyOnWriteLinkedList 可以做分段锁，并且性能很高。其次，复制性能没有数组来的快。



Vincent

2019-05-16

👍

既然读多写少，说明数据结构变更频率很少。那么数组结构适合这个场景，链表是适合写多的场景



Geek_bbbda...

2019-05-16

👍

链表读操作时间复杂度高，用copy on write 也解决不了这个问题，天生不适合读多场景。



Zach_

2019-05-13

👍

COWLinkedList 违背了 读多写少 读高效的初衷了哇

展开 ∨



Zach_

2019-05-13



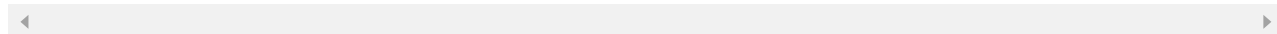
Sting Long ... 居然可以和CoW、Lock联系起来!

跟着老师默默修行!

希望这个专栏永远不要停!...

展开 ▾

作者回复: 感谢捧场 😊



刘鹏

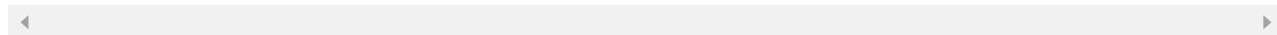
2019-05-10



服务下线了, 如果数据不一致, 会不会有请求发到下线了的服务器

展开 ▾

作者回复: rpc的客户端和服务提供端会建立一个长连接, 定时发心跳, 并不完全依赖注册中心的数据。很多rpc的服务端提供了手动下线功能, 能解决你说的这个问题



刘鹏

2019-05-10



一种策略

展开 ▾



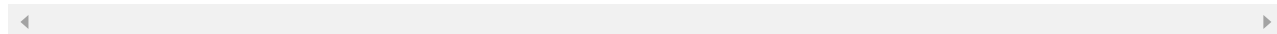
WL

2019-05-08



请问一下老师copyonwrite如果与volatile结合使用是不是就可以实现强一致性了?

作者回复: 好像没这么简单



张三

2019-05-06



上一篇说包装类型、String 是享元模式，这篇说是Copy-on-Write，是两种模式都有吗？

作者回复: 都有



张天屹

2019-05-05



写时复制用空间换时间，如果频繁写则会新建很多数组对象。数组结构写慢读快，链表结构写快读慢。适用于CopyOnWrite的读多写少场景肯定就不适合链表结构了。

展开 ▾