**29 Copy-on-Write模式：不是延时策略的COW**[](#29-copy-on-writecow)

* Copy-on-Write 模式的应用领域
* 一个真实的案例
* 总结
* 思考

在上一篇文章我们讲到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，而是创建一个新字符串，这种方法在解决不可变对象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使用Copy-on-Write方法解决，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**[](#copy-on-write)

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）给子进程，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。

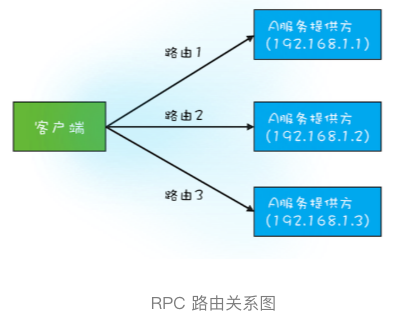
除了上面我们说的 Java 领域、操作系统领域，很多其他领域也都能看到 Copy-on-Write 的身影：Docker 容器镜像的设计是 Copy-on-Write，甚至分布式源码管理系统 Git 背后的设计思想都有 Copy-on-Write……

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**[](#_1)

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 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。



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>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。

*// 路由信息*

**public** **final** **class** **Router{**

**private** **final** String ip**;**

**private** **final** Integer port**;**

**private** **final** String iface**;**

*// 构造函数*

**public** **Router(**String ip**,**

Integer port**,** String iface**){**

**this.**ip **=** ip**;**

**this.**port **=** port**;**

**this.**iface **=** iface**;**

**}**

*// 重写 equals 方法*

**public** **boolean** **equals(**Object obj**){**

**if** **(**obj **instanceof** Router**)** **{**

Router r **=** **(**Router**)**obj**;**

**return** iface**.**equals**(**r**.**iface**)** **&&**

ip**.**equals**(**r**.**ip**)** **&&**

port**.**equals**(**r**.**port**);**

**}**

**return** **false;**

**}**

**public** **int** **hashCode()** **{**

*// 省略 hashCode 相关代码*

**}**

**}**

*// 路由表信息*

**public** **class** **RouterTable** **{**

*//Key: 接口名*

*//Value: 路由集合*

ConcurrentHashMap**<**String**,** CopyOnWriteArraySet**<**Router**>>**

rt **=** **new** ConcurrentHashMap**<>();**

*// 根据接口名获取路由表*

**public** Set**<**Router**>** **get(**String iface**){**

**return** rt**.**get**(**iface**);**

**}**

*// 删除路由*

**public** **void** **remove(**Router router**)** **{**

Set**<**Router**>** set**=**rt**.**get**(**router**.**iface**);**

**if** **(**set **!=** **null)** **{**

set**.**remove**(**router**);**

**}**

**}**

*// 增加路由*

**public** **void** **add(**Router router**)** **{**

Set**<**Router**>** set **=** rt**.**computeIfAbsent**(**

route**.**iface**,** r **->**

**new** CopyOnWriteArraySet**<>());**

set**.**add**(**router**);**

**}**

**}**

**总结**[](#_2)

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。

**思考**[](#_3)

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

数组在内存地址是连续的，天然适合copy，链表是分散的。

主要是ArrayList的数据存储是数组，复制可能只需要移动一个内存页或者多个连续的内存空间就可以，而且数组在复制的时候是知道数据集的大小的(动态扩容后也还是数组，只是预先申请了一些未来使用的空间)，而LinkdList底层实现为使用Node<?>链表，存储位置分散且大小不可控，如果使用COW可能会适得其反。这应该也是一种用空间换时间的策略吧。这么来看，除非事先限定了数据的存储区域，不然用COW还是数组方便些吧。

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竟态条件，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 wirte，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间窗口，要不要考虑需要根据具体场景定夺