写时复制，写时拷贝，写时分裂

（Copy-on-write，简称COW）是计算机资源管理方面的一种优化技术，有着广泛的应用，比如内存管理（进程的 fork），数据存储（qemu-kvm 虚拟机镜像乃至 Docker 的 AUFS 文件系统），软件开发(Java的Copy On Write容器)等等。

其核心思想是，如果有多个调用者（callers）同时要求使用同一个资源（如内存或磁盘上的数据存储），他们会共同获取相同的指针指向该资源，直到某个调用者试图修改资源的内容时，利用引用计数，系统才会真正复制一份专用副本（private copy）给该调用者，把跟新的内容写入该副本中，而其他调用者所见到的最初的资源仍然保持不变，从而节省创建多个完整副本时带来的空间和时间上的开销。这过程对其他的调用者都是透明的（transparently）。此作法主要的优点是如果调用者没有修改该资源，就不会有副本（private copy）被创建，因此多个调用者只是读取操作时可以共享同一份资源。

资源

+-------+

| 1 |

+-------+

用户 A ----> | 2 | <----用户B

+-------+

| 3 |

+-------+

当 A 需更新 Block 1 时，A 新建 New 1 Block，并把新的内容写入该 Block，从而避免影响 B 的 Block 1。同理，当 B 需更新 Block 3 时，B 新建 New 3 Block，并把新的内容写入该 Block，避免影响 A 的 Block 3。

资源

+-------+ +-------+

| New 1 | | 1 |

+-------+ +-------+

用户A ----> | 2 | <----用户B

+-------+ +-------+

A see: New 1, 2, 3 | 3 | | New 3 | B see: 1, 2, New 3

+-------+ +-------+

■内存管理中的应用

一般把这种被共享访问的页面标记为只读。当一个task试图向内存中写入数据时，内存管理单元（MMU）抛出一个异常，内核处理该异常时为该task分配一份物理内存并复制数据到此内存，重新向MMU发出执行该task的写操作。

Fork()

早期的 Unix 在实现 fork 系统调用时，并没有使用该技术，创建新进程的开销很大。出于效率考虑，Copy On Write 技术引入到进程中，fork 之后的父进程和子进程完全共享数据段、代码段、堆和栈等的完全副本，而且内核将共享的地址空间的访问权限改变为只读，如果父进程和子进程中的任何一个试图修改这些区域，则内核职位修改区域的那块内存制作一个副本，通常是虚拟存储系统中的一“页”(引自 Unix 高级环境编程)。

■数据存储中的应用

Linux等的文件管理系统使用了写时复制策略。

数据库服务器也一般采用了写时复制策略，为用户提供一份snapshot。

qcow

qcow是由QEMU（托管虚拟机监视器）使用的磁盘镜像文件的文件格式。是“QEMU写入时复制(QEMU Copy on write)”的缩写，使用了延迟存储分配，直到它实际需要的磁盘存储优化策略。qcow格式的文件可以包含通常与特定客户机操作系统相关联的各种磁盘镜像。存在两个版本的格式：qcow和qcow2，它们分别使用.qcow和.qcow2文件扩展名。

qcow磁盘镜像的一个主要特性是，具有此格式的文件可以随着数据添加而增长。这允许比原始磁盘镜像更小的文件大小，即将整个镜像空间分配给文件，即使它的一部分是空的。这对于不支持孔的文件系统（例如FAT32）特别有用。

qcow格式还允许通过使用写入时复制将对只读基本镜像所做的更改存储在单独的qcow文件上。这个新的qcow文件包含基本镜像的路径，以便在需要时引用它。当必须从该新图像读取特定数据片段时，如果是新的并且存储在那里，则从其检索内容; 如果不是，则从基本图像提取数据。

可选功能包括AES加密和基于zlib的透明解压缩。

Qcow镜像的一个缺点是它们不能直接作为原始磁盘镜像安装。 需要能够读取qcow文件的实用程序才能装载它。

qcow2

qcow2是qcow格式的更新版本，旨在取代它。 与原来的区别是，qcow2通过一个用于存储快照的新的灵活模型支持多个快照。和普通的 Raw 格式镜像相比，具有以下优点：

更小的空间占用

支持 Copy On Write.

支持多级快照

支持压缩

支持 AES

经过多次优化，Qcow2 镜像的性能已经逼近 Raw 格式的镜像，所以 OpenStack 的虚拟机多采用 Qcow2 镜像。

ide 驱动时的性能对比：

cache off writethrough writeback

Old qcow2 (0.10.5) 16:52 min 28:58 min 6:02 min

New qcow2 (0.11.0-rc1) 5:44 min 9:18 min 6:11 min

raw 5:41 min 7:24 min 6:03 min

virtio 驱动时的性能对比：

cache off writeback

Old qcow2 (0.10.5) 31:09 min 8:00 min

New qcow2 (0.11.0-rc1) 18:35 min 8:41 min

raw 8:48 min 7:51 min

除了镜像格式以外，很多文件系统都使用了 Copy-On-Write 技术，以节省空间、提升效率和更好的支持快照。

■软件开发中的应用

在C++的STL中，由于被质疑指责过性能问题，为了提升性能，曾经也有过Copy-On-Write的玩法，例如C++标准程序库中的std::string类，在C++98/C++03标准中是允许写时复制策略。但在C++11标准中为了提高并行性取消了这一策略。GCC从版本5开始，std::string不再采用COW策略。参见陈皓的《C++ STL String类中的Copy-On-Write》，后来，因为有很多线程安全上的事，就被去掉了。这里只给个连接，有兴趣的话，参照下吧。

标准C＋＋类string的Copy-On-Write技术（一）

<http://blog.csdn.net/haoel/article/details/24058>

标准C＋＋类string的Copy-On-Write技术（二）

<http://blog.csdn.net/haoel/article/details/24065>

标准C＋＋类string的Copy-On-Write技术（三）

<http://blog.csdn.net/haoel/article/details/24077>

Java中的Copy-On-Write容器

从JDK1.5开始Java并发包里提供了两个使用CopyOnWrite机制实现的并发容器,它们是CopyOnWriteArrayList和CopyOnWriteArraySet。CopyOnWrite容器非常有用，可以在非常多的并发场景中使用到。

什么是CopyOnWrite容器

CopyOnWrite容器即写时复制的容器。通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。这样做的好处是我们可以对CopyOnWrite容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。所以CopyOnWrite容器也是一种读写分离的思想，读和写不同的容器。

CopyOnWriteArrayList的实现原理

在使用CopyOnWriteArrayList之前，我们先阅读其源码了解下它是如何实现的。以下代码是向ArrayList里添加元素，可以发现在添加的时候是需要加锁的，否则多线程写的时候会Copy出N个副本出来。

public boolean add(T e) {

final ReentrantLock lock = this.lock;

lock.lock();

try {

Object[] elements = getArray();

int len = elements.length;

// 复制出新数组

Object[] newElements = Arrays.copyOf(elements, len + 1);

// 把新元素添加到新数组里

newElements[len] = e;

// 把原数组引用指向新数组

setArray(newElements);

return true;

} finally {

lock.unlock();

}

}

final void setArray(Object[] a) {

array = a;

}

读的时候不需要加锁，如果读的时候有多个线程正在向ArrayList添加数据，读还是会读到旧的数据，因为写的时候不会锁住旧的ArrayList。

public E get(int index) {

return get(getArray(), index);

}

JDK中并没有提供CopyOnWriteMap，我们可以参考CopyOnWriteArrayList来实现一个，基本代码如下：

**package** javay.cow;

**import** java.util.Collection;

**import** java.util.HashMap;

**import** java.util.Map;

**import** java.util.Set;

/\*\*

\* A thread-safe version of {@link Map} in which all operations that change the

\* Map are implemented by making a new copy of the underlying Map.

\*

\* While the creation of a new Map can be expensive, this class is designed for

\* cases in which the primary function is to read data from the Map, not to

\* modify the Map. Therefore the operations that do not cause a change to this

\* class happen quickly and concurrently.

\*/

**public** **class** CopyOnWriteMap<K, V> **implements** Map<K, V>, Cloneable {

**private** **volatile** Map<K, V> internalMap;

/\*\*

\* Creates a new instance of CopyOnWriteMap.

\*/

**public** CopyOnWriteMap() {

internalMap = **new** HashMap<K, V>();

}

/\*\*

\* Creates a new instance of CopyOnWriteMap with the specified initial size

\*

\* **@param** initialCapacity

\* The initial size of the Map.

\*/

**public** CopyOnWriteMap(**int** initialCapacity) {

internalMap = **new** HashMap<K, V>(initialCapacity);

}

/\*\*

\* Creates a new instance of CopyOnWriteMap in which the

\* initial data being held by this map is contained in

\* the supplied map.

\*

\* **@param** data

\* A Map containing the initial contents to be placed into this class.

\*/

**public** CopyOnWriteMap(Map<K, V> data) {

internalMap = **new** HashMap<K, V>(data);

}

/\*\*

\* Adds the provided key and value to this map.

\*

\* **@see** java.util.Map#put(java.lang.Object, java.lang.Object)

\*/

**public** V put(K key, V value) {

**synchronized** (**this**) {

Map<K, V> newMap = **new** HashMap<K, V>(internalMap);

V val = newMap.put(key, value);

internalMap = newMap;

**return** val;

}

}

/\*\*

\* Removed the value and key from this map based on the

\* provided key.

\*

\* **@see** java.util.Map#remove(java.lang.Object)

\*/

**public** V remove(Object key) {

**synchronized** (**this**) {

Map<K, V> newMap = **new** HashMap<K, V>(internalMap);

V val = newMap.remove(key);

internalMap = newMap;

**return** val;

}

}

/\*\*

\* Inserts all the keys and values contained in the

\* provided map to this map.

\*

\* **@see** java.util.Map#putAll(java.util.Map)

\*/

**public** **void** putAll(Map<? **extends** K, ? **extends** V> newData) {

**synchronized** (**this**) {

Map<K, V> newMap = **new** HashMap<K, V>(internalMap);

newMap.putAll(newData);

internalMap = newMap;

}

}

/\*\*

\* Removes all entries in this map.

\*

\* **@see** java.util.Map#clear()

\*/

**public** **void** clear() {

**synchronized** (**this**) {

internalMap = **new** HashMap<K, V>();

}

}

//

// Below are methods that do not modify

// the internal Maps

/\*\*

\* Returns the number of key/value pairs in this map.

\*

\* **@see** java.util.Map#size()

\*/

**public** **int** size() {

**return** internalMap.size();

}

/\*\*

\* Returns true if this map is empty, otherwise false.

\*

\* **@see** java.util.Map#isEmpty()

\*/

**public** **boolean** isEmpty() {

**return** internalMap.isEmpty();

}

/\*\*

\* Returns true if this map contains the provided key, otherwise

\* this method return false.

\*

\* **@see** java.util.Map#containsKey(java.lang.Object)

\*/

**public** **boolean** containsKey(Object key) {

**return** internalMap.containsKey(key);

}

/\*\*

\* Returns true if this map contains the provided value, otherwise

\* this method returns false.

\*

\* **@see** java.util.Map#containsValue(java.lang.Object)

\*/

**public** **boolean** containsValue(Object value) {

**return** internalMap.containsValue(value);

}

/\*\*

\* Returns the value associated with the provided key from this

\* map.

\*

\* **@see** java.util.Map#get(java.lang.Object)

\*/

**public** V get(Object key) {

**return** internalMap.get(key);

}

/\*\*

\* This method will return a read-only {@link Set}.

\*/

**public** Set<K> keySet() {

**return** internalMap.keySet();

}

/\*\*

\* This method will return a read-only {@link Collection}.

\*/

**public** Collection<V> values() {

**return** internalMap.values();

}

/\*\*

\* This method will return a read-only {@link Set}.

\*/

**public** Set<Entry<K, V>> entrySet() {

**return** internalMap.entrySet();

}

@Override

**public** Object clone() {

**try** {

**return** **super**.clone();

} **catch** (CloneNotSupportedException e) {

**throw** **new** InternalError();

}

}

}

CopyOnWrite的应用场景

CopyOnWrite并发容器用于读多写少的并发场景。比如白名单，黑名单，商品类目的访问和更新场景，假如我们有一个搜索网站，用户在这个网站的搜索框中，输入关键字搜索内容，但是某些关键字不允许被搜索。这些不能被搜索的关键字会被放在一个黑名单当中，黑名单每天晚上更新一次。当用户搜索时，会检查当前关键字在不在黑名单当中，如果在，则提示不能搜索。实现代码如下：

**package** javay.cow;

**import** java.util.Map;

/\*\*

\* 黑名单服务

\*/

**public** **class** BlackListServiceImpl {

**private** **static** CopyOnWriteMap<String, Boolean> **blackListMap** = **new** CopyOnWriteMap<String, Boolean>(1000);

**public** **static** **boolean** isBlackList(String id) {

**return** **blackListMap**.get(id) == **null** ? **false** : **true**;

}

**public** **static** **void** addBlackList(String id) {

**blackListMap**.put(id, Boolean.***TRUE***);

}

/\*\*

\* 批量添加黑名单

\*

\* **@param** ids

\*/

**public** **static** **void** addBlackList(Map<String,Boolean> ids) {

**blackListMap**.putAll(ids);

}

}

使用CopyOnWriteMap需要注意两件事情：

1. 减少扩容开销。根据实际需要，初始化CopyOnWriteMap的大小，避免写时CopyOnWriteMap扩容的开销。

2. 使用批量添加。因为每次添加，容器每次都会进行复制，所以减少添加次数，可以减少容器的复制次数。如使用上面代码里的addBlackList方法。

CopyOnWrite的缺点

CopyOnWrite容器有很多优点，但是同时也存在两个问题，即内存占用问题和数据一致性问题。所以在开发的时候需要注意一下。

内存占用问题。因为CopyOnWrite的写时复制机制，所以在进行写操作的时候，内存里会同时驻扎两个对象的内存，旧的对象和新写入的对象（注意:在复制的时候只是复制容器里的引用，只是在写的时候会创建新对象添加到新容器里，而旧容器的对象还在使用，所以有两份对象内存）。如果这些对象占用的内存比较大，比如说200M左右，那么再写入100M数据进去，内存就会占用300M，那么这个时候很有可能造成频繁的Yong GC和Full GC。之前我们系统中使用了一个服务由于每晚使用CopyOnWrite机制更新大对象，造成了每晚15秒的Full GC，应用响应时间也随之变长。

针对内存占用问题，可以通过压缩容器中的元素的方法来减少大对象的内存消耗，比如，如果元素全是10进制的数字，可以考虑把它压缩成36进制或64进制。或者不使用CopyOnWrite容器，而使用其他的并发容器，如ConcurrentHashMap。

数据一致性问题。CopyOnWrite容器只能保证数据的最终一致性，不能保证数据的实时一致性。所以如果你希望写入的的数据，马上能读到，请不要使用CopyOnWrite容器。

其他的比如PHP也有写时复制(cow)的运用，只是都穿着外套而已，不多说了。

（下次吧）