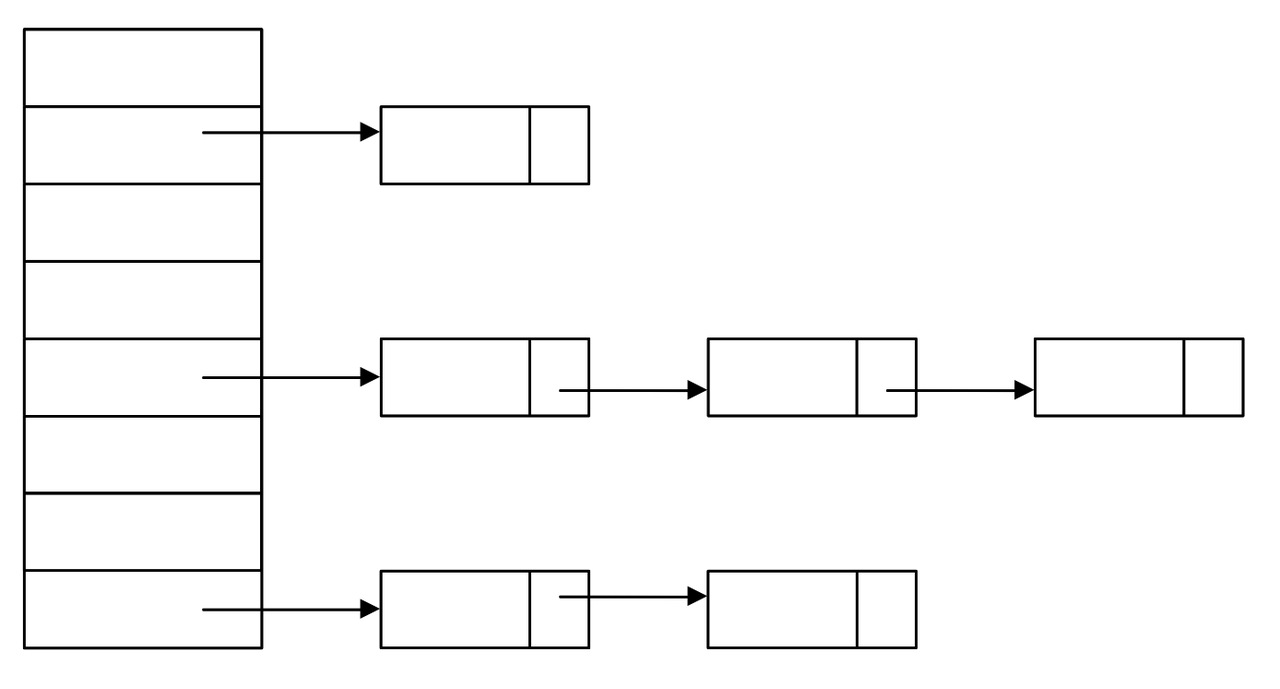
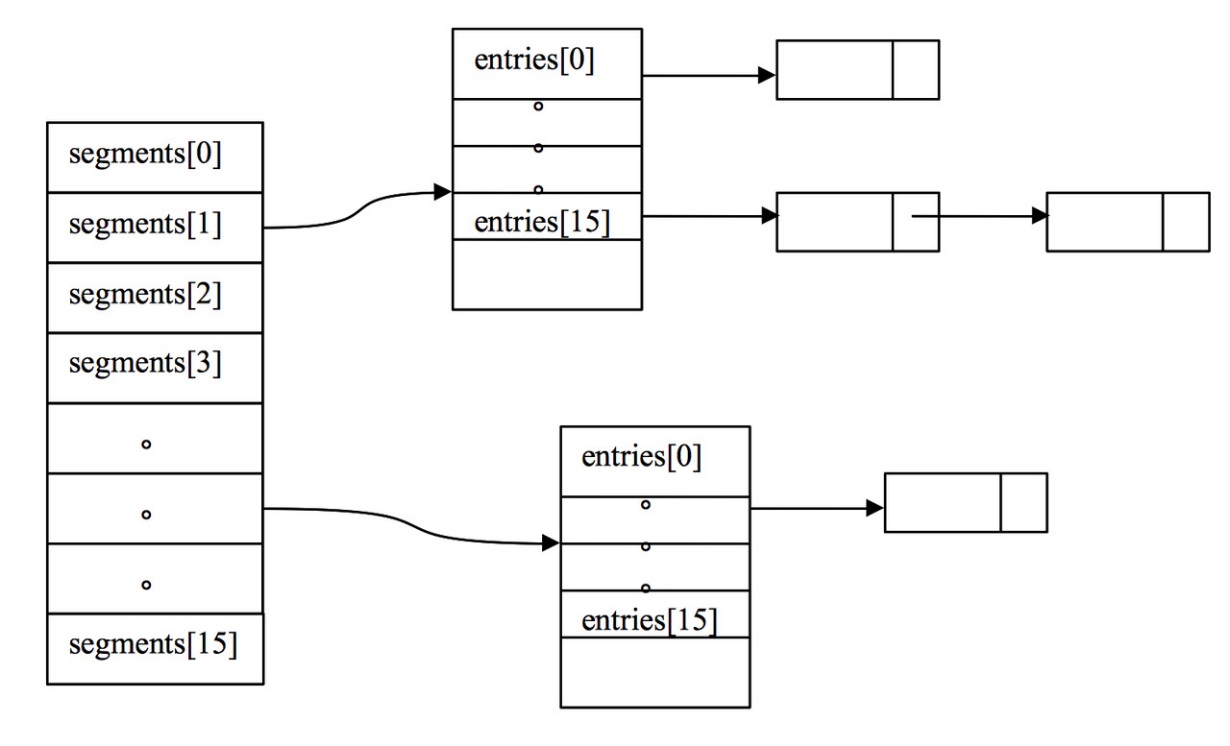
经过之前的铺垫，现在可以进入正题了。  
我们关注的操作有：get，put，remove 这3个操作。

对于哈希表，Java中采用链表的方式来解决hash冲突的。  
一个HashMap的数据结构看起来类似下图：  


实现了同步的HashTable也是这样的结构，它的同步使用锁来保证的，并且所有同步操作使用的是同一个锁对象。这样若有n个线程同时在get时，这n个线程要串行的等待来获取锁。

ConcurrentHashMap中对这个数据结构，针对并发稍微做了一点调整。  
它把区间按照并发级别(concurrentLevel)，分成了若干个segment。默认情况下内部按并发级别为16来创建。对于每个segment的容量，默认情况也是16。当然并发级别(concurrentLevel)和每个段(segment)的初始容量都是可以通过构造函数设定的。

创建好默认的ConcurrentHashMap之后，它的结构大致如下图：  


看起来只是把以前HashTable的一个hash bucket创建了16份而已。有什么特别的吗？没啥特别的。

继续看每个segment是怎么定义的：

static final class Segment<K,V> extends ReentrantLock implements Serializable

Segment继承了ReentrantLock，表明每个segment都可以当做一个锁。（ReentrantLock前文已经提到，不了解的话就把当做synchronized的替代者吧）这样对每个segment中的数据需要同步操作的话都是使用每个segment容器对象自身的锁来实现。只有对全局需要改变时锁定的是所有的segment。

上面的这种做法，就称之为**“分离锁（lock striping）”**。有必要对**“分拆锁”**和**“分离锁”**的概念描述一下：

**分拆锁(lock spliting)**就是若原先的程序中多处逻辑都采用同一个锁，但各个逻辑之间又相互独立，就可以拆(Spliting)为使用多个锁，每个锁守护不同的逻辑。  
分拆锁有时候可以被扩展，分成可大可小加锁块的集合，并且它们归属于相互独立的对象，这样的情况就是**分离锁(lock striping)**。（摘自《Java并发编程实践》）

看上去，单是这样就已经能大大提高多线程并发的性能了。还没完，继续看我们关注的get,put,remove这三个函数怎么保证数据同步的。

先看get方法：

public V get(Object key) {

int hash = hash(key); // throws NullPointerException if key null

return segmentFor(hash).get(key, hash);

}

它没有使用同步控制，交给segment去找，再看Segment中的get方法：

V get(Object key, int hash) {

if (count != 0) { // read-volatile // ①

HashEntry<K,V> e = getFirst(hash);

while (e != null) {

if (e.hash == hash && key.equals(e.key)) {

V v = e.value;

if (v != null) // ② 注意这里

return v;

return readValueUnderLock(e); // recheck

}

e = e.next;

}

}

return null;

}

它也没有使用锁来同步，只是判断获取的entry的value是否为null，为null时才使用加锁的方式再次去获取。

这个实现很微妙，没有锁同步的话，靠什么保证同步呢？我们一步步分析。

第一步，先判断一下 count != 0；count变量表示segment中存在entry的个数。如果为0就不用找了。  
假设这个时候恰好另一个线程put或者remove了这个segment中的一个entry，会不会导致两个线程看到的count值不一致呢？  
看一下count变量的定义： transient volatile int count;  
它使用了volatile来修改。我们前文说过，Java5之后，JMM实现了对volatile的保证：对volatile域的写入操作happens-before于每一个后续对同一个域的读写操作。  
所以，每次判断count变量的时候，即使恰好其他线程改变了segment也会体现出来。

第二步，获取到要该key所在segment中的索引地址，如果该地址有相同的hash对象，顺着链表一直比较下去找到该entry。当找到entry的时候，先做了一次比较： if(v != null) 我们用红色注释的地方。  
这是为何呢？

考虑一下，如果这个时候，另一个线程恰好新增/删除了entry，或者改变了entry的value，会如何？

先看一下HashEntry类结构。

static final class HashEntry<K,V> {

final K key;

final int hash;

volatile V value;

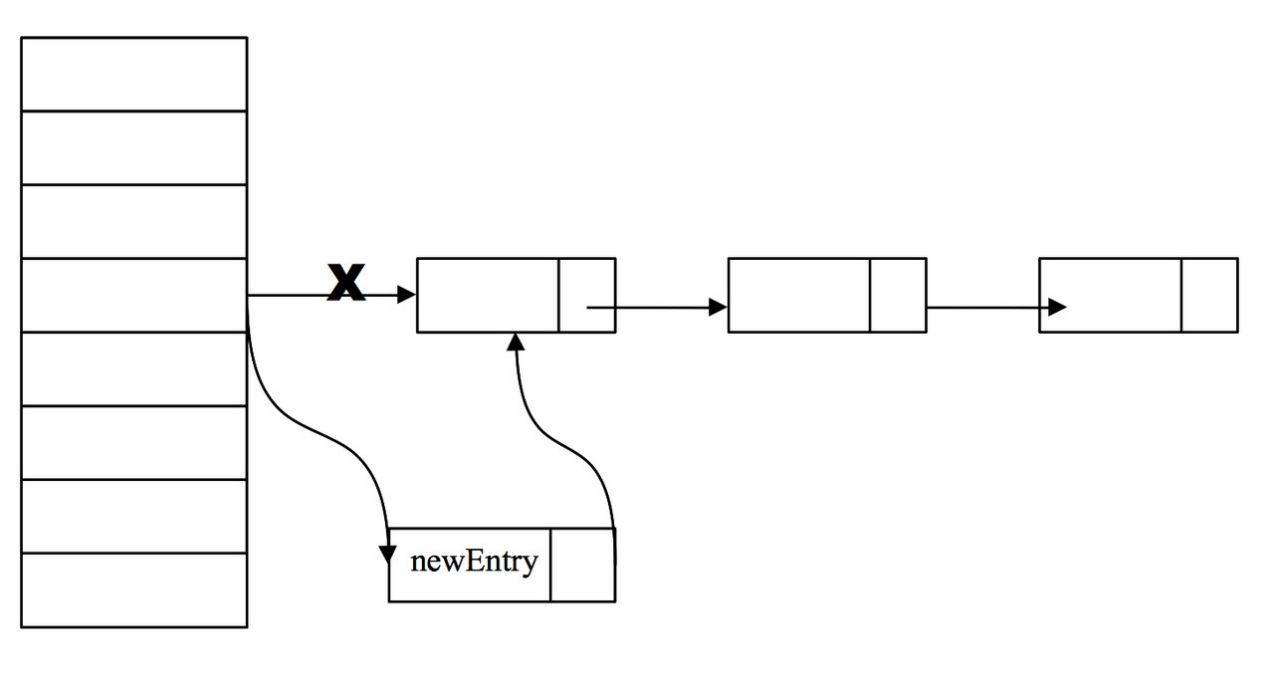
final HashEntry<K,V> next;

。。。

}

除了 value，其它成员都是final修饰的，也就是说value可以被改变，其它都不可以改变，包括指向下一个HashEntry的next也不能被改变。（那删除一个entry时怎么办？后续会讲到。）

1) 在get代码的①和②之间，另一个线程新增了一个entry  
如果另一个线程新增的这个entry又恰好是我们要get的，这事儿就比较微妙了。

下图大致描述了put 一个新的entry的过程。  


因为每个HashEntry中的next也是final的，没法对链表最后一个元素增加一个后续entry  
所以新增一个entry的实现方式只能通过头结点来插入了。

newEntry对象是通过 new HashEntry(K k , V v, HashEntry next) 来创建的。如果另一个线程刚好new 这个对象时，当前线程来get它。因为没有同步，就可能会出现当前线程得到的newEntry对象是一个没有完全构造好的对象引用。

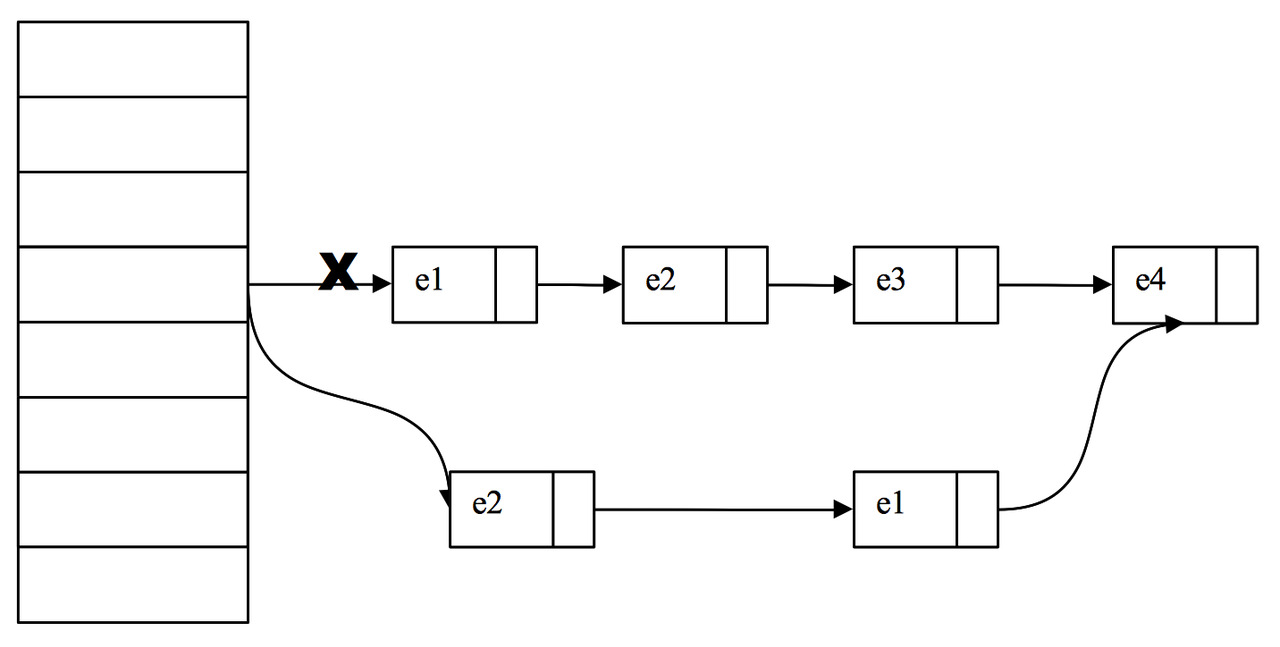
回想一下我们之前讨论的DCL的问题，这里也一样，没有锁同步的话，new 一个对象对于多线程看到这个对象的状态是没有保障的，这里同样有可能一个线程new这个对象的时候还没有执行完构造函数就被另一个线程得到这个对象引用。  
所以才需要判断一下：if (v != null) 如果确实是一个不完整的对象，则使用锁的方式再次get一次。

有没有可能会put进一个value为null的entry？ 不会的，已经做了检查，这种情况会抛出异常，所以 ②处的判断完全是出于对多线程下访问一个new出来的对象的状态检测。

2) 在get代码的①和②之间，另一个线程修改了一个entry的value  
value是用volitale修饰的，可以保证读取时获取到的是修改后的值。

3) 在get代码的①之后，另一个线程删除了一个entry

假设我们的链表元素是：e1-> e2 -> e3 -> e4 我们要删除 e3这个entry  
因为HashEntry中next的不可变，所以我们无法直接把e2的next指向e4，而是将要删除的节点之前的节点复制一份，形成新的链表。它的实现大致如下图所示：



如果我们get的也恰巧是e3，可能我们顺着链表刚找到e1，这时另一个线程就执行了删除e3的操作，而我们线程还会继续沿着旧的链表找到e3返回。  
这里没有办法实时保证了。

我们第①处就判断了count变量，它保障了在 ①处能看到其他线程修改后的。  
①之后到②之间，如果再次发生了其他线程再删除了entry节点，就没法保证看到最新的了。

不过这也没什么关系，即使我们返回e3的时候，它被其他线程删除了，暴漏出去的e3也不会对我们新的链表造成影响。

这其实是一种乐观设计，设计者假设 ①之后到②之间 发生被其它线程增、删、改的操作可能性很小，所以不采用同步设计，而是采用了事后（其它线程这期间也来操作，并且可能发生非安全事件）弥补的方式。  
而因为其他线程的“改”和“删”对我们的数据都不会造成影响，所以只有对“新增”操作进行了安全检查，就是②处的非null检查，如果确认不安全事件发生，则采用加锁的方式再次get。

这样做减少了使用互斥锁对并发性能的影响。可能有人怀疑remove操作中复制链表的方式是否代价太大，这里我没有深入比较，不过既然Java5中这么实现，我想new一个对象的代价应该已经没有早期认为的那么严重。

我们基本分析完了get操作。对于put和remove操作，是使用锁同步来进行的，不过是用的ReentrantLock而不是synchronized，性能上要更高一些。它们的实现前文都已经提到过，就没什么可分析的了。前面的章节中，我们涉及到的操作都是在单个Segment中进行的，但是ConcurrentHashMap有一些操作是在多个Segment中进行，比如size操作，ConcurrentHashMap的size操作也采用了一种比较巧的方式，来尽量避免对所有的Segment都加锁。对于size操作，前面我们提到了一个Segment中的有一个modCount变量，代表的是对Segment中元素的数量造成影响的操作的次数，这个值只增不减，size操作就是遍历了两次Segment，每次记录Segment的modCount值，然后将两次的modCount进行比较，如果相同，则表示期间没有发生过写入操作，就将原先遍历的结果返回，如果不相同，则就需要将所有的Segment都锁住，然后一个一个遍历了，具体的实现大家可以看ConcurrentHashMap的源码，这里就不贴了。

我们还需要知道一点，ConcurrentHashMap的迭代器不是Fast-Fail的方式，所以在迭代的过程中别其他线程添加/删除了元素，不会抛出异常，也不能体现出元素的改动。但也没有关系，因为每个entry的成员除了value都是final修饰的，暴漏出去也不会对其他元素造成影响。

最后，再来看一下Java6文档中对ConcurrentHashMap的描述（我们分析过的地方或者需要注意的使用了红色字体）：

支持获取的完全并发和更新的所期望可调整并发的哈希表。此类遵守与 Hashtable 相同的功能规范，并且包括对应于 Hashtable 的每个方法的方法版本。不过，**尽管所有操作都是线程安全的，但获取操作不 必锁定**，并且不 支持以某种防止所有访问的方式锁定整个表。此类可以通过程序完全与 Hashtable 进行互操作，这取决于其线程安全，而与其同步细节无关。

**获取操作（包括 get）通常不会受阻塞，因此，可能与更新操作交迭（包括 put 和 remove）**。获取会影响最近完成的 更新操作的结果。对于一些聚合操作，比如 putAll 和 clear，并发获取可能只影响某些条目的插入和移除。类似地，在创建迭代器/枚举时或自此之后，Iterators 和 Enumerations 返回在某一时间点上影响哈希表状态的元素。它们不会 抛出 ConcurrentModificationException。不过，迭代器被设计成每次仅由一个线程使用。

这允许通过可选的 concurrencyLevel 构造方法参数（默认值为 16）来引导更新操作之间的并发，该参数用作内部调整大小的一个提示。表是在内部进行分区的，试图允许指示无争用并发更新的数量。因为哈希表中的位置基本上是随意的，所以实际的并发将各不相同。理想情况下，应该选择一个尽可能多地容纳并发修改该表的线程的值。使用一个比所需要的值高很多的值可能会浪费空间和时间，而使用一个显然低很多的值可能导致线程争用。**对数量级估计过高或估计过低通常都会带来非常显著的影响。当仅有一个线程将执行修改操作，而其他所有线程都只是执行读取操作时，才认为某个值是合适的**。此外，重新调整此类或其他任何种类哈希表的大小都是一个相对较慢的操作，因此，在可能的时候，提供构造方法中期望表大小的估计值是一个好主意。

参考：  
http://www.javaeye.com/topic/344876

本来我的分析已经结束，不过正好看到Concurrency-interest邮件组中的一个问题，可以加深一下我们队ConcurrentHashMap的理解，同时也反驳了我刚开始所说的“ConcurrentHashMap完全可以替代HashTable”，我必须纠正一下。先看例子：

ConcurrentHashMap<String, Boolean> map = new ...;

Thread a = new Thread {

void run() {

map.put("first", true);

map.put("second", true);

}

};

Thread b = new Thread {

void run() {

map.clear();

}

};

a.start();

b.start();

a.join();

b.join();

I would expect that one of the following scenarios to be true (for the contents of the map) after this code runs:

Map("first" -> true, "second" -> true)

Map("second" -> true)

Map()

However, upon inspection of ConcurrentHashMap, it seems to me that the following scenario might also be true:

Map("first" -> true) ???

This seems surprising because “first” gets put before “second”, so if “second” is cleared, then surely “first” should be cleared too.

Likewise, consider the following code:

ConcurrentHashMap<String, Boolean> map = new ...;

List<String> myKeys = new ...;

Thread a = new Thread {

void run() {

map.put("first", true);

// more stuff

map.remove("first");

map.put("second", true);

}

};

Thread b = new Thread {

void run() {

Set<String> keys = map.keySet();

for (String key : keys) {

myKeys.add(key);

}

}

};

a.start();

b.start();

a.join();

b.join();

I would expect one of the following scenarios to be true for the contents of myKeys after this code has run:

List()

List("first")

List("second")

However, upon closer inspection, it seems that the following scenario might also be true:

List("first", "second") ???

This is surprising because “second” is only ever put into the map after “first” is removed. They should never be in the map simultaneously, but an iterator might perceive them to be so.

对于这两个现象的解释：ConcurrentHashMap中的clear方法：

public void clear() {

for (int i = 0; i < segments.length; ++i)

segments[i].clear();

}

如果线程b先执行了clear，清空了一部分segment的时候，线程a执行了put且正好把“first”放入了“清空过”的segment中，而把“second”放到了还没有清空过的segment中，就会出现上面的情况。

第二段代码，如果线程b执行了迭代遍历到first，而此时线程a还没有remove掉first，那么即使后续删除了first，迭代器里不会反应出来，也不抛出异常，这种迭代器被称为“弱一致性”(weakly consistent)迭代器。

*Doug Lea* 对这个问题的回复中提到：

We leave the tradeoff of consistency-strength versus scalability  
as a user decision, so offer both synchronized and concurrent versions  
of most collections, as discussed in the j.u.c package docs  
http://java.sun.com/javase/6/docs/api/java/util/concurrent/package-summary.html

大意是我们将“一致性强度”和“扩展性”之间的对比交给用户来权衡，所以大多数集合都提供了synchronized和concurrent两个版本。

通过他的说法，我必须纠正我一开始以为ConcurrentHashMap完全可以代替HashTable的说法，如果你的环境要求“强一致性”的话，就不能用ConcurrentHashMap了，它的get，clear方法和迭代器都是“弱一致性”的。不过真正需要“强一致性”的场景可能非常少，我们大多应用中ConcurrentHashMap是满足的。