****ConcurrentHashMap****是Java5中新增加的一个线程安全的Map集合，可以用来替代****HashTable****。对于ConcurrentHashMap是如何提高其效率的，可能大多人只是知道它使用了多个锁代替HashTable中的单个锁，也就是****锁分离技术（Lock Stripping）****。实际上，ConcurrentHashMap对提高并发方面的优化，还有一些其它的技巧在里面（比如你是否知道在get操作的时候，它是否也使用了锁来保护？）。

## ConcurrentMap

*提供其他原子 putIfAbsent、remove、replace 方法的 Map。 内存一致性效果：当存在其他并发 collection 时，将对象放入 ConcurrentMap 之前的线程中的操作 happen-before 随后通过另一线程从 ConcurrentMap 中访问或移除该元素的操作。*

*我们不关心ConcurrentMap中新增的接口，重点理解一下内存一致性效果中的******“happens-before”******是怎么回事。因为要想从根本上讲明白，这个是无法避开的。这又不得不从Java存储模型来谈起了。*

## 理解JAVA存储模型(JMM)的Happens-Before规则

在解释该规则之前，我们先看一段多线程访问数据的代码例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | **public** **class** Test1 {  **private** **int** a=1, b=2;    **public** **void** foo(){  // 线程1          a=3;          b=4;      }    **public** **int** getA(){ // 线程2  **return** a;      }  **public** **int** getB(){ // 线程2  **return** b;      }  } |

上面的代码，当线程1执行foo方法的时候，线程2访问getA和getB会得到什么样的结果？  
答案：

|  |  |
| --- | --- |
| 1  2  3  4 | A：a=1, b=2  // 都未改变  B：a=3, b=4  // 都改变了  C：a=3, b=2  //  a改变了，b未改变  D：a=1, b=4  //  b改变了，a未改变 |

上面的A,B,C都好理解，但是D可能会出乎一些人的预料。  
一些不了解JMM的同学可能会问怎么可能 b=4语句会先于 a=3 执行？

这是一个多线程之间内存可见性（Visibility）顺序不一致的问题。有两种可能会造成上面的D选项。

****1) Java编译器的重排序(Reording)操作有可能导致执行顺序和代码顺序不一致。****  
关于Reording：

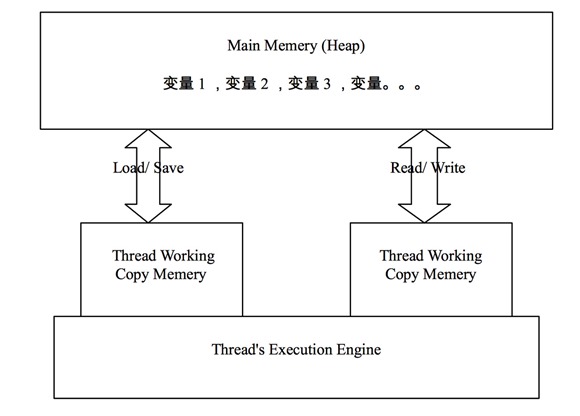
*Java语言规范规定了JVM要维护内部线程类似顺序化语义(within-thread as-is-serial semantics)：只要程序的最终结果等同于它在严格的顺序化环境中执行的结果，那么上述所有的行为都是允许的。*

上面的话是《Java并发编程实践》一书中引自Java语言规范的，感觉翻译的不太好。简单的说：假设代码有两条语句，代码顺序是语句1先于语句2执行；那么只要语句2不依赖于语句1的结果，打乱它们的顺序对最终的结果没有影响的话，那么真正交给CPU去执行时，他们的顺序可以是没有限制的。可以允许语句2先于语句1被CPU执行，和代码中的顺序不一致。

重排序（Reordering）是JVM针对现代CPU的一种优化，Reordering后的指令会在性能上有很大提升。(不知道这种优化对于多核CPU是否更加明显，也或许和单核多核没有关系。)

因为我们例子中的两条赋值语句，并没有依赖关系，无论谁先谁后结果都是一样的，所以就可能有Reordering的情况，这种情况下，对于其他线程来说就可能造成了可见性顺序不一致的问题。

****2) 从线程工作内存写回主存时顺序无法保证。****  
下图描述了JVM中主存和线程工作内存之间的交互：

[](http://images.cnitblog.com/blog/587773/201409/062059128755645.jpg" \o "1" \t "http://www.importnew.com/_blank)

JLS中对线程和主存互操作定义了6个行为，分别为load，save，read，write，assign和use，这些操作行为具有原子性，且相互依赖，有明确的调用先后顺序。这个细节也比较繁琐，我们暂不深入追究。先简单认为线程在修改一个变量时，先拷贝入线程工作内存中，在线程工作内存修改后再写回主存(Main Memery)中。

假设例子中Reording后顺序仍与代码中的顺序一致，那么接下来呢？有意思的事情就发生在线程把Working Copy Memery中的变量写回Main Memery的时刻。线程1把变量写回Main Memery的过程对线程2的可见性顺序也是无法保证的。  
上面的列子，a=3; b=4; 这两个语句在 Working Copy Memery中执行后，写回主存的过程对于线程2来说同样可能出现先b=4；后a=3；这样的相反顺序。

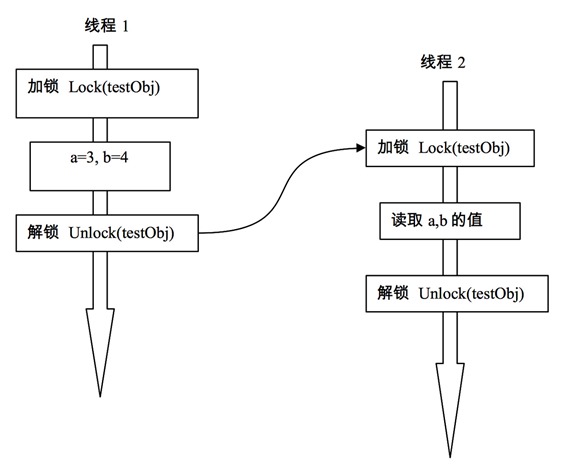
正因为上面的那些问题，JMM中一个重要问题就是：如何让多线程之间，对象的状态对于各线程的“可视性”是顺序一致的。它的解决方式就是 Happens-before 规则：  
JMM为所有程序内部动作定义了一个偏序关系，叫做happens-before。要想保证执行动作B的线程看到动作A的结果（无论A和B是否发生在同一个线程中），A和B之间就必须满足happens-before关系。

我们现在来看一下“Happens-before”规则都有哪些（摘自《Java并发编程实践》）：

*① 程序次序法则：线程中的每个动作A都happens-before于该线程中的每一个动作B，其中，在程序中，所有的动作B都能出现在A之后。  
② 监视器锁法则：对一个监视器锁的解锁 happens-before于每一个后续对同一监视器锁的加锁。 ③ volatile变量法则：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。  
④ 线程启动法则：在一个线程里，对Thread.start的调用会happens-before于每个启动线程的动作。  
⑤ 线程终结法则：线程中的任何动作都happens-before于其他线程检测到这个线程已经终结、或者从Thread.join调用中成功返回，或Thread.isAlive返回false。  
⑥ 中断法则：一个线程调用另一个线程的interrupt happens-before于被中断的线程发现中断。  
⑦ 终结法则：一个对象的构造函数的结束happens-before于这个对象finalizer的开始。  
⑧ 传递性：如果A happens-before于B，且B happens-before于C，则A happens-before于C*

我们重点关注的是②，③，这两条也是我们通常编程中常用的。  
后续分析ConcurrenHashMap时也会看到使用到锁(ReentrantLock)，Volatile，final等手段来保证happens-before规则的。

使用锁方式实现“Happens-before”是最简单，容易理解的。



早期Java中的锁只有最基本的synchronized，它是一种互斥的实现方式。在Java5之后，增加了一些其它锁，比如ReentrantLock，它基本作用和synchronized相似，但提供了更多的操作方式，比如在获取锁时不必像synchronized那样只是傻等，可以设置定时，轮询，或者中断，这些方法使得它在获取多个锁的情况可以避免死锁操作。

而我们需要了解的是ReentrantLock的性能相对synchronized来说有很大的提高。（不过据说Java6后对synchronized进行了优化，两者已经接近了。）在ConcurrentHashMap中，每个hash区间使用的锁正是ReentrantLock。

Volatile可以看做一种轻量级的锁，但又和锁有些不同。  
a) 它对于多线程，不是一种互斥（mutex）关系。  
b) 用volatile修饰的变量，不能保证该变量状态的改变对于其他线程来说是一种“原子化操作”。

在Java5之前，JMM对Volatile的定义是：保证读写volatile都直接发生在main memory中，线程的working memory不进行缓存。它只承诺了读和写过程的可见性，并没有对Reording做限制，所以旧的Volatile并不太可靠。在Java5之后，JMM对volatile的语义进行了增强。就是我们看到的③ volatile变量法则。

那对于“原子化操作”怎么理解呢？看下面例子：

|  |  |
| --- | --- |
| 1  2  3  4  5 | **private** **static** **volatile** **int** nextSerialNum = 0;    **public** **static** **int** generateSerialNumber(){  **return** nextSerialNum++;  } |

上面代码中对nextSerialNum使用了volatile来修饰，根据前面“Happens-Before”法则的第三条Volatile变量法则，看似不同线程都会得到一个新的serialNumber

问题出在了 ****nextSerialNum++**** 这条语句上，它不是一个原子化的，实际上是****read-modify-write****三项操作，这就有可能使得在线程1在write之前，线程2也访问到了nextSerialNum，造成了线程1和线程2得到一样的serialNumber。  
所以，在使用Volatile时，需要注意  
a)  需不需要互斥；  
b) 对象状态的改变是不是原子化的。

最后也说一下final 关键字。

****不变模式（immutable）****是多线程安全里最简单的一种保障方式。因为你拿他没有办法，想改变它也没有机会。  
不变模式主要通过final关键字来限定的。在JMM中final关键字还有特殊的语义。Final域使得确保初始化安全性（initialization safety）成为可能，初始化安全性让不可变形对象不需要同步就能自由地被访问和共享。

****3）经过前面的了解，下面我们用Happens-Before规则理解一个经典问题：双重检测锁(DCL)为什么在java中不适用？****

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | **public** **class** LazySingleton {  **private** **int** someField;  **private** **static** LazySingleton instance;    **private** LazySingleton(){  **this**.someField = **new** Random().nextInt(200) + 1; // (1)      }    **public** **static** LazySingleton getInstance() {  **if** (instance == **null**) {// (2)  **synchronized** (LazySingleton.**class**) { // (3)  **if** (instance == **null**) { // (4)                  instance = **new** LazySingleton(); // (5)                }              }          }  **return** instance; // (6)      }    **public** **int** getSomeField() {  **return** **this**.someField;  // (7)      }  } |

我想简单的用对象创建期间的实际场景来分析一下：（注意，这种场景是我个人的理解，所看的资料也是非官方的，不完全保证正确。如果发现不对请指出。

假设线程1执行完(5)时，线程2正好执行到了(2)；  
看看 new LazySingleton(); 这个语句的执行过程： 它不是一个原子操作，实际是由多个步骤，我们从我们关注的角度简化一下，简单的认为它主要有2步操作好了：  
a） 在内存中分配空间，并将引用指向该内存空间。  
b） 执行对象的初始化的逻辑(和操作)，完成对象的构建。

此时因为线程1和线程2没有用同步，他们之间不存在“Happens-Before”规则的约束，所以在线程1创建LazySingleton对象的 a),b)这两个步骤对于线程2来说会有可能出现a)可见，b)不可见  
造成了线程2获取到了一个未创建完整的lazySingleton对象引用，为后边埋下隐患。

之所以这里举到 DCL这个例子，是因为我们后边分析ConcurrentHashMap时，也会遇到相似的情况。  
对于对象的创建，出于乐观考虑，两个线程之间没有用“Happens-Before规则来约束”另一个线程可能会得到一个未创建完整的对象，这种情况必须要检测，后续分析ConcurrentHashMap时再讨论。

**单例模式：双重检测锁(DCL)为什么在java中不适用？**

前面的双重检查锁定示例代码的第7行（instance = new Singleton();）创建一个对象。这一行代码可以分解为如下的三行伪代码：

memory = allocate(); //1：分配对象的内存空间

ctorInstance(memory); //2：初始化对象

instance = memory; //3：设置instance指向刚分配的内存地址

上面三行伪代码中的2和3之间，可能会被重排序（在一些JIT编译器上，这种重排序是真实发生的，详情见参考文献1的“Out-of-order writes”部分）。2和3之间重排序之后的执行时序如下：

memory = allocate(); //1：分配对象的内存空间

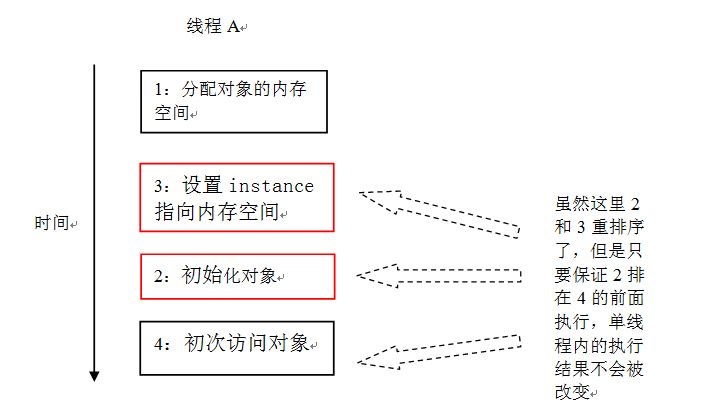
instance = memory; //3：设置instance指向刚分配的内存地址

//注意，此时对象还没有被初始化！

ctorInstance(memory); //2：初始化对象

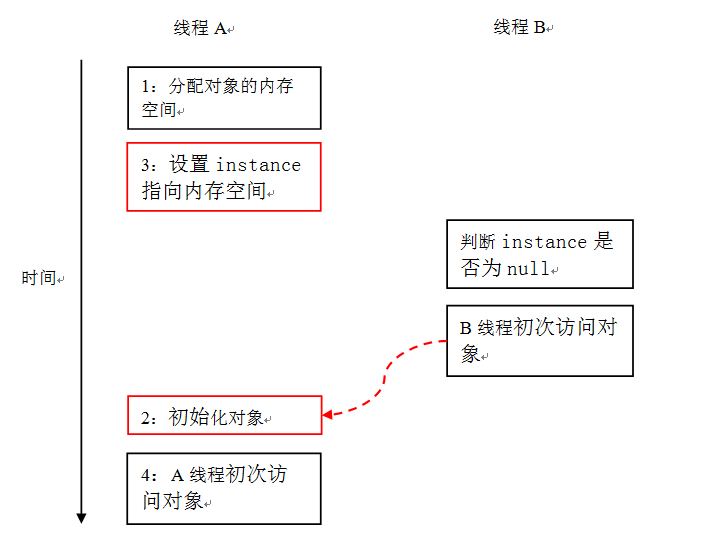
根据《The Java Language Specification, Java SE 7 Edition》（后文简称为java语言规范），所有线程在执行java程序时必须要遵守intra-thread semantics。intra-thread semantics保证重排序不会改变单线程内的程序执行结果。换句话来说，intra-thread semantics允许那些在单线程内，不会改变单线程程序执行结果的重排序。上面三行伪代码的2和3之间虽然被重排序了，但这个重排序并不会违反intra-thread semantics。这个重排序在没有改变单线程程序的执行结果的前提下，可以提高程序的执行性能。

为了更好的理解intra-thread semantics，请看下面的示意图（假设一个线程A在构造对象后，立即访问这个对象）：



如上图所示，只要保证2排在4的前面，即使2和3之间重排序了，也不会违反intra-thread semantics。

下面，再让我们看看多线程并发执行的时候的情况。请看下面的示意图：



由于单线程内要遵守intra-thread semantics，从而能保证A线程的程序执行结果不会被改变。但是当线程A和B按上图的时序执行时，B线程将看到一个还没有被初始化的对象。

※注：本文统一用红色的虚箭线标识错误的读操作，用绿色的虚箭线标识正确的读操作。

回到本文的主题，DoubleCheckedLocking示例代码的第7行（instance = new Singleton();）如果发生重排序，另一个并发执行的线程B就有可能在第4行判断instance不为null。线程B接下来将访问instance所引用的对象，但此时这个对象可能还没有被A线程初始化！下面是这个场景的具体执行时序：

|  |  |  |
| --- | --- | --- |
| 时间 | 线程A | 线程B |
| t1 | A1：分配对象的内存空间 |  |
| t2 | A3：设置instance指向内存空间 |  |
| t3 |  | B1：判断instance是否为空 |
| t4 |  | B2：由于instance不为null，线程B将访问instance引用的对象 |
| t5 | A2：初始化对象 |  |
| t6 | A4：访问instance引用的对象 |  |

这里A2和A3虽然重排序了，但java内存模型的intra-thread semantics将确保A2一定会排在A4前面执行。因此线程A的intra-thread semantics没有改变。但A2和A3的重排序，将导致线程B在B1处判断出instance不为空，线程B接下来将访问instance引用的对象。此时，线程B将会访问到一个还未初始化的对象。

在知晓了问题发生的根源之后，我们可以想出两个办法来实现线程安全的延迟初始化：

1. 不允许2和3重排序；
2. 允许2和3重排序，但不允许其他线程“看到”这个重排序。

后文介绍的两个解决方案，分别对应于上面这两点。1.volatile,static class

**JDK1.6分析**

**ConcurrentHashMap采用 分段锁的机制，实现并发的更新操作，底层采用数组+链表+红黑树的存储结构。**

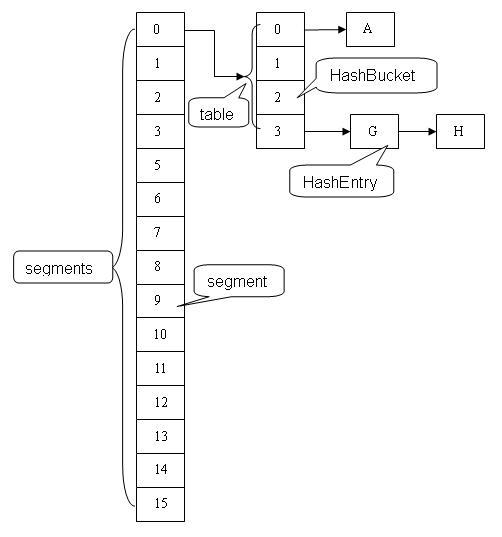
**其包含两个核心静态内部类 Segment和HashEntry。**

Segment继承ReentrantLock用来充当锁的角色，每个 Segment 对象守护每个散列映射表的若干个桶。

HashEntry 用来封装映射表的键 / 值对；

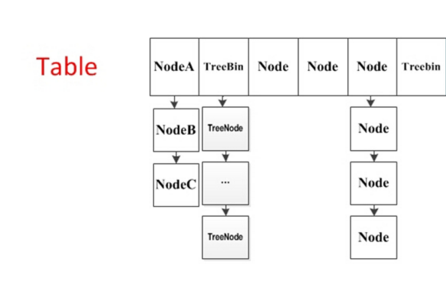
每个桶是由若干个 HashEntry 对象链接起来的链表。

一个 ConcurrentHashMap 实例中包含由若干个 Segment 对象组成的数组，下面我们通过一个图来演示一下 ConcurrentHashMap 的结构：



JDK1.8分析

1.8的实现已经抛弃了Segment分段锁机制，**利用CAS+Synchronized来保证并发更新的安全，底层依然采用数组+链表+红黑树的存储结构。**



**重要概念**

在开始之前，有些重要的概念需要介绍一下：

**table**：默认为null，初始化发生在第一次插入操作，默认大小为16的数组，用来存储Node节点数据，扩容时大小总是2的幂次方。

**nextTable**：默认为null，扩容时新生成的数组，其大小为原数组的两倍。

**sizeCtl** ：默认为0，用来控制table的初始化和扩容操作，具体应用在后续会体现出来。

-1 代表table正在初始化

-N 表示有N-1个线程正在进行扩容操作

其余情况：

1、如果table未初始化，表示table需要初始化的大小。

2、如果table初始化完成，表示table的容量，默认是table大小的0.75倍，居然用这个公式算0.75（n - (n >>> 2)）。

**Node**：保存key，value及key的hash值的数据结构。

class Node<K,V> implements Map.Entry<K,V> {

final int hash;

final K key;

volatile V val;

volatile Node<K,V> next;

... 省略部分代码

}

其中value和next都用volatile修饰，保证并发的可见性。

**ForwardingNode**：一个特殊的Node节点，hash值为-1，其中存储nextTable的引用。

final class ForwardingNode<K,V> extends Node<K,V> {

final Node<K,V>[] nextTable;

ForwardingNode(Node<K,V>[] tab) {

super(MOVED, null, null, null);

this.nextTable = tab;

}

}

**只有table发生扩容的时候，ForwardingNode才会发挥作用**，作为一个占位符放在table中表示当前节点为null或则已经被移动。