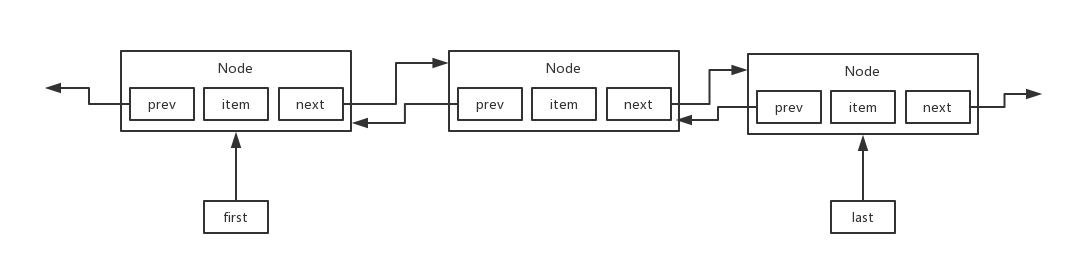
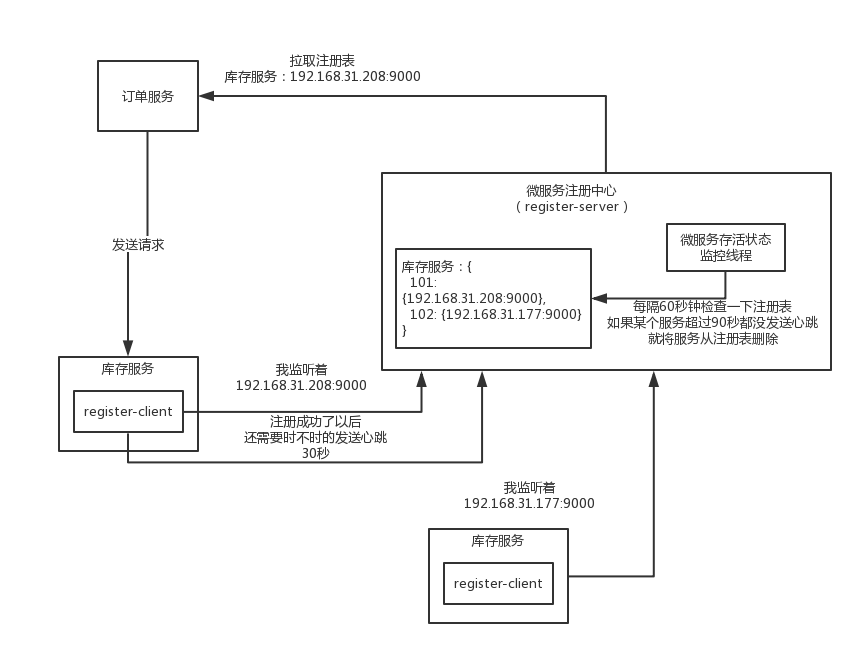
# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/06\_LinkedList源码剖析（二）：双向链表数据结构/01\_LinkedList数据结构.png**



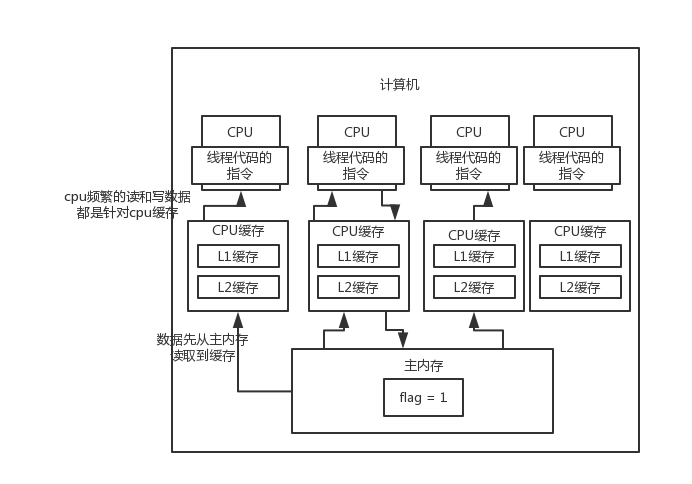
# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~17资料/06\_案例实战：微服务存活状态监控线程的实现/01\_微服务注册中心案例背景(1).png**



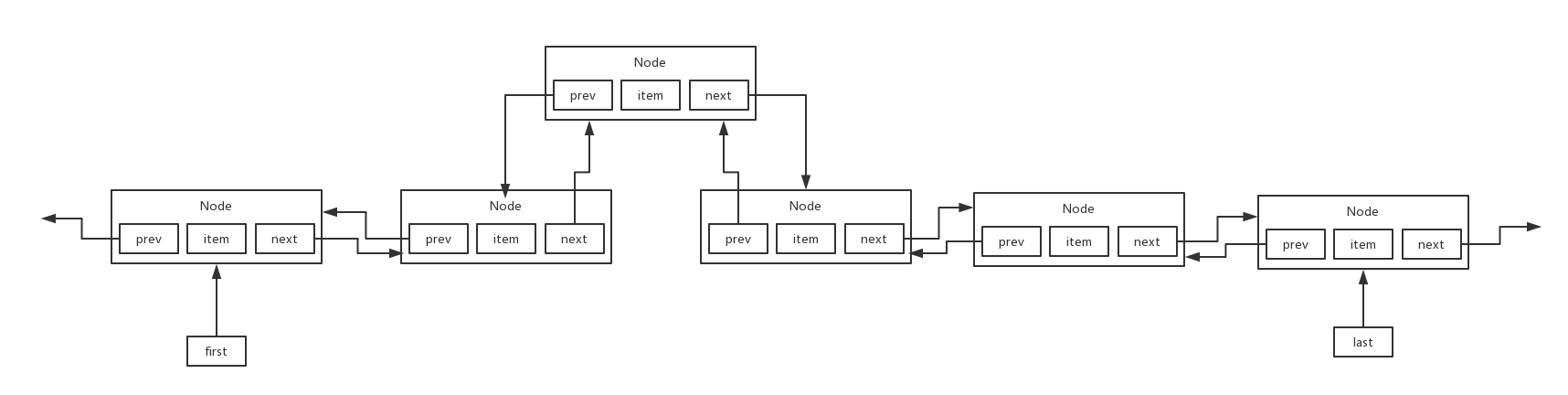
# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~17资料/02\_案例引入：微服务注册中心系列案例背景引入/01\_微服务注册中心案例背景.png**



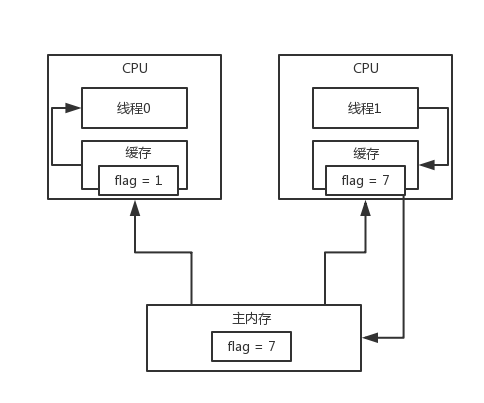
# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/18~30资料/19\_图解主内存以及cpu的多级缓存模型的实现原理/02\_cpu缓存模型.png**



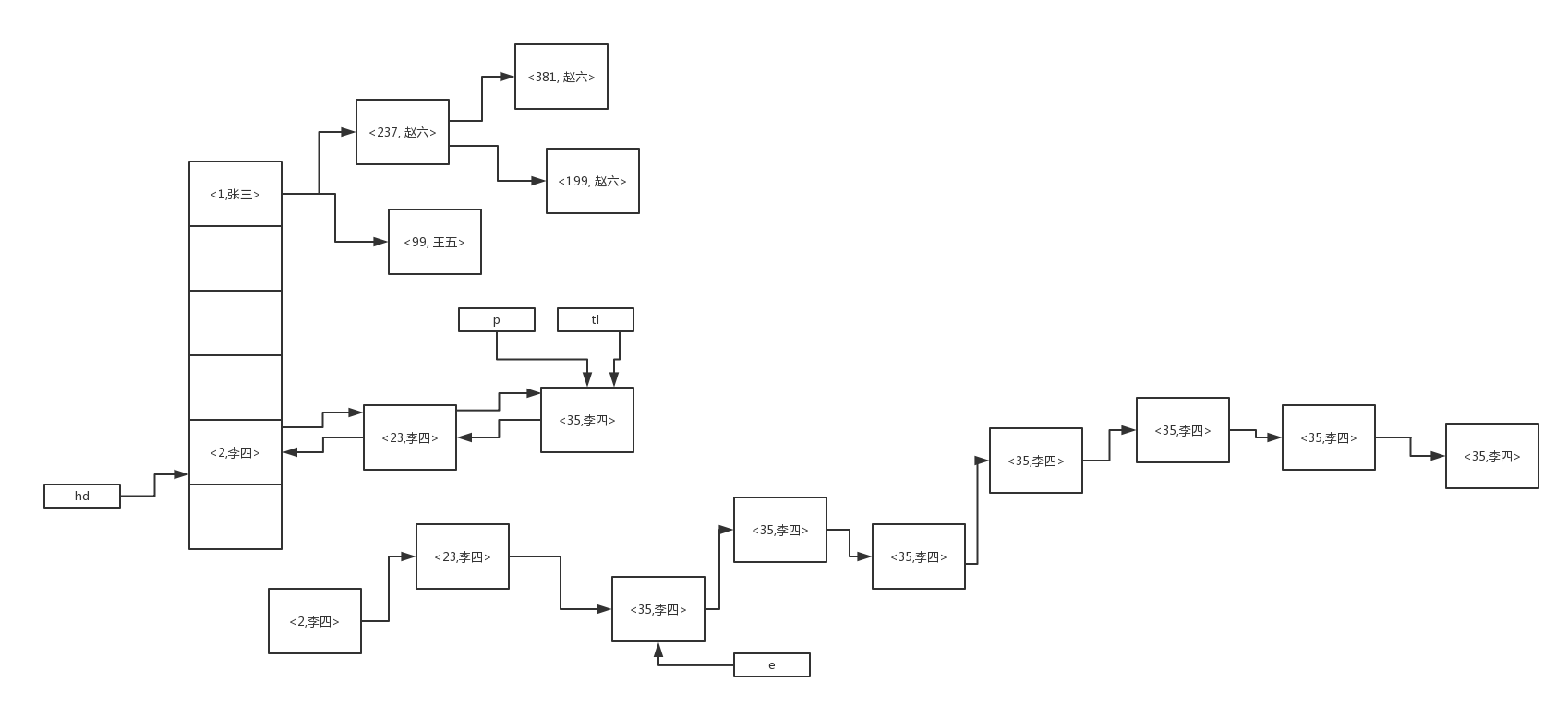
# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/07\_LinkedList源码剖析（三）：插入元素的原理/02\_LinkedList插入元素.png**



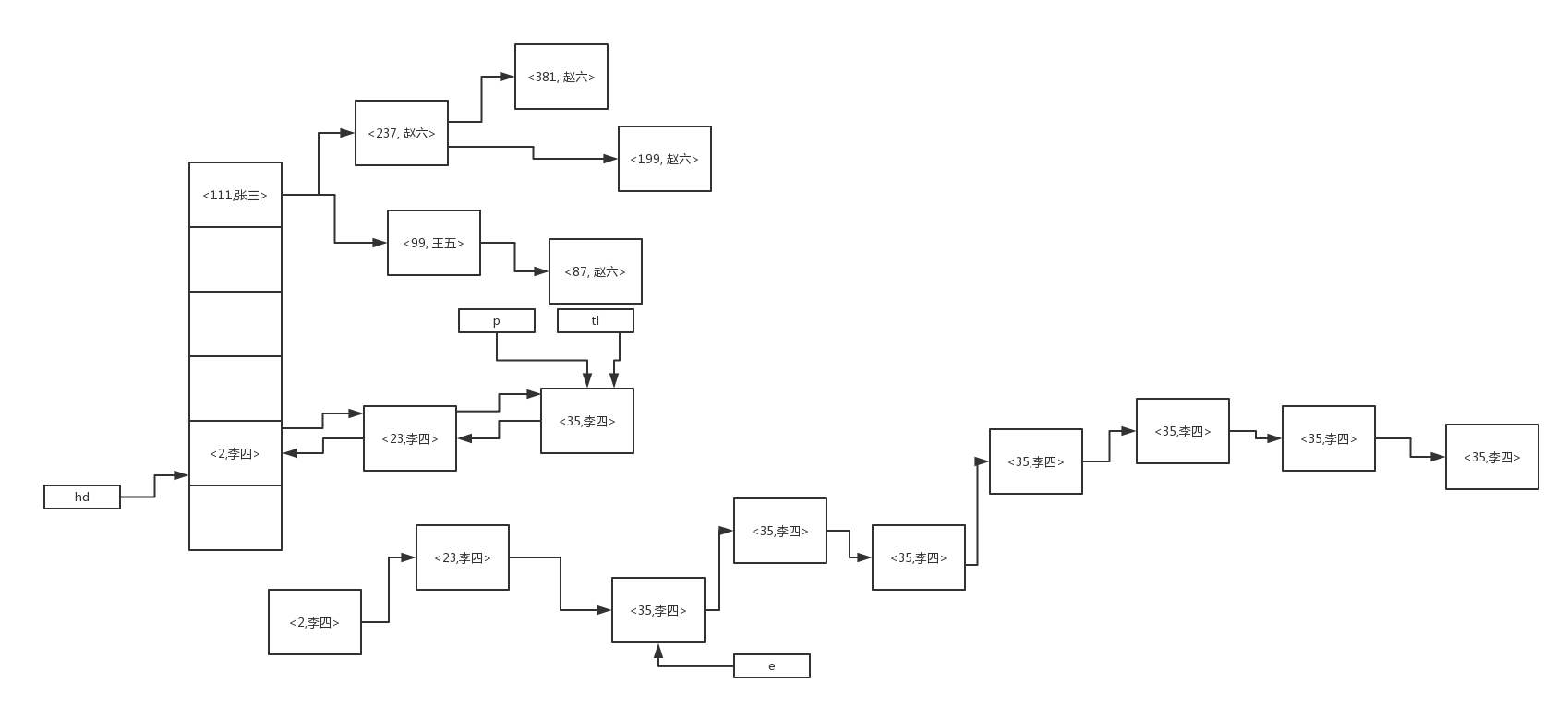
# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/18~30资料/20\_多线程并发运行时可能引发的数据不一致问题/03\_cpu缓存模型下的并发问题.png**



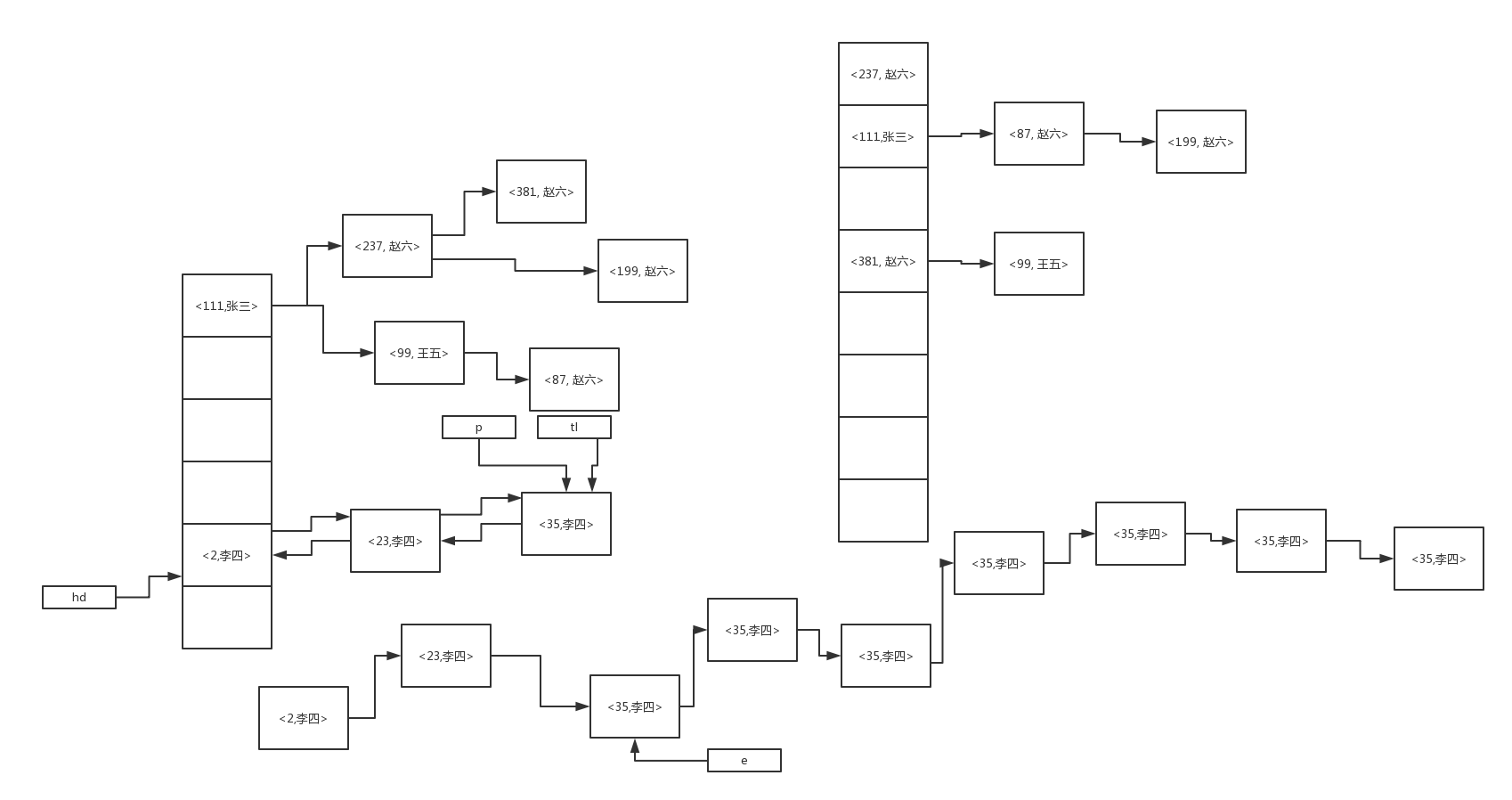
# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/17\_HashMap源码剖析（六）：JDK 1.8引入红黑树优化hash冲突/03\_hashmap数据结构(1).png**



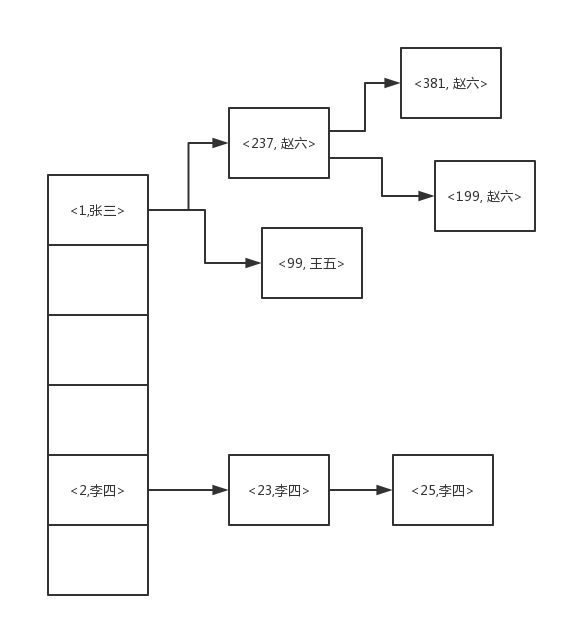
# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/18\_HashMap源码剖析（七）：通过红黑树来解决hash冲突/03\_hashmap数据结构(2).png**



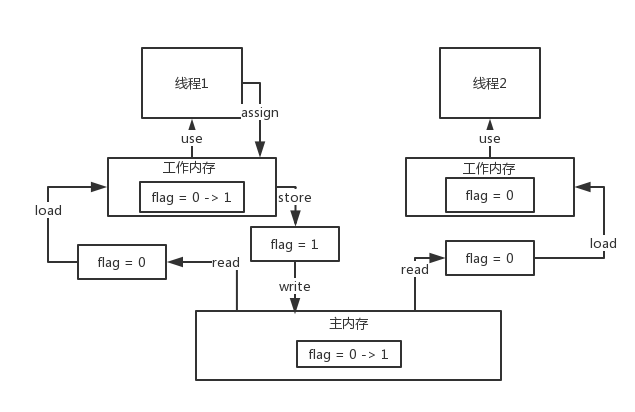
# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/19\_HashMap源码剖析（八）：基于数组的扩容原理图解/03\_hashmap数据结构(3).png**



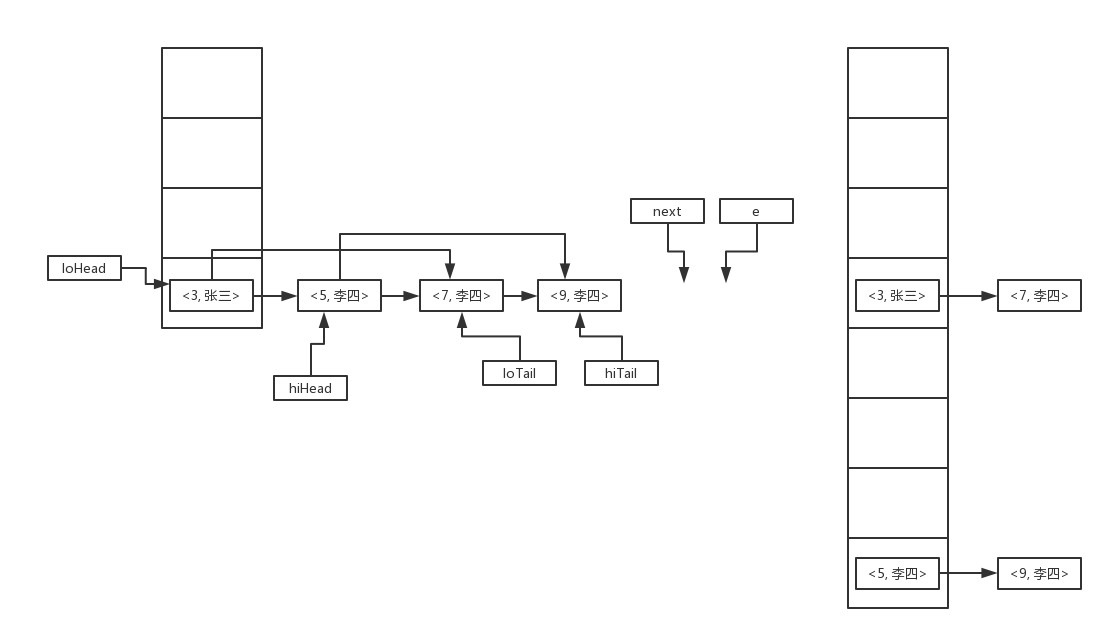
# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/12\_HashMap源码剖析（一）：数组+链表+红黑树的数据结构/03\_hashmap数据结构.png**



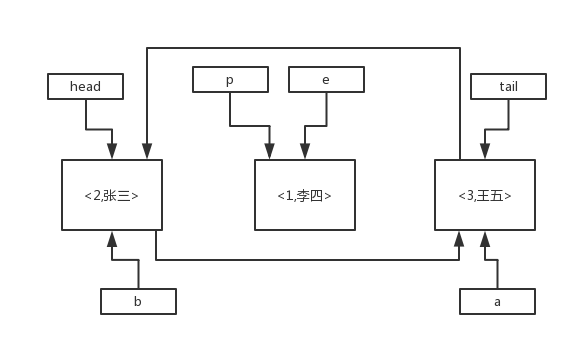
# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/18~30资料/22\_图解Java内存模型以及多线程并发问题的发生/03\_java内存模型.png**



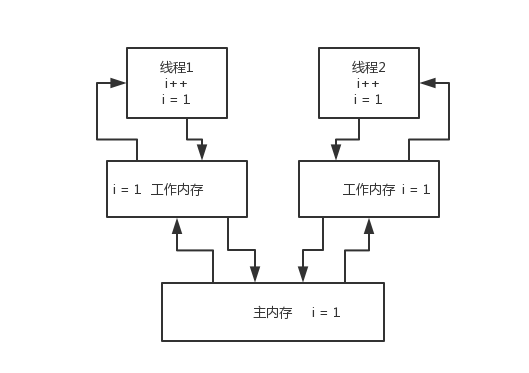
# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/20\_HashMap源码剖析（九）：JDK 1.8的高性能rehash算法/04\_hashmap的resize原理.png**



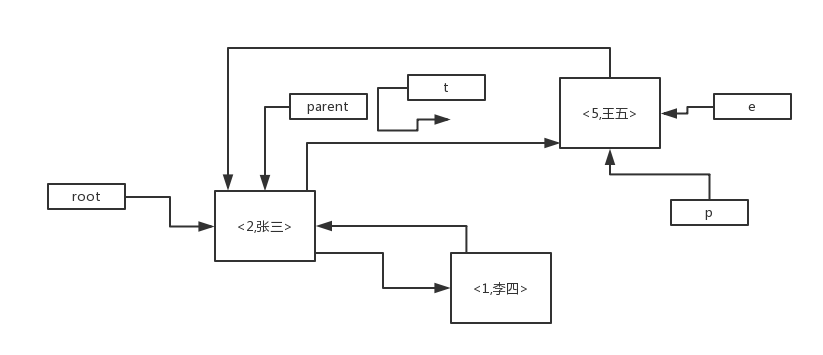
# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/22\_LinkedHashMap源码剖析：有顺序的map数据结构/05\_LinkedHashMap工作原理.png**



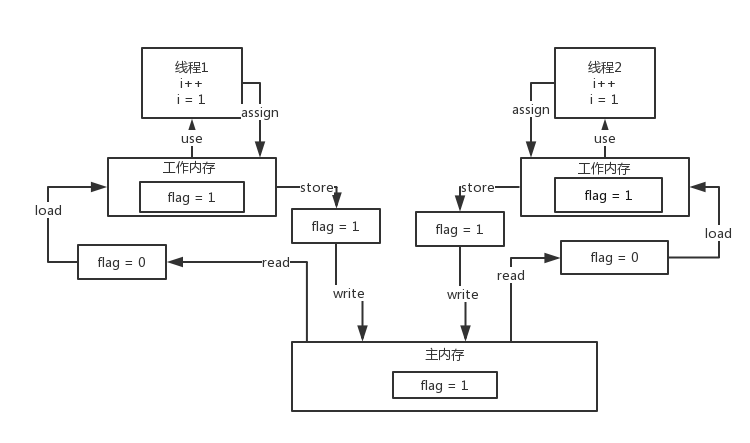
# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/18~30资料/23\_并发编程中的三大特性：可见性、原子性、有序性/05\_原子性问题.png**



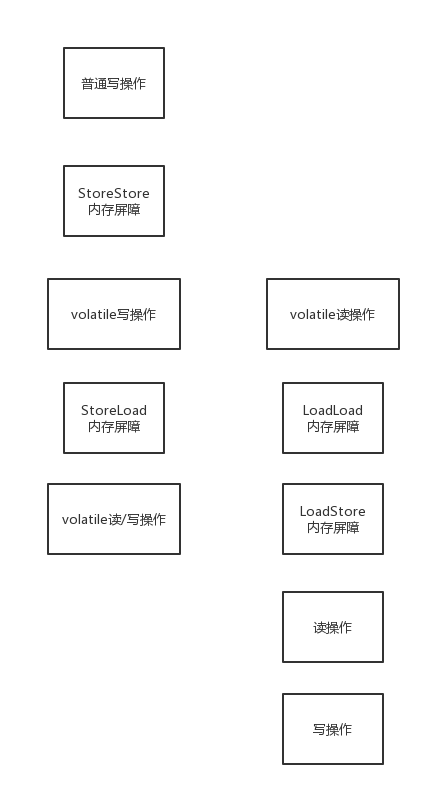
# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/23\_TreeMap源码剖析：自定义排序规则的红黑树map数据结构/06\_TreeMap工作原理(1).png**



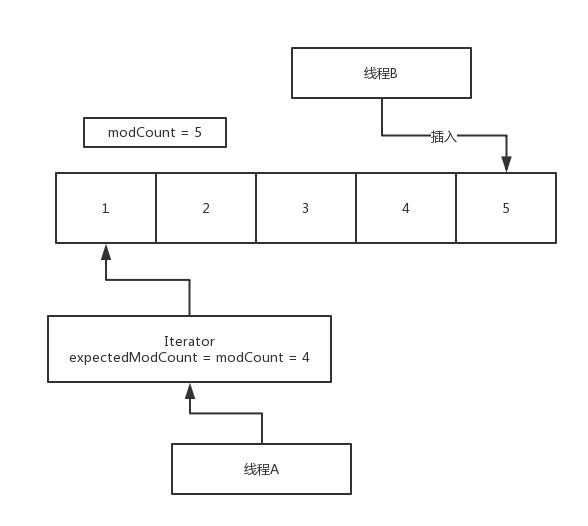
# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/18~30资料/25\_几乎没人解释的清楚：volatile为什么无法保证原子性？/06\_volatile无法保证原子性.png**



# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/18~30资料/27\_volatile的底层实现原理：lock指令以及内存屏障/07\_volatile和内存屏障.png**



# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/25\_Iterator迭代器应对多线程并发修改的fail fast机制/07\_迭代器的fail fast机制.png**



# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/125\_生产环境的锁故障：死锁现象演示以及jstack分析死锁问题/dump.txt**

2018-12-08 17:55:52

Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.151-b12 mixed mode):

"Thread-1" #11 daemon prio=5 os\_prio=0 tid=0x0000000026dc6000 nid=0x8a6c waiting on condition [0x00000000279ce000]

java.lang.Thread.State: WAITING (parking)

at sun.misc.Unsafe.park(Native Method)

- parking to wait for <0x0000000715e60e10> (a java.util.concurrent.locks.ReentrantReadWriteLock$NonfairSync)

at java.util.concurrent.locks.LockSupport.park(LockSupport.java:175)

at java.util.concurrent.locks.AbstractQueuedSynchronizer.parkAndCheckInterrupt(AbstractQueuedSynchronizer.java:836)

at java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireQueued(AbstractQueuedSynchronizer.java:870)

at java.util.concurrent.locks.AbstractQueuedSynchronizer.acquire(AbstractQueuedSynchronizer.java:1199)

at java.util.concurrent.locks.ReentrantReadWriteLock$WriteLock.lock(ReentrantReadWriteLock.java:943)

at com.zhss.demo.register.server.ServiceRegistryCache$CacheMapSyncDaemon.run(ServiceRegistryCache.java:176)

- locked <0x0000000715e60de8> (a java.lang.Object)

"Thread-0" #10 daemon prio=5 os\_prio=0 tid=0x0000000026d94000 nid=0x9ea4 waiting on condition [0x00000000278cf000]

java.lang.Thread.State: TIMED\_WAITING (sleeping)

at java.lang.Thread.sleep(Native Method)

at com.zhss.demo.register.server.ServiceRegistry$RecentlyChangedQueueMonitor.run(ServiceRegistry.java:279)

"Service Thread" #9 daemon prio=9 os\_prio=0 tid=0x0000000026d17000 nid=0x97d0 runnable [0x0000000000000000]

java.lang.Thread.State: RUNNABLE

"C1 CompilerThread2" #8 daemon prio=9 os\_prio=2 tid=0x0000000026cf1000 nid=0x8adc waiting on condition [0x0000000000000000]

java.lang.Thread.State: RUNNABLE

"C2 CompilerThread1" #7 daemon prio=9 os\_prio=2 tid=0x0000000026cdc800 nid=0x9c94 waiting on condition [0x0000000000000000]

java.lang.Thread.State: RUNNABLE

"C2 CompilerThread0" #6 daemon prio=9 os\_prio=2 tid=0x0000000026cd9000 nid=0x267c waiting on condition [0x0000000000000000]

java.lang.Thread.State: RUNNABLE

"Attach Listener" #5 daemon prio=5 os\_prio=2 tid=0x000000002592e000 nid=0x9994 waiting on condition [0x0000000000000000]

java.lang.Thread.State: RUNNABLE

"Signal Dispatcher" #4 daemon prio=9 os\_prio=2 tid=0x0000000026cd8800 nid=0x9dbc runnable [0x0000000000000000]

java.lang.Thread.State: RUNNABLE

"Finalizer" #3 daemon prio=8 os\_prio=1 tid=0x000000002590a800 nid=0x8710 in Object.wait() [0x0000000026c6f000]

java.lang.Thread.State: WAITING (on object monitor)

at java.lang.Object.wait(Native Method)

- waiting on <0x0000000715d88ec8> (a java.lang.ref.ReferenceQueue$Lock)

at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:143)

- locked <0x0000000715d88ec8> (a java.lang.ref.ReferenceQueue$Lock)

at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:164)

at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:209)

"Reference Handler" #2 daemon prio=10 os\_prio=2 tid=0x0000000002dc0000 nid=0x9750 in Object.wait() [0x0000000026b6f000]

java.lang.Thread.State: WAITING (on object monitor)

at java.lang.Object.wait(Native Method)

- waiting on <0x0000000715d86b68> (a java.lang.ref.Reference$Lock)

at java.lang.Object.wait(Object.java:502)

at java.lang.ref.Reference.tryHandlePending(Reference.java:191)

- locked <0x0000000715d86b68> (a java.lang.ref.Reference$Lock)

at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:153)

"main" #1 prio=5 os\_prio=0 tid=0x000000000283e800 nid=0x16f8 waiting for monitor entry [0x000000000293f000]

java.lang.Thread.State: BLOCKED (on object monitor)

at com.zhss.demo.register.server.ServiceRegistryCache.get(ServiceRegistryCache.java:95)

- waiting to lock <0x0000000715e60de8> (a java.lang.Object)

at com.zhss.demo.register.server.RegisterServerController.fetchFullRegistry(RegisterServerController.java:119)

at com.zhss.demo.register.server.RegisterServer.main(RegisterServer.java:34)

"VM Thread" os\_prio=2 tid=0x00000000258e7800 nid=0x933c runnable

"GC task thread#0 (ParallelGC)" os\_prio=0 tid=0x0000000002ce7800 nid=0x7054 runnable

"GC task thread#1 (ParallelGC)" os\_prio=0 tid=0x0000000002ce9000 nid=0x9ed8 runnable

"GC task thread#2 (ParallelGC)" os\_prio=0 tid=0x0000000002cea800 nid=0x9c8c runnable

"GC task thread#3 (ParallelGC)" os\_prio=0 tid=0x0000000002cec000 nid=0x9c9c runnable

"VM Periodic Task Thread" os\_prio=2 tid=0x0000000026d60800 nid=0xa2c4 waiting on condition

JNI global references: 6

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/163\_Semaphore源码剖析之如何基于AQS等待指定数量的线程（一）/笔记 .doc**

源码

state，一开始是你构造Semaphore的时候初始化传递的一个0

直接就将state - permits = 0 - 1 = -1 = remaining

此时通过state的操作，发现remaining = -1，小于0 的，此时就会将当前线程封装为一个节点加入AQS等待队列里去，而且会park挂起线程

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/182~201资料/192\_如何基于复杂的指针移动完成单向链表的入队？/笔记.doc**

基于CAS实现的线程并发安全性，非阻塞

LinkedQueue，底层一定是基于链表来实现的，所以一定会有Node类数据结构，Node指向对方串成一个链表，单向的，双向的，head和tail两个指针都是指向了链表中的头节点和尾节点

if (p.casNext(null, newNode)) {

这是一个CAS操作，就是把空节点的next指针指向了新的节点，同一时间只有一个线程可以执行成功这个操作

此时如果再次往队列里入队一个李四元素

此时如果再次往队列里入队一个王五元素

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/182~201资料/196\_如何在peek获取队头元素的时候基于CAS安全更新head指针？/笔记.doc**

peek，获取队头的元素，但是队头的元素是不出队列的，读一下队列的元素是什么就可以了，没什么特别的，poll源码，peek源码非常的简单

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/182~201资料/199\_同理分析一下contains如何基于volatile不受队列变化的影响/笔记.doc**

队列还有一个contains，读操作，判断对立中是否包含某个元素

遍历到一半，前面的头部的节点被出队，你之前已经对头部的元素判断过了，是否是你想要查找的元素

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/182~201资料/186\_基于写时复制机制实现多次拷贝数组删除list元素/笔记.doc**

list.remove，写时复制机制

[张三, 李四, 王二, 麻子]

list.remove(1)，我要删除李四这个兄弟，拷贝数组，也是要做一些复杂点的操作，新的数组，新数组的大小是3

张三复制到新数组里去，把王二和麻子复制到新数组里去，新数组里就是[张三, 王二, 麻子]

len = 4

index = 1

numMoved = 4 - 1 - 1 = 2

如果你要删除的是最后一个元素，index = 3

numMoved = 4 - 3 - 1 = 0，此时是不需要移动任何元素的

setArray(Arrays.copyOf(elements, len - 1));

[张三, 李四, 王二]

Object[] newElements = new Object[len - 1];

新数组的大小：len - 1，比老数组的大小少1，对不对

System.arraycopy(elements, 0, newElements, 0, index);

把老数组里的，从index = 0开始的元素，截止到index = 1的元素，只能是“张三”，复制到新数组的从index = 0位置开始的地方

[null, null, null]

[张三, null, null]

System.arraycopy(elements, index + 1, newElements, index, numMoved);

从老数组的index = 2，拷贝2个元素，index = 2和index = 3的两个元素，给拷贝到了新数组里去，从新数组的index = 1的位置开始放置

[张三, 王二, 麻子]

再把新数组走一个volatile写，设置为CopyOnWriteArrayList底层的数组

CopyOnWrite理解的透彻，增删改的时候，都是先复制一个数组出来，对新的复制数组进行修改，最后将修改好的新数组设置为底层数组

list.get(1)，读操作

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/182~201资料/191\_ConcurrentLinkedQueue：线程安全的链表结构无界队列/笔记.doc**

ConcurrentLinkedQueue，队列，无界，队列的大小是不限制的，LinkedList，源码分析过，线程安全的无界队列

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/182~201资料/197\_统统CAS保证线程安全性：多线程安全的并发删除队列元素/笔记.doc**

从队列里删除指定位置的一个元素

first = peek，获取队头的一个元素，使用队列的时候，remove操作很少很少会用的，不是玩儿，从队列里删除一个元素

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/182~201资料/187\_List读操作不需要加锁的秘密：写时复制多副本机制/笔记.doc**

list.get，源码实现，写操作的源码，独占锁 + 写时复制

其实就是直接从底层数组里来读取数据，通过index定位对应位置的元素就可以了

假设此时我要add一个赵六进去

写数据的时候一定要CopyOnWrite，如何解决读写并发的问题，写数据的时候，如何安全的读数据，ConcurrentHashMap里面是直接操作一个数组的，对数组读写全部是走的volatile的操作

CopyOnWriteArrayList设计思想，并不是基于CAS执行读写操作，写数据的时候，全部复制一个副本，新的数组，对新的数组来修改，修改好了设置回去就可以了。读数据，只有两种情况

第一种：我读到的老数组的数据

第二种：其他线程更新好了数组，volatile写，我读到的是新数组的数据

我不需要依赖任何一种加锁的机制来保证数据读写并发的安全性，我甚至都不需要依赖于Unsafe.getObjectVolatile()，volatile读机制，来读取数组里的元素，我直接就是最最简单，最最高效

最最高性能的，读就是直接读当前数组的数据即可，要么读的是老数据，要么读的是最新的数据，都有可能

写数据更新的是复制好的另外一个副本数组，同一时间大量的线程读数据的时候，都是在读老数组的数据，读写之间是没有任何的并发冲突问题的，读和写之间是没有锁的冲突的，写的是副本数组

CopyOnWrite机制，写副本数组，跟读就没关系了，只要写完成之后，走一个volatile写，设置最新的数组，自然读操作就会读到最新数组的元素了，只有一个线程可以写，但是写的同时可以允许大量的线程来并发读

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/182~201资料/193\_多线程并发入队的时候如何基于CAS保证线程安全性？/笔记.doc**

麻子和赵六两个元素是同时往队列入队，但是麻子CAS成功了，导致麻子就入队了，赵六的CAS就会失败

ConcurrentLinkedQueue.offer，保证多线程并发安全的奥秘，就在于一行代码：

if (p.casNext(null, newNode)) {

上面那行代码保证了是通过CAS操作把元素入队，同一时间只有一个线程可以入队成功的，其他的线程就会入队失败，此时会不断的for循环，变换指针，尝试让指针往后挪动到最后一个尾部节点

再次尝试用CAS将自己入队，加入队尾

上面的过程会通过for循环保证不断的尝试直到成功为止

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/182~201资料/182\_CopyOnWriteArrayList：线程安全的List数据结构/笔记.doc**

并发包下的线程安全的数据结构，ArrayList

ConcurrentHashMap -> 如何来实现分段加锁，JDK 1.7的hashmap并发死循环

多线程场景下，经常会出现多个线程并发的访问同一个内存数据结构的情况，HashMap、ArrayList、LinkedQueue、HashSet、BlockingQueue

只要出现了内存数据结构，然后你的系统是有多线程并发的场景的，很可能会出现说多个线程并发的访问内存里的数据结构

并发包下面，就干脆实现了一堆常用的线程安全的数据结构，多个线程访问HashMap是线程安全的，不会把HashMap里的数据改错，ConcurrentHashMap。ArrayList数据结构，内存里面，多线程要并发的访问这个数据结构

CopyOnWriteArrayList，写时复制机制的ArrayList，可以保证线程并发的安全性

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/182~201资料/201\_案例实战：基于线程安全的无界队列重构最近变更服务队列机制/笔记.doc**

ConcurrentLinkedQueue之后，来用一下，在微服务注册中心的项目里

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/182~201资料/190\_案例实战：重构分布式存储系统的集群注册机制/笔记.doc**

参考一下hadoop hdfs源码，hdfs分布式存储系统里面，这块就是他用到了CopyOnWriteArrayList，写不是特别的多，大量的读操作的时候，就比如说大量的对list进行遍历，但是更新list里的元素很少

分布式存储系统的案例里来使用一下

在这里面，大家去回顾一下hadoop的架构原理，NameNode、DataNode，注册机制，发送心跳，发送心跳的代码简单来写一下，不太合适

提供一个方法，来遍历ServiceActor，对每个ServiceActor做一些事情

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/182~201资料/194\_再来看看基于复杂指针移动实现的单向链表的出队操作实现/笔记.doc**

poll，从队头出队

if (item != null && p.casItem(item, null)) {

基于CAS操作，将队列的头部元素的item值设置为null，因为是CAS操作，出队的操作只能是同一时间就一个线程可以成功

通过一堆指针的变化，找到队列的头结点，通过CAS保证只有一个线程可以执行出队的操作，队列头部Node的item设置为null，原来的head Node给脱离出队列让jvm gc掉，head指针指向最新的头部的null节点

最后就是返回一个“张三”值，队列头部的值

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/182~201资料/188\_对List进行元素迭代时采用的副本快照机制分析/笔记.doc**

读一个list，除了get操作以外，迭代list，他是如何来迭代的

Iterator迭代器里面是包含了一个snapshot，指向的就是CopyOnWriteArrayList当前的数组，array对象，你创建了一个Iterator迭代器对象之后，在迭代的过程中，都是基于快照数组在进行遍历的

如果此时有另外一个线程更新了数组的元素，设置了array，但是此时对迭代是没有任何影响的，你的迭代器里有一个snapshot指针指向了老的数组对象，他会一直用这个老的数组完成遍历

迭代也是基于快照的机制来实现的，读操作其实跟写操作都是半毛钱关系没有的，CopyOnWrite机制，这样的一套设计思想，保证了就是说你的读的并发能力是很强的，不需要加锁的

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/182~201资料/198\_从读写并发的角度分析一下size获取队列大小的局限性/笔记.doc**

offer和poll，入队，出队

peek，remove一般不用的，倒不用太纠结，提一下

size => 看一下队列的大小，有多少个元素，并没有任何的锁的机制，他就是直接从头节点开始遍历，遍历每个链表中的每个节点，count++

如果在遍历的过程中，有人执行入队或者是出队，此时你觉得会怎么样

入队，你从队头开始遍历，遍历到一半儿的时候，有人在队列尾部进行入队的操作，你能及时的看到吗？

入队最最核心的操作，就是设置队列尾部节点的next指针，设置好了以后，新的节点就入队了，volatile写，你遍历的时候立马就可以看到的

如果是从队头开始遍历的时候，遍历到了一半儿，此时有人从队头出队，这个就没办法了，没法防止这个事情，你如果已经遍历到了一半儿了，此时是感知不到的

ConcurrentLinkedQueue，都是为了优化多线程并发性能，牺牲掉了一些数据一致性的，为了保证多线程写队列的高并发性能，大量的采用CAS无锁化的操作，很多读操作，尤其是常见的size，不涉及到任何锁

size之类的操作，都是一瞬间的快照你看到的数量，他不一定准确的，你拿到了一个size，可能你刚刚拿到一个size说是队列的大小是10，结果立马队列大小变成了5，有可能的，你对这种并发安全集合的使用

线程安全的集合包，使用的时候也要考虑到多线程并发的一些数据不一致的问题

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/182~201资料/184\_基于写时复制机制实现在list中加入一个元素/笔记.doc**

add操作，实现的源码，JDK源码，属于JDK的一部分，代表了你的非常底层以及核心的一块技术能力，写JDK的人都是世界级的Java大牛，读JDK的源码，基本上代表了世界顶级的java代码的水准

volatile的运用、CAS的运用，设计思想，都是非常厉害的

CopyOnWrite：写时复制的机制，大量的写操作都是基于写时复制的机制来实现的

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

elements：代表的是当前CopyOnWriteArrayList内部的数组，Arrays.copyOf复制的操作，就是把当前数组复制到了新的数组里去，新的数组的长度是多少呢？len + 1，newElements，就是一个全新的数组

新数组里包含了老数组所有的元素，而且长度还多了1位

CopyOnWrite，写数据的时候，不是直接在当前的数组里写的，他是先把老数组复制到新数组里来，大小为len + 1，接着是对新数组进行更新操作

ArrayList就不大一样了，先有一个固定大小的数组，往里面放数据，如果达到一定的程度，就会扩容这个数组

newElements[len] = e;

把新数组的最后一位的元素设置成要添加的元素，你的更新的操作此时都是发生在新数组里的，跟老数组是没关系的，写时复制的机制，写数据的时候，复制一个新的数组，然后在新的数组里更新元素

setArray(newElements);

最后再把新的数组设置为CopyOnWriteArrayList对应的一个数组，volatile写保证说，只要他一写，其他线程可以立马读到

老数组稍后就会被jvm垃圾回收掉了，已经没有人使用他了

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/182~201资料/189\_CopyOnWriteArrayList核心思想：弱一致性提升读并发/笔记.doc**

CopyOnWriteArrayList：弱一致性

多个线程并发的读写list，中间一定是有一段时间，是复制数组被修改好了，还没设置给array；但是此时其他线程读到的都是老数组的数据，这个过程中，多个线程看到的数据是不一致的，人家修改了数据没有立马被人读到

弱一致性 -> 最终一致性

优点：读和写不互斥的，写和写互斥，同一时间就一个人可以写，但是写的同时可以允许其他所有人来读；读和读也是并发的；读写锁机制还要好；他也不涉及到Unsafe.getObjectVolatile

使用场景：多线程并发安全性，可以选用他；尽可能是读多写少的场景，大量的读是不被影响的；可能有一个线程刚刚发起了写，此时别的线程读到的还是旧的数据，也有这种可能，还好

ArrayList，synchronized(list)，ReadWriteLock来操作这个ArrayList

缺点：空间换时间，写的时候，经常内存里会出现复制出来的一模一样的副本，对内存消耗过大，副本机制保证了保证读写并发优化，大量的并发读不需要锁互斥，list如果很大，可能你要考虑在线上运行的时候，可能经常

内存占用会是list大小的几倍

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/182~201资料/200\_单单基于CAS就能保证队列迭代遍历的过程没任何问题吗？/笔记.doc**

iterator机制，是基于副本快照机制来实现的，创建了一个迭代器之后，他只会用当时的一个副本快照来完成遍历，数据读不一致的问题

队列：offer、poll、size、iterator

你在迭代的就是ConcurrentLinkedQueue内部的那个单向链表，所以你迭代的过程中，很可能会有新的元素入队，也可能会有已经遍历过的队头的元素会出队，你一定要在使用集合的时候要考虑到这些问题

很多时候多线程访问一个内存数据结构，你直接用并发包里的类就可以的，符合你的需求，如果他原生的读写并发的支持无法满足你的需求，其实很多时候可能还是要对内存数据结构自己手动上锁的

synchronized，ReadWriteLock

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/182~201资料/183\_采用ReentrantLock保证修改元素时的线程并发安全性/笔记.doc**

/\*\*

\* Creates an empty list.

\*/

public CopyOnWriteArrayList() {

setArray(new Object[0]);

}

从上面这段构造函数的代码，就可以看出来，CopyOnWriteArrayList其实也是底层基于数组来实现的，List数据结构，要实现各种线程安全性

/\*\* The array, accessed only via getArray/setArray. \*/

private transient volatile Object[] array;

核心的底层数据结构是数组，volatile，保证 多线程读写的可见性，只要有一个线程修改了这个数组，其他的线程立马是可以读到的

/\*\* The lock protecting all mutators \*/

final transient ReentrantLock lock = new ReentrantLock();

每一个CopyOnWriteArrayList底层都对应一个数据结构，Object[]数组，同时还对应了一个ReentrantLock独占锁，就是用独占锁来保证说要修改Object[]数组的时候，必须加独占锁，此时只能有一个线程获取锁

独占锁保证了，只有一个线程可以来修改底层的数组里的数据

增删改操作的时候，都必须先获取一把ReentrantLock独占锁，保证同一时间只能有一个线程来操作底层的数组数据结构，更新CopyOnWriteArrayList的多线程并发的安全性就被保证了，多线程并发写的时候

写并发并不是特别的好，ConcurrentHashMap分段加锁，CAS非阻塞式的分段加锁，分段的粒度很细，所以并发写的性能是特别好的，并发读是通过volatile读来保证一定是读到最新的数据的

并发写CopyOnWriteArrayList的性能是较差的，基本上所有的线程都需要串行起来写CopyOnwriteArrayList，一个线程先写完，下一个线程才能写

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/182~201资料/185\_基于写时复制机制实现拷贝相同大小数组更新list元素/笔记.doc**

list.set，写时复制的机制

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

将老数组复制到一个新数组里去，新老两个数组的大小是一样的，都是len，修改一个元素，并不是删除元素，也不是新增元素

newElements[index] = element;

对复制之后的一个新数组的指定index位置的元素设置为element元素

setArray(newElements);

他就会将修改后的新数组设置为CopyOnWriteArrayList底层的数组

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/182~201资料/195\_那么多线程并发出队的时候如何基于CAS保证顺序出队？/笔记.doc**

同一时间多个线程都执行poll操作的时候，会如何？

模拟两个线程同时来执行poll出队的操作，线程1执行成功了CAS操作，开始出队；对于线程2来说，线程2通过指针的变化指向了最新的一个头节点，再次尝试用CAS操作进行出队

多个线程同时出队的时候，通过CAS + for，可以保证线程并发安全性，一个头节点同一个时间只能被一个线程出队，出队了之后，别的线程出队失败，再次尝试对最新的一个头节点尝试出队

线程安全的无界内存队列，ConcurrentLinkedQueue，给大家分析清楚了，入队+出队都是基于CAS来做的，保证同一时间只有一个线程可以入队或者是出队，其他线程如果入队或者出队失败了，会通过for循环再次尝试入队或者是出队

整个过程没有任何锁，全部是CAS，非阻塞的无界队列，多线程并发操作的性能会比用锁的那种机制会更好

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/31~52资料/31\_写一个多线程i++的小程序体验一下线程安全问题/笔记.doc**

我们来写一个多线程i++的程序，体验一下，多线程如果是并发的修改一个数据，会有什么样的线程并发安全问题

volatile，解决的对一个共享数据，有人写，有人读，多个线程并发读和写的可见性的问题，多个线程对一个共享数据并发的写，可能会导致数据出错，原子性的问题

观察，直接让多个线程对同一个变量并发的写，是有问题的，会出现很多的数据是不正常的

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/31~52资料/34\_synchronized在代码中的各种常见使用方法详解/笔记.doc**

synchronized锁两种东西，一种是对某个实例对象来加锁，另外一种是对这个类进行加锁。对类加锁，也是在针对一个对象实例进行加锁，其实他的意思就是对那个类的Class对象进行加锁

其实大家现在都知道，synchronized可以对两种对象加锁，对象实例，Class对象

你要是直接synchronized修饰一个普通的方法，那么就是对当前这个对象实例在加锁，访问同一个对象实例的synchronized方法，同一时间只有一个线程可以做到，如果是下面那种synchronized代码片段，也是这个意思：

synchronized(myObject) {

}

但是如果是两个线程，分别进入不同的对象的synchronized方法或者代码片段，这个没事，因为是在不同的对象上加锁，要理解清楚这一点

其实synchronized一个代码片段，有更加常见的一种写法，就是用this，其实意思就是基于当前这个对象实例来加锁：

synchronized(this) {

}

你要是synchronized一个静态方法，就是对这个类的Class对象加锁，兄弟，别告诉我不知道这个Class对象是什么啊，每个类都对应了一个Class对象，那么对同一个类的synchronized静态方法，同一时间只能有一个线程加锁进入其中，下面的那个代码片段，也是这个意思：

synchronized(MyObject.class) {

}

我经常会校招，招聘一些应届生，一般来说如果说面试官人手不足的话，我也会当校招的三面的面试官，亲自去当三面，校招，最常问的一个问题，synchronized普通方法，synchronized静态方法，他们的区别是什么

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/31~52资料/42\_案例实战：基于synchronized实现自我保护机制的开启/笔记.doc**

每分钟的实际的心跳次数，随着服务的注册、下线、故障，自我机制的阈值会不断的变动，在ServiceAliveMonitor进行故障摘除之前，可以先判断一下是否要触发自我保护机制，上一分钟的心跳次数 < 期望的心跳次数 \* 0.85，小于的话

认为自己的网络故障了，人家的心跳发不过来，此时就进入自我保护机制，不再摘除任何服务实例

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/31~52资料/38\_案例实战：微服务注册中心的自我保护机制的思想/笔记.doc**

自我保护

spring cloud eureka，微服务注册中心里，有一个东西，他是说有一个机制，自我保护，给大家简单的解释一下他的思想，如果是学习过spring cloud源码剖析的课程的话，是不听我讲自己都知道

register-server部署的机器出现了故障，导致网络有了问题，导致大量的服务实例都没有办法发送心跳信息过来，导致register-server上的ServiceAliveMonitor线程在检查的时候，会发现大面积的出现各个服务实例超过90秒没发送心跳

我一火起来，直接把所有的服务实例都给摘除 => 靠谱吗？

自我保护，什么情况下可以让register-server认为是自己的网络问题导致别人的心跳发不过来呢？其实有一个比例，spring cloud eureka提供的一个比例是25%，超过25%的服务实例的心跳都没法过来

如果ServiceAliveMonitor发现超过25%的服务实例的心跳，都没及时更新，就可以认为是可能register-server自己部署的机器的网络出现了故障，导致人家的心跳发不过来

就开始自动进入自我保护机制，他不再摘除任何的服务实例，避免说在自己网络故障的情况下，一下子就摘除50%，60%的服务实例，导致注册表的数据出现严重的数据缺失。如果后面再次运行的时候，发现已经有超过85%的服务实例其实恢复了发送心跳了

此时ServiceAliveMonitor可以自动退出自我保护的状态，可以继续去检查是不是某个服务实例他的心跳没有发送过来，没有及时更新，超过90秒还没更新，此时认为这个服务实例就宕机了，摘除

收集一下每分钟的心跳总次数，比如说你现在有10个服务实例，按理说每分钟应该有20次心跳，但是某一分钟你发现收到的心跳次数只有8次，此时就发现8 < 20 \* 0.85，也就是说，发现有超过25%的服务实例的心跳没有正常的发送

如果出现这样的情况，认为是自己的网络出问题，自动进入自我保护的机制

就不再摘除任何的服务实例，避免说一下子摘除大量的服务实例，导致注册表的数据出现问题，将自己的注册表的数据给保护住了

如果某一分钟你收到的心跳的次数，达到了18次，18 > 20 \* 0.85，此时就认为是我的网络肯定是正常了，此时就退出自我保护机制，就可以正常的检查服务实例的心跳是否在90秒内更新过，如果没更新过，就可以自动摘除这个故障实例

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/31~52资料/49\_案例实战：分布式存储系统的创建目录功能实现/笔记.doc**

这一讲完成了核心的在内存文件目录树中加入一个目录节点的代码实现，通过这个东西给大家演示了一把，内存里的文件目录树肯定是多线程并发写的资源，所以说必须得把他用synchronized给保护起来

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/31~52资料/33\_体验一下synchronized加锁如何解决多线程并发安全问题/笔记.doc**

加锁，syncrhonized，一旦说某个线程加了一把锁之后，就会保证，其他的线程没法去读取和修改这个变量的值了，同一时间，只有一个线程可以读这个数据以及修改这个数据，别的线程都会卡在尝试获取锁那儿

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/31~52资料/52\_案例实战：基于wait与notify实现edits log批量刷磁盘/笔记.doc**

此时已经到了线程如何等待别人刷完磁盘，自己再刷的这个事儿

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/31~52资料/39\_案例实战：基于synchronized实现服务的心跳计数器/笔记.doc**

记录一下每分钟有多少心跳发送过来

ServiceAliveMonitor线程每次尝试摘除服务实例的时候，都会检查一下上一分钟的心跳次数是否满足超过85%的服务实例的心跳都正常的条件，如果不满足，就进入自我保护机制，避免随意摘除大量的服务实例

Atomic那个系列的时候，我们会来优化这个心跳计数器，到时候我们可以对时间分片的划分，以及ServiceAliveMonitor触发自我保护机制的一些缺陷，到时候统一来分析一下，或者等到最后的时候

这个JDK底层技术系列都结束了之后，我们就会开发出来一个完整微服务注册中心的中间加你，spring cloud eureka实现了里面的机制、架构和细节，但是的话，我们最后会分析eureka的机制，一系列的问题，架构、机制的问题，大量开始改造微服务注册中心中间件他的代码和架构

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/31~52资料/47\_案例实战：分布式存储系统的edits log机制介绍/笔记.doc**

hadoop的架构原理，分布式存储，hdfs edits log机制，主要是作为操作日志记录到磁盘里去，如果说NameNode突然宕机的话，内存中的元数据可以通过edits log来进行恢复，edits log的原理和机制

分段加锁、内存双缓冲，复杂的机制，手写实现这套机制

石杉的架构笔记，《大规模集群下Hadoop NameNode如何承载每秒上千次高并发请求》，主要就是分析的edits log里面的那套机制

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/31~52资料/44\_wait与notify的底层原理：monitor以及wait set/笔记.doc**

wait与notify实现的一个底层原理，synchronized的原理，主要是monitor

必须得对同一个对象实例进行加锁、wait、notify，这样的话，他们其实操作的才是通一个对象实例里的monitor相关的计数器、wait set

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/31~52资料/43\_手写一个简易内存队列来体验一下wait与notify的作用/笔记.doc**

synchronized，多线程并发写一个数据，结合微服务注册中心里的一些案例之后，我们来体验了一下，如何来在项目里使用synchronized，配合着synchronized要来学习一下，wait和notify这两个东西

wait和notify / notifyAll还是挺有用的，在多线程开发中，还是挺常见的，我后面会用一个案例来给大家演示一下，在分布式系统里，如何使用wait和notifyall，线程通信，某个线程可以处于等待状态，其他线程可以来通知他唤醒他

wait和notify其实主要是用来控制线程的，你手头有多个线程在运行，你可以用各种各样的手段来控制他们，volatile、synchronzied、wait和notify，有意思，你必须是在各种真实的项目里用过，你才能真正的体会到他们的用法

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/31~52资料/37\_案例实战：基于synchronized解决注册表的并发读写问题/笔记.doc**

关于多线程并发写一个数据的安全问题，我们现在学到的就是用synchronized，内存注册表肯定是有多线程并发安全的问题的，所以我们可以考虑，先用synchronized来保护一下，对注册表的读写请求，都用synchronized关键字类保护

ConcurrentHashMap，线程安全的数据结构，我们可以用ConcurrentHashMap替换synchronized保护措施，分段加锁的机制，并发的支持更好

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/31~52资料/35\_图解synchronized底层原理（jvm指令以及monitor锁）/笔记.doc**

之前有一些同学去一线互联网大厂里去面试，聊并发编程这块的内容，问的比较深一点，就说synchronized的底层原理是什么呢？他当时就答不出来了

其实synchronized底层的原理，是跟jvm指令和monitor有关系的

你如果用到了synchronized关键字，在底层编译后的jvm指令中，会有monitorenter和monitorexit两个指令

monitorenter

// 代码对应的指令

monitorexit

那么monitorenter指令执行的时候会干什么呢？

每个对象都有一个关联的monitor，比如一个对象实例就有一个monitor，一个类的Class对象也有一个monitor，如果要对这个对象加锁，那么必须获取这个对象关联的monitor的lock锁

他里面的原理和思路大概是这样的，monitor里面有一个计数器，从0开始的。如果一个线程要获取monitor的锁，就看看他的计数器是不是0，如果是0的话，那么说明没人获取锁，他就可以获取锁了，然后对计数器加1

这个monitor的锁是支持重入加锁的，什么意思呢，好比下面的代码片段

synchronized(myObject) {

// 一大堆的代码

synchronized(myObject) {

// 一大堆的代码

}

}

如果一个线程第一次synchronized那里，获取到了myObject对象的monitor的锁，计数器加1，然后第二次synchronized那里，会再次获取myObject对象的monitor的锁，这个就是重入加锁了，然后计数器会再次加1，变成2

这个时候，其他的线程在第一次synchronized那里，会发现说myObject对象的monitor锁的计数器是大于0的，意味着被别人加锁了，然后此时线程就会进入block阻塞状态，什么都干不了，就是等着获取锁

接着如果出了synchronized修饰的代码片段的范围，就会有一个monitorexit的指令，在底层。此时获取锁的线程就会对那个对象的monitor的计数器减1，如果有多次重入加锁就会对应多次减1，直到最后，计数器是0

然后后面block住阻塞的线程，会再次尝试获取锁，但是只有一个线程可以获取到锁

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/31~52资料/32\_结合java内存模型来说说多线程安全问题的产生原因/笔记.doc**

java内存模型，给大家解释一下原子性问题是怎么回事，多个线程同时并发写一个变量，会出现一些问题，根子原因还是java内存模型，volatile是无法保证原子性的，他其实底层那套机制，MESI缓存一致性协议，强制刷主存，过期其他线程的工作内存的缓存

sycnrhonized、wait、notify、ReadWriteLock、死锁、活锁

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/31~52资料/51\_案例实战：基于synchronized实现edits log的分段加锁机制/笔记.doc**

下一讲：最后实现基于wait和notify的线程等待的机制

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/31~52资料/41\_案例实战：基于synchronized实现自我保护触发阈值的修改/笔记.doc**

已经可以记录每一分钟他的心跳的次数

需要知道按理来说他需要每分钟收到多少次的心跳才不会进入自我保护的机制

你如果注册了一个服务实例，在注册的时候，是不是就要去修改一下自我保护触发的阈值，正常的每分钟的心跳次数就会是2

如果一个服务实例被摘除了，或者是某个服务实例下线了，此时每分钟正常的心跳次数就需要减2

然后每次你的心跳次数变动之后，那个0.85阈值是不是也要修改

如果你此时注册了4个服务实例，8次心跳，8 \* 0.85 = 7，阈值，如果某一分钟的心跳次数小于了7，就自动触发自我保护机制

会导致他频繁的进入自我保护机制，也没关系，我们后面可以来在最后那块，优化这个机制，这个机制还是挺有必要的，保护的就是自己出现网络故障的时候不要随意摘除服务实例，springcloud eureka实现的这套机制很不完善

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/31~52资料/50\_案例实战：edits log的全局txid机制以及双缓冲机制实现/笔记.doc**

全局txid机制，双缓冲机制

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/31~52资料/36\_案例实战：微服务的定时注册表拉取机制实现/笔记.doc**

服务注册的时候会写注册表，服务存活监控线程会读注册表+写注册表，各个微服务还会拉取注册表（读注册表），多线程并发的写和读这个注册表，可能会导致一些数据的问题所产生，项目完善一下

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/31~52资料/48\_案例实战：分布式存储系统NameNode代码框架实现/笔记.doc**

动手写写代码，hadoop，hdfs，namenode，edits log，都已经比较熟悉了，咱们直接动手撸代码，把edits log那套机制给他撸出来

如果说要做edits log，场景，你现在执行一个命令，hadoop fs -mkdir /usr/warehosue，创建一个目录，非常简单，两件：在内存里的文件目录树中加入对应的目录节点；在磁盘里写入一条edits log，记录本次元数据的修改

hdfs client去创建目录的话，会给hdfs NameNode发送一个rpc接口调用的请求，调用人家的mkdir()接口，在那个接口里就会完成上述的两件事情

接下来咱们其实主要是做两件事情，第一件是在内存文件目录树中，加入进去对应的一个目录节点，第二件事情是在edits log写入磁盘文件

FSNamesystem，其实是作为NameNode里元数据操作的核心入口，负责管理所有的元数据的操作，但是在里面的话呢，他可能会调用其他的组件完成相关的事情

FSDirectory，专门负责管理内存中的文件目录树

FSEditLog，专门负责管理写入edits log到磁盘文件里去

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/31~52资料/45\_wait与notify在代码中使用时的注意事项总结/笔记.doc**

wait与sleep的区别：前者释放锁，后者不释放锁

wait()，必须是有人notify唤醒他

wait(timeout)，阻塞一段时间，然后自己唤醒，继续争抢锁

wait与notify，必须在synchronized代码块中使用，因为必须是拥有monitor lock的线程才可以执行wait与notify操作

因此wait与notify，必须与synchornized一起，对同一个对象进行使用，这样他们对应的monitor才是一样的

notify()与notifyall()：前者就唤醒block状态的一个线程，后者唤醒block状态的所有线程

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/31~52资料/40\_案例实战：微服务关闭时的服务下线机制的实现/笔记.doc**

我们其实是要做一下，服务关闭的时候，需要发送一个请求到register-server通知人家说我下线了

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/31~52资料/46\_案例实战：分布式存储系统案例背景引入/笔记.doc**

wait和notify的实战，还是得用分布式存储系统这个案例来做比较合适

hadoop，hdfs，edits log

什么是hadoop？到底什么是hdfs？到底什么是hdfs NameNode和DataNode？

石杉的架构笔记，《兄弟，用大白话告诉你小白都能听懂的Hadoop架构原理》，用非常浅显的语言分享了一下hadoop到底是个什么东西？什么是大数据？为什么要用大数据的技术？Hadoop的架构原理到底是什么？

搞大数据的应该都知道，如果是搞java开发的很多同学是不知道的

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/129~150资料/144\_采用写缓冲器和无效队列优化MESI协议的实现性能/笔记.doc**

MESI协议如果每次写数据的时候都要发送invalidate消息等待所有处理器返回ack，然后获取独占锁后才能写数据，那可能就会导致性能很差了，因为这个对共享变量的写操作，实际上在硬件级别变成串行的了

所以为了解决这个问题，硬件层面引入了写缓冲器和无效队列

写缓冲器的作用是，一个处理器写数据的时候，直接把数据写入缓冲器，同时发送invalidate消息，然后就认为写操作完成了，接着就干别的事儿了，不会阻塞在这里。接着这个处理器如果之后收到其他处理器的ack消息之后

才会把写缓冲器中的写结果拿出来，通过对cache entry设置为E加独占锁，同时修改数据，然后设置为M

其实写缓冲器的作用，就是处理器写数据的时候直接写入缓冲器，不需要同步阻塞等待其他处理器的invalidate ack返回，这就大大提升了硬件层面的执行效率了

包括查询数据的时候，会先从写缓冲器里查，因为有可能刚修改的值在这里，然后才会从高速缓存里查，这个就是存储转发

引入无效队列，就是说其他处理器在接收到了invalidate消息之后，不需要立马过期本地缓存，直接把消息放入无效队列，就返回ack给那个写处理器了，这就进一步加速了性能，然后之后从无效队列里取出来消息，过期本地缓存即可

通过引入写缓冲器和无效队列，一个处理器要写数据的话，这个性能其实很高的，他直接写数据到写缓冲器，发送一个validate消息出去，就立马返回，执行别的操作了；其他处理器收到invalidate消息之后直接放入无效队列，立马就返回invalidate ack

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/129~150资料/142\_高速缓存的数据结构：拉链散列表、缓存条目以及地址解码/笔记.doc**

如果这周的课不讲，只是靠着之前的课，volatile和synchronized的原理，也能说，但是说的比较浅层一些，主要是从基础的层面来聊一下他的原理，底层的细节肯定是不行的，但是这周的课讲完了

volatile和synchronized

原子性、可见性和有序性三个方面分别来聊，这两个关键字对那几个“性”的保障是通过什么来实现的。聊到他们会加哪些内存屏障，怎么加，这些内存屏障的效果，结合底层硬件层面的一个初步的原理，来给面试官聊一下。

还是有一些遗憾的，内存屏障在硬件层面的实现的原理，到底是怎么回事，能不能再细一点，再深入一些，让大家在面试的时候聊到volatile和synchronized，直接震慑式的回答。硬件层面的一些原理

MESI协议在硬件层面的实现机制，光靠初步的MESI协议是无法保证可见性和有序性的

内存屏障在硬件层面的细致的原理，到底是如何控制那些硬件的交互和行为，最终实现可见性和有序性的保障的

volatile和synchronized，底层，彻底通透

synchronized的一些JVM对锁的优化，讲解一下；CAS底层其实也是要靠这套硬件级别的原理来给说清楚，compareAndSwap操作到底是如何在底层实现原子性的，这个东西我之前也没讲

ThreadLocal，源码基本；ReentrantLock，读写锁，源码级别

处理器高速缓存的底层数据结构实际是一个拉链散列表的结构，就是有很多个bucket，每个bucket挂了很多的cache entry，每个cache entry由三个部分组成：tag、cache line和flag，其中的cache line就是缓存的数据

tag指向了这个缓存数据在主内存中的数据的地址，flag标识了缓存行的状态，另外要注意的一点是，cache line中可以包含多个变量的值

处理器会操作一些变量，怎么在高速缓存里定位到这个变量呢？

那么处理器在读写高速缓存的时候，实际上会根据变量名执行一个内存地址解码的操作，解析出来3个东西，index、tag和offset。index用于定位到拉链散列表中的某个bucket，tag是用于定位cache entry，offset是用于定位一个变量在cache line中的位置

如果说可以成功定位到一个高速缓存中的数据，而且flag还标志着有效，则缓存命中；否则不满足上述条件，就是缓存未命中。如果是读数据未命中的话，会从主内存重新加载数据到高速缓存中，现在处理器一般都有三级高速缓存，L1、L2、L3，越靠前面的缓存读写速度越快

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/129~150资料/147\_在复杂的硬件模型之上的Java内存模型是如何大幅简化的/笔记.doc**

java内存模型是对底层的硬件模型，cpu缓存模型，做了大幅度的简化，提供一个抽象和统一的模型给java程序员易于理解，很多时候如果要理解一些技术的本质，还是要深入到底层去研究的

volatile，原子性，可见性，有序性，加了一些内存屏障可以避免前后各种读写指令重排

synchronized，原子性，可见性，有序性，没有提到

CAS，cas指令到硬件级别，实现了一个原子性的cas操作

先把通俗易懂，简单的原理和模型给大家说一下，然后立马大量的实战，实战出真知，没感觉，整天听我讲内存屏障，枯燥死的

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/129~150资料/132\_到底有哪些操作在Java规范中是不保证原子性的呢？/笔记.doc**

所有变量的简单赋值写操作，jva语言规范原生给你保证原子性的；32位java虚拟机里的long/double是不保证赋值写的原子性的；volatile可以解决这个问题；不保证原子性的一些操作呢？

i++

i = y + 1

i = x \* y ==> 先把x和y分别从主内存里加载到工作内存里面来，然后再从工作内存里加载出来执行计算（处理器），计算后的结果写回到工作内存里去，最后还要从工作内存里把i的最新的值刷回主内存

CAS，AtomicInteger => compareAndSet

你敢说他是原子的？

volatile x = 1;

volatile y = 2;

volatile i = x \* y;

我之前给大家已经说过了，画图都演示过了

FSDirectory dir = ...

synchronized(dir) {

dir.add();

dir.remove();

dir.insert();

}

加锁

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/129~150资料/134\_深入探秘有序性：Java程序运行过程中发生指令重排的几个地方/笔记.doc**

我们写好的代码在实际执行的时候那个顺序可能在很多环节都会被人给重排序，一旦重排序之后，在多线程并发的场景下，就有可能会出现一些问题

（1）自己写的源代码中的执行顺序：这个是我们自己写的代码，一般来说就是按照我们自己脑子里想的样子来写

（2）编译后的代码的执行顺序：java里有两种编译器，一个是静态编译器（javac），一个是动态编译器（JIT）。javac负责把.java文件中的源代码编译为.class文件中的字节码，这个一般是程序写好之后进行编译的。JIT负责把.class文件中的字节码编译为JVM所在操作系统支持的机器码，一般在程序运行过程中进行编译。

在这个编译的过程中，编译器是很有可能调整代码的执行顺序的，为了提高代码的执行效率，很可能会调整代码的执行顺序，JIT编译器对指令重排的还是挺多的

（3）处理器的执行顺序：哪怕你给处理器一个代码的执行顺序，但是处理器还是可能会重排代码，更换一种执行顺序，JIT编译好的指令的时候，还是可能会调整顺序

（4）内存重排序：有可能你这个处理器在实际执行指令的过程中，在高速缓存和写缓冲器、无效队列等等，硬件层面的组件，也可能会导致你的指令的执行看起来的顺序跟想象的不太一样

上述就是在我们写好java代码之后，从编译到执行的过程中，可能代码的执行顺序可能会有指令重排的地方，只要有指令重排就有一定可能造成程序执行异常

但是编译器和处理器不是胡乱的重排序的，他们会遵循一个关键的规则，就是数据依赖规则，如果说一个变量的结果依赖于之前的代码执行结果，那么就不能随意进行重排序，要遵循数据的依赖

比如说：

int a = 3;

int b = 5;

int c = a \* b;

那第三行代码依赖于上面两行代码，第一行和第二行代码可以重排序，但是第三行代码必须放在最下面

此外，之前给大家介绍过happens-before原则，就是有一些基本的规则是要遵守的，不会让你胡乱的重排序

在遵守一定的规则的前提下，有好几个层面的代码和指令都可能出现重排序

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/129~150资料/139\_深入分析synchronized是如何通过加锁保证原子性的/笔记.doc**

synchronized底层的原理，monitor，没有特别细化，给大家详细说明一下synchronized实现原子性的核心原理

之前给大家简单说过synchronized加锁的原理，说白了，就是在进入加锁代码块的时候加一个monitorenter的指令，然后针对锁对象关联的monitor累加加锁计数器，同时标识自己这个线程加了锁

通过monitor里的加锁计数器可以实现可重入的加锁

在出锁代码块的时候，加一个monitorexit的指令，然后递减锁计数器，如果锁计数为0，就会标志当前线程不持有锁，释放锁

然后wait和notify关键字的实现也是依托于monitor实现的，有线程执行wait之后，自己会加入一个waitset中等待唤醒获取锁，notifyall操作会从monitor的waitset中唤醒所有的线程，让他们竞争获取锁

这边来深入分析一下那个加锁的底层原理

MyObject lock = new MyObject();

synchronized(lock) {

}

java对象都是分为对象头和实例变量两块的，其中实例变量就是大家平时看到的对象里的那些变量数据。然后对象头包含了两块东西，一个是Mark Word（包含hashCode、锁数据、GC数据，等等），另一个是Class Metadata Address（包含了指向类的元数据的指针）

在Mark Word里就有一个指针，是指向了这个对象实例关联的monitor的地址，这个monitor是c++实现的，不是java实现的。这个monitor实际上是c++实现的一个ObjectMonitor对象，里面包含了一个\_owner指针，指向了持有锁的线程。

ObjectMonitor里还有一个entrylist，想要加锁的线程全部先进入这个entrylist等待获取机会尝试加锁，实际有机会加锁的线程，就会设置\_owner指针指向自己，然后对\_count计数器累加1次

各个线程尝试竞争进行加锁，此时竞争加锁是在JDK 1.6以后优化成了基于CAS来进行加锁，理解为跟之前的Lock API的加锁机制是类似的，CAS操作，操作\_count计数器，比如说将\_count值尝试从0变为1

如果成功了，那么加锁成功了；如果失败了，那么加锁失败了

然后释放锁的时候，先是对\_count计数器递减1，如果为0了就会设置\_owner为null，不再指向自己，代表自己彻底释放锁

如果获取锁的线程执行wait，就会将计数器递减，同时\_owner设置为null，然后自己进入waitset中等待唤醒，别人获取了锁执行notify的时候就会唤醒waitset中的线程竞争尝试获取锁

有人会问，那尝试加锁这个过程，也就是对\_count计数器累加操作，是怎么执行的？如何保证多线程并发的原子性呢？很简单，JDk 1.6之后，对synchronized内的加锁机制做了大量的优化，这里就是优化为CAS加锁的

你如果说在之前把ReentrantLock底层的源码都读懂了，AQS的机制都读懂了之后，那么synchronized底层的实现差不多的，synchronized的ObjectMonitor的地位就跟ReentrantLock里的AQS是差不多的

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/129~150资料/131\_volatile原来还可以保证long和double变量写操作的原子性/笔记.doc**

杠精，文章里说volatile无法保证原子性？我们的文章是胡说八道

volatile对原子性保障的语义，在java里很有限的，几乎可以忽略不计。32位的java虚拟机里面，对long/double变量的赋值写是不原子的，此时如果对变量加上了volatile，就可以保证在32位java虚拟机里面，对long/double变量的赋值写是原子的了

int i = 0，原子性，volatile，java语言规范就规定了，原子性的

volatile long i;

多个线程执行：i = 30，此时就不要紧了，因为volatile修饰了，就可以保证这个赋值操作是原子的了

你以后出去面试也可能会遇到杠精面试官，你要说volatile是保证可见性和有序性的，不保证原子性，杠精面试官，素质差，很二，心胸很狭隘，volatile可以保证原子性，此时看过这一讲之后

i++，复杂的一些场景

resources = loadResources();

resources.execute();

ready = true;

杠精的思维模式，他们恰巧看过一些博客，知道32位虚拟机里的long/double不是原子的，volatile可以保证原子性，在外面格外的要凸显自己的水平，跟一般人不一样的地方。提醒一点，如果遇到杠精面试官，走人；杠精同事，杠回去，打压他

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/129~150资料/141\_再看volatile关键字对原子性、可见性以及有序性的保证/笔记.doc**

volatile对原子性的保证真的是非常的有限，其实主要就是32位jvm中的long/double类型变量的赋值操作是不具备原子性的，加上volatile就可以保证原子性了

volatile boolean isRunning = true;

线程1：

Release屏障

isRunning = false;

Store屏障 => 对于之前的讲解，更进了一步，原理，没有过多的牵扯到内存屏障的一些东西，可见性和有序性，主要都是基于各种内存屏障来实现的

线程2：

Load屏障

while(isRunning) {

Acquire屏障

// 代码逻辑

}

在volatile变量写操作的前面会加入一个Release屏障，然后在之后会加入一个Store屏障，这样就可以保证volatile写跟Release屏障之前的任何读写操作都不会指令重排，然后Store屏障保证了，写完数据之后，立马会执行flush处理器缓存的操作

在volatile变量读操作的前面会加入一个Load屏障，这样就可以保证对这个变量的读取时，如果被别的处理器修改过了，必须得从其他处理器的高速缓存（或者主内存）中加载到自己本地高速缓存里，保证读到的是最新数据；在之后会加入一个Acquire屏障，禁止volatile读操作之后的任何读写操作会跟volatile读指令重排序

跟之前讲解的volatie读写内存屏障的知识对比一下，其实你看一下是类似的意思的

那个Acquire屏障其实就是LoadLoad屏障 + LoadStore屏障，Release屏障其实就是StoreLoad屏障 + StoreStore屏障

好像有点不太一样，对吧？

其实不要对内存屏障这个东西太较真，因为说句实话，不同版本的JVM，不同的底层硬件，都可能会导致加的内存屏障有一些区别，所以这个本来就没完全一致的。你只要知道内存屏障是如何保证volatile的可见性和有序性的就可以了

看各种并发相关的书和文章，对内存屏障到底是加的什么屏障，莫衷一是，没有任何一个官方权威的说法，因为这个内存屏障太底层了，底层到了涉及到了硬件，硬件不同对内存屏障的实现是不一样的

内存屏障这个东西，大概来说，其实就是大概的给你说一下这个意思，尤其是Release屏障，Store屏障和Load屏障还好理解一些，比较简单，Acqurie屏障，莫衷一是，我也没法给你一个官方的定论

具体底层的硬件实现

如果你一定 要杠到底，到底加的准确的屏障是什么？到底是如何跟上下的指令避免重排的，你自己去研究吧。我之前看过很多的资料，做过很多的研究，硬件对这个东西的实现和承诺，莫衷一是，没有标准和官方定论。

两点：volatile读写前后会加屏障，避免跟前后的读写操作发生指令重排

volatile和synchronized保证可见性和有序性，原来都是通过各种内存屏障来实现的，因为加了内存屏障，就会有一些特殊的指令和实现，就可以保证可见性和有序性了，有序性在几个阶段的指令重排的问题

内存屏障对应的底层的一些基本的硬件级别的原理，也都讲清楚了

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/129~150资料/150\_再来看看CAS是如何基于MESI协议在底层硬件层面实现加锁的/笔记.doc**

无法发出指令来执行一个原子性的cas，先查出数据，比较一下，如果一样，就写数据。MESI协议有关系

volatile、synchronized、CAS、ThreadLocal、ReentrantReadWriteLock、锁优化、锁生产故障

并发的核心和关键的技术都到了硬件和源码的级别，大家都应该掌握的很好了

先讲线程安全的并发包下的集合，同步器组件，线程池，并发的核心技术，并发编程设计模式完全结合我们后续的微服务注册中心的项目完善、开发和实战来演练

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/129~150资料/137\_高速缓存和写缓冲器的内存重排序造成的视觉假象/笔记.doc**

处理器会将数据写入写缓冲器，这个过程是store；从高速缓存里读数据，这个过程是load。写缓冲器和高速缓存执行load和store的过程，都是按照处理器指示的顺序来的，处理器的重排处理器也是按照程序顺序来load和store的

但是有个问题，就是在其他的处理器看到的一个视觉假象而言，有可能会出现看到的load和store是重排序的，也就是内存重排序

处理器的乱序执行和推测执行，都是指令重排序，这次说的是内存重排序，因为都是发生在内存层面的写缓冲器和高速缓存中的

这个内存重排序，有4种可能性：

（1）LoadLoad重排序：一个处理器先执行一个L1读操作，再执行一个L2读操作；但是另外一个处理器看到的是先L2再L1

（2）StoreStore重排序：一个处理器先执行一个W1写操作，再执行一个W2写操作；但是另外一个处理器看到的是先W2再W1

（3）LoadStore重排序：一个处理器先执行一个L1读操作，再执行一个W2写操作；但是另外一个处理器看到的是先W2再L1

（3）StoreLoad重排序：一个处理器先执行一个W1写操作，再执行一个L2读操作；但是另外一个处理器看到的是先L2再W1

给大家举个例子，比如说写缓冲器为了提升性能，有可能先后到来W1和W2操作了之后，他先执行了W2操作，再执行了W1操作。那这个时候其他处理器看到的可不就是先W2再W1了，这就是StoreStore重排序

共享变量：

Resource resource = null;

Boolean resourceLoaded = false;

处理器0：

resource = loadResoureFromDisk();

resourceLoaded = true;

处理器1：

while(!resourceLoaded) {

try {

Thread.sleep(1000);

} catch(Exception) {

}

}

resource.execute();

类似上面的代码，很可能处理器0先写了resource，再写了resourceLoaded。结果呢，写缓冲器进行了内存重排序，先落地了resourceLoaded = true了，此时resource还是null。此时处理器1就会看到resourceLoaded = true，就会对resource对象执行execute()方法，此时就会有空指针异常的问题

反正类似的情况，高速缓存和写缓冲器都可以自己对Load和Store操作的结果落地到内存进行各种不同的重排序，进而造成上述4种内存重排序问题的发生

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/129~150资料/143\_结合硬件级别的缓存数据结构深入分析缓存一致性协议/笔记.doc**

因为有高速缓存的存在，所以就导致各个处理器可能对一个变量会在自己的高速缓存里有自己的副本，这样一个处理器修改了变量值，别的处理器是看不到的，所以就是为了这个问题引入了缓存一致性协议（MESI协议）

MESI协议规定：对一个共享变量的读操作可以是多个处理器并发执行的，但是如果是对一个共享变量的写操作，只有一个处理器可以执行，其实也会通过排他锁的机制保证就一个处理器能写

之前说过那个cache entry的flag代表了缓存数据的状态，MESI协议中划分为：

（1）invalid：无效的，标记为I，这个意思就是当前cache entry无效，里面的数据不能使用

（2）shared：共享的，标记为S，这个意思是当前cache entry有效，而且里面的数据在各个处理器中都有各自的副本，但是这些副本的值跟主内存的值是一样的，各个处理器就是并发的在读而已

（3）exclusive：独占的，标记为E，这个意思就是当前处理器对这个数据独占了，只有他可以有这个副本，其他的处理器都不能包含这个副本

（4）modified：修改过的，标记为M，只能有一个处理器对共享数据更新，所以只有更新数据的处理器的cache entry，才是exclusive状态，表明当前线程更新了这个数据，这个副本的数据跟主内存是不一样的

MESI协议规定了一组消息，就说各个处理器在操作内存数据的时候，都会往总线发送消息，而且各个处理器还会不停的从总线嗅探最新的消息，通过这个总线的消息传递来保证各个处理器的协作

下面来详细的图解MESI协议的工作原理，处理器0读取某个变量的数据时，首先会根据index、tag和offset从高速缓存的拉链散列表读取数据，如果发现状态为I，也就是无效的，此时就会发送read消息到总线

接着主内存会返回对应的数据给处理器0，处理器0就会把数据放到高速缓存里，同时cache entry的flag状态是S

在处理器0对一个数据进行更新的时候，如果数据状态是S，则此时就需要发送一个invalidate消息到总线，尝试让其他的处理器的高速缓存的cache entry全部变为I，以获得数据的独占锁。

其他的处理器1会从总线嗅探到invalidate消息，此时就会把自己的cache entry设置为I，也就是过期掉自己本地的缓存，然后就是返回invalidate ack消息到总线，传递回处理器0，处理器0必须收到所有处理器返回的ack消息

接着处理器0就会将cache entry先设置为E，独占这条数据，在独占期间，别的处理器就不能修改数据了，因为别的处理器此时发出invalidate消息，这个处理器0是不会返回invalidate ack消息的，除非他先修改完再说

接着处理器0就是修改这条数据，接着将数据设置为M，也有可能是把数据此时强制写回到主内存中，具体看底层硬件实现

然后其他处理器此时这条数据的状态都是I了，那如果要读的话，全部都需要重新发送read消息，从主内存（或者是其他处理器）来加载，这个具体怎么实现要看底层的硬件了，都有可能的

这套机制其实就是缓存一致性在硬件缓存模型下的完整的执行原理

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/129~150资料/149\_Java虚拟机对锁的优化：锁消除、锁粗化、偏向锁、自旋锁/笔记.doc**

从JDk 1.6开始，JVM就对synchronized锁进行了很多的优化

有个别同学完全没搞明白并发到底是怎么回事，一直追着问，什么是偏向锁，什么是自旋锁，锁是一种单独的锁类别。真是不懂并发技术，小白，小菜，too young too simple。其实是synchronized底层的优化和实现

synchronized说是锁，但是他的底层加锁的方式 可能不同，偏向锁的方式来加锁，自旋锁的方式来加锁，轻量级锁的方式来加锁

这些东西本身你只要了解一个概念就可以了，JDK 1.6开始对synchronized关键字做过哪些优化，有哪些加锁的方式，效果是什么，作用是什么，在实际的开发和使用中，根本就不需要你去过多的care一些东西

synchronized(this) {

}

（1）锁消除

锁消除是JIT编译器对synchronized锁做的优化，在编译的时候，JIT会通过逃逸分析技术，来分析synchronized锁对象，是不是只可能被一个线程来加锁，没有其他的线程来竞争加锁，这个时候编译就不用加入monitorenter和monitorexit的指令

这就是，仅仅一个线程争用锁的时候，就可以消除这个锁了，提升这段代码的执行的效率，因为可能就只有一个线程会来加锁，不涉及到多个线程竞争锁

（2）锁粗化

synchronized(this) {

}

synchronized(this) {

}

synchronized(this) {

}

这个意思就是，JIT编译器如果发现有代码里连续多次加锁释放锁的代码，会给合并为一个锁，就是锁粗化，把一个锁给搞粗了，避免频繁多次加锁释放锁

（3）偏向锁

这个意思就是说，monitorenter和monitorexit是要使用CAS操作加锁和释放锁的，开销较大，因此如果发现大概率只有一个线程会主要竞争一个锁，那么会给这个锁维护一个偏好（Bias），后面他加锁和释放锁，基于Bias来执行，不需要通过CAS

性能会提升很多

但是如果有偏好之外的线程来竞争锁，就要收回之前分配的偏好

可能只有一个线程会来竞争一个锁，但是也有可能会有其他的线程来竞争这个锁，但是其他线程唉竞争锁的概率很小

如果有其他的线程来竞争这个锁，此时就会收回之前那个线程分配的那个Bias偏好

（4）轻量级锁

如果偏向锁没能成功实现，就是因为不同线程竞争锁太频繁了，此时就会尝试采用轻量级锁的方式来加锁，就是将对象头的Mark Word里有一个轻量级锁指针，尝试指向持有锁的线程，然后判断一下是不是自己加的锁

如果是自己加的锁，那就执行代码就好了

如果不是自己加的锁，那就是加锁失败，说明有其他人加了锁，这个时候就是升级为重量级锁

（5）适应性锁

这是JIT编译器对锁做的另外一个优化，如果各个线程持有锁的时间很短，那么一个线程竞争锁不到，就会暂停，发生上下文切换，让其他线程来执行。但是其他线程很快释放锁了，然后暂停的线程再次被唤醒

也就是说在这种情况下，线程会频繁的上下文切换，导致开销过大

所以对这种线程持有锁时间很短的情况，是可以采取忙等策略的，也就是一个线程没竞争到锁，进入一个while循环不停等待，不会暂停不会发生线程上下文切换，等到机会获取锁就继续执行好了

一直追问我，什么自旋锁，不是什么事儿，当然，如果要站在jvm的底层层面，去说清楚的话，确实是比较复杂的，但是我觉得起码目前为止，暂时也没必要，各种锁底层是如何来实现的，完全可以等到以后jvm那块都讲过之后

再回过头来深入jvm底层的原理来剖析：偏向锁、自旋锁、轻量级锁，jvm层面的概念，栈侦，Load Record，不一定能听懂，基础的知识没有铺垫好，需要通过调节jvm的一些参数来优化底层synchronized里的各种加锁方式的使用

这样可以大幅度减少线程上下文的切换，而这种自旋等待获取锁的方式，就是所谓自旋锁，就是不断的自旋尝试获取锁

如果一个线程持有锁的时间很长，那么其他线程获取不到锁，就会暂停，发生上下文切换，让其他线程来执行，这种自己暂停获取锁的方式，就是所谓的重量级锁

这个根据不同情况自动调整的过程，就是适应锁的意思

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/129~150资料/133\_可见性涉及的底层硬件概念：寄存器、高速缓存、写缓冲器/笔记.doc**

从硬件的级别来考虑一下可见性的问题

每个处理器都有自己的寄存器（register），所以多个处理器各自运行一个线程的时候，可能导致某个变量给放到寄存器里去，接着就会导致各个线程没法看到其他处理器寄存器里的变量的值修改了

可见性的第一个问题，首先，就有可能在寄存器的级别，导致变量副本的更新，无法让其他处理器看到

然后一个处理器运行的线程对变量的写操作都是针对写缓冲来的（store buffer）并不是直接更新主内存，所以很可能导致一个线程更新了变量，但是仅仅是在写缓冲区里罢了，没有更新到主内存里去

这个时候，其他处理器的线程是没法读到他的写缓冲区的变量值的，所以此时就是会有可见性的问题，这是第二个可见性发生的场景

然后即使这个时候一个处理器的线程更新了写缓冲区之后，将更新同步到了自己的高速缓存里（cache，或者是主内存），然后还把这个更新通知给了其他的处理器，但是其他处理器可能就是把这个更新放到无效队列里去，没有更新他的高速缓存

此时其他处理器的线程从高速缓存里读数据的时候，读到的还是过时的旧值

可见性发生的问题

如果要实现可见性的话，其中一个方法就是通过MESI协议，这个MESI协议实际上有很多种不同的时间，因为他不过就是一个协议罢了，具体的实现机制要靠具体底层的系统如何实现

根据具体底层硬件的不同，MESI协议的实现是有区别的

比如说MESI协议有一种实现，就是一个处理器将另外一个处理器的高速缓存中的更新后的数据拿到自己的高速缓存中来更新一下，这样大家的缓存不就实现同步了，然后各个处理器的线程看到的数据就一样了

为了实现MESI协议，有两个配套的专业机制要给大家说一下：flush处理器缓存、refresh处理器缓存。

flush处理器缓存，他的意思就是把自己更新的值刷新到高速缓存里去（或者是主内存），因为必须要刷到高速缓存（或者是主内存）里，才有可能在后续通过一些特殊的机制让其他的处理器从自己的高速缓存（或者是主内存）里读取到更新的值

除了flush以外，他还会发送一个消息到总线（bus），通知其他处理器，某个变量的值被他给修改了

refresh处理器缓存，他的意思就是说，处理器中的线程在读取一个变量的值的时候，如果发现其他处理器的线程更新了变量的值，必须从其他处理器的高速缓存（或者是主内存）里，读取这个最新的值，更新到自己的高速缓存中

所以说，为了保证可见性，在底层是通过MESI协议、flush处理器缓存和refresh处理器缓存，这一整套机制来保障的

要记住，flush和refresh，这两个操作，flush是强制刷新数据到高速缓存（主内存），不要仅仅停留在写缓冲器里面；refresh，是从总线嗅探发现某个变量被修改，必须强制从其他处理器的高速缓存（或者主内存）加载变量的最新值到自己的高速缓存里去

内存屏障的使用，在底层硬件级别的原理，其实就是在执行flush和refresh，MESI协议是如何与内存屏障搭配使用的（flush、refresh）

volatile boolean isRunning = true;

isRunning = false; => 写volatile变量，就会通过执行一个内存屏障，在底层会触发flush处理器缓存的操作；while(isRunning) {}，读volatile变量，也会通过执行一个内存屏障，在底层触发refresh操作

之前给大家讲过那个volatile关键字的作用，对一个变量加了volatile修饰之后，对这个变量的写操作，会执行flush处理器缓存，把数据刷到高速缓存（或者是主内存）中，然后对这个变量的读操作，会执行refresh处理器缓存，从其他处理器的高速缓存（或者是主内存）中，读取最新的值

当然跟我们之前说的有一点点不一样，因为之前说的是写volatile变量的时候，一个是强制刷主内存，一个是过期掉其他处理器的高速缓存中的数据；读volatile变量的时候，会发现高速缓存中的值过期，然后强制从主内存加载最新值

其实这个东西吧，你没发现么，效果是一样的，他其实本质都是让一个线程写了volatie变量之后，另外一个变量立马可以读到volatile变量的值，只不过MESI协议的底层具体实现，根据cpu等硬件的不同，有多种不同的实现方式罢了

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/129~150资料/148\_面试的时候如何从内存屏障、硬件层面的原理来震慑面试官/笔记.doc**

volatile、synchronized

原子性这块，直接把底层的一些东西喷出来

硬件层面的原理 -> MESI协议在硬件层面运行的原理 -> 这套原理为何会导致可见性和有序性的问题 -> 各种内存屏障是如何在硬件层面解决可见性和有序性的问题 -> volatile和synchroized是如何加各种内存屏障来分别保证可见性和有序性的

行业里对并发这块知识掌握到这个层面的人，不多

很多人写并发的书，如果你把我们的课看完了，有并发的书，你去看看，XX并发实战，书里的内容很浅，你的水平可能已经超过部分写并发书籍的作者了

是个面试官，主要不是技术特别牛的，一般的人多会被你给震慑到，从硬件层面开始画图

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/129~150资料/138\_synchronized锁同时对原子性、可见性以及有序性的保证/笔记.doc**

原子性、可见性、有序性，三块东西，都重新从比较细节和底层的层面，都在硬件的级别去给大家说了一下，到底是怎么回事，为什么会发生这个问题，从底层的层面来说了一下，以及大体上有没有什么办法可以来解决这些问题

原子性，基本的赋值写操作都是可以保证原子性的，复杂的操作是无法保证原子性的

可见性，MESI协议、flush、refresh，配合起来，才可以解决可见性

有序性，三个层次，最后一个层次有4种重排（LoadLoad、StoreStore、LoadStore、StoreLoad）

synchronized关键字，同时可以保证原子性、可见性以及有序性的

原子性的层面而言，他加了以后，有一个加锁和释放锁的机制，加锁了之后，同一段代码就只有他可以执行了

可见性，可以保证可见性的，他会通过加入一些内存屏障，他在同步代码块对变量做的写操作，都会在释放锁的时候，全部强制执行flush操作，在进入同步代码块的时候，对变量的读操作，全部会强制执行refresh的操作

更新的数据，别的县城关只要进入代码块，就一定可以读到的

有序性，synchronized关键字，他会通过加各种各样的内存屏障，来保证说，解决LoadLoad、StoreStore等等重排序

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/129~150资料/135\_JIT编译器对创建对象的指令重排以及double check单例实践/笔记.doc**

JIT动态编译的时候，有可能会造成一个非常经典的指令重排

public class MyObject {

private Resource resource;

public MyObject() {

this.resource = loadResource(); // 从配置文件里加载数据构造Resource对象

}

public void execute() {

this.resource.execute();

}

}

// 线程1:

MyObject myObj = new MyObject(); => 这个是我们自己写的一行代码

// 线程2：

myObj.execute();

// 步骤1：以MyObject类作为原型，给他的对象实例分配一块内存空间，objRef就是指向了分配好的内存空间的地址的引用，指针

objRef = allocate(MyObject.class);

// 步骤2：就是针对分配好内存空间的一个对象实例，执行他的构造函数，对这个对象实例进行初始化的操作，执行我们自己写的构造函数里的一些代码，对各个实例变量赋值，初始化的逻辑

invokeConstructor(objRef);

// 步骤3：上两个步骤搞定之后，一个对象实例就搞定了，此时就是把objRef指针指向的内存地址，赋值给我们自己的引用类型的变量，myObj就可以作为一个类似指针的概念指向了MyObject对象实例的内存地址

myObj = objRef;

有可能JIT动态编译为了加速程序的执行速度，因为步骤2是在初始化一个对象实例，这个步骤是有可能很耗时的，比如说你可能会在里面执行一些网络的通信，磁盘文件的读写，都有可能

JIT动态编译，指令重排，为了加速程序的执行性能和效率，可能会重排为，步骤1 -> 步骤3 -> 步骤2

线程1，刚刚执行完了步骤1和步骤3，步骤2还没执行，此时myObj已经不是null了，但是MyObject对象实例内部的resource是null

线程2，直接调用myObj.execute()方法， 此时内部会调用resource.execute()方法，但是此时resource是null，直接导致空指针

double check单例模式里面，就是可能会出现这样的JIT指令重排，如果你不加volatile关键字，会导致一些问题的发生，volatile是避免说步骤1、步骤3、步骤2，必须全部执行完毕了，此时才能试用myObj对象实例

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/129~150资料/145\_硬件层面的MESI协议为何会引发有序性和可见性的问题？/笔记.doc**

MESI协议在硬件层面的原理其实大家都已经了解的很清晰了，对不对

可见性和有序性的问题

可见性：写缓冲器和无效队列导致的，写数据不一定立马写入自己的高速缓存（或者主内存），是因为可能写入了写缓冲器；读数据不一定立马从别人的高速缓存（或者主内存）刷新最新的值过来，invalidate消息在无效队列里面

有序性：

（1）StoreLoad重排序

int a = 0;

int c = 1;

线程1：

a = 1;

int b = c;

这个很简单吧，第一个是Store，第二个是Load。但是可能处理器对store操作先写入了写缓冲器，此时这个写操作相当于没执行，然后就执行了第二行代码，第二行代码的b是局部变量，那这个操作等于是读取a的值，是load操作

这就导致好像第二行代码的load先执行了，第一行代码的store后执行

第一个store操作写到写缓冲器里去了，导致其他的线程是读不到的，看不到的，好像是第一个写操作没执行一样；第二个load操作成功的执行了

StoreLoad重排，Store先执行，Load后执行；Load先执行，Store后执行

（2）StoreStore重排序

resource = loadResource();

loaded = true;

两个写操作，但是可能第一个写操作写入了写缓冲器，然后第二个写操作是直接修改的高速缓存，这个时候不就导致了两个写操作顺序颠倒了？

诸如此类的重排序，都可能会因为MESI的机制发生

可见性问题也是一样的，写入写缓冲器之后，没刷入高速缓存，导致别人读不到；读数据的时候，可能invalidate消息在无效队列里，导致没法立马感知到过期的缓存，立马加载最新的数据

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/129~150资料/136\_现代处理器为了提升性能的指令乱序和猜测执行的机制/笔记.doc**

指令乱序机制

指令不一定说是拿到了一个指令立马可以执行的，比如有的指令是要进行网络通信、磁盘读写，获取锁，很多种，有的指令不是立马就绪可以执行的，为了提升效率，在现代处理器里面都是走的指令的乱序执行机制

把编译好的指令一条一条读取到处理器里，但是哪个指令先就绪可以执行，就先执行，不是按照代码顺序来的。每个指令的结果放到一个重排序处理器中，重排序处理器把各个指令的结果按照代码顺序应用到主内存或者写缓冲器里

这就导致处理器可能压根儿就是乱序在执行我们代码编译后的指令

另外还有一个猜测执行，比如说if判断中有一坨代码，很可能先去执行if里的代码算出来结果，然后最后再来判断if是否成立

int sum = 0

if(flag) {

for(int i = 0; i < 10; i++) {

}

}

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/129~150资料/129\_再谈原子性：Java规范规定所有变量写操作都是原子的/笔记.doc**

topic，每一周发的课其实都是讲一个完整的topic，在大量的实战过了volatile、Atomic、ThreadLocal、synchronized、读写锁，写了很多的代码，而且对一些基本的原理都有了一定的掌握之后

回过头来，来看看这个java并发技术底层的原理，volatile、synchronized的对可见性、有序性的保障的语义，底层其实是基于内存屏障来实现的，硬件底层原理（高速缓存、写缓冲器、无效队列），各种内存屏障在底层硬件层面他的实现的原理

回过头来看看，volatile和synchronized通过各种内存屏障的使用，底层在硬件级别的实现原理到底是什么

可见性、有序性、原子性，都彻底通透了以后，硬件级别的原理，给大家再说一下CAS底层的硬件级别的原理

同步器，CountDownLatch、Semaphore，诸如此类的一些API，本身其实就是用起来非常简单的，其实就是用来做一些多线程同步的控制，剖析一下里面的源码， 再带着大家在微服务注册中心项目，以及轻量级分布式存储系统案例，做一下实战

并发包下的集合，ConcurrentHashMap，ConcurrentLinkedList，解析里面的源码，带着在各种项目来实战

线程池，底层源码，以及项目里的实战

并发技术的实战、底层源码/原理、纯手工写了微服务注册中心的大部分功能、轻量级的分布式存储系统的案例

跟着这个里面做也行，把微服务注册中心完全工业级的各种功能再完善一下，做成一个完全类似eureka，生产可用的微服务注册中心，初步优化他里面的一些问题，他的服务感知的时效性，秒级感知服务上下线，重构自我保护机制（解决里面的bug）

用一个http请求包，把register-client和register-server之间的网络通信给他跑通

尝试在spring cloud里面集成一下我们的这个东西，替换掉原生的eureka

并发课全部结束，第一个中间件项目结束，微服务注册中心

io、网络、netty、zk，讲完了以后，继续升级微服务注册中心的架构，升级架构更加的复杂和完美，网络通信绝对是要基于netty来重构

====================================================================

Applications apps;

线程1：

apps = loadedApps; // 原子的，不需要AtomicReference来处理

java语言规范里面，int i = 0，resource = loadedResoures，flag = true，各种变量的简单的赋值操作，规定都是原子的

包括引用类型的变量的赋值写操作，也是原子的

你赋值的时候，要保证没有人先赋值过，没有人修改过，你才能赋值，AtomicReference的CAS操作来实现了，之前给大家讲解过的

但是很多复杂的一些操作，i++，先读取i的值，再跟新i的值，i = y + 2，先读取y的值，再更新i的值，这种复杂操作，不是简单赋值写，他是有计算的过程在里面的，此时java语言规范默认是不保证原子性的

volatile，保证的可见性和有序性，原子性，杠精，偷换概念，胡说八道；i++，i = y + 2，不是volatile可以保证原子性的

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/129~150资料/130\_32位Java虚拟机中的long和double变量写操作为何不是原子的？/笔记.doc**

原子性这块，特例，32位虚拟机里的long/double类型的变量的简单赋值写操作，不是原子的，long i = 30，double c = 45.0，在32位虚拟机里就不是原子的，因为long和double是64位的

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

如果多个线程同时并发的执行long i = 30，long是64位的，就会导致有的线程在修改i的高32位，有的线程在修改i的低32位，多线程并发给long类型的变量进行赋值操作，在32位的虚拟机下，是有问题的

就可能会导致多线程给long i = 30赋值之后，导致i的值不是30，可能是-3333344429，乱码一样的数字，就是因为高低32位赋值错了，就导致二进制数字转换为十进制之后是一个很奇怪的数字

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/129~150资料/146\_内存屏障在硬件层面的实现原理以及如何解决各种问题/笔记.doc**

可见性问题：

Store屏障 + Load屏障

如果加了Store屏障之后，就会强制性要求你对一个写操作必须阻塞等待到其他的处理器返回invalidate ack之后，对数据加锁，然后修改数据到高速缓存中，必须在写数据之后，强制执行flush操作

他的效果，要求一个写操作必须刷到高速缓存（或者主内存），不能停留在写缓冲里

如果加了Load屏障之后，在从高速缓存中读取数据的时候，如果发现无效队列里有一个invalidate消息，此时会立马强制根据那个invalidate消息把自己本地高速缓存的数据，设置为I（过期），然后就可以强制从其他处理器的高速缓存中加载最新的值了

这就是refresh操作

为了解决有序性问题

内存屏障，Acquire屏障，Release屏障，但是都是由基础的StoreStore屏障,StoreLoad屏障，可以避免指令重排序的效果

StoreStore屏障，会强制让写数据的操作全部按照顺序写入写缓冲器里，他不会让你第一个写到写缓冲器里去，第二个写直接修改高速缓存了

resource = loadResource();

StoreStore屏障

loaded = true;

StoreLoad屏障，他会强制先将写缓冲器里的数据写入高速缓存中，接着读数据的时候强制清空无效队列，对里面的validate消息全部过期掉高速缓存中的条目，然后强制从主内存里重新加载数据

a = 1; // 强制要求必须直接写入高速缓存，不能停留在写缓冲器里，清空写缓冲器里的这条数据

int b = c;

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/129~150资料/140\_synchronized是如何使用内存屏障保证可见性和有序性的/笔记.doc**

int b = 0;

int c = 0;

synchronized(this) { -> monitorenter

Load内存屏障

Acquire内存屏障

int a = b;

c = 1; => synchronized代码块里面还是可能会发生指令重排

Release内存屏障

} -> monitorexit

Store内存屏障

java的并发技术底层很多都对应了内存屏障的使用，包括synchronized，他底层也是依托于各种不同的内存屏障来保证可见性和有序性的

按照可见性来划分的话，内存屏障可以分为Load屏障和Store屏障。

Load屏障的作用是执行refresh处理器缓存的操作，说白了就是对别的处理器更新过的变量，从其他处理器的高速缓存（或者主内存）加载数据到自己的高速缓存来，确保自己看到的是最新的数据。

Store屏障的作用是执行flush处理器缓存的操作，说白了就是把自己当前处理器更新的变量的值，都刷新到高速缓存（或者主内存）里去

在monitorexit指令之后，会有一个Store屏障，让线程把自己在同步代码块里修改的变量的值都执行flush处理器缓存的操作，刷到高速缓存（或者主内存）里去；然后在monitorenter指令之后会加一个Load屏障，执行refresh处理器缓存的操作，把别的处理器修改过的最新值加载到自己高速缓存里来

所以说通过Load屏障和Store屏障，就可以让synchronized保证可见性。

按照有序性保障来划分的话，还可以分为Acquire屏障和Release屏障。

在monitorenter指令之后，Load屏障之后，会加一个Acquire屏障，这个屏障的作用是禁止读操作和读写操作之间发生指令重排序。在monitorexit指令之前，会加一个Release屏障，这个屏障的作用是禁止写操作和读写操作之间发生重排序。

所以说，通过 Acquire屏障和Release屏障，就可以让synchronzied保证有序性，只有synchronized内部的指令可以重排序，但是绝对不会跟外部的指令发生重排序。

synchronized：

（1）原子性：加锁和释放锁，ObjectMonitor

（2）可见性：加了Load屏障和Store屏障，释放锁flush数据，加锁会refresh数据

（3）有序性：Acquire屏障和Release屏障，保证同步代码块内部的指令可以重排，但是同步代码块内部的指令和外面的指令是不能重排的

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/223~245资料/234\_最后来看看线程池关闭的过程中会涉及到什么？/笔记.doc**

关闭这个线程池，在整个系统都关闭的时候会来关闭这个线程池

Worker本身是一个AQS，执行任务的时候，state = 1；执行完一个任务，state = 0

尝试将Worker的state = 1，如果成功了，就说明这个Worker当前的state = 0，说明这个Worker当前是空闲的状态，没有执行任何一个任务，此时就可以中断掉这个Worker，实际上来说Worker内部的线程就会退出

正在执行任务的Worker，tryLock失败了，state = 1，不要去中断他了，让他执行完这个任务再说

任务队列里还有一些任务，等待那些任务会被执行完毕，把队列里的任务都执行完毕，如果队列里的任务都执行完毕了之后，再次执行getTask()的时候会有问题

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/223~245资料/244\_如何根据系统的业务场景需求定制自己的线程池/笔记.doc**

如果你的业务场景有一些需求的话，你完全可以自己直接构造ThreadPoolExecutor的线程池，传入一些参数符合你自己的业务需求的，fixed和cached、scheduled可以满足大部分场景的线程池的使用了

fixed，但是你希望队列是有界的，此时你就可以自己定制了

corePoolSize：线程池里的线程是0，少于这个数量，他会自动创建这个数量的线程的，只要线程池里的线程是少于corePoolSize，他后面就会以take阻塞的方式从队列里获取任务，这些线程是不会释放了

任务一定会直接入队，此时你是用什么队列？无界队列，有界队列，同步队列，LinkedBlockingQueue和SynchronousQueue，如果你看一些开源项目里对线程池的定制的源码的话

有界队列的话，此时如果队列满了，入队失败，就会尝试以非core方式创建线程，直接执行任务，最多线程池里的线程不能超过maximumPoolSize，如果线程池里的线程总数一旦超过了corePoolSize之后

超出corePoolSize数量的那些线程，在从队列获取的时候是走的poll + 超时时间（keepAliveTime）的方式，也就是如果一定时间空闲获取不到任务，自动退出释放线程，维持空闲的线程数量在小于等于corePoolSize

如果等待队列也满了，而且线程数量达到了maximumPoolSize，此时会直接执行reject策略，拒绝你提交任务

创建线程的时候，可以用自己的threadFactory，定制你的线程是不是要设置为daemon，需要不需要线程组的概念

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/223~245资料/227\_如何通过非阻塞的高性能CAS实现安全的线程创建/笔记.doc**

他一定会反复确保如果当前线程数量已经超过了corePoolSize之后，是绝对不让你创建新的core线程放入线程池的，直接会返回false，也就是代表着addWorker方法失败了，无法创建新core线程放入线程池中

if (compareAndIncrementWorkerCount(c))

如果当前线程数量 < corePoolSize的话，就会新创建一个线程出来放入线程池，在此之前先会尝试用CAS操作递增线程数量，如果说CAS成功的话，就可以在后面去创建一个新的线程，放入线程池

假设此时corePoolSize是4，线程数量是3，可能会有不同的线程来执行任务的提交，此时通过CAS可以保证安全的递增线程数量，此时只能有一个线程是CAS递增线程数量成功的，其他线程CAS会失败

如果说CAS成功了，就代表可以继续往下走，去真的创建一个线程出来

如果CAS失败了，再次for循环，判断线程数量是否小于corePoolSize，如果是的话，再次尝试进行CAS递增

CAS安全的来递增当前线程的数量

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/223~245资料/241\_不停往cached线程池提交任务时会导致CPU负载过高吗？/笔记.doc**

答案是：会，就可能会在系统高峰期导致大量的线程被创建出来，然后就是导致机器的CPU负载过高，更有甚至，就是线程太多导致内存溢出，线程太多了，导致CPU负载飙升。队列无限增长，内存飙升

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/223~245资料/235\_不限制线程数量的线程池：cached线程池/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/223~245资料/240\_cached线程池有线程的时候提交任务到队列又如何被执行/笔记.doc**

如果没有线程在从队列获取任务，此时无论你提交多少任务，SynchronousQueue入队都是失败的，offer都是返回false，此时都会直接创建线程来执行任务的，但是如果有线程空闲出来就会尝试从队列poll任务

如果队列为空，此时最多会等待60s空闲去poll一个任务出来，如果超过了60s没有poll到任务，此时这个线程自己就会释放掉

此时你提交任务会如何呢？

此时会通过SynchronousQueue实现一个match配对，offer入队的任务会给最后一个poll任务的线程去执行，如果有线程在poll队列的话，那么你入队都是可以成功的，会交给某个线程来执行的

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/223~245资料/225\_线程池的内部构造过程分析：找找核心成员变量有哪些/笔记.doc**

构造线程池的过程没有什么特别的地方，把握住一个线程池他包含的一些核心的东西

corePoolSize

maximumPoolSize

keepAliveTime

wokdQueue

threadFactory

rejectedHandler

分析源码的过程中，主要就是看他的线程池核心的工作机制，以及工作机制和上面这些东西之间的关系，搞清楚你在构造线程池的时候，传递进去的上面那些是可以如何改变线程池的工作行为

private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));

上面那个东西是一个极为核心和关键的成员变量，他是通过一个32位的integer的数字代表了线程池当前的状态以及线程池内的线程数量

32位的integer的数字，前3位是线程池的状态，后29位是线程池内的线程数量

通过这一个数字以及大量的二进制的运算，实现了，通过一个32位数字可以设置线程池的状态以及线程池的线程数量，通过二进制的运算来查询，修改

线程池的状态是：RUNNING，线程池内的线程数量是：0，刚开始线程池里是没有线程的

ConcurrentLinkedQueue：无界队列

LinkedBlockingQueue / ArrayBlockingQueue：有界队列，阻塞，如果队列满此时放数据会阻塞，如果对了空此时获取数据会阻塞，阻塞的源码

是用到了LinkedBlockingQueue的阻塞的特性，但是是当做无界队列来使用的，默认用了Integer.MAX\_VALUE，20多亿的数据量，几乎是可以无限制的放数据了，普通机器，如果队列里真的放了20多亿数据，肯定会内存溢出

Lock + AQS基于CAS、volatile、ThreadLocal

并发包下的集合：Lock、CAS、volatile

线程池：并发集合、Lock、CAS、volatile

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/223~245资料/229\_提交到线程池的任务是如何完成执行以及指标统计的/笔记.doc**

CAS递增线程数量 -> 创建Worker（theadFactory，AQS）-> 加入线程池（HashSet，独占锁） -> start线程 -> 执行提交的Runnable任务

Worker内部的thread线程，是通过threadFactory构造出来的，就是一个最最普通的thread，一旦启动之后，就会执行自己内部的Runnable业务逻辑，也就是我们传递进去的那个Runnable业务逻辑

只要thread start成功之后，这个提交任务到线程池的方法逻辑就结束掉了

Worker执行一个任务的时候，会通过自己的AQS机制更新一下自己的状态，相当于更新自己当前执行的线程是谁。state = 1，state = 0

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/223~245资料/226\_看看提交任务到线程池的源码执行流程是什么样子的/笔记.doc**

线程池构造，以及构造过程中牵扯的一些核心的一些东西，都给大家讲解了，现在大家应该都做到心里有数，线程池是跟哪些东西有关系的。提交一个Runnable类型的任务（封装了一段异步执行的业务逻辑）到线程池

线程池去执行一个任务的过程是什么样子的

1、第一种情况

如果说当前线程池里的线程数量是小于corePoolSize指定的数量的，此时就会创建一个新的线程出来，corePoolSize = 3，但是刚开始的时候，ctl里设置的线程池里的线程数量是0，刚开始是空的，没有线程的

一开始提交任务到线程池，肯定是会走第一种情况的

线程池里的线程数量一定是小于corePoolSize指定的数量的，此时一定会创建新的线程给放到线程池里去

刚开始你提交任务的时候，前4个任务，他其实都是会发现线程数量 < corePoolSize的，此时都会直接创建新线程放入线程池中，同时新线程负责执行我自己这个提交过过来的任务，干这个事儿

2、第二种情况

如果线程池里的线程数量 >= corePoolSize，此时就不会直接走第一种情况创建新的线程加入线程池，以及执行当前任务，不会干这个事儿了。此时唯一会干的，就是把当前的线程给压入到等待队列里去排队

排队过后做一些额外的处理，如果线程池已经关闭了，此时要从对了里把刚加入的任务给出队，线程池是空的，此时会创建一个线程放进去，还会处理一些额外的情况

一旦线程池里的线程数量达到了corePoolSize之后

提交第五个任务

3、第三种情况

如果线程池的线程数量 >= corePoolSize，无法创建新的线程，此时尝试入队等待也失败，可能队列满了，此时就会再次尝试创建新的线程（根据maximumPoolSize来了，尝试在创建额外的线程出来）

但是，如果创建额外线程都失败了，此时就会走reject策略，拒绝你提交任务

超过corePoolSize数量的线程，在keepAliveTime的时间范围内都空闲之后，比如说60s，额外创建的线程就会被回收掉

通过源码的注释的分析、画图、提交任务的源码流程的分析，我们可以搞清楚了线程池工作的核心流程：corePoolSize、workQueue、maximumPoolSize、rejectHandler

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/223~245资料/230\_核心线程满了之后如何将任务压入近乎无界的队列等待/笔记.doc**

比如说线程池里的线程数量已经是4了，再次提交第5个任务的时候，就会发现线程数量 = corePoolSize了，就不会再次创建新的线程了，入队等待的时候，走的是LinkedBLockingQueue的offer()方法

offer()方法实际上并不是阻塞的，非阻塞的3616118202

如果此时队列满，不是卡住等待人家take掉一个元素，而是直接返回false

使用的是近乎于无界的队列，所以基本上可以认为你offer入队的时候，永远是返回true，入队永远不会有满一说，入队都是成功的，可以无限的不停的入队

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/223~245资料/239\_非core线程执行完之后如何尝试从队列获取下一个任务/笔记.doc**

当前线程数量，一定是大于corePoolSize的，corePoolSize是0，无论你有多少个线程，都是属于非core线程，都会大于，这里的timed就是true

非core线程在从队列获取任务的时候走的是poll，非阻塞的，而且有超时时间，keepAliveTime，如果超过指定时间没获取到任务，此时就会当前线程就会自动释放掉，此时会进入SynchronousQueue的栈

他会位于栈顶，但是挂起当前线程的时间超时时间是60s，超过60s的话，就认为此次poll就失败了，null，此时就会返回一个null，此时就自动触发这个非core线程的释放了

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/223~245资料/231\_线程池中的工作线程如何从队列获取任务来执行/笔记.doc**

只要线程池的线程数量达到了corePoolSize之后，接下来任务都是直接入队的，无界队列，不停的入队，线程池里启动的线程是如何不停的从队列里获取任务来执行的呢

如果要允许线程的超时，两个条件满足一个就可以：配置项（allowCoreThreadTimedOut），当前线程数量超过了corePoolSize

allowCoreThreadTimedOut，默认是false，不允许core线程因为获取不到任务就超时退出；如果你设置为true的话，就会导致线程池中的core线程，如果超过一定的时间获取不到队列中的任务

就会认为是core线程空闲了一段时间，此时core线程就会允许退出，自己释放掉自己

当前线程数量是否超过了corePoolSize的数量，默认情况下是不会的，CAS保证在corePoolSize范围内的线程数量，但是cached线程池的时候，他是允许创建额外的非core线程，最多是达到maximumPoolSize的数量

超过了corePoolSize数量的额外的线程，都是允许timeout，也就是说，如果一定时间范围内没有从队列获取到任务，说明空闲了一段时间，此时就自动释放掉

fixed线程池中，暂时你可以认为，正常情况下是不会出现这个情况的，线程数量就是corePoolSize的数量

Runnable r = timed ?

workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :

workQueue.take();

fixed线程池的线程在从队列获取任务的时候，两个条件都不会满足，此时就是做的阻塞式的方法从LinkedBlockingQueue里获取任务

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/223~245资料/224\_通过最常见的fixed线程池来一窥线程池构造源码/笔记.doc**

线程池构造的过程

fixed线程池，搞了一堆固定数量的线程，配合了一个无界队列来处理你提交的任务，最多无论什么时候，只能有你指定数量的一些线程来处理任务。如果线程池里所有的线程都在繁忙的过程中，处理任务。

此时的话呢，就只能再次提交任务的时候，把任务给压入无界队列中等待

如果线程池里的某个线程挂掉了，此时他会自己启动一个新的线程加入到线程池里去

线程池里的线程会一直存活在线程池里，等待处理新提交过来的任务，直到你关闭这个线程池

ThreadPoolExecutor才是真正代表的是线程池管理器，管理了一个线程池，内部持有一个线程池，相当于创建了一个线程池

int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

HYPERLINK "eclipse-javadoc:%E2%98%82=demo-concurrent/F:\\/development\\/Java\\/jdk1.8.0\_151\\/jre\\/lib\\/rt.jar<java.util.concurrent(ThreadPoolExecutor.class%E2%98%83ThreadPoolExecutor~ThreadPoolExecutor~I~I~J~Ljava.util.concurrent.TimeUnit;~Ljava.util.concurrent.BlockingQueue\\<Ljava.lang.Runnable;>;%E2%98%82java.util.concurrent.TimeUnit" \o "in java.util.concurrent" ?TimeUnit? unit,

HYPERLINK "eclipse-javadoc:%E2%98%82=demo-concurrent/F:\\/development\\/Java\\/jdk1.8.0\_151\\/jre\\/lib\\/rt.jar<java.util.concurrent(ThreadPoolExecutor.class%E2%98%83ThreadPoolExecutor~ThreadPoolExecutor~I~I~J~Ljava.util.concurrent.TimeUnit;~Ljava.util.concurrent.BlockingQueue\\<Ljava.lang.Runnable;>;%E2%98%82java.util.concurrent.BlockingQueue" \o "in java.util.concurrent" ?BlockingQueue?<? HYPERLINK "eclipse-javadoc:%E2%98%82=demo-concurrent/F:\\/development\\/Java\\/jdk1.8.0\_151\\/jre\\/lib\\/rt.jar<java.util.concurrent(ThreadPoolExecutor.class%E2%98%83ThreadPoolExecutor~ThreadPoolExecutor~I~I~J~Ljava.util.concurrent.TimeUnit;~Ljava.util.concurrent.BlockingQueue\\<Ljava.lang.Runnable;>;%E2%98%82java.lang.Runnable" \o "in java.lang" ?Runnable?> workQueue

)

corePoolSize：线程池里应该有多少个线程

maximumPoolSize：如果线程池里的线程不够用了，等待队列还塞满了，此时有可能根据不同的线程池的类型，可能会增加一些线程出来，但是最多把线程数量增加到maximumPoolSize指定的数量

keepAliveTime + TimeUnit：如果你的线程数量超出了corePoolSize的话，超出corePoolSize指定数量的线程，就会在空闲keepAliveTime毫秒之后，就会自动被释放掉

workQueue：你的线程池的等待队列是什么队列

threadFactory：在线程池里创建线程的时候，你可以自己指定一个线程工厂，按照自己的方式创建线程出来

RejectedExecutionHandler：如果线程池里的线程都在执行任务，然后等待队列满了，此时增加额外线程也达到了maximumPoolSize指定的数量了，这个时候实在无法承载更多的任务了，此时就会执行这个东西

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/223~245资料/236\_提交任务到cached线程池时会先触发线程的创建吗？/笔记.doc**

corePoolSize：0

maximumPoolSize：Integer.MAX\_VALUE

keepAliveTime：60s

workQueue：SynchronousQueue

刚开始线程数量是0，此时是不是小于corePoolSize呢？当然不是了，corePoolSize = 0，所以此时第一种情况的条件是不会满足的

以上述方法构造出来的线程池，他在提交任务的时候是不会上来就直接创建一个线程的，解决了一个疑问：cached线程池，他是不会上来直接创建线程的，会走第二种情况，将任务压入队列中

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/223~245资料/228\_如何基于独占锁安全的将线程放入池子中以及启动任务/笔记.doc**

如果基于加独占锁来安全的创建线程以及放入线程池里，启动这个新的线程

Worker就是工作线程，他是一个AQS的子类，本身自己就可以直接基于AQS来实现独占锁的机制，负责执行你提交的Runnable任务，是一个组件，里面保存了一个核心的state，代表了工作线程的状态

Worker内部是基于threadFactory（有一个默认的线程工厂），你也可以自己手动指定一个线程工厂，此时就会按照线程工厂的策略来创建一个线程，放在Worker的内部，代表了执行任务的工作线程

ThreadPoolExecutor，核心的数据结构，就是维护了一堆线程的线程池，如果要操作这个核心数据结构，就必须要在ThreadPoolExecutor的层面加一个独占锁，此时只能是一个线程尝试来操作核心数据结构

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/223~245资料/245\_案例实战：服务注册中心集群同步机制基于线程池重构优化/笔记.doc**

把线程池在项目里来用一下

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/223~245资料/223\_上手用一把线程池以及他到底有什么作用和好处？/笔记.doc**

java并发编程课程，他的核心内容的最后一部分的课程了，线程池

如果你不搞线程池，在有些场景里可能会频繁的创建和销毁线程，性能开销是比较大的，性能不是特别好，线程池，搞一批线程，执行完一个任务之后，不要立即销毁这个线程，可以等待去执行下一个任务

100个任务要异步执行，如果频繁的创建和销毁线程，快速的创建和销毁一共是100个线程

线程池，里面就20个线程，先执行20个任务，每个线程执行一个任务，执行完了之后，线程不销毁，继续执行下一批20个任务，以此类推，就是复用了20个线程，执行了几百个任务

避免说频繁的创建和销毁线程

4种，fixed（固定数量的线程），cached（线程数量不固定，无论来多少任务都是不停的创建线程来执行，如果线程空闲了一定的时间，就会释放掉这个线程），single（线程池里就只有一个线程），scheduled（提交进去的线程，会在指定的时间过后才去执行）

可以自己手工定制线程池，实现你需要的一些功能

最最常见的线程池就是fixed线程池，线程数量是固定的，不会超过这个数量的线程，就执行你不断提交的任务

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/223~245资料/242\_cached线程池又会不会触发拒绝任务提交的机制呢？/笔记.doc**

答案是：不会

第一种fixed，是基于有限固定数量的线程处理源源不断涌入的任务，但是呢，无界队列，所以任务可以无限制的涌入和排队

第二种cached，是在需要的时候无限制的创建新的线程来处理新的任务，提交的任务几乎是不会排队的，永远能最快速度的得到执行，入队的时候先看看有没有人空闲在poll，如果有立马执行

4核8G，虚拟机，一般来说，线程池开启线程来异步处理任务，200以内，100~200的时候，线程机器的CPU负载就很高了，内存队列排队个几十万个任务，也还好，内存也没撑爆，但是如果你的线程一旦达到四五百个，线上机器的CPU负载过高的报警

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/223~245资料/238\_线程池已有线程在执行任务时再次提交任务会如何执行呢？/笔记.doc**

SynchronousQueue的原理，TransferStack来进行数据传递

put + take的方式，才能实现他原本希望实现的一个效果，如果put的时候没有人在take，此时就会将head指针指向put操作，put线程就park挂起

再次另外一个线程来put，head指针指向最新的线程put的数据，因为没有人take，所以也是挂起

他其实是用来干两个线程之间的数据传递的同步，put + take，take的时候，如果没人put，此时你会阻塞，直到有人put，是最后一次take的人先可以获取到别人put的数据，栈的效果

put的时候，如果没人take，此时你会阻塞住，多人put，最后一个put是在栈顶，此时有人take，先获取最后一次put的数据，栈，后进先出，queue，反过来，先入先出，默认的实现是栈的实现

SynchronousQueue的offer，他是不会阻塞的，如果没有take，offer的时候直接就是返回null，就是入队失败

此时如果有已经有一个非core线程在执行任务，再来第二个任务要提交，此时会如何呢？当前线程数量 = 1，corePoolSize = 0，1 < 0？不成立，所以此时还是不会直接创建一个core线程出来

此时还是会直接创建一个非core的线程来直接执行提交的任务

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/223~245资料/243\_single线程池和scheduled线程池自己研究的一点提示/笔记.doc**

大家自己去研究

corePoolSize: 1

maximumPoolSize: 1

keepAliveTime: 0

workQueue：无界队列

只有1个线程，不停的处理提交到无界队列的任务

corePoolSize: 10

maximumPoolSize: Integer.MAX\_VALUE

keepAliveTime: 0

workQueue: DelayedWorkQueue

必须是会延迟你指定的时间之后才会去执行那个任务

他肯定是刚开始默认先用你制定的那么多的corePoolSize来执行你的定时任务，有一些延迟任务，定时调度任务，如果任务太多了，估计可能是支持创建额外的线程来进行任务的执行，如果额外的线程

超过corePoolSize的线程如果空闲了以后，会不会释放掉

延迟队列，扔进去的任务会间隔指定的时间，比如说5秒，才可以从延迟队列里出来，被任务给poll出来以后，就会执行，任务对延迟队列的poll肯定是阻塞的，也就是说如果获取不到任务就阻塞起来

他会以固定的线程数量，来处理你提交的所有的延迟任务

猜测一下，核心就在于延迟队列，延迟队列是不是很有可能会第一次5秒后让任务出队被take到，后面每隔3秒都出队一次任务，让线程take到，每次出队的时候，肯定会判断一下，有没有符合延迟时间的任务出队了

如果没有符合要求的任务，他就会在这里被阻塞住，卡住

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/223~245资料/233\_maximumPoolSize和reject策略在fixed线程池中会用吗？/笔记.doc**

答案是：不会

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/223~245资料/232\_面试题：无界队列线程池在远程服务异常时内存飙升怎么办/笔记.doc**

网上有一篇很流行的文章

面试问题：使用无界队列的线程池，在远程服务异常的情况下导致内存飙升怎么办？

最后可能会导致JVM OOM，系统崩溃

如果要解决这个问题从哪个角度来切入，你可以考虑自己定制线程池，使用有界队列，不要使用无界队列，可以限制内存空间的使用，避免系统崩溃；无界队列，直接支撑前端的请求，不允许有界阻塞，到最后大不了就是系统崩溃

远程服务异常的话，可能会导致另外一个问题，你的线程不停的异常报错和崩溃退出

如果你提交到线程池的任务报错了，抛了异常出来，在线程池执行的过程中，Worker层面接收到一个异常，会直接throw抛出去

只要有一个Worker异常挂掉，此时就会把这个worker从线程池里给挪出去，然后判断一下，如果当前线程数量 < corePoolSize，就会重新创建一个Worker和线程放入线程池中，自己再搞一个补位

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/223~245资料/237\_第一个任务到cached线程池的入队失败以及非core线程执行/笔记.doc**

SynchronousQueue，入队

刚开始提交第一个任务的时候，SynchronousQueue的源码反复盘查了两遍，基本上可以确认说没有线程在等待获取任务的时候，入队直接是返回false的，不让你入队成功。必须要有人在等待获取任务，才能入队成功

就导致直接创建一个非core线程，Worker，core是true还是false，只不过是用来决定是跟corePoolSize对比还是maximumPoolSize对比，几乎可以认为永远可以不停的添加非core线程

直接就会把你的第一个线程给他执行完毕

cached线程池，提交第一个任务的时候，就是直接创建一个非core线程给你执行了

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/168\_最近很流行的面试题：HashMap为什么在高并发下会死循环？/笔记.doc**

比较流行的一道面试题：HashMap为什么在高并发下会出现死循环的问题？

JDK 1.7的HashMap确实会在多线程并发的环境下出现死循环的问题，首先这个问题是存在的。面试题我肯定会结合源码来分析一下，HashMap为什么会有多线程并发的安全问题？通过这个来引入ConcurrentHashMap为什么线程安全的

关系比较好的朋友，基本上都是一些P8~P9级别的朋友，他们的观点和态度是说，这个面试题是有意义的，有价值的，有一定工作年限的同学都应该知道这个东西

不这么看，观点，生僻。搞清楚，先看明白JDK 1.7的HashMap的源码，你还得从这个场景出发，到底这个源码下什么时候会出现死循环。麻烦了，都是靠每个人都自己猜测，一步一步画图推演那个源码

有脑子的人都知道，HashMap压根儿不会用在多线程并发的环境下，HashMap是线程并发不安全的。那他就不会出现死循环的

凭什么就一定要求，不会这道面试题，好像就是这个人水平不行

并发，集合的基础的技术，HashMap的原理，源码，现场动手画图，把他的源码原理给我演示一下，hash算法，rehash过程，扩容的过程，ConcurrentHashMap的原理是什么

面试圈子里，风气，开始问这个问题

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/178\_ConcurrentHashMap源码剖析之数组扩容迁移机制/笔记.doc**

什么时候会触发数组的扩容

无论如何，扩容了以后，他其实还是未来操作一个大的数组，对这个数组里的每个元素分段加锁的机制

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/175\_ConcurrentHashMap源码剖析之hash冲突导致的CAS加锁失败/笔记.doc**

put的时候，其实整体的流程还是跟之前是一样的，通过高低16位的异或的算法，降低hash冲突概率，hash值跟数组的大小做了一个取模算法，定位到数组的一个位置，跟之前不一样的一点

他是用的CAS原子性的操作，去对数组里的某个元素赋值的操作，保证多线程并发的安全性，CAS成功，key-value对就直接进到数组里去了

如果CAS失败了以后

两个或者是多个线程同时来进行并发的put操作的时候，如果不巧的定位都到了一个数组的元素的位置，此时大家都尝试进行CAS，只有一个线程是可以执行成功的，硬件层面走加锁的机制

其他的线程此时CAS设置数组的元素就会失败

万一有线程CAS失败了呢？此时会就什么都不会干，直接进入下一轮for循环，(f = tabAt(tab, i = (n - 1) & hash)) == null，此时会发现说数组的那个位置的元素已经不是null了，因为之前已经有人在数组的那个位置设置了一个Node

ConcurrentHashMap，为什么说可以保证线程并发安全性呢？首先多个线程过来对数组的同一个位置的元素赋值都是CAS，线程并发安全，原子性，只有一个线程会成功，这点就保证了ConcurrentHashMap并发安全

他这里其实已经初步体现出了分段加锁的机制

他并没有对整个一个大的数组进行synchronized加锁的机制，并没有，线程都会串行化的来执行，性能和效率是很低下的。

没有对一个数组全部进行加锁，仅仅是说对数组的同一个位置的元素赋值的时候，多个线程会基于CAS（隐含式的加锁），仅仅是对数组的那个位置的元素进行加锁而已，隐式的加锁，java代码里的锁

多个线程并发操作数组的时候，只有对同一个位置的元素赋值，才会用CAS隐式的加锁，CAS实现原子性操作

初步的体现出了分段加锁的思想

数组大小是16，有16个元素，同时可以允许最多是16个线程同时并发的来操作这个数组，如果16个线程操作的是数组的不同位置的元素的话，此时16个线程之间是没有任何锁的关系

数组大小是16,16个元素，CAS赋值的机制，实现了一个效果，这个数组有16把锁，每个元素是一把锁，只有竞争同一个位置的元素的多个线程，才会对一把锁进行争用的操作，大概可以这么来理解

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/162\_Semaphore：等待指定数量的线程完成任务的并发组件/笔记.doc**

Semaphore，等待指定数量的线程完成任务即可

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/174\_ConcurrentHashMap源码剖析之未分段数组的CAS加锁/笔记.doc**

put操作源码

int hash = spread(key.hashCode());

对key获取了hashCode，调用了spread算法，获取到了一个hash值

static final int spread(int h) {

return (h ^ (h >>> 16)) & HASH\_BITS;

}

他相当于是把hash值的高低16位都考虑到后面的hash取模算法里，这样就可以把hash值的高低16位的特征都放到hash取模算法来运算，有助于尽可能打散各个key在不同的数组的位置

降低hash冲突的概率

刚开始，table是null的话，此时就要初始化这个table

U.compareAndSwapInt(this, SIZECTL, sc, -1)：CAS操作，sizeCtl = -1

初始化一个table数组，默认的大小就是16

tabAt(tab, i = (n - 1) & hash)，这个就是hash取模的算法，定位的算法， return (Node<K,V>)U.getObjectVolatile(tab, ((long)i << ASHIFT) + ABASE);，数组和hash定位出来的位置，传递了进来

他在这里的意思，就是走一个线程安全的操作，Unsafe，如果此时数组那个位置的元素给返回出来，当前这个数组这里没有元素的话，此时就直接把你的key-value对放在数组的这个地方了

return U.compareAndSwapObject(tab, ((long)i << ASHIFT) + ABASE, c, v);

通过上述的代码，使用底层的CAS操作，直接将你的key-value对放在了数组的hash定位到的那个位置，为什么ConcurrentHashMap他说是线程安全的呢？

Unsafe，CAS的操作，都是线程安全的，只有一个线程可以在这里成功的将一个key-value对方在数组的一个地方里

如果多个线程并发来执行put的操作，都走到这里，可能就会有其他的线程，CAS往数组里赋值操作就会失败，如果CAS成功了，此时就直接break掉，put操作就成功了

探索出来了，ConcurrentHashMap凭什么说自己是线程安全的，靠什么实现线程安全，put操作里，初始化数组 -> 依靠线程安全的CAS对数组里的位置进行赋值，分段加锁的东西 ，我们此时还没看到

下一讲：如果两个线程并发操作，两个不同的key都定位到数组的同一个位置，有一个线程执行成功，另外一个线程执行CAS失败，如果一个线程CAS在数组里赋值失败了，会怎么样呢？

刚开始其实就一个数组，没有分段的概念，对数组里的赋值，是依托CAS，硬件级别的MESI协议的原理，CAS底层是如何在硬件层面来加锁的

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/153\_CountDownLatch源码剖析之如何基于AQS实现同步阻塞等待（二）/笔记.doc**

countDown()：其实就是把AQS的state给减1，走CAS来设置state的值；如果CAS成功了，就看看当前的state的值是不是0，如果是0就返回true，如果不是0，就返回false

第一个线程把state - 1，此时state = 1，此时这个返回的是false，相当于别的什么都不干了，就是仅仅把state的值给减1就可以了

如果是第二个线程再次来coutnDown操作，state = 0，如果此时state是0的话，就会触发一段逻辑，肯定会从AQS的等待队列中，唤醒之前等待的main线程，让他开始往下去执行，不要继续阻塞了

（1）await()，触发了一个线程入队阻塞等待

（2）countDown()，如果state == 0，唤醒队列里等待的所有的线程

（3）所有线程被唤醒，发现state == 0，就从await()方法里退出

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/165\_案例实战：数据分布式存储场景下的分布式计算架构介绍/笔记.doc**

也不是我来讲，公众号发的一篇文章：《兄弟，用大白话告诉你小白都能看懂的Hadoop架构原理》，讲的都是比较清晰的。关注里面一点，数据需要分布式存储，接着对分布式存储的数据需要进行分布式计算

一个1TB的大文件分散在多个机器上，如果说你要对这个1TB的大文件进行计算的话，你怎么来计算呢？分布式计算，分配多个计算任务到各个机器上去，每个计算任务负责那台机器上的数据的计算

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/164\_Semaphore源码剖析之如何基于AQS等待指定数量的线程（二）/笔记.doc**

state = 0

next = state + 1 = 1

执行CAS，无限循环，将state设置为1

将main线程从AQS等待队列中唤醒

avaiable = state = 1

remaining = 1 - 1 = 0

此时将state设置为0，接着返回0

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/160\_CyclicBarrier源码剖析之如何基于AQS实现任务分而治之（三）/笔记.doc**

await() -> 如何触发阻塞等待

最后一个await() -> 如何触发Runnable执行，重置一些东西，generation，count，唤醒之前阻塞的线程

之前阻塞的线程被唤醒之后，发现generation改变了，直接退出await()

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/156\_案例实战：slave节点注册时同步阻塞等待多个master注册完毕/笔记.doc**

这块代码来写一下

做一个预告：在并发课程里，微服务注册中心是一个项目的地位，就是随着课程的学习，会不断的完善他的代码，到最后所有的机制完备，成为一个工业级可以在生产环境使用的，不差于spring cloud eureka的中间件

在并发课程，分布式存储系统，作为一个案例集合的存在，代码比较demo。在IO + 网络两个课程里，分布式存储系统，就会作为一个项目的存在，就是我们会模仿hadoop hdfs，开发出来基本完整可用的分布式存储系统

在hadoop hdfs，大量的网络通信、磁盘读写，轻量级的分布式存储系统，省去hadoop hdfs一大堆的重量级的功能，实现一些核心的大文件存储存储的功能，IO、网络这些技术，大量的来进行运用

netty是一个关键的环节，重构微服务注册中心和分布式存储系统，两个项目的网络通信的环节，netty引入到两个项目里去

微服务注册中心：nacos、consul，吸收一些其他的微服务注册中心的架构的优点，来对eureka这套微服务注册中心进行大量的架构上的升级，做成一个超越spring cloud eureka的微服务注册中心中间件

eureka也已经停止维护了，我们完全可以通过架构班，来开源出去一个项目，如果大家以后有实力到一些中小型公司里去做架构师的话，把我们自己开发的微服务注册中心中间件，落地到你的公司里去

跟spring cloud来集成，替代掉eureka，committor，PMS；contributor；

分布式存储系统：java系统架构里，一般都有一个技术绕不过去，分布式文件系统，fastdfs，维护的也是相当的不活跃，找一个时机，先模仿hadoop hdfs来开发，把核心功能都实现，集群架构、HA高可用架构、元数据管理架构、大文件分布式存储架构、数据高可用容错架构、简单的分布式文件系统操作模型（创建目录、文件操作）

会看fastdfs的源码，吸收一些轻量级的分布式存储系统的架构特点、优势、功能，落地到自己的轻量级分布式存储系统里去，完善成工业级的概念。开源出去，发起成立一个开源社区，大家一起来维护

如果你能很努力的话，其他开源项目的贡献者，还挺难的。我手把手教你，带你写出来的，开源项目的committor，出去简历，履历，都是很好看的一个东西。哪怕是技术，也会让给你的技术影响力很大

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/177\_ConcurrentHashMap源码剖析之链表和红黑树解决hash冲突问题/笔记.doc**

如果有一个线程成功对数组的某个元素加锁了，synchronized

如何对这个位置的元素进行链表/红黑树的处理

if (e.hash == hash &&

((ek = e.key) == key ||

(ek != null && key.equals(ek)))) {

oldVal = e.val;

if (!onlyIfAbsent)

e.val = value;

break;

}

如果发现说我当前要put的key跟数组里这个位置的key是一样的，此时就对数组当前位置的元素的value值覆盖一下

如果两个key不同，就默认走链表的一个处理，此时就是把e.next = 新封装的一个节点，如下代码所示，e就是数组当前位置的元素

Node<K,V> pred = e;

if ((e = e.next) == null) {

pred.next = new Node<K,V>(hash, key,

value, null);

break;

}

if (binCount != 0) {

if (binCount >= TREEIFY\_THRESHOLD)

treeifyBin(tab, i);

if (oldVal != null)

return oldVal;

break;

}

上面这块代码的判断就是，如果一个链表的元素的数量超过了8，达到了一个阈值之后，就会将链表转换为红黑树。如果转换为红黑树以后，下次如果有hash冲突的问题，是直接把key-value对加入到红黑树里去

else if (f instanceof TreeBin) {

Node<K,V> p;

binCount = 2;

if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,

value)) != null) {

oldVal = p.val;

if (!onlyIfAbsent)

p.val = value;

}

}

出现hash冲突的时候，分段加锁成功了以后，就会做值覆盖/链表/红黑树，处理，前提条件，都是你对数组当前位置的元素synchronized加锁成功了才可以的

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/181\_案例实战：基于ConcurrentHashMap重构微服务注册中心的注册表/笔记.doc**

微服务注册中心里的服务注册表，典型的map数据结构吗？

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/167\_Exchange：支持两个线程之间进行数据交换的并发组件/笔记.doc**

最后exchanger，案例，真没想到在哪里用过

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/159\_CyclicBarrier源码剖析之如何基于AQS实现任务分而治之（二）/笔记.doc**

如果所有线程都执行了await()之后，触发什么事情呢

count == 0

就是触发执行在最开始传递进去的那个Runnable里面的代码逻辑

private void nextGeneration() {

// signal completion of last generation

trip.signalAll();

// set up next generation

count = parties;

generation = new Generation();

}

会通过Condition.signalAll()，唤醒在队列里排队的所有的线程，那些线程都会依次来尝试获取lock锁，因为那些线程此时被唤醒过后，都会进入Lock锁的AQS锁等待队列里去，真的回过头去自己看看

count重置成partities数量，3

线程1和线程2，一旦重新尝试获取锁之后，会怎么样

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/171\_图解剖析JDK 1.7 HashMap并发环境下死循环之死循环与丢数据/笔记.doc**

上一讲，两个线程同时对JDK 1.7的hashmap进行扩容，会导致环形链表

假如说，此时我要是来get一个值，get(k5)，k5的hash取模算法会定位到那个环形链表的位置，遍历环形链表，而且因为环形链表里可能没有k5的值，所以会导致在环形链表中无法找到对应的值来返回

间接的就会导致死循环那个环形链表，不停在里面遍历，无法退出，导致cpu直接100%，线上系统的各个线程可能都会被这个get操作给卡死

丢失数据：如果是线程1的newTable赋值给了hashmap里的table，采用了线程1的newTable之后，就会导致说，此时会导致<k3,v3>这条数据就永久丢失了，可能会被垃圾回收掉，同学

hashmap不是线程安全的，导致一些数据的问题，死循环、丢失数据，也是一个问题

JDK 1.8以后的hashmap，重构了这个算法，采用了红黑树的数据结构，优化了数组扩容的过程，他就不会再有死循环的问题

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/173\_ConcurrentHashMap源码剖析之初始化流程介绍/笔记.doc**

初步的先大致浏览一下ConcurrentHashMap初始化的过程

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/169\_JDK 1.7的HashMap工作原理：hash、链表以及扩容/笔记.doc**

put(key, value) => 对key执行hash算法 => 根据hash值用类似取模的算法 => 定位数组的某一个元素即可，如果数组那个地方是空的，此时就可以直接放在数组里就可以了，这个就是最基本的hash算法

如果说两个key的hash值，对数组取模算法之后，定位到了数组的同一个位置，此时怎么办呢？就会链表处理。也叫做hash冲突的处理

如果数组里承载的元素到了一定的数量之后，数组大小 \* loadFactor（0.75），此时就会进行resize，扩容，按照倍数扩容，搞一个更大的数组，此时就会将原来老数组里的元素，一一遍历，遍历了之后

就可以把那些数据的key，全部按照新数组的大小hash取模，定位到新的数组里去

回顾一下JDK 1.7的HashMap的原理：hash、链表、扩容

void transfer(Entry[] newTable) {

Entry[] src = table;

int newCapacity = newTable.length;

for (int j = 0; j < src.length; j++) {

Entry<K,V> e = src[j];

if (e != null) {

src[j] = null;

do {

Entry<K,V> next = e.next;

int i = indexFor(e.hash, newCapacity);

e.next = newTable[i];

newTable[i] = e;

e = next;

} while (e != null);

}

}

}

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/180\_JDK 1.8对ConcurrentHashMap做出的锁细粒度优化介绍/笔记.doc**

JDK 1.7的ConcurrentHashMap，稍微是有一些差距的

Segment，16个Segment，16个分段，每个Segment对应Node[]数组，每个Segment有一把锁，也就是说对一个Segment里的Node[]数组的不同的元素如果要put操作的话，其实都是要竞争一个锁，串行化来处理的

锁的粒度是比较粗的，因为一个Node[]数组是一把锁，但是他有多个Node[]数组

JDK 1.8的ConcurrentHashMap，优化了，锁粒度细化，他是就一个Node[]数组，正常会扩容的，但是他的锁粒度是针对的数组里的每个元素，每个元素的处理会加一把锁，不同的元素就会有不同的锁

大幅度的提升了多线程并发写ConcurrentHashMap的性能，降低了锁的冲突

读这个事情，是不加锁，他volatile读，保证说你读到的是当前最新的数据，内存屏障和硬件级别的原理，都给大家讲解过了

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/155\_案例实战：slave节点向主备两个master节点注册的机制介绍/笔记.doc**

Hadoop HDFS（分布式存储系统），NameNode其实分为主备两个节点的，各个DataNode都是会在启动的时候，会向两个NameNode都会去进行注册。也就是说让NameNode感知到集群里有哪些datanode的存在

DataNode作为数据存储节点，启动的时候会向主备两个NameNode都进行注册，模仿hadoop的源码来写一下

CountDownLatch来使用一下，让大家找找感觉

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/157\_CyclicBarrier：将工作任务给多线程分而治之的并发组件/笔记.doc**

下一个并发组件，很少用

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/170\_图解剖析JDK 1.7 HashMap并发环境下死循环之环形链表/笔记.doc**

第一步：多线程并发操作的时候，会导致一个环形链表

两个线程此时同时进来要插入一个元素，如果此时同时触发了resize的过程，同时进行hashMap的扩容

void transfer(Entry[] newTable) {

Entry[] src = table;

int newCapacity = newTable.length;

for (int j = 0; j < src.length; j++) {

Entry<K,V> e = src[j];

if (e != null) {

src[j] = null;

do {

Entry<K,V> next = e.next; // e = <k2,v2>，next = <k1,v1>

// 神奇的现象发生了，e = <k1,v1>，next = null

// 线程1执行到这里，此时e = <k1,v1>，next = <k2,v2>

// 此时，cpu发生上下文的切换，此时切换到了线程2来执行

// 线程2处理完了链表之后，cpu切换回线程1来执行

int i = indexFor(e.hash, newCapacity);

e.next = newTable[i];

newTable[i] = e;

e = next; // e = <k1,v1>

} while (e != null);

}

}

}

如果被两个线程同时来执行会如何？？？

JDK 1.7的hashmap扩容的源码，两个线程同时执行的时候，是如何导致环形链表的。某一个线程会把自己的newTable赋值给table，作为hashMap内部的数组

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/172\_ConcurrentHashMap：分段加锁提升并发性能的思想实践/笔记.doc**

锁优化的思路，拆分锁的粒度，类似于分布式锁优化的实践，把一份数据拆分为多份数据，对每一份数据片段来加锁，这样就可以提升多线程并发的效率

HashMap底层不就是一个数组的数据结构么？如果你要完全保证里的并发安全，如果你每次对数组做一些put、resize、get的操作的时候，你都是加锁，synchronized好了，此时就会导致并发性能非常的低下

所有的线程读写hashmap的过程都是串行化的，hashtable，就是采用的这种做法

读写锁，大量的读锁和写锁冲突的时候，也会导致多线程并发的效率大大的降低，也不行

ConcurrentHashMap，分段加锁，把一份数据拆分为多个segment，对每个段设置一把小锁，put操作，仅仅只是锁掉你的那个数据一个segment而已，锁一部分的数据，其他的线程操作其他segmetn的数据，跟你是没有竞争的

segment01，加锁；segment02，segment03。多个线程并发操作hashmap的效率

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/176\_ConcurrentHashMap源码剖析之synchronized分段加锁机制/笔记.doc**

一个数组，初步的分段加锁思想的体现，就是多个线程对同一个位置的元素赋值的时候是走的CAS操作，保证说对数组的这个位置的元素是线程安全的，数组有多少个元素，相当于就是有多少把小锁

因为只有多个线程竞争赋值同一个位置的元素，才会有锁的竞争的效果出现

我们先用屁股猜想一下，就是数组的一个位置的元素此时已经有人设置一个值了，我过来，此时只能用链表+红黑树来进行处理，JDK 1.8里的hashmap他是用链表+红黑树，来处理hash冲突的问题

此时就应该对数组的那个位置的元素进行链表+红黑树的处理，把冲突的多个key-value对挂成一个链表或者是红黑树

f就代表了数组当前位置的元素，Node节点

synchronized(f) {

}

这个就是所谓的JDK 1.8里的ConcurrentHashMap分段加锁的思想，淋漓尽致的体现，他仅仅就是对数组这个位置的元素加了一把锁而已

分段加锁呢？

数组如果有16个元素，最多就是对每个位置的元素加一把锁，synchronized锁，这个位置的元素直接加了一个锁，只有一个线程可以进锁来进行这个位置的链表+红黑树的处理，这个就是分段加锁

JDK 1.8的ConcurrentHashMap，并没有采取的是对整个数组加一把大的锁，而是对数组每个位置的元素都有一把小锁，synchronized，保证对数组的每个位置的元素同一时间只能有一个线程进入处理链表/红黑树就可以了

数组的中的每一个元素，无论是null情况下来赋值（CAS，分段加锁思想），有值情况下来处理链表/红黑树（synchronized，对元素本身加锁，更加是分段加锁思想），都是对数组的一个元素加锁而已

你的数组有多大，有多少个元素，你就有多少把锁

大幅度的提升了整个HashMap加锁的效率，对不对各位同学，考虑一下

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/151\_CountDownLatch：同步等待多个线程完成任务的并发组件/笔记.doc**

20讲左右的时间，java并发包下的同步组件

volatile、synchronized、CAS、读写锁、锁优化、生产锁故障、同步组件（次要的）、并发集合、线程池

并发编程设计模式，不一定会放在这里来讲解，会放到io、网络、netty之后，大幅度的重构和升级微服务注册中心的项目，大量的来运用并发编程设计模式

并发组件，他们全部都依赖于前面讲解的AQS，AQS是并发包下面的一个基础以及核心的组件，读写锁，重入锁，并发组件，都是基于AQS来实现的

如果你的一个线程启动了多个线程来执行一些任务，此时你的这个线程需要同步阻塞等待那些线程都执行完毕了，才可以继续往下走，此时可以用CountDownLatch

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/166\_案例实战：基于Semaphore实现分布式计算系统的推测执行机制/笔记.doc**

分布式计算系统，什么叫做推测执行。

比如说你有10台机器，每台机器上存储了100G的数据，你现在可以分配10个计算任务，每个计算任务分散在一个机器上负责计算 那台机器上的100G的数据，分布式计算。但是有一种可能，可能你在计算的过程中

可能会出现说某台机器上的计算任务执行的过慢，有这种可能，机器上的磁盘有点故障或者问题，或者是那台机器上当前cpu负载特别高，就导致计算任务运行的比较慢

推测执行机制，hadoop分布式存储的架构的话，肯定数据是有冗余存储的副本的，10台机器，每台机器是100G的数据。但是一份数据拆分为了10份，数据副本的概念，同一份数据可能在多台机器上都有副本

每台机器上，是有300G的数据，每个100G的原始数据都有另外2个副本，每个副本也是300G的数据，每一份数据，是100G原始数据 + 200G的副本数据，散落在10台机器上，每台机器上是有300G的数据

副本1在机器1，副本2在机器3，副本3在机器5

可以仅仅只是分配一个计算任务到机器1上去来进行计算，万一上面的计算任务特别的慢呢？内存里在进行长时间的fullgc导致计算卡顿。可以分配一个计算任务的多个副本到多台机器上去执行

分配3个计算任务到机器1、机器3、机器5上面去，每个计算任务都针对同一个副本来进行计算，看谁先计算完毕，就采用谁的计算结果。这套机制在线上负载特别高的情况下，是可以在一定的程度上提升分布式计算作业的性能的

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/152\_CountDownLatch源码剖析之如何基于AQS实现同步阻塞等待（一）/笔记.doc**

剖析一下CountDownLatch源码

protected int tryAcquireShared(int acquires) {

return (getState() == 0) ? 1 : -1;

}

刚开始的时候，state = 2，此时调用这个方法，state != 0，所以会返回-1

此时会将当前线程放入AQS的等待队列中，入队去等待，而且这里的话呢，图不用画，不过就是在AQS的队列里，加入了main线程

此时的话，main线程会直接通过park操作挂起，阻塞住，不能干任何事情了，就等待别人把他从队列里来唤醒

（1）state != 0

（2）将main线程封装为一个node，加入AQS的等待队列

（3）调用LockSupport.park()操作，挂起main线程

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/154\_案例实战：分布式存储系统的HA高可用架构原理介绍/笔记.doc**

分布式存储系统的案例，hadoop，《兄弟，用大白话告诉你小白都能看懂的hadoop架构原理》，自己去公众号里翻一下历史记录，找到这篇文章，仔细看几遍。edits log机制，active-standby两个namenode互相成主备，复制edits log数据。

两个namenode他们的数据是一模一样的，如果主节点宕机了，备节点会自动跟上去，成为主节点，对外提供服务。

分布式存储系统的HA高可用架构

DataNode是负责分布式数据存储的，NameNode是负责管理集群的元数据的，控制整个集群，你有一个1TB的大文件，此时NameNode会控制说把这个大文件用哪10台DataNode所在的机器来分别存储一部分

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/179\_ConcurrentHashMap源码剖析之查询操作是否涉及锁？/笔记.doc**

put，如何CAS加锁赋值，synchronized分段加锁处理链表/红黑树，扩容两倍大小的数组，核心的原理跟hashmap是一样的，CAS和synchronized分段加锁的思想体现，加锁的粒度是数组里的每个元素

get，涉及到锁，size，涉及到锁

他对读这个事情，是不用锁的，volatile底层硬件级别的原理，思考一下，volatile读操作的话，load屏障，绝对是在读取数据的时候，一定会嗅探一下，探查一下无效队列，如果说某个数据被别人修改过了

此时必须立马过期掉本地高速缓存里的缓存数据，invalid（I），然后再读的时候，就需要发送read消息到总线，从其他线程修改修改这个值的线程的高速缓存里，必须这个加载到最新的值

不加锁，但是他通过volatile读，尽可能给你保证说是读到了其他线程修改的一个最新的值，但是不需要加锁

记录了数组的每个位置挂载了多少个元素，每个位置都可能挂的是链表或者是红黑树，此时可能一个位置有多个元素，size方法，是帮助你去读到当前最新的一份数据，通过volatile的读操作

但是因为读操作是不加锁，他不定根可以保证说，你读的时候就一定没人修改了，很可能是你刚刚读完一份数据，就有人来修改

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/158\_CyclicBarrier源码剖析之如何基于AQS实现任务分而治之（一）/笔记.doc**

源码，CyclicBarrier初始化源码

线程1：

（1）ReentrantLock，加锁，保证多线程并发安全

（2）count = 3，--count = 2

（3）Condition.await()，底层，其实是释放了当前的lock锁，触发了把当前线程加入condition等待队列里，挂起当前线程

线程2：

（1）ReentrantLock，加锁，保证多线程并发安全

（2）count = 2，--count = 1

（3）Condition.await()，底层，其实是释放了当前的lock锁，触发了把当前线程加入condition等待队列里，挂起当前线程

线程3：

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/151~181资料/161\_案例实战：API服务中对多个接口并发调用后统一合并数据再返回/笔记.doc**

CyclicBarrier，给大家举一个例子，就是我们之前线上系统，很少，如果你去看一些知名的开源项目的话，也是一样的，用的很少

API服务，后面可以对应很多个其他的服务，他相当于是聚合一下多个服务的接口，但是他不是网关的作用

订单中心 -> 订单子系统 -> 订单相关的服务 -> 价格服务 + 订单服务 + 订单统计服务

订单中心其实是可以对外暴露一个统一的订单API服务，他暴露了订单中心里所有的订单相关的接口，他后台可以聚合各种其他服务的接口

有一个接口，查询订单列表接口，对应了底层的3个服务，价格服务，订单服务，订单统计服务，分别调用一个接口，查询出来3份数据，然后合并成一个完整的数据，返回给前端去进行展示

有一个页面，要展示什么数据，此时难道你要前端自己请求3个服务的接口，然后自行拼接数据吗？API聚合服务，提供同一的接口，聚合后台多个服务的接口，便于其他人来调用你的接口

API服务，一个接口，要调用后台3个服务的3个接口，一个服务一个服务的同步调用呢？还是开启3个线程，并发的去执行请求，同时去调用3个接口，等3个接口的数据都返回了之后，就合并成一份完整的数据

最后返回给前端

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/54~73资料/70\_案例实战：基于AtomicReference优化客户端缓存注册表/笔记.doc**

AtomicReference优化了一下，多个地方多个线程同时对一个对象变量的引用进行赋值的时候，可能导致的并发冲突的问题，就用AtomicReference的CAS操作来解决了，而没有使用加锁的重量级的方式

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/54~73资料/55\_double check布下小坑的说明以及volatile的真实作用/笔记.doc**

64kb，算错了，算成了64mb，人脑 不等于 电脑，短路，或者一下子惯性思维

double check给大家说volatile的作用，其实当时本意，大脑惯性思维了一下，要给大家说明白volatile对可见性的保证

volatile写还没全部执行结束，就自动让另外一个线程来执行volatile读，内存屏障，具体扣里面的细节，那就要写各种jvm指令，太麻烦了

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/54~73资料/67\_案例实战：客户端实现启动时拉取全量注册表/笔记.doc**

全量拉取注册表的代码就搞定了

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/54~73资料/71\_案例实战：基于AtomicStampedReference解决注册表缓存ABA问题/笔记.doc**

ABA问题，反复修改几次，重新修改回了最早的值，你以为没改过，其实已经中间修改过多次了，AtomicStampedReference

一开始stamp = 0

获取到一个Applications对象（01），还有一个stamp = 0邮戳

尝试CAS，发现stamp = 0与实际的stamp = 2不符合，CAS操作失败

再次获取到Applications对象（01），此时stamp = 2，再次尝试CAS操作成功

此时别的线程，反复的修改了几次这个Applications对象，比如Applications对象（02），stamp = 1；又修改回了Applications对象（01），但是此时stamp = 2

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/54~73资料/62\_案例实战：基于AtomicLong优化服务注册中心的心跳计数器/笔记.doc**

我们来看看，Atomic原子类在项目里来用一下，AtomicInteger、AtomicLong，AtomicBoolean，我们来用AtomicLong来优化一下咱们的服务注册中心内部的心跳计数器

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/54~73资料/59\_AtomicInteger源码剖析：无限重复循环以及CAS操作/笔记.doc**

private static final long valueOffset;

static {

try {

valueOffset = unsafe.objectFieldOffset

(AtomicInteger.class.getDeclaredField("value"));

} catch (Exception ex) { throw new Error(ex); }

}

private volatile int value;

类初始化的时候，来进行执行的，valueOffset，value这个字段在AtomicInteger这个类中的偏移量，在底层，这个类是有自己对应的结构的，无论是在磁盘的.class文件里，还是在JVM内存中

大概可以理解为：value这个字段具体是在AtomicInteger这个类的哪个位置，offset，偏移量，这个是很底层的操作，是通过unsafe来实现的。刚刚在类初始化的时候，就会完成这个操作的，final的，一旦初始化完毕，就不会再变更了

int l;

do {

// 会用getIntVolatile方法

// 从AtomicInteger对象实例，根据valueOffset偏移量，知道了value这个字段的位置

// 去获取到当前的value的值

l = getIntVolatile(paramObject, paramInt1);

} while(!compareAndSwapInt(paramObject, paramInt1, l, l + paramInt2));

return l;

// compareAndSwapInt()，CAS方法

// 他会拿你刚刚获取到的那个l的值，他认为当前的value的值

// 去跟底层当前目前AtomicInteger对象实例中的value的值去进行比较，如果是一样的话

// 这就是compare的过程

// 就会set的过程，也就是将value的值给设置为：l（之前拿到的值） + 1（递增的值）

// 如果l（获取到的值），跟AtomicInteger + valueOffset获取到的当前的值，不一样的话

// 此时compareAndSwapInt方法就会返回false

// while循环里拿到的是false的话，就会自动进入下一轮的循环

// 如果是成功的话，会返回一个l的值，是递增1之前的一个旧的值，所以会在外层方法中加1返回，告诉你当前累加1之后最新的值

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/54~73资料/68\_案例实战：客户端实现定时拉取增量注册表到本地合并/笔记.doc**

做完了全量拉取，这一讲就是增量拉取

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/54~73资料/65\_案例实战：服务注册中心的最近更新服务实例队列实现/笔记.doc**

从这一讲开始，我们要来实现服务注册表的增量拉取的机制

每隔30秒就拉取一次全量的注册表呢？就是说这个注册表可能并不是每一条数据都会变化的，每隔30秒，可能就少数几个服务实例会出现这个变化，并不需要每隔30秒吭哧吭哧把所有的注册表的数据都拉取过来

如果说你的服务实例有几万个，那么你的服务注册中心的注册表里可能有几万条数据，每次你得30秒拉取几万条数据的服务注册表，很坑爹的，对整个网络的开销，每次对服务注册中心的性能压力，都很大

服务启动的时候，先是全量拉取一次服务注册表，接着每隔30秒，拉取一次增量的注册表，这样的话，每隔30秒就是拉取最近30秒变化的少量的服务实例的信息即可

增量拉取注册表如何来实现呢？你搞一个队列，队列里面就是存放最近3分钟有变化的服务实例

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/54~73资料/66\_案例实战：服务注册中心提供全量和增量拉取注册表的接口/笔记.doc**

拆分了两个拉取注册标的接口，一个是全量，一个是增量

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/54~73资料/73\_Atomic原子类其他API的介绍以及自己学习的建议/笔记.doc**

AtomicIntegerArray => 数组类的API，就是你可以对数组元素CAS原子性的递增

AtomicIntegerFieldUpdater => API，百度一下

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/54~73资料/56\_i++和AtomicInteger之间的差别分析以及使用介绍/笔记.doc**

Java的原子类技术体系来讲一下，开多个线程来进行工作，以及基本的控制，interrupt，sleep；voaltile，一些标志位；synchronized，原子性，wait + notify线程管控的；Atomic系列的类，来优化一些加锁的并发性能

每一块在讲的时候，都是不断的完善微服务注册中心的代码

i++和AtomicInteger

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/54~73资料/60\_AtomicInteger源码剖析：底层CPU指令是如何实现CAS语义的/笔记.doc**

最最底层，用了一个native方法，不是java写的，走的是底层的c代码，可以通过发送一些cpu的指令，来确保说CAS的那个过程，绝对是原子的，具体是怎么来实现呢？以前的cpu会通过一些指令来锁掉某一小块的内存，后来会做了一些优化，他可以保证仅仅只有一个线程在同一时间可以对某块小内存中的数据，做CAS的操作

compare -> set，这是一系列的步骤，在执行这个步骤的时候，是每个线程都是原子的，有一个线程在执行CAS一系列的比较和设置的过程中，其他的线程是不能来执行的

cpu指令来实现

cpu会通过一些轻量级的锁小块内存的机制来实现

保证整个并发的性能要好的多

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/54~73资料/69\_案例实战：增量合并注册表后进行校验与全量纠正/笔记.doc**

案例代码写的差不多了，这里有一堆的问题，可以用Atomic系列的API来进行优化和解决的，接下来就一点一点的来做

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/54~73资料/58\_AtomicInteger源码剖析：仅限JDK内部使用的Unsafe类/笔记.doc**

Unsafe类是在JDK底层的一个类，而且的话人家限制好了，不允许你去实例化他以及使用他里面的方法的，首先人家的构造函数是私有化，不能自己手动去实例化他，其次，如果用Unsafe.getUnsafe()方法来获取一个实例

是不行的，在那个源码里，他会判断一下，如果当前是属于我们的用户的应用系统，识别到有我们的那个类加载器以后，就会报错，不让我们来获取实例

JDK源码里面，JDK自己内部来使用，不是对外的

Unsafe，封装了一些不安全的操作，指针相关的一些操作，就是比较底层了，主要就是Atomic原子类底层大量的运用了Unsafe

（1）volatile value

（2）Unsafe：核心类，负责执行CAS操作

（3）API接口：Atomic原子类的各种使用方式

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/54~73资料/64\_案例实战：基于LongAdder的分段CAS机制优化心跳计数器的效率/笔记.doc**

LongAdder，替代AtomicLong，我们完全可以对心跳计数器来使用LongAdder

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/54~73资料/57\_AtomicInteger中的CAS无锁化原理和思路介绍/笔记.doc**

AtomicInteger原子类底层的一些原理，AtomicLong、AtomicBoolean、AtomicReference、LongAdder，等等，大部分的技术我们都会结合分布式中间件项目实战来体验如何来在真实的场景下来使用

无锁化，乐观锁

判断此时此刻是否是某个值，如果是，则修改，如果不是则重新查询一个最新的值，再次执行判断，这个操作叫做CAS，Compare and Set

Atomic原子类底层核心的原理就是CAS，无锁化，乐观锁，每次尝试修改的时候，就对比一下，有没有人修改过这个值，没有人修改，自己就修改，如果有人修改过，就重新查出来最新的值，再次重复那个过程

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/54~73资料/54\_结合java内存模型了解synchronized对可见性的保证/笔记.doc**

主要是借着volatile解决可见性问题的思路，给大家说了一下，但是其实因为当时没有讲解synchronized原理，所以说当时就没有给大家说的更加的清晰一些，但是就是简单的提了一嘴而已，几分钟

synchronized本身是可以保证可见性的

double check本身，如果你不给instance单例加上volatile的话，出现的问题并不是可见性的问题，synchronzied是保证原子性，可见性

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/54~73资料/63\_Java 8的LongAdder是如何通过分段CAS机制优化多线程自旋问题的/笔记.doc**

Java 8提供的一个对AtomicLong改进后的一个类，LongAdder

大量线程并发更新一个原子类的时候，天然的一个问题就是自旋，会导致并发性能还是有待提升，比synchronized当然好很多了

分段迁移，某一个线程如果对一个Cell更新的时候，发现说出现了很难更新他的值，出现了多次自旋的一个问题，如果他CAS失败了，自动迁移段，他会去尝试更新别的Cell的值，这样的话就可以让一个线程不会盲目的等待一个cell的值

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/54~73资料/72\_案例实战：基于AtomicLong确保多线程拉取注册表版本不错乱/笔记.doc**

另外一个问题

其实要灵活的运用CAS操作来解决，这个是spring cloud eureka源码里的一个亮点，eureka-client端的缓存注册表，就是用AtomicReference来解决的多线程并发赋值的操作的原子性

其次他灵活运用AtomicLong来解决了多线程并发拉注册表，可能会导致注册表版本混乱的问题，他是如何来解决的？

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/54~73资料/61\_Atomic原子类体系的CAS语义存在的三大缺点分析/笔记.doc**

1、ABA问题：如果某个值一开始是A，后来变成了B，然后又变成了A，你本来期望的是值如果是第一个A才会设置新值，结果第二个A一比较也ok，也设置了新值，跟期望是不符合的。所以atomic包里有AtomicStampedReference类，就是会比较两个值的引用是否一致，如果一致，才会设置新值

假设一开始变量i = 1，你先获取这个i的值是1，然后累加了1，变成了2

但是在此期间，别的线程将i -> 1 -> 2 -> 3 -> 1

这个期间，这个值是被人改过的，只不过最后将这个值改成了跟你最早看到的值一样的值

结果你后来去compareAndSet的时候，会发现这个i还是1，就将它设置成了2，就设置成功了

说实话，用AtomicInteger，常见的是计数，所以说一般是不断累加的，所以ABA问题比较少见

2、无限循环问题：大家看源码就知道Atomic类设置值的时候会进入一个无限循环，只要不成功，就不停循环再次尝试，这个在高并发修改一个值的时候其实挺常见的，比如你用AtomicInteger在内存里搞一个原子变量，然后高并发下，多线程频繁修改，其实可能会导致这个compareAndSet()里要循环N次才设置成功，所以还是要考虑到的。

JDK 1.8引入的LongAdder来解决，是一个重点，分段CAS思路

3、多变量原子问题：一般的AtomicInteger，只能保证一个变量的原子性，但是如果多个变量呢？你可以用AtomicReference，这个是封装自定义对象的，多个变量可以放一个自定义对象里，然后他会检查这个对象的引用是不是一个。

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~17资料/03\_案例实战：以工作线程模式开启微服务的注册和心跳线程/笔记.doc**

注册的这个过程，就是完全可以开启一个线程去进行注册

还有一个就是时不时的发送心跳，通知微服务注册中心自己现在还活着

带入第一个概念，创建和启动一个线程，如果放在一些书里，或者视频里，都是一些概念的讲解，我们直接上案例

new Thread() {

public void run() {

}

}.start();

public class MyThread extends Thread {

public void run() {

}

}

new MyThread().start();

new Thread(new Runnable() {

public void run() {

}

}).start();

public class MyRunnable implements Runnable {

public void run() {

}

}

new Thread(new MyRunnable()).start();

用Thread，Runnable来封装线程的逻辑也好，你纵观各种开源项目的源码，我觉得用什么的都有，我个人是建议说

我们就用Thread来实现这个线程

两个工程

（1）register-server：负责接收各个服务的请求

是可以独立部署和启动的，启动了以后，他会以一个web工程的方式来启动，启动之后就是监听各个服务发送过来的http请求，注册、心跳、下线

我们到时候就可以认为是有一个main方法直接运行就启动也可以，现在很流行的spring boot，你直接运行main，人家就给你启动一个web服务器，你在工程里，如果使用spring web mvc实现了一个controller接口，就可以接受请求了

（2）register-client：组件，依赖包，各个服务需要引入这个依赖，在服务启动的时候就可以去让register-client来运行，来跟register-server进行通信，比如说完成这个注册，或者是心跳的通知

他不是独立启动的，他其实是一个依赖包，你可以把这个东西打包发布到maven nexus私服里去，你的公司里各个服务，必须依赖这个register-client，然后启动服务的时候，一般会调用regsiter-client的API，创建一个组件，启动这个组件

由register-client组件去跟register-server进行交互

（3）spring cloud eureka

人家Spring Cloud的微服务注册中心，eureka，大概就是这个意思，人家也是分eureka-server，是独立部署和启动的，就是一个web工程；eureka-client，各个服务都需要依赖eureka-client，服务启动就创建一个eureka-client实例；eureka-client帮各个服务跟eureka-server进行通信，注册、心跳、下线

（4）工作线程

我们启动了一个jvm进程，main线程，RegisterClientWorker线程

main线程负责启动了RegisterClientWorker线程，其实干完这些事情以后，main线程就结束了，结束了以后但是jvm进程不会退出？为什么呢，有一个工作线程，就是RegisterClientWorker线程一直在运行

所以jvm进程是不会退出的，会一直存在

只要有工作线程一直在运行，没有结束，那么jvm进程是不会退出的

大家一定要理解这个事情

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~17资料/04\_案例实战：微服务注册中心的服务注册功能实现/笔记.doc**

咱们不要玩儿概念，直接上案例，在案例里学东西

微服务注册中心里面，大家都记得我说的吧，各个服务会注册，此时会将自己的信息加入到一个内存的注册表里去，然后呢各个服务在运行的过程中，还会不断的发送心跳通知注册中心说，自己还活着

注册中心的话，就需要开启一个后台线程，不断的扫描和监控注册表里各个服务的心跳发送时间，如果超过一定时间还没发送心跳的话，那么不好意思，直接就是宣告这个服务死亡，从注册表里摘除掉

所以通过这个业务案例，我们就可以来玩儿一下开启线程的方法

一般是两种方式，要不然就是搞一个Thread类，重写里面的run()方法，然后创建一下实例后start()启动这个线程；要不然就是搞一个Runnable接口实现类，封装运行逻辑，然后传入Thread类构造

其实两种方式都可以，平时用的都很多，但是我一般比较喜欢用直接重写Thread的run方法的方式，这样子这个线程的类名称都可以自己来定，在系统里划分组件单元比较清晰

那我们就把这个案例来写一下

register-server，一般是以web工程的模式来启动的，他会类似于spring web mvc提供一些http的接口，让各个服务的register-client组件可以发送http请求到他这里来，当然也可以通过rpc的模式来做

在这个工程里面，给大家讲解一下，这种东西一般可以单拉一个工程出来，专门做一个叫做register-common的工程，也就是其他工程都需要依赖的一些公共的东西，类，可以 放在这个工程里面

我们在并发课程的案例里面，是不会涉及到网络的一些东西，IO，在后面是一个单独的课程，网络、IO、Netty一块儿讲，讲完了以后，就是把网络这块的东西加入微服务注册中心里面去，把整个微服务注册中心，做成一个完整的项目

仔细读一遍Spring Cloud Eureka的源码，按照他的源码，把微服务注册中心的核心机制全部手写实现一遍

我觉得大家对底层技术的开发就会非常的有把握了

每隔60秒运行一次来检查，如果某个服务超过90秒还没更新心跳来续约，那么就摘除这个服务实例

咱们用真实的分布式系统的项目，拆解大量的案例，来逐步逐步的跟，小孩子玩儿乐高一样，将所有的并发的知识点和技术，融入到项目里去，每个技术实现一个机制或者功能，就是一个案例，这就是我的这么一个并发课程的教学的思路

我觉得这种学习并发的模式，是最好的，效果是最好的，纯理论的教学，仅仅能够让你知道并发里的一些概念和技术，但是你怎么来用，其实你并不知道

咱们用了大量的时间，引入了项目案例，开发，各种启动线程，通过这两个案例，我相信大家对线程是什么东西，一般怎么来用，理解的非常深刻了

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~17资料/02\_案例引入：微服务注册中心系列案例背景引入/笔记.doc**

咱们这个并发课程系列里会有大量的案例，基本各种技术都是案例驱动，因为现在并发编程最大的学习痛点，就是缺少案例，各种书籍、博客、文章、视频，全部都是干讲概念，干讲API，有时候会有少数几个小demo

手写一个简易的连接池

营业厅叫号程序

所以很多人学了并发编程一大堆的概念，不知道怎么用，感受不到

大部分的人工作的时候，基本上来说，大家特别是做java开发的，并发编程很少用到，传统IT公司， 或者互联网公司，做一些业务系统，crud，业务，并发、IO、网络，这些底层的技术，很少用到

你出去面试，大厂，大互联网公司，都会深挖面试并发这块的内容

案例、大量的案例、脱胎于真实的复杂分布式系统的案例

并发编程这个东西，用在哪里？底层分布式系统，底层分布式中间件，微服务架构的Eureka注册中心，大数据里面的一些系统，大量用到了并发编程的一些东西，大数据系统本质都是一些复杂的分布式系统

复杂的分布式大数据系统里面，大量用到了并发、集合、IO、网络的一些东西

dubbo、RocketMQ

99%的工程师都不是做分布式系统开发的，大部分的工程师都是做业务系统开发的，crud，主要写web系统，spring mvc、用一用spring cloud等等这些技术

我这里的课程，引入的大量的案例，系列案例，微服务注册中心是一个系列案例，分布式存储系统也是一个系列案例，我会通过这些真实的系列案例出发， 把真实的分布式系统底层架构设计中，如何用到并发编程里的方方面面

把那些场景，抽出来，做成一个一个的案例，让大家体会在底层分布式系统里，并发是怎么来用的，通过这些案例，其实是很活生生很真实的，大家可以更好的体会并发是怎么来用的，第一个是开源的Spring Cloud里的微服务注册中心，Eureka

第二个是开源的大数据系统，分布式存储系统，HDFS

架构班课程正在讲，java架构课程，已经把Spring Cloud源码都剖析完了；大数据课程，正在剖析HDFS源码

在这个情况下，我觉得其实用这两个开源项目，分布式系统中的场景来作为大量的系列案例，给大家来讲解并发是如何使用的，我觉得是很好的

我会用最最言简意赅的语言，给大家来描述一下那两个开源项目的核心原理，站在这个背景下我们来做案例，基于并发手写一些他们里面的一些机制这样子

1、你一定可以很好的感悟到并发编程的技术，在大量的底层分布式系统里面是如何来使用的，完全脱离开仅仅是学会了一大堆的理论和概念的程度

2、建议，并发编程的课，最好的学习效果，是配合Sprnig Cloud源码 + HDFS源码两个课，一起来学，如果你把那两个源码课也学了，并发课也学了，你出去面试的时候，人家问到你任何一个并发的概念

出去面试，人家问到你并发的内容，volatile关键字，你不要光是讲原理，然后呢，兄弟，我看过hdfs的源码，edits log机制，在这个机制里面，多线程并发写edits log的时候，是如何通过volatile关键字来控制一个读写开关的

咱们这里就先引入一个未来会成为系列性的这么一个案例：微服务注册中心。之所以引入这个案例，最主要是之前架构班课程里，Spring Cloud源码都剖析过了，正好用里面的微服务注册中心作为案例，效果很好。

还没看过Spring Cloud源码的同学也没关系，我们这里大概说一下这个微服务注册中心的整体思路就ok了，这个都可以。

大体上来说，就是在一个微服务系统中，有大量的微服务，各个服务之间要互相调用啊，但是服务之间正常是不知道对方在哪里的，这个时候怎么办呢？就需要一个微服务注册中心，各个服务都把自己的地址注册过去，包括机器、端口等等，然后其他服务可以从注册中心拉取注册表，感知到其他服务的存在

而微服务注册中心就是在内存里维护的这个注册表数据，那么大家想，这个注册表在内存里，有服务注册和下线的时候都要被写，然后其他大量的服务要来读这个内存注册表数据结构，所以是不是会存在多线程并发读写冲突的问题？

这里就有大量的并发编程、锁冲突、锁优化等问题使用的空间了

我就会脱胎于Spring Cloud微服务注册中心，Eureka来给大家讲解一系列的并发实战的案例，告诉大家在一个微服务注册中心里，这个并发的技术是如何来运用的

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~17资料/16\_对一个线程执行interrupt到底会发生些什么事情？/笔记.doc**

给说说，interrupt这个东西是怎么来用的

interrupt这个东西的意思是什么呢？先给大家演示了第一种情况，如果你是while循环，可以判断如果没有被中断，那么就正常工作，如果别人中断了这个线程，那么while循环的条件判断里，就会发现说，isInterrupted，被中断了

被中断了以后，你的while循环发现了，就会退出循环，这个线程就终止了

interrupt打断一个线程，其实是在修改那个线程里的一个interrupt的标志位，打断他以后，interrupt标志位就会变成true，所以在线程内部，可以根据这个标志位，isInterrupted这个标志位来判断，是否要继续运行

并不是说，直接interrupt一下某个线程，直接就不让他运行了

还有一个更加常见的用法，就是说什么呢？打断一个线程的休眠，特别常用的一个组合，下一讲要给大家来讲解的

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~17资料/14\_解释一下从0开始手写微服务注册中心项目的计划/笔记.doc**

主要是基于spring cloud eureka和hadoop hdfs两个开源项目，来抽取大量的案例，案例驱动来讲解并发的各种技术，各种技术不要干讲，直接用在底层分布式系统里面。eureka是微服务注册中心，hdfs是分布式存储系统

我觉得我们就专注把eureka这种微服务注册中心，这一个项目给做好，不是抽取大量案例来做，而是通过做大量的系列的案例，最终从0开始不断迭代，开发出来一个完整的微服务注册中心，这种分布式系统

大项目驱动，每次学完一块并发的技术和知识以后，就不断的运用到分布式微服务注册中心里面去实战，开发，落地，让大家学以致用，等到并发系列课程全部结束了以后，你会发现，我们已经实现了微服务注册中心的大部分的功能

然后的话，我们会开始IO + 网络 + Netty的课程，那个课程可以跟这个并发的课程如出一辙，直接接着做这个大项目，直接继续在这个分布式微服务注册中心里面加入大量的东西，完成磁盘IO、网络通信相关的开发

最后，我们把jDK底层源码系列，集合、并发、IO、网络这些东西都全部学完，相当于就是我们也动手从0开始开发出来了一个大型的分布式微服务注册中心，这个东西其实是一个底层中间件的开发

一边学习底层技术，一边就从0开始动手完成了一个底层分布式系统 / 中间件的大项目的开发，到时候你出去面试找工作的时候，人家聊底层的技术，并发、集合、网络、IO、netty，你直接可以跟人家一方面聊聊开源的spring cloud eureka的源码，另外一方面，就是说可以跟人家说

你们是自己纯自研了微服务注册中心，中间件，然后在公司里来用，替代了开源的eureka，是有很多的问题，我们到最后也会再次来重读eureka的源码，分析eureka架构里面的一些问题，服务发现和下线的时效性太差了，自我保护机制有bug，还有很多其他的问题，我们都会在自研版本的微服务注册中心里解决这些开源项目的架构问题

达到这个效果，学习底层技术的效果，perfect，完美

我觉得这个思路很好，我们完全可以做到，spring cloud eureka的源码剖析都讲过了，大家如果去看过，都理解了开源项目的源码，现在一边学习底层技术，一边自研微服务注册中心，我觉得问题不大

集合、并发、网络、IO、netty，完美的一边学一边动手实战，真正掌握了底层技术是怎么用的，出去面试，你的竞争力太强了，深入阅读过集合、并发、Netty的源码，深入阅读过spirng cloud eureka的源码，自研过替代开源eureka的分布式微服务注册中心，做过架构改造和升级，等等

专注于一个大项目，手工自研微服务注册中心，工业级，大型的，分布式系统的开发，绝对不是demo，这套实现完以后，如果大家有兴趣的话，完全可以组成一个开源社区，大家一起写大量的测试用例，部署真实机器来测试，如果ok的话，跟spring cloud、spring boot整合一下，就可以在你公司里用的，效果一定会比eureka要好

可能会带少量的其他案例，就是真的是仅仅是一些小的案例，有些东西如果没法在大项目里实践的话，那么就用小案例来替代一下

hadoop hdfs就暂时搁置掉这个东西引入这个课程的计划

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~17资料/01\_5分钟告诉你线程是什么以及并发编程又是什么/笔记.doc**

最最言简意赅的语言，先解释第一个概念，线程是什么东西？

public class HelloWorld {

public static void main(String[] args) {

System.out.println(“Hello World......”);

new Thread() {

public void run() {

System.out.println(“另外一个线程干的事儿......”);

}

}.start();

// 还有一些其他的代码

}

}

咱们就来分析一下这段代码

如果说你要执行这段代码，是不是直接运行这个main方法，只要是学过线程的同学，都应该知道吧，执行main方法相当于其实是启动一个jvm进程

jvm进程里，是有很多线程的，首先第一个线程，你能看到的线程就是main线程

main线程就是负责执行你的main方法里的那些代码，比如说执行System.out.println这行代码，打印一些东西出来，就是干这个事儿的

只要你的线程执行完了这段代码之后，其实jvm进程他也就退出了

启动一个java的系统，通过执行一个main方法，java -jar这样的命令来启动，jvm进程，里面是有线程的，main线程，负责执行main方法里的代码

如果main线程执行完了以后，jvm进程默认就会直接退出

多线程并发编程是个什么意思呢？

一个jvm进程里，你除了人家默认给你开启的这个main线程，你还可以在main线程里开启别的线程，比如说上面，你可以通过Thread类开启别的线程，别的线程是跟main线程同时在运行的

没有先后顺序，多线程并发运行的时候，本质是CPU在执行各个线程的代码，一个CPU会有一个时间片算法，他一会儿执行main线程，一会儿执行Thread线程，看起来两个线程好像是在同时运行一样

只不过CPU执行每个线程的时间特别短，可能执行一次就几毫秒，几微妙，你是感觉不出来的，看起来好像是多个线程并发在运行一样

第二个点，什么是并发编程？

一句话：用多线程来编程，实现复杂的系统功能，让多个线程同时运行，干各种事情，最终完成一套复杂系统需要干的所有的事儿

1、控制多线程实现系统功能

2、Java内存模型以及volatile关键字

3、线程同步以及通信

4、锁优化

5、并发编程设计模式：基于多线程实现复杂系统架构

6、并发包以及线程池

7、案例、还是案例、大量的案例（脱胎于真实的复杂分布式系统）

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~17资料/15\_案例实战：基于join实现服务注册线程的阻塞式运行/笔记.doc**

join，概念

main线程里面，如果开启了一个其他线程，这个时候只要你一旦开启了其他线程之后，那么main线程就会跟其他线程开始并发的运行，一会执行main线程的代码，一会儿会执行其他线程的代码

main线程里面开启了一个线程，main线程如果对那个线程调用了join的方法，那么就会导致main线程会阻塞住，他会等待其他线程的代码逻辑执行结束，那个线程执行完毕，然后main线程才会继续往下走

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~17资料/05\_案例实战：微服务注册中心的心跳续约功能实现/笔记.doc**

咱们不要玩儿概念，直接上案例，在案例里学东西

微服务注册中心里面，大家都记得我说的吧，各个服务会注册，此时会将自己的信息加入到一个内存的注册表里去，然后呢各个服务在运行的过程中，还会不断的发送心跳通知注册中心说，自己还活着

注册中心的话，就需要开启一个后台线程，不断的扫描和监控注册表里各个服务的心跳发送时间，如果超过一定时间还没发送心跳的话，那么不好意思，直接就是宣告这个服务死亡，从注册表里摘除掉

所以通过这个业务案例，我们就可以来玩儿一下开启线程的方法

一般是两种方式，要不然就是搞一个Thread类，重写里面的run()方法，然后创建一下实例后start()启动这个线程；要不然就是搞一个Runnable接口实现类，封装运行逻辑，然后传入Thread类构造

其实两种方式都可以，平时用的都很多，但是我一般比较喜欢用直接重写Thread的run方法的方式，这样子这个线程的类名称都可以自己来定，在系统里划分组件单元比较清晰

那我们就把这个案例来写一下

register-server，一般是以web工程的模式来启动的，他会类似于spring web mvc提供一些http的接口，让各个服务的register-client组件可以发送http请求到他这里来，当然也可以通过rpc的模式来做

在这个工程里面，给大家讲解一下，这种东西一般可以单拉一个工程出来，专门做一个叫做register-common的工程，也就是其他工程都需要依赖的一些公共的东西，类，可以 放在这个工程里面

我们在并发课程的案例里面，是不会涉及到网络的一些东西，IO，在后面是一个单独的课程，网络、IO、Netty一块儿讲，讲完了以后，就是把网络这块的东西加入微服务注册中心里面去，把整个微服务注册中心，做成一个完整的项目

仔细读一遍Spring Cloud Eureka的源码，按照他的源码，把微服务注册中心的核心机制全部手写实现一遍

我觉得大家对底层技术的开发就会非常的有把握了

每隔60秒运行一次来检查，如果某个服务超过90秒还没更新心跳来续约，那么就摘除这个服务实例

咱们用真实的分布式系统的项目，拆解大量的案例，来逐步逐步的跟，小孩子玩儿乐高一样，将所有的并发的知识点和技术，融入到项目里去，每个技术实现一个机制或者功能，就是一个案例，这就是我的这么一个并发课程的教学的思路

我觉得这种学习并发的模式，是最好的，效果是最好的，纯理论的教学，仅仅能够让你知道并发里的一些概念和技术，但是你怎么来用，其实你并不知道

咱们用了大量的时间，引入了项目案例，开发，各种启动线程，通过这两个案例，我相信大家对线程是什么东西，一般怎么来用，理解的非常深刻了

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~17资料/12\_回头看看：sleep原来可以让线程暂停一段时间/笔记.doc**

微服务存活状态监控线程，其实一般会每次检查完一轮之后，就要停顿几秒钟，此时可以用Thread.sleep()这个方法，指定要等待多少毫秒。但是其实这个东西平时你要指定等待多少毫秒，还挺麻烦的

JDK 1.5之后就引入了TimeUnit这个类，很方便

TimeUnit.HOURS.sleep(1)

TimeUnit.MINUTES.sleep(5)

TimeUnit.SECONDS.sleep(30)

TimeUnit.MILLISECONDS.sleep(500)

如果用TimeUnit的话，你在外面怎么配？你要是配置休眠5分钟，还得加一个单位，代码里要判断一下你休眠的时间单位，如果是分钟，那么还得用TimeUnit.MINIUTE来进行休眠，不太方便

但是我要跟大家说一点，开源项目的话，在线程sleep这块，还是用的最最原始的sleep，因为可以通过毫秒数，动态的传入一个外面配置的一个值

500

30 \* 1000

1 \* 60 \* 1000

30 \* 60 \* 1000

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~17资料/08\_一般不常用到的ThreadGroup是个什么东西？/笔记.doc**

大体上来说，ThreadGroup就是线程组，其实意思就是你可以把一堆线程加入一个线程组里，那关键这个玩意儿有啥好处？好处大概就是，你可以将一堆线程作为一个整体，统一的管理和设置

实际上在java里，每个线程都有一个父线程的概念，就是在哪个线程里创建这个线程，那么他的父线程就是谁。举例来说，java都是通过main启动的，那么有一个主要的线程就是mian线程。在main线程里启动的线程，父线程就是main线程，就这么简单。

然后每个线程都必然属于一个线程组，默认情况下，你要是创建一个线程没指定线程组，那么就会属于父线程的线程组了，main线程的线程组就是main ThreadGroup。咱们来随手写一段代码看看不就得了

在java里面，线程都是有名字的，默认情况下，main线程的名字就是叫main。你其他的其他线程的名字，一般是叫做Thread-0之类的。ServiceAliveMonitor线程的父线程是main线程，默认的线程组也是main线程的线程组，叫做main

然后我们也可以手动创建一个线程组，将线程加入这个线程组中

但是线程组其实也有父线程组的概念，我们创建线程组的时候，如果没有手动指定他的父线程组，那么其实默认的父线程组就是main线程的线程组

这个也随手写一段代码来看看不就得了

现在我们再来看看，假如你要是心血来潮把一堆线程放到线程组里去，然后可以统一管理啥子个东西？

enumerate()：复制线程组里的线程

activeCount()：获取线程组里活跃的线程

getName()、getParent()、list()，等等

interrupt()：打断所有的线程

destroy()：一次性destroy所有的线程

当你真的需要用到他的时候，去查阅jDK的API文档，我觉得是最好的办法

我特别不喜欢的就是视频课程，照着API文档讲，讲一个一个API是怎么用的，那个是怎么用的，你就是学了一大堆的API的使用

文字类的书籍，跟视频类的课程

文字类的书籍能承载的信息量是有限的，语言能承载的信息量远远超出文字，语言可以解释的清楚的东西，比如现场画图什么的，分析啊什么的，远远超出文字

文字最擅长的就是记录的完整性，通过文字记录完整的API使用手册，API使用文档，写一本书，包含这个东西的方方面面，所有的细节

讲课不是这么回事，视频课程，剖析一些底层、源码、实战、现场画图、做项目，语言承载的信息量大，可以把很多书本和文字说不清楚的东西，给你讲清楚

现场敲API怎么用，那个API怎么用，API一大堆的用法，跟实际的真实的大型分布式系统联系不起来

其实有一个比较不错的人在讲并发类的课程，我觉得他很用心，很佩服，但是说实话，他的东西我看过，我个人觉得他是不会讲课的，因为在我讲课的理念来看，一个工程师需要的不是一个照着API文档讲解各种功能、方法怎么用的课程，那些东西，学习的时候大概过一眼就可以了

你需要用的时候，百度一下，或者是查一下API文档，立马就知道怎么回事，API文档都写的非常的清楚

学习，其实是学核心的原理、底层源码、实践经验，这些东西，这是最核心的，否则你就算把各种API的使用手敲1万遍代码，也还是以后会忘，而且我觉得意义不大

所以上面那些东西，我不讲，你就把握住一点：默认线程会加入父线程的ThreadGroup，或者你自己手动创建ThreadGroup，ThreadGroup也有父ThreadGroup，ThreadGroup可以包裹一大堆的线程，然后统一做一些操作，比如统一复制、停止、销毁，等等

大概把握住这个点就ok了

但是跟大家说一下，JDK虽然提供了ThreadGroup，但是一般平时自己开发，或者是很多的开源项目里，ThreadGrdoup很少用，其实如果你要自己封装一堆线程的管理组件，我觉得你完全可以自己写

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~17资料/13\_搜遍开源项目几乎都找不到的冷门yield又是什么东西？/笔记.doc**

担心说某个线程一直长时间霸占着CPU，导致其他的线程很少得到机会来执行，所以设计了一个yield方法，你调用之后，可以尝试说当前线程先别执行了，CPU，兄弟，你可以去执行其他线程了

如果你要用这个方法的话，必须在严格的测试环境下，做大量的测试，验证说，你在你需要的场景下，使用了yield方法，真的可以达到你需要的效果。很多人很少可以正确的使用这个yeild方法，这个方法常见于debug和test场景下的程序

在这样的一些场景下，他可以复现因为锁争用导致的一些bug

他也可以用于设计一些并发控制的工具，比如说在java.util.concurrent.locks包下的一些类

hadoop这个开源项目，大数据领域里炙手可热，最知名的分布式大数据系统，这个东西的源码超过百万行了，我就对这个hadoop的源码尝试搜索过，几乎都找不到yield的使用，仅仅在极个别的测试用例里面，人家有人用过一两次yeild方法

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~17资料/09\_几乎从来不会动手设置的线程优先级又是个啥东西？/笔记.doc**

设置线程优先级，理论上可以让优先级高的线程先尽量多执行，但是其实一般实践中很少弄这个东西，因为这是理论上的，可能你设置了优先级，人家cpu结果也还是没按照这个优先级来执行线程

这个优先级一般是在1~10之间

而且ThreadGroup也可以指定优先级，线程优先级不能大于ThreadGroup的优先级

但是一般就是用默认的优先级就ok了，默认他会用父线程的优先级，就是5

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~17资料/07\_案例实战：以daemon模式运行微服务存活状态监控线程/笔记.doc**

啥是daemon线程，啥是非daemon线程？

简单来说，一般工作线程是非daemon线程，后台线程是daemon线程

默认创建的线程就是非daemon的，我们称之为工作线程，上一讲已经给大家演示过了，如果你的main()方法启动之后就是要开启几个永久无限循环工作的线程，来处理一些请求之类的，比如web服务器

那么那些线程就是工作线程

你的main方法虽然执行完了，但是那些线程因为一直在运行，所以jvm进程是不会退出的，大家上一讲已经看到这个效果了

但是java里还有一个daemon线程的概念，这个意思是说，如果jvm里的工作线程都停止了，比如main线程之类的都执行完了，那么daemon线程就会跟着jvm进程一起退出，不会像工作线程一样阻止jvm进程退出

比如说常见的，假如说微服务注册中心负责接收请求的核心工作线程不知道为啥都停止了，那么说明这个微服务注册中心必须停止啊，结果你的那个监控微服务存活状态的线程一直在那儿运行着，卡着，会导致微服务注册中心没法退出的！因为jvm进程没法结束

所以说针对这种情况，一般会将后台运行的线程设置为daemon线程，如果jvm里只剩下了daemon线程，那么就会进程退出，所有daemon线程一起销毁了，不会阻止jvm进程退出。所以我们应该将微服务存活状态监控的线程，设置为daemon线程，这样如果工作线程都死了，那么jvm也就退出了，daemon线程也销毁了

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~17资料/11\_探索Thread源码（二）：线程启动过程/笔记.doc**

来看看对一个线程进行start的时候，是怎么回事

永远都不能对一个线程多次调用和执行start()方法，这个是不对的

if (threadStatus != 0)

throw new IllegalThreadStateException();

如果你的线程一旦执行过一次以后，那么他的threadStatus就一定会变为非0的一个状态，如果threadStatus是非0的状态，说明他之前已经被执行过了，所以这里会有一个判断，如果你对一个线程多次执行start()方法

人家会抛出一个异常，IllegalThreadStateException，非法的线程状态的异常

group.add(this);

group就是之前给分配的，如果你自己指定了那么就是你自己创建的那个ThreadGroup，否则的话就是你的父线程的threadGroup，这行代码，其实就是将当前线程加入了他属于的那个线程组

private native void start0();

会结合底层的一些代码和机制，实际的启动一个线程

一旦是start0()成功的启动之后，他就会去执行我们覆盖掉的那个run()方法，或者是如果你传入进去的是那个Runnalbe对象，人家就会执行那个Runnable对象的方法

@Override

public void run() {

if (target != null) {

target.run();

}

}

如果你是

new Thread(new Runnable() {

public void run() {

}

}).start();

传递进去了一个Runnable对象，就是在thread类里是target的东西，会判断一下，如果target不为null的话，那么此时就会执行target的run方法。反之，如果你是直接自己用Thread类继承了一个子类的话，那么你会重写这个run()方法，start0()启动线程之后，就会来执行你的run()方法

大家从这里需要注意的几个点：

（1）一旦启动了线程之后，就不能再重新启动了，多次调用start()方法，因为启动之后，threadStatus就是非0的状态了，此时就不能重新调用了

（2）你启动线程之后，这个线程就会加入之前处理好的那个线程组中

（3）启动一个线程实际上走的是native方法，start0()，会实际的启动一个线程

（4）一个线程启动之后就会执行run()方法

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~17资料/17\_案例实战：基于interrupt实现微服务优雅关闭心跳线程/笔记.doc**

结合interrupt来做一个实战

在一个分布式系统里面，一般会有一些核心的工作线程，现在如果这个系统要关闭，一般会设计一个shutdown方法，在这个方法里面，会设置各个工作线程是否需要运行的标志位为false

对各个工作线程都执行interrupt

因为各个工作线程可能都在不断的while循环运行，但是每次执行完一次之后，都会进入休眠的状态，sleep 30秒

如果系统要尽快停止，那么就应该用interrupt打断各个工作线程的休眠，让他们判断是否运行的标志位为false，就立刻退出

只要所有的工作线程都结束了以后，jvm进程就会自动退出了

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~17资料/06\_案例实战：微服务存活状态监控线程的实现/笔记.doc**

咱们不要玩儿概念，直接上案例，在案例里学东西

微服务注册中心里面，大家都记得我说的吧，各个服务会注册，此时会将自己的信息加入到一个内存的注册表里去，然后呢各个服务在运行的过程中，还会不断的发送心跳通知注册中心说，自己还活着

注册中心的话，就需要开启一个后台线程，不断的扫描和监控注册表里各个服务的心跳发送时间，如果超过一定时间还没发送心跳的话，那么不好意思，直接就是宣告这个服务死亡，从注册表里摘除掉

所以通过这个业务案例，我们就可以来玩儿一下开启线程的方法

一般是两种方式，要不然就是搞一个Thread类，重写里面的run()方法，然后创建一下实例后start()启动这个线程；要不然就是搞一个Runnable接口实现类，封装运行逻辑，然后传入Thread类构造

其实两种方式都可以，平时用的都很多，但是我一般比较喜欢用直接重写Thread的run方法的方式，这样子这个线程的类名称都可以自己来定，在系统里划分组件单元比较清晰

那我们就把这个案例来写一下

register-server，一般是以web工程的模式来启动的，他会类似于spring web mvc提供一些http的接口，让各个服务的register-client组件可以发送http请求到他这里来，当然也可以通过rpc的模式来做

在这个工程里面，给大家讲解一下，这种东西一般可以单拉一个工程出来，专门做一个叫做register-common的工程，也就是其他工程都需要依赖的一些公共的东西，类，可以 放在这个工程里面

我们在并发课程的案例里面，是不会涉及到网络的一些东西，IO，在后面是一个单独的课程，网络、IO、Netty一块儿讲，讲完了以后，就是把网络这块的东西加入微服务注册中心里面去，把整个微服务注册中心，做成一个完整的项目

仔细读一遍Spring Cloud Eureka的源码，按照他的源码，把微服务注册中心的核心机制全部手写实现一遍

我觉得大家对底层技术的开发就会非常的有把握了

每隔60秒运行一次来检查，如果某个服务超过90秒还没更新心跳来续约，那么就摘除这个服务实例

咱们用真实的分布式系统的项目，拆解大量的案例，来逐步逐步的跟，小孩子玩儿乐高一样，将所有的并发的知识点和技术，融入到项目里去，每个技术实现一个机制或者功能，就是一个案例，这就是我的这么一个并发课程的教学的思路

我觉得这种学习并发的模式，是最好的，效果是最好的，纯理论的教学，仅仅能够让你知道并发里的一些概念和技术，但是你怎么来用，其实你并不知道

咱们用了大量的时间，引入了项目案例，开发，各种启动线程，通过这两个案例，我相信大家对线程是什么东西，一般怎么来用，理解的非常深刻了

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~17资料/10\_探索Thread源码（一）：初始化过程/笔记.doc**

我们之前通过微服务注册中心一系列案例，或者项目的开发，了解了一些基本的线程的使用，线程的创建和开启，现在的话呢，我们来探索一下，Thread类的源码，初始化，启动线程的过程

init(null, null, "Thread-" + nextThreadNum(), 0);

默认情况下，如果你不指定线程的名称，那么自动给你生成的线程名称就是，Thread-0，Thread-1，以此类推的一大堆的线程

Thread parent = currentThread();

你创建线程的时候，获取到的是currentThread()，是当前创建你的那个线程，比如说一般来说就是那个main线程，main线程在创建ServiceAliveMonitor线程，所以说此时创建线程的过程中，获取到的currentThread()就是main线程

你创建一个线程的时候，默认他的父线程就是创建他的那个线程，比如main线程创建ServiceAliveMonitor线程，此时ServiceAliveMonitor线程的父线程就是main线程

下面这段代码的意思，就是说threadGroup是不指定的，他就会自动给你处理一下，给你分配一个线程组，每个线程必须属于一个ThreadGroup的。如果你没有指定线程组，那么你默认的线程组就是父线程的线程组

如果你的父线程是main线程的话，那么你的线程组就是main线程的线程组（main线程组）

默认情况下，如果你没有指定你是否为daemon的话，那么你的daemon的状态是由父线程决定的，就是说如果你的父线程是daemon线程，那么你也是daemon线程；同理，你的优先级如果没有指定的话，那么就跟父线程的优先级保持一致

if (g == null) {

/\* Determine if it's an applet or not \*/

/\* If there is a security manager, ask the security manager

what to do. \*/

if (security != null) {

g = security.getThreadGroup();

}

/\* If the security doesn't have a strong opinion of the matter

use the parent thread group. \*/

if (g == null) {

g = parent.getThreadGroup();

}

}

每个线程其实都有一个线程id，threadId，第一个分配的线程，它的id是1，之后的线程是2,3,4,5，这样子，依次分配各个线程的id

总结一下Thread初始化的过程蕴含的你需要知道的一些点：

（1）创建你的线程，就是你的父线程

（2）如果你没有指定ThreadGroup，你的ThreadGroup就是父线程的ThreadGroup

（3）你的daemon状态默认是父线程的daemon状态

（4）你的优先级默认是父线程的优先级

（5）如果你没有指定线程的名称，那么默认就是Thread-0格式的名称

（6）你的线程id是全局递增的，从1开始

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/202~222资料/204\_基于独占锁保证入队线程安全性的实现原理/笔记.doc**

入队，put，这一块的操作，如何基于独占锁，保证入队的多线程并发的安全性，以及入队的一个链表操作的过程

之前给大家讲过，线程是可以被中断的，在lock加锁的过程中，还没完成加锁，他会有一个判断，你是否被中断，如果你加锁的线程被中断了，此时加锁会失败，抛出一个线程中断的异常出来

你这个加锁的过程是可以被中断掉的

同一时间只有一个线程可以去入队的，其他线程就需要基于ReentrantLock的AQS排队等待机制去等待，非公平锁

AtomicInteger之前的值是0，所以此时拿到的c是0，接着再对0进行increment，累加为1

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/202~222资料/221\_那么如何对基于数组的线程安全队列进行查询操作呢？/笔记.doc**

size和iterator

直接加独占锁，此时此刻是没有任何一个线程可以出队或者是入队的

AtomicInteger来实现数据的写和读的，相对来说也是比较准确的，因为没有加锁，所以肯定还是会有一些入队和出队的操作同时在执行的

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/202~222资料/210\_案例实战：服务注册中心集群的三层队列异步同步机制/笔记.doc**

集群机制，向任何一个实例发送注册、下线、心跳的请求，都会同步到其他的实例上去，所有实例的内存注册表数据是一致的，以此形成一个集群

集群同步机制，三层队列批量同步机制，仿照他来实现一下

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/202~222资料/207\_那么如果队列是空的又如何阻塞出队线程以及唤醒呢？/笔记.doc**

如果队列是空的，此时线程如何被阻塞，后续如果队列有数据了是如何唤醒的

如果有人put了一个数据到队列里去，很明显应该唤醒take阻塞的线程，通知人家说现在可以获取数据了

c = 0，count = 1

如果此时c是0的话，代表的意思就是说，之前队列是空的，如果队列是空的，就可能有线程执行take操作被阻塞住了，此时就尝试唤醒take阻塞的线程

每次take完了一条数据，你都需要判断一下，c > 1，c == 2，我本次take之前，队列里起码有2条数据，那么我本次take完了之后，队列里起码还有一条数据，只要队列里起码还有1条数据的话

我此时就可以尝试对notEmpty换新一个take阻塞的线程

两把锁

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/202~222资料/211\_案例实战：集群间的异步同步机制以及数据最终一致性的分析/笔记.doc**

集群间采取了异步同步的机制和数据最终一致性的问题

最终一致性，在一段时间内，有一个小的时间窗口，集群里的各个服务注册中心实例的内存注册表里的数据可能是不一致的，所以此时你的服务进行注册表的抓取的时候，从不同的服务注册中心实例里看到的数据可能都是不一样的

最终集群里的各个服务注册中心实例的数据最终一定会一致的，他们互相之间在同步数据，最终数据一定会一样的

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/202~222资料/212\_案例实战：心跳过期下线服务为什么不需要进行集群同步？/笔记.doc**

服务注册、服务下线、发送心跳，这三种请求都是需要在集群之间进行同步的

发送心跳 -> 自动服务故障感知 -> 自动下线服务 -> 自我保护机制

服务注册 + 服务下线 -> 服务注册表的变动 -> 服务发现

自动下线故障服务的时候，是否需要进行集群之间的同步呢？可以做，也可以不这么做，eureka里面就没这么做

其他的服务注册中心实例本身都是会同步到服务B的心跳的，其他的服务注册中心实例也会感知到服务B的过期，就会自动在内存注册表里摘除服务B的实例

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/202~222资料/219\_如何基于一把独占锁实现基于数组的有界队列入队与出队？/笔记.doc**

如何基于独占锁实现线程安全的入队和出队，基于数组来实现

一把独占锁，锁掉整个数组，同一时间只有一个线程可以入队或者出队，跟LinkedBlockingQUeue不一样的，入队和出队是不能同时进行的

默认情况下从数组的index = 0的位置开始入队

count就是用来保存你的队列里当前有多少个元素

[null, null, null, 赵六, null, null, null, null, null, null]

count = 3

putIndex = 3

takeIndex = 0

x = 张三 = 队头的元素

takeIndex = 1

count = 2

takeIndex = 2

count = 1

takeIndex = 3

count = 0

putIndex = 4

count = 1

性能相对来说还是挺高的，出队和入队的操作，其实都是直接基于数组的index来实现的

链表的实现出队和入队性能也很高的，直接挂node

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/202~222资料/216\_案例实战：实现第三层队列用于存放以及发送同步batch/笔记.doc**

做第三层的队列

每个AbstractRequest会占用多少字节的内存，假设大概来说一个request，占用30个字节

平均一个batch包含100个request，3000个字节，3kb

10000 -> 30mb

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/202~222资料/205\_如何使用另外一把独占锁实现出队的线程安全性呢？/笔记.doc**

通过独占锁就可以保证同一时间只有一个线程可以执行出队的操作

两把独占锁提升并发的性能，出队和入队用的是不同的锁，多个线程并发出队和入队的时候，出队和入队两个操作是可以同时执行的，没有锁冲突的

这个两把锁的设计思想也是一个锁优化的思想，锁功能进行了拆分，用不同的锁控制不同功能的并发冲突

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/202~222资料/209\_案例实战：微服务注册中心的集群机制、高并发以及高可用/笔记.doc**

3616118202

项目实战，微服务注册中心的集群机制 -> 高并发 -> 高可用

每台机器承载的并发请求都是有限制的，单台机器足够了，每秒可能就几百次请求，但是未来慢慢的微服务规模越来越庞大，服务实例数量越来越多，最后就可以向单台机器请求的并发达到每秒几千次，上万次

所有的服务可以用负载均衡的方式来请求服务注册中心集群里的任意一个实例，用一个集群抗下每秒几千次甚至上万次的高并发的请求，集群里的每个服务注册中心实例也就抗个每秒几百次请求

服务注册中心仅仅只是部署了一台机器的话，那么如果这台机器他宕机了，就会导致服务注册中心整体就不可用了，但是如果你部署了一个集群的话，那么任何一台机器宕机，不会导致服务注册中心不可用的

其他的服务注册中心实例还是可用的

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/202~222资料/208\_如何对线程安全的链表结构有界队列进行查询？/笔记.doc**

size，遍历

AtomicInteger底层是CAS + volatile，拿到的基本是比较准备的一个值，跟ConcurrentLinkedQueue获取的size

100%准确的一个size，锁掉整个队列，不允许人入队和出队，获取到的size一定是准确的

LinkedBlockingQueue是直接锁了整个队列，直接把两把锁都给锁掉了，如果size也要获取类似的操作，也是这样子fullyLock

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/202~222资料/203\_基于链表结构的有界队列以及两把独占锁提升并发性能/笔记.doc**

画图分析一下有界队列的原理

Condition，如果你获取了一把锁，然后调用了Condition.await()，此时会释放锁，当前线程会进入一个cnodition等待队列，事后人家来唤醒你，Cnodition.signal()，这个时候会从condition等待队列中

把你加入到wait等待队列里去，然后人家释放锁，就会唤醒wait等待队列里的线程尝试来获取锁

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/202~222资料/220\_基于数组的队列又是如何实现线程的阻塞以及唤醒？/笔记.doc**

队列满的时候如何阻塞和唤醒，队列空的时候如何阻塞和唤醒

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/202~222资料/222\_关于优先级队列和延迟队列的API使用自学提示/笔记.doc**

优先级队列：PriorityBlockingQueue，你可以给队列里的每个元素都安排一个优先级，他的话内部会基于二叉树的数据结构来存储，根据优先级对队列中的元素进行排序，你出队的时候是按照优先级的大小来出队的

延迟队列：DelayQueue，你可以给这个压入队列的每个元素设置一个delay延迟时间，压入队列一个元素，delay time是500ms，那么必须等500ms过后这个元素才能从队列里出队的方法里获取到

java并发包下面的，我个人是不打算给大家来讲解和分析的，这两个东西真的很少很少，在单机场景下，对单个虚拟机而言，内部用优先级队列和延迟队列，是非常少见的，MQ那块，消息中间件那块

MQ那块，很多技术方案都会依赖于优先级队列和延迟队列来实现的

结合MQ那块对优先级队列和延迟队列的功能支持，同时结合MQ相关的技术方案来分析和讲解

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/202~222资料/217\_案例实战：完成微服务注册中心集群之间的异步内存数据同步/笔记.doc**

开一个接口来接收同步过来的batch，落地到自己内存里去

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/202~222资料/218\_基于数组实现的有界队列：ArrayBlockingQueue/笔记.doc**

基于链表实现的阻塞队列，LinkedBlockingQueue；基于数组实现的有界阻塞队列，整体原理跟LinkedBlockingQueue，只不过单向链表，ArrayBlockingQueue用的是数组来实现，这块这讲带着大家快速的写一下demo

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/202~222资料/215\_案例实战：使用第二层队列基于时间间隔和数量生成同步batch/笔记.doc**

第二层队列用什么来实现呢？

LinkedBlockingQueue，一方面是独占锁，有界的，对第二个队列的入队和出队的操作都是少数的后台线程在执行，并不是大量线程高并发在写入的

有界的特性，避免说在异常情况下的过度内存消耗和膨胀，是合理的，是可行的

参照eureka，按照时间来设置一个interval，每隔500ms打包成一个batch

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/202~222资料/206\_队列满时入队线程是如何阻塞以及被唤醒的呢？/笔记.doc**

blocking queue，阻塞，队列满需要阻塞线程不让放了，队列空需要阻塞线程不让获取

如果队列满了以后，是如何阻塞线程的；后续如果有人从队列里take掉了一个元素，队列不满了，是如何唤醒那个线程的

队列满了以后，直接会调用put锁对应的condition进行阻塞等待的操作，当前线程被挂起，释放锁，进入condition等待队列

只要发现队列不满了，比如说有人take掉了一个元素，此时是不是就可以唤醒处于阻塞的线程重新获取put锁再去执行入队的操作

假设count = 10，此时getAndDecrement()，获取到的c = 10，但是count = 9

此时c = capacity，也就代表的意思是什么呢？本次take之前队列是满的，但是本次take之后队列肯定不是满的了。主要队列之前是满的，很可能有人在put的时候处于阻塞的状态，此时就可以把处于put阻塞状态的线程唤醒

你在执行put操作的时候，每次执行完了put操作之后，c + 1（代表了当前的count），小于capacity的话，就说明当前的队列没满，此时就可以也是执行一下put锁等待对了的signal方法尝试唤醒一个阻塞的线程

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/202~222资料/213\_案例实战：对服务注册、服务下线以及发送心跳接入集群同步/笔记.doc**

服务注册、服务下线、发送心跳，接入集群同步机制

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/202~222资料/214\_案例实战：基于无界CAS无锁化队列实现第一层队列/笔记.doc**

实现第一层队列

ConcurrentLinkedQueue，无界队列，纯CAS实现的无锁机制，多线程并发的性能很高

LinkedBlockingQueue，有界队列，基于两把独占锁实现的，多线程并发性能并不是很好，但是可以实现指定大小的有界的限制，避免内存无限制的膨胀，而且还可以针对有界这个特点，实现写入和获取的阻塞

第一层队列一定是高并发的入队的，所以用LinkedBlockingQueue未必合适

第一层队列是有界还是无界的，如果做成有界的，限制大小，万一有点什么意外，可能会导致在有界队列这里直接打满队列，会导致整个服务注册中心实例所有线程全部阻塞在这里，假死的状态

对于这种情况而言，用无界队列实现第一层队列，更加合适一些，这个是对eureka做的一点改进

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/202~222资料/202\_线程安全的有界队列：LinkedBlockingQueue/笔记.doc**

无界队列，有界队列

无界队列：ConcurretedLinkedQueue，边界，大小限制，他就是一个单向的链表，无限制的往里面去存放数据，再从队列里获取数据

不停的疯狂的往无界队列里塞入数据的话，可能会导致内存溢出

提了一个所谓的面试题，使用无界队列的线程池，在远程服务异常情况下，会有什么问题。不停的可以往线程池里扔任务，搞一个线程请求远程服务，就导致请求卡死，一直阻塞无法请求通

就会导致不停的增长线程，最后线程超多，一直到把内存撑爆

有界队列：LinkedBlockingQueue，链表，有界，有大小限制，如果超过了限制，你往队列里塞数据就会被阻塞住，就不让你往队列里塞数据了，好处就在于说可以限制内存队列的大小，避免说内存队列无限制的增长，最后撑爆内存

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/74~99资料/93\_基于CAS实现多线程并发同时只有一个可以加读锁/笔记.doc**

dd

int r = sharedCount(c);：拿到state的高16位的值，代表了读锁的加锁次数

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/74~99资料/84\_tryLock如何实现加锁等待一段时间过后放弃？/笔记.doc**

在实际开发的时候，我们可能会使用到tryLock操作，有的时候你不希望一直阻塞在那里尝试加锁

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/74~99资料/77\_AQS如何基于无锁化的CAS机制实现高性能的加锁/笔记.doc**

大家就会留意到 我之前为什么先讲了Atomic原子类的系列，Atomic原子类 + Volatile可见性，更加是最最底层的java并发包实现各种东西的一些基础中的 基础，必须得从那两块东西开始讲起

AQS底层加锁、释放锁，都是大量的基于CAS的操作来实现的，底层是基于 NonfairSync的 lock操作来实现加锁的

final void lock() {

if (compareAndSetState(0, 1))

setExclusiveOwnerThread(Thread.currentThread());

else

acquire(1);

}

if (compareAndSetState(0, 1))：AQS里有一个核心的变量，state，代表了锁的状态；看一下state是否是0？如果是0的话，代表没人加过锁，此时我就可以加锁，把这个state设置为1

CAS可以无锁化的保证一个数值修改的原子性

compareAndSetState(0, 1)：相当于是在尝试加锁，底层原来是基于Unsafe来实现的，JDK内部使用的API，指针操作，基于cpu指令实现原子性的CAS，Atomic原子类底层也是基于Unsafe来实现的CAS操作

return unsafe.compareAndSwapInt(this, stateOffset, expect, update);

这行代码可以保证说，在一个原子操作中，如果发现值是我们期望的这个expect值，说明符合要求，没人修改过，此时可以将这个值设置为update，state如果是0的话，就修改为1，代表加锁成功了

这个操作是CAS原子性的

如果加锁成功了，compareAndSetState(0, 1)返回的是true，此时就说明加锁成功，他需要设置一下自己是当前加锁的线程

setExclusiveOwnerThread(Thread.currentThread());

设置当前线程自己是加了一个独占锁的线程，标识出来自己是加锁的线程

CAS -> AQS的state变量 -> exclusiveOwnerThread（当前加锁线程），Lock API性能比较好，CAS无锁化，乐观锁的思路，反复的比较

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/74~99资料/81\_用第三个线程尝试加锁失败彻底图解AQS队列等待机制/笔记.doc**

第二个线程尝试加锁失败了，入队列阻塞等待；第三个线程尝试加锁失败了，此时底层的入队算法是如何运作的，如何阻塞等待

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/74~99资料/94\_多线程加读锁时的重复循环自旋尝试完成加锁/笔记.doc**

如果有一个线程先加了读锁以后，其他线程过来同时要加读锁会怎么样呢？

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/74~99资料/88\_读写锁中的写锁是如何基于AQS的state变量完成加锁的？/笔记.doc**

就来玩玩儿读写锁

优化微服务注册中心的注册表的读写并发操作，读写锁

公平和非公平，之前都解释过了，所以说，默认人家都是非公平锁

Thread current = Thread.currentThread();

// 获取到一个state = 0

int c = getState();

// w，剧透一下，人家读写锁，非常聪明的利用state的值

// 二进制值里面的高低16位分别代表了读锁和写锁，AQS就一个，state

// state二进制值的高16位代表了读锁，低16位代表了写锁

// 可以认为下面的w就是从c（二进制值）通过位运算

// 获取到了state的低16位，代表了写锁的状态

int w = exclusiveCount(c);

// 如果c != 0，说明有人加过锁，但是此时c = 0

if (c != 0) {

// (Note: if c != 0 and w == 0 then shared count != 0)

if (w == 0 || current != getExclusiveOwnerThread())

return false;

if (w + exclusiveCount(acquires) > MAX\_COUNT)

throw new Error("Maximum lock count exceeded");

// Reentrant acquire

setState(c + acquires);

return true;

}

// 非公平锁，此时一定会去尝试加锁

// 如果是公平锁，此时会判断如果队列中有等待线程，就不加锁

if (writerShouldBlock() ||

!compareAndSetState(c, c + acquires))

return false;

setExclusiveOwnerThread(current);

return true;

基本跟之前看到的是一样的，如果说你来加写锁的话，state += 1，锁占用线程

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/74~99资料/97\_居然还可以用Condition实现wait和notify的效果？/笔记.doc**

Condition -> demo

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/74~99资料/86\_锁释放过后如何对AQS队列中唤醒阻塞线程尝试抢占锁？/笔记.doc**

此时我们是可以看到，他会唤醒处于队头的元素

如果一个线程来释放锁的话，他除了更新state和锁占有线程以外，他其实主要干的一个事儿就是用LockSupport的unpark操作唤醒了一个处于队头的一个线程

队头线程此时被unpark唤醒之后会干什么？

private final boolean parkAndCheckInterrupt() {

// 某一个线程其实是在这里会被挂起

LockSupport.park(this);

return Thread.interrupted();

}

for (;;) {

final Node p = node.predecessor();

if (p == head && tryAcquire(arg)) {

setHead(node);

p.next = null; // help GC

failed = false;

return interrupted;

}

if (shouldParkAfterFailedAcquire(p, node) &&

parkAndCheckInterrupt())

interrupted = true;

}

如果一旦被unpark唤醒之后，就会在这里苏醒过来，重新进入一个for循环里面

此时线程2会再次尝试去获取锁，因为他是队头线程，他的上一个节点一定就是那个head指针指向的节点了

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/74~99资料/79\_AQS的本质：为啥叫做异步队列同步器？真相大白！/笔记.doc**

线程2过来尝试加锁，此时的源码会如何走向呢？

final boolean nonfairTryAcquire(int acquires) {

// 先获取到当前的线程 -> 线程1

final Thread current = Thread.currentThread();

// 获取state变量值的过程，JDK源码里大量的运用了volatile，可见性的问题，保证一些关键变量，修改 -> 读取的可见性

int c = getState();

// 为什么会有这段代码呢？其实进入到这里，代表他之前一定是看到state != 0，才会进入到这里

// 就是人家代码的健壮性，怕的是之前state != 0，所以加锁失败了，但是进入到这里，人家再次判断一下，如果state是0，那么再次尝试加锁，就怕中间有人释放了锁

if (c == 0) {

if (compareAndSetState(0, acquires)) {

setExclusiveOwnerThread(current);

return true;

}

}

// 也就是说没有人释放锁，state != 0

// 再次判断，如果执行这个方法的线程 = exclusiveOwnerThread（加锁的线程）

// 代表的就是一个线程在可重入的加锁

// 之前他自己加过锁，然后在这里他就再次加锁

else if (current == getExclusiveOwnerThread()) {

// 此时，c = 1

// nextc = c(1) + acquires(1) = 2

// 其实就是代表了一个线程可重入加锁了1次，2代表了加锁的次数

int nextc = c + acquires;

if (nextc < 0) // overflow

throw new Error("Maximum lock count exceeded");

// 修改这个state的值，volatile保证了可见性

setState(nextc);

return true;

}

// 如果已经有一个线程加了锁，其他线程此时会走到这里

// 此时方法认为加锁失败，返回false

return false;

}

public final void acquire(int arg) {

// 此时加锁失败，第一个条件是false

// 开始走第二个条件，调用acquireQueued()方法

// 将当前线程入队阻塞等待

if (!tryAcquire(arg) &&

acquireQueued(addWaiter(Node.EXCLUSIVE), arg))

selfInterrupt();

}

addWaiter(Node.EXCLUSIVE)：EXCLUSIVE（排他性，独占锁，同一时间只能有一个线程获取到锁，此时是排他锁，独占锁）

Node node = new Node(Thread.currentThread(), mode);

将当前线程（线程2）封装成了一个Node，mode = EXCLUSIVE（排他锁，尝试获取一个排他锁，但是失败了），研究一下Node里面包含了一些什么东西？

// 如果一个线程无法获取到锁的话，会进入一个阻塞等待的状态

// 卡住不动，线程挂起，阻塞状态又细分为很多种不同的阻塞状态：

// CANCELED、SIGNAL、CONDITION、PROPAGATE

volatile int waitStatus;

// 一个节点可以有上一个节点，prev指针，指向了Node的上一个Node

volatile Node prev;

// 一个节点还可以有下一个节点，next指针，指向了Node的下一个Node

volatile Node next;

// Node里面封装了一个线程

volatile Thread thread;

// 可以认为是下一个等待线程

Node nextWaiter;

获取不到锁，处于等待状态的线程，会封装为一个Node，而且有指针，最后多个处于阻塞等待状态的线程可以封装为一个Node双向链表，JDK集合源码，就可以作为一个队列来实现了

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/74~99资料/95\_再次回头看看读锁占用时加写锁失败如何入队阻塞等待/笔记.doc**

0000 0000 0000 0010 0000 0000 0000 0000

w = 0

读锁，c != 0，c的高16位是不等于0，w = 0，低16位是0，有人加了读锁，此时是绝对不允许加写锁

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/74~99资料/80\_加锁失败的时候如何借助AQS异步入队阻塞等待？/笔记.doc**

3616118202

如果加锁失败的话，他会把自己封装成一个Node，Node原来是AQS底层非常关键的一个数据结构，双向链表 => 队列，估计就会将当前的线程封装为Node入队，阻塞等待，别人释放锁

private Node addWaiter(Node mode) {

// 将当前线程封装了一个Node

Node node = new Node(Thread.currentThread(), mode);

// Try the fast path of enq; backup to full enq on failure

Node pred = tail;

if (pred != null) {

node.prev = pred;

if (compareAndSetTail(pred, node)) {

pred.next = node;

return node;

}

}

enq(node);

return node;

}

for (;;) {

Node t = tail;

if (t == null) { // Must initialize

if (compareAndSetHead(new Node()))

tail = head;

} else {

node.prev = t;

if (compareAndSetTail(t, node)) {

t.next = node;

return t;

}

}

}

unsafe.compareAndSwapObject(this, headOffset, null, update);

headOffset -> 在AQS类里，head变量所在的位置，CAS操作的，判断一下，head变量是否为null，如果是null的话，就将head设置为空Node节点

compareAndSetTail(t, node)：尝试比较tail变量是否为t，如果为t的话，那么tail指针就指向node

acquireQueued(线程2代表的Node, 1)

final boolean acquireQueued(final Node node, int arg) {

boolean failed = true;

try {

boolean interrupted = false;

for (;;) {

// 获取到node的上一个节点

// prev指针指向的节点

final Node p = node.predecessor();

// 这个地方，其实会再次调用tryAcquire方法尝试加锁

// 如果加锁成功，其实是会将线程2对应的Node从队列中移除

if (p == head && tryAcquire(arg)) {

setHead(node);

p.next = null; // help GC

failed = false;

return interrupted;

}

// 如果说再次尝试加锁失败了

// 那么此时会判断一下，是否需要将当前线程挂起，阻塞等待

// 如果是需要的话，此时就会使用park操作挂起当前线程

if (shouldParkAfterFailedAcquire(p, node) &&

parkAndCheckInterrupt())

interrupted = true;

}

} finally {

if (failed)

cancelAcquire(node);

}

}

private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {

// pred -> 空Node

// 默认情况下，watiStatus应该是0，或者是空

int ws = pred.waitStatus;

if (ws == Node.SIGNAL)

/\*

\* This node has already set status asking a release

\* to signal it, so it can safely park.

\*/

return true;

if (ws > 0) {

/\*

\* Predecessor was cancelled. Skip over predecessors and

\* indicate retry.

\*/

do {

node.prev = pred = pred.prev;

} while (pred.waitStatus > 0);

pred.next = node;

} else {

/\*

\* waitStatus must be 0 or PROPAGATE. Indicate that we

\* need a signal, but don't park yet. Caller will need to

\* retry to make sure it cannot acquire before parking.

\*/

// 将空Node的waitStatus设置为SIGNAL

compareAndSetWaitStatus(pred, ws, Node.SIGNAL);

}

return false;

}

private final boolean parkAndCheckInterrupt() {

// LockSupport的park操作，就是将一个线程进行挂起，不让你动了

// 必须得有另外一个线程来对当前线程执行unpark操作，唤醒挂起的线程

LockSupport.park(this);

return Thread.interrupted();

}

public final void acquire(int arg) {

// 先尝试加锁

// 如果加锁失败，addWaiter()方法将自己挂到队列中去

// 接着acquireQueued()方法负责park操作挂起当前线程，阻塞等待

if (!tryAcquire(arg) &&

acquireQueued(addWaiter(Node.EXCLUSIVE), arg))

selfInterrupt();

}

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/74~99资料/98\_基于AQS实现的Condition阻塞过程源码分析/笔记.doc**

画图

线程1 -> 加锁 -> 释放锁&阻塞 -> 线程2 -> 加锁 -> 唤醒线程1 -> 释放锁 -> 线程1 -> 再次加锁 -> 线程1再次释放锁

Condition.await()原理：将自己加入condition等待队列、释放锁、挂起自己

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/74~99资料/90\_写锁加锁失败时如何基于AQS队列完成入队阻塞等待？/笔记.doc**

线程1可重入的加了两次写锁，假设此时线程2来加写锁会如何呢？

线程2如果来加写锁，会被互斥，此时就会被卡住，阻塞，在队列里等待唤醒

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/74~99资料/76\_ReentractLock底层原来是基于AQS来实现锁的！/笔记.doc**

一步一步的来探索一下Java Lock API的底层的源码以及内部的原理

public ReentrantLock() {

sync = new NonfairSync();

}

默认的构造函数这里，创建了一个Sync，NonfairSync，看起来是一个非常关键的组件，很可能是底层专门用于加锁和释放锁的核心组件

public void lock() {

sync.lock();

}

ReentrantLock在进行加锁的时候，他其实是直接基于底层的Sync来实现的lock操作，但是如果是这样子的话，ReentractLock这个类其实就是比较外层的一个薄薄的封装的一个类了，Sync就是ReentrantLock底层的核心组件

Sync：关键组件

abstract static class Sync extends AbstractQueuedSynchronizer

Sync是 一个抽象的静态内部类，子类？AQS：AbstractQueuedSynchronizer，这个东西，抽象队列同步器，是java并发包各种并发工具（锁、同步器）的底层的基础性的组件，核心的，主要是依赖于他

AQS里关键的一些东西，一个是Node（自定义数据结构，可以组成一个双向链表，也就是所谓的一个队列），state（核心变量，加锁、释放锁都是基于state来完成的）

Sync就是AQS（子类实现，多线程同步组件的意思）

NonfairSync是Sync的一个子类，覆盖重写了几个方法，没什么特别的东西在里面，大概代表了一个Sync的具体实现

ReentractLock -> synchronized（可重入加锁的）-> AQS -> NonfairSync（非公平的同步组件），什么公平锁，非公平锁，后面给大家来说一下

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/74~99资料/92\_释放写锁的源码剖析以及对AQS队列唤醒阻塞线程的过程/笔记.doc**

释放写锁

加写锁、释放写锁的过程，通过这个图的不断的变化，就全部都搞清楚了，现在轮到线程3可以来加读锁了

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/74~99资料/87\_一种新奇的加锁玩法：读锁和写锁分开是怎么玩的？/笔记.doc**

ReentrantLock，锁API，是如何基于AQS来实现的

主要的作用是给大家剖析清楚java并发包底层的AQS的基本实现原理，ReentrantLock比较少用，syncrhonized是差不多，JDK 1.6以后，synchronized底层实现里面，里面也是做一些计数器的维护，加锁释放锁，CAS

如果需要用synchronized话，优先使用；真正开发中，或者看一些开源项目的源码的话，其实都是用的synchronized，读写锁，ReentrantReadWriteLock，ReentrantLock

Lock API，读写锁，可以加读锁，也可以加写锁

但是，读锁和写锁是互斥的，也就是说，你加了读锁之后，就不能加写锁；如果加了写锁，就不能加读锁

但是如果有人加了读锁之后，别人可以同时加读锁

如果你有一份数据，有人读，有人写，如果你全部都是用synchronized的话，会导致如果多个人读，也是要串行化，一个接一个的读

我们希望的效果是多个人可以同时来读，如果使用读锁和写锁分开的方式，就可以让多个人来读数据，多个人可以同时加读锁

如果有人在读数据，就不能有人写数据，读锁 -> 写锁 -> 互斥

如果有人在写数据，别人不能写数据，写锁 -> 写锁 -> 互斥；如果有人在写数据，别人也不能读数据，写锁 -> 读锁 > 互斥

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/74~99资料/75\_面试的时候容易被问懵的问题：谈谈你对AQS的理解？/笔记.doc**

有同学去一些互联网大厂里面试，比较喜欢问并发编程的一些东西，很多面试官问是在问，只不过是他也知道而已，并不一定说代表你用并发的技术做过很多复杂的系统，并不是这样子的

AQS，AbstractQueuedSynchronizer，抽象队列同步器，很多同学被问到绝对蒙圈，我目前为止反正是没看到什么特别让我印象深刻的AQS原理说明的文章，让人看不懂，大厂又喜欢问，很多出去，被问到AQS

ReentractLock、ReadWriteReentractLock，锁API底层都是基于AQS来实现的，一般我们自己不直接使用，但是是属于java并发包里的底层的API，专门支撑各种java并发类的底层的逻辑实现

大白话，非常简单，画一张图，让大家知道AQS是个什么东东？

因为马上我们就要开始剖析ReentrantLock的源码了，讲技术，必须讲到底层，分布式锁，分布式事务，spring cloud，我们就深入底层，读写锁，再来玩儿一下微服务注册中心里的服务注册表涉及到大量的synchronized加锁

读写锁来优化服务注册表的多线程并发加锁的效率

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/74~99资料/78\_如何巧妙的借助AQS中的state变量实现可重入式加锁？/笔记.doc**

假如说此时，线程1再次进入，可重入的来加锁

如果是一个线程可重入的加锁会是什么样子呢？是如何来实现的呢？

compareAndSetState(0, 1)：这个方法一定是false，会失败，此时state = 1，不是0，CAS操作会失败，返回false，此时会执行acquire(1)

acquire(1)：会走到AQS的方法里

tryAcquire(1)：此时首先会走这个方法，传递进去一个值是1，AQS的父类 实现是一个空，其实是留给子类来实现的

protected final boolean tryAcquire(int acquires) {

return nonfairTryAcquire(acquires);

}

nonfairTryAcquire(1)：这个方法会走到Sync（父类）

final boolean nonfairTryAcquire(int acquires) {

// 先获取到当前的线程 -> 线程1

final Thread current = Thread.currentThread();

// 获取state变量值的过程，JDK源码里大量的运用了volatile，可见性的问题，保证一些关键变量，修改 -> 读取的可见性

int c = getState();

// 为什么会有这段代码呢？其实进入到这里，代表他之前一定是看到state != 0，才会进入到这里

// 就是人家代码的健壮性，怕的是之前state != 0，所以加锁失败了，但是进入到这里，人家再次判断一下，如果state是0，那么再次尝试加锁，就怕中间有人释放了锁

if (c == 0) {

if (compareAndSetState(0, acquires)) {

setExclusiveOwnerThread(current);

return true;

}

}

// 也就是说没有人释放锁，state != 0

// 再次判断，如果执行这个方法的线程 = exclusiveOwnerThread（加锁的线程）

// 代表的就是一个线程在可重入的加锁

// 之前他自己加过锁，然后在这里他就再次加锁

else if (current == getExclusiveOwnerThread()) {

// 此时，c = 1

// nextc = c(1) + acquires(1) = 2

// 其实就是代表了一个线程可重入加锁了1次，2代表了加锁的次数

int nextc = c + acquires;

if (nextc < 0) // overflow

throw new Error("Maximum lock count exceeded");

// 修改这个state的值，volatile保证了可见性

setState(nextc);

return true;

}

return false;

}

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/74~99资料/74\_尝试一下另一种锁：ReentractLock的demo例子/笔记.doc**

synchronized那块东西，还聊了一下Atomic原子类系列的用法，Lock锁API，ReentractLock，可重入锁，ReadWriteReentractLock，读写锁

原生的关键字，wait+notify

Lock锁API，来加锁和释放锁，读写锁，非常常用，而且非常有用

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/74~99资料/85\_基于AQS实现的可重入锁释放过程的源码剖析/笔记.doc**

释放锁，线程1加了锁，总得释放

如何把锁给释放掉，另外一个是如果锁彻底释放了以后，如何让队列中的队头的那个线程来唤醒尝试获取锁

protected final boolean tryRelease(int releases) {

// getState() == 2

// releases == 1

// getState() - releases = 2 - 1 = 1

// c = 1

int c = getState() - releases;

// 当前线程不等于加锁的线程，说明不是你加的锁，结果你来释放锁

if (Thread.currentThread() != getExclusiveOwnerThread())

throw new IllegalMonitorStateException();

boolean free = false;

if (c == 0) {

free = true;

setExclusiveOwnerThread(null);

}

setState(c);

return free;

}

可重入加锁，加了2次，释放锁1次，再次释放锁

如何唤醒队头的元素，队头的元素唤醒之后是如何重新尝试加锁的呢？

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/74~99资料/83\_ReentractLock如何设置公平锁策略以及原理图解/笔记.doc**

如果说你希望每个人过来都要按照顺序排队来加锁，公平锁，每个人先来后到，先来的人先加锁，公平性的，公平锁

if (c == 0) {

if (!hasQueuedPredecessors() &&

compareAndSetState(0, acquires)) {

setExclusiveOwnerThread(current);

return true;

}

}

公平锁的核心，就是一行代码，每次加锁的时候，都要先判断一下，如果前面没有排队等待的线程的话，我就尝试加锁，否则是不能尝试加锁的

public final boolean hasQueuedPredecessors() {

// The correctness of this depends on head being initialized

// before tail and on head.next being accurate if the current

// thread is first in queue.

Node t = tail; // Read fields in reverse initialization order

Node h = head;

Node s;

return h != t &&

((s = h.next) == null || s.thread != Thread.currentThread());

}

h != t，如果h != t，说明head和tail不一样，如果一样代表了队列里有人在排队

但是如果说head的下一个节点是null，说明没人在排队，因为有一个是null，所以此时也是返回true；或者是s，也就是排在队头的节点，队头节点的线程如果不是当前线程，所以此时也是返回true

此时就是，会判断出来当前有人在排队，所以返回true

公平锁，任何一个线程过来会先判断一下，当前是否有人在排队，而且是不是自己在排队，如果不是的话，说明有别人在排队，此时自己不能尝试加锁，直接入队阻塞等待

先来后到的顺序，后来的人一定是排到队列里去等待

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/74~99资料/99\_基于AQS实现的Condition唤醒阻塞线程的源码分析/笔记.doc**

线程1 -> 加锁 -> 释放锁&阻塞 -> 线程2 -> 加锁 -> 唤醒线程1 -> 释放锁 -> 线程1 -> 再次加锁 -> 线程1再次释放锁

Condition.await()原理：将自己加入condition等待队列、释放锁、挂起自己

如果在加锁等待队列里有人阻塞，会有unpark的过程，唤醒加锁等待队列中的队头元素的那个过程

signal唤醒的过程，大概的意思，就是把condition等待队列中的元素，转化为一个加锁等待队列中的元素

功能和效果其实是一样的，只不过给大家借助AQS分析一下底层的源码，如果大家之前的源码都看懂了，看condition的源码，跟玩儿一样

源码，干的源码，java并发包的AQS、Lock的源码，视频看2遍，自己按照视频的思路，调试和分析源码，自己画图，全部都给他搞懂

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/74~99资料/91\_读写锁互斥：基于AQS的state二进制高低16位完成互斥判断/笔记.doc**

很多人问我是怎么学习的，上大学的时候，你必须得是个学霸，4年 + 硕士，把技术功底打的极为扎实，计算机、磁盘、IO、内存、网络、linux、操作系统、数据库、搜索引擎、分布式系统架构理论，理论知识，做一些实验项目，实习项目

进入工作之后，大公司，BAT，有技术挑战，有流量、数据量、高可用故障场景、性能差、系统复杂、大规模团队协同，不断的研究新技术，我一开始工作，就直接可以看各种开源项目的源码

看官网 -> 使用 + 基本原理，看源码 -> 底层 + 更加深厚的技术功底 + 积累更加优秀的开源架构思想 + 工作里可以用新技术解决问题 + 遇到问题你都可以解决 + 让你跟那些99%的普通工程师拉开差距 + 公司内部升职/外部跳槽面试

不停的用技术解决各种复杂的业务、架构、技术挑战，架构的规模越来越大，5个人负责一个小系统，50个人负责一个大的产品线复杂系统的架构，200人，负责一个核心大业务群的涉及到几十个甚至上百个系统的超大规模架构，积累起来的设计架构、搭建组织结构，能力，国内罕有

比较牛的一些人，Google搜索引擎首席架构师，世界顶尖的架构能力，庞大规模的系统的架构，很多细节他也要care，而且他要去整体把控、设计一个超大系统的架构，难度和能力是指数级的

Google搜索引擎，支撑5亿人来使用，这个架构怎么来设计？能干的下来这个事儿的，就是顶尖的架构师，关注了里面的一块东西，或者一部分的内容。是需要机遇的，可遇而不可求的，跟着时代

if (exclusiveCount(c) != 0 &&

getExclusiveOwnerThread() != current)

return -1;

exclusiveCount(c)：获取的是state的低16位，代表写锁的状态值，如果不等于0，说明有人加了写锁，而且还不是你加的那个写锁，此时是不能加读锁的，在这里就形成了一个基于state的二进制高低16

写锁 -> 读锁的互斥

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/74~99资料/82\_AQS默认的非公平加锁策略的运作原理图解/笔记.doc**

非公平锁和公平锁

分布式锁底层的原理，也给大家讲过，公平必须严格排队 -> 非公平随意抢占

释放锁的源码，state = 0，加锁线程 = null

非公平，哪怕是有很多的线程在队列里排队，但是当某个线程释放锁的一瞬间，很可能会有其他的晚到的线程突然争抢到了锁

就导致先来的那些线程傻乎乎的还在排队，大家去坐摩天轮，天津，售票员，说，现在不能买票，前面的人都在摩天轮里坐着呢，你得等一批人出来了才能买票进场。100个人在那儿傻乎乎的排队

突然，售票员大喊，前一批人出来了，可以买票啦，可以买1张票，此时突然一个愣头青，翻越了围栏，一下子冲到队列的最前面，先买了一张票，进去坐摩天轮。结果后面的人，排了1个小时，结果导致就没买到票

非公平

ReentrantLock，默认的策略是非公平锁

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/74~99资料/96\_读锁释放过程分析以及如何基于AQS唤醒阻塞加写锁的线程/笔记.doc**

读锁释放过程，唤醒别人来加写锁

java并发包里的ReentrantLock、ReentrantReadWriteLock、AQS底层原理全部清除了，如果出去面试，要么别人别问到你AQS，要么你就直接喷出来各种java并发包的底层源码，读写锁的互斥，state高低16位的判断，c、w、r，互斥

Lock -> Condition（wait和notify） -> 读写锁的实战 -> 锁优化

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/74~99资料/89\_基于AQS的state二进制高低16位判断实现写锁的可重入加锁/笔记.doc**

写锁的可重入加锁，state的二进制高低16位的判断了

如果是int类型的state不是0的话，那么他的二进制数值，32位，低16位一定不是0，如果低16位不是0的话，就代表他是加过写锁的

c != 0，w == 0，c肯定不是0，但是低16位是0，说明有人加了读锁，没有人加写锁，此时你要加写锁，而且你还不是之前加锁的那个线程

c != 0，w != 0，有人加过锁，之前加的是写锁，但是当前线程不是之前加锁的线程，此时也不让你加写锁，同一个时间，只能有一个线程加写锁，如果线程1比如加了写锁，线程2也要加写锁，是不行的

c != 0，w != 0，之前有人加过写锁，而且加写锁的是你自己

如果加写锁的人是你自己，说明你就是在可重入的加写锁，将state += 1

烧脑，这个高低16位的判断，稍微有点烧脑

c =1

w = 1，代表了c，int，32位，低16位的值，代表了写锁的可重入加锁次数

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/113\_锁优化案例：分布式存储系统edits log机制中的ThreadLocal实践/笔记.doc**

volatile、Atomic、ThreadLocal，锁优化的三大利器

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/105\_ThreadLocal源码剖析：线程本地副本的实现原理/笔记.doc**

分析一下ThreadLocal源码的实现

JDK里面的Thread类，内部有一个ThreadLocalMap内部类，代表了一个map，每个Thread线程对象自己内部就有一个核心的数据结构是map

这个map只能是某个线程自己内部可以使用的一份数据，是不是就是代表了线程本地的副本。一个Thread可以放多个ThreadLocal对应的本地变量副本

Thread {

ThreadLocalMap {

ThreadLocal（requestId）: 1L,

ThreadLocal（txid）：1L

}

}

requestId.get() -> Thread.ThreadLocalMap -> ThreadLocal（requestId） -> 1L

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/106\_ThreadLocal在分布式存储系统edits log案例中的实践/笔记.doc**

讲一下，在分布式存储系统案例，edtis log机制，txid的东西

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/102\_案例实战：服务注册表的多线程并发读写场景分析/笔记.doc**

Lock API锁，给大家讲完了，分析底层源码，熟悉了源码，你才熟悉了里面的工作原理，何况你出去面试的时候也很可能会有人跟你问这个源码

读写锁做一个项目实战

服务注册表，多线程并发的读写

读多写少的场景，大部分情况下都是各个服务每隔30秒来拉取一次服务注册表，要不是增量拉取，要不是全量拉取

两份数据，全量注册表 + 增量变化服务队列，注册 + 下线 -> 写这两份数据；在服务启动全量拉取注册表，以及每隔30秒增量拉取注册表的 -> 读这两份数据

读 > 写

启动服务 -> 注册；服务重启，宕机，关掉 -> 下线 -> 比较少的

多的，服务每隔30秒来拉取数据，全量注册表的条数，增量变化服务队列

读写锁分离，读操作加读锁，大量的读操作都是可以并发起来的；写操作加写锁；synchronized，读的时候，大量的线程在串行的读

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/125\_生产环境的锁故障：死锁现象演示以及jstack分析死锁问题/笔记.doc**

基于 服务注册表缓存机制的隐患代码，演示一下死锁的发生，jstack怎么来分析死锁问题

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/120\_锁优化策略：尽可能对不同功能分离锁的使用/笔记.doc**

就是如果可能的话，比如说你有一个锁，你看看能不能按照这个功能的不同，拆分为两把锁，在使用不同的功能的时候，可以用不同的锁，这样降低线程竞争锁的冲突。阻塞队列，人家在实现源码的时候，就使用了两把锁

队头是一把锁，队列尾巴是一把锁，你从队列尾巴插进去是加的一把锁，从队头消费数据使用的是另外一把锁，入队和出队的操作，就不会因为锁产生冲突了

案例，并发集合的源码的时候，我们可以来看一下，锁分离的策略，在后面我们优化微服务注册中心的架构的时候，考虑

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/128\_生产环境的锁故障：线程饥饿、活锁以及公平锁策略解决思路/笔记.doc**

线程饥饿，活锁

非公平锁，公平锁，是可能会导致你很早就过来排队等待竞争那个锁，但是呢，由于非公平策略，高并发的环境下，导致大量的线程抢占锁的问题很严重，可能会导致有的线程等待了很久很久一直获取不到锁

线程饥饿

要解决线程饥饿的问题，非公平锁的策略调整为公平锁，大家按照顺序来排队，先到的先加锁，绝对不会出现线程饥饿

活锁 -> 线程一直在运行着，但是始终无法完成自己的任务

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/117\_锁优化案例：分布式存储系统edits log的分段加锁机制/笔记.doc**

分布式存储系统案例，edits log的分段加锁机制，是如何尽可能的减少加锁的时间的

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/119\_锁优化案例：每秒上千订单场景的分布式锁高并发优化实战/笔记.doc**

《每秒上千订单场景下的分布式锁高并发优化实践》

带大家来做一个案例，分布式锁的一个优化案例，锁优化的思路，其实用的就是上一讲说的那个降低加锁粒度的策略

石杉的架构笔记： 《每秒上千订单场景下的分布式锁高并发优化实践》

比较详细的记录了一下我们之前积累的一套分布式锁在高并发场景下的优化实践，分段加锁，降低加锁粒度，仅仅对部分数据加锁，不要对完整的数据来加锁

电商的下单，扣减库存的例子来举例，直接用分布式锁的话会导致下单的并发性能很差，但是如果用了优化策略之后就可以大幅度的提升分布式锁的并发能力

加分布式锁，查询库存，如果库存充足，则扣减库存，释放分布式锁 -> 保证库存不会超发

分段加锁，将一个库存拆分为N个库存段，10个库存段，每个库存段是100个库存，一个请求过来，直接就是做一个随机算法，直接随机从10个库存段里挑选出来一个库存段，来进行加锁，执行业务逻辑

大量的线程过来，并发性一下子提升了10倍，100个线程，争用一个锁；10个线程争用一个锁，并发性能一下子就提升了10倍

如果说一个分段库存是0呢？

如果说一个分段库存不是0呢？但是你要购买的数量大于那个分段的库存的数量，此时怎么办呢？对更多的分段库存加锁，合并扣减库存。

分布式锁在高并发场景下的优化的案例实践，电商的库存扣减的例子来举例，做两个分段扣库存的策略

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/109\_锁优化案例：服务优雅停机机制中的volatile标志位修改实践/笔记.doc**

回顾一下，volatile的锁优化，我们之前大量的已经用过了

如果你要基于synchronized也可以实现可见性，你在synchronized里做的写操作，一定会刷回主存，后面你在synchronized里做的读操作，会强制性从主存拉读取数据，保证了可见性的问题

项目实践了，锁优化，体会的价值是无价的

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/122\_锁优化策略：尽量减少高并发场景中线程对锁的争用/笔记.doc**

（1）减少锁占用的时间

（2）数据的分段加锁

（3）减少线程对锁的抢占和争用的频率：微服务注册中心，完全可以用这个策略来进一步的优化服务注册表的并发冲突问题

读写锁主要是解决了大量的读请求不会串行化，读请求可以并发起来

写锁和读锁，还是会冲突问题，如果有服务实例注册、下线，写锁，此时还是会短暂的影响人家来读注册表的数据，读锁是加不上的

降低锁竞争的频率

《【双11狂欢的背后】微服务注册中心如何承载大型系统的千万级访问？》

讲了服务注册表有一套多级缓存的机制

建议还是去看一下，java架构，大数据架构，尽量建议先看spring cloud源码分析的课，那个课程是给大家一个分布式服务框架源码的剖析，算是一个架构系列开始的启蒙课，spring clodu源码本身是比较简单的

eureka多级缓存架构和机制

ReadOnlyMap，ReadWriteMap

读请求，全部是从ReadOnlyMap里面来走的，如果这一级缓存里没有，则从ReadWriteMap里找，如果也没有，则从服务注册表来加载

ReadOnlyMap，大量的线程并发的读服务注册表的数据，在这里是不需要加锁的，直接缓存，大家直接读，不需要加读锁，大量的读锁和写锁的冲突了

只有在ReadOnlyMap里面没数据的时候，此时会加一个synchronized锁，只有一个线程可以去找ReadWriteMap，去找服务注册表的数据，此时会加服务注册表的读锁，但是找到了书数据，填充完了量级缓存之后

释放synchronized锁，前天线程又可以直接读缓存里的数据

将服务注册表的读锁，降低到了，很少的频率，某个线程发现缓存里没数据，要直接读服务注册表的数据的时候，此时才会加读锁

如果更新数据之后，此时会过期掉ReadWriteMap，有一个后台线程，会隔30秒后，去过期掉ReadOnlyMap里的数据，此时缓存清空

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/111\_锁优化案例：服务心跳计数器中的Atomic原子类落地使用/笔记.doc**

Atomic在实战中到底是如何来使用的，如何优化锁的呢？

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/123\_锁优化案例：采用多级缓存机制降低对服务注册表的锁争用/笔记.doc**

代码实现一下多级缓存机制，文章，spring cloud源码剖析的课，eureka源码的课，好好的思考一下，大幅度的降低了高频繁的读操作对服务注册表加读锁的请求，避免了频繁的服务注册表的读锁和写锁的竞争的冲突

系统刚启动的时候，会有一个线程来填充各级缓存的数据

此后的30秒内，大家全部都是读缓存数据的，不会涉及到任何加锁的行为

在这个过程中，如果有人更新注册表的数据的话，一方面会对注册表本身加写锁，另外一方面对缓存加一个锁，那么会过期掉readWriteMap的缓存。此时所有加的锁，是不会对高频的读请求有任何的锁的冲突和影响的

在写数据的期间，读数据不涉及任何读写锁的冲突，直接读的cache数据

有一个后台的线程，可能会过30秒之后，对缓存加个锁，然后同步两个缓存的数据，在这个过程中，实际上来说也是不会对高频的读操作施加任何的影响的

只有在此时，会有线程感知到缓存数据是null，重新填充数据，重新填充数据的时候，会涉及到重新从服务注册表查数据，然后加读锁，此时就是一个线程加了一个读锁，而且是很快的行为，大量的降低了频繁的读操作，可能频繁的跟写操作，读写锁冲突的问题

大幅度的降低了读写锁的互斥冲突的问题。避免了说高频的读操作，对服务注册表频繁的读取，频繁的加读锁，导致跟服务注册表的写锁频繁的互斥和冲突

各种锁的运用，加了两级缓存，尽可能的把读写锁的互斥和冲突，降低到了最低

为什么是两级缓存？不是一级缓存呢？

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/107\_多线程并发运行的环境中大量的锁争用对性能的影响/笔记.doc**

多个线程自顾自的来运行肯定是没关系的，但是问题就在于说，如果多个线程开启了之后来访问一些共享的数据，内存里共享的成员变量，就会比较坑爹了，你一定要对一些数据进行并发的修改

如果要同时修改数据，而且是比较复杂的更新的一些操作，对自我保护机制的阈值进行了较为复杂的同时并发的读和写，加锁，synchronized

线程通信（控制），wait和notify，分布式存储系统的edits log机制来演示了一下，是如何在真实工业级系统中来使用的

比较简单，累加一个数值，synchronized就太重了，可以用Atomic原子类，CAS的原理，如何通过CAS无锁化的思想，来实现并发更细数值

（1）线程创建以及控制

（2）可见性以及volatile的使用

（3）synchronized、wait以及notify的使用

（4）Atomic原子类

（5）AQS、Lock以及Condition，读写锁

（6）ThreadLocal线程本地副本

JDK 1.6以后对synchornized做了很多的优化，性能比较好；如果可以用读写锁拆分，Lock

讲到这里，咱们的话就是并发讲了不少的东西了

如果你volatile也不好用，Atomic也不好用，ThreadLocal也不好用，读写锁也不好用，必须得用synchronized加锁互斥，才能保证多线程并发的安全的更新共享数据，高并发场景下，大量的线程都在争用这个锁的话

可能导致大量的线程在高并发的时候，一个一个串行化的来执行一些操作，可能一旦你的并发技术用的不好的话，锁争用太剧烈，会导致你的系统的吞吐量会大幅度的下降，导致你的系统无法承载高并发的访问

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/121\_锁优化策略：避免在循环中频繁的加锁以及释放锁/笔记.doc**

循环中频繁加锁

如果是正常的代码流程，尽量避免在for循环里频繁的加锁释放锁

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/116\_锁优化策略：尽可能减少线程对锁占用的时间/笔记.doc**

读写锁用不了，synchronized锁，读写锁，加锁，有一个核心点，尽量保证你加锁的时间是很短的，不要在加锁之后，执行一些磁盘文件读写、网络IO读写，导致锁占用的时间过于长

一般 来说，我们建议，加锁，尽量就是操作一下内存里的数据就可以了，不要在锁里面去执行一些耗时的一些操作，比如说执行数据库操作，SQL，或者是别的一些东西，可能会导致占用锁的时间

就会导致线程并发的吞吐量大幅度的下降，并发能力就很弱，性能很差

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/127\_生产环境的锁故障：锁死问题的产生原因以及解决思路/笔记.doc**

死锁，是并发编程里，线上生产环境里，最大的问题

锁死，synchronized，wait陷入永久等待，要靠其他线程来notify唤醒他，代码写的不好的话，有的线程陷入了一个wait状态，永久等待，结果其他的线程你的代码里忘了notify的一行代码

jstack -> dump快照，分析线程栈里的调用情况，线程的状态，有人处于wait的状态

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/126\_生产环境的锁故障：优化注册表缓存机制中的死锁隐患的代码/笔记.doc**

改写那个导致死锁的那块代码就可以了

避免说，线程1先加锁1然后加锁2；线程2先加锁2然后加锁；大家按照一样的顺序来加锁就可以了

线程1先加锁1然后加锁2，线程2先加锁1然后加锁2，死锁问题了

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/112\_锁优化策略：数据允许多副本场景优先使用ThreadLocal/笔记.doc**

volatile、CAS

如果你不需要多个线程共享读写一个数据的话，可以让每个线程保持一个本地变量的副本的话，那么你其实可以搞一个ThreadLocal，让每个线程都维护一个变量的副本，每个线程就操作自己本地的副本就可以了

txid，requestId，每个线程就把自己的那个值放到自己本地副本里，后续自己来修改和读取就可以了，跟其他的线程之间就没有什么冲突了

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/104\_ThreadLocal：让每个线程都有一个副本/笔记.doc**

并发编程里非常常用的一个东西，ThreadLocal，线程本地副本

多线程并发安全问题主要出现在哪里呢？多个线程并发访问同一个共享数据的时候，才会有问题，java内存模型，并发修改同一个数据的时候，可能会导致数据错乱，必须要加一些并发同步机制

volatile可见性 -> 原子性，Atomic数值累加的原子性

共享数据吗？给每个线程拷贝一个线程自己本地的变量副本，每个线程就直接操作自己的本地副本就ok了，然后就跟其他的线程就没有冲突了

Long orderId，Long txid，数值，读一个数值 + 修改这个数值，每个线程就更新自己本地的数值就可以了，Long requestId，做成共享的，直接就每个线程自己维护一个副本，读取和更新自己的副本就可以了

避免多个线程并发的访问同一个共享的数据

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/114\_锁优化策略：读多写少需要加锁的场景优先使用读写锁/笔记.doc**

volatile、Atomic、ThreadLocal，都不适用呢？

多线程并发访问一块共享数据，就需要加锁了，优先考虑读写锁，synchronized重量级

读多写少的场景，读写锁分离，读锁 -> 大量的线程并发的读，写锁 -> 写数据其他人不能写同时来写，也不能有人来读

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/101\_学员问题答疑：分布式存储系统案例中有一个bug/笔记.doc**

分布式存储系统案例，代码里有一个bug

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/118\_锁优化策略：尽可能减少线程对数据加锁的粒度/笔记.doc**

（1）分段加锁，实践，尽可能的减少一个线程占用锁的时间

（2）尽可能减少你对数据加锁的粒度

比方说，你手上有一份数据，里面包含了多个子数据，你加锁，可以对一整块完整的大数据来加锁，别人只要访问这一大块数据，都会有锁的争用的问题。你也可以选择降低加锁的粒度，你仅仅对大块数据里的部分子数据加锁

如果别的线程去请求这个大块数据里，其他的子数据的话，就不会跟你的锁产生冲突

更少的数据，或者是对更少的代码进行加锁

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/100\_学员问题答疑：AtomicReference用的好像有点问题？/笔记.doc**

AtomicReference在项目里用的有问题，解释的太明白，没有理解我的意思

问题1：AtomicReference不是这么用的

问题2：AtomicLong，保证了版本不会错乱，就可以不需要使用AtomicReference了呢？

AtomicReference呢？AtomicLong，多个变量，要实现原子性的更新

public class MyObject {

private int a = 0;

private int b = 0;

private int c = 0;

}

AtomicReference<MyObject> myObjectRef = new AtomicReference<MyObject>(myObject);

每次修改了三个abc的变量的值以后，就搞一个新的MyObject的对象封装新的值，更新myObjectRef变量的时候，CAS，先get一下，CAS，比较一下，如果之前没有人更新过这个对象，那么我就来更新

CAS失败了呢？此时你需要重新get一下最新的值，再次修改，修改完了以后搞一个新的对象，再次CAS赋值

原子性

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/108\_锁优化策略：标志位修改等可见性场景优先使用volatile/笔记.doc**

如果你一旦开启了多线程，分布式系统，中间件系统

多线程访问一些共享的变量或者数据，所以以后你在写这种并发的代码的时候，你要考虑清楚一点，你的多个线程并发访问一些共享数据的时候，到底是怎么个并发访问的场景，是并发的写呢？

还是并发的读呢？

如果仅仅只是有一些线程会来写一个变量，标志位，另外一个线程是来读取这个标志位的值，那么此时优先使用volatile

能不用锁尽量别用锁，就会比较麻烦，可能会导致锁争用和冲突，如果没弄好的，锁的问题，可能会导致系统的吞吐量、性能会大幅度的降低

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/103\_案例实战：基于读写锁优化服务注册表的读写加锁操作/笔记.doc**

实战，对服务注册表里面设置两把锁，一个是读锁，一个是写锁

读的操作都是可以并发的执行的

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/115\_锁优化案例：服务注册表的并发读写场景采用读写锁分离策略/笔记.doc**

服务注册表的读写锁分离

读多，写少，每隔30秒拉拉取增量注册表，发送心跳，大多数的操作都是走读锁，并发的来实现读，synchronized的话，大量的读都是串行化，一个一个的读，导致服务注册中心的并发的性能很差

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/110\_锁优化策略：数值递增场景优先使用Atomic原子类/笔记.doc**

多线程并发访问一些共享数据，先分析和判断一下，那个变量是不是有人读，有人写，直接就是volatile就可以了。如果大家都要写，再判断一下，仅仅只是简单的数值累加或者变更，数值的一些操作

建议大家可以用 Atomic原子类，CAS机制，无锁化，并发性要比synchronized要好不少的，相当于简单的更新数值，都要一个一个线程依次加锁进入，修改，再释放锁，这个过程其实是并发性还是要差一些的

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/100~128资料/124\_生产环境的锁故障：注册表缓存机制中潜在的死锁问题/笔记.doc**

注册表的缓存机制，里面其实是有死锁的潜在的隐患问题的

死锁问题，必须得有两把锁，线程1先获取到了锁1，然后尝试获取锁2；但是线程2获取到了锁2，尝试获取锁1

死锁

尤其容易出现死锁的地方，就是一段代码里，先加了一个锁，然后没释放锁，直接尝试获取另外一个锁；另外一段代码，有类似的思路，只不过获取锁的顺序，反了一下，获取了另外一把锁，然后尝试获取第一把锁

jstack怎么获取dump快照然后来看一下死锁发生的

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/18\_HashMap源码剖析（七）：通过红黑树来解决hash冲突/笔记.doc**

假设现在某个地方已经是一颗红黑树了

如果此时在那个地方再次出现一个hash冲突的话，怎么办呢？此时就应该是在红黑树里插入一个节点了，不是说挂链表了，红黑树是一个平衡的二叉查找树，平衡的，插入的时候还挺复杂的，变色、旋转

e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);

这行代码里面，就会调用TreeNode.putTreeVal()方法，这个方法的源码我们不看了，你大概都可以想象到就可以基于红黑树的规则，保持平衡的前提下，插入一个节点在红黑树里面，大概 是这个意思，代码会非常的复杂，我们不看了

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/13\_HashMap源码剖析（二）：核心成员变量的作用分析/笔记.doc**

为什么要带着大家看一下源码？？

你也可以就是简单看一些博客，说一下原理，你看过源码以后，对集合包的原理的理解和深度就完全不一样了，比如ArrayList，数组，扩容，但是如果你看完了源码以后，你的理解会更加的深刻一些

你在面试表达的时候，你理解的深度所表达出来的东西，都是不一样的

static final int DEFAULT\_INITIAL\_CAPACITY = 1 << 4; // aka 16

应该是数组的默认的初始大小，是16，这个跟ArrayList是不一样的，初始的默认大小是10

static final float DEFAULT\_LOAD\_FACTOR = 0.75f;

这个数组的大小，一般会自己手动指定一下，就跟你用ArrayList一样，你需要去预估一下你的这个数据结构里会放多少key-value对，指定的大一些，避免频繁的扩容

这个参数，默认的负载因子，如果你在数组里的元素的个数达到了数组大小（16） \* 负载因子（0.75f），默认是达到12个元素，就会进行数组的扩容

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

final int hash;

final K key;

V value;

Node<K,V> next;

}

这是一个很关键的内部类，他其实是代表了一个key-value对，里面包含了key的hash值，key，value，还有就是可以有一个next的指针，指向下一个Node，也就是指向单向链表中的下一个节点

通过这个next指针，就可以形成一个链表

transient Node<K,V>[] table;

这个是什么东东？Node<K, V>[]，这个数组就是所谓的map里的核心数据结构的数组，数组的元素就可以看到是Node类型的，天然就可以挂成一个链表，单向链表，Node里面只有一个next指针

transient int size;

这个size代表的是就是当前hashmap中有多少个key-value对，如果这个数量达到了指定大小 \* 负载因子，那么就会进行数组的扩容

int threshold;

这个值，其实就是说capacity（就是默认的数组的大小），就是说capacity \* loadFactory，就是threshold，如果size达到了threshold，那么就会进行数组的扩容，如果面试这么回答，你就是在打自己的脸

负载因子，默认值，threshold，扩容，扩容，rehash的算法，JDK 1.7以前的rehash是怎么做的，性能有多差，JDK 1.8以后是如何优化的rehash的算法，东西很多的

final float loadFactor;，默认就是负载因子，默认的值是0.75f，你也可以自己指定，如果你指定的越大，一般就越是拖慢扩容的速度，一般不要修改

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/25\_Iterator迭代器应对多线程并发修改的fail fast机制/笔记.doc**

java集合中，迭代器在迭代的时候，他的fail-fast机制

ConcurrentModificationException，并发修改的异常，这个机制就叫做fail fast

modCount就是用来实现fail fast机制的，各个集合里面其实都有这个modCount的概念，只要这个集合被修改了，那么就会对modCount++

modificationCount，修改次数，只要你修改一次，就会更新这个，add、remove、set

比如说在迭代一个ArrayList之前，已经插入了4个元素，此时modCount = 4，在你获取和初始化一个迭代器的时候，里面的expectedModCount就会被初始化为modCount

throw new ConcurrentModificationException();，并发修改冲突异常

java集合包下的类，都是非线程安全的，所以说里面都设计了针对并发修改集合的问题，有fail fast机制，modCount

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/02\_ArrayList源码剖析（一）：基本原理以及优缺点/笔记.doc**

JDK集合包下面的源码，就把java里面的核心以及常用的集合的源码，都给大家来讲一下，ArrayList，这个东西，最最最最常用的一种集合

只要是刚入行的年轻的java工程师，也会有人告诉他说，兄弟，ArrayList，我告诉你，原理就是底层基于数组来实现，缺点是什么呢？优点是什么呢？

数组的长度是固定的，java里面数组都是定长数组，比如数组大小设置为100，此时你不停的往ArrayList里面塞入这个数据，此时元素数量超过了100以后，此时就会发生一个数组的扩容，就会搞一个更大的数组，把以前的数组拷贝到新的数组里面去

这个数组扩容+元素拷贝的过程，相对来说会慢一些

不要频繁的往arralist里面去塞数据，导致他频繁的数组扩容，避免扩容的时候较差的性能影响了系统的运行

缺点二：数组来实现，数组你要是往数组的中间加一个元素，是不是要把数组中那个新增元素后面的元素全部往后面挪动一位，所以说，如果你是往ArrayList中间插入一个元素，性能比较差，会导致他后面的大量的元素挪动一个位置

优点，基于数组来实现，非常适合随机读，你可以随机的去读数组中的某个元素，list.get(10)，相当于是在获取第11个元素，这个随机读的性能是比较高的，随机读，list.get(2)，list.get(20)，随机读list里任何一个元素

因为基于数组来实现，他在随机获取数组里的某个元素的时候，性能很高，他可以基于他底层对数组的实现来快速的随机读取到某个元素，直接可以通过内存地址来定位某个元素。

ArrayList，常用，如果你不会频繁的在里面插入一些元素，不会导致频繁的元素的位置移动、数组扩容，就是有一批数据，查询出来，灌入ArrayList中，后面不会频繁插入元素了，主要就是遍历这个集合，或者是通过索引随机读取某个元素

使用ArrayList还是比较合适的

若果你涉及到了频繁的插入元素到list中的话，尽量还是不要用ArrayList，数组，定长数组，长度是固定的，元素大量的移动，数组的扩容+元素的拷贝

开发系统的时候，大量的场景，需要一个集合，里面可以按照顺序灌入一些数据，ArrayList的话呢，他的最最主要的功能作用，就是说他里面的元素是有顺序的，我们在系统里的一些数据，都是需要按照我插入的顺序来排列的

一个班级里面，有一堆学生，需要对这些学生按照插入的顺序来排序，ArrayList，而且后面这里面的数据不会频繁的变化的，无论插入元素，还是随机删除某个元素

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/10\_Vector和Stack：栈数据结构的源码剖析/笔记.doc**

栈，Vector和Stack两个来实现的，Stack代表了一个栈这种数据结构，他是继承自Vector来实现的，Vector是一种类似于ArrayList（基于数组来实现的）数据结构，有序的集合，Stack是一种基于数组来实现的栈数据结构

栈，先进后出，跟队列的先进先出是不一样

队列：一般是队尾巴进去开始排队，从队头开始出来，排队的过程，先进先出

栈：进去的时候直接压入栈底，出来的时候是从栈的最上面一个元素开始先出来，先进后出

栈的压入栈底的时候，push()方法，几乎就跟ArrayList.add()方法的实现源码是一样的，就是放在数组的按顺序排列的位置上

public synchronized void addElement(E obj) {

modCount++;

ensureCapacityHelper(elementCount + 1);

elementData[elementCount++] = obj;

}

ArrayList每次扩容是1.5倍，capacity + (capacity >> 1) = 1.5

Vector每次扩容默认是2倍，默认情况下是直接扩容两倍，2倍

看JDK的源码，还可以学习一下人家是如何使用System.copy、Arrays.copyOf，人家是如何使用JDK封装好的一些基础工具类方法的

压入栈底的时候，elementData[size++] = element，这样的一行代码

pop()方法，从栈顶弹出来一个元素，最后一个压入栈的元素，会通过pop()方法弹出来，先是使用elementData[size - 1]获取最后一个元素，返回给用户，removeElementAt(size - 1)删除了最后一个元素

10 - (10 - 1) - 1 = 0

j = 0

elementData[size - 1] = null，直接将最后一个元素设置为Null就可以了

这个非常的简单，栈的使用，一般就是push() + pop()，压入栈底，从栈顶弹出来，这个东西呢，非常的简单

布置了一个作业，这个作业，自己纯手写，将ArrayList、LinkedList、Stack三种数据结构，都自己手写来实现一下

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/08\_LinkedList源码剖析（四）：获取元素的原理/笔记.doc**

获取头部元素，尾部元素，中间的某个元素

getFirst() == peek()：获取头部的元素，他其实就是直接返回first指针指向的那个Node他里面的数据，他们都是返回头部的元素。getFirst()如果是对空list调用，会抛异常；peek()对空list调用，会返回null

getLast()：获取尾部的元素

public E get(int index) {

checkElementIndex(index);

return node(index).item;

}

随机获取某个位置的元素，这个操作是ArrayList的强项，是通过数组来，直接通过数组的index就可以定位到那个元素，性能超高

LinkedList而言，get(int index)这个方法，是他的弱项，也是他的缺点，如果他要获取某个随机位置的元素，需要使用node(index)这个方法，是要进行链表的遍历，会判断一下index和size >> 1进行比较，如果在前半部分，就会从头部开始遍历；如果在后半部分，就会从尾部开始遍历

学一下人家JDK里面的链表遍历的优雅写法，候选人来我这里面试，我出一个特别简单的题目，就是自己实现一个双向链表的数据结构，写一个链表遍历的算法，代码，写出来，很多人在那儿纠结半天

连一个链表遍历的代码都写不好，coding功底有多差

好好看一些JDK源码，或者是刷一些leetcode上面的一些数据结构和算法题，编码的功底就会比较好，写出的代码，看着比较优雅

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/16\_HashMap源码剖析（五）：hash冲突时的链表处理/笔记.doc**

就是说，假设某两个key的hash值一样的，两个key不同，hash值一样， 这个概率其实很低很低，除非是什么呢？就是说你自己乱写了hashCode()方法，你自己人为的制造了两个不同的key，但是hash只一样

两个key的hash值不一样，但是通过寻址算法，定位到了数组的同一个key上去，此时就会出现典型的hash冲突，默认情况下，会用单向链表来处理

if ((p = tab[i = (n - 1) & hash]) == null)

这个分支，他的意思是说tab[i]，i就是hash定位到的数组index，tab[i]如果为空，也就是hash定位到的这个位置是空的，之前没有任何人在这里，此时直接是放一个Node在数组的这个位置即可

else

如果进入else，就说明通过hash定位到的数组位置，是已经有了Node了

if (p.hash == hash &&

((k = p.key) == key || (key != null && key.equals(k))))

e = p;

// 如果满足上述条件，说明是相同的key，覆盖旧的value

// map.put(1, “张三”)

// map.put(1, “李四”)

if (e != null) { // existing mapping for key

V oldValue = e.value;

// 张三就是oldValue

if (!onlyIfAbsent || oldValue == null)

e.value = value;

// value是新的值，是李四

// e.value = value，也就是将数组那个位置的Node的value设置为了新的李四，这个值

afterNodeAccess(e);

return oldValue;

}

上面那坨代码，其实说白了，就是相同的key在进行value的覆盖

如果上面那个if不成立，说明人家的key是不一样的，hash值不一样，或者是key不一样

else if (p instanceof TreeNode)：这个分支，是下下讲要给大家分析的，是说，如果这个位置已经是一颗红黑树的话，会怎么来处理

else {

直到进入这个else分支，才是说，key不一样，出现了hash冲突，然后此时还不是红黑树的数据结构，还是链表的数据结构，在这里，就会通过链表来处理

if (binCount >= TREEIFY\_THRESHOLD - 1) // -1 for 1st

treeifyBin(tab, hash);

break;

这串东西的意思，就是说如果当前链表的长度（binCount），大于等于TREEIFY\_THRESHOLD - 1的话，如果链表的长度大于等于8的话，链表的总长度达到8的话，那么此时就需要将这个链表转换为一个红黑树的数据结构

这个如何转换，咱们下一讲来看一下

再来想一想，假设如果此时已经链表有2个节点了，那么此时如果再来一个hash冲突，再挂一个节点，会怎么来挂呢？

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/15\_HashMap源码剖析（四）：put操作原理以及hash寻址算法/笔记.doc**

我们这一讲来看一下正常的put操作的流程

public V put(K key, V value) {

return putVal(hash(key), key, value, false, true);

}

hash(key) -> hash

key

value

false

true

假设，hashmap是空的，数组大小就是默认的16，负载因子就是默认的12

if ((tab = table) == null || (n = tab.length) == 0)

n = (tab = resize()).length;

刚开始table数组是空的，所以会分配一个默认大小的一个数组，数组大小是16，负载因子是0.75，threshold是12

(n - 1) & hash

n = 16

n - 1 = 15

15 & hash

1111 1111 1111 1111 0000 0101 1000 0011

0000 0000 0000 0000 0000 0000 0000 1111

-> 与操作

就是必须都是1，才是1，否则就是0

0000 0000 0000 0000 0000 0000 0000 0011，转成10进制，就是3，index = 3

他的hash寻址的算法， 并不是说用hash值对数组大小取模，取模就可以将任意一个hash值定位到数组的一个index那儿去，取模的操作性能不是太高

位运算，性能很高，&与操作，来实现取模的效果

他优化以后的一个效果，就是说他的数组刚开始的初始值，以及未来每次扩容的值，都是2的n次方

16，2的4次方

32，2的5次方

64，2的6次方

也就是说他后面每次扩容，数组的大小就是2的n次方，只要保证数组的大小是2的n次方，就可以保证说，(n - 1) & hash，可以保证就是hash % 数组.length取模的一样的效果，也就是说通过(n - 1) & hash，就可以将任意一个hash值定位到数组的某个index里去

因为他不想用取模，取模的性能相对较低，这个是他的一个提升性能的优化点

这也是hashmap底层原理里面的重要部分

i = (n - 1) & hash，i就是最后寻址算法获取到的那个hash值对应的数组的index

tab[i]不就是直接定位到了数组的那个位置了么？

刚开始肯定是空，就直接创建一个Node出来，代表了一个key-value对，放在数组的那个位置就可以了

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/05\_LinkedList源码剖析（一）：基本原理以及优缺点/笔记.doc**

LinkedList，底层是基于链表来实现的

刚入行的工程师，都知道ArrayList和LinkedList的区别，这个东西是java面试的时候基础面试题，都是面校招生，工作经验在2年以内的人的

LinkedList，链表，一个节点挂着另外一个节点

优点在哪儿？往这个里面中间插入一些元素，或者不停的往list里插入元素，都没关系，因为人家是链表，中间插入元素，不需要跟ArrayList数组那样子，挪动大量的元素的，不需要，人家直接在链表里加一个节点就可以了

如果你不断的往LinkedList中插入一些元素，大量的插入，就不需要像ArrayList数组那样还要去扩容啊什么的，人家是一个链表，就是不断的把新的节点挂到链表上就可以了

LinkedList的优点，就是非常适合各种元素频繁的插入里面去

LinkedList的缺点，不太适合在随机的位置，获取某个随机的位置的元素，比如LinkedList.get(10)，这种操作，性能就很低，因为他需要遍历这个链表，从头开始遍历这个链表，直到找到index = 10的这个元素为止

ArrayList.get(10)，不需要遍历，直接根据内存的地址，根据你指定的index，直接定位到那个元素，不需要遍历数组什么的

ArrayList和LinkedList区别，数组和链表的区别，优缺点主要就是这样子

使用场景

（1）ArrayList：一般场景，都是用ArrayList来代表一个集合，只要别频繁的往里面插入和灌入大量的元素就可以了，遍历，或者随机查，都可以

（2）LinkedList：适合，频繁的在list中插入和删除某个元素，然后尤其是LinkedList他其实是可以当做队列来用的，这个东西的话呢，我们后面看源码的时候，可以来看一下，先进先出，在list尾部怼进去一个元素，从头部拿出来一个元素。如果要在内存里实现一个基本的队列的话，可以用LinkedList

我们之前在电商系统开发的时候，在第一个版本中，用到了内存队列，用的就是LinkedList，他里面基于链表实现，天然就可以做队列的数据结构，先进先出，链表来实现，特别适合频繁的在里面插入元素什么的，也不会导致数组扩容

其实在工作中使用的时候，都是用的是java并发包下面的一些线程安全的集合，这个东西等到讲java并发包的时候，我们可以再来看一下

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/19\_HashMap源码剖析（八）：基于数组的扩容原理图解/笔记.doc**

我跟大家说一下，hashmap底层是基于数组来实现的核心的数据结构，如果是用数组的话，就天然会有一个问题，就跟ArrayList一样，就是数组如果满了，就必须要扩容，hashmap所以也是有扩容的问题存在的

这一讲我们先不看源码，先来画图说一下hashmap扩容的原理

非常简单，2倍扩容 + rehash，每个key-value对，都会基于key的hash值重新寻址找到新数组的新的位置

本来那个数组的长度假设是16，现在的话新数组的长度是32

本来所有的key的hash，对16取模是一个位置，比如说是index = 5；但是如果对32取模，可能就是index = 11,，位置可能变化

问你，hashmap扩容的原理，数组，一次扩容多大，2倍，rehash过程，基于key的hash值重新在新的数组里找到新的位置，很多key在新数组的位置都不一样了，如果是之前冲突的这个key可能就会在新的数组里分布在不同的位置上了

这个原理大体上是JDK 1.7以前的原理，现在的话呢，JDK 1.8以后，都是数组大小是2的n次方扩容，用的是与操作符来实现hash寻址的算法，来进行扩容的时候，rehash

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/03\_ArrayList源码剖析（二）：核心方法的原理/笔记.doc**

咱们来启动一个demo工程，在里面写写集合的代码，跟进去看看各种集合的实现原理，直接可以看JDK底层的源码

默认的构造函数，直接初始化一个ArrayList实例的话，会将内部的数组做成一个默认的空数组，{}，Object[]，他有一个默认的初始化的数组的大小的数值，是10，也就是我们可以认为他默认的数组初始化的大小就是只有10个元素

ArrayList的话，玩儿好的话，一般来说，你应该都不是使用这个默认的构造函数，你构造一个ArrayList的话，基本上来说就是默认他里面不会有太频繁的插入、移除元素的操作，大体上他里面有多少元素，你应该可以推测一下的

基本上最好是给ArrayList构造的时候，给一个比较靠谱的初始化的数组的大小，比如说，100个数据，1000,10000，避免数组太小，往里面塞入数据的时候，导致数据不断的扩容，不断的搞新的数组

ensureCapacityInternal(size + 1); // Increments modCount!!

你每次往ArrayList中塞入数据的时候，人家都会判断一下，当前数组的元素是否塞满了，如果塞满的话，此时就会扩容这个数组，然后将老数组中的元素拷贝到新数组中去，确保说数组一定是可以承受足够多的元素的

（1）add()方法的源码

public boolean add(E e) {

ensureCapacityInternal(size + 1); // Increments modCount!!

elementData[size++] = e;

return true;

}

elementData = []

size = 0

element[0] = “张三”

size++

elementData = [“张三”]

size = 1

elementData[1] = “李四”

size++

elementData = [“张三”, “李四”]

size = 2

elementData = [“张三”, “李四”, “王五”]

size = 3

（2）set()方法的源码

set(3, “赵六”) => 典型的数组越界

set(1, “赵六”)

E oldValue = elementData(index);

// 获取到了1这个位置的元素的值，李四

oldValue = “李四”

elementData[1] = “赵六”

elementData = [“张三”, “赵六”, “王五”]

返回了一个”李四”

（3）add(index, element)方法的源码

add(1, “麻子”)

index = 1

size = 3

System.arraycopy(elementData, index, elementData, index + 1, size - index);

System.arraycopy(elementData, 1, elementData, 2, 2);

你可以认为他这个方法的意思，就是说：elementeData这个数组，从第1位开始（第二个元素），拷贝2个元素，到elementData这个数组（还是原来的这个数组），从第2位开始（第三个元素开始）

elementData = [“张三”, “赵六”, “赵六”, “王五”]

elementData[1] = “麻子”

elementData = [“张三”, “麻子”, “赵六”, “王五”]

size++

size = 4

（4）get()方法的源码

这个方法是最简单了，直接elementData[index]，基于数组直接定位到这个元素，获取这个元素，这个是ArrayList性能最好的一个操作，优点所在

（5）remove()方法的源码

remove(1)

size = 4

index = 1

elementData = [“张三”, “麻子”, “赵六”, “王五”]

int numMoved = size - index - 1;

numMoved = 4 - 1 - 1 = 2

相当于是不是要把后面的“赵六”和“王五”都要往前面挪动一位

System.arraycopy(elementData, index + 1, elementData, index, numMoved);

System.arraycopy(elementData, 2, elementData, 1, 2);

把elementData数组中，从index = 2开始的元素，一共有2个元素，拷贝到elementData数组中（原来的数组里），从index = 1开始，进行拷贝

elementData = [“张三”, “赵六”, “王五, “王五”]

elementData[--size] = null

elementData[3] = null

elementData = [“张三”, “赵六”, “王五”]

size = 3

（6）源码分析的总结

remove()

add(index, element)

这个两个方法，都会导致数组的拷贝，大量元素的挪动，性能都不是太高，基于数组来做这种随机位置的插入和删除，其实性能真的不是太高

add()、add(index, element)，这两个方法，都可能会导致数组需要扩容，数组长度是固定的，默认初始大小是10个元素，如果不停的往数组里塞入数据，可能会导致瞬间数组不停的扩容，影响系统的性能

set()、get()，定位到随机的位置，替换那个元素，或者是获取那个元素，这个其实还是比较靠谱的，基于数组来实现随机位置的定位，性能是很高的

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/24\_HashSet、LinkedHashSet、TreeSet的源码剖析/笔记.doc**

Set的源码其实没什么好讲的，在面试的时候，主要人家就是问hashmap呢？ArrayList、LinkedList都很简单，没什么太多好问的。主要复杂的就是hashmap。linkedhashmap其实就是基于hashmap的。treemap。set都是直接基于map来实现的。

比如说HashSet就是基于HashMap来实现的

HashMap是不允许key重复的，他底层是一个数组，如果你的key重复了，你会hash寻址到数组的同一个位置去，然后覆盖原来的值

HashSet，他其实就是说一个集合，里面的元素是无序的，他里面的元素是没有重复的，HashMap的key是无顺序的，你插入进去的顺序，跟你迭代遍历的顺序是不一样的，而且HashMap的key是没有重复的，HashSet是不是直接就可以基于HashMap来实现啊

你不断的往HashSet里放入一些元素，人家底层就是不断的put到HashMap里去就ok了，如果你是从HashSet里进行遍历，人家就是直接遍历HashMap的key就可以了

LinkedHashSet，他是有顺序的set，也就是维持了插入set的这个顺序，你迭代LinkedHashSet的顺序跟你插入的顺序是一样的，底层是不是直接就可以基于LinkedHashMap来实现的

TreeSet，默认是根据你插入进去的元素的值来排序的，而且可以定制Comparator，自己决定排序的算法和逻辑，他底层是不是可以基于TreeMap来实现

Set底层的Map，只有key是有值的，value都是null值，都是空的

HashSet底层是基于HashMap来实现的，所以底层也是有数组的，扩容的问题，你可以在构造HashSet的时候就传入数组的大小

面试的时候，可能会有人问到你Set底层的实现原理是什么呢？Map来实现的，其实就是在map的key里放置，set的源码没多少行代码，非常的简单的，value都是一个空的对象

LinkedHashSet.add()方法，底层会调用LinkedHashMap.put()方法，此时在这个方法里就会记住加入元素的顺序，在一个链表中，后面你遍历的时候，是从LinkedHashMap里遍历元素，人家是直接遍历维护好的链表的

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/20\_HashMap源码剖析（九）：JDK 1.8的高性能rehash算法/笔记.doc**

JDK 1.8以后，为了提升rehash这个过程的性能，不是说简单的用key的hash值对新数组.length取模，取模给大家讲过，性能较低，所以JDK 1.8以后hash寻址这块，统一都是用的这个位操作

n - 1 0000 0000 0000 0000 0000 0000 0000 1111

hash1 1111 1111 1111 1111 0000 1111 0000 0101

&结果 0000 0000 0000 0000 0000 0000 0000 0101 = 5（index = 5的位置）

n - 1 0000 0000 0000 0000 0000 0000 0000 1111

hash2 1111 1111 1111 1111 0000 1111 0001 0101

&结果 0000 0000 0000 0000 0000 0000 0000 0101 = 5（index = 5的位置）

此时，上面两个hash值会出现hash碰撞的问题，使用链表，或者是红黑树来解决

如果数组的长度扩容之后 = 32，重新对每个hash值进行寻址，也就是用每个hash值跟新数组的length - 1进行与操作

n-1 0000 0000 0000 0000 0000 0000 0001 1111

hash1 1111 1111 1111 1111 0000 1111 0000 0101

&结果 0000 0000 0000 0000 0000 0000 0000 0101 = 5（index = 5的位置）

n-1 0000 0000 0000 0000 0000 0000 0001 1111

hash2 1111 1111 1111 1111 0000 1111 0001 0101

&结果 0000 0000 0000 0000 0000 0000 0001 0101 = 21（index = 21的位置）

00101 = 5

10101 = 21

hash2的位置，从原来的5变成了21，规律是什么？

也就是说，JDK 1.8，扩容一定是2的倍数，从16到32到64到128

就可以保证说，每次扩容之后，你的每个hash值要么是停留在原来的那个index的地方，要么是变成了原来的index（5） + oldCap（16） = 21

所以说，这个就是JDK 1.8以后，数组扩容的时候，元素重新寻址的一个过程和原理

如果面试官问你，hashmap的底层原理

（1）hash算法：为什么要高位和低位做异或运算？

（2）hash寻址：为什么是hash值和数组.length - 1进行与运算？

（3）hash冲突的机制：链表，超过8个以后，红黑树

（4）扩容机制：数组2倍扩容，重新寻址（rehash），hash & n - 1，判断二进制结果中是否多出一个bit的1，如果没多，那么就是原来的index，如果多了出来，那么就是index + oldCap，通过这个方式，就避免了rehash的时候，用每个hash对新数组.length取模，取模性能不高，位运算的性能比较高

通过这个方式的话，可以有效的将原来冲突在一个位置的多个key，给分散到新数组的不同的位置去

if (++size > threshold)

resize();

意思是说，每次你如果是put了一个新的key-value对之后，人家就会size++，每次都会比较一下size和threshold（数组的长度 \* 负载因子），resize()方法就是在扩容

else if ((newCap = oldCap << 1) < MAXIMUM\_CAPACITY &&

oldCap >= DEFAULT\_INITIAL\_CAPACITY)

newThr = oldThr << 1; // double threshold

newCap = oldCap << 1，就是乘以2，新数组的大小是老数组的2倍

Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];

新的数组是老数组的大小的两倍

if (e.next == null)

newTab[e.hash & (newCap - 1)] = e;

如果e.next是null的话，这个位置的元素不是链表，也不是红黑树

那么此时就是用e.hash & newCap（新数组的大小） - 1，进行与运算，直接定位到新数组的某个位置，然后直接就放在新数组里了

else if (e instanceof TreeNode)

((TreeNode<K,V>)e).split(this, newTab, j, oldCap);

如果这个位置是一个红黑树的话，此时会调用split()方法，人家肯定会去里面遍历这颗红黑树，然后将里面每个节点都进行重新hash寻址，找到新数组的某个位置

else { // preserve order

进入这个else分支的话，证明是链表

if (loTail != null) {

loTail.next = null;

newTab[j] = loHead;

}

if (hiTail != null) {

hiTail.next = null;

newTab[j + oldCap] = hiHead;

}

这块代码，大概的意思，就是我们之前给大家讲解的那样子的

大概的原理跟我们上面给大家讲解的那套东西是差不多那个意思，就是说他会判断一下，如果是一个链表里的元素的话，那么要么是直接放在新数组的原来的那个index，要么就是原来的index + oldCap

留一个小作业，就跟我上面讲的那种判断与运算结果变化的意思，是一样的，但是不是完全的一样，我倒是觉得大家可以自己好好的去推算一下这段代码块，一些hash值在进行与运算的时候，会如何变化

当然了，hashmap里面很多代码的细节，比如说红黑树的一些细节，然后包括resize那里，最后有一小坨代码没有给大家去推算，但是大体上核心的源码，核心的原理，给大家就讲清楚了，扣清楚每个细节的话，其实还是挺麻烦的

自身对hashmap技术的掌握来看，或者是你要以后出去面试来看，掌握到这个程度基本就够了，算是水平不错了

我还是想给大家就resize的原理做一点补充，担心大家看不懂那一块核心的代码

链表那块怎么来玩儿的代码

hash1 1111 1111 1111 1111 0000 1111 0000 0101

oldCap（16） 0000 0000 0000 0000 0000 0000 0001 0000

&运算 0000 0000 0000 0000 0000 0000 0000 0000

hash2 1111 1111 1111 1111 0000 1111 0001 0101

oldCap（16） 0000 0000 0000 0000 0000 0000 0001 0000

&运算 0000 0000 0000 0000 0000 0000 0001 0000 = 16 != 0

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/11\_手写List、队列、Stack数据结构的作业布置/笔记.doc**

我也发现了，很多同学coding的基础还是比较弱的，不是说你写过多少行代码，不是说你写过20年的代码，coding就一定很好的，写出来的代码比较糟糕，你有没有模仿和写过一些比较优秀的代码

ArrayList和LinkedList，自己基于数组和双向链表来实现这两种数据结构，以及他们里面的核心的一些操作，add、get、remove，插入元素、获取元素、删除元素，把这几种操作都自己实现一样，模仿JDK源码，自己写一下代码

我觉得对大家锻炼自己的优秀的coding能力，很有帮助

同时再手写一下Stack的数据结构，基于LinkedList相当于可以手写一下队列的数据结构，可以自己实现一下

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/23\_TreeMap源码剖析：自定义排序规则的红黑树map数据结构/笔记.doc**

TreeMap，顾名思义，底层是基于红黑树做的数据结构，不是传统意义上的那红HashMap，他天然就可以按照你的key的自然顺序来排序，既然人家是按照key的大小来进行排序和迭代输出的

人家肯定是用了一个按照key大小排序的数据结构，我们可以想到的就是红黑树，所以他底层是基于红黑树来实现的按key排序就可以了

static final class Entry<K,V> implements Map.Entry<K,V> {

K key;

V value;

Entry<K,V> left;

Entry<K,V> right;

Entry<K,V> parent;

boolean color = BLACK;

}

面试的时候，人家问到你TreeMap，你把这套东西给他讲一下，也就差不多了

透露一个面试的小秘诀，你要掌握主动性，比如人家让你聊hashmap，你直接夸夸夸，源码级别的，位运算，现场给他推算一下，显示你的JDK源码，LinkedHashMap、TreeMap，双向链表，如何基于插入顺序维护链表，迭代器是在迭代链表；底层是一颗红黑树，左小右大院里，挂成一个红黑树

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/22\_LinkedHashMap源码剖析：有顺序的map数据结构/笔记.doc**

LinkedHashMap，如果你把HashMap都听懂之后，LinkedHashMap其实并不难

HashMap，比如你放了一堆key-value对进去，后面的话呢如果你要遍历这个HashMap的话，遍历的顺序，并不是按照你插入的key-value的顺序来的

LinkedHashMap，他会记录你插入key-value的顺序， 如果你在遍历的时候，他是按照插入key-value对的顺序给你遍历出来的

LinkedHashMap是HashMap的一个子类，如果在面试的时候，可能会有的面试官问你，LinkedHashMap和TreeMap的区别，他们都可以维持key的顺序，只是LinkedHashMap底层是基于链表来实现的，TreeMap是基于红黑树来实现顺序的

LinkedHashMap其实原则上来说一些基本的原理和操作跟HashMap是差不多的，唯一主要的区别就是你在插入、覆盖、删除，他会记录一下key-value对的顺序，用一个链表来记录，在遍历的时候，就可以按照这个顺序来遍历

在调用LinkedHashMap的put()方法的时候，一定会调用到HashMap的put()方法里面去，调用完put()方法，插入一个key-value对之后，其实就会调用afterNodeInsertion(evict);，这个方法就会去回调LinkedHahsMap里面的子类的实现

void afterNodeInsertion(boolean evict) { // possibly remove eldest

LinkedHashMap.Entry<K,V> first;

if (evict && (first = head) != null && removeEldestEntry(first)) {

K key = first.key;

removeNode(hash(key), key, null, false, true);

}

}

所以说就是在这里，回调了这个方法，这个方法里面，他就是实现了LinkedHashMap的逻辑，来记录插入key-value对的顺序，用一个链表来记录

Node<K,V> newNode(int hash, K key, V value, Node<K,V> e) {

LinkedHashMap.Entry<K,V> p =

new LinkedHashMap.Entry<K,V>(hash, key, value, e);

linkNodeLast(p);

return p;

}

将节点封装为了一个LinkedHashMap.Entry对象，然后使用linkNodeLast(p)这个东西，将这个节点挂到一个链表里去

private void linkNodeLast(LinkedHashMap.Entry<K,V> p) {

LinkedHashMap.Entry<K,V> last = tail;

tail = p;

if (last == null)

head = p;

else {

p.before = last;

last.after = p;

}

}

一开始这个链表里就一个节点，所以tail和head两个指针，都会指向这个p

覆盖，如果是你再次将某个key的值覆盖一下，会怎么样呢？

如果我们是做key值的覆盖，可以看到，你多次覆盖一个值，不会改变他的顺序，LinkedHashMap有一个参数的，你可以在构造的时候传入进去，accessOrder，默认他是false，如果是默认为false的话，那么你比如说你get一个key，或者是覆盖这个key的值，都不会改变他在链表里的顺序

但是如果accessOrder是true的话，那么如果你get一个key，或者是覆盖这个key的值，就会导致个key-value对顺序会在链表里改变，他会被挪动到链表的尾部去，如果你把accessOrder指定为true，你每次修改一个key的值，或者是get访问一下这个key，都会导致这个key挪动到链表的尾部去

你删除某个元素的时候，就会将那个元素从链表里给摘除

在迭代的时候，LinkedHashMap里面会从链表的头部开始迭代，这样通过这个链表就可以维持他的一个顺序

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/04\_ArrayList源码剖析（三）：数组扩容以及元素拷贝/笔记.doc**

ArrayList里面最关键的一块，就是如果数组填充满了以后，如何进行扩容

ensureCapacityInternal(size + 1);

假设我们现在用的是默认的一个数组的大小，也就是10，现在呢已经往这个数组中添加了10个元素了，此时数组的size = 10，capacity = 10

此时调用add()方法插入一个元素，插入第11个元素咯，肯定是插入不进去的

ensureCapacityInternal(11)

calculateCapacity(elementData, minCapacity)

elementData已经填充了10个元素了，minCapacity = 11

elementData.length是默认的值，其实就是10，也就是说默认的情况下，这个数组最多只能放10个元素

要放第11个元素，elementData.length，最多只能放10个，同时现在已经放了10个了，你现在是要放第11个元素，所以此时会对数组进行扩容

int newCapacity = oldCapacity + (oldCapacity >> 1);

oldCapacity = 10

oldCapacity >> 1 = oldCapacity / 2 = 5

newCapacity = 15

elementData = Arrays.copyOf(elementData, newCapacity);

数组扩容的时候，他是怎么扩的，老的大小 + 老的大小 >> 1（相当于除以2），实现了一个数组的拷贝

最新的数组，变成了可以容纳15个元素的数组，但是此时数组中只有10个元素

看完了这个JDK的源码，非常简单的，算扩容的新数组的大小，你就知道是怎么来计算的，搞一个新数组，Arrays.copyOf()工具方法，完成老数组到新数组的一个拷贝

ArrayList源码，其实大家就已经看完了

基于数组来玩儿，最大的问题就是不要频繁的往里面灌入大量的数据，导致频繁的数组的扩容，新老数组元素拷贝，中间的位置插入元素，删除元素，会导致大量的元素的移动

随机获取一个元素，get(10)操作，性能极高，他适合随机读，不适合大量频繁的写入以及插入

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/07\_LinkedList源码剖析（三）：插入元素的原理/笔记.doc**

在尾部插入元素、在头部插入元素、在中间插入元素

add()，默认就是在队列的尾部插入一个元素，在那个双向链表的尾部插入一个元素

add(index, element)，是在队列的中间插入一个元素

addFirst()，在队列的头部插入一个元素

addLast()，跟add()方法是一样的，也是在尾部插入一个元素

add(2, element)，在index = 2的位置插入一个元素进去

Node<E> node(int index) {

// assert isElementIndex(index);

index = 2

假设是从first头结点开始遍历

i = 0

x = x.next = 第二个节点（index = 1）

i = 1

x = x.next = 第三个节点（index = 2）

i = 2，不成立，i = 2 = index，不小于index了

大家平时去外面面试的时候，不是会遇到一些数据结构和算法的题么，有些同学，可能连链表遍历的代码都不一定会写，这段代码就是非常经典的一个链表遍历的代码，通过头结点，不断的for循环，x指针不断的指向x.next，就可以实现链表的遍历的作用和功能

if (index < (size >> 1)) {

Node<E> x = first;

for (int i = 0; i < index; i++)

x = x.next;

return x;

} else {

// 就是如果index >= size / 2，说明要找的节点是在队列的后半部分

// 此时就是从队列的尾部往前遍历

Node<E> x = last;

for (int i = size - 1; i > index; i--)

x = x.prev;

return x;

}

}

上面的那段代码，还是比较经典的，大家一个可以学习一下JDK里的源码，是如何进行链表遍历的，这个是一个亮点。另外一个的话，要明白LinkedList在队列中间插入一个元素的原理，首先先找到那个index位置的元素，找这个元素的过程，会做一下判断，看看这个index是在前半部分还是后半部分，此时可以决定从头部还是尾部来进行遍历

这个其实就是说是在LinkedList中间插入一个元素的过程，涉及到了一些双向链表指针变换的过程

研究一下JDK的底层的源码，有另外一个好处，就是JDK的代码一般写的都是很精简，特别注重代码质量，JDK的代码的质量都是比较高的，数据结构的基础薄弱，JDK源码，学习一些最最基础的入门级别的数据结构的一些东西

完全可以学习一下JDK源码的实现，人家的各种list、queue、stack、map、set的数据结构的实现原理，代码是怎么写的，非常值得你来学习，我甚至建议大家如果有空闲的同学，仿照一下JDK对一些集合的实现原理，自己来实现一些LinkedList、ArrayList，把里面的核心方法的代码写一下

其实你如果能全部都手写一下，对你的一些编码的功底，基础的技术能力，都有很好的夯实的作用

是获取index位置的Node

size >> 1，比如说现在是size = 4，size >> 1相当于是size / 2，整除，得到的是2

index < (size >> 1)，如果index < (size / 2)，如果要插入的位置，是在队列的前半部分，那么就会从队列头部开始遍历，找到index那个位置的Node

LinkedList可以作为一个队列来使用

offer() == add()，就是在队列尾部入队，将一个元素插入队列尾部，offerFirst()，offerLast()

poll()，从队列头部出队

peek()，获取队列头部的元素，但是头部的元素不出队

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/17\_HashMap源码剖析（六）：JDK 1.8引入红黑树优化hash冲突/笔记.doc**

如果说出现大量的hash冲突之后，假设某给位置挂的一个链表特别的长，就很恶心了，如果链表长度太长的话，会导致有一些get()操作的时间复杂度就是O(n)，正常来说，table[i]数组索引直接定位的方式的话，O(1)

但是如果链表，大量的key冲突，会导致get()操作，性能急剧下降，导致很多的问题

所以说JDK 1.8以后人家优化了这块东西，会判断，如果链表的长度达到8的时候，那么就会将链表转换为红黑树，如果用红黑树的话，get()操作，即使对一个很大的红黑树进行二叉查找，那么时间复杂度会变成O(logn)，性能会比链表的O(n)得到大幅度的提升

链表转红黑树是怎么搞的

当你遍历一个链表达到第7个节点的时候，binCount是6

当你遍历到第8个节点，此时binCount是7，同时你挂上了第9个节点，然后就会发现binCount >= 7，达到了临界值，也就是说，当你的链表节点的数量超过8的时候，此时就会将链表转换为红黑树

eureka、hystrix那种代码，是烂代码

JDK的源码，极度简洁，高深莫测，一看这个代码就不是菜鸟写出来的

我们来分析这个JDK源码，不是来分析数据结构的源码的，你能看到，JDK里面的红黑树的实现，很复杂的，查找等等逻辑的话，其实很复杂，我不打算带着大家一点点来看红黑树的逻辑了，我们就把红黑树当做一个黑盒子来用就可以了

TreeNode<K,V> hd = null, tl = null;

do {

TreeNode<K,V> p = replacementTreeNode(e, null);

if (tl == null)

hd = p;

else {

p.prev = tl;

tl.next = p;

}

tl = p;

} while ((e = e.next) != null);

是将链表转换为了红黑树，这种源码解释起来极为麻烦

上面那个do while循环执行完了以后，先是将单向链表转换为了TreeNode类型组成的一个双向链表

if ((tab[index] = hd) != null)

hd.treeify(tab);

接下来针对双向链表，将双向链表转换为一颗红黑树，直接记住这个结论就ok了，当链表的长度超过8的时候，链表就先是变成双向链表，然后是变成红黑树

如果有hash冲突，要么是key不一样，结果你的hashCode()方法乱写，导致hash值一样；但是大多数是key不一样，同时hash值也不一样，但是呢，hash寻址过后找到的这个数组中的位置是一样的，出现了hash碰撞

此时先是挂链表，如果链表长度超过了8，就将链表转为红黑树

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/09\_LinkedList源码剖析（五）：删除元素的原理/笔记.doc**

removeLast()

removeFirst() == poll()

remove(int index)

说一下，很多同学说，搞不懂一些数据结构和算法，其实说实话，大学里面学的那些数据结构和算法，不难，你呢拿到一段算法，要搞明白里面的一些原理，很简单，构造一些数据，输入参数

每个参数输入进去，代码执行的过程，跟我这样子，画一幅图出来

算法运行完了以后，你就通过这个图帮助自己理解了那个算法的运行原理，如果算法有一些边界条件，你可以这个构造一些边界条件的值，输入到算法里面，运行，同样还是画图，一边运行算法，一边画图

冒泡、快排、二分查找，理论上来说，一般的算法，你都可以通过画图来理解

红黑树，TreeMap，HashMap底层现在也有红黑树了，都是画图，输入一些参数，通过代码运行，图不断的变动，理解这个数据结构和算法的原理

双向链表，来实现的LinkedList数据结构，你应该看到的是他底层的一个双向队列的数据结构，插入、获取、删除，都可以从队头、队尾来实现，完全可以当做一个队列来用，offer()往队尾插入元素，poll()从队头删除元素

如果你是往他这个里面不断的疯狂的插入数据，队头、队尾、队列中间，哪怕是插入大量的数据，他的优点就是在于他是基于链表来实现的，所以不会出现任何的大量元素的挪动，也不会出现说数组的扩容

在中间插入元素性能没有队头和队尾那么好，他要走一个遍历，链表，遍历到指定的位置，然后完成元素的插入

基于链表的数据结构，很适合做队列，这样的一个东西

缺点，如果是要随机位置获取一个元素，get(int index)这个方法，会导致说需要遍历，如果里面的数据很多的话，遍历的性能还是比较差的，才能获取到一个随机位置的元素，这个就是缺点

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/14\_HashMap源码剖析（三）：优化后的降低冲突概率的hash算法/笔记.doc**

正式来剖析hashmap的工作的源码，从哪个方法开始呢？

map.put(key, value) -> 对key进行hash算法，通过hash获取到对应的数组中的index位置

hash算法是怎么来玩儿的，是不是跟很多初级工程师在面试的时候回答一下说，就是一个简单的key的hash值，key.hashCode()方法返回的一个值吗 ？

这个东西，在hashmap里面是优化过后的hash算法，高性能的

public V put(K key, V value) {

return putVal(hash(key), key, value, false, true);

}

hash(key)，对key进行hash获取一个对应的hash值，key、value传入到putVal()方法里面去，将key-value对儿根据其hash值找到对应的数组位置

hash(key)方法，里面的算法是什么呢，JDK源码里面，涉及到了大量的位运算，我需要给大家来手动计算一下位运算操作符的计算过程和结果

static final int hash(Object key) {

int h;

return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

}

这个方法是计算出来了key的hash值

h = key.hashCode()：这个就是直接获取了key的hash值，通过的是hashCode()方法

1111 1111 1111 1111 1111 1010 0111 1100

h >>> 16，这个是位运算的操作，这个东西是把32位的二进制的数字，所有的bit往右侧右移了16位

1111 1111 1111 1111 1111 1010 0111 1100

-> h >>> 16

0000 0000 0000 0000 1111 1111 1111 1111

-> h ^ (h >>> 16)

1111 1111 1111 1111 1111 1010 0111 1100

0000 0000 0000 0000 1111 1111 1111 1111

1111 1111 1111 1111 0000 0101 1000 0011

他这么做，其实是考虑到，将他的高16位和低16位进行一个异或运算，必须记住这么一个结论，在面试的时候被问到hashmap的话，必须得说出来这个意思

就是因为，后面在用这个hash值定位到数组的index的时候，也有一个位运算

但是的话呢，一般那个后面的位运算，一般都是用低16位在进行运算，所以说如果你不把hash值的高16位和低16位进行运算的话，那么就会导致你后面在通过hash值找到数组index的时候，只有hash值的低16位参与了运算

提前在hash()函数里面，就会把高16位和低16位进行一下异或运算，就可以保证说，在hash值的低16位里面，可以同时保留他的高16位和低16位的特征，大家一定要记住这个结论

相当于是在后面定位到数组index的位运算的时候，哪怕只有低16位参与了运算，其实运算的时候，他的hash值的高16位和低16位的特征都参与到了运算定位到那个数组的index，好好的把我说的话回忆几遍，好好的体会一下

这么做有什么好处呢？为什么要保证同时将高16位和低16位的特征同时纳入运算，考虑到数组index的定位中去呢？因为这样子可以保证降低hash冲突的概率，如果说直接用hash值的低16位去运算定位数组index的话，可能会导致一定的hash冲突

为什么要做这么一个操作呢？hash算法里，为什么是hash值跟hash >>> 16位的结果，异或运算的结果呢？他这么做的话，这里牵扯到很多数学的概念，我觉得作为大家一个java码农来说，你只要记住一个结论

目标是什么呢？通过这样的方式算出来的hash值，可以降低hash冲突的概率

很多key，可能值不同，但是hash值可能是相同的，如果key不同，但是hash值相同，或者是hash值不同，但是到数组的index相同，那么都会出现hash冲突

通过上面的这个操作，计算出来的hash值可以降低hash冲突概率

hash算法，不是你想的那么简单的，是做了一个>>> 16，^，这样可以有效降低hash冲突的概率

0000 0000 0000 0000 0000 0000 0000 1111

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/06\_LinkedList源码剖析（二）：双向链表数据结构/笔记.doc**

你出去面试的时候，ArrayList和LinkedList的区别是什么

如果是比较一般的工程师，90%的人的比重，我见过的人里，一般他们就会说，ArrayList是数组实现的，LinkedList是链表实现的，回答到一个比较基本的程度，因为没看过JDK集合包的源码

水平比较好的工程师，真正底层技术很扎实的，深入的，ArrayList的源码我看过，给你讲一下，数组，add、remove、get、set他的一些基于数组实现的原理是什么，数组扩容，元素移动的原理是什么，优缺点是什么

LinkedList，如果技术真的好的话，不要说LinkedList是基于链表实现的，说LinkedList是基于双向链表实现的，你可以现场画一下他的数据结构，同时的话呢，就是可以把他的一些常见操作的原理图，在现场来画一下，指针，双向链表的数据结构，node是怎么变化的

LinkedList优缺点

我们项目里一般什么时候，我在哪个项目，哪个业务场景下用过ArrayList，LinkedList

在队列头部加一个元素 / 获取一个元素 / 删除一个元素、在队列尾部加一个元素 / 获取一个元素 / 删除一个元素、在队列中间插入一个元素 / 获取一个元素 / 删除一个元素

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/01\_为什么要讲JDK源码剖析系列课程？/笔记.doc**

大家之前已经学习过一些底层的源码和知识了，比如说spring cloud框架源码，bytetcc事务框架源码，redisson以及curator分布式锁源码。接下来我们在分布式那个阶段，需要学习zookeeper的实战和源码剖析，dubbo的实战和源码剖析。

所以说，大家思考一下，你们之前看那些源码的时候，有没有发现说很多地方都看不懂的，比如说一些基础性的代码逻辑，各种内存里的数据结构，list、map、set等等看，还有并发、网络请求、磁盘IO等等这些东西。3616118202

各种各样的系统，技术，也好，底层，最最核心的一块，其实都是集合（在内存里面存放数据）、并发（分布式系统，底层都会开多个线程，并发的处理一些东西，这里会涉及到一些锁，同步，等等）、IO（读写磁盘上的文件里的数据，发起网络IO，通过网络读写数据）、网络（如何在分布式系统中，各个机器建立网络连接，互相发送请求进行通信）

带上的大量的系统机制、核心流程、架构、业务代码、设计模式、面向对象

很多地方，一些基础性的代码细节，我们都没讲解，不是我不讲，是大家其实听不懂，那些基础你们都没有！

所以说我们现在要讲zookeeper和dubbo了，那些源码的难度上升了一个档次，说实话，我之前故意就给大家挑的是比较简单的源码，spring cloud的源码还算是比较简单一些的，包括分布式事务和分布式锁的源码

所以说，现在开始上难的源码了，那肯定得把基础性的技术都讲一下，否则大家是真的看不懂了，讲出来效果也不是太好

对基础技术的掌握，并不是说那么简单的，掌握一些基本的JDK API的使用，ArrayList、HashMap，简单知道怎么用就可以了，其实这些JDK API底层都涉及到一些比较复杂的机制，包括并发，锁底层的原理 -> JVM这块，BIO、NIO、AIO，磁盘文件读写的一些知识，网络，http、tcp、网络协议、rpc，这些东西你是否理解

如果你对这些底层的技术只是浅浅的知道一些的话，是不足以让你有足够的基础去研究一些比较复杂的技术的，比如zk、dubbo、rocketmq、sharding-jdbc，那些技术都需要你的底层的技术功底比较好

这是一方面

另外一方面，你出去面试，特别面一些大公司，只要是国内排名比较靠前的一些互联网公司，哪怕是B轮、C轮、D轮的知名创业公司，面试比较难了，基本上java基础我们必问，而且主要就是这么几块东西，集合、并发、IO、网络、jvm、设计模式，这几块，基本上就够成了所谓的java基础吧

比如集合这块，hashmap源码，是最常见会问的

很多人就是简单会用，那去大厂面试很容易被干倒，再正常不过了，所以还是要好好研究一下这些东西的

后面讲JDK集合源码，分成3个步骤来走，第一个是图解原理，第二个是弄一个模拟的案例来用一用，第三个是分析里面的源码实现

集合（list、map、set），并发（基础知识，推荐大家看一些书，我们来讲解一些比较高级一些的内容，这块东西有可能会让大家自行去学习一些基础的知识，学并发之前，需要去掌握一些jvm的知识，精讲，研究一下并发包底层的一些源码），IO + 网络（不一定会讲源码，核心、重要的基础知识和原理，给大家讲清楚），netty（源码，非常知名的进行分布式系统之间底层网络通信的框架）

这些东西都搞完了以后，我们可以回过头再看一遍spring cloud eureka的服务注册中心的源码，里面大量的集合、并发、IO、网络的东西，我们可以手写一个微服务注册中心的简化的一个版本，JDK底层知识搞一个串讲，搞一个实战开发的项目

（1）内存里面通过一些数据结构保存服务注册表

（2）多线程，并发读写服务注册表，里面就有并发、锁的一些问题

（3）各个服务与服务注册中心之间的通信，网络、IO的通信，这块可以采用netty来写

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/12\_HashMap源码剖析（一）：数组+链表+红黑树的数据结构/笔记.doc**

hashmap其实是我们整个JDK集合包源码剖析的重点

hashmap底层原理或者是底层源码的话，大公司都经常这么问，比如说我，也是比较偏好追问hashmap的底层原理

校招、初级、中级、高级、资深，hashmap都有不同的问法，基础的，可能随便问你几个问题，hash冲突的时候怎么解决，链表，用链表来处理，就算是刚参加完培训的同学，都该知道这个东西

初级的问题里，一般人可能会问你，你说一说hashmap的原理，对key进行hash，找到对应的位置，放在里面，查询的时候，也是对key进行hash，去找那个key-value对

JDK 1.8开始优化了hashmap的数据结构，链表 -> 红黑树，解决hash冲突的问题

hashmap的扩容是怎么处理的，扩容的原理

资深，JDK 1.8以后hashmap的底层做了哪些优化，让你聊聊hashmap底层的源码，源码层面聊聊hashmap的原理

你聊聊迭代java集合的时候，fail-fast机制是什么东西，这个在开发的时候是一个比较常见的异常，ConcurrentModificationException，大概是这样的一个名字，java集合相关的，让你说说是什么？

fail-fast实现的一个源码级别的原理

并发包里面的东西会问的比较多一些，这个东西我后面给大家来讲解，这个集合包源码结束了，就是并发包，把并发包里的东西，好好的搞一下

初步的介绍一下JDK 1.8开始的hashmap的基本的数据结构和原理

map.put(1, “张三”)

map.put(2, “李四”)

对你的key，进行一个hashCode()的一个运算，获取你的key的hash值

常规的一个做法，就是用这个hash值对数组的长度进行取模，根据取模的结果，将key-value对方在数组中的某个一个元素上

map.get(1)，这个东西，同理的，对key获取一个hash值，根据hash值对数组长度取模，就知道这个key对应的key-value对在哪里，return array[5]，就可以直接根据hash值定位到数组里哪个元素，然后就返回了，性能超高

如果说，某两个key，对应的hash值，不幸的是一样的，怎么办呢？

map.put(“手动阀”, “张三”)

map.put(“聊扣扣”, “李四”)

“手动阀”和”聊扣扣”，如果假设他们的hash值是一样的，怎么办呢？但是这是两个不一样的key啊，hash值一样会导致他们放到同一个数组的索引的位置上去，此时如何处理呢？其实在JDK 1.8以前，链表

如果有很多的hash冲突，也就是说多个key的hash值，是一样的；或者也可能是多个key的hash值不一样，但是不同的hash值对一个数组的length取模，获取到的这个数组的index位置，是一样的

map.put(99, “王五”)，假设99的hash值是特殊的一个值，但是这个99的hash值对数组长度取模以后，获取到的index位置也是index = 5，此时也会导致他的这个元素挂在数组上，成一个链表

hash冲突

如果你此时map.get(237)，要获取赵六的那个key-value对，如果出现大量的key冲突之后，对长链表遍历找一个key-value对，性能是O(n)，如果是直接根据array[index]获取到某个元素，性能是O(1)

JDK 1.8以后，优化了一下，如果一个链表的长度超过了8，就会自动将链表转换为红黑树，查找的性能，是O(logn)，这个性能是比O(n)要高的，如果你此时要搜索map.get(381)，如果是链表的话，你必须要遍历5个节点

现在是红黑树，如果大家连红黑树是什么都不知道的话，建议百度一下，红黑树

（1）红黑树是二叉查找树，左小右大，根据这个规则可以快速查找某个值

（2）但是普通的二叉查找树，是有可能出现瘸子的情况，只有一条腿，不平衡了，导致查询性能变成O(n)，线性查询了

（3）红黑树，红色和黑色两种节点，有一大堆的条件限制，尽可能保证树是平衡的，不会出现瘸腿的情况

（4）如果插入节点的时候破坏了红黑树的规则和平衡，会自动重新平衡，变色（红 <-> 黑），旋转，左旋转，右旋转

如果是红黑树的话，找3个节点，就可以找到那个381的值

JDK 1.8，在链表长度为8以后，要链表 -> 红黑树，链表的遍历性能，时间复杂度是O(n)，红黑树是O(logn)，所以如果出现了大量的hash冲突以后，红黑树的性能比链表高得多，几倍到几十倍

JDK 1.8以后，hashmap的数据结构是，数组 + 链表 + 红黑树

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/01~25资料/21\_HashMap源码剖析（十）：get与remove操作的原理分析/笔记.doc**

如果明白了put原理，hash算法、hash寻址、链表+红黑树，基本上get、remove操作看一眼就ok了，一看就明白什么意思

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/18~30资料/18\_手写一个小程序来体验一下volatile关键字的作用/笔记.doc**

很重要的一个环节，我们可以启动多个线程来工作

多个线程，首先如果要共用一个变量，有一个线程专门负责修改这个变量；另外一个线程，专门负责读取这个变量的值

这个时候，大家觉得会不会有什么问题？

在实际的系统运行过程中，可能会产生一个问题，就是说，Thread1修改变量的值，他修改了这个变量的值，结果呢，发现就是说，Thread0，在他修改变量值为true之后，没那么快能感知到flag = true了

Thread1，已经将flag = true设置好了

Thread0，比如说在一段时间范围内，还是读到了flag = false，在一小段时间范围内，可能Thread0会感知不到Thread1对flag的值修改为true，他读到的可能还是flag = false的这么一个旧的值

volatile这个东西，很多人把他看的很高深，其实很简单

他应该是并发编程里面非常常见的一种东西，如果你观察大量的开源项目的话，你会发现人家一般会大量的运用volatile来编程

只要开了多个线程，一定会有一些这种问题，某个线程修改一个变量值，其他线程要立刻感知到这个变量值的变化，但是如果你不用volatile，会有问题

有线程修改了一个变量的值，结果其他的线程感知不到这个变量值的修改

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/18~30资料/22\_图解Java内存模型以及多线程并发问题的发生/笔记.doc**

Java内存模型是跟cpu缓存模型是类似的，基于cpu缓存模型来建立的java内存模型，只不过java内存模型是标准化的，屏蔽掉底层不同的计算机的区别

线程的工作内存和主内存

read（从主存读取），load（将主存读取到的值写入工作内存），use（从工作内存读取数据来计算），assign（将计算好的值重新赋值到工作内存中），store（将工作内存数据写入主存），write（将store过去的变量值赋值给主存中的变量）

在java内存模型下，多线程并发运行的问题回顾

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/18~30资料/23\_并发编程中的三大特性：可见性、原子性、有序性/笔记.doc**

也就是并发编程过程中，可能会产生的三类问题

（1）可见性

之前一直给大家代码演示，画图演示，其实说的就是并发编程中可见性问题

（2）原子性

用这个java内存模型和多线程的场景，以i++的简单操作，给大家来画图分析一下

对于一个i++的操作，只要是多个线程并发运行来执行这行代码，其实的话，他都是不保证原子性的，如果保证原子性的，第一个线程i++，i = 1；第二个线程，i++，i = 2

经常在网上看到，有人就是volatile，第一句话，volatile是轻量级的锁，或者是轻量级同步机制，拍死

锁是什么？volatile是轻量级的锁，误导了无数的工程师，锁那个方向去发展，开始误解，volatile，i++，还有一些基本的操作，可以保证线程的并发访问的安全性

锁，synchonized、lock，某个线程一旦锁了这个资源以后，其他的线程就不能访问了，有人锁了就不能访问了

轻量级的锁，有一个线程在修改的时候，别人就不能访问了吗？

volatile，他的解释，就是保证线程之间可见性的一个关键字

（3）有序性

对于代码，同时还有一个问题是指令重排序，编译器和指令器，有的时候为了提高代码执行效率，会将指令重排序，就是说比如下面的代码

flag = false;

//线程1:

prepare(); // 准备资源

flag = true;

//线程2:

while(!flag){

Thread.sleep(1000);

}

execute(); // 基于准备好的资源执行操作

重排序之后，让flag = true先执行了，会导致线程2直接跳过while等待，执行某段代码，结果prepare()方法还没执行，资源还没准备好呢，此时就会导致代码逻辑出现异常。

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/18~30资料/24\_现场手绘图讲解volatile是如何保证可见性的？/笔记.doc**

直接基于java内存模型那张图来讲

如何能够说，加了volatile以后就可以保证多线程的可见性呢？

有一点，只要flag变成了1，然后线程不是要将flag = 1写回工作内存吗？assign操作，此时如果这个flag变量是加了volatile关键字的话，那么此时会这样子，就是说一定会强制保证说assign之后，就立马执行store + write，刷回到主内存里去

保证只要工作内存一旦变为flag = 1，主内存立马变成flag = 1

此外，如果这个变量是加了volatile关键字的话，此时他就会让其他线程的工作内存中的这个flag变量的缓存，会过期

线程2如果再从工作内存里读取flag变量的值，发现他已经过期了，此时就会重新从主内存里来加载这个flag = 1的值

通过volatile关键字，可以实现的一个效果就是说，有一个线程修改了值，其他线程可以立马感知到这个值

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/18~30资料/20\_多线程并发运行时可能引发的数据不一致问题/笔记.doc**

接着上一讲cpu缓存模型，cpu在他和主内存之间加了一层缓存

主内存的数据会被加载到cpu本地缓存里去，cpu后面会读写自己的缓存

特别多线程并发运行的时候，会引发什么问题呢？

cpu缓存模型，其实默认情况下是有问题的，特别是多线程并发运行的时候，导致说各个cpu的本地缓存，跟主内存，没有同步，一个数据，在各个地方，可能都不一样，就会导致数据的不一致

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/18~30资料/25\_几乎没人解释的清楚：volatile为什么无法保证原子性？/笔记.doc**

volatile是不保证原子性，为什么保证不了原子性呢？

volatile i = 0

两个线程，i++，此时为什么不能保证原子性？数据还是可能会出错呢？

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/18~30资料/19\_图解主内存以及cpu的多级缓存模型的实现原理/笔记.doc**

演示了一下开了多线程以后，如果多个线程共用一个共享变量，有人写，有人读，那么其实是有问题的，有可能会导致有的线程没法及时读到比人修改的变量的值，一直读到的就是老的值

volatile关键字，就是用来解决这个问题的，尽量让你修改了一个变量之后，其他的线程可以立即看到这个变量的最新的值

cpu缓存模型 -> java内存模型 -> 原子性、可见性、有序性 -> volatile的作用 -> volatile的底层原理 -> volatile实战

上网搜一些volatile的博客，那些人写的天花烂坠，描述的一点都不清楚

操作系统层面的东西

遇到一个问题，现代的计算机技术，内存的读写速度没什么突破，cpu如果要频繁的读写主内存的话，会导致性能较差，计算性能就会低，这样的不适应现代计算机技术的发展

现代的计算机，一般来说，都不是这么玩儿的

换了一种玩法，给cpu加了几层缓存

主要就是说，cpu可以直接操作自己对应的告诉缓存，不需要直接频繁的跟主内存通信，这个是现代计算机技术的一个进步，这样可以保证cpu的计算的效率非常的高

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/18~30资料/26\_基于happens-before原则来看volatile如何保证有序性/笔记.doc**

java中有一个happens-before原则：

编译器、指令器可能对代码重排序，乱排，要守一定的规则，happens-before原则，只要符合happens-before的原则，那么就不能胡乱重排，如果不符合这些规则的话，那就可以自己排序

程序次序规则：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作

锁定规则：一个unLock操作先行发生于后面对同一个锁额lock操作

volatile变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作

volatile变量写，再是读，必须保证是先写，再读

传递规则：如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C

线程启动规则：Thread对象的start()方法先行发生于此线程的每个一个动作

线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生

线程终结规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行

对象终结规则：一个对象的初始化完成先行发生于他的finalize()方法的开始

上面这8条原则的意思很显而易见，就是程序中的代码如果满足这个条件，就一定会按照这个规则来保证指令的顺序。

但是如果没满足上面的规则，那么就可能会出现指令重排，就这个意思。这8条原则是避免说出现乱七八糟扰乱秩序的指令重排，要求是这几个重要的场景下，比如是按照顺序来，但是8条规则之外，可以随意重排指令。

//线程1:

prepare(); // 准备资源

volatile flag = true;

//线程2:

while(!flag){

sleep()

}

execute(); // 基于准备好的资源执行操作

比如这个例子，如果用volatile来修饰flag变量，一定可以让prepare()指令在flag = true之前先执行，这就禁止了指令重排。因为volatile要求的是，volatile前面的代码一定不能指令重排到volatile变量操作后面，volatile后面的代码也不能指令重排到volatile前面。

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/18~30资料/29\_案例实战：基于volatile优化微服务的优雅关闭机制/笔记.doc**

volatile都学过了以后，在项目的开发中，一般同学如果做业务类系统，crud的系统的话，其实是一般不会用多线程编程，你的代码里不会启动多个线程做复杂的工作和操作，那么就必然不会用到voaltile

多线程访问一个共享变量，可见性

微服务注册中心的中间件，项目里，我们可以有大量的真实的场景可以来实战

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/18~30资料/27\_volatile的底层实现原理：lock指令以及内存屏障/笔记.doc**

volatile底层原理，如何实现保证可见性的呢？如何实现保证有序性的呢？

（1）lock指令

二期有一个学员出去面试，面一个大的互联网公司，volatile底层，java内存模型、问题、voaltile是如何保证可见性的，缓存一致性协议，但是后面有人家追问了一句，说volatile了以后，具体是发送了什么指令，去实现了什么效果

对volatile修饰的变量，执行写操作的话，JVM会发送一条lock前缀指令给CPU，CPU在计算完之后会立即将这个值写回主内存，同时因为有MESI缓存一致性协议，所以各个CPU都会对总线进行嗅探，自己本地缓存中的数据是否被别人修改

如果发现别人修改了某个缓存的数据，那么CPU就会将自己本地缓存的数据过期掉，然后这个CPU上执行的线程在读取那个变量的时候，就会从主内存重新加载最新的数据了

lock前缀指令 + MESI缓存一致性协议

（2）内存屏障：禁止重排序

volatille是如何保证有序性的？加了volatile的变量，可以保证前后的一些代码不会被指令重排，这个是如何做到的呢？

去年的时候，一期学员，当时说在一个XX群里，看到一个兄弟说，volatile很简单，他其实底层就是插入了XX内存屏障，XX内存屏障，就可以保证指令不会重排，当时那个学员说那个问题看着好高深，好难

你只要把我的volatile这块讲解，听懂了，画图，其实你就全理解了

指令重排是怎么回事，volatile就不会指令重排

Load1：

int localVar = this.variable

Load2：

int localVar = this.variable2

LoadLoad屏障：Load1；LoadLoad；Load2，确保Load1数据的装载先于Load2后所有装载指令，他的意思，Load1对应的代码和Load2对应的代码，是不能指令重排的

Store1：

this.variable = 1

StoreStore屏障

Store2：

this.variable2 = 2

StoreStore屏障：Store1；StoreStore；Store2，确保Store1的数据一定刷回主存，对其他cpu可见，先于Store2以及后续指令

LoadStore屏障：Load1；LoadStore；Store2，确保Load1指令的数据装载，先于Store2以及后续指令

StoreLoad屏障：Store1；StoreLoad；Load2，确保Store1指令的数据一定刷回主存，对其他cpu可见，先于Load2以及后续指令的数据装载

volatile的作用是什么呢？

volatile variable = 1

this.variable = 2 => store操作

int localVariable = this.variable => load操作

对于volatile修改变量的读写操作，都会加入内存屏障

每个volatile写操作前面，加StoreStore屏障，禁止上面的普通写和他重排；每个volatile写操作后面，加StoreLoad屏障，禁止跟下面的volatile读/写重排

每个volatile读操作后面，加LoadLoad屏障，禁止下面的普通读和voaltile读重排；每个volatile读操作后面，加LoadStore屏障，禁止下面的普通写和volatile读重排

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/18~30资料/30\_案例实战：基于volatile优化微服务存活状态检查机制/笔记.doc**

不要看volatile的项目中的实战，如此的短暂和简单，真的不简单，看过一些视频课程，或者是一些并发类的书，你之前只要是没做过中间件研发的同学，你一定对并发的实战运用了解的很浅

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/18~30资料/21\_总线加锁机制和MESI缓存一致性协议的工作原理/笔记.doc**

最早的时候，其实人家用的是一个机制，叫做总线加锁机制

已经没有人来用了，他大概的意思是说，某个cpu如果要读一个数据，会通过一个总线，对这个数据加一个锁，其他的cpu就没法去读和写这个数据了，只有当这个cpu修改完了以后，其他cpu可以读到最新的数据

大概是这个意思

这个总线加锁机制，效率太差了，一旦说多个线程出现了对某个共享变量的访问之后，就会导致说，可能串行化的问题，多个cpu多线程并发运行的时候，效率很差

MESI协议，缓存一致性协议

基本上就是可以保证说，我们上一讲讲的那个多线程并发的问题没有了

cpu嗅探机制

MESI协议，就是这样的一整套的机制来保证说，cpu缓存模型下，不会出现说我们演示的那个多线程并发读写变量，没有办法及时的感知到

特别是volatile关键字，底层涉及到的一些指令，之前确实是有一个同学出去面试，京东，人家就问他并发这块的内容，java内存模型，volatile的原理，他大概说出来了这套东西，但是人家问的很深

到底底层是如何实现这套MESI的机制，通过哪些指令，这个指令干了什么事情，才能保证说，我刚才说的那种效果，修改本地缓存，立马刷主存，其他cpu本地缓存立马工期，重新从主存加载

:lock前缀指令 -> 内存屏障

# **File Path: /Users/xiaotingting/Downloads/学习资料/08\_JDK源码剖析系列unzip/18~30资料/28\_double check单例模式的实现缺陷以及volatile的优化/笔记.doc**

volatile他的实际的用途和场景

double check的单例模式