目录

[聊聊并发（一）——深入分析Volatile的实现原理 3](#_Toc389065635)

[引言 3](#_Toc389065636)

[术语定义 3](#_Toc389065637)

[Volatile的官方定义 4](#_Toc389065638)

[为什么要使用Volatile 4](#_Toc389065639)

[Volatile的实现原理 4](#_Toc389065640)

[Volatile的使用优化 5](#_Toc389065641)

[参考资料 6](#_Toc389065642)

[关于作者 6](#_Toc389065643)

[聊聊并发（二）——Java SE1.6中的Synchronized 6](#_Toc389065644)

[1 引言 6](#_Toc389065645)

[2 术语定义 6](#_Toc389065646)

[3 同步的基础 7](#_Toc389065647)

[4 同步的原理 7](#_Toc389065648)

[4.1 Java对象头 7](#_Toc389065649)

[4.2 锁的升级 8](#_Toc389065650)

[4.3 偏向锁 8](#_Toc389065651)

[4.4 轻量级锁 9](#_Toc389065652)

[5 锁的优缺点对比 10](#_Toc389065653)

[6 参考源码 11](#_Toc389065654)

[7 参考资料 11](#_Toc389065655)

[作者简介 11](#_Toc389065656)

[聊聊并发（三）——JAVA线程池的分析和使用 11](#_Toc389065657)

[1. 引言 11](#_Toc389065658)

[2. 线程池的使用 12](#_Toc389065659)

[3. 线程池的分析 14](#_Toc389065660)

[4. 合理的配置线程池 15](#_Toc389065661)

[5. 线程池的监控 16](#_Toc389065662)

[6. 参考资料 17](#_Toc389065663)

[作者介绍 17](#_Toc389065664)

[聊聊并发（四）——深入分析ConcurrentHashMap 17](#_Toc389065665)

[术语定义 17](#_Toc389065666)

[线程不安全的HashMap 17](#_Toc389065667)

[效率低下的HashTable容器 18](#_Toc389065668)

[锁分段技术 18](#_Toc389065669)

[ConcurrentHashMap的结构 18](#_Toc389065670)

[ConcurrentHashMap的初始化 20](#_Toc389065671)

[定位Segment 21](#_Toc389065672)

[ConcurrentHashMap的get操作 22](#_Toc389065673)

[ConcurrentHashMap的Put操作 23](#_Toc389065674)

[ConcurrentHashMap的size操作 24](#_Toc389065675)

[参考资料 24](#_Toc389065676)

[作者介绍 24](#_Toc389065677)

[聊聊并发（五）——原子操作的实现原理 24](#_Toc389065678)

[1. 引言 24](#_Toc389065679)

[2. 术语定义 25](#_Toc389065680)

[3. 处理器如何实现原子操作 25](#_Toc389065681)

[3.1 处理器自动保证基本内存操作的原子性 25](#_Toc389065682)

[3.3 使用缓存锁保证原子性 26](#_Toc389065683)

[4. JAVA如何实现原子操作 27](#_Toc389065684)

[4.1 使用循环CAS实现原子操作 27](#_Toc389065685)

[4.2 使用锁机制实现原子操作 29](#_Toc389065686)

[5. 参考资料 29](#_Toc389065687)

[作者介绍 30](#_Toc389065688)

[聊聊并发（六）——ConcurrentLinkedQueue的实现原理分析 30](#_Toc389065689)

[1. 引言 30](#_Toc389065690)

[2. ConcurrentLinkedQueue的介绍 30](#_Toc389065691)

[3. ConcurrentLinkedQueue的结构 30](#_Toc389065692)

[4. 入队列 31](#_Toc389065693)

[5. 出队列 34](#_Toc389065694)

[6. 参考资料 36](#_Toc389065695)

[作者介绍 36](#_Toc389065696)

[**聊聊并发（七）——Java中的阻塞队列** 36](#_Toc389065697)

[1. 什么是阻塞队列？ 37](#_Toc389065698)

[2. Java里的阻塞队列 37](#_Toc389065699)

[3. 阻塞队列的实现原理 41](#_Toc389065700)

[4. 参考资料 46](#_Toc389065701)

[5. 作者介绍 46](#_Toc389065702)

[**聊聊并发（八）——Fork/Join框架介绍** 46](#_Toc389065703)

[1. 什么是Fork/Join框架 46](#_Toc389065704)

[2. 工作窃取算法 47](#_Toc389065705)

[3. Fork/Join框架的介绍 48](#_Toc389065706)

[4. 使用Fork/Join框架 49](#_Toc389065707)

[5. Fork/Join框架的异常处理 51](#_Toc389065708)

[6. Fork/Join框架的实现原理 52](#_Toc389065709)

[7. 参考资料 54](#_Toc389065710)

[8. 作者介绍 54](#_Toc389065711)

[**聊聊并发-Java中的Copy-On-Write容器** 54](#_Toc389065712)

[**聊聊并发——生产者消费者模式** 58](#_Toc389065713)

[为什么要使用生产者和消费者模式 59](#_Toc389065714)

[什么是生产者消费者模式 59](#_Toc389065715)

[生产者消费者模式实战 59](#_Toc389065716)

[多生产者和多消费者场景 63](#_Toc389065717)

[小结 67](#_Toc389065718)

聊聊并发（一）——深入分析Volatile的实现原理

<http://www.infoq.com/cn/articles/ftf-java-volatile>

## 引言

在多线程并发编程中synchronized和Volatile都扮演着重要的角色，Volatile是**轻量级的synchronized**，它在多处理器开发中保证了共享变量的“可见性”。可见性的意思是当一个线程修改一个共享变量时，另外一个线程能读到这个修改的值。

它在某些情况下比synchronized的开销更小，本文将深入分析在硬件层面上Inter处理器是如何实现Volatile的，通过深入分析能帮助我们正确的使用Volatile变量。

## 术语定义

|  |
| --- |
|  |

|  |  |  |
| --- | --- | --- |
| 术语 | 英文单词 | 描述 |
| 共享变量 |  | 在多个线程之间能够被共享的变量被称为共享变量。共享变量包括所有的实例变量，静态变量和数组元素。他们都被存放在堆内存中，Volatile只作用于共享变量。 |
| 内存屏障 | Memory Barriers | 是一组处理器指令，用于实现对内存操作的顺序限制。 |
| 缓冲行 | Cache line | 缓存中可以分配的最小存储单位。处理器填写缓存线时会加载整个缓存线，需要使用多个主内存读周期。 |
| 原子操作 | Atomic operations | 不可中断的一个或一系列操作。 |
| 缓存行填充 | cache line fill | 当处理器识别到从内存中读取操作数是可缓存的，处理器读取整个缓存行到适当的缓存（L1，L2，L3的或所有） |
| 缓存命中 | cache hit | 如果进行高速缓存行填充操作的内存位置仍然是下次处理器访问的地址时，处理器从缓存中读取操作数，而不是从内存。 |
| 写命中 | write hit | 当处理器将操作数写回到一个内存缓存的区域时，它首先会检查这个缓存的内存地址是否在缓存行中，如果存在一个有效的缓存行，则处理器将这个操作数写回到缓存，而不是写回到内存，这个操作被称为写命中。 |
| 写缺失 | write misses the cache | 一个有效的缓存行被写入到不存在的内存区域。 |

## Volatile的官方定义

Java语言规范第三版中对volatile的定义如下： java编程语言允许线程访问共享变量，为了确保共享变量能被准确和一致的更新，线程应该确保通过排他锁单独获得这个变量。Java语言提供了volatile，在某些情况下比锁更加方便。如果一个字段被声明成volatile，java线程内存模型确保所有线程看到这个变量的值是一致的。

## 为什么要使用Volatile

Volatile变量修饰符如果使用**恰当**的话，它比synchronized的**使用和执行成本会更低**，因为它不会引起线程上下文的切换和调度。

## Volatile的实现原理

那么Volatile是如何来保证可见性的呢？在x86处理器下通过工具获取JIT编译器生成的汇编指令来看看对Volatile进行写操作CPU会做什么事情。

|  |  |
| --- | --- |
| Java代码： | instance = new Singleton();//instance是volatile变量 |
| 汇编代码： | 0x01a3de1d: movb $0x0,0x1104800(%esi);  0x01a3de24: **lock** addl $0x0,(%esp); |

有volatile变量修饰的共享变量进行写操作的时候会多第二行汇编代码，通过查IA-32架构软件开发者手册可知，lock前缀的指令在多核处理器下会引发了两件事情。

* 将当前处理器缓存行的数据会写回到系统内存。
* 这个写回内存的操作会引起在其他CPU里缓存了该内存地址的数据无效。

处理器为了提高处理速度，不直接和内存进行通讯，而是先将系统内存的数据读到内部缓存（L1,L2或其他）后再进行操作，但操作完之后不知道何时会写到内存，如果对声明了Volatile变量进行写操作，JVM就会向处理器发送一条Lock前缀的指令，将这个变量所在缓存行的数据写回到系统内存。但是就算写回到内存，如果其他处理器缓存的值还是旧的，再执行计算操作就会有问题，所以在多处理器下，为了保证各个处理器的缓存是一致的，就会实现缓存一致性协议，每个处理器通过嗅探在总线上传播的数据来检查自己缓存的值是不是过期了，当处理器发现自己缓存行对应的内存地址被修改，就会将当前处理器的缓存行设置成无效状态，当处理器要对这个数据进行修改操作的时候，会强制重新从系统内存里把数据读到处理器缓存里。

这两件事情在IA-32软件开发者架构手册的第三册的多处理器管理章节（第八章）中有详细阐述。

**Lock前缀指令会引起处理器缓存回写到内存**。Lock前缀指令导致在执行指令期间，声言处理器的 LOCK# 信号。在多处理器环境中，LOCK# 信号确保在声言该信号期间，处理器可以独占使用任何共享内存。（因为它会锁住总线，导致其他CPU不能访问总线，不能访问总线就意味着不能访问系统内存），但是在最近的处理器里，LOCK＃信号一般不锁总线，而是锁缓存，毕竟锁总线开销比较大。在8.1.4章节有详细说明锁定操作对处理器缓存的影响，对于Intel486和Pentium处理器，在锁操作时，总是在总线上声言LOCK#信号。但在P6和最近的处理器中，如果访问的内存区域已经缓存在处理器内部，则不会声言LOCK#信号。相反地，它会锁定这块内存区域的缓存并回写到内存，并使用缓存一致性机制来确保修改的原子性，此操作被称为“缓存锁定”，**缓存一致性机制会阻止同时修改被两个以上处理器缓存的内存区域数据**。

**一个处理器的缓存回写到内存会导致其他处理器的缓存无效**。IA-32处理器和Intel 64处理器使用MESI（修改，独占，共享，无效）控制协议去维护内部缓存和其他处理器缓存的一致性。在多核处理器系统中进行操作的时候，IA-32 和Intel 64处理器能嗅探其他处理器访问系统内存和它们的内部缓存。它们使用嗅探技术保证它的内部缓存，系统内存和其他处理器的缓存的数据在总线上保持一致。例如在Pentium和P6 family处理器中，如果通过嗅探一个处理器来检测其他处理器打算写内存地址，而这个地址当前处理共享状态，那么正在嗅探的处理器将无效它的缓存行，在下次访问相同内存地址时，强制执行缓存行填充。

## Volatile的使用优化

著名的Java并发编程大师Doug lea在JDK7的并发包里新增一个队列集合类LinkedTransferQueue，他在使用Volatile变量时，用一种追加字节的方式来优化队列出队和入队的性能。

追加字节能优化性能？这种方式看起来很神奇，但如果深入理解处理器架构就能理解其中的奥秘。让我们先来看看LinkedTransferQueue这个类，它使用一个内部类类型来定义队列的头队列（Head）和尾节点（tail），而这个内部类PaddedAtomicReference相对于父类AtomicReference只做了一件事情，就将共享变量追加到64字节。我们可以来计算下，一个对象的引用占4个字节，它追加了15个变量共占60个字节，再加上父类的Value变量，一共64个字节。

/\*\* head of the queue \*/

private transient final PaddedAtomicReference < QNode > head;

/\*\* tail of the queue \*/

private transient final PaddedAtomicReference < QNode > tail;

static final class PaddedAtomicReference < T > extends AtomicReference < T > {

// enough padding for 64bytes with 4byte refs

Object p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, pa, pb, pc, pd, pe;

PaddedAtomicReference(T r) {

super(r);

}

}

public class AtomicReference < V > implements java.io.Serializable {

private volatile V value;

//省略其他代码 ｝

**为什么追加64字节能够提高并发编程的效率呢**？ 因为对于英特尔酷睿i7，酷睿， Atom和NetBurst， Core Solo和Pentium M处理器的L1，L2或L3缓存的高速缓存行是64个字节宽，不支持部分填充缓存行，这意味着如果队列的头节点和尾节点都不足64字节的话，处理器会将它们都读到同一个高速缓存行中，在多处理器下每个处理器都会缓存同样的头尾节点，当一个处理器试图修改头接点时会将整个缓存行锁定，那么在缓存一致性机制的作用下，会导致其他处理器不能访问自己高速缓存中的尾节点，而队列的入队和出队操作是需要不停修改头接点和尾节点，所以在多处理器的情况下将会严重影响到队列的入队和出队效率。Doug lea使用追加到64字节的方式来填满高速缓冲区的缓存行，避免头接点和尾节点加载到同一个缓存行，使得头尾节点在修改时不会互相锁定。

那么是不是在使用Volatile变量时都应该追加到64字节呢？不是的。在两种场景下不应该使用这种方式。第一：**缓存行非64字节宽的处理器**，如P6系列和奔腾处理器，它们的L1和L2高速缓存行是32个字节宽。第二：**共享变量不会被频繁的写**。因为使用追加字节的方式需要处理器读取更多的字节到高速缓冲区，这本身就会带来一定的性能消耗，共享变量如果不被频繁写的话，锁的几率也非常小，就没必要通过追加字节的方式来避免相互锁定。

## 参考资料

* [JVM执行篇：使用HSDIS插件分析JVM代码执行细节](http://www.infoq.com/cn/articles/zzm-java-hsdis-jvm)
* [内存屏障和并发](http://www.infoq.com/cn/articles/memory_barriers_jvm_concurrency)
* [Intel 64和IA-32架构软件开发人员手册](http://www.intel.com/products/processor/manuals/)

## 关于作者

方腾飞，阿里巴巴资深软件开发工程师，致力于高性能网络编程，目前在公司从事询盘管理和长连接服务器OpenComet的开发工作。博客地址：[http://ifeve.com](http://ifeve.com/)

# 聊聊并发（二）——Java SE1.6中的Synchronized

<http://www.infoq.com/cn/articles/java-se-16-synchronized>

## 1 引言

在多线程并发编程中Synchronized一直是元老级角色，很多人都会称呼它为重量级锁，但是随着Java SE1.6对Synchronized进行了各种优化之后，有些情况下它并不那么重了，本文详细介绍了Java SE1.6中为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁，以及锁的存储结构和升级过程。

## 2 术语定义

|  |  |  |
| --- | --- | --- |
| 术语 | 英文 | 说明 |
| CAS | Compare and Swap | 比较并设置。用于在硬件层面上提供原子性操作。在Intel 处理器中，比较并交换通过指令cmpxchg实现。比较是否和给定的数值一致，如果一致则修改，不一致则不修改。 |

## 3 同步的基础

Java中的每一个对象都可以作为锁。

* 对于同步方法，锁是当前实例对象。
* 对于静态同步方法，锁是当前对象的Class对象。
* 对于同步方法块，锁是Synchonized括号里配置的对象。

当一个线程试图访问同步代码块时，它首先必须得到锁，退出或抛出异常时必须释放锁。那么锁存在哪里呢？锁里面会存储什么信息呢？

## 4 同步的原理

JVM规范规定JVM基于进入和退出Monitor对象来实现方法同步和代码块同步，但两者的实现细节不一样。代码块同步是使用monitorenter和monitorexit指令实现，而方法同步是使用另外一种方式实现的，细节在JVM规范里并没有详细说明，但是方法的同步同样可以使用这两个指令来实现。monitorenter指令是在编译后插入到同步代码块的开始位置，而monitorexit是插入到方法结束处和异常处， JVM要保证每个monitorenter必须有对应的monitorexit与之配对。任何对象都有一个 monitor 与之关联，当且一个monitor 被持有后，它将处于锁定状态。线程执行到 monitorenter 指令时，将会尝试获取对象所对应的 monitor 的所有权，即尝试获得对象的锁。

### 4.1 Java对象头

锁存在Java对象头里。如果对象是数组类型，则虚拟机用3个Word（字宽）存储对象头，如果对象是非数组类型，则用2字宽存储对象头。在32位虚拟机中，一字宽等于四字节，即32bit。

|  |  |  |
| --- | --- | --- |
| 长度 | 内容 | 说明 |
| 32/64bit | Mark Word | 存储对象的hashCode或锁信息等。 |
| 32/64bit | Class Metadata Address | 存储到对象类型数据的指针 |
| 32/64bit | Array length | 数组的长度（如果当前对象是数组） |

Java对象头里的Mark Word里默认存储对象的HashCode，分代年龄和锁标记位。32位JVM的Mark Word的默认存储结构如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 25 bit | 4bit | 1bit  是否是偏向锁 | 2bit  锁标志位 |
| 无锁状态 | 对象的hashCode | 对象分代年龄 | 0 | 01 |

在运行期间Mark Word里存储的数据会随着锁标志位的变化而变化。Mark Word可能变化为存储以下4种数据：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **锁状态** | 25 bit | | | 4bit | 1bit | 2bit |
| 23bit | | 2bit | 是否是偏向锁 | 锁标志位 |
| 轻量级锁 | 指向栈中锁记录的指针 | | | | | 00 |
| 重量级锁 | 指向互斥量（重量级锁）的指针 | | | | | 10 |
| GC标记 | 空 | | | | | 11 |
| 偏向锁 | 线程ID | Epoch | | 对象分代年龄 | 1 | 01 |

在64位虚拟机下，Mark Word是64bit大小的，其存储结构如下：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **锁状态** | 25bit | 31bit | 1bit | 4bit | 1bit | | 2bit |
|  |  | cms\_free | 分代年龄 | 偏向锁 | | 锁标志位 |
| 无锁 | unused | hashCode |  |  | 0 | 01 | |
| 偏向锁 | ThreadID(54bit) Epoch(2bit) | |  |  | 1 | 01 | |

### 4.2 锁的升级

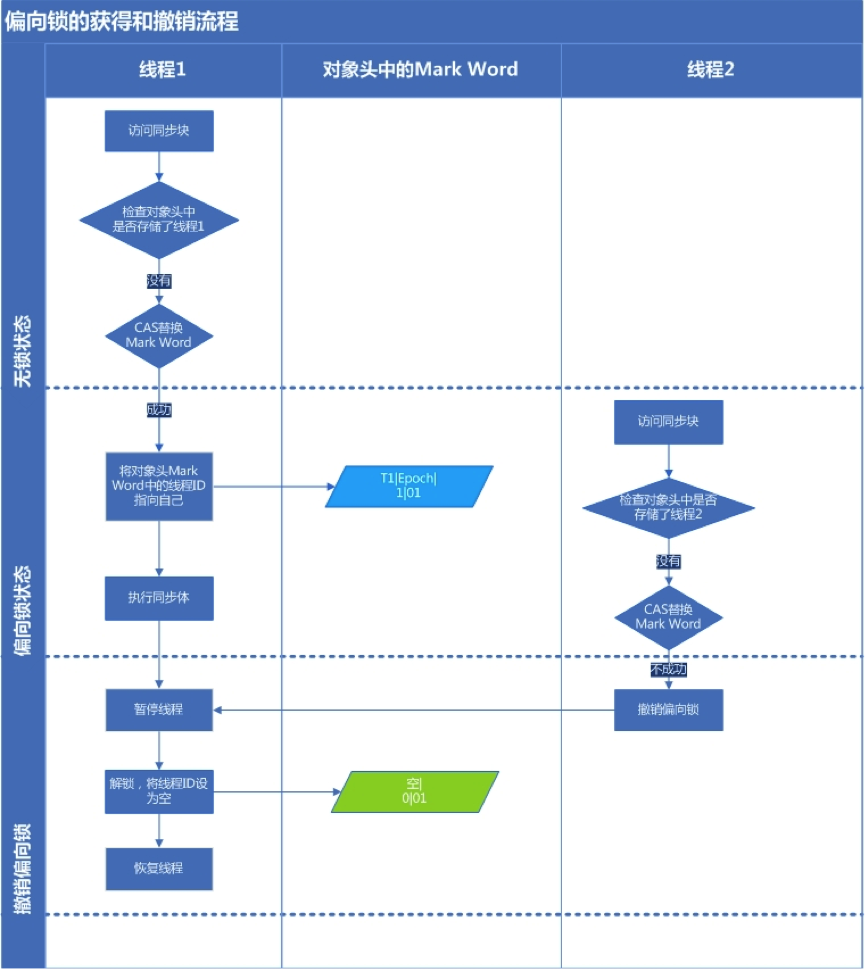
Java SE1.6为了减少获得锁和释放锁所带来的性能消耗，引入了“偏向锁”和“轻量级锁”，所以在Java SE1.6里锁一共有四种状态，无锁状态，偏向锁状态，轻量级锁状态和重量级锁状态，它会随着竞争情况逐渐升级。锁可以升级但不能降级，意味着偏向锁升级成轻量级锁后不能降级成偏向锁。这种锁升级却不能降级的策略，目的是为了提高获得锁和释放锁的效率，下文会详细分析。



### 4.3 偏向锁

Hotspot的作者经过以往的研究发现大多数情况下锁不仅不存在多线程竞争，而且总是由同一线程多次获得，为了让线程获得锁的代价更低而引入了偏向锁。当一个线程访问同步块并获取锁时，会在对象头和栈帧中的锁记录里存储锁偏向的线程ID，以后该线程在进入和退出同步块时不需要花费CAS操作来加锁和解锁，而只需简单的测试一下对象头的Mark Word里是否存储着指向当前线程的偏向锁，如果测试成功，表示线程已经获得了锁，如果测试失败，则需要再测试下Mark Word中偏向锁的标识是否设置成1（表示当前是偏向锁），如果没有设置，则使用CAS竞争锁，如果设置了，则尝试使用CAS将对象头的偏向锁指向当前线程。

偏向锁的撤销：偏向锁使用了一种等到竞争出现才释放锁的机制，所以当其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁。偏向锁的撤销，需要等待全局安全点（在这个时间点上没有字节码正在执行），它会首先暂停拥有偏向锁的线程，然后检查持有偏向锁的线程是否活着，如果线程不处于活动状态，则将对象头设置成无锁状态，如果线程仍然活着，拥有偏向锁的栈会被执行，遍历偏向对象的锁记录，栈中的锁记录和对象头的Mark Word要么重新偏向于其他线程，要么恢复到无锁或者标记对象不适合作为偏向锁，最后唤醒暂停的线程。下图中的线程1演示了偏向锁初始化的流程，线程2演示了偏向锁撤销的流程。

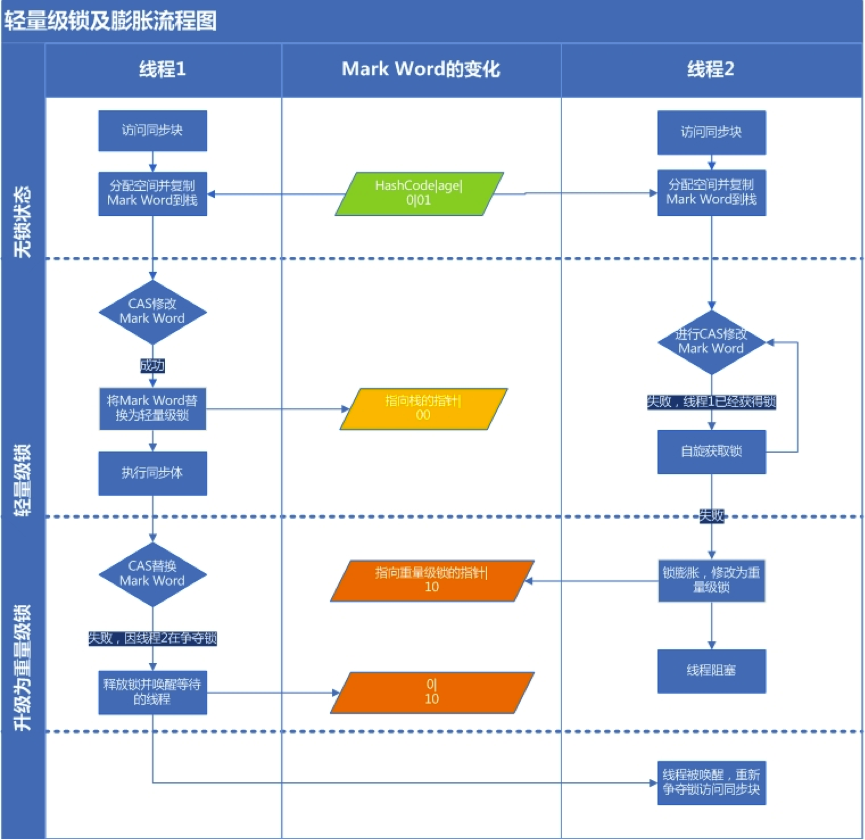


关闭偏向锁：偏向锁在Java 6和Java 7里是默认启用的，但是它在应用程序启动几秒钟之后才激活，如有必要可以使用JVM参数来关闭延迟-XX：BiasedLockingStartupDelay = 0。如果你确定自己应用程序里所有的锁通常情况下处于竞争状态，可以通过JVM参数关闭偏向锁-XX:-UseBiasedLocking=false，那么默认会进入轻量级锁状态。

### 4.4 轻量级锁

轻量级锁加锁：线程在执行同步块之前，JVM会先在当前线程的栈桢中创建用于存储锁记录的空间，并将对象头中的Mark Word复制到锁记录中，官方称为Displaced Mark Word。然后线程尝试使用CAS将对象头中的Mark Word替换为指向锁记录的指针。如果成功，当前线程获得锁，如果失败，表示其他线程竞争锁，当前线程便尝试使用自旋来获取锁。

轻量级锁解锁：轻量级解锁时，会使用原子的CAS操作来将Displaced Mark Word替换回到对象头，如果成功，则表示没有竞争发生。如果失败，表示当前锁存在竞争，锁就会膨胀成重量级锁。下图是两个线程同时争夺锁，导致锁膨胀的流程图。



因为自旋会消耗CPU，为了避免无用的自旋（比如获得锁的线程被阻塞住了），一旦锁升级成重量级锁，就不会再恢复到轻量级锁状态。当锁处于这个状态下，其他线程试图获取锁时，都会被阻塞住，当持有锁的线程释放锁之后会唤醒这些线程，被唤醒的线程就会进行新一轮的夺锁之争。

## 5 锁的优缺点对比

|  |  |  |  |
| --- | --- | --- | --- |
| 锁 | 优点 | 缺点 | 适用场景 |
| 偏向锁 | 加锁和解锁不需要额外的消耗，和执行非同步方法比仅存在纳秒级的差距。 | 如果线程间存在锁竞争，会带来额外的锁撤销的消耗。 | 适用于只有一个线程访问同步块场景。 |
| 轻量级锁 | 竞争的线程不会阻塞，提高了程序的响应速度。 | 如果始终得不到锁竞争的线程使用自旋会消耗CPU。 | 追求响应时间。  同步块执行速度非常快。 |
| 重量级锁 | 线程竞争不使用自旋，不会消耗CPU。 | 线程阻塞，响应时间缓慢。 | 追求吞吐量。  同步块执行速度较长。 |

## 6 参考源码

本文一些内容参考了[HotSpot](http://hg.openjdk.java.net/hsx/hotspot-main/hotspot/file/61b82be3b1ff/)源码 。对象头源码markOop.hpp。偏向锁源码biasedLocking.cpp。以及其他源码ObjectMonitor.cpp和BasicLock.cpp。

## 7 参考资料

* [偏向锁](http://www.oracle.com/technetwork/java/javase/tech/biasedlocking-oopsla2006-preso-150106.pdf)
* [java-overview-and-java-se6](http://pdffinder.net/Java-Overview-and-Java-SE-6-What's-New.html) Synchronization Optimization章节
* Dave Dice [“Synchronization in Java SE 6”](http://home.comcast.net/~pjbishop/Dave/MustangSync.pdf)
* [Java SE 6 Performance White Paper](http://java.sun.com/performance/reference/whitepapers/6_performance.html#2.1.3) 2.1章节
* [JVM规范（Java SE 7）](http://docs.oracle.com/javase/specs/jvms/se7/html/index.html)
* [Java语言规范（JAVA SE7）](http://docs.oracle.com/javase/specs/jls/se7/html/)
* [周志明的《深入理解Java虚拟机》](http://book.douban.com/subject/6522893/)
* [Java偏向锁实现原理](http://kenwublog.com/theory-of-java-biased-locking)

## 作者简介

**方腾飞**，阿里巴巴资深软件开发工程师，致力于高性能网络和并发编程，目前在公司从事询盘管理和长连接服务器OpenComet的开发工作。 博客地址：[http://ifeve.com](http://ifeve.com/) 微博地址：<http://weibo.com/kirals>

# 聊聊并发（三）——JAVA线程池的分析和使用

<http://www.infoq.com/cn/articles/java-threadPool>

## 1. 引言

合理利用线程池能够带来三个好处。第一：降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。第二：提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。第三：提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。但是要做到合理的利用线程池，必须对其原理了如指掌。

## 2. 线程池的使用

**线程池的创建**

我们可以通过ThreadPoolExecutor来创建一个线程池。

new ThreadPoolExecutor(corePoolSize, maximumPoolSize, keepAliveTime, milliseconds,runnableTaskQueue, handler);

创建一个线程池需要输入几个参数：

* corePoolSize（线程池的基本大小）：当提交一个任务到线程池时，线程池会创建一个线程来执行任务，即使其他空闲的基本线程能够执行新任务也会创建线程，等到需要执行的任务数大于线程池基本大小时就不再创建。如果调用了线程池的prestartAllCoreThreads方法，线程池会提前创建并启动所有基本线程。
* runnableTaskQueue（任务队列）：用于保存等待执行的任务的阻塞队列。 可以选择以下几个阻塞队列。
  + ArrayBlockingQueue：是一个基于数组结构的有界阻塞队列，此队列按 FIFO（先进先出）原则对元素进行排序。
  + LinkedBlockingQueue：一个基于链表结构的阻塞队列，此队列按FIFO （先进先出） 排序元素，吞吐量通常要高于ArrayBlockingQueue。静态工厂方法Executors.newFixedThreadPool()使用了这个队列。
  + SynchronousQueue：一个不存储元素的阻塞队列。每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于LinkedBlockingQueue，静态工厂方法Executors.newCachedThreadPool使用了这个队列。
  + PriorityBlockingQueue：一个具有优先级的无限阻塞队列。
* maximumPoolSize（线程池最大大小）：线程池允许创建的最大线程数。如果队列满了，并且已创建的线程数小于最大线程数，则线程池会再创建新的线程执行任务。值得注意的是如果使用了无界的任务队列这个参数就没什么效果。
* ThreadFactory：用于设置创建线程的工厂，可以通过线程工厂给每个创建出来的线程设置更有意义的名字。
* RejectedExecutionHandler（饱和策略）：当队列和线程池都满了，说明线程池处于饱和状态，那么必须采取一种策略处理提交的新任务。这个策略默认情况下是AbortPolicy，表示无法处理新任务时抛出异常。以下是JDK1.5提供的四种策略。
  + AbortPolicy：直接抛出异常。
  + CallerRunsPolicy：只用调用者所在线程来运行任务。
  + DiscardOldestPolicy：丢弃队列里最近的一个任务，并执行当前任务。
  + DiscardPolicy：不处理，丢弃掉。
  + 当然也可以根据应用场景需要来实现RejectedExecutionHandler接口自定义策略。如记录日志或持久化不能处理的任务。
* keepAliveTime（线程活动保持时间）：线程池的工作线程空闲后，保持存活的时间。所以如果任务很多，并且每个任务执行的时间比较短，可以调大这个时间，提高线程的利用率。
* TimeUnit（线程活动保持时间的单位）：可选的单位有天（DAYS），小时（HOURS），分钟（MINUTES），毫秒(MILLISECONDS)，微秒(MICROSECONDS, 千分之一毫秒)和毫微秒(NANOSECONDS, 千分之一微秒)。

**向线程池提交任务**

我们可以使用execute提交的任务，但是execute方法没有返回值，所以无法判断任务是否被线程池执行成功。通过以下代码可知execute方法输入的任务是一个Runnable类的实例。

threadsPool.execute(new Runnable() {

@Override

public void run() {

// TODO Auto-generated method stub

}

});

我们也可以使用submit 方法来提交任务，它会返回一个future,那么我们可以通过这个future来判断任务是否执行成功，通过future的get方法来获取返回值，get方法会阻塞住直到任务完成，而使用get(long timeout, TimeUnit unit)方法则会阻塞一段时间后立即返回，这时有可能任务没有执行完。

Future<Object> future = executor.submit(harReturnValuetask);

try {

Object s = future.get();

} catch (InterruptedException e) {

// 处理中断异常

} catch (ExecutionException e) {

// 处理无法执行任务异常

} finally {

// 关闭线程池

executor.shutdown();

}

**线程池的关闭**

我们可以通过调用线程池的shutdown或shutdownNow方法来关闭线程池，它们的原理是遍历线程池中的工作线程，然后逐个调用线程的interrupt方法来中断线程，所以无法响应中断的任务可能永远无法终止。但是它们存在一定的区别，shutdownNow首先将线程池的状态设置成STOP，然后尝试停止所有的正在执行或暂停任务的线程，并返回等待执行任务的列表，而shutdown只是将线程池的状态设置成SHUTDOWN状态，然后中断所有没有正在执行任务的线程。

只要调用了这两个关闭方法的其中一个，isShutdown方法就会返回true。当所有的任务都已关闭后,才表示线程池关闭成功，这时调用isTerminaed方法会返回true。至于我们应该调用哪一种方法来关闭线程池，应该由提交到线程池的任务特性决定，通常调用shutdown来关闭线程池，如果任务不一定要执行完，则可以调用shutdownNow。

## 3. 线程池的分析

流程分析：线程池的主要工作流程如下图：



从上图我们可以看出，当提交一个新任务到线程池时，线程池的处理流程如下：

1. 首先线程池判断**基本线程池**是否已满？没满，创建一个工作线程来执行任务。满了，则进入下个流程。
2. 其次线程池判断**工作队列**是否已满？没满，则将新提交的任务存储在工作队列里。满了，则进入下个流程。
3. 最后线程池判断**整个线程池**是否已满？没满，则创建一个新的工作线程来执行任务，满了，则交给饱和策略来处理这个任务。

**源码分析**。上面的流程分析让我们很直观的了解了线程池的工作原理，让我们再通过源代码来看看是如何实现的。线程池执行任务的方法如下：

public void execute(Runnable command) {

if (command == null)

throw new NullPointerException();

//如果线程数小于基本线程数，则创建线程并执行当前任务

if (poolSize >= corePoolSize || !addIfUnderCorePoolSize(command)) {

//如线程数大于等于基本线程数或线程创建失败，则将当前任务放到工作队列中。

if (runState == RUNNING && workQueue.offer(command)) {

if (runState != RUNNING || poolSize == 0)

ensureQueuedTaskHandled(command);

}

//如果线程池不处于运行中或任务无法放入队列，并且当前线程数量小于最大允许的线程数量，

则创建一个线程执行任务。

else if (!addIfUnderMaximumPoolSize(command))

//抛出RejectedExecutionException异常

reject(command); // is shutdown or saturated

}

}

**工作线程**。线程池创建线程时，会将线程封装成工作线程Worker，Worker在执行完任务后，还会无限循环获取工作队列里的任务来执行。我们可以从Worker的run方法里看到这点：

public void run() {

try {

Runnable task = firstTask;

firstTask = null;

while (task != null || (task = getTask()) != null) {

runTask(task);

task = null;

}

} finally {

workerDone(this);

}

}

## 4. 合理的配置线程池

要想合理的配置线程池，就必须首先分析任务特性，可以从以下几个角度来进行分析：

1. 任务的性质：CPU密集型任务，IO密集型任务和混合型任务。
2. 任务的优先级：高，中和低。
3. 任务的执行时间：长，中和短。
4. 任务的依赖性：是否依赖其他系统资源，如数据库连接。

任务性质不同的任务可以用不同规模的线程池分开处理。CPU密集型任务配置尽可能小的线程，如配置Ncpu+1个线程的线程池。IO密集型任务则由于线程并不是一直在执行任务，则配置尽可能多的线程，如2\*Ncpu。混合型的任务，如果可以拆分，则将其拆分成一个CPU密集型任务和一个IO密集型任务，只要这两个任务执行的时间相差不是太大，那么分解后执行的吞吐率要高于串行执行的吞吐率，如果这两个任务执行时间相差太大，则没必要进行分解。我们可以通过Runtime.getRuntime().availableProcessors()方法获得当前设备的CPU个数。

优先级不同的任务可以使用优先级队列PriorityBlockingQueue来处理。它可以让优先级高的任务先得到执行，需要注意的是如果一直有优先级高的任务提交到队列里，那么优先级低的任务可能永远不能执行。

执行时间不同的任务可以交给不同规模的线程池来处理，或者也可以使用优先级队列，让执行时间短的任务先执行。

依赖数据库连接池的任务，因为线程提交SQL后需要等待数据库返回结果，如果等待的时间越长CPU空闲时间就越长，那么线程数应该设置越大，这样才能更好的利用CPU。

建议使用有界队列，有界队列能增加系统的稳定性和预警能力，可以根据需要设大一点，比如几千。有一次我们组使用的后台任务线程池的队列和线程池全满了，不断的抛出抛弃任务的异常，通过排查发现是数据库出现了问题，导致执行SQL变得非常缓慢，因为后台任务线程池里的任务全是需要向数据库查询和插入数据的，所以导致线程池里的工作线程全部阻塞住，任务积压在线程池里。如果当时我们设置成无界队列，线程池的队列就会越来越多，有可能会撑满内存，导致整个系统不可用，而不只是后台任务出现问题。当然我们的系统所有的任务是用的单独的服务器部署的，而我们使用不同规模的线程池跑不同类型的任务，但是出现这样问题时也会影响到其他任务。

## 5. 线程池的监控

通过线程池提供的参数进行监控。线程池里有一些属性在监控线程池的时候可以使用

* taskCount：线程池需要执行的任务数量。
* completedTaskCount：线程池在运行过程中已完成的任务数量。小于或等于taskCount。
* largestPoolSize：线程池曾经创建过的最大线程数量。通过这个数据可以知道线程池是否满过。如等于线程池的最大大小，则表示线程池曾经满了。
* getPoolSize:线程池的线程数量。如果线程池不销毁的话，池里的线程不会自动销毁，所以这个大小只增不+
* getActiveCount：获取活动的线程数。

通过扩展线程池进行监控。通过继承线程池并重写线程池的beforeExecute，afterExecute和terminated方法，我们可以在任务执行前，执行后和线程池关闭前干一些事情。如监控任务的平均执行时间，最大执行时间和最小执行时间等。这几个方法在线程池里是空方法。如：

protected void beforeExecute(Thread t, Runnable r) { }

## 6. 参考资料

* Java并发编程实战。
* JDK1.6源码

## 作者介绍

**方腾飞**，花名清英，淘宝资深开发工程师，关注并发编程，目前在广告技术部从事无线广告联盟的开发和设计工作。个人博客：[http://ifeve.com](http://ifeve.com/) 微博：<http://weibo.com/kirals> 欢迎通过我的微博进行技术交流。

# 聊聊并发（四）——深入分析ConcurrentHashMap

<http://www.infoq.com/cn/articles/ConcurrentHashMap>

## 术语定义

|  |  |  |
| --- | --- | --- |
| 术语 | 英文 | 解释 |
| 哈希算法 | hash algorithm | 是一种将任意内容的输入转换成相同长度输出的加密方式，其输出被称为哈希值。 |
| 哈希表 | hash table | 根据设定的哈希函数H(key)和处理冲突方法将一组关键字映象到一个有限的地址区间上，并以关键字在地址区间中的象作为记录在表中的存储位置，这种表称为哈希表或散列，所得存储位置称为哈希地址或散列地址。 |

## 线程不安全的HashMap

因为多线程环境下，使用HashMap进行put操作会引起死循环，导致CPU利用率接近100%，所以在并发情况下不能使用HashMap，如以下代码

final HashMap<String, String> map = new HashMap<String, String>(2);

Thread t = new Thread(new Runnable() {

@Override

public void run() {

for (int i = 0; i < 10000; i++) {

new Thread(new Runnable() {

@Override

public void run() {

map.put(UUID.randomUUID().toString(), "");

}

}, "ftf" + i).start();

}

}

}, "ftf");

t.start();

t.join();

## 效率低下的HashTable容器

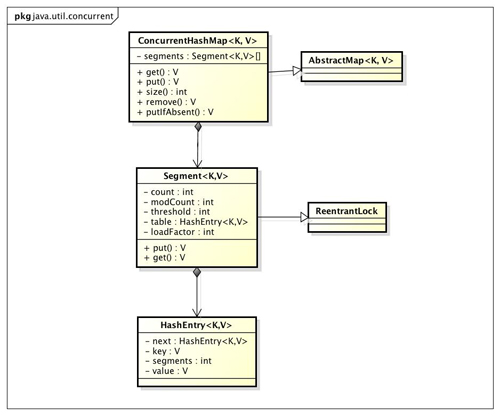
HashTable容器使用synchronized来保证线程安全，但在线程竞争激烈的情况下HashTable的效率非常低下。因为当一个线程访问HashTable的同步方法时，其他线程访问HashTable的同步方法时，可能会进入阻塞或轮询状态。如线程1使用put进行添加元素，线程2不但不能使用put方法添加元素，并且也不能使用get方法来获取元素，所以竞争越激烈效率越低。

## 锁分段技术

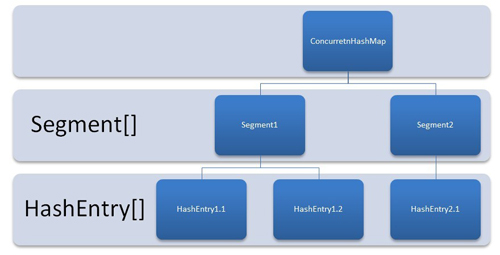
HashTable容器在竞争激烈的并发环境下表现出效率低下的原因是所有访问HashTable的线程都必须竞争同一把锁，那假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提高并发访问效率，这就是ConcurrentHashMap所使用的锁分段技术，首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。

## ConcurrentHashMap的结构

我们通过ConcurrentHashMap的类图来分析ConcurrentHashMap的结构。



ConcurrentHashMap是由Segment数组结构和HashEntry数组结构组成。Segment是一种可重入锁ReentrantLock，在ConcurrentHashMap里扮演锁的角色，HashEntry则用于存储键值对数据。一个ConcurrentHashMap里包含一个Segment数组，Segment的结构和HashMap类似，是一种数组和链表结构， 一个Segment里包含一个HashEntry数组，每个HashEntry是一个链表结构的元素， 每个Segment守护者一个HashEntry数组里的元素,当对HashEntry数组的数据进行修改时，必须首先获得它对应的Segment锁。



## ConcurrentHashMap的初始化

ConcurrentHashMap初始化方法是通过initialCapacity，loadFactor, concurrencyLevel几个参数来初始化segments数组，段偏移量segmentShift，段掩码segmentMask和每个segment里的HashEntry数组 。

初始化segments数组。让我们来看一下初始化segmentShift，segmentMask和segments数组的源代码。

if (concurrencyLevel > MAX\_SEGMENTS)

concurrencyLevel = MAX\_SEGMENTS;

// Find power-of-two sizes best matching arguments

int sshift = 0;

int ssize = 1;

while (ssize < concurrencyLevel) {

++sshift;

ssize <<= 1;

}

segmentShift = 32 - sshift;

segmentMask = ssize - 1;

this.segments = Segment.newArray(ssize);

由上面的代码可知segments数组的长度ssize通过concurrencyLevel计算得出。为了能通过按位与的哈希算法来定位segments数组的索引，必须保证segments数组的长度是2的N次方（power-of-two size），所以必须计算出一个是大于或等于concurrencyLevel的最小的2的N次方值来作为segments数组的长度。假如concurrencyLevel等于14，15或16，ssize都会等于16，即容器里锁的个数也是16。注意concurrencyLevel的最大大小是65535，意味着segments数组的长度最大为65536，对应的二进制是16位。

初始化segmentShift和segmentMask。这两个全局变量在定位segment时的哈希算法里需要使用，sshift等于ssize从1向左移位的次数，在默认情况下concurrencyLevel等于16，1需要向左移位移动4次，所以sshift等于4。segmentShift用于定位参与hash运算的位数，segmentShift等于32减sshift，所以等于28，这里之所以用32是因为ConcurrentHashMap里的hash()方法输出的最大数是32位的，后面的测试中我们可以看到这点。segmentMask是哈希运算的掩码，等于ssize减1，即15，掩码的二进制各个位的值都是1。因为ssize的最大长度是65536，所以segmentShift最大值是16，segmentMask最大值是65535，对应的二进制是16位，每个位都是1。

初始化每个Segment。输入参数initialCapacity是ConcurrentHashMap的初始化容量，loadfactor是每个segment的负载因子，在构造方法里需要通过这两个参数来初始化数组中的每个segment。

if (initialCapacity > MAXIMUM\_CAPACITY)

initialCapacity = MAXIMUM\_CAPACITY;

int c = initialCapacity / ssize;

if (c \* ssize < initialCapacity)

++c;

int cap = 1;

while (cap < c)

cap <<= 1;

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

this.segments[i] = new Segment<K,V>(cap, loadFactor);

上面代码中的变量cap就是segment里HashEntry数组的长度，它等于initialCapacity除以ssize的倍数c，如果c大于1，就会取大于等于c的2的N次方值，所以cap不是1，就是2的N次方。segment的容量threshold＝(int)cap\*loadFactor，默认情况下initialCapacity等于16，loadfactor等于0.75，通过运算cap等于1，threshold等于零。

## 定位Segment

既然ConcurrentHashMap使用分段锁Segment来保护不同段的数据，那么在插入和获取元素的时候，必须先通过哈希算法定位到Segment。可以看到ConcurrentHashMap会首先使用Wang/Jenkins hash的变种算法对元素的hashCode进行一次再哈希。

private static int hash(int h) {

h += (h << 15) ^ 0xffffcd7d;

h ^= (h >>> 10);

h += (h << 3);

h ^= (h >>> 6);

h += (h << 2) + (h << 14);

return h ^ (h >>> 16);

}

之所以进行再哈希，其目的是为了减少哈希冲突，使元素能够均匀的分布在不同的Segment上，从而提高容器的存取效率。假如哈希的质量差到极点，那么所有的元素都在一个Segment中，不仅存取元素缓慢，分段锁也会失去意义。我做了一个测试，不通过再哈希而直接执行哈希计算。

System.out.println(Integer.parseInt("0001111", 2) & 15);

System.out.println(Integer.parseInt("0011111", 2) & 15);

System.out.println(Integer.parseInt("0111111", 2) & 15);

System.out.println(Integer.parseInt("1111111", 2) & 15);

计算后输出的哈希值全是15，通过这个例子可以发现如果不进行再哈希，哈希冲突会非常严重，因为只要低位一样，无论高位是什么数，其哈希值总是一样。我们再把上面的二进制数据进行再哈希后结果如下，为了方便阅读，不足32位的高位补了0，每隔四位用竖线分割下。

0100｜0111｜0110｜0111｜1101｜1010｜0100｜1110

1111｜0111｜0100｜0011｜0000｜0001｜1011｜1000

0111｜0111｜0110｜1001｜0100｜0110｜0011｜1110

1000｜0011｜0000｜0000｜1100｜1000｜0001｜1010

可以发现每一位的数据都散列开了，通过这种再哈希能让数字的每一位都能参加到哈希运算当中，从而减少哈希冲突。ConcurrentHashMap通过以下哈希算法定位segment。

final Segment<K,V> segmentFor(int hash) {

return segments[(hash >>> segmentShift) & segmentMask];

}

默认情况下segmentShift为28，segmentMask为15，再哈希后的数最大是32位二进制数据，向右无符号移动28位，意思是让高4位参与到hash运算中， (hash >>> segmentShift) & segmentMask的运算结果分别是4，15，7和8，可以看到hash值没有发生冲突。

## ConcurrentHashMap的get操作

Segment的get操作实现非常简单和高效。先经过一次再哈希，然后使用这个哈希值通过哈希运算定位到segment，再通过哈希算法定位到元素，代码如下：

public V get(Object key) {

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

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

}

get操作的高效之处在于整个get过程不需要加锁，除非读到的值是空的才会加锁重读，我们知道HashTable容器的get方法是需要加锁的，那么ConcurrentHashMap的get操作是如何做到不加锁的呢？原因是它的get方法里将要使用的共享变量都定义成volatile，如用于统计当前Segement大小的count字段和用于存储值的HashEntry的value。定义成volatile的变量，能够在线程之间保持可见性，能够被多线程同时读，并且保证不会读到过期的值，但是只能被单线程写（有一种情况可以被多线程写，就是写入的值不依赖于原值），在get操作里只需要读不需要写共享变量count和value，所以可以不用加锁。之所以不会读到过期的值，是根据java内存模型的happen before原则，对volatile字段的写入操作先于读操作，即使两个线程同时修改和获取volatile变量，get操作也能拿到最新的值，这是用volatile替换锁的经典应用场景。

transient volatile int count;

volatile V value;

在定位元素的代码里我们可以发现定位HashEntry和定位Segment的哈希算法虽然一样，都与数组的长度减去一相与，但是相与的值不一样，定位Segment使用的是元素的hashcode通过再哈希后得到的值的高位，而定位HashEntry直接使用的是再哈希后的值。其目的是避免两次哈希后的值一样，导致元素虽然在Segment里散列开了，但是却没有在HashEntry里散列开。

hash >>> segmentShift) & segmentMask//定位Segment所使用的hash算法

int index = hash & (tab.length - 1);// 定位HashEntry所使用的hash算法

## ConcurrentHashMap的Put操作

由于put方法里需要对共享变量进行写入操作，所以为了线程安全，在操作共享变量时必须得加锁。Put方法首先定位到Segment，然后在Segment里进行插入操作。插入操作需要经历两个步骤，第一步判断是否需要对Segment里的HashEntry数组进行扩容，第二步定位添加元素的位置然后放在HashEntry数组里。

是否需要扩容。在插入元素前会先判断Segment里的HashEntry数组是否超过容量（threshold），如果超过阀值，数组进行扩容。值得一提的是，Segment的扩容判断比HashMap更恰当，因为HashMap是在插入元素后判断元素是否已经到达容量的，如果到达了就进行扩容，但是很有可能扩容之后没有新元素插入，这时HashMap就进行了一次无效的扩容。

如何扩容。扩容的时候首先会创建一个两倍于原容量的数组，然后将原数组里的元素进行再hash后插入到新的数组里。为了高效ConcurrentHashMap不会对整个容器进行扩容，而只对某个segment进行扩容。

## ConcurrentHashMap的size操作

如果我们要统计整个ConcurrentHashMap里元素的大小，就必须统计所有Segment里元素的大小后求和。Segment里的全局变量count是一个volatile变量，那么在多线程场景下，我们是不是直接把所有Segment的count相加就可以得到整个ConcurrentHashMap大小了呢？不是的，虽然相加时可以获取每个Segment的count的最新值，但是拿到之后可能累加前使用的count发生了变化，那么统计结果就不准了。所以最安全的做法，是在统计size的时候把所有Segment的put，remove和clean方法全部锁住，但是这种做法显然非常低效。 因为在累加count操作过程中，之前累加过的count发生变化的几率非常小，所以ConcurrentHashMap的做法是先尝试2次通过不锁住Segment的方式来统计各个Segment大小，如果统计的过程中，容器的count发生了变化，则再采用加锁的方式来统计所有Segment的大小。

那么ConcurrentHashMap是如何判断在统计的时候容器是否发生了变化呢？使用modCount变量，在put , remove和clean方法里操作元素前都会将变量modCount进行加1，那么在统计size前后比较modCount是否发生变化，从而得知容器的大小是否发生变化。

## 参考资料

1. JDK1.6源代码。
2. 《Java并发编程实践》。
3. [Java并发编程之ConcurrentHashMap](http://www.goldendoc.org/2011/06/juc_concurrenthashmap/) 。

## 作者介绍

**方腾飞**，花名清英，淘宝资深开发工程师，关注并发编程，目前在广告技术部从事无线广告联盟的开发和设计工作。个人博客：[http://ifeve.com](http://ifeve.com/) 微博：<http://weibo.com/kirals> 欢迎通过我的微博进行技术交流。

# 聊聊并发（五）——原子操作的实现原理

<http://www.infoq.com/cn/articles/atomic-operation>

## 1. 引言

原子（atom）本意是“不能被进一步分割的最小粒子”，而原子操作（atomic operation）意为"不可被中断的一个或一系列操作" 。在多处理器上实现原子操作就变得有点复杂。本文让我们一起来聊一聊在Intel处理器和Java里是如何实现原子操作的。

## 2. 术语定义

|  |  |  |
| --- | --- | --- |
| 术语 | 英文 | 解释 |
| 缓存行 | Cache line | 缓存的最小操作单位 |
| 比较并交换 | Compare and Swap | CAS操作需要输入两个数值，一个旧值（期望操作前的值）和一个新值，在操作期间先比较下旧值有没有发生变化，如果没有发生变化，才交换成新值，发生了变化则不交换。 |
| CPU流水线 | CPU pipeline | CPU流水线的工作方式就象工业生产上的装配流水线，在CPU中由5~6个不同功能的电路单元组成一条指令处理流水线，然后将一条X86指令分成5~6步后再由这些电路单元分别执行，这样就能实现在一个CPU时钟周期完成一条指令，因此提高CPU的运算速度。 |
| 内存顺序冲突 | Memory order violation | 内存顺序冲突一般是由假共享引起，假共享是指多个CPU同时修改同一个缓存行的不同部分而引起其中一个CPU的操作无效，当出现这个内存顺序冲突时，CPU必须清空流水线。 |

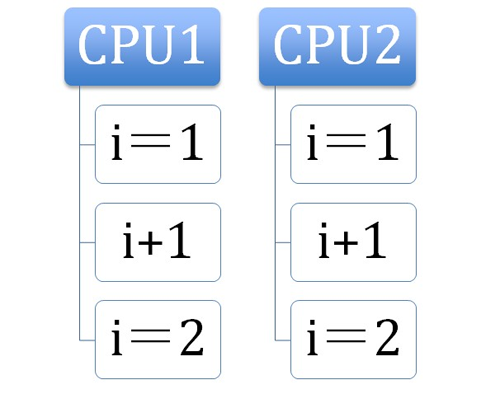
## 3. 处理器如何实现原子操作

32位IA-32处理器使用基于对缓存加锁或总线加锁的方式来实现多处理器之间的原子操作。

### 3.1 处理器自动保证基本内存操作的原子性

首先处理器会自动保证基本的内存操作的原子性。处理器保证从系统内存当中读取或者写入一个字节是原子的，意思是当一个处理器读取一个字节时，其他处理器不能访问这个字节的内存地址。奔腾6和最新的处理器能自动保证单处理器对同一个缓存行里进行16/32/64位的操作是原子的，但是复杂的内存操作处理器不能自动保证其原子性，比如跨总线宽度，跨多个缓存行，跨页表的访问。但是处理器提供总线锁定和缓存锁定两个机制来保证复杂内存操作的原子性。

第一个机制是通过总线锁保证原子性。如果多个处理器同时对共享变量进行读改写（i++就是经典的读改写操作）操作，那么共享变量就会被多个处理器同时进行操作，这样读改写操作就不是原子的，操作完之后共享变量的值会和期望的不一致，举个例子：如果i=1,我们进行两次i++操作，我们期望的结果是3，但是有可能结果是2。如下图



（例1）

原因是有可能多个处理器同时从各自的缓存中读取变量i，分别进行加一操作，然后分别写入系统内存当中。那么想要保证读改写共享变量的操作是原子的，就必须保证CPU1读改写共享变量的时候，CPU2不能操作缓存了该共享变量内存地址的缓存。

处理器使用总线锁就是来解决这个问题的。所谓总线锁就是使用处理器提供的一个LOCK＃信号，当一个处理器在总线上输出此信号时，其他处理器的请求将被阻塞住,那么该处理器可以独占使用共享内存。

### 3.3 使用缓存锁保证原子性

第二个机制是通过缓存锁定保证原子性。在同一时刻我们只需保证对某个内存地址的操作是原子性即可，但总线锁定把CPU和内存之间通信锁住了，这使得锁定期间，其他处理器不能操作其他内存地址的数据，所以总线锁定的开销比较大，最近的处理器在某些场合下使用缓存锁定代替总线锁定来进行优化。

频繁使用的内存会缓存在处理器的L1，L2和L3高速缓存里，那么原子操作就可以直接在处理器内部缓存中进行，并不需要声明总线锁，在奔腾6和最近的处理器中可以使用“缓存锁定”的方式来实现复杂的原子性。所谓“缓存锁定”就是如果缓存在处理器缓存行中内存区域在LOCK操作期间被锁定，当它执行锁操作回写内存时，处理器不在总线上声言LOCK＃信号，而是修改内部的内存地址，并允许它的缓存一致性机制来保证操作的原子性，因为缓存一致性机制会阻止同时修改被两个以上处理器缓存的内存区域数据，当其他处理器回写已被锁定的缓存行的数据时会起缓存行无效，在例1中，当CPU1修改缓存行中的i时使用缓存锁定，那么CPU2就不能同时缓存了i的缓存行。

但是有两种情况下处理器不会使用缓存锁定。第一种情况是：当操作的数据不能被缓存在处理器内部，或操作的数据跨多个缓存行（cache line），则处理器会调用总线锁定。第二种情况是：有些处理器不支持缓存锁定。对于Inter486和奔腾处理器,就算锁定的内存区域在处理器的缓存行中也会调用总线锁定。

以上两个机制我们可以通过Inter处理器提供了很多LOCK前缀的指令来实现。比如位测试和修改指令BTS，BTR，BTC，交换指令XADD，CMPXCHG和其他一些操作数和逻辑指令，比如ADD（加），OR（或）等，被这些指令操作的内存区域就会加锁，导致其他处理器不能同时访问它。

## 4. JAVA如何实现原子操作

在java中可以通过锁和循环CAS的方式来实现原子操作。

### 4.1 使用循环CAS实现原子操作

JVM中的CAS操作正是利用了上一节中提到的处理器提供的CMPXCHG指令实现的。自旋CAS实现的基本思路就是循环进行CAS操作直到成功为止，以下代码实现了一个基于CAS线程安全的计数器方法safeCount和一个非线程安全的计数器count。

public class Counter {

private AtomicInteger atomicI = new AtomicInteger(0);

private int i = 0;

public static void main(String[] args) {

final Counter cas = new Counter();

List<Thread> ts = new ArrayList<Thread>(600);

long start = System.currentTimeMillis();

for (int j = 0; j < 100; j++) {

Thread t = new Thread(new Runnable() {

@Override

public void run() {

for (int i = 0; i < 10000; i++) {

cas.count();

cas.safeCount();

}

}

});

ts.add(t);

}

for (Thread t : ts) {

t.start();

}

// 等待所有线程执行完成

for (Thread t : ts) {

try {

t.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

System.out.println(cas.i);

System.out.println(cas.atomicI.get());

System.out.println(System.currentTimeMillis() - start);

}

/\*\*

\* 使用CAS实现线程安全计数器

\*/

private void safeCount() {

for (;;) {

int i = atomicI.get();

boolean suc = atomicI.compareAndSet(i, ++i);

if (suc) {

break;

}

}

}

/\*\*

\* 非线程安全计数器

\*/

private void count() {

i++;

}

}

在java并发包中有一些并发框架也使用了自旋CAS的方式来实现原子操作，比如LinkedTransferQueue类的Xfer方法。CAS虽然很高效的解决原子操作，但是CAS仍然存在三大问题。ABA问题，循环时间长开销大和只能保证一个共享变量的原子操作。

ABA问题。因为CAS需要在操作值的时候检查下值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。ABA问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加一，那么A－B－A 就会变成1A-2B－3A。  
从Java1.5开始JDK的atomic包里提供了一个类AtomicStampedReference来解决ABA问题。这个类的compareAndSet方法作用是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

public boolean compareAndSet

(V expectedReference,//预期引用

V newReference,//更新后的引用

int expectedStamp, //预期标志

int newStamp) //更新后的标志

循环时间长开销大。自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。如果JVM能支持处理器提供的pause指令那么效率会有一定的提升，pause指令有两个作用，第一它可以延迟流水线执行指令（de-pipeline）,使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突（memory order violation）而引起CPU流水线被清空（CPU pipeline flush），从而提高CPU的执行效率。

只能保证一个共享变量的原子操作。当对一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作，但是对多个共享变量操作时，循环CAS就无法保证操作的原子性，这个时候就可以用锁，或者有一个取巧的办法，就是把多个共享变量合并成一个共享变量来操作。比如有两个共享变量i＝2,j=a，合并一下ij=2a，然后用CAS来操作ij。从Java1.5开始JDK提供了AtomicReference类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行CAS操作。

### 4.2 使用锁机制实现原子操作

锁机制保证了只有获得锁的线程能够操作锁定的内存区域。JVM内部实现了很多种锁机制，有偏向锁，轻量级锁和互斥锁，有意思的是除了偏向锁，JVM实现锁的方式都用到的循环CAS，当一个线程想进入同步块的时候使用循环CAS的方式来获取锁，当它退出同步块的时候使用循环CAS释放锁。详细说明可以参见文章Java SE1.6中的Synchronized。

## 5. 参考资料

1. [Java SE1.6中的Synchronized](http://www.infoq.com/cn/articles/java-se-16-synchronized)
2. [Intel 64和IA-32架构软件开发人员手册](http://www.intel.com/products/processor/manuals/)
3. [深入分析Volatile的实现原理](http://www.infoq.com/cn/articles/ftf-java-volatile)

## 作者介绍

**方腾飞**，花名清英，淘宝资深开发工程师，关注并发编程，目前在广告技术部从事无线广告联盟的开发和设计工作。个人博客：[http://ifeve.com](http://ifeve.com/) 微博：<http://weibo.com/kirals> 欢迎通过我的微博进行技术交流。

# 聊聊并发（六）——ConcurrentLinkedQueue的实现原理分析

<http://www.infoq.com/cn/articles/ConcurrentLinkedQueue>

## 1. 引言

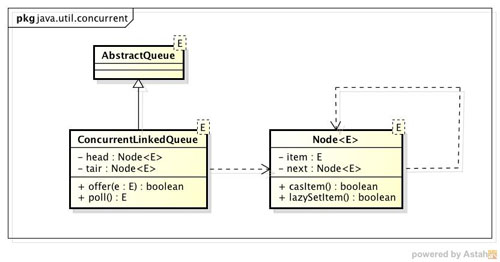
在并发编程中我们有时候需要使用线程安全的队列。如果我们要实现一个线程安全的队列有两种实现方式：一种是使用阻塞算法，另一种是使用非阻塞算法。使用阻塞算法的队列可以用一个锁（入队和出队用同一把锁）或两个锁（入队和出队用不同的锁）等方式来实现，而非阻塞的实现方式则可以使用循环CAS的方式来实现，本文让我们一起来研究下Doug Lea是如何使用非阻塞的方式来实现线程安全队列ConcurrentLinkedQueue的，相信从大师身上我们能学到不少并发编程的技巧。

## 2. ConcurrentLinkedQueue的介绍

ConcurrentLinkedQueue是一个基于链接节点的无界线程安全队列，它采用先进先出的规则对节点进行排序，当我们添加一个元素的时候，它会添加到队列的尾部，当我们获取一个元素时，它会返回队列头部的元素。它采用了“wait－free”算法来实现，该算法在Michael & Scott算法上进行了一些修改, Michael & Scott算法的详细信息可以参见[参考资料一](http://www.cs.rochester.edu/u/michael/PODC96.html)。

## 3. ConcurrentLinkedQueue的结构

我们通过ConcurrentLinkedQueue的类图来分析一下它的结构。



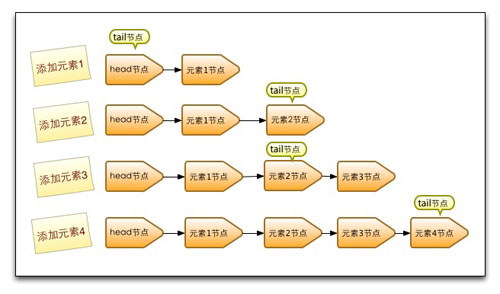
（图1）

ConcurrentLinkedQueue由head节点和tair节点组成，每个节点（Node）由节点元素（item）和指向下一个节点的引用(next)组成，节点与节点之间就是通过这个next关联起来，从而组成一张链表结构的队列。默认情况下head节点存储的元素为空，tair节点等于head节点。

private transient volatile Node<E> tail = head;

## 4. 入队列

**入队列就是将入队节点添加到队列的尾部**。为了方便理解入队时队列的变化，以及head节点和tair节点的变化，每添加一个节点我就做了一个队列的快照图。



（图二）

* 第一步添加元素1。队列更新head节点的next节点为元素1节点。又因为tail节点默认情况下等于head节点，所以它们的next节点都指向元素1节点。
* 第二步添加元素2。队列首先设置元素1节点的next节点为元素2节点，然后更新tail节点指向元素2节点。
* 第三步添加元素3，设置tail节点的next节点为元素3节点。
* 第四步添加元素4，设置元素3的next节点为元素4节点，然后将tail节点指向元素4节点。

通过debug入队过程并观察head节点和tail节点的变化，发现入队主要做两件事情，第一是将入队节点设置成当前队列尾节点的下一个节点。第二是更新tail节点，如果tail节点的next节点不为空，则将入队节点设置成tail节点，如果tail节点的next节点为空，则将入队节点设置成tail的next节点，所以tail节点不总是尾节点，理解这一点对于我们研究源码会非常有帮助。

上面的分析让我们从单线程入队的角度来理解入队过程，但是多个线程同时进行入队情况就变得更加复杂，因为可能会出现其他线程插队的情况。如果有一个线程正在入队，那么它必须先获取尾节点，然后设置尾节点的下一个节点为入队节点，但这时可能有另外一个线程插队了，那么队列的尾节点就会发生变化，这时当前线程要暂停入队操作，然后重新获取尾节点。让我们再通过源码来详细分析下它是如何使用CAS算法来入队的。

public boolean offer(E e) {

if (e == null) throw new NullPointerException();

//入队前，创建一个入队节点

Node<E> n = new Node<E>(e);

retry:

//死循环，入队不成功反复入队。

for (;;) {

//创建一个指向tail节点的引用

Node<E> t = tail;

//p用来表示队列的尾节点，默认情况下等于tail节点。

Node<E> p = t;

for (int hops = 0; ; hops++) {

//获得p节点的下一个节点。

Node<E> next = succ(p);

//next节点不为空，说明p不是尾节点，需要更新p后在将它指向next节点

if (next != null) {

//循环了两次及其以上，并且当前节点还是不等于尾节点

if (hops > HOPS && t != tail)

continue retry;

p = next;

}

//如果p是尾节点，则设置p节点的next节点为入队节点。

else if (p.casNext(null, n)) {

//如果tail节点有大于等于1个next节点，则将入队节点设置成tair节点，更新失败了也

没关系，因为失败了表示有其他线程成功更新了tair节点。

if (hops >= HOPS)

casTail(t, n); // 更新tail节点，允许失败

return true;

}

// p有next节点,表示p的next节点是尾节点，则重新设置p节点

else {

p = succ(p);

}

}

}

}

**从源代码角度来看整个入队过程主要做二件事情**。第一是定位出尾节点，第二是使用CAS算法能将入队节点设置成尾节点的next节点，如不成功则重试。

**第一步定位尾节点**。tail节点并不总是尾节点，所以每次入队都必须先通过tail节点来找到尾节点，尾节点可能就是tail节点，也可能是tail节点的next节点。代码中循环体中的第一个if就是判断tail是否有next节点，有则表示next节点可能是尾节点。获取tail节点的next节点需要注意的是p节点等于p的next节点的情况，只有一种可能就是p节点和p的next节点都等于空，表示这个队列刚初始化，正准备添加第一次节点，所以需要返回head节点。获取p节点的next节点代码如下

final Node<E> succ(Node<E> p) {

Node<E> next = p.getNext();

return (p == next) ? head : next;

}

**第二步设置入队节点为尾节点**。p.casNext(null, n)方法用于将入队节点设置为当前队列尾节点的next节点，p如果是null表示p是当前队列的尾节点，如果不为null表示有其他线程更新了尾节点，则需要重新获取当前队列的尾节点。

**hops的设计意图**。上面分析过对于先进先出的队列入队所要做的事情就是将入队节点设置成尾节点，doug lea写的代码和逻辑还是稍微有点复杂。那么我用以下方式来实现行不行？

public boolean offer(E e) {

if (e == null)

throw new NullPointerException();

Node<E> n = new Node<E>(e);

for (;;) {

Node<E> t = tail;

if (t.casNext(null, n) && casTail(t, n)) {

return true;

}

}

}

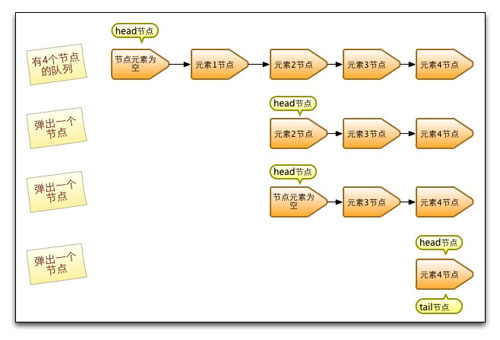
让tail节点永远作为队列的尾节点，这样实现代码量非常少，而且逻辑非常清楚和易懂。但是这么做有个缺点就是每次都需要使用循环CAS更新tail节点。如果能减少CAS更新tail节点的次数，就能提高入队的效率，所以doug lea使用hops变量来控制并减少tail节点的更新频率，并不是每次节点入队后都将 tail节点更新成尾节点，而是当 tail节点和尾节点的距离大于等于常量HOPS的值（默认等于1）时才更新tail节点，tail和尾节点的距离越长使用CAS更新tail节点的次数就会越少，但是距离越长带来的负面效果就是每次入队时定位尾节点的时间就越长，因为循环体需要多循环一次来定位出尾节点，但是这样仍然能提高入队的效率，因为从本质上来看它通过增加对volatile变量的读操作来减少了对volatile变量的写操作，而对volatile变量的写操作开销要远远大于读操作，所以入队效率会有所提升。

private static final int HOPS = 1;

还有一点需要注意的是入队方法永远返回true，所以不要通过返回值判断入队是否成功。

## 5. 出队列

出队列的就是从队列里返回一个节点元素，并清空该节点对元素的引用。让我们通过每个节点出队的快照来观察下head节点的变化。



从上图可知，并不是每次出队时都更新head节点，当head节点里有元素时，直接弹出head节点里的元素，而不会更新head节点。只有当head节点里没有元素时，出队操作才会更新head节点。这种做法也是通过hops变量来减少使用CAS更新head节点的消耗，从而提高出队效率。让我们再通过源码来深入分析下出队过程。

public E poll() {

Node<E> h = head;

// p表示头节点，需要出队的节点

Node<E> p = h;

for (int hops = 0;; hops++) {

// 获取p节点的元素

E item = p.getItem();

// 如果p节点的元素不为空，使用CAS设置p节点引用的元素为null,如果成功则返回p节点的元素。

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

if (hops >= HOPS) {

//将p节点下一个节点设置成head节点

Node<E> q = p.getNext();

updateHead(h, (q != null) ? q : p);

}

return item;

}

// 如果头节点的元素为空或头节点发生了变化，这说明头节点已经被另外一个线程修改了。那么获取p节点的下一个节点

Node<> next = succ(p);

// 如果p的下一个节点也为空，说明这个队列已经空了

if (next == null) {

// 更新头节点。

updateHead(h, p);

break;

}

// 如果下一个元素不为空，则将头节点的下一个节点设置成头节点

p = next;

}

return null;

}

首先获取头节点的元素，然后判断头节点元素是否为空，如果为空，表示另外一个线程已经进行了一次出队操作将该节点的元素取走，如果不为空，则使用CAS的方式将头节点的引用设置成null，如果CAS成功，则直接返回头节点的元素，如果不成功，表示另外一个线程已经进行了一次出队操作更新了head节点，导致元素发生了变化，需要重新获取头节点。

## 6. 参考资料

* [简单，快速和实用的阻塞和非阻塞并发队列算法](http://www.cs.rochester.edu/u/scott/papers/1996_PODC_queues.pdf)。
* [非阻塞算法在容器里的实现](http://www.ibm.com/developerworks/cn/java/j-lo-concurrent/index.html)。
* JDK1.6中ConcurrentLinkedQueue源码和注释。

## 

## 作者介绍

**方腾飞**，花名清英，淘宝资深开发工程师，关注并发编程，目前在广告技术部从事无线广告联盟的开发和设计工作。个人博客：[http://ifeve.com](http://ifeve.com/) 微博：<http://weibo.com/kirals>欢迎通过我的微博进行技术交流。

**聊聊并发（七）——Java中的阻塞队列**

<http://www.infoq.com/cn/articles/java-blocking-queue>

## 1. 什么是阻塞队列？

阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。这两个附加的操作是：在队列为空时，获取元素的线程会等待队列变为非空。当队列满时，存储元素的线程会等待队列可用。阻塞队列常用于生产者和消费者的场景，生产者是往队列里添加元素的线程，消费者是从队列里拿元素的线程。阻塞队列就是生产者存放元素的容器，而消费者也只从容器里拿元素。

阻塞队列提供了四种处理方法:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 方法\处理方式 | 抛出异常 | 返回特殊值 | 一直阻塞 | 超时退出 |
| 插入方法 | add(e) | offer(e) | put(e) | offer(e,time,unit) |
| 移除方法 | remove() | poll() | take() | poll(time,unit) |
| 检查方法 | element() | peek() | 不可用 | 不可用 |

* 抛出异常：是指当阻塞队列满时候，再往队列里插入元素，会抛出IllegalStateException("Queue full")异常。当队列为空时，从队列里获取元素时会抛出NoSuchElementException异常 。
* 返回特殊值：插入方法会返回是否成功，成功则返回true。移除方法，则是从队列里拿出一个元素，如果没有则返回null
* 一直阻塞：当阻塞队列满时，如果生产者线程往队列里put元素，队列会一直阻塞生产者线程，直到拿到数据，或者响应中断退出。当队列空时，消费者线程试图从队列里take元素，队列也会阻塞消费者线程，直到队列可用。
* 超时退出：当阻塞队列满时，队列会阻塞生产者线程一段时间，如果超过一定的时间，生产者线程就会退出。

## 2. Java里的阻塞队列

JDK7提供了7个阻塞队列。分别是

* ArrayBlockingQueue ：一个由数组结构组成的有界阻塞队列。
* LinkedBlockingQueue ：一个由链表结构组成的有界阻塞队列。
* PriorityBlockingQueue ：一个支持优先级排序的无界阻塞队列。
* DelayQueue：一个使用优先级队列实现的无界阻塞队列。
* SynchronousQueue：一个不存储元素的阻塞队列。
* LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。
* LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。

ArrayBlockingQueue是一个用数组实现的有界阻塞队列。此队列按照先进先出（FIFO）的原则对元素进行排序。默认情况下不保证访问者公平的访问队列，所谓公平访问队列是指阻塞的所有生产者线程或消费者线程，当队列可用时，可以按照阻塞的先后顺序访问队列，即先阻塞的生产者线程，可以先往队列里插入元素，先阻塞的消费者线程，可以先从队列里获取元素。通常情况下为了保证公平性会降低吞吐量。我们可以使用以下代码创建一个公平的阻塞队列：

ArrayBlockingQueue fairQueue = new ArrayBlockingQueue(1000,true);

访问者的公平性是使用可重入锁实现的，代码如下：

public ArrayBlockingQueue(int capacity, boolean fair) {

if (capacity <= 0)

throw new IllegalArgumentException();

this.items = new Object[capacity];

lock = new ReentrantLock(fair);

notEmpty = lock.newCondition();

notFull = lock.newCondition();

}

LinkedBlockingQueue是一个用链表实现的有界阻塞队列。此队列的默认和最大长度为Integer.MAX\_VALUE。此队列按照先进先出的原则对元素进行排序。

PriorityBlockingQueue是一个支持优先级的无界队列。默认情况下元素采取自然顺序排列，也可以通过比较器comparator来指定元素的排序规则。元素按照升序排列。

DelayQueue是一个支持延时获取元素的无界阻塞队列。队列使用PriorityQueue来实现。队列中的元素必须实现Delayed接口，在创建元素时可以指定多久才能从队列中获取当前元素。只有在延迟期满时才能从队列中提取元素。我们可以将DelayQueue运用在以下应用场景：

* 缓存系统的设计：可以用DelayQueue保存缓存元素的有效期，使用一个线程循环查询DelayQueue，一旦能从DelayQueue中获取元素时，表示缓存有效期到了。
* 定时任务调度。使用DelayQueue保存当天将会执行的任务和执行时间，一旦从DelayQueue中获取到任务就开始执行，从比如TimerQueue就是使用DelayQueue实现的。

队列中的Delayed必须实现compareTo来指定元素的顺序。比如让延时时间最长的放在队列的末尾。实现代码如下：

public int compareTo(Delayed other) {

if (other == this) // compare zero ONLY if same object

return 0;

if (other instanceof ScheduledFutureTask) {

ScheduledFutureTask x = (ScheduledFutureTask)other;

long diff = time - x.time;

if (diff < 0)

return -1;

else if (diff > 0)

return 1;

else if (sequenceNumber < x.sequenceNumber)

return -1;

else

return 1;

}

long d = (getDelay(TimeUnit.NANOSECONDS) -

other.getDelay(TimeUnit.NANOSECONDS));

return (d == 0) ? 0 : ((d < 0) ? -1 : 1);

}

**如何实现Delayed接口**

我们可以参考ScheduledThreadPoolExecutor里ScheduledFutureTask类。这个类实现了Delayed接口。首先：在对象创建的时候，使用time记录前对象什么时候可以使用，代码如下：

ScheduledFutureTask(Runnable r, V result, long ns, long period) {

super(r, result);

this.time = ns;

this.period = period;

this.sequenceNumber = sequencer.getAndIncrement();

}

然后使用getDelay可以查询当前元素还需要延时多久，代码如下：

public long getDelay(TimeUnit unit) {

return unit.convert(time - now(), TimeUnit.NANOSECONDS);

}

通过构造函数可以看出延迟时间参数ns的单位是纳秒，自己设计的时候最好使用纳秒，因为getDelay时可以指定任意单位，一旦以纳秒作为单位，而延时的时间又精确不到纳秒就麻烦了。使用时请注意当time小于当前时间时，getDelay会返回负数。

**如何实现延时队列**

延时队列的实现很简单，当消费者从队列里获取元素时，如果元素没有达到延时时间，就阻塞当前线程。

long delay = first.getDelay(TimeUnit.NANOSECONDS);

if (delay <= 0)

return q.poll();

else if (leader != null)

available.await();

SynchronousQueue是一个不存储元素的阻塞队列。每一个put操作必须等待一个take操作，否则不能继续添加元素。SynchronousQueue可以看成是一个传球手，负责把生产者线程处理的数据直接传递给消费者线程。队列本身并不存储任何元素，非常适合于传递性场景,比如在一个线程中使用的数据，传递给另外一个线程使用，SynchronousQueue的吞吐量高于LinkedBlockingQueue 和 ArrayBlockingQueue。

LinkedTransferQueue是一个由链表结构组成的无界阻塞TransferQueue队列。相对于其他阻塞队列，LinkedTransferQueue多了tryTransfer和transfer方法。

transfer方法。如果当前有消费者正在等待接收元素（消费者使用take()方法或带时间限制的poll()方法时），transfer方法可以把生产者传入的元素立刻transfer（传输）给消费者。如果没有消费者在等待接收元素，transfer方法会将元素存放在队列的tail节点，并等到该元素被消费者消费了才返回。transfer方法的关键代码如下：

Node pred = tryAppend(s, haveData);

return awaitMatch(s, pred, e, (how == TIMED), nanos);

第一行代码是试图把存放当前元素的s节点作为tail节点。第二行代码是让CPU自旋等待消费者消费元素。因为自旋会消耗CPU，所以自旋一定的次数后使用Thread.yield()方法来暂停当前正在执行的线程，并执行其他线程。

tryTransfer方法。则是用来试探下生产者传入的元素是否能直接传给消费者。如果没有消费者等待接收元素，则返回false。和transfer方法的区别是tryTransfer方法无论消费者是否接收，方法立即返回。而transfer方法是必须等到消费者消费了才返回。

对于带有时间限制的tryTransfer(E e, long timeout, TimeUnit unit)方法，则是试图把生产者传入的元素直接传给消费者，但是如果没有消费者消费该元素则等待指定的时间再返回，如果超时还没消费元素，则返回false，如果在超时时间内消费了元素，则返回true。

LinkedBlockingDeque是一个由链表结构组成的双向阻塞队列。所谓双向队列指的你可以从队列的两端插入和移出元素。双端队列因为多了一个操作队列的入口，在多线程同时入队时，也就减少了一半的竞争。相比其他的阻塞队列，LinkedBlockingDeque多了addFirst，addLast，offerFirst，offerLast，peekFirst，peekLast等方法，以First单词结尾的方法，表示插入，获取（peek）或移除双端队列的第一个元素。以Last单词结尾的方法，表示插入，获取或移除双端队列的最后一个元素。另外插入方法add等同于addLast，移除方法remove等效于removeFirst。但是take方法却等同于takeFirst，不知道是不是Jdk的bug，使用时还是用带有First和Last后缀的方法更清楚。

在初始化LinkedBlockingDeque时可以设置容量防止其过渡膨胀。另外双向阻塞队列可以运用在“工作窃取”模式中。

## 3. 阻塞队列的实现原理

如果队列是空的，消费者会一直等待，当生产者添加元素时候，消费者是如何知道当前队列有元素的呢？如果让你来设计阻塞队列你会如何设计，让生产者和消费者能够高效率的进行通讯呢？让我们先来看看JDK是如何实现的。

使用通知模式实现。所谓通知模式，就是当生产者往满的队列里添加元素时会阻塞住生产者，当消费者消费了一个队列中的元素后，会通知生产者当前队列可用。通过查看JDK源码发现ArrayBlockingQueue使用了Condition来实现，代码如下：

private final Condition notFull;

private final Condition notEmpty;

public ArrayBlockingQueue(int capacity, boolean fair) {

//省略其他代码

notEmpty = lock.newCondition();

notFull = lock.newCondition();

}

public void put(E e) throws InterruptedException {

checkNotNull(e);

final ReentrantLock lock = this.lock;

lock.lockInterruptibly();

try {

while (count == items.length)

notFull.await();

insert(e);

} finally {

lock.unlock();

}

}

public E take() throws InterruptedException {

final ReentrantLock lock = this.lock;

lock.lockInterruptibly();

try {

while (count == 0)

notEmpty.await();

return extract();

} finally {

lock.unlock();

}

}

private void insert(E x) {

items[putIndex] = x;

putIndex = inc(putIndex);

++count;

notEmpty.signal();

}

当我们往队列里插入一个元素时，如果队列不可用，阻塞生产者主要通过LockSupport.park(this);来实现

public final void await() throws InterruptedException {

if (Thread.interrupted())

throw new InterruptedException();

Node node = addConditionWaiter();

int savedState = fullyRelease(node);

int interruptMode = 0;

while (!isOnSyncQueue(node)) {

LockSupport.park(this);

if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)

break;

}

if (acquireQueued(node, savedState) && interruptMode != THROW\_IE)

interruptMode = REINTERRUPT;

if (node.nextWaiter != null) // clean up if cancelled

unlinkCancelledWaiters();

if (interruptMode != 0)

reportInterruptAfterWait(interruptMode);

}

继续进入源码，发现调用setBlocker先保存下将要阻塞的线程，然后调用unsafe.park阻塞当前线程。

public static void park(Object blocker) {

Thread t = Thread.currentThread();

setBlocker(t, blocker);

unsafe.park(false, 0L);

setBlocker(t, null);

}

unsafe.park是个native方法，代码如下：

public native void park(boolean isAbsolute, long time);

park这个方法会阻塞当前线程，只有以下四种情况中的一种发生时，该方法才会返回。

* 与park对应的unpark执行或已经执行时。注意：已经执行是指unpark先执行，然后再执行的park。
* 线程被中断时。
* 如果参数中的time不是零，等待了指定的毫秒数时。
* 发生异常现象时。这些异常事先无法确定。

我们继续看一下JVM是如何实现park方法的，park在不同的操作系统使用不同的方式实现，在linux下是使用的是系统方法pthread\_cond\_wait实现。实现代码在JVM源码路径src/os/linux/vm/os\_linux.cpp里的 os::PlatformEvent::park方法，代码如下：

void os::PlatformEvent::park() {

int v ;

for (;;) {

v = \_Event ;

if (Atomic::cmpxchg (v-1, &\_Event, v) == v) break ;

}

guarantee (v >= 0, "invariant") ;

if (v == 0) {

// Do this the hard way by blocking ...

int status = pthread\_mutex\_lock(\_mutex);

assert\_status(status == 0, status, "mutex\_lock");

guarantee (\_nParked == 0, "invariant") ;

++ \_nParked ;

while (\_Event < 0) {

status = pthread\_cond\_wait(\_cond, \_mutex);

// for some reason, under 2.7 lwp\_cond\_wait() may return ETIME ...

// Treat this the same as if the wait was interrupted

if (status == ETIME) { status = EINTR; }

assert\_status(status == 0 || status == EINTR, status, "cond\_wait");

}

-- \_nParked ;

// In theory we could move the ST of 0 into \_Event past the unlock(),

// but then we'd need a MEMBAR after the ST.

\_Event = 0 ;

status = pthread\_mutex\_unlock(\_mutex);

assert\_status(status == 0, status, "mutex\_unlock");

}

guarantee (\_Event >= 0, "invariant") ;

}

}

pthread\_cond\_wait是一个多线程的条件变量函数，cond是condition的缩写，字面意思可以理解为线程在等待一个条件发生，这个条件是一个全局变量。这个方法接收两个参数，一个共享变量\_cond，一个互斥量\_mutex。而unpark方法在linux下是使用pthread\_cond\_signal实现的。park 在windows下则是使用WaitForSingleObject实现的。

当队列满时，生产者往阻塞队列里插入一个元素，生产者线程会进入WAITING (parking)状态。我们可以使用jstack dump阻塞的生产者线程看到这点：

"main" prio=5 tid=0x00007fc83c000000 nid=0x10164e000 waiting on condition [0x000000010164d000]

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

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

- parking to wait for <0x0000000140559fe8> (a java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject)

at java.util.concurrent.locks.LockSupport.park(LockSupport.java:186)

at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:2043)

at java.util.concurrent.ArrayBlockingQueue.put(ArrayBlockingQueue.java:324)

at blockingqueue.ArrayBlockingQueueTest.main(ArrayBlockingQueueTest.java:11)

## 4. 参考资料

* [JDK6.0阻塞队列API文档](http://www.cjsdn.net/doc/jdk60/java/util/concurrent/BlockingQueue.html)
* JDK1.7源码
* [JVM Park的windows实现](http://hg.openjdk.java.net/hsx/hotspot-main/hotspot/file/61b82be3b1ff/src/os/windows/vm/os_windows.cpp)
* [JVM Park的linux实现代码](http://hg.openjdk.java.net/hsx/hotspot-main/hotspot/file/61b82be3b1ff/src/os/linux/vm/os_linux.cpp)

## 5. 作者介绍

**方腾飞**，花名清英，并发编程网站站长。目前在阿里巴巴微贷事业部工作。并发编程网：[http://ifeve.com](http://ifeve.com/)，个人微博：<http://weibo.com/kirals>，欢迎通过我的微博进行技术交流。

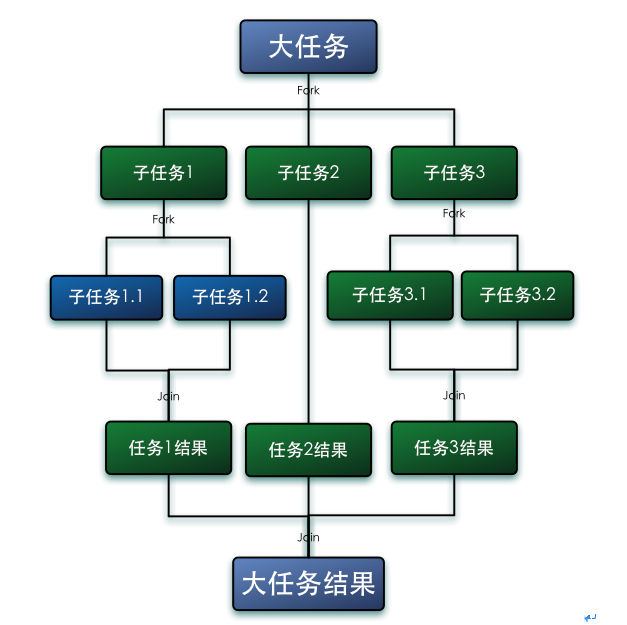
**聊聊并发（八）——Fork/Join框架介绍**

<http://www.infoq.com/cn/articles/fork-join-introduction>

## 1. 什么是Fork/Join框架

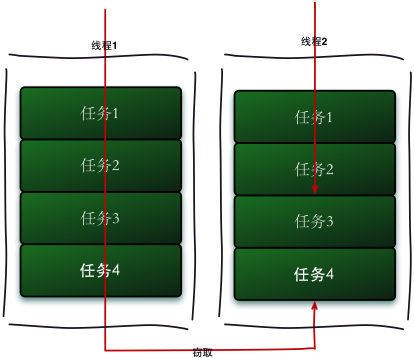
Fork/Join框架是Java7提供了的一个用于并行执行任务的框架， 是一个把大任务分割成若干个小任务，最终汇总每个小任务结果后得到大任务结果的框架。

我们再通过Fork和Join这两个单词来理解下Fork/Join框架，Fork就是把一个大任务切分为若干子任务并行的执行，Join就是合并这些子任务的执行结果，最后得到这个大任务的结果。比如计算1+2+。。＋10000，可以分割成10个子任务，每个子任务分别对1000个数进行求和，最终汇总这10个子任务的结果。Fork/Join的运行流程图如下：



## 2. 工作窃取算法

工作窃取（work-stealing）算法是指某个线程从其他队列里窃取任务来执行。工作窃取的运行流程图如下：



那么为什么需要使用工作窃取算法呢？假如我们需要做一个比较大的任务，我们可以把这个任务分割为若干互不依赖的子任务，为了减少线程间的竞争，于是把这些子任务分别放到不同的队列里，并为每个队列创建一个单独的线程来执行队列里的任务，线程和队列一一对应，比如A线程负责处理A队列里的任务。但是有的线程会先把自己队列里的任务干完，而其他线程对应的队列里还有任务等待处理。干完活的线程与其等着，不如去帮其他线程干活，于是它就去其他线程的队列里窃取一个任务来执行。而在这时它们会访问同一个队列，所以为了减少窃取任务线程和被窃取任务线程之间的竞争，通常会使用双端队列，被窃取任务线程永远从双端队列的头部拿任务执行，而窃取任务的线程永远从双端队列的尾部拿任务执行。

工作窃取算法的优点是充分利用线程进行并行计算，并减少了线程间的竞争，其缺点是在某些情况下还是存在竞争，比如双端队列里只有一个任务时。并且消耗了更多的系统资源，比如创建多个线程和多个双端队列。

## 3. Fork/Join框架的介绍

我们已经很清楚Fork/Join框架的需求了，那么我们可以思考一下，如果让我们来设计一个Fork/Join框架，该如何设计？这个思考有助于你理解Fork/Join框架的设计。

第一步分割任务。首先我们需要有一个fork类来把大任务分割成子任务，有可能子任务还是很大，所以还需要不停的分割，直到分割出的子任务足够小。

第二步执行任务并合并结果。分割的子任务分别放在双端队列里，然后几个启动线程分别从双端队列里获取任务执行。子任务执行完的结果都统一放在一个队列里，启动一个线程从队列里拿数据，然后合并这些数据。

Fork/Join使用两个类来完成以上两件事情：

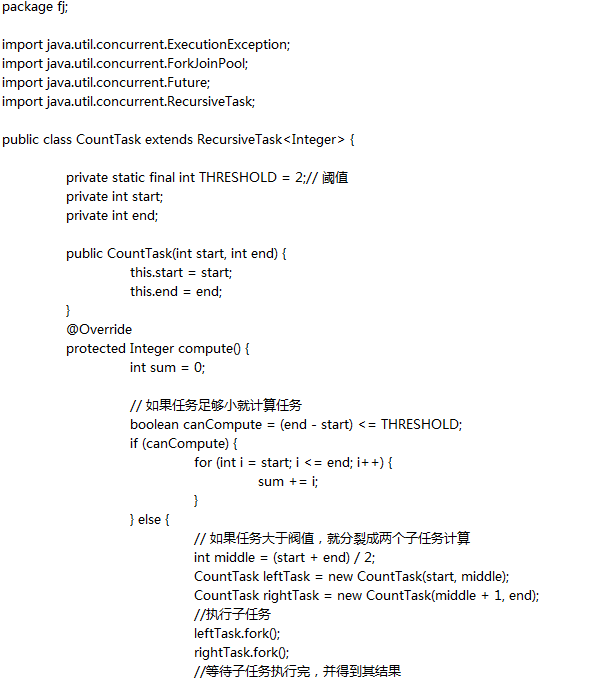
* ForkJoinTask：我们要使用ForkJoin框架，必须首先创建一个ForkJoin任务。它提供在任务中执行fork()和join()操作的机制，通常情况下我们不需要直接继承ForkJoinTask类，而只需要继承它的子类，Fork/Join框架提供了以下两个子类：
  + RecursiveAction：用于没有返回结果的任务。
  + RecursiveTask ：用于有返回结果的任务。
* ForkJoinPool ：ForkJoinTask需要通过ForkJoinPool来执行，任务分割出的子任务会添加到当前工作线程所维护的双端队列中，进入队列的头部。当一个工作线程的队列里暂时没有任务时，它会随机从其他工作线程的队列的尾部获取一个任务。

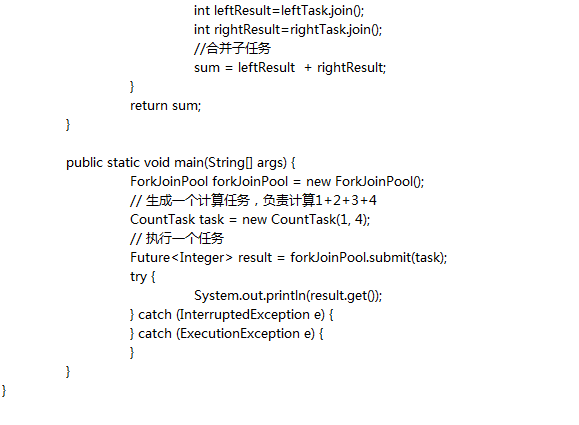
## 4. 使用Fork/Join框架

让我们通过一个简单的需求来使用下Fork／Join框架，需求是：计算1+2+3+4的结果。

使用Fork／Join框架首先要考虑到的是如何分割任务，如果我们希望每个子任务最多执行两个数的相加，那么我们设置分割的阈值是2，由于是4个数字相加，所以Fork／Join框架会把这个任务fork成两个子任务，子任务一负责计算1+2，子任务二负责计算3+4，然后再join两个子任务的结果。

因为是有结果的任务，所以必须继承RecursiveTask，实现代码如下：





通过这个例子让我们再来进一步了解ForkJoinTask，ForkJoinTask与一般的任务的主要区别在于它需要实现compute方法，在这个方法里，首先需要判断任务是否足够小，如果足够小就直接执行任务。如果不足够小，就必须分割成两个子任务，每个子任务在调用fork方法时，又会进入compute方法，看看当前子任务是否需要继续分割成孙任务，如果不需要继续分割，则执行当前子任务并返回结果。使用join方法会等待子任务执行完并得到其结果。

## 5. Fork/Join框架的异常处理

ForkJoinTask在执行的时候可能会抛出异常，但是我们没办法在主线程里直接捕获异常，所以ForkJoinTask提供了isCompletedAbnormally()方法来检查任务是否已经抛出异常或已经被取消了，并且可以通过ForkJoinTask的getException方法获取异常。使用如下代码：

if(task.isCompletedAbnormally())

{

System.out.println(task.getException());

}

getException方法返回Throwable对象，如果任务被取消了则返回CancellationException。如果任务没有完成或者没有抛出异常则返回null。

## 6. Fork/Join框架的实现原理

ForkJoinPool由ForkJoinTask数组和ForkJoinWorkerThread数组组成，ForkJoinTask数组负责存放程序提交给ForkJoinPool的任务，而ForkJoinWorkerThread数组负责执行这些任务。

ForkJoinTask的fork方法实现原理。当我们调用ForkJoinTask的fork方法时，程序会调用ForkJoinWorkerThread的pushTask方法异步的执行这个任务，然后立即返回结果。代码如下：

public final ForkJoinTask fork() { ((ForkJoinWorkerThread) Thread.currentThread()) .pushTask(this); return this; }

pushTask方法把当前任务存放在ForkJoinTask 数组queue里。然后再调用ForkJoinPool的signalWork()方法唤醒或创建一个工作线程来执行任务。代码如下：

final void pushTask(ForkJoinTask t) {

ForkJoinTask[] q; int s, m;

if ((q = queue) != null) { // ignore if queue removed

long u = (((s = queueTop) & (m = q.length - 1)) << ASHIFT) + ABASE;

UNSAFE.putOrderedObject(q, u, t);

queueTop = s + 1; // or use putOrderedInt

if ((s -= queueBase) <= 2)

pool.signalWork();

else if (s == m)

growQueue();

}

}

ForkJoinTask的join方法实现原理。Join方法的主要作用是阻塞当前线程并等待获取结果。让我们一起看看ForkJoinTask的join方法的实现，代码如下：

public final V join() {

if (doJoin() != NORMAL)

return reportResult();

else

return getRawResult();

}

private V reportResult() {

int s; Throwable ex;

if ((s = status) == CANCELLED)

throw new CancellationException();

if (s == EXCEPTIONAL && (ex = getThrowableException()) != null)

UNSAFE.throwException(ex);

return getRawResult();

}

首先，它调用了doJoin()方法，通过doJoin()方法得到当前任务的状态来判断返回什么结果，任务状态有四种：已完成（NORMAL），被取消（CANCELLED），信号（SIGNAL）和出现异常（EXCEPTIONAL）。

* 如果任务状态是已完成，则直接返回任务结果。
* 如果任务状态是被取消，则直接抛出CancellationException。
* 如果任务状态是抛出异常，则直接抛出对应的异常。

让我们再来分析下doJoin()方法的实现代码：

private int doJoin() {

Thread t; ForkJoinWorkerThread w; int s; boolean completed;

if ((t = Thread.currentThread()) instanceof ForkJoinWorkerThread) {

if ((s = status) < 0)

return s;

if ((w = (ForkJoinWorkerThread)t).unpushTask(this)) {

try {

completed = exec();

} catch (Throwable rex) {

return setExceptionalCompletion(rex);

}

if (completed)

return setCompletion(NORMAL);

}

return w.joinTask(this);

}

else

return externalAwaitDone();

}

在doJoin()方法里，首先通过查看任务的状态，看任务是否已经执行完了，如果执行完了，则直接返回任务状态，如果没有执行完，则从任务数组里取出任务并执行。如果任务顺利执行完成了，则设置任务状态为NORMAL，如果出现异常，则纪录异常，并将任务状态设置为EXCEPTIONAL。

## 7. 参考资料

* JDK1.7源码
* http://ifeve.com/fork-join-5/

## 8. 作者介绍

**方腾飞**，花名清英，并发编程网站站长。目前在阿里巴巴微贷事业部工作。并发编程网：[http://ifeve.com](http://ifeve.com/)，个人微博：<http://weibo.com/kirals>，欢迎通过我的微博进行技术交流。

**聊聊并发-Java中的Copy-On-Write容器**

Copy-On-Write简称COW，是一种用于程序设计中的优化策略。其基本思路是，从一开始大家都在共享同一个内容，当某个人想要修改这个内容的时候，才会真正把内容Copy出去形成一个新的内容然后再改，这是一种延时懒惰策略。从JDK1.5开始Java并发包里提供了两个使用CopyOnWrite机制实现的并发容器,它们是CopyOnWriteArrayList和CopyOnWriteArraySet。CopyOnWrite容器非常有用，可以在非常多的并发场景中使用到。

#### 什么是CopyOnWrite容器

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

#### CopyOnWriteArrayList的实现原理

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

|  |  |
| --- | --- |
| 01 | public boolean add(T e) { |
| 02 | final ReentrantLock lock = this.lock; | |

|  |  |  |
| --- | --- | --- |
| 03 | lock.lock(); | |
| 04 | try { |

|  |  |
| --- | --- |
| 05 |  |
| 06 | Object[] elements = getArray(); | |

|  |  |
| --- | --- |
| 07 |  |
| 08 | int len = elements.length; | |

|  |  |  |
| --- | --- | --- |
| 09 | // 复制出新数组 | |
| 10 |  |

|  |  |  |
| --- | --- | --- |
| 11 | Object[] newElements = Arrays.copyOf(elements, len + 1); | |
| 12 | // 把新元素添加到新数组里 |

|  |  |
| --- | --- |
| 13 |  |
| 14 | newElements[len] = e; | |

|  |  |  |
| --- | --- | --- |
| 15 | // 把原数组引用指向新数组 | |
| 16 |  |

|  |  |  |
| --- | --- | --- |
| 17 | setArray(newElements); | |
| 18 |  |

|  |  |  |
| --- | --- | --- |
| 19 | return true; | |
| 20 |  |

|  |  |  |
| --- | --- | --- |
| 21 | } finally { | |
| 22 |  |

|  |  |  |
| --- | --- | --- |
| 23 | lock.unlock(); | |
| 24 |  |

|  |  |  |
| --- | --- | --- |
| 25 | } | |
| 26 |  |

|  |  |
| --- | --- |
| 27 | } |
| 28 |  |

|  |  |  |
| --- | --- | --- |
| 29 | final void setArray(Object[] a) { | |
| 30 | array = a; |

|  |  |
| --- | --- |
| 31 | } |

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

|  |  |
| --- | --- |
| 1 | public E get(int index) { |
| 2 | return get(getArray(), index); | |

|  |  |
| --- | --- |
| 3 | } |

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

|  |  |  |
| --- | --- | --- |
| 01 | import java.util.Collection; | |
| 02 | import java.util.Map; |

|  |  |  |
| --- | --- | --- |
| 03 | import java.util.Set; | |
| 04 |  |

|  |  |  |
| --- | --- | --- |
| 05 | public class CopyOnWriteMap<K, V> implements Map<K, V>, Cloneable { | |
| 06 | private volatile Map<K, V> internalMap; |

|  |  |
| --- | --- |
| 07 |  |
| 08 | public CopyOnWriteMap() { | |

|  |  |  |
| --- | --- | --- |
| 09 | internalMap = new HashMap<K, V>(); | |
| 10 | } |

|  |  |
| --- | --- |
| 11 |  |
| 12 | public V put(K key, V value) { | |

|  |  |
| --- | --- |
| 13 |  |
| 14 | synchronized (this) { | |

|  |  |  |
| --- | --- | --- |
| 15 | Map<K, V> newMap = new HashMap<K, V>(internalMap); | |
| 16 | V val = newMap.put(key, value); |

|  |  |  |
| --- | --- | --- |
| 17 | internalMap = newMap; | |
| 18 | return val; |

|  |  |  |
| --- | --- | --- |
| 19 | } | |
| 20 | } |

|  |  |
| --- | --- |
| 21 |  |
| 22 | public V get(Object key) { | |

|  |  |  |
| --- | --- | --- |
| 23 | return internalMap.get(key); | |
| 24 | } |

|  |  |
| --- | --- |
| 25 |  |
| 26 | public void putAll(Map<? extends K, ? extends V> newData) { | |

|  |  |
| --- | --- |
| 27 | synchronized (this) { |
| 28 | Map<K, V> newMap = new HashMap<K, V>(internalMap); | |

|  |  |  |
| --- | --- | --- |
| 29 | newMap.putAll(newData); | |
| 30 | internalMap = newMap; |

|  |  |  |
| --- | --- | --- |
| 31 | } | |
| 32 | } |

|  |  |
| --- | --- |
| 33 | } |

实现很简单，只要了解了CopyOnWrite机制，我们可以实现各种CopyOnWrite容器，并且在不同的应用场景中使用。

#### CopyOnWrite的应用场景

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

|  |  |  |
| --- | --- | --- |
| 01 | package com.ifeve.book; | |
| 02 |  |

|  |  |  |
| --- | --- | --- |
| 03 | import java.util.Map; | |
| 04 |  |

|  |  |  |
| --- | --- | --- |
| 05 | import com.ifeve.book.forkjoin.CopyOnWriteMap; | |
| 06 |  |

|  |  |
| --- | --- |
| 07 | /\*\* |
| 08 | \* 黑名单服务 | |

|  |  |
| --- | --- |
| 09 | \* |
| 10 | \* @author fangtengfei | |

|  |  |
| --- | --- |
| 11 | \* |
| 12 | \*/ | |

|  |  |  |
| --- | --- | --- |
| 13 | public class BlackListServiceImpl { | |
| 14 |  |

|  |  |  |
| --- | --- | --- |
| 15 | private static CopyOnWriteMap<String, Boolean> blackListMap = newCopyOnWriteMap<String, Boolean>( | |
| 16 | 1000); |

|  |  |
| --- | --- |
| 17 |  |
| 18 | public static boolean isBlackList(String id) { | |

|  |  |  |
| --- | --- | --- |
| 19 | return blackListMap.get(id) == null ? false : true; | |
| 20 | } |

|  |  |
| --- | --- |
| 21 |  |
| 22 | public static void addBlackList(String id) { | |

|  |  |  |
| --- | --- | --- |
| 23 | blackListMap.put(id, Boolean.TRUE); | |
| 24 | } |

|  |  |
| --- | --- |
| 25 |  |
| 26 | /\*\* | |

|  |  |  |
| --- | --- | --- |
| 27 | \* 批量添加黑名单 | |
| 28 | \* |

|  |  |  |
| --- | --- | --- |
| 29 | \* @param ids | |
| 30 | \*/ |

|  |  |  |
| --- | --- | --- |
| 31 | public static void addBlackList(Map<String,Boolean> ids) { | |
| 32 | blackListMap.putAll(ids); |

|  |  |  |
| --- | --- | --- |
| 33 | } | |
| 34 |  |

|  |  |
| --- | --- |
| 35 | } |

代码很简单，但是使用CopyOnWriteMap需要注意两件事情：

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

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

#### CopyOnWrite的缺点

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

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

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

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

关于C++的STL中，曾经也有过Copy-On-Write的玩法，参见陈皓的《[C++ STL String类中的Copy-On-Write](http://blog.csdn.net/haoel/article/details/24058)》，后来，因为有很多线程安全上的事，就被去掉了。

**聊聊并发——生产者消费者模式**

<http://www.infoq.com/cn/articles/producers-and-consumers-mode>

在并发编程中使用生产者和消费者模式能够解决绝大多数并发问题。该模式通过平衡生产线程和消费线程的工作能力来提高程序的整体处理数据的速度。

## 为什么要使用生产者和消费者模式

**在线程世界里，生产者就是生产数据的线程，消费者就是消费数据的线程。**在多线程开发当中，如果生产者处理速度很快，而消费者处理速度很慢，那么生产者就必须等待消费者处理完，才能继续生产数据。同样的道理，如果消费者的处理能力大于生产者，那么消费者就必须等待生产者。为了解决这个问题于是引入了生产者和消费者模式。

## 什么是生产者消费者模式

生产者消费者模式是通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者彼此之间不直接通讯，而通过阻塞队列来进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。

这个阻塞队列就是用来给生产者和消费者解耦的。纵观大多数设计模式，都会找一个第三者出来进行解耦，如工厂模式的第三者是工厂类，模板模式的第三者是模板类。在学习一些设计模式的过程中，如果先找到这个模式的第三者，能帮助我们快速熟悉一个设计模式。

## 生产者消费者模式实战

我和同事一起利用业余时间开发的Yuna工具中使用了生产者和消费者模式。首先我先介绍下Yuna工具，在阿里巴巴很多同事都喜欢通过邮件分享技术文章，因为通过邮件分享很方便，同学们在网上看到好的技术文章，复制粘贴发送就完成了分享，但是我们发现技术文章不能沉淀下来，对于新来的同学看不到以前分享的技术文章，大家也很难找到以前分享过的技术文章。为了解决这问题，我们开发了Yuna工具。Yuna取名自我喜欢的一款游戏最终幻想里的女主角。

首先我们申请了一个专门用来收集分享邮件的邮箱，比如share@alibaba.com，同学将分享的文章发送到这个邮箱，让同学们每次都抄送到这个邮箱肯定很麻烦，所以我们的做法是将这个邮箱地址放在部门邮件列表里，所以分享的同学只需要象以前一样向整个部门分享文章就行，Yuna工具通过读取邮件服务器里该邮箱的邮件，把所有分享的邮件下载下来，包括邮件的附件，图片，和邮件回复，我们可能会从这个邮箱里下载到一些非分享的文章，所以我们要求分享的邮件标题必须带有一个关键字，比如［内贸技术分享］，下载完邮件之后，通过confluence的web service接口，把文章插入到confluence里，这样新同事就可以在confluence里看以前分享过的文章，并且Yuna工具还可以自动把文章进行分类和归档。

为了快速上线该功能，当时我们花了三天业余时间快速开发了Yuna1.0版本。在1.0版本中我并没有使用生产者消费模式，而是使用单线程来处理，因为当时只需要处理我们一个部门的邮件，所以单线程明显够用，整个过程是串行执行的。在一个线程里，程序先抽取全部的邮件，转化为文章对象，然后添加全部的文章，最后删除抽取过的邮件。代码如下：

public void extract() {

logger.debug("开始" + getExtractorName() + "。。");

//抽取邮件

List<Article> articles = extractEmail();

//添加文章

for (Article article : articles) {

addArticleOrComment(article);

}

//清空邮件

cleanEmail();

logger.debug("完成" + getExtractorName() + "。。");

}

Yuna工具在推广后，越来越多的部门使用这个工具，处理的时间越来越慢，Yuna是每隔5分钟进行一次抽取的，而当邮件多的时候一次处理可能就花了几分钟，于是我在Yuna2.0版本里使用了生产者消费者模式来处理邮件，首先生产者线程按一定的规则去邮件系统里抽取邮件，然后存放在阻塞队列里，消费者从阻塞队列里取出文章后插入到conflunce里。代码如下：

public class QuickEmailToWikiExtractor extends AbstractExtractor {

private ThreadPoolExecutor threadsPool;

private ArticleBlockingQueue<ExchangeEmailShallowDTO> emailQueue;

public QuickEmailToWikiExtractor() {

emailQueue= new ArticleBlockingQueue<ExchangeEmailShallowDTO>();

int corePoolSize = Runtime.getRuntime().availableProcessors() \* 2;

threadsPool = new ThreadPoolExecutor(corePoolSize, corePoolSize, 10l, TimeUnit.SECONDS,

new LinkedBlockingQueue<Runnable>(2000));

}

public void extract() {

logger.debug("开始" + getExtractorName() + "。。");

long start = System.currentTimeMillis();

//抽取所有邮件放到队列里

new ExtractEmailTask().start();

// 把队列里的文章插入到Wiki

insertToWiki();

long end = System.currentTimeMillis();

double cost = (end - start) / 1000;

logger.debug("完成" + getExtractorName() + ",花费时间：" + cost + "秒");

}

/\*\*

\* 把队列里的文章插入到Wiki

\*/

private void insertToWiki() {

//登录wiki,每间隔一段时间需要登录一次

confluenceService.login(RuleFactory.USER\_NAME, RuleFactory.PASSWORD);

while (true) {

//2秒内取不到就退出

ExchangeEmailShallowDTO email = emailQueue.poll(2, TimeUnit.SECONDS);

if (email == null) {

break;

}

threadsPool.submit(new insertToWikiTask(email));

}

}

protected List<Article> extractEmail() {

List<ExchangeEmailShallowDTO> allEmails = getEmailService().queryAllEmails();

if (allEmails == null) {

return null;

}

for (ExchangeEmailShallowDTO exchangeEmailShallowDTO : allEmails) {

emailQueue.offer(exchangeEmailShallowDTO);

}

return null;

}

/\*\*

\* 抽取邮件任务

\*

\* @author tengfei.fangtf

\*/

public class ExtractEmailTask extends Thread {

public void run() {

extractEmail();

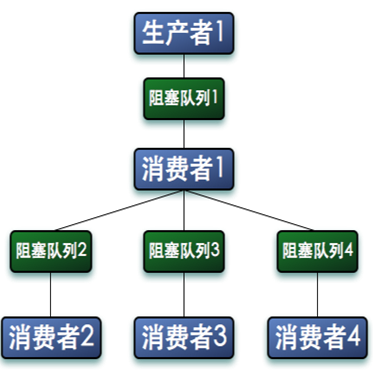
}

}

}

## 多生产者和多消费者场景

在多核时代，多线程并发处理速度比单线程处理速度更快，所以我们可以使用多个线程来生产数据，同样可以使用多个消费线程来消费数据。而更复杂的情况是，消费者消费的数据，有可能需要继续处理，于是消费者处理完数据之后，它又要作为生产者把数据放在新的队列里，交给其他消费者继续处理。如下图：



我们在一个长连接服务器中使用了这种模式，生产者1负责将所有客户端发送的消息存放在阻塞队列1里，消费者1从队列里读消息，然后通过消息ID进行hash得到N个队列中的一个，然后根据编号将消息存放在到不同的队列里，每个阻塞队列会分配一个线程来消费阻塞队列里的数据。如果消费者2无法消费消息，就将消息再抛回到阻塞队列1中，交给其他消费者处理。

以下是消息总队列的代码；

/\*\*

\* 总消息队列管理

\*

\* @author tengfei.fangtf

\*/

public class MsgQueueManager implements IMsgQueue{

private static final Logger LOGGER

= LoggerFactory.getLogger(MsgQueueManager.class);

/\*\*

\* 消息总队列

\*/

public final BlockingQueue<Message> messageQueue;

private MsgQueueManager() {

messageQueue = new LinkedTransferQueue<Message>();

}

public void put(Message msg) {

try {

messageQueue.put(msg);

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

public Message take() {

try {

return messageQueue.take();

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

return null;

}

}

启动一个消息分发线程。在这个线程里子队列自动去总队列里获取消息。

/\*\*

\* 分发消息，负责把消息从大队列塞到小队列里

\*

\* @author tengfei.fangtf

\*/

static class DispatchMessageTask implements Runnable {

@Override

public void run() {

BlockingQueue<Message> subQueue;

for (;;) {

//如果没有数据，则阻塞在这里

Message msg = MsgQueueFactory.getMessageQueue().take();

//如果为空，则表示没有Session机器连接上来，

需要等待，直到有Session机器连接上来

while ((subQueue = getInstance().getSubQueue()) == null) {

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

//把消息放到小队列里

try {

subQueue.put(msg);

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

}

}

使用Hash算法获取一个子队列。

/\*\*

\* 均衡获取一个子队列。

\*

\* @return

\*/

public BlockingQueue<Message> getSubQueue() {

int errorCount = 0;

for (;;) {

if (subMsgQueues.isEmpty()) {

return null;

}

int index = (int) (System.nanoTime() % subMsgQueues.size());

try {

return subMsgQueues.get(index);

} catch (Exception e) {

//出现错误表示，在获取队列大小之后，队列进行了一次删除操作

LOGGER.error("获取子队列出现错误", e);

if ((++errorCount) < 3) {

continue;

}

}

}

}

使用的时候我们只需要往总队列里发消息。

//往消息队列里添加一条消息

IMsgQueue messageQueue = MsgQueueFactory.getMessageQueue();

Packet msg = Packet.createPacket(Packet64FrameType.

TYPE\_DATA, "{}".getBytes(), (short) 1);

messageQueue.put(msg);

## 小结

本章讲解了生产者消费者模式，并给出了实例。读者可以在平时的工作中思考下哪些场景可以使用生产者消费者模式，我相信这种场景应该非常之多，特别是需要处理任务时间比较长的场景，比如上传附件并处理，用户把文件上传到系统后，系统把文件丢到队列里，然后立刻返回告诉用户上传成功，最后消费者再去队列里取出文件处理。比如调用一个远程接口查询数据，如果远程服务接口查询时需要几十秒的时间，那么它可以提供一个申请查询的接口，这个接口把要申请查询任务放数据库中，然后该接口立刻返回。然后服务器端用线程轮询并获取申请任务进行处理，处理完之后发消息给调用方，让调用方再来调用另外一个接口拿数据。

另外Java中的线程池类其实就是一种生产者和消费者模式的实现方式，但是实现方法更高明。生产者把任务丢给线程池，线程池创建线程并处理任务，如果将要运行的任务数大于线程池的基本线程数就把任务扔到阻塞队列里，这种做法比只使用一个阻塞队列来实现生产者和消费者模式显然要高明很多，因为消费者能够处理直接就处理掉了，这样速度更快，而生产者先存，消费者再取这种方式显然慢一些。

我们的系统也可以使用线程池来实现多生产者消费者模式。比如创建N个不同规模的Java线程池来处理不同性质的任务，比如线程池1将数据读到内存之后，交给线程池2里的线程继续处理压缩数据。线程池1主要处理IO密集型任务，线程池2主要处理CPU密集型任务。