# 并发关键字

目录

[并发关键字 1](#_Toc47762481)

[volatile 2](#_Toc47762482)

[volatile简介 2](#_Toc47762483)

[volatile实现原理 2](#_Toc47762484)

[volatile的happens-before关系 3](#_Toc47762485)

[volatile的内存语义 4](#_Toc47762486)

[volatile和synchronized的区别与联系 8](#_Toc47762487)

[final 8](#_Toc47762488)

[final的简介 8](#_Toc47762489)

[final的具体使用场景 9](#_Toc47762490)

[final关键字举例 13](#_Toc47762491)

[final的实现原理 20](#_Toc47762492)

[何谓悲观锁与乐观锁 21](#_Toc47762493)

[悲观锁 21](#_Toc47762494)

[乐观锁 22](#_Toc47762495)

[乐观锁常见的两种实现方式 22](#_Toc47762496)

[乐观锁的缺点 23](#_Toc47762497)

[CAS与synchronized的使用情景 23](#_Toc47762498)

[AQS 24](#_Toc47762499)

[AQS 简单介绍 24](#_Toc47762500)

[AQS 原理 25](#_Toc47762501)

[AQS 对资源的共享方式 26](#_Toc47762502)

[Exclusive（独占） 26](#_Toc47762503)

[Share（共享） 29](#_Toc47762504)

[AQS 底层使用了模板方法模式 29](#_Toc47762505)

[Semaphore(信号量)-允许多个线程同时访问 30](#_Toc47762506)

[CountDownLatch （倒计时器） 31](#_Toc47762507)

[CountDownLatch 的三种典型用法 32](#_Toc47762508)

[CountDownLatch 的不足 33](#_Toc47762509)

[CyclicBarrier(循环栅栏) 33](#_Toc47762510)

[CyclicBarrier 的应用场景 34](#_Toc47762511)

[CyclicBarrier 和 CountDownLatch 的区别 39](#_Toc47762512)

[并发容器 39](#_Toc47762513)

[JDK 提供的并发容器总结 39](#_Toc47762514)

[ConcurrentHashMap 40](#_Toc47762515)

[CopyOnWriteArrayList 40](#_Toc47762516)

[BlockingQueue 42](#_Toc47762517)

[BlockingQueue 简单介绍 42](#_Toc47762518)

[ArrayBlockingQueue 42](#_Toc47762519)

[PriorityBlockingQueue 43](#_Toc47762520)

[ConcurrentSkipListMap 43](#_Toc47762521)

## volatile

### volatile简介

在上一篇文章中我们深入理解了Java关键字-synchronized，我们知道在java中还有一大神器就是关键volatile，可以说是和synchronized各领风骚，其中奥妙，我们来共同探讨下。

通过上一篇的文章我们了解到synchronized是阻塞式同步，在线程竞争激烈的情况下会升级为重量级锁。而volatile就可以说是Java虚拟机提供的最轻量级的同步机制。但它同时不容易被正确理解，也至于在并发编程中很多程序员遇到线程安全的问题就会使用synchronized。Java内存模型告诉我们，各个线程会将共享变量从主内存中拷贝到工作内存，然后执行引擎会基于工作内存中的数据进行操作处理。线程在工作内存进行操作后何时会写到主内存中？这个时机对普通变量是没有规定的，而针对volatile修饰的变量给Java虚拟机特殊的约定，线程对volatile变量的修改会立刻被其他线程所感知，即不会出现数据脏读的现象，从而保证数据的“可见性”。

现在我们有了一个大概的印象就是：被volatile修饰的变量能够保证每个线程能够获取该变量的最新值，从而避免出现数据脏读的现象

### volatile实现原理

volatile是怎样实现了？比如一个很简单的Java代码：

instance = new Instancce() //instance是volatile变量

在生成汇编代码时会在volatile修饰的共享变量进行写操作的时候会多出Lock前缀的指令（具体的大家可以使用一些工具去看一下，这里我就只把结果说出来）。我们想这个Lock指令肯定有神奇的地方，那么Lock前缀的指令在多核处理器下会发现什么事情了？主要有这两个方面的影响：

将当前处理器缓存行的数据写回系统内存；

这个写回内存的操作会使得其他CPU里缓存了该内存地址的数据无效

为了提高处理速度，处理器不直接和内存进行通信，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，但操作完不知道何时会写到内存。如果对声明了volatile的变量进行写操作，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，当处理器对这个数据进行修改操作的时候，会重新从系统内存中把数据读到处理器缓存里。因此，经过分析我们可以得出如下结论：

Lock前缀的指令会引起处理器缓存写回内存；

一个处理器的缓存回写到内存会导致其他处理器的缓存失效；

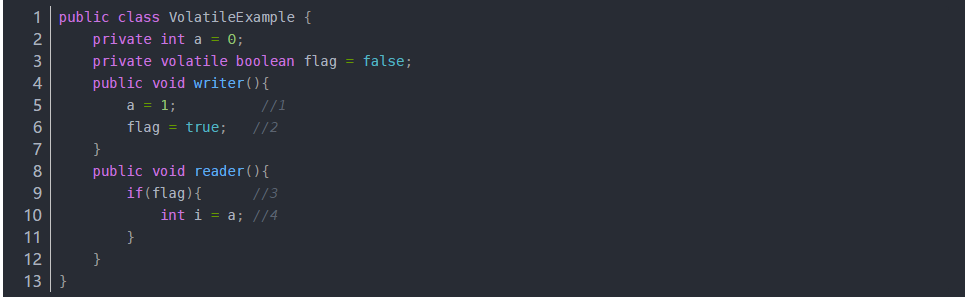
当处理器发现本地缓存失效后，就会从内存中重读该变量数据，即可以获取当前最新值。

这样针对volatile变量通过这样的机制就使得每个线程都能获得该变量的最新值。

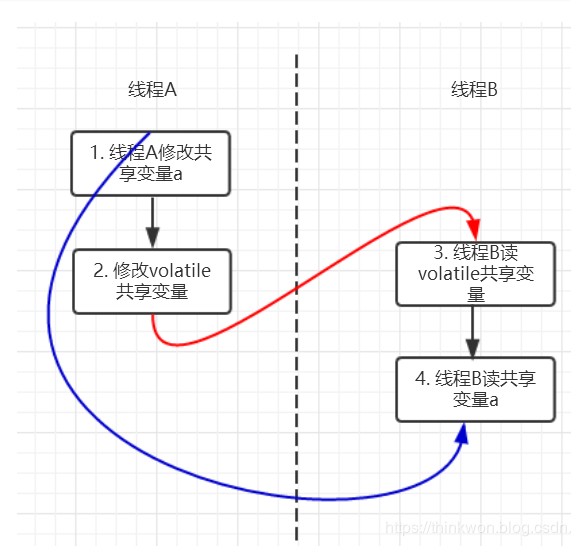
### volatile的happens-before关系

经过上面的分析，我们已经知道了volatile变量可以通过缓存一致性协议保证每个线程都能获得最新值，即满足数据的“可见性”。我们继续延续上一篇博客分析问题的方式（我一直认为思考问题的方式是属于自己，也才是最重要的，也在不断培养这方面的能力），我一直将并发分析的切入点分为两个核心，三大性质。两大核心：JMM内存模型（主内存和工作内存）以及happens-before；三条性质：原子性，可见性，有序性（关于三大性质的总结在以后得文章会和大家共同探讨）。废话不多说，先来看两个核心之一：volatile的happens-before关系。

在六条happens-before规则中有一条是：volatile变量规则：对一个volatile域的写，happens-before于任意后续对这个volatile域的读。下面我们结合具体的代码，我们利用这条规则推导下：



上面的实例代码对应的happens-before关系如下图所示：

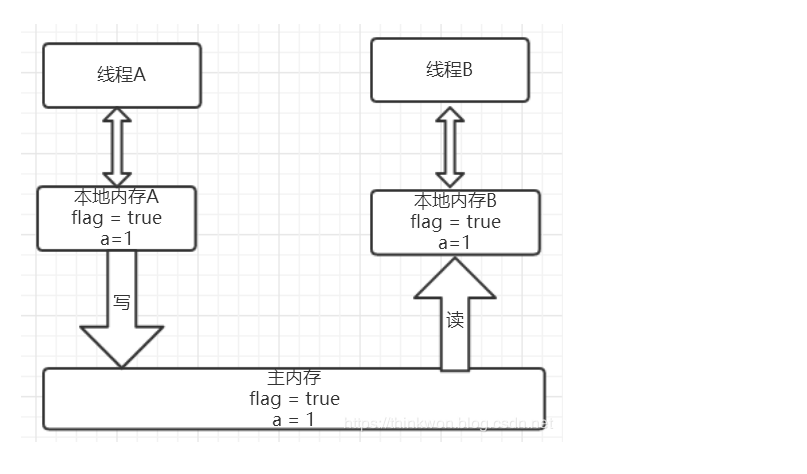


加锁线程A先执行writer方法，然后线程B执行reader方法，图中每一个箭头两个节点就代码一个happens-before关系，黑色的代表根据程序顺序规则推导出来，红色的是根据volatile变量的写happens-before 于任意后续对volatile变量的读，而蓝色的就是根据传递性规则推导出来的。这里的2 happen-before 3，同样根据happens-before规则定义：如果A happens-before B，则A的执行结果对B可见，并且A的执行顺序先于B的执行顺序，我们可以知道操作2执行结果对操作3来说是可见的，也就是说当线程A将volatile变量 flag更改为true后线程B就能够迅速感知。

### volatile的内存语义

还是按照两个核心的分析方式，分析完happens-before关系后我们现在就来进一步分析volatile的内存语义（按照这种方式去学习，会不会让大家对知识能够把握的更深，而不至于不知所措，如果大家认同我的这种方式，不妨给个赞，小弟在此谢过，对我是个鼓励）。还是以上面的代码为例，假设线程A先执行writer方法，线程B随后执行reader方法，初始时线程的本地内存中flag和a都是初始状态，下图是线程A执行volatile写后的状态图。

当volatile变量写后，线程中本地内存中共享变量就会置为失效的状态，因此线程B再需要读取从主内存中去读取该变量的最新值。下图就展示了线程B读取同一个volatile变量的内存变化示意图。



从横向来看，线程A和线程B之间进行了一次通信，线程A在写volatile变量时，实际上就像是给B发送了一个消息告诉线程B你现在的值都是旧的了，然后线程B读这个volatile变量时就像是接收了线程A刚刚发送的消息。既然是旧的了，那线程B该怎么办了？自然而然就只能去主内存去取啦。

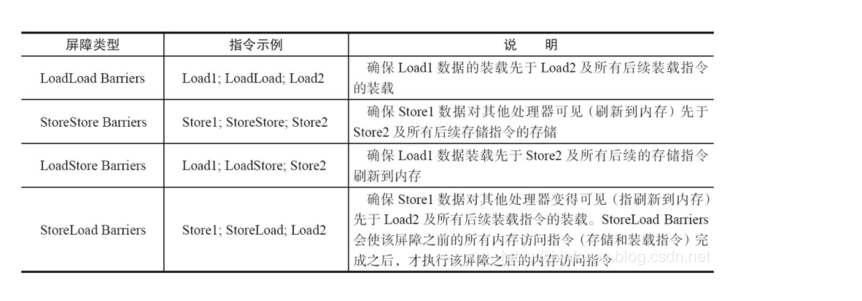
好的，我们现在两个核心：happens-before以及内存语义现在已经都了解清楚了。是不是还不过瘾，突然发现原来自己会这么爱学习（微笑脸），那我们下面就再来一点干货----volatile内存语义的实现。

volatile的内存语义实现

我们都知道，为了性能优化，JMM在不改变正确语义的前提下，会允许编译器和处理器对指令序列进行重排序，那如果想阻止重排序要怎么办了？答案是可以添加内存屏障。

内存屏障

JMM内存屏障分为四类见下图，



Java编译器会在生成指令系列时在适当的位置会插入内存屏障指令来禁止特定类型的处理器重排序。为了实现volatile的内存语义，JMM会限制特定类型的编译器和处理器重排序，JMM会针对编译器制定volatile重排序规则表：



"NO"表示禁止重排序。为了实现volatile内存语义时，编译器在生成字节码时，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序。对于编译器来说，发现一个最优布置来最小化插入屏障的总数几乎是不可能的，为此，JMM采取了保守策略：

在每个volatile写操作的前面插入一个StoreStore屏障；

在每个volatile写操作的后面插入一个StoreLoad屏障；

在每个volatile读操作的后面插入一个LoadLoad屏障；

在每个volatile读操作的后面插入一个LoadStore屏障。

需要注意的是：volatile写是在前面和后面分别插入内存屏障，而volatile读操作是在后面插入两个内存屏障

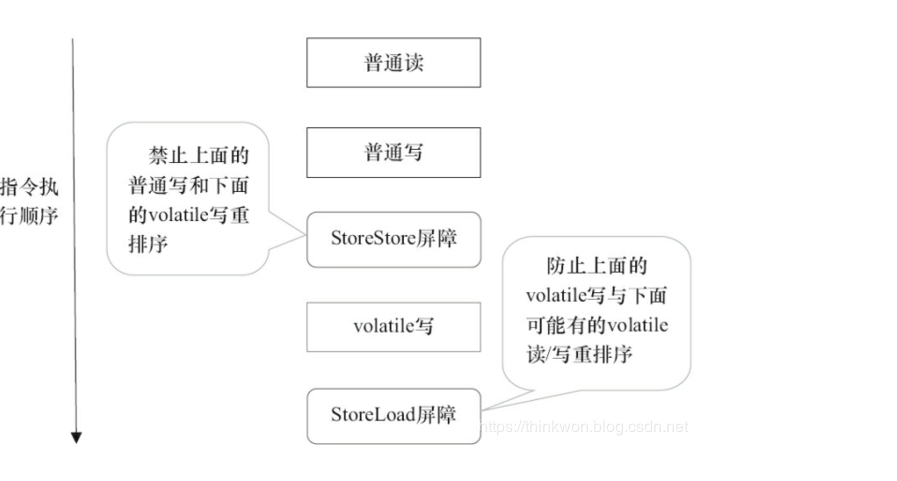
StoreStore屏障：禁止上面的普通写和下面的volatile写重排序；

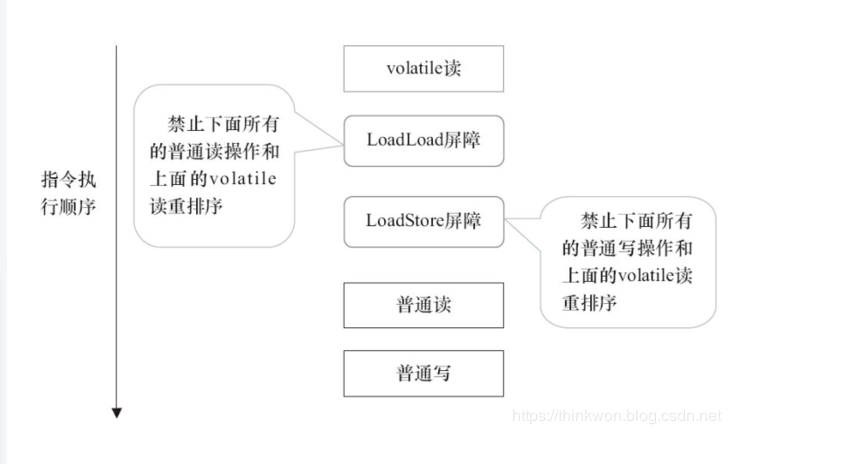
StoreLoad屏障：防止上面的volatile写与下面可能有的volatile读/写重排序

LoadLoad屏障：禁止下面所有的普通读操作和上面的volatile读重排序

LoadStore屏障：禁止下面所有的普通写操作和上面的volatile读重排序

下面以两个示意图进行理解，图片摘自相当好的一本书《Java并发编程的艺术》。





一个示例

我们现在已经理解volatile的精华了，文章开头的那个问题我想现在我们都能给出答案了。更正后的代码为：



注意不同点，现在已经将isOver设置成了volatile变量，这样在main线程中将isOver改为了true后，thread的工作内存该变量值就会失效，从而需要再次从主内存中读取该值，现在能够读出isOver最新值为true从而能够结束在thread里的死循环，从而能够顺利停止掉thread线程。现在问题也解决了，知识也学到了。（如果觉得还不错，请点赞，是对我的一个鼓励。

### volatile和synchronized的区别与联系

1.volatile本质是在告诉jvm当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取；synchronized则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住；  
2.volatile仅能使用在变量级别；synchronized则可以使用在变量、方法、和类级别的；  
3.volatile仅能实现变量的修改可见性，不能保证原子性；而synchronized则可以保证变量的修改可见性和原子性；  
4.volatile不会造成线程的阻塞；synchronized可能会造成线程的阻塞；  
5.volatile标记的变量不会被编译器优化；synchronized标记的变量可以被编译器优化。

## final

### final的简介

final可以修饰变量，方法和类，用于表示所修饰的内容一旦赋值之后就不会再被改变，比如String类就是一个final类型的类。即使能够知道final具体的使用方法，final在多线程中存在的重排序问题很容易忽略，希望能够一起做下探讨。

### final的具体使用场景

final能够修饰变量，方法和类，也就是final使用范围基本涵盖了Java每个地方，下面就分别以锁修饰的位置：变量，方法和类分别来说一说。

变量

在Java中变量，可以分为成员变量以及方法局部变量。因此也是按照这种方式依次来说，以避免漏掉任何一个死角。

final成员变量

通常每个类中的成员变量可以分为类变量（static修饰的变量）以及实例变量。针对这两种类型的变量赋初值的时机是不同的，类变量可以在声明变量的时候直接赋初值或者在静态代码块中给类变量赋初值。而实例变量可以在声明变量的时候给实例变量赋初值，在非静态初始化块中以及构造器中赋初值。类变量有两个时机赋初值，而实例变量则可以有三个时机赋初值。当final变量未初始化时系统不会进行隐式初始化，会出现报错。这样说起来还是比较抽象，下面用具体的代码来演示。（代码涵盖了final修饰变量所有的可能情况，耐心看下去会有收获的）



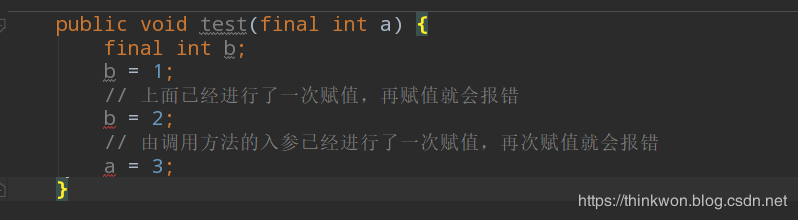
看上面的图片已经将每种情况整理出来了，这里用截图的方式也是觉得在IDE出现红色出错的标记更能清晰的说明情况。现在我们来将这几种情况归纳整理一下：

类变量：必须要在静态初始化块中指定初始值或者声明该类变量时指定初始值，而且只能在这两个地方之一进行指定；

实例变量：必要要在非静态初始化块，声明该实例变量或者在构造器中指定初始值，而且只能在这三个地方进行指定。

final局部变量

final局部变量由程序员进行显式初始化，如果final局部变量已经进行了初始化则后面就不能再次进行更改，如果final变量未进行初始化，可以进行赋值，当且仅有一次赋值，一旦赋值之后再次赋值就会出错。下面用具体的代码演示final局部变量的情况：



现在我们来换一个角度进行考虑，final修饰的是基本数据类型和引用类型有区别吗？

final基本数据类型 VS final引用数据类型

通过上面的例子我们已经看出来，如果final修饰的是一个基本数据类型的数据，一旦赋值后就不能再次更改，那么，如果final是引用数据类型了？这个引用的对象能够改变吗？我们同样来看一段代码。



当我们对final修饰的引用数据类型变量person的属性改成22，是可以成功操作的。通过这个实验我们就可以看出来当final修饰基本数据类型变量时，不能对基本数据类型变量重新赋值，因此基本数据类型变量不能被改变。而对于引用类型变量而言，它仅仅保存的是一个引用，final只保证这个引用类型变量所引用的地址不会发生改变，即一直引用这个对象，但这个对象属性是可以改变的。

宏变量

利用final变量的不可更改性，在满足一下三个条件时，该变量就会成为一个“宏变量”，即是一个常量。

使用final修饰符修饰；

在定义该final变量时就指定了初始值；

该初始值在编译时就能够唯一指定。

注意：当程序中其他地方使用该宏变量的地方，编译器会直接替换成该变量的值

方法

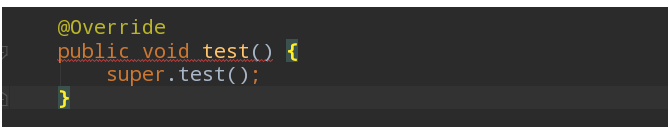
重写？

当父类的方法被final修饰的时候，子类不能重写父类的该方法，比如在Object中，getClass()方法就是final的，我们就不能重写该方法，但是hashCode()方法就不是被final所修饰的，我们就可以重写hashCode()方法。我们还是来写一个例子来加深一下理解：

先定义一个父类，里面有final修饰的方法test();



然后FinalExample继承该父类FinalExampleParent，当重写test()方法时出现报错



通过这个现象我们就可以看出来被final修饰的方法不能够被子类所重写。

重载？



可以看出被final修饰的方法是可以重载的。经过我们的分析可以得出如下结论：

1. 父类的final方法是不能够被子类重写的

2. final方法是可以被重载的

类

当一个类被final修饰时，表名该类是不能被子类继承的。子类继承往往可以重写父类的方法和改变父类属性，会带来一定的安全隐患，因此，当一个类不希望被继承时就可以使用final修饰。还是来写一个小例子：

public final class FinalExampleParent {

public final void test() {

}

}

父类会被final修饰，当子类继承该父类的时候，就会报错，如下图：

### final关键字举例

final经常会被用作不变类上，利用final的不可更改性。我们先来看看什么是不变类。

不变类

不变类的意思是创建该类的实例后，该实例的实例变量是不可改变的。满足以下条件则可以成为不可变类：

使用private和final修饰符来修饰该类的成员变量；

提供带参的构造器用于初始化类的成员变量；

仅为该类的成员变量提供getter方法，不提供setter方法，因为普通方法无法修改fina修饰的成员变量；

如果有必要就重写Object类 的hashCode()和equals()方法，应该保证用equals()判断相同的两个对象其Hashcode值也是相等的。

JDK中提供的八个包装类和String类都是不可变类，我们来看看String的实现。

/\*\* The value is used for character storage. \*/

private final char value[];

可以看出String的value就是final修饰的，上述其他几条性质也是吻合的。

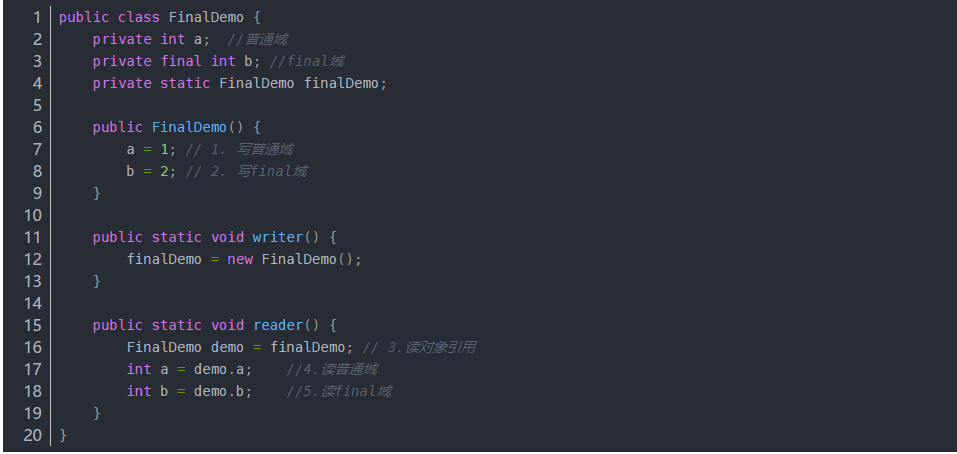
多线程中你真的了解final吗

上面我们聊的final使用，应该属于Java基础层面的，当理解这些后我们就真的算是掌握了final吗？有考虑过final在多线程并发的情况吗？在Java内存模型中我们知道Java内存模型为了能让处理器和编译器底层发挥他们的最大优势，对底层的约束就很少，也就是说针对底层来说Java内存模型就是一弱内存数据模型。同时，处理器和编译为了性能优化会对指令序列有编译器和处理器重排序。那么，在多线程情况下，final会进行怎样的重排序？会导致线程安全的问题吗？下面，就来看看final的重排序。

final域重排序规则

final域为基本类型

先看一段示例性的代码：



假设线程A在执行writer()方法，线程B执行reader()方法。

写final域重排序规则

写final域的重排序规则禁止对final域的写重排序到构造函数之外，这个规则的实现主要包含了两个方面：

JMM禁止编译器把final域的写重排序到构造函数之外；

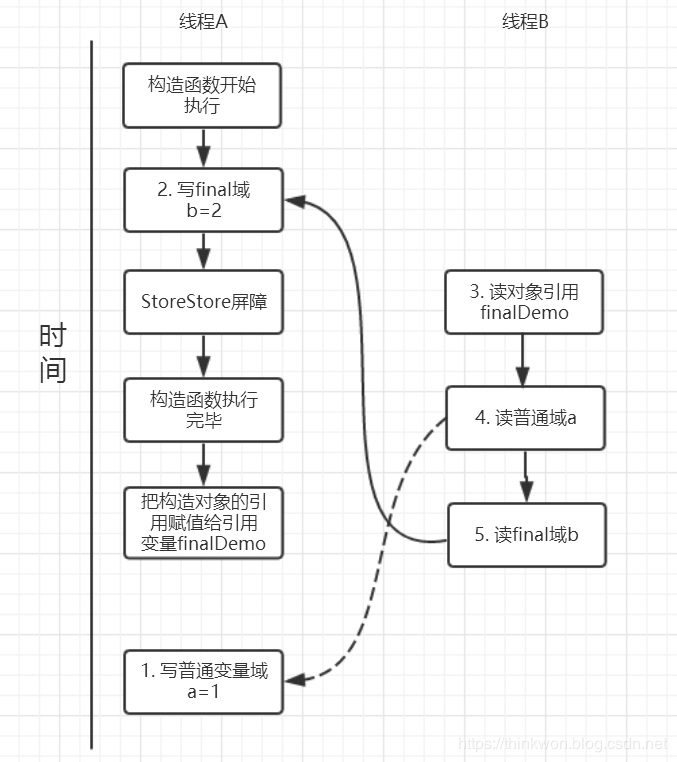
编译器会在final域写之后，构造函数return之前，插入一个storestore屏障（关于内存屏障可以看这篇文章）。这个屏障可以禁止处理器把final域的写重排序到构造函数之外。

我们再来分析writer方法，虽然只有一行代码，但实际上做了两件事情：

构造了一个FinalDemo对象；

把这个对象赋值给成员变量finalDemo。

我们来画下存在的一种可能执行时序图，如下：



由于a,b之间没有数据依赖性，普通域（普通变量）a可能会被重排序到构造函数之外，线程B就有可能读到的是普通变量a初始化之前的值（零值），这样就可能出现错误。而final域变量b，根据重排序规则，会禁止final修饰的变量b重排序到构造函数之外，从而b能够正确赋值，线程B就能够读到final变量初始化后的值。

因此，写final域的重排序规则可以确保：在对象引用为任意线程可见之前，对象的final域已经被正确初始化过了，而普通域就不具有这个保障。比如在上例，线程B有可能就是一个未正确初始化的对象finalDemo。

读final域重排序规则

读final域重排序规则为：在一个线程中，初次读对象引用和初次读该对象包含的final域，JMM会禁止这两个操作的重排序。（注意，这个规则仅仅是针对处理器），处理器会在读final域操作的前面插入一个LoadLoad屏障。实际上，读对象的引用和读该对象的final域存在间接依赖性，一般处理器不会重排序这两个操作。但是有一些处理器会重排序，因此，这条禁止重排序规则就是针对这些处理器而设定的。

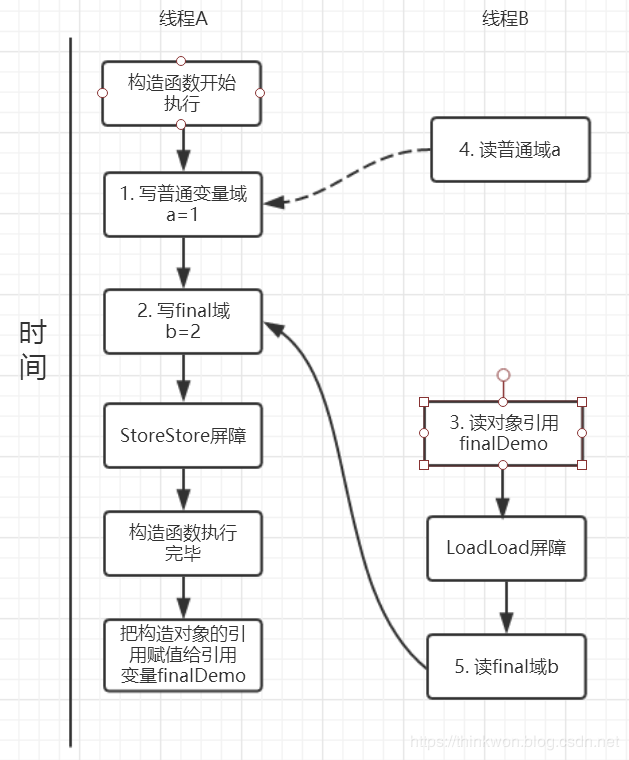
read()方法主要包含了三个操作：

初次读引用变量finalDemo;

初次读引用变量finalDemo的普通域a;

初次读引用变量finalDemo的final与b;

假设线程A写过程没有重排序，那么线程A和线程B有一种的可能执行时序为下图：



读对象的普通域被重排序到了读对象引用的前面就会出现线程B还未读到对象引用就在读取该对象的普通域变量，这显然是错误的操作。而final域的读操作就“限定”了在读final域变量前已经读到了该对象的引用，从而就可以避免这种情况。

读final域的重排序规则可以确保：在读一个对象的final域之前，一定会先读这个包含这个final域的对象的引用。

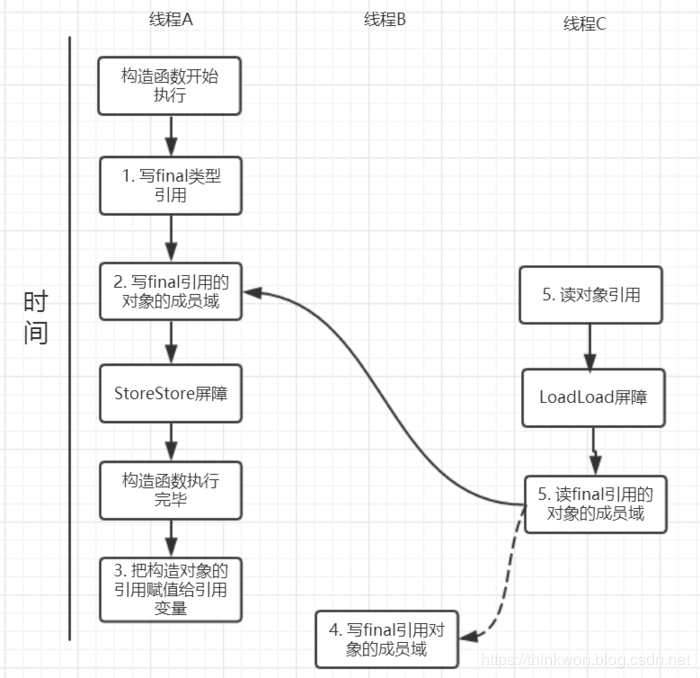
final域为引用类型

我们已经知道了final域是基本数据类型的时候重排序规则是怎么的了？如果是引用数据类型了？我们接着继续来探讨。

对final修饰的对象的成员域写操作

针对引用数据类型，final域写针对编译器和处理器重排序增加了这样的约束：在构造函数内对一个final修饰的对象的成员域的写入，与随后在构造函数之外把这个被构造的对象的引用赋给一个引用变量，这两个操作是不能被重排序的。注意这里的是“增加”也就说前面对final基本数据类型的重排序规则在这里还是使用。这句话是比较拗口的，下面结合实例来看。

针对上面的实例程序，线程线程A执行wirterOne方法，执行完后线程B执行writerTwo方法，然后线程C执行reader方法。下图就以这种执行时序出现的一种情况来讨论（耐心看完才有收获）。



由于对final域的写禁止重排序到构造方法外，因此1和3不能被重排序。由于一个final域的引用对象的成员域写入不能与随后将这个被构造出来的对象赋给引用变量重排序，因此2和3不能重排序。

对final修饰的对象的成员域读操作

JMM可以确保线程C至少能看到写线程A对final引用的对象的成员域的写入，即能看下arrays[0] = 1，而写线程B对数组元素的写入可能看到可能看不到。JMM不保证线程B的写入对线程C可见，线程B和线程C之间存在数据竞争，此时的结果是不可预知的。如果可见的，可使用锁或者volatile。

关于final重排序的总结

按照final修饰的数据类型分类：

基本数据类型:

final域写：禁止final域写与构造方法重排序，即禁止final域写重排序到构造方法之外，从而保证该对象对所有线程可见时，该对象的final域全部已经初始化过。

final域读：禁止初次读对象的引用与读该对象包含的final域的重排序。

引用数据类型：

额外增加约束：禁止在构造函数对一个final修饰的对象的成员域的写入与随后将这个被构造的对象的引用赋值给引用变量 重排序

### final的实现原理

上面我们提到过，写final域会要求编译器在final域写之后，构造函数返回前插入一个StoreStore屏障。读final域的重排序规则会要求编译器在读final域的操作前插入一个LoadLoad屏障。

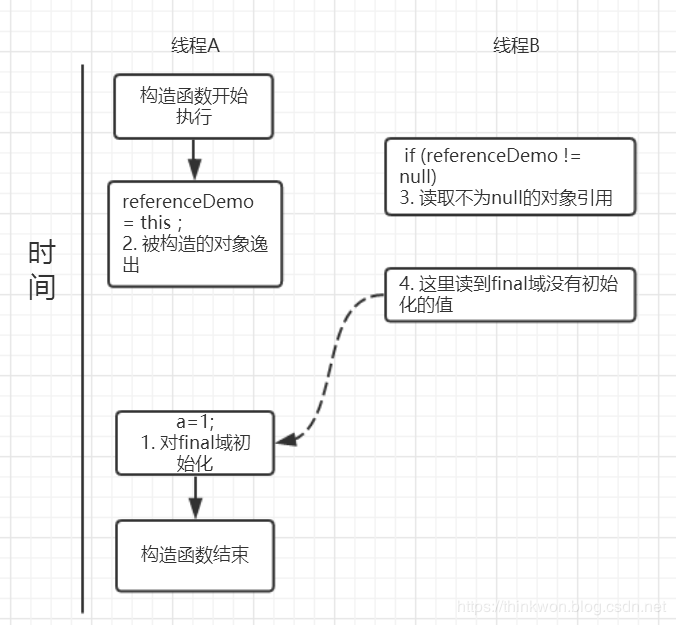
很有意思的是，如果以X86处理为例，X86不会对写-写重排序，所以StoreStore屏障可以省略。由于不会对有间接依赖性的操作重排序，所以在X86处理器中，读final域需要的LoadLoad屏障也会被省略掉。也就是说，以X86为例的话，对final域的读/写的内存屏障都会被省略！具体是否插入还是得看是什么处理器

为什么final引用不能从构造函数中“溢出”

这里还有一个比较有意思的问题：上面对final域写重排序规则可以确保我们在使用一个对象引用的时候该对象的final域已经在构造函数被初始化过了。但是这里其实是有一个前提条件的，也就是：在构造函数，不能让这个被构造的对象被其他线程可见，也就是说该对象引用不能在构造函数中“溢出”。以下面的例子来说：



可能的执行时序如图所示：



假设一个线程A执行writer方法另一个线程执行reader方法。因为构造函数中操作1和2之间没有数据依赖性，1和2可以重排序，先执行了2，这个时候引用对象referenceDemo是个没有完全初始化的对象，而当线程B去读取该对象时就会出错。尽管依然满足了final域写重排序规则：在引用对象对所有线程可见时，其final域已经完全初始化成功。但是，引用对象“this”溢出，该代码依然存在线程安全的问题。

# 何谓悲观锁与乐观锁

乐观锁对应于生活中乐观的人总是想着事情往好的方向发展，悲观锁对应于生活中悲观的人总是想着事情往坏的方向发展。这两种人各有优缺点，不能不以场景而定说一种人好于另外一种人。

## 悲观锁

总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁（共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程）。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。Java中synchronized和ReentrantLock等独占锁就是悲观锁思想的实现。

## 乐观锁

总是假设最好的情况，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号机制和CAS算法实现。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于write\_condition机制，其实都是提供的乐观锁。在Java中java.util.concurrent.atomic包下面的原子变量类就是使用了乐观锁的一种实现方式CAS实现的。

两种锁的使用场景

从上面对两种锁的介绍，我们知道两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下（多读场景），即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果是多写的情况，一般会经常产生冲突，这就会导致上层应用会不断的进行retry，这样反倒是降低了性能，所以一般多写的场景下用悲观锁就比较合适。

### 乐观锁常见的两种实现方式

乐观锁一般会使用版本号机制或CAS算法实现。

#### 1. 版本号机制

一般是在数据表中加上一个数据版本号version字段，表示数据被修改的次数，当数据被修改时，version值会加一。当线程A要更新数据值时，在读取数据的同时也会读取version值，在提交更新时，若刚才读取到的version值为当前数据库中的version值相等时才更新，否则重试更新操作，直到更新成功。

**举一个简单的例子：** 假设数据库中帐户信息表中有一个 version 字段，当前值为 1 ；而当前帐户余额字段（ balance ）为 $100 。

1. 操作员 A 此时将其读出（ version=1 ），并从其帐户余额中扣除 $50（ $100-$50 ）。
2. 在操作员 A 操作的过程中，操作员B 也读入此用户信息（ version=1 ），并从其帐户余额中扣除 $20 （ $100-$20 ）。
3. 操作员 A 完成了修改工作，将数据版本号加一（ version=2 ），连同帐户扣除后余额（ balance=$50 ），提交至数据库更新，此时由于提交数据版本大于数据库记录当前版本，数据被更新，数据库记录 version 更新为 2 。
4. 操作员 B 完成了操作，也将版本号加一（ version=2 ）试图向数据库提交数据（ balance=$80 ），但此时比对数据库记录版本时发现，操作员 B 提交的数据版本号为 2 ，数据库记录当前版本也为 2 ，不满足 “ 提交版本必须大于记录当前版本才能执行更新 “ 的乐观锁策略，因此，操作员 B 的提交被驳回。

这样，就避免了操作员 B 用基于 version=1 的旧数据修改的结果覆盖操作员A 的操作结果的可能。

#### 2. CAS算法

即**compare and swap（比较与交换）**，是一种有名的**无锁算法**。无锁编程，即不使用锁的情况下实现多线程之间的变量同步，也就是在没有线程被阻塞的情况下实现变量的同步，所以也叫非阻塞同步（Non-blocking Synchronization）。**CAS算法**涉及到三个操作数

* 需要读写的内存值 V
* 进行比较的值 A
* 拟写入的新值 B

当且仅当 V 的值等于 A时，CAS通过原子方式用新值B来更新V的值，否则不会执行任何操作（比较和替换是一个原子操作）。一般情况下是一个**自旋操作**，即**不断的重试**。

关于自旋锁，大家可以看一下这篇文章，非常不错：[《 面试必备之深入理解自旋锁》](https://blog.csdn.net/qq_34337272/article/details/81252853)

### 乐观锁的缺点

ABA 问题是乐观锁一个常见的问题

**1 ABA 问题**

如果一个变量V初次读取的时候是A值，并且在准备赋值的时候检查到它仍然是A值，那我们就能说明它的值没有被其他线程修改过了吗？很明显是不能的，因为在这段时间它的值可能被改为其他值，然后又改回A，那CAS操作就会误认为它从来没有被修改过。这个问题被称为CAS操作的 **"ABA"问题。**

JDK 1.5 以后的 AtomicStampedReference 类就提供了此种能力，其中的 compareAndSet 方法就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。

**2 循环时间长开销大**

自旋CAS（也就是不成功就一直循环执行直到成功）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。

**3 只能保证一个共享变量的原子操作**

CAS 只对单个共享变量有效，当操作涉及跨多个共享变量时 CAS 无效。但是从 JDK 1.5开始，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.所以我们可以使用锁或者利用AtomicReference类把多个共享变量合并成一个共享变量来操作。

### CAS与synchronized的使用情景

简单的来说CAS适用于写比较少的情况下（多读场景，冲突一般较少），synchronized适用于写比较多的情况下（多写场景，冲突一般较多）

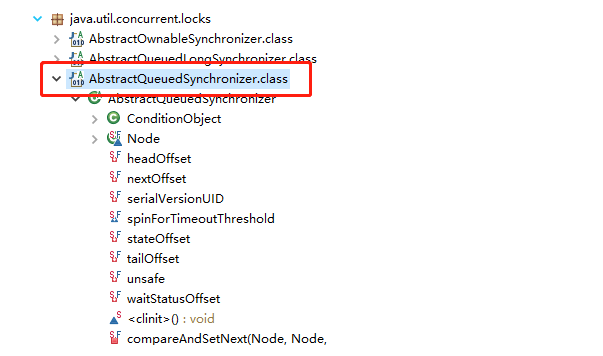
1. 对于资源竞争较少（线程冲突较轻）的情况，使用synchronized同步锁进行线程阻塞和唤醒切换以及用户态内核态间的切换操作额外浪费消耗cpu资源；而CAS基于硬件实现，不需要进入内核，不需要切换线程，操作自旋几率较少，因此可以获得更高的性能。
2. 对于资源竞争严重（线程冲突严重）的情况，CAS自旋的概率会比较大，从而浪费更多的CPU资源，效率低于synchronized。

补充： Java并发编程这个领域中synchronized关键字一直都是元老级的角色，很久之前很多人都会称它为 **“重量级锁”** 。但是，在JavaSE 1.6之后进行了主要包括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** 以及其它**各种优化**之后变得在某些情况下并不是那么重了。synchronized的底层实现主要依靠 **Lock-Free** 的队列，基本思路是 **自旋后阻塞**，**竞争切换后继续竞争锁**，**稍微牺牲了公平性，但获得了高吞吐量**。在线程冲突较少的情况下，可以获得和CAS类似的性能；而线程冲突严重的情况下，性能远高于CAS。

# AQS

## AQS 简单介绍

AQS 的全称为（AbstractQueuedSynchronizer），这个类在 java.util.concurrent.locks 包下面。



AQS 是一个用来构建锁和同步器的框架，使用 AQS 能简单且高效地构造出应用广泛的大量的同步器，比如我们提到的 ReentrantLock，Semaphore，其他的诸如 ReentrantReadWriteLock，SynchronousQueue，FutureTask(jdk1.7) 等等皆是基于 AQS 的。当然，我们自己也能利用 AQS 非常轻松容易地构造出符合我们自己需求的同步器。

## AQS 原理

在面试中被问到并发知识的时候，大多都会被问到“请你说一下自己对于 AQS 原理的理解”。下面给大家一个示例供大家参考，面试不是背题，大家一定要加入自己的思想，即使加入不了自己的思想也要保证自己能够通俗的讲出来而不是背出来。

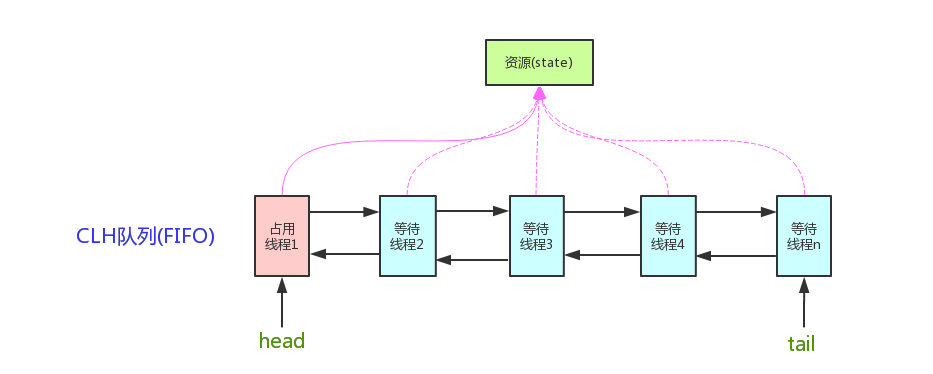
下面大部分内容其实在 AQS 类注释上已经给出了，不过是英语看着比较吃力一点，感兴趣的话可以看看源码。

AQS 原理概览

AQS 核心思想是，如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设置为锁定状态。如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个机制 AQS 是用 CLH 队列锁实现的，即将暂时获取不到锁的线程加入到队列中。

CLH(Craig,Landin,and Hagersten)队列是一个虚拟的双向队列（虚拟的双向队列即不存在队列实例，仅存在结点之间的关联关系）。AQS 是将每条请求共享资源的线程封装成一个 CLH 锁队列的一个结点（Node）来实现锁的分配。

看个 AQS(AbstractQueuedSynchronizer)原理图：



AQS 使用一个 int 成员变量来表示同步状态，通过内置的 FIFO 队列来完成获取资源线程的排队工作。AQS 使用 CAS 对该同步状态进行原子操作实现对其值的修改。



状态信息通过 protected 类型的getState，setState，compareAndSetState进行操作



## AQS 对资源的共享方式

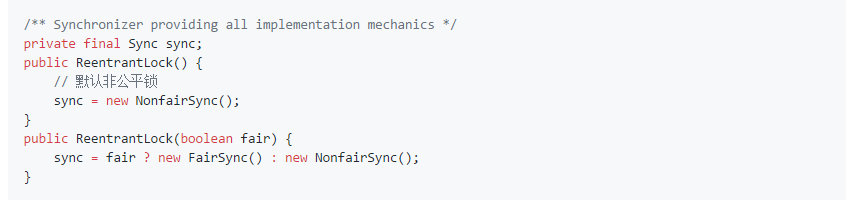
### Exclusive（独占）

只有一个线程能执行，如 ReentrantLock。又可分为公平锁和非公平锁,ReentrantLock 同时支持两种锁,下面以 ReentrantLock 对这两种锁的定义做介绍：

* 公平锁：按照线程在队列中的排队顺序，先到者先拿到锁
* 非公平锁：当线程要获取锁时，先通过两次 CAS 操作去抢锁，如果没抢到，当前线程再加入到队列中等待唤醒。

说明：下面这部分关于 ReentrantLock 源代码内容节选自：<https://www.javadoop.com/post/AbstractQueuedSynchronizer-2> ，这是一篇很不错文章，推荐阅读。

**下面来看 ReentrantLock 中相关的源代码：**

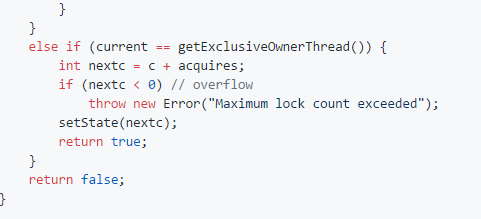
ReentrantLock 默认采用非公平锁，因为考虑获得更好的性能，通过 boolean 来决定是否用公平锁（传入 true 用公平锁）。

ReentrantLock 中公平锁的 lock 方法



非公平锁的 lock 方法：





总结：公平锁和非公平锁只有两处不同：

1. 非公平锁在调用 lock 后，首先就会调用 CAS 进行一次抢锁，如果这个时候恰巧锁没有被占用，那么直接就获取到锁返回了。
2. 非公平锁在 CAS 失败后，和公平锁一样都会进入到 tryAcquire 方法，在 tryAcquire 方法中，如果发现锁这个时候被释放了（state == 0），非公平锁会直接 CAS 抢锁，但是公平锁会判断等待队列是否有线程处于等待状态，如果有则不去抢锁，乖乖排到后面。

公平锁和非公平锁就这两点区别，如果这两次 CAS 都不成功，那么后面非公平锁和公平锁是一样的，都要进入到阻塞队列等待唤醒。

相对来说，非公平锁会有更好的性能，因为它的吞吐量比较大。当然，非公平锁让获取锁的时间变得更加不确定，可能会导致在阻塞队列中的线程长期处于饥饿状态。

### Share（共享）

多个线程可同时执行，如 Semaphore/CountDownLatch。Semaphore、CountDownLatCh、 CyclicBarrier、ReadWriteLock 我们都会在后面讲到。

ReentrantReadWriteLock 可以看成是组合式，因为 ReentrantReadWriteLock 也就是读写锁允许多个线程同时对某一资源进行读。

不同的自定义同步器争用共享资源的方式也不同。自定义同步器在实现时只需要实现共享资源 state 的获取与释放方式即可，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS 已经在上层已经帮我们实现好了。

## AQS 底层使用了模板方法模式

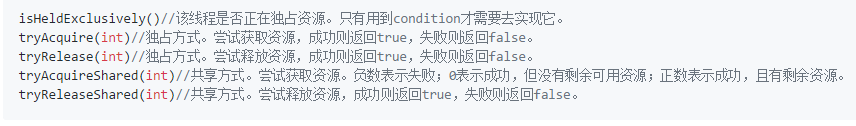
同步器的设计是基于模板方法模式的，如果需要自定义同步器一般的方式是这样（模板方法模式很经典的一个应用）：

1. 使用者继承 AbstractQueuedSynchronizer 并重写指定的方法。（这些重写方法很简单，无非是对于共享资源 state 的获取和释放）
2. 将 AQS 组合在自定义同步组件的实现中，并调用其模板方法，而这些模板方法会调用使用者重写的方法。

这和我们以往通过实现接口的方式有很大区别，这是模板方法模式很经典的一个运用，下面简单的给大家介绍一下模板方法模式，模板方法模式是一个很容易理解的设计模式之一。

模板方法模式是基于”继承“的，主要是为了在不改变模板结构的前提下在子类中重新定义模板中的内容以实现复用代码。举个很简单的例子假如我们要去一个地方的步骤是：购票buyTicket()->安检securityCheck()->乘坐某某工具回家ride()->到达目的地arrive()。我们可能乘坐不同的交通工具回家比如飞机或者火车，所以除了ride()方法，其他方法的实现几乎相同。我们可以定义一个包含了这些方法的抽象类，然后用户根据自己的需要继承该抽象类然后修改 ride()方法。

**AQS 使用了模板方法模式，自定义同步器时需要重写下面几个 AQS 提供的模板方法：**



默认情况下，每个方法都抛出 UnsupportedOperationException。 这些方法的实现必须是内部线程安全的，并且通常应该简短而不是阻塞。AQS 类中的其他方法都是 final ，所以无法被其他类使用，只有这几个方法可以被其他类使用。

以 ReentrantLock 为例，state 初始化为 0，表示未锁定状态。A 线程 lock()时，会调用 tryAcquire()独占该锁并将 state+1。此后，其他线程再 tryAcquire()时就会失败，直到 A 线程 unlock()到 state=0（即释放锁）为止，其它线程才有机会获取该锁。当然，释放锁之前，A 线程自己是可以重复获取此锁的（state 会累加），这就是可重入的概念。但要注意，获取多少次就要释放多么次，这样才能保证 state 是能回到零态的。

再以 CountDownLatch 以例，任务分为 N 个子线程去执行，state 也初始化为 N（注意 N 要与线程个数一致）。这 N 个子线程是并行执行的，每个子线程执行完后 countDown()一次，state 会 CAS(Compare and Swap)减 1。等到所有子线程都执行完后(即 state=0)，会 unpark()主调用线程，然后主调用线程就会从 await()函数返回，继续后余动作。

一般来说，自定义同步器要么是独占方法，要么是共享方式，他们也只需实现tryAcquire-tryRelease、tryAcquireShared-tryReleaseShared中的一种即可。但 AQS 也支持自定义同步器同时实现独占和共享两种方式，如ReentrantReadWriteLock。

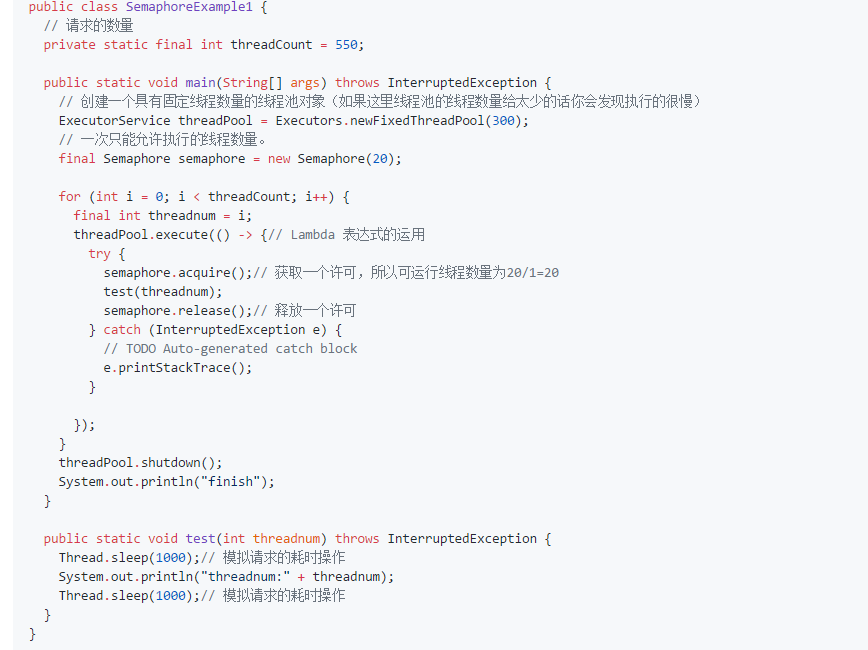
推荐两篇 AQS 原理和相关源码分析的文章：

* <http://www.cnblogs.com/waterystone/p/4920797.html>
* <https://www.cnblogs.com/chengxiao/archive/2017/07/24/7141160.html>

## Semaphore(信号量)-允许多个线程同时访问

synchronized 和 ReentrantLock 都是一次只允许一个线程访问某个资源，Semaphore(信号量)可以指定多个线程同时访问某个资源。

示例代码如下：



执行 acquire 方法阻塞，直到有一个许可证可以获得然后拿走一个许可证；每个 release 方法增加一个许可证，这可能会释放一个阻塞的 acquire 方法。然而，其实并没有实际的许可证这个对象，Semaphore 只是维持了一个可获得许可证的数量。 Semaphore 经常用于限制获取某种资源的线程数量。

当然一次也可以一次拿取和释放多个许可，不过一般没有必要这样做：

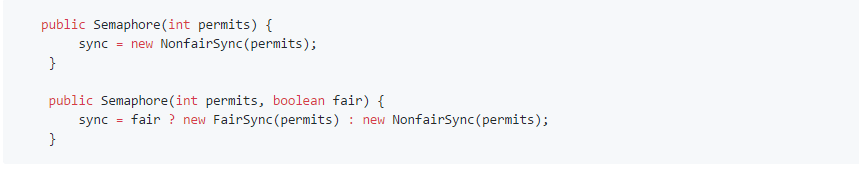


除了 acquire方法之外，另一个比较常用的与之对应的方法是tryAcquire方法，该方法如果获取不到许可就立即返回 false。

Semaphore 有两种模式，公平模式和非公平模式。

* **公平模式：** 调用 acquire 的顺序就是获取许可证的顺序，遵循 FIFO；
* **非公平模式：** 抢占式的。

Semaphore 对应的两个构造方法如下：



这两个构造方法，都必须提供许可的数量，第二个构造方法可以指定是公平模式还是非公平模式，默认非公平模式。

[issue645补充内容](https://github.com/Snailclimb/JavaGuide/issues/645) ：Semaphore与CountDownLatch一样，也是共享锁的一种实现。它默认构造AQS的state为permits。当执行任务的线程数量超出permits,那么多余的线程将会被放入阻塞队列Park,并自旋判断state是否大于0。只有当state大于0的时候，阻塞的线程才能继续执行,此时先前执行任务的线程继续执行release方法，release方法使得state的变量会加1，那么自旋的线程便会判断成功。 如此，每次只有最多不超过permits数量的线程能自旋成功，便限制了执行任务线程的数量。

由于篇幅问题，如果对 Semaphore 源码感兴趣的朋友可以看下这篇文章：<https://juejin.im/post/5ae755366fb9a07ab508adc6>

## CountDownLatch （倒计时器）

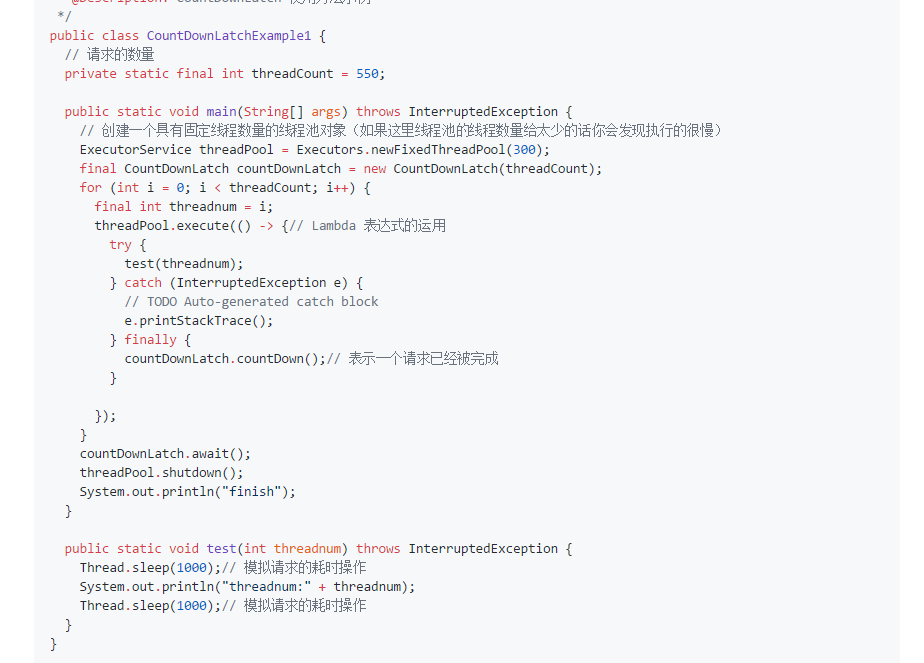
CountDownLatch允许 count 个线程阻塞在一个地方，直至所有线程的任务都执行完毕。在 Java 并发中，countdownlatch 的概念是一个常见的面试题，所以一定要确保你很好的理解了它。

CountDownLatch是共享锁的一种实现,它默认构造 AQS 的 state 值为 count。当线程使用countDown方法时,其实使用了tryReleaseShared方法以CAS的操作来减少state,直至state为0就代表所有的线程都调用了countDown方法。当调用await方法的时候，如果state不为0，就代表仍然有线程没有调用countDown方法，那么就把已经调用过countDown的线程都放入阻塞队列Park,并自旋CAS判断state == 0，直至最后一个线程调用了countDown，使得state == 0，于是阻塞的线程便判断成功，全部往下执行。

### CountDownLatch 的三种典型用法

1. 某一线程在开始运行前等待 n 个线程执行完毕。将 CountDownLatch 的计数器初始化为 n ：new CountDownLatch(n)，每当一个任务线程执行完毕，就将计数器减 1 countdownlatch.countDown()，当计数器的值变为 0 时，在CountDownLatch上 await() 的线程就会被唤醒。一个典型应用场景就是启动一个服务时，主线程需要等待多个组件加载完毕，之后再继续执行。
2. 实现多个线程开始执行任务的最大并行性。注意是并行性，不是并发，强调的是多个线程在某一时刻同时开始执行。类似于赛跑，将多个线程放到起点，等待发令枪响，然后同时开跑。做法是初始化一个共享的 CountDownLatch 对象，将其计数器初始化为 1 ：new CountDownLatch(1)，多个线程在开始执行任务前首先 coundownlatch.await()，当主线程调用 countDown() 时，计数器变为 0，多个线程同时被唤醒。
3. 死锁检测：一个非常方便的使用场景是，你可以使用 n 个线程访问共享资源，在每次测试阶段的线程数目是不同的，并尝试产生死锁。

**CountDownLatch 的使用示例**



上面的代码中，我们定义了请求的数量为 550，当这 550 个请求被处理完成之后，才会执行System.out.println("finish");。

与 CountDownLatch 的第一次交互是主线程等待其他线程。主线程必须在启动其他线程后立即调用 CountDownLatch.await() 方法。这样主线程的操作就会在这个方法上阻塞，直到其他线程完成各自的任务。

其他 N 个线程必须引用闭锁对象，因为他们需要通知 CountDownLatch 对象，他们已经完成了各自的任务。这种通知机制是通过 CountDownLatch.countDown()方法来完成的；每调用一次这个方法，在构造函数中初始化的 count 值就减 1。所以当 N 个线程都调 用了这个方法，count 的值等于 0，然后主线程就能通过 await()方法，恢复执行自己的任务。

如果对CountDownLatch源码感兴趣的朋友，可以查看： [【JUC】JDK1.8源码分析之CountDownLatch（五）](https://www.cnblogs.com/leesf456/p/5406191.html)

### CountDownLatch 的不足

CountDownLatch 是一次性的，计数器的值只能在构造方法中初始化一次，之后没有任何机制再次对其设置值，当 CountDownLatch 使用完毕后，它不能再次被使用。

**CountDownLatch 相常见面试题：**

解释一下 CountDownLatch 概念？

CountDownLatch 和 CyclicBarrier 的不同之处？

给出一些 CountDownLatch 使用的例子？

CountDownLatch 类中主要的方法？

## CyclicBarrier(循环栅栏)

CyclicBarrier 和 CountDownLatch 非常类似，它也可以实现线程间的技术等待，但是它的功能比 CountDownLatch 更加复杂和强大。主要应用场景和 CountDownLatch 类似。

CountDownLatch的实现是基于AQS的，而CycliBarrier是基于 ReentrantLock(ReentrantLock也属于AQS同步器)和 Condition 的.

CyclicBarrier 的字面意思是可循环使用（Cyclic）的屏障（Barrier）。它要做的事情是，让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活。CyclicBarrier 默认的构造方法是 CyclicBarrier(int parties)，其参数表示屏障拦截的线程数量，每个线程调用await方法告诉 CyclicBarrier 我已经到达了屏障，然后当前线程被阻塞。

再来看一下它的构造函数：

其中，parties 就代表了有拦截的线程的数量，当拦截的线程数量达到这个值的时候就打开栅栏，让所有线程通过。

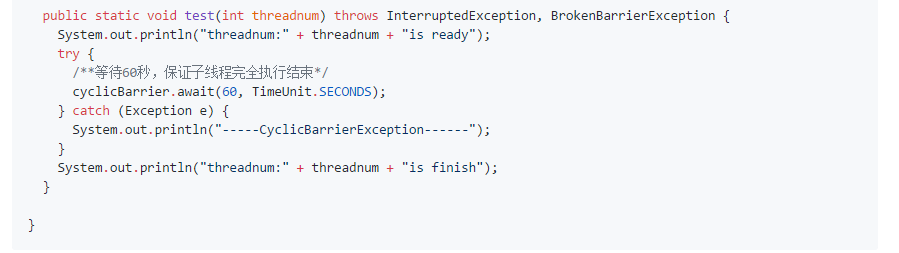
### CyclicBarrier 的应用场景

CyclicBarrier 可以用于多线程计算数据，最后合并计算结果的应用场景。比如我们用一个 Excel 保存了用户所有银行流水，每个 Sheet 保存一个帐户近一年的每笔银行流水，现在需要统计用户的日均银行流水，先用多线程处理每个 sheet 里的银行流水，都执行完之后，得到每个 sheet 的日均银行流水，最后，再用 barrierAction 用这些线程的计算结果，计算出整个 Excel 的日均银行流水。

**CyclicBarrier 的使用示例**

示例 1：





运行结果，如下：

threadnum:0is ready

threadnum:1is ready

threadnum:2is ready

threadnum:3is ready

threadnum:4is ready

threadnum:4is finish

threadnum:0is finish

threadnum:1is finish

threadnum:2is finish

threadnum:3is finish

threadnum:5is ready

threadnum:6is ready

threadnum:7is ready

threadnum:8is ready

threadnum:9is ready

threadnum:9is finish

threadnum:5is finish

threadnum:8is finish

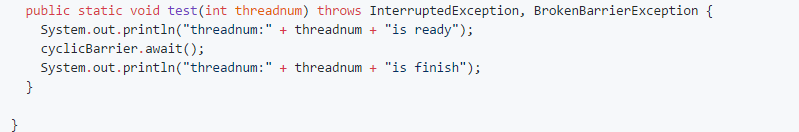
threadnum:7is finish

threadnum:6is finish

......

可以看到当线程数量也就是请求数量达到我们定义的 5 个的时候， await方法之后的方法才被执行。

另外，CyclicBarrier 还提供一个更高级的构造函数CyclicBarrier(int parties, Runnable barrierAction)，用于在线程到达屏障时，优先执行barrierAction，方便处理更复杂的业务场景。示例代码如下：



运行结果，如下：

threadnum:0is ready

threadnum:1is ready

threadnum:2is ready

threadnum:3is ready

threadnum:4is ready

------当线程数达到之后，优先执行------

threadnum:4is finish

threadnum:0is finish

threadnum:2is finish

threadnum:1is finish

threadnum:3is finish

threadnum:5is ready

threadnum:6is ready

threadnum:7is ready

threadnum:8is ready

threadnum:9is ready

------当线程数达到之后，优先执行------

threadnum:9is finish

threadnum:5is finish

threadnum:6is finish

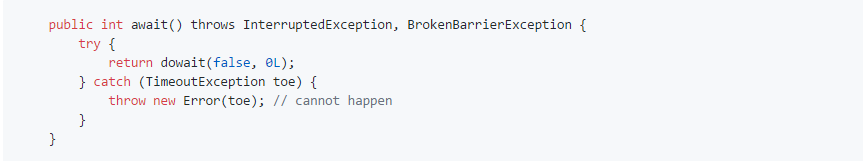
threadnum:8is finish

threadnum:7is finish

......

**CyclicBarrier源码分析**

当调用 CyclicBarrier 对象调用 await() 方法时，实际上调用的是dowait(false, 0L)方法。 await() 方法就像树立起一个栅栏的行为一样，将线程挡住了，当拦住的线程数量达到 parties 的值时，栅栏才会打开，线程才得以通过执行。



dowait(false, 0L)：

// 当线程数量或者请求数量达到 count 时 await 之后的方法才会被执行。上面的示例中 count 的值就为 5。







总结：CyclicBarrier 内部通过一个 count 变量作为计数器，cout 的初始值为 parties 属性的初始化值，每当一个线程到了栅栏这里了，那么就将计数器减一。如果 count 值为 0 了，表示这是这一代最后一个线程到达栅栏，就尝试执行我们构造方法中输入的任务。

### CyclicBarrier 和 CountDownLatch 的区别

**下面这个是国外一个大佬的回答：**

CountDownLatch 是计数器，只能使用一次，而 CyclicBarrier 的计数器提供 reset 功能，可以多次使用。但是我不那么认为它们之间的区别仅仅就是这么简单的一点。我们来从 jdk 作者设计的目的来看，javadoc 是这么描述它们的：

CountDownLatch: A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.(CountDownLatch: 一个或者多个线程，等待其他多个线程完成某件事情之后才能执行；) CyclicBarrier : A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point.(CyclicBarrier : 多个线程互相等待，直到到达同一个同步点，再继续一起执行。)

对于 CountDownLatch 来说，重点是“一个线程（多个线程）等待”，而其他的 N 个线程在完成“某件事情”之后，可以终止，也可以等待。而对于 CyclicBarrier，重点是多个线程，在任意一个线程没有完成，所有的线程都必须等待。

CountDownLatch 是计数器，线程完成一个记录一个，只不过计数不是递增而是递减，而 CyclicBarrier 更像是一个阀门，需要所有线程都到达，阀门才能打开，然后继续执行。

**ReentrantLock 和 ReentrantReadWriteLock**

ReentrantLock 和 synchronized 的区别在上面已经讲过了这里就不多做讲解。另外，需要注意的是：读写锁 ReentrantReadWriteLock 可以保证多个线程可以同时读，所以在读操作远大于写操作的时候，读写锁就非常有用了。

# 并发容器

## JDK 提供的并发容器总结

JDK 提供的这些容器大部分在 java.util.concurrent 包中。

* **ConcurrentHashMap:** 线程安全的 HashMap
* **CopyOnWriteArrayList:** 线程安全的 List，在读多写少的场合性能非常好，远远好于 Vector.
* **ConcurrentLinkedQueue:** 高效的并发队列，使用链表实现。可以看做一个线程安全的 LinkedList，这是一个非阻塞队列。
* **BlockingQueue:** 这是一个接口，JDK 内部通过链表、数组等方式实现了这个接口。表示阻塞队列，非常适合用于作为数据共享的通道。
* **ConcurrentSkipListMap:** 跳表的实现。这是一个 Map，使用跳表的数据结构进行快速查找。

## ConcurrentHashMap

我们知道 HashMap 不是线程安全的，在并发场景下如果要保证一种可行的方式是使用 Collections.synchronizedMap() 方法来包装我们的 HashMap。但这是通过使用一个全局的锁来同步不同线程间的并发访问，因此会带来不可忽视的性能问题。

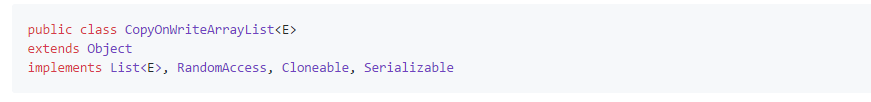
所以就有了 HashMap 的线程安全版本—— ConcurrentHashMap 的诞生。在 ConcurrentHashMap 中，无论是读操作还是写操作都能保证很高的性能：在进行读操作时(几乎)不需要加锁，而在写操作时通过锁分段技术只对所操作的段加锁而不影响客户端对其它段的访问。

关于 ConcurrentHashMap 相关问题，我在 [Java 集合框架常见面试题](https://github.com/Snailclimb/JavaGuide/blob/master/docs/java/collection/Java%E9%9B%86%E5%90%88%E6%A1%86%E6%9E%B6%E5%B8%B8%E8%A7%81%E9%9D%A2%E8%AF%95%E9%A2%98.md) 这篇文章中已经提到过。下面梳理一下关于 ConcurrentHashMap 比较重要的问题：

* [ConcurrentHashMap 和 Hashtable 的区别](https://github.com/Snailclimb/JavaGuide/blob/master/docs/java/collection/Java%E9%9B%86%E5%90%88%E6%A1%86%E6%9E%B6%E5%B8%B8%E8%A7%81%E9%9D%A2%E8%AF%95%E9%A2%98.md#concurrenthashmap-%E5%92%8C-hashtable-%E7%9A%84%E5%8C%BA%E5%88%AB)
* [ConcurrentHashMap 线程安全的具体实现方式/底层具体实现](https://github.com/Snailclimb/JavaGuide/blob/master/docs/java/collection/Java%E9%9B%86%E5%90%88%E6%A1%86%E6%9E%B6%E5%B8%B8%E8%A7%81%E9%9D%A2%E8%AF%95%E9%A2%98.md#concurrenthashmap%E7%BA%BF%E7%A8%8B%E5%AE%89%E5%85%A8%E7%9A%84%E5%85%B7%E4%BD%93%E5%AE%9E%E7%8E%B0%E6%96%B9%E5%BC%8F%E5%BA%95%E5%B1%82%E5%85%B7%E4%BD%93%E5%AE%9E%E7%8E%B0)

## CopyOnWriteArrayList

**CopyOnWriteArrayList 简介**



在很多应用场景中，读操作可能会远远大于写操作。由于读操作根本不会修改原有的数据，因此对于每次读取都进行加锁其实是一种资源浪费。我们应该允许多个线程同时访问 List 的内部数据，毕竟读取操作是安全的。

这和我们之前在多线程章节讲过 ReentrantReadWriteLock 读写锁的思想非常类似，也就是读读共享、写写互斥、读写互斥、写读互斥。JDK 中提供了 CopyOnWriteArrayList 类比相比于在读写锁的思想又更进一步。为了将读取的性能发挥到极致，CopyOnWriteArrayList 读取是完全不用加锁的，并且更厉害的是：写入也不会阻塞读取操作。只有写入和写入之间需要进行同步等待。这样一来，读操作的性能就会大幅度提升。**那它是怎么做的呢？**

**CopyOnWriteArrayList 是如何做到的？**

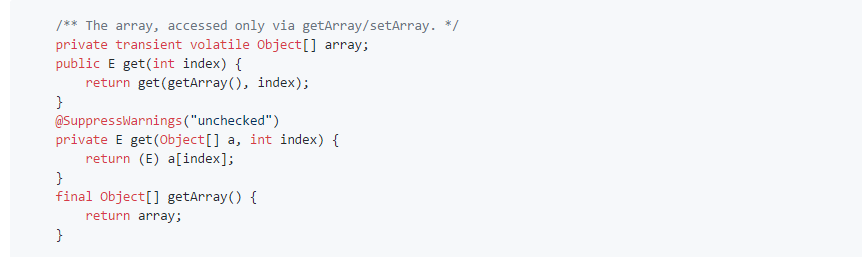
CopyOnWriteArrayList 类的所有可变操作（add，set 等等）都是通过创建底层数组的新副本来实现的。当 List 需要被修改的时候，我并不修改原有内容，而是对原有数据进行一次复制，将修改的内容写入副本。写完之后，再将修改完的副本替换原来的数据，这样就可以保证写操作不会影响读操作了。

从 CopyOnWriteArrayList 的名字就能看出CopyOnWriteArrayList 是满足CopyOnWrite 的 ArrayList，所谓CopyOnWrite 也就是说：在计算机，如果你想要对一块内存进行修改时，我们不在原有内存块中进行写操作，而是将内存拷贝一份，在新的内存中进行写操作，写完之后呢，就将指向原来内存指针指向新的内存，原来的内存就可以被回收掉了。

**CopyOnWriteArrayList 读取和写入源码简单分析**

**CopyOnWriteArrayList 读取操作的实现**

读取操作没有任何同步控制和锁操作，理由就是内部数组 array 不会发生修改，只会被另外一个 array 替换，因此可以保证数据安全。



**CopyOnWriteArrayList 写入操作的实现**

CopyOnWriteArrayList 写入操作 add() 方法在添加集合的时候加了锁，保证了同步，避免了多线程写的时候会 copy 出多个副本出来。

ConcurrentLinkedQueue

Java 提供的线程安全的 Queue 可以分为**阻塞队列**和**非阻塞队列**，其中阻塞队列的典型例子是 BlockingQueue，非阻塞队列的典型例子是 ConcurrentLinkedQueue，在实际应用中要根据实际需要选用阻塞队列或者非阻塞队列。 **阻塞队列可以通过加锁来实现，非阻塞队列可以通过 CAS 操作实现。**

从名字可以看出，ConcurrentLinkedQueue这个队列使用链表作为其数据结构．ConcurrentLinkedQueue 应该算是在高并发环境中性能最好的队列了。它之所有能有很好的性能，是因为其内部复杂的实现。

ConcurrentLinkedQueue 内部代码我们就不分析了，大家知道 ConcurrentLinkedQueue 主要使用 CAS 非阻塞算法来实现线程安全就好了。

ConcurrentLinkedQueue 适合在对性能要求相对较高，同时对队列的读写存在多个线程同时进行的场景，即如果对队列加锁的成本较高则适合使用无锁的 ConcurrentLinkedQueue 来替代。

## BlockingQueue

### BlockingQueue 简单介绍

上面我们己经提到了 ConcurrentLinkedQueue 作为高性能的非阻塞队列。下面我们要讲到的是阻塞队列——BlockingQueue。阻塞队列（BlockingQueue）被广泛使用在“生产者-消费者”问题中，其原因是 BlockingQueue 提供了可阻塞的插入和移除的方法。当队列容器已满，生产者线程会被阻塞，直到队列未满；当队列容器为空时，消费者线程会被阻塞，直至队列非空时为止。

BlockingQueue 是一个接口，继承自 Queue，所以其实现类也可以作为 Queue 的实现来使用，而 Queue 又继承自 Collection 接口。下面是 BlockingQueue 的相关实现类：

**下面主要介绍一下:ArrayBlockingQueue、LinkedBlockingQueue、PriorityBlockingQueue，这三个 BlockingQueue 的实现类。**

### ArrayBlockingQueue

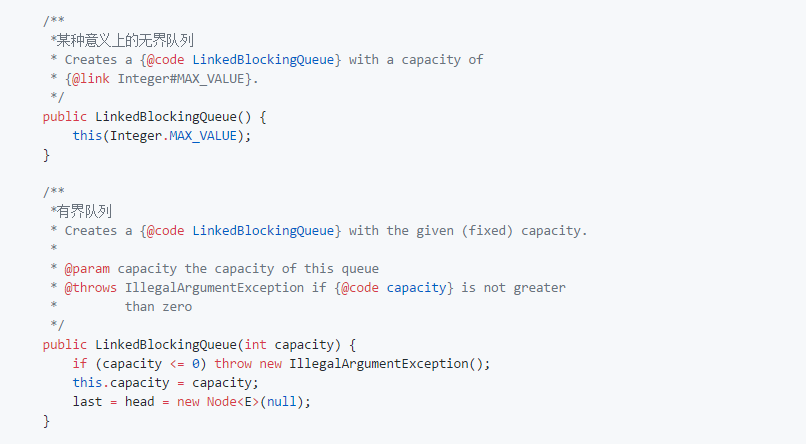
**ArrayBlockingQueue** 是 BlockingQueue 接口的有界队列实现类，底层采用**数组**来实现。ArrayBlockingQueue 一旦创建，容量不能改变。其并发控制采用可重入锁来控制，不管是插入操作还是读取操作，都需要获取到锁才能进行操作。当队列容量满时，尝试将元素放入队列将导致操作阻塞;尝试从一个空队列中取一个元素也会同样阻塞。

ArrayBlockingQueue 默认情况下不能保证线程访问队列的公平性，所谓公平性是指严格按照线程等待的绝对时间顺序，即最先等待的线程能够最先访问到 ArrayBlockingQueue。而非公平性则是指访问 ArrayBlockingQueue 的顺序不是遵守严格的时间顺序，有可能存在，当 ArrayBlockingQueue 可以被访问时，长时间阻塞的线程依然无法访问到 ArrayBlockingQueue。如果保证公平性，通常会降低吞吐量。如果需要获得公平性的 ArrayBlockingQueue，可采用如下代码：

LinkedBlockingQueue

**LinkedBlockingQueue** 底层基于**单向链表**实现的阻塞队列，可以当做无界队列也可以当做有界队列来使用，同样满足 FIFO 的特性，与 ArrayBlockingQueue 相比起来具有更高的吞吐量，为了防止 LinkedBlockingQueue 容量迅速增，损耗大量内存。通常在创建 LinkedBlockingQueue 对象时，会指定其大小，如果未指定，容量等于 Integer.MAX\_VALUE。

**相关构造方法:**



### PriorityBlockingQueue

**PriorityBlockingQueue** 是一个支持优先级的无界阻塞队列。默认情况下元素采用自然顺序进行排序，也可以通过自定义类实现 compareTo() 方法来指定元素排序规则，或者初始化时通过构造器参数 Comparator 来指定排序规则。

PriorityBlockingQueue 并发控制采用的是 **ReentrantLock**，队列为无界队列（ArrayBlockingQueue 是有界队列，LinkedBlockingQueue 也可以通过在构造函数中传入 capacity 指定队列最大的容量，但是 PriorityBlockingQueue 只能指定初始的队列大小，后面插入元素的时候，**如果空间不够的话会自动扩容**）。

简单地说，它就是 PriorityQueue 的线程安全版本。不可以插入 null 值，同时，插入队列的对象必须是可比较大小的（comparable），否则报 ClassCastException 异常。它的插入操作 put 方法不会 block，因为它是无界队列（take 方法在队列为空的时候会阻塞）。

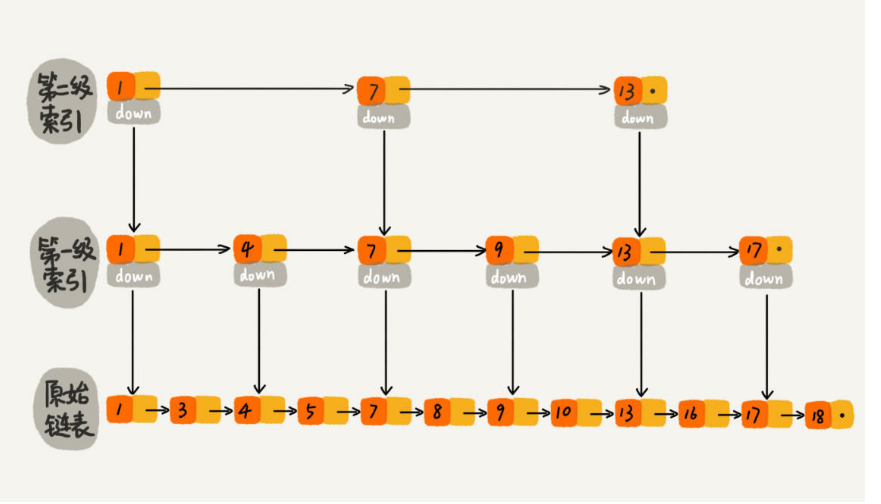
**推荐文章：**

《解读 Java 并发队列 BlockingQueue》

<https://javadoop.com/post/java-concurrent-queue>

### ConcurrentSkipListMap

下面这部分内容参考了极客时间专栏[《数据结构与算法之美》](https://time.geekbang.org/column/intro/126?code=zl3GYeAsRI4rEJIBNu5B/km7LSZsPDlGWQEpAYw5Vu0=&utm_term=SPoster" \o "《数据结构与算法之美》)以及《实战 Java 高并发程序设计》。

**为了引出 ConcurrentSkipListMap，先带着大家简单理解一下跳表。**

对于一个单链表，即使链表是有序的，如果我们想要在其中查找某个数据，也只能从头到尾遍历链表，这样效率自然就会很低，跳表就不一样了。跳表是一种可以用来快速查找的数据结构，有点类似于平衡树。它们都可以对元素进行快速的查找。但一个重要的区别是：对平衡树的插入和删除往往很可能导致平衡树进行一次全局的调整。而对跳表的插入和删除只需要对整个数据结构的局部进行操作即可。这样带来的好处是：在高并发的情况下，你会需要一个全局锁来保证整个平衡树的线程安全。而对于跳表，你只需要部分锁即可。这样，在高并发环境下，你就可以拥有更好的性能。而就查询的性能而言，跳表的时间复杂度也是 **O(logn)** 所以在并发数据结构中，JDK 使用跳表来实现一个 Map。

跳表的本质是同时维护了多个链表，并且链表是分层的，

最低层的链表维护了跳表内所有的元素，每上面一层链表都是下面一层的子集。

跳表内的所有链表的元素都是排序的。查找时，可以从顶级链表开始找。一旦发现被查找的元素大于当前链表中的取值，就会转入下一层链表继续找。这也就是说在查找过程中，搜索是跳跃式的。如上图所示，在跳表中查找元素 18。

查找 18 的时候原来需要遍历 18 次，现在只需要 7 次即可。针对链表长度比较大的时候，构建索引查找效率的提升就会非常明显。

从上面很容易看出，**跳表是一种利用空间换时间的算法。**

使用跳表实现 Map 和使用哈希算法实现 Map 的另外一个不同之处是：哈希并不会保存元素的顺序，而跳表内所有的元素都是排序的。因此在对跳表进行遍历时，你会得到一个有序的结果。所以，如果你的应用需要有序性，那么跳表就是你不二的选择。JDK 中实现这一数据结构的类是 ConcurrentSkipListMap。