Thread 底层调用start的 native方法

并行 及 并发 概念？

并行，多个线程同时执行的概念。

并发，单位时间内，线程处理数据的能力，吞吐量。

Java默认是多线程

高并发编程的意义及好处？

平衡使用我们的线程(好处)

不要cpu 过累(意义)

面试题

1.run 和start方法的区别？

.run()时函数调用和线程没有任何关系，.start()方法执行native底层函数，系统最终调度到run函数，这才是线程。

2. 如何控制线程的执行顺序？

.join() 函数控制，让t2获取执行权力，能够做到顺序执行

3. 多线程中的并行和并发是什么？

四个车道，四辆车并行的行驶，就是并行。

四个车道，五秒钟多少的车流量，多少的吞吐量，是并发。单位时间内的执行效率

4. 在Java中能不能指定CPU去执行某个线程？

不能，Java是做不到的，唯一能够去干预的是C语言调用内核的API去执行才行。

5. 项目开发过程中，你会考虑线程优先级么？

不会考虑优先级，为什么呢？因为线程的优先级很依赖与系统的平台，所以这个优先级无法对号入座，无法做到你想象中的优先级，属于

不稳定，有风险，因为某些开源框架，也不能依靠线程优先级来设置自己想要的优先级顺序，这个是不可靠的。

例：Java线程优先级有十级，而此时操作系统优先级指有2~3级，那就对应不上了。

6. sleep 和 wait 有什么区别？

sleep是休眠，等休眠时间一国，才有执行的资格，注意是有资格，并不能马上就会被执行，什么时候执行，取决于cpu系统调度。

wait是登台，需要唤醒，唤醒后，才有执行的资格。

sleep()：线程进入睡眠状态，不会释放锁

wait()：调动方法之前，必须要持有锁。调用了wait()方法以后，锁就会被释放，进入锁的等待队列，方法返回后重新拿到锁

含义不同: sleep 无条件可以休眠， wait是某些原因与条件 需要等待一下。

7. 在Java中能不能强制中断线程的执行

虽然提供了stop函数，但是此函数不推荐使用。因为这种方式很暴力，很危险，例如:下载图片5kb，只下载了4kb等我们可以使用interrupt来

处理我线程的停止，但是注意interrupt只是协作式的方式，并不能绝对保证终中断，并不是抢占式。

8. 如何让出当前线程的执行权？

yield() ：让出cpu的执行权，将线程从运行转到可运行状态，但是下个时间片，该线程依然有可能被再次选中运行

9. notify()和notifyAll()区别？

notify()可能发生信号丢失的情况。并且可能会导致死锁现象。

锁池:假设线程A已经拥有了某个对象(注意:不是类)的锁，而其它的线程想要调用这个对象的某个synchronized方法(

或者synchronized块)，由于这些线程在进入对象的synchronized方法之前必须先获得该对象的锁的拥有权，

但是该对象的锁目前正被线程A拥有，所以这些线程就进入了该对象的锁池中。

等待池:假设一个线程A调用了某个对象的wait()方法，线程A就会释放该对象的锁后，进入到了该对象的等待池中

如果线程调用了对象的 wait()方法，那么线程便会处于该对象的等待池中，等待池中的线程不会去竞争该对象的锁。

当有线程调用了对象的 notifyAll()方法（唤醒所有 wait 线程）或 notify()方法（只随机唤醒一个 wait 线程），

被唤醒的的线程便会进入该对象的锁池中，锁池中的线程会去竞争该对象锁。也就是说，调用了notify后只要一个线程会由等待池进入锁池，

而notifyAll会将该对象等待池内的所有线程移动到锁池中，等待锁竞争

优先级高的线程竞争到对象锁的概率大，假若某线程没有竞争到该对象锁，它还会留在锁池中，唯有线程再次调用 wait()方法，

它才会重新回到等待池中。而竞争到对象锁的线程则继续往下执行，直到执行完了 synchronized 代码块，它会释放掉该对象锁，

这时锁池中的线程会继续竞争该对象锁。

唤醒线程，另一种解释可以说是将线程由等待池移动到锁池，notifyAll调用后，会将全部线程由等待池移到锁池，然后参与锁的竞争

，竞争成功则继续执行，如果不成功则留在锁池等待锁被释放后再次参与竞争。而notify只会唤醒一个线程。

10.调用yield() 、sleep()、wait()、notify()等方法对锁有何影响？

yield()：让出时间片，不会释放锁

sleep()：线程进入睡眠状态，不会释放锁

wait()：调动方法之前，必须要持有锁。调用了wait()方法以后，锁就会被释放，进入锁的等待队列，方法返回后重新拿到锁

notify()：调动方法之前，必须要持有锁，调用notify()方法本身不会释放锁的。而是通知等待队列中的某一个线程，同步代码块执行完毕后才会释放锁

notifyAll()：同notify，有一点不同在于，notifyAll会发出n个信号（n=等待线程数），而notify只会发出一个信号，通常情况下，尽量选择notifyAll

11.join()函数作用？

线程A，执行了线程B的join方法，线程A必须要等待B执行完成了以后，线程A才能继续自己的工作

12.sleep 和 wait 哪个函数才会清除interrupt中断标记？

sleep 在抛出异常的时候，捕获异常之前就已经清除中断标记。

锁、多线程、唤醒机制

13．有几种新启线程的方式？(两种)

1.继承Thread类

2.实现Runnable接口

实现Callable 严格意义上不算。实现Callable的接口对象会封装到FutureTask。FutureTask实现RunnableFuture。RunnableFuture 接口继承了Runnable和Future接口。

线程的生命周期?

1. 初始 NEW
2. 运行 RUNNING
3. 就绪 REDAY
4. 等待 WAITING
5. 阻塞 BLOCK
6. 终止 TERMINATED

运行：获取到时间片cpu调度。

调用wait进入等待 调用notify唤醒

调用synchronized修饰的方法中，未获取到锁 进入阻塞状态

调用sleep进入等待态

调用lock没有拿到锁，进入阻塞态？不会！只是进入等待、等待超时。

只有在synchronized修饰才会进入阻塞态。

锁概念 有一个线程进入，加锁后，其他线程无法进入。

synchronized 隐式锁 可重入锁 内部会完成 锁定解锁底层逻辑。JDK内置锁 无法修改。需要关键字 synchronized

lock 显式锁 可以手动控制锁定解锁

唤醒机制

等待区域:wait()

获取对象的锁🔒

synchronized(对象锁){

wait() //释放锁

}

通知区域:notify()

获取对象的锁🔒

synchronized(对象锁){

notify()

}

死锁

死锁是指 两个或两个以上的进程在执行过程中，由于竞争资源或者用于彼此通信造成的一种阻塞的现象，若无外力作用，他们

都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁。

条件

1.多个操作者(M>=2)争夺多个资源(N>=2) (N<=M)

2.争夺资源的顺序不对

3.拿到资源不放手

规范

1.互斥条件

2.请求和保持

3.不剥夺

4.循环

死锁解决，规定争夺资源的顺序，lock.try尝试拿锁。

活锁

线程A、B 锁 1、2

线程A 持有锁1 想 获取锁2 发现线程B持有锁2 释放所有锁 重复 持有锁1 想 获取锁2

线程B 持有锁2 想 获取锁1 发现线程A持有锁1 释放所有锁 重复 持有锁2 想 获取锁1

两个线程都在不停工作。

锁饥饿

ThreadLocal 自定义存在问题。 map存在竞争。

ThreadLocal 实现原理？ thread 变量

class Thread{//持有ThreadLocal.ThreadLocalMap 成员变量

}

class ThreadLocalMap{//持有一个Entry[]数组

Entry[] table

}

class Entry{//每个元素，持有ThreadLocal Object value

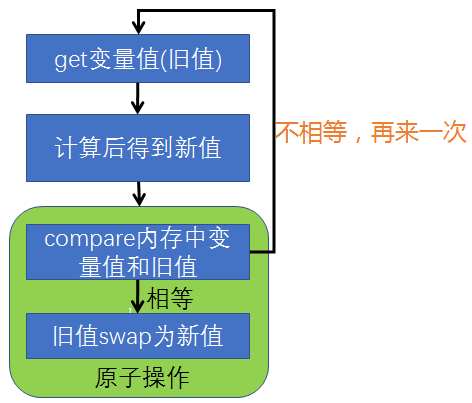
ThreadLocal<?> k ,

Object v

}

CAS基本原理

Compare And Swap 比较和交换



CAS原理

利用现代处理器都支持CAS的指令

循环这个指令，知道成功为止

执行流程

1.get变量值(旧值)

2.计算后得到新值

3.compare内存中变量值和旧值

4.相等 旧值swap为新值

5.不相等 重复操作

CAS问题

ABA问题

开销问题

只能保证一个共享变量的原子操作

Atomic 原子变量类。cas 操作

如何实现原子操作？

1.synchronized

2.compare and swap(轻量级)

悲观锁(synchronized)

乐观锁(cas)

synchronized执行时，阻塞时会上下文切换，非常耗时。

cas 不会进入阻塞

ABA问题

线程1 A-->B 线程执行A-->B

线程2 A-->C-->A

解决ABA问题 版本戳

AtomicReference

AtomicMarkableReference(是否改动)

AtomicStampedReference(是否改动，改动次数)

AtomicReference 解决只能保证一个共享变量的原子操作，可以支持改两个变量。

CAS 比较交换操作 为原子操作，由CPU保证。

线程池

AsynTask内部 线程池

构造函数各个参数的含义？

CorePoolSize 核心线程数

maximumPoolSize 最大线程数

keepAliveTime 存活时间

TimeUnit

ThreadFactory

BlockingQueue<Runnable> 超过的线程进入阻塞队列

RejectedExecutionHandler 默认拒绝策略

1. AbortPolicy 直接抛出异常(默认)
2. DiscardPolicy 把最新提交的任务废弃
3. CallerRunsPolicy 让调用者线程执行任务
4. DiscardOldestPolicy 直接丢弃队列中最老的

keepAliveTime TimeUnit 根据这两个参数来控制线程的存活时间，线程池超过数值就会销毁。

什么是线程池？为什么要用线程池？

Thread 线程 操作系统

T1：创建

T2：任务执行

T3：销毁

为了统一管理，节省内存。用到线程池。

1. 线程池的创建
2. 提交任务
3. 关闭线程池

Excute

Submit futureTask 有返回值

Shutdown尝试关闭线程池，没有执行的

ShutdownNow 线程的中断，协作机制

合理配置线程

任务特性：

CPU密集型 机器的核心数 磁盘虚拟内存 最多+1 可能会产生页缺失，为保证cpu跑满+1

IO密集型 机器的核心数\*2

混合型

JDK中的线程池工作原理？

创建线程池值后，会根据CorePoolSize创建出线程 ，当有提交任务，CorePool里边会执行任务，当线程都在占用，继续有任务提交，超出核心线程数，会创建新线程并添加到BlockingQueue阻塞队列中。当阻塞队列中也满了，根据maximumPool继续创建线程执行。当线程数超出maximumPool大小，使用拒绝策略。

DMA 中断 OS 零拷贝 控制器

OS 内核空间 用户空间 mmap directmeory

# Java并发基础知识补全

## 启动

启动线程的方式只有：

1、X extends Thread;，然后X.start

2、X implements Runnable；然后交给Thread运行

## 线程的状态

Java中线程的状态分为6种：

1. 初始(NEW)：新创建了一个线程对象，但还没有调用start()方法。

2. 运行(RUNNABLE)：Java线程中将就绪（ready）和运行中（running）两种状态笼统的称为“运行”。

线程对象创建后，其他线程(比如main线程）调用了该对象的start()方法。该状态的线程位于可运行线程池中，等待被线程调度选中，获取CPU的使用权，此时处于就绪状态（ready）。就绪状态的线程在获得CPU时间片后变为运行中状态（running）。

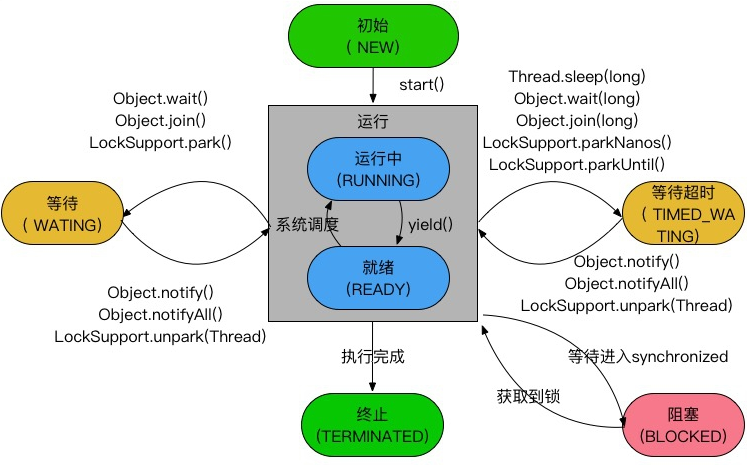
3. 阻塞(BLOCKED)：表示线程阻塞于锁。

4. 等待(WAITING)：进入该状态的线程需要等待其他线程做出一些特定动作（通知或中断）。1

5. 超时等待(TIMED\_WAITING)：该状态不同于WAITING，它可以在指定的时间后自行返回。

6. 终止(TERMINATED)：表示该线程已经执行完毕。

状态之间的变迁如下图所示



## 死锁

### 概念

是指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁。

举个例子：A和B去按摩洗脚，都想在洗脚的时候，同时顺便做个头部按摩，13技师擅长足底按摩，14擅长头部按摩。

这个时候A先抢到14，B先抢到13，两个人都想同时洗脚和头部按摩，于是就互不相让，扬言我死也不让你，这样的话，A抢到14，想要13，B抢到13，想要14，在这个想同时洗脚和头部按摩的事情上A和B就产生了死锁。怎么解决这个问题呢？

第一种，假如这个时候，来了个15，刚好也是擅长头部按摩的，A又没有两个脑袋，自然就归了B，于是B就美滋滋的洗脚和做头部按摩，剩下A在旁边气鼓鼓的，这个时候死锁这种情况就被打破了，不存在了。

第二种，C出场了，用武力强迫A和B，必须先做洗脚，再头部按摩，这种情况下，A和B谁先抢到13，谁就可以进行下去，另外一个没抢到的，就等着，这种情况下，也不会产生死锁。

所以总结一下：

死锁是必然发生在多操作者（M>=2个）情况下，争夺多个资源（N>=2个，且N<=M）才会发生这种情况。很明显，单线程自然不会有死锁，只有B一个去，不要2个，打十个都没问题；单资源呢？只有13，A和B也只会产生激烈竞争，打得不可开交，谁抢到就是谁的，但不会产生死锁。同时，死锁还有几个要求，1、争夺资源的顺序不对，如果争夺资源的顺序是一样的，也不会产生死锁；

2、争夺者拿到资源不放手。

#### 学术化的定义

死锁的发生必须具备以下四个必要条件。

1）互斥条件：指进程对所分配到的资源进行排它性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求资源，则请求者只能等待，直至占有资源的进程用毕释放。

2）请求和保持条件：指进程已经保持至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有，此时请求进程阻塞，但又对自己已获得的其它资源保持不放。

3）不剥夺条件：指进程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放。

4）环路等待条件：指在发生死锁时，必然存在一个进程——资源的环形链，即进程集合{P0，P1，P2，···，Pn}中的P0正在等待一个P1占用的资源；P1正在等待P2占用的资源，……，Pn正在等待已被P0占用的资源。

理解了死锁的原因，尤其是产生死锁的四个必要条件，就可以最大可能地避免、预防和解除死锁。

只要打破四个必要条件之一就能有效预防死锁的发生。

打破互斥条件：改造独占性资源为虚拟资源，大部分资源已无法改造。

打破不可抢占条件：当一进程占有一独占性资源后又申请一独占性资源而无法满足，则退出原占有的资源。

打破占有且申请条件：采用资源预先分配策略，即进程运行前申请全部资源，满足则运行，不然就等待，这样就不会占有且申请。

打破循环等待条件：实现资源有序分配策略，对所有设备实现分类编号，所有进程只能采用按序号递增的形式申请资源。

避免死锁常见的算法有有序资源分配法、银行家算法。

### 危害

1、线程不工作了，但是整个程序还是活着的2、没有任何的异常信息可以供我们检查。3、一旦程序发生了发生了死锁，是没有任何的办法恢复的，只能重启程序，对正式已发布程序来说，这是个很严重的问题。

### 解决

关键是保证拿锁的顺序一致

两种解决方式

* 1. 内部通过顺序比较，确定拿锁的顺序；

2、采用尝试拿锁的机制。

## 其他线程安全问题

### 活锁

两个线程在尝试拿锁的机制中，发生多个线程之间互相谦让，不断发生同一个线程总是拿到同一把锁，在尝试拿另一把锁时因为拿不到，而将本来已经持有的锁释放的过程。

解决办法：每个线程休眠随机数，错开拿锁的时间。

### 线程饥饿

低优先级的线程，总是拿不到执行时间

## ThreadLocal辨析

#### 与Synchonized的比较

ThreadLocal和Synchonized都用于解决多线程并发訪问。可是ThreadLocal与synchronized有本质的差别。synchronized是利用锁的机制，使变量或代码块在某一时该仅仅能被一个线程訪问。而ThreadLocal为每个线程都提供了变量的副本，使得每个线程在某一时间訪问到的并非同一个对象，这样就隔离了多个线程对数据的数据共享。

#### ThreadLocal的使用

ThreadLocal类接口很简单，只有4个方法，我们先来了解一下：

• void set(Object value)

设置当前线程的线程局部变量的值。

• public Object get()

该方法返回当前线程所对应的线程局部变量。

• public void remove()

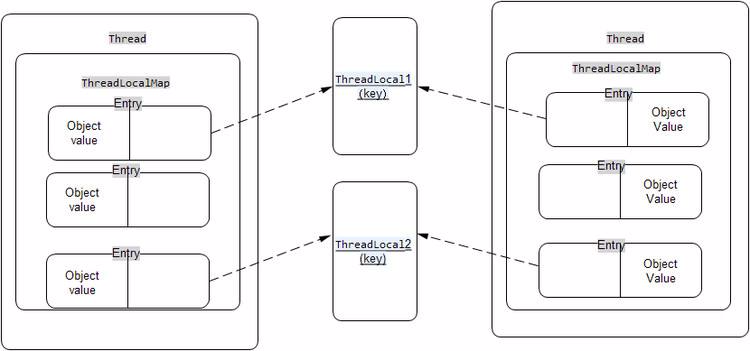
将当前线程局部变量的值删除，目的是为了减少内存的占用，该方法是JDK 5.0新增的方法。需要指出的是，当线程结束后，对应该线程的局部变量将自动被垃圾回收，所以显式调用该方法清除线程的局部变量并不是必须的操作，但它可以加快内存回收的速度。

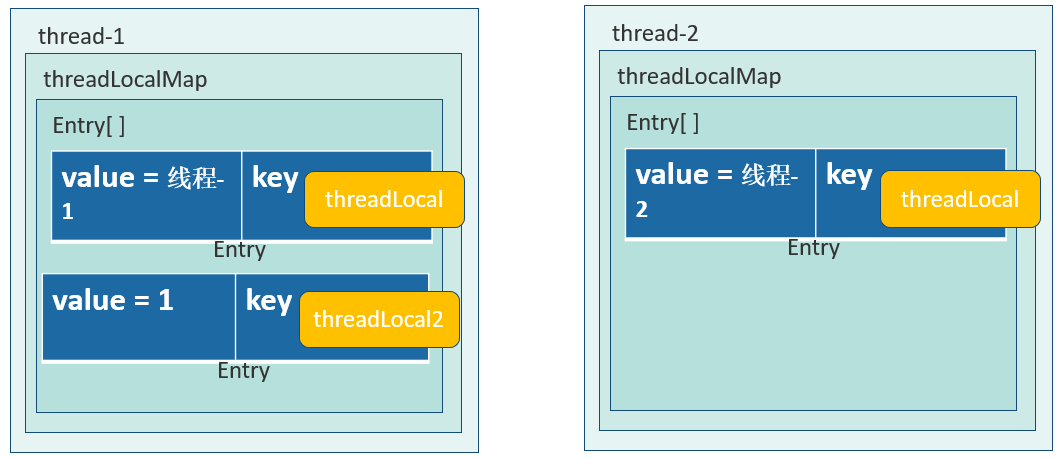
• protected Object initialValue()

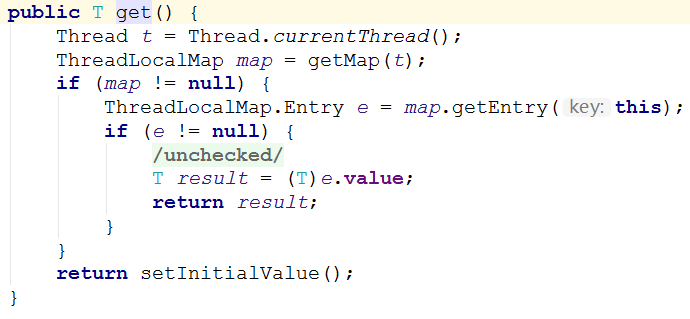
返回该线程局部变量的初始值，该方法是一个protected的方法，显然是为了让子类覆盖而设计的。这个方法是一个延迟调用方法，在线程第1次调用get()或set(Object)时才执行，并且仅执行1次。ThreadLocal中的缺省实现直接返回一个null。

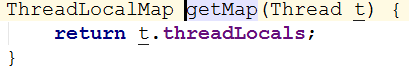
public final static ThreadLocal<String> RESOURCE = new ThreadLocal<String>();RESOURCE代表一个能够存放String类型的ThreadLocal对象。此时不论什么一个线程能够并发访问这个变量，对它进行写入、读取操作，都是线程安全的。

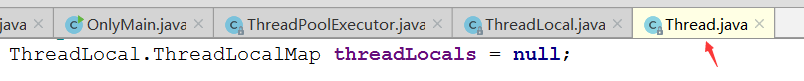
#### 实现解析





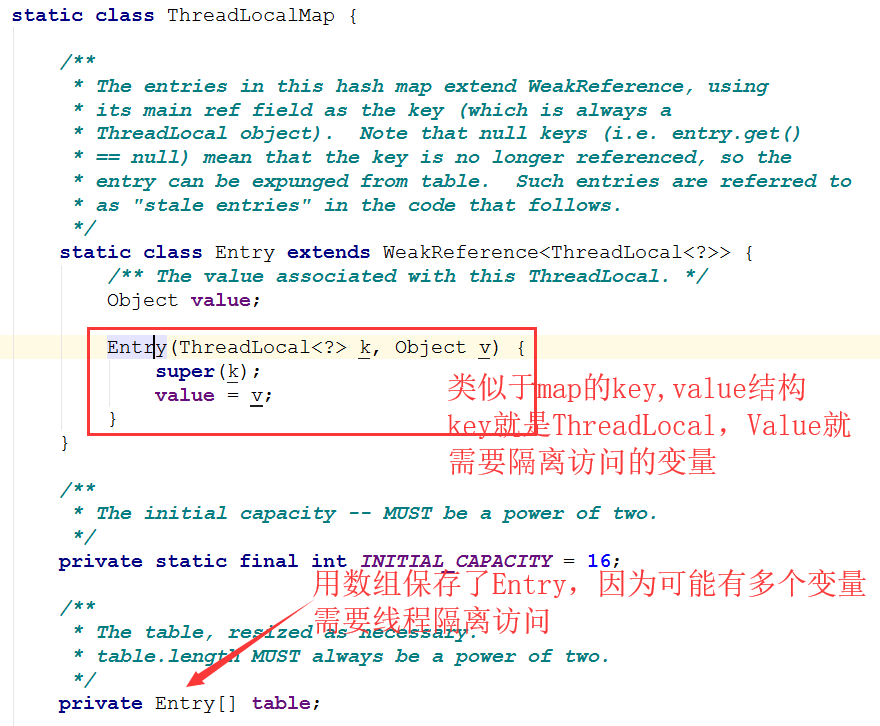




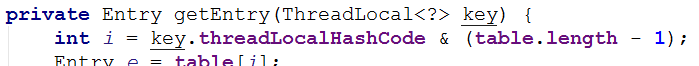


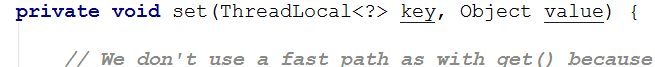
上面先取到当前线程，然后调用getMap方法获取对应的ThreadLocalMap，ThreadLocalMap是ThreadLocal的静态内部类，然后Thread类中有一个这样类型成员，所以getMap是直接返回Thread的成员。

看下ThreadLocal的内部类ThreadLocalMap源码：



可以看到有个Entry内部静态类，它继承了WeakReference，总之它记录了两个信息，一个是ThreadLocal<?>类型，一个是Object类型的值。getEntry方法则是获取某个ThreadLocal对应的值，set方法就是更新或赋值相应的ThreadLocal对应的值。





回顾我们的get方法，其实就是拿到**每个线程独有的ThreadLocalMap**

然后再用ThreadLocal的当前实例，拿到Map中的相应的Entry，然后就可以拿到相应的值返回出去。当然，如果Map为空，还会先进行map的创建，初始化等工作。

# CAS基本原理

## 什么是原子操作？如何实现原子操作？

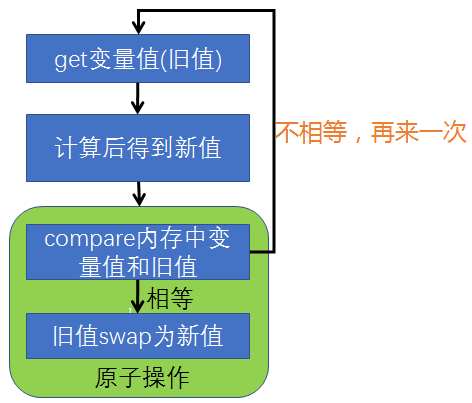
假定有两个操作A和B(A和B可能都很复杂)，如果从执行A的线程来看，当另一个线程执行B时，要么将B全部执行完，要么完全不执行B，那么A和B对彼此来说是原子的。

实现原子操作可以使用锁，锁机制，满足基本的需求是没有问题的了，但是有的时候我们的需求并非这么简单，我们需要更有效，更加灵活的机制，synchronized关键字是基于阻塞的锁机制，也就是说当一个线程拥有锁的时候，访问同一资源的其它线程需要等待，直到该线程释放锁，

这里会有些问题：首先，如果被阻塞的线程优先级很高很重要怎么办？其次，如果获得锁的线程一直不释放锁怎么办？（这种情况是非常糟糕的）。还有一种情况，如果有大量的线程来竞争资源，那CPU将会花费大量的时间和资源来处理这些竞争，同时，还有可能出现一些例如死锁之类的情况，最后，其实锁机制是一种比较粗糙，粒度比较大的机制，相对于像计数器这样的需求有点儿过于笨重。

实现原子操作还可以使用当前的处理器基本都支持CAS()的指令，只不过每个厂家所实现的算法并不一样，每一个CAS操作过程都包含三个运算符：一个内存地址V，一个期望的值A和一个新值B，操作的时候如果这个地址上存放的值等于这个期望的值A，则将地址上的值赋为新值B，否则不做任何操作。

CAS的基本思路就是，如果这个地址上的值和期望的值相等，则给其赋予新值，否则不做任何事儿，但是要返回原值是多少。循环CAS就是在一个循环里不断的做cas操作，直到成功为止。



## CAS实现原子操作的三大问题

### ABA问题。

因为CAS需要在操作值的时候，检查值有没有发生变化，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1，那么A→B→A就会变成1A→2B→3A。举个通俗点的例子，你倒了一杯水放桌子上，干了点别的事，然后同事把你水喝了又给你重新倒了一杯水，你回来看水还在，拿起来就喝，如果你不管水中间被人喝过，只关心水还在，这就是ABA问题。

如果你是一个讲卫生讲文明的小伙子，不但关心水在不在，还要在你离开的时候水被人动过没有，因为你是程序员，所以就想起了放了张纸在旁边，写上初始值0，别人喝水前麻烦先做个累加才能喝水。

### 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。

### 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。

还有一个取巧的办法，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Java 1.5开始，JDK提供了AtomicReference类来保证引用对象之间的原子性，就可以把多个变量放在一个对象里来进行CAS操作。

## Jdk中相关原子操作类的使用

### AtomicInteger

•int addAndGet（int delta）：以原子方式将输入的数值与实例中的值（AtomicInteger里的value）相加，并返回结果。

•boolean compareAndSet（int expect，int update）：如果输入的数值等于预期值，则以原子方式将该值设置为输入的值。

•int getAndIncrement()：以原子方式将当前值加1，注意，这里返回的是自增前的值。

•int getAndSet（int newValue）：以原子方式设置为newValue的值，并返回旧值。

### AtomicIntegerArray

主要是提供原子的方式更新数组里的整型，其常用方法如下。

•int addAndGet（int i，int delta）：以原子方式将输入值与数组中索引i的元素相加。

•boolean compareAndSet（int i，int expect，int update）：如果当前值等于预期值，则以原子方式将数组位置i的元素设置成update值。

需要注意的是，数组value通过构造方法传递进去，然后AtomicIntegerArray会将当前数组复制一份，所以当AtomicIntegerArray对内部的数组元素进行修改时，不会影响传入的数组。

### 更新引用类型

原子更新基本类型的AtomicInteger，只能更新一个变量，如果要原子更新多个变量，就需要使用这个原子更新引用类型提供的类。Atomic包提供了以下3个类。

#### AtomicReference

原子更新引用类型。

#### AtomicStampedReference

利用版本戳的形式记录了每次改变以后的版本号，这样的话就不会存在ABA问题了。这就是AtomicStampedReference的解决方案。AtomicMarkableReference跟AtomicStampedReference差不多， AtomicStampedReference是使用pair的int stamp作为计数器使用，AtomicMarkableReference的pair使用的是boolean mark。 还是那个水的例子，AtomicStampedReference可能关心的是动过几次，AtomicMarkableReference关心的是有没有被人动过，方法都比较简单。

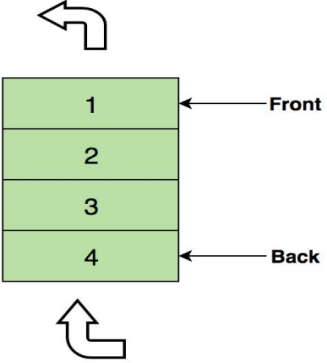
#### AtomicMarkableReference：

原子更新带有标记位的引用类型。可以原子更新一个布尔类型的标记位和引用类型。构造方法是AtomicMarkableReference（V initialRef，booleaninitialMark）。

# 阻塞队列和线程池原理

## 阻塞队列

### 队列



队列是一种特殊的线性表，特殊之处在于它只允许在表的前端（front）进行删除操作，而在表的后端（rear）进行插入操作，和栈一样，队列是一种操作受限制的线性表。进行插入操作的端称为队尾，进行删除操作的端称为队头。

在队列中插入一个队列元素称为入队，从队列中删除一个队列元素称为出队。因为队列只允许在一端插入，在另一端删除，所以只有最早进入队列的元素才能最先从队列中删除，故队列又称为先进先出（FIFO—first in first out）线性表。

### 什么是阻塞队列

1）支持阻塞的插入方法：意思是当队列满时，队列会阻塞插入元素的线程，直到队列不满。

2）支持阻塞的移除方法：意思是在队列为空时，获取元素的线程会等待队列变为非空。

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整体处理数据的速度。

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。

为了解决这种生产消费能力不均衡的问题，便有了生产者和消费者模式。生产者和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信，而是通过阻塞队列来进行通信，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。

阻塞队列常用于生产者和消费者的场景，生产者是向队列里添加元素的线程，消费者是从队列里取元素的线程。阻塞队列就是生产者用来存放元素、消费者用来获取元素的容器。



·抛出异常：当队列满时，如果再往队列里插入元素，会抛出IllegalStateException（"Queuefull"）异常。当队列空时，从队列里获取元素会抛出NoSuchElementException异常。

·返回特殊值：当往队列插入元素时，会返回元素是否插入成功，成功返回true。如果是移除方法，则是从队列里取出一个元素，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，直到队列可用或者响应中断退出。当队列空时，如果消费者线程从队列里take元素，队列会阻塞住消费者线程，直到队列不为空。

·超时退出：当阻塞队列满时，如果生产者线程往队列里插入元素，队列会阻塞生产者线程一段时间，如果超过了指定的时间，生产者线程就会退出。

### 常用阻塞队列

·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。

以上的阻塞队列都实现了BlockingQueue接口，也都是线程安全的。

#### 有界无界？

有限队列就是长度有限，满了以后生产者会阻塞，无界队列就是里面能放无数的东西而不会因为队列长度限制被阻塞，当然空间限制来源于系统资源的限制，如果处理不及时，导致队列越来越大越来越大，超出一定的限制致使内存超限，操作系统或者JVM帮你解决烦恼，直接把你 OOM kill 省事了。

无界也会阻塞，为何？因为阻塞不仅仅体现在生产者放入元素时会阻塞，消费者拿取元素时，如果没有元素，同样也会阻塞。

#### 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。默认情况下不保证线程公平的访问队列，所谓公平访问队列是指阻塞的线程，可以按照阻塞的先后顺序访问队列，即先阻塞线程先访问队列。非公平性是对先等待的线程是非公平的，当队列可用时，阻塞的线程都可以争夺访问队列的资格，有可能先阻塞的线程最后才访问队列。初始化时有参数可以设置

#### 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。

#### Array实现和Linked实现的区别

1. 队列中锁的实现不同

ArrayBlockingQueue实现的队列中的锁是没有分离的，即生产和消费用的是同一个锁；

LinkedBlockingQueue实现的队列中的锁是分离的，即生产用的是putLock，消费是takeLock

2. 在生产或消费时操作不同

ArrayBlockingQueue实现的队列中在生产和消费的时候，是直接将枚举对象插入或移除的；

LinkedBlockingQueue实现的队列中在生产和消费的时候，需要把枚举对象转换为Node<E>进行插入或移除，会影响性能

3. 队列大小初始化方式不同

ArrayBlockingQueue实现的队列中必须指定队列的大小；

LinkedBlockingQueue实现的队列中可以不指定队列的大小，但是默认是Integer.MAX\_VALUE

#### PriorityBlockingQueue

PriorityBlockingQueue是一个支持优先级的无界阻塞队列。默认情况下元素采取自然顺序升序排列。也可以自定义类实现compareTo()方法来指定元素排序规则，或者初始化PriorityBlockingQueue时，指定构造参数Comparator来对元素进行排序。需要注意的是不能保证同优先级元素的顺序。

#### 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。

DelayQueue非常有用，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。

#### 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，非常适合传递性场景。SynchronousQueue的吞吐量高于LinkedBlockingQueue和ArrayBlockingQueue。

#### LinkedTransferQueue

多了tryTransfer和transfer方法，

（1）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。

（2）tryTransfer方法

tryTransfer方法是用来试探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。

#### LinkedBlockingDeque

LinkedBlockingDeque是一个由链表结构组成的双向阻塞队列。所谓双向队列指的是可以从队列的两端插入和移出元素。双向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。

多了addFirst、addLast、offerFirst、offerLast、peekFirst和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度膨胀。另外，双向阻塞队列可以运用在“工作窃取”模式中。

## 线程池

### 为什么要用线程池？

Java中的线程池是运用场景最多的并发框架，几乎所有需要异步或并发执行任务的程序都可以使用线程池。在开发过程中，合理地使用线程池能够带来3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。假设一个服务器完成一项任务所需时间为：T1 创建线程时间，T2 在线程中执行任务的时间，T3 销毁线程时间。 如果：T1 + T3 远大于 T2，则可以采用线程池，以提高服务器性能。线程池技术正是关注如何缩短或调整T1,T3时间的技术，从而提高服务器程序性能的。它把T1，T3分别安排在服务器程序的启动和结束的时间段或者一些空闲的时间段，这样在服务器程序处理客户请求时，不会有T1，T3的开销了。

第三：提高线程的可管理性。线程是稀缺资源，如果无限制地创建，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分配、调优和监控。

### ThreadPoolExecutor 的类关系

Executor是一个接口，它是Executor框架的基础，它将任务的提交与任务的执行分离开来。

ExecutorService接口继承了Executor，在其上做了一些shutdown()、submit()的扩展，可以说是真正的线程池接口；

AbstractExecutorService抽象类实现了ExecutorService接口中的大部分方法；

ThreadPoolExecutor是线程池的核心实现类，用来执行被提交的任务。

ScheduledExecutorService接口继承了ExecutorService接口，提供了带"周期执行"功能ExecutorService；

ScheduledThreadPoolExecutor是一个实现类，可以在给定的延迟后运行命令，或者定期执行命令。ScheduledThreadPoolExecutor比Timer更灵活，功能更强大。

### 线程池的创建各个参数含义

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,BlockingQueue<Runnable> workQueue,ThreadFactory threadFactory,RejectedExecutionHandler handler)

#### corePoolSize

线程池中的核心线程数，当提交一个任务时，线程池创建一个新线程执行任务，直到当前线程数等于corePoolSize；

如果当前线程数为corePoolSize，继续提交的任务被保存到阻塞队列中，等待被执行；

如果执行了线程池的prestartAllCoreThreads()方法，线程池会提前创建并启动所有核心线程。

#### maximumPoolSize

线程池中允许的最大线程数。如果当前阻塞队列满了，且继续提交任务，则创建新的线程执行任务，前提是当前线程数小于maximumPoolSize

#### keepAliveTime

线程空闲时的存活时间，即当线程没有任务执行时，继续存活的时间。默认情况下，该参数只在线程数大于corePoolSize时才有用

#### TimeUnit

keepAliveTime的时间单位

#### workQueue

workQueue必须是BlockingQueue阻塞队列。当线程池中的线程数超过它的corePoolSize的时候，线程会进入阻塞队列进行阻塞等待。通过workQueue，线程池实现了阻塞功能。

一般来说，我们应该尽量使用有界队列，因为使用无界队列作为工作队列会对线程池带来如下影响。

1）当线程池中的线程数达到corePoolSize后，新任务将在无界队列中等待，因此线程池中的线程数不会超过corePoolSize。

2）由于1，使用无界队列时maximumPoolSize将是一个无效参数。

3）由于1和2，使用无界队列时keepAliveTime将是一个无效参数。

4）更重要的，使用无界queue可能会耗尽系统资源，有界队列则有助于防止资源耗尽，同时即使使用有界队列，也要尽量控制队列的大小在一个合适的范围。

#### threadFactory

创建线程的工厂，通过自定义的线程工厂可以给每个新建的线程设置一个具有识别度的线程名，当然还可以更加自由的对线程做更多的设置，比如设置所有的线程为守护线程。

Executors静态工厂里默认的threadFactory，线程的命名规则是“pool-数字-thread-数字”。

#### RejectedExecutionHandler

线程池的饱和策略，当阻塞队列满了，且没有空闲的工作线程，如果继续提交任务，必须采取一种策略处理该任务，线程池提供了4种策略：

（1）AbortPolicy：直接抛出异常，默认策略；

（2）CallerRunsPolicy：用调用者所在的线程来执行任务；

（3）DiscardOldestPolicy：丢弃阻塞队列中靠最前的任务，并执行当前任务；

（4）DiscardPolicy：直接丢弃任务；

当然也可以根据应用场景实现RejectedExecutionHandler接口，自定义饱和策略，如记录日志或持久化存储不能处理的任务。

### 线程池的工作机制

1）如果当前运行的线程少于corePoolSize，则创建新线程来执行任务（注意，执行这一步骤需要获取全局锁）。

2）如果运行的线程等于或多于corePoolSize，则将任务加入BlockingQueue。

3）如果无法将任务加入BlockingQueue（队列已满），则创建新的线程来处理任务。

4）如果创建新线程将使当前运行的线程超出maximumPoolSize，任务将被拒绝，并调用RejectedExecutionHandler.rejectedExecution()方法。

### 提交任务

execute()方法用于提交不需要返回值的任务，所以无法判断任务是否被线程池执行成功。

submit()方法用于提交需要返回值的任务。线程池会返回一个future类型的对象，通过这个future对象可以判断任务是否执行成功，并且可以通过future的get()方法来获取返回值，get()方法会阻塞当前线程直到任务完成，而使用get（long timeout，TimeUnit unit）方法则会阻塞当前线程一段时间后立即返回，这时候有可能任务没有执行完。

### 关闭线程池

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中的任意一个，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方法来关闭线程池，如果任务不一定要执行完，则可以调用shutdownNow方法。

### 合理地配置线程池

要想合理地配置线程池，就必须首先分析任务特性

要想合理地配置线程池，就必须首先分析任务特性，可以从以下几个角度来分析。

•任务的性质：CPU密集型任务、IO密集型任务和混合型任务。

•任务的优先级：高、中和低。

•任务的执行时间：长、中和短。

•任务的依赖性：是否依赖其他系统资源，如数据库连接。

性质不同的任务可以用不同规模的线程池分开处理。

CPU密集型任务应配置尽可能小的线程，如配置Ncpu+1个线程的线程池。由于IO密集型任务线程并不是一直在执行任务，则应配置尽可能多的线程，如2\*Ncpu。

混合型的任务，如果可以拆分，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量将高于串行执行的吞吐量。如果这两个任务执行时间相差太大，则没必要进行分解。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，或者可以使用优先级队列，让执行时间短的任务先执行。

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儿，比如几千。

如果当时我们设置成无界队列，那么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。

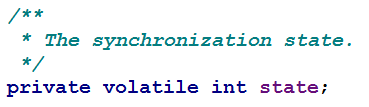
## AbstractQueuedSynchronizer

### 学习AQS的必要性

队列同步器AbstractQueuedSynchronizer（以下简称同步器或AQS），是用来构建锁或者其他同步组件的基础框架，它使用了一个int成员变量表示同步状态，通过内置的FIFO队列来完成资源获取线程的排队工作。并发包的大师（Doug Lea）期望它能够成为实现大部分同步需求的基础。

### AQS使用方式和其中的设计模式

AQS的主要使用方式是继承，子类通过继承AQS并实现它的抽象方法来管理同步状态，在AQS里由一个int型的state来代表这个状态，在抽象方法的实现过程中免不了要对同步状态进行更改，这时就需要使用同步器提供的3个方法（getState()、setState(int newState)和compareAndSetState(int expect,int update)）来进行操作，因为它们能够保证状态的改变是安全的。



在实现上，子类推荐被定义为自定义同步组件的静态内部类，AQS自身没有实现任何同步接口，它仅仅是定义了若干同步状态获取和释放的方法来供自定义同步组件使用，同步器既可以支持独占式地获取同步状态，也可以支持共享式地获取同步状态，这样就可以方便实现不同类型的同步组件（ReentrantLock、ReentrantReadWriteLock和CountDownLatch等）。

同步器是实现锁（也可以是任意同步组件）的关键，在锁的实现中聚合同步器。可以这样理解二者之间的关系：

锁是面向使用者的，它定义了使用者与锁交互的接口（比如可以允许两个线程并行访问），隐藏了实现细节；

同步器面向的是锁的实现者，它简化了锁的实现方式，屏蔽了同步状态管理、线程的排队、等待与唤醒等底层操作。锁和同步器很好地隔离了使用者和实现者所需关注的领域。

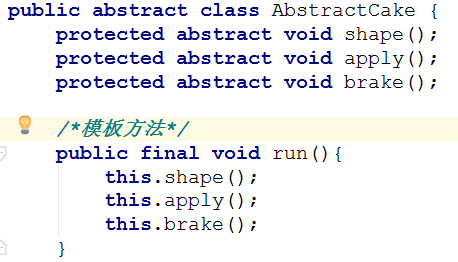
实现者需要继承同步器并重写指定的方法，随后将同步器组合在自定义同步组件的实现中，并调用同步器提供的模板方法，而这些模板方法将会调用使用者重写的方法。

#### 模板方法模式

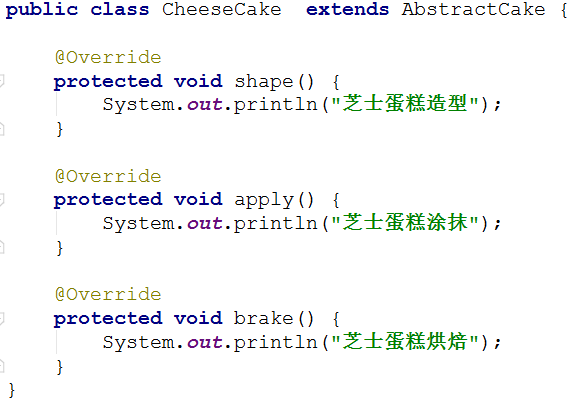
同步器的设计基于模板方法模式。模板方法模式的意图是，定义一个操作中的算法的骨架，而将一些步骤的实现延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。我们最常见的就是Spring框架里的各种Template。

##### 实际例子

我们开了个蛋糕店，蛋糕店不能只卖一种蛋糕呀，于是我们决定先卖奶油蛋糕，芝士蛋糕和慕斯蛋糕。三种蛋糕在制作方式上一样，都包括造型，烘焙和涂抹蛋糕上的东西。所以可以定义一个抽象蛋糕模型

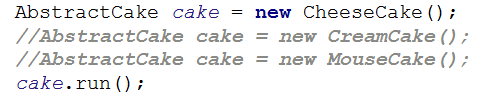


然后就可以批量生产三种蛋糕

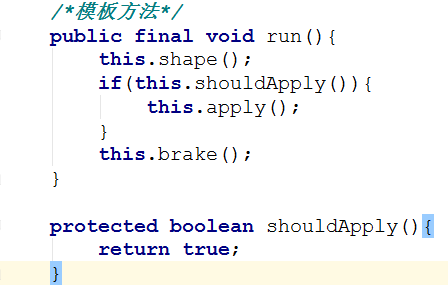




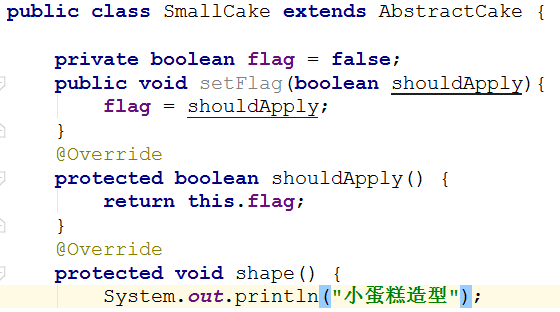




这样一来，不但可以批量生产三种蛋糕，而且如果日后有扩展，只需要继承抽象蛋糕方法就可以了，十分方便，我们天天生意做得越来越赚钱。突然有一天，我们发现市面有一种最简单的小蛋糕销量很好，这种蛋糕就是简单烘烤成型就可以卖，并不需要涂抹什么食材，由于制作简单销售量大，这个品种也很赚钱，于是我们也想要生产这种蛋糕。但是我们发现了一个问题，抽象蛋糕是定义了抽象的涂抹方法的，也就是说扩展的这种蛋糕是必须要实现涂抹方法，这就很鸡儿蛋疼了。怎么办？我们可以将原来的模板修改为带钩子的模板。



做小蛋糕的时候通过flag来控制是否涂抹，其余已有的蛋糕制作不需要任何修改可以照常进行。



### AQS中的方法

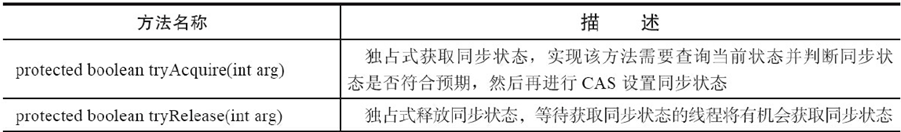
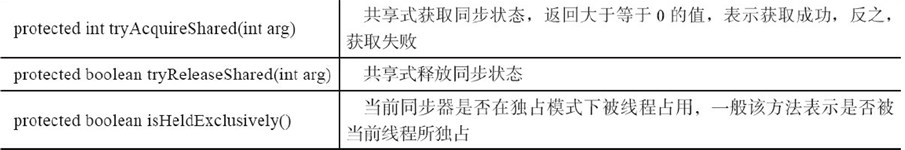
#### 模板方法

实现自定义同步组件时，将会调用同步器提供的模板方法，



这些模板方法同步器提供的模板方法基本上分为3类：独占式获取与释放同步状态、共享式获取与释放、同步状态和查询同步队列中的等待线程情况。

#### 可重写的方法

#### 访问或修改同步状态的方法

重写同步器指定的方法时，需要使用同步器提供的如下3个方法来访问或修改同步状态。

•getState()：获取当前同步状态。

•setState(int newState)：设置当前同步状态。

•compareAndSetState(int expect,int update)：使用CAS设置当前状态，该方法能够保证状态设置的原子性。

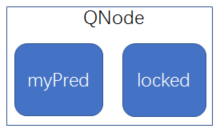
### CLH队列锁

CLH队列锁即Craig, Landin, and Hagersten (CLH) locks。

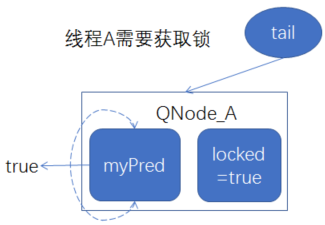
CLH队列锁也是一种基于链表的可扩展、高性能、公平的自旋锁，申请线程仅仅在本地变量上自旋，它不断轮询前驱的状态，假设发现前驱释放了锁就结束自旋。

当一个线程需要获取锁时：

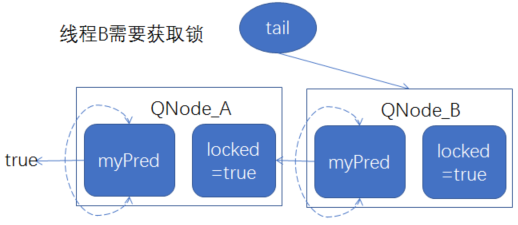
1. 创建一个的QNode，将其中的locked设置为true表示需要获取锁，myPred表示对其前驱结点的引用



1. 线程A对tail域调用getAndSet方法，使自己成为队列的尾部，同时获取一个指向其前驱结点的引用myPred

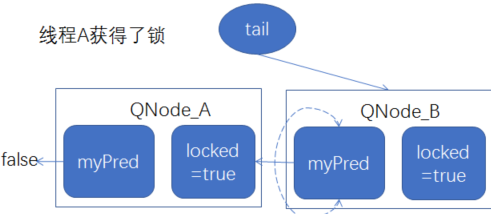


线程B需要获得锁，同样的流程再来一遍



3.线程就在前驱结点的locked字段上旋转，直到前驱结点释放锁(前驱节点的锁值 locked == false)

4.当一个线程需要释放锁时，将当前结点的locked域设置为false，同时回收前驱结点



如上图所示，前驱结点释放锁，线程A的myPred所指向的前驱结点的locked字段变为false，线程A就可以获取到锁。

CLH队列锁的优点是空间复杂度低（如果有n个线程，L个锁，每个线程每次只获取一个锁，那么需要的存储空间是O（L+n），n个线程有n个myNode，L个锁有L个tail）。CLH队列锁常用在SMP体系结构下。

Java中的AQS是CLH队列锁的一种变体实现。

### ReentrantLock的实现

#### 锁的可重入

重进入是指任意线程在获取到锁之后能够再次获取该锁而不会被锁所阻塞，该特性的实现需要解决以下两个问题。

1）线程再次获取锁。锁需要去识别获取锁的线程是否为当前占据锁的线程，如果是，则再次成功获取。

2）锁的最终释放。线程重复n次获取了锁，随后在第n次释放该锁后，其他线程能够获取到该锁。锁的最终释放要求锁对于获取进行计数自增，计数表示当前锁被重复获取的次数，而锁被释放时，计数自减，当计数等于0时表示锁已经成功释放。

nonfairTryAcquire方法增加了再次获取同步状态的处理逻辑：通过判断当前线程是否为获取锁的线程来决定获取操作是否成功，如果是获取锁的线程再次请求，则将同步状态值进行增加并返回true，表示获取同步状态成功。同步状态表示锁被一个线程重复获取的次数。

如果该锁被获取了n次，那么前(n-1)次tryRelease(int releases)方法必须返回false，而只有同步状态完全释放了，才能返回true。可以看到，该方法将同步状态是否为0作为最终释放的条件，当同步状态为0时，将占有线程设置为null，并返回true，表示释放成功。

#### 公平和非公平锁

ReentrantLock的构造函数中，默认的无参构造函数将会把Sync对象创建为NonfairSync对象，这是一个“非公平锁”；而另一个构造函数ReentrantLock(boolean fair)传入参数为true时将会把Sync对象创建为“公平锁”FairSync。

nonfairTryAcquire(int acquires)方法，对于非公平锁，只要CAS设置同步状态成功，则表示当前线程获取了锁，而公平锁则不同。tryAcquire方法，该方法与nonfairTryAcquire(int acquires)比较，唯一不同的位置为判断条件多了hasQueuedPredecessors()方法，即加入了同步队列中当前节点是否有前驱节点的判断，如果该方法返回true，则表示有线程比当前线程更早地请求获取锁，因此需要等待前驱线程获取并释放锁之后才能继续获取锁。

# 深入理解并发编程和归纳总结

## JMM基础-计算机原理

Java内存模型即Java Memory Model，简称JMM。JMM定义了Java 虚拟机(JVM)在计算机内存(RAM)中的工作方式。JVM是整个计算机虚拟模型，所以JMM是隶属于JVM的。Java1.5版本对其进行了重构，现在的Java仍沿用了Java1.5的版本。Jmm遇到的问题与现代计算机中遇到的问题是差不多的。

物理计算机中的并发问题，物理机遇到的并发问题与虚拟机中的情况有不少相似之处，物理机对并发的处理方案对于虚拟机的实现也有相当大的参考意义。

根据《Jeff Dean在Google全体工程大会的报告》我们可以看到



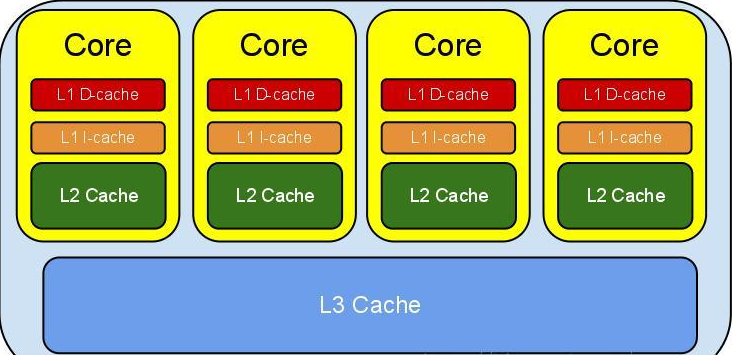
计算机在做一些我们平时的基本操作时，需要的响应时间是不一样的。

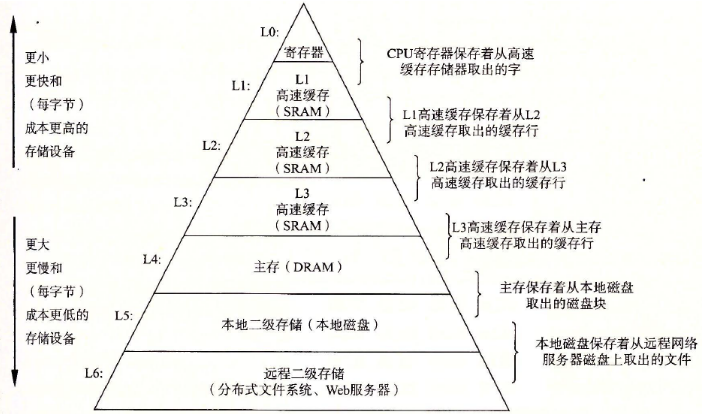
（以下案例仅做说明，并不代表真实情况。）

如果从内存中读取1M的int型数据由CPU进行累加，耗时要多久？

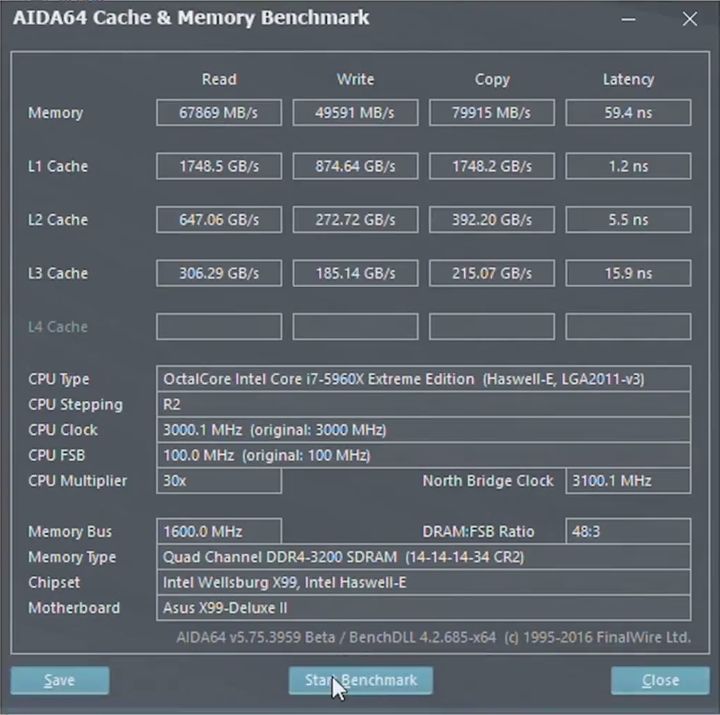
做个简单的计算，1M的数据，Java里int型为32位，4个字节，共有1024\*1024/4 = 262144个整数 ，则CPU 计算耗时：262144 \*0.6 = 157 286 纳秒，而我们知道从内存读取1M数据需要250000纳秒，两者虽然有差距（当然这个差距并不小，十万纳秒的时间足够CPU执行将近二十万条指令了），但是还在一个数量级上。但是，没有任何缓存机制的情况下，意味着每个数都需要从内存中读取，这样加上CPU读取一次内存需要100纳秒，262144个整数从内存读取到CPU加上计算时间一共需要262144\*100+250000 = 26 464 400 纳秒，这就存在着数量级上的差异了。

而且现实情况中绝大多数的运算任务都不可能只靠处理器“计算”就能完成，处理器至少要与内存交互，如读取运算数据、存储运算结果等，这个I/O操作是基本上是无法消除的（无法仅靠寄存器来完成所有运算任务）。早期计算机中cpu和内存的速度是差不多的，但在现代计算机中，cpu的指令速度远超内存的存取速度,由于计算机的存储设备与处理器的运算速度有几个数量级的差距，所以现代计算机系统都不得不加入一层读写速度尽可能接近处理器运算速度的高速缓存（Cache）来作为内存与处理器之间的缓冲：将运算需要使用到的数据复制到缓存中，让运算能快速进行，当运算结束后再从缓存同步回内存之中，这样处理器就无须等待缓慢的内存读写了。





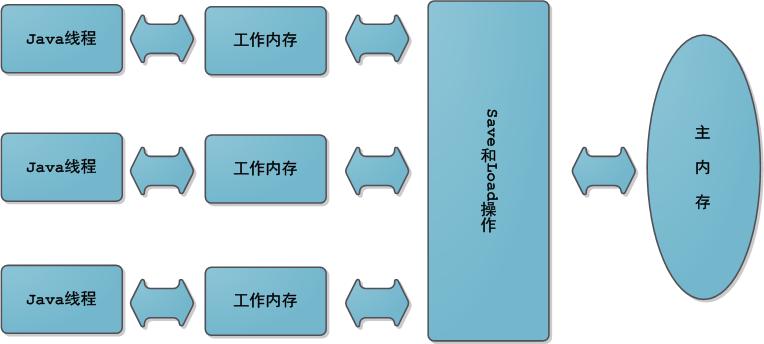
在计算机系统中，寄存器划是L0级缓存，接着依次是L1，L2，L3（接下来是内存，本地磁盘，远程存储）。越往上的缓存存储空间越小，速度越快，成本也更高；越往下的存储空间越大，速度更慢，成本也更低。从上至下，每一层都可以看做是更下一层的缓存，即：L0寄存器是L1一级缓存的缓存，L1是L2的缓存，依次类推；每一层的数据都是来至它的下一层，所以每一层的数据是下一层的数据的子集。

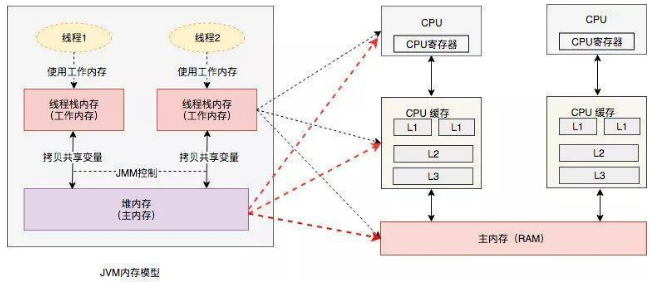


在现代CPU上，一般来说L0， L1，L2，L3都集成在CPU内部，而L1还分为一级数据缓存（Data Cache，D-Cache，L1d）和一级指令缓存（Instruction Cache，I-Cache，L1i），分别用于存放数据和执行数据的指令解码。每个核心拥有独立的运算处理单元、控制器、寄存器、L1、L2缓存，然后一个CPU的多个核心共享最后一层CPU缓存L3

## Java内存模型（JMM）

从抽象的角度来看，JMM定义了线程和主内存之间的抽象关系：线程之间的共享变量存储在主内存（Main Memory）中，每个线程都有一个**私有**的本地内存（Local Memory），本地内存中存储了该线程以读/写共享变量的副本。本地内存是JMM的一个抽象概念，并不真实存在。它涵盖了缓存、写缓冲区、寄存器以及其他的硬件和编译器优化。





### 可见性

可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

由于线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存中的变量，那么对于共享变量V，它们首先是在自己的工作内存，之后再同步到主内存。可是并不会及时的刷到主存中，而是会有一定时间差。很明显，这个时候线程 A 对变量 V 的操作对于线程 B 而言就不具备可见性了 。

要解决共享对象可见性这个问题，我们可以使用volatile关键字或者是加锁。

### 原子性

原子性：即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。

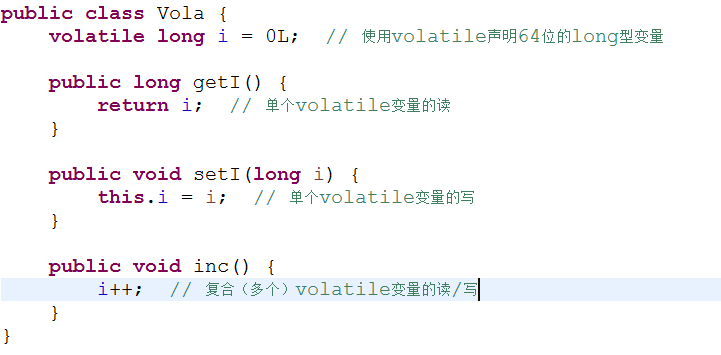
我们都知道CPU资源的分配都是以线程为单位的,并且是分时调用,操作系统允许某个进程执行一小段时间，例如 50 毫秒，过了 50 毫秒操作系统就会重新选择一个进程来执行（我们称为“任务切换”），这个 50 毫秒称为“时间片”。而任务的切换大多数是在时间片段结束以后,

那么线程切换为什么会带来bug呢？因为操作系统做任务切换，可以发生在任何一条CPU 指令执行完！注意，是 CPU 指令，CPU 指令，CPU 指令，而不是高级语言里的一条语句。比如count++，在java里就是一句话，但高级语言里一条语句往往需要多条 CPU 指令完成。其实count++包含了三个CPU指令！

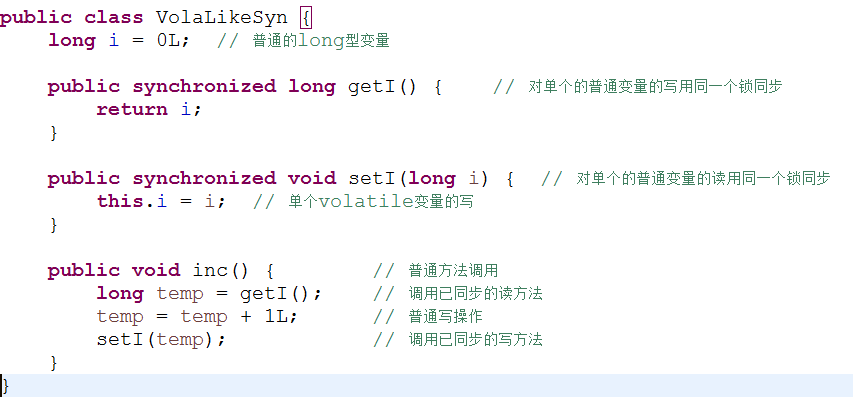
## volatile详解

### volatile特性

可以把对volatile变量的单个读/写，看成是使用同一个锁对这些单个读/写操作做了同步



可以看成



所以volatile变量自身具有下列特性：

可见性。对一个volatile变量的读，总是能看到（任意线程）对这个volatile变量最后的写入。

原子性：对任意单个volatile变量的读/写具有原子性，但类似于volatile++这种复合操作不具有原子性。

volatile虽然能保证执行完及时把变量刷到主内存中，但对于count++这种非原子性、多指令的情况，由于线程切换，线程A刚把count=0加载到工作内存，线程B就可以开始工作了，这样就会导致线程A和B执行完的结果都是1，都写到主内存中，主内存的值还是1不是2

### volatile的实现原理

通过对OpenJDK中的unsafe.cpp源码的分析，会发现被volatile关键字修饰的变量会存在一个“lock:”的前缀。

Lock前缀，Lock不是一种内存屏障，但是它能完成类似内存屏障的功能。Lock会对CPU总线和高速缓存加锁，可以理解为CPU指令级的一种锁。

同时该指令会将当前处理器缓存行的数据直接写会到系统内存中，且这个写回内存的操作会使在其他CPU里缓存了该地址的数据无效。

## synchronized的实现原理

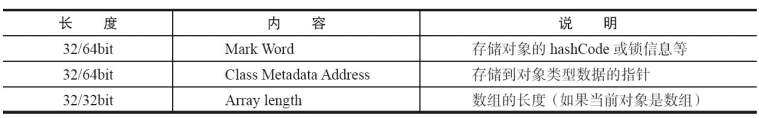
Synchronized在JVM里的实现都是基于进入和退出Monitor对象来实现方法同步和代码块同步，虽然具体实现细节不一样，但是都可以通过成对的MonitorEnter和MonitorExit指令来实现。

对同步块，MonitorEnter指令插入在同步代码块的开始位置，当代码执行到该指令时，将会尝试获取该对象Monitor的所有权，即尝试获得该对象的锁，而monitorExit指令则插入在方法结束处和异常处，JVM保证每个MonitorEnter必须有对应的MonitorExit。

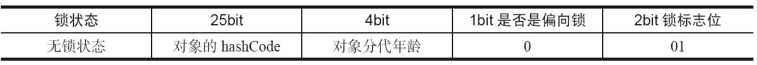
对同步方法，从同步方法反编译的结果来看，方法的同步并没有通过指令monitorenter和monitorexit来实现，相对于普通方法，其常量池中多了ACC\_SYNCHRONIZED标示符。

JVM就是根据该标示符来实现方法的同步的：当方法被调用时，调用指令将会检查方法的 ACC\_SYNCHRONIZED 访问标志是否被设置，如果设置了，执行线程将先获取monitor，获取成功之后才能执行方法体，方法执行完后再释放monitor。在方法执行期间，其他任何线程都无法再获得同一个monitor对象。

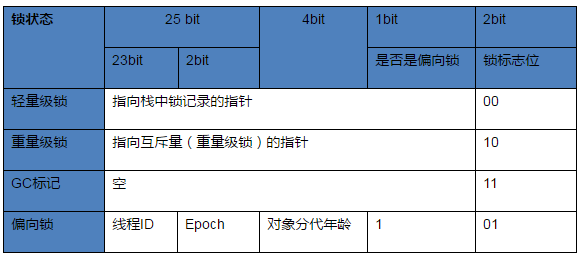
synchronized使用的锁是存放在Java对象头里面，



具体位置是对象头里面的MarkWord，MarkWord里默认数据是存储对象的HashCode等信息，



但是会随着对象的运行改变而发生变化，不同的锁状态对应着不同的记录存储方式



### 了解各种锁

#### 自旋锁

##### 原理

自旋锁原理非常简单，如果持有锁的线程能在很短时间内释放锁资源，那么那些等待竞争锁的线程就不需要做内核态和用户态之间的切换进入阻塞挂起状态，它们只需要等一等（自旋），等持有锁的线程释放锁后即可立即获取锁，这样就避免用户线程和内核的切换的消耗。

但是线程自旋是需要消耗CPU的，说白了就是让CPU在做无用功，线程不能一直占用CPU自旋做无用功，所以需要设定一个自旋等待的最大时间。

如果持有锁的线程执行的时间超过自旋等待的最大时间扔没有释放锁，就会导致其它争用锁的线程在最大等待时间内还是获取不到锁，这时争用线程会停止自旋进入阻塞状态。

##### 自旋锁的优缺点

自旋锁尽可能的减少线程的阻塞，这对于锁的竞争不激烈，且占用锁时间非常短的代码块来说性能能大幅度的提升，因为自旋的消耗会小于线程阻塞挂起操作的消耗！

但是如果锁的竞争激烈，或者持有锁的线程需要长时间占用锁执行同步块，这时候就不适合使用自旋锁了，因为自旋锁在获取锁前一直都是占用cpu做无用功，占着XX不XX，线程自旋的消耗大于线程阻塞挂起操作的消耗，其它需要cup的线程又不能获取到cpu，造成cpu的浪费。

##### 自旋锁时间阈值

自旋锁的目的是为了占着CPU的资源不释放，等到获取到锁立即进行处理。但是如何去选择自旋的执行时间呢？如果自旋执行时间太长，会有大量的线程处于自旋状态占用CPU资源，进而会影响整体系统的性能。因此自旋次数很重要

JVM对于自旋次数的选择，jdk1.5默认为10次，在1.6引入了适应性自旋锁，适应性自旋锁意味着自旋的时间不在是固定的了，而是由前一次在同一个锁上的自旋时间以及锁的拥有者的状态来决定，基本认为一个线程上下文切换的时间是最佳的一个时间。

JDK1.6中-XX:+UseSpinning开启自旋锁； JDK1.7后，去掉此参数，由jvm控制；

### 锁的状态

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，目的是为了提高获得锁和释放锁的效率。

#### 偏向锁

引入背景：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，减少不必要的CAS操作。

偏向锁，顾名思义，它会偏向于第一个访问锁的线程，如果在运行过程中，同步锁只有一个线程访问，不存在多线程争用的情况，则线程是不需要触发同步的，减少加锁／解锁的一些CAS操作（比如等待队列的一些CAS操作），这种情况下，就会给线程加一个偏向锁。 如果在运行过程中，遇到了其他线程抢占锁，则持有偏向锁的线程会被挂起，JVM会消除它身上的偏向锁，将锁恢复到标准的轻量级锁。它通过消除资源无竞争情况下的同步原语，进一步提高了程序的运行性能。

偏向锁获取过程：

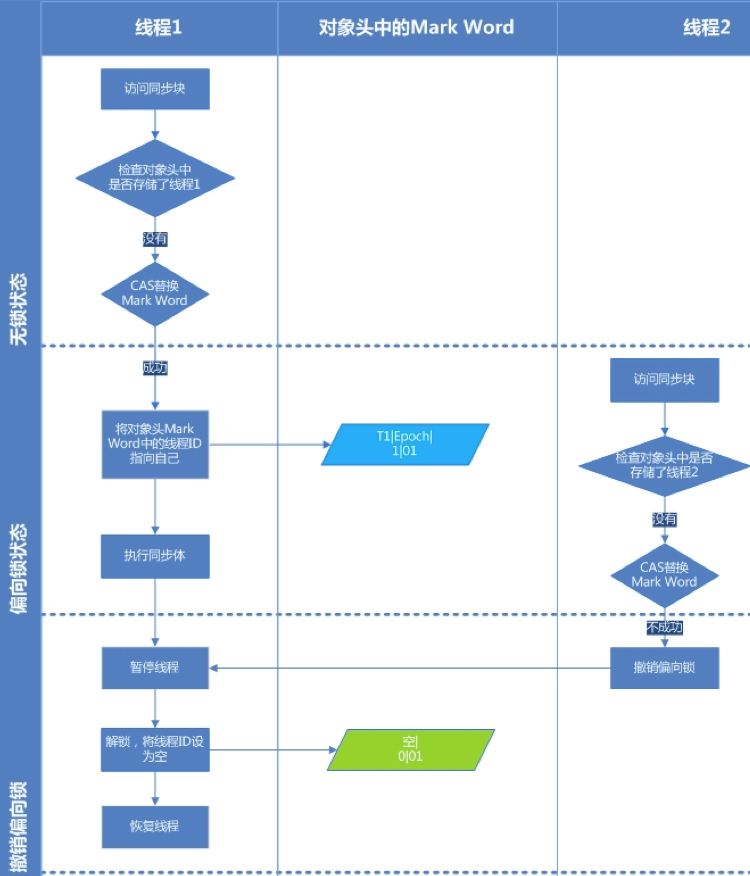
步骤1、 访问Mark Word中偏向锁的标识是否设置成1，锁标志位是否为01，确认为可偏向状态。

步骤2、 如果为可偏向状态，则测试线程ID是否指向当前线程，如果是，进入步骤5，否则进入步骤3。

步骤3、 如果线程ID并未指向当前线程，则通过CAS操作竞争锁。如果竞争成功，则将Mark Word中线程ID设置为当前线程ID，然后执行5；如果竞争失败，执行4。

步骤4、 如果CAS获取偏向锁失败，则表示有竞争。当到达全局安全点（safepoint）时获得偏向锁的线程被挂起，偏向锁升级为轻量级锁，然后被阻塞在安全点的线程继续往下执行同步代码。（撤销偏向锁的时候会导致stop the word）

步骤5、 执行同步代码。



偏向锁的释放：

偏向锁的撤销在上述第四步骤中有提到。偏向锁只有遇到其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放偏向锁，线程不会主动去释放偏向锁。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，判断锁对象是否处于被锁定状态，撤销偏向锁后恢复到未锁定（标志位为“01”）或轻量级锁（标志位为“00”）的状态。

偏向锁的适用场景

始终只有一个线程在执行同步块，在它没有执行完释放锁之前，没有其它线程去执行同步块，在锁无竞争的情况下使用，一旦有了竞争就升级为轻量级锁，升级为轻量级锁的时候需要撤销偏向锁，撤销偏向锁的时候会导致stop the word操作；

在有锁的竞争时，偏向锁会多做很多额外操作，尤其是撤销偏向所的时候会导致进入安全点，安全点会导致stw，导致性能下降，这种情况下应当禁用。

jvm开启/关闭偏向锁

开启偏向锁：-XX:+UseBiasedLocking -XX:BiasedLockingStartupDelay=0

关闭偏向锁：-XX:-UseBiasedLocking

#### 轻量级锁

轻量级锁是由偏向锁升级来的，偏向锁运行在一个线程进入同步块的情况下，当第二个线程加入锁争用的时候，偏向锁就会升级为轻量级锁；

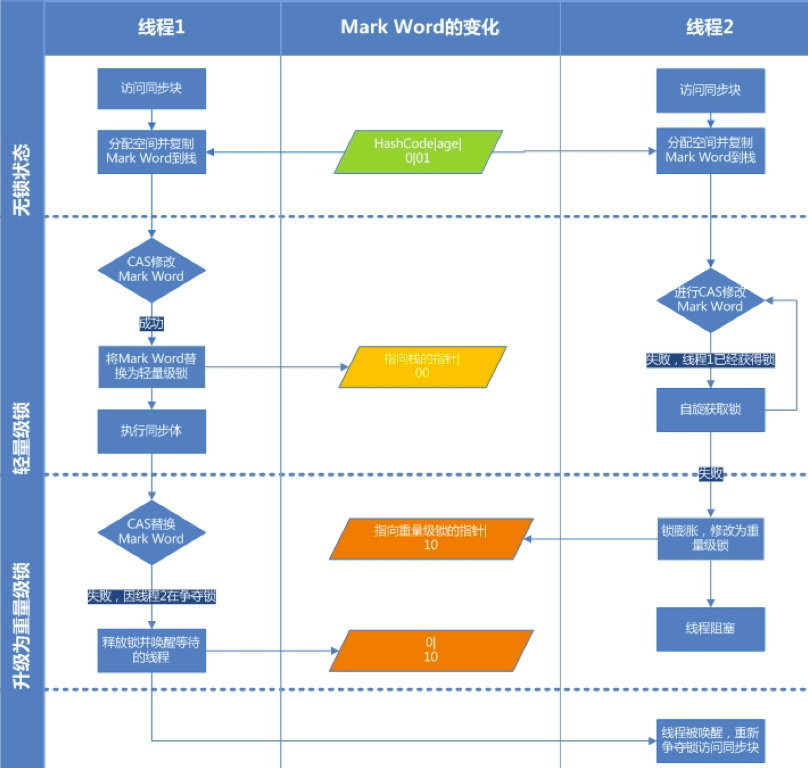
轻量级锁的加锁过程：

在代码进入同步块的时候，如果同步对象锁状态为无锁状态且不允许进行偏向（锁标志位为“01”状态，是否为偏向锁为“0”），虚拟机首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的Mark Word的拷贝，官方称之为 Displaced Mark Word。

拷贝成功后，虚拟机将使用CAS操作尝试将对象的Mark Word更新为指向Lock Record的指针，并将Lock record里的owner指针指向object mark word。如果更新成功，则执行步骤4，否则执行步骤5。

如果这个更新动作成功了，那么这个线程就拥有了该对象的锁，并且对象Mark Word的锁标志位设置为“00”，即表示此对象处于轻量级锁定状态

如果这个更新操作失败了，虚拟机首先会检查对象的Mark Word是否指向当前线程的栈帧，如果是就说明当前线程已经拥有了这个对象的锁，那就可以直接进入同步块继续执行。否则说明多个线程竞争锁，当竞争线程尝试占用轻量级锁失败多次之后，轻量级锁就会膨胀为重量级锁，重量级线程指针指向竞争线程，竞争线程也会阻塞，等待轻量级线程释放锁后唤醒他。锁标志的状态值变为“10”，Mark Word中存储的就是指向重量级锁（互斥量）的指针，后面等待锁的线程也要进入阻塞状态。



#### 不同锁的比较

