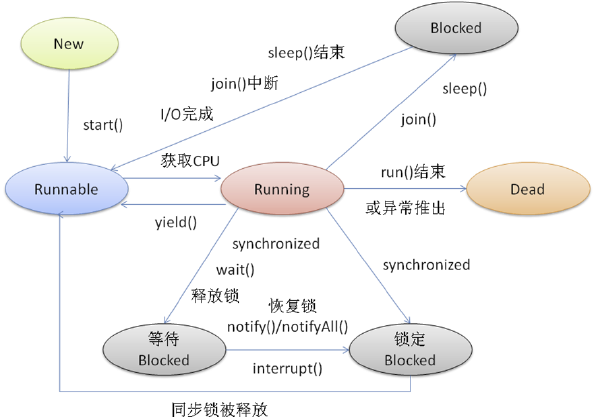
## 线程状态转换



线程在Running的过程中可能会遇到阻塞(Blocked)情况

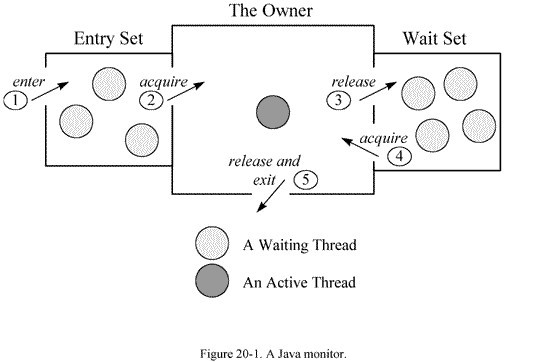
调用join()和sleep()方法，sleep()时间结束或被打断，join()中断,IO完成都会回到Runnable状态，等待JVM的调度。

调用wait()，使该线程处于等待池(wait blocked pool),直到notify()/notifyAll()，线程被唤醒被放到锁定池(lock blocked pool )，释放同步锁使线程回到可运行状态（Runnable）

对Running状态的线程加同步锁(Synchronized)使其进入(lock blocked pool ),同步锁被释放进入可运行状态(Runnable)。

## 线程协作

synchronized, wait, notify 是任何对象都具有的同步工具。让我们先来了解他们



**monitor:**

他们是应用于同步问题的人工线程调度工具。讲其本质，首先就要明确monitor的概念，Java中的每个对象都有一个监视器，来监测并发代码的重入。在非多线程编码时该监视器不发挥作用，反之如果在synchronized 范围内，监视器发挥作用。

wait/notify必须存在于synchronized块中。并且，这三个关键字针对的是同一个监视器（某对象的监视器）。这意味着wait之后，其他线程可以进入同步块执行。

当某代码并不持有监视器的使用权时（如图中5的状态，即脱离同步块）去wait或notify，会抛出java.lang.IllegalMonitorStateException。也包括在synchronized块中去调用另一个对象的wait/notify，因为不同对象的监视器不同，同样会抛出此异常。

### 2.1、线程同步

#### 2.1.1、synchronized

代码块：如下，在多线程环境下，synchronized块中的方法获取了lock实例的monitor，如果实例相同，那么只有一个线程能执行该块内容。

public class Thread1 implements Runnable {

Object lock;

public void run() {

synchronized(lock){

..do something

}

}

}

直接用于方法： 相当于上面代码中用lock来锁定的效果，实际获取的是Thread1类的monitor。更进一步，如果修饰的是static方法，则锁定该类所有实例。

public class Thread1 implements Runnable {

public synchronized void run() {

..do something

}

}

synchronized的执行过程：

1. 检测Mark Word里面是不是当前线程的ID，如果是，表示当前线程处于偏向锁

2. 如果不是，则使用CAS将当前线程的ID替换Mard Word，如果成功则表示当前线程获得偏向锁，置偏向标志位1

3. 如果失败，则说明发生竞争，撤销偏向锁，进而升级为轻量级锁。

4. 当前线程使用CAS将对象头的Mark Word替换为锁记录指针，如果成功，当前线程获得锁

5. 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。

6. 如果自旋成功则依然处于轻量级状态。

7. 如果自旋失败，则升级为重量级锁。

#### 2.1.2、Lock

若要实现更细粒度的控制，我们可以使用锁（ReentrantLock）。

(线程可以进入任何一个它已经拥有的锁所同步着的代码块，叫重入锁。获得了锁，这时就无法再调用加锁方法，这时必须先释放锁才能调用，所以称这种锁为不可重入锁，也叫自旋锁。)

private Lock lock;

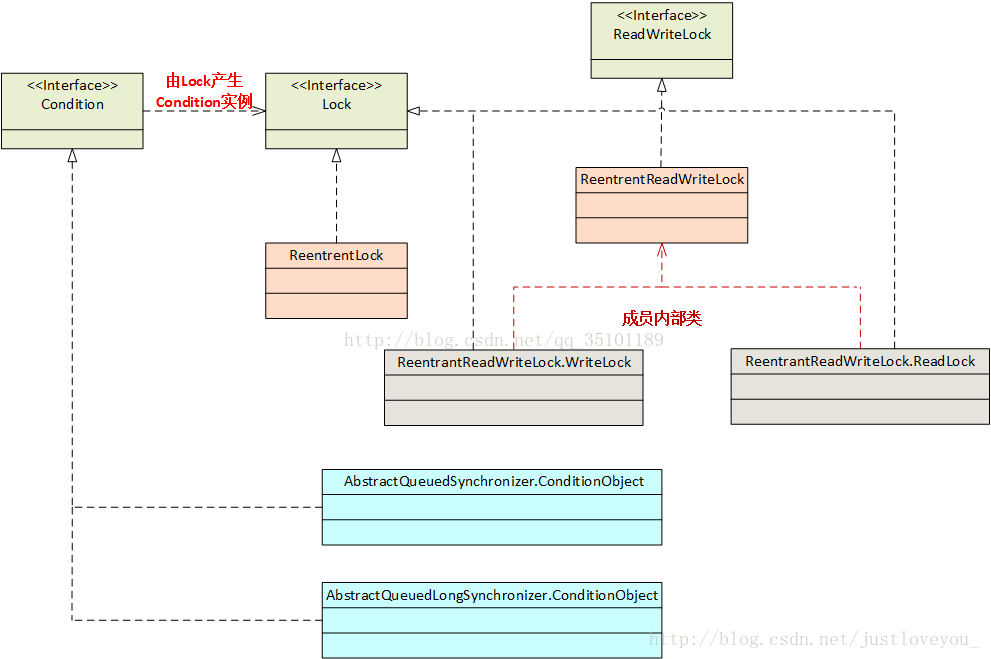
public int func(int value) {

lock.lock();

// ...

lock.unlock();

}



| **类别** | **synchronized** | **Lock** |
| --- | --- | --- |
| 存在层次 | Java的关键字，在jvm层面上 | 是一个类 |
| 锁的释放 | 1、以获取锁的线程执行完同步代码，释放锁 2、线程执行发生异常，jvm会让线程释放锁 | 在finally中必须释放锁，不然容易造成线程死锁 |
| 锁的获取 | 假设A线程获得锁，B线程等待。如果A线程阻塞，B线程会一直等待 | 分情况而定，Lock有多个锁获取的方式，具体下面会说道，大致就是可以尝试获得锁，线程可以不用一直等待 |
| 锁状态 | 无法判断 | 可以判断 |
| 锁类型 | 可重入 不可中断 非公平 | 可重入 可判断 可公平（两者皆可） |
| 性能 | 少量同步 | 大量同步 |

1）Lock是一个接口，而synchronized是Java中的关键字，synchronized是内置的语言实现；

2）synchronized在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生；而Lock在发生异常时，如果没有主动通过unLock()去释放锁，则很可能造成死锁现象，因此使用Lock时需要在finally块中释放锁；

3）Lock可以让等待锁的线程响应中断，而synchronized却不行，使用synchronized时，等待的线程会一直等待下去，不能够响应中断；

4）通过Lock可以知道有没有成功获取锁，而synchronized却无法办到。

5）Lock可以提高多个线程进行读操作的效率。

1. Lock

public interface Lock {

void lock();

void lockInterruptibly() throws InterruptedException;

boolean tryLock();

boolean tryLock(long time, TimeUnit unit) throws InterruptedException;

void unlock();

Condition newCondition();

}

lock()、tryLock()、tryLock(long time, TimeUnit unit)和lockInterruptibly()是用来获取锁的。unLock()方法是用来释放锁的。

如果采用Lock，必须主动去释放锁，并且在发生异常时，不会自动释放锁。

tryLock()方法是有返回值的，它表示用来尝试获取锁，如果获取成功，则返回true，如果获取失败（即锁已被其他线程获取），则返回false，也就说这个方法无论如何都会立即返回。

tryLock(long time, TimeUnit unit)方法和tryLock()方法是类似的，只不过区别在于这个方法在拿不到锁时会等待一定的时间，在时间期限之内如果还拿不到锁，就返回false。如果如果一开始拿到锁或者在等待期间内拿到了锁，则返回true。

lockInterruptibly()方法比较特殊，当通过这个方法去获取锁时，如果线程正在等待获取锁，则这个线程能够响应中断，即中断线程的等待状态。也就使说，当两个线程同时通过lock.lockInterruptibly()想获取某个锁时，假若此时线程A获取到了锁，而线程B只有在等待，那么对线程B调用threadB.interrupt()方法能够中断线程B的等待过程。注意，当一个线程获取了锁之后，是不会被interrupt()方法中断的。因为本身在前面的文章中讲过单独调用interrupt()方法不能中断正在运行过程中的线程，只能中断阻塞过程中的线程。

1. ReentrantLock

ReentrantLock，意思是“可重入锁”，关于可重入锁的概念在下一节讲述。ReentrantLock是唯一实现了Lock接口的类，并且ReentrantLock提供了更多的方法。

1. ReadWriteLock

ReadWriteLock也是一个接口，在它里面只定义了两个方法：Lock readLock();Lock writeLock();一个用来获取读锁，一个用来获取写锁。也就是说将文件的读写操作分开，分成2个锁来分配给线程，从而使得多个线程可以同时进行读操作。

**读锁**：如果没有任何写操作线程锁定 ReadWriteLock，并且没有任何写操作线程要求一个写锁(但还没有获得该锁)。因此，可以有多个读操作线程对该锁进行锁定。

**写锁**：如果没有任何读操作或者写操作。因此，在写操作的时候，只能有一个线程对该锁进行锁定。

1. ReentrantReadWriteLock

ReentrantReadWriteLock里面提供了很多丰富的方法，不过最主要的有两个方法：readLock()和writeLock()用来获取读锁和写锁。

#### 2.1.3、BlockingQueue

java.util.concurrent.BlockingQueue 接口有以下阻塞队列的实现：

FIFO 队列 ：LinkedBlockingQueue、ArrayListBlockingQueue（固定长度）

优先级队列 ：PriorityBlockingQueue

提供了阻塞的 take() 和 put() 方法：如果队列为空 take() 将一直阻塞到队列中有内容，如果队列为满 put() 将阻塞到队列有空闲位置。它们响应中断，当收到中断请求的时候会抛出 InterruptedException，从而提前结束阻塞状态。

### 2.2、线程通信

1、synchronized, wait, notify结合

wait()、notify() 和 notifyAll() 三者实现了线程之间的通信。wait() 会在等待时将线程挂起，而不是忙等待，并且只有在 notify() 或者 notifyAll() 到达时才唤醒。只有在同步控制方法或同步控制块里才能调用 wait() 、notify() 和 notifyAll()。

2、Lock/Condition

使用Lock对像来充当同步监视器,使用Condition对象来暂停指定线程，唤醒指定线程！

3、volatile、AtomicInteger共享变量

volatile修饰的变量值直接存在main memory里面，子线程对该变量的读写直接写入main memory，而不是像其它变量一样在local thread里面产生一份copy。volatile能保证所修饰的变量对于多个线程可见性，即只要被修改，其它线程读到的一定是最新的值。

1. 管道

管道流是JAVA中线程通讯的常用方式之一，基本流程如下：

1）创建管道输出流PipedOutputStream pos和管道输入流PipedInputStream pis

2）将pos和pis匹配，pos.connect(pis);

3）将pos赋给信息输入线程，pis赋给信息获取线程，就可以实现线程间的通讯了

sleep() 和 yield() 并没有释放锁，但是 wait() 会释放锁。实际上，只有在同步控制方法或同步控制块里才能调用 wait() 、notify() 和 notifyAll()。

wait() 和 sleep() 的区别:

wait() 是 Object 类的方法，而 sleep() 是 Thread 的静态方法；

wait() 会放弃锁，而 sleep() 不会。

典型场景生产者消费者问题

/\*\*

\* 生产者生产出来的产品交给店员

\*/

public synchronized void produce()

{

if(this.product >= MAX\_PRODUCT)

{

try

{

wait();

System.out.println("产品已满,请稍候再生产");

}

catch(InterruptedException e)

{

e.printStackTrace();

}

return;

}

this.product++;

System.out.println("生产者生产第" + this.product + "个产品.");

notifyAll(); //通知等待区的消费者可以取出产品了

}

/\*\*

\* 消费者从店员取产品

\*/

public synchronized void consume()

{

if(this.product <= MIN\_PRODUCT)

{

try

{

wait();

System.out.println("缺货,稍候再取");

}

catch (InterruptedException e)

{

e.printStackTrace();

}

return;

}

System.out.println("消费者取走了第" + this.product + "个产品.");

this.product--;

notifyAll(); //通知等待去的生产者可以生产产品了

}

### 2.3、结束线程

#### 2.3.1、阻塞

一个线程进入阻塞状态可能有以下原因：

1、调用 Thread.sleep() 方法进入休眠状态；

2、通过 wait() 使线程挂起，直到线程得到 notify() 或 notifyAll() 消息（或者 java.util.concurrent 类库中等价的 signal() 或 signalAll() 消息；

3、等待某个 I/O 的完成；

4、试图在某个对象上调用其同步控制方法，但是对象锁不可用，因为另一个线程已经获得了这个锁。

#### 2.3.2、中断

使用中断机制即可终止阻塞的线程。

使用 interrupt() 方法来中断某个线程，它会设置线程的中断状态。Object.wait(), Thread.join() 和 Thread.sleep() 三种方法在收到中断请求的时候会清除中断状态，并抛出 InterruptedException。

应当捕获这个 InterruptedException 异常，从而做一些清理资源的操作。

**不可中断的阻塞**

不能中断 I/O 阻塞和 synchronized 锁阻塞。

**Executor 的中断操作**

Executor 避免对 Thread 对象的直接操作，但是使用 interrupt() 方法必须持有 Thread 对象。Executor 使用 shutdownNow() 方法来中断所有它里面的所有线程，shutdownNow() 方法会发送 interrupt() 调用给所有线程。

如果只想中断一个线程，那么使用 Executor 的 submit() 而不是 executor() 来启动线程，就可以持有线程的上下文。submit() 将返回一个泛型 Futrue，可以在它之上调用 cancel()，如果将 true 传递给 cancel()，那么它将会发送 interrupt() 调用给特定的线程。

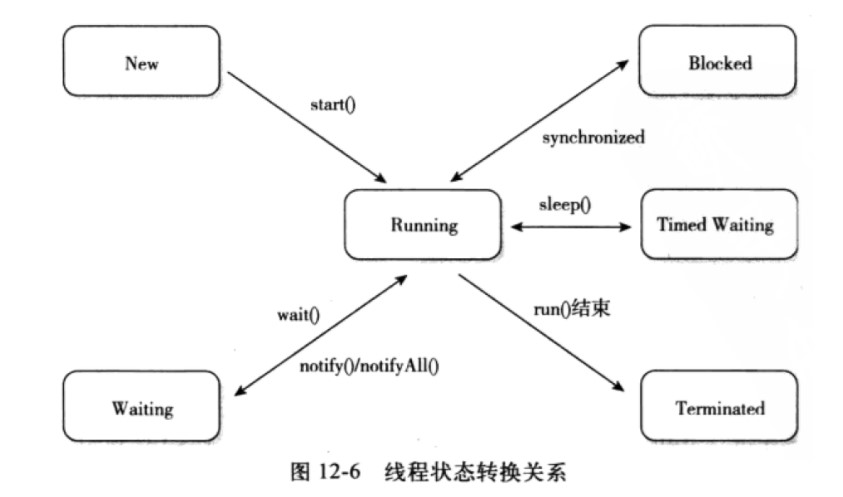
**检查中断**

通过中断的方法来终止线程，需要线程进入阻塞状态才能终止。如果编写的 run() 方法循环条件为 true，但是该线程不发生阻塞，那么线程就永远无法终止。

interrupt() 方法会设置中断状态，可以通过 interrupted() 方法来检查中断状，从而判断一个线程是否已经被中断。

interrupted() 方法在检查完中断状态之后会清除中断状态，这样做是为了确保一次中断操作只会产生一次影响。

线程状态转换：



1、NEW（新建）：创建后尚未启动的线程。

2、RUNNABLE（运行）：处于此状态的线程有可能正在执行，也有可能正在等待着 CPU 为它分配执行时间。

3、BLOCKED（阻塞）：阻塞与等待的区别是，阻塞在等待着获取到一个排它锁，这个时间将在另一个线程放弃这个锁的时候发生；而等待则是在等待一段时间，或者唤醒动作的发生。在程序等待进入同步区域的时候，线程将进入这种状态。

4、Waiting（无限期等待）：处于这种状态的进行不会被分配 CPU 执行时间，它们要等待其它线程显示地唤醒。以下方法会让线程进入这种状态：

5、TIMED\_WAITING（限期等待）：处于这种状态的线程也不会被分配 CPU 执行时间，不过无序等待其它线程显示地唤醒，在一定时间之后它们会由系统自动唤醒。

6、TERMINATED（死亡）

以下方法会让线程陷入无限期的等待状态：

1、没有设置 Timeout 参数的 Object.wait() 方法

2、没有设置 Timeout 参数的 Thread.join() 方法

3、LockSupport.park() 方法

以下方法会让线程进入限期等待状体：

Thread.sleep()

设置了 Timeout 参数的 Object.wait() 方法

设置了 Timeout 参数的 Thread.join() 方法

LockSupport.parkNanos() 方法

LockSupport.parkUntil() 方法

### 2.4、volatile

多线程的内存模型：main memory（主存）、working memory（线程栈），在处理数据时，线程会把值从主存load到本地栈，完成操作后再save回去(volatile关键词的作用：每次针对该变量的操作都激发一次load and save)。

针对多线程使用的变量如果不是volatile或者final修饰的，很有可能产生不可预知的结果（另一个线程修改了这个值，但是之后在某线程看到的是修改之前的值）。其实道理上讲同一实例的同一属性本身只有一个副本。但是多线程是会缓存值的，本质上，volatile就是不去缓存，直接取值。在线程安全的情况下加volatile会牺牲性能。

### 2.5、线程安全实现

1、synchronized加锁

synchronized锁方法: 因为snchroized 定义的方法，在同一时刻只能执行一个。线程在获得该对象的锁后才能够继续执行方法。

synchronized(object)锁对象: 同一时刻只能有一个线程能够获得该对象的锁。

synchronized(this\function)锁对象: 同一时刻只能有一个线程能够获得本实例的锁。

syschronized(class)只对方法区中(static的成员和方法)能够起到同步的作用。在同一时刻只能有一个获得了锁的对象执行这个方法。Class锁对类的所有对象实例起作用。

总结：如果它作用的对象是非静态的，则它取得的锁是对象；如果synchronized作用的对象是一个静态方法或一个类，则它取得的锁是对类，该类所有的对象同一把锁。

2、Lock等加锁(见5.6)

3、不共享状态或不可变对象: String、基本类型的包装类、BigInteger、BigDecimal。

4、线程本地存储ThreadLocal

5、互斥量（wait(),notify()）

6、volatile共享变量不取缓存，轻量级锁

## 线程类

基本线程类指的是Thread类，Runnable接口，Callable接口

### 3.1、Thread

**Thread 类实现了Runnable接口，启动一个线程的方法：**

MyThread my = new MyThread();

　　my.start();

**Thread类相关方法：**

//当前线程可转让cpu控制权，让别的就绪状态线程运行（切换）

public static Thread.yield()

//暂停一段时间

public static Thread.sleep()

//在一个线程中调用other.join(),将等待other执行完后才继续本线程。

public join()

//后两个函数皆可以被打断

public interrupte()

**关于中断**：它并不像stop方法那样会中断一个正在运行的线程。线程会不时地检测中断标识位，以判断线程是否应该被中断（中断标识值是否为true）。终端只会影响到wait状态、sleep状态和join状态。被打断的线程会抛出InterruptedException。  
Thread.interrupted()检查当前线程是否发生中断，返回boolean  
synchronized在获锁的过程中是不能被中断的。

中断是一个状态！interrupt()方法只是将这个状态置为true而已。所以说正常运行的程序不去检测状态，就不会终止，而wait等阻塞方法会去检查并抛出异常。如果在正常运行的程序中添加while(!Thread.interrupted()) ，则同样可以在中断后离开代码体

**Thread类最佳实践**：  
写的时候最好要设置线程名称 Thread.name，并设置线程组 ThreadGroup，目的是方便管理。在出现问题的时候，打印线程栈 (jstack -pid) 一眼就可以看出是哪个线程出的问题，这个线程是干什么的。

### 3.2、Runnable

与Thread类似，实现Runnable接口必须重写其run方法，调用Thread.start()启动线程。Thread类是实现了Runnable接口的。

### 3.3、Callable

实现Callable接口必须重写其call方法，调用Thread.start()启动线程。

实现接口会更好一些，因为：

Java 不支持多重继承，因此继承了 Thread 类就无法继承其它类，但是可以实现多个接口。

类可能只要求可执行即可，继承整个 Thread 类开销会过大。

Runnable和Callable的区别是：

(1)Callable规定的方法是call()，Runnable规定的方法是run()。其中Runnable可以提交给Thread来包装下，直接启动一个线程来执行，而Callable则一般都是提交给ExecuteService来执行。

(2)Callable的任务执行后可返回值，而Runnable的任务是不能返回值得

(3)call方法可以抛出异常，run方法不可以

(4)运行Callable任务可以拿到一个Future对象，表示异步计算的结果。

### 3.4、ThreadLocal

**概述：**

ThreadLocal 是线程的局部变量， 是每一个线程所单独持有的，其他线程不能对其进行访问， 通常是类中的 private static 字段，是对该字段初始值的一个拷贝，它们希望将状态与某一个线程（例如，用户 ID 或事务 ID）相关联

我们知道有时候一个对象的变量会被多个线程所访问，这时就会有线程安全问题，当然我们可以使用synchorinized 关键字来为此变量加锁，进行同步处理，从而限制只能有一个线程来使用此变量，但是加锁会大大影响程序执行效率，此外我们还可以使用ThreadLocal来解决对某一个变量的访问冲突问题。

当使用ThreadLocal维护变量的时候 为每一个使用该变量的线程提供一个独立的变量副本，即每个线程内部都会有一个该变量，这样同时多个线程访问该变量并不会彼此相互影响，因此他们使用的都是自己从内存中拷贝过来的变量的副本， 这样就不存在线程安全问题，也不会影响程序的执行性能。

但是要注意，虽然ThreadLocal能够解决上面说的问题，但是由于在每个线程中都创建了副本，所以要考虑它对资源的消耗，比如内存的占用会比不使用ThreadLocal要大。

**实现：**

ThreadLocal 的几个方法： ThreadLocal 可以存储任何类型的变量对象， get返回的是一个Object对象，但是我们可以通过泛型来制定存储对象的类型。

public T get() { } // 用来获取ThreadLocal在当前线程中保存的变量副本

public void set(T value) { } //set()用来设置当前线程中变量的副本

public void remove() { } //remove()用来移除当前线程中变量的副本

protected T initialValue() { } //initialValue()是一个protected方法，一般是用来在使用时进行重写的

Thread 在内部是通过ThreadLocalMap来维护ThreadLocal变量表， 在Thread类中有一个threadLocals 变量，是ThreadLocalMap类型的，它就是为每一个线程来存储自身的ThreadLocal变量的， ThreadLocalMap是ThreadLocal类的一个内部类，这个Map里面的最小的存储单位是一个Entry， 它使用ThreadLocal作为key， 变量作为 value，这是因为在每一个线程里面，可能存在着多个ThreadLocal变量。Entry(ThreadLocal<?> k, Object v)。

初始时，在Thread里面，threadLocals为空，当通过ThreadLocal变量调用get()方法或者set()方法，就会对Thread类中的threadLocals进行初始化，并且以当前ThreadLocal变量为键值，以ThreadLocal要保存的副本变量为value，存到threadLocals。

然后在当前线程里面，如果要使用副本变量，就可以通过get方法在threadLocals里面查找

**唯一性保证：**

对于每一个ThreadLocal对象，都有一个final修饰的int型的threadLocalHashCode不可变属性，对于基本数据类型，可以认为它在初始化后就不可以进行修改，所以可以唯一确定一个ThreadLocal对象。但是如何保证两个同时实例化的ThreadLocal对象有不同的threadLocalHashCode属性：在ThreadLocal类中，还包含了一个static修饰的AtomicInteger（提供原子操作的Integer类）成员变量（即类变量）和一个static final修饰的常量（作为两个相邻nextHashCode的差值）。由于nextHashCode是类变量，所以每一次调用ThreadLocal类都可以保证nextHashCode被更新到新的值，并且下一次调用ThreadLocal类这个被更新的值仍然可用，同时AtomicInteger保证了nextHashCode自增的原子性。

## 线程调度

线程的调度

1、调整线程优先级：Java线程有优先级，优先级高的线程会获得较多的运行机会。

Java线程的优先级用整数表示，取值范围是1~10，Thread类有以下三个静态常量：

static int MAX\_PRIORITY

线程可以具有的最高优先级，取值为10。

static int MIN\_PRIORITY

线程可以具有的最低优先级，取值为1。

static int NORM\_PRIORITY

分配给线程的默认优先级，取值为5。

Thread类的setPriority()和getPriority()方法分别用来设置和获取线程的优先级。

每个线程都有默认的优先级。主线程的默认优先级为Thread.NORM\_PRIORITY。

线程的优先级有继承关系，比如A线程中创建了B线程，那么B将和A具有相同的优先级。

JVM提供了10个线程优先级，但与常见的操作系统都不能很好的映射。如果希望程序能移植到各个操作系统中，应该仅仅使用Thread类有以下三个静态常量作为优先级，这样能保证同样的优先级采用了同样的调度方式。

2、线程睡眠：Thread.sleep(long millis)方法，使线程转到阻塞状态。millis参数设定睡眠的时间，以毫秒为单位。当睡眠结束后，就转为就绪（Runnable）状态。sleep()平台移植性好。

3、线程等待：Object类中的wait()方法，导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 唤醒方法。这个两个唤醒方法也是Object类中的方法，行为等价于调用 wait(0) 一样。

4、线程让步：Thread.yield() 方法，暂停当前正在执行的线程对象，把执行机会让给相同或者更高优先级的线程。

5、线程加入：join()方法，等待其他线程终止。在当前线程中调用另一个线程的join()方法，则当前线程转入阻塞状态，直到另一个进程运行结束，当前线程再由阻塞转为就绪状态。

6、线程唤醒：Object类中的notify()方法，唤醒在此对象监视器上等待的单个线程。如果所有线程都在此对象上等待，则会选择唤醒其中一个线程。选择是任意性的，并在对实现做出决定时发生。线程通过调用其中一个 wait 方法，在对象的监视器上等待。 直到当前的线程放弃此对象上的锁定，才能继续执行被唤醒的线程。被唤醒的线程将以常规方式与在该对象上主动同步的其他所有线程进行竞争；例如，唤醒的线程在作为锁定此对象的下一个线程方面没有可靠的特权或劣势。类似的方法还有一个notifyAll()，唤醒在此对象监视器上等待的所有线程。

注意：Thread中suspend()和resume()两个方法在JDK1.5中已经废除，不再介绍。因为有死锁倾向。

## 并发工具类concurrent

Executor                  ：具体Runnable任务的执行者。

ExecutorService           ：一个线程池管理者，其实现类有多种，我会介绍一部分。我们能把Runnable,Callable提交到池中让其调度。

Semaphore                 ：一个计数信号量

ReentrantLock             ：一个可重入的互斥锁定 Lock，功能类似synchronized，但要强大的多。

Future                    ：是与Runnable,Callable进行交互的接口，比如一个线程执行结束后取返回的结果等等，还提供了cancel终止线程。

### 5.1、BlockingQueue

阻塞队列。

CompletionService         : ExecutorService的扩展，可以获得线程执行结果的

CountDownLatch            ：一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。

CyclicBarrier             ：一个同步辅助类，它允许一组线程互相等待，直到到达某个公共屏障点

Future                    ：Future 表示异步计算的结果。

ScheduledExecutorService ：一个 ExecutorService，可安排在给定的延迟后运行或定期执行的命令。

### 5.2、ConcurrentHashMap

ConcurrentHashMap 和 java.util.HashTable 类很相似，但 ConcurrentHashMap 能够提供比 HashTable 更好的并发性能。在你从中读取对象的时候 ConcurrentHashMap 并不会把整个 Map 锁住。此外，在你向其中写入对象的时候，ConcurrentHashMap 也不会锁住整个 Map。它的内部只是把 Map 中正在被写入的部分进行锁定。

ConcurrentHashMap使用的是分段锁的概念，把一个大的MAP拆分成几个类似hashtable结构,根据key.hashCode()来决定把key放到哪个HashTable中。就是把Map分成了N个Segment，Segment是一种可重入锁ReentrantLock，put和get的时候，都是现根据key.hashCode()算出放到哪个Segment中，只有在同一个分段内才存在竞态关系。容器里锁的个数也是segments数组的长度。

**ConcurrentHashMap和HashMap的区别：**

(1) ConcurrentHashMap对整个桶数组进行了分段，而HashMap则没有

(2)ConcurrentHashMap在每一个分段上都用锁进行保护，从而让锁的粒度更精细一些，并发性能更好，而HashMap没有锁机制，不是线程安全的

(3)ConcurrentHashMap中的HashEntry相对于HashMap中的Entry有一定的差异性：HashEntry中的value以及next都被volatile修饰，这样在多线程读写过程中能够保持它们的可见性

**ConcurrentHashMap和HashTable的区别**

(1)HashTable的线程安全使用的是一个单独的全部Map范围的锁，ConcurrentHashMap抛弃了HashTable的单锁机制，使用了锁分离技术，使得多个修改操作能够并发进行,只有进行SIZE()操作时ConcurrentHashMap会锁住整张表。

(2)HashTable的put和get方法都是同步方法，  而ConcurrentHashMap的get方法多数情况都不用锁，put方法需要锁。

但是ConcurrentHashMap不能替代HashTable,因为两者的迭代器的一致性不同的，hash table的迭代器是强一致性的，而concurrenthashmap是弱一致的。 ConcurrentHashMap的get，clear，iterator 都是弱一致性的。(要求更新过的数据能被后续的访问都能看到，这是**强一致性**。如果能容忍后续的部分或者全部访问不到，则是**弱一致性**。)

另外一个不同点是，在被遍历的时候，即使是 ConcurrentHashMap 被改动，它也不会抛 ConcurrentModificationException。尽管 Iterator 的设计不是为多个线程的同时使用。更多关于 ConcurrentMap 和 ConcurrentHashMap 的细节请参考官方文档。

### 5.3、CyclicBarrier

两个线程在栅栏旁等待对方。通过调用 CyclicBarrier 对象的 await() 方法，两个线程可以实现互相等待。一旦 N 个线程在等待 CyclicBarrier 达成，所有线程将被释放掉去继续运行。

### 5.4、Semaphore

java.util.concurrent.Semaphore 类是一个计数信号量。这就意味着它具备两个主要方法：

acquire()

release()

计数信号量由一个指定数量的 “许可” 初始化。每调用一次 acquire()，一个许可会被调用线程取走。每调用一次 release()，一个许可会被返还给信号量。因此，在没有任何 release() 调用时，最多有 N 个线程能够通过 acquire() 方法，N 是该信号量初始化时的许可的指定数量。这些许可只是一个简单的计数器。这里没啥奇特的地方。

Semaphore 用法

信号量主要有两种用途：

保护一个重要(代码)部分防止一次超过 N 个线程进入。

在两个线程之间发送信号。

保护重要部分

如果你将信号量用于保护一个重要部分，试图进入这一部分的代码通常会首先尝试获得一个许可，然后才能进入重要部分(代码块)，执行完之后，再把许可释放掉。

比如这样：

Semaphore semaphore = new Semaphore(1);

//critical section

semaphore.acquire();

...

semaphore.release();

### 5.5、ThreadPoolExecutor

java.util.concurrent.ThreadPoolExecutor 是 ExecutorService 接口的一个实现。ThreadPoolExecutor 使用其内部池中的线程执行给定任务(Callable 或者 Runnable)。常用的构造方法有下面四个：

ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue)

ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory)

ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

RejectedExecutionHandler handler)

ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory,

RejectedExecutionHandler handler)

**构造方法参数说明**

* corePoolSize

核心线程数，默认情况下核心线程会一直存活，即使处于闲置状态也不会受存keepAliveTime限制。除非将allowCoreThreadTimeOut设置为true。

* maximumPoolSize

线程池所能容纳的最大线程数。超过这个数的线程将被阻塞。当任务队列为没有设置大小的LinkedBlockingDeque时，这个值无效。

* keepAliveTime

非核心线程的闲置超时时间，超过这个时间就会被回收。

* unit

指定keepAliveTime的单位，如TimeUnit.SECONDS。当将allowCoreThreadTimeOut设置为true时对corePoolSize生效。

* workQueue

线程池中的任务队列.

常用的有三种队列，SynchronousQueue, LinkedBlockingQueue,ArrayBlockingQueue。

SynchronousQueue：

“是这样 一种阻塞队列，其中每个 put 必须等待一个 take，反之亦然。同步队列没有任何内部容量。翻译一下：这是一个内部没有任何容量的阻塞队列，任何一次插入操作的元素都要等待相对的删除/读取操作，否则进行插入操作的线程就要一直等待，反之亦然。

ArrayBlockingQueue：

一个由数组支持的有界阻塞队列。此队列按 FIFO（先进先出）原则对元素进行排序。新元素插入到队列的尾部，队列获取操作则是从队列头部开始获得元素。这是一个典型的“有界缓存区”，固定大小的数组在其中保持生产者插入的元素和使用者提取的元素。一旦创建了这样的缓存区，就不能再增加其容量。试图向已满队列中放入元素会导致操作受阻塞；试图从空队列中提取元素将导致类似阻塞。

LinkedBlockingQueue：

LinkedBlockingQueue是我们在ThreadPoolExecutor线程池中常用的等待队列。它可以指定容量也可以不指定容量。由于它具有“无限容量”的特性，所以我还是将它归入了无限队列的范畴（实际上任何无限容量的队列/栈都是有容量的，这个容量就是Integer.MAX\_VALUE）。

LinkedBlockingQueue的实现是基于链表结构，而不是类似ArrayBlockingQueue那样的数组。但实际使用过程中，不需要关心它的内部实现，如果指定了LinkedBlockingQueue的容量大小，那么它反映出来的使用特性就和ArrayBlockingQueue类似了。

LinkedBlockingDeque:

LinkedBlockingDeque是一个基于链表的双端队列。LinkedBlockingQueue的内部结构决定了它只能从队列尾部插入，从队列头部取出元素；但是LinkedBlockingDeque既可以从尾部插入/取出元素，还可以从头部插入元素/取出元素。

PriorityBlockingQueue:

PriorityBlockingQueue是一个按照优先级进行内部元素排序的无限队列。存放在PriorityBlockingQueue中的元素必须实现Comparable接口，这样才能通过实现compareTo()方法进行排序。优先级最高的元素将始终排在队列的头部；PriorityBlockingQueue不会保证优先级一样的元素的排序，也不保证当前队列中除了优先级最高的元素以外的元素，随时处于正确排序的位置。

LinkedTransferQueue:

LinkedTransferQueue也是一个无限队列，它除了具有一般队列的操作特性外（先进先出），还具有一个阻塞特性：LinkedTransferQueue可以由一对生产者/消费者线程进行操作，当消费者将一个新的元素插入队列后，消费者线程将会一直等待，直到某一个消费者线程将这个元素取走，反之亦然。

* threadFactory:

线程工厂，提供创建新线程的功能。ThreadFactory是一个接口，只有一个方法

public interface ThreadFactory {

Thread newThread(Runnable r);

}

* RejectedExecutionHandler:

RejectedExecutionHandler也是一个接口，只有一个方法

public interface RejectedExecutionHandler {

void rejectedExecution(Runnable var1, ThreadPoolExecutor var2);

}

当线程池中的资源已经全部使用，添加新线程被拒绝时，会调用RejectedExecutionHandler的rejectedExecution方法。

**线程池规则**

线程池的线程执行规则跟任务队列有很大的关系。

* 下面都假设任务队列没有大小限制：
  1. 如果线程数量<=核心线程数量，那么直接启动一个核心线程来执行任务，不会放入队列中。
  2. 如果线程数量>核心线程数，但<=最大线程数，并且任务队列是LinkedBlockingDeque的时候，超过核心线程数量的任务会放在任务队列中排队。
  3. 如果线程数量>核心线程数，但<=最大线程数，并且任务队列是SynchronousQueue的时候，线程池会创建新线程执行任务，这些任务也不会被放在任务队列中。这些线程属于非核心线程，在任务完成后，闲置时间达到了超时时间就会被清除。
  4. 如果线程数量>核心线程数，并且>最大线程数，当任务队列是LinkedBlockingDeque，会将超过核心线程的任务放在任务队列中排队。也就是当任务队列是LinkedBlockingDeque并且没有大小限制时，线程池的最大线程数设置是无效的，他的线程数最多不会超过核心线程数。
  5. 如果线程数量>核心线程数，并且>最大线程数，当任务队列是SynchronousQueue的时候，会因为线程池拒绝添加任务而抛出异常。
* 任务队列大小有限时
  1. 当LinkedBlockingDeque塞满时，新增的任务会直接创建新线程来执行，当创建的线程数量超过最大线程数量时会抛异常。
  2. SynchronousQueue没有数量限制。因为他根本不保持这些任务，而是直接交给线程池去执行。当任务数量超过最大线程数时会直接抛异常。

**线程池状态**

　　在ThreadPoolExecutor中定义了一个volatile变量，另外定义了几个static final变量表示线程池的各个状态：

volatile int runState;

static final int RUNNING = 0;

static final int SHUTDOWN = 1;

static final int STOP = 2;

static final int TERMINATED = 3;

　　runState表示当前线程池的状态，它是一个volatile变量用来保证线程之间的可见性；

　　下面的几个static final变量表示runState可能的几个取值。

　　当创建线程池后，初始时，线程池处于RUNNING状态；

　　如果调用了shutdown()方法，则线程池处于SHUTDOWN状态，此时线程池不能够接受新的任务，它会等待所有任务执行完毕；

　　如果调用了shutdownNow()方法，则线程池处于STOP状态，此时线程池不能接受新的任务，并且会去尝试终止正在执行的任务；

当线程池处于SHUTDOWN或STOP状态，并且所有工作线程已经销毁，任务缓存队列已经清空或执行结束后，线程池被设置为TERMINATED状态。

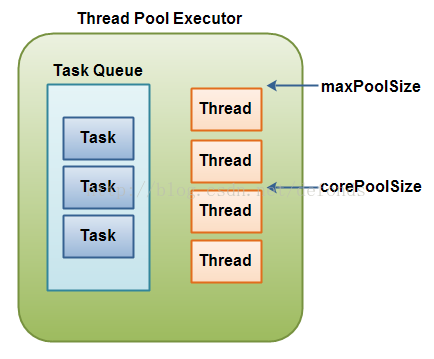
ThreadPoolExecutor 包含的线程池能够包含不同数量的线程。池中线程的数量由以下变量决定：

corePoolSize

maximumPoolSize

当一个任务委托给线程池时，如果池中线程数量低于 corePoolSize，一个新的线程将被创建，即使池中可能尚有空闲线程。如果内部任务队列已满，而且有至少 corePoolSize 正在运行，但是运行线程的数量低于 maximumPoolSize，一个新的线程将被创建去执行该任务。

ThreadPoolExecutor 图解：



创建一个 ThreadPoolExecutor

ThreadPoolExecutor 有若干个可用构造子。比如：

int corePoolSize = 5; //核心线程数，默认情况下核心线程会一直存活

int maxPoolSize = 10; //线程池所能容纳的最大线程数

long keepAliveTime = 5000; //非核心线程的闲置超时时间，超过这个时间就会被回收

TimeUnit unit = TimeUnit.MILLISECOND; //指定keepAliveTime的单位

workQueue //线程池中的任务队列,常用的有三种队列:SynchronousQueue,LinkedBlockingDeque,ArrayBlockingQueue

ExecutorService threadPoolExecutor =

new ThreadPoolExecutor(

corePoolSize,

maxPoolSize,

keepAliveTime,

TimeUnit.MILLISECONDS,

new LinkedBlockingQueue<Runnable>()

);

合理的使用线程池能够带来3个很明显的好处：

1.降低资源消耗：通过重用已经创建的线程来降低线程创建和销毁的消耗

2.提高响应速度：任务到达时不需要等待线程创建就可以立即执行。

3.提高线程的可管理性：线程池可以统一管理、分配、调优和监控。

### 5.6、Lock

既然 Lock 是一个接口，在你的程序里需要使用它的实现类之一来使用它。以下是一个简单重入锁示例：(线程可以进入任何一个它已经拥有的锁所同步着的代码块，叫重入锁。获得了锁，这时就无法再调用加锁方法，这时必须先释放锁才能调用，所以称这种锁为不可重入锁，也叫自旋锁。)

Lock lock = new ReentrantLock();

lock.lock();

//critical section

lock.unlock();

首先创建了一个 Lock 对象。之后调用了它的 lock() 方法。这时候这个 lock 实例就被锁住啦。任何其他再过来调用 lock() 方法的线程将会被阻塞住，直到锁定 lock 实例的线程调用了 unlock() 方法。最后 unlock() 被调用了，lock 对象解锁了，其他线程可以对它进行锁定了。

Java Lock 实现

java.util.concurrent.locks 包提供了以下对 Lock 接口的实现类：

ReentrantLock

Lock 和 synchronized 代码块的主要不同点

一个 Lock 对象和一个 synchronized 代码块之间的主要不同点是：

synchronized 代码块不能够保证进入访问等待的线程的先后顺序。

你不能够传递任何参数给一个 synchronized 代码块的入口。因此，对于 synchronized 代码块的访问等待设置超时时间是不可能的事情。

synchronized 块必须被完整地包含在单个方法里。而一个 Lock 对象可以把它的 lock() 和 unlock() 方法的调用放在不同的方法里。

1）Lock是一个接口，而synchronized是Java中的关键字，synchronized是内置的语言实现；

2）synchronized在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生；而Lock在发生异常时，如果没有主动通过unLock()去释放锁，则很可能造成死锁现象，因此使用Lock时需要在finally块中释放锁；

3）Lock可以让等待锁的线程响应中断，而synchronized却不行，使用synchronized时，等待的线程会一直等待下去，不能够响应中断；

4）通过Lock可以知道有没有成功获取锁，而synchronized却无法办到。

5）Lock可以提高多个线程进行读操作的效率。

Lock 的方法

Lock 接口具有以下主要方法：

lock()

lockInterruptibly()

tryLock()

tryLock(long timeout, TimeUnit timeUnit)

unlock()

lock() 将 Lock 实例锁定。如果该 Lock 实例已被锁定，调用 lock() 方法的线程将会阻塞，直到 Lock 实例解锁。

lockInterruptibly() 方法将会被调用线程锁定，除非该线程被打断。此外，如果一个线程在通过这个方法来锁定 Lock 对象时进入阻塞等待，而它被打断了的话，该线程将会退出这个方法调用。

tryLock() 方法试图立即锁定 Lock 实例。如果锁定成功，它将返回 true，如果 Lock 实例已被锁定该方法返回 false。这一方法永不阻塞。tryLock(long timeout, TimeUnit timeUnit) 的工作类似于 tryLock() 方法，除了它在放弃锁定 Lock 之前等待一个给定的超时时间之外。

unlock() 方法对 Lock 实例解锁。一个 Lock 实现将只允许锁定了该对象的线程来调用此方法。其他(没有锁定该 Lock 对象的线程)线程对 unlock() 方法的调用将会抛一个未检查异常(RuntimeException)。

**ReadWriteLock** :

java.util.concurrent.locks.ReadWriteLock 读写锁是一种先进的线程锁机制。它能够允许多个线程在同一时间对某特定资源进行读取，但同一时间内只能有一个线程对其进行写入。

读写锁的理念在于多个线程能够对一个共享资源进行读取，而不会导致并发问题。并发问题的发生场景在于对一个共享资源的读和写操作的同时进行，或者多个写操作并发进行。

**读锁**：如果没有任何写操作线程锁定 ReadWriteLock，并且没有任何写操作线程要求一个写锁(但还没有获得该锁)。因此，可以有多个读操作线程对该锁进行锁定。

**写锁**：如果没有任何读操作或者写操作。因此，在写操作的时候，只能有一个线程对该锁进行锁定。

java.util.concurrent.locks 包提供了 ReadWriteLock 接口的以下实现类：

ReentrantReadWriteLock。

ReadWriteLock readWriteLock = new ReentrantReadWriteLock();

readWriteLock.readLock().lock();

// multiple readers can enter this section

// if not locked for writing, and not writers waiting

// to lock for writing.

readWriteLock.readLock().unlock();

readWriteLock.writeLock().lock();

// only one writer can enter this section,

// and only if no threads are currently reading.

readWriteLock.writeLock().unlock();

注意如何使用 ReadWriteLock 对两种锁实例的持有。一个对读访问进行保护，一个队写访问进行保护。

### 5.7、AtomicInteger

AtomicInteger 类为我们提供了一个可以进行原子性读和写操作的 int 变量，它还包含一系列先进的原子性操作，比如 compareAndSet()。AtomicInteger 类位于 java.util.concurrent.atomic 包，因此其完整类名为 java.util.concurrent.atomic.AtomicInteger。

AtomicInteger 类包含有一些方法，通过它们你可以增加 AtomicInteger 的值，并获取其值。这些方法如下：

addAndGet()

getAndAdd()

getAndIncrement()

incrementAndGet()

### 5.8、Future

JDK内置的Future主要使用到了Callable接口和FutureTask类。

Callable是类似于Runnable的接口，实现Callable接口的类和实现Runnable的类都是可被其他线程执行的任务。Callable的类型参数是返回值的类型，被线程执行后，可以返回值，这个返回值可以被Future拿到。

Future保存异步计算的结果。实际应用中可以启动一个计算，将Future对象交给某个线程，然后执行其他操作。Future对象的所有者在结果计算好之后就可以获得它。Future接口具有下面的方法：

public interface Future<V> {

boolean cancel(boolean mayInterruptIfRunning);

boolean isCancelled();

boolean isDone();

V get() throws InterruptedException, ExecutionException;

V get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException, TimeoutException;

}

1. cancel方法用来取消任务，如果取消任务成功则返回true，如果取消任务失败则返回false。参数mayInterruptIfRunning表示是否允许取消正在执行却没有执行完毕的任务，如果设置true，则表示可以取消正在执行过程中的任务。如果任务已经完成，则无论mayInterruptIfRunning为true还是false，此方法肯定返回false，即如果取消已经完成的任务会返回false；如果任务正在执行，若mayInterruptIfRunning设置为true，则返回true，若mayInterruptIfRunning设置为false，则返回false；如果任务还没有执行，则无论mayInterruptIfRunning为true还是false，肯定返回true。
2. isCancelled方法表示任务是否被取消成功，如果在任务正常完成前被取消成功，则返回 true。
3. isDone方法表示任务是否已经完成，若任务完成，则返回true；
4. get()方法用来获取执行结果，这个方法会产生阻塞，会一直等到任务执行完毕才返回；
5. get(long timeout, TimeUnit unit)用来获取执行结果，如果在指定时间内，还没获取到结果，就直接返回null。

FutureTask包装器是一种非常便利的机制，同时实现了Future和Runnable接口。FutureTask有2个构造方法：

public FutureTask(Callable<V> callable) {

if (callable == null)

throw new NullPointerException();

this.callable = callable;

this.state = NEW; // ensure visibility of callable

}

public FutureTask(Runnable runnable, V result) {

this.callable = Executors.callable(runnable, result);

this.state = NEW; // ensure visibility of callable

}

通常，我们会使用Callable示例构造一个FutureTask对象，并将它提交给线程池进行处理，下面我们将展示这个内置的Future模式的使用：

public class RealData implements Callable<String> {

private String param;

public RealData(String param){

this.param = param;

}

@Override

public String call() throws Exception {

StringBuffer sb = new StringBuffer();

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

sb.append(param);

try {

Thread.sleep(100);

}catch (InterruptedException e){

}

}

return sb.toString();

}

}

public class FutureMain {

public static void main(String[] args)

throws ExecutionException, InterruptedException {

//构造FutureTask

FutureTask<String> futureTask = new FutureTask<String>(new RealData("xxx"));

ExecutorService executorService = Executors.newFixedThreadPool(1);

//执行FutureTask，发送请求

//在这里开启线程进行RealData的call()执行

executorService.submit(futureTask);

System.out.println("请求完毕。。。");

try {

//这里可以进行其他额外的操作，这里用sleep代替其他业务的处理

Thread.sleep(200);

}catch (InterruptedException e) {

e.printStackTrace();

}

//获取call()方法的返回值

//如果此时call()方法没有执行完成，则依然会等待

System.out.println("真实数据:"+futureTask.get());

}

}

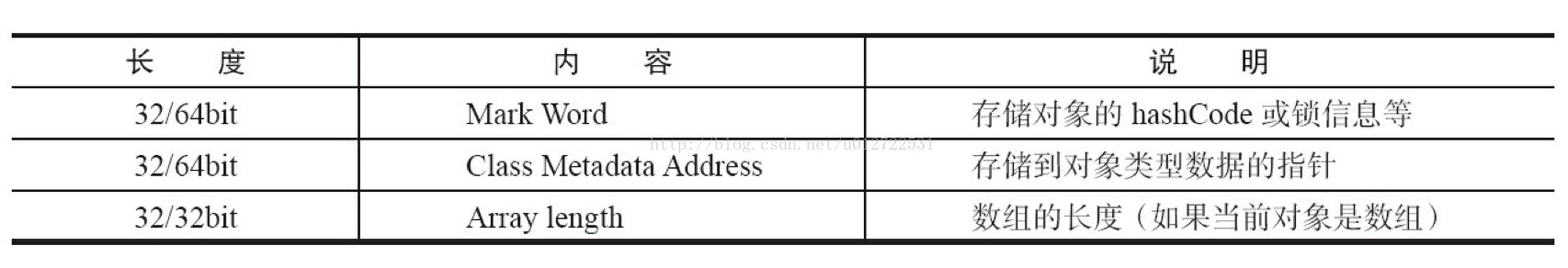
上述代码就是使用Future模式的典型。构造FutureTask时使用Callable接口，告诉FutureTask我们需要的数据应该有返回值。然后将FutureTask提交给线程池，接下来我们不用关心数据是怎么产生的，可以去做其他的业务逻辑处理，然后在需要的时候调用FutureTask.get()得到实际的数据。

## 偏向锁、轻量级锁、自旋锁和重量级锁

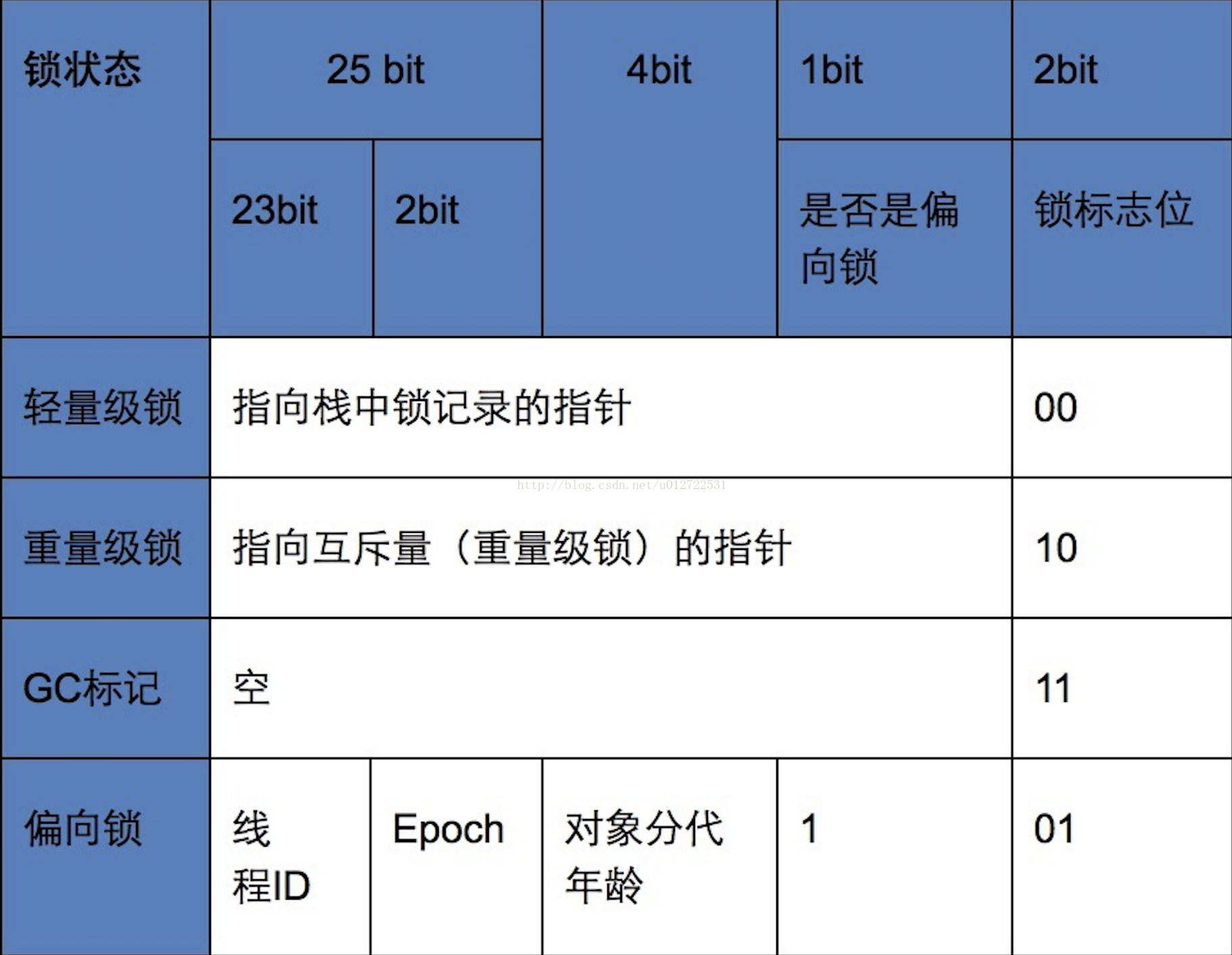
在学习sychronized关键字及其实现细节的时候，发现java中的三种锁，偏向锁，轻量级锁，重量级锁其实也有很多值得探究的地方，引入偏向锁是为了在无多线程竞争的情况下尽量减少不必要的轻量级锁执行路径，因为轻量级锁的获取及释放依赖多次CAS原子指令，而偏向锁只需要在置换ThreadID的时候依赖一次CAS原子指令，代价就是一旦出现多线程竞争的情况就必须撤销偏向锁。

**Java对象头的长度**

Java的对象头通常由两个部分组成，一个是Mark Word存储对象的hashCode或者锁信息,另一个是Class Metadata Address用于存储对象类型数据的指针，如果对象是数组，还会有一个部分存储的是数据的长度



**对象头中Mark Word布局**



偏向锁和轻量级锁是在Java1.6中引入的，并且规定锁只可以升级而不可以降级，这就意味着偏向锁升级成轻量级锁后不能降级成偏向锁。这种策略是为了提高获得锁和释放锁的效率。

### 1、偏向锁

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，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。

偏向锁的来源是因为Hotsopt的作者研究发现大多数情况下，锁不仅不存在多线程竞争，而且总是由统一线程多次获得，而线程的阻塞和唤醒需要CPU从用户态转为核心态，频繁的阻塞和唤醒对CPU来说是一件负担很重的工作，为了让线程获得锁的代驾更低而引入了偏向锁。

偏向锁获得锁的过程分为以下几步：

1）初始时对象的Mark Word位为1，表示对象处于可偏向的状态，并且ThreadId为0，这时该对象是biasable&unbiased状态，可以加上偏向锁进入2）。如果一个线程试图锁住biasable&biased并且ThreadID不等于自己ID的时候，由于锁竞争应该直接进入4）撤销偏向锁。

2）线程尝试用CAS将自己的ThreadID放置到Mark Word中相应的位置，如果CAS操作成功进入到3），否则进入4）

3）进入到这一步代表当前没有锁竞争，Object继续保持biasable状态，但此时ThreadID已经不为0了，对象处于biasable&biased状态

4）当线程执行CAS失败，表示另一个线程当前正在竞争该对象上的锁。当到达全局安全点时（cpu没有正在执行的字节）获得偏向锁的线程将被挂起，撤销偏向（偏向位置0），如果这个线程已经死了，则把对象恢复到未锁定状态（标志位改为01），如果线程还活着，则把偏向锁置0，变成轻量级锁（标志位改为00），释放被阻塞的线程，进入到轻量级锁的执行路径中，同时被撤销偏向锁的线程继续往下执行。

5）运行同步代码块

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。

参考文章：<http://www.cnblogs.com/javaminer/p/3892288.html?utm_source=tuicool&utm_medium=referral>

### 2、轻量级锁

如果说偏向锁是只允许一个线程获得锁，那么轻量级锁就是允许多个线程获得锁，但是只允许他们顺序拿锁，不允许出现竞争，也就是拿锁失败的情况，轻量级锁的步骤如下：

1）线程1在执行同步代码块之前，JVM会先在当前线程的栈帧中创建一个空间用来存储锁记录，然后再把对象头中的Mark Word复制到该锁记录中，官方称之为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，则获得锁，进入步骤3）。如果失败执行步骤2）

2）线程自旋，自旋成功则获得锁，进入步骤3）。自旋失败，则膨胀成为重量级锁，并把锁标志位变为10，线程阻塞进入步骤3）

3）锁的持有线程执行同步代码，执行完CAS替换Mark Word成功释放锁，如果CAS成功则流程结束，CAS失败执行步骤4）

4）CAS执行失败说明期间有线程尝试获得锁并自旋失败，轻量级锁升级为了重量级锁，此时释放锁之后，还要唤醒等待的线程

轻量级锁的释放：

释放锁线程视角：由轻量锁切换到重量锁，是发生在轻量锁释放锁的期间，之前在获取锁的时候它拷贝了锁对象头的markword，在释放锁的时候如果它发现在它持有锁的期间有其他线程来尝试获取锁了，并且该线程对markword做了修改，两者比对发现不一致，则切换到重量锁。因为重量级锁被修改了，所有display mark word和原来的markword不一样了。

怎么补救，就是进入mutex前，compare一下obj的markword状态。确认该markword是否被其他线程持有。此时如果线程已经释放了markword，那么通过CAS后就可以直接进入线程，无需进入mutex，就这个作用。

尝试获取锁线程视角：如果线程尝试获取锁的时候，轻量锁正被其他线程占有，那么它就会修改markword，修改重量级锁，表示该进入重量锁了。

还有一个注意点：等待轻量锁的线程不会阻塞，它会一直自旋等待锁，并如上所说修改markword。这就是自旋锁，尝试获取锁的线程，在没有获得锁的时候，不被挂起，而转而去执行一个空循环，即自旋。在若干个自旋后，如果还没有获得锁，则才被挂起，获得锁，则执行代码。

### 3、自旋锁

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cup的，说白了就是让cup在做无用功，如果一直获取不到锁，那线程也不能一直占用cup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仍然没有释放锁，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再唤醒的操作的消耗，这些操作会导致线程发生两次上下文切换！

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，同时有大量线程在竞争一个锁，会导致获取锁的时间很长，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。所以这种情况下我们要关闭自旋锁；

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。因此自旋的周期选的额外重要！

JVM对于自旋周期的选择，jdk1.5这个限度是一定的写死的，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，同时JVM还针对当前CPU的负荷情况做了较多的优化

1.如果平均负载小于CPUs则一直自旋

2.如果有超过(CPUs/2)个线程正在自旋，则后来线程直接阻塞

3.如果正在自旋的线程发现Owner发生了变化则延迟自旋时间（自旋计数）或进入阻塞

4.如果CPU处于节电模式则停止自旋

5.自旋时间的最坏情况是CPU的存储延迟（CPU A存储了一个数据，到CPU B得知这个数据直接的时间差）

6.自旋时会适当放弃线程优先级之间的差异

自旋锁的开启：

JDK1.6中-XX:+UseSpinning开启；

-XX:PreBlockSpin=10 为自旋次数；

JDK1.7后，去掉此参数，由jvm控制

### 4、重量级锁Synchronized、lock

**Synchronized的作用**

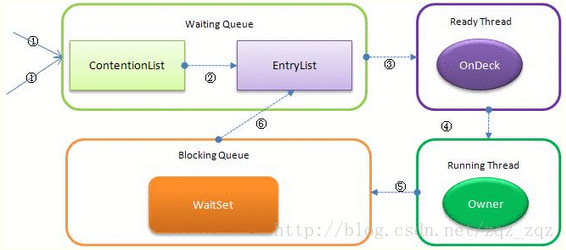
在JDK1.5之前都是使用synchronized关键字保证同步的，Synchronized的作用相信大家都已经非常熟悉了；

它可以把任意一个非NULL的对象当作锁。

1. 作用于方法时，锁住的是对象的实例(this)；
2. 当作用于静态方法时，锁住的是Class实例，又因为Class的相关数据存储在永久带PermGen（jdk1.8则是metaspace），永久带是全局共享的，因此静态方法锁相当于类的一个全局锁，会锁所有调用该方法的线程；
3. synchronized作用于一个对象实例时，锁住的是所有以该对象为锁的代码块。

Synchronized的实现

实现如下图所示；



它有多个队列，当多个线程一起访问某个对象监视器的时候，对象监视器会将这些线程存储在不同的容器中。

1. Contention List：竞争队列，所有请求锁的线程首先被放在这个竞争队列中；
2. Entry List：Contention List中那些有资格成为候选资源的线程被移动到Entry List中；
3. Wait Set：哪些调用wait方法被阻塞的线程被放置在这里；
4. OnDeck：任意时刻，最多只有一个线程正在竞争锁资源，该线程被成为OnDeck；
5. Owner：当前已经获取到所资源的线程被称为Owner；
6. !Owner：当前释放锁的线程。

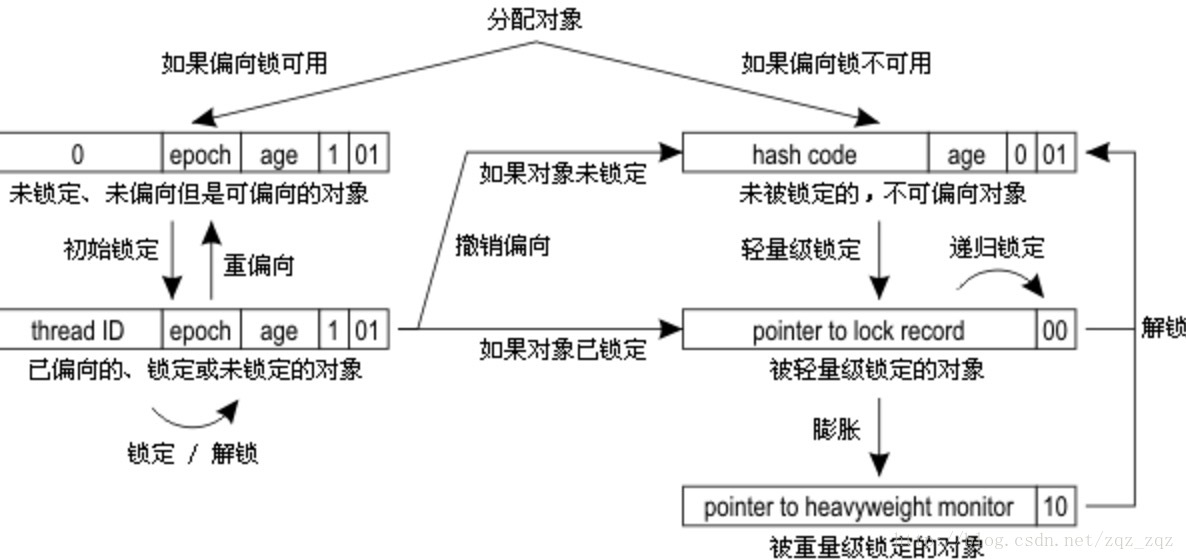
JVM每次从队列的尾部取出一个数据用于锁竞争候选者（OnDeck），但是并发情况下，ContentionList会被大量的并发线程进行CAS访问，为了降低对尾部元素的竞争，JVM会将一部分线程移动到EntryList中作为候选竞争线程。Owner线程会在unlock时，将ContentionList中的部分线程迁移到EntryList中，并指定EntryList中的某个线程为OnDeck线程（一般是最先进去的那个线程）。Owner线程并不直接把锁传递给OnDeck线程，而是把锁竞争的权利交给OnDeck，OnDeck需要重新竞争锁。这样虽然牺牲了一些公平性，但是能极大的提升系统的吞吐量，在JVM中，也把这种选择行为称之为“竞争切换”。

OnDeck线程获取到锁资源后会变为Owner线程，而没有得到锁资源的仍然停留在EntryList中。如果Owner线程被wait方法阻塞，则转移到WaitSet队列中，直到某个时刻通过notify或者notifyAll唤醒，会重新进去EntryList中。

处于ContentionList、EntryList、WaitSet中的线程都处于阻塞状态，该阻塞是由操作系统来完成的（Linux内核下采用pthread\_mutex\_lock内核函数实现的）。

Synchronized是非公平锁。 Synchronized在线程进入ContentionList时，等待的线程会先尝试自旋获取锁，如果获取不到就进入ContentionList，这明显对于已经进入队列的线程是不公平的，还有一个不公平的事情就是自旋获取锁的线程还可能直接抢占OnDeck线程的锁资源。

**总结：**



## 分布式锁

**什么是锁？**

* 在单进程的系统中，当存在多个线程可以同时改变某个变量（可变共享变量）时，就需要对变量或代码块做同步，使其在修改这种变量时能够线性执行消除并发修改变量。
* 而同步的本质是通过锁来实现的。为了实现多个线程在一个时刻同一个代码块只能有一个线程可执行，那么需要在某个地方做个标记，这个标记必须每个线程都能看到，当标记不存在时可以设置该标记，其余后续线程发现已经有标记了则等待拥有标记的线程结束同步代码块取消标记后再去尝试设置标记。这个标记可以理解为锁。
* 不同地方实现锁的方式也不一样，只要能满足所有线程都能看得到标记即可。如 Java 中 synchronize 是在对象头设置标记，Lock 接口的实现类基本上都只是某一个 volitile 修饰的 int 型变量其保证每个线程都能拥有对该 int 的可见性和原子修改，linux 内核中也是利用互斥量或信号量等内存数据做标记。
* 除了利用内存数据做锁其实任何互斥的都能做锁（只考虑互斥情况），如流水表中流水号与时间结合做幂等校验可以看作是一个不会释放的锁，或者使用某个文件是否存在作为锁等。只需要满足在对标记进行修改能保证原子性和内存可见性即可。

**什么是分布式？**

分布式的 CAP 理论告诉我们:

任何一个分布式系统都无法同时满足一致性（Consistency）、可用性（Availability）和分区容错性（Partition tolerance），最多只能同时满足两项。

目前很多大型网站及应用都是分布式部署的，分布式场景中的数据一致性问题一直是一个比较重要的话题。基于 CAP理论，很多系统在设计之初就要对这三者做出取舍。在互联网领域的绝大多数的场景中，都需要牺牲强一致性来换取系统的高可用性，系统往往只需要保证最终一致性。

分布式场景

此处主要指集群模式下，多个相同服务同时开启.

在许多的场景中，我们为了保证数据的最终一致性，需要很多的技术方案来支持，比如分布式事务、分布式锁等。很多时候我们需要保证一个方法在同一时间内只能被同一个线程执行。在单机环境中，通过 Java 提供的并发 API 我们可以解决，但是在分布式环境下，就没有那么简单啦。

* 分布式与单机情况下最大的不同在于其不是多线程而是多进程。
* 多线程由于可以共享堆内存，因此可以简单的采取内存作为标记存储位置。而进程之间甚至可能都不在同一台物理机上，因此需要将标记存储在一个所有进程都能看到的地方。
* **什么是分布式锁？**
* 当在分布式模型下，数据只有一份（或有限制），此时需要利用锁的技术控制某一时刻修改数据的进程数。
* 与单机模式下的锁不仅需要保证进程可见，还需要考虑进程与锁之间的网络问题。（我觉得分布式情况下之所以问题变得复杂，主要就是需要考虑到网络的延时和不可靠。。。一个大坑）
* 分布式锁还是可以将标记存在内存，只是该内存不是某个进程分配的内存而是公共内存如 Redis、Memcache。至于利用数据库、文件等做锁与单机的实现是一样的，只要保证标记能互斥就行。

**我们需要怎样的分布式锁？**

* 可以保证在分布式部署的应用集群中，同一个方法在同一时间只能被一台机器-上的一个线程执行。
* 这把锁要是一把可重入锁（避免死锁）
* 这把锁最好是一把阻塞锁（根据业务需求考虑要不要这条）
* 这把锁最好是一把公平锁（根据业务需求考虑要不要这条）
* 有高可用的获取锁和释放锁功能
* 获取锁和释放锁的性能要好

### 8.1基于数据库做分布式锁

基于乐观锁

**基于表主键唯一做分布式锁**

**思路：**利用主键唯一的特性，如果有多个请求同时提交到数据库的话，数据库会保证只有一个操作可以成功，那么我们就可以认为操作成功的那个线程获得了该方法的锁，当方法执行完毕之后，想要释放锁的话，删除这条数据库记录即可。

上面这种简单的实现有以下几个问题：

* 这把锁强依赖数据库的可用性，数据库是一个单点，一旦数据库挂掉，会导致业务系统不可用。
* 这把锁没有失效时间，一旦解锁操作失败，就会导致锁记录一直在数据库中，其他线程无法再获得到锁。
* 这把锁只能是非阻塞的，因为数据的 insert操作，一旦插入失败就会直接报错。没有获得锁的线程并不会进入排队队列，要想再次获得锁就要再次触发获得锁操作。
* 这把锁是非重入的，同一个线程在没有释放锁之前无法再次获得该锁。因为数据中数据已经存在了。
* 这把锁是非公平锁，所有等待锁的线程凭运气去争夺锁。
* 在 MySQL 数据库中采用主键冲突防重，在大并发情况下有可能会造成锁表现象。

**当然，我们也可以有其他方式解决上面的问题。**

* 数据库是单点？搞两个数据库，数据之前双向同步，一旦挂掉快速切换到备库上。
* 没有失效时间？只要做一个定时任务，每隔一定时间把数据库中的超时数据清理一遍。
* 非阻塞的？搞一个 while 循环，直到 insert 成功再返回成功。
* 非重入的？在数据库表中加个字段，记录当前获得锁的机器的主机信息和线程信息，那么下次再获取锁的时候先查询数据库，如果当前机器的主机信息和线程信息在数据库可以查到的话，直接把锁分配给他就可以了。
* 非公平的？再建一张中间表，将等待锁的线程全记录下来，并根据创建时间排序，只有最先创建的允许获取锁。
* 比较好的办法是在程序中生产主键进行防重。

**基于表字段版本号做分布式锁**

这个策略源于 mysql 的 mvcc 机制，使用这个策略其实本身没有什么问题，唯一的问题就是对数据表侵入较大，我们要为每个表设计一个版本号字段，然后写一条判断 sql 每次进行判断，增加了数据库操作的次数，在高并发的要求下，对数据库连接的开销也是无法忍受的。

基于悲观锁

**基于数据库排他锁做分布式锁**

在查询语句后面增加for update，数据库会在查询过程中给数据库表增加排他锁 (注意： InnoDB 引擎在加锁的时候，只有通过索引进行检索的时候才会使用行级锁，否则会使用表级锁。这里我们希望使用行级锁，就要给要执行的方法字段名添加索引，值得注意的是，这个索引一定要创建成唯一索引，否则会出现多个重载方法之间无法同时被访问的问题。重载方法的话建议把参数类型也加上。)。当某条记录被加上排他锁之后，其他线程无法再在该行记录上增加排他锁。  
  
我们可以认为获得排他锁的线程即可获得分布式锁，当获取到锁之后，可以执行方法的业务逻辑，执行完方法之后，通过connection.commit()操作来释放锁。  
  
这种方法可以有效的解决上面提到的无法释放锁和阻塞锁的问题。

* 阻塞锁？ for update语句会在执行成功后立即返回，在执行失败时一直处于阻塞状态，直到成功。
* 锁定之后服务宕机，无法释放？使用这种方式，服务宕机之后数据库会自己把锁释放掉。

但是还是无法直接解决数据库单点和可重入问题。

这里还可能存在另外一个问题，虽然我们对方法字段名使用了唯一索引，并且显示使用 for update 来使用行级锁。但是，MySQL 会对查询进行优化，即便在条件中使用了索引字段，但是否使用索引来检索数据是由 MySQL 通过判断不同执行计划的代价来决定的，如果 MySQL 认为全表扫效率更高，比如对一些很小的表，它就不会使用索引，这种情况下 InnoDB 将使用表锁，而不是行锁。如果发生这种情况就悲剧了。。。

还有一个问题，就是我们要使用排他锁来进行分布式锁的 lock，那么一个排他锁长时间不提交，就会占用数据库连接。一旦类似的连接变得多了，就可能把数据库连接池撑爆。

**优缺点**

**优点**：简单，易于理解  
  
  
**缺点**：会有各种各样的问题（操作数据库需要一定的开销，使用数据库的行级锁并不一定靠谱，性能不靠谱）

### 8.2基于 Redis 做分布式锁

**基于 REDIS 的 SETNX()、EXPIRE() 方法做分布式锁**

**setnx()**

setnx 的含义就是 SET if Not Exists，其主要有两个参数 setnx(key, value)。该方法是原子的，如果 key 不存在，则设置当前 key 成功，返回 1；如果当前 key 已经存在，则设置当前 key 失败，返回 0。

**expire()**

expire 设置过期时间，要注意的是 setnx 命令不能设置 key 的超时时间，只能通过 expire() 来对 key 设置。

**使用步骤**

1、setnx(lockkey, 1) 如果返回 0，则说明占位失败；如果返回 1，则说明占位成功

2、expire() 命令对 lockkey 设置超时时间，为的是避免死锁问题。

3、执行完业务代码后，可以通过 delete 命令删除 key。

这个方案其实是可以解决日常工作中的需求的，但从技术方案的探讨上来说，可能还有一些可以完善的地方。**比如，如果在第一步 setnx 执行成功后，在 expire() 命令执行成功前，发生了宕机的现象，那么就依然会出现死锁的问题，所以如果要对其进行完善的话，可以使用 redis 的 setnx()、get() 和 getset() 方法来实现分布式锁。**

**基于 REDIS 的 SETNX()、GET()、GETSET()方法做分布式锁**

这个方案的背景主要是在 setnx() 和 expire() 的方案上针对可能存在的死锁问题，做了一些优化。

**getset()**

这个命令主要有两个参数 getset(key，newValue)。该方法是原子的，对 key 设置 newValue 这个值，并且返回 key 原来的旧值。假设 key 原来是不存在的，那么多次执行这个命令，会出现下边的效果：

* getset(key, “value1”) 返回 null 此时 key 的值会被设置为 value1
* getset(key, “value2”) 返回 value1 此时 key 的值会被设置为 value2
* 依次类推！

**使用步骤**

* setnx(lockkey, 当前时间+过期超时时间)，如果返回 1，则获取锁成功；如果返回 0 则没有获取到锁，转向 2。
* get(lockkey) 获取值 oldExpireTime ，并将这个 value 值与当前的系统时间进行比较，如果小于当前系统时间，则认为这个锁已经超时，可以允许别的请求重新获取，转向 3。
* 计算 newExpireTime = 当前时间+过期超时时间，然后 getset(lockkey, newExpireTime) 会返回当前 lockkey 的值currentExpireTime。
* 判断 currentExpireTime 与 oldExpireTime 是否相等，如果相等，说明当前 getset 设置成功，获取到了锁。如果不相等，说明这个锁又被别的请求获取走了，那么当前请求可以直接返回失败，或者继续重试。
* 在获取到锁之后，当前线程可以开始自己的业务处理，当处理完毕后，比较自己的处理时间和对于锁设置的超时时间，如果小于锁设置的超时时间，则直接执行 delete 释放锁；如果大于锁设置的超时时间，则不需要再锁进行处理。

import cn.com.tpig.cache.redis.RedisService;

import cn.com.tpig.utils.SpringUtils;

//redis分布式锁

public final class RedisLockUtil {

private static final int defaultExpire = 60;

private RedisLockUtil() {

//

}

/\*\*

\* 加锁

\* @param key redis key

\* @param expire 过期时间，单位秒

\* @return true:加锁成功，false，加锁失败

\*/

public static boolean lock(String key, int expire) {

RedisService redisService = SpringUtils.getBean(RedisService.class);

long status = redisService.setnx(key, "1");

if(status == 1) {

redisService.expire(key, expire);

return true;

}

return false;

}

public static boolean lock(String key) {

return lock2(key, defaultExpire);

}

/\*\*

\* 加锁

\* @param key redis key

\* @param expire 过期时间，单位秒

\* @return true:加锁成功，false，加锁失败

\*/

public static boolean lock2(String key, int expire) {

RedisService redisService = SpringUtils.getBean(RedisService.class);

long value = System.currentTimeMillis() + expire;

long status = redisService.setnx(key, String.valueOf(value));

if(status == 1) {

return true;

}

long oldExpireTime = Long.parseLong(redisService.get(key, "0"));

if(oldExpireTime < System.currentTimeMillis()) {

//超时

long newExpireTime = System.currentTimeMillis() + expire;

long currentExpireTime = Long.parseLong(redisService.getSet(key, String.valueOf(newExpireTime)));

if(currentExpireTime == oldExpireTime) {

return true;

}

}

return false;

}

public static void unLock1(String key) {

RedisService redisService = SpringUtils.getBean(RedisService.class);

redisService.del(key);

}

public static void unLock2(String key) {

RedisService redisService = SpringUtils.getBean(RedisService.class);

long oldExpireTime = Long.parseLong(redisService.get(key, "0"));

if(oldExpireTime > System.currentTimeMillis()) {

redisService.del(key);

}

}

}

public void drawRedPacket(long userId) {

String key = "draw.redpacket.userid:" + userId;

boolean lock = RedisLockUtil.lock2(key, 60);

if(lock) {

try {

//领取操作

} finally {

//释放锁

RedisLockUtil.unLock(key);

}

} else {

new RuntimeException("重复领取奖励");

}

}

### 8.3基于 REDLOCK 做分布式锁

Redlock 是 Redis 的作者 antirez 给出的集群模式的 Redis 分布式锁，它基于 N 个完全独立的 Redis 节点（通常情况下 N 可以设置成 5）。

算法的步骤如下：

* 1、客户端获取当前时间，以毫秒为单位。
* 2、客户端尝试获取 N 个节点的锁，（每个节点获取锁的方式和前面说的缓存锁一样），N 个节点以相同的 key 和 value 获取锁。客户端需要设置接口访问超时，接口超时时间需要远远小于锁超时时间，比如锁自动释放的时间是 10s，那么接口超时大概设置 5-50ms。这样可以在有 redis 节点宕机后，访问该节点时能尽快超时，而减小锁的正常使用。
* 3、客户端计算在获得锁的时候花费了多少时间，方法是用当前时间减去在步骤一获取的时间，只有客户端获得了超过 3 个节点的锁，而且获取锁的时间小于锁的超时时间，客户端才获得了分布式锁。
* 4、客户端获取的锁的时间为设置的锁超时时间减去步骤三计算出的获取锁花费时间。
* 5、如果客户端获取锁失败了，客户端会依次删除所有的锁。

使用 Redlock 算法，可以保证在挂掉最多 2 个节点的时候，分布式锁服务仍然能工作，这相比之前的数据库锁和缓存锁大大提高了可用性，由于 redis 的高效性能，分布式缓存锁性能并不比数据库锁差。

但是，有一位分布式的专家写了一篇文章《How to do distributed locking》，质疑 Redlock 的正确性。

<https://mp.weixin.qq.com/s/1bPLk_VZhZ0QYNZS8LkviA>

<https://blog.csdn.net/jek123456/article/details/72954106>

**优缺点**

**优点：** 性能高

**缺点：**

失效时间设置多长时间为好？如何设置的失效时间太短，方法没等执行完，锁就自动释放了，那么就会产生并发问题。如果设置的时间太长，其他获取锁的线程就可能要平白的多等一段时间。

### 8.4基于 REDISSON 做分布式锁

redisson 是 redis 官方的分布式锁组件。GitHub 地址：<https://github.com/redisson/redisson>

上面的这个问题 ——> 失效时间设置多长时间为好？这个问题在 redisson 的做法是：每获得一个锁时，只设置一个很短的超时时间，同时起一个线程在每次快要到超时时间时去刷新锁的超时时间。在释放锁的同时结束这个线程。

### 8.5基于 ZooKeeper 做分布式锁

**ZOOKEEPER 锁相关基础知识**

* zk 一般由多个节点构成（单数），采用 zab 一致性协议。因此可以将 zk 看成一个单点结构，对其修改数据其内部自动将所有节点数据进行修改而后才提供查询服务。
* zk 的数据以目录树的形式，每个目录称为 znode， znode 中可存储数据（一般不超过 1M），还可以在其中增加子节点。
* 子节点有三种类型。序列化节点，每在该节点下增加一个节点自动给该节点的名称上自增。临时节点，一旦创建这个 znode 的客户端与服务器失去联系，这个 znode 也将自动删除。最后就是普通节点。
* Watch 机制，client 可以监控每个节点的变化，当产生变化会给 client 产生一个事件。

**ZK 基本锁**

* 原理：利用临时节点与 watch 机制。每个锁占用一个普通节点 /lock，当需要获取锁时在 /lock 目录下创建一个临时节点，创建成功则表示获取锁成功，失败则 watch/lock 节点，有删除操作后再去争锁。临时节点好处在于当进程挂掉后能自动上锁的节点自动删除即取消锁。
* 缺点：所有取锁失败的进程都监听父节点，很容易发生羊群效应，即当释放锁后所有等待进程一起来创建节点，并发量很大。

**ZK 锁优化**

* 原理：上锁改为创建临时有序节点，每个上锁的节点均能创建节点成功，只是其序号不同。只有序号最小的可以拥有锁，如果这个节点序号不是最小的则 watch 序号比本身小的前一个节点 (公平锁)。

步骤：

* 1.在 /lock 节点下创建一个有序临时节点 (EPHEMERAL\_SEQUENTIAL)。
* 2.判断创建的节点序号是否最小，如果是最小则获取锁成功。不是则取锁失败，然后 watch 序号比本身小的前一个节点。
* 3.当取锁失败，设置 watch 后则等待 watch 事件到来后，再次判断是否序号最小。
* 4.取锁成功则执行代码，最后释放锁（删除该节点）。

import java.io.IOException;

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

import java.util.concurrent.CountDownLatch;

import java.util.concurrent.TimeUnit;

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

import org.apache.zookeeper.CreateMode;

import org.apache.zookeeper.KeeperException;

import org.apache.zookeeper.WatchedEvent;

import org.apache.zookeeper.Watcher;

import org.apache.zookeeper.ZooDefs;

import org.apache.zookeeper.ZooKeeper;

import org.apache.zookeeper.data.Stat;

public class DistributedLock implements Lock, Watcher{

private ZooKeeper zk;

private String root = "/locks";//根

private String lockName;//竞争资源的标志

private String waitNode;//等待前一个锁

private String myZnode;//当前锁

private CountDownLatch latch;//计数器

private int sessionTimeout = 30000;

private List<Exception> exception = new ArrayList<Exception>();

/\*\*

\* 创建分布式锁,使用前请确认config配置的zookeeper服务可用

\* @param config 127.0.0.1:2181

\* @param lockName 竞争资源标志,lockName中不能包含单词lock

\*/

public DistributedLock(String config, String lockName){

this.lockName = lockName;

// 创建一个与服务器的连接

try {

zk = new ZooKeeper(config, sessionTimeout, this);

Stat stat = zk.exists(root, false);

if(stat == null){

// 创建根节点

zk.create(root, new byte[0], ZooDefs.Ids.OPEN\_ACL\_UNSAFE,CreateMode.PERSISTENT);

}

} catch (IOException e) {

exception.add(e);

} catch (KeeperException e) {

exception.add(e);

} catch (InterruptedException e) {

exception.add(e);

}

}

/\*\*

\* zookeeper节点的监视器

\*/

public void process(WatchedEvent event) {

if(this.latch != null) {

this.latch.countDown();

}

}

public void lock() {

if(exception.size() > 0){

throw new LockException(exception.get(0));

}

try {

if(this.tryLock()){

System.out.println("Thread " + Thread.currentThread().getId() + " " +myZnode + " get lock true");

return;

}

else{

waitForLock(waitNode, sessionTimeout);//等待锁

}

} catch (KeeperException e) {

throw new LockException(e);

} catch (InterruptedException e) {

throw new LockException(e);

}

}

public boolean tryLock() {

try {

String splitStr = "\_lock\_";

if(lockName.contains(splitStr))

throw new LockException("lockName can not contains \\u000B");

//创建临时子节点

myZnode = zk.create(root + "/" + lockName + splitStr, new byte[0], ZooDefs.Ids.OPEN\_ACL\_UNSAFE,CreateMode.EPHEMERAL\_SEQUENTIAL);

System.out.println(myZnode + " is created ");

//取出所有子节点

List<String> subNodes = zk.getChildren(root, false);

//取出所有lockName的锁

List<String> lockObjNodes = new ArrayList<String>();

for (String node : subNodes) {

String \_node = node.split(splitStr)[0];

if(\_node.equals(lockName)){

lockObjNodes.add(node);

}

}

Collections.sort(lockObjNodes);

System.out.println(myZnode + "==" + lockObjNodes.get(0));

if(myZnode.equals(root+"/"+lockObjNodes.get(0))){

//如果是最小的节点,则表示取得锁

return true;

}

//如果不是最小的节点，找到比自己小1的节点

String subMyZnode = myZnode.substring(myZnode.lastIndexOf("/") + 1);

waitNode = lockObjNodes.get(Collections.binarySearch(lockObjNodes, subMyZnode) - 1);

} catch (KeeperException e) {

throw new LockException(e);

} catch (InterruptedException e) {

throw new LockException(e);

}

return false;

}

public boolean tryLock(long time, TimeUnit unit) {

try {

if(this.tryLock()){

return true;

}

return waitForLock(waitNode,time);

} catch (Exception e) {

e.printStackTrace();

}

return false;

}

private boolean waitForLock(String lower, long waitTime) throws InterruptedException, KeeperException {

Stat stat = zk.exists(root + "/" + lower,true);

//判断比自己小一个数的节点是否存在,如果不存在则无需等待锁,同时注册监听

if(stat != null){

System.out.println("Thread " + Thread.currentThread().getId() + " waiting for " + root + "/" + lower);

this.latch = new CountDownLatch(1);

this.latch.await(waitTime, TimeUnit.MILLISECONDS);

this.latch = null;

}

return true;

}

public void unlock() {

try {

System.out.println("unlock " + myZnode);

zk.delete(myZnode,-1);

myZnode = null;

zk.close();

} catch (InterruptedException e) {

e.printStackTrace();

} catch (KeeperException e) {

e.printStackTrace();

}

}

public void lockInterruptibly() throws InterruptedException {

this.lock();

}

public Condition newCondition() {

return null;

}

public class LockException extends RuntimeException {

private static final long serialVersionUID = 1L;

public LockException(String e){

super(e);

}

public LockException(Exception e){

super(e);

}

}

}

**优缺点**

**优点：**

有效的解决单点问题，不可重入问题，非阻塞问题以及锁无法释放的问题。实现起来较为简单。

**缺点：**

性能上可能并没有缓存服务那么高，因为每次在创建锁和释放锁的过程中，都要动态创建、销毁临时节点来实现锁功能。ZK 中创建和删除节点只能通过 Leader 服务器来执行，然后将数据同步到所有的 Follower 机器上。还需要对 ZK的原理有所了解。

### 8.6基于 Consul 做分布式锁

DD 写过类似文章，其实主要利用 Consul 的 Key / Value 存储 API 中的 acquire 和 release 操作来实现。

文章地址：<http://blog.didispace.com/spring-cloud-consul-lock-and-semphore/>

**使用分布式锁的注意事项**

1、注意分布式锁的开销

2、注意加锁的粒度

3、加锁的方式