多线程学习笔记

### 线程实现4种方式

#### 继承Thread类创建线程

Thread类本质上是实现了Runnable接口的一个实例，代表一个线程的实例。启动线程的唯一方法就是通过Thread类的start()实例方法。start()方法是一个native方法，它将启动一个新线程，并执行run()方法。这种方式实现多线程很简单，通过自己的类直接extend Thread，并复写run()方法，就可以启动新线程并执行自己定义的run()方法。代码简单就不写示例了。

最佳实践：创建线程最好给线程指定一个线程名称，这样方便管理。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。

#### 实现Runnable接口创建线程

如果自己的类已经extends另一个类，就无法直接extends Thread，此时，可以实现一个Runnable接口，并且实现run方法。

#### 实现Callable接口通过FutureTask包装器来创建Thread线程

Callable接口（也只有一个方法）定义如下：

|  |
| --- |
| public interface Callable<V>{  V call() throws Exception;  } |

创建线程代码如下：

|  |
| --- |
| public class SomeCallable<V> extends OtherClass implements Callable<V> {  @Override  public V call() throws Exception {  // TODO Auto-generated method stub  return null;  }  } |

|  |
| --- |
| Callable<V> oneCallable = new SomeCallable<V>();  //由Callable<Integer>创建一个FutureTask<Integer>对象：  FutureTask<V> oneTask = new FutureTask<V>(oneCallable);  //注释：FutureTask<Integer>是一个包装器，它通过接受Callable<Integer>来创建，它同时实现了Future和Runnable接口。  //由FutureTask<Integer>创建一个Thread对象：  Thread oneThread = new Thread(oneTask);  oneThread.start(); |

#### 使用ExecutorService、Callable、Future实现有返回结果的线程

ExecutorService、Callable、Future三个接口实际上都是属于Executor框架。返回结果的线程是在JDK1.5中引入的新特征，有了这种特征就不需要再为了得到返回值而大费周折了。而且自己实现了也可能漏洞百出。

可返回值的任务必须实现Callable接口。类似的，无返回值的任务必须实现Runnable接口。

执行Callable任务后，可以获取一个Future的对象，在该对象上调用get就可以获取到Callable任务返回的Object了。

注意：get方法是阻塞的，即：线程无返回结果，get方法会一直等待。

再结合线程池接口ExecutorService就可以实现传说中有返回结果的多线程了。

下面提供了一个完整的有返回结果的多线程测试例：

|  |
| --- |
| /\*\*  \* 有返回值的线程  \*/  @SuppressWarnings("unchecked")  public class Test {  public static void main(String[] args) throws ExecutionException,  InterruptedException {  System.out.println("----程序开始运行----");  Date date1 = new Date();  int taskSize = 5;  // 创建一个线程池  ExecutorService pool = Executors.newFixedThreadPool(taskSize);  // 创建多个有返回值的任务  List<Future> list = new ArrayList<Future>();  for (int i = 0; i < taskSize; i++) {  Callable c = new MyCallable(i + " ");  // 执行任务并获取Future对象  Future f = pool.submit(c);  // System.out.println(">>>" + f.get().toString());  list.add(f);  }  // 关闭线程池  pool.shutdown();  // 获取所有并发任务的运行结果  for (Future f : list) {  // 从Future对象上获取任务的返回值，并输出到控制台  System.out.println(">>>" + f.get().toString());  }  Date date2 = new Date();  System.out.println("----程序结束运行----，程序运行时间【"  + (date2.getTime() - date1.getTime()) + "毫秒】");  }  }    public class MyCallable implements Callable<Object> {  private String taskNum;    MyCallable(String taskNum) {  this.taskNum = taskNum;  }    public Object call() throws Exception {  System.out.println(">>>" + taskNum + "任务启动");  Date dateTmp1 = new Date();  Thread.sleep(1000);  Date dateTmp2 = new Date();  long time = dateTmp2.getTime() - dateTmp1.getTime();  System.out.println(">>>" + taskNum + "任务终止");  return taskNum + "任务返回运行结果,当前时间【" + time + "毫秒】";  }  } |

#### 总结

从上面4个实例可以看出实现线程的方式也可以说是3种方式，分别是继承Threa类，实现Runnable接口、实现Callable（一种使用FutureTask，一种使用J.U.C的线程池）。

### 终止线程三种方式

有三种方法可以使终止线程：

* 使用退出标志，使线程正常退出，也就是当run方法完成后线程终止。
* 使用stop方法强行终止线程（这个方法不推荐使用，因为stop会导致线程立马停止，这样可能会导致一些锁或数据、资源等释放操作无法得到正确执行，这样就可能出现不可预料的结果）。
* 使用interrupt方法中断线程。

#### 使用退出标志终止线程

当run方法执行完后，线程就会退出。但有时run方法是永远不会结束的。如在服务端程序中使用线程进行监听客户端请求，或是其他的需要循环处理的任务。在这种情况下，一般是将这些任务放在一个循环中，如while循环。如果想让循环永远运行下去，可以使用while（true）{……}来处理。但要想使while循环在某一特定条件下退出，最直接的方法就是设一个boolean类型的标志，并通过设置这个标志为true或false来控制while循环是否退出。下面给出了一个利用退出标志终止线程的例子。

|  |
| --- |
| public class ThreadFlag extends Thread{  public volatile boolean exit = false;  public void run(){  while (!exit){//执行逻辑};  }  public static void main(String[] args) throws Exception{  ThreadFlag thread = new ThreadFlag();  thread.start();  sleep(5000); // 主线程延迟5秒  thread.exit = true; // 终止线程thread  thread.join();  System.out.println("线程退出!");  }  } |

上述代码中，join() 的作用：让“主线程”等待“子线程”结束之后才能继续运行。也就是说执行到【thread.join()】代码时，主线程（main）会等待thread线程执行结束后，主线程才继续执行。

#### 使用stop方法终止线程

使用stop方法可以强行终止正在运行或挂起的线程。我们可以使用如下的代码来终止线程：thread.stop();

虽然使用上面的代码可以终止线程，但使用stop方法是很危险的，就象突然关闭计算机电源，而不是按正常程序关机一样，可能会产生不可预料的结果，因此，并不推荐使用stop方法来终止线程。

#### 使用interrupt方法终止线程

使用interrupt方法来终端线程可分为两种情况：

* 线程处于阻塞状态，如使用了sleep方法。
* 使用while（！isInterrupted（））{……}来判断线程是否被中断。

在第一种情况下使用interrupt方法，sleep方法将抛出一个InterruptedException例外，而在第二种情况下线程将直接退出。下面的代码演示了在第一种情况下使用interrupt方法。

|  |
| --- |
| public class ThreadInterrupt extends Thread{  public void run(){  try{  sleep(50000); // 延迟50秒  } catch (InterruptedException e){  System.out.println(e.getMessage());  }  }  public static void main(String[] args) throws Exception{  Thread thread = new ThreadInterrupt();  thread.start();  System.out.println("在50秒之内按任意键中断线程!");  System.in.read();  thread.interrupt();  thread.join();  System.out.println("线程已经退出!");  }  } |

上面代码的运行结果如下：

|  |
| --- |
| 在50秒之内按任意键中断线程!  sleep interrupted  线程已经退出! |

在调用interrupt方法后， sleep方法抛出异常，然后输出错误信息：sleep interrupted.

注意：在Thread类中有两个方法可以判断线程是否通过interrupt方法被终止。一个是静态的方法interrupted（），一个是非静态的方法isInterrupted（），这两个方法的区别是interrupted用来判断当前线是否被中断，而isInterrupted可以用来判断其他线程是否被中断。因此，while （！isInterrupted（））也可以换成while （！Thread.interrupted（））。

### 线程状态

Java中的线程的生命周期大体可分为5种状态：

* 新建(NEW)：新创建了一个线程对象。
* 可运行(RUNNABLE)：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 。
* 运行(RUNNING)：可运行状态(runnable)的线程获得了cpu 时间片（timeslice） ，执行程序代码。
* 阻塞(BLOCKED)：阻塞状态是指线程因为某种原因放弃了cpu 使用权，也即让出了cpu timeslice，暂时停止运行。直到线程进入可运行(runnable)状态，才有机会再次获得cpu timeslice 转到运行(running)状态。阻塞的情况分三种：
* 等待阻塞：运行(running)的线程执行o.wait()方法，JVM会把该线程放入等待队列(waitting queue)中。
* 同步阻塞：运行(running)的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池(lock pool)中。
* 其他阻塞：运行(running)的线程执行Thread.sleep(long ms)或t.join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入可运行(runnable)状态。
* 死亡(DEAD)：线程run()、main() 方法执行结束，或者因异常退出了run()方法，则该线程结束生命周期。死亡的线程不可再次复生。

线程的状态图如下：



### 内存模型

一个现代计算机通常由两个或者多个CPU。其中一些CPU还有多核。从这一点可以看出，在一个有两个或者多个CPU的现代计算机上同时运行多个线程是可能的。每个CPU在某一时刻运行一个线程是没有问题的。这意味着，如果你的Java程序是多线程的，在你的Java程序中每个CPU上一个线程可能同时（并发）执行。

每个CPU都包含一系列的寄存器，它们是CPU内内存的基础。CPU在寄存器上执行操作的速度远大于在主存上执行的速度。这是因为CPU访问寄存器的速度远大于主存。

每个CPU可能还有一个CPU缓存层。实际上，绝大多数的现代CPU都有一定大小的缓存层。CPU访问缓存层的速度快于访问主存的速度，但通常比访问内部寄存器的速度还要慢一点。一些CPU还有多层缓存，但这些对理解Java内存模型如何和内存交互不是那么重要。只要知道CPU中可以有一个缓存层就可以了。

一个计算机还包含一个主存。所有的CPU都可以访问主存。主存通常比CPU中的缓存大得多。

通常情况下，当一个CPU需要读取主存时，它会将主存的部分读到CPU缓存中。它甚至可能将缓存中的部分内容读到它的内部寄存器中，然后在寄存器中执行操作。当CPU需要将结果写回到主存中去时，它会将内部寄存器的值刷新到缓存中，然后在某个时间点将值刷新回主存。

当CPU需要在缓存层存放一些东西的时候，存放在缓存中的内容通常会被刷新回主存。CPU缓存可以在某一时刻将数据局部写到它的内存中，和在某一时刻局部刷新它的内存。它不会再某一时刻读/写整个缓存。通常，在一个被称作“cache lines”的更小的内存块中缓存被更新。一个或者多个缓存行可能被读到缓存，一个或者多个缓存行可能再被刷新回主存。

想象一下，共享对象被初始化在主存中。跑在CPU上的一个线程将这个共享对象读到CPU缓存中。然后修改了这个对象。只要CPU缓存没有被刷新会主存，对象修改后的版本对跑在其它CPU上的线程都是不可见的。这种方式可能导致每个线程拥有这个共享对象的私有拷贝，每个拷贝停留在不同的CPU缓存中。

使用Java中的volatile关键字。volatile关键字可以保证直接从主存中读取一个变量，如果这个变量被修改后，总是会被写回到主存中去。同时会通知其他线程该变量cpu的缓存行无效，其他线程将直接从主存中读取该变量的值到cpu缓存中。

如果想了解指令重排序和happens-before原则的知识点或更多关于内存模型的知识，可以阅读：<https://www.cnblogs.com/dolphin0520/p/3920373.html>

以下为代码示例：

|  |
| --- |
| public class VolatileTest implements Runnable{  private volatile boolean tag = false;  private static CountDownLatch latch = new CountDownLatch(10);  @Override  public void run() {  if(tag){  System.out.println("tag为："+tag);  }else{  System.out.println("tag111111为："+tag);  tag = true;  System.out.println("tag2222为："+tag);  }  latch.countDown();  }  public static void main(String[] args) throws InterruptedException {  ExecutorService executorService = Executors.newFixedThreadPool(10);  VolatileTest test = new VolatileTest();  for(int i = 0; i < 10; i++){  executorService.submit(test);  }  latch.await();  executorService.shutdown();  System.out.println("测试完毕");  }  } |

上面代码示例其中的一次执行结果如下：

|  |
| --- |
| tag111111为：false  tag111111为：false  tag111111为：false  tag2222为：true  tag2222为：true  tag2222为：true  tag为：true  tag为：true  tag为：true  tag为：true  tag为：true  tag为：true  tag为：true  测试完毕 |

Volatile修饰的变量能保证变量原子操作下的线程安全，非原子操作无法保证线程安全，如下面代码：

|  |
| --- |
| volatile int count;  public void add(){  count++;  } |

上述代码的add方法在多线程环境下存在线程安全的问题，比如有2个线程（A和B），A将count的值（0）读到cpu缓存中，并进行加1，但还没赋值给count的节点前任意时刻，线程B读到cpu缓存的count值也为0，并且进行++操作，所以最终获取的值为1，而不是2。这里++操作就不是原子操作，所以哪怕使用volatile修饰也存在线程安全问题。

### Synchronized

线程安全是并发编程中的重要关注点，应该注意到的是，造成线程安全问题的主要诱因有两点，一是存在共享数据(也称临界资源)，二是存在多条线程共同操作共享数据。因此为了解决这个问题，我们可能需要这样一个方案，当存在多个线程操作共享数据时，需要保证同一时刻有且只有一个线程在操作共享数据，其他线程必须等到该线程处理完数据后再进行，这种方式有个高尚的名称叫互斥锁，即能达到互斥访问目的的锁，也就是说当一个共享数据被当前正在访问的线程加上互斥锁后，在同一个时刻，其他线程只能处于等待的状态，直到当前线程处理完毕释放该锁。在 Java 中，关键字 synchronized可以保证在同一个时刻，只有一个线程可以执行某个方法或者某个代码块(主要是对方法或者代码块中存在共享数据的操作)，同时我们还应该注意到synchronized另外一个重要的作用，synchronized可保证一个线程的变化(主要是共享数据的变化)被其他线程所看到（保证可见性，完全可以替代Volatile功能），这点确实也是很重要的。

synchronized关键字最主要有以下3种应用方式：

* 修饰实例方法，作用于当前实例加锁，进入同步代码前要获得当前实例的锁
* 修饰静态方法，作用于当前类对象加锁，进入同步代码前要获得当前类对象的锁
* 修饰代码块，指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。

#### synchronized的可重入性

从互斥锁的设计上来说，当一个线程试图操作一个由其他线程持有的对象锁的临界资源时，将会处于阻塞状态，但当一个线程再次请求自己持有对象锁的临界资源时，这种情况属于重入锁，请求将会成功。在java中synchronized是基于原子性的内部锁机制，是可重入的，因此在一个线程调用synchronized方法的同时在其方法体内部调用该对象另一个synchronized方法，也就是说一个线程得到一个对象锁后再次请求该对象锁，是允许的，这就是synchronized的可重入性。简单代码如下：

|  |
| --- |
| public class SynchronziedTest implements Runnable{  public synchronized void test1(){  System.out.println("开始执行test1");  test2();  System.out.println("执行test1结束");  }  public synchronized void test2(){  System.out.println("执行test2");  }  @Override  public void run() {  test1();  }  public static void main(String[] args) {  new Thread(new SynchronziedTest()).start();  }  } |

上面示例执行结果如下：

|  |
| --- |
| 开始执行test1  执行test2  执行test1结束 |

线程获取SynchronziedTest对象锁进入test1方法后，如果在test1方法中调用test2方法（该方法也被synchronized）是被允许的，我们就说该锁是可重入锁（或该锁有可重入的特性）。

#### 文章推荐

<http://cmsblogs.com/?p=2071>

### wait、notify和notifyAll

wait、notify和notifyAll主要注意的有以下方面：

* wait、notify和notifyAll方法必须在同步代码块中使用
* sleep和wait的区别：sleep不会释放对象锁，wait会释放对象锁
* notify和notifyAll方法调用后，尽量立即退出临界区，就是说尽量不要在notify和notifyAll方法后面写一些耗时的操作
* 多线程中校验某个条件的变化用if还是用while？如：

while(条件不满足){this.wait();}还是if(条件不满足){this.wait();}，答案是使用while。

当一个线程正在某一个对象的同步方法中运行时调用了这个对象的wait()方法，那么这个线程将释放该对象的独占锁并被放入这个对象的等待队列。注意，wait()方法强制当前线程释放对象锁。这意味着在调用某对象的wait()方法之前，当前线程必须已经获得该对象的锁。因此，线程必须在某个对象的同步方法或同步代码块中才能调用该对象的wait()方法。在调用对象的notify()和notifyAll()方法之前,调用线程必须已经得到该对象的锁。因此,必须在某个对象的同步方法或同步代码块中才能调用该对象的notify()或notifyAll()方法。

wait、notify和notifyAll方法只是实现线程间通信的一种方式，更多内容可以查看：<http://ifeve.com/thread-signaling/>

### 死锁

死锁是两个或更多线程阻塞着等待其它处于死锁状态的线程所持有的锁。死锁通常发生在多个线程同时但以不同的顺序请求同一组锁的时候。

例如，如果线程1锁住了A，然后尝试对B进行加锁，同时线程2已经锁住了B，接着尝试对A进行加锁，这时死锁就发生了。线程1永远得不到B，线程2也永远得不到A，并且它们永远也不会知道发生了这样的事情。为了得到彼此的对象（A和B），它们将永远阻塞下去。这种情况就是一个死锁。

一般造成死锁必须同时满足如下4个条件：

* 互斥条件：线程使用的资源必须至少有一个是不能共享的；
* 请求与保持条件：至少有一个线程必须持有一个资源并且正在等待获取一个当前被其它线程持有的资源；
* 非剥夺条件：分配资源不能从相应的线程中被强制剥夺；
* 循环等待条件：第一个线程等待其它线程，后者又在等待第一个线程。

因为要产生死锁，这4个条件必须同时满足，所以要防止死锁的话，只需要破坏其中一个条件即可。一些推荐做法如下：

* 尽量使用tryLock(long timeout, TimeUnit unit)的方法(ReentrantLock、ReentrantReadWriteLock)，设置超时时间，超时可以退出防止死锁。
* 尽量使用java.util.concurrent(jdk 1.5以上)包的并发类代替手写控制并发，比较常用的是ConcurrentHashMap、ConcurrentLinkedQueue、AtomicBoolean等等，实际应用中java.util.concurrent.atomic十分有用，简单方便且效率比使用Lock更高
* 尽量降低锁的使用粒度，尽量不要几个功能用同一把锁
* 尽量减少同步的代码块

比较简单的方法：所有线程都按顺序获取资源，比如资源ABCD ，那所有线程必须获取了A资源再去 申请B资源 ， 依次类推。

其实我们在操作资源的时候，可以使用到队列。让线程按照顺序操作资源，这样的话，就不会造成死锁现象。

按顺序获取资源，是破坏循环等待的条件，算是比较好的方法但是也是有缺点的：

* 是限制的进程对资源的要求，而已给系统中所有的资源合理编号也是件难事，同时增加了系统开销。
* 为了遵循按编号申请的次序，暂不使用的资源也需要提前申请，从而增加了进程对资源的占用时间。

对于这个问题，单一的临界资源可以使用资源分配图算法，多类的临界资源可以使用“银行家算法”。

### ThreadLocal

#### 作用

ThreadLocal的作用是提供线程内的局部变量，这种变量在线程的生命周期内起作用，减少同一个线程内多个函数或者组件之间一些公共变量的传递的复杂度。也可以归纳为以下两点：

1。每个线程中都有一个自己的ThreadLocalMap类对象，可以将线程自己的对象保持到其中，各管各的，线程可以正确的访问到自己的对象。

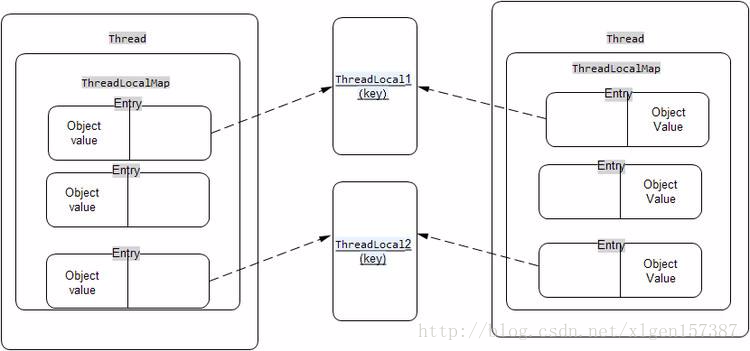
2。将一个共用的ThreadLocal静态实例作为key，将不同对象的引用保存到不同线程的ThreadLocalMap中，然后在线程执行的各处通过这个静态ThreadLocal实例的get()方法取得自己线程保存的那个对象，避免了将这个对象作为参数传递的麻烦。

#### 使用场景

最常见的ThreadLocal使用场景为用来解决数据库连接、Session管理等。我们平常开发常用于传递登录的用户信息，不用每次从session中去取，同时减少user信息到处传递的麻烦。比如拦截器中校验用户信息后，将用户信息放到ThreadLocalMap中，这样无论是controoler层还是service层（或其他层）需要获取用户信息，只需要从ThreadLocal中获取即可。当然ThreadLocal也不能乱用，如果存储的内容过多，并且释放不及时（后面会继续介绍），可能会导致内存泄漏的问题。

#### 源码分析

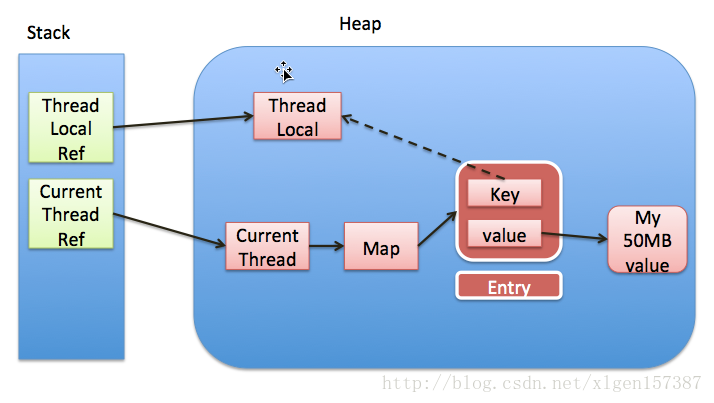
##### Thread、ThreadLocal、ThreadLocalMap、Entry之间的关系



上图中描述了：一个Thread中只有一个ThreadLocalMap，一个ThreadLocalMap中可以有多个ThreadLocal对象，其中一个ThreadLocal对象对应一个ThreadLocalMap中一个的Entry实体（也就是说：一个Thread可以依附有多个ThreadLocal对象）。

##### ThreadLocal各类引用关系

在ThreadLocal的生命周期中，都存在这些引用。（ 实线代表强引用，虚线代表弱引用）。



ThreadLocal到Entry对象key的引用断裂，而不及时的清理Entry对象，可能会造成OOM内存溢出！

##### 引用的类型

我们对引用的理解也许很简单，就是：如果 reference类型的数据中存储的数值代表的是另外一块内存的起始地址，就称这块内存代表着一个引用。但是书上说的这种方式过于狭隘，一个对象在这种定义下只有被引用或者没有被引用两种状态，对于如何描述一些“食之无味，弃之可惜”的对象就显得无能为力。我们希望能描述这样一类对象：当内存空间还足够时，则能保留在内存之中；如果内存在进行垃圾收集后还是非常紧张，则可以抛弃这些对象。很多系统的缓存功能都符合这样的应用场景。

一般的引用类型分为：强引用（ Strong Reference）、软引用（ Soft Reference）、弱引用（ Weak Reference）、虚引用（ Phantom Reference）四种，这四种引用强度依次逐渐减弱。下边是四中类型的介绍：

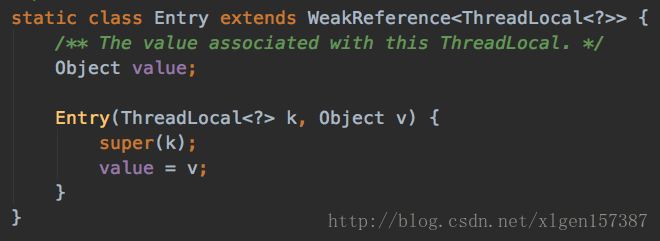
* 强引用：就是指在程序代码之中普遍存在的，类似“Object obj = new Object()”这类的引用，只要强引用还存在，垃圾收集器永远不会回收掉被引用的对象，也就是说即使Java虚拟机内存空间不足时，GC收集器也绝不会回收该对象，如果内存空间不够就会导致内存溢出。
* 软引用：用来描述一些还有用，但并非必需的对象。对于软引用关联着的对象，在系统将要发生内存溢出异常之前，将会把这些对象列进回收范围之中并进行回收，以免出现内存溢出。如果这次回收还是没有足够的内存，才会抛出内存溢出异常。在 JDK 1.2 之后，提供了 SoftReference 类来实现软引用。软引用适合引用那些可以通过其他方式恢复的对象，例如：数据库缓存中的对象就可以从数据库中恢复，所以软引用可以用来实现缓存。等会会介绍MyBatis中的使用软引用实现缓存的案例。
* 弱引用：也是用来描述非必需对象的，但是它的强度比软引用更弱一些，被弱引用关联的对象只能生存到下一次垃圾收集发生之前。当垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。在 JDK 1.2 之后，提供了 WeakReference 类来实现弱引用。ThreadLocal使用到的就有弱引用。
* 虚引用：也称为幽灵引用或者幻影引用，它是最弱的一种引用关系。一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。为一个对象设置虚引用关联的唯一目的就是希望能在这个对象被收集器回收时收到一个系统通知。在 JDK 1.2 之后，提供了PhantomReference 类来实现虚引用。

各引用类型的生命周期及作用：



##### ThreadLocal中的弱引用

上述我们知道了当垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。我们的ThreadLocal中ThreadLocalMap中的Entry类的key就是弱引用的，如下：



而弱引用会在垃圾收集器工作的时候进行回收，也就是说，只要执行垃圾回收，这些对象就会被回收，也就是上述图B中的虚线连接的地方断开了，就成了一个没有key的Entry。这里暂时不做过多的介绍，建议多看写得不错的一篇文章：<https://blog.csdn.net/xlgen157387/article/details/78513735?ref=myread>

##### 内存泄漏

synchronized是用时间换空间、ThreadLocal是用空间换时间，为什么这么说？

因为synchronized操作数据，只需要在主存存一个变量即可，就阻塞等共享变量，而ThreadLocal是每个线程都创建一块小的堆工作内存。一个线程对应一块工作内存，线程可以存储多个ThreadLocal。那么假设，开启1万个线程，每个线程创建1万个ThreadLocal，也就是每个线程维护1万个ThreadLocal小内存空间，而且当线程执行结束以后，假设这些ThreadLocal里的Entry还不会被回收，那么将很容易导致堆内存溢出。

怎么办？难道JVM就没有提供什么解决方案吗？

ThreadLocal当然有想到，所以他们把ThreadLocal里的Entry设置为弱引用，当垃圾回收的时候，回收ThreadLocal。

什么是弱引用？

Key使用强引用：也就是上述说的情况，引用的ThreadLocal的对象被回收了，ThreadLocal的引用ThreadLocalMap的Key为强引用并没有被回收，如果不手动回收的话，ThreadLocal将不会回收那么将导致内存泄漏。

Key使用弱引用：引用的ThreadLocal的对象被回收了，ThreadLocal的引用ThreadLocalMap的Key为弱引用，如果内存回收，那么将ThreadLocalMap的Key将会被回收，ThreadLocal也将被回收。value在ThreadLocalMap调用get、set、remove的时候就会被清除。

比较两种情况，我们可以发现：由于ThreadLocalMap的生命周期跟Thread一样长，如果都没有手动删除对应key，都会导致内存泄漏，但是使用弱引用可以多一层保障：弱引用ThreadLocal不会内存泄漏，对应的value在下一次ThreadLocalMap调用set,get,remove的时候会被清除。

那按你这么说，既然JVM有保障了，还有什么内存泄漏可言？

ThreadLocalMap使用ThreadLocal对象作为弱引用，当垃圾回收的时候，ThreadLocalMap中Key将会被回收，也就是将Key设置为null的Entry。如果线程迟迟无法结束，也就是ThreadLocal对象将一直不会回收，回顾到上面存在很多线程+TheradLocal，那么也将导致内存泄漏。

其实，在ThreadLocal中，当调用remove、get、set方法的时候，会清除为null的弱引用，也就是回收ThreadLocal。

总结：

JVM利用设置ThreadLocalMap的Key为弱引用，来避免内存泄露。

JVM利用调用remove、get、set方法的时候，回收弱引用。

当ThreadLocal存储很多Key为null的Entry的时候，而不再去调用remove、get、set方法，那么将导致内存泄漏。

当使用static ThreadLocal的时候，延长ThreadLocal的生命周期，那也可能导致内存泄漏。因为，static变量在类未加载的时候，它就已经加载，当线程结束的时候，static变量不一定会回收。那么，比起普通成员变量使用的时候才加载，static的生命周期加长将更容易导致内存泄漏危机。

##### 我的总结

我们使用tomcat为容器，用户发送请求，到请求结束这整个过程结束后，处理请求的线程未必会被销毁，有可能会放到tomcat的线程池中一直在重复被利用，所以这种情况下，ThreadLocal会可能出现内存泄漏的问题，或者说ThreadLocalMap会一直占用内存资源，下面我们分别分析ThreadLocal2种不同的使用情况。

ThreadLocal对象不使用static修饰的情况：

* 由于ThreadLocalMap的key是弱引用，所以每次gc后，key会被回收，而线程未必会被销毁（如容器为tomcat），那么ThreadLocal中会出现key为null，value为存储的对象，这样会造成资源浪费（还有其他问题）。虽然ThreadLocal中的get、set方法会在每次调用的时候清除这类key为null的数据，但是如果线程一直不调用get和set方法呢？那内存就会一直被占用着，最后就可能出现内存泄漏的问题。
* 如果线程被销毁，那ThreadLocalMap会被回收，这里主要是有2个问题：
* 使用过程中发生gc（ThreadLocal的get方法调用前），当前线程获取副本数据将为空(这样就容易出现莫名其妙的bug)；所以jdk推荐使用static修饰ThreadLocal；这里关于获取副本数据为空的情况是不会出现的，这点做保留只是为了自己不会再次出现这种错误的理解（因为只有将ThreadLocal对象设置为null且当前线程销毁后，gc才回收ThreadLocalMap对象），关于推荐使用static修饰ThreadLocal，只是为了延长ThreadLocal的生命周期。
* 在线程被销毁前（线程可能会多次使用后才销毁），内存浪费（可能出现内存泄漏问题），除非调用了ThreadLocal的get、set、remove方法，不然key为null，value有内容的数据一直不会被回收，并占据着内存。

ThreadLocal对象使用static修饰的情况：

* 由于ThreadLocal使用static修饰，那么ThreadLocalMap的key不再是弱引用，所以每次gc后，key不会被回收。同样因为线程未必会被销毁，所以ThreadLocalMap中的内容永远不会被垃圾回收器回收。这里更为严重的是，就算线程在某一时间被销毁了，但是因为key为强引用（static修饰不会被回收），所以ThreadLocalMap中的内容也不会被回收，这样内存就一直被浪费着，最后可能出现内存泄漏的问题。这里有点注意的是，key是否回收不重要，因为他用static修饰，只有一份，最重要的是value内的数据回收是关键。

##### 最佳实现

* 每次使用完ThreadLocal，都调用它的remove()方法，清除数据。
* 如果需要延长ThreadLocal生命周期，推荐使用static修饰。

##### 如何学习

我觉得多阅读源代码，配合以下文章进行学习，这样关于ThreadLocal的知识应该是能尽可能掌握好了，下面是文章链接：

<https://blog.csdn.net/xlgen157387/article/details/78513735?ref=myread>

<https://www.cnblogs.com/qiuyong/p/7091689.html>

<https://www.zhihu.com/question/23089780>

<http://www.importnew.com/22039.html>

### AQS

#### 概述

Java的内置锁一直都是备受争议的，在JDK 1.6之前，synchronized这个重量级锁其性能一直都是较为低下，虽然在1.6后，进行大量的锁优化策略,但是与Lock相比synchronized还是存在一些缺陷的：虽然synchronized提供了便捷性的隐式获取锁释放锁机制（基于JVM机制），但是它却缺少了获取锁与释放锁的可操作性，可中断、超时获取锁，且它为独占式在高并发场景下性能大打折扣。

在介绍Lock之前，我们需要先熟悉一个非常重要的组件，掌握了该组件JUC包下面很多问题都不在是问题了。该组件就是AQS。

AQS，AbstractQueuedSynchronizer，即队列同步器。它是构建锁或者其他同步组件的基础框架（如ReentrantLock、ReentrantReadWriteLock、Semaphore等），JUC并发包的作者（Doug Lea）期望它能够成为实现大部分同步需求的基础。它是JUC并发包中的核心基础组件。

AQS解决了实现同步器时涉及当的大量细节问题，例如获取同步状态、FIFO同步队列。基于AQS来构建同步器可以带来很多好处。它不仅能够极大地减少实现工作，而且也不必处理在多个位置上发生的竞争问题。

在基于AQS构建的同步器中，只能在一个时刻发生阻塞，从而降低上下文切换的开销，提高了吞吐量。同时在设计AQS时充分考虑了可伸缩行，因此J.U.C中所有基于AQS构建的同步器均可以获得这个优势。

#### 基本功能

同步器至少要有以下两种类型的方法acquire和release

* acquire：至少要有一个操作能实现对调用线程的阻塞，直到同步器允许它进行操作。
* release：至少要有一个操作能用一种方式解锁一个或者更多个已经阻塞的线程改变同步状态。

同时，同步器还需要支持以下几种功能：

* 非阻塞式的同步过程尝试(tryLock)
* 可选的超时机制，可以允许程序放弃等待
* 可以通过中断执行取消

而为了适应不同的同步器，同步器要支持两种模式:

* 独占式 exclusive。要保证一次只有一个线程可以经过阻塞点
* 共享式 shared。可以允许多个线程阻塞点

#### 性能要求

AQS对性能改进的关注点不是主要在于减小空间的开销和时间的开销。

原因是：对于开发者来说，只在需要的时候构建同步器，实在没有必要为了这部分空间的消耗去压缩空间。与此同时，同步器大部分情况是用在多线程的情况下，产生一些竞争也是可以想象到的。

|  |
| --- |
| AQS的性能重点在于可扩展性 |

AQS的几个改进目标：

* 可预见性的保证同步器的效率，甚至在其发生竞争的情形下。
* 减少那些已经被允许通过阻塞点的线程但是没有通过的消耗时间。

对比自旋锁来说，自旋锁的响应速度很快，但是在线程竞争特别激烈的情况下，由于大量的内存读取，会降低其响应的速度。

|  |
| --- |
| AQS的框架必须能够提供一些监视和检查的基本操作，以便用户发现和缓解瓶颈。例如：提供一个方式，决定多少个线程会被阻塞。 |

#### 设计与实现

acquire和release操作的伪代码可以很容易的写出来如下：

|  |
| --- |
| acquire{  while (同步状态不允许获取) {  若当前线程没有入队，那么就将其入队；  可能会阻塞当前线程；  }  如果它已经入队了，就将其出队  }  release {  更新同步状态  if(状态表示允许一个阻塞的线程去acquire)  释放一个或多个队列中的线程  } |

##### 同步状态State

AQS采用了int（4个字节）的变量来持有同步状态。使用getState， setState， compareAndSetState方法来进行状态的获取和更新。

以上的方法，它们依赖于volatile这个机制来进行读写，compare-and-swap机制去实现compareAndSetState方法。（CAS操作如果不清楚的可以自行搜索相关的内容）

AQS是一个抽象类，它的tryAcquire和tryRelease方法都需要子类去实现。两个方法都支持传入一个int类型的参数。这个参数主要用来实现不同子类功能的。【例如】：reentrant lock，当在返回一个条件等待后重新去获取锁权限是，它会重新建立一个递归计数。

##### 阻塞

AQS没有采用Thread.suspend和Thread.resume这两种方式，以上两种方式都有严重的安全问题，容易造成死锁等。

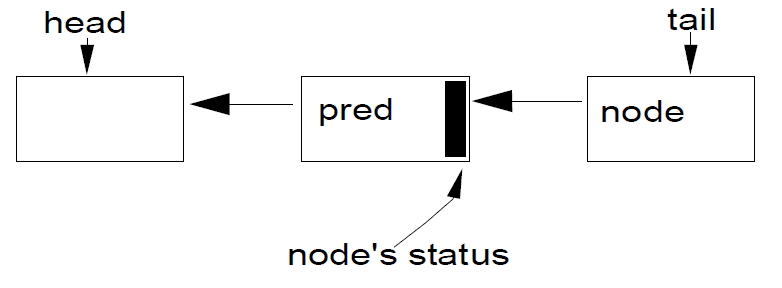
AQS采用了java.util.concurrent.locks包下的LockSupport类。该类可以响应中断操作，可以设置超时时间等。此机制与Win32内的“消费事件”机制，Linux NPTL线程库的方式类似。

##### 核心队列

AQS框架的核心是阻塞线程的队列。也就是一个FIFO（先进先出）队列。AQS不支持基于优先级的同步器。

AQS的锁策略采用的CLH而不是MCS，原因是CLH要比MCS更适合处理取消和超时。

CLH队列的入队和出队操作是与它的锁操作息息相关。它有两个原子操作更新域，head和taiil。初始化时，将指向一个虚假的节点。



每个节点的release状态都保存在它的前驱节点内，while (pred.status != RELEASED);后就可以开始自旋。若持有前驱节点的域，CLH锁可以处理超时和其他形式的取消操作。

AQS对CLH机制有两点修改。

<https://www.cnblogs.com/waterystone/p/4920797.html>

https://blog.csdn.net/teen11/article/details/69371876?locationNum=16&fps=1