链滴

并发编程学习笔记

作者：skyesx
原文链接：https://ld246.com/article/1478096664889
来源网站：链滴

<h2 id="toc_h2_0">并发编程基础</h2>
<h3 id="toc_h3_1">上下文切换</h3>
上下文切换是CPU中一个消耗较大的操作，要尽量减少上下文切换，一些方法：

无锁并发编程。避免使用锁，用Hash等方法，分散竞争
CAS算法。也叫乐观锁算法，低冲突的时候特别高效率
避免创建不必要的线程。
协程。在单线程里实现多任务的调度，并在单线程里维持多个任务间的切换(稍了解)

生产环境中减少上下文切换可以用jstack dump线程快照，并分析原因
<h3 id="toc_h3_2">死锁</h3>
同样可以通过jstack分析
<h2 id="toc_h2_3">JAVA并发机制的底层实现原理</h2>
<h3 id="toc_h3_4">volatile的应用</h3>
某线程对volatile 变量进行写入操作后，其他线程读取volatile变量时，能获得volatile的最新值
X86汇编实现原理：写入volatile变量时，JVM会向处理器发送一条LOCK前缀的指令，将这个变对应的缓存行写入主存。同时，其他的处理器会嗅探总线上传播的数据以此判断自己的缓存是否过期若发现过期，则将对应的缓存行置为失效,更多内容参考 缓存一致性（Cache Coherency）入门
<h3 id="toc_h3_5">synchronized的实现原理与应用</h3>
基础设定：每一个JAVA对象都可以作为锁

对于实例同步方法，锁是当前实例对象
对于普通静态同步方法，锁是当前的类Class对象
对于同步方法块，锁是括号里指定的对象

<h4 id="toc_h4_6">JAVA对象头</h4>
synchronized用的锁是存在JAVA对象头里的。
若对象为数组，那么 对象头长度为 3字宽（1字宽 为 32bit/64bit 视虚拟机位数而定） 若对象数组，那么 对象头长度为 2字宽
JAVA对象头
<table>
<thead>
<tr><th>位置</th><th>内容</th><th>描述</th></tr>
</thead>
<tbody>
<tr>
<td>第一字宽</td>
<td>MarkWord</td>
<td>存储对象的hashCode或锁信息</td>
</tr>
<tr>
<td>第二字宽</td>
<td>Class Metadata Address</td>
<td>存储到对象类型数据的指针</td>
</tr>
<tr>
<td>第三字宽</td>
<td>Array Length</td>
<td>数组的长度（如果对象是数组）</td>
</tr>

</tbody>
</table>
MarkWord里，最后两位为 锁标志位，Markword其余位置 存储信息的含义 会随着 锁标志位改变而改变。64位与32位JVM存储的Markword格式不一样
32位JVM锁标志位 的含义：

01 及偏向锁位置为0 则为 无锁状态
01 及偏向锁位置为1 则为 偏向锁状态
00 表示轻量级锁
10 表示重量级锁
11 表示GC标志

<h4 id="toc_h4_7">锁的升级与对比</h4>
JAVA SE 1.6为减少获得锁和释放锁带来的性能消耗，引入了 “偏向锁” 和 “轻量级锁” 。 在JVA SE 1.6 中，锁一共有4种状态：无锁，偏向锁，轻量级锁，重量级锁。这几个状态会随着竞争情逐渐升级。锁从偏向锁升级到非偏向锁后，不会再变成偏向锁。
<h5 id="toc_h5_8">1.重量级锁</h5>
使用系统的互斥量实现。 使用睡眠唤醒机制竞争锁，适用于高竞争的场景，追求高吞吐量。
<h5 id="toc_h5_9">2.轻量级锁</h5>
使用自旋提高竞争效率，追求高响应。当另外一个进程竞争一个已被当前线程获取的锁时，锁会胀，升级成重量级锁。升级成重量级锁后，释放锁后，再获取锁，会再次变成轻量级锁。
轻量级锁及重量级锁 都会将markword写入到对应获取了锁的线程的栈桢里，并在markword里存一个指针，指向 对应的栈桢。释放锁时，把这个栈桢保存的内容覆盖回 markword里
<h5 id="toc_h5_10">3.偏向锁</h5>
经HotSpot作者研究发现，大多数情况下，锁不仅不存在多线程竞争，而且总是由同一线程多次得，为了让线程获得锁的代价更低而引入偏向锁
线程使用CAS向一个可偏向，且未偏向的对象 竞争偏向锁，若竞争成功，则在markword上写上己的线程编号，后续访问检查markword，如果依然markword还是写着自己的线程编号，则无需CA竞争偏向锁。
若有另外一个线程竞争这个锁时，偏向锁就会视情况，看当前是否处于锁定状态，若是，则偏向变成轻量级锁，若否，则变成无锁状态，但markword会标志其不可偏向。
<h5 id="toc_h5_11">锁的优缺点对比</h5>
<table>
<thead>
<tr><th>锁类型</th><th>优点</th><th>缺点</th><th>适用场景</th></tr>
</thead>
<tbody>
<tr>
<td>重量级锁</td>
<td>在竞争大的时候，使用唤醒机制能节约CPU资源</td>
<td>在竞争较少的情况下，重量级锁会使得线程进入等待状态。当有线程释放锁资源时，才会唤醒程，让线程竞争锁</td>
<td>悲观锁思想，适用于竞争大，追求吞吐量的场景</td>
</tr>
<tr>
<td>轻量级锁</td>
<td>在竞争小的时候，线程可以快速获得锁，免去进入等待队列的时间</td>
<td>当竞争大的时候会加重CPU消耗</td>
<td>采用乐观锁思想，适用于竞争较少的场景</td>
</tr>
<tr>
<td>偏向锁</td>
<td>在无竞争的情况下，偏向锁的性能可接近非同步方法的性能</td>
<td>当竞争发生时，偏向锁的撤销需等到全局安全点，竞争消耗较大</td>

<td>适用于无线程竞争的场景</td>
</tr>
</tbody>
</table>
<h4 id="toc_h4_12">原子操作的实现原理</h4>
<h5 id="toc_h5_13">CPU实现原子操作的方式</h5>

总线锁（锁住时，其他核都不能访问任意内存）
缓存锁（只锁特定的缓存行。但存在不能锁缓存行的情况，如要锁的数据大于一个缓存行）

<h5 id="toc_h5_14">JAVA实现原子操作</h5>

使用循环CAS（利用CPU提供的CMPXCHG指令）
存在问题：

ABA问题
自旋CAS损耗CPU
只能保证一个共享变量的原子操作

<s>使用锁机制实现原子操作</s>

<h2 id="toc_h2_15">JAVA内存模型</h2>
<h3 id="toc_h3_16">JAVA内存模型的基础</h3>
<h4 id="toc_h4_17">并发编程模型的两个关键问题</h4>

线程之间如何通讯

一种形式：共享内存（通过写读内存中的公共状态进行隐式通讯）
另一种形式：消息传递（没有公共状态，通过发送消息来显示进行通讯）

线程之间如何同步（用于控制线程间操作发生相对顺序的机制）

共享内存的并发模型里，同步是显式进行的（程序员必须显式指定某些操作的互斥执行）
消息传递的并发模型里，同步是隐式进行的（消息发送一定在消息接收前）

<h4 id="toc_h4_18">JAVA内存的抽象结构</h4>
每个线程都有其独立的 工作内存，然后有一个公共的主存。
线程写入一个共享变量何时对另外一个线程可见由JMM决定。
<h4 id="toc_h4_19">指令重排序</h4>
指令可能重排序的位置：
编译重排序：在不改变单线程程序语义的情况下，改变指令执行顺序 CPU指令重排序：在不影数据依赖的情况下，对CPU指令重排序，以提高CPU流水线使用效率。意思跟编译重排序差不多 内存统重排序：CPU通过缓存操作主存，且读写缓存都是批量的。因此这个也算是指令重排序。
JMM会根据自身的定义，在某些代码位置插入一些 内存屏障（memory barriers） 阻止一些特的重排序,这里的 重排序指代的是 主存 的重排序，而非CPU缓存中操作的重排序。
<h4 id="toc_h4_20">并发编程模型的分类</h4>
现代的CPU都会使用缓存来填补 CPU操作与内存操作之间的速度差异。CPU可以批量读取数据缓存中，也可以批量将数据写回到内存里。
那么这样一种场景就出现了：CPU写了缓存，但还没会写到内存，此时又需要从主存中读取数据载到缓存中。

如果上述场景要求：

加载内存数据到缓存前，必须回写缓存到内存中，那么CPU整体的性能将会大大降低，无法充分挥缓存的作用。

无需写回内存就可以从内存加载新数据的话能最大程度利用缓存的性能，但是！这样的话对于存的实际操作顺序就由 程序代码定义中的 先写后读 变成了 先读后写！当有其他线程依赖于这些 读/写的变量进行逻辑判断的话，就会出现难以理解的现象。

然而由于第二种选择能大大提升CPU的效率，因此，基本所有CPU都允许2中的情况出现，也就说，允许 写-读 重排序出现。当然，这种重排序允许的前提是不修改单线程的语义，也就是在单线程，写-读重排序操作之间没有数据依赖关系。
可以猜测出除了，写读重排序外，还有另外三种重排序形式，一共四种重排序

读读
读写
写写
写读

按照我们之前的分析，我们可以判断，读读，写写这两种重排序对 优化缓存使用方面都没有较的作用，但或许能为CPU流水线乱序执行提供较为宽松的支持。
至于读写重排序，可以优化的场景就是，缓存满了，还要读入新的数据，这时就把一些写操作提进行，更改完后写回内存，再读取内存数据覆盖写回了的那部分的缓存
根据那种重排序规则是允许的，设定了几种并发编程模型。
只允许 写读 重排序的：X86，SPARC-TSO 允许所有的重排序的：IA64,PowerPC
上面的重排序都是性能优化项，但如果涉及到线程同步，那么必须在某些关键的时序中禁止重排，这样其他线程才能准确的获得数据状态。为了禁止重排序，内存屏障就被设计出来了。
JMM定义了以下四种内存屏障：

读读屏障
写写屏障
读写屏障
写读屏障

其中写读屏障同时具有其他3种屏障的作用。同时该屏障也是一个消耗很大的操作，因为需要将有的缓存回写到内存中。
<h4 id="toc_h4_21">Happens-before</h4>
happens-before是一系列规则，它定义了哪些操作有happens-before的关系。若A happens-before B，那么A所有操作都是对B可见的。
一个Happens-before规则的背后对应的就是 JMM调整编译器重排序，调整CPU重排序的过程。/p>
JMM通过happens-before规则将程序员从繁琐的重排序规则中解脱出来。
<h3 id="toc_h3_22">顺序一致性</h3>
顺序一致性内存模型是一个理论参考模型，这个模型最为符合常人的逻辑思维习惯。在这个模型，所有线程对内存的操作都马上对其他线程可见，且线程内的操作不会进行重排序，所有线程的所有作都是串行执行的，不存在并发的关系。
<h4 id="toc_h4_23">数据竞争与顺序一致性</h4>
JMM定义数据竞争： 线程A写变量X 线程B读变量X 且读写操作之间没有通过同步来排序
当没有正确同步时，程序会出现出一些难以理解的结果。但只要进行正确同步，那么JMM将会证其执行结果与在 顺序一致性内存模型下 执行的结果一致
<h3 id="toc_h3_24">Volatile的内存语义</h3>

一个声明为volatile的变量 对其进行读取操作时，总能获得其他线程修改的最新的值 对volatile量的读写是原子的，包括double和long
从JDK5开始，volatile变量的写读可以实现线程之间的通讯，即通过volatile的读写能建立起不线程事件之间的happens-before关系

volatile变量的写 与 锁的释放 有相同的内存语义，JMM会把当前线程的本地内存都刷新到主存去
volatile变量的读 与 锁的获取 有相同的内存语义。JMM会把该线程对应的所有本地内存都置为效，然后从主存中读取所有所需的数据

写读volatile变量实际上是一个消息发送的过程。
<h4 id="toc_h4_25">volatile内存语义的实现</h4>
volatile变量写-读之间要组成一个happens-before的关系的话
volatile写之前要加入一个store-store屏障，volatile写之后要加一个store-load屏障

volatile写之前 加入store-store屏障是为了防止volatile写的变量 先于 代码顺序中的其他写操作到内存中，实现happens-before关系。
volatile写之后 加入store-load屏障是为了防止 volatile写之后的 读/写 语句先于volatile的结果入到内存，store-load是一个全能屏障，防止后续写操作重排序的原因也是为了实现happens-befor，至于防止后续的读操作的重排序，也是为了实现happens-before,防止一种情况出现：提前读取本被其他线程修改的变量（其他线程修改的依据是本线程写入的volatile写而构建的happens-before关）

volatile读之后要加入一个loadload屏障以及loadStore屏障

loadload，loadStore屏障防止 volatile读后面的 读操作先于volatile读,如果顺序反了，那么happens-before关系就不成立了
为什么不防止代码顺序中 volatile前的读在执行时后于volatile读？是因为这里需要保证 代码顺序中 volatile读后的操作 后于 volatile写？

在X86的实现里，只会出现 写-读重排序（写读重排序最能提高缓存使用效率，效能提高最大）因此X86JVM只需要处理掉store-load重排序即可。因此在X86处理器中，写volatile的消耗较大，而volatile基本与正常变量读取的消耗一致。
<h4 id="toc_h4_26">为什么要增强VOLATILE语义</h4>
在JSR-133之前，volatile变量的读写只管其自身的读写，而不保证其他变量也刷新到内存（即保证happens-before关系），但为了实现HAPPENS-BEFORE关系，提供一个更简易统一的编程模，因此专家们做了增强volatile语义的调整
<h3 id="toc_h3_27">锁的内存语义</h3>
锁的获取-释放会建立起happens-before关系

当线程获取锁时，JMM会把线程对应的本地内存置为无效，从而使得被monitor保护的临界区代必须从主内存中读取共享变量。与volatile读一致

当线程释放锁时，JMM会把本地内存中的所有共享变量刷新到内存中。与volatile写一致

以上这么一个 写内存 读内存的过程 本质上是消息传递的过程。
<h4 id="toc_h4_28">锁内存语义的实现</h4>
以ReentranceLock的实现来讲述
ReentranceLock继承自Lock接口，ReetranceLock内部有一个抽象类Sync继承自AbstractQueudSynchroinzer（简称AQS,其核心是维护了一个volatile变量state）。Sync抽象类下有两个实现，一是公平锁实现，一个是非公平锁的实现。

<h5 id="toc_h5_29">我们先看公平锁的实现，</h5>
tryAcquire是获取锁的基本操作，其声明自AQS，各种乐观、悲观的锁获取实现都会调用tryAcqu
re
伪代码：
<pre><code class="hljs">boolean
/span> tryAcquire(<span class=
hljs-keyword">int i){
  if(AQS.state == 0</spa
>){
    if(锁获取的排队队列没有人在排队或自己是排队中的
一人 && CAS AQS的STATE为i成功){
      将锁所归属的线程置为本线程
      return true

    }else{
      return false</spa
>;
    }
  }else{
    if(获取锁的是自己){
      将AQS的state置为原值+i
      return true

    }else{
      return false</spa
>;
    }
}</code></pre>
释放锁tryRealease声明自ABS，当锁TOTALLY FREE的时候返回true,否则返回false.返回true是
知AQS去通知那些等待线程可以去竞争锁了。
<pre><code class="hljs">boolean tryR
alse(int i)
/span>{
  if(尝试释放的线程不是锁拥有者){
    抛出异常
  }else{
    int newState = ABS.STATE-i;
    boolean free = false</sp
n>;
    if(newState == 0</sp
n>){
      将当前锁的的拥有线程置为null
      free = true;
    }
    将AQS的state置为newState//将这个最后赋值是为了将当前
程为null的结果一起刷新到内存中
    return free;
<pre><code class="highlight-chroma"><span class="highlight
cl">}
</code></pre>
</code> <code class="hljs">}</code> </pre>
<h5 id="toc_h5_30">非公平锁</h5>
非公平锁的释放实现与公平锁一样，但 获取稍有不同，非公平锁不考虑等待队列中的排队顺序</
>

```
<pre><code class="hljs">boolean tryAccquire(int i){
    if(AQS.<span class="hljs-keyword">state</span>==<span class="hljs-number">0</span>
{
        if(获取锁的是本线程){
            AQS.<span class="hljs-keyword">state</span> = AQS.<span class="hljs-keyword">s
ate</span> + i;
            return true;
        }else{
            return false;
        }
    }else{
        if(CAS ABS.<span class="hljs-keyword">state</span> = i 成功){
            将当前线程设为AQS的互斥占有者
            return true;
        }else{
            return false;
        }
    }
}</code></pre>
<h5 id="toc_h5_31">AQS.CAS的实现</h5>
<p>AQS的CAS方法为 compareAndSetState(),其调用Unsafe类的compareAndSwapInt方法实现
这个方法有volatile变量读与写的语义。</p>
<p>具有volatile读与写的内存语义意味着编译器不能对 CAS与CAS前后的代码进行重排序。</p>
<p>这个语义的实现在X86里不由内存屏障实现，而由CPU指令 lock cmpxchg实现。lock前缀在单
CPU里是不必要的。</p>
<p>lock前缀的语义如下： 1.确保内存中的 读-改-写 原子执行。执行过程中会 锁总线或者锁缓存 2.
止该指令与 之前之后的指令重排序 3.把缓冲区的数据写入到内存中</p>
<h4 id="toc_h4_32">concurrent包的实现</h4>
<p>JAVA线程通讯方式有以下几种</p>
<ol>
<li>线程A写volatile变量X，随后线程B读volatile变量X</li>
<li>A CAS X ,B读X</li>
<li>A CAS X, B CAS X</li>
<li>A 写 X, B CAS X</li>
</ol>
<p>CAS 及 volatile是整个concurrent包实现的基石 CAS 及 volatile 构成了 AQS,原子变量类，非
塞数据解雇 然后又由这3个构成了concurrent包</p>
<h3 id="toc_h3_33">final域的语义</h3>
<h4 id="toc_h4_34">重排序规则</h4>
<p>对象里的基本对象：</p>
<pre><code class="hljs"><span class="hljs-class"><span class="hljs-keyword">class</span
 <span class="hljs-title">FinalTest</span></span>{
    <span class="hljs-keyword">int</span> i;
    <span class="hljs-keyword">final</span> <span class="hljs-keyword">int</span> j;
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">FinalTest(){
</span></span><span class="highlight-line"><span class="highlight-cl">    i = &lt;span cla
s="hljs-number" style="box-sizing: border-box; color: teal;"&gt;10&lt;/span&gt;;
</span></span><span class="highlight-line"><span class="highlight-cl">    j = &lt;span cla
s="hljs-number" style="box-sizing: border-box; color: teal;"&gt;11&lt;/span&gt;;
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
</code><p><code class="hljs">}</code></p></pre><p></p>
<ul>
```

构造函数中给final对象赋值与随后把这个构造对象的引用传给一个引用变量的操作不能重排序（
果不这做，那其他线程可能访问到未初始化的final对象，实现的方法就是在final域的赋值后，初始化
数结束前 添加一个store-store屏障）
初次读一个包含final域的对象引用，与随后初次读这个final域，这两个操作不能重排序（在大多
处理器中，这是个有依赖关系的操作，不会被重排序，但少数处理器会对其进行重排序，因此 需要加
loadload屏障来解决这个问题，否则外部也有可能读取到未经初始化的final变量）

对象里的对象是final域的话，新增一条重排序规则：
<pre><code class="hljs">class</span
FinalTest2{
<pre><code class="highlight-chroma"><span class="highlight
cl"><span class="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight:
bold;">final <span class="hljs-built_in" style="box-sizing: border-box; colo
: #0086b3;">Object obj;
FinalTest2(){
  obj = <span c
ass="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">ne
 <span class="hljs-built_in" style="box-sizing: border-box; color: #0086b3;"&
t;Object();
}
</code></pre>
</code> <code class="hljs">}</code> </pre>

构造函数内一个final域对象的初始化函数执行完成 与 在构造函数外将 这个对象赋值给一个引用
量 这两个操作不能重排序，否则将有可能使得其他线程读到未初始化完成的final对象

要保证未经初始化的final域不被其他线程看到，实际上还需要另外一个保障：在构造函数中，不
this的引用暴露给其他线程。如以下的初始化函数就将this暴露出去了，这可能导致外部访问到未经
始化的final域。
class FinalTest3{
<pre><code class="hljs">final <span class="hljs-built_
n">Object obj;
  static FinalTest3 leak;
FinalTest3(){
</code> <code class="hljs">obj = new <span clas
="hljs-built_in">Object();
leak = this;
}</code> </pre>
}

有了以上的条件的保证，那么对象里的final域被其他线程可见时，都是初始化完成的

final语义在X86中的实现：因x86只会对store-load重排序，因此无需刻意加入内存屏障即可实现f
nal语义

<h3 id="toc_h3_35">Happens-before</h3>
程序员希望有一个强的内存模型，这样，编程会显得更加直观，更加便于理解。 但是编译器，
化器希望有一个弱的内存模型，这样他们的发挥空间更多，能得到更高的执行性能。
为了在性能与易用性方面取得平衡， 专家们设计了一套 Happens-before规则。程序员只要基于
appens-before规则来编写程序，那么就能够编写出符合预期的程序，Happens-before实现的具体
节由JVM屏蔽。

JMM把happens-before要求禁止的重排序分为了两类：

会影响执行结果的重排序
不会影响执行结果的重排序

对于会影响结果的重排序，JMM一定会禁止其重排序。 但对于Happens-before规则要求禁止但不会影响（单线程）执行结果的重排序，JMM不作要求，可重排亦可不重排。
如此设计，为程序员提供了一个足够强的内存模型，也给编译器和CPU足够的自由优化空间。</
>
Happens-before起源于Lamport的一篇论文,这篇论文可以看我博文中的对应的解析。
在JSR-133中对happens-before关系的定义如下

如果一个操作Happens-before另外一个操作，那么第一个操作的执行结果将对第二个操作可见且第一个操作的执行顺序在第二个操作之前。这是对程序员的保证。
两个操作之间存在hb关系，并不意味着JAVA平台的具体实现必须要按照happens-before关系定的顺序来执行。如果重排序后的执行结果与按hb关系来执行的结果一致，那么这个重排序不非法。是JMM对编译器及处理器重排序的约束原则。

具体的HB规则

程序顺序规则，代码顺序前面的语句HB后面的语句
监视器锁规则，解锁HB加锁
volatile变量规则，写HB读
传递性
start()规则
join规则

<h3 id="toc_h3_36">双重检查锁定</h3>
<pre><code class="hljs">class</span
 DoubleCheckExample{
<pre><code class="highlight-chroma"><span class="highlight
cl"><span class="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight:
bold;">private <span class="hljs-keyword" style="box-sizing: border-box;
olor: #333333; font-weight: bold;">volatile Object obj;

<span class="hl
s-function" style="box-sizing: border-box;">Object <span class="hljs-title" style="box-si
ing: border-box; color: #990000; font-weight: bold;">getLazyObj<span clas
="hljs-params" style="box-sizing: border-box;">(){
    <span class=
hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">if</sp
n>(obj == <span class="hljs-keyword" style="box-sizing: border-box; color: #333333; fo
t-weight: bold;">null){
        <span clas
="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">synch
onized(<span class="hljs-keyword" style="box-sizing: border-box; color: #33
333; font-weight: bold;">this){
            <span cl
ss="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">if&lt
/span>(obj == <span class="hljs-keyword" style="box-sizing: border-box; color: #33333
; font-weight: bold;">null){
                obj = &l
;span class="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;
>new Object();

            }
        }
    }
    <span class=
hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">return&lt
/span> obj;
}
</code></pre>
</code> <code class="hljs">}</code> </pre>
上述加上volatile的作用是，防止初始化时 对obj变量赋值操作跟初始化Obj操作的 赋值重排序。
/p>
以上延时加载还可以使用基于类初始化的解决方案。
一个类，一般情况下会尽量的延迟初始，直到不初始化会影响程序的正确性为止。JVM会保证类
始化只执行一遍，因此，这个机制可以被利用作延迟初始化。实现如下：
<pre><code class="hljs">class</span
 OutClass{
    private static I
nerClass{
        static OutClass lazyObj = <span class="hljs-keywo
d">new OutClass();
    }
<pre><code class="highlight-chroma"><span class="highlight
cl"><span class="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight:
bold;">public <span class="hljs-keyword" style="box-sizing: border-box; c
lor: #333333; font-weight: bold;">static getOutClass{
    <span class=
hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">return&lt
/span> InnerCLass.lazyObj;
}
</code></pre>
</code> <code class="hljs">}</code> </pre>
<h3 id="toc_h3_37">JAVA内存模型总结</h3>
顺序一致性内存模型是一个理论的参考模型，在设计JMM及CPU内存模型时都会以此为参照，
放松一些内存模型的限制，以期达到更好的性能。
对不存在依赖关系的指令：

允许store-load重排序，得到TSO(total store oredering)内存模型
1的基础上允许store-store重排序，得到PSO(partial store order)内存模型
继续允许 load-store 及 loadload重排序，得到RMO(Relaxed Memory Order)内存模型

由于JMM的内存模型强于大多数的CPU内存模型，因此JMM需要在特定的位置插入内存屏障，
实现JMM的Happens-before关系。JMM会通过插入内存屏障为不同平台的程序员提供一个一致的
存模型JMM.
可见性保证：

单线程程序不存在可见性问题
正确同步的多线程程序的执行将具有与顺序一致性内存模型一样的结果。
未同步、未正确同步的多线程程序，JMM为其提供了最小安全保障：读取到的值要么是某个线程
入的值，要么是默认值。

64位数据的非原子性，跟这个最小安全保障没关系。也就是说，这个最小安全保障并不保证不会
取到写到一半的64位数据
JSR-133对旧内存模型的修补

禁止了对volatile变量读写操作的一些重排序，以实现happens-before语义
增强final的语义，为其添加了两个重排序规则，保证了在 构造函数this不leak的情况下，final变被其他线程访问到时，都经过了初始化。

<h2 id="toc_h2_38">JAVA并发编程基础</h2>
<h3 id="toc_h3_39">线程</h3>
<h4 id="toc_h4_40">线程状态： </h4>

NEW 初始状态，线程被构建 还没有执行start方法
RUNNABLE 运行中，包括 就绪(READY) 及 运行(RUNNING) 两种
BLOCKED 阻塞状态，表示线程阻塞于锁
WATING 等待状态，表示线程等待 其他的线程的通知/中断
TIME_WATING 超时等待状体，超过一定时间后自动返回
TERMINATED 终止状态，线程已经执行完毕。

NEW->RUNNABLE{Thread.start()}

RUNNABLE->WATING{Object.wait(),Thread.join(),LockSupport.park()}

WAITING -> RUNABLE {Object.notify(),Object.notifyAll(),LockSupport.unpark(Thread)}
/p>

RUNNABLE -> BLOCKED {等待进入synchronized方法或方法块}

BLOCKED -> RUNNABLE {成功获取monitor}

RUNABLE -> TIME_WAITING { Object.wait(long),Thread.sleep(long),LockSupport.park
anos(long),LockSupport.parkUntil(),Thread.join(long) }

TIME_WAITING -> RUNNABLE {超时等待时间结束，Object.notify(),Object.notifyAll,Lock
upport.unpark(Thread)}

8.RUNNABLE -> TERMINATED {程序执行完毕}
<ol start="9">
READY -> RUNNING {系统调度}
RUNNING -> READY {Thread.yield(),系统调度}

<h4 id="toc_h4_41">Daemon线程</h4>
Deamon线程是后台支持线程。当不存在前台线程，只有Daemon线程时，整个程序就会退出。
/p>
当没有前台线程时，Daemon线程有可能随时终止，不管代码执行到哪里。如在Deamon线程中
finally块，也不一定会执行。
<h4 id="toc_h4_42">中断</h4>
中断就是其他线程向本线程打招呼。至于打招呼后要干嘛，则自行定义。有很多声明抛出Interup
edException的方法都会将 Interupted标识复位，然后抛出异常，等待你处理。 自己在代码中也可

主动监测中断，并处理中断
中断最常用的目的是 终止/取消 任务。
<h4 id="toc_h4_43">过期的suspend(),resume(),stop()</h4>
被禁用的原因是suspend()执行时并没有释放锁等资源，可能造成死锁。stop()方法会马上停止程，导致一些打开的资源没有释放。suspend,resume可以使用等待/通知机制代替
<h4 id="toc_h4_44">安全的终止线程</h4>

通过中断通知
通过共享变量通知

<h3 id="toc_h3_45">线程间通讯</h3>
<h4 id="toc_h4_46">volatile</h4>
<h4 id="toc_h4_47">synchroized</h4>
<h4 id="toc_h4_48">等待/通知机制</h4>
生产者消费者模式用于隔离事件 与 响应 的实现。但响应线程何时执行是一个难以处理的问题。家希望有一个 高效的通知机制（无需自旋浪费CPU，响应时间也要足够的快），因此等待通知机制就发明出来了。

notify() 通知一个在对象上等待的线程，使其从wait()方法返回。而返回的前提是该线程获取到对象的锁。
notifyALl() 通知所有等待在该对象上的线程。
wait()

这个机制构建于synchronized（monitor）的基础上，进行notify,wait都需要获得对应对象的monitor.线程wait之后被notify但没有获得monitor锁时，会处于blocked状态
<h4 id="toc_h4_49">管道输入/输出流</h4>
这个管道流 与 文件流/网络流 的最主要区别为，其用于线程间通讯，介质为内存
PipedReader/PipedWriter
<h4 id="toc_h4_50">Thread.join</h4>
<h4 id="toc_h4_51">ThreadLocal</h4>
<h2 id="toc_h2_52">JAVA中的锁</h2>
锁是用来控制多个线程访问共享资源的方式，其目的与synchroinzed一致。synchronized的使更便捷；锁的使用需要显式处理比较麻烦，但其可以实现一些比较复杂的锁的形式，如

非阻塞地获取锁
能被中断地获取锁
超时获取锁

<h3 id="toc_h3_53">Lock Api</h3>

lock()
lockInterruptibly()
tryLock() 尝试一次，若无法获取则马上返回
tryLock(long time,TimeUnit unit) 超时获取锁，可被中断
unLock()
newCondition() 获取等待通知组件，该组件和当前的锁绑定，当前线程只有获得了锁，才能调该组件的wait()方法，而调用后，将会释放锁。

<h3 id="toc_h3_54">队列同步器</h3>
AbstractQueuedSynchronizer是用来实现锁及其他同步组件的基础框架，它用state变量表示步状态，使用内置的FIFO队列来完成资源获取线程的排队工作。
其主要使用方式是继承。同时推荐在所需实现的工具的内部，用一个内部类继承AQS，并用聚合形式进行工作。
同步器简化了锁的实现，屏蔽了 同步状态管理，线程排队，等待与唤醒等底层操作。

<h4 id="toc_h4_55">同步队列</h4>

这个队列是一个先进先出的队列，在AQS内部用于维护同步线程的顺序,当线程尝试获取同步状态败时，就会往这个队列通过自旋CAS在队列末加一个节点，然后自旋获取同步状态的操作，若获取失，则阻塞，等待被唤醒（可能是同步状态释放，也可能是中断导致），唤醒后再自旋。
同步队列节点有以下状态：
CANCELLED,值为1，表示由于 超时 或者 中断 等原因取消等待
SIGNAL,值为-1，后续节点的线程处于等待的状态，当前节点 如果释放了 同步状态 或者 被取消将会通知后续节点，使得后续节点得以运行。
CONDITION，值为-2，节点在等待队列中
PROPATION,值为-3，表示下一次共享式同步状态获取将会无条件的被传播下去
INITIAL,值为0，初始状态

同步队列的头结点是获得锁的线程所在结点，或者刚刚释放锁的结点。但有一种情况例外，那就队列为空，tryAcquire一次就成功，那么等待队列不会加上当前线程所在的NODE。但是缺少了这个ODE，获取同步状态失败的线程将无法统一后续处理，因此获取同步状态失败的线程发现没有头结点，会给其初始化一个头结点，然后自己的结点加在头结点之后。然后在自旋CAS中将前置NODE的waiStatus改为signal，这样 释放同步状态时，处于等待状态的线程才会被唤醒。
获取锁的顺序基本上是队列中的排队顺序，除非有线程没有加入到队列中排队，直接用tryAquir（and so on...）一次就获得了锁。如ReentrantLock的公平锁和非公平锁的实现的区别就在于，公平在判断同步队列存在排队节点时，则不允许其获得锁，而是让其也加入同步队列中进行排队。
使用tryAquireShared的时候，如果成功，且当前线程在队列中，那么将会依次唤醒后续shared点。后续shared节点线程将会将前面获取成功的节点出队列，并将自己置为头结点，直到最后一个连的shared节点。

<h4 id="toc_h4_56">CONDITION</h4>
跟OBJECT的notify一样一样的用法。但比其功能更强大，能设立多个CONDITION，根据场景NTIFY不同队列的线程。进入WATING首先要获得锁，然后进入CONDITION的WATING队列，然后释锁。被NOTIFY后会进入 同步队列 ，等待获取之前的锁，然后继续运行。
<h3 id="toc_h3_57">一些锁的实现思路</h3>
<h4 id="toc_h4_58">TWINS LOCK</h4>
只允许两个线程获得锁的不可重入锁
因为是两个的线程都可获得锁，因此用AQS的SHARE相关方法会比较合适。虽然在一个线程释锁，就唤醒另外一个锁的场景用 独占形式的方法 也可以。但AQS的SHARE形式比较合适。
将可获得锁的线程个数参数化为 N（TWINS则N=2）,将status初始化为N，表示还有N个资源可获取。
tryAcquireShared伪代码如下：
<pre><code class="hljs">int</spa> tryAcquire(int acquire){
    for(;;){//循环CAS是因有可能同时两个线程都在竞争锁，且两个都有可能竞争成功，为了节约进入队列进行等待的消耗，因采用循环CAS
        int status = getStatus();
        int newStatus = status - acquire;
        if(status > 0</spa> && compareAndSetStatus(status,newStatus))
            return newStatus;/大于0表示可唤醒排队中的线程尝试获取同步状态，等于0表示无需通知后续等待线程
        else
            return newStatus;/小于0表示获取失败
    }
}</code></pre>

tryReleaseSHared伪代码如下：
<pre><code class="hljs">boolean
/span> tryRelease(<span class=
hljs-keyword">int release){
    for(;;){//存在同时释放
的场景
        int status = getStatus();
        int newStatus = status + release;
        if(compareAndSetStatus(status,newStatus)){
            return true

        }
    }
}</code></pre>
<h4 id="toc_h4_59">ReentranceLock</h4>
可重入锁。排它，但锁拥有者可重入。
实现思路，通过重写 tryAcquire,tryRelease 实现。
<pre><code class="hljs">//本实现为非公平锁实现。若要实现公
锁，则需要检测自己是否队列第二个节点，若是，才尝试去获取锁。
pubulic boolean tryAcquire<sp
n class="hljs-params">(int acquire){
    int status = getStatus();
    if(status != 0){
        //检测自己是否OWNER
        if(Thread.currentThread() == getExclusiveThread())

            setStatus(status + acquire);
            return true;
        } else {
            return false;
        }
    }else{
        if(compareAndSetStatus(<span class="hljs-numbe
">0,acquire)){
            setExclusiveThread(Thread.currentThread());
            return true;
        }else{
            return false;
        }
    }
}
public boolean tryRelease(<span class="hljs-keyword
>int release){
if(Thread.currentThread() != getExclusiveThread()){
throw new Illegal
ccessException();
}
<pre><code class="highlight-chroma"><span class="highlight
cl"><span class="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight:
bold;">int newStatus = getStatus() - release;
<span class="hl
s-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">if</span
gt;(newStatus < <span class="hljs-number" style="box-sizing: border-box; color: teal

">0){
    <span class=
hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">throw&lt
/span> <span class="hljs-keyword" style="box-sizing: border-box; color: #333333; font
weight: bold;">new OverFlowException();
}<span class="hl
s-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">else</sp
n> {
    setStatus(newS
atus);
    <span class=
hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">if</sp
n>(newStatus == <span class="hljs-number" style="box-sizing: border-box; color: teal;
>0){
        setExclusiveT
read(null);
        <span clas
="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">retur
 true</span&
t;;
    }<span class=
hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">else</
pan>{
        <span clas
="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">retur
 false</span&
t;;
    }
}
</code></pre>
</code> <code class="hljs">}</code> </pre>
<h4 id="toc_h4_60">ReentranceReadWriteLock</h4>
可重入，读锁共享，写锁独占，写锁可降级读锁，可区分公平非公平实现。
实现思路： 利用status高16位表示读锁的总重入次数，低16位表示写锁的重入次数。 读锁中每线程的重入次数使用各自的ThreadLocal记录，staus记录的是总次数。写锁获取者只有一个，无需类的ThreadLocal变量。
<pre><code class="hljs">boolean tryA
quire(int acquire)<
span>{
    int status = getStatus();
    int writeStatus = getWriteStatus(status);
    int readStatus = getReadStatus(status);
    if(writeStatus != 0</spa
>){
        if(getExclusiveThread() = Thread.currentThread()){
            setStatus(calcStatusFromWR(writeStatus+acquire,readStatus))
            return true;
        }else{
            return false;
        }
    }else{
        if(readStatus == 0</s
an>){
            if(isSecondNodeMatter() && compar

AndSetStatus(0,calcStatusFromWR(acquire,<span class=
hljs-number">0))){//isSecondNodeMatter方法用于保
公平性
setExcusiveThread(Thread.currentThread());
return true

}else{
return false</span
;
}
}else{
retun false;
}
}
}
int <span class="hljs-t
tle">tryAcquireShared(int<
span> acquire){
<pre><code class="highlight-chroma"><span class="highlight
cl"><span class="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight:
bold;">for(;;){
<span cl
ss="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">int&l
;/span> status = getStatus();
<span class=
hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">int</s
an> writeStatus = getWriteStatus(status);
<span class=
hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">int</s
an> readStatus = getReadStatus(status);

<span class=
hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">if</sp
n>(writeStatus != <span class="hljs-number" style="box-sizing: border-box; color: teal;
>0){
<span clas
="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">if</
pan>(Tread.currentThread() == getExclusiveThread()){
setStatus(c
lcStatusFromWR(writeStatus,readStatus + acquire));
<span cl
ss="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">retu
n
;<span class="hljs-comment" style="box-sizing: border-box; color: #999988; f
nt-style: italic;">//acquire success but not propogation
}<span clas
="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">else&lt
/span>{
<span cl
ss="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">retu
n -
;<span class="hljs-comment" style="box-sizing: border-box; color: #999988; f
nt-style: italic;">//can't acquire
}

    }<span class=
hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">else</
pan>{
        <span clas
="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">if</
pan>(isSecondNodeMatter() && compareAndSetStatus(status,calcStat
sFromWR(0</
pan>,readStatus + acquire))){
            threadLoca
Acquire.add(acquire);
            <span cl
ss="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">retu
n
;
        }
    }
}
</code></pre>
}
boolean tryRelease<span c
ass="hljs-params">(int release){
int status = getStatus();
int writeStatus = getWriteStatus(status);
int readStatus = getReadStatus(status);
int newWriteStatus = writeStatus + release;
if(newWriteStatus == 0</s
an>){
setExclusiveThread(null);
setStatus(calcStatusFromWR(newStatus,readStatus));
return true;
}else{
setStatus(calcStatusFromWR(newStatus,readStatus));
return false;
}
}
boolean tryReleaseShared
span class="hljs-params">(int release)

int threadLocalAcquired = <span class="hljs-keyword"
this.threadLocalAcquired.get();
if(threadLocalAcquired <= <span class="hljs-number
>0){
throw new Runti
eException();
}
<pre><code class="highlight-chroma"><span class="highlight
cl"><span class="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight:
bold;">int newLocalAcquired = threadLocalAcquired - release;
<span class="hl
s-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">if</span
gt;(newLOcalAcquired < <span class="hljs-number" style="box-sizing: border-box;
olor: teal;">0){
    <span class=
hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">throw&lt

/span> new RuntimeException();
} else</sp n> if(newLocalAcquired > 0){
    for</s an>(;;){
        int< span> status = getStatus();
        int< span> writeStatus = getWriteStatus(status);
        int< span> readStatus = getReadStatus(status);
        if</ pan>(writeStatus > 0){
            setStatus(c lcStatusFormWR(writeStatus,readStatus - released));
        } else& t;/span> {
            if&lt /span>(!compareAndSetStatus(status,calcStatusFormWR(writeStatus,readStatus - released)){
                c ntinue;
            }
        }
        this&lt /span>.threadLocalAcquired.set(newLocalAcquired);
        retur false</span& t;;
    }
} else</sp n>{
    for</s an>(;;){
        int<

span> status = getStatus();
        <span clas
="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">int<
span> writeStatus = getWriteStatus(status);
        <span clas
="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">if</
pan>(writeStatus > <span class="hljs-number" style="box-sizing: border-box; co
or: teal;">0){
            setStatus(c
lcStatusFormWR(writeStatus,readStatus - released));
            <span cl
ss="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">this
lt;/span>.threadLocalAcquired.<span class="hljs-built_in" style="box-sizing: border-box;
color: #0086b3;">set(newLocalAcquired);
            <span cl
ss="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">retu
n false</span
gt;;
        } <span cla
s="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">else&
t;/span> {
            <span cl
ss="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">int&l
;/span> newReadStatus = readStatus - released;
            <span cl
ss="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">if&lt
/span>(compareAndSetStatus(status,calcStatusFormWR(writeStatus,newReadStatus))){
                i
(newReadStatus = <span class="hljs-number" style="box-sizing: border-box;
olor: teal;">0){
                    <s
an class="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;"&g
;return true</
pan>;
                }<spa
 class="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">e
se{
                    <s
an class="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;"&g
;return false</
pan>;
                }
            }
        }

        <span clas
="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">retur
 <span class="hljs-literal" style="box-sizing: border-box;">false</span&
t;;
    }
}
</code></pre>
</code> <code class="hljs">}</code> </pre>

<h2 id="toc_h2_61">JAVA并发容器和框架</h2>
<h3 id="toc_h3_62">ConcurrentHashMap</h3>
HashMap非线程安全，HashTable效率低下
ConcurrentHashMap使用锁分段技术降低锁竞争。具体实现类似于（不太恰当）一个ConcurenHashMap由多个HashMap组成，每个HashMap都对应于一段HashCode对应的KEY，PUT进去的时需要获得跟HashMap对应的锁。
<h3 id="toc_h3_63">ConcurrentLinkedQueue</h3>
FIFO队列 使用CAS入队及出队。
核心优化技巧是，tail指针(volatile变量),head指针(volatile变量)都允许不是指向队列第一个节点最后一个结点,允许其与第一个节点/最后一个结点的距离小于等于HOPS变量（默认为1）。这么做的因是volatile写的消耗是远远大于volatile读的。而如果要保持 tail永远指向最后一个结点，那么需要两个volatile变量（tail变量 及 node.next,head的情况也类似），因此设计了允许 head不指向第一节点，而是通过链表向前查找的形式查找第一个节点的算法。
<h3 id="toc_h3_64">阻塞队列</h3>
一个队列能够阻塞地插入（队列满则阻塞等待）及阻塞地移除（队列空则阻塞地等待） 那么它是阻塞队列。
阻塞队列的 入队出队 共有四组方法。这四组方法在队列不可用时提供不同的响应，如：抛出异，返回特殊值，一直阻塞，超时退出。
JDK的七个阻塞队列：

ArrayBlockingQueue:数组结构有界阻塞队列
LinkedBlockingQUeue:链表结构无界阻塞队列
PriorityBlockingQueue:支持优先级的无界阻塞队列
DelayQueue:一个使用优先队列实现的无界阻塞队列，其优先顺序根据元素距离执行时间的长短排序。其可以用于实现 缓存定时过期/定时任务 等场景
SynchronousQueue:一个不保存元素的队列，需要 生产者消费者 一起 生产/消费。本队列消费度较快。
LinkedTransferQueue:链表结构无界阻塞队列。与LinkedBlockingQueue的区别是，有tryTranser方法，如果有消费者正在等待消费，那么生产者将会直接将消费元素传递给消费者
LinkedBlockingDeque:链表结构无界双向阻塞队列

阻塞队列的主要实现原理：使用通知模式实现（即使用Condition，不符合条件则循环await）阻塞队列会维护两个Condition（两个等待队列）:full,empty.队列满后，生产者执行put操作时就会阻塞然后在等待队列中等待。当消费者取出元素后，notify full队列中的元素，让其加入同步队列，获取同步状态成功，则可以put数据到queue中。当队列为空，需要消费时的情况与此类似。
<h3 id="toc_h3_65">Fork/Join框架</h3>
分拆任务多线程计算，再合并返回结果的框架。
<h2 id="toc_h2_66">JAVA中的13个原子操作类</h2>
<h3 id="toc_h3_67">原子更新基本类型</h3>

AtomicBoolean
AtomicInteger
AtomicLong

<h3 id="toc_h3_68">原子更新数组</h3>

AtomicLongArray
AtomicIntegerArray
AtomicReferenceArray

<h3 id="toc_h3_69">原子更新引用类型</h3>

AtomicReference
AtomicReferenceFieldUpdater

AtomicMarkableReference ?? 不太理解....

<h3 id="toc_h3_70">原子更新字段类</h3>

AtomicIntegerFieldUpdater
... 用于原子更新已有的字段。

<h2 id="toc_h2_71">并发工具类</h2>
<h3 id="toc_h3_72">CountDownLatch</h3>
等待倒数完成，所有等待的线程都可以执行。不可重用。
<h3 id="toc_h3_73">CyclicBarrier</h3>
等待指定个数的线程进入await阶段后，一起开始执行。
<h3 id="toc_h3_74">Semaphore</h3>
允许指定数量的资源，资源被消耗完后，只能阻塞等待。重入也会减少资源数
<h3 id="toc_h3_75">Exchanger</h3>
两个线程交换数据后继续执行
<h2 id="toc_h2_76">线程池</h2>

coreThreadCount：当线程池中的线程小于coreThreadCount的话，无论有没有空闲线程，都创建新的线程
maxThreadCount:当等待队列已满时，就会新增线程处理提交的任务，如果总的线程数已经达了maxThreadCount，那么就会按照设定的策略（abort,discard,callerRuns,discardOldestQueueNde）处理无法执行的任务。当线程空闲下来时，会将线程关闭到coreThreadCount个。如果队列是界队列，那么maxThreadCount这个参数无效。

shutdown/shutdownNow方法

共同点：拒绝接受新的任务。
不同点：第一个只会对没有执行任务的线程发出中断请求，第二个会对所有运行中线程发出中断求。

建议使用有界队列，这样能及早发现性能问题，并便于排查。
<h2 id="toc_h2_77">Executor框架</h2>

Executor接口

里面只有一个方法 void execute(Runnable task)
接口解耦(decoupling)了 任务的提交 与 任务的运行调度
主要的运用场景类似 将一个Runnable实例提交到Executor里执行，而非手动创建线程，然后调Thread.start()

Future接口

表征一个异步执行的结果
通过get方法可以阻塞地获得执行结构
可以调用Cancel方法尝试终止未完成的任务

传入参数false，表示若任务已开始执行的话就不尝试中断,让任务继续执行，但是任务的状态变成Canceled，isDone返回true
传入参数true,表示任务已经开始执行则发送中断命令，并将任务状态置为已中断,isDone返回tru

cancel成功 并不代表 任务执行过程退出了,只是该任务的返回将会被忽略

方法isDone

其表征是否已经执行完成。
对于Cancel方法执行成功时，不管 任务是否继续执行中

方法isCanceled

当任务执行完成前被成功执行cancel方法，那么isCanceled返回true.

RunnableFuture

继承了Runnable接口及Future接口的一个接口

FutureTask类

RunnableFuture的一个实现
可以将Runnable实例或者Callable实例封装成RunableFuture。
因实现了Runable接口，其可以用Executor调度，也可以直接新建线程调度，也可以在同一个线内调度

ExecutorService接口

继承自Executor接口
里面新增了两类功能

关闭Executor服务相关的方法
shutdown():执行后，拒绝执行接收新任务，但已经在队列中的任务会继续处理，当线程没有任执行时，执行中断操作，通知其退出。
shutdownNow():与shutdown()类似，只不过不再执行已经在队列中但尚未开始的任务，无论线有没有在执行任务，都会对其发出中断请求，请求其退出。
isShutdown():是否调用了shutdown方法
isTerminate():调用shutdown方法后是否所有的线程都已结束任务。
awaitTerminate():等待所有线程执行完毕

新增监控任务运行状态及结果的方法(通过返回的future方法)
Future<T> submit()
List<Future<T>> invokeAll()

AbstractExecutorService

故名思义，其为ExecutorService实现了一个骨架类，提供了invokeAny,invokeAll，将Runnabl封装成Callable等操作
抽象一些公用的部分到这里

ThreadPoolExecutor

ExecutorService的一个重要实现类,继承自AbstractExecutorService
JDK建议使用Executors的工厂方法创建该类的实例

Delayed接口

实现了Comparable接口，使其可以在 优先级队列中进行排序
包含方法getDelay,用于获取本对象距离到时时间还有多长的时间

ScheduledFuture

Mix-in Delayed 及 Future 两个接口
其为ScheduledExecutorService一些有返回值得异步方法的返回结果的表征

RunnableScheduledFuture接口

MIX-IN了RunnableFuture接口及ScheluedFuture
该接口实例的run方法会将执行结果放入到Future的相关输出中
该方法还提供一个方法isPeriodic,表示本任务是否需要周期执行，若是，则将其再放入等待执行队列中

ScheduledExecutorService接口

扩展自ExecutorService
新增一些任务的定时调度功能，如

指定时间后，执行某个task
每隔一段间隔，执行重复执行某个task

ScheduledThreadPoolExecutor类

是ScheduledExecutorService接口的主要实现类，但是JDK建议不直接使用本类，而是使用Executors的工厂方法来创建
使用一个优先队列来维护需要执行的任务的先后顺序(DelayQUeue,内部包含优先级队列，且当列头结点的到时时间到才能从中取出节点。)

