# 第一章 并发编程的挑战

在并发编程时，若希望通过多线程执行任务让程序运行得更快，会面临非常多的挑战： 上下文切换问题，死锁问题，受限于硬件和软件的资源限制问题。

## 1.1 上下文切换

CPU通过时间片分配算法来执行任务，当前任务执行一个时间片后会切换到下一个，切换之前会保存上一个任务的状态。

任务从保存到再加载的过程就是一次上下文切换。上下文切换会影响多线程的执行速度。

**减少上下文切换的方法**：

* 无锁并发编程：多线程处理数据时，可以用一定的方法来避免锁的使用。
* CAS算法：Java的Atomic包使用CAS算法来更新数据，不需要锁
* 使用最少线程：避免创建不需要的线程
* 协程：在单线程中实现多任务的调度，并在单线程里维持多个任务间的切换

减少上下文切换实战：

* 用jstack命令dump线程信息： jstack {pid} > d:/dump12
* 统计所有线程分别处于什么状态：[grep java.lang.Thread.State dump12 | awk ‘{print $2$3$4$5}’ | sort | uniq –c ]
* 打开dump文件查看处于WAITING的线程在做什么
* 减少WAITING状态的线程数目，修改线程池的配置信息，将maxThreads降低

## 1.2 避免死锁的常见方法

* 避免一个线程同时获取多个锁
* 避免一个线程在锁内占用多个资源，尽量保证每个锁只占用一个资源
* 尝试使用定时锁，使用lock.tryLock(timeout)来代替内部锁
* 对于数据库锁，加锁和解锁必须在一个数据库连接里，否则会解锁失败

## 1.3 资源限制

资源限制是指程序并发执行速度受限于计算机硬件资源（带宽、CPU等）或软件资源。

资源限制引发的问题：

受限于资源，并发执行的代码仍然在串行执行，由于上下文的切换，程序不会加快执行，反而会更慢。

解决方法：

对于硬件资源的限制可以使用集群并发执行；对于软件资源的限制可以考虑资源池复用

根据不同的资源限制调整程序的并发度。比如下载文件程序依赖两个资源：带宽和磁盘读写速度；数据库操作涉及数据库连接数，SQL语句执行速度。

# 第二章 并发机制的底层实现原理

Java中所使用的并发机制依赖于JVM 实现和CPU指令

## 2.1 volatile的应用

volatile是轻量级的synchronized，它在并发处理器开发中保证了共享变量的“可见性”，使用恰当的话，比synchronized的使用和执行成本更低，因为它不会引起线程上下文切换。

**Java线程内存模型能够确保所有线程看到的volatile声明变量的值是一致的**。

CPU术语

* **缓冲行**：缓存中可以分配的最小存储单位，处理器填写缓存线时会加载整个缓存线，需要多个主内存读周期
* **原子操作**：不可中断的一个或一系列操作
* **缓存行填充**：当处理器识别到内存中读取操作时可缓存的，处理器读取整个缓存行到适当缓存
* **缓存命中**：若高速缓存行填充操作的内存位置仍然是下次处理器访问的地址，处理器从缓存中读取，而不是从内存中读取
* **写命中**：如果会写的操作数在有效的缓存中时，则将改操作数回写到缓存中，而不是内存中。
* **写缺失**：一个有效的缓存行被写到不存在的内存区域

volatile的实现原则：处理器缓存回写到内存；一个处理器的缓存回写到内存，会导致其他处理器的缓存无效。

## 2.2 synchronized的实现原理与应用

使用synchronized实现同步的基础：Java中每个对象都可以作为锁：

* 对于普通同步方法：锁是当前实例对象
* 对于静态同步方法：锁是当前类的Class对象
* 对于同步方法块：锁是synchronized括号里配置的对象

JVM是基于进入和退出Monitor对象来实现方法同步和代码块同步的。

synchronized用的锁是存在Java对象头里的，如果对象是数组，则虚拟机用3个字宽存储对象头。

java中的锁一共有四种状态：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态。这几个状态会随着竞争情况逐渐升级，但不可降级。

**偏向锁**：

考虑到大多数情况下，锁不仅不存在多线程竞争，而且总是由同一线程多次获得，所以引入了偏向锁。

偏向锁使用了一种等待竞争出现才释放锁的机制，偏向锁的撤销需要等待全局安全点。

-XX:BiasedLockingStartupDelay=0 来关闭偏向锁的启动延迟

-XX:UseBiasedLocking=false 当应用程序所有的锁通常都是处于竞争情况是，可以默认不是用偏向锁，默认会进入到轻量级锁状态

**轻量级锁**：

线程在执行同步块之前，JVM会首先在当前线程的栈桢中穿件用于存储锁记录的空间，并将对象头中的Mark Word复制到锁记录中。

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，若成功则获得锁；若失败，表示其他线程竞争锁，当前线程使用自旋来获取锁，或升级成重量级锁。

**锁的优缺点对比**

|  |  |  |  |
| --- | --- | --- | --- |
| 锁 | 优点 | 缺点 | 使用场景 |
| 偏向锁 | 加解锁无需额外消耗，与非同步方法仅有纳秒级差距 | 若存在锁竞争，会带来额外的锁撤销消耗 | 只有一个线程访问同步块场景 |
| 轻量级锁 | 竞争的线程不会阻塞，提高程序的相应速度 | 若时钟得不到锁的竞争线程，自旋消耗CPU | 追求响应时间  同步块执行速度快 |
| 重量级锁 | 线程竞争不使用自旋，不消耗CPU | 线程阻塞，响应时间慢 | 追求吞吐量  同步块执行时间较长 |

## 2.3 原子操作实现原理

处理器实现原子操作有两个机制：

* **总线锁保证原子性**：总线锁就是使用处理器提供的一个#LOCK信号，当一个处理器在总线上输出此信号时，其他处理器的请求被阻塞，该处理器独占共享内存。
* **缓存锁保证原子性**：锁定缓存行的数据，使用缓存一致性来保证原子性。跨多个缓存行时不能使用。

Java中可以使用CAS实现原子操作。

CAS实现原子操作的三大问题：

* **ABA问题**：CAS需要在操作值时检查值是否发生了变化.若值由A变为B，再变为A，则认为没有变化，可以使用AtomicStampedReference来解决ABA问题。
* **循环时间长开销大**：自旋CAS长时间执行不成功，会给CPU带来较大的开销
* **只能保证一个共享变量的原子操作**：使用AtomicReference类来保证引用对象之间的原子性，可以将多个变量放到一个对象里。

# 第三章 Java内存模型（JMM）

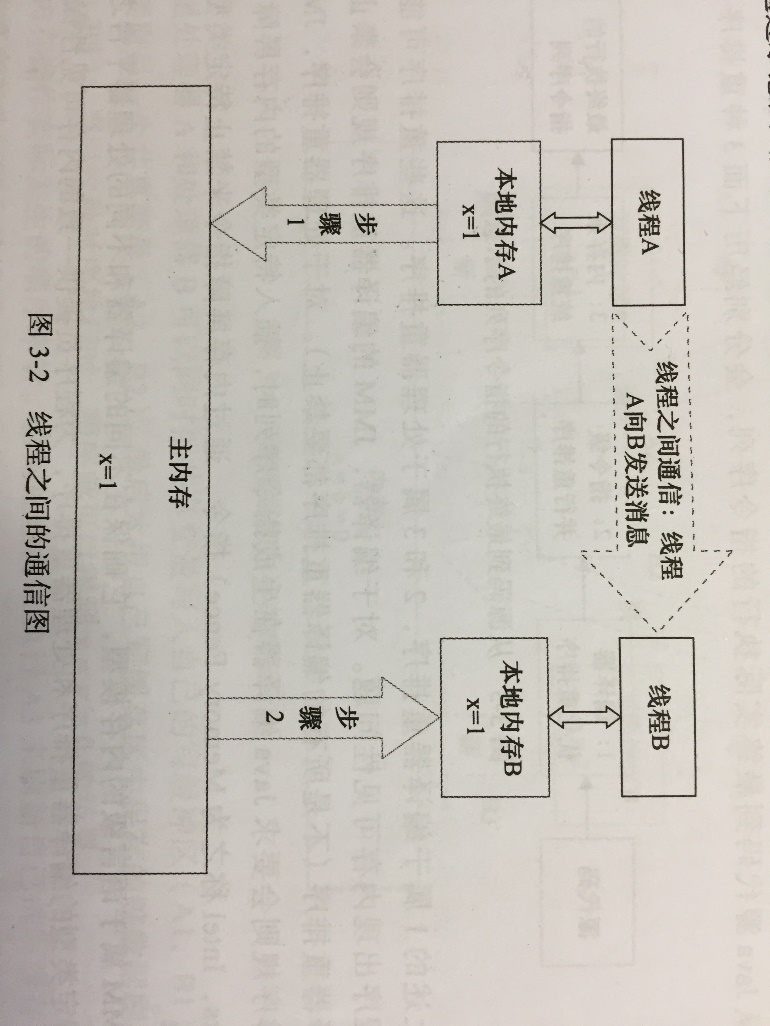
Java内存模型的基础，顺序一致性，同步原语

## 1. Java内存基础

线程之间如何通信和线程之间如何同步是并发变成需要处理的两个关键问题。在命令式编程中，线程的同步机制一般有两种：共享内存和消息传递。

Java中，所有实例域、静态域和数组元素都存储在堆内存中，堆内存在线程之间共享，所以会有可见性问题。

JMM通过控制主内存与每个线程的本地内存之间的交互，来为Java程序提供内存可见性保护。



在执行程序时，为了提高性能，编译器和处理器常常会对指令进行重排序：

* 编译器优化的重排序：编译器在不改变单线程语义的前提下，重排语句的执行顺序
* 指令级并行的重排序：如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序
* 内存系统的重排序：由于处理器使用了缓存和读/写缓冲区，这使得加载和存储操作看上去可能是在乱序执行

JMM的处理器重排序规则会要求Java在编译器生成指令序列时，插入特定的内存屏障指令，来禁止特定类型的处理器重排序，保证一致的内存可以见性。

### 1.1 写缓冲区的优势

写缓冲区可以保证指令流水线持续运行，它可以避免由于处理器停顿下来等待想内存写入数据而产生的延迟。同时通过以批处理的方式刷新写缓冲区，以及合并写缓冲区中对同一内存地址的多次写，减少对内存总线的占用。

### 1.2 happens-before

JSR-133内存模型使用happens-before的概念来阐述操作之间的内存可见性。在JMM中，如果一个操作执行的结果需要对另一个操作可见，那么这两个操作之间必须存在happens-before关系。

happens-before关系如下：

* 程序顺序规则：一个线程中的每个操作，happens-before于该线程中的任意后续操作
* 监视器锁规则：对一个锁的解锁，happens-before于随后对这个锁的加锁
* volatile变量规则：对一个volatile域的写，happens-before于任意后续对这个volatile的读
* 传递性：如果A happens-before B，且B happens-before C，那么A happens-before C

## 2. 重排序

重排序是指编译器和处理器为了优化程序性能而对指令序列进行重新排序的一种手段。

### 2.1 数据依赖性

如果两个操作访问同一个变量，且其中一个操作为写，此时两个操作之间存在数据依赖。只要重排序两个操作的执行顺序，执行结果就会改变。编译器和处理器在重排序时，会遵守数据依赖性。

这里的依赖性仅针对于单个处理器中执行的指令序列和单个线程中执行的操作。

### 2.2 重排序对多线程的影响

|  |
| --- |
| **示例代码** |
| class ReorderExample**{**  int a **=** 0**;**  boolean flag **=** **false;**    public void writer**()** **{**  a **=** 1**;** // 1  flag **=** **true;** // 2  **}**    public void reader**()** **{**  **if** **(**flag**)** **{** // 3  int i **=** a **\*** a**;** // 4  **}**  **}**  **}** |

## 3. 顺序一致性

Java内存模型规范对数据竞争的定义：在一个线程中写一个变量，在另一个线程中读同一个变量，而且写和读没有通过同步来排序。

如果成行是正确同步的，程序的执行将具有顺序一致性--即程序的执行结果与改程序在顺序一致性内存模型中执行结果相同。

未同步程序在JMM中不但整体的执行顺序是无序的，而且所有线程看到的操作执行顺序也可能不一致。只有当一个线程把本地内存中写过的数据刷新到主内存之后，这个写操作才能对其他线程可见。在这种情况下，该线程和其他线程看到的操作执行顺序不一致。

对于未同步或未正确同步的多线程程序，JMM只提供最小安全性：线程执行时读取到的值，要么是之前某个线程写入的值，要么是默认值（0,null, false），不会是无中生有的值。

JMM不保证对64位的long型和double型变量的写操作具有原子性（32位处理器），可以使用volatile来声明，保证其原子性。

## 4. volatile的内存语义

对volatile变量的单个读/写（只是读和写操作，不包含复合操作），可看成是使用同一个锁对这些单个读/写操作做了同步。

volatile变量具有以下特性：

* 可见性：对一个volatile变量的读，总能看到所有线程对这个volatile变量最后的写入
* 原子性：对任意单个volatile变量的读/写具有原子性，但类似于volatile++这种复合操作，不具备原子性

### 4.1 volatile写-读的内存语义

volatile写的内存语义： 当写一个volatile变量时，JMM会把该线程对应的本地内存中的共享变量的值刷新到住内存中

volatile读的内存语义：当读取一个volatile变量时，JMM会把该线程对应的本地内存置为无效，线程接下来将从主内存中读取共享变量。

### 4.2 volatile内存语义的实现

|  |  |  |  |
| --- | --- | --- | --- |
| **是否能重排序** | **第二个操作** | | |
| **第一个操作** | **普通读/写** | **volatile读** | **volatile写** |
| 普通读/写 |  |  | NO |
| volatile读 | NO | NO | NO |
| volatile写 |  | NO | NO |

上表中表示在程序中，当第一个操作为普通变量的读或写时，如果第二个操作是volatile写，则编译器不能重排序这两个操作。

从上表中可以看出

* 当第二个操作是volatile写时，不管第一个操作是什么，都不能重排序，这个规则确保volatile写之前的操作不会被编译器重排序到volatile写之后
* 当第一个操作是volatile读时，不管第二个操作是什么，都不能重排序。这个规则确保volatile读之后的操作不会被编译器重排序到volatile读之前
* 当第一个操作是volatile写，第二个操作是volatile读时，不能重排序

正确使用volatile变量的条件：对变量的写操作不依赖于当前值；该变量没有包含在具有其他变量的不变式中（条件判断等）

编译器不会对volatile读与volatile读后面的任意内存操作重排序；编译器不会对volatile写与volatile写前面的任意内存操作重排序。

## 5. 锁的内存语义

锁是Java并发编程中最重要的同步机制。锁除了让临界区互斥执行外，还可以让释放锁的线程向获取同一个锁的线程发送消息。

当线程释放锁时，JMM会把该线程对应的本地内存中的共享变量刷新到主内存中。

当线程获取锁时，JMM会把该线程对应的本地变量内存置为无效。

### 5.1 锁内存语义的实现

ReentrantLock的实现依赖于Java同步框架AbstractQueuedSynchronizer，该框架使用一个整型的volatile变量来维护同步状态。通过CAS获取锁，CAS同时具有volatile读和volatile写的语义：

* 公平锁和非公平锁释放时，最后都要写一个volatile变量state
* 公平锁获取时，首先会去读volatile变量
* 非公平锁获取时，首先会用CAS更新volatile变量，这个操作同时具有volatile读和写的内存语义

### 5.2 concurrent包的实现

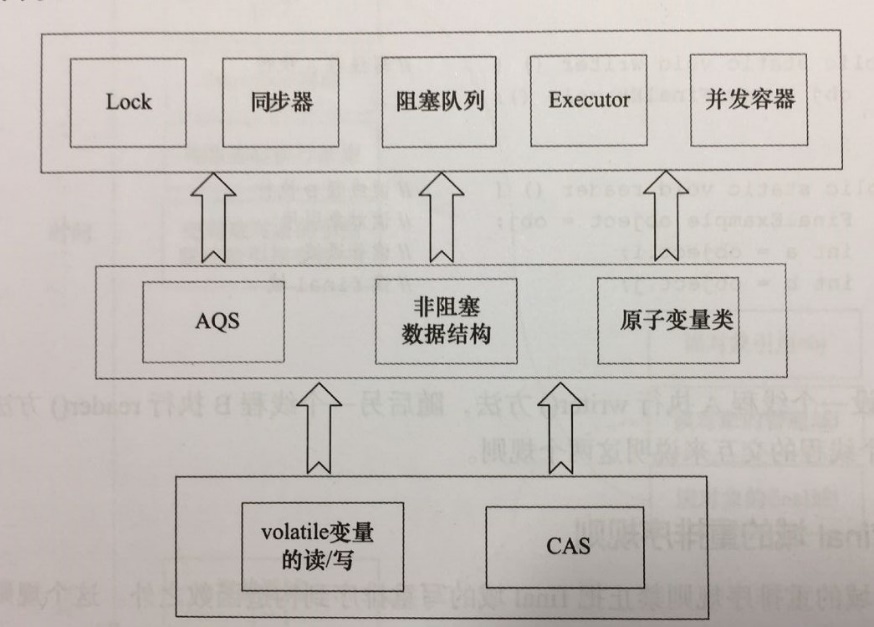
因Java的CAS同时具有volatile读和volatile写的内存语义，因此Java线程之间的通信现在有了以下4种方式：

* A线程写volatile变量，随后B线程读取这个变量
* A线程写volatile变量，随后B线程用CAS更新这个volatile变量
* A线程使用CAS更新一个volatile变量，随后B线程使用CAS更新这个volatile变量
* A线程使用CAS更新一个volatile变量，随后B线程读取这个volatile变量

Java的CAS会使用现代处理器提供的高效机器级别的原子指令，这些原子指令以原子方式对内存执行读-改-写操作。这是concurrent包得以实现的基石：

首先声明共享变量volatile；然后使用CAS的原子条件的更新来实现线程之间的同步；同时配合以volatile的读/写和CAS所具有的volatile读和写的内存语义来实现线程之间的通信。

下图中展示了concurrent包实现的示意图。



## 6. final域的内存语义

对于final域，编译器和处理器要遵守两个重排序规则

1. 构造函数内对一个final域的写入，与随后把这个被构造函数对象的引用赋值给一个引用变量，这两个操作之间不能重排序（JMM禁止编译器把final域的写重排序到构造函数之外）。

2. 初次读一个包含final域的对象的引用，与随后初次读这个final域，这两个操作之间不能重排序

|  |
| --- |
| public class FinalExample **{**  int i**;**  final int j**;**  static FinalExampleobj**;**  public FinalExample **()** **{**  i **=** 1**;**  j **=** 2**;**  **}**  public static void writer **()** **{** // 写线程A执行  obj **=** **new** FinalExample**();**  **}**  public static void reader **()** **{**  FinalExample object **=** obj**;** // 读对象引用  int a **=** object**.**i**;** // 读普通域  int b **=** object**.**j**;** // 读final域  **}**  **}** |

写final域的重排序规则可以确保：在对象引用被任意线程可见之前，对象的final域已经被正确的初始化过了，而普通对象不具有这个保障。（线程B看到对象obj时，很可能obj对象还没有构造完成，i的写入操作还没有完成）

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

对于引用类型，写final域的重排序规则对编译器和处理器增加了如下约束：在构造函数内对一个final引用对象的成员域的写入，与随后在构造函数外把这个被构造函数对象的引用赋值给一个引用变量，这两个操作不能被重排序。

final引用不能从构造函数中“溢出” P60

## 7. happens-before 原则

JMM中定义了如下happens-before规则：

* 程序顺序规则：一个线程中的每个操作，happens-before于该线程的任意后续操作
* 监视器锁规则：对一个锁的解锁，happens-before于随后对这个锁的加锁
* volatile变量规则：对某个volatile变量的写，happens-before于任意后续对这个变量的读。
* 传递性
* start()规则：如果线程A执行操作ThreadB.start()，那么A线程的ThreadB.start()操作happens-before于线程B中的任意操作
* join()规则：如果线程A执行操作ThreadB.join()并成功返回，那么线程B中的任意操作happens-before于线程A从ThreadB.join()成功返回

## 8.双重检查锁定与延迟初始化

|  |
| --- |
| public class DoubleCheckedLocking **{**  private static Instance instance**;**  public static Instance getInstance **()** **{**  **if** **(**instance **==** **null)** **{** // 第一次检查  synchronized **(**DoubleCheckedLocking**.**class**)** **{** // 加锁  **if** **(**instance **==** **null)** **{** // 第二次检查  instance **=** **new** Instance**();** // 问题根源在此  **}**  **}**  **}**  **return** instance**;**  **}**  **}** |

上面代码的问题根源(instance=new Instance())创建了一个对象，这一行代码可以分解成3行伪代码：

memory = allocate(); // 1：分配对象的内存空间

ctorInstance(memory); // 2: 初始化对象

instance = memory; // 3: 设置instance指向新分配的空间

上面代码中2和3可能会被重排序，线程B可能会访问到一个未初始化的对象。

解决方案：

* 使用volatile来修饰Instance，volatile禁止了2和3的重排序
* 基于类的初始化解决方案：JVM在类的初始化阶段，JVM会获取一个锁，这个所可以同步多个线程对同一个类的初始化

volatile除了可以对静态字段实现延迟实例化外，还可以对实例字段实现延迟初始化，基于类的实例化只能实例化静态字段。字段延迟初始化降低了初始化类或创建实例的开销，但增加了访问被延迟初始化字段的开销。测试后发现使用volatile耗时少（不知道测试正确与否）

|  |
| --- |
| **基于类初始化的解决方案** |
| public class InstanceFactory **{**  private static class InstanceHolder **{**  public static Instance instance **=** **new** Instance**();**  **}**  public static Instance getInstance**()** **{**  **return** InstanceHolder**.**instance**;**  **}**  **}** |
| **基于volatile的解决方案** |
| public class DoubleCheckedLocking **{**  private static volatile Instance instance**;**  public static Instance getInstance **()** **{**  **if** **(**instance **==** **null)** **{** // 第一次检查  synchronized **(**DoubleCheckedLocking**.**class**)** **{** // 加锁  **if** **(**instance **==** **null)** **{** // 第二次检查  instance **=** **new** Instance**();** // 问题根源在此  **}**  **}**  **}**  **return** instance**;**  **}**  **}** |

# 第四章 Java并发编程基础

线程：又称为轻量级进程，在一个进程里可以创建多个线程，这些线程都拥有各自的计数器、堆栈和局部变量等属性，并能够访问共享的内存变量。

## 1. 线程简介

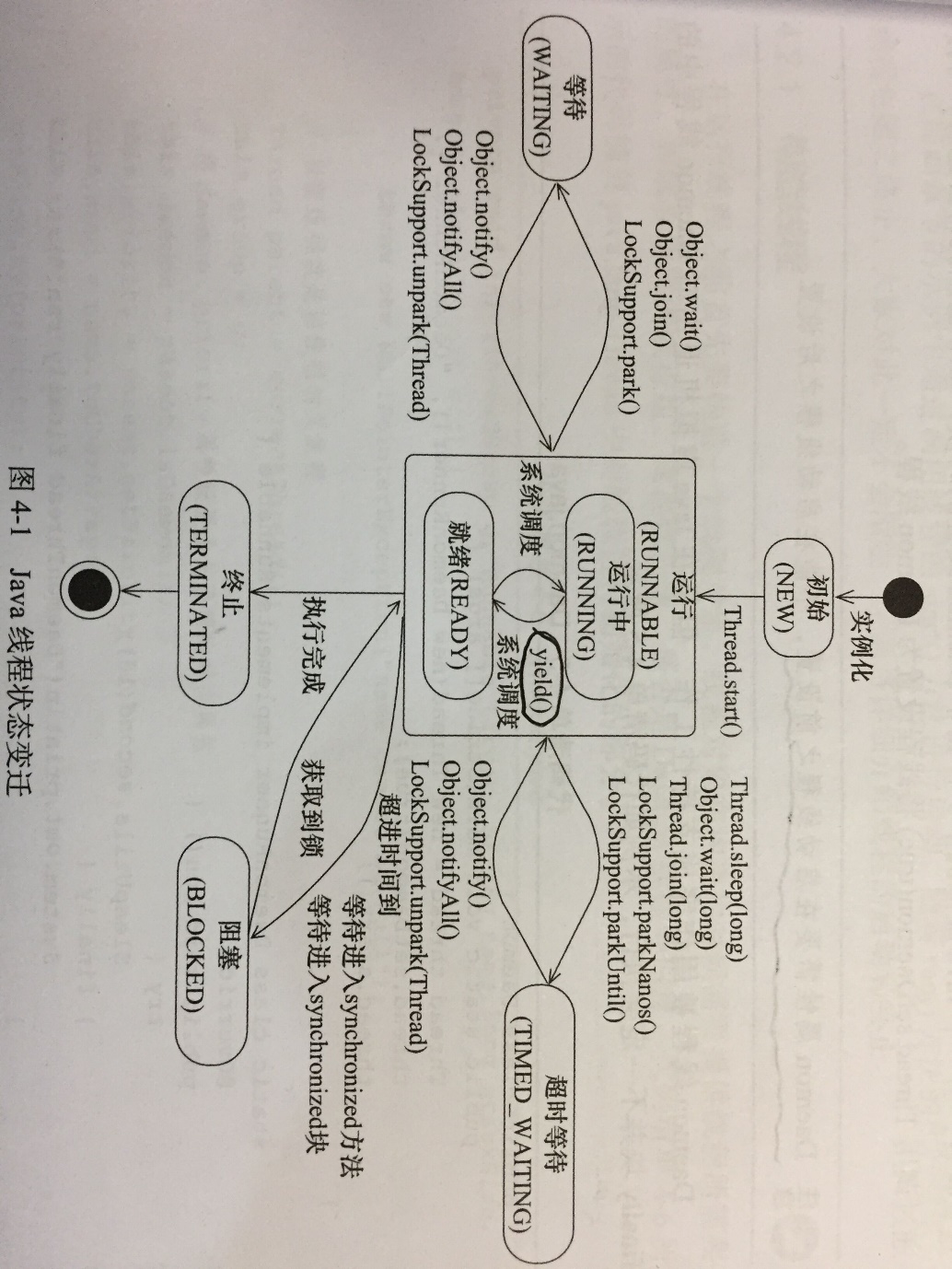
多线程的优势：一个线程在一个时刻只能运行在一个处理器核心上，多个线程运行在多个核心上，处理速度更快；对复杂的业务逻辑可以将数据一致性不强的操作派发给其他线程处理（消息队列）。

### 1.1 线程优先级

设置线程优先级时，针对频繁阻塞（休眠或者I/O操作）的线程需要设置较高优先级，而偏重计算的线程则设置较低的优先级，确保处理器不会被独占。

**线程优先级不能作为程序正确性的依赖**，因为操作系统可以完全不用理会Java线程对优先级的设定。

### 1.2 线程的状态



由上图可以看出，线程创建之后，调用start()方法开始运行。当线程执行wait()方法之后，线程进入等待状态。进入等待状态的线程需要依靠其他线程的通知才能够返回到运行状态，而超市等待状态相当于在等待状态的基础上增加了超时限制，也就是超时时间到达时将会返回运行状态。在没有获取到锁的情况下，线程会进入到阻塞状态。线程执行Runnable的run()方法之后会进入到终止状态。

|  |  |
| --- | --- |
| 状态名称 | 说明 |
| NEW | 初始状态，线程被构建，但是还没有调用start() |
| RUNNABLE | 运行状态，Java线程将操作系统中的就绪和运行两种成为“运行中” |
| BLOCKED | 阻塞状态，表示线程阻塞于锁。指线程阻塞在进入synchronized |
| WAITING | 等待状态，进入该状态表示当前线程需要等待其他线程做出一些特定的动作（通知或中断），阻塞在Lock接口的线程（concurrent包） |
| TIME\_WAITING | 超时等待，它可以在指定的时间自动返回 |
| TERMINATED | 终止状态 |

### 1.3 Daemon线程

Daemon线程是一种支持型线程，因为它被用作程序中后台调度以及支持工作。当一个Java虚拟机不存在非Daemon线程时，Java虚拟机将会退出。

Daemon属性需要在启动线程之前设置：thread.setDaemon(true)

构建Daemon线程时，不能依靠finally块中的内容来确保执行关闭或清理资源，因为Daemon线程的finally块并不一定会被执行。

## 2. 启动和终止线程

启动一个线程前，最好为这个线程设置线程名称，因为这样在使用jstack分析程序或者排查问题时，会更容易些。

中断可以理解为一个线程的标记位属性，它表示一个运行中的线程是否被其他线程中断。线程通过isInterrupted()来进行判断是否被中断，也可以调用静态方法Thread.interrupted()对当前线程的中断标识位进行复位

|  |
| --- |
| **安全地终止线程** |
| public class ShutDown**{**  public static void main**(**String **[]** args**)** **throws** Exception **{**  Runner one **=** **new** Runner**();**  Thread countThread **=** **new** Thread**(**one**,** "CountThread"**);**  countThread**.**start**();**  // sleep 1s, main线程对CountThread进行中断，使CountThread能够感知中断而结束  TimeUtil**.**SECONDS**.**sleep**(**1**);**  countThread**.**interrupt**();**  Runner two **=** **new** Runner**();**  countThread **=** **new** Thread**(**two**,** "countThread"**);**  countThread**.**start**();**  // sleep 1s, main线程对CountThread进行中断，使CountThread能够感知on而结束  TimeUtil**.**SECONDS**.**sleep**(**1**);**  two**.**cancel**();**  **}**    private static class Runner **implements** Runnable **{**  private long i**;**  private volatile boolean on **=** **true;**  public void run**()** **{**  **while** **(**on **&&** **!**Thread**.**currentThread**().**isInterrupted**())** **{**  i**++;**  **}**  System**.**out**.**println**(**"Count i = " **+** i**);**  **}**  public void cancel**()** **{**  on **=** **false;**  **}**  **}**  **}** |

## 3. 线程间通信

### 3.1 volatile和synchronized关键字

关键字volatile可以用来修改字段，就是告知程序任何对该变量的访问均需要从共享内存中获取，而对它的改变也必须同步刷新到共享内存中，它能保证所有线程对变量访问的可见性。

关键字synchronized可以修饰方法或以同步块的形式来使用，它主要确保多个线程在同一时刻，只能有一个线程处于方法或者同步块中，它保证了线程对变量访问的可见性和排他性。synchronized使用监视器来实现同步

### 3.2 等待/通知机制

等待/通知机制是任意Java对象都具备的

|  |  |
| --- | --- |
| **方法名称** | **描述** |
| 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了对象的锁 |
| notifyAll() | 通知所有等待在该对象上的线程，竞争获取锁 |
| wait() | 线程进入到WAITING状态，只有等待另外线程的通知或被中断才会返回，调用wait()方法后，会释放对象的锁 |
| wait(long) | 等待n毫秒，如果没有通知就超时返回 |
| wait(long,int) | 对超时更细力度的控制，可以达到纳秒 |

使用wait()、notify()和notifyAll()的注意细节：

* 使用wait()、notify()和notifyAll()时需要先对调用对象加锁
* 调用wait()后，线程状态由RUNNING变为WAITING，并将当前线程放置到对象的等待队列中
* notify()或notifyAll()调用后，等待线程依旧不会从wait()返回，需要调用notify()或notifyAll()的线程释放解锁之后，等待线程才有机会从wait()返回
* notify()方法将一个等待线程从其等待队列中移动到同步队列中，而notifyAll()则将等待队列中的所有线程全部同步到同步队列，被移动的线程状态由WAITING变为BLOCKED。
* 从wait()返回的前提是获得了对象的锁。

|  |
| --- |
| **消费者(等待方)** |
| synchronized **(**object**)** **{** // 获取对象的锁  **while** **(**condition**)** **{** // 如果条件不满足，调用wait方法，被通知后仍要检查条件  object**.**wait**();**  **}**  // handle // 条件满足执行对应的逻辑  **}** |
| **生产者(通知方)** |
| synchronized **(**object**)** **{** // 获取对象的锁  change condition // 改变条件  object**.**notifyAll**();** // 通知所有等待在对象上的线程  **}** |

### 3.3 管道输入/输出流

管道输入/输出流主要用于线程之间的数据传输，而传输媒介为内存。包括了4种实现：PipedOutputStream、PipedInputStream、PipedReader和PipedWriter,前两种面向字节，后两种面向字符。

public class Piped {  
 static class Print implements Runnable {  
 private PipedReader in;  
 public Print(PipedReader in) {  
 this.in = in;  
 }  
 public void run() {  
 int receive = 0;  
 try {  
 while ((receive = in.read()) != -1) {  
 System.*out*.print((char) receive);  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 public static void main(String [] args) throws Exception {  
 PipedWriter out = new PipedWriter();  
 PipedReader in = new PipedReader();  
 // 将输入流和输出流进行连接，否则在使用时会抛出IOException  
 out.connect(in);  
 Thread printThread = new Thread(new Print(in), "Print Thread");  
 printThread.start();  
 int receive = 0;  
 while ((receive = System.*in*.read()) != -1) {  
 out.write(receive);  
 }  
 out.close();  
 }  
}

### 3.4 Thread.join()的使用

如果一个线程A执行了thread.join()语句，其含义是：当前线程A等待thread线程终止之后才能从thread.join()返回

当程序终止时，join方法中，在调用线程自身的notifyAll()方法，会通知所有等待在该线程对象上的线程。

### 3.5 ThreadLocal的使用

ThreadLocal，即线程变量，是一个以ThreadLocal对象为键、任意对象为值的存储结构，一个线程可以根据一个ThreadLocal对象查询到绑定到这个线程上的一个值。

通过set(T)来设置一个值，在当前线程下通过get()方法获取到原先设置的值。

# 第五章 Java中的锁

锁是用来控制多个线程访问共享资源的方式，一个锁能够防止多个线程同时访问共享资源（有些锁允许多个线程并发访问资源）。

在finally块中释放锁，目的是保证在获取锁之后，最终能释放。不要将获取锁的过程写在try块中，因为如果在获取锁时发生了异常，异常抛出的同时，也会导致锁无故释放。

|  |  |
| --- | --- |
| **特性** | **描述** |
| 尝试非阻塞地获取锁 | 当前线程尝试获取锁，如果这一时刻锁没有被其他线程获取到，则成功获取并持有锁 |
| 能被中断地获取锁 | 与synchronized不同，获取到锁的线程能够响应中断，当获取到锁的线程被中断时，中断异常被抛出，同时锁会被释放 |
| 超时获取锁 | 在指定的截止时间之前获取锁，如果无法获取锁，则返回 |

Lock的API

|  |  |
| --- | --- |
| 方法名称 | 描述 |
| void lock() | 获取锁，调用该方法当前线程会获取锁，获取锁后，返回 |
| void lockInterruptibly() throws InterruptedException | 可中断地获取锁，和lock()方法的不同之处在于该方法会响应中断，即在锁的获取中可以中断当前线程 |
| boolean tryLock() | 尝试非阻塞获取锁，调用该方法后立即返回，如果能够获取则返回true，否则返回false |
| boolean tryLock(long time, TimeUnit unit) throws InterruptedException | 超时获取锁，当前线程在以下3中情况下会返回：   * 当线程在超时时间内获得了锁 * 当前线程在超时时间内被中断 * 超时时间结束，返回false |
| void unlock() | 释放锁 |
| Condition newCondition() | 获取等待通知组件，该组件和当前锁绑定，当前线程只有获得了锁，才能调用该组件的wait()方法。调用后，当前线程将锁释放 |

## 1. 队列同步器

队列同步器AbstractQueueSynchronizer是用来构建锁或者其他同步组件的基础框架，使用一个int成员变量表示同步状态，通过内置的FIFO队列来完成资源获取线程的排队工作。

同步器的主要使用方式是继承，子类通过继承同步器并实现它的抽象方法来管理同步状态。同步器提供了3个方法：getState():获取当前同步状态；setState(int state):设置当前同步状态；compareAndSetState(int expect, int update):使用CAS设置当前状态，该方法能够保证状态设置的原子性。

锁是面向使用者的，它定义了使用者与锁交互的接口，隐藏了实现细节；同步器是面向锁的实现者，它简化了锁的实现方式，屏蔽了同步状态管理、线程排队、等待与唤醒等底层操作。

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

## 2. 队列同步器的实现

### 2.1. 同步队列

同步器依赖内部的同步队列（FIFO双向队列）来完成同步状态的管理，当前线程获取同步状态失败时，同步器会将当前线程以及等待状态等信息构造成为一个节点并将其加入到同步队列中，同时阻塞当前线程。当同步状态释放时，会把首节点中的线程唤醒，使其再次尝试获取同步状态。

|  |  |
| --- | --- |
| 属性类型与名称 | 描述 |
| int waitStatus | 等待状态，包含如下状态   1. CANCELLED，值为1，表示同步队列中等待的线程等待超时或者被中断，需要取消等待，节点进入该状态后将不会变化 2. SIGNAL，值为-1，后继节点的线程处于等待状态，而当前节点的线程如果释放了同步状态或者被取消，将会通知后继节点，使后继节点得以运行 3. CONDITION，值为-2，节点在等待队列中，等待在Condition上，当其他线程对Condition调用了signal()后，该节点从等待队列转移到同步队列中并加入到对同步状态的获取中 4. PROPAGATE，值为-3，表示下一次共享式同步状态获取将会无条件地传播下去 5. INITIAL，值为0，初始状态 |
| Node prev | 前驱节点，当节点加入到同步队列时被设置（队尾添加） |
| Node next | 后继节点 |
| Node nextWaiter | 等待队列中的后继节点。如果当前节点时共享的，那么这个字段将是一个SHAREED常亮 |
| Thread thread | 获取同步状态的线程 |

总结：在获取同步状态时，同步器维护一个同步队列，获取状态失败的线程都会被加入到队列中并在队列中自旋；移除队列的条件是前驱节点为头结点且成功获取了同步状态。在释放同步状态时，同步器调用tryRelease(int arg)方法释放同步状态，然后唤醒头节点的后继节点。

### 2.2 共享式同步状态获取与释放

共享式获取与独占式获取最主要的区别在于同一时刻能否有多个线程同时获取到同步状态。

共享式访问资源时，其他共享方式的访问均被允许，而独占式访问被阻塞；独占式资源被访问时，同一时刻，其他访问均被阻塞。

### 2.3 重入锁

重进入是指任意线程在获取到锁之后能够再次获取该锁而不会被锁阻塞，需要解决两个问题：

**线程再次获取锁**：锁需要去识别获取锁的线程是否为当前占有锁的线程

**锁的最终释放**：线程重复n次获取了锁，也要释放n次锁。

synchronized隐式的支持锁的重进入。

公平锁是指锁的获取顺序符合请求的绝对时间顺序（FIFO）

非公平锁被设定为默认实现：公平锁为了保证锁的获取是按照FIFO原则，而代价是进行大量的线程切换，开销比较大；非公平锁中，释放锁的线程再次竞争获取锁的几率非常大，线程切换较少，开销比公平锁低。

### 2.4 读写锁

读写锁在同一时刻可以允许多个线程访问，但是在写线程访问时，所有的读线程和其他写线程都被阻塞。读写锁维护了一对锁，一个读锁和一个写锁，通过分离读锁和写锁，使得并发性比一般的排它锁有了很大的提升

在读多于写的情况下，读写锁能够提供比排它锁更好的并发量和吞吐量。

锁降级是指当前持有写锁，再次获取到读锁，随后释放写锁的过程。锁降级主要是为了保证数据的可见性，如果当前线程不获取读锁而直接释放写锁，假设此刻另一个线程（T）获取了写锁并修改了数据，那么当前线程无法感知线程T的数据更新。两个线程同时获取读锁，一个线程获取写锁，修改完成后，再获取读锁

## 3. LockSupport工具

LockSupport定义了一组的公共静态方法，这些方法提供了最基本的线程阻塞和唤醒功能。

|  |  |
| --- | --- |
| **方法名称** | **描述** |
| void park() | 阻塞当前线程，如果调用unpark或者当前线程中断，才能从park返回 |
| void parkNanos(long nanos) | 阻塞当前线程，最长不超过nanos纳秒，在park基础上增加了超时返回 |
| void parkUtil(long deadline) | 阻塞当前线程，直到deadline |
| void unpart(Thread thread) | 唤醒处于阻塞状态的线程thread |

## 4. Condition接口

任意一个Java对象，都拥有一组监视器方法，主要包括wait()、wait(long timeout)、notify()以及notifyAll()方法，这些方法与synchronized配合，可以实现等待\通知模式。Condition接口也提供了类似的功能特性

|  |  |  |
| --- | --- | --- |
| **对比项** | **Object Monitor** | **Condition** |
| 前置条件 | 获取对象的锁 | 调用Lock.lock()获取锁  调用Lock.newCondition()获取Condition对象 |
| 调用方式 | 直接调用ob.wait() | 直接调用 condition.wait() |
| 等待队列个数 | 一个 | 多个 |
| 线程释放锁并进入等待状态 | 支持 | 支持 |
| 当前线程释放锁并进入等待状态，在等待状态中不响应中断 | 不支持 | 支持 |
| 线程释放锁并进入超时等待 | 支持 | 支持 |
| 当前线程释放锁并进入等待状态到将来的某个时间 | 不支持 | 支持 |
| 唤醒等待队列中的一个线程 | 支持 | 支持 |
| 唤醒等待队列中的全部线程 | 支持 | 支持 |

Condition接口示例

public class ConditionUseCase {  
 Lock lock = new ReentrantLock();  
 Condition condition = **lock.newCondition()**;  
  
 public void conditionWait() throws InterruptedException {  
 lock.lock();  
 try{  
 condition.**await()**;  
 } finally {  
 lock.unlock();  
 }  
 }  
  
 public void conditionSignal() {  
 lock.lock();  
 try {  
 condition.signal();  
 } finally {  
 lock.unlock();  
 }  
 }  
}

获取一个Condition必须通过Lock的newCondition方法。

# 第六章 Java并发容器和框架

## 1. ConcurrentHashMap的实现原理与使用

HashMap是线程不安全的，在多线程环境下，使用HashMap进行put操作会引起死循环。因为多线程会导致HashMap的Entry链表形成环形数据结构，Entry的next节点永远都不为空。

HashTable效率低下，一个线程操作HashTable，会对整个HashTable变量加锁，阻塞其他线程访问该变量。

ConcurrentHashMap使用了锁分段技术，首先将数据分为一段一段地存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一段数据时，其他段的数据也能被其他线程访问。

ConcurrentHashMap在插入元素之前，首先判断是否需要扩容，这样比插入之后判断的好处是，如果扩容之后没有元素插入，那就会进行了一次无效的扩容。

## 2. ConcurrentLinkedQueue

如果要实现一个线程安全的队列有两种方式：一种是使用阻塞算法，另一种是使用非阻塞算法。使用阻塞算法的队列可以用一个锁（入队和出队用同一把锁）或两个锁来实现。非阻塞队列可以使用循环CAS的方式来实现。

ConcurrentLinkedQueue是一个基于链接点的误解无界线程安全队列，它采用FIFO的方式，使用了CAS算法来实现。

## 3. 阻塞队列

阻塞队列是一个支持阻塞的插入方法和阻塞的移除方法的队列。阻塞插入是当队列满时，队列会阻塞插入元素的线程，知道队列不满；阻塞移除时当队列为空时，获取元素的线程会等待队列变为非空。

无界阻塞队列，队列不会出现满的情况，所示使用put或offer方法永远不会被阻塞，offer方法会一直返回true。

## 4. Fork/Join框架

Fork/Join框架是Java 7提供的一个用于并行执行任务的框架，是一个把大任务分割成若干个小任务，最终汇总每个小任务结果后得到大任务结果的框架。

工作窃取算法是指某个线程从其他队列窃取任务来执行。优点是充分利用线程进行并行计算，减少线程间的竞争。缺点是在某些情况下还存在竞争，比如双端队列里只有一个任务时。而且该算法会消耗更多的系统资源，比如创建多个线程和多个双端队列。

# 第七章 Java的13个原子操作类

在Atomic包中一共提供了13个类，属于4中类型的原子更新方式，分别是原子更新基本类型、原子更新数组、原子更新引用和原子更新属性。

## 1. 原子更新基本类型

AtomicBoolean, AtomicInteger, AtomicLong. 以AtomicInteger，其常用方法：

* int addAndGet(int delta): 以原子方式将输入的值与实例中的值相加，并返回结果
* boolean compareAndSet(int expect, int update): CAS的方式更新
* int getAndIncrement(): 以原子方式将当前值加1，返回自增前的值
* int getAndSet(int newValue): 以原子方式设置为newValue的值，并返回旧值。

原子更新char、float和double变量时可以先转换成整型来实现

## 2. 原子更新数组

Atomic提供了4个类：

* AtomicIntegerArray:原子更新数组里的元素
* AtomicLongArray：原子更新长整型数组里的元素
* AtomicReferenceArray： 原子更新引用类型数组里的元素
* AtomicIntegerArray：原子更新数组里的整型：
  + int addAndGet(int i, int delta)
  + boolean compareAndSet(int position,int expect,int update)

数组通过构造方法传递进去，然后原子更新数组类会将当前数组复制一份，所以原子更新数组对元素修改时，不会影响传入的数组。

## 3. 原子更新引用类型

如果要原子更新多个变量，就要使用原子更新引用类型：

* AtomicReference：原子更新引用类型
* AtomicReferenceFieldUpdater:原子更新引用类型里的字段
* AtomicMarkableReference: 原子更新带有标记位的引用类型

public static AtomicReference<Person> *atomicPersonRef* = new AtomicReference<>();  
public static void main(String [] args) {  
 Person p = new Person("wang", 20);  
 *atomicPersonRef*.set(p);  
 Person updateUeser = new Person("pan",20);  
 *atomicPersonRef*.compareAndSet(p, updateUeser);  
}

## 4. 原子更新字段类

如果需要原子地更新某个类里的某个字段，就需要使用原子更新字段类，Atomic包提供了3个类：

* AtomicIntegerFieldUpdater: 原子更新整型的字段的更新器
* AtomicLongFieldUpdater: 原子更新长整型字段的更新器
* AtomicStampedReference: 原子更新带有版本号的引用类型，解决使用CAS进行更新时可能出现的ABA问题

使用原子更新字段的类时：用静态方法newUpdater()创建一个更新器，设置想要更新的类和属性；更新类的字段必须使用public volatile修饰

private static AtomicIntegerFieldUpdater<User> *updater* =  
 AtomicIntegerFieldUpdater.*newUpdater*(User.class, "old");  
public static void main(String [] args) {  
 User conan = new User("conan", 10);  
 System.*out*.println(*updater*.getAndIncrement(conan));  
 System.*out*.println(*updater*.get(conan));  
}  
  
public static class User {  
 private String name;  
 public volatile int old;

}

# 第八章 Java中的并发工具类

JDK的并发包里提供了几个非常有用的工具类：CountDownLatch、CyclicBarrier和Semaphore工具类。Exchanger工具类则提供了在线程间交换数据的一种手段。

## 1. 等待多线程完成的CountDownLatch

CountDownLatch允许一个或多个线程等待其他线程完成操作。

CountDownLatch的构造函数接收一个int类型的参数作为计数器，如果想等待N个点完成，就出入N。

调用CountDownLatch的countDown方法时，N就会减1，CountDownLatch的await方法会阻塞当前线程，直到N变为零。由于countDown方法可以用在任何地方，所以这里的N，可以是N个线程，也可以是1个线程的N个执行步骤。

## 2. 同步屏障CyclicBarrier

让一组线程达到一个屏障（同步点）时被阻塞，知道最后一个线程达到屏障时，屏障才会打开，所有被屏障拦截的线程才会继续运行。

其默认的构造方法是CyclicBarrier(int parties), 其参数表示屏障拦截的线程数量，每个线程调用await方法告诉CyclicBarrier我已到达屏障，然后该线程被阻塞。

CyclicBarrier的计数器可以使用reset()重置。

## 3. 控制并发线程数的Semaphore

Semaphore用来控制同时访问特定资源的线程数量，通过协调各个线程，以保证合理的使用公共资源。

## 4. 线程间交换数据的Exchanger

Exchanger用于线程间协作的工具类，它可以在线程间进行数据交换。它提供一个同步点，在这个同步点，两个线程可以交换彼此的数据，这两个线程通过exchange方法交换数据，如果第一个线程先执行exchange()方法，它会一直等待第二个线程也执行exchange()方法，当两个线程都达到同步点时，两个线程就可以交换数据。

# 第九章 Java中的线程池

线程池的好处：

* 降低资源消耗：通过重复利用已创建的线程降低线程创建和销毁造成的消耗
* 提高响应速度：当任务到达时，任务可以不需要等线程创建就能执行
* 提高线程的可管理性。线程是稀缺资源，如果无限制地创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一分配、调优和监控。

## 1. 线程池的实现原理

线程池的处理流程如下：

1. 线程池判断核心线程池里的线程是否都在执行任务。如果不是，则创建一个新的工作线程来执行任务；否则进入下一个流程
2. 线程池判断工作队列是否已满，如果工作队列没有满，则将新提交的任务存储在这个工作队列。如果工作队列满了，则进入下个流程。
3. 线程池判断线程池的线程是否都处于工作状态，如果没有，则创建一个新的工作线程来执行任务。否则交给饱和线程来处理这个任务。

execute方法的执行分为4中情况：

1. 如果当前运行的线程小于corePoolSize，则创建新的线程来执行任务（该步骤需要获取全局锁）
2. 如果运行的线程等于或多于corePoolSize，则将任务加入BlockingQueue
3. 如果队列已满，则创建新的线程来处理任务（该步骤需要获取全局锁）
4. 如果创建新线程将使当前运行的线程数超出了maximumPoolSize，任务将被拒绝。

## 2. 线程池的使用

execute() 用于提交不需要返回值的任务；

submit()方法用于提交需要返回值的任务。线程池会返回一个Future类型的对象，通过该对象可以判断任务是否执行完成，并且可以通过get()方法获取返回值，get()方法会阻塞当前线程线程直到任务完成。

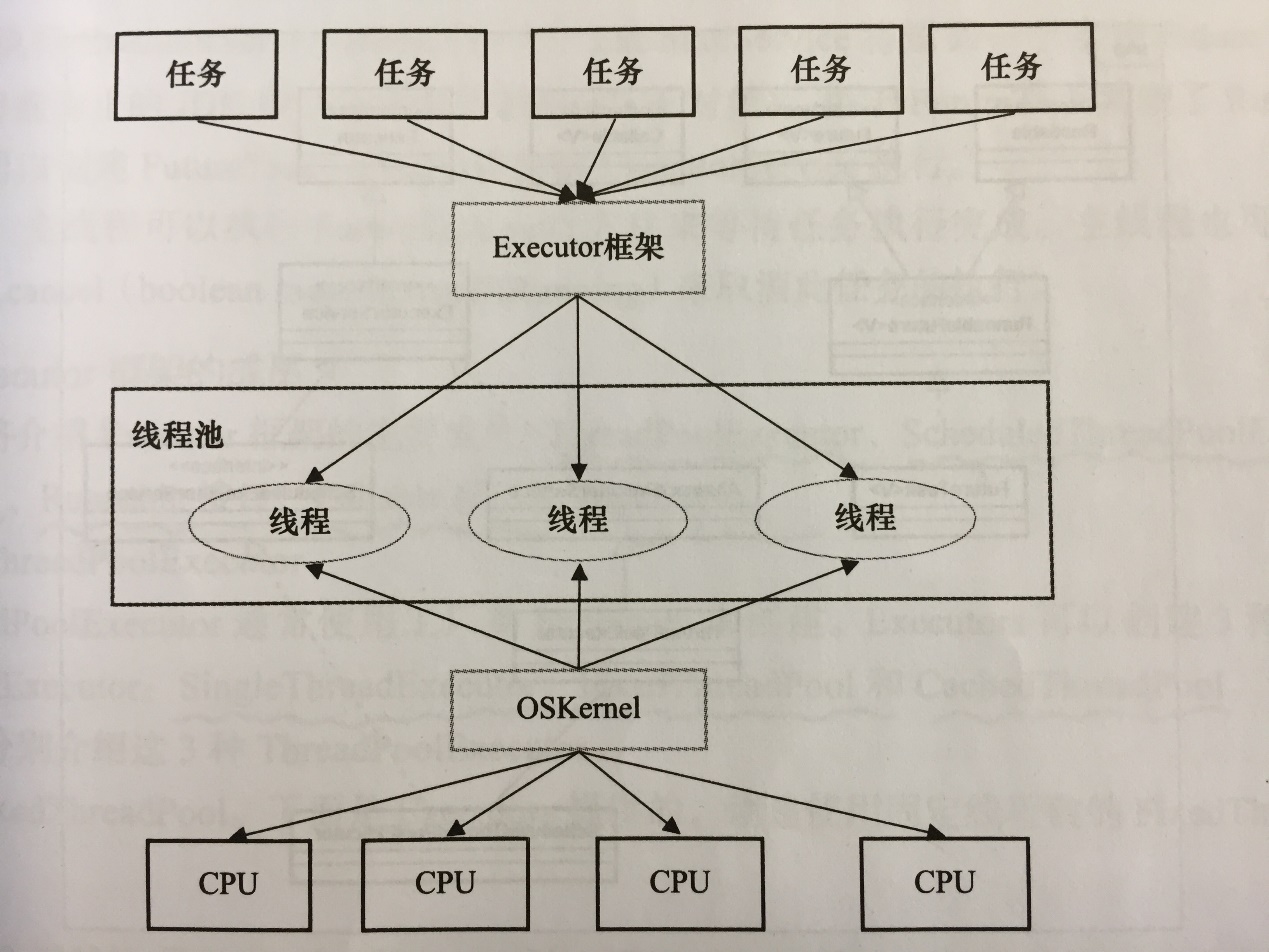
shutdown()和shutdownNow()可以关闭线程池，通过遍历线程池中的工作线程，然后逐个调用线程的interrupt方法来中断线程，所以无法响应中断的任务无法终止。shutdownNow首先将线程池的状态设置为STOP，然后尝试停止所有的正在执行或暂停的线程，并返回等待执行任务的列表。shutdown只是将线程池的状态设置为SHUTDOWN，然后中断所有没有正在执行任务的线程。

在系统中大量使用线程池，则有必要对线程池进行监控，方便在出现问题时，可以根据线程池的使用状况快速定位问题。

* taskCount: 线程池需要执行的任务数量
* completedTaskCount: 线程池在运行过程中已完成的任务数量，小于或等于taskCount
* largestPoolSize: 线程池里曾经创建过的最大线程数量。通过这个数据可以知道线程池是否曾经满过。
* getPoolSize: 线程池的线程数量。
* getActiveCount: 获取活动的线程数。

通过扩展线程池进行监控：重写线程池的beforeExecute、afterExecute和terminated方法，可以在任务执行前、执行后和线程池关闭前来执行一些代码进行监控。

# 第十章 Executor框架



上图是Executor框架的两级调度模型：应用程序通过Executor框架控制上层的调度；而下层的调度由操作系统内核控制，下层的调度不受应用程序的控制。

## 1. Executor框架

Executor框架主要有3部分构成：

* 任务：包括任务所需要的接口：Runnable或Callable
* 任务的执行:核心接口Executor，以及继承自Executor的ExecutorService接口。Executor有两个关键实现类：ThreadPoolExecutor和ScheduledThreadPoolExecutor（延迟后执行或定期执行任务）。
* 异步计算结果：包括接口Future和实现Future接口的FutureTask类。

Executors可以创建3种类型的THreadPoolExecutor：SingleThreadExecutors、FixedThreadPool和CachedThreadPool

* FixedThreadPool用于创建固定线程数的FixedThreadPool，适用于为了满足资源管理需求，而需要限制当前线程数量，适用于负载比较重的服务器。该线程池使用无界队列LinkedBlockingQueue作为线程池的工作队列。
* SingleThreadExecutor创建单个线程，适用于需要保证顺序地执行各个任务；并且在任意时间点，不会有多个线程活动的应用场景。该线程池使用无界队列LinkedBlockingQueue作为线程池的工作队列。
* CachedThreadPool根据需要创建新线程，该线程池是大小无界的线程池，适用于执行很多短期异步任务的小程序或负载较轻的服务器。该线程池使用没有容量的SynchronousQueue作为线程池的工作队列。如果主线程提交任务的速度高于线程处理任务的速度，那么CachedThreadPool将会不断创建新线程直到耗尽CPU和内存资源。

Executors可以创建两种类型的ScheduledThreadPoolExecutor：

* ScheduledThreadPoolExecutor：包含若干个线程
* SingleThreadScheduledExecutor：只包含一个线程

调用scheduleAtFixedRate()方法或scheduleWithFixedDelay()方法时，会向ScheduledThreadPoolExecutor的DelayQueue添加一个实现了RunnableScheduledFuture接口的ScheduledFutureTask。

# 附录A 技巧

# 附录B 代码

## 1. 自制连接池

public class DefaultThreadPool<Job extends Runnable> implements ThreadPool<Job> {  
 // 线程最大限制数  
 private static final int *MAX\_WORKDER\_NUMBER* = 10;  
 // 线程池默认的数量  
 private static final int *DEFAULT\_WORKER\_NUMBERS* = 5;  
 // 线程池最小数量  
 private static final int *MIN\_WORDER\_NUMBERS* = 1;  
 // 工作列表  
 private final LinkedList<Job> jobs = new LinkedList<Job>();  
 // 工作者列表  
 private final List<Worker> workers = Collections.*synchronizedList*(new ArrayList<Worker>());  
 // 工作者线程的数量  
 private int workerNum = *DEFAULT\_WORKER\_NUMBERS*;  
 // 线程编号  
 private AtomicLong threadNum = new AtomicLong();  
  
 public DefaultThreadPool() {  
 initializeWorkers(*DEFAULT\_WORKER\_NUMBERS*);  
 }  
  
 public DefaultThreadPool(int num) {  
 workerNum = num > *MAX\_WORKDER\_NUMBER* ? *MAX\_WORKDER\_NUMBER* :  
 num < *MIN\_WORDER\_NUMBERS* ? *MIN\_WORDER\_NUMBERS* : num;  
 initializeWorkers(workerNum);  
 }  
  
 @Override  
 public void execute(Job job) {  
 if (job != null) {  
 synchronized (jobs) {  
 jobs.addLast(job);  
 jobs.notify();  
 }  
 }  
 }  
  
 @Override  
 public void shutdown() {  
 for (Worker worker: workers) {  
 worker.shutdown();  
 }  
 }  
  
 @Override  
 public void addWorkers(int num) {  
 synchronized (jobs) {  
 if (num + this.workerNum > *MAX\_WORKDER\_NUMBER*) {  
 num = *MAX\_WORKDER\_NUMBER* - this.workerNum;  
 }  
 initializeWorkers(num);  
 this.workerNum += num;  
 }  
 }  
  
 @Override  
 public void removeWorkers(int num) {  
 synchronized (jobs) {  
 if (num >= this.workerNum) {  
 throw new IllegalArgumentException("beyond workNum");  
 }  
 int count = 0;  
 while (count < num) {  
 Worker worker = workers.get(count);  
 if (workers.remove(worker)) {  
 worker.shutdown();  
 count ++;  
 }  
 }  
 this.workerNum -= count;  
 }  
 }  
  
 @Override  
 public int getJobSize() {  
 return jobs.size();  
 }  
  
 private void initializeWorkers (int num) {  
 for (int i=0; i<num; i++) {  
 Worker worker = new Worker();  
 workers.add(worker);  
 Thread thread = new Thread(worker, "ThreadPool-Worker-" + threadNum.incrementAndGet());  
 thread.start();  
 }  
 }  
  
 class Worker implements Runnable {  
 // 是否工作  
 private volatile boolean running = true;  
 public void run() {  
 while (running){  
 Job job = null;  
 synchronized (jobs) {  
 while (jobs.isEmpty()) {  
 try {  
 jobs.wait();  
 } catch (InterruptedException e) {  
 Thread.*currentThread*().interrupt();  
 return;  
 }  
 }  
 // 取出一个Job  
 job = jobs.removeFirst();  
 }  
 if (job != null) {  
 try {  
 job.run();  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
  
 public void shutdown() {  
 running = false;  
 }  
 }  
}

## 2.模拟HttpServer

public class SimpleHttpServer {  
 // 处理HttpRequest的线程池  
 static ThreadPool<HttpRequestHandler> *threadPool* = new DefaultThreadPool<>(1);  
 // SimpleHttpServer的根路径  
 static String *basePath*;  
 static ServerSocket *serverSocket*;  
 static int *port* = 8080;  
  
 public static void setPort(int port) {  
 if (port > 0) {  
 SimpleHttpServer.*port* = port;  
 }  
 }  
  
 public static void setBasePath(String basePath) {  
 if (basePath != null && new File(basePath).exists() && new File(basePath).isDirectory()) {  
 SimpleHttpServer.*basePath* = basePath;  
 }  
 }  
  
 // 启动SimpleHttpServer  
 public static void start() throws Exception {  
 *serverSocket* = new ServerSocket(*port*);  
 Socket socket = null;  
 while ((socket = *serverSocket*.accept()) != null) {  
 *threadPool*.execute(new HttpRequestHandler(socket));  
 }  
 *serverSocket*.close();  
 }  
  
 static class HttpRequestHandler implements Runnable {  
 private Socket socket;  
 public HttpRequestHandler(Socket socket) {  
 this.socket = socket;  
 }  
  
 public void run() {  
 String line = null;  
 BufferedReader br = null;  
 BufferedReader reader = null;  
 PrintWriter out = null;  
 InputStream in = null;  
  
 try {  
 reader = new BufferedReader(new InputStreamReader(socket.getInputStream()));  
 String header = reader.readLine();  
 // 由相对路径计算绝对路径  
 String filePath = *basePath* + header.split(" ")[1];  
 out = new PrintWriter(socket.getOutputStream());  
 if (filePath.endsWith("jpg") || filePath.endsWith("ico")) {  
 in = new FileInputStream(filePath);  
 ByteArrayOutputStream baos = new ByteArrayOutputStream();  
 int i= 0;  
 while ((i = in.read()) != -1) {  
 baos.write(i);  
 }  
 byte[] arrray = baos.toByteArray();  
 out.println("HTTP/1.1 200 OK");  
 out.println("Server: Molly");  
 out.println("Content-Type: image/jpeg");  
 out.println("Content-Length: " + arrray.length);  
 out.println("");  
 socket.getOutputStream().write(arrray, 0, arrray.length);  
 } else {  
 br = new BufferedReader(new InputStreamReader(new FileInputStream(filePath)));  
 out = new PrintWriter(socket.getOutputStream());  
 out.println("HTTP/1.1 200 OK");  
 out.println("Server: Molly");  
 out.println("Content-Type: text/html; charset=UTF-8");  
 out.println("");  
 while ((line = br.readLine()) != null) {  
 out.println(line);  
 }  
 }  
 out.flush();  
 } catch (IOException e) {  
 e.printStackTrace();  
 } finally {  
 close(br, in, reader, out, socket);  
 }  
  
 }  
  
 private void close(Closeable... closeables) {  
 if (closeables != null) {  
 for (Closeable closeable: closeables) {  
 try {  
 closeable.close();;  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
 }  
}

## 3.自定义同步器

public class Mutex {  
 // 静态内部类，自定义同步器  
 private static class Sync extends AbstractQueuedSynchronizer {  
 // 是否处于占用状态  
 protected boolean isHeldExclusively () {  
 return getState() == 1;  
 }  
 public boolean tryAcquire (int acquires) {  
 if (compareAndSetState(0, 1)) {  
 setExclusiveOwnerThread(Thread.*currentThread*());  
 return true;  
 }  
 return false;  
 }  
 protected boolean tryRelease (int release) {  
 if (getState() == 0) throw new IllegalMonitorStateException();  
 setExclusiveOwnerThread(null);  
 setState(0);  
 return true;  
 }  
 // 返回一个Condition，每个condition都包含一个condition队列  
 Condition newCondition () {  
 return new ConditionObject();  
 }  
 }  
  
 // 仅需要将操作代理到Sync上即可  
 private final Sync sync = new Sync();  
 public void lock () {sync.acquire(1);}  
 public boolean tryLock() {return sync.tryAcquire(1);}  
 public void unlock() {sync.release(1);}  
 public Condition newCondition () {return sync.newCondition();}  
 public boolean isLocked() {return sync.isHeldExclusively();}  
 public boolean hasQueuedThreads () { return sync.hasQueuedThreads(); }  
 public void lockInterruptible() throws InterruptedException {  
 sync.acquireInterruptibly(1);  
 }  
 public boolean tryLock (long timeout, TimeUnit unit) throws InterruptedException {  
 return sync.tryAcquireNanos(1, unit.toNanos(timeout));  
 }  
}

4. 线程间通信--使用通道Piped

public class Piped {  
 static class Print implements Runnable {  
 private PipedReader in;  
 public Print(PipedReader in) {  
 this.in = in;  
 }  
 public void run() {  
 int receive = 0;  
 try {  
 while ((receive = in.read()) != -1) {  
 System.*out*.print((char) receive);  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
 public static void main(String [] args) throws Exception {  
 PipedWriter out = new PipedWriter();  
 PipedReader in = new PipedReader();  
 // 将输入流和输出流进行连接，否则在使用时会抛出IOException  
 out.connect(in);  
 Thread printThread = new Thread(new Print(in), "Print Thread");  
 printThread.start();  
 int receive = 0;  
 while ((receive = System.*in*.read()) != -1) {  
 out.write(receive);  
 }  
 out.close();  
 }  
}

## 5. 模拟阻塞双向队列

public class BoundedQueue<T> {  
 private Object[] items;  
 private int addIndex, removeIndex, count;  
 private Lock lock = new ReentrantLock();  
 private Condition notEmpty = lock.newCondition();  
 private Condition notFull = lock.newCondition();  
  
 public BoundedQueue(int size) {  
 items = new Object[size];  
 }  
 public void add(T t) throws InterruptedException {  
 lock.lock();  
 try {  
 while (count == items.length) {  
 notFull.await();  
 }  
 items[addIndex] = t;  
 if (++addIndex == items.length) {  
 addIndex = 0;  
 }  
 ++ count;  
 notEmpty.signal();  
 } finally {  
 lock.unlock();  
 }  
 }  
 public T remove() throws InterruptedException {  
 lock.lock();  
 try {  
 while (count == 0) {  
 notEmpty.await();  
 }  
 Object x = items[removeIndex];  
 if (++removeIndex == items.length) {  
 removeIndex = 0;  
 }  
 --count;  
 notFull.signal();  
 return (T) x;  
 } finally {  
 lock.unlock();  
 }  
 }  
}