# volatile

## 背景案例

|  |
| --- |
| /\*\*  \* volatile 关键字 测试 任务  \*/  @Getter  public class MyRunnable implements Runnable {  private boolean flag = false;  @Override  public void run() {  try {  Thread.sleep(3000);  } catch (InterruptedException e) {  e.printStackTrace();  }  this.flag = true;  }  } |

|  |
| --- |
| /\*\*  \* volatile 关键字 测试  \*/  public class VolatileTest {  public static void main(String[] args) throws  InterruptedException {  //创建任务  MyRunnable runnable = new MyRunnable();  //创建线程  new Thread(runnable).start();  //主线程轮询  while(true) {  if(runnable.isFlag()) {  System.out.println("跳出循环, flag=" +  runnable.isFlag());  break;  }  }  }  } |

## 现象分析

**案例发现的问题**：线程1启动，3秒之后将自己的成员变量flag改为true,主线程启动后直接while(true)轮询 成员变量，结果是，一直访问到的值为false,但是当在while(true)中让主线程sleep1秒，就会访问到flag=true.

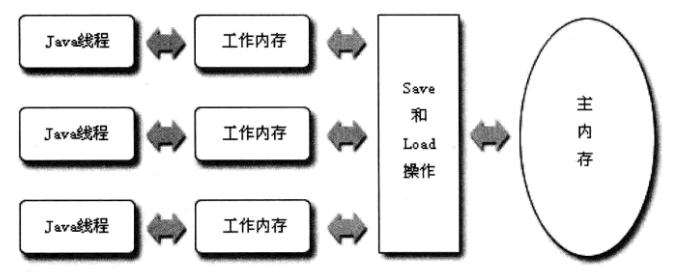
## 原因分析

1. 因为在MyRunnable类中，flag是成员变量,初始值false，是放在堆中(主存)，是各个线程共享的变量，当线程1启动时，过了3秒之后将成员变量flag的值复制到自己的缓存中，并将其修改为true,然后将值同步到堆中（主存）;
2. 在main线程启动之后，会轮询取成员变量flag的值（也是讲flag的值复制到自己的缓存中），开始读取的flag=false,由于while(true)这个玩意太猛了，执行太快，没办法让main线程去堆中去看看flag是否发生变化。就直接使用自己线程中的缓存的flag值，所以一直显示flag=false

## 内存可见性

在jvm当中，为了充分使用多核cpu计算效率，将一些共享的值放在一个主内存中，如果线程想去操作共享值时，将该变量从主存中复制到自己的内存中，操作完毕后再同步到主存中，因为每个线程是独享内存空间的（变量放在线程私有栈中），因此当多个线程操作共享数据时，彼此是不可见的。这就引发了 **内存可见性问题。**

**注意：当线程操作完后，其他线程再去操作时，如果条件允许，还是会去从主存中获取值，但是while(true)，太猛了，不给线程机会，为了提高效率。只能从自己的私有的缓存中，直接获取值。**



## 线程不安全的根本原因

**在《深入理解Java虚拟机：JVM高级特性与最佳实践》（12.3.1节“主内存与工作内存”,P319）中，给出了一个原因【原因1】：**

“每条线程有自己的工作内存，线程的工作内存中保存了被该线程使用到的变量的主内存副本拷贝，线程对该变量的所有操作都必须在工作内存中进行，而不能直接读写主内存中的变量。不同线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完成。”

**因此**

**线程内的关于外部变量的语境，与真实外部语境不一致。是导致线程不安全的根本原因**

**案例：**

|  |
| --- |
| public void subtract(){  try{  synchronized(lock){  i**f(ValueObject.list.size() == 0)**{  lock.wait();  }  ValueObject.list.remove(0);  }  }catch(InterruptedException e){  e.printStackTrace(); |

**执行过程中会出现异常。**

**线程B被唤醒后能执行操作 ValueObject.list.remove(0); 的先决条件 if(ValueObject.list.size() == 0) 就是 线程内的关于外部变量的语境，而出现异常的原因就在于该此时先决条件已经发生变化（ 与真实外部语境不一致，list.size为0）**

## 方案

volatile 关键字：当多个线程进行操作共享数据时，可以保证内存中的数据可见，相比较于synchronized是一种轻量级的同步策略。**我们可以理解，使用volatile关键字修饰的变量，线程操作之前，必须从主存中获取。**

**注意：1. volatile 不具备“互斥信性”**

1. **volatile 不能保证变量的“原子性”**

# ThreadLocal

## 什么ThreadLocal

* **生活中例子：**

假设有统一批次、统一品牌的衣服（口袋数是一样的），每个衣服（线程）上有好多口袋，并且每个口袋都有自己的编号，并且每个口袋只能装一个东西，而ThreadLocal就是口袋的编号，根据这个编号我们很容易找到口袋里的物品，并且不同的衣服的口袋是完全没有关系，但是口袋的编号是可以复用。

* **源码中的理解：**

ThreadLocal是Thread一个类别的存储，在Thread中有一个成员变量

（ThreadLocal.ThreadLocalMap threadLocals），ThreadLocalMap中存储了多个Entry<ThreadLocal<?>, Object>[]>,由此可知，在一个线程中，可以有自己的全局的变量，而这个变量其实是由多个Entry组成的，Entry中的key是ThreadLocal,value就是你要存储的值。

## ThreadLocal源码

### ThreadLocal类结构

--ThreadLocal

--ThreadLocalMap：是ThreadLocal静态内部类

--Entry extends WeakReference：是ThreadLocalMap静态内部类

### ThreadLocal源码精髓

Thread 的成员变量 ThreadLocal.ThreadLocalMap threadLocals

ThreadLocalMap 的成员变量 Entry[] table，Entry存入数组中位置利用Hash运算。

Entry extends WeakReference<ThreadLocal<?>>，其中的key是

ThreadLocal

1. ThreadLocal.set(T value);

获取当前线程，从当前线程中获取ThreadLocalMap，然后计算当前ThreadLocalMap的threadLocalHashCode值，并根据hashCode从当前的ThreadLocalMap中的table（Entry[]）获取对应的Entry，如果Entry存在就直接存入Entry中，否则新建Entry,并存入数组总

1. ThreadLocal.get();

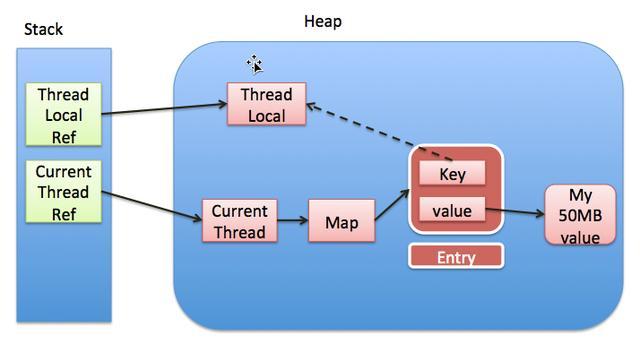
获取当前线程，从当前线程中获取ThreadLocalMap，然后计算当前ThreadLocalMap的threadLocalHashCode值，并根据hashCode从当前的ThreadLocalMap中的table（Entry[]）获取对应的Entry,最后将Entry.value返回。

1. ThreadLocal.createMap(Thread t, T firstValue);

用于创建新的ThreadLocalMap并赋值给Thread.threadLocals

1. new ThreadLocalMap(this, firstValue)

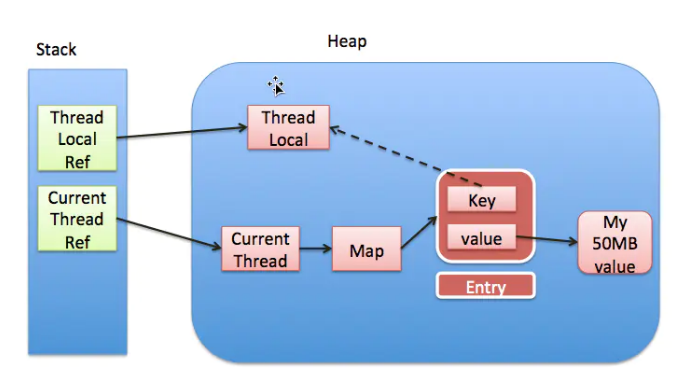
ThreadLocalMap的构造函数，主要是初始化Entry[] table,java8中数据的初始长度是16



## 面试题

### ThreadLocal内存泄漏问题

* **原因：**通过之前的分析已经知道，当使用ThreadLocal保存一个value时，会在ThreadLocalMap中的数组插入一个Entry对象，按理说key-value都应该以强引用保存在Entry对象中，但在ThreadLocalMap的实现中，key被保存到了WeakReference对象中。这就导致了一个问题，ThreadLocal在没有外部强引用时，发生GC时会被回收，如果创建ThreadLocal的线程一直持续运行，那么这个Entry对象中的value就有可能一直得不到回收，发生内存泄露。



上图中，实线代表强引用，虚线代表的是弱引用，如果threadLocal外部强引用被置为null(threadLocalInstance=null)的话，threadLocal实例就没有一条引用链路可达，很显然在gc(垃圾回收)的时候势必会被回收，因此entry就存在key为null的情况，无法通过一个Key为null去访问到该entry的value。同时，就存在了这样一条引用链：threadRef->currentThread->threadLocalMap->entry->valueRef->valueMemory,导致在垃圾回收的时候进行可达性分析的时候,value可达从而不会被回收掉，但是该value永远不能被访问到，这样就存在了内存泄漏。当然，如果线程执行结束后，threadLocal，threadRef会断掉，因此threadLocal,threadLocalMap，entry都会被回收掉。可是，在实际使用中我们都是会用线程池去维护我们的线程，比如在Executors.newFixedThreadPool()时创建线程的时候，为了复用线程是不会结束的，所以threadLocal内存泄漏就值得我们关注。

* **策略：**既然已经发现有内存泄露的隐患，自然有应对的策略，在调用ThreadLocal的**get()、set()**可能会清除ThreadLocalMap中key为null的Entry对象，这样对应的value就没有GC Roots可达了，下次GC的时候就可以被回收，当然如果调用remove方法，肯定会删除对应的Entry对象。如果使用ThreadLocal的set方法之后，没有显示的调用remove方法，就有可能发生内存泄露，所以养成良好的编程习惯十分重要，使用完ThreadLocal之后，记得调用remove方法

|  |
| --- |
| ThreadLocal<String> localName = new ThreadLocal();  try {  localName.set("占小狼");  // 其它业务逻辑  } finally {  localName.remove();  } |

### ThreadLocal用在什么地方

ThreadLocal是用在多线程的场景的，是线程私用的，可以保存线程上下文信息，在任意需要的地方可以获取；

例如：用ThreadLocal存储Connection，从而各个DAO可以获取同一Connection，可以进行事务回滚，提交等操作。

ThreadLocal无法解决线程间共享对象的更新问题，ThreadLocal对象建议使用static修饰。这个变量是针对一个线程内所有操作贡献的，所以设置为静态变量，所有此类实例共享次静态变量，也就是说在类第一次被使用时转载，只分配一块存储空间，所有此类的实例（只要是这个线程内定义的）都可以操控这个变量。

### hash冲突

* **问题产生背景：**ThreadLoalMap中的数组table,没有链表结构，是初始化一个大小16的Entry数组，Entry对象用来保存每一个key-value键值对，只不过这里的key永远都是ThreadLocal对象，通过ThreadLocal对象的set方法，结果把ThreadLocal对象自己当做key，放进了ThreadLoalMap中。
* **Hash冲突：**因为table没有链表结构，那发生hash冲突了怎么办？如下是源码：

|  |
| --- |
| private void set(ThreadLocal<?> key, Object value) {  Entry[] tab = table;  int len = tab.length;  int i = key.threadLocalHashCode & (len-1);  for (Entry e = tab[i];  e != null;  e = tab[i = nextIndex(i, len)]) {  ThreadLocal<?> k = e.get();  if (k == key) {  e.value = value;  return;  }  if (k == null) {  replaceStaleEntry(key, value, i);  return;  }  }  tab[i] = new Entry(key, value);  int sz = ++size;  if (!cleanSomeSlots(i, sz) && sz >= threshold)  rehash();  } |

在插入过程中，根据ThreadLocal对象的hash值，定位到table中的位置i，过程如下：

1、如果当前位置是空的，那么正好，就初始化一个Entry对象放在位置i上；

2、不巧，位置i已经有Entry对象了，如果这个Entry对象的key正好是即将设置的key，那么重新设置Entry中的value；

3、很不巧，位置i的Entry对象，和即将设置的key没关系，那么只能找下一个空位置；

这样的话，在get的时候，也会根据ThreadLocal对象的hash值，定位到table中的位置，然后判断该位置Entry对象中的key是否和get的key一致，如果不一致，就判断下一个位置

可以发现，set和get如果冲突严重的话，效率很低，因为ThreadLoalMap是Thread的一个属性，所以即使在自己的代码中控制了设置的元素个数，但还是不能控制其它代码的行为。

### 为什么使用弱引用？（Entry extends WeakReference）

**如果使用强引用：**

假设threadLocal使用的是强引用，在业务代码中执行threadLocalInstance==null操作，以清理掉threadLocal实例的目的，但是因为threadLocalMap的Entry强引用threadLocal，因此在gc的时候进行可达性分析，threadLocal依然可达，对threadLocal并不会进行垃圾回收，这样就无法真正达到业务逻辑的目的，出现逻辑错误

**如果使用弱引用：**

假设Entry弱引用threadLocal，尽管会出现内存泄漏的问题，但是在threadLocal的生命周期里（set,getEntry,remove）里，都会针对key为null的脏entry进行处理。

从以上的分析可以看出，使用弱引用的话在threadLocal生命周期里会尽可能的保证不出现内存泄漏的问题，达到安全的状态。

# 原子性

## i++ 和 ++i的原理

**java代码**

public void test1() {

int i = 0;

int j = i++;

}

public void test2() {

int i = 0;

int j = ++i;

**再通过 javap -c demo.class 进行反编译，得到反编译代码如下：**

|  |
| --- |
| **public void test1();**  **Code:**  **0: iconst\_0 // 入栈常数0**  **1: istore\_1 // 赋值1号存储单元为常数0（i = 0）**  **2: iload\_1 // 加载1号存储单元值到寄存器（记为stack[0] = 0）**  **3: iinc 1, 1 // 递增1号存储单元值（i = i + 1 = 0 + 1 = 1）**  **6: istore\_2 // 赋值2号存储单元为寄存器值（j = stack[0] = 0）**  **7: return // 返回（此时：i = 1，j = 0）**    **public void test2();**  **Code:**  **0: iconst\_0 // 入栈常数0**  **1: istore\_1 // 赋值1号存储单元为常数0（i = 0）**  **2: iinc 1, 1 // 递增1号存储单元值（i = i + 1 = 0 + 1 = 1）**  **5: iload\_1 // 加载1号存储单元值到寄存器（记为stack[0] = 1）**  **6: istore\_2 // 赋值2号存储单元为寄存器值（j = stack[0] = 1）**  **7: return // 返回（此时：i = 1，j = 1）** |

## volatile不能保证变量的原子性

变量的原子性，是指操作的原子性，例如想要对共享变量进行修改，需要进行【读、改、写】三步操作，此时，volatile不能保证这3步操作的原子性，因此，volatile不能完全避免线程不安全的问题。

## 方案

使用原子变量：java.util.concurrent.atomic包下的提供常用的原子变量。

# 原子变量

定义：java.util.concurrent.atomic包下的提供常用的原子变量。

## 原子变量保证原子性的方案

1. 使用volatile 保证内存的可见性；
2. CAS（Compare-And-Swap）算法保证数据的原子性

## CAS 算法

CAS 算法是硬件对于并发操作共享数据的支持。

算提供的api有3个参数：

V : 读取时候的内存值

A : 预估值，更新的时候读取的内存的值

B : 更新值

只有V = A 时，才能更新B成功。

通常使用 死循环配合CAS算法，形成自旋

## 模拟cas算法

|  |
| --- |
| /\*\*  \* 模拟cas算法  \*/  public class CompareAndSwapTest {  private int value;  /\*\*  \* 获取内存值  \*/  public synchronized int getValue() {  return value;  }  /\*\*  \* 比较and替换  \* @param expectedValue 预估值  \* @param newValue 新值  \* @return 内存值  \*/  public synchronized int compareAndSwap(int expectedValue, int newValue) {  //获取内存值  int oldValue = value;  //比较  if(oldValue == expectedValue) {  this.value = newValue;  }  return oldValue;  }  public synchronized boolean compareAndSet(int expectedValue, int newValue) {  return expectedValue == compareAndSwap(expectedValue, newValue);  }  } |

|  |
| --- |
| /\*\*  \* 测试模拟cas  \*/  public class Client {  public static void main(String[] args) {  CompareAndSwapTest swapTest = new CompareAndSwapTest();  for (int i = 0; i < 100; i++) {  new Thread(() -> {  int expectedValue = swapTest.getValue();  boolean isSwap = swapTest.compareAndSet(expectedValue, (int)(Math.random() \* 101));  System.out.println(isSwap);  }).start();  }  }  } |

# ConcurrentHashMap

## 简介

1. java5.0在java.util.concurrent包中提供了多种并发容器类来改进同步容器的性能。
2. ConcurrentHashMap同步容器是java5增加的一个线程安全的哈希表。对与多线程的操作，介于HashMap与HashTable之间。内存采用“锁分段”机制替代Hashtable的独占锁，进而提高性能。
3. ConcurrentHashMap、ConcurrentListMap、ConcurrentSkipListSet、

CopyOnWriteArrayList、CopyOnWriteArraySet。当期望许多线程访问一个给定collection时，ConcurrentHashMap通常优与同步的HashMap，ConcurrentSkipListMap通常优与同步的TreeMap。当期望的读数和遍历远远大于列表更新数时，CopyOnWriteArrayList优于同步的ArrayList

## hashtable

* 简介：
* 优缺点：
* 源码精髓：
* 复合操作线程不安全

例如：“若不存在则添加”、“若存在则删除”

if(!table.contains()) {\*\*\*\*\*}

原因是：contains()方法单独加锁，put()方法单独加锁，整个流程不是原子性。

* 保证线程措施

## HashMap

* 简介：
* 优缺点：
* 源码精髓：
* 线程不安全点：
* 常用保证线程安全的措施

## ConcurrentHashMap

### 保证线程安全的思想

1.8之前使用锁分段机制实现线程安全

通过concurrentLevel（并发级别），将整个数组分成若干segment（段），在每个段内的操作是线程安全的。

1.8使用CAS算法。

### 源码精髓

## CopyOnWriteArrayList

* 使用场景：

添加操作多时，效率低，因为每次添加会进行复制，开销非常大，并发迭代操作时可以选择。

* 测试案例

|  |
| --- |
| /\*\*  \* 线程安全的list遍历添加操作  \*/  public class ArrayListRunnable implements Runnable {  private static List<String> list  = Collections.synchronizedList(new ArrayList<>());  static {  list.add("AA");  list.add("BB");  list.add("CC");  }  @Override  public void run() {  Iterator<String> iterator = list.iterator();  while (iterator.hasNext()) {  System.out.println(iterator.next());  list.add("AA");  }  }  } |

|  |
| --- |
| /\*\*  \* 赋值并写入集合 遍历添加任务  \*/  public class CopyOnWriteListRunnable implements Runnable {  private static List<String> list = new CopyOnWriteArrayList<>();  static {  list.add("AA");  list.add("BB");  list.add("CC");  }  @Override  public void run() {  Iterator<String> iterator = list.iterator();  while (iterator.hasNext()) {  System.out.println(iterator.next());  list.add("AA");  }  }  } |

|  |
| --- |
| /\*\*  \* 遍历添加测试  \*/  public class Test {  public static void main(String[] args) {  //测试线程安全的List,报错ConcurrentModificationException  ArrayListRunnable listRunnable = new ArrayListRunnable();  for (int i = 0; i < 10; i++) {  new Thread(listRunnable).start();  }  //测试CopyOnWriteList  CopyOnWriteListRunnable writeListRunnable = new CopyOnWriteListRunnable();  for (int i = 0; i < 10; i++) {  new Thread(writeListRunnable).start();  }  }  } |

# CountDownLatch

## 简介

Java 5.0 在 java.util.concurrent 包中提供了多种并发容器类来改进同步容器的性能。

* 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。

* 闭锁可以延迟线程的进度直到其到达终止状态，闭锁可以用来确保某些活

动直到其他活动都完成才继续执行：

* 确保某个计算在其需要的所有资源都被初始化之后才继续执行;
* 确保某个服务在其依赖的所有其他服务都已经启动之后才启动;
* 等待直到某个操作所有参与者都准备就绪再继续执行。

## 案例：

|  |
| --- |
| /\*\*  \* 闭锁 测试任务  \*/  public class CountDownLatchRunnable implements Runnable {  private CountDownLatch latch;  public CountDownLatchRunnable(CountDownLatch latch) {  this.latch = latch;  }  @Override  public void run() {  try {  for (int i = 0; i < 10; i++) {  int j = i + 1;  System.out.println("线程：" + Thread.currentThread().getName() + "值：" + j);  }  } finally {  latch.countDown();  }  }  } |

|  |
| --- |
| /\*\*  \* 闭锁测试类  \*/  public class Test {  public static void main(String[] args) throws InterruptedException {  //10个线程执行相同任务  CountDownLatch latch = new CountDownLatch(10);  CountDownLatchRunnable runnable = new CountDownLatchRunnable(latch);  long start = System.currentTimeMillis();  for (int i = 0; i < 10; i++) {  new Thread(runnable).start();  }  latch.await();  long end = System.currentTimeMillis();  System.out.println("共计消耗：" + (end - start));  }  } |

# CountDownLatch和CyclicBarrier的理解和区别

## 理解和区别

从字面上理解，CountDown表示减法计数，Latch表示门闩的意思，计数为0的时候就可以打开门闩了。

Cyclic Barrier表示循环的障碍物。两个类都含有这一个意思：对应的线程都完成工作之后再进行下一步动作，也就是大家都准备好之后再进行下一步。

然而两者最大的区别是，进行下一步动作的动作实施者是不一样的。这里的“动作实施者”有两种，一种是主线程（即执行main函数），另一种是执行任务的其他线程，后面叫这种线程为“其他线程”，区分于主线程。对于**CountDownLatch，当计数为0的时候，下一步的动作实施者是main函数**；对于CyclicBarrier，**下一步动作实施者是“其他线程”。**

## 举例说明CountDownLatch

对于CountDownLatch，其他线程为游戏玩家，比如英雄联盟，**主线程为控制游戏开始的线程**。在所有的玩家都准备好之前，主线程是处于等待状态的，也就是游戏不能开始。当所有的玩家准备好之后，下一步的动作实施者为主线程，即开始游戏。

我们使用代码模拟这个过程，我们模拟了三个玩家，在三个玩家都准备好之后，游戏才能开始。代码的输出结果为:

|  |
| --- |
| 正在等待所有玩家准备好  player0 已经准备好了, 所使用的时间为 1.235s  player2 已经准备好了, 所使用的时间为 1.279s  player3 已经准备好了, 所使用的时间为 1.358s  player1 已经准备好了, 所使用的时间为 2.583s  开始游戏 |

CountDownLatch的代码：



## 举例说明CyclicBarrier

对于CyclicBarrier，假设有一家公司要全体员工进行团建活动，活动内容为翻越三个障碍物，每一个人翻越障碍物所用的时间是不一样的。但是公司要求所有人在翻越当前障碍物之后再开始翻越下一个障碍物，也就是所有人翻越第一个障碍物之后，才开始翻越第二个，以此类推。类比地，每一个员工都是一个“其他线程”。当所有人都翻越的所有的障碍物之后，程序才结束。而主线程可能早就结束了，这里我们不用管主线程。

我们使用代码来模拟上面的过程。我们设置了三个员工和三个障碍物。可以看到所有的员工翻越了第一个障碍物之后才开始翻越第二个的，下面是运行结果：

|  |
| --- |
| main function is finished.  队友1, 通过了第0个障碍物, 使用了 1.432s  队友0, 通过了第0个障碍物, 使用了 1.465s  队友2, 通过了第0个障碍物, 使用了 2.26s  队友1, 通过了第1个障碍物, 使用了 1.542s  队友0, 通过了第1个障碍物, 使用了 2.154s  队友2, 通过了第1个障碍物, 使用了 2.556s  队友1, 通过了第2个障碍物, 使用了 1.426s  队友2, 通过了第2个障碍物, 使用了 2.603s  队友0, 通过了第2个障碍物, 使用了 2.784s |

CyclicBarrier代码



# Callable

## 简介

* Java 5.0 在 java.util.concurrent 提供了一个新的创建执行

线程的方式：Callable 接口

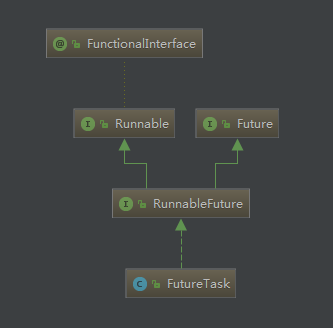
* Callable 接口类似于 Runnable，两者都是为那些其实例可

能被另一个线程执行的类设计的。但是 Runnable 不会返

回结果，并且无法抛出经过检查的异常，Callable可以。

* Callable 需要依赖FutureTask ，FutureTask 也可以用作闭

锁。



## 创建线程的方式

1. 实现Runable接口
2. 实现Thread
3. 实现Callable
4. 利用线程池

## 案例

|  |
| --- |
| /\*\*  \* callable 测试  \*/  public class MyCallable implements Callable<Integer> {  //相较于Runnable,有返回值、可以抛出异常  @Override  public Integer call() throws Exception {  int sum = 0;  for (int i = 0; i < 1000; i++) {  sum = sum + i;  }  return sum;  }  } |

|  |
| --- |
| /\*\*  \* 测试类  \*/  public class CallableTest {  public static void main(String[] args) throws Exception {  //callable的目的就是能获取线程执行的结果  MyCallable callable = new MyCallable();  //需要使用FutureTask对Callable进行封装，用于获取结果  FutureTask<Integer> task = new FutureTask<>(callable);  //线程  new Thread(task).start();  //通过FutureTask.get()方法获取返回值，内部是自旋判断线程状态  System.out.println(task.get());  }  } |

# Lock 同步锁

## 解决多线安全问题的方式：

1. **synchronized关键字（隐式）**，此关键字，我的理解是可以跟类、实例分别要锁，主要的执行方案是：
2. **同步代码块**

* 类级别：synchronized (CallableTest.class) {do something…}
* 实例级别：synchronized (new CallableTest()) {do something…}

1. **同步方法**

* 类级别：public static synchronized int compareAndSwap() {}
* 实例级别：public synchronized int compareAndSwap() {}

1. **jdk1.5同步锁Lock**

**注意：lock是显示锁，需要通过lock()方法上锁，必须通过unlock()方法进行释放锁。更加灵活**

## 案例

|  |
| --- |
| /\*\*  \* 卖票任务  \* 解决多线安全问题的方式：  \* 1.同步代码块 synchronized 隐式锁,比同步方法灵活。  \* 2.同步方法 synchronized 隐式锁  \* 3.同步锁 lock 显示锁,需要通过 lock() 和 unlock(),注意，一定 在finally内加unlock()方法。  \*/  public class TickRunnable implements Runnable {  private Lock lock = new ReentrantLock();  private int tickNum;  public TickRunnable(int tickNum) {  this.tickNum = tickNum;  }  @Override  public void run() {  while(true) {  lock.lock();  try {  if (tickNum > 0) {  Thread.sleep(200);  System.out.println(Thread.currentThread().getName() + "售票。余票：" + --tickNum);  } else {  break;  }  } catch (Exception e) {  e.printStackTrace();  }finally {  lock.unlock();  }  }  }  } |

|  |
| --- |
| /\*\*  \* 售票客户端  \*/  public class TickClient {  public static void main(String[] args) {  TickRunnable runnable = new TickRunnable(100);  new Thread(runnable, "1号窗口").start();  new Thread(runnable, "2号窗口").start();  new Thread(runnable, "3号窗口").start();  }  } |

# 生产者\_消费者案例

## 案例简述

* **角色：**

店员：数据存储，并对外数据插入，和数据删除的方法。

生产者：产品、数据增加的具体的执行者。

消费者：产品、数据减少的具体的消费者。

* **案例考察的点：**

1. 多线程情况下，操作（增、减）保证库存数据的线程安全。
2. 多个生产者生产数据时，（生产过快）当库存数据已满，怎么保证数据不丢失。
3. 多个消费者消费数据时，（消费过快）当库存数据为空，怎么保证消费者获取数据正确且不重复的数据。
4. 怎么避免线程等待之后，无法被唤醒，导致线程不能结束。
5. 怎么防止wait的虚假唤醒

* **解决方案：**

1. **使用synchronized解决线程安全问题，使用等待（wait）、唤醒（notify、notifyAll）解决线程通信问题,在wait之后，调用一次notifyAll，保证线程在wait状态唤醒之后，继续执行，也能执行notifyAll，使用while(condition)，让唤醒后的线程继续执行判断条件。**
2. **使用Lock解决线程安全问题，使用condition解决线程通信问题。**

## 解决代码一

**注释：使用synchronized解决线程安全问题，使用等待（wait）、唤醒（notify、notifyAll）解决线程通信问题**。



## 解决代码二



# 线程间通信-线程按序交替案例

## 简介

* **描述：**

编写一个程序，开启 3 个线程，这三个线程的 ID 分别为 A、B、C，每个线程将自己的 ID 在屏幕上打印 10 遍，要求输出的结果必须按顺序显示。

如：ABCABCABC…… 依次递归

* **案例考点：**

1. 怎么让线程不是自己的执行的时候睡眠等待；
2. 怎么让线程之间进行通信
3. 怎么让自己的执行条件，保证原子性（线程安全）。

* **解决方案：**

1. 用Lock.lock() + 判断条件，保证了 执行条件的线程安全和打印的线程安全，即Condition.await 与 Condition.singal 之间的就是你需要执行的代码逻辑。因为线程被唤醒之后，接着以前的位置继续执行。；
2. 用Condition进行线程间通信，其中一定要有判断条件，来触发await,注意signal的执行，是没有判断条件，因为线程被唤醒之后会接着以前的位置继续执行。

## 解决代码



# 读-写锁

## 简介

* ReadWriteLock 维护了一对相关的锁，一个用于只读操作， 另一个用于写入操作。只要没有 writer，读取锁可以由 多个 reader 线程同时保持。写入锁是独占的。。
* ReadWriteLock 读取操作通常不会改变共享资源，但执行 写入操作时，必须独占方式来获取锁。对于读取操作占多数的数据结构。 ReadWriteLock 能提供比独占锁更高 的并发性。而对于只读的数据结构，其中包含的不变性可以完全不需要考虑加锁操作。
* 读写、写写，互斥；读读不互斥。

## 代码演示



# 线程八锁

## 关键点

* **无论是对象或者实例都只有一把锁，只能有一个线程持有。具体表现的形式如下：**

1. 对象中的非静态方法synchronized，是属于this(实例级别)；
2. 对象中的静态方法synchronized，是属于.class(类级别的)

* **针对以上概念，出现加锁的方式：**

1. **public void synchronized() {} //实例锁**
2. **public static void synchronized() {} //对象锁**
3. **synchronized(new Object()) {} //实例锁**
4. **synchronized(Object) {} //对象锁**

## 具体描述

* **前提是，同一个实例，一个对象里面如果有多个synchronized方法，某一个时刻内，只要一个线程去调用 其中的一个synchronized方法了，其它的线程都只能等待，换句话说，某一个时刻 内，只能有唯一一个线程去访问这些synchronized方法**
* **前提是，同一个实例，锁的是当前对象this（实例级别），被锁定后，其它的线程都不能进入到当前对象的其它的 synchronized方法**
* **加个普通方法后发现和同步锁无关**
* **换成两个对象后，不是同一把锁了，情况立刻变化。**
* **都换成静态同步方法后，情况又变化**
* **所有的非静态同步方法用的都是同一把锁——实例对象本身，也就是说如果一个实 例对象的非静态同步方法获取锁后，该实例对象的其他非静态同步方法必须等待获 取锁的方法释放锁后才能获取锁，可是别的实例对象的非静态同步方法因为跟该实 例对象的非静态同步方法用的是不同的锁，所以毋须等待该实例对象已获取锁的非 静态同步方法释放锁就可以获取他们自己的锁。**
* **所有的静态同步方法用的也是同一把锁——类对象本身，这两把锁是两个不同的对 象，所以静态同步方法与非静态同步方法之间是不会有竞态条件的。但是一旦一个 静态同步方法获取锁后，其他的静态同步方法都必须等待该方法释放锁后才能获取 锁，而不管是同一个实例对象的静态同步方法之间，还是不同的实例对象的静态同 步方法之间，只要它们同一个类的实例对象！**

# 线程池

## 释义

线程池提供了一个线程队列，队列中保存所有等待状态的线程。避免了创建与销毁额外的开销，提高了响应的速度。

## 线程池体系结构

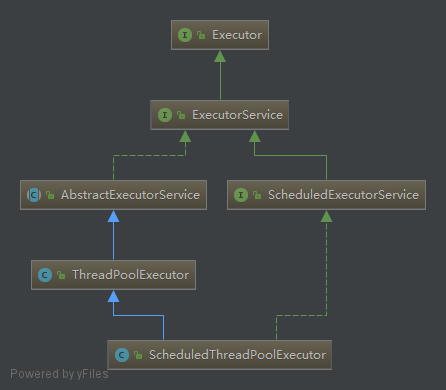
java.util.concurrent.Excutor:负责线程的使用与调度的根接口

|--ExecutorService 子接口：线程池的主要接口

|--ThreadPollExecutor 线程池的实现类

|--ScheduledExecutorService 子接口：负责线程的调度

|--ScheduledThreadPoolExecutor:继承ThreadPollExecutor，实现ScheduledExecutorService



## 工具类

**Executors:**

1. **ExecutorService newFixedThreadPool(int nThreads) //固定大小线程池**
2. **ExecutorService newCachedThreadPool() //缓存线程池，线程池的数量不固定，可以根据需求自动的更改数量**
3. **ExecutorService newSingleThreadExecutor() //创建单个线程池，线程池只有一个线程**
4. **ScheduledExecutorService newScheduledThreadPool(int corePoolSize) 、**

**//创建固定大小的线程池，可以延迟或定时的执行任务。**

## 线程池使用

1. **创建线程池，（也可以定义通用的线程池，供多个业务使用）**
2. **为线程池提交任务（Runnable、Callable）**
3. **关闭线程池（如果是自己业务中使用，需要关闭，通用的话，不需要关闭）；**
4. **获取结果（如果是多个话，可以通过while遍历Future,根据isDone判断是否线程完毕，从而获取结果）**



## 线程池的核心参数

**以下是摘抄ThreadPoolExecutor类中**

**public ThreadPoolExecutor(int corePoolSize,**

**int maximumPoolSize,**

**long keepAliveTime,**

**TimeUnit unit,**

**BlockingQueue<Runnable> workQueue,**

**ThreadFactory threadFactory,**

**RejectedExecutionHandler handler)**

1. **corePoolSize：**核心线程数
2. **maximumPoolSize：**最大线程池数
3. **keepAliveTime：**当线程数大于核心时，这是多余空闲线程在终止前等待新任务的最长时间。
4. **unit：**是keepAliveTime等待时间的单位，例如**TimeUnit.SECONDS**
5. **workQueue:** 用于在任务执行之前保留任务的队列。此队列将只保存{@code execute}方法提交的{@code Runnable}或者Callable任务。
6. **threadFactory:** 创建线程的工厂
7. **handler：**拒绝策略。jdk默认提供了四种拒绝策略

* CallerRunsPolicy：触发拒绝策略，只要线程池没有关闭的话，则使用调用线程直接运行任务。一般并发比较小，性能要求不高，不允许失败。但是，由于调用者自己运行任务，如果任务提交速度过快，可能导致程序阻塞，性能效率上必然的损失较大
* AbortPolicy：丢弃任务，并抛出拒绝执行 RejectedExecutionException 异常信息。线程池默认的拒绝策略。必须处理好抛出的异常，否则会打断当前的执行流程，影响后续的任务执行
* DiscardPolicy：直接丢弃，其他啥都没有
* DiscardOldestPolicy：当触发拒绝策略，只要线程池没有关闭的话，丢弃阻塞队列 workQueue 中最老的一个任务，并将新任务加入

## 线程池工作流程简述

线程池在创建的时候，根据核心线程数（**corePoolSize**）初始创建固定数量的线程，这部分线程随着线程池创建而创建，线程池销毁而销毁。当提交任务数大于核心线程数时，会将多余任务存入到工作队列中（**workQueue**）。当队列饱和后，如果还有多余任务，则会继续创建线程，直至达到最大线程数（**maximumPoolSize**），此时，再多余的任务，则会触发线程池的拒绝策略了。

另外，当线程池中非核心线程，如果超过保持活跃时间（**keepAliveTime**）没有接受到新任务，则会被销毁。

## 根据核心参数剖析四种常用线程池

### 固定大小的线程池

new ThreadPoolExecutor(nThreads, nThreads,

0L, TimeUnit.MILLISECONDS,

new LinkedBlockingQueue<Runnable>())

核心线程数等于最大线程数，保持活跃时间为0毫秒，工作队列长度是Integer.MAX\_VALUE，拒绝策略：AbortPolicy；

基于参数设置，我们可知，固定大小的线程池 如果提交任务数量超过 核心线程数，直接放入 工作队列中，因此我们使用固定大小的线程池，需要注意，如果提交的任务 最好是**执行时间短，不会有短时间大量的并发，并且任务间没有执行顺序限制。否则有可能将整个内存吃掉，影响整个程序的运行。**

### 单个线程的线程池

new ThreadPoolExecutor(1, 1,

0L, TimeUnit.MILLISECONDS,

new LinkedBlockingQueue<Runnable>())

核心线程数、最大线程数都等于1，保持活跃时间0，工作队列长度是Integer.MAX\_VALUE，拒绝策略：AbortPolicy。

其实单个线程的线程池，是固定大小线程池的特例。因为跟固定大小的线程池有同样的特点，因为只有1个核心线程，所以，提交的任务是可以有序的执行。

### SynchronousQueue

* **定义：**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，甚至连一个队列的容量都没有。
* **使用场景：**它非常适合于传递性设计，在这种设计中，在一个线程中运行的对象要将某些信息、事件或任务传递给在另一个线程中运行的对象，它就必须与该对象同步。
* **线程安全：**对于正在等待的生产者和使用者线程而言，此类支持可选的公平排序策略。默认情况下不保证这种排序。但是，使用公平设置为 true 所构造的队列可保证线程以 FIFO 的顺序进行访问。 公平通常会降低吞吐量，但是可以减小可变性并避免得不到服务。
* **描述：**SynchronousQueue 内部没有容量，但是由于一个插入操作总是对应一个移除操作，反过来同样需要满足。那么一个元素就不会再SynchronousQueue 里面长时间停留，一旦有了插入线程和移除线程，元素很快就从插入线程移交给移除线程。也就是说这更像是一种信道（管道），资源从一个方向快速传递到另一方 向。需要特别说明的是，尽管元素在SynchronousQueue 内部不会“停留”，但是并不意味之SynchronousQueue 内部没有队列。实际上SynchronousQueue 维护者线程队列，也就是插入线程或者移除线程在不同时存在的时候就会有线程队列。既然有队列，同样就有公平性和非公平性特性，公平性保证正在等待的插入线 程或者移除线程以FIFO的顺序传递资源。显然这是一种快速传递元素的方式，也就是说在这种情况下元素总是以最快的方式从插入着（生产者）传递给移除着（消费者），这在多任务队列中是最快处理任务的方式。

### 缓存线程池

new ThreadPoolExecutor(0, Integer.MAX\_VALUE,

60L, TimeUnit.SECONDS,

new SynchronousQueue<Runnable>())

核心线程数=0，最大线程池数= Integer.MAX\_VALUE，保持活跃时间60s,

工作队列SynchronousQueue，拒绝策略：AbortPolicy

从参数分析，改线程池没有初始核心线程，任务来时，先入SynchronousQueue队列，当线程没有任务时，超过60s,就会被销毁。注意，SynchronousQueue队列，只能是存入和取走是一对操作。也就是说，这种线程池适合**短时间任务量特别多、任务执行时间短。但是要注意防止大量任务涌入，造成程序瘫痪。**

### DelayedWorkQueue

* **简述：**

任务队列会根据任务延时时间的不同进行排序，延时时间越短地就排在队列的前面，先被获取执行。

队列是先进先出的数据结构，就是先进入队列的数据，先被获取。但是有一种特殊的队列叫做优先级队列，它会对插入的数据进行优先级排序，保证优先级越高的数据首先被获取，与数据的插入顺序无关。

### 调度线程池

super(corePoolSize, Integer.MAX\_VALUE, 0, NANOSECONDS,

new DelayedWorkQueue())

核心线程数时存入的，最大线程池数= Integer.MAX\_VALUE，保持活跃时间0，工作队列为延时队列（可根据延时时间进行排序，时间越短排在前面）

根据线程池参数，可知，线程初始创建corePoolSize个数的线程，如果多余任务，存入到延时队列中，创建新的线程从延时队列中获取任务，非核心线程执行完任务后，就被销毁。

# ForkJoinPool 分支/ 合并框架 工作窃取

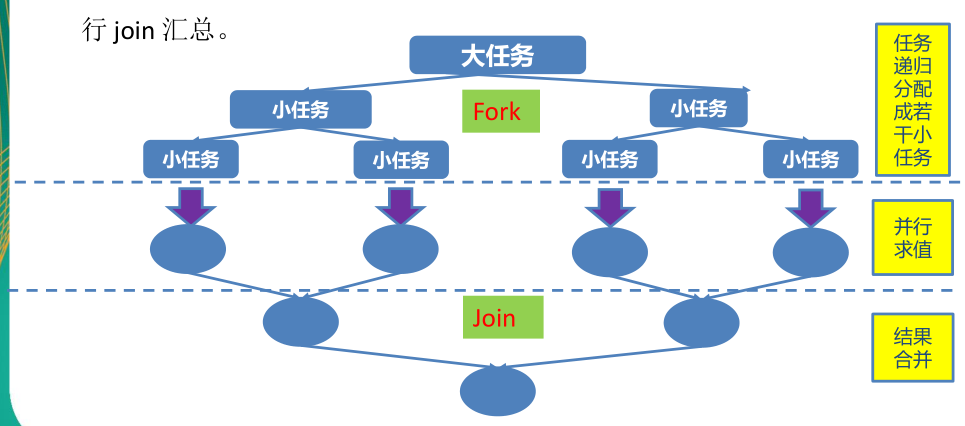
## 释义

* **Fork/Join 框架：**

就是在必要的情况下，将一个大任务，进行拆分(fork)成

若干个小任务（拆到不可再拆时），再将一个个的小任务运算的结果进

行 join 汇总。



* **工作窃取：**

**假设主机cpu是4核的，当遇到大任务时，将其拆分成小任务，分别放入4个线程的队列中，分别执行，执行完毕后进行求和。但是有可能其中一个线程在执行某个任务时候，耗时较多，导致在相同情况下，其他线程已经执行完了自己队列中的任务，此时，这个线程就会随机向没有执行完的线程的工作队列末尾进行偷取一个任务，进行执行。**

**A线程 B线程 C线程 D线程**

窃取

## 案例：使用工作窃取执行 0-1亿求和

分支、合并的关键点：

1. 任务的开始值、任务的结束值
2. 任务拆分粒度的边界值（即拆分到什么时候为止）
3. 以什么规则进行拆分（案例中，是以中间值进行拆分）

