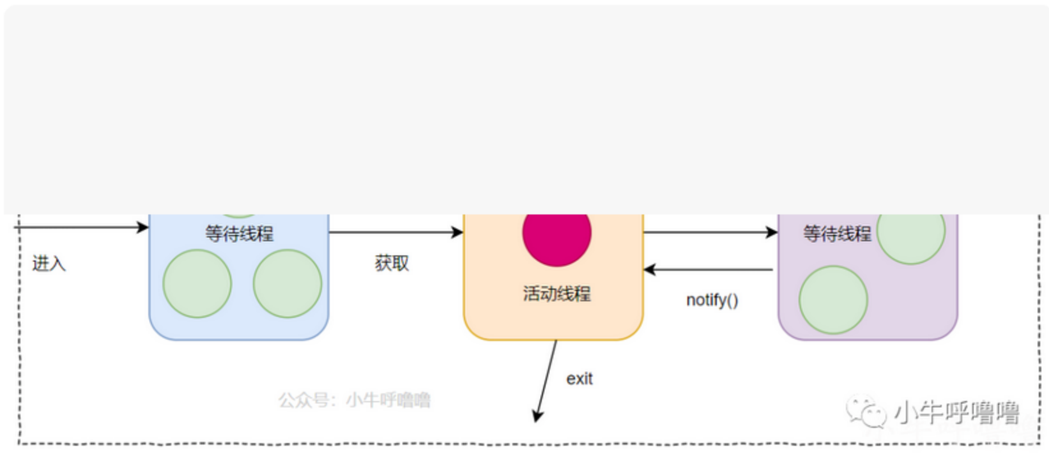
关于synchronized你知道的不知道的都在这了

JAVA旭阳 2023-05-08 08:12 Posted on 福建



前言

大家好，我是呼噜噜，在之前的文章中https://mp.weixin.qq.com/s/0Ii636KQ9sWwX-OhdlPIYw，我们知道了 `volatile关键字` 可以保证可见性、有序性，但无法保证原子性的。今天我们来聊聊synchronized关键字，其可以同时保证三者，实现线程安全。

线程安全

在介绍 `synchronized关键字` 之前，我们得强调一下什么是线程安全，所谓线程安全：

> **当多个线程同时访问一个对象时， 如果不用考虑这些线程在运行时环境下的调度和交替执行， 也不需要进行额外的同步， 或者在调用方进行任何其他的协调操作， 调用这个对象的行为都可以获得正确的结果， 那就称这个对象是线程安全的。**

什么是synchronized关键字？

在 Java 早期版本中，synchronized 属于 **重量级锁**，效率低下；不过在 Java 6 之后，Java 官方对从 JVM 层面对 synchronized 较大优化，所以现在的 synchronized 锁效率也优化得非常不错。目前不论是各种开源框架还是 JDK 源码都大量使用了 synchronized 关键字

synchronized实现方式

`synchronized` 的使用其实比较简单，可以用它来修饰实例方法和静态方法，也可以用来修饰代码块。我们需要注意的是 `synchronized` 是一个对象锁，也就是它锁的是一个对象。我们无论使用哪一种方法， `synchronized` 都需要有一个锁对象

1. 修饰实例方法
2. 修饰静态方法
3. 修饰代码块

1.修饰实例方法

synchronized修饰实例方法, 在方法上加上synchronized关键字即可。

```
public class SynchronizedTest1 {
    public synchronized void test() {
        System.out.println("synchronized 修饰 方法");
    }
}
```

此时，synchronized加锁的对象就是这个方法所在实例的本身,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**。

补充一个常见的面试题：构造方法可以用synchronized关键字修饰吗？

> 不能，也不需要，因为构造方法本身就是线程安全的

2.修饰静态方法

synchronized修饰静态方法的使用与实例方法并无差别，在静态方法上加上synchronized关键字即可

```
public static synchronized void test(){
       i++;
}
```

由于静态方法不属于任何一个实例对象，归整个类所有，不依赖于类的特定实例，被类的所有实例共享。给静态方法加 `synchronized` 锁，会作用于类的所有对象实例 ，进入同步代码前要获得 **当前静态方法所在类的Class对象的锁**。

有一点我们需要知道：如果一个线程 A 调用一个实例对象的非静态 synchronized 方法，而线程 B 需要调用这个实例对象所属类的静态 synchronized 方法，是允许的，不会发生互斥现象，**因为访问静态 synchronized 方法占用的锁是当前类的锁，而访问非静态 synchronized 方法占用的锁是当前实例对象锁**。

3.修饰代码块

synchronized修饰代码块需要传入一个对象。

```
public class SynchronizedTest2 {
    public void test() {
        synchronized (this) {
            System.out.println("synchronized 修饰 代码块");
        }
    }
}
```

此时synchronized加锁对象即为传入的这个对象实例，指定加锁对象，进入同步代码库前要获得给定对象的锁需要注意的是这里的**this **：

> 1. synchronized(object) ，表示进入同步代码库前要获得 **给定对象的锁**
> 2. synchronized(类.class) ，表示进入同步代码前要获得 **给定 Class 的锁**
> 3. 最好不要使用 synchronized(String a) ，因为在 JVM 中，字符串常量池具有缓存功能，**如果我们多次加锁,会加锁在同一个对象上**😂

synchronized关键字底层原理

想要搞清楚synchronized关键字的底层原理，我们得看下其对应的底层字节码

synchronized修饰实例方法

以上文SynchronizedTest1类为例子，其中synchronized关键字修饰实例方法

获取SynchronizedTest1.class的字节码：

```
javac SynchronizedTest1.java
javap -c -v SynchronizedTest1.class

Classfile /D:/ideaProjects/src/main/java/com/zj/ideaprojects/demo/test2/SynchronizedTest1.class
  Last modified 2022-10-28; size 466 bytes
  MD5 checksum a28131024850f35538b6ad60621b8838
  Compiled from "SynchronizedTest1.java"
public class com.zj.ideaprojects.demo.test2.SynchronizedTest1
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #6.#14         // java/lang/Object."<init>":()V
   #2 = Fieldref           #15.#16        // java/lang/System.out:Ljava/io/PrintStream;
   #3 = String             #17            // synchronized 修饰 方法
   #4 = Methodref          #18.#19        // java/io/PrintStream.println:(Ljava/lang/String;)V
   #5 = Class              #20            // com/zj/ideaprojects/demo/test2/SynchronizedTest1
   #6 = Class              #21            // java/lang/Object
   #7 = Utf8               <init>
   #8 = Utf8               ()V
   #9 = Utf8               Code
  #10 = Utf8               LineNumberTable
  #11 = Utf8               test
  #12 = Utf8               SourceFile
  #13 = Utf8               SynchronizedTest1.java
  #14 = NameAndType        #7:#8          // "<init>":()V
  #15 = Class              #22            // java/lang/System
  #16 = NameAndType        #23:#24        // out:Ljava/io/PrintStream;
  #17 = Utf8               synchronized 修饰 方法
  #18 = Class              #25            // java/io/PrintStream
  #19 = NameAndType        #26:#27        // println:(Ljava/lang/String;)V
  #20 = Utf8               com/zj/ideaprojects/demo/test2/SynchronizedTest1
  #21 = Utf8               java/lang/Object
  #22 = Utf8               java/lang/System
  #23 = Utf8               out
  #24 = Utf8               Ljava/io/PrintStream;
  #25 = Utf8               java/io/PrintStream
  #26 = Utf8               println
  #27 = Utf8               (Ljava/lang/String;)V
{
  public com.zj.ideaprojects.demo.test2.SynchronizedTest1();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
        line 3: 0

  public synchronized void test(); //需要注意的地方!
    descriptor: ()V
    flags: ACC_PUBLIC, ACC_SYNCHRONIZED //需要注意的地方!
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
         3: ldc           #3                  // String synchronized 修饰 方法
         5: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/Stri
         8: return
      LineNumberTable:
        line 5: 0
        line 6: 8
}
SourceFile: "SynchronizedTest1.java"
```

> 笔者已将字节码中 需要注意的地方，标注出来了

我们可以发现当`synchronized`修饰方法时，jvm往字节码中添加一个`ACC_SYNCHRONIZED标识符`，它的作用是：当一个线程访问方法时，会去检查是否存在`ACC_SYNCHRONIZED标识符`，如果存在，则先要获得对应的monitor锁，然后执行方法。当方法执行结束(不管是正常return还是抛出异常)都会释放对应的monitor锁。如果此时有其他线程也想要访问这个方法时，会因得不到`monitor锁`而阻塞。当同步方法中抛出异常且方法内没有捕获，则在向外抛出时会先释放已获得的`monitor锁`;大家看到这里就会有疑问：monitor锁是什么？

monitor锁是什么？

`monitor`字面意思就是"监视器"，也叫管程。它的出现是为了解决操作系统级别关于线程同步原语的使用复杂性，类似于语法糖，对复杂操作进行封装。在HotSpot源码中，Monitor是基于C++实现的，详情可见objectMonitor。每个对象中都内置了一个`ObjectMonitor`对象。

monitor的作用：**限制同一时刻，只有一个线程能进入monitor框定的临界区(持有权)，达到线程互斥，保护临界区中临界资源的安全，实现程序线程安全的目的**。它还具有管理进程，线程状态的功能，Java中Object类提供了notify和wait方法来对线程进行控制，其实它们都依赖于monitor对象，**这就是wait/notify等方法只能在同步的块或者方法中才能被调用的根本原因**

ObjectMonitor中有3个重要部分，分别为`_ower,_WaitSet和_EntryList`

1. 如果没有其他线程正在持有对象的Monitor，入口队列的"等待线程"需要和等待队列被唤醒的"等待线程"竞争（CPU调度）,选出一个线程来获取对象的Monitor，执行受保护的代码段，执行完毕后释放Monitor；**如果已经有线程持有对象的Monitor，那么需要等待其释放Monitor后再进行竞争**。
2. 如果一个线程是从等待队列中被notify()方法唤醒后(**调用notify方法，并不意味着释放了Monitor，必须要等同步代码块结束后才会释放Monitor)**，获取到的Monitor，它会去读取它自己先前保存的PC计数器中的地址，从它调用wait方法的地方继续执行

那monitor锁究竟是什么呢？呼噜噜查看了JVM官方文档中关于`Synchronization`的部分：`monitor锁`就是Java基于monitor机制的实现的**重量级锁**。在Java中，**每一个对象实例都会关联一个Monitor对象**,既可以与对象一起创建销毁，也可以在线程试图获取对象锁时自动生成。当这个Monitor对象被线程持有后，它便处于锁定状态；如果线程没有获取到monitor会被阻塞。

那么对象实例是怎么做到关联一个Monitor对象，一起创建销毁的？换句话说**如何通过java对象可以获取到和它对应的监视器**，这就需要我们去了解**Java对象内存布局**

Java对象内存布局

在JVM中，对象在内存中存储的布局可以分为三个区域，分别是对象头、实例数据以及填充数据。

- **对象头（object header）**：对象头又被分为两部分，分别为：Mark Word(标记字段)、Class Pointer(类型指针)。如果是数组，那么还会有数组长度。我们等会重点讨论
- **实例数据（Instance Data）**：主要是存放类的数据信息，父类的信息，对象字段属性信息。这部分内存按4字节对齐。
- **对齐填充（Padding）**：由于虚拟机要求对象起始地址必须是8字节的整数倍。填充数据不是必须存在的，仅仅是为了字节对齐。

对象头：主要有两部分，分别为：Mark Word和 Class Pointer

1. Class Pointer(类型指针)

即类型指针，是对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。

2. Mark Word(标记字段)

用于存储对象自身的运行时数据，如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等等。我们去看openjdk_1.8的源码，对应路径`/openjdk/hotspot/src/share/vm/oops`，Mark Word对应到C++的代码`markOop.hpp`，可以从注释中看到它们的组成

32位：

	23bit	2bit		是否偏向锁	索标志位
无锁态	对象的hashCode		分代年龄	0	01
偏向锁	线程ID	Epoch	分代年龄	1	01
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向互斥锁（重量级锁）的指针				10
GC标记	空				11

64位：

锁状态	56bit		1bit	4bit	1bit	2bit
					是否偏向锁	索标志位
无锁态	unused：25bit	对象的hashCode:31bit	unused	分代年龄	0	01
偏向锁	线程ID:54bit	Epoch：2bit	unused	分代年龄	1	01
轻量级锁	指向栈中锁记录的指针					00
重量级锁	指向互斥锁（重量级锁）的指针					10
GC标记	空					11

从上图中，我们发现当对象头中 `锁状态` 为**重量级锁时，对象头的MarkWord存储了指向堆中的Monitor对象的指针**。相信大家看的这里，会恍然大悟！

synchronized修饰代码块

以上文SynchronizedTest2类为例子，其中synchronized关键字修饰代码块

获取SynchronizedTest2.class的字节码：

```
javac -encoding utf-8 SynchronizedTest2.java
javap -c -v SynchronizedTest2.class

Classfile /D:/ideaProjects/src/main/java/com/zj/ideaprojects/demo/test2/SynchronizedTest2.class
  Last modified 2022-10-28; size 575 bytes
  MD5 checksum ac915d460a3da67f6c76c5ed2aae01f1
  Compiled from "SynchronizedTest2.java"
public class com.zj.ideaprojects.demo.test2.SynchronizedTest2
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #6.#18         // java/lang/Object."<init>":()V
   #2 = Fieldref           #19.#20        // java/lang/System.out:Ljava/io/PrintStream;
   #3 = String             #21            // synchronized ▒▒▒ ▒▒▒▒
   #4 = Methodref          #22.#23        // java/io/PrintStream.println:(Ljava/lang/String;)V
   #5 = Class              #24            // com/zj/ideaprojects/demo/test2/SynchronizedTest2
   #6 = Class              #25            // java/lang/Object
   #7 = Utf8               <init>
   #8 = Utf8               ()V
   #9 = Utf8               Code
  #10 = Utf8               LineNumberTable
  #11 = Utf8               test
  #12 = Utf8               StackMapTable
  #13 = Class              #24            // com/zj/ideaprojects/demo/test2/SynchronizedTest2
  #14 = Class              #25            // java/lang/Object
  #15 = Class              #26            // java/lang/Throwable
  #16 = Utf8               SourceFile
  #17 = Utf8               SynchronizedTest2.java
  #18 = NameAndType        #7:#8          // "<init>":()V
  #19 = Class              #27            // java/lang/System
  #20 = NameAndType        #28:#29        // out:Ljava/io/PrintStream;
  #21 = Utf8               synchronized ▒▒▒ ▒▒▒▒
  #22 = Class              #30            // java/io/PrintStream
  #23 = NameAndType        #31:#32        // println:(Ljava/lang/String;)V
  #24 = Utf8               com/zj/ideaprojects/demo/test2/SynchronizedTest2
  #25 = Utf8               java/lang/Object
  #26 = Utf8               java/lang/Throwable
  #27 = Utf8               java/lang/System
  #28 = Utf8               out
  #29 = Utf8               Ljava/io/PrintStream;
  #30 = Utf8               java/io/PrintStream
  #31 = Utf8               println
  #32 = Utf8               (Ljava/lang/String;)V
{
  public com.zj.ideaprojects.demo.test2.SynchronizedTest2();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
        line 3: 0

  public void test();
    descriptor: ()V
    flags: ACC_PUBLIC  //注意这里!
    Code:
      stack=2, locals=3, args_size=1
         0: aload_0
         1: dup
         2: astore_1
         3: monitorenter     //注意这里!
         4: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
         7: ldc           #3                  // String synchronized ▒▒▒ ▒▒▒▒
         9: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/Stri
        12: aload_1
        13: monitorexit     //注意这里!
        14: goto          22
        17: astore_2
        18: aload_1
        19: monitorexit    //注意这里!
        20: aload_2
        21: athrow
        22: return
      Exception table:
         from    to  target type
```

```
            4    14    17   any
           17    20    17   any
      LineNumberTable:
        line 5: 0
        line 6: 4
        line 7: 12
        line 8: 22
      StackMapTable: number_of_entries = 2
        frame_type = 255 /* full_frame */
          offset_delta = 17
          locals = [ class com/zj/ideaprojects/demo/test2/SynchronizedTest2, class java/lang/Object
          stack = [ class java/lang/Throwable ]
        frame_type = 250 /* chop */
          offset_delta = 4
}
SourceFile: "SynchronizedTest2.java"
```

我们可以发现：synchronized 同步语句块的在字节码中的实现，是使用了 monitorenter 和 monitorexit 指令，其中 monitorenter 指令指向同步代码块的开始位置，monitorexit 指令则指明同步代码块的结束位置。

1. 每个对象都拥有一个monitor，当monitor被占用时，就会处于锁定状态，线程执行monitorenter指令时会获取monitor的所有权。
2. 当monitor计数为0时，说明该monitor还未被锁定，此时线程会进入monitor并将monitor的计数器设为1，并且该线程就是monitor的所有者。如果此线程已经获取到了monitor锁，再重新进入monitor锁的话，那么会将计时器count的值加1。
3. 如果有线程已经占用了monitor锁，此时有其他的线程来获取锁，那么此线程将进入阻塞状态，待monitor的计时器count变为0，这个线程才会获取到monitor锁。
4. 只有拿到了monitor锁对象的线程才能执行monitorexit指令。在执行 monitorexit 指令后，将锁计数器设为 0，表明锁被释放，其他线程可以尝试获取锁。
5. 如果获取对象锁失败，那当前线程就要阻塞等待，直到锁被另外一个线程释放为止

有个奇怪的现象不知道大家有没有发现？为什么 `monitorenter指令` 只出现了一次，但是 `monitorexit指令` 却出现了2次？

> 因为编译器必须保证无论同步代码块中的代码以何种方式结束，代码中每次调用monitorenter必须执行对应的monitorexit指令。如果没有执行 monitorexit指令，monitor一直被占用，其他线程都无法获取，这是非常危险的。
>
> 这个就很像"try catch finally"中的 `finally` ，不管程序运行结果如何，必须要执行 `monitorexit指令` ，释放monitor所有权

小结一下：

1. 同步代码块是通过monitorenter和monitorexit指令来实现；同步方式是通过方法中的access_flags中设置ACC_SYNCHRONIZED标识符来实现，ACC_SYNCHRONIZED标识符会去隐式调用这两个指令：monitorenter和monitorexit
2. synchronized修饰方法、修饰代码块，归根到底，都是通过**竞争monitor所有权**来实现同步的
3. 每个java对象都会与一个monitor相关联，可以由线程获取和释放
4. monitor通过维护一个计数器来记录锁的获取，重入，释放情况

锁优化

为什么说JDK早期，Synchronized是重量级锁呢？在JVM中monitorenter和monitorexit字节码依赖于底层的操作系统的 `Mutex Lock` 来实现的，但是由于使用 `Mutex Lock` 需要将**当前线程挂起并从用户态切换到内核态来申请锁资源，还需要经过一个中断的调用，申请完之后还需要从内核态返回到用户态**。整个切换过程是非常消耗资源的，如果程序中存在大量的锁竞争，那么会引起程序频繁的在用户态和内核态进行切换，严重影响到程序的性能。

> 在Linux系统架构中可以分为用户空间和内核，我们的程序都运行在用户空间，进入用户运行状态就是所谓的用户态。在用户态可能会涉及到某些操作如I/O调用，就会进入内核中运行，此时进程就被称为内核运行态，简称内核态。
>
> 1. **内核：** 本质上可以理解为一种软件，控制计算机的硬件资源，并提供上层应用程序运行的环境。
> 2. **用户空间：** 上层应用程序活动的空间。应用程序的执行必须依托于内核提供的资源，包括CPU资源、存储资源、I/O资源等。
> 3. **系统调用：** 为了使上层应用能够访问到这些资源，内核必须为上层应用提供访问的接口：即系统调用。

为了解决这一问题，在JDK1.6对Synchronized进行大量的优化，锁自旋、锁粗化、锁消除，锁膨胀等技术，在这部分扩展内容比较多，我们接下来一一道来。

自旋锁

在jdk1.6前多线程竞争锁时，当一个线程A获取锁时，它会阻塞其他所有正在竞争的线程，这样对性能带来了极大的影响。在挂起线程和恢复线程的操作都需要转入内核态中完成，这些操作对系统的并发性能带来了很大的压力。由于在实际环境中，**很多线程的锁定状态只会持续很短的一段时间，会很快释放锁**，为了如此短暂的时间去挂起和阻塞其他所有竞争锁的线程，是非常浪费资源的，我们完全可以让另一个没有获取到锁的线程在门外等待一会(自旋)，但**不放弃CPU的执行时间**，等待持有锁的线程A释放锁,就里面去获得锁。这其实就是**自旋锁**

但是我们也无法保证线程获取锁之后，就一定很快释放锁。万一遇到有线程，长时间不释放锁，其会带来更多的性能开销。因为在线程自旋时，始终会占用CPU的时间片，如果锁占用的时间太长，那么自旋的线程会消耗掉CPU资源。**所以我们需要对锁自旋的次数有所限制，如果自旋超过了限定的次数仍然没有成功获取到锁，就应该重新使用传统的方式去挂起线程了**。在JDK定义中，自旋锁默认的自旋次数为10次，用户可以使用参数`-XX:PreBlockSpin`来更改。

后来也有改进型的**自适应自旋锁**，自适应意味着自旋的次数不在固定，而是由前一次在同一个锁上的自旋时间和锁的拥有者的状态共同决定。如果在同一个锁对象上，自旋等待刚刚成功获得过锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也是很可能再次成功的，进而它将会允许线程自旋相对更长的时间。如果对于某个锁，线程很少成功获得过，则会相应减少自旋的时间甚至直接进入阻塞的状态，避免浪费处理器资源。笔者感觉这个跟CPU的分支预测，有异曲同工之妙

锁粗化

一般来说，同步块的作用范围应该尽可能小，缩短阻塞时间，如果存在锁竞争，那么等待锁的线程也能尽快获取锁但某些情况下，可能会对同一个锁频繁访问，或者有人在循环里面写上了synchronized关键字，为了降低短时间内大量的锁请求、释放带来的性能损耗，Java虚拟机发现了之后会**适当扩大加锁的范围,以避免频繁的拿锁释放锁的过程**。将多个锁请求合并为一个请求,这就是**锁粗化**

```
public class LockCoarseningTest {
 public String test() {
  StringBuffer sb = new StringBuffer();
  for(int i = 0; i < 100; i++) {
   sb.append("test");
  }
  return sb.toString();
 }
}
```

append() 为同步方法，短时间内大量进行锁请求、锁释放，JVM 会自动进行锁粗化，将加锁范围扩大至 for 循环外部，从而只需要进行一次锁请求、锁释放

锁消除

锁消除：通过运行时JIT编译器的逃逸分析来消除一些没有在当前同步块以外被其他线程共享的数据的锁保护，通过逃逸分析也可以在线程本的Stack上进行对象空间的分配(同时还可以减少Heap上的垃圾收集开销)。其实就是即时编译器通过对运行上下文的扫描，对不可能存在共享资源竞争的锁进行消除，从而节约大量的资源开销，提高效率

```
public class LockEliminateTest {
 static int i = 0;

 public void method1() {
  i++;
 }

 public void method2() {
  Object obj = new Object();
  synchronized (obj) {
   i++;
  }
 }
}
```

method2() 方法中的 obj 为局部变量，显然不可能被共享，对其加锁也毫无意义，故被即时编译器消除

锁膨胀

锁膨胀方向：`无锁 → 偏向锁 → 轻量级锁 → 重量级锁`偏向锁、轻量级锁，这两个锁既是一种优化策略，也是一种膨胀过程，接下来我们分别聊聊

偏向锁

在大多数情况下虽然加了锁，但是没有锁竞争的发生，甚至是同一个线程反复获得这个锁，那么多次的获取锁和释放锁会带来很多不必要的性能开销和上下文切换。偏向锁就为了针对这种情况而出现的

偏向锁指，**锁偏向于第一个获取他的线程**，若接下来的执行过程中，该锁一直没有被其他线程获取，则持有偏向锁的线程永远不需要再进行同步。**这样就在无锁竞争的情况下避免在锁获取过程中执行不必要的获取锁和释放锁操作**。

偏向锁的具体过程:

1. 首先JVM要设置为可用偏向锁。然后当一个进程访问同步块并且获得锁的时候，会在对象头和栈帧的锁记录里面存储取得偏向锁的线程ID。
2. 等下一次有线程尝试获取锁的时候，首先检查这个对象头的MarkWord是不是储存着这个线程的ID。如果是，那么直接进去而不需要任何别的操作。
3. 如果不是，那么分为两种情况：

> - 对象的偏向锁标志位为0（当前不是偏向锁），说明发生了竞争，已经膨胀为轻量级锁，这时使用CAS操作尝试获得锁。
> - 偏向锁标志位为1，说明还是偏向锁不过请求的线程不是原来那个了。这时只需要使用CAS尝试把对象头偏向锁从原来那个线程指向目前求锁的线程。

轻量级锁

在实际情况中，大部分的锁，在整个同步生命周期内都不存在竞争，在无锁竞争的情况下完全可以避免调用操作系统层面的**重量级互斥锁，**可以通过**CAS原子指令**就可以完成锁的获取及释放。当存在锁竞争的情况下，执行CAS指令失败的线程将调用操作系统互斥锁进入到阻塞状态，当锁被释放的时候被唤醒。当升级为轻量级锁之后，`MarkWord`的结构也会随之变为轻量级锁结构。JVM会利用CAS尝试把对象原本的`MarkWord`更新为`Lock Record`的指针，成功就说明加锁成功，改变锁标志位为00，然后执行相关同步操作。轻量级锁所适应的场景是**线程交替执行同步块的场合**，如果存在同一时间访问同一锁的场合，就会导致轻量级锁就会失效，进而膨胀为重量级锁。

> **CAS**（Compare-And-Swap）：顾名思义**比较并替换**。这是一个由CPU硬件提供并实现的原子操作.可以被认为是一种**乐观锁**，会以一种更加乐观的态度对待事情，认为自己可以操作成功。当多个线程操作同一个共享资源时，仅能有一个线程同一时间获得锁成功，在乐观锁中，其他线程发现自己无法成功获得锁，并不会像悲观锁那样阻塞线程，而是直接返回，可以去选择再次重试获得锁，也可以直接退出
>
> CAS机制所保证的只是一**个变量**的原子性操作，无法保证整个代码块的原子性

最后再小结一下，锁的优缺点对比：

锁	优点	缺点	使用场景
偏向锁	加锁和解锁不需要CAS操作，没有额外的性能消耗，和执行非同步方法相比仅存在纳秒级的差距	如果线程间存在锁竞争，会带来额外的锁撤销的消耗	适用于只有一个线程访问同步块的场景
轻量级锁	竞争的线程不会阻塞，提高了响应速度	如线程成始终得不到锁竞争的线程，使用自旋会消耗CPU性能	追求响应时间，同步块执行速度非常快
重量级锁	线程竞争不适用自旋，不会消耗CPU	线程阻塞，响应时间缓慢，在多线程下，频繁的获取释放锁，会带来巨大的性能消耗	追求吞吐量，同步块执行速度较长

最高效的是偏向锁，尽量使用偏向锁，如果不能（发生了竞争）就膨胀为轻量级锁，当发生锁竞争时，轻量级锁的CAS操作会自动失效，锁再次膨胀为重量级锁。**锁一般是只能升级但不能降级**，这种锁升级却不能降级的策略，目的是**为了提高获得锁和释放锁的效率。（**hotspot其实是可以发生锁降级的，但触发锁降级的条件比较苛刻**）**

偏向锁，轻量级锁，只需在用户态就可以实现，而不需要进行用户态和内核态之间的切换

经过如此多的锁优化，如今的 synchronized 锁效率非常不错，目前不论是各种开源框架还是 JDK 源码都大量使用了 synchronized 关键字。

synchronized关键字实现单例模式

我们来看一个经典的例子，利用 `synchronized关键字` 实现单例模式

```
/**
 * 懒汉 - 双层校验锁
 */
public class SingleDoubleCheck {
    private static SingleDoubleCheck instance = null;

    private SingleDoubleCheck(){}//将构造器 私有化，防止外部调用

    public static SingleDoubleCheck getInstance() {
        if (instance == null) { //part 1
            synchronized (SingleDoubleCheck.class) {
                if (instance == null) { //part 2
                    instance = new SingleDoubleCheck();//part 3
                }
            }
        }
        return instance;
    }
}
```

> 对单例模式感兴趣的话，见拓展：https://mp.weixin.qq.com/s/TyiCfVMeeDwa-2hd9N9XJQ

synchronized 和 volatile 的区别？

synchronized 关键字和 volatile 关键字是两个互补的存在，而不是对立的存在

1. volatile 关键字是线程同步的轻量级实现，所以 volatile性能肯定比synchronized关键字要好 。但是 volatile 关键字只能用于变量而 synchronized 关键字可以修饰方法以及代码块 。
2. volatile 关键字能保证数据的可见性，但不能保证数据的原子性。synchronized 关键字两者都能保证。
3. volatile关键字主要用于解决变量在多个线程之间的可见性，而 synchronized 关键字解决的是多个线程之间访问资源的同步性。
4. volatile只能修饰实例变量和类变量，而synchronized可以修饰方法，以及代码块。

尾语

本文拓展内容确实有点多，很开心你能看到最后，我们再简明地回顾一下synchronized 的特性

1. 原子性：确保线程互斥的访问同步代码。synchronized保证只有一个线程拿到锁，进入同步代码块操作共享资源，因此具有原子性。

2. 可见性：保证共享变量的修改能够及时可见。当某线程进入synchronized代码块前后，线程会获得锁，清空工作内存，从主内存拷贝共享变量最新的值到工作内存成为副本，执行代码，将修改后的副本的值刷新回主内存中，线程释放锁。其他获取不到锁的线程会阻塞等待，所以变量的值一直都是最新的。

3. 有序性：synchronized内的代码和外部的代码禁止排序，至于内部的代码，则不会禁止排序，但是由于只有一个线程进入同步代码块，因此在同步代码块中相当于是单线程的，根据 as-if-serial 语义，即使代码块内发生了重排序，也不会影响程序执行的结果。

4. 悲观锁：synchronized是悲观锁。每次使用共享资源时都认为会和其他线程产生竞争，所以每次使用共享资源都会上锁。

5. 独占锁（排他锁）：synchronized是独占锁（排他锁）。该锁一次只能被一个线程所持有，其他线程被阻塞。

6. 非公平锁：synchronized是非公平锁。线程获取锁的顺序可以不按照线程的阻塞顺序。允许新来的线程有可能立即获得监视器，而在等待区中等候已久的线程可能再次等待。这样有利于提高性能，但是也可能会导致饥饿现象

7. 可重入锁：synchronized是可重入锁。持锁线程可以再次获取自己的内部的锁，可一定程度避免死锁。

参考资料：

https://openjdk.org/groups/hotspot/docs/HotSpotGlossary.html

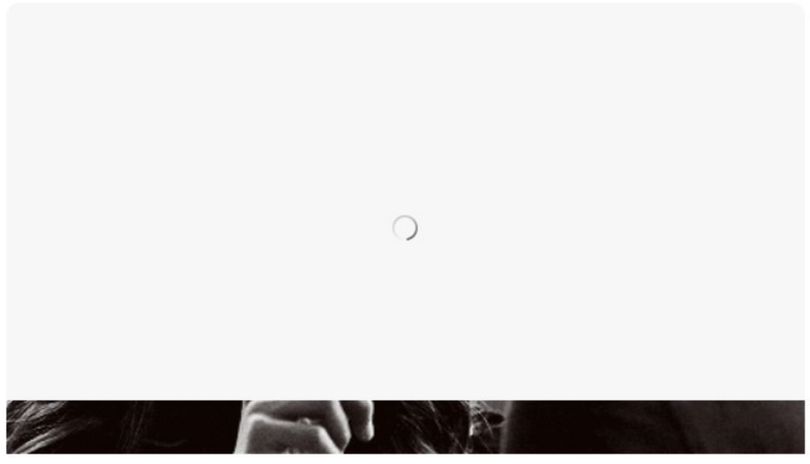
《深入理解java虚拟机》

《Java并发编程的艺术》

https://www.cnblogs.com/qingshan-tang/p/12698705.html

https://www.cnblogs.com/jajian/p/13681781.html

您的在看和点赞对我真的很重要!



JAVA旭阳

旭阳，希望自己能像初升的太阳一样，对任何事情充满希望~~

142篇原创内容

公众号

Read more

People who liked this content also liked

redis九大数据类型及场景案例实现

好好学技术

Hystrix Dashboard图形化监控

一只IT攻城狮

正则系列之RegExp使用正则表达式

前端老L