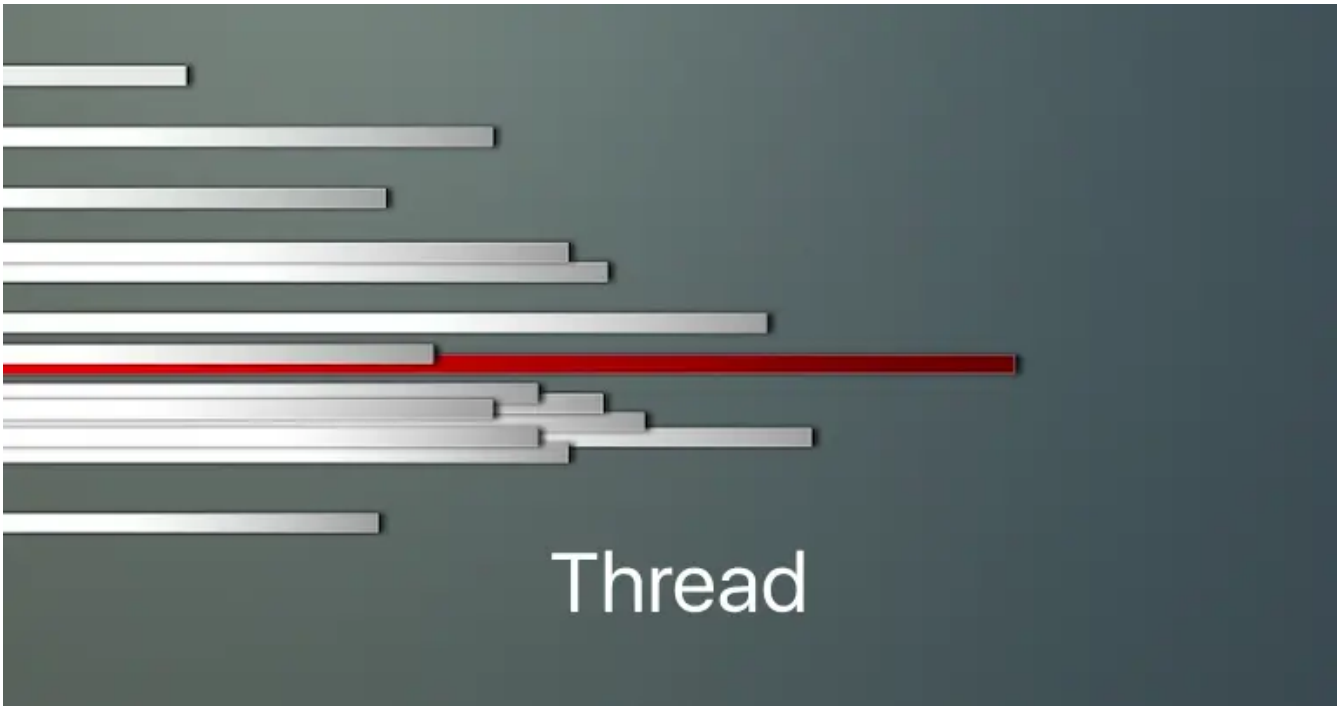
Java线程内存模型

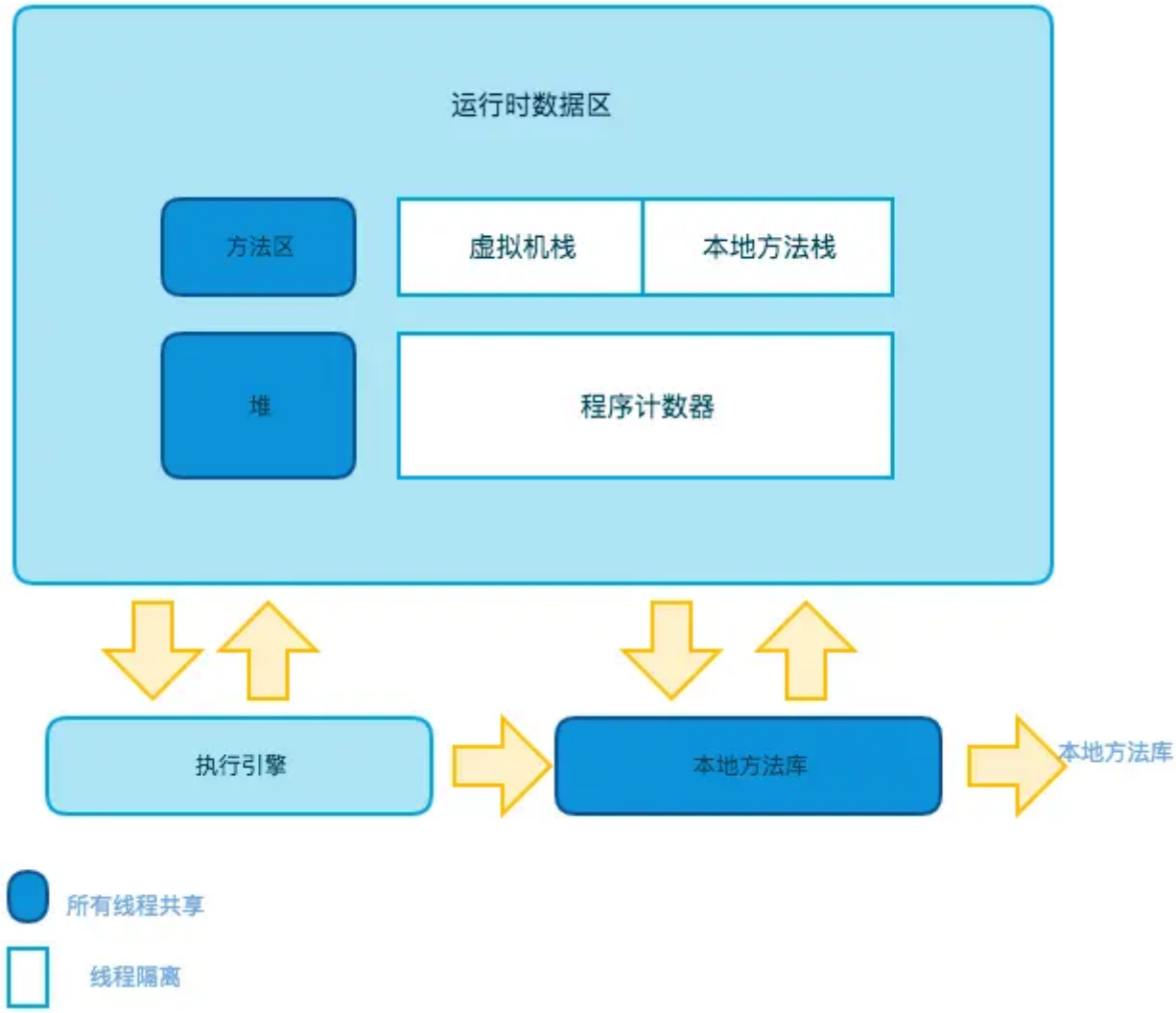
MaxZing 关注 IP属地: 上海

2017.11.07 17:35:22 字数 1,817 阅读 1,552



知道JVM内存模型可以在学习多线程的时候更加了解锁的机制和工作方式。下面是我的学习笔记，比较初级。

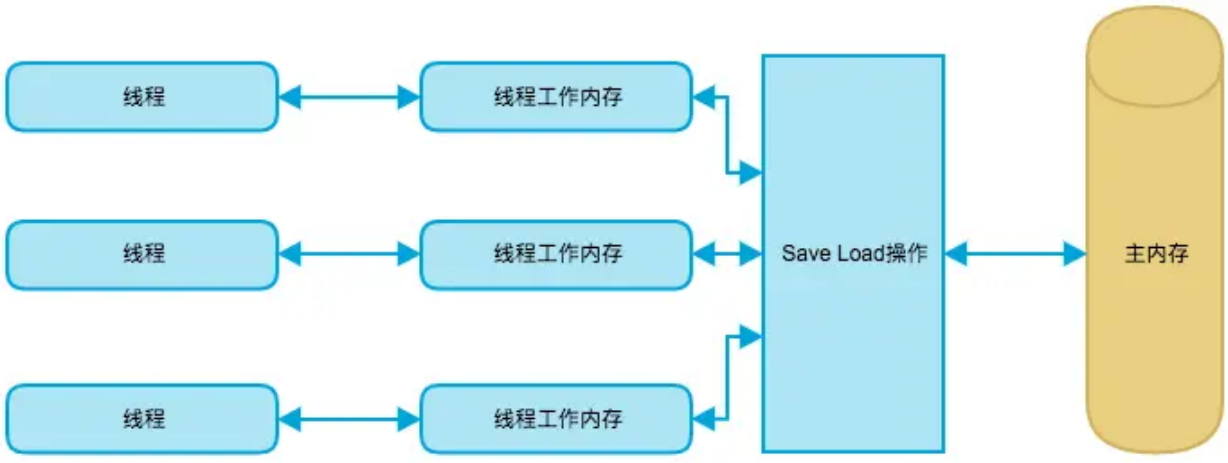
0x01 内存模型图的思维转换



JVM定义了Java的虚拟内存模型，跟C/C++不一样的是，虚拟内存将物理内存划分了不同的区域，而C/C++是直接映射物理内存的。

笼统的来说，jvm一般将内存分为栈和堆，栈用来存储静态方法和静态变量，而堆用来存储对象和普通变量。

但是如果从线程的角度，内存模型会变成下图的样子



在这个模型中，变量是在主内存中的，线程各自有各自的工作内存，不会出现相互干扰。

工作内存通过跟主内存之间的操作，实现变量数据的交换共享。而线程工作内存是相互隔离的。这样各线程工作的时候不会对其他线程的工作数据产生影响

0x02 工作内存和主内存之间的操作（单个线程）

两块内存有8种操作。

1. **(lock - unlock)** lock将一个主内存变量标记成线程独占，unlock将独占的变量释放
2. **(read - load)** read 将主内存的变量读取到CPU中，load操作将read到的变量存入到工作内存中，一定会成对出现
3. **(use - assign)** use将工作内存中的变量传递给执行的代码中，当代码需要使用变量值的字节码时，需要这个操作。 assign 赋值操作，将代码中赋值指令出现时，把收到的变量赋值到工作内存中
4. **(store - write)** store 将工作内存的变量传送回主内存，但是只是传送，write操作才会将值写入到主内存。而且这两个一定会成对出现

> - read load ; store write只能成对操作，不能出现只读不用，只返回不存储
> - 不允许线程丢弃assign操作，用完的变量一定会传回主内存，也不允许将未assign的变量从工作内存写回主内存
> - 变量只能从主内存中创建，未初始化的变量线程不能load 或 assign
> - 变量只能被一条线程lock，而且可以lock很多次，必须执行相应条数的unlock才会被释放
> - 线程只能unlock自己lock的变量，未被lock的变量不能执行unlock，不允许unlock其他线程lock的变量
> - lock变量操作会将工作内存的变量清空，使用这个变量时，相应使用这个变量时，需要重新load 和assign
> - unlock 前，变量会被重新写入主内存

正如上面4条说明，每两对操作基本都是对应的，成对出现的。（不是绝对）

0x03 特殊的内存操作

下面说一下特例情况

synchronized对内存可见性的影响

首先synchronized关键字在线程同步上，安全性几乎是万能的，导致被滥用的一塌糊涂。
其次synchronized关键字保证了两条线程遵循happens-before的设计原则，两个线程必须一先一后执行。

> 它会阻止其它线程获取当前对象锁，这样就使得当前对象中被synchronized关键字保护的代码块无法被其它线程访
>
> volatile和synchronized的区别

最后synchronized关键字对内存的影响是，当一个线程从一个synchronized块出来时，内存一定会刷新成最新的数据，保证变量的可见性。

接着是volatile关键字

volatile 修饰的变量。面试容易被问，因为大家喜欢用synchronized 来进行同步，忽略了了volatile这个关键字

那么他们的区别是什么？
“可见性”！但是一定要记住，这里的可见性不是绝对的立即可见，虽然volatile变量会将变量的变化及时的反应到其他线程的工作内存中去，但是并不表示被修饰的变量在各线程中会一致，只是使用的时候会被刷新，所以执行引擎不会发现不一致的情况，而实际上，被修饰的变量也会有不同的工作内存中值不一致的情况

volatile 关键字只能保证线程取值的时候，是一致的，等取到工作内存中进行操作时，如果变量被其他线程存回去的话，这时候工作内存的变量就会出现不一致。
下面一段代码会出现比预期值小的情况

```
public class TestVolatail {
    public static volatile int added = 0;

    public static void increase() {
        added++;//++ 操作是先取出来，然后再加一，如果想线程安全，试试juc里面的atomic包下的类
    }

    public static final int ADD_ACCOUNTER = 10;
```

```
        public static void main(String[] args) {
            Thread t[] = new Thread[ADD_ACCOUNTER];
            for (int i = 0; i < ADD_ACCOUNTER; i++) {
                t[i] = new Thread(() -> {
                    for (int m = 0; m < 10000; m++) {
                        increase();
                    }
                });
                t[i].start();
            }

            while (Thread.activeCount() > 1) {
                Thread.yield();
            }

            System.out.println("最终累加值: " + added);
        }
    }
```

输出结果：

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
objc[42341]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualM
Connected to the target VM, address: '127.0.0.1:56136', transport: 'socket'
最终累加值: 99172
Disconnected from the target VM, address: '127.0.0.1:56136', transport: 'socket'

Process finished with exit code 0
```

10个线程，每个对变量做10000次累加，并不是100000，变小了是因为某些线程正在累加的时候，新的较小的值已经覆盖了正在使用的变量，别的线程来取会以这个较小的值为准。

> **所以volatile只能保证可见性，不能保证一致性**

> 面试的时候最好提一下volatile会禁止jvm做指令重排序，java1.5之后才变成这样的，本篇暂且不表，坑后面再填

long 和 double的特殊性

首先 long 和 double是64位的，这一点是毋庸置疑的。

所以上述8种操作，在面对long类型和double类型的时候，有4种操作`read load store write`是需要连续执行两次的。虽然JVM中，虽然是允许读取64位数据不是原子操作，但是在商用的JVM中，这些操作都是原子的

而且JVM规范里，强烈“建议”将读取和存放 64位数据的操作做成原子操作的。所以……
另外，多核心处理器中的同一个变量缓存也不是及时同步的，即使有个byte，使用时也会加锁。

附上JVM规范和R大在知乎上的回答：

> JVM规范：17.7. Non-atomic Treatment of double and long

> R大的回答截取： *简单说就是规范说的是：*
>
> > - 实现对普通long与double的读写不要求是原子的（但如果实现为原子操作也OK）
> > - 实现对volatile long与volatile double的读写必须是原子的（没有选择余地）
>
> (*64位JVM的long和double读写也不是原子操作么？ - RednaxelaFX的回答 - 知乎 https://www.zhihu.com/question/38816432/answer/78944479*)

如果想亲自试一试，可以看看：证明32位java对long和double的写操作不是原子性的

0x04 总结
以上只是对内存结构和部分读写规则的总结笔记，线程是java必过的硬指标，后面再更新新的学习笔记。

---

by：zing
https://micorochio.github.io/2017/11/10/the_thread_RAM_modle_in_jvm/