内存模型

1. java 内存模型

很多人将【java 内存结构】与【java 内存模型】傻傻分不清，【java 内存模型】是 Java Memory Model（JMM）的意思。

关于它的权威解释，请参考 https://download.oracle.com/otn-pub/jcp/memory_model-1.0-pfd-spec-oth-JSpec/memory_model-1_0-pfd-spec.pdf?AuthParam=1562811549_4d4994cbd5b59d964cd2907ea22ca08b

简单的说，JMM 定义了一套在多线程读写共享数据时（成员变量、数组）时，对数据的可见性、有序性、和原子性的规则和保障

1.1 原子性

原子性在学习线程时讲过，下面来个例子简单回顾一下：

问题提出，两个线程对初始值为 0 的静态变量一个做自增，一个做自减，各做 5000 次，结果是 0 吗?

1.2 问题分析

以上的结果可能是正数、负数、零。为什么呢？因为 Java 中对静态变量的自增，自减并不是原子操作。

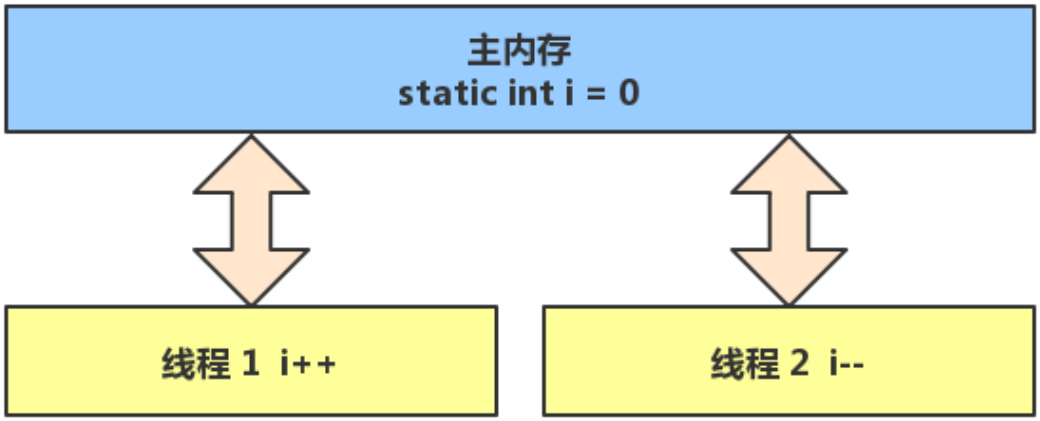
例如对于 `i++` 而言（i 为静态变量），实际会产生如下的 JVM 字节码指令：

```
getstatic     i  // 获取静态变量i的值
iconst_1         // 准备常量1
iadd             // 加法
putstatic     i  // 将修改后的值存入静态变量i
```

而对应 `i--` 也是类似：

```
getstatic     i  // 获取静态变量i的值
iconst_1         // 准备常量1
isub             // 减法
putstatic     i  // 将修改后的值存入静态变量i
```

而 Java 的内存模型如下，完成静态变量的自增，自减需要在主存和线程内存中进行数据交换：



如果是单线程以上 8 行代码是顺序执行（不会交错）没有问题：

```
// 假设i的初始值为0
getstatic     i  // 线程1-获取静态变量i的值 线程内i=0
iconst_1         // 线程1-准备常量1
iadd             // 线程1-自增 线程内i=1
putstatic     i  // 线程1-将修改后的值存入静态变量i 静态变量i=1
getstatic     i  // 线程1-获取静态变量i的值 线程内i=1
iconst_1         // 线程1-准备常量1
isub             // 线程1-自减 线程内i=0
putstatic     i  // 线程1-将修改后的值存入静态变量i 静态变量i=0
```

但多线程下这 8 行代码可能交错运行（为什么会交错？思考一下）：
出现负数的情况：

```
// 假设i的初始值为0
getstatic     i  // 线程1-获取静态变量i的值 线程内i=0
getstatic     i  // 线程2-获取静态变量i的值 线程内i=0
iconst_1         // 线程1-准备常量1
iadd             // 线程1-自增 线程内i=1
putstatic     i  // 线程1-将修改后的值存入静态变量i 静态变量i=1
iconst_1         // 线程2-准备常量1
isub             // 线程2-自减 线程内i=-1
putstatic     i  // 线程2-将修改后的值存入静态变量i 静态变量i=-1
```

出现正数的情况：

```
// 假设i的初始值为0
getstatic     i  // 线程1-获取静态变量i的值 线程内i=0
getstatic     i  // 线程2-获取静态变量i的值 线程内i=0
iconst_1         // 线程1-准备常量1
iadd             // 线程1-自增 线程内i=1
iconst_1         // 线程2-准备常量1
isub             // 线程2-自减 线程内i=-1
putstatic     i  // 线程2-将修改后的值存入静态变量i 静态变量i=-1
putstatic     i  // 线程1-将修改后的值存入静态变量i 静态变量i=1
```

1.3 解决方法

`synchronized`（同步关键字）

语法

```
synchronized( 对象 ) {
    要作为原子操作代码
}
```

用 `synchronized` 解决并发问题：

```
static int i = 0;
static Object obj = new Object();

public static void main(String[] args) throws InterruptedException {
    Thread t1 = new Thread(() -> {
        for (int j = 0; j < 5000; j++) {
            synchronized (obj) {
                i++;
            }
        }
    });

    Thread t2 = new Thread(() -> {
        for (int j = 0; j < 5000; j++) {
            synchronized (obj) {
                i--;
            }
        }
    });
    t1.start();
    t2.start();

    t1.join();
    t2.join();
    System.out.println(i);
}
```

如何理解呢：你可以把 obj 想象成一个房间，线程 t1，t2 想象成两个人。

当线程 t1 执行到 synchronized(obj) 时就好比 t1 进入了这个房间，并反手锁住了门，在门内执行 count++ 代码。

这时候如果 t2 也运行到了 synchronized(obj) 时，它发现门被锁住了，只能在门外等待。

当 t1 执行完 synchronized{} 块内的代码，这时候才会解开门上的锁，从 obj 房间出来。t2 线程这时才可以进入 obj 房间，反锁住门，执行它的 count-- 代码。

> 注意：上例中 t1 和 t2 线程必须用 synchronized 锁住同一个 obj 对象，如果 t1 锁住的是 m1 对象，t2 锁住的是 m2 对象，就好比两个人分别进入了两个不同的房间，没法起到同步的效果。

2. 可见性

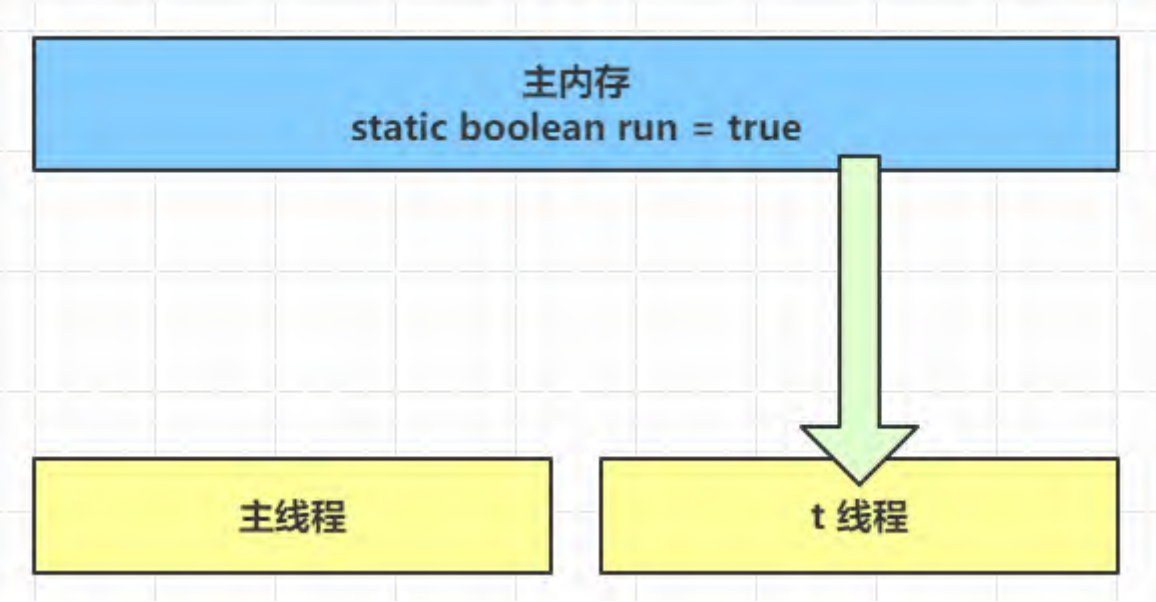
2.1 退不出的循环

先来看一个现象，main 线程对 run 变量的修改对于 t 线程不可见，导致了 t 线程无法停止：

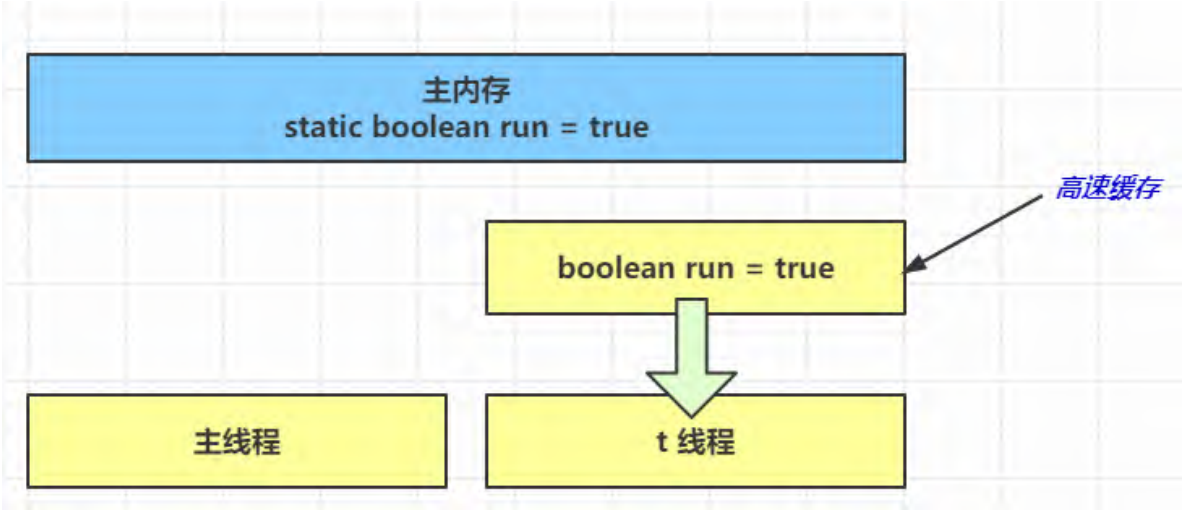
```
static boolean run = true;

public static void main(String[] args) throws InterruptedException {
    Thread t = new Thread(()->{
        while(run){
            // ....
        }
    });
    t.start();

    Thread.sleep(1000);
    run = false; // 线程t不会如预想的停下来
}
```

为什么呢？分析一下：

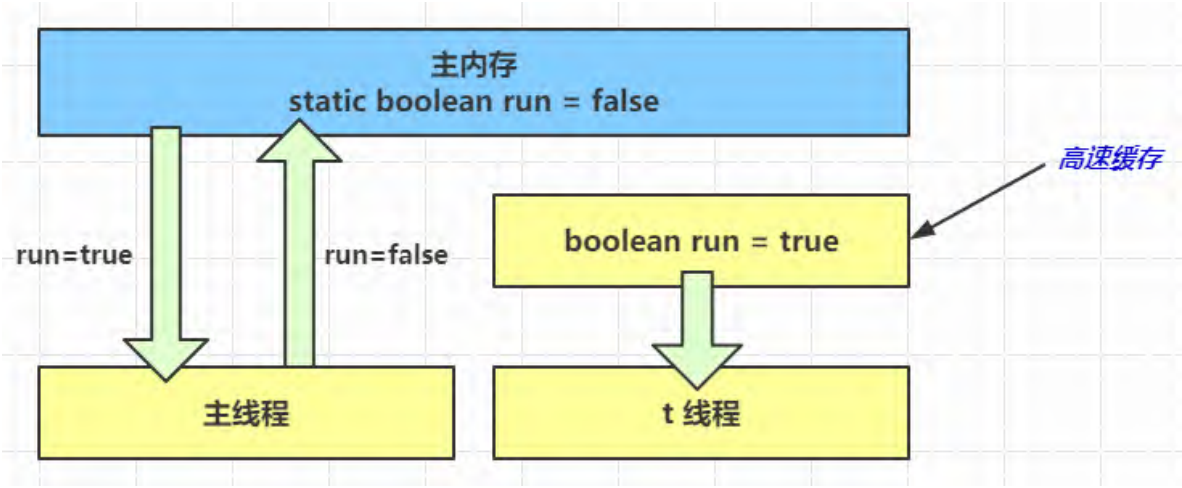
1. 初始状态， t 线程刚开始从主内存读取了 run 的值到工作内存。



2. 因为 t 线程要频繁从主内存中读取 run 的值，JIT 编译器会将 run 的值缓存至自己工作内存中的高速缓存中，减少对主存中 run 的访问，提高效率



3. 1 秒之后，main 线程修改了 run 的值，并同步至主存，而 t 是从自己工作内存中的高速缓存中读取这个变量的值，结果永远是旧值



2.2 解决方法

volatile（易变关键字）

它可以用来修饰成员变量和静态成员变量，他可以避免线程从自己的工作缓存中查找变量的值，必须到主存中获取它的值，线程操作 volatile 变量都是直接操作主存

2.3 可见性

前面例子体现的实际就是可见性，它保证的是在多个线程之间，一个线程对 volatile 变量的修改对另一个线程可见， 不能保证原子性，仅用在一个写线程，多个读线程的情况：
上例从字节码理解是这样的：

```
getstatic     run   // 线程 t 获取 run true
getstatic     run   // 线程 t 获取 run true
getstatic     run   // 线程 t 获取 run true
getstatic     run   // 线程 t 获取 run true
putstatic     run  //  线程 main 修改 run 为 false,  仅此一次
getstatic     run   // 线程 t 获取 run false
```

比较一下之前我们将线程安全时举的例子：两个线程一个 i++ 一个 i-- ，只能保证看到最新值，不能解决指令交错

```
// 假设i的初始值为0
getstatic     i  // 线程1-获取静态变量i的值 线程内i=0
getstatic     i  // 线程2-获取静态变量i的值 线程内i=0
iconst_1         // 线程1-准备常量1
iadd             // 线程1-自增 线程内i=1
putstatic     i  // 线程1-将修改后的值存入静态变量i 静态变量i=1
iconst_1         // 线程2-准备常量1
isub             // 线程2-自减 线程内i=-1
putstatic     i  // 线程2-将修改后的值存入静态变量i 静态变量i=-1
```

> **注意**
> synchronized 语句块既可以保证代码块的原子性，也同时保证代码块内变量的可见性。但缺点是synchronized是属于重量级操作，性能相对更低
>
> 如果在前面示例的死循环中加入 System.out.println() 会发现即使不加 volatile 修饰符，线程 t 也能正确看到对 run 变量的修改了，想一想为什么?

3. 有序性

3.1 诡异的结果

```java
int num = 0;
boolean ready = false;

// 线程1 执行此方法
public void actor1(I_Result r) {
    if(ready) {
        r.r1 = num + num;
    } else {
        r.r1 = 1;
    }
}

// 线程2 执行此方法
public void actor2(I_Result r) {
    num = 2;
    ready = true;
}
```

I_Result 是一个对象，有一个属性 r1 用来保存结果，问，可能的结果有几种?

有同学这么分析

情况1：线程1 先执行，这时 ready = false，所以进入 else 分支结果为 1

情况2：线程2 先执行 num = 2，但没来得及执行 ready = true，线程1 执行，还是进入 else 分支，结果为1

情况3：线程2 执行到 ready = true，线程1 执行，这回进入 if 分支，结果为 4（因为 num 已经执行过了）

但我告诉你，结果还有可能是 0 😁😁😁，信不信吧!

这种情况下是：线程2 执行 ready = true，切换到线程1，进入 if 分支，相加为 0，再切回线程2 执行 num = 2

相信很多人已经晕了 😳😳😳

这种现象叫做指令重排，是 JIT 编译器在运行时的一些优化，这个现象需要通过大量测试才能复现：

借助 java 并发压测工具 jcstress https://wiki.openjdk.java.net/display/CodeTools/jcstress

```
mvn archetype:generate  -DinteractiveMode=false -
DarchetypeGroupId=org.openjdk.jcstress -DarchetypeArtifactId=jcstress-java-test-
archetype -DgroupId=org.sample -DartifactId=test -Dversion=1.0
```

创建 maven 项目，提供如下测试类

```
@JCStressTest
@Outcome(id = {"1", "4"}, expect = Expect.ACCEPTABLE, desc = "ok")
@Outcome(id = "0", expect = Expect.ACCEPTABLE_INTERESTING, desc = "!!!!")
@State
public class ConcurrencyTest {

    int num = 0;
    boolean ready = false;
    @Actor
    public void actor1(I_Result r) {
        if(ready) {
            r.r1 = num + num;
        } else {
            r.r1 = 1;
        }
    }

    @Actor
    public void actor2(I_Result r) {
        num = 2;
        ready = true;
    }

}
```

执行

```
mvn clean install
java -jar target/jcstress.jar
```

会输出我们感兴趣的结果，摘录其中一次结果：

```
*** INTERESTING tests
  Some interesting behaviors observed. This is for the plain curiosity.

  2 matching test results.
      [OK] test.ConcurrencyTest
    (JVM args: [-XX:-TieredCompilation])
  Observed state   Occurrences              Expectation  Interpretation
```

```
                0         1,729   ACCEPTABLE_INTERESTING  !!!!
                1    42,617,915               ACCEPTABLE  ok
                4     5,146,627               ACCEPTABLE  ok

      [OK] test.ConcurrencyTest
    (JVM args: [])
  Observed state   Occurrences              Expectation  Interpretation
                0         1,652   ACCEPTABLE_INTERESTING  !!!!
                1    46,460,657               ACCEPTABLE  ok
                4     4,571,072               ACCEPTABLE  ok
```

可以看到，出现结果为 0 的情况有 638 次，虽然次数相对很少，但毕竟是出现了。

3.2 解决方法

volatile 修饰的变量，可以禁用指令重排

```
@JCStressTest
@Outcome(id = {"1", "4"}, expect = Expect.ACCEPTABLE, desc = "ok")
@Outcome(id = "0", expect = Expect.ACCEPTABLE_INTERESTING, desc = "!!!!")
@State
public class ConcurrencyTest {

    int num = 0;
    volatile boolean ready = false;
    @Actor
    public void actor1(I_Result r) {
        if(ready) {
            r.r1 = num + num;
        } else {
            r.r1 = 1;
        }
    }

    @Actor
    public void actor2(I_Result r) {
        num = 2;
        ready = true;
    }

}
```

结果为：

```
*** INTERESTING tests
  Some interesting behaviors observed. This is for the plain curiosity.

  0 matching test results.
```

3.3 有序性理解

JVM 会在不影响正确性的前提下，可以调整语句的执行顺序，思考下面一段代码

```
static int i;
static int j;

// 在某个线程内执行如下赋值操作
i = ...; // 较为耗时的操作
j = ...;
```

可以看到，至于是先执行 i 还是 先执行 j，对最终的结果不会产生影响。所以，上面代码真正执行时，既可以是

```
i = ...; // 较为耗时的操作
j = ...;
```

也可以是

```
j = ...;
i = ...; // 较为耗时的操作
```

这种特性称之为『指令重排』，多线程下『指令重排』会影响正确性，例如著名的 double-checked locking 模式实现单例

```
public final class Singleton {
    private Singleton() { }
    private static Singleton INSTANCE = null;
    public static Singleton getInstance() {
        // 实例没创建，才会进入内部的 synchronized代码块
        if (INSTANCE == null) {
            synchronized (Singleton.class) {
                // 也许有其它线程已经创建实例，所以再判断一次
                if (INSTANCE == null) {
                    INSTANCE = new Singleton();
                }
            }
        }
        return INSTANCE;
    }
}
```

以上的实现特点是：

- 懒惰实例化
- 首次使用 getInstance() 才使用 synchronized 加锁，后续使用时无需加锁

但在多线程环境下，上面的代码是有问题的，`INSTANCE = new Singleton()` 对应的字节码为：

```
0: new           #2                  // class cn/itcast/jvm/t4/Singleton
3: dup
4: invokespecial #3                  // Method "<init>":()V
7: putstatic     #4                  // Field
INSTANCE:Lcn/itcast/jvm/t4/Singleton;
```

其中 4 7 两步的顺序不是固定的，也许 jvm 会优化为：先将引用地址赋值给 INSTANCE 变量后，再执行构造方法，如果两个线程 t1，t2 按如下时间序列执行：

```
时间1  t1 线程执行到 INSTANCE = new Singleton();
时间2  t1 线程分配空间，为Singleton对象生成了引用地址（0 处）
时间3  t1 线程将引用地址赋值给 INSTANCE，这时 INSTANCE != null（7 处）
时间4  t2 线程进入getInstance() 方法，发现 INSTANCE != null（synchronized块外），直接返回 INSTANCE
时间5  t1 线程执行Singleton的构造方法（4 处）
```

这时 t1 还未完全将构造方法执行完毕，如果在构造方法中要执行很多初始化操作，那么 t2 拿到的是将是一个未初始化完毕的单例

对 INSTANCE 使用 volatile 修饰即可，可以禁用指令重排，但要注意在 JDK 5 以上的版本的 volatile 才会真正有效

3.4 happens-before

happens-before 规定了哪些写操作对其它线程的读操作可见，它是可见性与有序性的一套规则总结，抛开以下 happens-before 规则，JMM 并不能保证一个线程对共享变量的写，对于其它线程对该共享变量的读可见

- 线程解锁 m 之前对变量的写，对于接下来对 m 加锁的其它线程对该变量的读可见

```
static int x;
static Object m = new Object();

new Thread(()->{
    synchronized(m) {
        x = 10;
    }
},"t1").start();

new Thread(()->{
    synchronized(m) {
        System.out.println(x);
    }
},"t2").start();
```

- 线程对 volatile 变量的写，对接下来其它线程对该变量的读可见

```
volatile static int x;

new Thread(()->{
    x = 10;
},"t1").start();

new Thread(()->{
    System.out.println(x);
},"t2").start();
```

- 线程 start 前对变量的写，对该线程开始后对该变量的读可见

```
static int x;

x = 10;

new Thread(()->{
    System.out.println(x);
},"t2").start();
```

- 线程结束前对变量的写，对其它线程得知它结束后的读可见（比如其它线程调用 t1.isAlive() 或 t1.join()等待它结束）

```
static int x;

Thread t1 = new Thread(()->{
    x = 10;
},"t1");
t1.start();

t1.join();
System.out.println(x);
```

- 线程 t1 打断 t2（interrupt）前对变量的写，对于其他线程得知 t2 被打断后对变量的读可见（通过t2.interrupted 或 t2.isInterrupted）

```
static int x;

public static void main(String[] args) {
    Thread t2 = new Thread(()->{
        while(true) {
            if(Thread.currentThread().isInterrupted()) {
                System.out.println(x);
                break;
            }
        }
    },"t2");
    t2.start();

    new Thread(()->{
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        x = 10;
        t2.interrupt();
    },"t1").start();

    while(!t2.isInterrupted()) {
        Thread.yield();
    }
    System.out.println(x);
```

```
}
```

- 对变量默认值（0，false，null）的写，对其它线程对该变量的读可见
- 具有传递性，如果 x hb-> y 并且 y hb-> z 那么有 x hb-> z

> 变量都是指成员变量或静态成员变量
>
> 参考： 第17页

4. CAS 与 原子类

4.1 CAS

CAS 即 `Compare and Swap`，它体现的一种乐观锁的思想，比如多个线程要对一个共享的整型变量执行 +1 操作：

```
// 需要不断尝试
while(true) {
  int 旧值 = 共享变量 ; // 比如拿到了当前值 0
  int 结果 = 旧值 + 1; // 在旧值 0 的基础上增加 1 ，正确结果是 1

  /*
    这时候如果别的线程把共享变量改成了 5，本线程的正确结果 1 就作废了，这时候
    compareAndSwap 返回 false，重新尝试，直到：
    compareAndSwap 返回 true，表示我本线程做修改的同时，别的线程没有干扰
  */
  if( compareAndSwap ( 旧值, 结果 )) {
    // 成功，退出循环
  }
}
```

获取共享变量时，为了保证该变量的可见性，需要使用 volatile 修饰。结合 CAS 和 volatile 可以实现无锁并发，适用于竞争不激烈、多核 CPU 的场景下。

- 因为没有使用 synchronized，所以线程不会陷入阻塞，这是效率提升的因素之一
- 但如果竞争激烈，可以想到重试必然频繁发生，反而效率会受影响

CAS 底层依赖于一个 Unsafe 类来直接调用操作系统底层的 CAS 指令，下面是直接使用 Unsafe 对象进行线程安全保护的一个例子

```
import sun.misc.Unsafe;
import java.lang.reflect.Field;
public class TestCAS {
    public static void main(String[] args) throws InterruptedException {
        DataContainer dc = new DataContainer();
        int count = 5;
```

```
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < count; i++) {
                dc.increase();
            }
        });
        t1.start();
        t1.join();
        System.out.println(dc.getData());
    }
}


class DataContainer {
    private volatile int data;
    static final Unsafe unsafe;
    static final long DATA_OFFSET;

    static {
        try {
            // Unsafe 对象不能直接调用，只能通过反射获得
            Field theUnsafe = Unsafe.class.getDeclaredField("theUnsafe");
            theUnsafe.setAccessible(true);
            unsafe = (Unsafe) theUnsafe.get(null);
        } catch (NoSuchFieldException | IllegalAccessException e) {
            throw new Error(e);
        }
        try {
            // data 属性在 DataContainer 对象中的偏移量，用于 Unsafe 直接访问该属性
            DATA_OFFSET =
unsafe.objectFieldOffset(DataContainer.class.getDeclaredField("data"));
        } catch (NoSuchFieldException e) {
            throw new Error(e);
        }
    }

    public void increase() {
        int oldValue;
        while(true) {
            // 获取共享变量旧值，可以在这一行加入断点，修改 data 调试来加深理解
            oldValue = data;
            // cas 尝试修改 data 为 旧值 + 1，如果期间旧值被别的线程改了，返回 false
            if (unsafe.compareAndSwapInt(this, DATA_OFFSET, oldValue, oldValue +
1)) {
                return;
            }
        }
    }

    public void decrease() {
        int oldValue;
        while(true) {
            oldValue = data;
            if (unsafe.compareAndSwapInt(this, DATA_OFFSET, oldValue, oldValue -
1)) {
                return;
            }
        }
    }
```

```
    public int getData() {
        return data;
    }
}
```

4.2 乐观锁与悲观锁

- CAS 是基于乐观锁的思想：最乐观的估计，不怕别的线程来修改共享变量，就算改了也没关系，我吃亏点再重试呗。
- synchronized 是基于悲观锁的思想：最悲观的估计，得防着其它线程来修改共享变量，我上了锁你们都别想改，我改完了解开锁，你们才有机会。

4.3 原子操作类

juc（java.util.concurrent）中提供了原子操作类，可以提供线程安全的操作，例如：AtomicInteger、AtomicBoolean等，它们底层就是采用 CAS 技术 + volatile 来实现的。

可以使用 AtomicInteger 改写之前的例子：

```
// 创建原子整数对象
private static AtomicInteger i = new AtomicInteger(0);

public static void main(String[] args) throws InterruptedException {
    Thread t1 = new Thread(() -> {
        for (int j = 0; j < 5000; j++) {
            i.getAndIncrement();  // 获取并且自增  i++
//                i.incrementAndGet();  // 自增并且获取  ++i
        }
    });
```

```
    Thread t2 = new Thread(() -> {
        for (int j = 0; j < 5000; j++) {
            i.getAndDecrement(); // 获取并且自减  i--
        }
    });

    t1.start();
    t2.start();
    t1.join();
    t2.join();
    System.out.println(i);
}
```

5. synchronized 优化

Java HotSpot 虚拟机中，每个对象都有对象头（包括 class 指针和 Mark Word）。Mark Word 平时存储这个对象的 `哈希码`、`分代年龄`，当加锁时，这些信息就根据情况被替换为 `标记位`、`线程锁记录指针`、`重量级锁指针`、`线程ID` 等内容

5.1 轻量级锁

如果一个对象虽然有多线程访问，但多线程访问的时间是错开的（也就是没有竞争），那么可以使用轻量级锁来优化。这就好比：

学生（线程 A）用课本占座，上了半节课，出门了（CPU时间到），回来一看，发现课本没变，说明没有竞争，继续上他的课。
如果这期间有其它学生（线程 B）来了，会告知（线程A）有并发访问，线程 A 随即升级为重量级锁，进入重量级锁的流程。

而重量级锁就不是那么用课本占座那么简单了，可以想象线程 A 走之前，把座位用一个铁栅栏围起来

假设有两个方法同步块，利用同一个对象加锁

```
static Object obj = new Object();
public static void method1() {
    synchronized( obj ) {
        // 同步块 A
        method2();
    }
}
public static void method2() {
    synchronized( obj ) {
        // 同步块 B
    }
}
```

每个线程都的栈帧都会包含一个锁记录的结构，内部可以存储锁定对象的 Mark Word

线程 1	对象 Mark Word	线程 2
访问同步块 A，把 Mark 复制到线程 1 的锁记录	01（无锁）	-
CAS 修改 Mark 为线程 1 锁记录地址	01（无锁）	-
成功（加锁）	00（轻量锁）线程 1 锁记录地址	-
执行同步块 A	00（轻量锁）线程 1 锁记录地址	-
访问同步块 B，把 Mark 复制到线程 1 的锁记录	00（轻量锁）线程 1 锁记录地址	-
CAS 修改 Mark 为线程 1 锁记录地址	00（轻量锁）线程 1 锁记录地址	-
失败（发现是自己的锁）	00（轻量锁）线程 1 锁记录地址	-
锁重入	00（轻量锁）线程 1 锁记录地址	-
执行同步块 B	00（轻量锁）线程 1 锁记录地址	-
同步块 B 执行完毕	00（轻量锁）线程 1 锁记录地址	-
同步块 A 执行完毕	00（轻量锁）线程 1 锁记录地址	-
成功（解锁）	01（无锁）	-
-	01（无锁）	访问同步块 A，把 Mark 复制到线程 2 的锁记录
-	01（无锁）	CAS 修改 Mark 为线程 2 锁记录地址
-	00（轻量锁）线程 2 锁记录地址	成功（加锁）
-	...	...

5.2 锁膨胀

如果在尝试加轻量级锁的过程中，CAS 操作无法成功，这时一种情况就是有其它线程为此对象加上了轻量级锁（有竞争），这时需要进行锁膨胀，将轻量级锁变为重量级锁。

```
static Object obj = new Object();
public static void method1() {
    synchronized( obj ) {
        // 同步块
    }
}
```

线程 1	对象 Mark	线程 2
访问同步块，把 Mark 复制到线程 1 的锁记录	01（无锁）	-
CAS 修改 Mark 为线程 1 锁记录地址	01（无锁）	-
成功（加锁）	00（轻量锁）线程 1 锁记录地址	-
执行同步块	00（轻量锁）线程 1 锁记录地址	-
执行同步块	00（轻量锁）线程 1 锁记录地址	访问同步块，把 Mark 复制到线程 2
执行同步块	00（轻量锁）线程 1 锁记录地址	CAS 修改 Mark 为线程 2 锁记录地址
执行同步块	00（轻量锁）线程 1 锁记录地址	失败（发现别人已经占了锁）
执行同步块	00（轻量锁）线程 1 锁记录地址	CAS 修改 Mark 为重量锁
执行同步块	10（重量锁）重量锁指针	阻塞中
执行完毕	10（重量锁）重量锁指针	阻塞中
失败（解锁）	10（重量锁）重量锁指针	阻塞中
释放重量锁，唤起阻塞线程竞争	01（无锁）	阻塞中
-	10（重量锁）	竞争重量锁
-	10（重量锁）	成功（加锁）
-	...	...

5.3 重量锁

重量级锁竞争的时候，还可以使用自旋来进行优化，如果当前线程自旋成功（即这时候持锁线程已经退出了同步块，释放了锁），这时当前线程就可以避免阻塞。

在 Java 6 之后自旋锁是自适应的，比如对象刚刚的一次自旋操作成功过，那么认为这次自旋成功的可能性会高，就多自旋几次；反之，就少自旋甚至不自旋，总之，比较智能。

- 自旋会占用 CPU 时间，单核 CPU 自旋就是浪费，多核 CPU 自旋才能发挥优势。
- 好比等红灯时汽车是不是熄火，不熄火相当于自旋（等待时间短了划算），熄火了相当于阻塞（等待时间长了划算）
- Java 7 之后不能控制是否开启自旋功能

自旋重试成功的情况

线程 1 （cpu 1 上）	对象 Mark	线程 2 （cpu 2 上）
-	10（重量锁）	-
访问同步块，获取 monitor	10（重量锁）重量锁指针	-
成功（加锁）	10（重量锁）重量锁指针	-
执行同步块	10（重量锁）重量锁指针	-
执行同步块	10（重量锁）重量锁指针	访问同步块，获取 monitor
执行同步块	10（重量锁）重量锁指针	自旋重试
执行完毕	10（重量锁）重量锁指针	自旋重试
成功（解锁）	01（无锁）	自旋重试
-	10（重量锁）重量锁指针	成功（加锁）
-	10（重量锁）重量锁指针	执行同步块
-	...	...

自旋重试失败的情况

线程 1（cpu 1 上）	对象 Mark	线程 2（cpu 2 上）
-	10（重量锁）	-
访问同步块，获取 monitor	10（重量锁）重量锁指针	-
成功（加锁）	10（重量锁）重量锁指针	-
执行同步块	10（重量锁）重量锁指针	-
执行同步块	10（重量锁）重量锁指针	访问同步块，获取 monitor
执行同步块	10（重量锁）重量锁指针	自旋重试
执行同步块	10（重量锁）重量锁指针	自旋重试
执行同步块	10（重量锁）重量锁指针	自旋重试
执行同步块	10（重量锁）重量锁指针	阻塞
-	...	...

5.4 偏向锁

轻量级锁在没有竞争时（就自己这个线程），每次重入仍然需要执行 CAS 操作。Java 6 中引入了偏向锁来做进一步优化：只有第一次使用 CAS 将线程 ID 设置到对象的 Mark Word 头，之后发现这个线程 ID 是自己的就表示没有竞争，不用重新 CAS.

- 撤销偏向需要将持锁线程升级为轻量级锁，这个过程中所有线程需要暂停（STW）
- 访问对象的 hashCode 也会撤销偏向锁
- 如果对象虽然被多个线程访问，但没有竞争，这时偏向了线程 T1 的对象仍有机会重新偏向 T2，重偏向会重置对象的 Thread ID
- 撤销偏向和重偏向都是批量进行的，以类为单位
- 如果撤销偏向到达某个阈值，整个类的所有对象都会变为不可偏向的
- 可以主动使用 -XX:-UseBiasedLocking 禁用偏向锁

可以参考这篇论文：https://www.oracle.com/technetwork/java/biasedlocking-oopsla2006-wp-149958.pdf

假设有两个方法同步块，利用同一个对象加锁

```
static Object obj = new Object();
public static void method1() {
    synchronized( obj ) {
        // 同步块 A
        method2();
    }
}
public static void method2() {
    synchronized( obj ) {
        // 同步块 B
    }
}
```

线程 1	对象 Mark
访问同步块 A，检查 Mark 中是否有线程 ID	101（无锁可偏向）
尝试加偏向锁	101（无锁可偏向）对象 hashCode
成功	101（无锁可偏向）线程ID
执行同步块 A	101（无锁可偏向）线程ID
访问同步块 B，检查 Mark 中是否有线程 ID	101（无锁可偏向）线程ID
是自己的线程 ID，锁是自己的，无需做更多操作	101（无锁可偏向）线程ID
执行同步块 B	101（无锁可偏向）线程ID
执行完毕	101（无锁可偏向）对象 hashCode

5.5 其它优化

1. 减少上锁时间

同步代码块中尽量短

2. 减少锁的粒度

将一个锁拆分为多个锁提高并发度，例如：

- ConcurrentHashMap
- LongAdder 分为 base 和 cells 两部分。没有并发争用的时候或者是 cells 数组正在初始化的时候，会使用 CAS 来累加值到 base，有并发争用，会初始化 cells 数组，数组有多少个 cell，就允许有多少线程并行修改，最后将数组中每个 cell 累加，再加上 base 就是最终的值
- LinkedBlockingQueue 入队和出队使用不同的锁，相对于LinkedBlockingArray只有一个锁效率要高

3. 锁粗化

多次循环进入同步块不如同步块内多次循环
另外 JVM 可能会做如下优化，把多次 append 的加锁操作粗化为一次（因为都是对同一个对象加锁，没必要重入多次）

```
new StringBuffer().append("a").append("b").append("c");
```

4. 锁消除

JVM 会进行代码的逃逸分析，例如某个加锁对象是方法内局部变量，不会被其它线程所访问到，这时候就会被即时编译器忽略掉所有同步操作。

5. 读写分离

CopyOnWriteArrayList
ConyOnWriteSet

参考：

https://wiki.openjdk.java.net/display/HotSpot/Synchronization

http://luojinping.com/2015/07/09/java锁优化/

https://www.infoq.cn/article/java-se-16-synchronized

https://www.jianshu.com/p/9932047a89be

https://www.cnblogs.com/sheeva/p/6366782.html

https://stackoverflow.com/questions/46312817/does-java-ever-rebias-an-individual-lock