**Race Conditions and Critical Sections**

* [Critical Sections](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html#critical-sections)
* [Race Conditions in Critical Sections](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html#race-conditions-in-critical-sections)
* [Preventing Race Conditions](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html)
* [Critical Section Throughput](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html#critical-section-throughput)

*竞争条件*是在临界区间内可能发生的特殊情况。临界区间是一个被多线程执行的代码区间，并且其中对于线程的执行顺序使得临界区间的并发执行的结果有所不同。

当多线程执行一个临界区间的结果可能不同，取决于线程执行的顺序，临界区间就被说是包含一个竞争条件。词条竞争条件源于这样的隐喻，即线程正在竞争通过临界区段，并且该竞争的结果影响临界区段执行的结果。

这可能听起来有点复杂，因此我将在下面的章节中详细阐述竞争条件和临界区间。

**Critical Sections**

在同一应用程序中运行多个线程本身不会引起问题。当多个线程访问相同的资源时会出现问题。例如，相同的内存（变量、数组或对象）、系统（数据库、Web服务等）或文件。

事实上，只有一个或多个线程写入这些资源时才会出现问题。如果资源不改变，则允许多个线程读取相同的资源是安全的。

这是一个临界区间Java代码示例，如果被多个线程执行则可能失败：

public class Counter {

protected long count = 0;

public void add(long value){

this.count = this.count + value;

}

}

想像如果有两个线程，A和B，执行相加方法在Counter类的相同实例上。没有方法知道什么时候操作系统在两个线程之间切换。add()方法里的代码没有被JVM作为一个单原子指令执行。而是将它作为一组较小的指令集执行，类似：

1. 从内存读取this.count到寄存器
2. 增加值到寄存器
3. 寄存器写入内存

观察发生了什么随着接下来线程A和B的混合执行：

this.count = 0;

A: Reads this.count into a register (0)

B: Reads this.count into a register (0)

B: Adds value 2 to register

B: Writes register value (2) back to memory. this.count now equals 2

A: Adds value 3 to register

A: Writes register value (3) back to memory. this.count now equals 3

这两个线程想要加2和3到计数器上。因此值应当是5，当两个线程完成执行。然而，因为两个线程的执行是交错的，结果最终是不同的。

执行序列例举以上，两个线程都从内存读取值0。然后它们增加独立的值，2和3，给值，然后写结果到内存中。不是5，this.count中剩下的值将是最后一个线程写的值。在上面的情况下，它是线程A，但它也可以是线程B。

**Race Conditions in Critical Sections**

早前例子中add()方法中的代码包含一个临界区间。当多线程执行这个临界区间，竞争条件发生。

更正式地说，两个线程竞争同一资源的情况称为竞争条件，其中访问资源的顺序很重要。导致竞争条件的代码区间称为临界区间。

**Preventing Race Conditions**

为了避免竞争条件发生你必须确保临界区间被一个原子指令执行。那意味着一旦一个单线程正在执行它，没有别的线程可以执行它直到第一个线程离开临界区间。

竞争条件可以被避免通过临界区间的适当的线程同步。线程同步可以通过使用Java同步代码块([**synchronized block of Java code**](http://tutorials.jenkov.com/java-concurrency/synchronized.html))获得。线程同步也可以被获取通过使用其他的同步建构像锁([**locks**](http://tutorials.jenkov.com/java-concurrency/locks.html))或者原子变量如原子整型([**java.util.concurrent.atomic.AtomicInteger**](http://tutorials.jenkov.com/java-concurrent-util/atomicinteger.html))。

**Critical Section Throughput**

对于较小的临界区间，使整个临界区间变成一个同步块可能有用。但是，对较大的临界区间可能打碎临界区间为较小的临界区间，从而允许多线程执行每个较小的临界区间，是有益处的。这可以减少共享资源的争用，从而增加总临界区间的吞吐量。

这是一个非常简化的Java代码实例展示出我的意思：

public class TwoSums {

private int sum1 = 0;

private int sum2 = 0;

public void add(int val1, int val2){

synchronized(this){

this.sum1 += val1;

this.sum2 += val2;

}

}

}

注意add()方法如何将值添加到两个不同的和成员变量。为了防止竞争条件，在Java同步块内执行求和。通过这个实现，只有一个线程可以同一时刻执行求和。

然而，由于两个求和变量彼此独立，所以可以将它们的求和拆分为两个独立的同步块，如下所示：

public class TwoSums {

private int sum1 = 0;

private int sum2 = 0;

private Integer sum1Lock = new Integer(1);

private Integer sum2Lock = new Integer(2);

public void add(int val1, int val2){

synchronized(this.sum1Lock){

this.sum1 += val1;

}

synchronized(this.sum2Lock){

this.sum2 += val2;

}

}

}

现在两个线程可以同时执行add()方法。一个线程在第一个同步块内部，另一个线程在第二个同步块内部。这两个同步块在不同的块上同步，因此两个不同的线程可以独立执行两个块。这样线程将会彼此等待更短时间执行add()方法。

当然这个例子非常简单。在现实生活中的共享资源中，关键部分的分解可能要复杂得多，并且需要对执行顺序的可能性进行更多的分析。