# 出自:

腾讯课堂 700多分钟干货实战Java多线程高并发高性能实战全集 , 我学习完了之后,我做了个笔记.

# 需求

在一线大厂的日常开发工作中,我们经常需要对业务平台(系统)\纯技术中间件的相关功能进行高并发\高性能的性能测试

假如说你是架构师,请用多线程相关基础知识实现单机环境下,在3秒内实现1亿的计数,并统计亿级计数的耗时.

上面做法很显然单个线程是很慢的, 你用单个线程在三秒内去实现一亿的计数器,这个肯定是搞不定的,肯定是需要多线程的,多线程跑的话需要统计你多个线程计数耗时是多少,就是说呢多个线程从1一起数到一亿,那整个耗时是多少.

# 头脑风暴

## 1.本次实战中要保证什么数据的并发安全?

一亿的数据量,如果用单线程跑的话三秒肯定是跑不完的,所以得需要多线程去计数,这个计数不能多也不能少,不能是错的.

## 2.本次实战案例中,有几大类的线程角色?

并发线程处理: 并发线程去记数

统计线程:统计线程去记录耗时

所以就有两大类线程角色

# 分析

1.亿级高并发高性能计数器的特点是并发量很大,如何有效保证一亿并发呢,单个线程肯定是不行的,需要多线程去做

2.性能要求比较高,要求是3秒以内,这个肯定单线程是不行的,性能不行,得需要多线程一起计算.

3.多线程来计算需要并发安全性问题,多个线程,大家一起来计数,不能多也不能少,必须要求正好是一个亿.

4.有两大类线程,一个是计数类的线程,一个是统计时间的线程,统计线程等到所有的计数线程结束之后再做统计,再计算耗时是多少.

# 核心问题

我们可以从哪些具体的大的角度实现亿级并发计数?

1.安全性问题

多个线程一起并发计数,怎么保证并发安全性

2.线程等待怎么处理

计数线程做完了计数之后,统计线程才需要统计计数线程的耗时.

线程等待可以用 join , CountDownLatch 线程状态TERMINATED来做

3.并发安全性怎么保证

用锁计数 , 原子操作类

# 演示

这里有11种,其实还有更多的方案,不仅仅11种

## 先说结论

建议用下面两种方式,性能是最好的:

8 LongAdder+ CountDownLatch

9 .LongAccumulator + CountDownLatch

## 1.AtomicLong+线程join方式

AtomicLong是原子操作类，多线程操作的时候，不会出现线程安全问题

thread 其实可以换成线程池的方式,

计算线程执行完了之后,调用join方法进行线程等待, 然后主线程开始统计计算的结果

下面的方法执行完了结果是 748 毫秒

package com.yrxy.thread.case4;  
  
import java.util.concurrent.atomic.AtomicLong;  
  
public class Counter1 {  
  
 public static AtomicLong inc = new AtomicLong();  
  
 public void increase() {  
 inc.getAndIncrement();  
 }  
  
 public static void main(String[] args) throws InterruptedException {  
 for (int i = 0; i < 10; i++) {  
 inc.set(0);  
 count();  
 }  
 }  
  
 private static void count() throws InterruptedException {  
 final Counter1 test = new Counter1();  
 Thread th;  
 long t1 = System.currentTimeMillis();  
 // 这里是1000个线程, 每个线程计数100000 .  
 for (int i = 0; i < 1000; i++) {  
 th = new Thread(() -> {  
 for (int j = 0; j < 100000; j++) {  
 test.increase();  
 }  
 });  
 th.start();  
 //计数线程做完之后就调用join操作进行等待.  
 th.join();  
 }  
  
 long t2 = System.currentTimeMillis();  
 System.out.println("Counter1 , " + String.format("结果：%s,耗时(ms)：%s", inc, (t2 - t1)));  
 }  
  
}

执行结果：

Counter1 , 结果：100000000,耗时(ms)：706  
Counter1 , 结果：100000000,耗时(ms)：638  
Counter1 , 结果：100000000,耗时(ms)：634  
Counter1 , 结果：100000000,耗时(ms)：646  
Counter1 , 结果：100000000,耗时(ms)：642  
Counter1 , 结果：100000000,耗时(ms)：651  
Counter1 , 结果：100000000,耗时(ms)：638  
Counter1 , 结果：100000000,耗时(ms)：656  
Counter1 , 结果：100000000,耗时(ms)：639  
Counter1 , 结果：100000000,耗时(ms)：696  
  
进程已结束，退出代码为 0

## 2. ReentrantLock+线程join方式

这种方式 计数器 inc不是线程安全的,所有在操作这个inc变量累加之前需要先获取锁,加完再释放锁.

这个操作性能不如方式1使用原子类的方式

执行结果是 : 1855 毫秒

package com.yrxy.thread.case4;  
  
import java.util.concurrent.locks.Lock;  
import java.util.concurrent.locks.ReentrantLock;  
  
public class Counter2 {  
   
 public static int inc = 0;  
 Lock lock = new ReentrantLock();  
   
 public void increase() {  
 // 加锁  
 lock.lock();  
 try {  
 // 由于这个inc不是线程安全的,所以为了保证线程安全问题,累加的时候需要加锁,加完了之后再释放锁.  
 //这种方式不如原子类的性能高  
 inc++;  
 } finally{  
 //释放锁  
 lock.unlock();  
 }  
 }  
   
 public static void main(String[] args) throws InterruptedException {  
 inc = 0;  
 count();  
  
 }  
  
 private static void count() throws InterruptedException {  
 final Counter2 test = new Counter2();  
 Thread th;  
 long t1 = System.currentTimeMillis();  
 //1000 个线程, 每个线程计算100000  
 for(int i=0;i<1000;i++){  
 th= new Thread(() -> {  
 for(int j=0;j<100000;j++)  
 test.increase();  
 });  
 th.start();  
 //线程等待  
 th.join();  
 }  
  
 long t2 = System.currentTimeMillis();  
 System.out.println("Counter2 , "+String.format("结果：%s,耗时(ms)：%s", test.inc, (t2 - t1)));  
 }  
}

执行结果是 :

Counter2 , 结果：100000000,耗时(ms)：1855  
Counter2 , 结果：100000000,耗时(ms)：1804  
Counter2 , 结果：100000000,耗时(ms)：1789  
Counter2 , 结果：100000000,耗时(ms)：1793  
Counter2 , 结果：100000000,耗时(ms)：1791  
Counter2 , 结果：100000000,耗时(ms)：1788  
Counter2 , 结果：100000000,耗时(ms)：1796  
Counter2 , 结果：100000000,耗时(ms)：1794  
Counter2 , 结果：100000000,耗时(ms)：1805  
Counter2 , 结果：100000000,耗时(ms)：1792  
  
进程已结束，退出代码为 0

## 3.Synchronized+线程join方式

这种方式和方式2几乎是一样的,区别就是将ReentrantLock 换成了Synchronized , 这种方式执行效率不如 ReentrantLock 的方式,效率比ReentrantLock 慢了100多ms

package com.yrxy.thread.case4;  
  
  
  
public class Counter3 {  
   
 public static int inc = 0;  
 public synchronized void increase() {  
 inc++;  
 }  
   
 public static void main(String[] args) throws InterruptedException {  
 inc = 0;  
 count();  
  
 }  
  
 private static void count() throws InterruptedException {  
 final Counter3 test = new Counter3();  
 Thread th;  
 long t1 = System.currentTimeMillis();  
 for(int i=0;i<1000;i++){  
 th= new Thread(){  
 public void run() {  
 for(int j=0;j<100000;j++)  
 test.increase();  
 };  
 };  
 th.start();  
 th.join();  
 }  
  
 long t2 = System.currentTimeMillis();  
 System.out.println("Counter3, "+String.format("结果：%s,耗时(ms)：%s", test.inc, (t2 - t1)));  
 }  
}

执行结果:

Counter3, 结果：100000000,耗时(ms)：2027  
Counter3, 结果：100000000,耗时(ms)：1998  
Counter3, 结果：100000000,耗时(ms)：2001  
Counter3, 结果：100000000,耗时(ms)：1989  
Counter3, 结果：100000000,耗时(ms)：1975  
Counter3, 结果：100000000,耗时(ms)：1978  
Counter3, 结果：100000000,耗时(ms)：1978  
Counter3, 结果：100000000,耗时(ms)：1987  
Counter3, 结果：100000000,耗时(ms)：1985  
Counter3, 结果：100000000,耗时(ms)：1987  
  
进程已结束，退出代码为 0

## 4.AtomicLong+线程状态Thread.State.TERMINATED

这种效果跟方式1区别就是将join 变成线程状态了,, 这种方式执行结果是 1936 ms ,效率不如join方式

package com.yrxy.thread.case4;  
  
import java.util.concurrent.atomic.AtomicLong;  
  
public class Counter4 {  
  
 public static AtomicLong inc = new AtomicLong();  
   
 public void increase() {  
 inc.getAndIncrement();  
 }  
   
 public static void main(String[] args) throws InterruptedException {  
 inc.set(0);  
 count();  
 }  
  
 private static void count() throws InterruptedException {  
 final Counter4 test = new Counter4();  
 long t1 = System.currentTimeMillis();  
 Thread th = null;  
 for(int i=0;i<1000;i++){  
 th= new Thread(){  
 public void run() {  
 for(int j=0;j<100000;j++)  
 test.increase();  
 };  
 };  
 th.start();  
 // 线程内部状态类,如果线程不是完成状态,那么就继续等待  
 while(th.getState()!=Thread.State.TERMINATED){  
 Thread.sleep(1);  
 }  
 }  
  
 long t2 = System.currentTimeMillis();  
 System.out.println("Counter4, "+String.format("结果：%s,耗时(ms)：%s", test.inc, (t2 - t1)));  
 }  
  
}

执行结果:

Counter4, 结果：100000000,耗时(ms)：1935  
Counter4, 结果：100000000,耗时(ms)：1884  
Counter4, 结果：100000000,耗时(ms)：1925  
Counter4, 结果：100000000,耗时(ms)：1893  
Counter4, 结果：100000000,耗时(ms)：1903  
Counter4, 结果：100000000,耗时(ms)：1872  
Counter4, 结果：100000000,耗时(ms)：1891  
Counter4, 结果：100000000,耗时(ms)：1890  
Counter4, 结果：100000000,耗时(ms)：1896  
Counter4, 结果：100000000,耗时(ms)：1867  
  
进程已结束，退出代码为 0

## 5.AtomicLong+线程状态isAlive

和方式5区别就是用isAlive来判断状态,性能不太好, 执行结果是: 1895毫秒

package com.yrxy.thread.case4;  
  
import java.util.concurrent.atomic.AtomicLong;  
  
  
public class Counter5 {  
  
 public static AtomicLong inc = new AtomicLong();  
   
 public void increase() {  
 inc.getAndIncrement();  
 }  
   
 public static void main(String[] args) throws InterruptedException {  
 inc.set(0);  
 count();  
 }  
  
 private static void count() throws InterruptedException {  
 final Counter5 test = new Counter5();  
 Thread th = null;  
 long t1 = System.currentTimeMillis();  
 for(int i=0;i<1000;i++){  
 th= new Thread(){  
 public void run() {  
 for(int j=0;j<100000;j++)  
 test.increase();  
 };  
 };  
 th.start();  
 // 用的是 isAlive 判断线程状态  
 while(th.isAlive()){  
 Thread.sleep(1);  
 }  
 }  
  
 long t2 = System.currentTimeMillis();  
 System.out.println("Counter5, "+String.format("结果：%s,耗时(ms)：%s", test.inc, (t2 - t1)));  
 }  
  
}

执行结果:

Counter5, 结果：100000000,耗时(ms)：1912  
Counter5, 结果：100000000,耗时(ms)：1902  
Counter5, 结果：100000000,耗时(ms)：1892  
Counter5, 结果：100000000,耗时(ms)：1878  
Counter5, 结果：100000000,耗时(ms)：1900  
Counter5, 结果：100000000,耗时(ms)：1908  
Counter5, 结果：100000000,耗时(ms)：1882  
Counter5, 结果：100000000,耗时(ms)：1886  
Counter5, 结果：100000000,耗时(ms)：1904  
Counter5, 结果：100000000,耗时(ms)：1893  
  
进程已结束，退出代码为 0

## 6.synchronized+CountDownLatch

这种方式除了第一个用了2000ms,后面都是用了700 800ms ,不知道为啥这样,等我研究研究

package com.yrxy.thread.case4;  
  
import java.util.concurrent.CountDownLatch;  
import java.util.concurrent.ExecutionException;  
  
public class counter6 {  
  
 static int count = 0;  
  
 public static synchronized void incr() {  
 count++;  
 }  
  
 public static void main(String[] args) throws ExecutionException, InterruptedException {  
 for (int i = 0; i < 10; i++) {  
 count = 0;  
 count();  
 }  
 }  
  
 private static void count() throws InterruptedException {  
 long t1 = System.currentTimeMillis();  
 //1000个计数器  
 int threadCount = 1000;  
 CountDownLatch countDownLatch = new CountDownLatch(threadCount);  
 for (int i = 0; i < threadCount; i++) {  
 new Thread(() -> {  
 try {  
 for (int j = 0; j < 100000; j++) {  
 incr();  
 }  
 } finally {  
 // 递减锁存计数  
 countDownLatch.countDown();  
 }  
 }).start();  
 }  
 //等待  
 countDownLatch.await();  
 long t2 = System.currentTimeMillis();  
 System.out.println("Counter6, " + String.format("结果：%s,耗时(ms)：%s", count, (t2 - t1)));  
 }  
  
}

执行结果:

Counter6, 结果：100000000,耗时(ms)：1972  
Counter6, 结果：100000000,耗时(ms)：838  
Counter6, 结果：100000000,耗时(ms)：853  
Counter6, 结果：100000000,耗时(ms)：825  
Counter6, 结果：100000000,耗时(ms)：860  
Counter6, 结果：100000000,耗时(ms)：877  
Counter6, 结果：100000000,耗时(ms)：874  
Counter6, 结果：100000000,耗时(ms)：847  
Counter6, 结果：100000000,耗时(ms)：833  
Counter6, 结果：100000000,耗时(ms)：840  
  
进程已结束，退出代码为 0

## 7.AtomicLong+CountDownLatch

耗时：1921

package com.yrxy.thread.case4;
  
   
import java.util.concurrent.CountDownLatch;
  
import java.util.concurrent.ExecutionException;
  
import java.util.concurrent.atomic.AtomicLong;
  
   
public class counter7 {
  
   
 static AtomicLong count = new AtomicLong(0);
  
   
 public static void incr() {
  
 count.incrementAndGet();
  
 }
  
   
 public static void main(String[] args) throws ExecutionException, InterruptedException {
  
 for (int i = 0; i < 10; i++) {
  
 count.set(0);
  
 count();
  
 }
  
 }
  
   
 private static void count() throws InterruptedException {
  
 long t1 = System.currentTimeMillis();
  
 int threadCount = 1000;
  
 CountDownLatch countDownLatch = new CountDownLatch(threadCount);
  
 for (int i = 0; i < threadCount; i++) {
  
 new Thread(() -> {
  
 try {
  
 for (int j = 0; j < 100000; j++) {
  
 incr();
  
 }
  
 } finally {
  
 countDownLatch.countDown();
  
 }
  
 }).start();
  
 }
  
 countDownLatch.await();
  
 long t2 = System.currentTimeMillis();
  
 System.out.println("Counter7, " + String.format("结果：%s,耗时(ms)：%s", count, (t2 - t1)));
  
 }
  
   
}

输出结果:

Counter7, 结果：100000000,耗时(ms)：1921
  
Counter7, 结果：100000000,耗时(ms)：1830
  
Counter7, 结果：100000000,耗时(ms)：1835
  
Counter7, 结果：100000000,耗时(ms)：1870
  
Counter7, 结果：100000000,耗时(ms)：1827
  
Counter7, 结果：100000000,耗时(ms)：1830
  
Counter7, 结果：100000000,耗时(ms)：1860
  
Counter7, 结果：100000000,耗时(ms)：1830
  
Counter7, 结果：100000000,耗时(ms)：1839
  
Counter7, 结果：100000000,耗时(ms)：1840
  
   
进程已结束，退出代码为 0

## 8.LongAdder+ CountDownLatch

性能非常的好,执行耗时就 :206毫秒

package com.yrxy.thread.case4;
  
   
import java.util.concurrent.CountDownLatch;
  
import java.util.concurrent.ExecutionException;
  
import java.util.concurrent.atomic.LongAdder;
  
   
   
public class Counter8 {
  
 static LongAdder count = new LongAdder();
  
   
 public static void incr() {
  
 count.increment();
  
 }
  
   
 public static void main(String[] args) throws ExecutionException, InterruptedException {
  
 for (int i = 0; i < 10; i++) {
  
 count.reset();
  
 m1();
  
 }
  
 }
  
   
 private static void m1() throws ExecutionException, InterruptedException {
  
 long t1 = System.currentTimeMillis();
  
 int threadCount = 1000;
  
 CountDownLatch countDownLatch = new CountDownLatch(threadCount);
  
 for (int i = 0; i < threadCount; i++) {
  
 new Thread(() -> {
  
 try {
  
 for (int j = 0; j < 100000; j++) {
  
 incr();
  
 }
  
 } finally {
  
 countDownLatch.countDown();
  
 }
  
 }).start();
  
 }
  
 countDownLatch.await();
  
 long t2 = System.currentTimeMillis();
  
 System.out.println("Counter8, " + String.format("结果：%s,耗时(ms)：%s", count.sum(), (t2 - t1)));
  
 }
  
   
   
}

执行结果:

Counter8, 结果：100000000,耗时(ms)：206
  
Counter8, 结果：100000000,耗时(ms)：145
  
Counter8, 结果：100000000,耗时(ms)：133
  
Counter8, 结果：100000000,耗时(ms)：141
  
Counter8, 结果：100000000,耗时(ms)：132
  
Counter8, 结果：100000000,耗时(ms)：178
  
Counter8, 结果：100000000,耗时(ms)：194
  
Counter8, 结果：100000000,耗时(ms)：186
  
Counter8, 结果：100000000,耗时(ms)：160
  
Counter8, 结果：100000000,耗时(ms)：158
  
   
进程已结束，退出代码为 0

## 9.LongAccumulator + CountDownLatch

性能也很好,差不多 189毫秒

package com.yrxy.thread.case4;
  
   
import java.util.concurrent.CountDownLatch;
  
import java.util.concurrent.ExecutionException;
  
import java.util.concurrent.atomic.LongAccumulator;
  
   
/\*\*
  
 \*/
  
public class Counter9 {
  
   
 static LongAccumulator count = new LongAccumulator((x, y) -> x + y, 0L);
  
   
 public static void incr() {
  
 count.accumulate(1);
  
 }
  
   
 public static void main(String[] args) throws ExecutionException, InterruptedException {
  
 for (int i = 0; i < 10; i++) {
  
 count.reset();
  
 m1();
  
 }
  
 }
  
   
 private static void m1() throws ExecutionException, InterruptedException {
  
 long t1 = System.currentTimeMillis();
  
 int threadCount = 1000;
  
 CountDownLatch countDownLatch = new CountDownLatch(threadCount);
  
 for (int i = 0; i < threadCount; i++) {
  
 new Thread(() -> {
  
 try {
  
 for (int j = 0; j < 100000; j++) {
  
 incr();
  
 }
  
 } finally {
  
 countDownLatch.countDown();
  
 }
  
 }).start();
  
 }
  
 countDownLatch.await();
  
 long t2 = System.currentTimeMillis();
  
 System.out.println("Counter9, "+String.format("结果：%s,耗时(ms)：%s", count.longValue(), (t2 - t1)));
  
 }
  
   
}

运行结果

Counter9, 结果：100000000,耗时(ms)：189
  
Counter9, 结果：100000000,耗时(ms)：160
  
Counter9, 结果：100000000,耗时(ms)：152
  
Counter9, 结果：100000000,耗时(ms)：128
  
Counter9, 结果：100000000,耗时(ms)：132
  
Counter9, 结果：100000000,耗时(ms)：165
  
Counter9, 结果：100000000,耗时(ms)：150
  
Counter9, 结果：100000000,耗时(ms)：161
  
Counter9, 结果：100000000,耗时(ms)：167
  
Counter9, 结果：100000000,耗时(ms)：158

## 10.LongAdder+join方式

耗时: 1051毫秒

package com.yrxy.thread.case4;
  
   
import java.util.concurrent.atomic.LongAdder;
  
   
public class Counter10 {
  
   
 public static LongAdder inc = new LongAdder();
  
   
 public static void main(String[] args) throws InterruptedException {
  
 for (int i = 0; i < 10; i++) {
  
 inc.reset();
  
 count();
  
 }
  
   
 }
  
   
 private static void count() throws InterruptedException {
  
 final Counter10 test = new Counter10();
  
 Thread th;
  
 long t1 = System.currentTimeMillis();
  
 for (int i = 0; i < 1000; i++) {
  
 th = new Thread(() -> {
  
 for (int j = 0; j < 100000; j++)
  
 test.increase();
  
 });
  
 th.start();
  
 th.join();
  
 }
  
   
 long t2 = System.currentTimeMillis();
  
 System.out.println("Counter10 , " + String.format("结果：%s,耗时(ms)：%s", test.inc, (t2 - t1)));
  
 }
  
   
 public void increase() {
  
 inc.increment();
  
 }
  
   
}

运行结果

Counter10 , 结果：100000000,耗时(ms)：1051
  
Counter10 , 结果：100000000,耗时(ms)：1081
  
Counter10 , 结果：100000000,耗时(ms)：1061
  
Counter10 , 结果：100000000,耗时(ms)：1089
  
Counter10 , 结果：100000000,耗时(ms)：1044
  
Counter10 , 结果：100000000,耗时(ms)：1057
  
Counter10 , 结果：100000000,耗时(ms)：1063
  
Counter10 , 结果：100000000,耗时(ms)：1064
  
Counter10 , 结果：100000000,耗时(ms)：1065
  
Counter10 , 结果：100000000,耗时(ms)：1074

## 11.LongAccumulator+join方式

执行 1067毫秒

package com.yrxy.thread.case4;
  
   
import java.util.concurrent.atomic.LongAccumulator;
  
   
public class Counter11 {
  
   
 static LongAccumulator inc = new LongAccumulator((x, y) -> x + y, 0L);
  
   
 public static void main(String[] args) throws InterruptedException {
  
 for (int i = 0; i < 10; i++) {
  
 inc.reset();
  
 count();
  
 }
  
   
 }
  
   
 private static void count() throws InterruptedException {
  
 final Counter11 test = new Counter11();
  
 Thread th;
  
 long t1 = System.currentTimeMillis();
  
 for (int i = 0; i < 1000; i++) {
  
 th = new Thread(() -> {
  
 for (int j = 0; j < 100000; j++)
  
 test.increase();
  
 });
  
 th.start();
  
 th.join();
  
 }
  
   
 long t2 = System.currentTimeMillis();
  
 System.out.println("Counter11 , " + String.format("结果：%s,耗时(ms)：%s", test.inc, (t2 - t1)));
  
 }
  
   
 public void increase() {
  
 inc.accumulate(1);
  
 }
  
   
}

运行结果

Counter11 , 结果：100000000,耗时(ms)：1144
  
Counter11 , 结果：100000000,耗时(ms)：1120
  
Counter11 , 结果：100000000,耗时(ms)：1104
  
Counter11 , 结果：100000000,耗时(ms)：1113
  
Counter11 , 结果：100000000,耗时(ms)：1096
  
Counter11 , 结果：100000000,耗时(ms)：1072
  
Counter11 , 结果：100000000,耗时(ms)：1067
  
Counter11 , 结果：100000000,耗时(ms)：1053
  
Counter11 , 结果：100000000,耗时(ms)：1062
  
Counter11 , 结果：100000000,耗时(ms)：1050
  
   
进程已结束，退出代码为 0