发件人：方堃 fangkun119@hotmail.com
主题：
日期：2021年3月4日 下午2:02
收件人：

CH21 多线程(E)：仿真、性能、乐观容器、乐观锁、活动对象

2016年10月8日 星期六
19:08

仿真：模拟三个场景

1. 用线程模拟银行出纳员 P737
2. 用线程模拟一个饭店 P741
3. 用线程模拟汽车工厂的任务分发 P744

性能调优

目录

1.互斥技术比较 【目录】

测试代码如何些	不正确性能测试代码 P748 本例演示了所谓的“微基准测试”危险[9]，这个术语通常指在隔离的、脱离上下文环境的情况下对某个特性进行性能测试。当然，你仍旧必须编写测试来验证诸如“**Lock比synchronized更快**”这样的断言，但是你需要在编写这些测试的时候意识到，在编译过程中和在运行时实际会发生什么。 上面的示例存在着大量的问题。首先也是最重要的是，我们只有在这些互斥存在竞争的情况下，才能看到真正的性能差异，因此必须有多个任务尝试着访问互斥代码区。而在上面的示例中，每个互斥都是由单个的**main()**线程在隔离的情况下测试的。
编译器对synchronized的优化	其次，当编译器看到**synchronized**关键字时，有可能会执行特殊的优化，甚至有可能会注意到这个程序是单线程的。编译器甚至可能会识别出**counter**被递增的次数是固定数量的，因此会预先计算出其结果。不同的编译器和运行时系统在这方面会有所差异，因此很难确切了解将会发生什么，但是我们需要防止编译器去预测结果的可能性。
正确的测试代码 P749 测试代码的设计模式、思路 P754 剔除编译器优化因素的方法 P754	为了创建有效的测试，我们必须使程序更加复杂。首先我们需要多个任务，但并不只是会修改内部值的任务，还包括读取这些值的任务（否则优化器可以识别出这些值从来都不会被使用）。另外，计算必须足够复杂和不可预测，以使得编译器没有机会执行积极优化。这可以通过预加载一个大型的随机**int**数组（预加载可以减小在主循环上调用**Random.nextInt()**所造成的影响），并在计算总和时使用它们来实现；
测试结论	• Lock和Atomic的性能好于synchronized • 互斥保护的方法体内部的开销，可能远大于获得互斥锁所需的开销，因此Loc,/Atomic的性能优势很可能被冲淡 • Atomic使用范围有限制，Lock的代码可读性不如synchronized

2.免锁容器 【目录】

历史	Java 1.1	Vector, Hashtable中提供了很多synchronized方法，缺点是不需要多线程时，带来了额外的大量开销

	Java 1.2	新的容器类库是不同步的，并且Collections提供了static的同步装饰方法
		开销仍然是基于synchronized加锁机制的
		`Collection<String> c = Collections.synchronizedCollection(new ArrayList<String>());`
		`List<String> list = Collections.synchronizedList(new ArrayList<String>());`
		`Set<String> s = Collection.synchronizedSet(new HashSet<String>());`
		`SortedSet<String> ss = Collection.synchronizedSortedSet(new TreeSet<String>());`
		`Set<String> ss2 = ss;`
		`SortedMap<String, String> sm = Collection.synchronizedMap(new HashMap<String,String>());`
		`Map<String, String> sm2 = sm;`

	Java SE5	提供了更加轻量级的同步类库，例如基于CopyOnWrite，读写并行并且只有写完成的版本才对读可见等

容器举例	CopyOnWriteArrayList	机制	写操作会把整个数组都重写一遍，然后做一个原子切换
		优点	写的时候，读操作可以并发进行 多个迭代器同时遍历和修改时，不会抛出ConcurrentModificationException
		缺点	整个数组都会被重写
	CopyOnWriteArraySet	与CopyOnWriteArrayList类似	
	ConcurrentHashMap	使用类似的技术，正在写入的元素对于读不可见，写完成后，读操作可以见到最新的版本 区别是，写入的不是整个HashMap，而仅仅是被修改的元素，改动点小 不会抛出ConcurrentModificationException	
	ConcurrentLinkedQueue	与ConcurrentHashMap类似	

性能影响	乐观锁思想（要参照读写比并测试）	乐观锁概念，就是假定读多写少时，使用偏向于读的同步机制，会更有助于提升性能 但是实际上，读写比多少时，应该使用偏向于读的同步机制，还需要根据测试数据来决定 确定使用乐观锁思路时，可以使用CopyOnWriteArrayList，ConcurrentHashMap等机制		
	测试代码框架	P755， P757		

	对比1：ArrayList	对比项	Collections.synchronizedList(...)	`return Collections.synchronizedList(`
				`  new ArrayList<Integer>(`
				`  new CountingIntegerList(containerSize)));`

			CopyOnWriteArrayList	`return new CopyOnWriteArrayList<Integer>(` `  new CountingIntegerList(containerSize));`
		结论	结论：即使读者数：写者数比达到5:5，CopyOnWriteArrayList仍然占优	

耗时：68.1 v.s 249.5
数组长度怎样？从命令行参数传入的，参数2，应该是10

数据

```
Type                          Read time      Write time
Synched ArrayList 10r 0w      232158294700            0
Synched ArrayList 9r 1w       198947618203    24918613399
readTime + writeTime =        223866231602
Synched ArrayList 5r 5w       117367305062   132176613508
readTime + writeTime =        249543918570
CopyOnWriteArrayList 10r 0w      758386889             0
CopyOnWriteArrayList 9r 1w       741305671     136145237
readTime + writeTime =           877450908
CopyOnWriteArrayList 5r 5w       212763075    67967464300
readTime + writeTime =         68180227375
```

对比2：HashMap

对比项

Collections.synchronizedMap(...)	`return Collections.synchronizedMap(new HashMap<Integer,Integer>(MapData.map(new CountingGenerator.Integer(), new CountingGenerator.Integer(), containerSize)));`
ConcurrentHashMap(...)	`return new ConcurrentHashMap<Integer,Integer>(MapData.map(new CountingGenerator.Integer(), new CountingGenerator.Integer(), containerSize));`

结论

ConcurrentHashMap占优，并且ConcurrentHashMap对写者数量的增加不敏感
这是因为ConcurrentHashMap使用了不同的技术来最小化写操作对性能的影响

数据

```
Type                          Read time      Write time
Synched HashMap 10r 0w        306052025049            0
Synched HashMap 9r 1w         428319156207    47697347568
readTime + writeTime =        476016503775
Synched HashMap 5r 5w         243956877760   244012003202
readTime + writeTime =        487968880962
ConcurrentHashMap 10r 0w       23352654318             0
ConcurrentHashMap 9r 1w        18833089400    1541853224
readTime + writeTime =         20374942624
ConcurrentHashMap 5r 5w        12037625732   11850489099
readTime + writeTime =         23888114831
*///:~
```

3.部分使用了乐观锁的Atomic类【目录】

内容

- Atomic类提供了decrementAndSet()、compareAndSet()类型的操作
- 部分Atomic类的这些操作使用了乐观锁：即假设读的频率远大于写：

函数参数包括了旧值、新值
先与旧值比较：如果相同，就Set；如果不同，就失败

- 因为存在了失败的概率，因此必须考虑如何容错，如果不能兼容失败情况，就需要回退重新借助synchronized方法

例子

数据

```
//: concurrency/FastSimulation.java
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class FastSimulation {
  static final int N_ELEMENTS = 100000;
  static final int N_GENES = 30;
  static final int N_EVOLVERS = 50;
  static final AtomicInteger[][] GRID =
    new AtomicInteger[N_ELEMENTS][N_GENES];
  static Random rand = new Random(47);
```

```
//二维AtomicInteger数组
static final AtomicInteger[][] GRID
   = new AtomicInteger[N_ELEMENTS][N_GENES];
```

```
  public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < N_ELEMENTS; i++)
      for(int j = 0; j < N_GENES; j++)
        GRID[i][j] = new AtomicInteger(rand.nextInt(1000));
    for(int i = 0; i < N_EVOLVERS; i++)
      exec.execute(new Evolver());
    TimeUnit.SECONDS.sleep(5);
    exec.shutdownNow();
  }
} /* (Execute to see output) *///:~
```

```
//main
//线程池
ExecutorService execSvc = Executors.newCachedThreadPool();
//填充二维数组
for (int i = 0; ...) {
  for (int j = 0; ...) {
    GRID[i][j] = new AtomicInteger(rand.nextInt(1000));
//启动一组线程
for (int i = 0; ...)
  execSvc.execute(new Evolver());
//主线程sleep之后关闭ExecutorService，同时发送中断让工作线程结束
TimeUnit.SENCONDS.sleep(5);
execSvc.shutdownNow();
```

线程

```
static class Evolver implements Runnable {
  public void run() {
    while(!Thread.interrupted()) {
      // Randomly select an element to work on:
      int element = rand.nextInt(N_ELEMENTS);
      for(int i = 0; i < N_GENES; i++) {
        int previous = element - 1;
        if(previous < 0) previous = N_ELEMENTS - 1;
        int next = element + 1;
        if(next >= N_ELEMENTS) next = 0;
        int oldvalue = GRID[element][i].get();
        // Perform some kind of modeling calculation:
        int newvalue = oldvalue +
          GRID[previous][i].get() + GRID[next][i].get();
        newvalue /= 3; // Average the three values
        if(!GRID[element][i]
          .compareAndSet(oldvalue, newvalue)) {
          // Policy here to deal with failure. Here, we
          // just report it and ignore it; our model
          // will eventually deal with it.
          print("Old value changed from " + oldvalue);
        }
      }
    }
  }
}
```

```
//随机取一个元素，以及其前后相邻的元素
int randElemIdx = rand.nextInt(N_ELEMENTS);
for (int i = 0; i < N_GENES; ++i) {
  int preElemIdx = randElemIdx - 1;
  preElemIdx = (preElemIdx < 0) ? N_ELEMENTS - 1 : preElemIdx;
  int nextElemIdx = nextElemIdx + 1;
  nextElemIdx = (nextElemIdx >= N_ELEMENTS) ? 0 : nextElemIdx;
  //计算新值(相邻元素值求平均)
  int oldValue = GRID[element][i].get();
  int newValue = oldValue + GRID[preElemIdx][i].get() + GRID[next][i].get();
  newValue /= 3;
  //在写-写冲突少场景下，CAS操作可以用轻量级的方式提高并发
  if (!GRID[randElemIdx][i].compareAndSet(oldValue, newValue))) {
    ...//CAS成功则输出，失败则放弃
  }
}
```

4.读写锁【目录】

性能影响

读写锁对于性能的影响是不可确定的，影响因素包括如下，唯一可明确的方法是做实验

- 读取频率 / **ReadWriteLock**是否能够提高程序的性能是完全不可确定的，它取决于诸如数据被读取的频

	修改频率	率与被修改的频率相比较的结果，读取和写入操作的时间（锁将更复杂，因此短操作并不能带来好处），有多少线程竞争以及是否在多处理机器上运行等因素。最终，唯一可以了解**ReadWriteLock**是否能够给你的程序带来好处的方式就是用试验来证明。

例子

用读写锁包裹的容器

```
//: concurrency/ReaderWriterList.java
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class ReaderWriterList<T> {
  private ArrayList<T> lockedList;
        // Make the ordering fair:
        private ReentrantReadWriteLock lock =
          new ReentrantReadWriteLock(true);
        public ReaderWriterList(int size, T initialValue) {
          lockedList = new ArrayList<T>(
            Collections.nCopies(size, initialValue));
        }
        public T set(int index, T element) {
          Lock wlock = lock.writeLock();
          wlock.lock();
          try {
            return lockedList.set(index, element);
          } finally {
            wlock.unlock();
          }
        }
        public T get(int index) {
          Lock rlock = lock.readLock();
          rlock.lock();
          try {
            // Show that multiple readers
            // may acquire the read lock:
            if(lock.getReadLockCount() > 1)
              print(lock.getReadLockCount());
            return lockedList.get(index);
          } finally {
            rlock.unlock();
          }
        }
        public static void main(String[] args) throws Exception {
          new ReaderWriterListTest(30, 1);
        }
      }
```

```
//读写锁
ReentrantReadWriteLock reentrantReadWriteLock
 = new ReentrantReadWriteLock(true)
//写操作
public T set(...) {
 Lock wlock = reentrantReadWriteLock.writeLock();
 wlock.lock();
 try {
  ...
 } finally {
  wlock.unlock();
 }
}
//读操作
public T get(...) {
 Lock rlock = reentrantReadWriteLock.readLock();
 rlock.lock();
 try {
  ...
 } finally {
  rlock.unlock();
 }
}
```

线程代码

```
  class ReaderWriterListTest {
    ExecutorService exec = Executors.newCachedThreadPool();
    private final static int SIZE = 100;
    private static Random rand = new Random(47);
    private ReaderWriterList<Integer> list =
      new ReaderWriterList<Integer>(SIZE, 0);
    private class Writer implements Runnable {
      public void run() {
        try {
          for(int i = 0; i < 20; i++) { // 2 second test
            list.set(i, rand.nextInt());
            TimeUnit.MILLISECONDS.sleep(100);
          }
        } catch(InterruptedException e) {
          // Acceptable way to exit
        }
        print("Writer finished, shutting down");
        exec.shutdownNow();
      }
    }
    private class Reader implements Runnable {
      public void run() {
        try {
          while(!Thread.interrupted()) {
            for(int i = 0; i < SIZE; i++) {
              list.get(i);
              TimeUnit.MILLISECONDS.sleep(1);
            }
          }
        } catch(InterruptedException e) {
          // Acceptable way to exit
        }
      }
    }
    public ReaderWriterListTest(int readers, int writers) {
      for(int i = 0; i < readers; i++)
        exec.execute(new Reader());
      for(int i = 0; i < writers; i++)
        exec.execute(new Writer());
    }
  } /* (Execute to see output) *///:~
```

```
reader线程get()
writer线程set()
```

4.活动对象【目录】

用途	简化多线程编程，不使用互斥锁之类的同步机制，但同时能避免多线程竞争
活动对象	活动对象、又称行动者，这类对象： 1.每个对象都维护着自己的worker_thread和item_queue 2.所有请求都进入item_queue 排队，任何时候，一个活动对象只能处理一个item，这样就item串行化了 3.实现方式： 向活动对象发消息 -> 消息转变成任务 -> 任务插入到活动对象的队列中 -> 等待该任务在之后某个时刻运行 借助 Java SE5 的 Future 来实现

例子

公用代码

```
//: concurrency/ActiveObjectDemo.java
// Can only pass constants, immutables, "disconnected
// objects," or other active objects as arguments
// to asynch methods.
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class ActiveObjectDemo {
  private ExecutorService ex =
    Executors.newSingleThreadExecutor();
  private Random rand = new Random(47);
  // Insert a random delay to produce the effect
  // of a calculation time:
```

```
//线程池中只有一个线程(对于每个活动对象ActiveObjectDomo来说)
ExecutorService singleThreadExecutor
  = Executors.newSingleThreadExecutor();
```

		private void pause(int factor) {
		try {
		TimeUnit.MILLISECONDS.sleep(
		100 + rand.nextInt(factor));
		} catch(InterruptedException e) {
		print("sleep() interrupted");
		}
		}	

	添加任务	public Future<Integer>
		calculateInt(final int x, final int y) {
		return ex.submit(new Callable<Integer>() {
		public Integer call() {
		print("starting " + x + " + " + y);
		pause(500);
		return x + y;
		}
		});
		}
		public Future<Float>
		calculateFloat(final float x, final float y) {
		return ex.submit(new Callable<Float>() {
		public Float call() {
		print("starting " + x + " + " + y);
		pause(2000);
		return x + y;
		}
		});
		}
			//调用该函数会向线程池提交任，每次只能运行一个任务
			public Future<Integer>
			calculateInt(final int x, final int y) {
			return singleThreadExecutor.submit(new Callable<Integer>() {
			public Integer call() {
			...
			}
			});
			}
			//调用该函数会向线程池提交另一种任务，线程池每次只能运行一个任务
			public Future<Float>
			calciulateFloat(final float x, final float y) {
			return singleThreadExecutor.submit(new Callable<Float>() {
			public Float call() {
			...
			}
			});
			}

	Demo
	Driver
		public static void main(String[] args) {
		ActiveObjectDemo d1 = new ActiveObjectDemo();
		// Prevents ConcurrentModificationException:
		List<Future<?>> results =
		new CopyOnWriteArrayList<Future<?>>();
		for(float f = 0.0f; f < 1.0f; f += 0.2f)
		results.add(d1.calculateFloat(f, f));
		for(int i = 0; i < 5; i++)
		results.add(d1.calculateInt(i, i));
		print("All asynch calls made");
		while(results.size() > 0) {
		for(Future<?> f : results)
		if(f.isDone()) {
		try {
		print(f.get());
		} catch(Exception e) {
		throw new RuntimeException(e);
		}
		results.remove(f);
		}
		}
		d1.shutdown();
		}
		} /* Output: (85% match)
		All asynch calls made
		starting 0.0 + 0.0
		starting 0.2 + 0.2
		0.0
		starting 0.4 + 0.4
		0.4
		starting 0.6 + 0.6
		0.8
			//创建一个ActiveObject，同一时间内就只有一个工作线程
			//如果创建量个ActiveObject，同一时间内就有两个工作线程
			ActiveObjectDemo activeObject = new ActiveObjectDeom();
			//CopyOnWriteArrayList<>通过多版本策略以更多的写操作来减少切换开销
			List<Future<?>> resultList = new CopyOnWriteArrayList<Future<?>>();
			//向ActiveObject添加任务
			for (float f = 0.0f; f < 1.0f; f += .2f)
			resultList.add(activeObject.calculateFloat(f, f));
			for (float i = 0; i < 5; ++i)
			resultList.add(activeObject.calculateInt(i, i));
			//因为ActiveObjectDemo使用的时SingleThreadExecutor，任务串行进行
			//接收处理完的任务
			while (resultList.size() > 0) {
			for (Future<?> future : resultList)
			...
			//因为resultList是CopyOnWriteList
			//remove发生在读操作时不会引发ConcurrentModificationException
			resultList.remove(future);
			}
			}
			//关闭ActiveObject
			activeObject.shutdown();