Java Unit Stress Test

Java Unit Tests sind ein zentraler Bestandteil für die Qualitätssicherung der programmierten Java Software. Neben den Funktionstests gemäss Use Cases oder einfachen Methodentests geht das Augenmerk auf die parallele (gleichzeitige) Ausführung und damit Threadsafety oft vergessen. Moderne Java Anwendungen laufen in der Regel im Hintergrund asynchron und damit parallel ab. Unit Tests sollten die korrekte und stabile Ausführung unter Last oder Stress aufzeigen. Ein weiteres Ziel von Stress Tests ist das Auffinden von Memory Leaks, welche es leider auch in Java jederzeit geben kann.Dieser Blog zeigt auf wie man mit einfachsten Mitteln, mit nur einer Java Klasse, diese Stress Tests programmieren kann.Das folgende Listing zeigt die Klasse ch.std.test.StressTest:

package ch.std.test;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.concurrent.*;

public class StressTest {
  private Callable callable;
  private int cycles;
  private int runsPerCycle;
  private List<Exception> exceptionList;

  public StressTest(Callable callable, int cycles, int runsPerCycle) {
    this.callable = callable;
    this.cycles = cycles;
    this.runsPerCycle = runsPerCycle;
    this.exceptionList = Collections.synchronizedList(new ArrayList());
  }

  public void test() throws Exception {
    ExecutorService executor = Executors.newFixedThreadPool(this.runsPerCycle);
    for (int i = 0; i < this.cycles; i++) {
      this.exceptionList.clear();
      List<Callable<V>> callables = new ArrayList<>();
      for (int j = 0; j < this.runsPerCycle; j++) {
        callables.add(callable);
      }
      List<Future<V>> futures = executor.invokeAll(callables);
      for (Future<V> future : futures) {
        try {
          if (!future.isDone()) {
            this.exceptionList.add(new Exception("future is not done"));
          } else {
            V f = future.get();
          }
        } catch (CancellationException ce) {
          this.exceptionList.add(ce);
        } catch (ExecutionException ee) {
          this.exceptionList.add(ee);
        } catch (InterruptedException ie) {
          this.exceptionList.add(ie);
          Thread.currentThread().interrupt();
        } catch (Exception e) {
          this.exceptionList.add(e);
        }
      }
      if (!this.exceptionList.isEmpty()) {
        for (Exception e: this.exceptionList) {
          System.out.println(e.getMessage());
        }
        throw new Exception("test with exception, see list");
      }
    }
  }

  public void add(Exception e) {
    this.exceptionList.add(e);
  }

  public List<Exception> getExceptionList() {
    return Collections.unmodifiableList(exceptionList);
  }

  public void printResult(PrintStream ps) {
    if (this.exceptionList.isEmpty()) {
      ps.println("no exceptions");
      return;
    }
    for (Exception exception : exceptionList) {
      ps.println(exception.getMessage());
    }
  }
}

Mit dem folgenden Listing zeigen wir 3 Tests. Der 1. Test fügt synchron 100 Zahlen in ein HashSet. Solcher funktioniert ohne Probleme und das HashSet enthält korrekt die 100 Einträge.Der 2. Test arbeitet parallel mit dem StressTest. 100 Threads werden via Executor Service und dem fixen ThreadPool gestartet. Alle Threads arbeiten mit dem gleichen Set. Jeder Run entfernt die Zahl, berechnet den Set HashCode und fügt die Zahl wieder ins Set ein. Hier resultiert in der Regel eine ConcurrencyException, oder das Set verfügt über eine falsche Anzahl(size).Der 3. Test arbeitet analog dem 2. Test, verwendet aber ein Threadsafe Set. Dieser Test funktioniert auch mit dem StressTest einwandfrei.

package ch.std.unittest.demo;
import java.util.Collections;
import java.util.HashSet;
import java.util.Set;

import org.junit.Assert;
import org.junit.Test;

import ch.std.test.StressTest;

public class StressTestDemo {
  @Test
  public void testSyncSetSingleRun() {
    Set iSet = new HashSet<>();
    for(int i = 0; i< 100; i++) {
      iSet.remove(i);
      iSet.add(i);
    }
    Assert.assertEquals(100, iSet.size());
  }
  @Test
  public void testHashSetMultipleRun() throws Exception {
    Set iSet = new HashSet<>();
    StressTest stressTest = new StressTest<>(() -> {
      for(int i

= 0; i< 100; i++) {
  iSet.remove(i);// force exception
  iSet.hashCode(); // force exception
  iSet.add(i);
  }
  return iSet.size();
  }, 1, 100);
 stressTest.test();
  stressTest.printResult(System.out);
  Assert.assertEquals(100, iSet.size());
  Assert.assertTrue(stressTest.getExceptionList().isEmpty());
  }
 @Test
  public void testConcurrentHashSetMultipleRun() throws Exception {
  Set iSet = Collections.synchronizedSet(new HashSet());
  StressTest stressTest = new StressTest<>(() -> {
  for(int i = 0; i< 100; i++) {
  iSet.remove(i);
 iSet.hashCode();
  iSet.add(i);
  }
  return iSet.size();
  }, 1, 100);
 stressTest.test();
  stressTest.printResult(System.out);
  Assert.assertEquals(100, iSet.size());
  Assert.assertTrue(stressTest.getExceptionList().isEmpty());
  }
}

Der folgende ScreenShot zeigt das Verhalten der 3 Unit Tests:Der StressTest zeigt auf, dass die HashSet Klasse nicht threadsafe ist.Es ist in jedem Projekt zentral, dass relevante Codeteile auf ihre korrekt parallele Ausführung getestet werden. Insbesondere Server Komponenten wie Web oder REST Services sind hier besonders zu berücksichtigen.Die StressTest Klasse erleichtert uns hier die Arbeit.

Feedback

War dieser Blog für Sie wertvoll. Wir danken für jede Anregung und Feedback

Kontakt

Simtech AG
Finkenweg 23
3110 Münsingen
Schweiz

Impressum

Das Copyright für sämtliche Inhalte dieser Website liegt bei Simtech AG, Schweiz.
Beachten Sie auch unsere Hinweise zum Urheberrecht, Datenschutz und Haftungsauschluss.
Jeder Hinweis auf Fehler nehmen wir gerne entgegen.

Copyright

2024 Simtech AG, All rights reserved, Powered by stack.ch written in Golang by Daniel Schmutz

https://www.simtech-ag.ch/testHashSetMultipleRun