(FULL THREAD =http://infinispan-developer-list.980875.n3.nabble.com/infinispan-dev-Infinispan-embedded-off-heap-cache-td4026102.html)

Jan 15, 2014; 2:26pm

Re: [infinispan-dev] Infinispan embedded off-heap cache

49 posts

FYI. Some results from a Test that Peter Lawrey just wrote wrt to comparing Netty allocater vs. OpenHFT allocater's direct invoke of Unsafe malloc/free. Indeed, Netty's use of a PooledHeap approach does result in a 100% speed improvement (wrt to allocation events). However, OpenHFT has a huge advantage wrt its underlying BytesMarshallable capability to blazing serialize/deserialize 'back to the heap!' value object COPY transports (that could then be viewed as NIO-operable ByteBuffer).

Interesting.

Moral of the story? Both Netty and OpenHFT should likely both be significant contributors to this ambition to deliver a compelling off-heap Cache<K,V> capability to ISPN.

---peter.lawrey@higherfrequencytrading.com wrote: --------------------

The first thing I noticed is that allocating using the Pooled Heap is twice as fast on my machine, Netty creating/freeing 256 bytes is 11 million vs DirectStore 5.6 million per second. Note: HHM avoids doing this at all and I suspect this difference is not important for HHM.

I re-wrote one of their tests as a performance test. Given they don't appear to performance test their object serialization is a worry ;) but it also means I probably didn't do it as optimally as it could be. In the following test I serialize and deserialize an object with four fields String, int, double, Enum using the same writeExternalizable/readExternalizble code.

Netty: Serialization/Deserialization latency: 327,499 us avg
Netty: Serialization/Deserialization latency: 97,419 us avg
Netty: Serialization/Deserialization latency: 54,232 us avg
Netty: Serialization/Deserialization latency: 58,950 us avg
Netty: Serialization/Deserialization latency: 53,177 us avg
Netty: Serialization/Deserialization latency: 53,189 us avg
Netty: Serialization/Deserialization latency: 53,672 us avg
Netty: Serialization/Deserialization latency: 52,871 us avg

Netty: Serialization/Deserialization latency: 52,211 us avg
Netty: Serialization/Deserialization latency: 51,924 us avg
DirectStore: Externalizable latency: 6,899 us avg
DirectStore: Externalizable latency: 825 us avg
DirectStore: Externalizable latency: 496 us avg
DirectStore: Externalizable latency: 494 us avg
DirectStore: Externalizable latency: 385 us avg
DirectStore: Externalizable latency: 212 us avg
DirectStore: Externalizable latency: 201 us avg
DirectStore: Externalizable latency: 197 us avg
DirectStore: Externalizable latency: 199 us avg
DirectStore: Externalizable latency: 203 us avg

The code is

```
/*
 * Copyright 2012 The Netty Project
 *
 * The Netty Project licenses this file to you under the Apache License,
 * version 2.0 (the "License"); you may not use this file except in compliance
 * with the License. You may obtain a copy of the License at:
 *
 *   http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
 * WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
 * License for the specific language governing permissions and limitations
 * under the License.
 */
package io.netty.handler.codec.marshalling;

import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandler;
import io.netty.channel.embedded.EmbeddedChannel;
import net.openhft.lang.io.Bytes;
import net.openhft.lang.io.DirectBytes;
import net.openhft.lang.io.DirectStore;
import net.openhft.lang.io.serialization.BytesMarshallable;
import org.jboss.marshalling.MarshallerFactory;
import org.jboss.marshalling.Marshalling;
import org.jboss.marshalling.MarshallingConfiguration;
import org.jboss.marshalling.Unmarshaller;
import org.jetbrains.annotations.NotNull;
import org.junit.Test;
```

```
import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;
import java.lang.annotation.RetentionPolicy;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNull;
import static org.junit.Assert.assertTrue;

public class SerialMarshallingEncoderTest extends SerialCompatibleMarshallingEncoderTest
{

   @Override
   protected ByteBuf truncate(ByteBuf buf) {
      buf.readInt();
      return buf;
   }

   @Override
   protected ChannelHandler createEncoder() {
      return new MarshallingEncoder(createProvider());
   }

   @Test
   public void testMarshallingPerf() throws Exception {
      MyData testObject = new MyData("Hello World", 1, 2.0, RetentionPolicy.RUNTIME);

      final MarshallerFactory marshallerFactory = createMarshallerFactory();
      final MarshallingConfiguration configuration = createMarshallingConfig();
      Unmarshaller unmarshaller = marshallerFactory.createUnmarshaller(configuration);

      for (int t = 0; t < 10; t++) {
         long start = System.nanoTime();
         int RUNS = 10000;
         for (int i = 0; i < RUNS; i++) {
            EmbeddedChannel ch = new EmbeddedChannel(createEncoder());
            ch.writeOutbound(testObject);
            assertTrue(ch.finish());

            ByteBuf buffer = ch.readOutbound();

            unmarshaller.start(Marshalling.createByteInput(truncate(buffer).nioBuffer()));
            MyData read = (MyData) unmarshaller.readObject();
            assertEquals(testObject, read);
```

```
            assertEquals(-1, unmarshaller.read());

            assertNull(ch.readOutbound());
            buffer.release();
        }
        long average = (System.nanoTime() - start) / RUNS;
        System.out.printf("Netty: Serialization/Deserialization latency: %,d us avg%n",
average);
    }

    unmarshaller.finish();
    unmarshaller.close();
  }

  @Test
  public void testMarshallingPerfDirectStore() throws Exception {
    MyData testObject = new MyData("Hello World", 1, 2.0, RetentionPolicy.RUNTIME);
    MyData testObject2 = new MyData("test", 12, 222.0, RetentionPolicy.CLASS);

    DirectStore ds = DirectStore.allocateLazy(256);
    DirectBytes db = ds.createSlice();
    for (int t = 0; t < 10; t++) {
      long start = System.nanoTime();
      int RUNS = 10000;
      for (int i = 0; i < RUNS; i++) {
        db.reset();
        testObject.writeExternal(db);
        long position = db.position();
        db.reset();
        testObject2.readExternal(db);
        assertEquals(testObject, testObject2);

        assertEquals(position, db.position());
      }
      long average = (System.nanoTime() - start) / RUNS;
      System.out.printf("DirectStore: Externalizable latency: %,d us avg%n", average);
    }
    ds.free();
  }

  public static class MyData implements Externalizable {
    String text;
    int value;
    double number;
    RetentionPolicy policy;
```

```
    public MyData() {
    }

    public MyData(String text, int value, double number, RetentionPolicy policy) {
       this.text = text;
       this.value = value;
       this.number = number;
       this.policy = policy;
    }

    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
       out.writeUTF(text);
       out.writeInt(value);
       out.writeDouble(number);
       out.writeUTF(policy.name());
    }

    @Override
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException
{
       text = in.readUTF();
       value = in.readInt();
       number = in.readDouble();
       policy = RetentionPolicy.valueOf(in.readUTF());
    }

    @Override
    public boolean equals(Object o) {
       if (this == o) return true;
       if (o == null || getClass() != o.getClass()) return false;

       MyData myData = (MyData) o;

       if (Double.compare(myData.number, number) != 0) return false;
       if (value != myData.value) return false;
       if (policy != myData.policy) return false;
       if (text != null ? !text.equals(myData.text) : myData.text != null) return false;

       return true;
    }
  }
}
```

On 15 January 2014 16:59, Peter Lawrey <peter.lawrey@gmail.com> wrote:

Good question. I suspect there is a bunch of things it is not doing, but I will investigate.

On 15 January 2014 16:50, Ben Cotton <bendcotton@gmail.com> wrote:
Simplifying my question, is there something that Netty's jemalloc() like off-heap allocation management does that is somehow different (advantageous?) when compared with straightforward usage of Unsafe malloc/free ?