ABOUT BYRON KIOURTZOGLOU

Byron is a master software engineer working in the IT and Telecom domains. He is an applications developer in a wide variety of applications/services. He is currently acting as the team leader and technical architect for a proprietary service creation and integration platform for both the IT and Telecom industries in addition to a in-house big data real-time analytics solution. He is always fascinated by SOA, middleware services and mobile development. Byron is co-founder and Executive Editor at Java Code Geeks.

Java Best Practices – High performance Serialization

Posted by: Byron Kiourtzoglou in Core Java July 23rd, 2010 8 Comments 938 Views

Continuing our series of articles concerning proposed practices while working with the Java programming language, we are going to discuss and demonstrate how to utilize Object Serialization for high performance applications.

All discussed topics are based on use cases derived from the development of mission critical, ultra high performance production systems for the telecommunication industry.

Prior reading each section of this article it is highly recommended that you consult the relevant Java API documentation for detailed information and code samples.

All tests are performed against a Sony Vaio with the following characteristics :

- System : openSUSE 11.1 (x86_64)
- Processor (CPU) : Intel(R) Core(TM)2 Duo CPU T6670 @ 2.20GHz
- Processor Speed : 1,200.00 MHz
- Total memory (RAM) : 2.8 GB
- Java : OpenJDK 1.6.0_0 64-Bit

The following test configuration is applied :

- Concurrent worker Threads : 200
- Test repeats per worker Thread : 1000
- Overall test runs : 100

High performance Serialization

Serialization is the process of converting an object into a stream of bytes. That stream can then be sent through a socket, stored to a file and/or database or simply manipulated as is. With this article we do not intend to present an in depth description of the serialization mechanism, there are numerous articles out there that provide this kind of information. What will be discussed here is our proposition for utilizing serialization in order to achieve high performance results.

The three main performance problems with serialization are :

- Serialization is a recursive algorithm. Starting from a single object, all the objects that can be reached from that object by following instance variables, are also serialized. The default behavior can easily lead to unnecessary Serialization overheads
- Both serializing and deserializing require the serialization mechanism to discover information about the instance it is serializing. Using the default serialization mechanism, will use reflection to discover all the field values. Furthermore if you don't explicitelly set a „serialVersionUID" class attribute, the serialization mechanism has to compute it. This involves going through all the fields and methods to generate a hash. The aforementioned procedure can be quite slow
- Using the default serialization mechanism, all the serializing class description information is included in the stream, such as :
  - The description of all the serializable superclasses
  - The description of the class itself
  - The instance data associated with the specific instance of the class

To solve the aforementioned performance problems you can use Externalization instead. The major difference between these two methods is that Serialization writes out class descriptions of all the serializable superclasses along with the information associated with the instance when viewed as an instance of each individual superclass. Externalization, on the other hand, writes out the identity of the class (the name of the class and the appropriate „serialVersionUID" class attribute) along with the superclass structure and all the information about the class hierarchy. In other words, it stores all the metadata, but writes out only the local instance information. In short, Externalization eliminates almost all the reflective calls used by the serialization mechanism and gives you complete control over the marshalling and demarshalling algorithms, resulting in dramatic performance improvements.

Of course, Externalization efficiency comes at a price. The default serialization mechanism adapts to application changes due to the fact that metadata is automatically extracted from the class definitions. Externalization on the other hand isn't very flexible and requires you to rewrite your marshalling and demarshalling code whenever you change your class definitions.

What follows is a short demonstration on how to utilize Externalization for high performance applications. We will start by providing the "Employee" object to perform serialization and deserialization operations. Two flavors of the "Employee" object will be used. One suitable for standard serialization operations and another that is modified so as to able to be externalized.

Below is the first flavor of the "Employee" object :

```
001 package com.javacodegeeks.test;
002
003 import java.io.Serializable;
004 import java.util.Date;
005 import java.util.List;
006
007 public class Employee implements Serializable {
008
009  private static final long serialVersionUID = 365777329397454389OL;
010
011  private String firstName;
012  private String lastName;
013  private String socialSecurityNumber;
014  private String department;
015  private String position;
016  private Date hireDate;
017  private Double salary;
018  private Employee supervisor;
019  private List<string> phoneNumbers;
020
021  public Employee() {
022  }
023
024  public Employee(String firstName, String lastName,
025    String socialSecurityNumber, String department, String position,
026    Date hireDate, Double salary) {
027   this.firstName = firstName;
028   this.lastName = lastName;
029   this.socialSecurityNumber = socialSecurityNumber;
030   this.department = department;
031   this.position = position;
032   this.hireDate = hireDate;
033   this.salary = salary;
034  }
035
036  public String getFirstName() {
037   return firstName;
038  }
039
040  public void setFirstName(String firstName) {
041   this.firstName = firstName;
042  }
043
044  public String getLastName() {
045   return lastName;
046  }
047
048  public void setLastName(String lastName) {
049   this.lastName = lastName;
050  }
051
052  public String getSocialSecurityNumber() {
053   return socialSecurityNumber;
054  }
055
056  public void setSocialSecurityNumber(String socialSecurityNumber) {
057   this.socialSecurityNumber = socialSecurityNumber;
058  }
059
060  public String getDepartment() {
061   return department;
062  }
063
064  public void setDepartment(String department) {
065   this.department = department;
066  }
067
068  public String getPosition() {
069   return position;
070  }
071
072  public void setPosition(String position) {
073   this.position = position;
074  }
075
076  public Date getHireDate() {
077   return hireDate;
078  }
079
080  public void setHireDate(Date hireDate) {
081   this.hireDate = hireDate;
```

```
 }

 public Double getSalary() {
  return salary;
 }

 public void setSalary(Double salary) {
  this.salary = salary;
 }

 public Employee getSupervisor() {
  return supervisor;
 }

 public void setSupervisor(Employee supervisor) {
  this.supervisor = supervisor;
 }

 public List<string> getPhoneNumbers() {
  return phoneNumbers;
 }

 public void setPhoneNumbers(List<string> phoneNumbers) {
  this.phoneNumbers = phoneNumbers;
 }

}
```

Things to notice here :

- We assume that the following fields are mandatory :
  - "firstName"
  - "lastName"
  - "socialSecurityNumber"
  - "department"
  - "position"
  - "hireDate"
  - "salary"

Following is the second flavor of the "Employee" object :

```
package com.javacodegeeks.test;

import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;
import java.util.Arrays;
import java.util.Date;
import java.util.List;

public class Employee implements Externalizable {

 private String firstName;
 private String lastName;
 private String socialSecurityNumber;
 private String department;
 private String position;
 private Date hireDate;
 private Double salary;
 private Employee supervisor;
 private List<string> phoneNumbers;

 public Employee() {
 }

 public Employee(String firstName, String lastName,
   String socialSecurityNumber, String department, String position,
   Date hireDate, Double salary) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.socialSecurityNumber = socialSecurityNumber;
  this.department = department;
  this.position = position;
  this.hireDate = hireDate;
  this.salary = salary;
 }

 public String getFirstName() {
  return firstName;
 }

 public void setFirstName(String firstName) {
  this.firstName = firstName;
 }

 public String getLastName() {
  return lastName;
 }

 public void setLastName(String lastName) {
  this.lastName = lastName;
 }

 public String getSocialSecurityNumber() {
  return socialSecurityNumber;
 }
```

```
public void setSocialSecurityNumber(String socialSecurityNumber) {
 this.socialSecurityNumber = socialSecurityNumber;
}

public String getDepartment() {
 return department;
}

public void setDepartment(String department) {
 this.department = department;
}

public String getPosition() {
 return position;
}

public void setPosition(String position) {
 this.position = position;
}

public Date getHireDate() {
 return hireDate;
}

public void setHireDate(Date hireDate) {
 this.hireDate = hireDate;
}

public Double getSalary() {
 return salary;
}

public void setSalary(Double salary) {
 this.salary = salary;
}

public Employee getSupervisor() {
 return supervisor;
}

public void setSupervisor(Employee supervisor) {
 this.supervisor = supervisor;
}

public List<string> getPhoneNumbers() {
 return phoneNumbers;
}

public void setPhoneNumbers(List<string> phoneNumbers) {
 this.phoneNumbers = phoneNumbers;
}

public void readExternal(ObjectInput objectInput) throws IOException,
  ClassNotFoundException {

 this.firstName = objectInput.readUTF();
 this.lastName = objectInput.readUTF();
 this.socialSecurityNumber = objectInput.readUTF();
 this.department = objectInput.readUTF();
 this.position = objectInput.readUTF();
 this.hireDate = new Date(objectInput.readLong());
 this.salary = objectInput.readDouble();

 int attributeCount = objectInput.read();

 byte[] attributes = new byte[attributeCount];

 objectInput.readFully(attributes);

 for (int i = 0; i < attributeCount; i++) {
  byte attribute = attributes[i];

  switch (attribute) {
  case (byte) 0:
   this.supervisor = (Employee) objectInput.readObject();
   break;
  case (byte) 1:
   this.phoneNumbers = Arrays.asList(objectInput.readUTF().split(";"));
   break;
  }
 }

}

public void writeExternal(ObjectOutput objectOutput) throws IOException {

 objectOutput.writeUTF(firstName);
 objectOutput.writeUTF(lastName);
 objectOutput.writeUTF(socialSecurityNumber);
 objectOutput.writeUTF(department);
 objectOutput.writeUTF(position);
 objectOutput.writeLong(hireDate.getTime());
 objectOutput.writeDouble(salary);

 byte[] attributeFlags = new byte[2];

 int attributeCount = 0;

 if (supervisor != null) {
  attributeFlags[0] = (byte) 1;
  attributeCount++;
 }
 if (phoneNumbers != null && !phoneNumbers.isEmpty()) {
  attributeFlags[1] = (byte) 1;
```

```
    attributeCount++;
   }

   objectOutput.write(attributeCount);

   byte[] attributes = new byte[attributeCount];

   int j = attributeCount;

   for (int i = 0; i < 2; i++)
    if (attributeFlags[i] == (byte) 1) {
     j--;
     attributes[j] = (byte) i;
    }

   objectOutput.write(attributes);

   for (int i = 0; i < attributeCount; i++) {
    byte attribute = attributes[i];

    switch (attribute) {
    case (byte) 0:
     objectOutput.writeObject(supervisor);
     break;
    case (byte) 1:
     StringBuilder rowPhoneNumbers = new StringBuilder();
     for(int k = 0; k < phoneNumbers.size(); k++)
      rowPhoneNumbers.append(phoneNumbers.get(k) + ";");
     rowPhoneNumbers.deleteCharAt(rowPhoneNumbers.lastIndexOf(";"));
     objectOutput.writeUTF(rowPhoneNumbers.toString());
     break;
    }
   }

  }
}
```

Things to notice here :

- We implement the "writeExternal" method for marshalling the "Employee" object. All mandatory fields are written to the stream
- For the "hireDate" field we write only the number of milliseconds represented by this Date object. Assuming that the demarshaller will be using the same timezone as the marshaller the milliseconds value is all the information we need to properly deserialize the "hireDate" field. Keep in mind that we could serialize the entire "hireDate" object by using the "objectOutput.writeObject(hireDate)" operation. In that case the default serialization mechanism would kick in resulting in speed degradation and size increment for the resulting stream
- All the non mandatory fields ("supervisor" and "phoneNumbers") are written to the stream only when they have actual (not null) values. To implement this functionality we use the "attributeFlags" and "attributes" byte arrays. Each position of the "attributeFlags" array represents a non mandatory field and holds a "marker" indicating whether the specific field has a value. We check each non mandatory field and populate the "attributeFlags" byte array with the corresponding markers. The "attributes" byte array indicates the actual non mandatory fields that must be written to the stream by means of "position". For example if both "supervisor" and "phoneNumbers" non mandatory fields have actual values then "attributeFlags" byte array should be [1,1] and "attributes" byte array should be [0,1]. In case only "phoneNumbers" non mandatory field has a non null value "attributeFlags" byte array should be [0,1] and "attributes" byte array should be [1]. By using the aforementioned algorithm we can achieve minimal size footprint for the resulting stream. To properly deserialize the "Employee" object non mandatory parameters we must write to the steam only the following information :
  - The overall number of non mandatory parameters that will be written (aka the "attributes" byte array size – for the demarshaller to parse)
  - The "attributes" byte array (for the demarshaller to properly assign field values)
  - The actual non mandatory parameter values
- For the "phoneNumbers" field we construct and write to the stream a String representation of its contents. Alternatively we could serialize the entire "phoneNumbers" object by using the "objectOutput.writeObject(phoneNumbers)" operation. In that case the default serialization mechanism would kick in resulting in speed degradation and size increment for the resulting stream
- We implement the "readExternal" method for demarshalling the "Employee" object. All mandatory fields are written to the stream. For the non mandatory fields the demarshaller assigns the appropriate field values according to the protocol described above

For the serialization and deserialization processes we used the following four functions. These functions come in two flavors. The first pair is suitable for serializing and deserializing Externalizable object instances, whereas the second pair is suitable for serializing and deserializing Serializable object instances.

```
01 public static byte[][] serializeObject(Externalizable object) throws Exception {
02   ByteArrayOutputStream baos = null;
03   ObjectOutputStream oos = null;
04   byte[][] res = new byte[2][];
05
06   try {
07    baos = new ByteArrayOutputStream();
08    oos = new ObjectOutputStream(baos);
09
10    object.writeExternal(oos);
11    oos.flush();
12
13    res[0] = object.getClass().getName().getBytes();
14    res[1] = baos.toByteArray();
15
16   } catch (Exception ex) {
17    throw ex;
18   } finally {
19    try {
20     if(oos != null)
21      oos.close();
22    } catch (Exception e) {
```

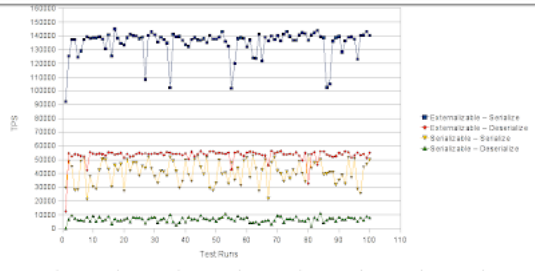
```
    e.printStackTrace();
   }
  }

  return res;
 }
```

```
public static Externalizable deserializeObject(byte[][] rowObject) throws Exception {
  ObjectInputStream ois = null;
  String objectClassName = null;
  Externalizable res = null;

  try {

   objectClassName = new String(rowObject[0]);
   byte[] objectBytes = rowObject[1];

   ois = new ObjectInputStream(new ByteArrayInputStream(objectBytes));

   Class objectClass = Class.forName(objectClassName);
   res = (Externalizable) objectClass.newInstance();
   res.readExternal(ois);

  } catch (Exception ex) {
   throw ex;
  } finally {
   try {
    if(ois != null)
     ois.close();
   } catch (Exception e) {
    e.printStackTrace();
   }

  }

  return res;

 }
```

```
public static byte[] serializeObject(Serializable object) throws Exception {
  ByteArrayOutputStream baos = null;
  ObjectOutputStream oos = null;
  byte[] res = null;

  try {
   baos = new ByteArrayOutputStream();
   oos = new ObjectOutputStream(baos);

   oos.writeObject(object);
   oos.flush();

   res = baos.toByteArray();

  } catch (Exception ex) {
   throw ex;
  } finally {
   try {
    if(oos != null)
     oos.close();
   } catch (Exception e) {
    e.printStackTrace();
   }
  }

  return res;
 }
```

```
public static Serializable deserializeObject(byte[] rowObject) throws Exception {
  ObjectInputStream ois = null;
  Serializable res = null;

  try {

   ois = new ObjectInputStream(new ByteArrayInputStream(rowObject));
   res = (Serializable) ois.readObject();

  } catch (Exception ex) {
   throw ex;
  } finally {
   try {
    if(ois != null)
     ois.close();
   } catch (Exception e) {
    e.printStackTrace();
   }

  }

  return res;

 }
```

Below we present a performance comparison chart between the two aforementioned approaches



The horizontal axis represents the number of test runs and the vertical axis the average transactions per second (TPS) for each test run. Thus higher values are better. As you can see by using the Externalizable approach you can achieve superior performance gains when serializing and deserializing compared to the plain Serializable approach.

Lastly we must pinpoint that we performed our tests providing values for all non mandatory fields of the "Employee" object. You should expect even higher performance gains if you do not use all the non mandatory parameters for your tests, either when comparing between the same approach and most importantly when cross comparing between the Externalizable and Serializable approaches.

Happy coding!

Justin

Related Articles :

- Java Best Practices – DateFormat in a Multithreading Environment
- Java Best Practices – Vector vs ArrayList vs HashSet
- Java Best Practices – String performance and Exact String Matching
- Java Best Practices – Queue battle and the Linked ConcurrentHashMap
- Java Best Practices – Char to Byte and Byte to Char conversions

Tagged with: JAVA BEST PRACTICES SERIALIZATION

(***0*** *rating,* ***0*** *votes*)

You need to be a registered member to rate this. 8 Comments 938 Views Tweet it!

LIKE THIS ARTICLE? READ MORE FROM JAVA CODE GEEKS

8 Leave a Reply

Join the discussion...

Subscribe ▾ newest oldest most voted

Guest

Proagent

thanks for this nice article,
can you tell us what the tool you used for performance comparison ?

+ 0 − Reply 5 years ago

Guest

Byron Kiourtzoglou

I have created a in-house performance measurement app

+ 0 − Reply 5 years ago

Guest

Proagent

cool , its open source ?

+ 0 − Reply 5 years ago

Guest

Byron Kiourtzoglou

Unfortunately no, its a proprietary code, but you can make your own very easily!

Regards

+ 0 − Reply 5 years ago

Guest

Andre

Tweaked your code a bit, this is 4 times faster roughly … package com.javacodegeeks.test;; import java.io.ByteArrayInputStream; import java.io.ByteArrayOutputStream; import java.io.DataInputStream; import java.io.DataOutputStream; import java.io.IOException; public class DataMessageTransmission_test { private DataMessageTransmission_test employee = this; private String firstName; private String lastName; private String socialSecurityNumber; private String department; private String position; private long hireDate; private Double salary; private String supervisor; private String[] phoneNumbers; private static byte[][]serial; public DataMessageTransmission_test() {} public DataMessageTransmission_test(String firstName, String lastName,String socialSecurityNumber, String department, String position, long hireDate, Double salary) { employee.firstName = firstName; employee.lastName = lastName; employee.socialSecurityNumber = socialSecurityNumber; employee.department = department; employee.position = position; employee.hireDate = hireDate; employee.salary… Read more »

+ 0 − Reply 5 years ago

Guest

Charles

Hello Andre,

Could you please email your code to play with? The one posted seems having some problem….
Thanks in advance.

Charles_L_chan (at) me (dot) com

+ 0 − Reply 5 years ago

Guest

Rüdiger Möller

You should checkout https://code.google.com/p/fast-serialization/ . This library outperforms manual serialization in many cases.

+ 0 − Reply 5 years ago

Johan Parent

Guest

Hi Justin, Great article! You show a some useful techniques to use the ObjectOutput and ObjectInput APIs to get the best out of the JDK serialization algorithm. Having tried fast-serialization myself I can confirm it is indeed very fast. Great piece of code. Application servers and other JavaEE technologies do not always allow the use of an alternative serialization mechanism. If you are bound to the default JDK serialization you may want to take a look at Externalizer4J. It optimizes the serialization using techniques similar to the ones described in this post. But it does so automatically by analyzing the… Read more »

+ 0 − Reply

3 years ago