JsonPerformanceVS

- 主流JSON引擎性能比较（GSON，FASTJSON，JACKSON，JSONSMART）

前言

测试目的： Purpose

> 测试当前主流Json引擎的序列化与反序列化性能，包括JSON,FASTJSON,JACKSON and SMARTJSON。

> Test the performance of the current mainstream engine，including JSON,FASTJSON,JACKSON and SMARTJSON

```
JSON序列化(Object => JSON)
JSON反序列化(JSON => Object)
```

预告结论：Conclusion pre：

- 1、 当数据小于 100K 的时候，建议使用 GSON。
- 2、 当数据100K 与 1M 的之间时候，建议使用各个JSON引擎性能差不多
- 3、 当数据大与 1M 的时候，建议使用 JACKSON 与 FASTJSON。

1. when the data size is less than 100k, i recommand you to use GSON.
2. when the data size is between 100k and 1M, choose what you like ,because their performance is similar.
3. when the data size is greater than 1M,i recommand you to use JACKSON or FASTJSON because of their high efficiency and stability.

任何错误与不如请不吝赐教，留言指出。谢谢。

一、硬件介绍 Hardware

```
MacBook Pro (13-inch, 2017, Four Thunderbolt 3 Ports
Processor: 3.1 GHz Intel Core i5
Memory: 8 GB 2133 MHz LPDDR3
disk : 256G
```

二、JVM配置 The Configuration of JVM

```
java version "1.8.0_161"
Java(TM) SE Runtime Environment (build 1.8.0_161-b12)
```

```
Java HotSpot(TM) 64-Bit Server VM (build 25.161-b12, mixed mode)

---

 -Xmx6g -Xms4g -XX:+UseG1GC
```

- JsonPerformanceVS,git代码地址

三、参与测试的JSON引擎介绍 The type of Json engine

```
// 选用目前最主流的JSON引擎:
public enum JsonTypeEnum {
    FASTJSON(0),
    GSON(1),
    JACKSON(2),
    JSONSMART(3);
}


--- 使用版本介绍，都是较新的并且使用人数最多的:
--- The most used and latest version
<!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-
databind (Mar 26, 2018) -->
    <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-databind</artifactId>
      <version>2.9.5</version>
    </dependency>


    <!-- https://mvnrepository.com/artifact/com.alibaba/fastjson (Mar 15, 2018) --
>
    <dependency>
      <groupId>com.alibaba</groupId>
      <artifactId>fastjson</artifactId>
      <version>1.2.47</version>
    </dependency>

    <!-- https://mvnrepository.com/artifact/com.google.code.gson/gson   (May 22,
2018) -->
    <dependency>
      <groupId>com.google.code.gson</groupId>
      <artifactId>gson</artifactId>
      <version>2.8.5</version>
    </dependency>

    <!-- https://mvnrepository.com/artifact/net.minidev/json-smart  (Mar 26, 2017)
-->
    <dependency>
      <groupId>net.minidev</groupId>
      <artifactId>json-smart</artifactId>
```

```
        <version>2.3</version>
    </dependency>
```

四、 测试步骤

```
    1. 数据准备:
    JSON序列化(Object => JSON):
        测试样本数量为1 10 100 1000 10000 100000个,
        处理同一个样本的时候，先把样本Java对象保存在文件中。每个 json引擎测试 1000 次，排列后，去掉前50与后50，对剩下的900 次结果求平均值作为最终的速度为最终的测试数据。
    JSON反序列化(JSON => Object)
        测试样本数量为1 10 100 1000 10000 100000个,
        处理同一个样本的时候，先把样本Java对象生成对应的各个引擎序列化后的字符串保存在文件（或者redis），然后读取出来进行反序列化。
        每个 json引擎测试 1000 次，排列后，去掉前50与后50，对剩下的900 次结果求平均值作为最终的速度为最终的测试数据。

    控制变量，生成样板数据sample，序列化到
        sampleSize_1_list_10_mapNum_10_samplesObject.txt
        sampleSize_10_list_10_mapNum_10_samplesObject.txt
             (根据 samples_object 生成4份json，下面类似 1 : 4)
                type_0_sampleSize_10_list_10_mapNum_10_samplesJson.txt
                type_1_sampleSize_10_list_10_mapNum_10_samplesJson.txt
                type_2_sampleSize_10_list_10_mapNum_10_samplesJson.txt
                type_3_sampleSize_10_list_10_mapNum_10_samplesJson.txt
        sampleSize_100_list_10_mapNum_10_samplesObject.txt
        sampleSize_1000_list_10_mapNum_10_samplesObject.txt
        sampleSize_10000_list_10_mapNum_10_samplesObject.txt
        sampleSize_100000_list_10_mapNum_10_samplesObject.txt
         (后期为了优化读取的速度，全部保存到redis里面去)

        序列化测试:
    1、 控制变量，把 samples_object.txt 反序列化生成对象 bean ，循环转 json，测试时间。
    2、 测试结果 result ，1000次测试 （每次测试4个引擎处理的是不同样本，1000个样本）
    3、 测试结果 result2 ，1000次测试 （每次测试4个引擎处理的是同一个样本，1个样本）

        反序列化测试:
    1、 控制变量，把 samples_samplesJson.txt 反序列化生成对象 bean 测试时间。
    2、 测试结果 result ，1000次测试 （每次测试4个引擎处理的是不同样本，1000个样本，无须用到 txt）
    3、 测试结果 result2 ，1000次测试 （每次测试4个引擎处理的是同一个样本，1个样本）
```

五、测试结果统计

```
    1、排除特殊干扰项，1000个测试结果，排序后，去除前50个小值与后50大值，900个结果取平均值。
```

- 同一样本：

> 序列化 ： anaysisOneSample(averageCost(ms))：

engine	sampleNum	1	10	100	1000	10000	100000
FASTJSON(0)	-	89.86	85.65	102.22	138.07	303.29	1068.61
GSON(1)	-	11.74	14.55	43.95	95.11	411.44	2129.12
JACKSON(2)	-	67.54	69.67	85.31	109.95	256.29	934.18
JSONSMART(3)	-	38.36	40.37	63.87	92.54	350.25	2005.55

> 反序列化： anaysisOneSample(averageCost(ms))

engine	sampleNum	1	10	100	1000	10000	100000
FASTJSON(0)	-	89.36	90.76	111.49	218.28	734.27	5924.24
GSON(1)	-	14.17	22.79	85.78	271.87	1090.57	7368.88
JACKSON(2)	-	81.17	84.46	109.7	232	792.78	5896.77
JSONSMART(3)	-	195.42	202.67	231.64	361.62	956.28	6498.74

- 1000个样本：

> 序列化 ： anaysisDifferentSample(averageCost(ms))：

engine	sampleNum	1	10	100	1000	10000	100000
FASTJSON(0)	-	90.64	91.76	95.9	152.34	247.38	966.73
GSON(1)	-	12.01	14.44	40.15	109.25	372.09	1789.78
JACKSON(2)	-	68.66	70.61	90.96	128.83	194.67	909.17
JSONSMART(3)	-	36.27	37.87	55.48	109.62	281.41	1772.59

> 反序列化： anaysisDifferentSample(averageCost(ms))

engine	sampleNum	1	10	100	1000	10000	100000
FASTJSON(0)	-	89.86	85.65	102.22	138.07	303.29	1068.82
GSON(1)	-	11.74	14.55	43.95	95.11	411.44	2129.63
JACKSON(2)	-	67.54	69.65	85.31	109.95	256.29	934.39
JSONSMART(3)	-	38.36	40.37	63.87	92.54	350.24	2005.36

性能总结：

> 1、序列化:

处理不变样本:
a. 在样本量为: 1 10 100 1000, 也就是对象大小为1k 10k 100k 1M 的时候, GSON的性能一直领先。
在这三个量级的情况下, GSON > JSONSMART > JACKSON > FASTJSON .
b. 在样本量为: 10000 100000, 也就是对象大小为: 10M 100M 的时候, GSON 和 JSONSMART 开始变慢, JSON 变慢最明显, 排倒数第一。
JACKSON > FASTJSON > JSONSMART > GSON

处理1000个样本:
测试结果同上。

2、反序列化

处理不变样本:
a.在样本量为1 10 100的时候, 也就是对象大小为1k 10k 100k 的时候, GSON的性能一直领先, 特别是样本量 1 的时候, 性能是排序最慢的 JSONSMART 的14倍, 排序次快的 JACKSON 的 5.8 倍。
在这三个量级的情况下, GSON > JACKSON > FASTJSON > JSONSMART
b.在样本量 1000 , 也就是对象大小为 1M 的时候, JSON 变慢的最明显。
在这个量级下, FASTJSON > JACKSON > GSON > SMARTJSON
c.在样本量 10000 , 也就是对象大小为 10M 的时候, JSON 变慢的最明显,SMARTJSON 反超 GSON。
在这个量级下, FASTJSON > JACKSON > SMARTJSON > GSON 。
d. 在样本量 100000 , 也就是对象大小为 100M 的时候,JACKSON 反超 FASTJSON
在这个量级下, JACKSON > FASTJSON > SMARTJSON > GSON 。
期待后期样本量 * 10情况下 JACKSON 与 FASTJSON 的PK

处理1000个样本:
a.在样本量为1 10 100的时候, 也就是对象大小为1k 10k 100k 的时候, GSON的性能一直领先.
在这三个量级的情况下, GSON > JSONSMART > JACKSON > FASTJSON
b.在样本量 1000 , 也就是对象大小为 1M 的时候, SMARTJSON 反超 GSON
在这个量级下, SMARTJSON > GSON > FASTJSON > JACKSON
c.在样本量 10000 100000, 也就是对象大小为 10M 100M的时候, JSON 变慢的最明显,JACKSON 与 FASTJSON 性能最优最稳定
在这个量级下, JACKSON > FASTJSON > SMARTJSON > GSON 。

总结:
当数据小于 100K 的时候, 建议使用 GSON。
当数据100K 与 1M 的之间时候, 建议使用各个JSON引擎性能差不多
当数据大与 1M 的时候, 建议使用 JACKSON 与 FASTJSON。

- 对象大小(FileSize(kb)):

engine	sampleNum	1	10	100	1000	10000	100000
objectseriable	-	1635	10369	94730	951066	9506635	94953271
FASTJSON(0)	-	1068	10541	102395	1033459	10336676	103254176
GSON(1)	-	10801	10801	105892	1068117	10682736	106719677
JACKSON(2)	-	1068	10541	102395	1033459	10336676	103254176

engine	sampleNum	1	10	100	1000	10000	100000
JSONSMART(3)	-	1085	10881	106968	1078781	10789216	107785985

- 对象大小(FileSize(human))：

engine	sampleNum	1	10	100	1000	10000	100000
objectseriable	-	1.6K	10K	93K	929K	9.1M	91M
FASTJSON(0)	-	1.0K	10K	100K	1.0M	9.9M	98M
GSON(1)	-	1.1K	11K	103K	1.0M	10M	102M
JACKSON(2)	-	1.0K	10K	100K	1.0M	9.9M	98M
JSONSMART(3)	-	1.1K	11K	104K	1.0M	10M	103M

- 空间占用大小结论：

```
a.有人常说，序列化后的文件大小一定比转json的小，其实不一样。在1个样本的时候，占了1.6k，占比最高。
b.随着样本量的上升，序列化占空间逐渐变小，变最小。
```

六、 注意事项；

```
    1. 为了避免垃圾回收带来的影响，每一次running只针对一个样板数据，调用4个json引擎
    2. json引擎具有强化能力，即使你换了样本，下次running同样效率会高很多，所以每一次测试都重启JVM，保证测试结果。
    3. 用G1垃圾收集器，定义好初始内存
    4. 对象序列化的目的：不同json转成的字符串，不一定能被其他json工具成功转换回来。如fastjson 把一个bean转成string，gson转回来后报错。
```

七、shell脚本

```
#!/bin/bash
# @author: Jeb_lin
# @Date: 2018-07-13


for i in {1..1000}
do

    java -jar -Xmx6g -Xms4g -XX:+UseG1GC JsonPerformanceVS.jar 100000 10 10
    echo 'shell, i-> '$i
done

echo 'OK'
```

七、答疑

1、为什么不写一个for循环1000次，而要重新启动JVM
答：一方面不想因为内存问题(GC)影响到测试结果，另一方面，你自己试试，基本循环到第5次之后，后面的速度都是10毫秒内的，不具备统计意义。无论事序列化还是反序列化，都会出现这种情况。

2、为什么写个 for 循环 1000遍会越来越快。
答：根据《深入了解Java虚拟机》第11章表述，被多次调用的方法与被多次执行的循环体，会触发JIT编译器进行程序的优化，最终将字节码转换为本地代码，大幅度提高执行效率。

3、为什么要把对象序列化后的信息写到redis
答：可以写到文件，但是每一次调用测试序列化的时候，读redis比读文件更省时间。

八、其他 Else

1、本例子的测试数据 The Testing data of this project
链接:https://pan.baidu.com/s/1YVamgy4LCIl9QiMffoJ4Ww 密码:puhe