# 性能测试

由于软硬件及其复杂，本结果只具有参考意义。

代码： <https://github.com/zhang-xzhi/perftest>

测试方法：

预热1：默认跑10000次，触发JIT。，根据当前avg时间调整下次loop的次数。期望一个suite可以在2s左右。

预热2：根据预热1计算的loop次数，重新跑一次，由于代码已经JIT过，理论上，本次run的时候avg会变小，根据当前avg时间调整下次loop的次数。期望一个suite可以在2s左右。

暂停1s,等待JIT完成。

实际执行。

时间单位为ns。

# JVM参数



java

-client/-server

-XX:+PrintCompilation

-Xms1024m

-Xmx1024m

-Xmn680m

-XX:SurvivorRatio=8

-XX:PermSize=240m

-XX:MaxPermSize=240m

-XX:+UseConcMarkSweepGC

-XX:+UseParNewGC

-XX:CMSInitiatingOccupancyFraction=70

-XX:+UseCMSInitiatingOccupancyOnly

-XX:CMSFullGCsBeforeCompaction=0

-XX:+UseCMSCompactAtFullCollection

-XX:+CMSClassUnloadingEnabled

-XX:+PrintGCDetails

-XX:+PrintGCDateStamps

# 异常

## 配置

loop=20000 suite=5 delta=0.2

## 结果

|  |  |  |
| --- | --- | --- |
| pertestcase | server | client |
| ExceptionOnlyCreateException\_1 | 712 | 741 |
| ExceptionOnlyCreateException\_2 | 849 | 833 |
| ExceptionOnlyCreateException\_4 | 838 | 895 |
| ExceptionOnlyCreateException\_8 | 990 | 1036 |
| ExceptionOnlyCreateException\_16 | 1195 | 1407 |
| ExceptionOnlyCreateException\_32 | 2040 | 2299 |
| ExceptionOnlyCreateException\_64 | 3351 | 3929 |
| ExceptionOnlyCreateException\_128 | 6252 | 7208 |
| ExceptionOnlyCreateException\_256 | 12076 | 13424 |
| ExceptionOnlyCreateException\_512 | 21119 | 26056 |
| ExceptionOnlyCreateException\_1024 | 43667 | 52557 |
| ExceptionWithTryCatch\_1 | 42 | 12 |
| ExceptionWithTryCatch\_2 | 9 | 10 |
| ExceptionWithTryCatch\_4 | 10 | 14 |
| ExceptionWithTryCatch\_8 | 16 | 24 |
| ExceptionWithTryCatch\_16 | 25 | 52 |
| ExceptionWithTryCatch\_32 | 45 | 100 |
| ExceptionWithTryCatch\_64 | 95 | 205 |
| ExceptionWithTryCatch\_128 | 181 | 408 |
| ExceptionWithTryCatch\_256 | 368 | 817 |
| ExceptionWithTryCatch\_512 | 664 | 1724 |
| ExceptionWithTryCatch\_1024 | 1294 | 3988 |
| ExceptionThrowAndCatch\_1 | 883 | 1070 |
| ExceptionThrowAndCatch\_2 | 978 | 959 |
| ExceptionThrowAndCatch\_4 | 994 | 1226 |
| ExceptionThrowAndCatch\_8 | 1508 | 1646 |
| ExceptionThrowAndCatch\_16 | 1915 | 2457 |
| ExceptionThrowAndCatch\_32 | 3654 | 4654 |
| ExceptionThrowAndCatch\_64 | 6273 | 8482 |
| ExceptionThrowAndCatch\_128 | 11627 | 17455 |
| ExceptionThrowAndCatch\_256 | 23781 | 31009 |
| ExceptionThrowAndCatch\_512 | 43768 | 61659 |
| ExceptionThrowAndCatch\_1024 | 87996 | 123532 |

## 简要分析

ExceptionOnlyCreateException 只创建异常。Deepth表示递归多少次。

ExceptionThrowAndCatch 创建异常，抛出，捕获。Deepth表示递归多少次。

ExceptionWithTryCatch 方式上有try catch块，但是实际运行中，无异常抛出。Deepth表示递归多少次。

创建异常由于要fillInStackTrace，所以较慢。

方法调用递归越深，时间消耗越大。

# 反射

## 配置

loop=20000 suite=5 delta=0.2

## 结果

|  |  |  |
| --- | --- | --- |
| pertestcase | server | client |
| ReflectionConstructor | 112 | 553 |
| ReflectionConstructorInvoke | 10 | 66 |
| ReflectionConstructor\_DirectAccess | 8 | 15 |
| ReflectionField | 491 | 466 |
| ReflectionFieldInvoke | 6 | 58 |
| ReflectionField\_DirectAccess | 5 | 5 |
| ReflectionMethod | 222 | 658 |
| ReflectionMethodInvoke | 6 | 15 |
| ReflectionMethod\_DirectAccess | 5 | 40 |
| ReflectionNewInstance | 13 | 109 |

## 简要分析

测试了构造函数，方法，field的3种操作方式。

S1:

ReflectionConstructor

ReflectionField

ReflectionMethod

为使用class.get得到Constructor,Field,Method对象。

S2:

ReflectionConstructorInvoke

ReflectionFieldInvoke

ReflectionMethodInvoke

为使用Constructor,Field,Method对象来构造对象，取值，调用方法。

S3:

ReflectionConstructor\_DirectAccess

ReflectionField\_DirectAccess

ReflectionMethod\_DirectAccess

为使用java代码直接构造对象，取值，调用方法。

可以看到S1:S2:S3各自相差一个数量级。

ReflectionNewInstance 由于class的实现对Constructor对象做了缓存，因此，class.newInstance的时间比ReflectionConstructorInvoke稍慢一点。

server 和 client相比，性能明显提升。

# ArrayCopy

## 配置

loop=20000 suite=5 delta=0.2

## 结果

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| length of bytes | loop copy  (server) | loop copy  (client) | SystemArrayCopy  (server) | SystemArrayCopy  (client) |
| 512 | 50 | 864 | 23 | 45 |
| 1024 | 78 | 1645 | 65 | 92 |
| 1536 | 104 | 2520 | 59 | 90 |
| 2048 | 123 | 3333 | 72 | 96 |
| 2560 | 143 | 3884 | 90 | 105 |
| 3072 | 166 | 4465 | 124 | 198 |
| 3584 | 186 | 5257 | 130 | 150 |
| 4096 | 222 | 6063 | 168 | 409 |
| 4608 | 233 | 7293 | 165 | 182 |
| 5120 | 304 | 7488 | 204 | 192 |
| 5632 | 290 | 8472 | 205 | 206 |
| 6144 | 304 | 9557 | 217 | 227 |
| 6656 | 318 | 9901 | 234 | 262 |
| 7168 | 340 | 10397 | 250 | 268 |
| 7680 | 361 | 12160 | 267 | 277 |
| 8192 | 420 | 11935 | 368 | 812 |
| 8704 | 413 | 12792 | 306 | 314 |
| 9216 | 428 | 13883 | 318 | 341 |
| 9728 | 455 | 14229 | 331 | 353 |
| 10240 | 475 | 14958 | 355 | 369 |

## 简要分析

在client mode下，Loop方式copy 字节数组和System.arrayCopy相比是很慢的。

在server mode下，Loop方式copy 字节数组和System.arrayCopy相比稍慢一点。

# Time

## 配置

loop=20000 suite=5 delta=0.2

## 结果

|  |  |  |
| --- | --- | --- |
| pertestcase | server | client |
| Time\_CurrentTimeMillis | 19 | 20 |
| Time\_NanoTime | 36 | 36 |
| Time\_NewDate | 27 | 30 |

# String

## 配置

loop=1000 suite=5 delta=0.2

## 结果

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| loop length | StrBuF  (S) | StrBuf  (C) | StrBuilder  (S) | StrBuilder  (C) | Str  (S) | Str  (C) |
| 1 | 113 | 37 | 81 | 35 | 5089 | 6313 |
| 2 | 71 | 81 | 29 | 101 | 14557 | 12666 |
| 4 | 149 | 103 | 105 | 92 | 20481 | 55913 |
| 8 | 345 | 227 | 120 | 196 | 81924 | 108892 |
| 16 | 540 | 411 | 228 | 425 | 313046 | 454132 |
| 32 | 604 | 844 | 470 | 800 | 1357065 | 1854443 |