各种API性能/编码优化技巧

温绍锦

微博 http://weibo.com/wengaotie

性能计数单位

- 1 sec
  = 1,000 ms 毫秒
  = 1,000,000 μs 微秒
  = 1,000,000,000 ns 纳秒
  = 1,000,000,000,000 ps 皮秒

- 访问一次内存（Cacheline）大约10~15ns

获取当前时间API性能

操作	耗时(nano)	说明
System.currentMillis()	52	在linux-2.6.18下慢10倍，大约534ns
System.nano()	47	在linux-2.6.18下慢12倍，大约658ns

- 在linux老版本（2.6.18）中，System.currentMillis的性能是很差的，如果你的程序中，大量调用需要获取当前时间，请升级操作系统版本。Linux-2.6.32版本以上都是好的.
- 关键在于升级操作系统

异常性能

操作	耗时(nano)	
new Exception()	1061	
new Exception(), throw, catch	1092	
new Exception().getStackTrace()	5786	获取堆栈信息
Thread.getStackTrace()	6560	
new Exception(), throw, catch/none fillStackTrace	23	重载为空的fillStackTrace方法
Throw try cach	9	抛出的是同一个Exception

- 异常的开销在于fillStackTrace方法
- fillStackTrace方法在Exception的构造函数中调用
- try/cach/throw都是非常快的
- Thread.getStackTrace很慢，和Exception.getStackTrace一样慢
- **关键在于fillStackTrace和getStackTrace这两个方法**

反射性能

操作	耗时(nano)	
Class.newInstance	12	
Constructor.new	13	
Method.invoke	4	缓存了Method
Field.get	15	缓存了Field
Class.forname	560~1000	系统类会快一些
Class.getField	243	
Class.getMethod	250	
Class.getConstrcutor	100	

- 反射的性能是很好的，只要缓存了Class/Constructor/Method/Field
- Class.forname/getField/getMethod/getConstructor都很慢
- **关键在于缓存Class/Constructor/Field/Method等元数据对象**

字符串性能（一）

第一种写法：

```
for (int i = 0, len = str.length(); i len; ++i) {
      char ch = str.charAt(i);
}
```

第二种写法：

```
char[] chars = str.toCharArray();
for (int i = 0; i < chars.length; ++i) {
      char ch = chars[i];
}
```

第一种写法比第二种写法速度更快。长度为1000的String，第一种写法250ns，第二种写法会产生gc，700~7000都可能。

结论：charAt比toCharArray然后访问char数组快

字符串性能

- StringBuilder是非线程安全版本的StringBuffer，性能更好
- 在方法内使用StringBuffer，可能会被javac编译时替换为StringBuilder
- 他们有共同被instrinsic的方法
  - 三个构造函数： () / (int) / (String)
  - 三个append方法：append(char) / append(int) / append(String)
  - toString方法

并发性能（一）

操作	耗时 nano
volatile int ++	15
volatile long int ++	15
AtomicLong.increment()	17
AtomicInteger.increment()	17
ReentrantLock.lock/unlock	38
Unfair ReentrantLock.lock/unlock	37
synchronized	36

主要消耗在一次Cacheline上

主要消耗在两次Cacheline上

- volatile 变量读取的性能和普通变量一样，修改的性能很差
- JDK 6/7中，syncrhonized性能比ReentrantLock略好，在JDK 5中则很差。

并发性能（二）

- ReentranLock有两种模式，unfair（缺省）和fair。
- 非竞争情况下，fair和unfair差不多
- 竞争情况下，unfair模式性能好得多的，竞争越激烈，相差越多，几倍到数十百倍。
- Unfair模式tryLock(long, TimeUnit)方法会有饥饿现象

并发性能（三）-公平

- 性能
  - synchronized > Unfair Lock > Fair L ock
- 公平性
  - Fair Lock > synchronized > Unfair Lock
- 结论
  - 如果不需要多个Condition和tryLock，使用synchronized

并发性能（四）-统计信息

通过ManagementFactory

.getThreadMXBean()

.getThreadInfo(currentThreadId)

来获取线程的如下信息：

blockedTime

blockedCount

waitedTime

waitedCount

通过这些信息来了解你的程序多线程之间的竞争信息，这些数值越小越好。

对象大小

```
class UUID {
    long mostSigBits;
    long leastSigBits;
}
```

```
class UUID {
    byte[] data; // length 16
}
```

第一种写法更节省内存

对象大小

```
class Stats {
    AtomicLong count;
}

class Stats {
    volatile long count;
    static AtomicLongFieldUpdater countUpdater;
    long incrementAndGet() {
        countUpdater.incrementAndGet(this);
    }
}
```

第二种写法比较麻烦，但是更节省内存

对象大小

- 使用int表示IP地址
- LongHashMap/ConcurrentLongHashMap

数组操作性能

- 当数组的长度大于8做拷贝操作时，尽量使用System.arrayCopy提升性能。

```
for (int i = 0; i < src.length; ++i) {
    dest[i] = src[i];
}
```

当src.length，也就是copy的数据量很小时，上面的操作比System.arrayCopy快。

安全API性能-随机数

算法	耗时 nano
UUID.randomUUID	2,934
secureRandom.nextBytes(new byte[16])	2936
Random.nextBytes(new byte[16])	76
threadLocalRandom.nextBytes(new byte[16])	42

- UUID.randomUUID是通过SecureRandom来实现的，性能和SecureRandom.nextBytes基本一致
- JDK7中的ThreadLocalRandom性能非常好，如果项目中需要，可以backport。

安全API-对称加密

算法	KeySize	数据长度	加密耗时(nano)	解密耗时(nano)
AES	128	1024	12,841	14,388
AES	192	1024	14,772	15,965
AES	256	1024	16,672	17,859
DES	56	1024	37,891	39,260
DESede	168	1024	113,259	114,719
Blowfish	128	1024	22,970	20,601
RC2	20	1024	41,096	30,573

- AES加密算法比DES/DESede/Blowfish/RC2都要好
- AES在三种密钥长度128/192/256下的性能差别不大

RSA算法性能

算法	KeySize	数据长度	加密耗时(nano)	解密耗时(nano)
RSA	512	32	38,372	308,731
RSA	768	32	61,475	816,705
RSA	1024	32	102,841	1,668,865
RSA	2048	32	323,458	9,973,702
RSA	3072	32	679,590	
RSA	4096	32	1,167,000	
RSA	6144	32	2,486,000	

- RSA支持的KeySize范围很广512~65536，加解密的性能随着KeySize变大迅速下降
- RSA算法的解密比加密慢得多，大约一个数量级

安全API-摘要算法

算法	数据长度	耗时(nano)
MD2	1024	181,057
MD5	1024	4,747
SHA-1	1024	8,059
SHA-256	1024	13,350
SHA-384	1024	9,771
SHA-512	1024	9,800

- 常用的MD5和SHA性能都相当好
- MD5比SHA大约慢一倍

优化技巧

- 尽量使用final/static
- 尽量调用JVM的intrinsic方法（查JDK源码的vmSymbols.hpp文件）
- 分支判断，针对热点分支进行特别处理
- 在关键的性能处理时，if通常比switch更靠谱。
- 通过intel vtune等更工具看代码的热点，重点优化调用频率高的代码