

记一次JavaSDK性能从8000提升至30000的过程

原创 李辉忠 FISCO BCOS开源社区 2月11日



李辉忠

FISCO BCOS 研发负责人

— AUTHOR | 作者 —

缘起

FISCO BCOS在中国信通院可信区块链测评中达到2万+ TPS的交易处理能力，在同类产品中处于领先水平。此次测试标的是底层平台，目的是压测得出底层平台的性能上限，主要评估目标是底层平台的交易处理能力。

交易构造是由客户端（集成了SDK）完成，客户端通常可以非常容易实现平行扩展。

SDK要完成交易构造的过程，实现交易组包、编码、签名、发送等一系列操作，这些过程本身是无状态的，客户端可以通过多线程的方式进行扩展，一个客户端性能达到瓶颈，可以增加更多客户端进行扩展。

虽然“堆机器”的平行扩展方式可以解决发送端的性能问题，但是机器本身就是珍贵的资源，进一步优化算法效率，提高资源利用率，将大有裨益。于是，我们打算先测试一下JavaSDK目前的性能，主要评测生成交易的性能。

生成交易包括交易组包、参数编码、交易编码、交易签名等过程，其中，交易签名是最重要的环节，测试数据如下：

8核机器，测试本地生成50W笔交易的耗时

完全并行：每秒生成**8498笔**交易，平均每笔交易耗时0.12毫秒

完全串行：每秒生成1504笔交易，平均每笔交易耗时0.66毫秒

对比C++实现交易签名，这个性能实属不高。根据以往经验判断，这里大有优化空间啊！于是团队开始了一次JavaSDK的性能优化之路。

过程

//////////

关于性能优化，社区做过多次分享，可以补充阅读下列文章：

[FISCO BCOS共识优化之路](#)

[区块链的同步及其性能优化方法](#)

[FISCO BCOS中交易池及其优化策略](#)

[FISCO BCOS性能优化——工具篇](#)

[FISCO BCOS的速度与激情：性能优化方案最全解密](#)

[让木桶没有短板，FISCO BCOS全面推进并行化改造](#)

这些性能优化的过程中，感触最深的有两句话：

『过早的优化是万恶之源』

『没有任何证据支撑的优化是万恶之源』

优化要靠数据说话，而获取数据需要靠有效的分析工具，所以此次JavaSDK的性能优化，首要任务就是确定采用什么性能分析工具。

工具：发现java自带的分析工具挺管用

先把热点确定出来吧 @octopuswang(王章)

星期一下午 5:40

Jprofiler 这个工具可以试试

octopuswang(王章)

已经在跑了，java有自带的工具 HPROF

采用HPROF跑了一次，得到下面这样的数据报告：

```
CPU SAMPLES BEGIN (total = 418417) Mon Feb 3 18:39:03 2020
rank self accum count trace method
1 45.42% 45.42% 190034 301592
java.math.MutableBigInteger.divideKnuth
2 9.52% 54.93% 39816 301222
sun.nio.ch.EPollArrayWrapper.epollWait
3 2.51% 57.44% 10500 301555
java.math.MutableBigInteger.divideMagnitude
4 1.56% 59.01% 6546 301550
java.math.MutableBigInteger.divideMagnitude
5 1.52% 60.52% 6341 301885
java.security.AccessController.doPrivileged
6 1.34% 61.86% 5606 300734 java.util.Arrays.copyOfRange
7 1.30% 63.16% 5427 301559 java.util.Arrays.copyOfRange
```

数据显示，热点在很底层的库函数。这个结论并不符合预期，说明 SDK本身代码不是热点，性能优化比较难入手。

octopuswang(王章)

跑了几组数据，跟这个结果类似

octopuswang(王章)

这个工具结果不太友好，找不到图形界面工具分析

octopuswang(王章)

java.math.MutableBigInteger.divideKnuth 大头在这块，整形计算

星期一 下午 8:37

还有救么。。。

星期一 下午 8:44

octopuswang(王章)

这些明显不是我们自己的代码，我打算找个可视化工具再看下具体是在那里，能不能优化明天给个结论 出来

幸运的是，很快好消息就来了：采用java自带的另一个工具jvisualvm，较好地可视化输出性能分析数据，同时也很直观地显示出JavaSDK本身存在热点。

热点 - 方法	自用...	自用...	自用...	总时间	...
org.fisco.bcos.web3j.tx.ExtendedRawTransactionManager.createTransaction()	...	(41.9%)	1,533,9...	188,597,115 ms	1...
org.fisco.bcos.web3j.crypto.gm.sm2.crypto.asymmetric.SM2Algorithm.createRandom()	...	(40.1%)	1,490,0...	180,208,293 ms	1...
org.bouncycastle.math.ec.ECFieldElementFp.modReduce()	...	(7.4%)	33,203,...	33,203,585 ms	33...
jdk.internal.misc.Unsafe.park[native]()	...	(4.6%)	248 ms	20,855,905 ms	24...
org.bouncycastle.math.ec.ECFieldElementFp.modAdd()	...	(1%)	4,271,9...	4,271,907 ms	4...
org.bouncycastle.math.ec.ECFieldElementFp.modMult()	...	(0.8%)	3,730,6...	36,355,693 ms	36...

分析：找到一个“伪热点”

[illegible]

@wheatli(李辉忠) @catli(李陈希) 热点在这两个函数，这两个函数都使用 SecureRandom 生成随机数，这个类的随机数非常的慢。

createTransaction 随机数用于交易的 nonce 参数。createRandom 随机数用于国密签名接口。

If you want a cryptographically strong random numbers in Java, you use `SecureRandom`. Unfortunately, **SecureRandom can be very slow**. If it uses `/dev/random` on Linux, it can block waiting for sufficient

创建交易的 nonce 参数没必要用这么强的随机数，可以优化下。

通过jvisualvm工具分析出来，最大的热点是在生成随机数，这让我等有些吃惊。

catli(李陈希)

随机数竟然这么耗时。。可以试下 uuid

上面分析的是国密版本，对于非国密如何呢？跑了一遍，热点依旧可见在createTransaction，那就试试uuid吧。

octopuswang(王章)

命令	命令时间	命令时间	命令时间	命令时间
my \$time_start = (time);	0.000000	0.000000	0.000000	0.000000
my \$time_end = (time);	0.000000	0.000000	0.000000	0.000000
my \$time_diff = (\$time_end - \$time_start);	0.000000	0.000000	0.000000	0.000000
my \$time_diff = (\$time_end - \$time_start);	0.000000	0.000000	0.000000	0.000000

octopuswang(王章)

非国密的。。

按照李大神的思路，nonce 换成 uuid 算法试一下

但是，打脸，来得是那叫一个快，UUID的实现也是基于SecureRandom。

晕死，用 UUID 跑的结果跟 `SecureRandom` 一样，还一直在纠结，刚翻了下 UUID 的代码，直接就看到这句 `static final SecureRandom numberGenerator = new SecureRandom();` 🤔

群聊出现好长时间的静默.....

那就研究一下 `SecureRandom` 吧，看看为什么它会这么慢，也了解 Java 随机数的一些相关背景知识。

终于，下面这个知识点让我们重拾兴奋。

(<https://zhuanlan.zhihu.com/p/72697237>)

性能检测

简析，基准：100000 随机数，单线程

- 1、`Random`：2 毫秒
- 2、`ThreadLocalRandom`：1 毫秒
- 3、`SecureRandom`
 - 1) 默认算法，即 `SHAR1PRNG`：80 毫秒左右。
 - 2) `NativePRNG`：90 毫秒左右。
- 4、`SplittableRandom`：1 毫秒

我那句“采用 `ThreadLocalRandom` 替换 `SecureRandom` 计算 Nonce”还未敲字出去，新一轮打脸已经开始了。

octopuswang(王章)

ThreadLocalRandom 这个下午已经跑过了，这个跟 Random 的效果差不多，非国密替换之后跟李大神同等测试条件下并行可以跑到 9500-10000 之间。
其他的明天继续看下。

这个方向的讨论为本次性能优化之旅打开了另一扇思考之门~

octopuswang(王章)

@catli(李陈希) 李大神，这个测试程序也有问题，这个是根据之前合约压力测试的程序修改的，里面会启动非常多的线程，不适合用来做这种 cpu 密集型的压力测试，线程调用会占用很多的资源

catli(李陈希)

是说如果只启动较少数量的线程，性能还会提高？

octopuswang(王章)

我觉得是，这个测试里面其他逻辑太多了

catli(李陈希)

其他逻辑占用的计算资源应该不算多吧？

octopuswang(王章)

不只是线程，里面还有加锁操作

讨论很有道理，再仔细看数据，从数据层面也印证了这个想法。

wheatli(李辉忠):

热点 - 方法	自用时间	自用时间 (CPU)	总时间	总时间 (CPU)
org.fisco.bcos.sdk.jvmlib.bcosclient.BcosClient.createTransaction ()	26.888 ms	6.188 ms	30.188 ms	
jdk.internal.reflect.GeneratedMethodAccessor47.invoke ()	0.000 ms	100.317 ms	0.000 ms	
org.bouncycastle.math.ec.custom.sec.SecP256K1Field.multiply ()	45.918 ms	38.109 ms	84.029 ms	
org.bouncycastle.math.ec.custom.sec.SecP256K1Field.square ()	30.499 ms	38.118 ms	68.618 ms	

根据这个来看，createTx的总耗时，和后面的两个是相当的

所以即使优化了它，也不会有太大的提升

感觉还是没有摸清门道，还需再挖一下

实验测试证明这是一个正确的方向，线程数降下来，这个热点就消失了。随机数那里出现热点是因为压测并发线程数开启太多，太多线程抢占资源导致随机数获取较慢。

再分析：找到令人震惊的热点

经过第一轮分析，算是找到一个“伪热点”，但性能提升依旧收效甚微。革命尚未成功，同志还需努力！

降低线程数再跑一次性能分析，得到性能数据如下：

热点 - 方法	自用时间 [%]	自用时间	自用时间 (CPU)	总时间	总时间 (CPU)
org.bouncycastle.math.ec.custom.sec.SecP256K1Field.multiply ()	84.98...	84.987 ms	84.987 ms	104.104 ms	104.104 ms
org.bouncycastle.math.ec.custom.sec.SecP256K1Field.square ()	56.26...	56.269 ms	56.269 ms	71.811 ms	71.811 ms
org.bouncycastle.math.ec.custom.sec.SecP256K1Field.reduce ()	36.05...	36.059 ms	36.059 ms	36.059 ms	36.059 ms
jdk.internal.reflect.GeneratedMethodAccessor47.invoke ()	32.67...	32.677 ms	32.677 ms	32.677 ms	32.677 ms
org.fisco.bcos.sdk.jvmlib.bcosclient.BcosClient.createTransaction ()	21.95...	21.951 ms	21.951 ms	238.662 ms	238.662 ms
org.bouncycastle.math.ec.custom.sec.SecP256K1Field.square ()	16.05...	16.057 ms	16.057 ms	16.057 ms	16.057 ms
org.bouncycastle.math.ec.custom.sec.SecP256K1Field.multiply ()	13.57...	13.576 ms	13.576 ms	88.309 ms	88.309 ms
org.bouncycastle.math.ec.custom.sec.SecP256K1Field.multiplyAddToExt ()	11.05...	11.055 ms	11.055 ms	11.055 ms	11.055 ms
org.bouncycastle.math.ec.custom.sec.SecP256K1Field.square ()	9.864...	9.864 ms	9.864 ms	13.529 ms	13.529 ms

乍一看，热点分布又是很底层的基础库操作，焦虑又弥漫心头...

伴着凌晨的静谧，群聊再一次陷入了无言的沉默.....直到，我又抛出一个疑问。

函数 - 方法	占用时间 (%)	占用时间	占用时间 (CPU)	占用时间	占用时间 (CPU)
org.bouncycastle.math.ec.custom.sec.Sec256K1Field.multiply()	84.88	132.29s	84.887 ms	134.104 ms	134.104 ms
org.bouncycastle.math.ec.custom.sec.Sec256K1Field.square()	16.28	11.47s	16.289 ms	17.911 ms	17.911 ms
org.bouncycastle.math.ec.custom.sec.Sec256K1Field.reduce()	26.83	19.49s	26.839 ms	28.089 ms	28.089 ms
jdk.internal.reflect.GeneratedMethodAccessor1.invoke()	32.67	9.99s	32.677 ms	33.677 ms	33.677 ms
org.bouncycastle.math.ec.custom.sec.Sec256K1Field.add()	21.95	15.75s	21.951 ms	23.052 ms	23.052 ms
org.bouncycastle.math.ec.fiat.sabfvm.FiatSabFvm	14.95	10.75s	14.957 ms	16.057 ms	16.057 ms
org.bouncycastle.crypto.signers.ECDSASigner.generateSignature()	13.97	12.49s	13.978 ms	15.209 ms	15.209 ms
org.bouncycastle.math.ec.custom.sec.Sec256K1Field.multiplyFieldSub()	11.98	12.99s	11.989 ms	13.089 ms	13.089 ms
org.bouncycastle.math.ec.custom.sec.Sec256K1Field.square()	9.994	12.99s	9.994 ms	11.529 ms	11.529 ms

SDK为什么会有recover？不是只有签名么

238.750ms，好像就差不多等于签名几行的总耗时之和

是不是就是recover调用他们这几个

章鱼哥（王章）秒回，团队的激情苏醒了~为他这个回复速度，也为这个回复结果！

octopuswang(王章)

```
public Sign.SignatureData signMessage(byte[] message, ECKeypair keyPair) {
    BigInteger privateKey = keyPair.getPrivateKey();
    BigInteger publicKey = keyPair.getPublicKey();

    byte[] messageHash = Hash.sha3(message);

    ECDSASignature sig = sign(messageHash, privateKey);
    // Now we have to work backwards to figure out the recid needed to recover the signature.
    int recid = -1;
    for (int i = 0; i < 4; i++) {
        BigInteger k = Sign.recoverFromSignature(i, sig, messageHash);
        if (k != null && k.equals(publicKey)) {
            recid = i;
            break;
        }
    }
}
```

签名的时候有这个调用

签名算法的实现，居然是先签名然后再做验签，而验签就是为了获得一个叫recoveryID的值（关于ECDSA的recovery原理，将在另一篇详细展开）。

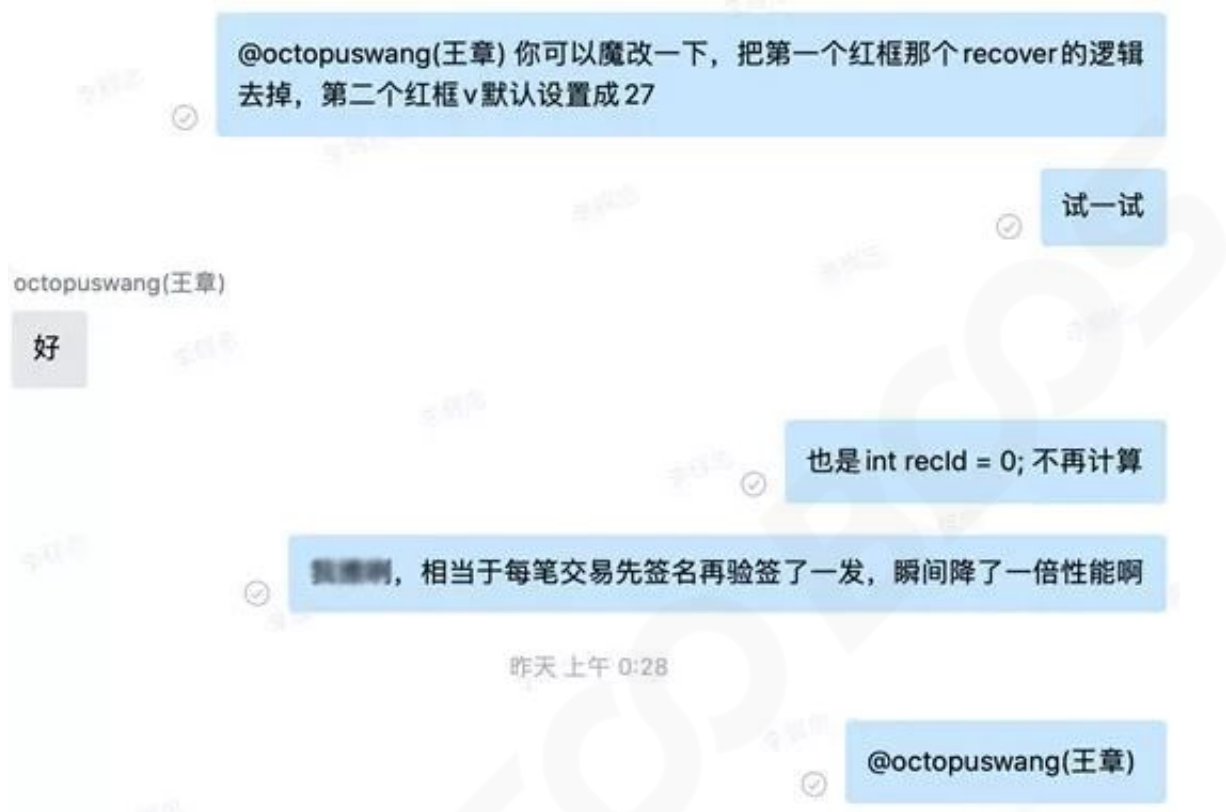
这里讲述一下我们为什么会兴奋。

recoveryID设置的目的是为了给未来使用者可以从签名快速恢复出公钥，如果没有这个recoveryID，恢复公钥就需要遍历4种可能性，然而，这里的实现方式在生成签名的时候是通过遍历查找来计算出recoveryID。

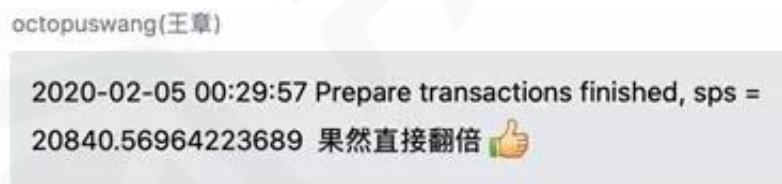
这种做法完全没有减小实际开销，只是采用“乾坤大挪移”把开销转移到签名环节了而已。实际上，recoveryID是有更快的计算方式可得到的，下一节可见。【这部分代码继承自web3j，之前没有

深入考究其实现方式，当前web3j还仍然是这种实现】

出于老码农的本能，发现症状就想要第一时间先搞清楚问题影响边界在哪里，于是就有了这样的分析和尝试：



同样老司机的章鱼哥快速给出了可喜的结论：



到这里，心里有底了！至少把recoverFromSignature干掉是能够翻倍提升性能的，至于怎么干掉，额...再说呗。

出于好奇，想再看看此时的性能数据会怎样（幸亏好奇了一下）。



©

©

30062.53006253006

CPU 样例 线程 CPU 时间		线程 Dump				
热点 - 方法		自用时间...	自用时间	自用时间...	总时间	总时间...
org.fisco.bcos.channel.test.parallel.precompile.DagTransfer.userTransferSeq()		...	(0%)	0.000 ms	183,746 ms	181,982 ms
org.fisco.bcos.channel.test.parallel.precompile.PerformanceDTTest\$.run()		...	(0%)	0.000 ms	183,746 ms	181,982 ms
org.fisco.bcos.web3j.tx.Contract.createTransactionSeq()		...	(0%)	0.000 ms	182,332 ms	180,568 ms
org.fisco.bcos.web3j.tx.ManagedTransaction.createSeq()		...	(0.1%)	176 ms	177,762 ms	175,999 ms
org.fisco.bcos.web3j.tx.ExtendedRawTransactionManager.sign()		...	(0.2%)	489 ms	176,408 ms	174,644 ms
org.fisco.bcos.web3j.crypto.ExtendedTransactionEncoder.signMessage()		...	(0%)	0.000 ms	174,772 ms	173,009 ms
org.fisco.bcos.web3j.crypto.ECDSASign.signMessage()		...	(0%)	66.4 ms	166,518 ms	164,755 ms
org.fisco.bcos.web3j.crypto.ECDSASign.sign()		...	(0%)	0.000 ms	163,011 ms	161,247 ms
org.bouncycastle.crypto.signers.ECDSASigner.generateSignature()		...	(12%)	24,173 ms	162,556 ms	160,793 ms
org.bouncycastle.math.ec.AbstractECMultiplier.multiply()		...	(0%)	0.000 ms	108,065 ms	106,302 ms
org.bouncycastle.math.ec.FixedPointCombMultiplier.multiplyPositive()		...	(1%)	1,936 ms	107,353 ms	105,589 ms
org.bouncycastle.math.ec.custom.sec.SecP256K1Point.twicePlus()		...	(0.7%)	1,497 ms	81,421 ms	81,421 ms
org.bouncycastle.math.ec.custom.sec.SecP256K1Point.add()		...	(1.3%)	2,648 ms	49,771 ms	49,771 ms
org.bouncycastle.math.ec.custom.sec.SecP256K1Field.multiply()		...	(17.8%)	35,856 ms	42,653 ms	42,653 ms
org.bouncycastle.math.ec.custom.sec.SecP256K1Point.twice()		...	(0.3%)	558 ms	30,510 ms	30,510 ms
org.bouncycastle.math.ec.custom.sec.SecP256K1Field.square()		...	(11.7%)	23,489 ms	29,290 ms	29,290 ms
org.bouncycastle.math.ec.custom.sec.SecP256K1Curve\$.lookup()		...	(8.8%)	17,587 ms	19,935 ms	19,935 ms
jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke()		...	(0%)	0.000 ms	18,798 ms	18,798 ms
jdk.internal.reflect.GeneratedMethodAccessor14.invoke()		...	(0%)	0.000 ms	18,612 ms	18,612 ms
jdk.internal.reflect.GeneratedMethodAccessor18.invoke()		...	(0.1%)	176 ms	18,436 ms	18,110 ms

凌晨一点，总算收获满意的成果，心情是棒棒的...

一秒之后，脑回路蹦出来个问题“国密的会怎样呢？”

明天再说吧（国密的优化会在另一篇再详细讲）。



填坑：recoveryID的计算方法

由于对Java密码学算法库（bc-java）不够熟悉，此处也遇到不少坑，最终通过继承密码算法库，将更多所需参数暴露返回给上层，总算实现了Java版本的recoveryID计算。

```

1 // Now we have to work backwards to figure out the recId needed to recover
2     ECPoint ecPoint = sig.p;
3     BigInteger affineXCoordValue = ecPoint.normalize().getAffineXCoord();
4     BigInteger affineYCoordValue = ecPoint.normalize().getAffineYCoord();
5
6     int recId = affineYCoordValue.and(BigInteger.ONE).intValue();
7     recId |= (affineXCoordValue.compareTo(sig.r) != 0 ? 2 : 0);
8     if (sig.s.compareTo(halfCurveN) > 0) {
9         sig.s = Sign.CURVE.getN().subtract(sig.s);
10        recId = recId ^ 1;
11    }

```

关于ECDSA算法的recover机制和recoveryID生成原理，将在后续的推送中详细展开，敬请期待。

后话

//////////

每一次做性能优化，都是一次很爽的体验，少不了熬夜，但永不缺激情。对代码进行抽丝剥茧，经历反复多次的发现瓶颈、无情打脸、重拾信心过程，最后到达柳暗花明。

再次用那两句话结尾：

过早的优化是万恶之源

没有任何数据支撑的优化是万恶之源

共勉！

..... **FISCO BCOS**

下载地址↓↓↓

<https://github.com/FISCO-BCOS/FISCO-BCOS>

