高性能纠删码编码

2018-04-12 02:54:00

https://www.jianshu.com/p/024f96a0d56a

ISA-L 是什么东西？

intel®-storage-acceleration-library Intel存储加速库，包括两个大类：加密和非加密的。非加密的 crc，izip，erase-code，加密的包括sha512，sha256，md5，sha1等。

核心技术就是使用intel sse/avx/avx2/avx256的扩展指令，并行运算多个流的方法。单线程比openssl要快2～8倍。

现在ISA-L已经开源：

https://github.com/01org/isa-l

https://github.com/01org/isa-l_crypto

---

http://www.zhixing123.cn/computer/57905.html

随着数据的存储呈现出集中化（以分布式存储系统为基础的云存储系统）和移动化（互联网移动终端）的趋势，数据可靠性愈发引起大家的重视。集群所承载的数据量大大上升，但存储介质本身的可靠性进步却很小，这要求我们必须以更加经济有效的方式来保障数据安全。

副本与纠删码都是通过增加冗余数据的方式来保证数据在发生部分丢失时，原始数据不发生丢失。但相较于副本，纠删码能以低得多的存储空间代价获得相似的可靠性。比如3副本下，存储开销为3，因为同样的数据被存储了三份，而在10+3（将原始数据分为10份，计算3份冗余）的纠删码策略下，存储开销为为1.3。采用纠删码能够极大地减少存储系统的存储开销，减少硬件、运维和管理成本，正是这样巨大的收益驱使各大公司纷纷将纠删码应用于自己的存储系统，比如Google、Facebook、Azure、EMC等等国际巨头，在国内以淘宝、华为、七牛云等为代表的公司也在自己的存储系统上应用了纠删码。

最典型的纠删码算法是里德-所罗门码（Reed-Solomon码，简称RS码）。RS码最早应用于通信领域，经过数十年的发展，其在存储系统中得到广泛应用，比如光盘中使用RS码进行容错，防止光盘上的划痕导致数据不可读；生活中经常使用的二维码就利用了RS码来提高识别的成功率。近年RS码在分布式存储系统中的应用被逐渐推广，一方面是分布式存储系统存储的存储容量和规模增大的需求；另一方面是由于纠删码编码速度在近年得到迅猛提升。随着对高性能纠删码引擎在实际系统中应用需要，也催生了对纠删码在具体系统中实现的各种优化手段。并为相关的决策者带来了困扰——究竟什么样的编码引擎才是高效的呢？

我们将以这个问题展开对纠删码技术的剖析，帮助企业更全面，深入的了解纠删码在存储系统中的应用并更好地做出技术选型。本系列文章将从纠删码的基本原理开始，随后引出如何判断编码引擎优劣这个问

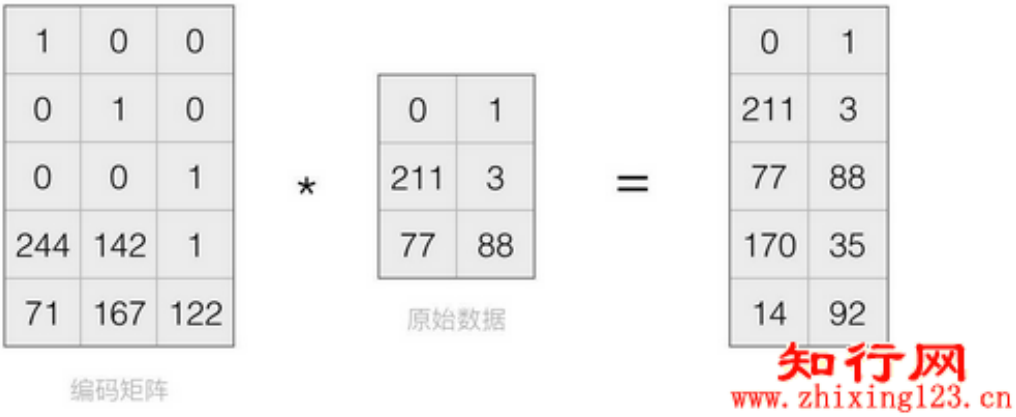
题，接下来将深度分析代码实现，帮助开发者顺利完成定制开发。

本文作为系列首篇，我们将一起探讨纠删码的编码原理与如何选择编码引擎这两个问题。

一、纠删码编码原理

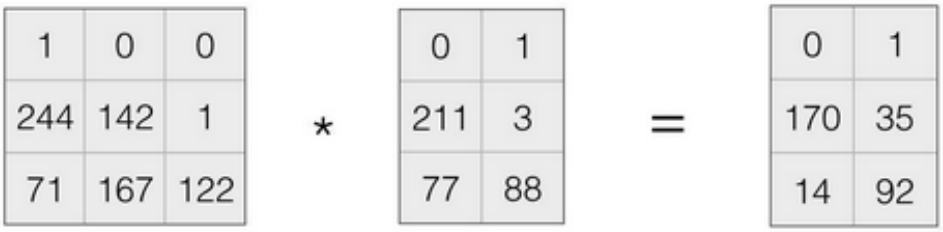
在展开分析之前，我们先来看一看RS码是如何工作的。

下图展示了3+2（3份数据，2份冗余）下对2字节长度的数据进行编码与数据修复过程：



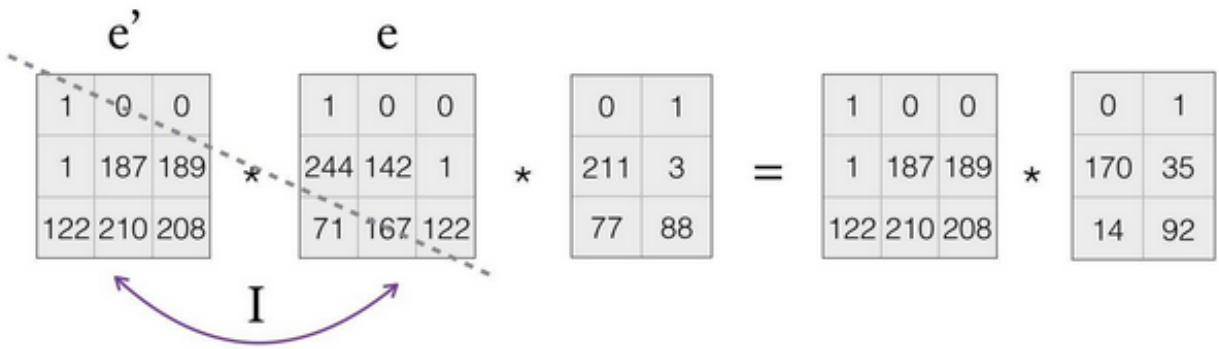
为了计算冗余数据，首先我们需要选举出一个合适的编码矩阵。编码矩阵的上部为一个单位矩阵，这样保证了在编码后原始数据依然可以直接读取。通过计算编码矩阵和原始数据的乘积，可以到最终的结果。

下面介绍解码过程，当1，2两块数据丢失，即：



当数据块发生丢失，在编码矩阵中去掉相应行，等式仍然保持成立。这为我们接下来恢复原始数据提供了依据。

原始数据的修复过程如下：



为了恢复数据，首先我们求剩余编码数据的逆矩阵，等式两边乘上这个逆矩阵仍然保持相等。与此同时，互逆矩阵的乘积为单位矩阵，因此可以被消掉。那么所求得的逆矩阵与剩余块的数据的乘积就是原始数据了。

数据编码以字节为单位，如果将被编码数据看做一个「数组」，「数组」中每个元素是一个字节，数据按照字节顺序被编码。编码过程是计算编码矩阵中元素和「数组」的乘积过程。为保证乘积的运算结果仍旧在一个字节大小以内（即0-255），必须应用到有限域[1]。有限域上的算术运算不同于通常实数的运算规则。我们通常事先准备好乘法表，并在算术运算时对每一次乘法进行查表得到计算结果。早期的编码引擎之所以性能不佳，是因为逐字节查表的性能是非常低的。倘若能一次性对多字节进行查表以及相应的吞吐和运算，引擎的工作效率必将大幅度提升。

许多CPU厂商提供了包含更多位数的寄存器（大于64位），这类寄存器和相应支持的运算使得用户程序可以同时对大于机器位数的数据进行运算，支持这类寄存器和运算的指令称之为SIMD（SingleInstructionMultipleData）指令集，比如Intel支持的SSE指令集最大支持128bits的数据运算，AVX2指令集最大支持512bits的数据运算。它们为我们对一个「数组」数据分别执行相同的操作，提高了数据运算的并行性。目前，市面上所有高性能的纠删码引擎均采用了该项技术以提高编解码性能。

二、编码引擎评判标准

我们将从以下几个关键指标来对编码引擎进行分析：

1、高编/解码速度；

2、参数可配置；

3、代码简洁、稳定；

4、降低修复开销等。

2.1 高编/解码速度

无须多言，编/解码性能是最基本也是最重要的指标。对于一款性能优异的引擎来说，应该同时满足以下几个指标：

根据CPU的特性自动选择最优的指令集进行加速。上文提到，依赖于SIMD技术RS码编码性能有了大幅度的提高。其中，我们可以利用多种指令集扩展以供加速，引擎应该能够自主发现最优解

不亚于目前最出色的几款引擎的性能表现（详见第三章著名引擎对比）

通过SIMD加速，性能会有大幅度攀升。我们还可以将逐字节查表（下称基本方法）的编码速度与利用SIMD技术加速的编码速度做对比，两者应该有数倍的差距

编/解码速度稳定，对于不同尺寸的数据块会有相近的性能表现。由于系统缓存的影响，当被编码数据的大小和缓存大小相当时，编码应该具有最快的速度。当编码数据的大小大于缓存大小时，内存带宽成为编码速度的瓶颈，文件大小和编码时间呈现近似线性关系。这样，数据编码时间是可预期的，用户的服务质量也是可保障的。在实际中，我们对于大文件进行定长分块，依次编码，分块大小和缓存大小保持一定关系。

下图展示了在10+4策略下，不同大小的数据块的编码速度变化趋势[2]：

注：

测试平台：MacBookPro(Retina,13-inch,Mid2014)，2.6GHzi5-4278U(3MBL3CacheSize),8GB1600MHzDDR3

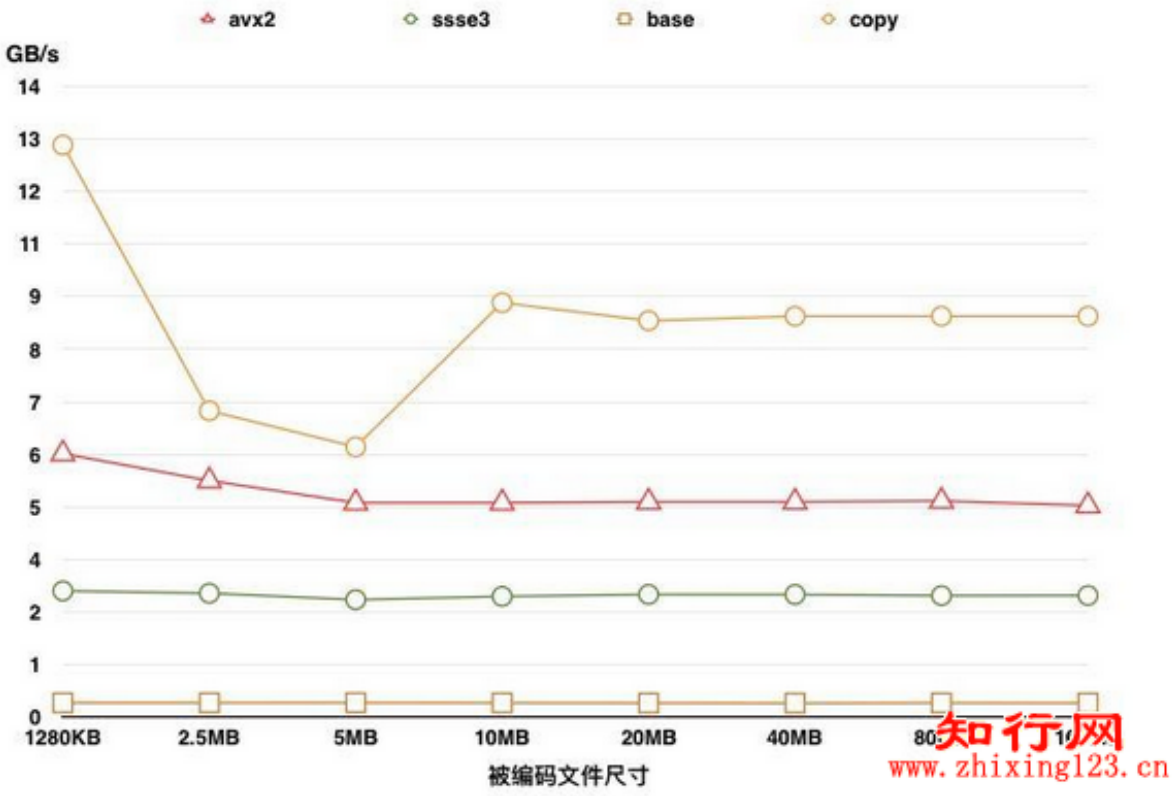
编/解码速度计算公式:在k+m策略下，每一个数据块的尺寸计作s,编/解码m个数据块的耗时计作t,则速度=(k*s)/t

测试方法：在内存中生成随机数据，运行若干次编／解码，取平均值

分别执行了avx2指令集,ssse3指令集,基本方法(base)这三种编码方案

被编码文件尺寸指，每一个数据块的尺寸与总的数据块个数的乘积，即原始数据的总大小

作为对比，利用go语言自带的copy函数（copy），对k个数据块进行内存拷贝。copy同样使用了SIMD技术进行加速



另外，解码速度应该大于或等于编码速度（视丢失的数据块数量而定），下图为10+4策略下修复不同数量的原始数据的速度对比[2]：

注：

测试平台与上文的编码测试相同

lostdata=丢失数据块数目（个）

原始数据块每块大小为128KB，总大小为1280KB

2.2 参数可配置

一款合理的纠删码引擎必须能做到编码策略在理论范围内可随意切换，这指的是如果要将编码策略进行变化时，仅需从接口传入不同参数而不需要改动引擎本身。这大大降低了后续的开发和维护所需要的精力。一个可配置参数的编码引擎可以根据数据的冷热程度和数据重要程度选择不同的编码系数，比如可靠性要求高的数据可以选择更多冗余。

2.3 代码简洁、稳定

为了利用SIMD加速我们不得不引入汇编代码或者封装后的CPU指令，因此代码形式并不常见。为了增强可读性可将部分逻辑抽离到高级语言，然而会损失部分性能，这其中的利弊需要根据团队的研发实力进行权衡。

接下来的可维护性也非常重要。首先是接口稳定，不会随着新技术的引入而导致代码大规模重构；另外代码必须经过有合理的测试模块以便在后续的更新中校验新算法。

比如早先的SIMD加速是基于SSE指令集扩展来做的，随后Intel又推出AVX指令集进一步提高了性能，引擎应该能即时跟上硬件进步的步伐。再比方说，再生码[5]（可以理解为能减少修复开销的纠删码）是将来发展的趋势，但我们不能因为算法的升级而随意改变引擎的接口。

2.4 降低修复开销

纠删码的一大劣势便是修复代价数倍于副本方案。k+m策略的RS码在修复任何一个数据块时，都需要k份的其他数据从磁盘上读取和在网络上传输。比如10+4的方案下，丢失一个数据块将必须读取10个块来修复，整个修复过程占用了大量磁盘I/O和网络流量，并使得系统暴露在一种降级的不稳定状态。因此，实际系统中应该尽量避免使用过大的k值。

再生码便是为了缓解数据修复开销而被提出的，它能够极大减少节点失效时所需要的吞吐的数据量。然而其复杂度大，一方面降低了编码速度，另外一方面牺牲了传统RS码的一些优秀性质，在工程实现上的难度也大于传统纠删码。

三、著名引擎对比

目前被应用最广泛并采用了SIMD加速的引擎有如下几款：

1.Intel出品的ISA-L[4]

2.J.S.Plank教授领导的Jerasure[5]

3.klauspost的个人项目（inGolang)[6]

这三款引擎的执行效率都非常高，在实现上略有出入，以下是具体分析：

3.1 ISA-L

纠删码作为ISA-L库所提供的功能之一，其性能应该是目前业界最佳。需要注意的是Intel采用的性能测试方法与学术界常用的方式略有出路，其将数据块与冗余块的尺寸之和除以耗时作为速度，而一般的方法是不包含冗余块的。另外，ISA-L未对vandermonde矩阵做特殊处理，而是直接拼接单位矩阵作为其编码矩阵，因此在某些参数下会出现编码矩阵线性相关的问题。好在ISA-L提供了cauchy矩阵作为第二方案。

ISA-L之所以速度快，一方面是由于Intel谙熟汇编优化之道，其次是因为它将整体矩阵运算搬迁到汇编中进行。但这导致了汇编代码的急剧膨胀，令人望而生畏。

另外ISA-L支持的指令集扩展丰富，下至SSSE3，上到AVX512，平台适应性最强。

3.2 Jerasure2.0

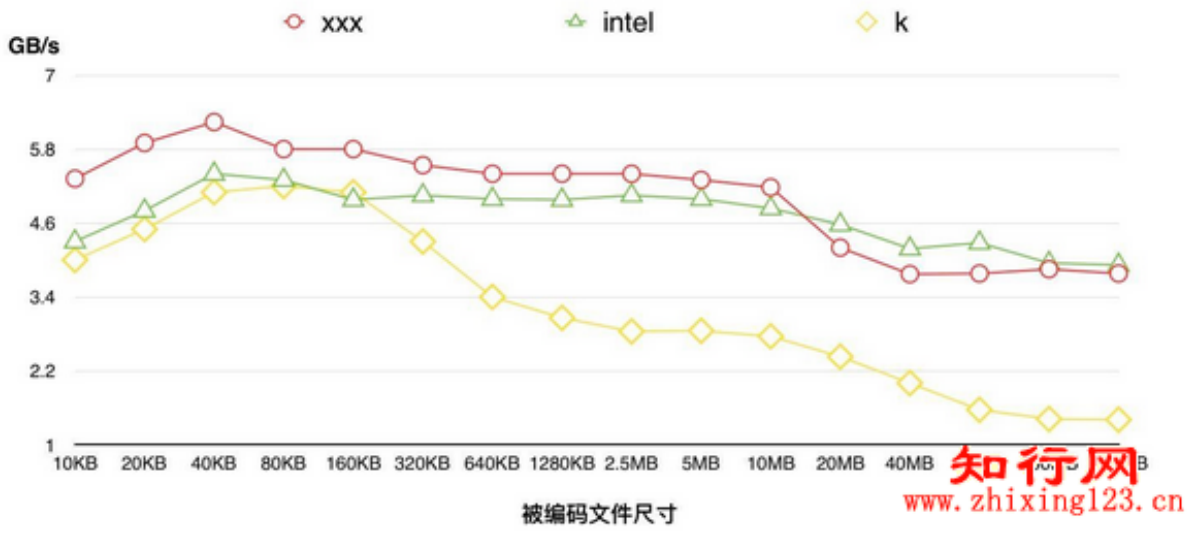
不同于ISA-L直接使用汇编代码，Jerasure2.0使用C语言封装后的指令，这样代码更加的友好。另外Jerasure2.0不仅仅支持GF(2^8)有限域的计算，其还可以进行GF(2^4)-GF(2^{128})之间的有限域。并且除了RS码，还提供了CauchyReed-Solomoncode（CRS码）等其他编码方法的支持。它在工业应用之外，其学术价值也非常高。目前其是使用最为广泛的编码库之一。目前Jerasure2.0并不支持AVX加速，尽管如此，在仅使用SSE的情况下，Jerasure2.0依然提供了非常高的性能表现。不过其主要作者之一JamesS.Plank教授转了研究方向，另外一位作者Greenan博士早已加入工业界。因此后续的维护将是个比较大的问题。

3.3 klauspost的ReedSolomon

klauspost利用Golang的汇编支持，友好地使用了SIMD技术，此款引擎的SIMD加速部分是目前我看到的实现中最为简洁的，矩阵运算的部分逻辑被移到了外层高级语言中，加上Golang自带的汇编支持，使得汇编代码阅读起来更佳的友好。不过Go并没有集成所有指令，部分指令不得不利用YASM等汇编编译器将指令编译成字节序列写入汇编文件中。一方面导致了指令的完全不可读，另外一方面这部分代码的语法风格是Intel而非Golang汇编的AT&T风格，平添了迷惑。这款引擎比较明显的缺陷有两点：1.对于较大的数据块，编码速度会有巨大的下滑；2.修复速度明显慢于编码速度。

3.4 编码速度对比

我在这里选取了IntelISA-L（图中intel),klauspost的ReedSolomon(图中k),以及自研的一款引擎2这三款引擎进行编码效率的对比,这三款引擎均支持avx2加速。测试结果如下：



注：

编码速度计算公式，测试方法与上一节相同。其中isa-l默认的速度计算方式与公式有冲突，需要修改为一致

测试平台：AWSt2.microIntel®Xeon®CPUE5-2676v3@2.40GHz,Memory1GB

编码方案：10+4

klauspost的引擎默认开了并发，测试中需要将并发数设置为1

四、自己实现一款引擎

可能是由于对开源库后续维护问题的担忧，也有可能是现有方案并不能满足企业对某些特定需求和偏好，很多公司选择了自研引擎。那么如何写出高效的代码呢？在上面的简单介绍中，受限于篇幅我跳过了很多细节。比如SIMD技术是如何为纠删码服务的，以及如何利用CPUCache做优化等诸多重要问题。我们会在后续的文章中逐步展开其实现，欢迎大家继续关注。

我们简单了解了Reed-SolomonCodes（RS码）的编／解码过程，以及编码引擎的评判标准。但并没有就具体实现进行展开，本篇作为《纠删码技术详解》的下篇，我们将主要探讨工程实现的问题。

这里先简单提炼一下实现高性能纠删码引擎的要点：首先，根据编码理论将矩阵以及有限域的运算工程化，接下来主要通过SIMD指令集以及缓存优化工作来进行加速运算。也就是说，我们可以将RS的工程实现划分成两个基本步骤：

将数学理论工程化

进一步的工程优化

这需要相关研发工程师对以下内容有所掌握：

有限域的基本概念，包括有限域的生成与运算

矩阵的性质以及乘法规则

计算机体系结构中关于CPU指令以及缓存的理论

接下来，我们将根据这两个步骤并结合相关基础知识展开实现过程的阐述。

一理论工程化

以RS码为例，纠删码实现于具体的存储系统可以分为几个部分：编码、解码和修复过程中的计算都是在有限域上进行的；编码过程即是计算生成矩阵（范德蒙德或柯西矩阵）和所有数据的乘积；解码则是计算解码矩阵（生成矩阵中某些行向量组成的方阵的逆矩阵）和重建数据的乘积。

1.1有限域运算

有限域是纠删码中运算的基础域，所有的编解码和重建运算都是基于某个有限域的。不止是纠删码，一般的编码方法都在有限域上进行，比如常见的AES加密中也有有限域运算。使用有限域的一个重要原因是计算机并不能精确执行无限域的运算，比如有理数域和虚数域。

此外，在有限域上运算另一个重要的好处是运算后的结果大小在一定范围内，这是因为有限域的封闭性决定的，这也为程序设计提供了便利。比如在RS中，我们通常使用$GF(2^8)$，即0～255这一有限域，这是因为其长度刚好为1字节，便于我们对数据进行存储和计算。

在确定了有限域的大小之后，通过有限域上的生成多项式可以找到该域上的生成元[1]，进而通过生成元的幂次遍历有限域上的元素，利用这一性质我们可以生成相应的指数表。通过指数表我们可以求出对数表，再利用指数表与对数表最终生成乘法表。关于本原多项式的生成以及相关运算表的计算可以参考我在开源库中的数学工具。[2]

有了乘法表，我们就可以在运算过程中直接查表获得结果，而不用进行复杂的多项式运算了。同时也不难发现，查表优化将会成为接下来工作的重点与难点。

1.2选择生成矩阵

生成矩阵（GM,generatormatrix）定义了如何将原始数据块编码为冗余数据块，RS码的生成矩阵是一个n行k列矩阵，将k块原始数据块编码为n块冗余数据块。如果对应的编码是系统码（比如RAID），编码后包含了原始数据，则生成矩阵中包含一个k×k大小的单位矩阵和(n−k)×k的冗余矩阵，单位矩阵对应的是原始数据块，冗余矩阵对应的是冗余数据块。非系统码没有单位矩阵，整个生成矩阵都是冗余矩阵，因此编码后只有冗余数据块。通常我们会使用系统码以提高数据提取时的效率，那么接下来我们需要找到合适的冗余矩阵。

在解码过程中我们要对矩阵求逆，因此所采用的矩阵必须满足子矩阵可逆的性质。目前业界应用最多的两种矩阵是Vandermondematrix(范德蒙矩阵）和Cauchymatrix（柯西矩阵）。其中范德蒙矩阵历史最为悠久，但需要注意的是我们并不能直接使用范德蒙矩阵作为生成矩阵，而需要通过高斯消元后才能使用，这是因为在编码参数（k+m）比较大时会存在矩阵不可逆的风险。

柯西矩阵运算简单，只不过需要计算乘法逆元，我们可以提前计算好乘法逆元表以供生成编码矩阵时使用。创建以柯西矩阵为生成矩阵的编码矩阵的伪代码如下图所示：

```
// m 为编码矩阵
// rows为行数，cols为列数
// k×k 的单位矩阵
for j := 0; j < cols; j++ {
    m[j][j] = byte(1)
}
// mxk 的柯西矩阵
for i := cols; i < rows; i++ {
    for j := 0; j < cols; j++ {
        d := i ^ j
        a := inverseTable[d]     // 查乘法逆元表
        m[i][j] = byte(a)
    }
}
```


1.3矩阵求逆运算

有限域上的求逆方法和我们学习的线性代数中求逆方法相同，常见的是高斯消元法，算法复杂度是$O(n^3)$。过程如下：

在待求逆的矩阵右边拼接一个单位矩阵

进行高斯消元运算

取得到的矩阵左边非单位矩阵的部分作为求逆的结果，如果不可逆则报错

我们在实际的测试环境中发现，矩阵求逆的开销还是比较大的(大约6000ns/op)。考虑到在实际系统中，单盘数据重建往往需要几个小时或者更长（磁盘I/O占据绝大部分时间），求逆计算时间可以忽略不计。

二进一步的工程优化

2.1利用SIMD加速有限域运算

从上一篇文章可知，有限域上的乘法是通过查表得到的，每个字节和生成矩阵中元素的乘法结果通过查表得到，图1给出了按字节对原始数据进行编码的过程（生成多项式为$x^8+x^4+x^3+x^2+1$）。对于任意1字节来说，在$GF(2^8)$内有256种可能的值，所以没有元素对应的乘法表大小为256字节。每次查表可以进行一个字节数据的乘法运算，效率很低。

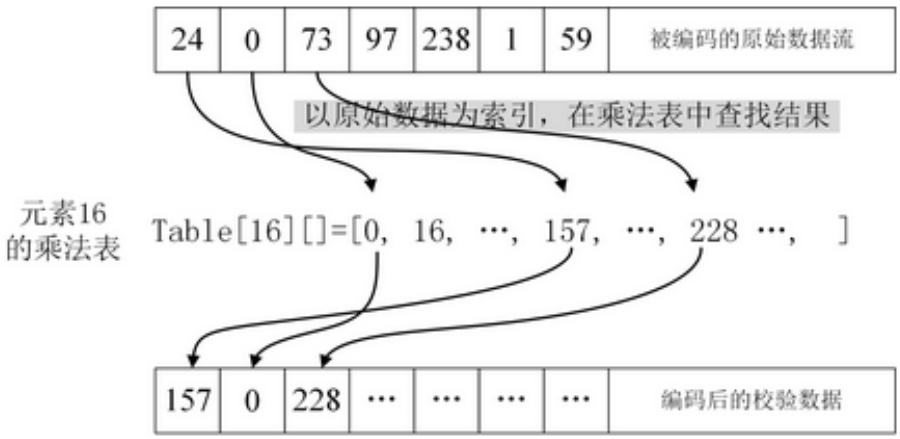


图 1, 按字节对原始数据进行编码

图 1, 按字节对原始数据进行编码

目前主流的支持SIMD相关指令的寄存器有128bit（XMM指令)、256bit(YMM指令)这两种容量，这意味着对于64位的机器来说，分别提供了2到4倍的处理能力，我们可以考虑采用SIMD指令并行地为更多数据进行乘法运算。

但每个元素的乘法表的大小为256Byte,这大大超出了寄存器容纳能力。为了达到利用并行查表的目的，我们采用分治的思想将两个字节的乘法运算进行拆分。

字节y与字节a的乘法运算过程可表示为，其中y(a)表示从y的乘法表中查询与x相乘结果的操作：

y(a)=y*a 我们将字节a拆分成高4位（al)与低4位(ar)两个部分，即（其中⊕为异或运算）：

a=(al<<4)⊕ar

这样字节a就表示为0-15与(0-15<<4)异或运算的结果了。

于是原先的y与a的乘法运算可表示为：

y(a)=y(al<<4)⊕y(ar)

由于ar与al的范围均为0-15（0-1111），字节y与它们相乘的结果也就只有16个可能的值了。这样原先256字节的字节y的乘法表就可以被2张16字节的乘法表替换了。

下面以根据本原多项式$x^8+x^4+x^3+x^2+1$生成的GF(2^8)为例，分别通过查询普通乘法表与使用拆分乘法表来演示16*100的计算过程。

16的完整乘法表为：

```
table = [0 16 32 48 64 80 96 112 128 144 160 176 192 208 224 240 29
13 61 45 93 77 125 109 157 141 189 173 221 205 253 237 58 42 26 10
122 106 90 74 186 170 154 138 250 234 218 202 39 55 7 23 103 119 71
87 167 183 135 151 231 247 199 215 116 100 84 68 52 36 20 4 244 228
212 196 180 164 148 132 105 121 73 89 41 57 9 25 233 249 201 217 16
9 185 137 153 78 94 110 126 14 30 46 62 206 222 238 254 142 158 174
190 83 67 115 99 19 3 51 35 211 195 243 227 147 131 179 163 232 248
200 216 168 184 136 152 104 120 72 88 40 56 8 24 245 229 213 197 18
1 165 149 133 117 101 85 69 53 37 21 5 210 194 242 226 146 130 178
162 82 66 114 98 18 2 50 34 207 223 239 255 143 159 175 191 79 95 1
11 127 15 31 47 63 156 140 188 172 220 204 252 236 28 12 60 44 92 7
6 124 108 129 145 161 177 193 209 225 241 1 17 33 49 65 81 97 113 1
66 182 134 150 230 246 198 214 38 54 6 22 102 118 70 86 187 171 155
139 251 235 219 203 59 43 27 11 123 107 91 75]
```


计算16*100可以直接查表得到：

table[100] = 14

16 的低 4 位乘法表，也就是 16 与 0-15 的乘法结果：

lowtable=[0163248648096112128144160176192208224240]

16的高4位乘法表，为16与0-15<<4的乘法结果：

hightable=[0295839116105788323224521020715612916618 7]

将100（01100100）拆分，则：

100 = 0110 << 4 ⊕ 0100

在低位表中查询0100（4），得：

lowtable[4]=64 在高位表中查询 0110（6），得：

hightable[6]=78

将两个查询结果异或：

result=64^{78}=1000000^{1001110}=1110=14

从上面的对比中，我们不难发现采用SIMD的新算法提高查表速度主要表现在两个方面：

减少了乘法表大小； 提高查表并行度（从1个字节到16甚至32个字节） 采用SIMD指令在大大降低了乘法表的规模的同时多了一次查表操作以及异或运算。由于新的乘法表每一部分只有16字节，我们可以顺利的将其放置于XMM寄存器中，从而利用SIMD指令集提供的指令来进行数据向量运算，将原先的逐字节查表

改进为并行的对16字节进行查表，同时异或操作也是16字节并行的。除此之外，由于乘法表的总体规模的下降，在编码过程中的缓存污染也被大大减轻了，关于缓存的问题我们会在接下来的小节中进行更细致的分析。 以上的计算过程以单个字节作为例子，下面我们一同来分析利用SIMD技术对多个字节进行运算的过程。基本步骤如下： 拆分保存原始数据的XMM寄存器中的数据向量，分别存储于不同的XMM寄存器中根据拆分后的数据向量对乘法表进行重排，即得到查表结果。我们可以将乘法表理解为按顺序排放的数组，数组长度为16，查表的过程可以理解为将拆分后的数据（数据范围为0-15）作为索引对乘法表数组进行重新排序。这样我们就可以通过排序指令完成查表操作了 将重排后的结果进行异或，得到最终的运算结果 以下是伪代码：

```
// 将原始数据的右移4bit
d2 = raw_data >> 4
// 将右移后的数据的每字节与15（即1111）做AND操作,得到数据高位
high_data = d2 AND 1111
// 原始数据的每字节与15（即1111）做AND操作,得到数据低位
low_data = raw_data AND 1111
// 以数据作为索引对乘法表进行了重排
for i, b = range low_data {
  low_ret[i]=low_table[b]
}
for i, b = range high_data {
  high_ret[i]=high_table[b]
}
// 异或两部分结果得到最终数据
ret = low_ret XOR high_ret
```


需要注意的是，要使用SIMD加速有限域运算，对CPU的最低要求是支持SSSE3扩展指令集。另外为了充分提高效率，我们应该事先对数据进行内存对齐操作，在SSSE3下我们需要将数据对齐到16Bytes，否则我们只能使用非对齐指令进行数据的读取和写入。在这一点上比较特殊的是Go语言，一方面Go支持直接调用汇编函数这为使用SIMD指令集提供了语言上的支持；但另外一方面Golang又隐藏了内存申请的细节，这使得指定内存对齐操作不可控，虽然我们也可以通过cgo或者汇编来实现，但这增加额外的负担。所幸，对于CPU来说一个Cacheline的大小为64byte，这在一定程度上可以帮助我们减少非对齐读写带来的惩罚。另外，根据Golang的内存对齐算法，对于较大的数据块，Golang是会自动对齐到32byte的，因此对齐或非对齐指令的执行效果是一致的。

2.2 写缓存友好代码

缓存优化通过两方面进行，其一是减少缓存污染；其二是提高缓存命中率。在尝试做到这两点之前，我们先来分析缓存的基本工作原理。

CPU缓存的默认工作模式是Write-Back,即每一次读写内存数据都需要先写入缓存。上文提到的Cacheline即为缓存工作的基本单位，其大小为固定的64byte，也就说哪怕从内存中读取1字节的数据，CPU也会将其余的63字节带入缓存。这样设计的原因主要是为了提高缓存的时间局域性，因为所要执行的数据大小通常远远超过这个数字，提前将数据读取至缓存有利于接下来的数据在缓存中被命中。

2.2.1 矩阵运算分块

矩阵运算的循环迭代中都用到了行与列，因此原始数据矩阵与编码矩阵的访问总有一方是非连续的，通过简单的循环交换并不能改善运算的空间局域性。因此我们通过分块的方法来提高时间局域性来减少缓存缺失。

分块算法不是对一个数组的整行或整列进行操作，而是对其子矩阵进行操作，目的是在缓存中的数据被替换之前，最大限度的利用它。

分块的尺寸不宜过大，太大的分块无法被装进缓存；另外也不能过小，太小的分块导致外部逻辑的调用次数大大上升，产生了不必要的函数调用开销，而且也不能充分利用缓存空间。

2.2.2 减少缓存污染

不难发现的是，编码矩阵中的系数并不会完全覆盖整个GF(2^8)，例如10+4的编码方案中，编码矩阵中校验矩阵大小为4×10，编码系数至多（可能会有重复）有10×4=40个。因此我们可以事先进行一个乘法表初始化的过程，比如生成一个新的二维数组来存储编码系数的乘法表。缩小表的范围可以在读取表的过程中对缓存的污染。

另外在定义方法集时需要注意的是避免结构体中的元素浪费。避免将不必要的参数扔进结构体中，如果每一个方法仅使用其中若干个元素，则其他元素白白侵占了缓存空间。

三指令级并行与数据级并行的深入优化

本节主要介绍如何利用AVX/AVX2指令集以及指令级并行优化来进一步提高性能表现。除此之外，我们还可以对汇编代码进行微调以取得微小的提升。比如，尽量避免使用R8-R15这8个寄存器，因为指令解码会比其他通用寄存器多一个字节。但很多汇编优化细节是和CPU架构设计相关的，书本上甚至Intel提供的手册也并不能提供最准确的指导（因为有滞后性），而且这些操作带来的效益并不显著，在这里就不做重点说明了。

3.1利用AVX2

在上文中我们已经知道如何将乘法表拆分成128bits的大小以适应XMM寄存器，那么对于AVX指令集来说，要充分发挥其作用，需要将乘法表复制到256bit的YMM寄存器。为了做到这一点，我们可以利用XMM寄存器为YMM寄存器的低位这一特性，仅使用一条指令来完成表的复制（Intel风格）：

vinserti128ymm0,ymm0,xmm0,1

这条指令作用是将xmm0寄存器中的数据拷贝到ymm0中,而剩余128位数据通过ymm0得到，其中立即数1表明xmm0拷贝的目的地是ymm0的高位。这条指令提供了两个sourceoperand（源操作数）以及一个destinationoperand（目标操作数），我们在这里使用ymm0寄存器同时作为源操作数和目标操作数来实现了表的复制操作。接下来我们便可以使用与SSSE3下同样的方式来进行单指令32byte的编码运算过程了。

由于使用了SSE与AVX这两种扩展指令集，我们需要避免AVX-SSETransitionPenalties[3]。之所以会有这种性能惩罚主要是由于SSE指令对YMM寄存器的高位一无所知，SSE指令与AVX指令的混用会导致机器不断的执行YMM寄存器的高位保存与恢复，这大大影响了性能表现。如果对指令不熟悉，难以避免指令混用，那么可以在RET前使用VZEROUPPER指令来清空YMM寄存器的高位。

3.2指令级并行(ILP)优化

程序分支指令的开销并不仅仅为指令执行所需要的周期，因为它们可能影响前端流水线和内部缓存的内容。我们可以通过如下技巧来减少分支指令对性能的影响，并且提高分支预测单元的准确性：

尽量少的使用分支指令

当贯穿(fall-through)更可能被执行时，使用向前条件跳转

当贯穿代码不太可能被执行时，使用向后条件跳转

向前跳转经常用在检查函数参数的代码块中，如果我们避免了传入长度为0的数据切片，这样可以在汇编中去掉相关的分支判断。在我的代码中仅有一条向后条件跳转指令，用在循环代码块的底部。需要注意的是，以上2、3点中的优化方法是为了符合静态分支预测算法的要求，然而在市场上基于硬件动态预测方法等处理器占主导地位，因此这两点优化可能并不会起到提高分支预测准确度的作用，更多的是良好的编程习惯的问题。

对于CPU的执行引擎来说，其往往包含多个执行单元实例，这是执行引擎并发执行多个微操做的基本原理。另外CPU内核的调度器下会挂有多个端口，这意味着每个周期调度器可以给执行引擎分发多个微操作。因此我们可以利用循环展开来提高指令级并行的可能性。

循环展开就是将循环体复制多次，同时调整循环的终止代码。由于它减少了分支判断的次数，因此可以将来自不同迭代的指令放在一起调度。

当然，如果循环展开知识简单地进行指令复制，最后使用的都是同一组寄存器，可能会妨碍对循环的有效调度。因此我们应当合理分配寄存器的使用。另外，如果循环规模较大，会导致指令缓存的缺失率上升。Intel的优化手册中指出，循环体不应当超过500条指令。[4]

四小结

以上内容较为完整的还原了纠删码引擎的实现过程，涉及到了较多的数学和硬件层面的知识，对于大部分工程师来说可能相对陌生，我们希望通过本系列文章的介绍能够为大家的工程实践提供些许帮助。但受限于篇幅，很多内容无法全面展开。比如，部分数学工具的理论与证明并没有得到详细的解释，还需要读者通过其他专业资料的来进行更深入的学习。