isa-l 中 ec_init_tables() 的用途

关注到这个问题的同学，想必对 ec_encode_data()的原理已经了解的差不多了。如果感觉还有欠缺，可以参考下面两篇论文和一篇博客。

论文：
A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems
Note: Correction to the 1997 Tutorial on Reed-Solomon Coding
这两篇论文都是美国田纳西大学 James S. Plank教授等人写的，James S. Plank教授是纠删码领域著名的学术专家，有很多理论和实践成果，详情可参见链接：James S. Plank。这两篇论文中，第二篇是对第一篇的修正，但不是说第一篇有多大的问题，而是存在某些细节问题。因此，从入门的角度来讲，这两篇论文都应该读一下。

博客：
高性能纠删码编码
这篇博客的内容比较丰富，并且提到了一个重要的点：在 ec_encode_data()中，每个元素的乘法表的大小为 256 Byte，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，采用分治的思想将两个字节的乘法运算进行拆分。暂时先记住这个结论，后面会用到。

从 isa-l中给出的使用示例来看，对数据字节进行编码时基本都遵循以下流程：

```
...
gf_gen_rs_matrix();
ec_init_tables();
ec_encode_data_base();
...
```

gf_gen_rs_matrix()很好理解，就是生成 Reel-Solomon-Van矩阵，这是 RS编码的基础，不用多解释。
如果只看 C语言版实现，ec_init_tables()的作用就不太好理解了，因为 RS编码理论中貌似没有提到这个步骤，为什么除了生成 RS矩阵外，还需要根据 RS矩阵生成一个 table？

接下来看了 ec_encode_data_base()的实现，还会有另外一个疑问：为什么数据字节是和 table中的元素相乘，而不是直接和矩阵中的元素相乘？

```
void ec_encode_data_base(int len, int srcs, int dests, unsigned char *v,
                         unsigned char **src, unsigned char **dest)
{
        int i, j, l;
        unsigned char s;
```

```
        for (l = 0; l < dests; l++) {
                for (i = 0; i < len; i++) {
                        s = 0;
                        for (j = 0; j < srcs; j++)
                                s ^= gf_mul(src[j][i], v[j * 32 + l * srcs
* 32 + 1]);

                        dest[l][i] = s;
                }
        }
}
```

仔细观察上面的代码可以发现，如果将 table数组划分成每 32个字节一组的话，数据字节刚好是和每组中下标为 1的字节在 GF(2^8)上相乘，为什么要这样做？32是哪来的？下标为 1的字节代表什么？下面依次回答这三个问题。

如果我们对 ec_init_tables()的实现原理和作用不了解的话，就会有上述疑问。在搞懂 ec_init_tables()的实现原理后我们知道，table中每 32个字节对应一个查找表，每个查找表由 RS矩阵中的一个元素生成，查找表的作用就是当数据字节和 RS矩阵中的某个元素相乘时，直接从该元素对应的查找表中得到结果。我猜测这就是为什么函数名中的 tables是复数形式，因为 ec_init_tables()的作用是为 RS矩阵（其实只针对下半部分的范德蒙德矩阵）中的每个元素生成查找表，也就是说生成了多个 tables，并且按顺序将这些 tables放入一个一维字符数组中，最后通过上面的 j * 32 + l * srcs * 32进行索引，其中，l * srcs * 32索引到 RS矩阵中的行，j * 32索引到 RS矩阵中的列。

为什么查找表的大小是 32字节？因为这里做了一个优化——在查找前，将数据字节分成了高 4位和低 4位，让他们分别去查找表中进行索引，然后将得到的结果进行异或，这个异或的结果就是 RS矩阵中对应元素与数据字节在 GF(2^8)上相乘的结果。需要补充的一点是，这里的查找表分为高 4位查找表和低 4位查找表，这两个查找表的内容是不一样的，因此需要由两个 16字节构成一个元素的查找表，所以查找表的大小是 32字节，感兴趣的同学可以看看 gf_vect_mul_init()的实现原理，该函数实现了针对某个元素生成对应查找表的功能，有点复杂，但慢慢看，也能看懂。

同样的，在搞懂查找表的生成原理后我们知道，查找表中下标为 1的字节正是生成该查找表的那个元素本身！所以从查找表中取下标为 1的字节与数据字节相乘的效果就等同于让 RS矩阵中的某个元素与数据字节直接相乘！

这么一看，ec_init_tables()生成的 table貌似没有用上啊！每次都只使用了查找表中的一个元素，而且这个元素还是原封不动的来自于 RS矩阵。这个 table到底有什么用呢？回答这个问题，需要看 ec_encode_data()的汇编实现。在汇编实现中，采用 SIMD指令，将原先的逐字节查表改为并行的对 16字节同时查表，并将 16字节中所有高 4位的查表结果和所有低 4位的查表结果进行 16字节异或，这就得到了 RS矩阵中某个元素与 16字节数据相乘的结果，这比逐字节查表的效率高多了！

如果将 ec_encode_data_base()稍加修改，用 C语言实现单字节查表，那么就能看出 table的用途了，代码如下所示：

```
void ec_encode_data_base(int len, int srcs, int dests, unsigned char *v,
                         unsigned char **src, unsigned char **dest)
{
        int i, j, l;
        unsigned char s, sval, *pos;

        for (l = 0; l < dests; l++) {
                for (i = 0; i < len; i++) {
                        s = 0;
                        for (j = 0; j < srcs; j++) {
                                sval = src[j][i];
                                pos = &v[j * 32 + l * srcs * 32];
                                s ^ = pos[sval & 0x0F] ^ pos[16 + (sval >>
4)];
                        }

                        dest[l][i] = s;
                }
        }
}
```


经测试，上述修改还能提升性能。可能的原因是，修改前是在 256字节的表中进行查找，修改后是在 32字节的表中进行查找，后者对 cache更友好。