密码学第六次实验报告

# 优化的AES算法

## 原理

### 对轮操作的优化

上一次的实验中，已经可以预想到AES的查表优化。因此这次的优化，可以借助AES的运算性质，通过查表优化的方法来进行。这样可以使AES加密使用到的运算次数大大减小。

具体的查表优化的方法是这样的。对任意一列，有：

其中为字节经过ByteSub部件后的结果，列下标需要模4。

根据每一个完整轮的加密方式，易知上式成立。

因此可以定义4个1024字节的表：

这些表中的取值范围为一个字节的所有可能取值。易知各表的大小确实是1024字i，而且能被事先计算。可以把这些表看成一种函数，这种函数输入一个字节值，产生一个32位的列向量，该向量是对应输入字节的函数。

这样轮函数对一列的操作就能简化成查表和异或：

显然这种方案能大大减小加密操作需要的时间。但是需要的空间略有增加，因此这种方案是一种以空间换时间的方案。

事实上，由于解密算法可以在结构上与加密算法相同，因此类似地可以给出一种对解密轮函数的查表优化方案。同样也是把轮函数对一列的操作简化成查表和异或：

其中为字节经过InvByteSub部件后的结果，列下标也需要模4,是对应的解密变换要查的表。但由于结构上的限制，其中应为经过InvMixColumn部件处理后的轮密钥的第列。

事实上，由于对轮密钥进行InvMixColumn操作相当于把轮密钥当作列向量，再左乘一个矩阵，因此也可以用类似的算法进行优化。

这样，就可以实现加解密的查表优化。

### 利用Python语言特性进行优化

Python是一种解释型语言，因此面对编译型语言有先天的性能劣势。但是，Python中提供了许多相对高效的语言特性，比如map操作、列表生成式、迭代器等。通过尽量广泛地使用这些语言特性，可以让代码效率进一步提高。

### 函数内联

由于Python是一种解释型语言，而且是动态类型语言，所以创建函数的开销比较大。内联函数在Python中能够起到一定的优化作用。可以把常调用的简单函数进行内联，从而提高效率。

### 共享密钥状态

由于对文件的加解密实现的是CFB模式，所以每轮使用的密钥相同，可以通过共享密钥状态来实现。实际上，共享木要状态只需要共享密钥列表和密钥数。这样可以进一步提高效率。

## 伪代码

用到的与AES相关的函数与上次的相应部件实现相同，因此伪代码也相同。但是加密算法和解密算法不同，因此只展示加密算法和解密算法的伪代码，其余函数的伪代码在第五次实验的实验报告中。

### 加密算法

**def** encrypt(p: bytes, k: bytes) -> bytes:

由p生成state

由k生成key\_state

key\_list = get\_key\_list(key\_state)

addroundkey(state, key\_list[0])

for i in range(9):

for j in range(4):

得到

把结果与第i轮的轮密钥的第j列相异或，得到下一轮的状态的第j列

用生成的下一轮的状态代替现在的状态

bytesub(state)

shiftrow(state)

addroundkey(state, key\_list[10])

**return** state转换成字节后的结果

### 解密算法

**def** decrypt(p: bytes, k: bytes) -> bytes:

由p生成state

由k生成key\_state

key\_list = get\_key\_list(key\_state)

addroundkey(state, key\_list[10])

for i in range(8, 0, -1):

for j in range(4):

得到

把第i轮的轮密钥用InvMixColumn部件处理

把结果与处理后的轮密钥的第j列相异或，得到下一轮的状态的第j列

用生成的下一轮的状态代替现在的状态

invbytesub()

invshiftrow()

addroundkey(state, key\_list[0])

**return** state转换成字节后的结果

## 分析

这种方案实际上用到了一定的空间，因为要存储8个表，每个表的大小为字节，即1KB。但是耗费8KB的存储空间，实际上的开销并不大，因为现代计算机很容易满足8KB的存储需求。这样，每轮只需要4次查表和5次异或运算，虽然时空复杂度仍然为常数级别，但是实际运行效率会大大提高。

## 测试

采用明文54776F204F6E65204E696E652054776F，密钥5468617473206D79204B756E67204675进行测试。得到密文为29C3505F571420F6402299B31A02D73A。这里明文、密文、密钥均采用16进制表示。采用相同的密钥解密密文，得到原来的明文。



可以看出，结果与未经优化的AES算法相同。经过验算，结果正确。

下面比较未经优化与优化后的AES算法的效率，数据仍然选用上文中的数据，重复运行20次得到运行时间和占用内存的平均值。得到结果如下：

|  |  |  |
| --- | --- | --- |
|  | 平均运行时间/s | 占用内存/MB |
| 未经优化的AES算法 | 0.008185 | 0.009019 |
| 优化后的AES算法 | 0.001164 | 0.006789 |

可以看出，经过优化，AES算法的运行效率有了成倍的提升。而且由于减少了中间结果的存放，占用内存也降低了。这就使得AES算法更加实用。

使用优化后的AES算法加密一个文件，大小为848.8KB。结果如下：

|  |  |  |
| --- | --- | --- |
|  | 加密平均运行时间/s | 解密平均运行时间/s |
| 未经优化的AES算法 | 144.043991 | 306.888531 |
| 优化后的AES算法 | 15.275411 | 24.879644 |

# 总结

## 优化的AES算法

其实这个算法属于对加密算法的优化，实际上技术难度并不是非常大。重要的一点是弄明白背后的数学原理，这对该算法的实现才是关键的。实际上，数学原理大体就是矩阵运算，因为AES的非线性性并不多。弄明白数学原理以后，实现就相对简单了。

## 失败的思路

### 减少与合并文件I/O

文件I/O是相对慢的部分，因此可以通过在代码中减小对文件的分段读写做到提升代码效率。

尝试减少和合并文件I/O，让对文件的读写集中，结果失败了。应该是因为Python中对文件的操作自带缓冲机制，多重缓冲反倒会适得其反，增加文件操作的延迟。

### 使用单例模式优化对密文和密钥状态的管理

对密文和密钥的状态，我采取的是每加密一次就创建一个新密文状态和密钥状态对象的解决方案。考虑到这种方案耗费的内存较大而且可能会增加垃圾收集器的负担，尝试采用单例模式进行优化。结果发现时间反倒增加了。

猜想可能是更小的零碎对象增加了垃圾收集器的工作量，而且引用计数难以管理。因此可以看出：如果有很多生灭较快的对象待回收，在Python中用其它对象把它们打包似乎是个好方法。

## 算法评估与优化

这次的算法评估与优化，还是沿用了以前的套路。但是，优化的内核越来越偏重数学了。这也是数学对算法优化的一个很典型的例子。

还有一个优化点，就是要明白底层实现。我用的Python，底层实现就相对完善，所以对某些更加接近硬件的语言的优化技巧不一定适用。这也是一个需要注意的地方。

## 系统设计与维护

这次良好的系统设计与维护起到了很好的作用。由于程序进行了模块化处理，所以我可以直接更改相关函数的代码，而且基本上不需要有其他考虑，因为代码结构相对清晰，而且几乎没有荣誉代码。由于上次实验打下的好基础，我可以方便地使用已经有的工具进行调试和逻辑验证，从而加快了实现的速度。同时，由于我以前写过了评测用的代码框架，因此这次评测使用的代码写起来速度能稍微快一些。这应该能体现良好系统设计的加速型回报。

## 对课程的建议

感觉以后可以多强调代码的模块化，这样更能使我们注重代码的可维护性，对以后的实践也是有帮助的。

## 总结

这次实验算一次代码优化实验。这次确实让我能够重视代码优化的重要性，我会在以后的实验中更加关注算法的性能，并且在实现中体现出来。