**个人文件加密系统**

**摘要：**本文实现了一个个人文件加密系统，提供多级密钥管理，动态修改可执行文件以获得每个用户使用不同的主密钥;通过各种关键信息计算出用户密钥而非直接记录用户密钥的加密文本，拥有较难破解的用户密钥；使用自定义个文件密钥加密文件；而且提供不需要解密而直接通过网络进行访问的方式，当然，本身不包含任何访问公共网络的逻辑，以保证用户放心。

**关键字：**个人加密系统 openssl 网络访问 修改动态链接库

**Abstract：**I have implemented a personal file encryption system, which provides multi-level key management. It will dynamically modify the executable file to make different have different master keys. It will calculate the user key through various information instead of directly recording the encrypted text on the disk, which makes the user hardly to crack. Using a custom file key to encrypt the file makes it more convenient. It also provides functions to access encrypted files through the network without decryption. Of course, it does not contain any access to the public network logic, because it may make the user worry about the reliability;

**Keywords：**personal file encryption system, openssl, access encrypted files through the network without decryption, modify executable file

**第一章 背景与意义**

**1.原因**

个人文件加密系，在办公区等工种场所、或者比较重要的个人文件（如密码本）等方面都有需求。移动端一般厂商自带加密功能，且被攻破的机会比较少（在没有root的情况下，厂商自带的加密系统文件是无法访问到的）；macOS/linux有比较完善的用户权限系统，并不太需要。

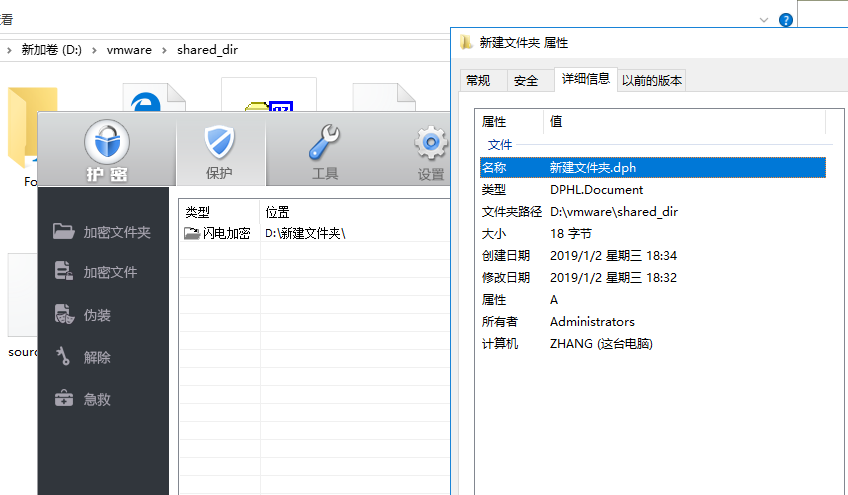
但是windows端则完全相反。

1. windows的文件权限系统设计并不友好，普通用户几乎没什么人用，windows自带的加密时针对磁盘的，而没有类似lvm 的技术支持，这会非常笨拙，危险；
2. 在完全单机的情况下，如何保护主密钥是一个很困难的问题，应为它就存储在本机，总能被找到，破解；
3. java/c#/python的加密/混淆机制并没有太好的作用，几乎是可以看到源码的，c++要好一些，但是也有反汇编然后修改关键函数返回的破解存在；
4. 加密过的文件在使用时得经过解密，写回磁盘才可以使用，但对大文件而言这一过程往往比较耗时，尤其是只读取文件时，写到磁盘显得毫无作用。

由于以上原因，市面上目前没有太好的相关软件，要么收费且不方便，要么必须联网，难以信任，难以推广。因此设计一款比较难破解，非常方便使用的的个人文件加密系统就很有必要。

**2.国内的研究现状和发展趋势**

国内的护密文件夹加密软件大师：



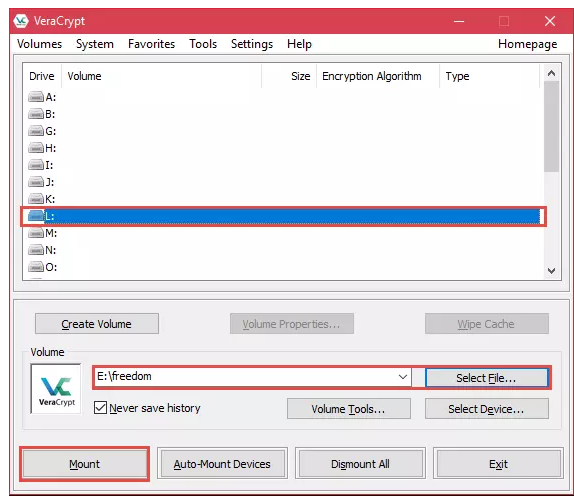
优点：小巧（4M），不依赖网络，在原位置创建.dph文件以链接到程序；缺点：单级密码，安全度不高,文件建相互依赖时不好处理。

国内的红线隐私保护系统



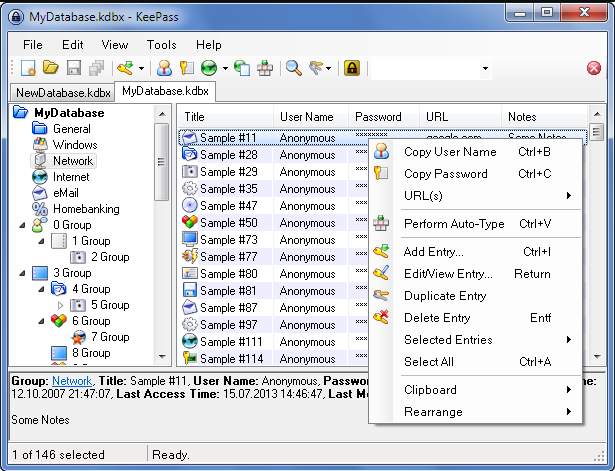
优点：可以直接打开被加密的文档资料进行编辑修改操作。无需手动解密再编辑，无需编辑保存后再次加密。使用方便，加密复杂，安全度高；缺点：是一款企业级软件的个人移植版，需要购买，需要联网，是一款半在线的加密软件，你的数据会被同步到它的服务器上，主密钥也存在服务器上。

国外的VeraCrypt：



优点：可以将加密目录挂载到一个虚拟盘符下，方便使用，采用二级加密机制；缺点：主密钥容易白爆破，虽然这个工具也提供通过密钥文件进行解密的方式，但这种方式一是便利性不够，二是密钥文件又要想办法加密和隐藏，且又不能放在VeraCrypt加密区。

国外的KeePass：



通过AES算法来加密文件，在加密时会要求输入AES加密的密钥，在输入密钥后，会自动生成原文件名+扩展名+.aes的加密文件。优点：可以使用很多加密秘钥，加密秘钥自行管理；缺点：用户管理秘钥复杂，容易弄混秘钥，使用时必须先解密，并不方便。

**第二章 加密算法对比与选择**

**1.对称加密与非对称加密**

对称加密，又称共享密钥加密，即通信双方用同一个密 钥分别对信息进行加密、解密运算。当双方需要进行通信时， 发送方通过加密算法用密钥对明文数据包加密，使其转变为 非授权用户无法读取的密文，然后通过媒介发送给接收方。 数据接收方在收到密文后，用与发送方相同的密钥对密文进 行解密运算，之后才能对数据进行读取利用[[1]](#endnote-0)。对称加密使用同一份密钥进行加密/解密，在网络应用中安全性不好，但是，在个人文件加密系统这种纯粹的本地应用中，他是比较适合的。

非对称密码算法是指通信的收发双方使用具有两组密 钥的密码，如 ＲSA 算法。一个作为公钥对外公开，另一个作为 私钥不对外公开，用来进行解密。尽管非对称密码算法需要使 用两个密码增加了密码的复杂性，但是它能够有效的提高信息 安全程度，获得广泛的好评。非对称密码算法是指通信的收发双方使用具有两组密 钥的密码，如 ＲSA 算法。一个作为公钥对外公开，另一个作为 私钥不对外公开，用来进行解密。尽管非对称密码算法需要使 用两个密码增加了密码的复杂性，但是它能够有效的提高信息安全程度，获得广泛的好评。[[2]](#endnote-1)在本应用中，由于文件按的加密解密，都是由文件主控完成的，不需要传输，所以非对称加密算法在本程序中就不需要了。

**2.各种对称加密算法的比较**

DES 加密算法采用 56 位密钥对 64 位明文数据进行有效 的处理，由于其算法是公开的，因此，DES 加密算法的保密 性主要是由密钥的安全性决定的，其密钥是一组对称的分组 密码。运用 DES 加密算法对数据进行加密前，要对所有的明 文进行科学合理的分组，然后再对每个分组依次进行二进制加密，完成后就会得到每组对应的密文数据，最后将各组对 应的密文数据按照一定的顺序连接起来，就能得到 DES 加密 算法的整个密文。算法只是对数据进行加密的一种基本技术， 而通过将这些技术进行有机结合就能得到 DES 加密算法。DES 加密算法是基于密钥作用于明文，也就是我们所了解的轮， 在 DES 之中一共设有 16 轮，这就表明采用该加密算法时，需 要在明文的各个分组进行 16 次相同的组合技术，这就能够大 大提高数据的安全性。 [[3]](#endnote-2)这是本程序中主要使用的加密算法方法之一，是一种比较基础，实用的算法，性能各方面基本满足要求，而且出错率少，可以使用。

3DES （Triple DES）是 DES 向 AES 过渡的加密算法， 针对 DES 密码长度过短，安全性略低而进行改进，利用 DES 形成了三重数据加密。3DES 采用 3 组 64 位密钥，若第一组 和第三组密钥相同，则被称为双密钥的 3DES，即加密 -- 解 密 -- 加密 (EDE) 模式，密钥实际为 56 × 2 = 112 bits。若三 组密钥均不相同，那么密钥长度实际为 56 × 3 = 168 bits， 安全性将会大大增强。[[4]](#endnote-3)本程序不会使用这种算法，因为他只是des的加强，但是用户密钥很难胡hi有这么长的密钥，更多的只是浪费；而且3des的软件实现版本性能都很一般，这里文件加密中性能是很重要的，因此更不会选他。

国际数据加密算法IDEA (International Data Encryption Algorithm) 密钥长度128位, 密钥空间是2128, 属于对称加密算法的一种, 该算法具有保密性强、加密速度快的特点。1990年, 曾被瑞士联邦技术学院的来学嘉X.J.Lai和Massey建议称为PES, 1992年Lai和Massey提高了该算法的抗差分分析能力, 并改成位IDEA。IDEA逐渐代替了DES, 并被誉为“好的分组密码”, 应用在许多行业和产品中。[[5]](#endnote-4)在本程序中也可以使用，但更多的只是一种替补。

AES 算法是为了代替安全性不能满足要求的 DES 算法而选 出的，主要由拓展密钥、加密模块和解密模块组成。AES 的分组 长度为 128 比特，且有三种可选的密钥长度：128 比特、192 比 特及 256 比特，分别对应的加密轮数为 10 轮、12 轮及 14 轮。[[6]](#endnote-5)AES算法，性能好，安全度高，将会作为本程序的主要加密算法，当然也会提供DES算法和IDEA算法，但他们更多的只是一种补充，为了提高破解难度而较少的使用一部分这些算法。

总的来说，对称加密算法DES比较慢，只能在安全要求不太高的场合中使用；3DES 更加慢，这是文件个人文件系统中无法忍受的；IDEA速度快，对密钥要求不但是容易攻破，只能作为一个替补，偶尔使用一下；AES具有速度快，有抗密码分析强度好，具有相对更好的安全性，将会被作为主力使用。

分组密码工作模式 ： ECB：电子密码本（最常用的，每次加密均产生独立的密 文分组，并且对其他的密文分组不会产生影响，也就是相同的 明文加密后产生相同的密文）。 CBC：密文链接（常用的，明文加密前需要先和前面的密 文进行异或运算，也就是相同的明文加密后产生不同的密文）。 除了这两种常用的工作模式，还有：CFB：密文反馈。OFB：输出反馈。CTR：计数器。 这五种工作模式主要是密码学中算法在进行推导演算的时候所应用到的。[[7]](#endnote-6)本程序主要使用ebc和cbc模式，随机使用多种加密算法，多种分组模式，有利于加强文件系统的安全性。

**3.算法库**

对称加密算法实现是非常复杂的，而且稳定性要求是非常高的，所以，我不在自己实现对称加密算法，而是使用openssl库所提供的各种加密算法，对他们进行一次适配器模式的包装，使用统一的接口进行调用。

Openssl项目是一个开放源代码的工具包,它实现了安全套接层协议(S LvZ/v3) 和传输层安全协议 ( TLSvl ) , 并带有 一个功效完整的 、具有通用性的加密技术库 。Openssl 工具包可分为三个 部分 : S SL 函数库 、 Crypto函数库和命令行工具。SS L函数库实现了安全套接层协议(S LvZ/v3)(S LvZ/v3)和传输层安全协议;Crypto函数库实现了大多数interne标准的加密算法;[[8]](#endnote-7)这里我们使用openssl的Crypto函数库，它实现了很多常见的加密解密算法，并将它们封装为BIO库（各种文件相关的操作和一些简单的算法）EVP库（各种复杂算法的实现，兼容bio库），抽象出了一套通用的逻辑。

source/sink类型的BIO是数据源，例如，sokect BIO和文件BIO。 而filter BIO就是把数据从一个BIO转换到另外一个BIO或应用接口，例如在加密BIO中，如果写操作，数据就会被加密，如果是读操作，数据就会被解密。Openssl EVP(high-level cryptographic functions)提供了丰富的密码学中的各种函数。Openssl 中实现了各种对称算法、摘要算法以及签名/验签算法。EVP 函数将这些具体的算法进行了封装，并且是支持bio库的，EVP系列函数主要封装了加密、摘要、编码三大类型的算法，他们都可以被当作一个bio的filter使用，这样就极大的简化了编程流程。

**第三章 工作环境搭建**

**1.总体说明**

本系统尽可能使用posix标准进行编写，Linux下使用gcc进行编译，windows下使用cygwin+gcc进行编译，对于必须使用平台相关组件的，使用宏命令进行区分编译。

整体架构采用mvc架构，model与controller结合较为紧密，在一起相互依赖实现，view层完全隔离，通过命令模式驱动，网络访问中使用socket（tcp）进行通信，本地gui通过pipe进行通信，即使没有gui，也可以通过命令行进行驱动。

**2.依赖及编译/获取方式：**

STL 库：c++标准库；获取方式：c++自带。

Posix标准库：为跨平台准备的一些底层操作库，如pthread获取方式：linux和cygwin自带。

boost 库：c++准标准库，为跨平台提供了很多高级工具，如boost：：filesyatem，boost：：log等，为跨平台准备了充分的环境。获取方式：cygwin : 包管理器中有 ；linux :apt install libboost-all-dev。

gtest 库：google的测试用例的驱动库；获取方式：通用：github获取源码，cmake -DCMAKE\_INSTALL\_PREFIX=/usr .. & make & make install。

openssl crypt库：提供各种加密解密算法；获取方式：；cygwin : 先安装nasm(包管理器) perl(包管理器)，然后github 获取源码，执行 ./Configure Cygwin-x86\_64 no-asm --prefix=/usr ；linux:直接apt获取。

yaml-cpp 0.6.0 ：yaml语言解析器；获取方式：通用：github获取源码， 编译安装 (注意如果是静态库,直接链接.a文件位置,eg; /usr/local/lib/libyaml-cpp.a，如果是动态库，则和其他库一样)

qt5（qt跨平台图像库，性能和美观都不错）；获取方式：；Linux：直接安装官网安装包；Window：直接安装官网安装包，和cygwin的交互在架构处会提到。

Openssl中使用的算法：编码：base64；散列：md5，sha256；

对称 DES-cbc DES-ebc AES-cbc

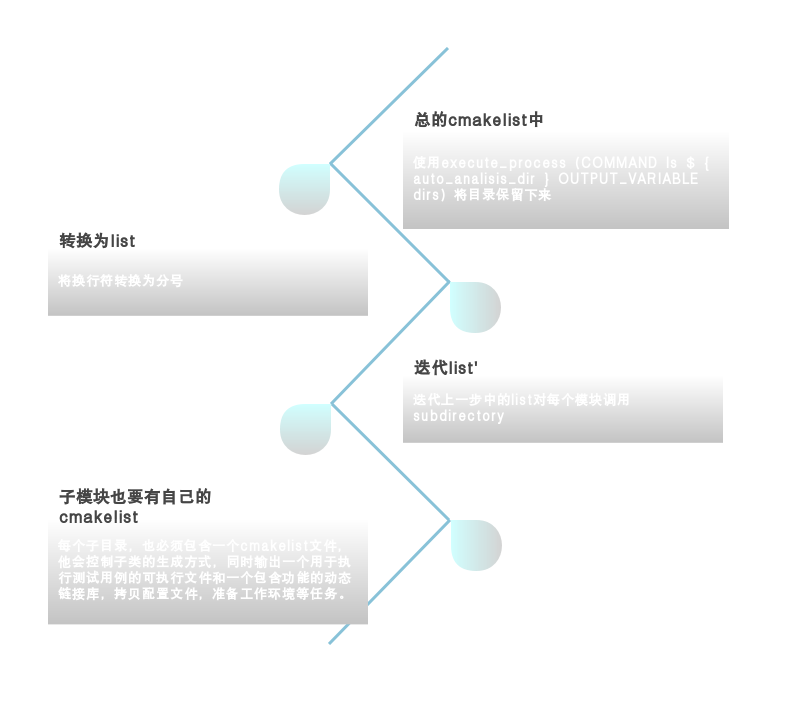
非对称 RSA(暂时没用到)

**3.自动加载子模块的cmake脚本**

构建一个自动化的cmake脚本，方便多模块使用：

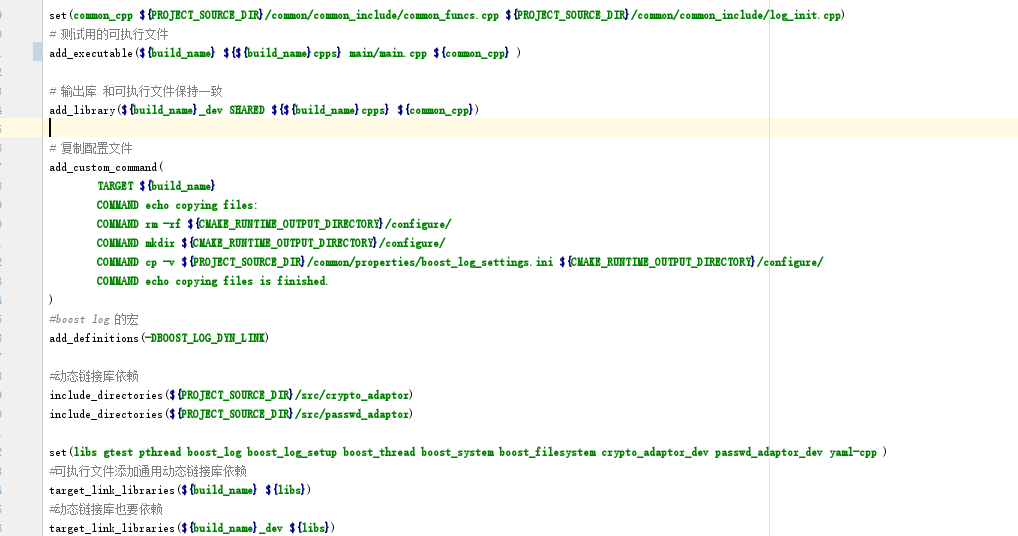
由于cmake没有可以直接使用的将某个子目录全部解析为子模块的功能，但是我梦必定会有很多的子模块，这样，对导致每次新加子模块都要修改cmakelist，这是很麻烦的一件事，本着没有轮子就造的精神，我自己写了一个自动加载子目录为子模块的脚本：

在项目根目处创建一个总cmake文件，用于设置通用的编译选项和加载子项目。cmake脚本中有一个execute\_process命令，它可以用来在执行cmake脚本之前调用一些可执行程序，并将结果保存在一个变量中，有了这个功能，我们就可以实现自动加载子模块了。



使用execute\_process ( COMMAND ls $ { auto\_analisis\_dir } OUTPUT\_VARIABLE dirs ) 命令，将auto\_analisis\_dir下的文件列出来，注意，ls命令在脚本中是通过换行符来区分文件的，但是cmake识别的列表是通过封号来划分的，所以使用string(REPLACE "\n" ";" RPLACE\_LIST ${dirs})将换行符替换为";"，这样，我们就得到了一个列表，使用foreach命令对每一个子文件夹调用subdirectory，这样，我们就可以不用修改cmakelist文件就实现自动加载子类了。

当然，对于每个子目录，也必须包含一个cmakelist文件，他会控制子类的生成方式，同时输出一个用于执行测试用例的可执行文件和一个包含功能的动态链接库，拷贝配置文件，准备工作环境等任务。



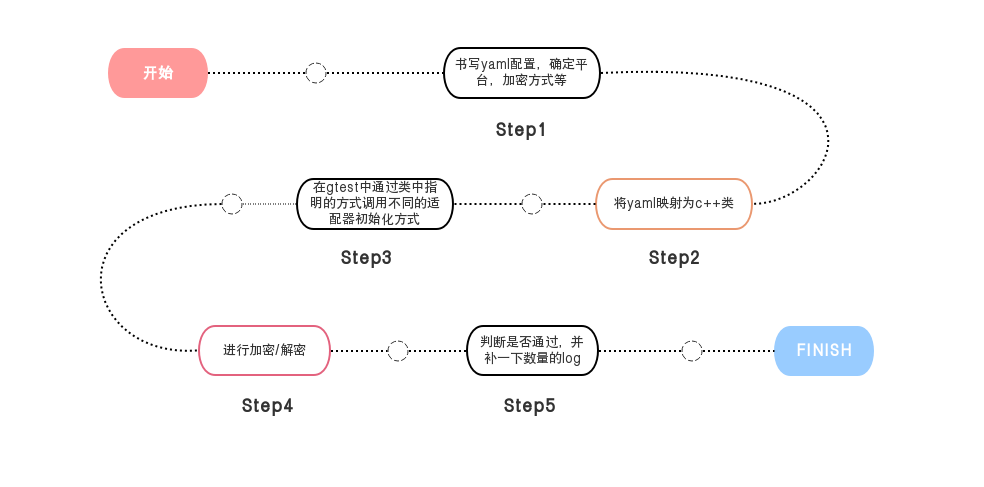
这样当新添加模快时，只要新建一个文件夹并复制子文件夹的cmakelist进行少量修改就可以同时实现输出库和测试可执行文件，十分方便。

版本控制

使用git作为版本控制工具，搭配gitee（类似github，只不过服务器在国内，速度快一些）进行开发。

**第四章 测试驱动**

单元测试是一个软件开发十分重要的过程，但是当功能较为多时，测试用例的就会变得十分繁琐；为了检测边界值，又需要复制很多重复的代码出来，十分丑陋，而且一旦功能有变，又需要修改大量的代码。这使得很多程序不进行测试开发，而大公司中有专门的测试人员负责测试（但个人项目根本不可能）。为了解决这个问题，我做出了如下设计：所有功能预先设计接口，，类似的功能使用统一的接口，根据接口开发出一个测试框架通过框架加载yaml配置，使用配置调用具体的测试用例，通过如下流程测试：



这样只需要给每个接口开发一个测试用例，至于具体要测哪些值，要测多少条数据，完全由yaml配置文件决定，程序简洁清晰，添加新的测试用例又十分简单。加密解密适配器的部分文件如下：

├── hash\_adaptor.cpp

├── hash\_adaptor.h

├── test.hpp

├── test\_main

│   ├── init\_tests\_yaml.cpp

│   ├── init\_tests\_yaml.h

│   ├── main.cpp

│   └── tests.cpp

└── work\_mode.h

这里实现了各种加密算法的适配器，每个适配器由需要好多的测试用例（因为每种加密适配器同时支持文件，字符串，二进制）但是通过adaptor\_base.h定义好接口后，就可以只写三个测试用例，然后通过yaml加载测试数据进来，进行测试。测试使用Google的gtest进行。一个例子如下：

测试用的model



这是测试时使用的数据，它指明了测试用例的组，测试名，可执行的平台，测试值，测试值的解释方式（文件按，base64数据，字符串），加密方式，和预期值，然后将这个配置映射到一个c++数组中。



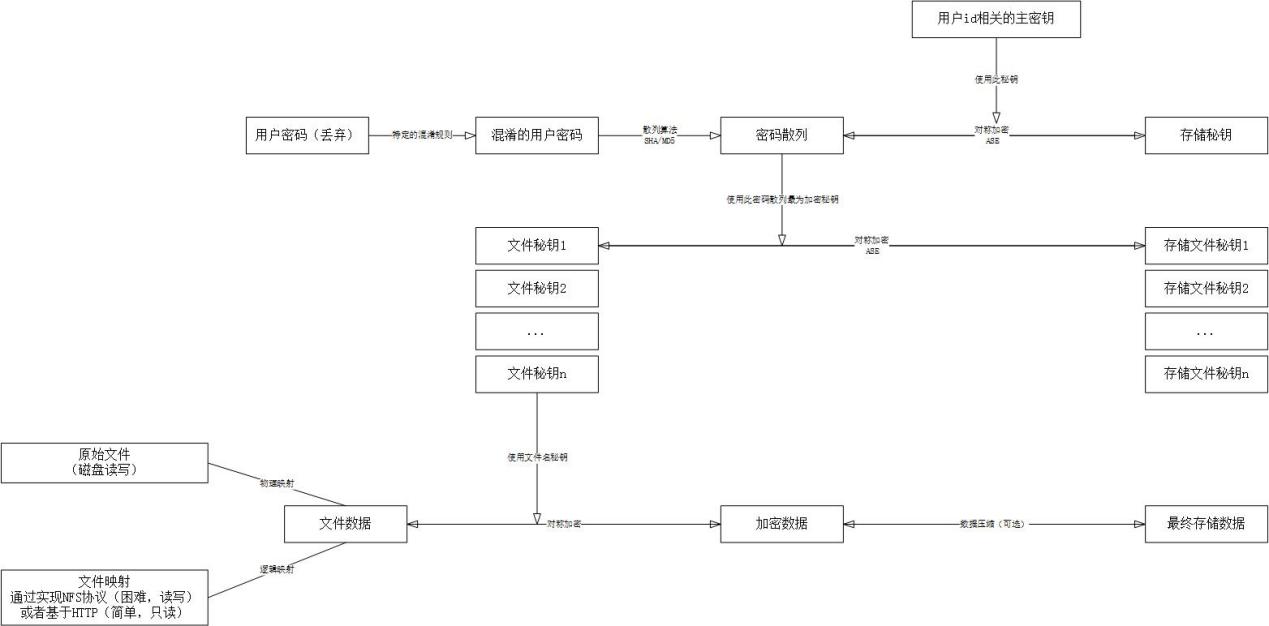
这样我们九八yaml配置，转换为c++对象，然后通过一个测试驱动，不断地循环这个数组，就可以实现对配置进行测试：

这样，进过简单的开发，就可以实现测试用例100%覆盖。当然这样也引入了一个问题：Google test 无法正确的识别到测试用例的个数，因为一个模块只用一个test，需要我们使用log进行追补。

**第五章 系统总体设计**

文件加密/解密的管理程序，采用多级密钥管理体制，主密钥用来保护用户密钥，用户密钥用于保护个人文件的加密/解密密钥；对称密钥算法作为文件加密/解密算法，采用随机密钥加密文件，并进行相关文件/密钥资源管理。

**1.总体架构**



1.1用户密码按固定的规则进行混淆，丢弃原始密码

1.2混淆文本进行散列计算

1.3散列文本使用主密钥进行AES对称加密，存储

2.1使用散列文本作为文件秘钥加密秘钥，进行加密存储

2.2构建一个文件秘钥池，使用多个文件秘钥

3.1原始文件物理映射为文件数据（为了逻辑统一）

3.2通过HTTP构建新的访问方式逻辑映射为文件数据

3.3随机（或指定）一个文件秘钥对文件数据进行加密（解密）

3.4加密后的数据进行压缩（可选）

3.5最终数据存盘

**2.总体模块说明**

主要采用适配器模式，mvc模式，组件模式（unity3d类似的解耦设计思想），单例模式，观察者模式，命令模式等设计思想，通过各种控制器与组件模式进行整合。

**2.1 主目录：**

├── CMakeLists.txt ：总的cmakelist

├── common ：通用的工具，配置文件,日志适配器等

├── LICENSE ：许可协议（BSD）和其他依赖的协议

├── README.en.md ：reamde的英文版

├── README.md ：readme的中文版

├── gui ：通过命令模式驱动的qt gui的实现

└── src ：各个功能模快的实现

**2.2 common:包含通用的工具，配置文件,日志适配器等**

├── common/common\_include ：日志适配器，常用方法，版本号等

├── common/properties ：配置文件

├── common/temlpate ：cmakelist的模板

**2.3 src：各个功能模快动态链接库和测试可执行文件的实现**

├── src/change\_rodata ：修改已编译动态链接库中的主密钥

├── src/crypto\_adaptor ：加密算法适配器

├── src/file\_coltroller ：文件管理器的逻辑实现

├── src/main\_controller ：主控，用于协调各个模快协作

├── src/main\_passwd ：主密钥的管理器

├── src/passwd\_adaptor ：用户秘钥与文件秘钥的管理器

└── src/web\_adaptor ：简单的网络实时解密浏览实现

2.4 gui:

└─personal\_file\_gui\_qt ：qt + qml 写的一个gui

**3.生成的可执行文件说明**

├── configure ：拷贝的配置文件

│   └── boost\_log\_settings.ini ：boostlog的配置文件

├── cygchange\_rodata\_dev.dll ：初始化时修改主密钥的链接库

├── cygcrypto\_adaptor\_dev.dll ：加密适配器（使用openssl）

├── cygfile\_controller\_dev.dll ：文件管理器，管理文件表

├── cygmain\_passwd\_dev.dll ：主密钥的存储的dll库

├── cygpasswd\_adaptor\_dev.dll ：用户密钥，文件密钥的适配器

├── cygweb\_adaptor\_dev.dll ：简单的网络浏览适配器

├── filedata ：加密文件数据

│   ├── data.7AWUCj

│   ├── data.B4WSQL

├── log ：输出的日志

├── main\_controller.exe ：主控，通过pipe进行获取命令

├── main\_controller\_cli.exe ：通过命令行驱动的pipe客户端

├── personal\_file\_gui\_qt.exe ：qt写的gui驱动的pipe客户端

└── sysdata ：系统数据

├── 2.filePasswd.data ：用户2的文件密钥

├── 3.filePasswd.data ：用户3的文件密钥

├── 3.index.data ：用户3的文件目录

├── init.data ：是否初始化的记录

├── pipe.data ：主控启动时的随机pipe位置记录

└── userKey.data ：获取用户密钥的关键数据

当然这只是本系统生成的动态链接库和可执行文件，其他以来的库就不再这里列举了。

**第六章 各个模快的详细说明与实现方式**

**1. common/common\_include**

common/common\_include/

├── common\_configure.h.in ：cmake的版本控制头文件

├── common\_funcs.cpp ：通用方法

├── common\_funcs.h ：通用方法的定义

├── common\_includes.h

├── log\_init.cpp ：log的适配器实现

├── log\_init.h ：log的适配器定义

└── README.md

common\_funcs.h：通用方法，包含几个内存安全的内存拷贝和读写超时检测

size\_t memncpy(void \*\_\_restrict dest, size\_t dest\_len, const void \*\_\_restrict src, size\_t src\_len);由于memcpy不会检测是否越界，这样会导致很可能写坏堆栈，造成很对安全问题，这里对memcpy进行了一次包装，这样就不会写坏堆栈了，类似strncpy的实现，额外传入了dest\_len，最多只拷贝这个长度的数据，并返回实际拷贝的长度，方便检测是否越界写了。

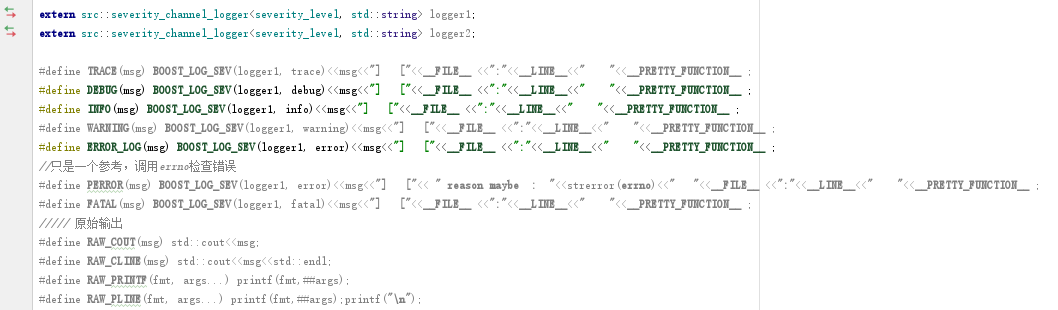
#define MEMNCPY(dest, dest\_len, src, src\_len) memcpy(dest, src, (src\_len < dest\_len) ? src\_len : dest\_len)通过宏定义实现的memncpy，由于函数调用还是比较耗费性能的，而inline也并不是一定会起作用的，但是这个函数一定会被频繁调用，所以通过宏定义实现，减少函数调用开销。

int test\_read\_timeout(int fd, long wait\_sec);检测读超时的函数（并不进行读操作）

int test\_write\_timeout(int fd, long wait\_sec);检测写超时的函数（并不进行读操作）

log\_init.h：log的适配器定义

通过适配器模式对boost.log的封装，方便在不修改代码的情况下更换log库。如刚开始时是使用log4cplus，但是由于它在cygwin环境下无法通过编译，而跟换为了boost.log，却不用修改任何代码。主要 包含以下方法的实现：



对常用的所有可能的输出都进行了包装，通过宏定义实现，也不会耗费性能，并且剥离了业务逻辑也日志框架，减少了对业务的侵入性，当由于各种原因要更换日志模块时，只需要在这里进行修改就好，完全不用改动业务代码。

**2.src/crypto\_adaptor：加密算法适配器**

├── adaptor\_base.h ：加密库适配器基类

├── DES\_func\_adaptor.cpp ：des适配器

├── DES\_func\_adaptor.h ：des适配器

├── AES\_func\_adaptor.cpp ：AES适配器

├── AES\_func\_adaptor.h ：AES适配器

├── base64\_adaptor.cpp ：base64适配器

├── base64\_adaptor.h ：base64适配器

├── CMakeLists.txt

├── hash\_adaptor.cpp ：md5，sha256适配器

├── hash\_adaptor.h ：md5，sha256适配器

├── test\_main

│   ├── init\_tests\_yaml.cpp ：测试配置加载

│   ├── init\_tests\_yaml.h ：测试配置加载

│   ├── main.cpp ：测试用例驱动的主函数

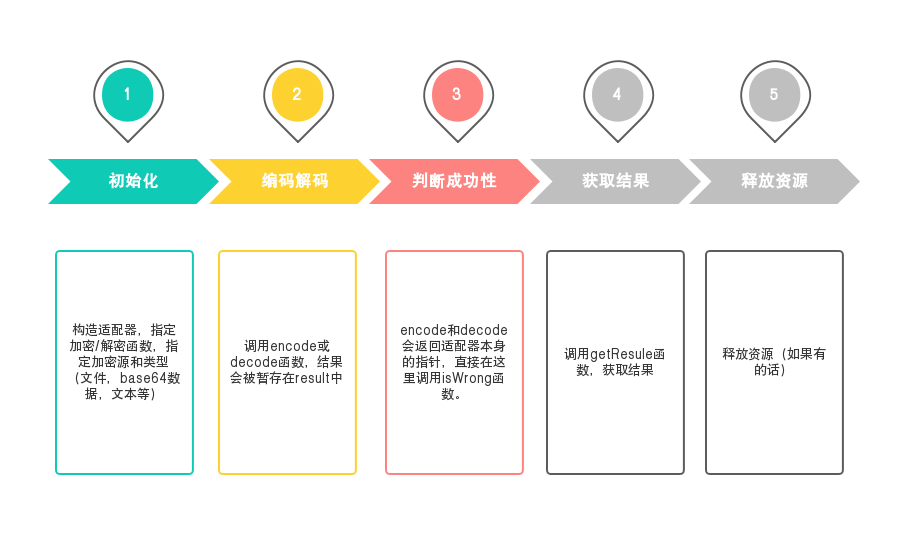
│   └── tests.cpp ：测试用例的定义

└── work\_mode.h ：工作模式如平台，加密方法等

对openssl crypt库进行适配器模式封装。各种加密适配器都继承自adaptor\_base：



这里只提供一些通用的接口，和为观察者模式准备的一些接口，至于具体的加密解密方法的定义，则有各个实现自己定义，应为各个库的接口都不相同。但是都遵循以下流程：



初始化：构造适配器，指定加密/解密函数，指定加密源和类型（文件，base64数据，文本等）



编码解码：调用encode或decode函数，结果会被暂存在result中

判断成功性：encode和decode会返回适配器本身的指针，直接在这里调用isWrong函数。



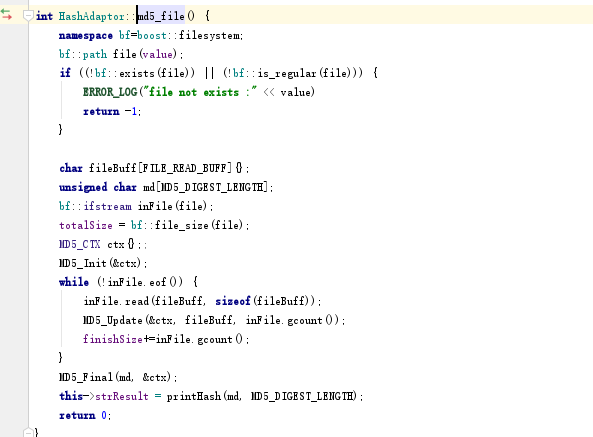
获取结果：调用getResule函数，获取结果



释放资源：释放资源（如果有的话）

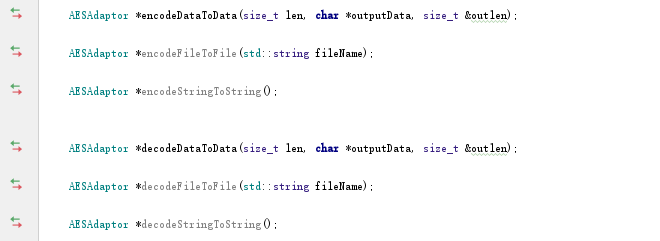
这样，就把各种复杂的加密解密逻辑该亏为三句代码，使用非常方便。

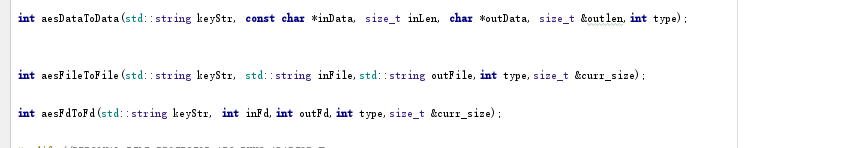
以hash\_adaptor为例：Hash\_adaptor只需要一个 encode ( CryptType eType );传入算法的enum类型即可完成hash函数。，至于到底时文件，数据，还是文本，则是在初始化时指明的。而各种函数的实现，则是通过对openssl的封装，如对MD5的封装，可以方便的实现MD5。



通过初始化MD5控制器->不断读入一个文件->然后调用update，同时更新观察者需要的已处理的文件大小->结束并刷新->输出结果；这个流程进行求解。

而对于des，aes库，则会变得复杂的多，因为不仅涉及到加密解密，还有对文件，二进制，甚至网络中的文件描述符进行操作。





对称加密算法的设计的就比较复杂，因为它要面对最复杂的业务。所以他提供了加密解密文件，数据，字符串，甚至文件描述符的功能。

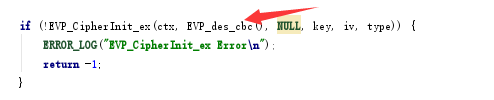
以对文件描述符的操作为例：



参数说明：KeyStr：密钥，通过文件密钥控制器获取；inFd：待加密/解密的文件描述符；outFd：输出的文件按描述符，支持文件，socket等，可以实现不用解密而通过网络浏览；type：加密（0）或者解密（1）；curr\_size：当前已经处理的大小，用于展示进度。

通过openssl提供的bio库，将网络的输出定义为一个bio文件，然后讲之与一个des\_cbc的过滤器相互串联，同时，指定秘钥，初始化方式，还有是加密还是解密，这样写入到过滤器中的数据就会被输出到文件描述符中，通过不断的读入文件->输出到过滤器->更新观察者需要的数据这一流程，就实现了对网络的加密解密输出，方便用户在无需解密的情况下，进行在线的数据的读取，甚至通过wifi进行共享。

至于加密解密方法，openssl的EVP库对这些复杂的加密算法进行了非常好的封装，同时还支持BIO。通过初始化一个文件bio->将输出文件绑定到bio上->初始化一个过滤器->将加密解密算法绑定到过滤器上->将输出文件绑定到过滤器之后->向过滤器写入文件->结束并刷新这一流程，就可以实现对任意加密算法的快捷使用，要修改加密算法，只需要修改下图中这一个地方就好，非常方便。



**3.src/file\_coltroller：文件管理器的逻辑实现**

├── CMakeLists.txt ：子模块cmakelist

├── file\_controller\_adaptor.cpp ：文件控制器主控适配器

├── file\_controller\_adaptor.h ：文件控制器主控适配器

├── file\_controller.cpp ：文件与文件表实现

├── file\_controller.h ：文件与文件表实现

└── main

├── init\_tests\_yaml.cpp ：测试用例加载器

├── init\_tests\_yaml.h ：测试用例加载器

├── main.cpp ：测试主类

└── tests.cpp ：测试驱动

设计思路与原因：

首先：文件树的方式有非常复杂的层次关系，对于大量的文件进行分层管理效率高，但是复杂的层次关系导致他非常脆弱，一点文件错误可能导致很多文件索引不到，要对这种情况进行处理需要非常复杂的逻辑，即使是现代的文件系统，很多也处理不好，而单层的文件结构有非常好的抗错误能力，但是文件多的时候性能不行，考虑到个人文件系统不会有太多的文件，但是稳定性非常重要，所以最后选择了单层文件模式，因此采取了单层文件结构+全路径名的方式实现而非文件树结构的方式。

对于文件表的存储实现方式，我有三种思路：通过map+yaml/json存储，通过map+new(dest)直接dump内存实现，通过sqlite实现。

map+yaml/json存储：需要写一些打包函数，存储，加载会慢一些；但是稳定性好；

map+new(dest)直接dump内存实现：通过c++<new>库，在指定的buff中new出map，然后对这个map进行操作，存储和加载则直接对这一段buff进行操作，实现简单，但是用户可能无法进行迁移，由32位系统换成64位系统，或者进行系统间迁移都会存在问题。而且，直接使用内存映射，会导致一旦某个位置出现文件错误，那么可能会造成整个列表出现偏移，那么所有数据都会丢失。而通过加密的字符串的形式存储，即使出现了文件错误，也只是导致他本身解析出错，最多影响邻居，不会有大的灾难。

sqlite：如果把前面两种实现方式看作redis，那么sqlite可以看做传统sql，由于单层化的模式，它的性能可能存在问题；每次查找都要去磁盘加载数据库，速度堪忧，尤其时当有其他线程在进行加密/解密操作时，会大量占用磁盘速度，这时，数据库进行更新会变得非常慢；而且sqlite开源版的加密是对数据进行加密，但是表结构是公开的，当用户使用较为简单的加密方式，如des-cbc时，这是非常危险的，而加密表结构的版本时要商业收费的，这会带来很大的安全隐患。因次在开发一段时间后只能放弃，换用其他方式进行持久化存储。

考虑到上述理由，我选择了map+yaml存储，使用aes算法和用户密钥（本设计中最安全的密钥）进行加密。

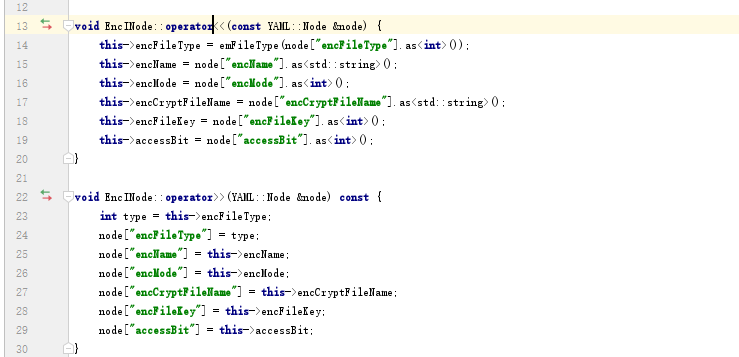
具体实现：

文件表采用类似unix方式，但只保留owner（r,w)other(r,w)，文件列表通过yaml加密存储，文件扁平化设计，只有一层，通过类似linux的iNode方式管理，通过路径虚拟出文件夹。

因次选择自己早轮子，考虑到yaml的c++库比我所接触到的json的c++库使用要方便很多，所以这里使用yaml-cpp作为序列化方式，采用一张hashmap在内存中存储数据，实现快速查找，通过yaml序列化这张表，然后进行加密，本地存储的思路进行管理。

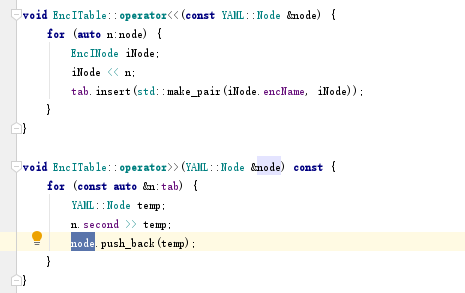
std::string encName：文件在显示时使用的名字；int encMode：文件加密的方式，如des-cbc，des-ebc等；std::string encCryptFileName：如果时加密文件，则它在磁盘中存储的文件名或索引文件名；int encFileKey：文件的控制权限位标记（r，w）；int encFileKey：用户的文件密钥id；int accessBit：访问权限。

这些就是全部要保存的数据了，比较简单，不涉及修改时间，访问时间之类的复杂逻辑，只是记录最基本的信息，而其他的信息，则使用encCryptFileName所指向的文件的数据。然后，通过yaml将其序列化为一段字符转，作为每一条记录的原始数据，或者从解密的yaml中加载进内存。



这样每一条元数据的加载与存储的问题就解决了。当然，这个元数据是不负责加密与解密的，因为它只是一条记录，可以是文件，也可以是文件夹，甚至可以是一个没有加密的文件，所以它只是一条记录，加密解密不应该由他负责。

然后是对这个元数据的管理，时通过一张map表进行管理的，同样使用yaml进行存储或者读取，通过yaml的push\_back函数,来记录表：

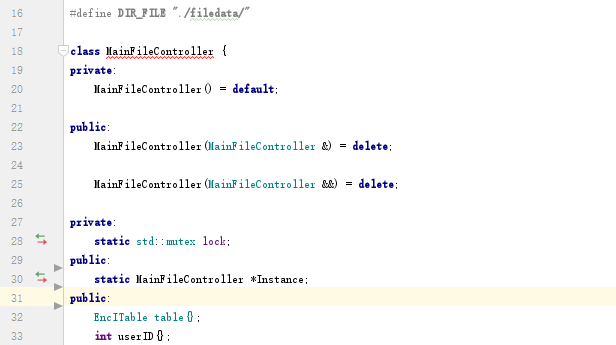


当然，这里也不会做加密解密操作，打的作用只是对上面的iNode进行管理，通过一个map进行管理，c++stl的map使用了红黑树，查找效率比较高，而且也实现了对整张表的序列化，这样可以直接在内从中进行修改，然后在后台同步回文件，速度很快。

至于文件夹，这里不做强制要求，可以加也可以不加，因为实现了std::map<std::string,EncINode> \* selectFileWithDirByName(std::string name,std::map<std::string,EncINode> \*result);这个方法，它会找出某个文件夹下的所有文件，如果某个文件深于两层，那么只会把它的父目录添加进result中，这样，在表中完全不用理会文件夹，而是在具体的拉某个目录时，再计算出文件夹，方便其他逻辑操作。

通过这个EncITable，完全实现了增删改查，当然这只是对记录，文件的加密解密操作不由他负责。

然后就是它的持有者了，这个持有者将通过组件模式，加载各种功能，通过单例模式保持全局唯一，实现文件操作的全部功能。

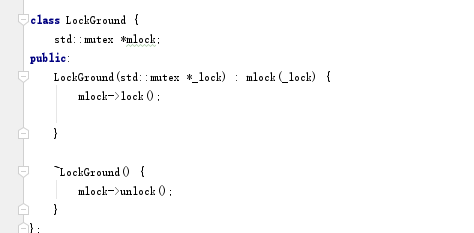
 首先，关掉拷贝，移动构造函数；MainFileController(MainFileController &) = delete;MainFileController(MainFileController &&) = delete;这是c++ 11的新特性。

然后让默认构造函数为private的MainFileController() = default;这也是c++ 11 的新特性。阻止它的拷贝，移动和构造，全方面防止手误导致单例模式被打破，使用饿汉模式进行单例实现（这里不需要懒汉模式的懒加载，应为他们启动时都是要构造的，至于加载表，则是推迟到了用户登录之后，饿汉模式不会影响想能，还不会有线程安全问题）

然后保持一个static的指针，和一个全局锁，当进行操作时必须加锁，这里是为多线程做准备，因为gui中一定会用到多线程操作，而除了dump和加载表，其他操作都是非常快的，这个消耗完全可以接受。

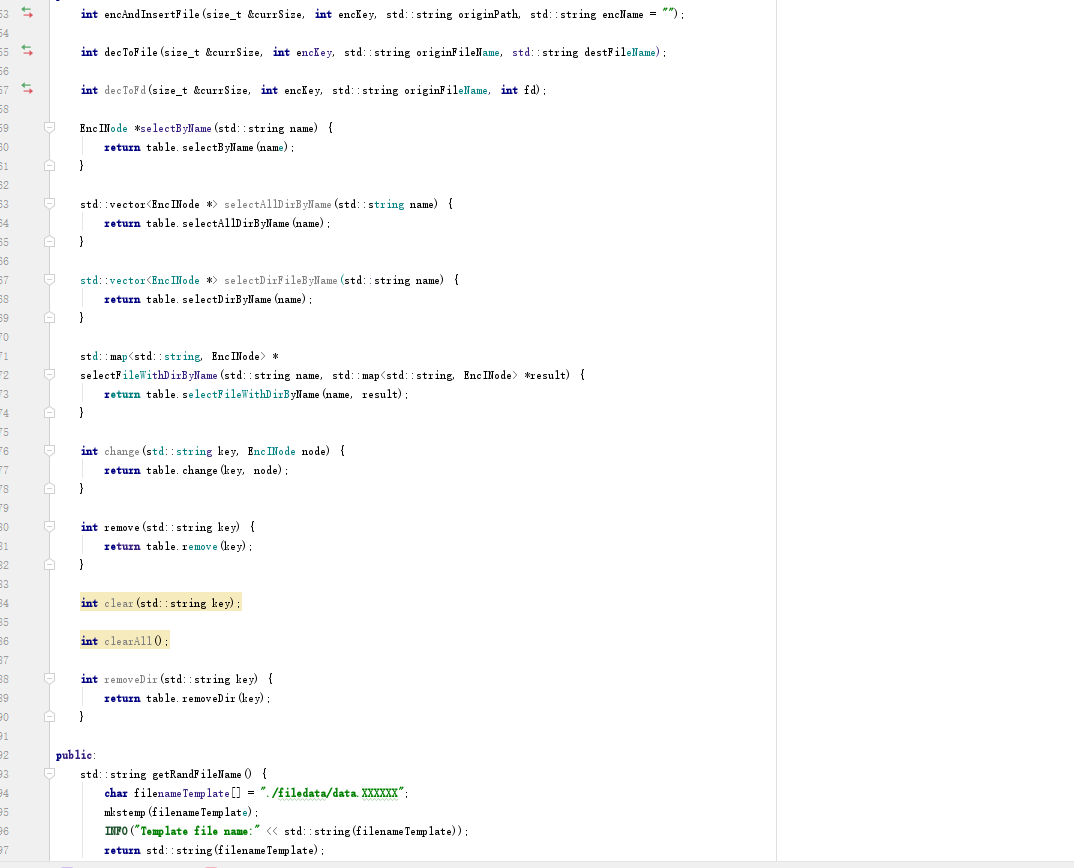
然后通过组件模式让他拥有文件表的增删改查的功能，并对所有文件相关的功能进行一次包装，对外之暴漏他的接口。

锁的释放：



LockGround为通过西沟函数实现的一个无论如何都会释放锁的ground，防止造成死锁，在lockGround被析构时调用释放锁的函数，这样，即使发生错误退出了本函数，也一定会释放锁的。

对外接口：



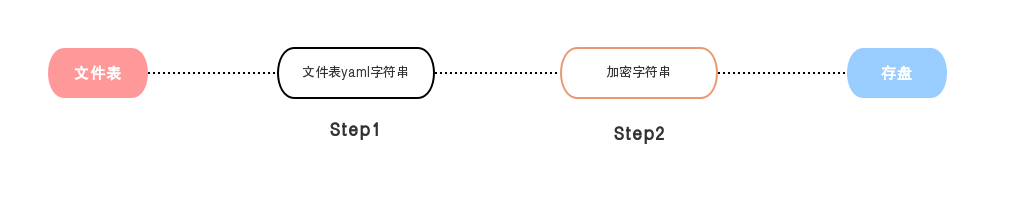
这就是文件管理模块对外提供的所有接口了，大部分都是对文件表的简单操作，但是有几个特殊的功能：

encAndInsertFile：加密文件并且添加到文件表中；decToFile：解密一个文件到指定的位置；decToFd：解密一个文件到指定的文件描述符（为网络访问做铺垫）；这三个函数都是对前面实现的各种功能的调用。

getRandFileName：获取一个不会重复的文件名（一定数量先）这里使用mktemp函数；使用"./filedata/data.XXXXXX"作为模板，会随机出一个不会重复的文件名，将XXXXXX替换为目标文件名。

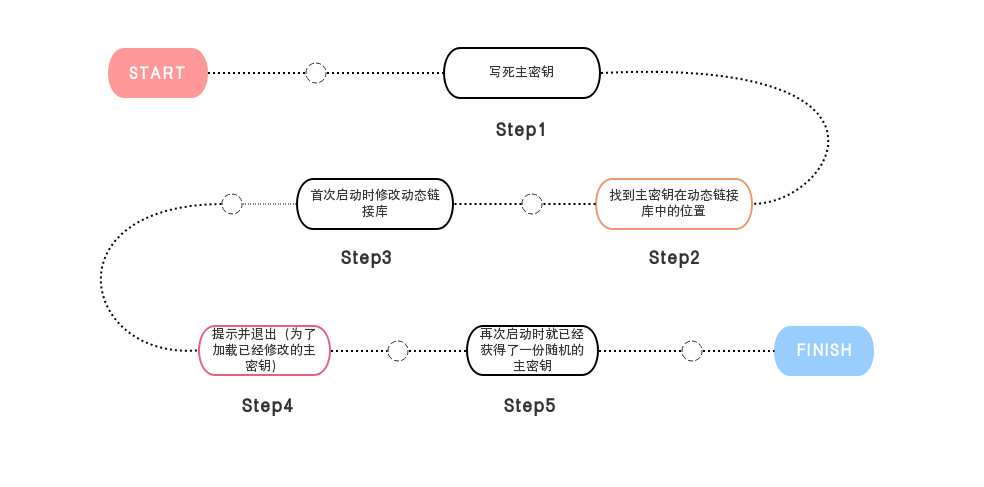
存盘/加载：

当要将表写回磁盘时，先将其序列化为一段yaml字符串，然后使用用户秘钥（后文再说它的实现方式）进行加密，最终写回磁盘，加载同理。



**4.修改动态链接库的主密钥：**

src/main\_passwd：主密钥的管理器和src/change\_rodata：修改已编译动态链接库中的主密钥，它们共同构成了主密钥：



4.1.安装时创建一个全局秘钥

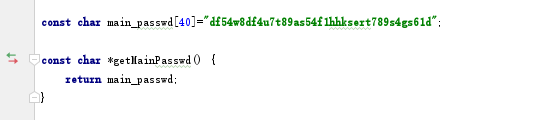
4.2.全局秘钥在代码中写死在动态链接库中

4.3.（初次启动）时修改该动态链接库中的秘钥文本

4.4.这样就获得了一份直接写在动态链接库中的随机公用秘钥

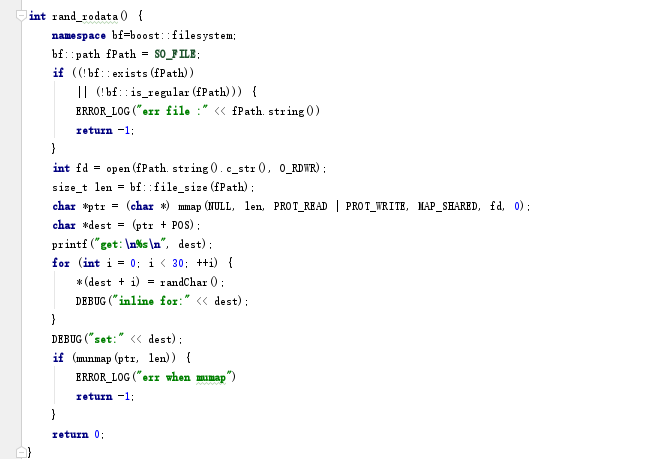
实现方法为：c++内存模型中.data保存有初始化的全局变量，.rodata保存全局常量，只要在源码中声明一个很长的字符串，那么它就会被保存在一个固定的地址，可以mmap该库，直接修改对应位置。

在src/main\_passwd，主密钥的管理器中，非常简单的记录了一段字符串做为主密钥：



它非常简单，就是返回一个写死的字符串，但是这是不能作为主密钥的，因为这意味着一旦一个用户的主密钥被枚举破解，那么所用用户的主密钥都将被破解。所以通过下面的包进行秘钥修改：

src/change\_rodata：修改已编译动态链接库中的主密钥



mmap()系统调用使得进程之间通过映射同一个普通文件实现共享内存。普通文件被映射到进程地址空间后，进程可以像访问普通内存一样对文件进行访问，不必再调用read()，write（）等操作。 实际上，mmap()系统调用并不是完全为了用于共享内存而设计的。它本身提供了不同于一般对普通文件的访问方式，进程可以像读写内存一样对普通文件的操作。[[9]](#endnote-8)使用mmap函数将这个so文件映射到内存中来，而非调用，这样我们只要能知道主密钥在动态链接库中的位置，就可以做到在用户使用时为他随机一份主密钥出来。

首先得找到主密钥在动态链接库中的偏移:

Linux下使用objdump -tT命令，可以查看一个动态链接库的符号表：

libmain\_passwd\_dev.so： 文件格式 elf64-x86-64

SYMBOL TABLE:

00000000000001c8 l d .note.gnu.build-id 0000000000000000 .note.gnu.build-id

00000000000001f0 l d .gnu.hash 0000000000000000 .gnu.hash

......

0000000000201018 l d .data 0000000000000000 .data

0000000000201020 l d .bss 0000000000000000 .bss

......

0000000000000000 l df \*ABS\* 0000000000000000 main\_passwd.cpp

00000000000005c0 l O .rodata 0000000000000028 \_ZL11main\_passwd

......

可以看到 .rodata在0x5c0位置，转换为十进制就是1472，而我们只有一个全局常量，那么它记录的就一定是主密钥；

使用一个测试main进行查看：

char \*ptr = (char \*) mmap(NULL, len, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd, 0);char \*dest = (ptr + pos);这里，直接将so映射到内存中了，然后对这个内存加上1472的偏移量，就会指向我们的主密钥了。我们可以通过GDB查看这段字符是不是我们的密钥：

(gdb) p dest

$2 = 0x7f19c61d05c0 "df54w8df4u7t89as54f1hhksert789s4gs61d"

这个和我们的主密钥是一致的，因此可以确定这里就是我们的主密钥，然后就像操作数组一样对主密钥进行修改，为了清楚一点，这里将目标改为了字母o，只是为了看每次改到哪里，具体实现中会改为随机字符：

[2019-04-10 22:28:28.390500] [0x00007f603b238b80] [debug] inline for:of54w8df4u7t89as54f1hhksert789s4gs61d] [/home/tao/linux\_based\_projects/personal\_file\_protector/src/change\_rodata/main.cpp:36 int main(int, char\*\*)

......

[2019-04-10 22:28:28. 391229] [0x0 00 07f603 b23 8b 80] [deb ug] set:o oooooo ooooo ooooo ooo ooooooo ooos4 gs61d] [/home/tao/linux\_based\_projects/personal\_file\_protector/src/change\_rodata/main.cpp:38 int main(int, char\*\*)

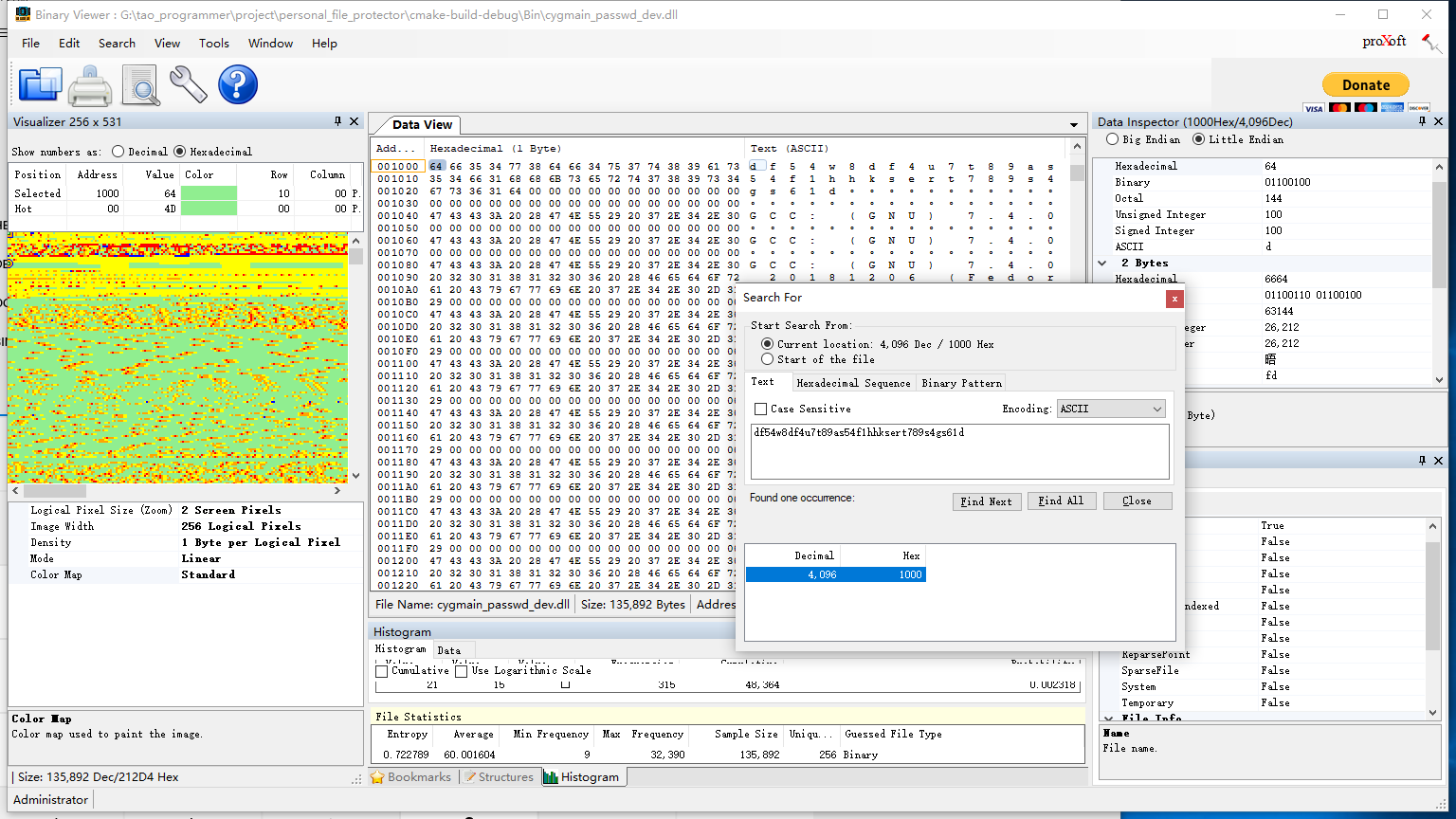
通过不断的日志我们确实可以看到主密钥一步步被修改，最后通过munmap(ptr,len)函数解除映射，将数据写回磁盘，这样改写会动态链接库，再次检查，发现库中的数据已经被修改：

(gdb) p dest

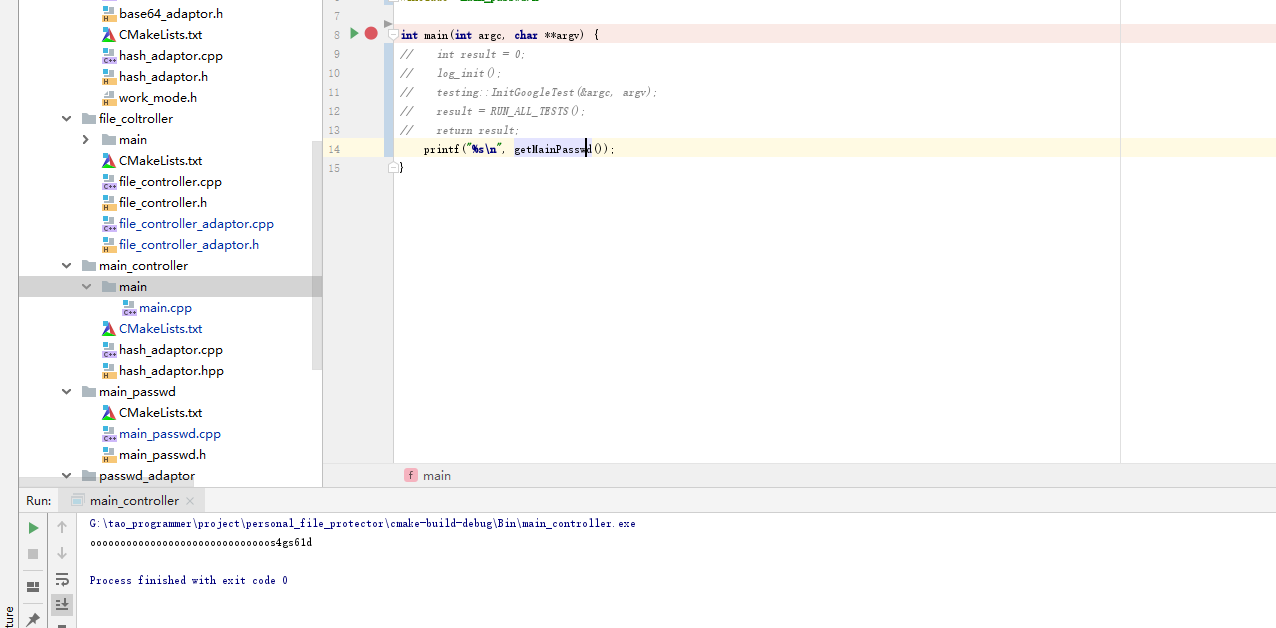
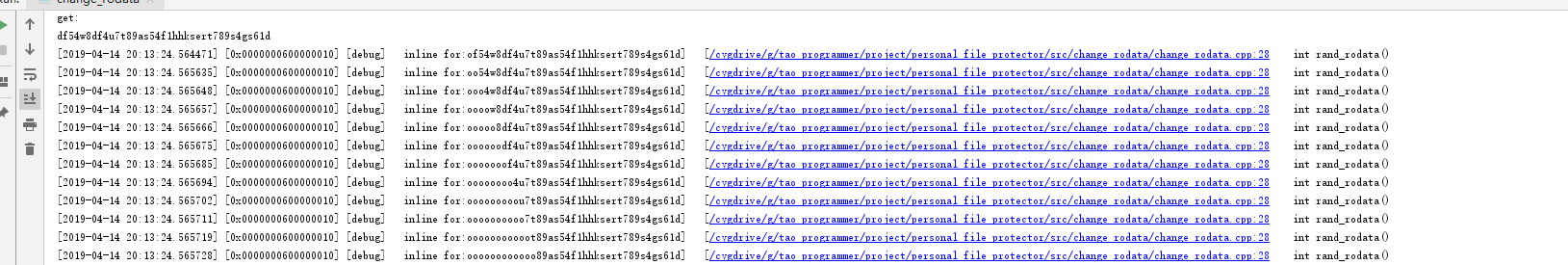
$1 = 0x7f28ca 5c05c0 "oooooo oooo oooooooooo ooooooo ooos4 gs61d"

在使用objdump检查库函数符号表发现无误。证明修改后的库可以使用。至于为什么不全修改，这里最好保留一两位buff，因为一旦修改超了，那么这个so文件将无法使用。

而在Windows下，动态链接库并没有这么规整的编码方式，Windows下色dll库会进行一些对齐操作，导致这一招不能使用，但是我们可以借助Binary Viewer这个工具直接查看二进制，并且搜索我们的主密钥，就可以找到对应的位置了：



可以看到，主密钥被对齐到了0x1000，即4096的位置，这是比较奇怪的，但是通过检查发现还是可以使用的：



我们将so库名修改为dll库名，并将偏移指定为4096，重复之前的操作，发现同样时可以使用的。

**5.src/passwd\_adaptor：用户秘钥与文件秘钥的管理器**

这部分主要分为文件秘钥与用户秘钥。他们对外提供两个非常简单的接口：

std::string get\_usr\_key(int para);

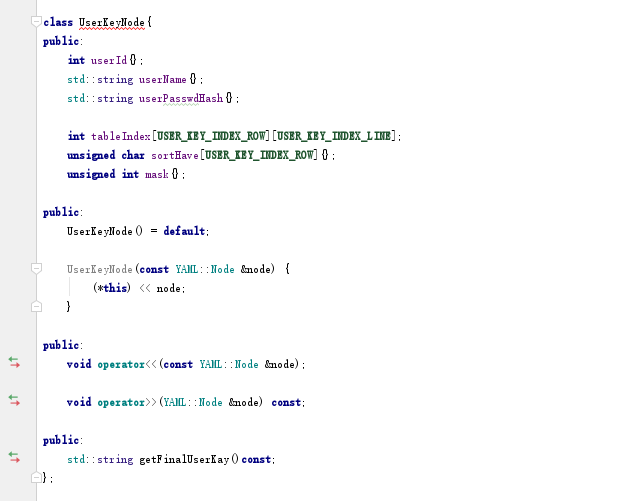
std::string get\_file\_key(int user,int para);

通过用户id（内部实现，用户无感）或与用户密钥，通过用户id和文件密钥id或与文件密钥。

用户名和密码的hash将通过主密钥加密，通过yaml存储，这个前文已将讲过类似的了这里不在细说。登录时用户输入用户名和密码，但是只是和密码的hash进行比较，即使有人破解了主密钥，也完全不能拿到其他人的密码，只拿到hash时没用的。

而文件秘钥也是类似的情况，通过一张yaml表记录某个用户的文件秘钥，每个秘钥都有id，通过id来获取文件秘钥。这样做的目的是为了以后用户可以自己指定文件秘钥做准备。

至于用户秘钥，这是最重要的一个，也是最需要防范的一个，所以对他做了比较复杂的逻辑：



这里记录了用户的id，用户名，密码，获取用户密钥的关键信息等信息，用户秒哟绝对不会直接加密存储到磁盘上。而是通过大量的关键信息（可能比直接存储用户密钥还要费空间）通过多次运算，最终得到用户密钥。

构建一张很大的秘钥段落表，写死在动态链接库中。



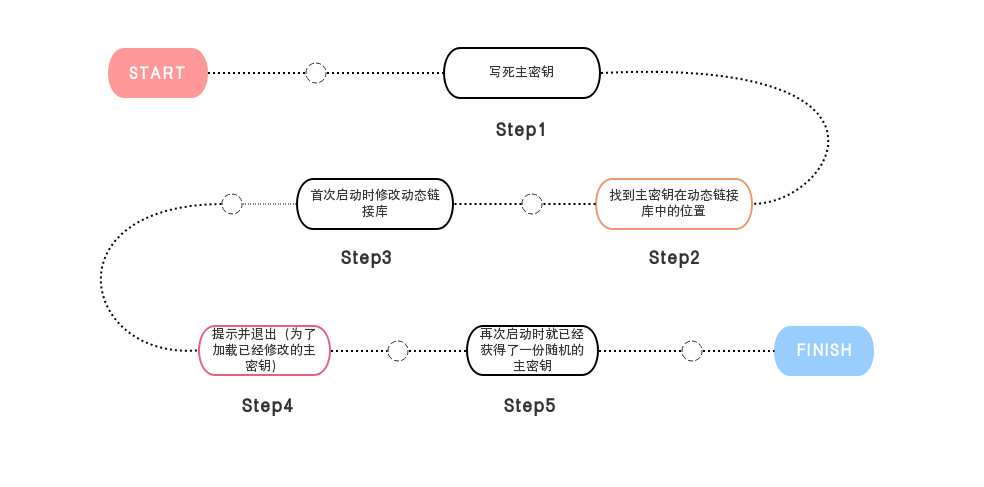
这是一张30\*500的随机字符表，用户密钥的部分数据就是从它里面得到的。

在获取用户密钥的信息中存储以下信息：一个30\*2的二维数组，指示使用密钥表中的那些数据；一个无符号的char[30],将它当short使用，指示了上面数组所指向的数据的排序权重。

然后按照上面的权重对所指向的表中的数据进行排序，构成了初版的用户密钥。

在记录一个无符号整数掩码，使用后30位，将排序好的秘钥，主密钥分割为30份，按这个掩码进行组合，只有掩码中指示可以使用的部分才会被记录到用户密钥结果中，其余部分将会被舍弃，而使用主密钥对应的部分，得到了第二版的用户密钥

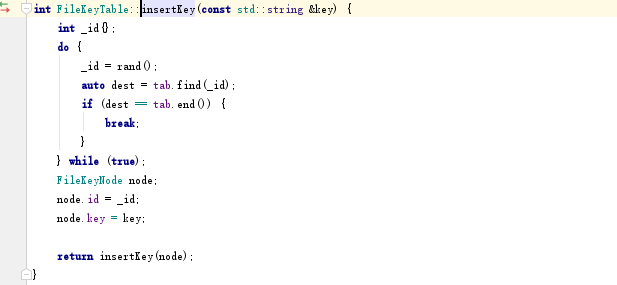
最后将得到的数据按一定的规则进行转换的到用户密钥，这个转换规则可以是base64编码，也可以是MD5或sha256求hash，也可以直接使用，这个可以 随时切换。



用户UserKeyNode通过一张UserKeyNodeTable进行存储，最终通过yaml存储，使用主密钥加密yaml表。至于具体的转换逻辑，这里就不再贴代码进行讲述了，上文类似的已经讲的比较清楚了。他会负责登录，注册，修改密码/用户名等操作，这些都比较简单，就不再细说。

对于文件密钥，因为他会经常变动，甚至用户还会加自己的用户密钥进来，因此，而我们又有一个比较安全的用户密钥，所以直接使用一张表保存id和密钥字符串就好，将序列化的yaml数据使用用户密钥加密存储。

当然他与用户表不同的是，文件密钥是有可能被删除的，所以他的id就不能使用简单的递增的方式了；



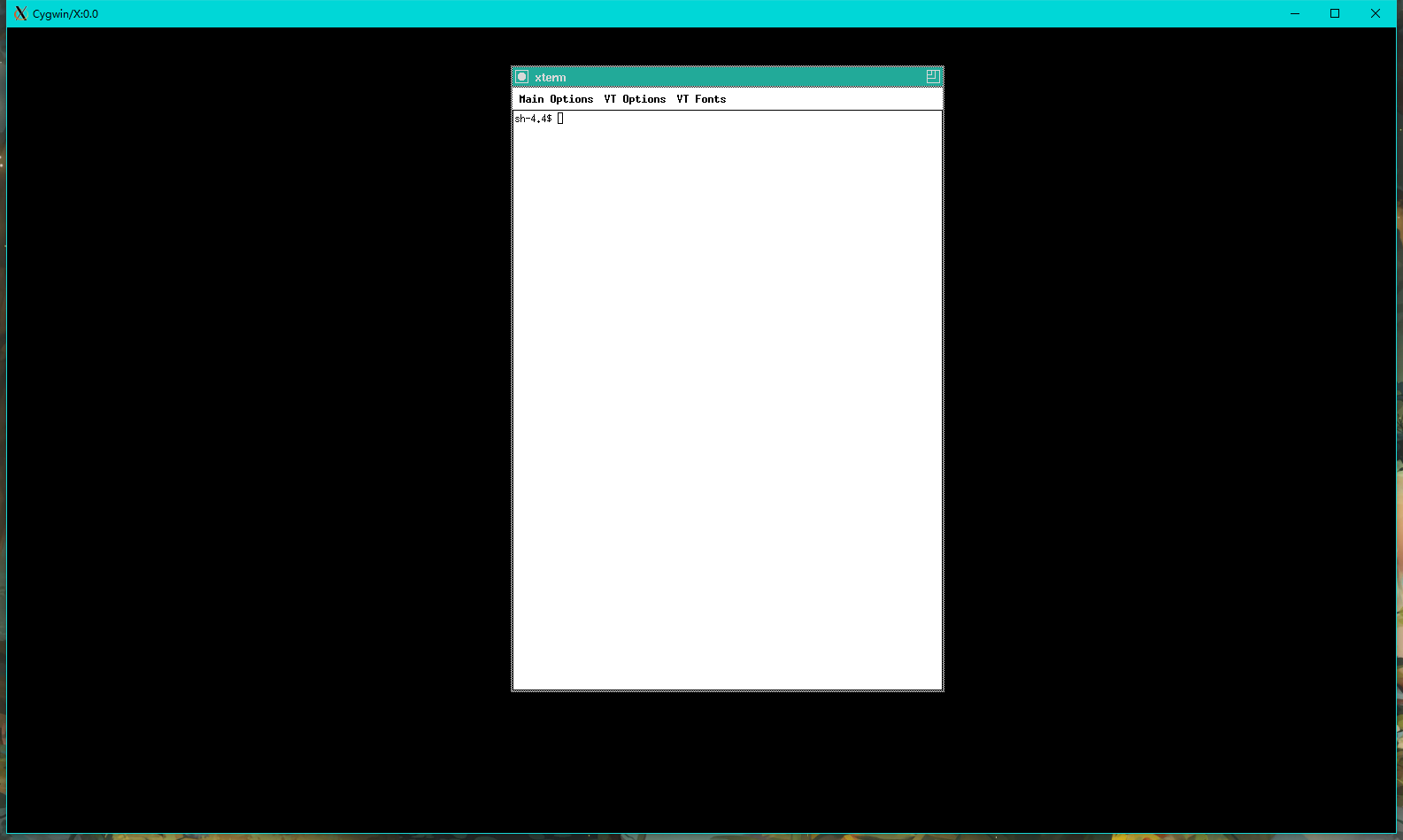
通过随机一个文件密钥->检查是不是已经在文件密钥表中了->再插入的逻辑，就可以使用不会重复的文件密钥了。

**6.src/gui\_qt**

使用核心层提供的接口利用QT库实现一个图形界面，调用逻辑层服务。

原来打算将qt的主界面也实现为一个动态链接库，同时交给main\_controller管理，main controller通过不同的编译选项或者命令行参数，决定是通过命令执行一些逻辑，还是启动qt图形界面。

但是这里有一个很严重的问题，Cygwin中的qt5必须在启动一个x11提供的舞台之后，才可以在这个舞台上展示他的窗口，这就变得非常丑。



就是在这个舞台上在提供窗口服务，原因是Windows不支持x协议。但是，这样的效果太差了，因此只能舍弃这种方式。

由于cygwin对于qt5支持不好，反而，qt是默认支持Windows的，所以又考虑不用费劲的使用cygwin在绕一圈，直接include main\_controller 和 web\_adaptor两个模块的头文件，然后在qt中通过动态链接库的方式加载已经编译好的动态链接库，这样，gui层就绕过了Cygwin。但是这样还是存在问题。虽然Cygwin生成的也是dll库，可以在Windows下直接运行，但是他们和Windows下直接使用vs或者mingw生成的dll库格式还是有所不同的，不能混合调用，这种方式也行不通。

最终我选择了一种万金油的方式：远程访问+命令模式驱动。使用管道（安全）或者socket（不安全）传输客户端的请求，通过命令模式驱动文件加密服务运作，这样，文件加密系统就被当作了一种服务，客户端这是一个命令生成器而已。这样即使客户端崩溃了，用户不小心点击退出了，也不会造成文件加密/解密被中断造成各种意外错误，十分稳定，当然通过管道传输数据肯定不如直接内存访问速度快，但是考虑到个人文件加密系统中，只是传输命令，这都是一些非常短的字符串，最长的也就是返回的某个文件夹下的文件名表，这也不会非常长，完全可以做到无感。

Gui也不一定要使用qt，通过这种方式，可以使用任意支持pipe的语言，如：通过jni技术使用javafx，mfc，python的pyqt等等，gui层完全被隔离出核心逻辑了。

首先要实现主控，塔方负责各种资源的调度，命令的解释，驱动整个系统运作，至于他的客户端是什么，完全不需要它关心，他只负责掌管各种适配器，和根据命令对适配器进行操作。

├── CMakeLists.txt ：子模块的cmakelist

├── main

│   ├── cli.cpp ：下调试使用的一个命令行客户端

│   ├── main.cpp ：启动服务的main函数

│   ├── main\_unix.hpp ：linux下的main函数实现

│   └── main\_win.hpp ：windows下的main函数实现

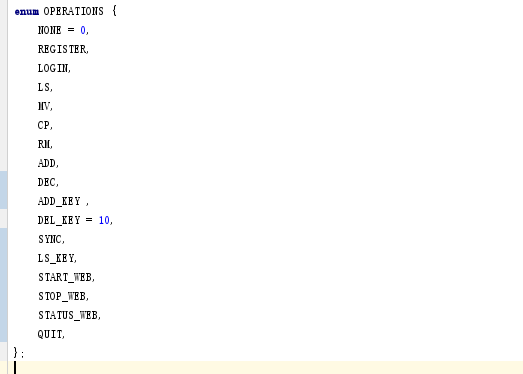
├── main\_controller.cpp ：命令模式驱动的主控

├── main\_controller.h ：命令模式驱动的主控

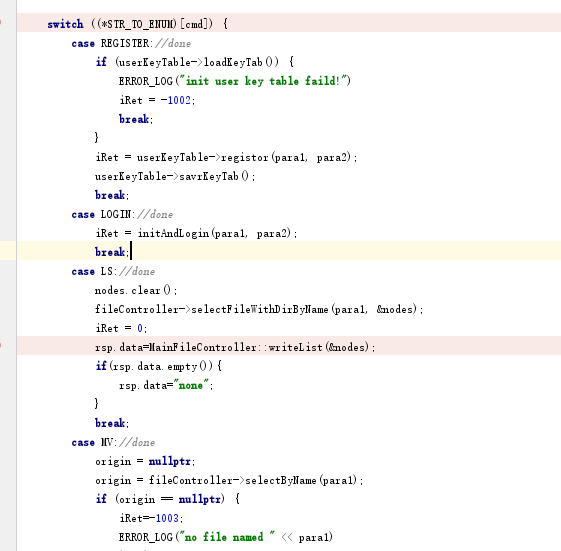
├── pip\_common\_win.hpp ：Windows下的pipe包装

├── pip\_common\_unix.hpp ：Linux下的pipe包装

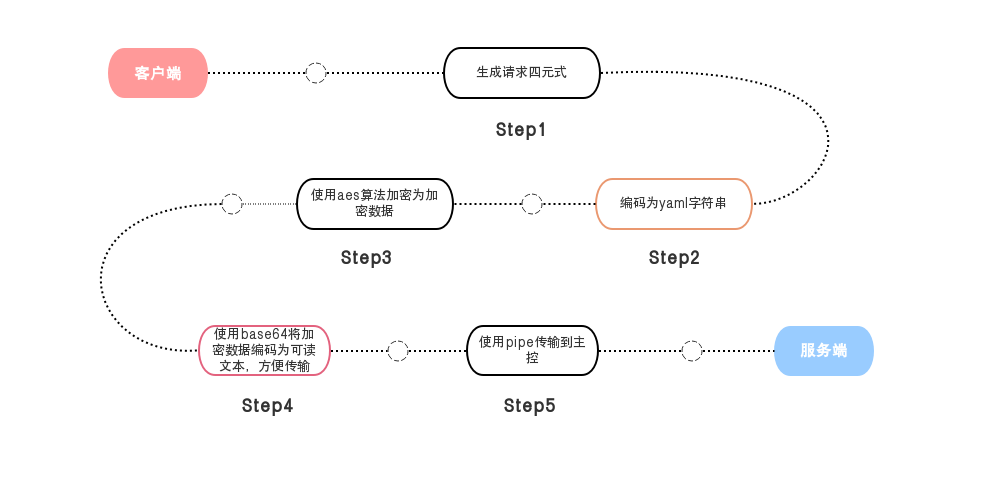
在主控中使用一个enum提供各种命令，通过一个以string为key，int为value的map让c++的switch具有字符串作为key的功能。



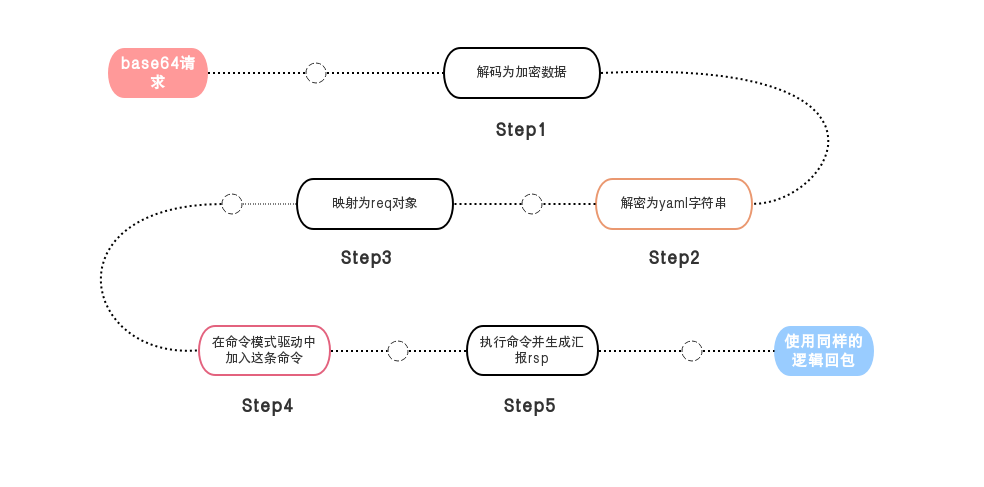
这就是所有的命令了，通过命令和3个额外字符串参数，可以实现常用的所有文件操作的命令。



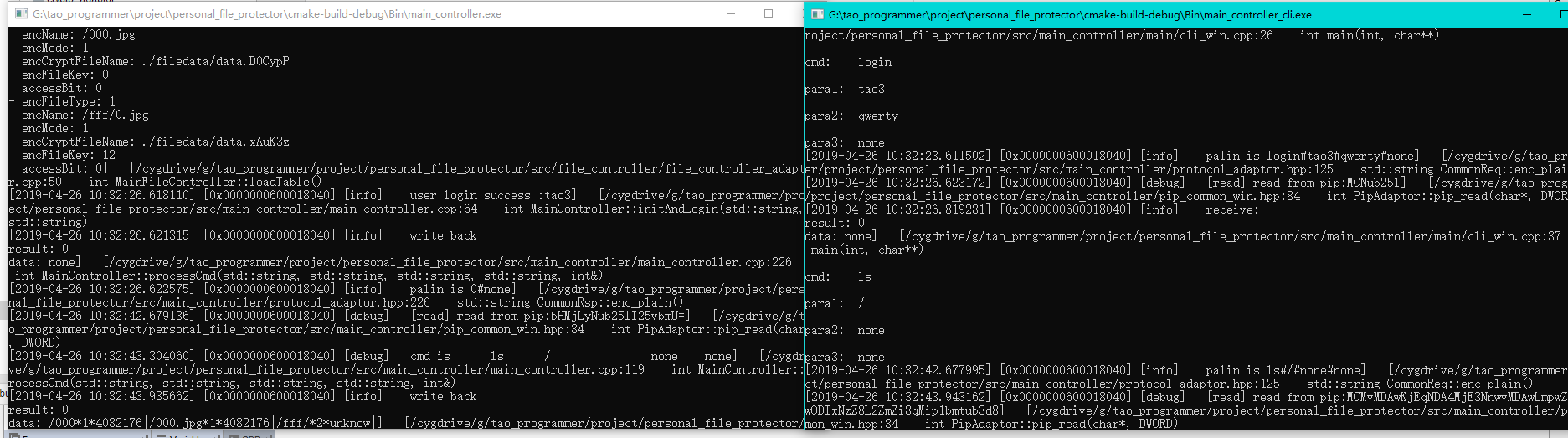
最终在switch中调用包装好的各种方法，驱动整个文件加密系统运作，至于访问的协议，采用req对象->yaml字符串->加密的数据->base64编码的数据的逻辑进行访问。



当请求到达主控后，解包请求，并通过命令模式驱动它允许，并回包。



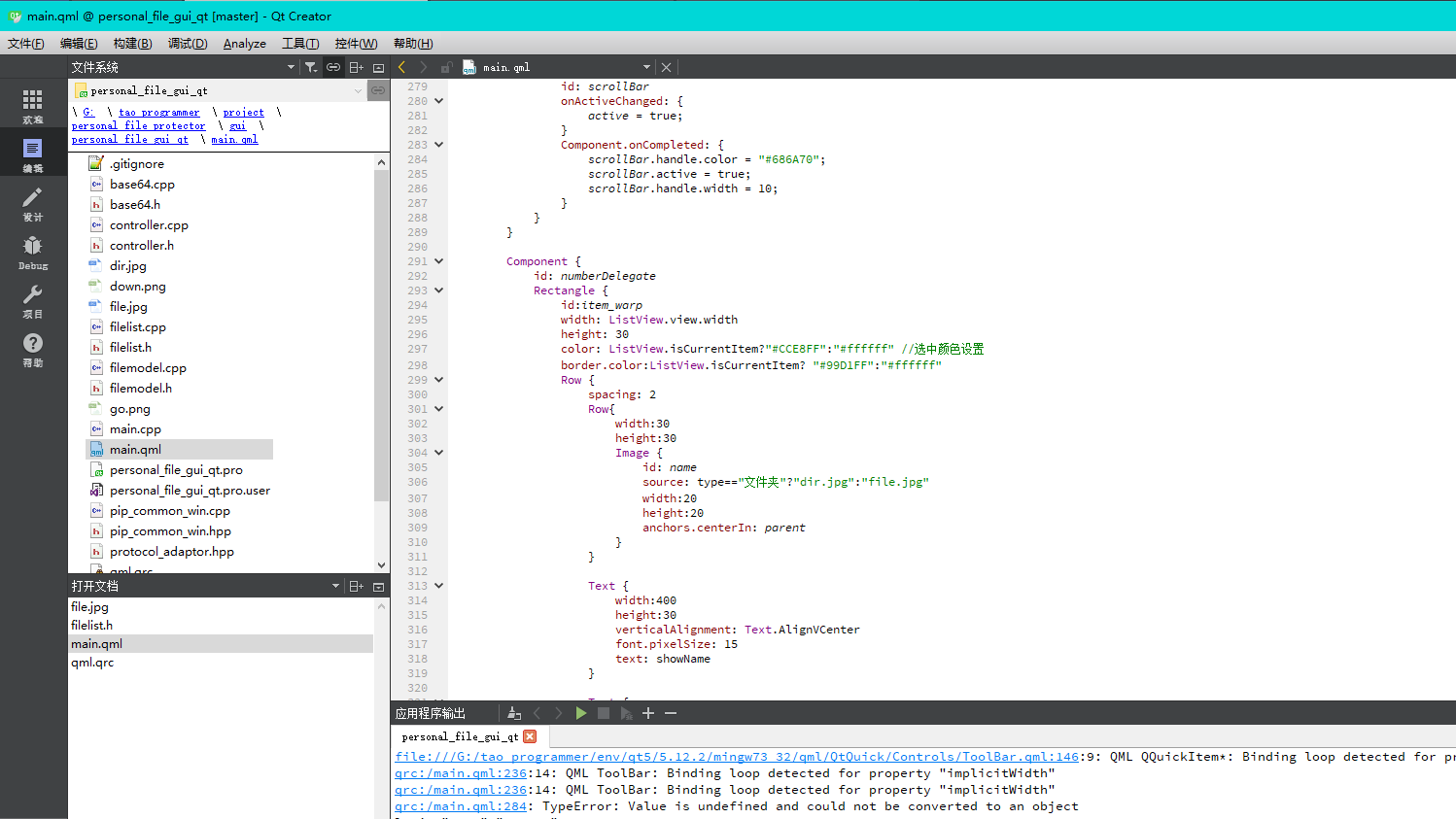
这样，我们就可以使用任意gui库生成我们的gui了，也可以非常简单的使用cli生成命令作为客户端驱动：



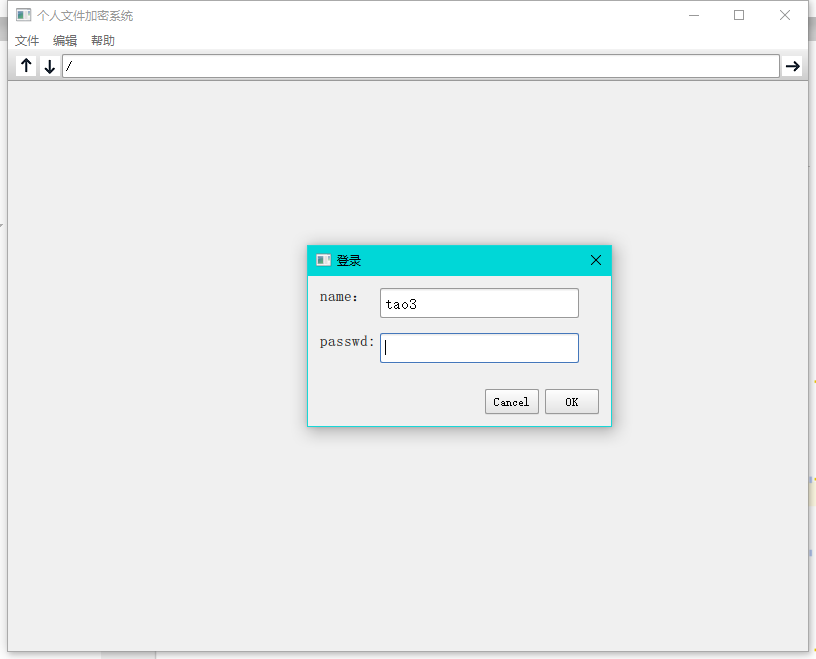
这是一个cli驱动的客户端，当然只是作为测试使用的，它输入测试数据要比gui方便很多。

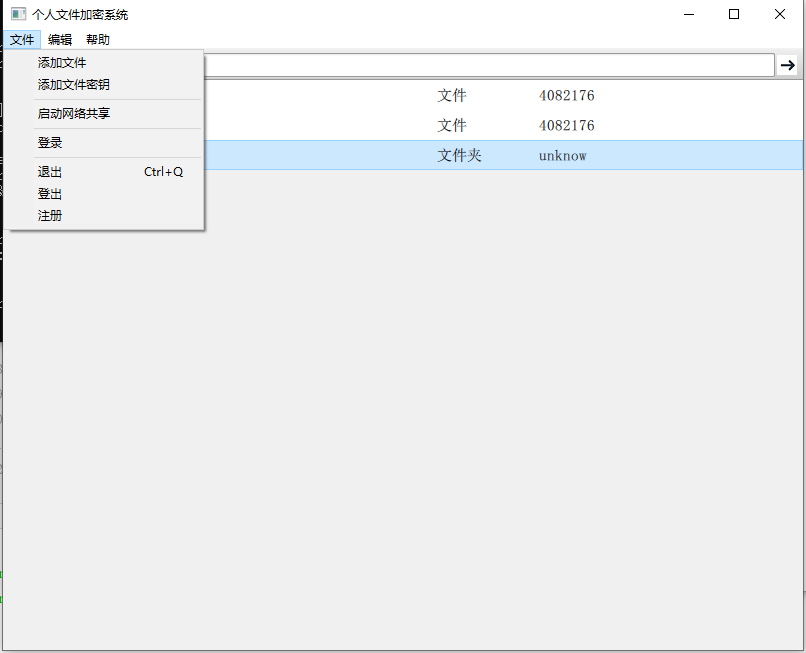
使用qt的qml实现gui，是c++跨平台gui实现的一种简便方式，写起来很想css+JavaScript，性能也很不错，通过信号和槽调用c++中的方法。

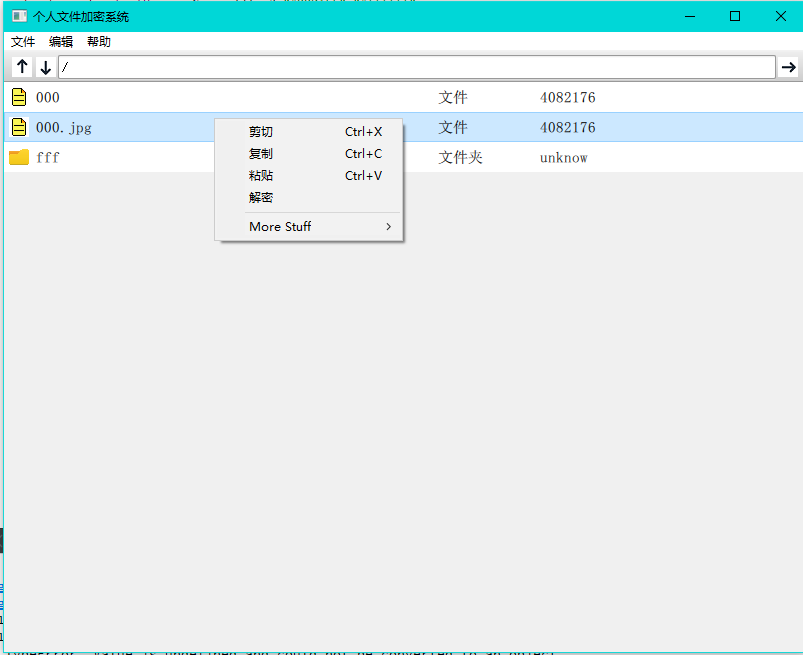
QML 是一种描述语言，并且它们已经被整合到 Qt开发环境中 ， QtQuick 的核心之一就是 QML 语 言 。 Qt5.0开始程序员使用 QquickView 类从 QT 应用程序中载入QML文件，并与之互动。 QML元素是先进的图形元素， 这些图形元素可以 叠加在任意图形界面上，像搭积木那样构建程序界面，这一特性使得 QML 既可以快速创建新的程序界面，也可以非常方便地改造已有的程序界面，QML 还可以在脚本里创建图形对象，并且支持各种图形特效，同时又能跟Qt写的 C++代码进行交互，使用起来非常方便。[[10]](#endnote-9)



下面生成命令和对结果的简单展示。







这里完全仿照windows资源管理器的操作习惯，除了双击时解密到指定文件夹和拖入文件按，拖出文件会慢一些外，其他的和Windows的资源管理器完全兼容。当然这里只是对功能的实现，界面的美化我并不擅长，所以只能先做到这样了。

**7.src/web\_adaptor：简单的网络实时解密浏览实现**

它的接口如下：



非常清晰简单，通过它的拉起的main\_controller将一个文件主控的地址给他，通过组件模式让他也拥有文件主控的功能。

由文件适配器提供获取文件夹下内容的方法和将文件解密到fd的方法，他们只是为了逻辑独立，使用适配器模式对文件主控的一个包装，方便以后更换文件主控，或者文件主控修改接口时，减少要修改的地方。

至于文件服务器，则是一个非常简单的html服务，访问者每次都带回来一个目标文件，如果是文件夹，则调用GetDirHtml，返回新的文件列表的html，如果是文件，那么就先写一个文件html响应头，然后将文件解密到socket的文件描述符中。

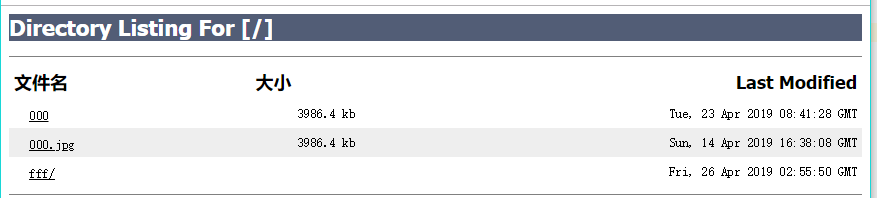
这样用户就可以通过浏览器不用解密就可以访问文件了，而且还支持通过WiFi进行手机访问，如果路由器允许，甚至可以实现外网访问。

但是对于多用户同时访问的实现，则走了一些弯路：刚开始时，采用unix下常用的select/poll模型进行开发：



但是，在实际体验时，发现select当一个用户在在线使用文件时，其他用户会卡死，思考了一下发下犯了一个低级错误:select函数用于确定一个或多个套接口的状态，对每一个套接口，调用者可查询它的可读性、可写性及错误状态信息，用fd\_set结构来表示一组等待检查的套接口，在调用返回时，这个结构存有满足一定条件的套接口组的子集，并且select()返回满足条件的套接口的数目。[[11]](#endnote-10)select本质上还是只有一个线程在工作，当这个线程上的用户调用解密函数时，这个函数不会退出，要花费很长时间才能完成，这段时间内，其他用户就无法得到响应了。

因此，与其在解密函数处调用线程分离主线程，还不如直接使用多线程版的网络模型，简单又不会卡住。Select/poll模型，适用于同时又很多链接同时打开，但同一时间只有很少的链接在活跃的场景，在这种场景下，建立大量的连接，如果使用线程的话，那么线程切换将会消耗大量的资源。而我们的场景刚好相反，只有很少的链接建立，但是，一旦激活传输文件，那么这个链接将会使用很长时间，所以这种情况下，为每一个链接建立一个连接，通过锁控制并发是最好的选择。通过拼接html网页文本，就可以实现简单的网页浏览了。



但是由于现在还没有对网络权限进行验证，依赖客户端的验证，这个模块还需要完善。基于上述理由，网络模块的命令中就只开放了ls命令，不允许网络客户端对文件进行操作。

**第七章 不足与期待**

**1.实现一个NFS服务器**

mmap一段内存，将文件数据也映射到这里来，进行读写访问（整个过程全在内存中，依赖内存的交换机制），最后对整块内存进行加密，写回。可以使用windows自带的mount技术，将NFS挂载到一个盘符下，直接使用windows资源管理器操作（复杂）。

NFS客户端和NFS服务端通讯过程：首先服务器端启动RPC服务，并开启111端口；服务器端启动NFS服务，并向RPC注册端口信息；客户端启动RPC（portmap服务），向服务端的RPC(portmap)服务请求服务端的NFS端口；服务端的RPC(portmap)服务反馈NFS端口信息给客户端；客户端通过获取的NFS端口来建立和服务端的NFS连接并进行数据的传输。

**2.脱离cygwin而实现全平台编译**

在之前的设计中，之所以使用cygwin而非windows自带的vs进行编译，是因为考虑到在不同的平台下,对文件的细节操作完全不同，所以使用cygwin进行统一，而且Windows对select / poll的支持非常弱，以前的设计中是会用到select / poll模型的，基于这些考量，最终选择了Cygwin作为Windows下的编译器。

但是使用Cygwin也不是没有代价的，在Cygwin中很多类库是没有官方版本的二进制包的，必须得自己下载源码进行编译。这是非常慢的。

而且，有一些类库，使用了汇编进行加速，那么这是后的编译就要非常小心，openssl库在编译时，又一些汇编实现的就无法指定平台，只能退而求其次，使用费汇编版本的通过编译，而最早使用的log4cplus日志库，就是由于最新版本对Cygwin有兼容问题，只能换掉，换成了boost：：log。

因此在实现中，我尽量避免使用和操作系统直接相关的函数，在设计中，如果这个量足够小，而且有可跨平台替代版本的话，就可以考虑取消Cygwin。

现在，在所有功能基本实现的情况下，回头统计了一下，发现和底层相关的调用，有：

mmap函数，win版和unix版是不一样的，但可以考虑通过编译宏来控制。

mkstemp，windows下暂时没找到，但是第三方库中又提供，或者自己写一个也行，只是随机一个不重复的文件名，在已经有所有资源的情况下并不难。

网络访问，这一块Windows和unix很多不一样，但是boost库有对一部网络io进行跨平台的封装，可以考虑使用boost：：asio库。

至于其他的操作，文件操作统一使用boost：：filesystem库，加密解密使用openssl crypt库，yaml使用yaml-cpp库，日志使用boost：：log库这些都是跨平台支持的，因此理论上只要把上面的三个问题解决一下就可以抛开Cygwin而实现真真的跨平台了。

**3.对网路访问进行权限验证**

现在是没有网络验证的，只是通过路径进行了用户的区分，但是当这个用户打开网络共享之后，所有可以访问到用户网络的人（如连接了该用户WiFi的人）都可以访问到该用户的文件。

要实现登录鉴权，本身有大量的例子可以参考，但是这又是一个非常庞大的功能，涉及到session的保存，过期，验证，等复杂的逻辑，因此，有两种解决方案：

**3.1自己实现简单的网络链接管理**

HTTP是无状态的，就是前后两个HTTP事务它们并不知道对方的信息。为了维护会话信息或用户信息，一般用Cookie或Session技术缓存信息。Cookie是存储在客户端的，Session是存储在服务端的。

客户端请求服务器时，如果请求的服务涉及Session的访问，如果请求中包含session id，则从session管理器中获取session，如果不包含session id，则会生成一个新的不冲突的session id。一般是通过Cookie随Request和Response在客户端和服务端间传输。在Cookie禁用情况下，也可由URL参数的形式进行通讯。[[12]](#endnote-11)

这种方式逻辑全部集中在自己的代码中，方便发布，没有外部依赖。

**3.2使用apache服务器的cgi功能**

继承一个apache服务器进去，通过cgi的形式实现网络访问，而非自己再去写网络访问逻辑。

Apache服务器本身是有session管理的，省去了自己实现，减少bug的可能。

但是这回导致项目中必须包含一个apache服务器出去，比较臃肿，而且apache所提供的大部分功能我都用不到。

这两种实现方式还得哉评估一下。

**4.网络穿透**

现在的网络访问是可以通过和用户在同一局域网下的用户访问，如该电脑的WiFi链接的手机，同一路由器下的其他用户等等，但是一旦离开了居于网，就无法在访问了，这是因为现在用的ipv4地址不够用，因此我们很难拥有一个通用的地址，大部分时候都是网络管理员分配的局域网地址，在网络出口哪里进行转化，这使得我们要在外网访问我们内网的服务器，会很麻烦，有以下两种解决方案：

**4.1 NET转换**

在NAT网关（如：路由器，局域网出口）上会有一张映射表，表上记录了内网向公网哪个IP和端口发起了请求，然后如果内网有主机向公网设备发起了请求，内网主机的请求数据包传输到了NAT网关上，那么NAT网关会修改该数据包的源IP地址和源端口为NAT网关自身的IP地址和任意一个不冲突的自身未使用的端口，并且把这个修改记录到那张映射表上。最后把修改之后的数据包发送到请求的目标主机，等目标主机发回了响应包之后，再根据响应包里面的目的IP地址和目的端口去映射表里面找到该转发给哪个内网主机。这样就实现了内网主机在没有公网IP的情况下，通过NAPT技术借助路由器唯一的一个公网IP来访问公网设备。[[13]](#endnote-12)

但这需要用户有权限修改网关或者让网关管理员修改，而且ISP分配给普通用户的ip地址通常不是固定的，这又需要我们经常更改链接地址。

**4.2 反向代理**

通过在公共端点和本地运行的 Web 服务器之间建立一个安全的通道，实现内网主机的服务可以暴露给外网。可以考虑把这个功能直接集成到一个工具中，在搭建一个反向代理工具，让有需要的用户拉起反向代理，而没需要的就不用理，杜绝加密管理器联公网的可能（这个加密管理器的设计基本原则之一）。

**5.文件压缩**

可以考虑使用压缩技术将加密后的文件进行压缩，这样可以节省磁盘空间。通用的7z提供的LZMA SDK就很不做，提供较好的c++支持。当然openssl本身也提供压缩支持：

6.1 LZMA SDK includes:

C++ source code of LZMA Encoder and Decoder

C++ source code for .7z compression and decompression (reduced version)

ANSI-C compatible source code for LZMA / LZMA2 / XZ compression and decompression

ANSI-C compatible source code for 7z decompression with example

C# source code for LZMA compression and decompression

Java source code for LZMA compression and decompression

lzma.exe for .lzma compression and decompression

7zr.exe to work with 7z archives (reduced version of 7z.exe from 7-Zip)

SFX modules to create self-extracting packages and installers

ANSI-C and C++ source code in LZMA SDK is subset of source code of 7-Zip.

LZMA features:

Compression speed: 3 MB/s on 3 GHz dual-core CPU.

Decompression speed:

20-50 MB/s on modern 3 GHz CPU (Intel, AMD, ARM).

5-15 MB/s on simple 1 GHz RISC CPU (ARM, MIPS, PowerPC).

Small memory requirements for decompression: 8-32 KB + DictionarySize

Small code size for decompression: 2-8 KB (depending on speed optimizations)

The LZMA decoder uses only CPU integer instructions and can be implemented for any modern 32-bit CPU.[[14]](#endnote-13)

根据7z的官方文档来看是很符合我们的需求的。在加密文件时，先把加密文件写到一个mmap的不存储到本地问文件映射中，然后同时在另一个线程读取这个文件，并进行压缩，输出到目标文件中，这是要注意一定要把已经压缩的部分ummap掉，不然处理大文件时内存会被占满，操作系统很难判断到底哪些该换出。

这个可控性比较高，7z的速度，压缩率都是值得信赖的。

Openssl的COMP\_zlib

COMP\_zlib会返回基于zlib库的COMP\_METHOD，和加密的逻辑保持一致，只需要要加密文件过滤器之后再加一个压缩过滤器就好，实现起来非常方便，由于时间原因，这里没有再进行包装实现。

**第八章 结语**

这个文件个人文件加密管理器就基本实现了，提供了三级密钥管理，主密钥通过so文件写死在动态链接库中，通过直接修改动态链接库来为每个用户提供不同的主密钥；用户密钥通过查表加排序，并和主密钥用掩码进行互补保持，使用主密钥加密的yaml文件按存储相关信息；文件密钥使用用户密钥加密；文件使用文件密钥进行加密；同时提供了gui访问和通过局域网网络不需要解密就进行访问，并且支持手机访问；当然除了主要功能之外，还有一些额外功能没有来的及实现，还需要继续优化。

1. W.Stallings & L.Brown, (2012) “Computer Security, Principlesand Practice”, 2nd/e, Prentice Hall. [↑](#endnote-ref-0)
2. 张华.RSA算法对信息保护的意义和作用[J].科技风,2018(31):78-79+87. [↑](#endnote-ref-1)
3. 李翔宇,于景泽.DES加密算法在保护文件传输中数据安全的应用[J].信息技术与信息化,2019(03):23-25. [↑](#endnote-ref-2)
4. 李青,陈靓,冯梅,李程辉,方静.浅析几种典型数据加密算法[J].信息系统工程,2017(11):148-149. [↑](#endnote-ref-3)
5. 魏革.IDEA加密解密算法的设计与实现策略探究[J].无线互联科技,2015(24):58-59. [↑](#endnote-ref-4)
6. 詹鹏伟,谢小姣.DES与AES算法实现及其在图像加密中的效率探究[J].网络安全技术与应用,2018(09):41-42. [↑](#endnote-ref-5)
7. 裴宏韬,孙建言,盖红玉,李稳.对称加密算法分析及用Java实现[J].电脑知识与技术,2017,13(18):46-47+61. [↑](#endnote-ref-6)
8. 张浩然,曾文潇,蒋同海.用Java和OpenSSL实现认证中心[J].计算机应用研究,2004(05):157-159. [↑](#endnote-ref-7)
9. <https://baike.baidu.com/item/mmap/1322217> 百度百科mmap函数 [↑](#endnote-ref-8)
10. 冯源.QML语言在显控界面开发中的应用[J].电脑编程技巧与维护,2018(02):62-64. [↑](#endnote-ref-9)
11. [https://baike.baidu.com/item/select%28%29/10082180?fr=aladdin](https://baike.baidu.com/item/select()/10082180?fr=aladdin) 百度百科，select函数 [↑](#endnote-ref-10)
12. <https://www.cnblogs.com/nick-huang/p/6660232.html> [↑](#endnote-ref-11)
13. <https://www.jianshu.com/p/cdc446e51675> [↑](#endnote-ref-12)
14. <https://www.7-zip.org/sdk.html> 7z document [↑](#endnote-ref-13)