Pyarmor Documentation

发行版本 *8.2.3*

Jondy Zhao

2023 年 06 月 05 日

Contents

版本

8.2.3

主页

https://pyarmor.dashingsoft.com/index-zh.html

邮箱

pyarmor@163.com

作者

赵俊德

CHAPTER 1

如何阅读本手册

Pyarmor 有完备的文档系统，这里是帮助用户如何快速找到需要的相关内容

- 第一部分: 基础教程 适合第一次使用 Pyarmor 的用户，这里以实例的形式一步接着一步的说明了加密脚本和包的最常用的场景。也可以先看看入门教程
- 第二部分: 应用实践 针对每一个特定的需求，说明在 Pyarmor 中应该如何去做，使用什么样的命令和选项去实现。阅读这部分内容需要对 Pyarmor 和 Python 都有一定的了解。
- 第三部分: 技术手册 从技术层的角度详细列出了所有的概念定义，命令手册，配置选项和错误信息代码。它适用于使用 Pyarmor 的高级用户，需要查找相关的参数和配置，了解这些配置项的可用值和不同值的作用和含义。
- 第四部分: 深入了解 这部分针对 Pyarmor 提供的功能，从如何实现的层面进行了详细的解释。阅读这部分内容需要完全掌握了 Pyarmor 使用到的主要概念，以及对 Python 脚本的执行过程有相当了解。它适用于需要对 Pyarmor 进行扩展和定制，以满足更高一层需求的用户。
- 第五部分: 许可模式和许可证类型 描述了*Pyarmor* 的最终用户许可协议，*Pyarmor* 的许可模式，不同的许可类型，以及如何购买*Pyarmor* 许可证。

CHAPTER 2

获取帮助

使用中遇到问题?

那么首先看看常见问题 –这里详细说明了对于常见问题的通用解决方法，大部分的用户问题按照通用方法的步骤，都可以得到解决。后面的章节则分类列出了用户经常提问的一些问题和解决方案，使用过程中出现问题来这里通常是最快的解决方式。

想找某一个特定的关键字，搜一下 总索引, 或者浏览文档总目录

还是没有找到？看一下如何在 *Github* 上提问.

发现 Pyarmor 的问题，请点击 问题报告 按照模版进行提交。

CHAPTER 3

目录

3.1 使用教程

3.1.1 入门教程

内容

Pyarmor 是什么

Pyarmor 是一个用于加密和保护*Python* 脚本的工具。它能够在运行时刻保护*Python* 脚本代码不被泄露，设置加密后脚本的使用期限，绑定加密脚本到硬盘、网卡等硬件设备。

功能特点:

- **无缝替换**: 加密后的脚本依然是一个有效的 *.py* 文件，在大多数情况下可以直接替换原来的 *.py* 脚本，而不影响脚本的使用。
- **均衡加密**: 提供了丰富的加密选项来平衡安全性和性能，能够满足大多数应用对安全性和性能的要求。
- **不可逆加密**: 能够直接重命名源代码中的函数，类，方法，变量和参数。
- **转换成为 C 代码**: 能够把模块中部分函数转换成为 C 代码，然后使用高优化选项直接编译 C 代码为机器指令来保护 Python 函数
- **限制加密脚本的使用范围**: 可以绑定加密脚本到指定的设备或者设置加密脚本的有效期
- **Themida 保护**: 使用 Themida 保护加密脚本（仅 Windows 平台可用）

安装

Pyarmor 是一个发布在 PyPI 的 Python 包，最方便的方式就是直接使用命令 **pip** 进行安装。

在 Linux 或者 Apple 平台，直接打开终端，然后执行下面的命令:

```
$ pip install -U pyarmor
```

在 Windows 下面，使用 `Win-r` 弹出命令输入框，输入 **cmd** 之后回车打开控制台，在控制台输入安装命令:

```
C:\> pip install -U pyarmor
```

安装完成之后，输入命令 **pyarmor --version** 并回车执行。如果安装成功，会显示出 Pyarmor 的版本信息。

不是所有的平台都被 Pyarmor 支持，所有支持的平台和架构请查看生成加密脚本的环境

加密脚本

下面是最简单的加密命令，用来加密一个脚本 foo.py:

```
$ pyarmor gen foo.py
```

子命令 gen 能够被替换成为 g 或者 generate:

```
$ pyarmor g foo.py
$ pyarmor generate foo.py
```

这个命令会生成一个加密脚本 dist/foo.py ，这也是一个正常的 Python 脚本，可以直接使用 Python 解释器执行:

```
$ python dist/foo.py
```

查看所有生成的文件:

```
$ ls dist/
...    foo.py
...    pyarmor_runtime_000000
```

除了加密脚本之外，可以看到还有另外一个目录 pyarmor_runtime_000000 ，这是运行加密脚本所依赖的一个*Python 包* 。

发布加密脚本

只拷贝加密脚本 dist/foo.py 本身到客户设备 是无法运行的，必须把输出目录下面的运行辅助包 一起拷贝过去才可以。

为什么呢? 看一下加密脚本 dist/foo.py 的内容就明白了:

```
from pyarmor_runtime_000000 import __pyarmor__
__pyarmor__(__name__, __file__, ...)
```

加密脚本需要从 pyarmor_runtime_000000 导入函数 __pyarmor__ ，这个包也是加密脚本的依赖包，加密脚本可以被当作一个正常脚本和依赖包 pyarmor_runtime_000000 来使用。

重要: 因为依赖包 pyarmor_runtime_000000 包含使扩展模块 ，所以加密脚本只能在相同系统，使用相同版本的 Python 才能运行。如果客户设备 的运行环境不一样，需要使用其他跨平台加密选项。

备注: 不需要安装 Pyarmor 到客户设备 ，运行加密脚本不需要 Pyarmor

加密包

现在来加密一个包，使用选项 *-O* 设置另外一个输出目录 dist2:

```
$ pyarmor gen -O dist2 src/mypkg
```

查看加密结果:

```
$ ls dist2/
...     mypkg
...     pyarmor_runtime_000000

$ ls dist2/mypkg/
...           __init__.py
```

测试一下导入加密后的包 dist2/mypkg:

```
$ cd dist2/
$ python -C 'import mypkg'
```

如果包里面还有其他子目录需要加密，那么使用选项 *-r* 来启用递归搜索模式:

```
$ pyarmor gen -O dist2 -r src/mypkg
```

发布加密包

虽然可以把整个目录 dist2 直接拷贝到客户设备，但是还有一种更好的方式，使用选项 *-i* 把运行辅助包保存到包目录内部:

```
$ pyarmor gen -O dist3 -r -i src/mypkg
```

查看输出目录:

```
$ ls dist3/
...     mypkg

$ ls dist3/mypkg/
...           __init__.py
...           pyarmor_runtime_000000
```

现在所有需要拷贝的文件都在加密包 dist3/mypkg 内部，只需要整个包目录拷贝到客户设备 上面就可以了。

备注: 可以比较一下 dist3/mypkg/__init__.py 和上一节生成到的加密文件 dist2/mypkg/

__init__.py 的内容更多的了解这个选项的作用。

封装加密包

再说一次，加密脚本就是正常的 Python 脚本，所以其他用来封装 Python 脚本的工具，例如 distutils，setuptools，以及 wheel 都可以用来封装加密脚本。

假设包 mypkg 的目录结构如下:

```
projects/
└── src/
    └── mypkg/
        ├── __init__.py
        ├── utils.py
        └── config.json
```

首先创建一个输出目录 projects/dist6 用来保存加密包:

```
$ cd projects
$ mkdir dist6
```

然后把所有数据文件到拷贝过去:

```
$ cp -a src/mypkg dist6/
```

接下来生成加密包，把所有加密后的 .py 文件保存到输出目录 .py:

```
$ pyarmor gen -O dist6 -i src/mypkg
```

最终的输出如下:

```
projects/
├── README.md
└── src/
    └── mypkg/
        ├── __init__.py
        ├── utils.py
        └── config.json
└── dist6/
    └── mypkg/
        ├── __init__.py
        ├── utils.py
        ├── config.json
        └── pyarmor_runtime_000000/__init__.py
```

比较一下 src/mypkg 和 dist6/mypkg，唯一的区别是后者多了一个目录 pyarmor_runtime_000000，最后要做的就是使用你熟悉的方式封装 dist6/mypkg

还不了解如何封装 Python 包？请参考这里学习 Python Packaging User Guide

设置加密脚本有效期

使用选项 -e 可以方便的设置加密脚本的有效期。例如，设置加密脚本有效期为 30 天:

```
$ pyarmor gen -O dist4 -e 30 foo.py
```

运行一下加密脚本 dist4/foo.py 来验证一下:

```
$ python dist4/foo.py
```

加密脚本使用 NTP 服务器来验证是否过期，如果当前设备不能访问网络，会报错退出。

也可以使用另外一种格式 YYYY-MM-DD 来设置有效期，例如:

```
$ pyarmor gen -O dist4 -e 2020-12-31 foo.py
```

运行一下 dist4/foo.py 要进行验证:

```
$ python dist4/foo.py
```

如果不需要验证网络时间，可以在有效期前面增加前缀 . 表示检查本地时间。例如:

```
$ pyarmor gen -O dist4 -e .30 foo.py
$ pyarmor gen -O dist4 -e .2020-12-31 foo.py
```

发布有时间限制的加密脚本和上面的方法是一样的，直接拷贝整个输出目录 dist4/ 到客户设备

绑定加密脚本到指定设备

假设客户设备 的硬件信息如下:

```
IPv4:                           128.16.4.10
Enternet Addr:                  00:16:3e:35:19:3d
Hard Disk Serial Number:        HXS2000CN2A
```

使用选项 -b 来绑定硬件信息到加密角本。例如，绑定 dist5/foo.py 到网卡以太网地址:

```
$ pyarmor gen -O dist5 -b 00:16:3e:35:19:3d foo.py
```

使用相同的选项来绑定 IPv4 地址和硬盘序列号:

```
$ pyarmor gen -O dist5 -b 128.16.4.10 foo.py
$ pyarmor gen -O dist5 -b HXS2000CN2A foo.py
```

组合多种硬件信息使用下面的格式:

```
$ pyarmor gen -O dist5 -b "00:16:3e:35:19:3d HXS2000CN2A" foo.py
```

只有设备的硬件信息都符合绑定的信息，加密脚本才能运行，否则报错退出。

发布绑定到设备的加密脚本和上面的方法是一样的，直接拷贝整个输出目录 `dist4/` 到客户设备

关于加密脚本必须要知道的

运行加密脚本需要一个扩展模块 `pyarmor_runtime`，它在运行辅助包 `pyarmor_runtime_000000` 目录下面:

```
$ ls dist6/mypkg/pyarmor_runtime_000000
...    __init__.py
...    pyarmor_runtime.so
```

使用二进制的扩展模块意味着加密脚本需要有为各个平台的预编译的扩展模块 `pyarmor_runtime`，所以加密脚本

- 只能运行在那些已经有预编译扩展模块的平台，所有支持的平台请参考生成加密脚本的环境
- 只能使用相同版本 CPython interpreter 解释器来运行，例如使用 Python 3.8 加密的脚本，无法被 Python 3.9 运行
- 一般不能被第三方解释器，例如 PyPy，IronPython 或者 Jython 等来运行

还有，在 Android 系统下面，`.py` 脚本可以在任意目录下面运行，但是扩展模块是动态库，就必须在系统特定的目录下面才能运行。

下一步的教程

根据你的需要进行选择下一步的教程

这里有完整的安装教程 包含如下内容:

- 从 Github 库直接安装 Pyarmor
- 如何从 Python 脚本中调用 Pyarmor
- 完整卸载

接下来是基础教程 包含的内容有:

- 使用更多选项加密脚本和包
- 使用外部密钥文件限制加密脚本的运行

- 本地化错误信息
- 生成不需要 Python 环境就可以独立运行的加密脚本

还有高级教程，有些功能在试用版中无法使用

- 如何使用两种不可逆的加密模式: RFT 模式和 BCC 模式 [pro]
- 定制错误退出方式
- 国际化错误消息
- 加密跨平台的加密脚本

很多用户可能对这里的内容感兴趣最高安全性和最快性能

如何阅读本手册

Pyarmor 有完备的文档系统，这里是帮助用户如何快速找到需要的相关内容

- 第一部分: 基础教程 适合第一次使用 Pyarmor 的用户，这里以实例的形式一步接着一步的说明了加密脚本和包的最常用的场景。也可以先看看入门教程
- 第二部分: 应用实践 针对每一个特定的需求，说明在 Pyarmor 中应该如何去做，使用什么样的命令和选项去实现。阅读这部分内容需要对 Pyarmor 和 Python 都有一定的了解。
- 第三部分: 技术手册 从技术层的角度详细列出了所有的概念定义，命令手册，配置选项和错误信息代码。它适用于使用 Pyarmor 的高级用户，需要查找相关的参数和配置，了解这些配置项的可用值和不同值的作用和含义。
- 第四部分: 深入了解 这部分针对 Pyarmor 提供的功能，从如何实现的层面进行了详细的解释。阅读这部分内容需要完全掌握了 Pyarmor 使用到的主要概念，以及对 Python 脚本的执行过程有相当了解。它适用于需要对 Pyarmor 进行扩展和定制，以满足更高一层需求的用户。
- 第五部分: 许可模式和许可证类型 描述了*Pyarmor* 的最终用户许可协议，*Pyarmor* 的许可模式，不同的许可类型，以及如何购买*Pyarmor* 许可证。

想找某一个特定的关键字，搜一下 总索引, 或者浏览文档总目录

还是没有找到？看一下如何在 *Github* 上提问.

3.1.2 完整安装教程

内容

前提条件

Pyarmor 需要 Python 运行动态库以及 C 库，缺少它们 Pyarmor 无法正常启动运行。

在 Linux 平台，如有必要请安装 Python 运行动态库。例如，使用下面的命令安装 Python 3.10 的运行动态库:

```
$ apt install libpython3.10
```

在 Darwin 平台，确保文件 `@rpath/lib/libpythonX.Y.dylib` 存在，这里 `X.Y` 表示 Python 的版本。例如:

```
@rpath/lib/libpython3.10.dylib
```

`@rpath` 是下列路径之一:

- @executable_path/..
- @loader_path/..
- /System/Library/Frameworks/Python.framework/Versions/3.10
- /Library/Frameworks/Python.framework/Versions/3.10

如果没有这个文件，请安装必要的包或者使用必要的选项重新编译 Python。

从 PyPI 直接安装

Pyarmor 发布在 PyPI 上面，最方便的方式就是使用命令 **pip** 直接安装。

在 Linux 和 MacOS，直接打开命令终端并运行下面的命令:

```
$ pip install -U pyarmor
```

在 Windows 环境下，需要使用 `Win-r` 打开命令输入框，然后输入 **cmd** 打开命令窗口，并运行下面的命令:

```
C:\> pip install -U pyarmor
```

安装完成之后，输入命令 **pyarmor --version** 并回车。如果安装成功，会显示安装的 Pyarmor 的版本信息。

如果需要跨平台发布加密脚本，还需要安装另外一个包*pyarmor.cli.runtime*:

```
$ pip install pyarmor.cli.runtime
```

并不是所有的平台都支持 Pyarmor，所有支持的运行平台请查看生成加密脚本的环境

安装的命令

- **pyarmor** 是最重要的一个，所有的工作基本都由它来完成，详细使用方法请参考命令手册
- **pyarmor-7** 是为了和老版本兼容的命令，它等价于 Pyarmor 7.x 的修正版本。

使用 Python 解释器直接运行 Pyarmor

pyarmor 等价于下面的命令:

```
$ python -m pyarmor.cli
```

从 Github 上面进行安装

也可以直接从 Pyarmor Github 安装 Pyarmor。下载库到本地，然后使用 pip 进行安装:

```
$ git clone https://github.com/dashingsoft/pyarmor
$ cd pyarmor
$ pip install .
```

你可以直接下载一个库的压缩文件 tar.gz 或者 zip ，解压之后在使用 pip 进行安装。

在 Python 脚本中调用 Pyarmor

首先创建一个任意的脚本，例如 `tool.py`

```
from pyarmor.cli.__main__ import main_entry

args = ['gen', 'foo.py']
main(args)
```

然后运行这个脚本:

```
$ python tool.py
```

以上只是一个示例说明，具体使用的加密选项和参数可以通过各种方式进行传递。

完全卸载

使用下面的命令可以完全卸载 Pyarmor:

```
$ pip uninstall pyarmor
$ pip uninstall pyarmor.cli.core
$ pip uninstall pyarmor.cli.runtime
$ rm -rf ~/.pyarmor
$ rm -rf ./.pyarmor
```

3.1.3 基础教程

内容

本教程仅适用于 Pyarmor 8.0+，使用下面的命令来查看版本信息:

```
$ pyarmor --version
```

如果 Pyarmor 的版本小于 8，那么请选择相应版本的文档 (在页面的左下角可以选择版本)，或者升级 Pyarmor 到正确版本。

在整个教程中，使用的例子结构如下:

```
project/
    ├── foo.py
    ├── queens.py
```

(续下页)

(接上页)

```
    └── joker/
        ├── __init__.py
        ├── queens.py
        └── config.json
```

Pyarmor 使用*pyarmor gen* 加密不同的脚本，它提供了丰富的选项，以满足不同应用关于性能和安全方面的各种需求。

这里仅仅介绍的是常用的一些功能，关于完整的命令选项请查阅命令手册

调试模式和跟踪日志

当加密过程出现问题的时候，检查控制台的输出日志可以帮助发现问题所在，而使用选项 `-d` 启用调试模式会打印更多的信息帮助发现错误:

```
$ pyarmor -d gen foo.py
```

跟踪日志用来记录那些函数被 Pyarmor 使用什么方式进行了保护，它通过下面的方式启用:

```
$ pyarmor cfg enable_trace=1
```

启用之后，每一次执行命令*pyarmor gen* 都会生成一个跟踪日志文件 `.pyarmor/pyarmor.trace.log` 记录相关的保护信息。例如:

```
$ pyarmor gen foo.py
$ cat .pyarmor/pyarmor.trace.log

trace.co              foo:1:<module>
trace.co              foo:5:hello
trace.co              foo:9:sum2
trace.co              foo:12:main
```

每一行开头的 `trace.co` 表示是默认的加密模式，后面的函数名称表示该函数使用默认的方式进行加密。

使用下面的方式禁用跟踪日志:

```
$ pyarmor cfg enable_trace=0
```

使用更多的选项加密脚本

对于脚本，可以使用这些选项来增加安全性:

```
$ pyarmor gen --enable-jit --mix-str --assert-call --private foo.py
```

选项*--enable-jit* 通过使用动态指令生成技术处理某些敏感数据来增加安全性。

选项*--mix-str*[1] 能够加密脚本中所有长度大于 8 的字符串。

选项*--assert-call* 能够确保加密脚本中的函数不会被替换。

选项*--private* 能够确保加密脚本的属性不能直接被 Python 解释器直接导入查看。

例如，

```
data = "abcdefgxyz"

def fib(n):
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b

if __name__ == '__main__':
    fib(n)
```

字符串常量 `abcdefgxyz` 和函数 `fib` 会被以如下方式进行保护

```
data = __mix_str__(b"******")

def fib(n):
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b

if __name__ == '__main__':
    __assert_call__(fib)(n)
```

如果函数 `fib` 是加密函数，那么 `__assert_call__(fib)` 返回原来的函数，否则抛出保护异常。

为了查看那些函数和字符串被保护，可以启用并检查跟踪日志:

[1] *--mix-str* 在试用版中不可用

```
$ pyarmor cfg enable_trace=1
$ pyarmor gen --mix-str --assert-call fib.py
$ cat .pyarmor/pyarmor.trace.log

trace.assert.call    fib:10:'fib'
trace.mix.str        fib:1:'abcxyz'
trace.mix.str        fib:9:'__main__'
trace.co             fib:1:<module>
trace.co             fib:3:fib
```

使用更多的选项加密包

如果是加密*Python 包*，不要使用选项*--private*而是使用其他两个选项:

```
$ pyarmor gen --enable-jit --mix-str --assert-call --assert-import --restrict joker/
```

选项*--assert-import*可以检查导入的模块，确保是没有被替换的加密模块。

选项*--restrict*可以确保加密模块只能在加密脚本内部使用，而不能被外部脚本导入。

默认情况下 `__init__.py` 中定义的函数和名称是可以被外部脚本使用的，其他模块定义的函数和名称是不能被外部脚本使用。让我们创建一个测试脚本 `dist/a.py` 来验证一下

```
import joker
print('import joker OK')
from joker import queens
print('import joker.queens OK')
```

运行这个测试脚本:

```
$ cd dist
$ python a.py
... import joker OK
... RuntimeError: unauthorized use of script
```

如果需要导出包中的其他模块，要么不使用选项*--restrict*，要么单独配置模块 `joker.queens` 不使用约束模式:

```
$ pyarmor cfg -p joker.queens restrict_module=0
```

再次加密和测试一下，这次应该可以正常运行:

```
$ pyarmor gen --restrict joker/

$ cd dist/
```

(续下页)

(接上页)

```
$ python a.py
... import joker OK
... import joker.queens
```

拷贝数据文件

很多包都有数据文件，并且运行的时候需要这些数据文件，但是默认情况下不会被 Pyarmor 拷贝到输出路径。

但是 Pyarmor 提供了三种方法可以解决这个问题

1. 在加密之前先把整个包全部拷贝到输出目录，然后加密脚本，这样加密脚本仅仅覆盖原来的 `.py` 文件，而数据文件保留不变:

   ```
   $ mkdir dist/joker
   $ cp -a joker/* dist/joker
   $ pyarmor gen -O dist -r joker/
   ```

2. 通过配置选项，让 Pyarmor 自动拷贝数据文件:

   ```
   $ pyarmor cfg data_files=*
   $ pyarmor gen -O dist -r joker/
   ```

3. 自己编写*加密插件* 来拷贝需要的数据文件

周期性检查运行密钥

使用下面的命令生成的加密脚本，运行的时候会每隔一个小时对运行密钥进行一次检查:

```
$ pyarmor gen --period 1 foo.py
```

绑定加密脚本到多个设备

使用选项 *-b* 多次可以绑定加密脚本到多个设备。

例如，两台设备 A 和 B，以太网地址分别是 `66:77:88:9a:cc:fa` 和 `f8:ff:c2:27:00:7f`，使用下面的命令绑定加密脚本到这两台设备:

```
$ pyarmor gen -b "66:77:88:9a:cc:fa" -b "f8:ff:c2:27:00:7f" foo.py
```

使用外部文件存放运行密钥

在加密脚本的时候指定使用外部密钥:

```
$ pyarmor gen --outer foo.py
```

在这种情况下，加密后的脚本无法直接运行:

```
$ python dist/foo.py
```

需要先使用*pyarmor gen key* 创建一个外部密钥:

```
$ pyarmor gen key -e 3
```

上面的命令会生成外部密钥文件 `dist/pyarmor.rkey` ，拷贝这个文件到运行辅助包:

```
$ cp dist/pyarmor.rkey dist/pyarmor_runtime_000000/
```

这样可以正常运行加密脚本 `dist/foo.py`:

```
$ python dist/foo.py
```

让我们在生成一个新的运行密钥文件，存放在另外一个目录:

```
$ pyarmor gen key -O dist/key2 -e 10

$ ls dist/key2/pyarmor.rkey
```

拷贝这个新文件到运行辅助包去替换原来的:

```
$ cp dist/key2/pyarmor.rkey dist/pyarmor_runtime_000000/
```

外部运行密钥必须至少包含一个约束条件，要么是有效期，要么是设备信息。

外部运行密钥的名称默认是 `pyarmor.rkey`

外部运行密钥也可以存放在其他路径，请参阅外部密钥 中的说明

如何本地化错误信息

运行错误信息可以使用本地语言进行替换定制，例如加密脚本过期之后可以提示用户自己设定的错误信息。

首先在当前目录创建子目录 `.pyarmor` ，然后在其中创建文件 `messages.cfg`:

```
$ mkdir .pyarmor
$ vi .pyarmor/message.cfg
```

编辑这个文件。这是一个 `.ini` 格式的文件，按照自己的需要修改 `error_N` 后面对应的错误信息

```
[runtime.message]

  error_1 = 脚本许可证已经过期
  error_2 = 脚本许可证不可用于当前设备
  error_3 = 缺少运行许可文件
  error_4 = 非法使用脚本

  error_5 = 脚本不支持当前 Python 版本
  error_6 = 脚本不支持当前系统

  error_7 = 加密模块的数据格式不正确
  error_8 = 加密函数的数据格式不正确
```

然后需要重新加密脚本:

```
$ pyarmor gen foo.py
```

如果你想所有的运行密钥错误都显示同样的错误信息，而其他类型的错误还是使用默认值，那么，修改成为下面的样子

```
[runtime.message]

  error_1 = 未授权使用脚本
  error_2 = 未授权使用脚本
```

然后重新加密脚本使之生效。

生成可独立运行的加密脚本

这里的打包是指生成可以在没有 Python 环境独立运行的可执行文件。

Pyarmor 需要使用 PyInstaller 打包好的可执行文件，然后替换其中的脚本为加密后的脚本。

单个可执行文件模式

首先使用 PyInstaller 的选项 `-F` 生成一个单独的可执行文件:

```
$ pyinstaller -F foo.py
```

然后把上一步生成的可执行文件文件 `dist/foo` 传递给 Pyarmor 进行处理:

```
$ pyarmor gen -O obfdist --pack dist/foo foo.py
```

这个命令会加密 `foo.py` ，并使用加密后的脚本替换 `dist/foo` 原来的脚本，处理完成替换原来的 `dist/foo` 。

最终输出的依旧是 dist/foo:

```
$ dist/foo
```

单个目录模式

首先使用 PyInstaller 生成单目录模式的包:

```
$ pyinstaller foo.py
```

所有需要的文件都存放在一个目录 dist/foo，其中保护一个可执行文件 dist/foo/foo 。

把这个可执行文件传递给 Pyarmor 进行处理:

```
$ pyarmor gen -O obfdist --pack dist/foo/foo foo.py
```

和上面一样，加密脚本替换原来的脚本，并输出替换后的可执行文件 dist/foo/foo

运行一下最后生成的可执行文件:

```
$ dist/foo/foo
```

如果打包出现问题，或者想对打包有更多的了解，请参阅详解可独立运行的加密脚本

3.1.4 高级教程

内容

使用 RFT 模式加密脚本 pro

RFT 模式 是一种不可逆的加密模式，它相当于把源代码重新写了一遍，把其中的函数，类，方法和变量的名称进行了重命名。

使用选项*--enable-rft* 启用 RTF 模式[1]:

```
$ pyarmor gen --enable-rft foo.py
```

启用加密模式之后，这个脚本

```
import sys

def sum2(a, b):
    return a + b

def main(msg):
    a = 2
    b = 6
    c = sum2(a, b)
    print('%s + %s = %d' % (a, b, c))

if __name__ == '__main__':
    main('pass: %s' % data)
```

会被转换成为

```
pyarmor__17 = __assert_armored__(b'\x83\xda\x03sys')

def pyarmor__22(a, b):
    return a + b

def pyarmor__16(msg):
    pyarmor__23 = 2
    pyarmor__24 = 6
    pyarmor__25 = pyarmor__22(pyarmor__23, pyarmor__24)
    pyarmor__14('%s + %s = %d' % (pyarmor__23, pyarmor__24, pyarmor__25))

if __name__ == '__main__':
    pyarmor__16('pass: %s' % pyarmor__20)
```

RFT 模式不会修改模块属性 `__all__` 中列出的名称，也不会修改函数所有的参数名称。例如

[1] 这个功能仅在*Pyarmor 专家版* 中可用

```
import re

__all__ = ['make_scanner']

def py_make_scanner(context):
    parse_obj = context.parse_object
    parse_arr = context.parse_array

make_scanner = py_make_scanner
```

上面的脚本会转换成为

```
pyarmor__3 = __assert_armored__(b'\x83e\x9d')

__all__ = ['make_scanner']

def pyarmor__1(context):
    pyarmor__4 = context.parse_object
    pyarmor__5 = context.parse_array

make_scanner = pyarmor__1
```

如果需要查看脚本转换的结果，可以使用下面的命令启用跟踪 RFT 模式[2]:

```
$ pyarmor cfg trace_rft=1
$ pyarmor gen --enable-rft foo.py
```

启用这个功能之后转换后的脚本会保存到目录 `.pyarmor/rft`:

```
$ cat .pyarmor/rft/foo.py
```

现在运行加密后的脚本:

```
$ python dist/foo.py
```

如果出现名称绑定的错误，可以尝试再次对其进行加密，这样可能会解决一些名称错误问题:

```
$ pyarmor gen --enable-rft foo.py
$ python dist/foo.py
```

如果重新两次加密之后还是无法运行加密脚本，请参考深入了解 *RFT* 模式 来解决名称错误问题。

[2] 这个功能仅用于 Python 3.9+

使用 BCC 模式加密脚本 pro

BCC 模式会把部分函数直接转换成为二进制代码，在根本上避免被还原成为 Python 函数。

BCC 模式需要配置*C* 编译器，对于 *Linux* 和 *Darwin* 来说，一般不需要进行配置，只要默认的 `gcc` 和 `clang` 能工作就可以。在 *Windows* 环境下面，可以使用下面任意一种方式配置 `clang.exe` ，目前其它编译器还不支持:

- 如果已经有 `clang.exe` ，只要在其它路径直接运行 `clang.exe` 不出错就可以。如果文件存在，但是无法在任意路径直接运行，可以配置环境变量 `PYARMOR_CC` 来指定这个文件，例如:

```
set PYARMOR_CC=C:\path\to\clang.exe
```

- 从 LLVM 官网 下载并安装预编译版本
- 从 Pyarmor 官网下载 clang-9.0.zip ，压缩包大小约为 26M 左右，里面只有一个可执行文件，解压后存放在根目录 下面，默认是 `%HOME%/.pyarmor`

配置好编译器之后，使用选项*--enable-bcc* 启用 BCC 模式[3]:

```
$ pyarmor gen --enable-bcc foo.py
```

模块级别的代码不会转换转换成为*C* 的函数，模块的任何函数如果使用了不被支持的特性，也不会转换成为*C* 函数，这么没有使用 BCC 模式加密的函数会根据选项使用其他方式进行加密。

为了查看那些函数被转换成为*C* 函数，使用下面的方式启用跟踪模式:

```
$ pyarmor cfg enable_trace=1
$ pyarmor gen --enable-bcc foo.py
```

然后查看跟踪日志，日志中会显示那些转换的函数所在的脚本和行号:

```
$ ls .pyarmor/pyarmor.trace.log
$ grep trace.bcc .pyarmor/pyarmor.trace.log

trace.bcc            foo:5:hello
trace.bcc            foo:9:sum2
trace.bcc            foo:12:main
```

运行加密后的脚本:

```
$ python dist/foo.py
```

如果加密脚本出错，可以根据错误信息里面报告的函数名称，那么通过设置让 BCC 模式不要把转换该函数。例如，如果运行的时候错误和函数 `sum2` 有关，那么不转换这个函数:

[3] 这个功能仅在*Pyarmor 专家版* 中可用

```
$ pyarmor cfg -p foo bcc:excludes "sum2"
```

选项 bcc:excludes 指定函数名称，选项 -p 指定模块名称。如果没有指定模块名称的话，其他所有模块中的同名函数也会被 BCC 模式忽略。增加更多的函数名称使用下面的命令格式:

```
$ pyarmor cfg -p foo bcc:excludes + "hello"
```

如果这样加密脚本还是不能运行，请参考*深入了解 BCC 模式* 来解决问题。

定制错误处理方式

当运行密钥已经过期，或者和当前设备不匹配，以及加密脚本保护异常，默认的处理方式是抛出运行错误的异常，并显示错误信息:

```
$ pyarmor gen -e 2020-05-05 foo.py
$ python dist/foo.py

Traceback (most recent call last):
  File "dist/foo.py", line 2, in <module>
    from pyarmor_runtime_000000 import __pyarmor__
  File "dist/pyarmor_runtime_000000/__init__.py", line 2, in <module>
    from .pyarmor_runtime import __pyarmor__
RuntimeError: this license key is expired (1:10937)
```

如果需要只显示错误信息，那么可以使用下面的方式进行配置:

```
$ pyarmor cfg on_error=1

$ pyarmor gen -e 2020-05-05 foo.py
$ python dist/foo.py

this license key is expired (1:10937)
```

如果需要直接退出，不显示任何信息，可以进行下面的配置:

```
$ pyarmor cfg on_error=2

$ pyarmor gen -e 2020-05-05 foo.py
$ python dist/foo.py

$
```

可以使用下面的任意一个命令恢复默认处理方式:

```
$ pyarmor cfg on_error=0
$ pyarmor cfg --reset on_error
```

这个设置仅仅影响的 Pyarmor 抛出的错误信息，不会影响脚本代码中本身抛出的异常

备注： 如果加密脚本不是被 Python 直接执行，这个选项可能无法不显示错误直接退出。

过滤加密字符串

默认情况下选项*--mix-str* 会加密脚本中所有长度大于 8 的字符串。

但是这样有时候可能会影响性能，如果需要的话，可以通过定制，只加密指定的敏感字符串。

如果只需要加密长度大于 10 的字符串，可以使用下面的设置:

```
$ pyarmor cfg mix.str:threshold 10
```

如果不需要加密两个字符串 __main__ 和 xyz ，使用下面的命令进行过滤:

```
$ pyarmor cfg mix.str:excludes ^ "__main__ xyz"
```

如果不需要加密长度超过 1000 的字符串，可以使用格式为 /pattern/ 的正则表达式来实现。例如:

```
$ pyarmor cfg mix.str:excludes ^ "/.{1000,}/"
```

重新恢复默认的过滤规则，使用下面的命令:

```
$ pyarmor cfg mix.str:excludes = ""
```

如果只需要加密长度在 8 和 32 之间的字符串，可以使用正则表达式和下面的命令:

```
$ pyarmor cfg mix.str:includes = "/.{8,32}/"
```

配置之后可以通过检查跟踪日志查看效果。

过滤需要保护的函数和模块

选项*--assert-call* 和*--assert-import* 能够对函数和模块进行保护，确保它们是被加密的，从而避免一些对加密脚本注入式的攻击。

但是有时候可能会做一些错误的判断，去保护一些没有加密的函数或者模块，这时候就需要增加一个过滤规则把这些没有加密的函数和模块排除出去。

例如，告诉*--assert-import* 忽略模块 json 和 inspect 使用下面的规则:

```
$ pyarmor cfg assert.import:excludes = "json inspect"
```

告诉*--assert-call* 忽略所有以 wintype_ 开头的函数使用下面的规则:

```
$ pyarmor cfg assert.call:excludes "/wintype_.*/"
```

除了保护了不该保护的函数之外，另外一种错误是有些加密函数却没有被自动识别出来并进行保护，这就需要使用下面的方法进行修正。

使用内联标识符修正加密脚本

在加密脚本之前，Pyarmor 扫描每一行，如果发现该行有内联标识符 `# pyarmor:` ，那么会这一行进行额外处理：删除内联标识符和其后的一个空格，保留其后的所有内容。

例如，这个脚本经过处理之后

```
print('start ...')

# pyarmor: print('this script is obfuscated')
# pyarmor: check_something()
```

就会变成下面的样子

```
print('start ...')

print('this script is obfuscated')
check_something()
```

使用内联标识符修正加密脚本的一个实例就是使用 `__assert_armored__` 来保护额外的模块或者函数

默认情况下*--assert-import* 能够自动识别使用 `import` 语句导入的模块并进行保护，但是使用其他方法导入的模块，不会进行处理。例如，使用下面的方式导入的模块就不会进行处理

```
m = __import__('abc')
```

如果我们需要确保模块 `abc` 是加密的，没有被替换的，可以使用加密脚本特有的内置函数*__assert_armored__()* 来检查 `m`

```
m = __import__('abc')
__assert_armored__(m)
```

但是这样修改后的脚本存在一个问题是没有加密的时候，这个脚本无法正常运行，很不方便调试。因为函数*__assert_armored__()* 只有在加密脚本中才存在。

内联标识符很好的解决了这个问题，我们对上面的脚本进行修正如下就可以完美解决这个问题

```
m = __import__('abc')
# pyarmor: __assert_armored__(m)
```

同样，有时候*--assert-call* 也会遗漏一些需要保护的函数，这时候也可以使用内联标识符和*__assert_armored__()* 进行人工保护。例如，下面的例子对函数 `self.foo.meth` 进行人工检查:

```
# pyarmor: __assert_armored__(self.foo.meth)
self.foo.meth(x, y, z)
```

错误信息支持多语言

Pyarmor 提供错误信息的多语言功能，可以根据*客户设备* 的语言设置显示不同语言的错误信息。

为了支持多语言，首先创建 `~/.pyarmor/messages.cfg`

$ mkdir .pyarmor $ vi .pyarmor/message.cfg

这是一个 `.ini` 格式的文件，增加一个节 `runtime.message` 和选项 `languages`:

```
[runtime.message]

languages = zh_CN zh_TW

error_1 = invalid license
error_2 = invalid license
```

这个例子中支持两种语言，其中语言代码和环境变量 *LANG* 中的设置一样。错误信息我们只定制了前两个错误:

- error_1: 许可证已经过期
- error_2: 许可证不可用于当前设备

默认错误信息设置为 `invalid license` ，也就是说除了指定的语言之外，在其他语言环境都显示这个默认错误信息。

定制不同语言的错误消息分别使用两个节 `runtime.message.zh_CN` 和 `runtime.message.zh_TW`

```
[runtime.message]

languages = zh_CN zh_TW

error_1 = invalid license
error_2 = invalid license

[runtime.message.zh_CN]

```

(续下页)

(接上页)

```
error_1 = 脚本超期
error_2 = 未授权设备

[runtime.message.zh_TW]

error_1 = 🄵本許可證已經過期
error_2 = 🄵本許可證不可用於當前設備
```

然后重新加密脚本:

```
$ pyarmor gen foo.py
```

当加密脚本运行的时候,它会检查当前设备的 *LANG* 来设置默认的语言。如果当前环境的语言设置不是 `zh_CN` 或者 `zh_TW` ，那么使用默认的错误信息。

环境变量 *PYARMOR_LANG* 可以用来指定加密脚本的使用的错误信息语言。如果这个环境变量设置，会忽略系统环境变量 *LANG* 。例如，使用下面的方式运行加密脚本将总是显示繁体中文的错误信息:

```
export PYARMOR_LANG=zh_TW
python dist/foo.py
```

生成跨平台加密脚本

在 8.1 版本加入.

生成跨平台运行的加密脚本需要使用选项*--platform* 指定目标设备的平台名称，这里列出了所有支持的运行平台 名称。

例如，在一台 Darwin 开发设备上面生成可以运行在 Windows 下面的加密脚本，需要使用下面的命令:

```
$ pyarmor gen --platform windows.x86_64 foo.py
```

Python 包*pyarmor.cli.runtime* 提供了其他平台的预编译扩展模块，跨平台加密需要首先安装这个包，如果没有安装的话，会提示安装。

如果需要加密脚本可以运行在多个平台上，可以使用*--platform* 多次，指定每一个运行的平台。例如，下面的命令生成的加密脚本可以运行在多个 x86_64 的操作系统:

```
$ pyarmor gen --platform windows.x86_64
              --platform linux.x86_64 \
              --platform darwin.x86_64 \
              foo.py
```

支持多个 Python 版本的加密脚本

在 8.3 版本加入.

下面是加密一个脚本 *foo.py* 同时支持 Python 3.8 和 3.9 的基本步骤

首先要为每一个 Python 版本安装 Pyarmor:

```
$ python3.8 -m pip install pyarmor
$ python3.9 -m pip install pyarmor
```

如果已经购买了 Pyarmor 的许可证，使用任意一个 Python 版本进行注册:

```
$ python3.8 -m pyarmor.cli reg pyarmor-regfile-xxxx.zip
```

同时需要启用内置插件 `MultiPythonPlugin`:

```
$ python3.8 -m pyarmor.cli cfg plugins + "MultiPythonPlugin"
```

使用每一个 Python 版本分别加密脚本，保存在不同的目录:

```
$ python3.8 -m pyarmor.cli gen -O dist1 foo.py
$ python3.9 -m pyarmor.cli gen -O dist2 foo.py
```

最后使用辅助脚本 `merge.py` 来合并两个输出目录:

```
$ python3.8 -m pyarmor.cli.merge -O dist dist1 dist2
```

最终输出的脚本在 `dist`:

```
$ python3.8 dist/foo.py
$ python3.9 dist/foo.py
```

3.1.5 定制和扩展

内容

Pyarmor 使用下面的方式进行定制和扩展

- 使用命令*pyarmor cfg* 修改默认配置
- 使用加密插件 对加密过程和输出文件进行扩展和定制
- 使用脚本补丁 对运行时刻的加密脚本进行扩展和定制

设置运行辅助包的名称

在 8.2 版本加入:[1]

默认情况下运行辅助包的名称是 `pyarmor_runtime_xxxxxx`

这个名称可以被配置成为任何合法的包名称。例如设定名称为 `my_runtime`:

```
pyarmor cfg package_name_format "my_runtime"
```

自定义需要保护的函数和模块

在 8.2 版本加入.

Pyarmor 8.2 新增加一个配置项 `auto_mode` 用来实现自定义需要保护的函数和模块，它的默认值为 `and`，这时候过滤方式和以前的版本是一样的。`and` 的含义是所有的操作对象除了是自动识别之外，还必须满足 `includes` 和 `excludes` 条件。

如果修改其值为 `or`，则表示除了自动识别的函数和模块之外，还需要保护 `includes` 里面的函数。例如，下面的命令,，除了保护自动识别的函数之外，还额外保护函数 `foo` 和 `koo`:

```
$ pyarmor cfg ast.call:auto_mode "or"
$ pyarmor cfg ast.call:includes "foo koo"

$ pyarmor gen --assert-call foo.py
```

下面的命令可以用来保护没有使用 `import` 语句直接导入的加密模块 `joker.card`:

```
$ pyarmor cfg ast.import:auto_mode "or"
$ pyarmor cfg ast.import:includes "joker.card"

$ pyarmor gen --assert-import joker/
```

[1] 试用版本不可以修改运行辅助包的名称，修改后的加密脚本无法运行

使用加密插件修正运行辅助包的依赖项

在 8.2 版本加入.

在使用 Dawin 的设备中，如果 Python 没有安装在标准路径，那么运行加密脚本的时候可能会因为找不到依赖的 Python 动态库而出现装载错误。

如果需要运行加密脚本的环境在这种设备，那么在加密脚本的时候需要修正运行辅助包的依赖库位置。

首先查看一些运行辅助包中动态库 `pyarmor_runtime.so` 的依赖项:

```
$ otool -L dist/pyarmor_runtime_000000/pyarmor_runtime.so

dist/pyarmor_runtime_000000/pyarmor_runtime.so:

    pyarmor_runtime.so (compatibility version 0.0.0, current version 1.0.0)
    ...
    @rpath/lib/libpython3.9.dylib (compatibility version 3.9.0, current version 3.9.0)
    ...
```

如果客户设备 上面没有 `@rpath/lib/libpython3.9.dylib` ，而是 `@rpath/lib/libpython3.9.so` ，那么加密脚本无法被装载。

这时候可以通过插件来修正这个问题，首先创建一个插件脚本 `.pyarmor/conda.py`:

```
__all__ = ['CondaPlugin']

class CondaPlugin:

    def _fixup(self, target):
        from subprocess import check_call
        check_call('install_name_tool -change @rpath/lib/libpython3.9.dylib @rpath/
↪lib/libpython3.9.so %s' % target)
        check_call('codesign -f -s - %s' % target)

    @staticmethod
    def post_runtime(ctx, source, target, platform):
        if platform.startswith('darwin.'):
            print('using install_name_tool to fix %s' % target)
            self._fixup(target)
```

启用这个插件脚本，然后重新加密脚本:

```
$ pyarmor cfg plugins + "conda"
$ pyarmor gen foo.py
```

请根据具体的环境修改上面的插件脚本以满足需要。

参见:

加密插件

使用脚本补丁绑定脚本到 Docker

在 8.2 版本加入.

假设我们要把脚本 app.py 绑定运行在两个 Docker 上面，它们的 id 分别是 docker-a1 ， docker-b2

那么，首先创建一个脚本补丁 .pyarmor/hooks/app.py

```
def _pyarmor_check_docker():
    cid = None
    with open("/proc/self/cgroup") as f:
        for line in f:
            if line.split(':', 2)[1] == 'name=systemd':
                cid = line.strip().split('/')[-1]
                break

    docker_ids = __pyarmor__(0, None, b'keyinfo', 1).decode('utf-8')
    if cid is None or cid not in docker_ids.split(','):
        raise RuntimeError('license is not for this machine')

_pyarmor_check_docker()
```

然后加密脚本，同时把 Docker 的信息存储到运行密钥 中:

```
$ pyarmor gen --bind-data "docker-a1,docker-b2" app.py
```

运行加密脚本以验证其效果，可以增加一些 print 语句在脚本补丁中进行调试。

参见:

脚本补丁 *__pyarmor__()*

使用其他网络时间服务来检查脚本有效期

在 8.2 版本加入.

默认情况下 Pyarmor 是请求 NTP 服务器来验证加密脚本的有效期，如果 NTP 端口没有开放，也可以通过脚本补丁 使用其他网络时间服务器来进行验证。

首先创建脚本补丁 .pyarmor/hooks/foo.py

```
def _pyarmor_check_worldtime(host, path):
    from http.client import HTTPSConnection
```

(续下页)

(接上页)

```
    expired = __pyarmor__(1, None, b'keyinfo', 1)
    conn = HTTPSConnection(host)
    conn.request("GET", path)
    res = conn.getresponse()
    if res.code == 200:
        data = res.read()
        s = data.find(b'"unixtime":')
        n = data.find(b',', s)
        current = int(data[s+11:n])
        if current > expire:
            raise RuntimeError('license is expired')
     else:
         raise RuntimeError('got network time failed')
_pyarmor_check_worldtime('worldtimeapi.org', '/api/timezone/Europe/Paris')
```

然后加密脚本，有效期的设置使用本地时间:

```
$ pyarmor gen -e .30 foo.py
```

这样就可以使用定制的代码检查网络时间。

参见:

脚本补丁 __pyarmor__()

保护运行辅助模块

在 8.2 版本加入.

下面的例子说明如何检查运行辅助模块 pyarmor_runtime.so 的文件内容来确保其没有被修改

首先创建一个补丁脚本 .pyarmor/hooks/foo.py:

```
1 def check_pyarmor_runtime(value):
2     from pyarmor_runtime_000000 import pyarmor_runtime
3     with open(pyarmor_runtime.__file__, 'rb') as f:
4         if sum(bytearray(f.read())) != value:
5             raise RuntimeError('unexpected %s' % filename)
6
7 check_pyarmor_runtime(EXCEPTED_VALUE)
```

第 7 行的 EXCEPTED_VALUE 需要被替换成为实际值，但是这里存在一个问题。每一次加密之后运行辅助模块 pyarmor_runtime.so 是不同的，所以必须在生成运行辅助模块的同时得到其文件字节总和。这个我们可以通过加密插件来实现，在生成辅助文件之后，自动计算字节总和，然后修改补丁脚本

```
# Plugin script: .pyarmor/myplugin.py

__all__ = ['RuntimePlugin', 'CondaPlugin']

class RuntimePlugin:

    @staticmethod
    def post_runtime(ctx, source, target, platform):
        with open(target, 'rb') as f:
            value = sum(bytearray(f.read()))
        with open('.pyarmor/hooks/foo.py', 'r') as f:
            source = f.read()
        source = source.replace('EXPECTED_VALUE', str(value))
        with open('.pyarmor/hooks/foo.py', 'r') as f:
            f.write(source)

class CondaPlugin:
    ...
```

然后启用这个插件:

```
$ pyarmor cfg plugins + "myplugin"
```

最后生成加密脚本，并进行验证:

```
$ pyarmor gen foo.py
$ python dist/foo.py
```

这个例子只是演示如何去做，并不能在实际项目中使用。任何公开源码的检查方式一般都可以找到相应的方法绕过，所以请编写自己私有的检查脚本，这样才能真正的提高安全性。

参见:

脚本补丁

在外部密钥中增加注释

在 8.2 版本加入.

加密脚本检查外部密钥 文件的时候会忽略头部的任何可打印的字符，所以可以在外部密钥文件的开始增加注释，来对这个密钥进行备注说明。

Pyarmor 提供了加密插件可以用来对外部密钥添加注释，下面这个例子会把所有的绑定信息打印到屏幕，并且把有效期写入到外部密钥中:

```
# Plugin script: .pyarmor/myplugin.py

from datetime import datetime

__all__ = ['CommentPlugin']

class CommentPlugin:

    @staticmethod
    def post_key(ctx, keyfile, **keyinfo):
        expired = None
        for name, value in keyinfo.items():
            print(name, value)
            if name == 'expired':
               expired = datetime.fromtimestamp(value).isoformat()

        if expired:
            print('patching runtime key')
            comment = '# expired date: %s\n' % expired
            with open(keyfile, 'rb') as f:
                keydata = f.read()
            with open(keyfile, 'wb') as f:
                f.write(comment.encode())
                f.write(keydata)
```

启用这个插件，然后生成一个外部密钥:

```
$ pyarmor cfg plugins + "myplugin"
$ pyarmor gen key -e 2023-05-06
```

查看外部密钥中的注释:

```
$ head -n 1 dist/pyarmor.rkey
```

参见:

加密插件

3.2 应用实践

3.2.1 最高安全性和最快性能

内容

如何生成最高安全性的脚本

下面的选项都可以用来增加安全性

- `--enable-rft` 几乎不影响性能，是推荐选项
- `--enable-bcc` 能增加函数执行速度，但是可能需要的内存会多一些
- `--enable-jit` 可以防止静态反编译
- `--enable-themida` 可以防止动态调试器，但是对性能影响比较大，并且仅在 Windwos 可用
- `--mix-str` 保护脚本的所有字符串常量
- `--obf-code 2` 能够同时增加反编译 Bytecode 的难度
- `pyarmor cfg mix_argnames=1` 保护函数参数，但是可能导致 annotations 不可用

下面的选项可以隐藏加密模块的属性，外部脚本无法直接导入和使用加密脚本

- 加密脚本使用`--private`，加密包使用`--restrict`

下面的选项可以防止加密脚本和加密脚本中函数被其他人替换成为自己的函数

- `--assert-call`
- `--assert-import`

如何生成最快的加密脚本

使用下面的选项可以生成性能最高的加密脚本，其安全性相当于 *.pyc*

- *--obf-code* 0
- *--obf-module* 0
- `pyarmor cfg restrict_module=0`

如果需要提高安全性，但是对性能又不要影响太大，最好的选择是同时启用*RFT 模式*

- *--enable-rft*

如果有敏感字符串，那么使用*--mix-str* 同时设置过滤条件，仅仅加密这些敏感字符串。如果没有过滤条件，所有的字符串常量都会加密，可能对性能会造成一点影响

- `pyarmor cfg mix.str:includes "/regular expression/"`
- *--mix-str*

不同类型应用的推荐选项

对于服务网络请求类型的应用

如果*RFT 模式* 已经足够安全（通过检查转换后的脚本来确定是否足够安全），那么使用下面的选项进行加密

- *--enable-rft*
- *--obf-code* 0
- *--obf-module* 0
- *--mix-str* 和过滤条件，如果不设置过滤条件能满足性能要求，也可以不设置

如果单独的*RFT 模式* 不能满足安全需求，那么使用下面的选项

- *--enable-rft*
- *--no-wrap*
- *--mix-str* 和过滤条件

对于其他大多数的应用

如果*RFT 模式* 和*BCC 模式* 可用，那么使用下面的选项

- *--enable-rft*
- *--enable-bcc*
- *--mix-str* 和过滤条件
- *--assert-import*

如果*RFT 模式* 和*BCC 模式* 不可用，使用下面的选项

- `--enable-jit`
- `--private`（加密脚本），或者`--restrict`（加密包）
- `--mix-str` 和过滤条件
- `--assert-import`
- `--obf-code` 2

如果需要保护关键函数避免被其他人替换成为没有加密的函数，使用下面的选项

- `--assert-call`，同时检查跟踪日志，确保关键函数被保护，必要的时候使用运行脚本补丁 对这些函数进行保护

如果性能允许的话，启用选项`--enable-themida`，这个选项还是能很大程度的防止调试器的攻击，但就是对性能影响大一些

重构脚本增加安全性

重构主脚本

Pyarmor 在装载完模块之后，会把模块级别的代码进行清理，清理之后即使通过注入方式也无法在获取代码。

但是对于主模块，因为它永远不会执行完，所以模块级别的代码不会被清理。通过把主模块中非必要的代码，移到其他模块，然后在导入这些代码，这样能够提高安全性。

3.2.2 保护运行时刻的数据安全性

Pyarmor 的核心功能是保护 Python 脚本无法被反编译，通过多种不可逆加密模式的实现，已经能够实现 Python 脚本无法使用任何方式完全反编译出来。但是对于内存数据，包括运行时刻的数据保护，Pyarmor 没有进行太多的保护。如果保护的重点是运行时刻的数据，那么就需要额外的工具来进行保护。

Pyarmor 可以确保各种使用 Python 自身提供的机制无法非法获取运行时刻的数据，但是前提是运行加密脚本的 Python 的解释器和扩展模块 pyarmor_runtime 不能被替换或者修改，尤其是在运行时刻使用调试器直接修改内存代码段，这些就是需要额外的方法和工具来提供保护。常见的保护方式有

- 使用操作系统提供的签名验证方式确保可执行文件和动态库没有被替换和修改
- 使用第三方的可执行文件的保护工具例如 VMProtect 等保护 Python 以及 pyarmor_runtime.pyd/.so
- Pyarmor 提供了一些保护选项，能够绑定脚本到解释器，发现调试器（弱）就退出等功能
- Pyarmor 提供了脚本补丁，可以用来添加自定义的函数，进行检测调试器等各种自定义的保护，这种需要专业的能力，但是能够提供足够高的安全性，没有人知道用户自定义的保护代码，哪怕是简单的保护，对任何一个黑客来说都需要花费时间去破解。而对于那些公共的保护技术，往往有成熟的工具可以直接绕过

基本的实现步骤

需要明白的一点是，如果运行 Python 脚本的解释器可以被定制，那么运行时刻的 Python 数据就没有秘密可言，所以必须要保证运行加密脚本的解释器不能被替换。

能实现这一点的加密模式目前 Pyarmor 只提供使用选项 *--pack* 的约束模式，然后使用外部工具保证生成的动态库不能被替换和修改，这样才能够确保无法直接通过 Python 自身的机制来获取运行时刻的数据。

首先是使用 PyInstaller 打包[1]:

```
$ pyinstaller foo.py
```

接着使用下面的命令加密脚本[2]:

```
$ pyarmor cfg check_debugger=1 check_interp=1
$ pyarmor gen --mix-str --assert-call --assert-import --private --pack dist/foo/foo␣
↪foo.py
```

然后使用其他方式来保护 `dist/foo/` 目录下面所有的可执行文件和动态库，外部工具要确保动态库不能被替换以及在运行时候的内存代码不能被修改。

典型的外部工具有 codesign，VMProtect 等。

备注

脚本补丁

为了进一步提高安全性，还可以使用脚本补丁来检查 PyInstaller 的装载代码，确保其没有被替换。

下面的例子只是演示如何实现，请不要直接在项目中使用，并且在不同 PyInstaller 版本中可能会出错，请根据自己的具体情况，参考这个示例编写自己的私有补丁。

```
1  # Hook script ".pyarmor/hooks/foo.py"
2
3  def protect_self():
4      from sys import modules
5
6      def check_module(name, checklist):
7          m = modules[name]
8          for attr, value in checklist.items():
9              if value != sum(getattr(m, attr).__code__.co_code):
10                 raise RuntimeError('unexpected %s' % m)
11
```

(续下页)

[1] PyInstaller 如果使用选项 `--onefile` 打包成为单个可执行文件，这个文件在执行的时候会解压到一个临时目录下面执行，需要保证第三方工具或者签名工具对解压后的文件也能够提供保护，否则是没有保护效果的，因为真正的动态库等相关文件都在这个解压后的目录下面。

[2] 不要在 Intel i686 系列的平台上使用配置项 `check_interp`，这个选项无法在这些平台工作。

(接上页)

```
    checklist__frozen_importlib = {}
    checklist__frozen_importlib_external = {}
    checklist_pyimod03_importers = {}

    check_module('_frozen_importlib', checklist__frozen_importlib)
    check_module('_frozen_importlib_external', checklist__frozen_importlib_external)
    check_module('pyimod03_importers', checklist_pyimod03_importers)

protect_self()
```

目前脚本里面的检查点为空（高亮的行），为了得到真正的检查点，需要先使用下面的一个假函数替换真正的 check_module

```
def check_module(name, checklist):
    m = modules[name]
    refs = {}
    for attr in dir(m):
        value = getattr(m, attr)
        if hasattr(value, '__code__'):
            refs[attr] = sum(value.__code__.co_code)
    print('    checklist_%s = %s' % (name, refs))
```

运行下面的命令以得到真正的检查点，代码行会打印在控制台:

```
$ pyinstaller foo.py
$ pyarmor gen --pack dist/foo/foo foo.py

...
checklist__frozen_importlib = {'__import__': 9800, ...}
checklist__frozen_importlib_external = {'_calc_mode': 2511, ...}
checklist_pyimod03_importers = {'imp_lock': 183, 'imp_unlock': 183, ...}
```

编辑脚本补丁，恢复原来的函数 check_module 并使用生成的代码替换空的检查点。

最后使用真正的补丁脚本来生成最终的包:

```
$ pyinstaller foo.py
$ pyarmor gen --pack dist/foo/foo foo.py
```

启动补丁

在 8.3 版本加入.

用户还可以编写自己的代码去检查调试器和其他任何反调试代码，代码可以使用 Python 实现，在扩展模块 pyarmor_runtime 被装载的时候自动调用。

基本配置方式是创建一个脚本 .pyarmor/hooks/pyarmor_runtime.py，定义一个函数*bootstrap()*来进行额外的检查。例如:

```
def bootstrap(user_data):
    from ctypes import windll
    if windll.kernel32.IsDebuggerPresent():
        print('found debugger')
        return False
```

3.2.3 打包使用外部密钥的加密脚本

下面说明如何加密并打包一个应用程序 src/myapp.py 并使用外部密钥

首先使用 PyInstaller 打包:

```
$ pyinstaller myapp.py
```

接着加密脚本并替换生成的可执行文件，同时指定使用外部密钥:

```
$ pyarmor gen --outer --pack dist/myapp/myapp myapp.py
```

然后生成外部密钥:

```
$ pyarmor gen key -O keylist -e 30
```

对于打包成为单个目录的模式来说，一般存放运行密钥到运行辅助包的目录，例如:

```
$ cp keylist/pyarmor.rkey dist/myapp/pyarmor_runtime_000000/
```

这样就可以在任何位置运行可执行文件 dist/myapp/myapp 。例如:

```
$ dist/myapp/myapp
```

如果是打包成为单个可执行文件，一般把外部密钥和可执行文件存放在相同目录下面，并且重命名为 可执行文件名称. 密钥名称。例如:

```
$ pyinstaller --onefile myapp.py
$ pyarmor gen --outer --pack dist/myapp myapp.py
$ pyarmor gen key -O keylist -e 30
$ cp keylist/pyarmor.rkey dist/myapp.pyarmor.rkey
```

然后这样就可以在任意目录运行可执行文件，例如:

```
$ dist/myapp
```

还有一种情况是把运行密钥存放在一个固定目录下面，然后使用环境变量来指定这个目录。例如:

```
$ export PYARMOR_RKEY=/opt/pyarmor/runtime_data
$ mkdir -p /opt/pyarmor/runtime_data
$ cp keylist/pyarmor.rkey /opt/pyarmor/runtime_data/
$ dist/foo
```

3.2.4 打包加密脚本成为 Wheel

测试项目的目录结构如下:

```
$ tree test-project

test-project
├── MANIFEST.in
├── pyproject.toml
├── setup.cfg
└── src
    └── parent
        ├── child
        │   └── __init__.py
        └── __init__.py
```

文件 MANIFEST.in 的内容如下:

```
recursive-include dist/parent/pyarmor_runtime_00xxxx *.so
```

文件 pyproject.toml

```
[build-system]
    requires = [
        "setuptools>=66.1.1",
        "wheel"
    ]
    build-backend = "setuptools.build_meta"
```

文件 setup.cfg

```
[metadata]
    name = parent.child
    version = attr: parent.child.VERSION

[options]
    package_dir =
        =dist/
```

(续下页)

(接上页)

```
    packages =
        parent
        parent.child
        parent.pyarmor_runtime_00xxxx

    include_package_data = True
```

src/parent/__init__.py 和 src/parent/child/__init__.py 是相同的:

```
VERSION = '0.0.1'
```

首先加密包:

```
$ cd test-project
$ pyarmor gen --recursive -i src/parent
```

加密成功之后输出的目录结构如下:

```
$ tree dist

dist
└── parent
    ├── child
    │   ├── __init__.py
    │   └── __pycache__
    │       └── __init__.cpython-311.pyc
    ├── __init__.py
    └── pyarmor_runtime_00xxxx
        ├── __init__.py
        └── pyarmor_runtime.so
```

接着就构建 wheel:

```
$ python -m build --skip-dependency-check --no-isolation
```

但是却报错了

```
* Building sdist...
Traceback (most recent call last):
  File "/usr/lib/python3/dist-packages/setuptools/config/expand.py", line 81, in __
↪getattr__
    return next(
           ^^^^^
StopIteration
```

(续下页)

(接上页)

```
The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "/usr/lib/python3/dist-packages/setuptools/config/expand.py", line 191, in␣
↪read_attr
    return getattr(StaticModule(module_name, spec), attr_name)
```

检查错误堆栈发现问题出现在 StaticModule，通过异常中文件名称和行号，查看在包 setuptools 源代码中这个类的定义，它使用了 ast.parse 直接解析源代码获取模块的局部变量，这当然在加密脚本中行不通的。为了解决这个问题，可以直接在加密后的脚本 dist/parent/child/__init__.py 中增加一行

```
from pyarmor_runtime_00xxxx import __pyarmor__
VERSION = '0.0.1'
...
```

但是默认情况下加密脚本是不允许修改的，为了能够修改这个脚本，需要使用下面的命令进行配置:

```
$ pyarmor cfg -p parent.child.__init__ restrict_module = 0
$ pyarmor gen --recursive -i src/parent
```

其中选项*pyarmor cfg -p* parent.child.__init__ 是只取消对单个模块的 parent/child/__init__.py 的限制，而其他模块依旧使用约束模式。

现在修改 dist/parent/child/__init__.py 之后再次运行下面的命令进行打包:

```
$ python -m build --skip-dependency-check --no-isolation
```

自定义运行辅助包的名称

如果你想自定义运行辅助包的名称为 libruntime，并且存放到子目录 parent.child 下面，你需要修改 MANIFEST.in:

```
recursive-include dist/parent/child/libruntime *.so
```

和 setup.cfg:

```
[options]
    ...
    packages =
        parent
        parent.child
        parent.child.libruntime
    ...
```

然后使用下面的配置进行加密:

```
$ pyarmor cfg package_name_format "libruntime"
$ pyarmor gen --recursive --prefix parent.child src/parent
```

在创建 wheel 之前不要忘记对 dist/parent/child/__init__.py 进行修改:

```
$ python -m build --skip-dependency-check --no-isolation
```

更进一步

为了能够在加密完成之后自动修改 dist/parent/child/__init__.py ，你可以创建一个加密插件 . pyarmor/myplugin.py:

```
__all__ = ['VersionPlugin']

class VersionPlugin:

    @staticmethod
    def post_build(ctx, inputs, outputs, pack):
        script = os.path.join(outputs[0], 'parent', 'child', '__init__.py')
        with open(script, 'a') as f:
            f.write("\nVERSION = '0.0.1'")
```

然后启用这个插件:

```
$ pyarmor cfg plugins + "myplugin"
```

这样以后每次打包只需要运行下面的命令就可以了:

```
$ pyarmor gen --recursive --prefix parent.child src/parent
$ python -m build --skip-dependency-check --no-isolation
```

3.2.5 打包脚本的时候保护系统库

在 8.2 版本加入.

在 8.2.2 版本发生变更: 不在支持使用选项*--restrict* 组合*--pack* 来保护系统库。

Pyarmor 对使用 PyInstaller 打包进行发布的方式提供了特别的保护，这种模式下面可以对系统库进行加密和保护。其实现的思路就是把所有的依赖包也列出了进行加密。

下面是一个示例，说明如何加密一个脚本 foo.py 并同时保护系统依赖库.

我们需要使用 PyInstaller 提供的功能来列出 foo.py 所有的依赖库。

首先生成 foo.spec:

```
$ pyi-makespec foo.py
```

然后修改这个 foo.spec:

```
a = Analysis(
    ...
)

# Patched by Pyarmor to generate file.list
_filelist = []
_package = None
for _src in sort([_src for _name, _src, _type in a.pure]):
    if _src.endswith('__init__.py'):
        _package = _src.replace('__init__.py', '')
        _filelist.append(_package)
    elif _package is None:
        _filelist.append(_src)
    elif not _src.startswith(_package):
        _package = None
        _filelist.append(_src)
with open('file.list', 'w') as _file:
    _file.write('\n'.join(_filelist))
# End of patch
```

接下来使用这个修改后的文件打包 foo.py，同时生成包含所有依赖库的文件 file.list:

```
$ pyinstaller foo.py
```

最后使用下面的选项加密脚本并重新打包:

```
$ pyarmor gen --assert-call --assert-import --pack dist/foo/foo foo.py @file.list
```

这个例子只是说明了基本的实现方法和步骤，请根据自己的实际情况编写自己的补丁脚本和使用必要的加密选项。如有必要，还可以人工修改生成的依赖库文件 file.list

3.2.6 解决加密过程中编码错误

默认的脚本编码为 utf-8，当加密脚本的时候出现编码错误，可以使用配置项指定正确的文件编码。例如下面的命令可以设置加密脚本使用的编码为 gbk:

```
$ pyarmor cfg encoding=gbk
```

同样也可以为定制的错误消息文件 messages.cfg 指定编码。例如，使用下面的命令可以设置定制的错误消息文件使用的编码为 gbk:

```
$ pyarmor cfg messages=messages.cfg:gbk
```

3.2.7 删除脚本中 Docstring

使用下面的配置可以删除加密脚本中 DocString:

```
$ pyarmor cfg optimize 2
```

配置项 `optimize` 可用值和作用请参考 compile

3.2.8 如何解决第三方库调用加密脚本存在的问题

内容

Pyarmor 提供了丰富的选项就是为了解决不同情况下遇到的问题，当使用一个复杂的第三方包调用加密脚本出现问题的时候，请首先花费一点时间看看命令手册，去了解所有的选项和作用，有一些选项就是专门针对不同的应用情况设计的。**Pyarmor 开发组不会去告诉应该使用那个选项就可以解决你遇到的问题**，而是你需要去学习和了解 Pyarmor，并找到适合自己需要的选项。

Python 在各个领域得到广泛的应用，有很多包我甚至从来都没有用过。对于我来说，不会去学习每一个包，然后确保其能够和 Pyarmor 兼容。通常的处理方式是用户报告相关的异常，根据异常的行号给出附近的源代码，我可以帮助分析哪些地方可能和 Pyarmor 发生冲突并给出相应的解决方案。

对于属于 Pyarmor 的问题，Pyarmor 开发组会尽快解决，但是有一些问题是 Pyarmor 自身无法解决的。

通常情况下，使用第三方包主要导致的问题来源有:

- 使用 `sys._getframe` 去访问函数的局部变量，或者其他运行框架的信息，但是加密脚本的运行框架和普通脚本是不一样的
- 使用 `inspect` 或者其他方式直接去访问函数的源代码（Byte code）等相关属性，它们所访问的属性正好就是 Pyarmor 所保护的
- 使用 `pickle` 或者类似功能序列号加密函数，然后把加密函数传递给其他进程或者线程执行，但是加密函数是无法使用普通方法序列化的

这些问题都来源于第三方包使用到了加密脚本修改过的底层对象，更多的不同之处请参考加密脚本和原来脚本的区别 。如果使用被修改的特性，就会出现不兼容的问题。特别是使用*BCC 模式* 进行加密的脚本，改变的内容更多。

对于这些问题，有一些常用的解决方案，可以首先尝试下面的方式

- 使用*RFT 模式* 和选项*--obf-code* 0

  RFT 模式 几乎不改变原来脚本的内部结构，所以一般不会导致不兼容性。使用*--obf-code* 禁用代码对象的加密也是为了保持内部对象的结构不改变。这是一种推荐的解决方案:

```
$ pyarmor gen --enable-rft --obf-code 0 /path/to/myapp
```

  首先使用最少的选项确保其能工作，然后在尝试更多的选项去增加安全性。例如:

```
$ pyarmor gen --enable-rft --obf-code 0 --mix-str /path/to/myapp
$ pyarmor gen --enable-rft --obf-code 0 --mix-str --assert-call /path/to/myapp
```

- 忽略有问题的模块

  在一个复杂的应用中，如果只是个别模块存在问题，可以不加密这些模块。例如，如果只有模块 `config.py` 不能正常工作，使用模块私有配置的方式把这个模块忽略掉:

```
$ pyarmor cfg -p myapp.config obf_code=0
$ pyarmor gen [other options] /path/to/myapp
```

  也可以按照原来的方法加密，只是加密之后使用原来的脚本直接把加密脚本替换:

```
$ pyarmor gen [other options] /path/to/myapp
$ cp /path/to/myapp/config.py dist/myapp/config.py
```

- 修改第三方库

  这是一个实例，如果使用别名"myapi" 去调用，抛出 404 错误，但是不使用别名工作正常。

```
@cherrypy.expose(alias='myapi')
    @cherrypy.tools.json_out()
    # pylint: disable=no-member
    @cherrypy.tools.authenticate()
    @cherrypy.tools.validateOptOut()
    @cherrypy.tools.validateHttpVerbs(allowedVerbs=['POST'])
    # pylint: enable=no-member
    def abc_xyz(self, arg1, arg2):
        """
        This is the doc string
        """
```

  导致问题出现的原因在于 `cherrypy.expose` 使用下面的语句

```
parents = sys._getframe(1).f_locals
```

因为 `sys._getframe(1)` 在加密脚本返回的不是期望的执行框架，把它修改成为下面的语句就可以在加密脚本正常使用:

```
parents = sys._getframe(2).f_locals
```

备注: 如果第三方包 `cheerypy` 也被其他程序使用，请为加密脚本创建一个私有的包

常用的第三方库

这里列出了一些常用到的第三方库以及可能的解决方案，欢迎大家分享自己的使用经验，提交 Pull request 增加新的第三方库。

表 1: 表-1. 常用第三方库列表

库	状态	备注
cherrypy	打补丁[1]	问题原因是使用了 sys._getframe
pandas	打补丁[1]	问题原因是使用了 sys._getframe
playwright	打补丁应该可以工作[2]	尚未验证
nuitka	使用 restrict_module = 0 之后应该可以工作	尚未验证

说明

pandas

另外一个实例是 `pandas`

```
import pandas as pd

class Sample:
    def __init__(self):
        self.df = pd.DataFrame(
            data={'name': ['Alice', 'Bob', 'Dave'],
            'age': [11, 15, 8],
            'point': [0.9, 0.1, 0.4]}
        )

    def func(self, val: float = 0.5) -> None:
        print(self.df.query('point > @val'))
```

(续下页)

[1] 通过打补丁的方式可以使用加密脚本

[2] 可以使用*RFT 模式* 加密的脚本

(接上页)

```
sampler = Sample()
sampler.func(0.3)
```

加密之后运行报错:

```
pandas.core.computation.ops.UndefinedVariableError: local variable 'val' is not␣
↪defined
```

同样需要对其打补丁，把 pandas scope.py 里面的 sys._getframe(self.level) 修改成为 sys._getframe(self.level+1) , sys._getframe(self.level+2) 或者 sys._getframe(self.level+3)

nuitka

把加密脚本作为普通脚本一样使用 Nuitka 进行处理，但是需要禁用约束模式:

```
$ pyarmor cfg restrict_module=0
```

如果不禁用约束模式，会导致校验错误 RuntimeError: unauthorized use of script

首先使用默认选项进行加密:

```
$ pyarmor gen foo.py
```

然后在尝试使用更多选项，但是约束相关的这些选项，例如*--private* , *--restrict* , *--assert-call* , *--assert-import* 可能无法使用。

3.2.9 注册和使用许可证

内容

前提条件

在注册许可证之前，需要满足下列条件

1. 一个许可证的激活文件，名称一般为 `pyarmor-regcode-xxxx.txt`，参考许可模式和许可证 购买合适的许可证
2. 安装 Pyarmor 8.2 以上的版本的设备，
3. 当前设备需要能够访问互联网
4. 确定许可证需要绑定的产品名称，如果是在非商业化产品中使用，产品名称使用 `non-profits`

使用基础版和专家版许可证

基本使用步骤:

1. 使用激活文件 进行初始登记，设定许可证绑定的产品名称
2. 初始登记完成之后会生成相应的注册文件，后续相关的注册都将使用这个文件
3. 在其他设备使用注册文件 进行注册

初始登记

在第一次注册的时候，需要使用 *-p* 指定许可证绑定的产品名称，如果在非商业化产品中的使用 Pyarmor，那么产品名称设定为 `non-profits`，该许可证不能被应用于商用产品中。

假设许可证绑定的产品名称为 `XXX`，那么使用下面的命令进行初始登记注册:

```
$ pyarmor reg -p "XXX" pyarmor-regcode-xxxx.txt
```

Pyarmor 会首先显示注册信息并请求确认，如果确认无误，输入 `yes` 并 `Enter` 继续下面的注册步骤，其他任何输入都会终止注册过程。

登记成功之后会同时生成一个相应的注册文件 `pyarmor-regfile-xxxx.zip`，这个文件被用来在其他设备上进行注册。

请不要再次使用激活文件 进行后续的注册，在初始登记之后这个文件失效，后续在任何设备上注册请使用注册文件 。

请妥善保管并备份注册文件，一旦丢失，Pyarmor 并不提供找回服务。

一旦登记成功，许可证绑定的产品名称就不能在进行修改。

绑定许可证到正在开发的产品

如果产品还在开发中，没有正式开始销售，允许初始化登记的时候设定产品的名称为 `TBD` 。例如:

```
$ pyarmor reg -p "TBD" pyarmor-regcode-xxxx.txt
```

绑定的产品名称为 `TBD` 的许可证，必须在六个月之内，使用下面的命令设置为正确的产品名称:

```
$ pyarmor reg -p "XXX" pyarmor-regcode-xxxx.txt
```

如果六个月之内还没有进行修改，那么产品名称会被自动设定为 `non-profits` ，并且不能在被修改。

在其他设备上面注册

拷贝注册文件 `pyarmor-regfile-xxxx.zip` 到其他需要注册的设备，然后运行下面的命令进行注册:

```
$ pyarmor reg pyarmor-regfile-xxxx.zip
```

查看注册信息:

```
$ pyarmor -v
```

注册成功之后所有的加密操作自动应用当前许可证，每一次加密操作需要联网验证许可证。

在 Docker 或者 CI pipeline 中注册

在 Docker 或者 CI pipeline 中注册 Pyarmor 的基本方法同上，但是同时运行的 Pyarmor 的 Docker 数量有限制，不能超过 100 个。

只允许在开发设备上安装和注册 Pyarmor。如果 Docker 镜像需要发送给客户，那么不允许在上面安装和注册 Pyarmor

使用集团版许可证

在 8.2 版本加入.

每一个集团版许可证称为一个组，一个组最多可以有 100 个离线设备，每一个设备有一个编号，依次从 1 到 100。

必须依次为离线设备进行编号，例如第一台设备编号为 1 ，第二台设备为 2 等等。

基本使用步骤:

1. 使用*激活文件* 进行初始登记，设定许可证绑定的产品名称，初始登记完成之后会生成相应的*注册文件*
2. 在离线设备上生成相应的组设备文件
3. 在联网设备上面使用*注册文件* 和组设备文件生成离线注册文件
4. 在离线设备上使用离线注册文件进行注册，离线注册文件只能在绑定的设备上使用

初始登记

购买集团版许可证之后，一个 `pyarmor-regcode-xxxx.txt` 的*激活文件* 会发送到注册邮箱中，这个文件用于第一次的注册登记。

集团版许可证的初始登记也需要在有网络的设备上进行，初始登记必须指定许可证绑定的产品名称，这里不允许使用 `TBD` 。假设产品名称是 `XXX` ，运行下面的命令进行初始登记:

```
$ pyarmor reg -p XXX pyarmor-regcode-xxxx.txt
```

登记成功之后会生成相应的*注册文件* `pyarmor-regfile-xxxx.zip` ，这个注册文件可用于后续的注册命令。

生成组设备文件

为了在离线设备上注册 Pyarmor，首先需要生成设备对应的组设备文件 `pyarmor-group-device.X` ，其中 `X` 是设备编号。

在每一台离线设备上安装 Pyarmor ，使用选项 *-g* 指定设备编号，生成相应的设备文件。

例如，在第一台离线设备上，运行下面的命令生成组设备文件 `pyarmor-group-device.1`:

```
$ pyarmor reg -g 1
```

生成离线设备注册文件

生成离线设备注册文件需要

- 安装 Pyarmor 8.2+ 并且可以访问互联网的设备
- 完成初始登记，并且已经生成注册文件 `pyarmor-regfile-xxxx.zip`
- 离线设备的组设备文件

把离线设备的组设备文件拷贝到进行初始登记的机器上（或者任何有互联网连接的设备），并存放在当前目录下面的子目录 `.pyarmor/group/`，然后使用下面的命令生成离线注册文件 `pyarmor-device-regfile-xxxx.1.zip`:

```
$ mkdir -p .pyarmor/group
$ cp pyarmor-group-file.1 .pyarmor/group/

$ pyarmor reg -g 1 /path/to/pyarmor-regfile-xxxx.zip
```

这条命令生成的离线注册文件只可以在第一台设备上使用。

离线设备注册

一旦生成离线设备注册文件之后，就可以拷贝到离线设备上面进行注册。例如，在第一台离线设备上，运行下面的命令:

```
$ pyarmor reg pyarmor-device-regfile-xxxx.1.zip
```

查看注册信息:

```
$ pyarmor -v
```

注册成功之后所有的加密操作自动应用集团版许可证，注册和加密都不需要联网验证。

支持多设备的离线注册文件

在 8.x 版本加入: 该功能尚未实现

如果需要为所有设备生成一个集成的离线注册文件，首先把所有组设备文件拷贝到指定目录，然后使用特殊组号 `0` 生成所有设备可用的离线注册文件 `pyarmor-device-regfile-xxxx.zip`:

```
$ cp pyarmor-group-file.1 pyarmor-group-file.2 pyarmor-group-file.3 .pyarmor/group/

$ pyarmor reg -g 0 /path/to/pyarmor-regfile-xxxx.zip
```

然后拷贝到每一台离线设备，使用 *-g* 指定设备编号进行注册:

```
$ pyarmor reg -g 1 pyarmor-device-regfile-xxxx.zip
```

升级老版本许可证

升级老版本的许可证请参考升级老版本许可证

3.3 技术手册

3.3.1 概念定义

BCC 模式

一种加密的方法，可以把脚本中的函数转换成为*C* 函数，然后使用优化选项编译生成机器指令。使用这种方式加密的函数是不可逆的，无法恢复成为原来的 Python 函数。

本地配置

一般在当前目录下会创建一个目录 `.pyarmor` ，用来存放本地配置，默认本地配置的文件是 `.pyarmor/config`

相同选项的本地配置会覆盖全局配置

参考*pyarmor cfg*

C

一种最古老的编程语言，现在依然生命力旺盛。

根目录

用来存放 Pyarmor 注册信息，全局配置文件等的路径，默认情况下是当前登陆用户的根目录下面的子目录 `~/.pyarmor`

使用 sudo 命令可能会改变 Pyarmor 的根目录

激活文件

一个文本文件，购买任何*Pyarmor 许可证* 之后，都会有一个相应的激活文件发送到注册邮箱。

激活文件主要用于第一次注册*Pyarmor 许可证* ，激活文件一旦完成初始登记，就无法在继续使用。

初始登记成功会同时生成相应的注册文件 ，后面在任何设备上进行注册都需要使用注册文件 。

加密插件

一个*Python* 脚本，在加密过程被调用，可以对输出的文件进行一些额外的操作。

参考加密插件

脚本补丁

一个 Python 脚本，可以被嵌入到加密脚本中，执行一些额外的检查。

考脚本补丁

JIT

JUST-IN-TIME 的缩写，是一种在运行时刻生成机器指令的技术，可以有效防止静态反编译工具对代码进行分析

开发机器

是指安装和运行 Pyarmor 的设备，在开发机器上对脚本进行加密

不是所有的平台都可以运行 Pyarmor，所有支持的运行环境请参考生成加密脚本的环境

客户设备

是指运行加密脚本的设备

客户设备上面不需要安装 Pyarmor

扩展模块

一个使用*C* 或者 C++ 语言编写的*Python* 模块

模块私有配置

每一个加密模块可以有自己的私有配置，一般存放在本地配置 的目录下面，和模块同名的 `module.ruler` 文件

相同选项的私有配置会覆盖本地配置

Pyarmor

Pyarmor 是一个用来加密 Python 脚本的工具。

Pyarmor 的组成部分

- *Pyarmor 项目*
- *pyarmor 包*

Pyarmor 包

一个*Python 包* ，它包含下列包

- *pyarmor*
- *pyarmor.cli*
- *pyarmor.cli.core*
- *pyarmor.cli.runtime*

Pyarmor 基础版

一种*Pyarmor 许可证* 类型

Pyarmor 集团版

一种*Pyarmor 许可证* 类型

Pyarmor 用户

使用*Pyarmor* 的组织机构或者开发人员

Pyarmor 项目

项目文件存放在 https://github.com/dashingsoft/pyarmor/

这里有 Pyarmor 开源部分的代码，提交的 问题报告 和最新的文档。

Pyarmor 许可证

由 Pyarmor 开发团队颁发，用于解锁 Pyarmor 试用版本中功能限制

请参考*Pyarmor 许可模式和许可证*

Pyarmor 专家版

一种*Pyarmor 许可证* 类型

Python

一种编程语言，官网地址 Python

Python 脚本

一个包含 Python 源代码的文件

外部概念 https://docs.python.org/3.11/glossary.html#term-module

Python 模块

要么是一个*Python 脚本* ，要么是一个*扩展模块*

Python 包

外部概念 https://docs.python.org/3.11/glossary.html#term-package

全局配置

存放 Pyarmor 运行配置的全局文件，全局配置文件必须存放在 Pyarmor 的*根目录* ，默认的文件名称是 `~/.pyarmor/config/global`

参考*pyarmor cfg*

RFT 模式

一种不可逆的加密模式，可以重命名 Python 脚本中的函数，类，方法，变量和参数

外部密钥

一个用于存储*运行密钥* 的文件，通常的名称为 `pyarmor.rkey`

外部密钥可以存放在下面的任何一个路径:

- *运行辅助包* 所在的路径
- 环境变量 *PYARMOR_RKEY* 指定的路径，路径中不能使用 `..` ，尾部不能是路径分隔符，一般用来指定一个绝对路径，例如 `/var/data`
- 当前路径

或者是文件：当前可执行文件的全路径名称 + `.pyarmor.rkey`

运行辅助包

一个*Python 包* ，名称一般为 `pyarmor_runtime_xxxxxx`

一般生成加密脚本的同时，也会生成相应的运行辅助包，运行加密脚本需要有相应的运行辅助包。

运行辅助文件

是指运行加密脚本需要的所有其他文件

通常情况下它等价于运行辅助包，如果使用了外部密钥，那么也包含外部密钥

运行密钥

保存加密脚本的运行设置和相关约束限制，包括脚本的有效期，脚本绑定的设备信息，也包括控制加密脚本行为的其他标志和设置。

通常情况下运行密钥被嵌入到运行辅助包里面，但是也可以是一个独立文件形式的外部密钥

运行平台

Pyarmor 定义的标准名称，用来标示运行 Pyarmor 和加密脚本的操作系统和 CPU 架构

下面列出了所有定义的运行平台:

- **Windows**
  - windows.x86_64
  - windows.x86
- **Many Linuxs**
  - linux.x86_64
  - linux.x86
  - linux.aarch64
  - linux.armv7
- **Apple Intel and Silicon**
  - darwin.x86_64
  - darwin.aarch64 or darwin.arm64
- **FreeBSD**
  - freebsd.x86_64
- **Alpine Linux (musl-c)**
  - alpine.x86_64
  - alpine.aarch64
- **Android**
  - android.x86_64
  - android.x86
  - android.aarch64
  - android.armv7

注册文件

一个 `.zip` 格式的压缩文件，主要用于除了初始登记之后的所有注册。

第一次初始登记使用激活文件，激活成功之后生成注册文件，之后在任何设备上都要使用注册文件进行注册。

3.3.2 命令手册

内容

Pyarmor 提供了丰富的选项来满足不同应用程序加密脚本的需求。

pyarmor

语法

pyarmor [options] <command> ...

选项

-h, --help	显示所有可用的子命令并退出
-v, --version	显示版本信息并退出
-q, --silent	在控制台不显示日志...
-d, --debug	打印更多的信息用于发现命令执行过程中问题...
--home PATH	设置根目录 ...

这些选项可以在命令 **pyarmor** 之后和下列子命令之前使用:

gen	生成加密脚本以及运行辅助文件
gen key	生成外部密钥文件
cfg	显示和配置加密环境
reg	激活和注册 Pyarmor

使用 **pyarmor <command> -h** 来查看每一个子命令的功能和所有的选项

描述

-q, --silent

不在控制台打印任何日志信息

例如:

```
pyarmor -q gen foo.py
```

-d, --debug

在控制台打印更多的信息用于发现命令执行过程中问题

加密脚本出现问题，或者想更多的了解加密过程，可以使用该选项打开调试模式。例如:

```
pyarmor -d gen foo.py
```

--home PATH[,GLOBAL[,LOCAL[,REG]]]

设置根目录，全局配置目录，本地配置目录以及注册信息所在的目录这个选项主要用于一台设备上需要使用多个不同的*Pyarmor* 许可证，可以为每一个许可证设定一个根目录，

它们的默认值分别是

- 根目录 是 ~/.pyarmor/
- 全局配置 目录是 ~/.pyarmor/config/
- 本地配置 目录是 ./.pyarmor/

默认存放注册文件的目录是根目录

这些目录配置都可以通过本选项进行设置。例如修改根目录为 ~/.pyarmor2/:

```
$ pyarmor --home ~/.pyarmor2 ...
```

这条命令同时修改

- 全局配置 目录是 ~/.pyarmor2/config/
- 注册文件所在目录是 ~/.pyarmor2/

下面的例子只修改全局配置 目录为 ~/.pyarmor/config2/:

```
$ pyarmor --home ,config2 ...
```

下面的命令只修改本地配置 目录为 /var/myproject

$ pyarmor --home „/var/myproject/ ...

当需要在一台设备上注册多个*Pyarmor* 许可证 的时候，可以为每一个许可证设置一个根目录 。例如:

```
$ pyarmor --home ~/.pyarmor1 reg pyarmor-regfile-2051.zip
$ pyarmor --home ~/.pyarmor2 reg pyarmor-regfile-2052.zip

$ pyarmor --home ~/.pyarmor1 gen project1/foo.py
$ pyarmor --home ~/.pyarmor2 gen project2/foo.py
```

在实际使用过程中，我们可能修改了很多配置，有时候出现问题之后，需要临时测试一下使用默认配置选项是否正常，但是又不想修改当前的配置。这时候可以通过设置全局配置目录和本地配置目录到一个不存在的路径来实现。例如:

```
$ pyarmor --home ,x,x, gen foo.py
```

参见:

PYARMOR_HOME

pyarmor gen

生成加密脚本和所有需要的运行辅助文件。

语法

pyarmor gen <options> <SCRIPT or PATH>

选项

-h, --help 显示选项列表并退出

-O PATH, --output PATH 设置输出目录...

-r, --recursive 递归搜索目录中的脚本...

-e DATE, --expired DATE 设置脚本有效期...

-b DEV, --bind-device DEV 绑定脚本到设备...

--bind-data DATA 存储自定义数据到运行密钥...

--period N 周期性检查运行密钥...

--outer 启用外部密钥文件...

--platform NAME 指定脚本运行的目标平台名称...

-i 保存运行辅助文件到加密包的内部...

--prefix PREFIX 设置导入运行辅助包的前缀名称...

--obf-module <0,1> 指定模块加密模式，默认是 1 ...

--obf-code <0,1,2> 指定代码加密模式，默认是 1 ...

--no-wrap 禁用包裹加密模式...

--enable <jit,rft,bcc,themida> 启用不同的保护特征...

--mix-str 混淆字符串常量...

--private 启用私有模块加密脚本...

--restrict 启用约束模式加密包...

--assert-import 确保导入的脚本是经过加密的...

--assert-call 确保调用的函数是经过加密的...

--pack BUNDLE 使用加密后的脚本替换打包成为可执行文件里面的原来脚本...

描述

该命令用来加密脚本和包，例如:

```
pyarmor gen foo.py
pyarmor gen foo.py goo.py koo.py
pyarmor gen src/mypkg
pyarmor gen src/pkg1 src/pkg2 libs/dbpkg
pyarmor gen -r src/mypkg
pyarmor gen -r main.py src/*.py libs/utils.py libs/dbpkg
```

所有在命令行列出的文件都会被作为 Python 脚本进行加密，即便它的扩展名不是 `.py` 。

所有命令行列出的目录都会被作为包来进行加密，在该目录下面的所有 `.py` 文件会被加密，如果包含子目录需要加密，使用选项 *-r* 来递归处理所有子目录。

不要使用 `pyarmor gen src/*` 来加密一个包，这样会加密目录下面的任何文件。

从 8.2.2 开始，支持使用文件列出所有需要加密的脚本和包, 然后使用前缀 `@` 来引用选项文件，例如:

```
pyarmor gen -r @filelist
```

文件 `filelist` 列出了需要加密的两个脚本和两个包:

```
src/foo.py
src/utils.py
libs/dbpkg
libs/config
```

-O PATH, **--output** PATH

设置加密脚本的输出路径，默认值是 `dist`

-r, --recursive

递归搜索目录下面的 Python 脚本，否则只搜索当前目录下面的脚本

-i

保存运行辅助文件到加密包的内部。例如，下面的命令把运行辅助包 存放到 dist/mypkg 目录下面:

```
$ pyarmor gen -r -i mypkg

$ ls dist/
...      mypkg/

$ ls dist/mypkg/
...            pyarmor_runtime_000000/
```

没有这个选项，输出的目录结构是:

```
$ ls dist/
...      mypkg/
...      pyarmor_runtime_000000/
```

这个选项同时会使用相对导入的方式导入运行辅助包，所以只能在加密包中使用，如果是单独加密脚本，不要使用这个选项。

--prefix PREFIX

设置导入运行辅助包的前缀

主要应用于同时加密多个包，但是运行辅助包 又被存放在一个包里面，使用该选项告诉其他包如何正确导入运行辅助包。例如:

```
$ pyarmor gen --prefix mypkg src/mypkg mypkg1 mypkg2
```

这个命令用于同时加密三个包，但是运行辅助包会存放在 dist/mypkg 里面，那么在 dist/mypkg/__init.py 里面，使用的是下面方式导入运行辅助包:

```
from .pyarmor_runtime_000000 import __pyarmor__
__pyarmor__(__name__, __file__, b'...')
```

而在 dist/mypkg1/__init.py 里面，使用的是下面方式导入运行辅助包:

```
from mypkg.pyarmor_runtime_000000 import __pyarmor__
__pyarmor__(__name__, __file__, b'...')
```

-e DATE, **--expired** DATE

设置加密脚本的有效期

支持的格式:

- 数字，表示从现在开始的天数
- YYYY-MM-DD，直接指定有效期

如果前面有字符”.”，那么表示使用本地时间判断，否则使用网络时间

例如:

```
pyarmor gen -e 30 foo.py
pyarmor gen -e 2022-12-31 foo.py
```

判断是否过期会读取 NTP 服务器的时间，所以在没有联网的机器上无法运行。

而下面的格式则使用本地时间验证有效期，例如:

```
pyarmor gen -e .30 foo.py
pyarmor gen -e .2022-12-31 foo.py
```

-b DEV, **--bind-device** DEV

绑定加密脚本到指定的机器，目前支持的硬件信息包括硬盘序列号，网卡 Mac 地址和 IPv4 地址。例如:

```
pyarmor gen -b 128.16.4.10 foo.py
pyarmor gen -b 52:38:6a:f2:c2:ff foo.py
pyarmor gen -b HXS2000CN2A foo.py
```

也可以和有效期组合使用:

```
pyarmor gen -e 30 -b 128.16.4.10 foo.py
```

如果需要同时指定一个设备的多个硬件信息，那么使用空格分开各项，例如:

```
pyarmor gen -b "128.16.4.10 52:38:6a:f2:c2:ff HXS2000CN2A" foo.py
```

这个选项可以使用多次，以绑定加密脚本到不同的机器，例如:

```
pyarmor gen -b 52:38:6a:f2:c2:ff -b 66:77:88:9a:cc:fa -b "f8:ff:c2:27:00:7f" foo.py
```

如果需要绑定其他网卡，那么使用尖括号把网址包含起来，例如:

```
pyarmor gen -b "<2a:33:50:46:8f>" foo.py
```

也可以使用下面的格式绑定一台机器上全部或者部分网卡，例如:

```
pyarmor gen -b "<2a:33:50:46:8f,f0:28:69:c0:24:3a>" foo.py
```

在 Linux 系统下，还可以指定的网络接口的名称，例如:

```
pyarmor gen -b "eth1/fa:33:50:46:8f:3d" foo.py
```

如果需要绑定其他硬盘，那么需要指定硬盘的名称，例如:

```
# 适用于 Windows，分别绑定第一个硬盘和第二个硬盘
pyarmor gen -b "/0:FV994730S6LLF07AY" foo.py
pyarmor gen -b "/1:KDX3298FS6P5AX380" foo.py

# 适用于 Linux，绑定到硬盘设备名称 "/dev/vda2"
pyarmor gen -b "/dev/vda2:KDX3298FS6P5AX380" foo.py
```

--bind-data DATA

DATA 可以是字符串或者 @FILENAME

这个选项可以存储任何数据到运行密钥 中，但是有长度限制，一般不超过 4096 个字节。主要用于用户扩展验证运行密钥的方式，在加密脚本中读取运行密钥中存放的数据，使用自己的算法进行校验和检查。

如果传入的参数以 @ 开头，那么读取后面的文件内容，否则直接把参数的内容存放到运行密钥中。

不管哪一种情况，存放到运行密钥中都是 Bytes 类型。

--period N

周期性的检查运行设置文件，单位为小时。默认情况下是导入加密模块的时候会检查运行设置信息，对于一些服务器应用，这个选项可以人工设置检查周期。

支持的格式为数字 + 单位，单位支持时分秒，没有单位则默认为小时：

- 1
- 1s
- 1m
- 1h

下面的例子是等价的，都是每隔一个小时重新检查一下:

```
$ pyarmor gen --period 1 foo.py
$ pyarmor gen --period 3600s foo.py
$ pyarmor gen --period 60m foo.py
$ pyarmor gen --period 1h foo.py
```

备注： 需要注意的是即便是设置了这个值，也必须是在运行加密函数的时候才进行检查。也就是说，如果一个无限循环没有调用任何加密函数，那么不会触发周期性检查事件。

--outer

启用外部密钥

加密脚本如果需要使用外部密钥，必须在加密的时候指定这个选项，否则外部密钥不起作用。

而一旦指定使用外部密钥，就必须使用*pyarmor gen key* 生成外部密钥文件，并拷贝到指定目录，否则加密脚本无法运行。

外部密钥文件的名称默认为 `pyarmor.rkey` ，可以通过配置文件修改默认值，例如，使用前面有点的 `.pyarmor.key`:

```
$ pyarmor cfg outer_keyname=".pyarmor.key"
```

这样外部密钥文件使用 **ls** 就无法看到。

需要注意的是一旦修改密钥文件之后，必须重新生成运行辅助包，原来的运行辅助包无法在使用，因为运行辅助包只会查找生成的时候指定的密钥文件名称。

--platform NAME

用于跨平台加密脚本指定运行加密脚本的目标平台

这个选项可以使用多次，也可以使用逗号把多个平台名称分开

平台名称必须是 Pyarmor 定义的*运行平台* 名称

不是所有的平台都可以组合在一起发布的

跨平台加密需要安装包*pyarmor.cli.runtime* ，只有包里面支持的平台才能使用

--private

启用私有模式来保护加密脚本

启用私有模式之后，函数的名称在执行框架中会被隐藏，并且加密后的脚本也不能被其他脚本导入。

选项*--restrict* 隐含启用私有模式。

--restrict

主要应用于保护加密包，保护包里面的模块，只能在包内部使用，不能被外部模块调用

这个选项隐含启用*--private*

当约束模式启用之后，除了 `__init__.py` 输出的名称之外，其他模块都不能被外部脚本导入和使用。

例如，使用下面的命令加密一个约束包 `dist/joker`:

```
$ pyarmor gen -i --restrict joker
$ ls dist/
...    joker/
```

然后在创建一个没有加密的脚本 `foo.py` 去导入加密包:

```
import joker
print('import joker should be OK')
from joker import queens
print('import joker.queens should fail')
```

运行结果如下:

```
$ cd dist
$ python foo.py
... import joker should be OK
... RuntimeError: unauthorized use of script
```

约束模式一般只用于加密包（目录），如果使用该选项单独加密脚本（文件）之后，脚本无法被直接运行。

--obf-module <0,1>

加密模块对应 .pyc，默认是 1

--obf-code <0,1,2>

加密模块中的每一个函数（二次加密），默认是 1

模式 2 是 Pyarmor 8.2 中新增加的，它能增加从 Bytecode 进行反编译的难度，同时会降低一点性能。它可以把部分表达式中的属性名称混淆，例如:

```
obj.attr          ==> getattr(obj, 'xxxx')
obj.attr = value  ==> setattr(obj, 'xxxx', value)
```

通常情况下，如果 RFT 模块可用的话，不需要使用这个选项。

--no-wrap

不使用包裹模式加密模块中函数

使用包裹模式加密函数是指在调用函数的时候解密函数，调用完成重新加密函数。禁用之后第一次调用函数的解密函数，但是调用完成之后不在加密，以后调用的时候也无需重新解密。禁用包裹模式，对于多次执行的函数可以提高性能。

如果*--obf-code* 是 0，那么使用选项没有任何作用。

--enable <jit,rft,bcc,themida>

启用不同的加密模式

--enable-jit

使用*JIT* 来处理一些敏感数据以增强安全性

--enable-rft

启用*RFT 模式* pro

--enable-bcc

启用*BCC 模式* pro

--enable-themida

使用 Themida 来保护加密脚本，仅 Windows 平台可用

--mix-str

混淆脚本中字符串常量 basic

--assert-call

启用自动检查函数功能，确保加密函数没有被替换

--assert-import

启用自动检查模块功能，确保加密的模块没有被替换

--pack BUNDLE

使用加密脚本替换 BUNDLE 里面的 Python 脚本

参数中 BUNDLE 是使用 PyInstaller 生成的可执行文件。

Pyarmor 首先加密脚本，接着使用加密后的脚本替换 BUNDLE 里面的同名 Python 脚本，最后使用修改后的可执行文件覆盖原文件。

pyarmor gen key

生成外部密钥 文件

语法

pyarmor gen key <options>

选项

-O PATH, --output PATH 输出路径

-e DATE, --expired DATE 设置有效期

--period N 定时检查运行密钥

-b DEV, --bind-device DEV 绑定加密脚本到指定设备

--bind-data DATA 存储自定义数据到运行密钥

描述

这个命令用来生成外部密钥 文件，它使用到这些选项基本是命令*pyarmor gen* 的子集，其作用和含义也一样，请参考上面的说明。

外部密钥必须至少包含 -e 或者 -b 一个选项，没有任何约束和限制的外部密钥存放安全风险，如果你确定需要这样的外部密钥，可以指定一万年的有效期，并且使用本地时间。

通常情况下外部密钥文件保存在 dist/pyarmor.rkey 。例如:

```
$ pyarmor gen key -e 30
$ ls dist/pyarmor.rkey
```

使用下面的命令保存外部密钥到其他路径:

```
$ pyarmor gen key -O dist/mykey2 -e 10
$ ls dist/mykey2/pyarmor.rkey
```

外部密钥的默认文件名称是 pyarmor.rkey，它不能在命令行进行修改。但是可以通过配置文件进行修改。例如，下面的命令把外部密钥文件名称修改为 sky.lic:

```
$ pyarmor cfg outer_keyname=sky.lic
$ pyarmor gen key -e 30
$ ls dist/sky.lic
```

外部密钥文件也可以存放在其他位置，在运行的时候会依次查找下面的路径:

- 运行辅助包的目录
- 环境变量 *PYARMOR_RKEY* 指定的路径，路径名称中不能包含 . .，尾部不能是路径分隔符，一般用来指定一个绝对路径，例如 /var/data
- 当前目录

如果在以上目录下找不到 pyarmor.rkey，会查看是否存在文件

- 当前可执行文件的全路径名称 + .pyarmor.rkey，例如 dist/foo/foo.exe.pyarmor.rkey

如果不存在，那么报错缺失运行密钥。

pyarmor cfg

显示和配置 Pyarmor 加密环境

语法

pyarmor cfg <options> [OPT[=VALUE]] ...

选项

-h, --help	显示选项和帮助信息然后退出
-p NAME	指定私有设置的模块名称
-g, --global	所有操作都是在全局配置 中进行
-r, --reset	恢复配置项的默认值
--encoding ENCODING	指定打开配置文件的编码

描述

查看所有的可用配置项:

```
$ pyarmor cfg
```

只查看一个选项 obf_module:

```
$ pyarmor cfg obf_module
```

查看所有以 obf 开头的选项:

```
$ pyarmor cfg obf*
```

设置一个整数型配置项的值，可以使用下面的任意一种方式:

```
$ pyarmor cfg obf_module 0
$ pyarmor cfg obf_module=0
$ pyarmor cfg obf_module =0
$ pyarmor cfg obf_module = 0
```

设置布尔型配置项的值:

```
$ pyarmor cfg wrap_mode 0
$ pyarmor cfg wrap_mode=1
```

设置字符串配置项的值:

```
$ pyarmor cfg outer_keyname "sky.lic"
$ pyarmor cfg outer_keyname = "sky.lic"
```

增加一个词到列表型配置项目使用运算符 + 。例如:

```
$ pyarmor cfg pyexts + ".pym"

Current settings
    pyexts = .py .pyw .pym
```

删除一个词使用 - 。例如:

```
$ pyarmor cfg pyexts - ".pym"

Current settings
    pyexts = .py .pyw
```

增加一行到配置项使用 ^ ，例如:

```
$ pyarmor cfg rft_excludes ^ "/win.*/"

Current settings
    rft_excludes = super
        /win.*/
```

恢复配置项的默认值，可以使用下面的任意一种格式:

```
$ pyarmor cfg rft_excludes ""
$ pyarmor cfg rft_excludes=""
$ pyarmor cfg -r rft_excludes
```

指定选项所在的组中使用前缀 `group:` 。例如修改组 `finder` 中的配置项 `excludes`:

```
$ pyarmor cfg finder:excludes "ast"
```

如果没有组前缀 `finder` ，那么其他组中的配置项 `excludes` 也会被修改。

配置组

组是多个配置项的集合，常用的组有

- finder: 如何搜索脚本
- builder: 如果加密脚本，大部分的选项是在这里
- runtime: 如何生成运行辅助包和运行密钥

另外还有几个不常用的组

- mix.str: 如何选择需要加密的字符串
- assert.call: 如何选择需要保护的函数
- assert.import: 如何选择需要保护的模块
- bcc: 如何转换 Python 函数为*C* 函数

-p NAME

指定模块私有配置 的模块名称

使用这个选项来处理特殊的模块，这些特殊模块需要使用不同的加密配置

所有的配置操作都是在该模块中，不影响其他模块。

对于包里面的模块，需要使用全路径名称来指定，例如 `pkgname.modname.submodule`

例如，这两个模块 `joker/__init__.py` 和 `joker/card.py` 不使用任何约束，包里面的其他模块都使用约束模式:

```
$ pyarmor cfg -p joker.__init__ restrict_module = 0
$ pyarmor cfg -p joker.card restrict_module = 0
$ pyarmor gen -r --restrict joker
```

-g, --global

所有操作都是在全局配置 中进行

没有指定这个选项的话，所有的操作都是在本地配置 中，通常就是 `./.pyarmor/` 。指定了这个选项，所有的操作都在全局配置 中，通常就是 `~/.pyarmor/config/`

-r, --reset

恢复配置项的默认值

pyarmor reg

激活，注册和升级 Pyarmor

语法

pyarmor reg [OPTIONS] [FILENAME]

选项

-h, --help 显示可用选项和帮助信息然后退出

-p NAME, --product NAME 指定许可证绑定的产品名称

-u, --upgrade 升级老版本的 Pyarmor 许可证

-g ID, --device ID 指定组内设备编号，仅用于集团版许可证注册

参数

参数 `FILENAME` 必须是下面任意一种文件

- `pyarmor-regcode-xxxx.txt` 激活文件，一般在购买许可证之后会发送到注册邮箱
- `pyarmor-regfile-xxxx.zip` 注册文件，初始登记之后自动生成

描述

检查当前设备的注册信息:

```
$ pyarmor -v
```

初始登记

初始登记使用下面的命令，产品名称必须输入，使用实际产品名称替换 `NAME` ，用于非商业化产品的使用名称 `non-profits`:

```
$ pyarmor reg -p NAME pyarmor-regcode-xxxx.txt
```

初始登记完成之后会生成相应的注册文件 `pyarmor-regfile-xxxx.zip` ，在其他设备以及后续的注册均使用这个文件，并且不需要输入产品名称:

```
$ pyarmor reg pyarmor-regfile-xxxx.zip
```

升级许可证

升级使用下面的命令，其中产品名称必须和原来的许可证绑定名称一致，如果不一致的话会被忽略:

```
$ pyarmor reg -p NAME pyarmor-regcode-xxxx.txt
```

升级完成之后会生成相应的注册文件 `pyarmor-regfile-xxxx.zip` ，在其他设备以及后续的注册均使用这个文件，并且不需要输入产品名称:

```
$ pyarmor reg pyarmor-regfile-xxxx.zip
```

集团版许可证

集团版许可证也需要在有网络的机器上进行初始登记，并生成相应的注册文件 `pyarmor-regfile-xxxx.zip`

一个集团版许可证最多使用 100 个离线设备，每一个设备都有一个编号，从 1 到 100。

为了在离线设备上注册 Pyarmor，需要为每一台设备分别生成相应的离线注册文件。

例如，为第一台设备生成离线注册文件，首先在第一台设备运行下面的命令，生成一个组内的设备文件 `pyarmor-group-device.1`:

```
$ pyarmor reg -g 1
```

然后把这个文件拷贝到进行初始登记的机器上面，并存放在指定目录 `.pyarmor/group/` 下面，再使用下面的命令生成离线注册文件 `pyarmor-device-regfile-xxxx.1.zip`:

```
$ mkdir -p .pyarmor/group
$ cp pyarmor-group-device.1 .pyarmor/group/

$ pyarmor reg -g 1 pyarmor-regfile-xxxx.zip
```

拷贝这个离线注册文件到第一台设备，运行下面的命令进行离线注册:

```
$ pyarmor reg pyarmor-device-regfile-xxxx.1.zip
```

对于第二台，第三台等设备，也需要进行相应的操作，设备的序号必须依次增加。

-p NAME, **--product** NAME

使用 `.txt` 文件注册时候指定许可证绑定的产品名称

使用 `.zip` 文件注册的时候不需要使用这个选项

非商业化产品中使用请设置名称为 `non-profits`

第一次注册或者升级的时候指定许可证绑定的产品名称

绑定的产品名称一旦设定之后就无法更改，除了一个特殊名称 `TBD`

如果产品名称设置为 `TBD`，那么在六个月之内可以在修改一次。这个主要用于产品还在开发阶段，尚未确定名称的时候使用，在产品正式销售之前需要修改为正确的产品名称。如果六个月之内还没有进行修改，那么产品名称会被自动设定为 `non-profits`，并且不能在被修改。

只有基础版和专家版许可证的产品名称可以被设定为 `TBD`，升级许可证，以及集团版许可证的必须指定实际使用的产品名称。

-u, --upgrade

升级老版本的许可证为 Pyarmor 8 的许可证

不是所有的老版本的许可证都可以升级到新版本，请参考许可模式和许可证 里面的升级说明

-g ID, **--group** ID

指定离线设备的组内编号，仅适用于集团版许可证

有效值为 1 到 100

环境变量

下列环境变量是在开发机器 上加密脚本的时候会使用到

PYARMOR_HOME

设置方法和选项*pyarmor --home* 相同

主要用来设置根目录，如果使用了选项*pyarmor --home*，这个环境变量会被忽略

PYARMOR_PLATFORM

设置 Pyarmor 的运行平台 名称

主要用来支持一些能够运行 Pyarmor 但是无法正确得到平台名称的系统

PYARMOR_CC

设置*BCC 模式* 使用的*C* 编译器

PYARMOR_CLI

仅当需要和 Pyarmor 7.x 兼容的时候使用

如果安装了 Pyarmor 8，但是又需要保持和 Pyarmor 7.x 的兼容性，例如不修改原来的构建环境的脚本等，那么需要输出设置环境变量为 7 。

例如，在 Linux 或者 Apple 下面:

```
export PYARMOR_CLI=7
pyarmor -h

PYARMOR_CLI=7 pyarmor -h
```

在 Windows 下面，使用下面的命令:

```
set PYARMOR_CLI=7
pyarmor -h
```

另外一种选择是直接使用命令 *pyarmor-7* 来和 Pyarmor 7.x 保持兼容。

3.3.3 生成加密脚本的环境

这里列出了和命令 **pyarmor** 相关的所有一切。

首先命令 **pyarmor** 运行在开发机器 上面，使用*supported Python versions* 运行在这些*supported platforms*

命令行选项，配置文件选项 ，加密插件 ，脚本补丁 和一些环境变量影响和改变着命令 **pyarmor** 的行为。

所有的命令行选项和相关的环境变量在命令手册 中有详细描述。

支持的 Python 版本

表 2: 表-1. 不同功能支持的 Python 版本表

Python	2.7	3.0~3.4	3.5~3.6	3.7~3.10	3.11	3.12+	备注
RFT 模式	No	No	No	Y	Y	N/y	[1]
BCC 模式	No	No	No	Y	Y	N/y	
pyarmor 8 基础功能	No	No	No	Y	Y	N/y	
pyarmor-7	Y	Y	Y	Y	No	No	

支持的平台

表 3: 表-2. 不同功能支持的平台列表（一）

OS	Windows	Apple		Linux[2]			
Arch	x86/x86_64	x86_64	arm64	x86/x86_64	aarch64	armv7	armv6
Themida 保护	Y	No	No	No	No	No	No
RFT 模式	Y	Y	Y	Y	Y	Y	No
BCC 模式	Y	Y	Y	Y	Y	N/y	No
pyarmor 8 基础功能	Y	Y	Y	Y	Y	Y	No
pyarmor-7[3]	Y	Y	Y	Y	Y	Y	Y

表 4: 表-3. 不同功能支持的平台列表（二）[Page 80, 4]

OS	FreeBSD	Alpine Linux		Android			
Arch	x86_64	x86_64	aarch64	x86/x86_64	aarch64	armv7	armv6
RFT 模式	Y	Y	Y	Y	Y	Y	No
BCC 模式	Y	Y	Y	Y	Y	Y	No
pyarmor 8 基础功能	Y	Y	Y	Y	Y	Y	No
pyarmor-7	Y	Y	Y	Y	Y	Y	Y

[1] N/y 意味着现在还不支持，但是将来会支持

[2] 这里的 Linux 使用的是 glibc

[3] pyarmor-7 支持更多的平台，参考 Pyarmor 7.x 文档.

注释

重要： pyarmor-7 是 Pyarmor 7.x 的问题修正版本，只支持老版本的许可证。在新版本的许可证下面使用可能会报错 `HTTP 401 error`

配置文件选项

有三种类型的配置文件

- 全局配置 文件，一个 `.ini` 格式的文件 `~/.pyarmor/config/global`
- 本地配置 文件，一个 `.ini` 格式的文件 `.pyarmor/config`
- 模块私有配置 每一个模块可以有自己的私有配置，存放在本地配置 的目录下面

使用命令*pyarmor cfg* 来查看和设置配置文件选项。

加密插件

在 8.2 版本加入.

加密插件是一个 Python 脚本，在生成输出文件之后被调用，可以对输出文件进行一些修改和调整，也可以做任何需要的事情。

加密插件常用的场景:

- 对输出目录的文件进行额外处理
- 修改加密脚本中导入运行辅助包的语句，以满足特殊的导入需求
- 增加注释信息到外部密钥 文件，这样通过注释使用外部脚本就可以读取运行配置信息
- 修改运行辅助包中扩展模块 `pyarmor_runtime` 的后缀，以避免名称冲突
- 使用 *install_name_tool* 修改扩展模块 `pyarmor_runtime` 的依赖包，解决有些 Darwin 环境无法装载扩展模块的问题

一个加密插件的脚本可以定义一个或者多个插件类，同时必须定义属性 `__all__` 来输出脚本中定义的插件类，没有输出的插件类不起作用

插件类的定义如下

class PluginName

static post_build(*ctx*, *inputs*, *outputs*, *pack=None*)

如果该方法存在，那么当所有加密文件都生成之后会被*pyarmor gen* 调用

参数

[4] 下表中平台是在 Pyarmor 8.3 新增加的

- **ctx** (*Context*) -- 加密环境
- **inputs** (*list*) -- 所有命令行输入的脚本和包名称
- **outputs** (*list*) -- 输出目录
- **pack** (*str*) -- 要么是 None，要么是选项*--pack* 指定的文件名称

static post_key(*ctx*, *keyfile*, ***keyinfo*)

如果该方法存在，那么当外部密钥 文件生成之后会被*pyarmor gen key* 调用

参数

- **ctx** (*Context*) -- 加密环境
- **keyfile** (*str*) -- 输出的外部密钥文件
- **keyinfo** (*dict*) -- 外部密钥绑定的信息

参数 `keyinfo` 中可能的项目有

Key expired
None 或者有效期（epoch）

Key devices
None 或者设备信息列表

Key data
None 或者是绑定的私有数据（Bytes）

Key period
None 或者是定时检查运行密钥的周期（秒）

static post_runtime(*ctx*, *source*, *dest*, *platform*)

如果该方法存在，那么当每一个运行平台的扩展模块 `pyarmor_runtime` 被生活之后会被*pyarmor gen* 调用，如果是生成多平台的脚本，它可能被调用多次。

参数

- **ctx** (*Context*) -- 加密环境
- **source** (*str*) -- 预编译的扩展模块文件名称
- **dest** (*str*) -- 输出的扩展模块文件名称
- **platform** (*str*) -- 扩展模块对应的运行平台 名称

为了启用插件脚本，需要进行配置。配置的时候不需要输入扩展名 `.py` ，直接使用脚本名称。例如:

```
$ pyarmor cfg plugins + "script name"
```

Pyarmor 按照顺序依次搜索同名的脚本:

- 当前路径

- 本地配置 目录，通常是 .pyarmor/
- 全局配置 目录，通常就是 ~/.pyarmor/

这里有一个示例插件脚本 fooplugin.py

```
__all__ = ['EchoPlugin']

class EchoPlugin:

    @staticmethod
    def post_runtime(ctx, source, dest, platform):
        print('-------- test fooplugin ----------')
        print('ctx is', ctx)
        print('source is', source)
        print('dest is', dest)
        print('platform is', platform)
```

把它保存到 .pyarmor/fooplugin.py 并启用它:

```
$ pyarmor cfg plugins + "fooplugin"
```

加密一个脚本，在控制台可以看到插件运行的输出信息:

```
$ pyarmor gen foo.py
```

禁用插件使用下面的方式:

```
$ pyarmor cfg plugins - "fooplugin"
```

脚本补丁

在 8.2 版本加入.

脚本补丁也是一个 Python 脚本，在加密的时候被嵌入到脚本的最前面，相当于直接在脚本的前面插入了插件源代码，这样加密脚本运行的时候就会首先执行插件代码。

当加密脚本的时候，Pyarmor 会在本地配置 和全局配置 目录下面查看有没有路径 hooks ，如果有的话，那么会查看有没有和加密脚本同名的文件，有的话就会被插入到加密脚本中。

例如，.pyarmor/hooks/foo.py 就是 foo.py 的补丁，而 .pyarmor/hooks/joker.card.py 是 joker/card.py 的补丁。

脚本补丁就是一个普通的 Python 脚本，但是它可以用两个特殊的内置函数*__pyarmor__()* and *__assert_armored__()* 来做一些加密脚本特有的事情。

需要注意的是补丁是直接插入到脚本的模块级别的代码中，所以要避免名称冲突，影响了原来脚本的执行。

参见:

__pyarmor__() __assert_armored__()

特殊脚本补丁

在 8.3 版本加入.

一般的脚本补丁是嵌入到了加密脚本中，如果需要在运行加密脚本之前就进行一些定制或者额外的检查，那么就需要使用到特殊的脚本补丁 .pyarmor/hooks/pyarmor_runtime.py，这个脚本补丁可以定义在加密脚本执行之前就被调用的函数。

首先创建脚本 .pyarmor/hooks/pyarmor_runtime.py，然后定义一个函数*bootstrap()*，这个函数在扩展模块 *pyarmor_runtime* 初始化的过程中被调用，其他代码都会被忽略。

bootstrap(*user_data*)

参数

user_data (*bytes*) -- 运行密钥里面的用户自定义数据

返回

如果返回 False，那么扩展模块 pyarmor_runtime 初始化失败，并且抛出保护异常

抛出

- **SystemExit** -- 直接退出，不显示调用堆栈
- **ohter Exception** -- 退出并且显示调用堆栈

An example script:

```
def bootstrap(user_data):
    # 必须在函数内容导入需要的名称，不要在模块级别导入
    import sys
    import time
    from struct import calcsize

    print('user data is', user_data)

    # 检查平台，不支持 32 位
    if sys.platform == 'win32' and calcsize('P'.encode()) * 8 == 32:
        raise SystemExit('no support for 32-bit windows')

    # 在 Windows 平台下面检查是否有调试器存在
    if sys.platform == 'win32':
        from ctypes import windll
        if windll.kernel32.IsDebuggerPresent():
            print('found debugger')
```

(续下页)

(接上页)

```
            return False

    # 在这个例子中，传入的自定义数据是时间戳
    if time.time() > int(user_data.decode()):
        return False
```

验证一下这个脚本，首先拷贝这个脚本到 .pyarmor/hooks/pyarmor_runtime.py，然后执行下面的命令:

```
$ pyarmor gen --bind-data 12345 foo.py
$ python dist/foo.py

user data is b'12345'
Traceback (most recent call last):
  File "dist/foo.py", line 2, in <module>
  ...
RuntimeError: unauthorized use of script (1:10325)
```

3.3.4 运行加密脚本的环境

加密脚本运行在客户设备 上面

支持的 Python 版本和平台

运行加密脚本支持的平台和 Python 版本和生成加密脚本的环境 是一样的。

环境变量

这里的环境变量会被加密脚本使用来决定一些运行设置

LANG

操作系统环境变量，加密脚本读取它来决定运行时刻的语言设置

PYARMOR_LANG

用来设置运行时刻使用的语言设置。

如果这个变量存在，那么 *LANG* 会被忽略

PYARMOR_RKEY

设置搜索外部密钥 的路径

第三方解释器的支持

对于第三方的解释器（例如 Jython 等）以及通过嵌入 Python C/C++ 代码调用加密脚本，只要第三方解释器能够和 CPython 扩展模块 兼容，就可以使用加密脚本。查看第三方解释器的文档，确认它是否支持 CPython 的扩展模块，

已知的一些问题

- *PyPy* 无法运行加密脚本，因为它完全不同于 *CPython* 。
- 在 Linux 下面装载 Python 动态库 *libpythonXY.so* 的时候 *dlopen* 必须设置 *RTLD_GLOBAL* ，否则加密脚本无法运行。
- Boost::python，默认装载 Python 动态库是没有设置 *RTLD_GLOAL* 的，运行加密脚本的时候会报错"No PyCode_Type found" 。解决方法就是在初始化的调用方法 *sys.setdlopenflags(os.RTLD_GLOBAL)* ，这样就可以共享动态库输出的函数和变量。
- 模块 *ctypes* 必须存在并且 *ctypes.pythonapi._handle* 必须被设置为 Python 动态库的句柄，PyArmor 会通过该句柄获取 Python C API 的地址。
- WASM 目前不支持，因为这需要把运行库的代码也编译成为 WASM，但是 WASM 是很容易就被反编译成为原来的 C 代码，为了安全性，所以目前没有支持 WASM 的计划。如果有更多的用户提出这个需求，会考虑实现一个轻量级的运行库，只支持能够运行 RFT 模式的加密脚本，但是目前还没有开发计划。

加密脚本内置函数

在 8.2 版本加入.

有两个特殊的函数无需导入，可以直接在加密脚本中使用，通常它们一般用在脚本补丁 中。

__pyarmor__(*arg*, *kwarg*, *name*, *flag*)

参数

- **name** (*bytes*) -- 必须是 `b'hdinfo'` 或者 `b'keyinfo'`
- **flag** (*int*) -- 必须是 `1`

查询当前设备硬件信息

当 `name` 等于 `b'hdinfo'` 时候，调用这个函数可以获取当前设备硬件信息

参数

- **arg** (*int*) -- 获取不同类型的设备信息，有效值：0，1，2，3
- **kwarg** (*str*) -- None，或者设备名称

返回

arg 为 0 返回硬盘序列号

返回

arg 为 1 返回网卡以太网地址

返回

arg 为 2 返回 IPv4 地址

返回

arg 为 3 返回当前设备的名称

返回类型

字符串

例如，

```
__pyarmor__(0, None, b'hdinfo', 1)
__pyarmor__(1, None, b'hdinfo', 1)
```

在 Linux 系统，`kwarg` 可以用来指定网卡或者硬盘的设备名称，例如:

```
__pyarmor__(0, "/dev/vda2", b'hdinfo', 1)
__pyarmor__(1, "eth2", b'hdinfo', 1)
```

在 Windows 系统，`kwarg` 还可以用来得到全部网卡或者全部硬盘的信息。例如:

```
__pyarmor__(0, "/0", b'hdinfo', 1)    # First disk
__pyarmor__(0, "/1", b'hdinfo', 1)    # Second disk

__pyarmor__(1, "*", b'hdinfo', 1)
__pyarmor__(1, "*", b'hdinfo', 1)
```

查询运行密钥中的数据和有效期

当 `name` 等于 `b'keyinfo'` 的时候，调用这个函数可以获取运行密钥中信息

参数

- **arg** (*int*) -- 指定获取的信息，有效值：0，1
- **kwarg** -- None，暂时没有使用

返回

arg 为 0 返回运行密钥绑定的私有数据，没有绑定的数据返回 `b''`

返回类型

Bytes

返回

arg 为 1 返回运行密钥的有效期 (epoch)，如果没有设置有效期返回 -1

返回类型

Long

返回

如果发生错误，那么返回 None

例如，

```
print('bind data is', __pyarmor__(0, None, b'keyinfo', 1))
print('expired epoch is' __pyarmor__(1, None, b'keyinfo', 1))
```

__assert_armored__(*arg*)

参数

arg(*object*) -- 模块或者函数

返回

如果 arg 指定的对象是被加密过的，那么返回 arg 自身，否则抛出保护异常

例如

```
m = __import__('abc')
__assert_armored__(m)

def hello(msg):
    print(msg)

__assert_armored__(hello)
hello('abc')
```

3.3.5 错误消息

这里列出了加密时候和运行加密脚本的时候的错误信息列表，并且给出了导致错误发生的可能原因以及解决方案。

如果错误信息没有在这里找到，那么一般情况下这种错误不是 Pyarmor 引起的，可能是因为系统环境配置不正确，缺失系统包等原因造成的。这部分原因不需要 Pyarmor 进行任何修改，只要把环境配置正确，安装必要的包等就可以解决。对于这种类型的问题，请直接在百度或者其他任何搜索引擎，网站和论坛等查找解决方案。

加密时候的错误消息

加密常见错误信息

下表列出的是运行命令 **pyarmor** 时候常见的错误信息，部分可能的原因和解决方案

表 5: 表-1. 加密时候错误信息表

错误信息	原因和解决方案
out of license	使用试用版或者当前许可证没有的功能 解决方案请参考许可模式和许可证
not machine id	当前设备的硬件信息发生了改变可能会造成这个错误 重现在当前设备注册 Pyarmor 可以解决这个问题
query machine id failed	如果无法获取到当前设备的相关硬件信息，会报这个错误
relative import ”%s” overflow	尝试直接加密一个脚本，但是脚本里面使用了相对导入的语句 解决方案: 直接加密脚本所在的包 (目录)，而不是单独加密一个文件

注册错误信息

下表列出一般发生在注册 Pyarmor 时候的错误信息

表 6: 表-1.1 注册时候错误信息表

错误信息	原因和解决方案
HTTP Error 400: Bad Request	请升级到 Pyarmor 8.2+，以显示正确的错误信息，然后采取相应的解决方案
HTTP Error 401: Unauthorized	在新版本的许可证下使用 pyarmor-7 命令 没有解决方案，pyarmor-7 只支持老版本的许可证
HTTP Error 503: Service Temporarily Unavailable	在 1 分钟之内使用了多次注册命令 许可证服务器一分钟之内至多允许同一个 IP 的三次请求，过多的请求将返回 503 错误
unknown license type OLD	在 Pyarmor 8 中使用老版本的许可证 请参阅这里的升级说明许可模式和许可证 解决方案：使用命令 `pyarmor-7`
This code has been used too many times	如果是在 CI/Docker 中使用 Pyarmor，请通过该订单的注册邮箱发送订单信息到 pyarmor@163.com 以解锁该订单

运行加密脚本的错误信息

Pyarmor 报告的错误信息

这里列出的是运行加密脚本的时候 Pyarmor 报出的错误信息，部分可能的原因和解决方案。

如果该消息有错误代码，那么用户可以使用消息代码对这个消息进行国际化和本地化处理。没有错误代码的消息无法进行定制。

表 7: 表-2. 运行加密脚本的错误信息表（Pyarmor）

错误代码	错误信息	原因和解决方案
	error code out of range	Internal error
error_1	this license key is expired	
error_2	this license key is not for this machine	
error_3	missing license key to run the script	
error_4	unauthorized use of script	
error_5	this Python version is not supported	
error_6	the script doesn't work in this system	
error_7	the format of obfuscated script is incorrect	可能的原因: 1. 加密脚本是由其他不兼容的 Pyarmor 生成的，请使用最新的 Pyarmor 进行加密 2. 无法获得运行辅助包所在的路径，也会报这个错误
error_8	the format of obfuscated function is incorrect	
	RuntimeError: Resource temporarily unavailable	设置了加密脚本的有效期，但是无法访问时间服务器导致的问题 这个问题无法解决，要么使用本地时间，要么用户使用脚本补丁自己进行校验

Python 报告的错误信息

通常情况下，这种错误不是 Pyarmor 造成的。

解决这里的问题只需要参考 Python 的文档，把加密脚本看作是普通脚本，就可以解决问题，也可以直接在百度或者 Python 相关的论坛网站找到答案。

表 8: 表-2.1 运行加密脚本的错误信息表（Python）

错误信息	原因和解决方案
ImportError: attempted relative import with no known parent package	1. `from .pyarmor_runtime_000000 import __pyarmor__` 解决方案：不要使用 *-i* 或者 *--prefix* 去加密脚本

3.4 深入了解

3.4.1 加密过程详解

本文档适用于下列用户

- 默认选项无法满足加密脚本的性能或者安全方面的需求
- 加密脚本出现问题不知道错在那里

阅读本文档需要一定的 Python 基础，了解 Shell 脚本，环境变量等相关知识。

一般用户在操作系统中都一个用户目录，默认情况 pyarmor 运行过程的配置和数据文件会存放在这里。在命令行使用下面的命令查看配置目录的位置:

```
$ echo $HOME/.pyarmor
```

或者在命令行输入 python 命令，打开 Python 控制台，运行下面的语句:

```
>>> import os
>>> print(os.path.expanduser("~/.pyarmor"))
```

备注: 在本文档的示例代码中，以 *$* 开头的命令都是 Shell 命令，而以 »> 开头的都是在 Python 控制台运行的脚本语句。

在本文档中，后面会使用 `~/.pyarmor/` 表示这个目录。

我们首先看下面是一个最简单的加密命令:

```
$ pyarmor gen foo.py
```

运行完成之后，加密脚本存放在目录 *dist/* 下面，输出的文件:

```
dist/
    foo.py
    pyarmor_runtime.so
```

在这个过程中，pyarmor 都做了什么呢?

设置运行环境

pyarmor 首先设置运行环境，主要的步骤如下

1. 创建运行环境 ctx，根据全局配置文件进行初始化
2. 如果当前目录存在 `.pyarmor/config` 文件，使用其中的选项修改 ctx
3. 读取命令行选项，修改运行环境 ctx

搜索脚本

接下来 pyarmor 根据命令行传入的文件或者目录，开始搜索需要加密的所有脚本。

配置项 `pyexts` 列出了需要加密的脚本扩展名，默认是 `[.py, .pyw]`

首先，命令行输入的脚本，即便扩展名不在 `pyexts` 之中，也会被增加到需要加密的脚本列表 ctx.resources

其次，命令行输入的目录下面，所有扩展名在 `pyexts` 之中的会被增加到列表

然后根据下面的两个配置项，决定下一步的搜索范围

- 选项*`--recursive`* 递归搜索所有的子目录
- 选项 `--find-all` 查找所有依赖的模块和包（尚未实现）

依赖脚本查找

这个功能尚未实现

当 `--find-all` 选项设定的时候，会自动查找加密脚本的依赖模块和包。

pyarmor 首先设定搜索路径

- 继承目前 Python 系统设置的搜索路径
- 把配置项 `pypaths` 中指定的路径增加到搜索列表
- 把当前目录增加到搜索列表

然后使用下面的算法查找脚本 *foo* 依赖模块:

1. 把脚本 foo.py 解析生成语法树 tree
2. 搜索树中所有的 import/from...import 语句
3. 把其中导入的模块作为依赖项
4. 根据模块名称和搜索路径，查找对应的模块文件
5. 把模块文件加入到需要加密的脚本列表

有一些隐含导入的模块无法通过上面的算法发现，这时候会使用脚本的规则文件来辅助查找。脚本的规则文件一般是人工创建，把依赖文件或者包名称添加到组 `finder` 下面，例如:

```
[finder]
includes = a.py b/c d.pt
```

脚本规则文件的命名和存放路径必须符合下面的规则，否则无法起作用:

- *~/.pyarmor/config/foo.rules* 存放在全局配置目录 rules 下面的同名文件
- *.pyarmor/foo.rules* 当前配置目录下的扩展名为.rules 的同名文件

如果规则文件存在，那么这里面指定的模块和文件都会被进行加密处理，并且如果指定了具体的文件名称，即便扩展名不在配置项 `.pyexts` 也会被增加到加密列表中。

源代码处理

找到所有需要处理的脚本之后，下一步就是依次对每一个脚本的源代码进行处理

源代码的处理算法主要步骤

- 读取文件内容，生成资源文件 *res*
- 遍历环境配置中所有源代码处理器，依次处理资源文件
- 处理完成之后把源代码编译成为语法树

文件的编码使用的是系统默认值，如果读取文件出现编码错误，修改配置文件的选项 `encoding` ，设定正确的值。

内联注释处理器

内置的源代码处理器是内联注释处理器，默认启用的。

内联注释处理器只是简单的把源代码中每一行包含的 `# pyarmor:` 字符串替换为空，这样可以使得原来被注释的代码生效。例如，原来的脚本如下

```
# pyarmor: print('inline pyarmor code')
def foo():
    i = 2
    # pyarmor: i += 1
    print('i is', i)
```

处理完之后，代码的内容就变成

```
print('inline pyarmor code')
def foo():
    i = 2
    i += 1
    print('i is', i)
```

这个的设计初衷是有时候需要执行一些只能在加密脚本中才能运行的代码，这样在没有加密的时候，调试起来很不方便。所以先把这部分的代码注释掉，在加密之前把注释自动去掉。

语法树处理

源代码处理完成之后，输出的是源代码对应的 *ast.Module* 语法树。

语法树的处理算法主要步骤

- 根据环境配置，找到所有的语法树处理器
- 遍历所有语法树处理器，依次处理资源文件
- 最后输出的是经过各个处理器修改后的语法树

为了节省内存，源代码处理完成，文件内容会被释放。如果语法树处理器需要从文件的源代码读取信息，需要重新打开文件读取。

每一个语法处理器接受的输入是上一个语法处理器的输出，所以后面的语法处理器处理的语法树可能和最初的源代码并不一致。

内置的语法树处理器主要有二个

- 函数调用处理器，通过修改 ast.Call，确保调用的函数是经过加密的
- 导入模块处理器，通过修改 ast.Import，确保导入的模块是经过加密的

模块代码处理

所有的语法树处理器完成之后，输出的修改后的 *ast.Module* 语法树。接下来就是对模块代码进行处理:

```
* 根据环境配置，找到所有的模块代码处理器
* 编译语法树为模块代码对象 mco
* 遍历所有模块代码处理器，依次处理资源文件的模块代码对象 mco
* 最后输出的是经过各个处理器修改后的模块代码对象
```

每一个模块代码处理器接受的输入是上一个模块代码处理器的输出，所以除了第一个代码处理器之外，后面的处理器输入的 mco 可能和语法树 mtree 不一致。

内置的模块代码处理器有二个

- 重构代码处理器，主要实现自动重命名模块中类，函数，属性和变量的名称
- 加密代码处理器，主要是对代码对象进行加密处理

参见:

--obf-code --mix-str --enable-rft --enable-bcc

加密脚本生成

下列相关选项影响加密脚本中导入运行辅助包 的方式

- *--prefix*
- *-i*

同名脚本的输出路径

如果存在同名脚本，例如:

```
pyarmor gen a/foo.py b/foo.py
```

默认输出分别是:

- a/foo.py -> `dist/foo.py`
- b/foo.py -> `dist-1/foo.py`

运行辅助文件生成

选项*--enable-themida* 和*--platform* 决定了运行辅助文件中需要那些动态库。

3.4.2 深入了解加密脚本

阅读本文档需要一定的 Python 基础，了解 Shell 脚本，环境变量等相关知识。

加密脚本就是普通 Python 文件

Pyarmor 加密后的脚本输出的是同名的 `.py` 文件和一个运行辅助包 。它们和普通 Python 模块一样，可以被 Python 解释器调用执行，这也是 Pyarmor 的一个加密特点，可以使用加密后的脚本无缝替换原来的脚本。

使用加密脚本完全和使用普通的 Python 脚本一样，例如，使用解释器直接运行:

```
python dist/foo.py
```

使用文本编辑器打开这个脚本，它的内容一般如下:

```
from pyarmor_runtime import __pyarmor__
__pyarmor__(__name__, __file__, b'\x28\x83....')
```

可以看到第一条语句就是导入扩展模块 `pyarmor_runtime` ，它就是和加密脚本在相同目录下的文件 `pyarmor_runtime.so` 。扩展模块 `pyarmor_runtime` 不是必须和加密脚本放在一起，只要它存在于 Python 搜索模块的任何路径，能被 Python 导入进来，加密脚本就可以正常使用。

如果运行加密脚本的时候提示模块 pyarmor_runtime 无法找到，首先要能找到扩展模块文件，然后在扩展模块文件所在的目录直接使用 Python 解释器导入这个模块:

```
cd dist/
python
>>> import pyarmor_runtime
```

如果不能正常导入，说明文件格式不正确，不适用于当前平台和 Python 版本。

扩展模块是二进制的动态库，有些没有加密的时候，脚本可以正常执行，加密之后无法运行。就是因为运行环境无法装载扩展模块而造成的。这种情况只要了解扩展模块的相关知识，设置运行环境允许加载扩展模块，都可以正常运行加密脚本。判断运行环境是否允许加载扩展模块的方法是把任何一个系统扩展模块拷贝到当前目录，看看能否导入。

在导入扩展模块 pyarmor_runtime 之前，所有的事情都是 Python 的自身功能，和 Pyarmor 和脚本是否加密都没有关系。解决这里出现的问题需要的就是学习 Python 相关的知识，特别是 Python 是如何根据模块名称去搜索和装载模块和扩展模块的。

扩展模块 pyarmor_runtime 第一次被导入的时候，会进行一些初始化工作，包括检查运行密钥 等，如果初始化失败，那么抛出异常退出。

装载加密模块

如果初始化正常完成，那么执行加密脚本的第二行语句，调用从扩展模块中导入的函数 __pyarmor__ 来完成对加密模块的装载工作。

把模块加载完成之后，就又把控制器交给 Python 解释器，执行解密后的模块。

装载加密函数

当 Python 解释器调用加密函数的时候，控制权交给扩展模块 pyarmor_runtime 进行解密，解密完成之后返回 Python 解释器继续执行。

函数调用返回之前，控制权重新交给扩展模块 pyarmor_runtime 进行加密和做一些保护清理工作，最后在返回给 Python 解释器继续执行。

运行辅助包

上面对加密脚本的说明中进行了简化，实际加密之后扩展模块 pyarmor_runtime 是在一个运行辅助包 里面，让我们查看一下加密后的目录就一目了然:

```
$ pyarmor gen foo.py

$ ls dist/

foo.py    pyarmor_runtime_000000

$ ls dist/pyarmor_runtime_000000
```

(续下页)

(接上页)

```
...     __init__.py
...     pyarmor_runtime.so
```

这里 dist/pyarmor_runtime_000000 就是运行辅助包，使用运行辅助包主要是为了能够支持在多平台运行的加密脚本，我们可以把预编译的不同平台扩展模块 pyarmor_runtime 都存放到包目录下面，然后在 __init__.py 里面根据不同的平台，导入相应平台的扩展模块，这样就可以让加密脚本运行在多平台下面。

运行辅助包也是一个正常的*Python 包*，它不是必须和加密脚本在一起，只要满足 Python 模块导入机制的要求，它可以存放在任何地方。

运行辅助包可以使用相对导入的方式，也可以使用绝对导入的方式。Pyarmor 提供了相关选项*-i* 和*--prefix* 来帮助加密脚本生成正确的导入语句。如果还不能满足需求，可以自己编写加密插件 来修改加密脚本的导入语句，从而确保能导入运行辅助包。

运行密钥

运行密钥保存对加密脚本的约束信息以及相关的一些运行设置。

运行密钥一般嵌入到扩展模块中，但是也可以使用外部密钥文件。

扩展模块在初始化的时候要验证运行密钥，验证失败就直接报错退出，只要验证成功之后才会继续执行。

运行密钥的验证模式只是在扩展模块初始化的时候进行验证，但是可以配置成为每一次导入加密模块都进行验证，也可以配置成为定时进行验证。

如果使用外部密钥文件，可以在其头部插入任何可读文本作为注释，这样通过这些注释，不需要在加密脚本内部就可以读取运行密钥的相关信息，从而做一些额外的处理。

用户可以在运行密钥中绑定任何私有数据（但是长度有一定限制，不能超过 4K），然后使用脚本补丁 在加密脚本自己去验证这些数据，从而实现对加密脚本的限制和约束。这里所有的数据格式和业务逻辑全部由用户自己控制，Pyarmor 只是提供了这种扩展机制。

约束模式

约束模式用来对加密脚本进行一定的约束和限制。

默认约束模式是不允许对加密后的脚本进行修改。

使用*--private* 之后不允许外部脚本导入加密脚本

使用*--restrict* 之后不运行外部脚本访问加密脚本中方法

如果需要禁用全部的约束，使用下面的命令:

```
$ pyarmor cfg restrict_module 0
```

一般情况下，是对某一个特殊脚本禁用约束，而其他脚本的约束不变，这样在配置的时候需要指定相应的模块:

```
$ pyarmor cfg -p NAME restrict_module 0
```

加密脚本和原来脚本的区别

加密脚本和原来的脚本相比，存在下列一些的不同:

- 加密脚本是和 Python 版本绑定的，例如使用 Python 3.5 加密的脚本，只能使用 Python 3.5 去运行，而无法使用 Python 3.6 去运行，但是可以使用不同补丁的 3.5 版本。
- 加密脚本是平台相关，因为使用到了动态库，不同的平台需要相应平台的动态库。平台根据操作系统，CPU 架构来进行区别，例如 32 位 X86 Windows，Linux Aarch64。
- 执行加密角本的 Python 不能是调试版，准确的说，不能是设置了 Py_TRACE_REFS 或者 Py_DEBUG 生成的 Python
- 使用 `sys.settrace`, `sys.setprofile`, `threading.settrace` 和 `threading.setprofile` 设置的回调函数在加密脚本中将被忽略，所以任何使用这些函数的工具无法正常工作。
- 模块 `inspect` 和其他任何第三方包如果试图访问加密脚本的 Byte Code 或者直接访问代码对象的某些属性，也会崩溃，失败或者得到错误的数据
- 使用 `cPickle` 或者其他序列化工具传递加密代码对象，传递之后的代码对象可能无法正常运行。
- `sys._getframe([n])` 可能得到的不是期望的运行框架，因为加密脚本可能增加了额外的运行框架。
- 加密脚本抛出异常中的行号和原来的脚本在个别情况下会不一样
- 代码块的属性 `__file__` 在加密脚本是 `<frozen name>`，而不是文件名称，在异常信息中会看到文件名的显示是 `<frozen name>`

  需要注意的是模块的属性 `__file__` 还和原来的一样，还是文件名称。加密下面的脚本并运行，就可以看到输出结果的不同:

```
def hello(msg):
    print(msg)

# The output will be 'foo.py'
print(__file__)

# The output will be '<frozen foo>'
print(hello.__file__)
```

有些选项也会影响到脚本的内部结构:

- `pyarmor cfg mix_argname=1` 会导致 annotations 无法使用

参见:

如何解决第三方库调用加密脚本存在的问题

第三方解释器的支持

对于第三方的解释器（例如 Jython 等）以及通过嵌入 Python C/C++ 代码调用加密脚本，只要第三方解释器能够和 CPython 扩展模块 兼容，就可以使用加密脚本。请自行查看第三方解释器的文档，确认它是否支持 CPython 的扩展模块。

已知的一些问题

- *PyPy* 无法运行加密脚本，因为它完全不同于 *CPython* 。
- 在 Linux 下面装载 Python 动态库 *libpythonXY.so* 的时候 *dlopen* 必须设置 *RTLD_GLOBAL* ，否则加密脚本无法运行。
- Boost::python，默认装载 Python 动态库是没有设置 *RTLD_GLOAL* 的，运行加密脚本的时候会报错”No PyCode_Type found” 。解决方法就是在初始化的调用方法 *sys.setdlopenflags(os.RTLD_GLOBAL)* ，这样就可以共享动态库输出的函数和变量。
- 模块 *ctypes* 必须存在并且 *ctypes.pythonapi._handle* 必须被设置为 Python 动态库的句柄，PyArmor 会通过该句柄获取 Python C API 的地址。
- WASM 目前不支持，因为这需要把运行库的代码也编译成为 WASM，但是 WASM 是很容易就被反编译成为原来的 C 代码，为了安全性，所以目前没有支持 WASM 的计划。如果有更多的用户提出这个需求，会考虑实现一个轻量级的运行库，只支持能够运行 RFT 模式的加密脚本，但是目前还没有开发计划。

3.4.3 详解可独立运行的加密脚本

使用 Pyarmor 8.0 生成可以独立运行的加密脚本，必须首先调用 PyInstaller 将脚本打包成为单独的可执行文件或者打包到一个目录，然后把打包生成的可执行文件通过选项 `--pack` 传递给*pyarmor gen* 来实现:

```
pyinstaller foo.py
pyarmor gen --pack dist/foo/foo foo.py
```

如果没有选项 `--pack` ，只是把脚本进行加密。至于那些脚本被加密，请参考*pyarmor gen* 中说明，这里不会自动加密依赖项。

如果有选项 `--pack` ，那么默认输出目录是 `.pyarmor/pack/dist` ，在加密完成之后，还进行下面的额外处理:

- 提取可执行文件中内容到一个临时目录 `.pyarmor/pack/`
- 使用加密脚本替换临时目录中同名的未加密脚本
- 把加密脚本的运行辅助文件 增加到临时目录中
- 根据把临时目录中所有内容重新生成可执行文件，并替换原来的可执行文件

自己动手打包加密脚本

如果使用上面的方式出现问题，或者打包好的可执行文件出现执行错误，那么请使用下面的方式自己打包加密脚本。

这需要你了解 如何使用 PyInstaller 和 如何使用 spec file，如果还不知道如何使用，请点击链接学习相关知识。

下面我们使用一个例子来说明如何手动打包加密脚本 `/path/to/src/foo.py`

- 首先使用 Pyarmor 加密这个脚本[1]:

```
cd /path/to/src
pyarmor gen -O obfdist -a foo.py
```

- 然后把*运行辅助包* 移到当前目录[2]:

```
mv obfdist/pyarmor_runtime_000000 ./
```

- 如果已经有 `foo.spec`，需要把运行辅助包增加到 `hiddenimports`

```
a = Analysis(
    ...
    hiddenimports=['pyarmor_runtime_000000'],
    ...
)
```

- 如果还没有这个文件，使用下面的命令生成 `foo.spec`[3]:

```
pyi-makespec --hidden-import pyarmor_runtime_000000 foo.py
```

- 修改 `foo.spec`，插入补丁代码到 `a = Analysis` 之后，这一步是重点

```
a = Analysis(
    ...
)

# Patched by PyArmor
_src = r'/path/to/src'
_obf = r'/path/to/src/obfdist'

_count = 0
for i in range(len(a.scripts)):
    if a.scripts[i][1].startswith(_src):
        x = a.scripts[i][1].replace(_src, _obf)
```

(续下页)

[1] 不要使用选项 *-i* 和 *--prefix* 加密脚本，其他选项可以尝试

[2] 这一步的目的只是为了方便让 *PyInstaller* 能够找到运行辅助包而不需要增加额外路径

[3] 其他 PyInstaller 的选项也可以在这里使用

(接上页)

```
        if os.path.exists(x):
            a.scripts[i] = a.scripts[i][0], x, a.scripts[i][2]
            _count += 1
if _count == 0:
    raise RuntimeError('No obfuscated script found')

for i in range(len(a.pure)):
    if a.pure[i][1].startswith(_src):
        x = a.pure[i][1].replace(_src, _obf)
        if os.path.exists(x):
            if hasattr(a.pure, '_code_cache'):
                with open(x) as f:
                    a.pure._code_cache[a.pure[i][0]] = compile(f.read(), a.pure[i][1],
↪ 'exec')
            a.pure[i] = a.pure[i][0], x, a.pure[i][2]
# Patch end.

pyz = PYZ(a.pure, a.zipped_data, cipher=block_cipher)
```

- 最后直接使用打过补丁的 `foo.spec` 来打包:

```
pyinstaller foo.spec
```

请根据你的具体情况，做如下修改

- 使用实际目录替换 `/path/to/src`
- 使用实际名称替换 `pyarmor_runtime_000000`

如何验证打包进去的是加密脚本

方法一，可以在 `foo.spec` 的补丁代码增加一些 print 语句，验证加密脚本已经替换了原来的脚本

方法二，可以在主脚本增加一些调试语句进行判断，例如

```
print('this is __pyarmor__', __pyarmor__)
```

如果不是加密脚本，这个语句会报错，只有在加密脚本中才能正常打印。

备注

Apple M1 的 segment fault

在 Apple M1 上打包，如果生产的加密脚本运行时候发生崩溃，请首先检查运行辅助包的签名:

```
$ codesign -v dist/foo/pyarmor_runtime_000000/pyarmor_runtime.so
```

如果签名非法，请重新进行签名:

```
$ codesign -f -s dist/foo/pyarmor_runtime_000000/pyarmor_runtime.so
```

如果使用了*--enable-bcc*或者*--enable-jit*进行加密，那么还需要启用 Allow Execution of JIT-compiled Code Entitlement

参见:

Using the latest code signature format

3.4.4 深入了解 RFT 模式

Pyarmor 的 RFT 模式能够对模块中的函数，类，方法，属性，变量等进行重命名，相当于把源代码重新写了一下，所以这种加密方式不可逆的。

启用 RFT 模式之后，模块中的所有定义的名称都会发生变化，外部脚本将无法使用原来的名称进行导入。如果需要输出某些名称被外部脚本使用，需要在模块中定义 `__all__`，这里面列出的名称会被保留。

RFT 模式会修改下列名称

- 函数
- 类
- 方法
- 全局变量
- 局部变量
- 内置名称
- 导入到名称

RFT 模式不会修改的名称

- 参数名称
- 调用函数的时候使用的关键字名称
- 使用模块属性 `__all__` 输出的所有名称
- 所有以 `__` 开头的名称

使用 RFT 模式的时候，必须把所有使用的脚本和包在同一条命令进行加密。Pyarmor 会分析脚本之间的调用关系，对导入的名称也进行正确的重命名。例如:

```
src/
    foo.py
    joker/
        __init__.py
        card.py
```

使用 RFT 模式加密脚本 foo.py 以及包 joker 的命令如下:

```
pyarmor gen --enable-rft foo.py src/joker/
```

Pyarmor 使用内置的自动规则和人工配置的规则来分析脚本，并进行重命名。虽然 RFT 模式的自动规则可以处理大多数的情况，但是对于复杂的脚本和包，肯定会存在一些名称没有被正确处理。因为 Python 语言自身的特点，Pyarmor 并不认为可以找到算法自动处理所有的情况，所以对特殊情况，必须要进行人工配置规则。RFT 模式的目标是不断更新和完善自动规则，对于不能自动处理的部分，能够自动生成相关的参考配置，用户通过这些参考配置生成人工规则。

对于复杂的脚本，不要期望 Pyarmor 会自动进行处理所有情况。例如，

```
foo().stack[2].count = 3
(a+b).tostr().get()
```

是否对属性 `stack` ，`count` ，`tostr` 和 `get` 进行重命名呢? 在有些情况下，使用静态的语法分析根本无法判断属性所在的变量类型，因为有些表达式的类型是动态执行的时候确定的，甚至同一个表达式会在不同的情况下返回不同的类型。

为了处理这种情况，RFT 模式提供了两种方法:

- **rft-auto-exclude**

  自动排除未知属性，这是默认方法。

  这种方法是搜索所有的脚本中所有的属性链，如果属性所在的类型名称无法确定，那么把这个属性名称增加到排除属性表，所有被排除的属性在其他任何脚本中都不会被重命名。

  排除属性表存放的文件名称是 `.pyarmor/rft_exclude_table`

  第一次运行 RFT 模式加密脚本的时候，这个表是空的。RFT 模式依次扫描各个脚本，发现无法处理的属性，就把它添加到这个表中。当所有的脚本都加密完成之后，这个表里面的名称就是所有无法处理的属性名称。

  这时候再次使用 RFT 模式使用相同的参数加密脚本，就可以保证所有的未知属性没有命名，基本不会出现名称绑定的错误。

  RFT 模式不会自动删除这个表 `.pyarmor/rft_exclude_table` ，而是不断增加新的未知属性到里面。如果有需要的话，可以把这个表删除掉，然后重新开始一个全新的重命名过程。

这种方法使用比较简单，但是可能会排除很多名称，查看排除属性表可以知道那些属性名称被保留下来了。

- **rft-auto-include**

  自动命名所有的类和方法。

  这种方法是首先扫描全部脚本，把脚本中函数，类和方法全部都进行重命名，确保这些名称能被重命名。

  对于那些属性链中无法处理的属性名称，保留下来不进行处理。

  用户需要运行加密脚本，如果出现名称错误，那么把这些名称人工进行排除，或者使用人工规则进行排除。

  不断重复上述步骤，直到没有名称错误出现。

  这种方法能够重命名大部分的名称，但是需要更多额外的工作。

启用 RFT 模式

使用下面的命令启用 RFT mode:

```
$ pyarmor gen --enable-rft foo.py
```

也可以使用 **pyarmor cfg** 通过配置文件启用:

```
$ pyarmor cfg enable_rft=1
$ pyarmor gen foo.py
```

启用 **rft-auto-include** 方法通过禁用 `rft_auto_exclude`:

```
$ pyarmor cfg rft_auto_exclude=0
```

重新启用 **rft-auto-exclude** 方法:

```
$ pyarmor cfg rft_auto_exclude=1
```

查看重命名之后的完整脚本

使用 RFT 模式修改后的脚本究竟能否满足安全需要？可能需要先看一下。

在 Python 3.9+ 启用 RFT 跟踪模式，可以输出相应的脚本:

```
$ pyarmor cfg trace_rft 1
$ pyarmor gen --enable-rft foo.py
$ ls .pyarmor/rft
```

foo.py 转换后对应的脚本是 .pyarmor/rft/foo.py:

```
$ cat .pyarmor/rft/foo.py
```

转换后的脚本表示的只是 RFT 模式所进行的修改，并不完全等价于加密后的脚本，它还会按照加密选项的设置进行下面的处理。例如 docstring 现在还保留着，但是如果使用了下面的配置项:

```
$ pyarmor cfg optimize 2
```

加密脚本最终会把所有 DocString 删除。

备注： Python 3.8 以及之前的版本不支持该功能。

查看重命名的名称

如果需要跟踪那些名称被重命名，可以同时启用日志跟踪和 RFT 跟踪选项，这时候会生成跟踪日志 .pyarmor/pyarmor.trace.log，里面有所有的重命名日志:

```
$ pyarmor cfg enable_trace=1 trace_rft=1
$ pyarmor gen --enable-rft foo.py
$ grep trace.rft .pyarmor/pyarmor.trace.log

trace.rft            foo:1 (import sys as pyarmor__1)
trace.rft            foo:12 (self.wScan->self.pyarmor__4)
```

第一个日志记录了 sys 被重命名为 pyarmor__1

第二条日志记录了属性 wScan 被重命名为 pyarmor__4

人工排除属性名称

如果加密后的脚本出现名称错误，最简单的解决方式就是把这个名称人工排除，也就是说，在所有的脚本中保留这个名称（属性）而不进行重命名。下面的命令把名称 mouse_keybd 保留下来:

```
$ pyarmor cfg rft_excludes + "mouse_keybd"
$ pyarmor gen --enable-rft foo.py
```

如果错误显示的名称是像 pyarmor__22 这样的格式，那么首先通过跟踪日志反向查找出原来的名称:

```
$ grep pyarmor__22 .pyarmor/pyarmor.trace.log

trace.rft            foo:65 (self.height->self.pyarmor__22)
trace.rft            foo:81 (self.height->self.pyarmor__22)
```

例如，上例中 height 是 pyarmor__22 原来的名称，那么使用下面的命令保留这个名称:

```
$ pyarmor cfg rft_excludes + "height"
$ pyarmor gen --enable-rft foo.py
$ python dist/foo.py
```

如果需要把这些人工排除的名称清除，使用下面的命令:

```
$ pyarmor cfg rft_excludes ""
```

处理模块属性 __all__

模块属性 __all__ 中定义的名称会被保留下来，可以输出名称供外部脚本使用。例如，下面的函数 foo 会被保留下来不进行重命名

```
__all__ = ['foo']

def foo(msg):
    print(msg)

def _private_foo(msg):
    print(msg)
```

如果输出的是一个类，那么类中的方法和属性也会被保留下来。

有时候可能需要重命名所有的类和方法，可以使用下面的选项让 RFT 模式忽略 __all__ 中定义的名称:

```
$ pyarmor cfg rft_export__all__ 0
```

处理特殊的导入语句

导入全部名称的语句—from module import *—会进行特殊的处理。

如果是从一个加密模块里面导入所有名称，那么 RFT 模式会直接解析源代码并直到属性 __all__ 的定义。

如果是从一个外部模块中导入所有名称，那么 RFT 模式会尝试直接导入这个模块，并且尝试得到模式属性 __all__ 。如果导入失败，会导致 RFT 模式报错退出。在这种情况下，需要设置环境变量 PYTHONPATH 或者其他任何方式让 Python 可以导入这个模块。

如果模块属性 __all__ 没有定义，那么模块的所有属性中不是以 _ 开头的名称都会被导入进来。

3.4.5 深入了解 BCC 模式

BCC 模式会把部分函数直接转换成为二进制代码，在根本上避免被还原成为 Python 函数。

BCC 模式需要配置*C* 编译器，对于 *Linux* 和 *Darwin* 来说，一般不需要进行配置，只要默认的 `gcc` 和 `clang` 能工作就可以。在 *Windows* 环境下面，可以使用下面任意一种方式配置 `clang.exe` ，目前其它编译器还不支持:

- 如果已经有 `clang.exe` ，只要在其它路径直接运行 `clang.exe` 不出错就可以。如果文件存在，但是无法在任意路径直接运行，可以配置环境变量 `PYARMOR_CC` 来指定这个文件，例如:

```
set PYARMOR_CC=C:\path\to\clang.exe
```

- 从 LLVM 官网 下载并安装预编译版本
- 从 Pyarmor 官网下载 clang-9.0.zip ，压缩包大小约为 26M 左右，里面只有一个可执行文件，解压后存放在根目录 下面，默认是 `%HOME%/.pyarmor`

启用 BCC 模式

配置好编译器之后，使用选项*--enable-bcc* 启用 BCC 模式:

```
$ pyarmor gen --enable-bcc foo.py
```

模块级别的代码不会转换转换成为*C* 的函数，模块的任何函数如果使用了不被支持的特性，也不会转换成为*C* 函数，这么没有使用 BCC 模式加密的函数会根据选项使用其他方式进行加密。

查看被 BCC 模式加密的函数

启用跟踪模式可以在跟踪日志文件 `.pyarmor/pyarmor.trace.log` 中记录那些函数被转换成为了*C* 函数。例如:

```
$ pyarmor cfg enable_trace=1
$ pyarmor gen --enable-bcc foo.py
```

查看跟踪日志中使用 `trace.bcc` 记录的内容:

```
$ ls .pyarmor/pyarmor.trace.log
$ grep trace.bcc .pyarmor/pyarmor.trace.log

trace.bcc            foo:5:hello
trace.bcc            foo:9:sum2
trace.bcc            foo:12:main
```

第一条日志记录的是 `foo.py` 第 5 行的函数 `hello` 被转换成为 C 函数第二条日志记录的是 `foo.py` 第 9 行的函数 `sum2` 被转换成为 C 函数

不转换特定的模块和函数

使用 BCC 模式加密脚本并不是完全和原来的脚本兼容，如果运行 BCC 模式加密脚本出现了兼容性问题，那么解决方案就是不要转换特定的模块，或者模块中特定的函数。

为了避免影响其他脚本，这里使用模块私有配置 来修改单独修改模块的设置。

让 BCC 模式忽略一个模块 `pkgname.modname` 使用下面的命令:

```
$ pyarmor cfg -p pkgname.modname bcc:disabled=1
```

忽略模块中一个函数使用下面的命令:

```
$ pyarmor cfg -p pkgname.modname bcc:excludes + "function name"
```

忽略更多的函数:

```
$ pyarmor cfg -p foo bcc:excludes + "hello foo2"
```

我们可以启用跟踪日志，查看这些语句的效果，看看这些模块和函数是否没有被 BCC 模式处理。例如:

```
$ pyarmor cfg enable_trace 1
$ pyarmor gen --enable-bcc foo.py
$ grep trace.bcc .pyarmor/pyarmor.trace.log
```

另外一个例子，忽略 `joker/card.py` 但是使用 BCC 模式加密包 `joker` 的其他模块:

```
$ pyarmor cfg -p joker.card bcc:disabled=1
$ pyarmor gen --enable-bcc /path/to/pkg/joker
```

使用两个配置项 `bcc:excludes` 和 `bcc:disabled` 可以最小限度的排除不被 BCC 模式支持的代码，从而确保其他脚本能正常使用 BCC 模式加密运行。

改变的脚本特性

使用 BCC 模式加密后脚本和原来的脚本存在一些额外的不同

- 部分异常的提示信息和原来不一样
- 如果不是在异常处理的过程中，直接调用没有参数的 *raise* 抛出的异常类型不同

  没有加密前

  ```
  >>> raise
  RuntimeError: No active exception to reraise
  ```

  在 BCC 模式加密的脚本中

```
>>> raise
UnboundlocalError: local variable referenced before assignment
```

- 函数对象的属性，尤其是以双下划线开始的属性，例如 __qualname__ 等等，在转换成为 C 函数之后都不存在，使用这些属性的函数加密后无法正常工作

不支持的特性

使用了下列特性的函数无法转换成为*C* 函数

```
unsupport_nodes = (
    ast.ExtSlice,

    ast.AsyncFunctionDef, ast.AsyncFor, ast.AsyncWith,
    ast.Await, ast.Yield, ast.YieldFrom, ast.GeneratorExp,

    ast.NamedExpr,

    ast.MatchValue, ast.MatchSingleton, ast.MatchSequence,
    ast.MatchMapping, ast.MatchClass, ast.MatchStar,
    ast.MatchAs, ast.MatchOr
)
```

如果调用了下列任意一个内置函数，那么该函数也无法转换成为*C* 函数:

- exec,
- eval
- super
- locals
- sys._getframe
- sys.exc_info

例如，下面这些函数都不会使用终极模式加密，因为它们或者使用了不支持的特性，或者调用了不支持的函数:

```
async def nested():
    return 42

def foo1():
    for n range(10):
        yield n
```

(续下页)

(接上页)

```
def foo2():
   frame = sys._getframe(2)
   print('parent frame is', frame)
```

3.4.6 性能和安全

关于安全性

Pyarmor 的核心功能是保护 Python 脚本无法被反编译，通过多种不可逆加密模式的实现，已经能够实现 Python 脚本无法使用任何方式完全反编译出来。

如果你看到有人宣称能够的破解 Pyarmor，请首先参考最高安全性和最快性能，使用你所可用的最高安全选项去加密一个简单的参考脚本，然后尝试使用网上所说的方法和工具进行破解。如果能够被破解，再把 Python 版本，Pyarmor 的版本，运行的平台，加密使用的选项，参考脚本以及破解的方法发送到 pyarmor@163.com

Pyarmor 并没有提供内存数据保护和很强的反调试保护，即便没有这些保护，Pyarmor 也能保证加密脚本无法被恢复成为原来的脚本。但是对于使用各种逆向工程方法直接修改内存运行数据，以及运行时刻的内存数据，并没有提供完整的保护机制。如何对这方面的要求很高，那就需要结合 Pyarmor 提供的功能以及使用其他反调试工具和技术一起来进行保护，详细说明请参考保护运行时刻的数据安全性

关于性能

Pyarmor 提供了大量的选项来平衡性能和安全性，用户可以根据需要选择不同的选项。这里列出了所有相关的选项以及其对安全性和性能方面的影响。

需要注意的是不同的脚本，以及不同的使用场景，相同的选项对性能的影响可能有很大的不同。这里的所有性能测试数据都是基于同一个简单的测试脚本，如果对性能有很高的要求，请使用相应的场景和脚本进行测试，这里的结果不一定有参考价值。

在运行任何 Python 脚本之前，除非特别说明，都要清除脚本对应的 `__pycache__` 。这个目录存放脚本对应 `.pyc` ，如果这个目录存在，就意味着执行时间里面没有包含编译脚本的时间，但是编译脚本（.py->.pyc）是特别花费时间的，和函数的执行时间相比，不是一个数量级别。

测试脚本 `benchmark.py` 的内容如下

```
import sys


class BenTest(object):

    def __init__(self):
        self.a = 1
        self.b = "b"
        self.c = []
        self.d = {}
```

(续下页)

(接上页)

```
def foo():
    ret = []
    for i in range(100000):
        ret.extend(sys.version_info[:2])
        ret.append(BenTest())
    return len(ret)
```

主脚本 testben.py 的内容如下

```
import benchmark
import sys
import time


def metric(func):
    if not hasattr(time, 'process_time'):
        time.process_time = time.clock

    def wrap(*args, **kwargs):
        t1 = time.process_time()
        result = func(*args, **kwargs)
        t2 = time.process_time()
        print('%-16s: %10.3f ms' % (func.__name__, ((t2 - t1) * 1000)))
        return result
    return wrap


def test_import():
    t1 = time.process_time()
    import benchmark2 as m2
    t2 = time.process_time()
    print('%-16s: %10.3f ms' % ('test_import', ((t2 - t1) * 1000)))
    return m2


@metric
def test_foo():
    benchmark.foo()


if __name__ == '__main__':
    print('Python %s.%s' % sys.version_info[:2])
```

(续下页)

(接上页)

```
    test_import()
    test_foo()
```

默认加密脚本的性能

首先比较不同 Python 版本下的加密和不加密脚本的性能

使用下面的脚本，分别运行没有加密和加密的脚本两次，其中第二次运行的主要目的是测试存在 __pycache__ 情况下的模块导入时间:

```
$ rm -rf dist __pycache__

$ cp benchmark.py benchmark2.py
$ python testben.py

Python 3.7
test_import      :   1.303 ms
test_foo         : 250.360 ms

$ python testben.py

Python 3.7
test_import      :   0.290 ms
test_foo         : 252.273 ms

$ pyarmor gen testben.py benchmark.py benchmark2.py
$ python dist/testben.py

Python 3.7
test_import      :   0.907 ms
test_foo         : 311.076 ms

$ python dist/testben.py

Python 3.7
test_import      :   0.454 ms
test_foo         : 359.138 ms
```

表 9: 表-1. 不同 Python 版本的加密脚本性能测试

时长 (毫秒)	导入模块时间		导入模块时间 2		执行函数时间	
Python 版本	未加密	加密	未加密	加密	未加密	加密
3.7	1.303	0.907	0.290	0.454	252.2	311.0
3.8	1.305	0.790	0.286	0.338	272.232	295.973
3.9	1.198	1.681	0.265	0.449	267.561	331.668
3.10	1.070	1.026	0.408	0.300	281.603	322.608
3.11	1.510	0.832	0.464	0.616	164.104	289.866

可以看的出来，和之前的版本相比，关于执行函数的时间 Python 3.11 比加密后的脚本快了很多，可能和它的指令缓存和性能优化有关系。

RFT 模式性能

RFT 模式是直接把源代码的函数，类，方法和变量进行重命名，所以不应该会影响性能。这里我们比较的是加密后的脚本和 RFT 模式加密的脚本性能数据。下表中的数据是使用下面的测试脚本获得的:

```
$ rm -rf dist
$ pyarmor gen testben.py benchmark.py benchmark2.py
$ python dist/testben.py

$ rm -rf dist
$ pyarmor gen --enable-rft testben.py benchmark.py benchmark2.py
$ python dist/testben.py
```

表 10: 表-2. 默认 RFT 模式性能测试

时长 (毫秒)	导入模块时间		执行函数时间		备注
Python 版本	加密	RFT 模式	加密	RFT 模式	
3.7	1.083	1.317	334.313	324.023	
3.8	0.774	1.109	239.217	241.697	
3.9	0.775	0.809	304.838	301.789	
3.10	2.182	1.049	310.046	339.414	
3.11	0.882	0.984	258.309	264.070	

接下来，我们组合 RFT 模式和 `--obf-code` 为 0 对脚本进行加密，然后和没有加密脚本的比较一下性能。下表中的数据是使用下面的测试脚本获得的:

```
$ rm -rf dist __pycache__
$ python testben.py

$ pyarmor gen --enable-rft --obf-code=0 testben.py benchmark.py benchmark2.py
$ python testben.py
```

表 11: 表-2.1 组合 RFT 模式性能测试

时长 (毫秒)	导入模块时间		执行函数时间		备注
Python 版本	未加密	组合模式	未加密	组合模式	
3.7	0.757	1.844	307.325	272.672	
3.8	0.791	0.747	276.865	243.436	
3.9	1.276	0.986	246.407	236.138	
3.10	2.563	1.142	256.583	260.196	
3.11	0.952	0.938	185.435	154.390	

BCC 模式性能

BCC 模式是把部分函数转换成为 C 函数，应该模块装载时间略长一些，函数执行的时间能稍微快一些。下表中的数据是使用下面的测试脚本获得的:

```
$ rm -rf dist __pycache__
$ python testben.py
$ python testben.py

$ pyarmor gen --enable-bcc testben.py benchmark.py benchmark2.py
$ python dist/testben.py
$ python dist/testben.py
```

表 12: 表-3. 不同 Python 版本的 BCC 模式性能测试

时长 (毫秒)	导入模块时间		导入模块时间 2		执行函数时间	
Python 版本	未加密	加密	未加密	加密	未加密	加密
3.7	1.086	1.177	0.342	0.391	344.640	271.426
3.8	1.099	1.397	0.351	0.400	291.244	251.520
3.9	1.229	1.076	0.538	0.362	306.594	254.458
3.10	1.267	0.999	0.255	0.796	302.398	247.154
3.11	1.146	1.056	0.273	0.536	206.311	189.582

不同选项的性能

使用不同选项的测试，为了便于比较，每一个选项尽可能的都是单独使用，除非特别情况必须组合使用。

例如，选项*--no-wrap* 的测试如下:

```
$ rm -rf dist __pycache__
$ pyarmor testben.py

$ pyarmor gen --no-wrap testben.py benchmark.py benchmark2.py
$ pyarmor dist/testben.py
```

(续下页)

(接上页)

```
Python 3.7
test_import     :      0.971 ms
test_foo        :    306.261 ms
```

表 13: 表-4. 不同选项对性能和安全性的影响

选项	性能影响	安全性
--no-wrap	增加性能	降低安全性
--obf-module 0	对性能影响不大	轻微降低安全性
--obf-code 0	显著增加性能	显著降低安全性
--obf-code 2	降低性能	显著增加安全性
--enable-rft	基本不影响性能	增强安全性
--enable-themida	显著降低性能	显著提高安全性，能有效防止反调试工具
--mix-str	降低性能	提高安全性，主要是保护字符串常量
--assert-call	降低性能	提高安全性，主要是防止注入和 Monkey Trick
--assert-import	轻微降低性能	提高安全性，主要是防止注入和 Monkey Trice
--private	降低性能	提高安全性，主要是防止模块属性被外部脚本读取
--restrict	降低性能	提高安全性，主要是防止模块属性被外部脚本读取

3.4.7 本地化和国际化的工作原理

加密过程

查找本地配置目录下面的 `messages.cfg` 没有则查找全局配置目录下面的 `messages.cfg`

如果存在定制的消息文件，那么把定制的消息存到运行辅助包里面

打开消息文件的编码方式默认是 `utf-8` ，如果需要使用其他编码方式，使用下面的命令进行配置:

```
$ pyarmor cfg messages=messages.cfg:gbk
```

参见:

表-2. 运行加密脚本的错误信息表（Pyarmor）

运行加密脚本的过程

确定默认语言

- 首先查看 *PYARMOR_LANG*
- 其次查看 *LANG*

查找定制的错误信息，首先匹配语言，然后匹配错误代码

没有找到则使用默认的错误信息

3.5 许可模式和许可证

内容

3.5.1 简介

本文档仅适用于 Pyarmor 8.0 之后的版本。

下载和安装本软件表示自动接受试用许可协议，试用版本有如下的限制

(1) 加密功能对脚本大小有限制，不能加密超过限制的大脚本。

(2) 混淆字符串功能在试用版中无法使用。

(3) RFT 加密模式，BCC 加密模式在试用版无法使用。

(4) 可以使用本软件加密非商业用途的脚本，未经许可不得用于商业用途。

(5) 运行辅助包的名称”pyarmor_runtime_000000” 不可以被设置和修改

(6) 不可以使用本软件提供任何形式的加密服务，不管是通过应用程序还是网络服务。

(7) 不支持 obf-code 大于 1 的任何加密模式

试用版本中功能限制，需要通过许可授权来解锁相关功能。

3.5.2 许可模式

许可授权需要通过购买相应的许可证来获取，购买许可证可以通过指定的网站购买。

本软件提供三种许可证，分别解锁不同的功能

- 基础版许可证

  基础版许可证解锁限制 (1) (2) (4) (5) (7).

  加密脚本的时候需要在线验证许可证

- 专家版许可证

  专家版许可证解锁限制 (1) (2) (3) (4) (5) (7).

  加密脚本的时候需要在线验证许可证

- 集团版许可证

  集团版许可证解锁限制 (1) (2) (3) (4) (5) (7).

  加密脚本不需要在线验证许可证

不管哪一种许可证，运行加密脚本的时候都无需验证许可证，本软件对于加密脚本的运行没有任何控制和限制。

每一个许可证都有一个 18 位字符长度唯一的编号，并授权给有且只有一种产品使用。也就是说，任何一种使用本软件进行保护的产品都有自己唯一的许可证编号，不允许两种不同产品使用相同的许可证编号。

如果用户有多种产品并且已经为第一种产品购买了许可证，那么当用户的第二种产品的销售收入小于当前许可证费用的 10 倍，那么第二种产品可以暂时使用第一种产品的许可证。一旦销售收入超过许可证费用的 10 倍，第二种产品就必须购买自己的许可证。这个原则也适用于用户的其他所有产品。

一种产品在本协议中指的是独立销售的软件所有组成部分，包括开发需要的各种设备，以及提供支持的服务器，云服务器等。一种产品也包括产品的当前版本，历史版本，以及将来的升级版本。一种产品也包括基础功能相同，组合不同特殊功能而形成的不同版本的产品，这种产品的特征是不同版本对外销售名称一样，只是通过辅助名称等来进行区分。

同一个许可下面，同时使用本软件的设备数目不超过 100 个。同时使用本软件是指从现在开始在 24 小时内曾经运行 pyarmor 命令的设备。这里的设备是指安装 Pyarmor 并使用 Pyarmor 对脚本进行加密的设备，不是指运行加密脚本的客户机器。

关于许可的更详细说明，请阅读 Pyarmor 最终用户许可协议

关于一种产品的示例说明

不用于销售的所有的 Python 脚本都属于一种特殊的产品 `non-profits`

Pyarmor 是一种产品，它所包含

- Pyarmor 基础版，专家版和集团版都属于 Pyarmor 这一种产品
- pyarmor-webui，为 Pyarmor 提供图形界面的工具，也属于 Pyarmor 这一种产品

- Pyarmor 的后台订单系统是用 django 开发的一个程序，这也属于 Pyarmor 这一种产品
- 开发 Pyarmor 所使用的笔记本电脑，测试 Pyarmor 使用的台式机，以及运行后台订单系统的云服务器都属于 Pyarmor 这一种产品
- Pyarmor 7.x 和 Pyarmor 8.x 也都属于 Pyarmor 一种产品。

Microsoft Office 产品系列不是一种产品，它包括的各个产品，例如 Microsoft Word 和 Microsoft Excel 是功能完全不同的两个产品，所以 Microsoft Office 不是一种产品。而 Microsoft Word 是一种产品，它的各个版本系列 Micorsoft Word 2003，Word 2007 等也都属于 Microsoft word 这一种产品。

不同许可证的功能列表

表 14: 表-1. 许可证功能表

功能	试用版	基础版	专家版	集团版	备注说明
基本加密功能	有	有	有	有	[1]
脚本有效期	有	有	有	有	[2]
绑定到设备	有	有	有	有	[3]
JIT 保护	有	有	有	有	[4]
Themedia 保护	有	有	有	有	[5]
Assert 保护	有	有	有	有	[6]
大脚本文件	无	有	有	有	[7]
混淆字符串	无	有	有	有	[8]
obf-code > 1	无	有	有	有	[9]
RFT 加密模式	无	无	有	有	[10]
BCC 加密模式	无	无	有	有	[11]

[1] 基本加密功能，是指没有使用任何选项的加密功能。

[2] 脚本有效期，是指能够限制加密脚本运行有效期的功能。

[3] 绑定到设备，是指能够限制加密脚本运行在指定设备的功能。

[4] JIT 保护，是指使用动态代码生成机制对加密脚本进行保护的功能。

[5] Themedia 保护，是指使用第三方工具 Themedia 对 Widnows 动态库进行保护的功能。

[6] Assert 保护，是指保护加密脚本不会被替换或者非法注入的保护功能。

[7] 大脚本文件，是指加密脚本的大小超过一定值，“无”表示不能加密大脚本，“有”表示可以加密大脚本。

[8] 混淆字符串，是指对脚本中的字符串常量进行混淆保护的功能。

[9] 从 Pyarmor 8.2 开始，obf-code 支持使用多种不同方式对函数进行加密

[10] RFT 加密模式，是指通过重命名脚本中的函数，类，方法和变量的名称来保护脚本的功能。

[11] BCC 加密模式，是指把 Python 脚本中部分函数转换成为对应的 C 函数，通过编译直接生成机器指令代码，从而对脚本进行保护的功能。

notes

3.5.3 购买

在浏览器中打开 Pyarmor 官网的购物车，支持微信和支付宝

https://pyarmor.dashingsoft.com/cart/order.html

在购物车页面选择需要的许可证类型，填写注册名称，并完成支付。

支付成功之后，在一个工作日之内激活文件会发送到注册邮箱，请按照激活文件中的方法和步骤完成注册和激活，或者参考这里注册和使用许可证 。

购买软件许可的费用是一次性收费，可以永久在购买本软件时候的版本中使用，但是许可证可能在任何一个升级版本中失效，Pyarmor 不承诺许可证可以在今后所有的升级版本中使用。

表 15: 表-2. 不同授权模式的价格列表（中国）

授权模式	不含税价格（人民币元）	含税价格（人民币元）	说明
基础版	298	359	
专家版	512	562	
集团版	868	918	

如果需要使用信用卡或者 Paypal 进行支付，可以直接在 MyCommerce 官网进行购买，这里的价格单位是美元，并且不能开国内增值税发票，只提供国际通用的电子发票（订单，发票相关事宜全部由 MyCommerce 处理）

https://order.mycommerce.com/product?vendorid=200089125&productid=301044051

表 16: 表-3. 不同授权模式的价格列表（国外）

授权模式	不含税价格（美元）	说明
基础版	52	
专家版	89	
集团版	158	

3.5.4 升级老版本许可证

不是所有的老版本的许可证都可以升级为新的许可证。

符合下列条件的老版本许可证可以免费升级到 Pyarmor 基础版许可证:

- 遵循新的 Pyarmor 最终用户许可协议
- 原来的许可证编号是以 `pyarmor-vax-` 开头的
- 原来许可证的注册文件 `pyarmor-regcode-xxxx.txt` 存在且不能被使用超过 100 次

- 原来的许可证的购买日期在 2023 年 6 月 1 日之前，原则上在 Pyarmor 8 发布之后依旧购买的老许可证不支持升级。

如果无法免费升级，请购买新的许可证。

老版本的许可证不支持升级到专家版和集团版。

免费升级到基础版

首先找到原来的许可证激活文件 `pyarmor-regcode-xxxx.txt`

然后安装 Pyarmor 8.2+

按照新的 Pyarmor 最终用户许可协议，需要为每一个许可证指定产品名称。这也意味着，如果老的许可证是被用于多种产品的话，升级之后就只能用于其中的一个，其他产品还需要购买新的许可证。

假定使用许可证的产品名称是 `XXX`，那么使用下面的命令进行升级:

```
$ pyarmor reg -u -p "XXX" pyarmor-regcode-xxxx.txt
```

升级成功之后会生成新的注册文件

在其他设备直接使用新的注册文件 进行注册:

```
$ pyarmor reg pyarmor-regfile-xxxx.zip
```

运行下面的命令检查升级后的许可证:

```
$ pyarmor -v
```

注册成功之后所有的加密操作自动应用当前许可证，每一次加密操作需要联网验证许可证。

3.6 常见问题

3.6.1 如何在 Github 上提问

Pyarmor 是一个命令行工具，提供了丰富的选项来应对不同的情况。对于简单的应用，使用默认选项就可以工作，不需要学习如何使用 Pyarmor。但是对于一些高级的功能，或者比较复杂的应用，尤其是需要使用大型的第三方包来调用加密脚本，用户需要一定的时间去学习和使用各种选项，并组合这些选项去解决加密过程中问题。

当加密脚本出现问题的时候，这不一定是 Pyarmor 的 Bug，很可能是这个脚本使用默认的选项和配置无法工作。这种情况下需要分析问题并找到 Pyarmor 解决这个问题的选项和设置，这就需要用户参考 Pyarmor 的命令手册去找到合适的选项。

Pyarmor Team 不会告诉用户需要使用什么选项去解决加密过程中遇到的问题，用户需要自己学习 Pyarmor 的各个选项如何使用，并根据自己的需要选择正确的选项

Pyarmor 有完整的文档系统，所有的功能和选项都已经在文档中描述，Pyarmor Team 能告诉你的，这些文档中都已经有说明。

在提问之前，请首先尝试下面的解决方案，这应该能够解决大部分的问题，并且避免了提交重复的问题:

- 如果使用的是命令 **pyarmor-7** 或者 Pyarmor < 8.0，请参阅 Pyarmor 7.x 在线文档
- 浏览一下文档总目录
- 如果还没有读过入门教程，首先把入门教程读一下，特别是最后面的一章，文档的组织结构和内容说明
- 参阅常见错误信息 里面的错误信息和解决方案
- 如果是使用 pack 方面的问题，请参阅详解可独立运行的加密脚本
- 如果是使用*RFT 模式* 的问题，请参阅使用 *RFT 模式*加密脚本 *pro*
- 如果是使用*BCC 模式* 的问题，请参阅使用 *BCC 模式*加密脚本 *pro*
- 如果是第三库无法正常调用加密脚本，请参阅如何解决第三方库调用加密脚本存在的问题
- 如果是和安全性能相关的问题，请参阅性能和安全
- 浏览一下本页后面的内容
- 运行没有加密的脚本，确保没有加密之前的脚本能够正确的运行
- 如果加密的是复杂的应用，请首先测试一个简单的脚本，确认它使用同样的加密选项能够工作
- 启用调试模式，跟踪日志，在控制台输出更多信息，仔细检查日志信息，确认问题所在
- 如果使用的不是最新版本的 Pyarmor，请升级到最新版本
- 搜索一下 问题报告
- 搜索一下 讨论区

如果你不想花费时间去阅读文档，请在 讨论区 提问，但是 Pyarmor Team 并不保证一定回答这部分的所有问题。

如果你发现 Pyarmor 中存在 Bug，请提交到 问题报告

Pyarmor Team 欢迎真正的 Bug 报告，并且会尽快解决这些 Bug。报告 Bug 请提供必要的信息以便 Pyarmor Team 能够重现问题，这样有助于快速解决问题，无法被 Pyarmor Team 重现的问题解决起来需要更长的周期。运行*pyarmor gen* 命令的时候，控制台的前四行包含加密使用的选项，Pyarmor 的版本信息，Python 的版本信息，以及当前的运行平台，请务必以文本的形式拷贝这四行到 Bug 报告中。对于无法重现的 Bug 报告，Pyarmor Team 会把它转入到讨论区，以后可以重现的时候，在转回到 Bug 并继续处理。为了方便处理 Bug，请尽量不要使用截图，而是直接拷贝文本。例如，一般报告应该以这样的方式开始:

````
**加密使用的选项和命令**
```
$ pyarmor cfg wrap_mode 1
$ pyarmor gen --assert-call foo.py
````

(续下页)
````

(接上页)

```
INFO     Python 3.9.0
INFO     Pyarmor 8.1.6 (pro), 005068, boot armor
INFO     Platform darwin.x86_64
```

3.6.2 热点问题

看到网上有破解 Pyarmor 的工具？请问 Pyarmor 能够防止这些破解工具吗？

Pyarmor 一般不会关注和了解这些破解工具，而是致力于研究 CPython 的源代码，以改进算法来提高安全性，所以没有办法直接回答这类问题。

可以确定的是，Pyarmor 提供了多种不可逆的加密模式，从原理上来说，是不可能把加密脚本恢复成为原来的脚本。

请参考最高安全性和最快性能，使用你可用的最高安全选项去加密一个简单的参考脚本，例如

```
import sys

def fib(n):
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

print('python version:', sys.version_info[:2])
print('this is fib(10)', fib(10))
```

然后尝试使用网上所说的方法和工具进行破解。

如果能够被破解，请把 Python 版本，Pyarmor 的版本，运行的平台，加密使用的选项，参考脚本以及破解的方法发送到 pyarmor@163.com

请不要在 Pyarmor 上发布任何破解工具的链接。

Pyarmor 的核心功能是防止加密脚本被还原，对使用各种逆向工程方法的内存拷贝，直接修改内存绕过加密脚本的约束设置并没有特别的保护。关于如何对运行数据进行保护的解决方案，请参考保护运行时刻的数据安全性

3.6.3 Apple 上的 Segment fault

1. 通常情况下，这是因为不正确的 code signature

如果是加密或者注册的时候发生了崩溃，请尝试对扩展模块 `pytransform3.so` 重新签名:

```
$ codesign -s - -f /path/to/lib/pythonX.Y/site-packages/pyarmor/cli/core/pytransform3.
↪so
```

如果是运行加密脚本的时候发生了崩溃，请尝试对扩展模块 `pyarmor_runtime.so` 重新签名:

```
$ codesign -s - -f dist/pyarmor_runtime_000000/pyarmor_runtime.so
```

请参阅 Apple 官方文档 Using the latest code signature format

2. 使用 otool 和 install_name_tool 解决依赖库问题

因为预编译的扩展模块需要一些依赖库，如果依赖库的位置不对，就可能直接崩溃。使用 `otool -L` 可以查看依赖库:

```
$ otool -L /path/to/lib/pythonX.Y/site-packages/pyarmor/cli/core/pytransform3.so

/path/to/lib/pythonX.Y/site-packages/pyarmor/cli/core/pytransform3.so:
    pytransform3.so (compatibility version 0.0.0, current version 1.0.0)
    @rpath/lib/libpython3.9.dylib (compatibility version 3.9.0, current version 3.9.0)
    ...
```

除了系统库之外，主要就是 Python 动态库 `@rpath/lib/libpython3.9.dylib`，其中默认配置的 `rpath` 为:

```
$ install_name_tool -id pytrnsform3.so \
        -change $deplib @rpath/lib/libpython$ver.dylib \
        -add_rpath @executable_path/.. \
        -add_rpath @loader_path/.. \
        -add_rpath /System/Library/Frameworks/Python.framework/Versions/$ver \
        -add_rpath /Library/Frameworks/Python.framework/Versions/$ver \
        build/$host/libs/cp$ver/$name.so
```

也可以使用下面的命令查看 `rpath`:

```
$ otool -l /path/to/lib/pythonX.Y/site-packages/pyarmor/cli/core/pytransform3.so
```

检查当前系统是否存在 `@rpath/lib/libpython3.9.dylib`，如果不存在这个文件的话，需要使用 `install_name_tool` 适配当前的 Python 安装环境，假设 Python 动态库是 `/usr/local/Python.framework/Versions/3.9/Python`:

```
$ install_name_tool -change @rpath/lib/libpython3.9.dylib /usr/local/Python.framework/
↪Versions/3.9/Python \
        /path/to/lib/pythonX.Y/site-packages/pyarmor/cli/core/pytransform3.so
```

对于 `dist/pyarmor_runtime_000000/pyarmor_runtime.so` 也是同样的，必须保证依赖库都存在，否则需要进行适配运行环境。

如何找到当前 Python 解释器对应的动态库，请自行搜索答案。注意有些预编译的 Python 没有使用动态库，那么是无法运行加密脚本的，需要重新编译支持动态库的版本。

请参阅 Apple 官方文档 Run-Path Dependent Libraries

3. 如果系统安装了多个 Python，确保链接到正确的动态库

4. 权限设置问题

Pyarmor 使用了 JIT 技术来提高安全性，在 Apple M1，这可能需要对 Python 进行某些设置。使用下面的命令检查 Python 的 entitlements 并进行必要的设置:

```
$ codesign -d --entitlements - $(which python)
```

请参阅 Apple 官方文档 Allow Execution of JIT-compiled Code Entitlement

5. 最后看一下 Apple 的 segment fault 日志，根据提示的错误信息搜索网络找解决方案

3.6.4 使用许可相关问题

我们采购流程需要双方签一个合同，请问是否支持呢？

Pyarmor 是一个工具产品，不额外签订其他合同，Pyarmor 最终用户许可协议 就是合同文件。

你们提到一个许可证授权给有且只有一个产品使用，表示的是我只能用来加密一个脚本吗？

> 你可以把“一个产品”替换成为“一种产品”来进行理解，一种产品指的是独立销售的软件所有组成部分，包括开发需要的各种设备，以及提供支持的服务器，云服务器等。一种产品也包括产品的当前版本，历史版本，以及将来的升级版本。一种产品也包括基础功能相同，组合不同特殊功能而形成的不同版本的产品，这种产品的特征是不同版本对外销售名称一样，只是通过辅助名称等来进行区分。
>
> 详细说明和例子请参考许可模式和许可证

是否可以对大文件提供试用许可证

> 不提供。
>
> Pyarmor 是一个小工具，大部分的功能在试用版中均可以得到验证。对于一些无法试用的高级功能，主要是 RFT 模式和 BCC 模式，这些模式都可以通过配置忽略把不支持的部分脚本或者函数进行忽略。也就是说，只要通过额外的配置，它们总是可以工作的。

我们有自己的 CICD，需要在镜像内对代码进行加密，那么这是否意味着我们在每次打镜像时都需要通过“pyarmor register pyarmor-regcode-xxxx.txt”来对 pyarmor 进行注册？我们的 CICD 会触发的比较频繁，那么如果比较频繁的进行 register 操作的话是否会影响 pyarmor 的使用?

激活文件 yarmor-regcode-xxxx.txt 一般只用于第一次的注册登记，使用超过十次之后就无法在进行注册。在 CICD 中新镜像使用的时候需要注册，但是不能使用这个文件，可以使用第一次注册登记时候生成的一个.zip 格式的注册文件。

具体使用方法请参考注册和使用许可证 里面的注册许可证中的说明

CHAPTER 4

索引表

- 总索引
- 模块索引
- 搜索

Python 模块索引

p

索引

符号

B

C

J

L

P

R