|  |
| --- |
| BAN CƠ YẾU CHÍNH PHỦ  **HỌC VIỆN KỸ THUẬT MẬT MÃ**  Logo HvKTMM  BÁO CÁO MÔN HỌC  **KỸ THUẬT LẬP TRÌNH**  **Đề tài: QUẢN LÝ LOGGING TRONG HỆ THỐNG MẠNG**  Nhóm sinh viên thực hiện:  **Nguyễn Quang Trung – AT150658**  **Nguyễn Đăng Long – AT150236**  **Hoàng Dương Tùng – AT160357**  **Dương Minh Nghĩa – AT160338**  **Nguyễn Thị Thùy Trang – AT160449**    Người hướng dẫn:  **KS. Nguyễn Mạnh Thắng**  Khoa An toàn thông tin – Học viện Kỹ thuật mật mã  ***Hà Nội - 2023*** |

**MỤC LỤC**

[DANH MỤC HÌNH ẢNH 4](#_Toc129019652)

[DANH MỤC BẢNG BIỂU 5](#_Toc129019653)

[LỜI NÓI ĐẦU 6](#_Toc129019654)

[CHƯƠNG 1: GIỚI THIỆU CHUNG VỀ LOGGING 7](#_Toc129019655)

[1.1. Giới về thiệu logging 7](#_Toc129019656)

[1.2. Logging trong hệ thống 7](#_Toc129019657)

[1.3. Các module trong logging 8](#_Toc129019658)

[1.3.1. Module logging 8](#_Toc129019659)

[1.3.2. Module logging.config 13](#_Toc129019660)

[1.3.3. Module logging.handlers 16](#_Toc129019661)

[CHƯƠNG 2: CÁC KIẾN THỨC VỀ LOGGING 21](#_Toc129019662)

[2.1. Các kiến thức cơ bản về logging 21](#_Toc129019663)

[2.1.1. Các cấp độ của python logging 21](#_Toc129019664)

[2.1.2. Logging vào trong file 23](#_Toc129019665)

[2.1.3. Logging từ nhiều module khác 24](#_Toc129019666)

[2.1.4. Biến dữ liệu trong logging 25](#_Toc129019667)

[2.1.5. Thay đổi định dạng hiển thị thông báo 26](#_Toc129019668)

[2.1.6. Hiển thị thời gian của thông báo 27](#_Toc129019669)

[2.2. Kiến thức nâng cao về logging 28](#_Toc129019670)

[2.2.1. Tạo trình ghi nhật ký (logger) 29](#_Toc129019671)

[2.2.2. Tạo trình ghi nhật ký với trình xử lý (handler) và trình định dạng (formatter) 29](#_Toc129019672)

[2.2.3. Tải và lưu cấu hình logging 31](#_Toc129019673)

[CHƯƠNG 3: THỰC NGHIỆM 34](#_Toc129019674)

[3.1. Giới thiệu 34](#_Toc129019675)

[3.1.1. Mô tả 34](#_Toc129019676)

[3.1.2. Yêu cầu 34](#_Toc129019677)

[3.1.3. Mục tiêu 35](#_Toc129019678)

[3.2. Triển khai, thử nghiệm 35](#_Toc129019679)

[3.2.1. Cài đặt 35](#_Toc129019680)

[3.2.2. Xây dựng 35](#_Toc129019681)

[3.2.3. Thực nghiệm 43](#_Toc129019682)

[KẾT LUẬN 44](#_Toc129019683)

[**TÀI LIỆU THAM KHẢO** 45](#_Toc129019684)

# DANH MỤC HÌNH ẢNH

[*Hình 1. 1 Module Logging 2*](#_Toc128696434)

[*Hình 3. 1 Mô hình mạng kểt nối của hai máy Kali và Ubuntu.......................... 25*](#_Toc128690688)

# DANH MỤC BẢNG BIỂU

[*Bảng 1.1 Bảng ưu tiên trong syslogHandler* 13](#_Toc128696521)

[*Bảng 1.2 Bảng cơ sở trong syslogHandler* 14](#_Toc128696522)

[*Bảng 2. 1 Bảng các cấp độ trong logging......................................................... 16*](#_Toc128690054)

[*Bảng 2. 2 Bảng thuộc tính xuất hiện trong chuỗi format 19*](#_Toc128690055)

# LỜI NÓI ĐẦU

Trong quá trình vận hành và sử dụng phần mềm, “phần mềm” có lúc vận hành tốt có lúc không, và để “phần mềm” thông báo trạng thái hoạt động của mình với người dùng, thì lập trình viên thường trong quá trình xây dựng và phát triển phần mềm, các lập trình viên sẽ đặt các lệnh try...except... tại các vị trí có thể gây ra lỗi để cảnh báo cho người dùng được biết về các lỗi xảy ra hoặc thông báo cho người dùng các vấn đề cần lưu ý.

Để muốn kiểm tra được lỗi của phần mềm thì một số lập trình viên có thói quen đặt câu lệnh print để hiện ra màn hình console các thông tin trong quá trình ứng dụng chạy để xử lý lỗi trước khi hoàn thành. Nhưng ngày nay giải pháp hay được mọi người sử dụng là ghi lỗi vào log file để ghi lại những lỗi, thông tin hay dữ liệu cần lưu ý trong quá trình sử dụng phần mềm hoặc giao thức nào đó. Cụ thể trong báo cáo này sử dụng các module logging. Đây là module tiêu chuẩn (được pack kèm bộ cài mặc định) do cộng đồng Python phát triển, dễ dàng sử dụng và phát triển.

Trong quá trình vận hành và sử dụng, phần mềm không phải lúc nào cũng hoạt động trơn tru. Khi đó phải tìm cách để phần mềm thông báo trạng thái hoạt động của mình với lập trình viên, người dùng. Ở môi trường develop ta hoàn toàn có thể sử dụng các công cụ mà IDE hỗ trợ để debug. Nhưng ở môi trường product thì không thể tái hiện được lỗi xảy ra trong quá trình vận hành. Để kiểm tra, một số lập trình viên có thói quen đặt lệnh in trên màn hình console kèm thông tin trong khi chạy ứng dụng để "gỡ lỗi trong khi chạy" (debug while running) trước khi xuất bản, sau đó thực hiện chuyển lệnh để comment ra ngoài hoặc xóa các dòng lệnh in đó, điều này cần thời gian để thực hiện và nếu xảy ra sự cố thì sẽ tiêu tốn rất nhiều thời gian để sửa lỗi. Giải pháp thường được sử dụng nhất là ghi lỗi vào tệp nhật ký (Log file). Cụ thể trong báo cáo này sử dụng module Logging trong Python. Đây là một module chuẩn (đi kèm với bộ cài đặt mặc định) dễ sử dụng và được phát triển bởi cộng đồng Python.

# GIỚI THIỆU CHUNG VỀ LOGGING

## Giới về thiệu logging

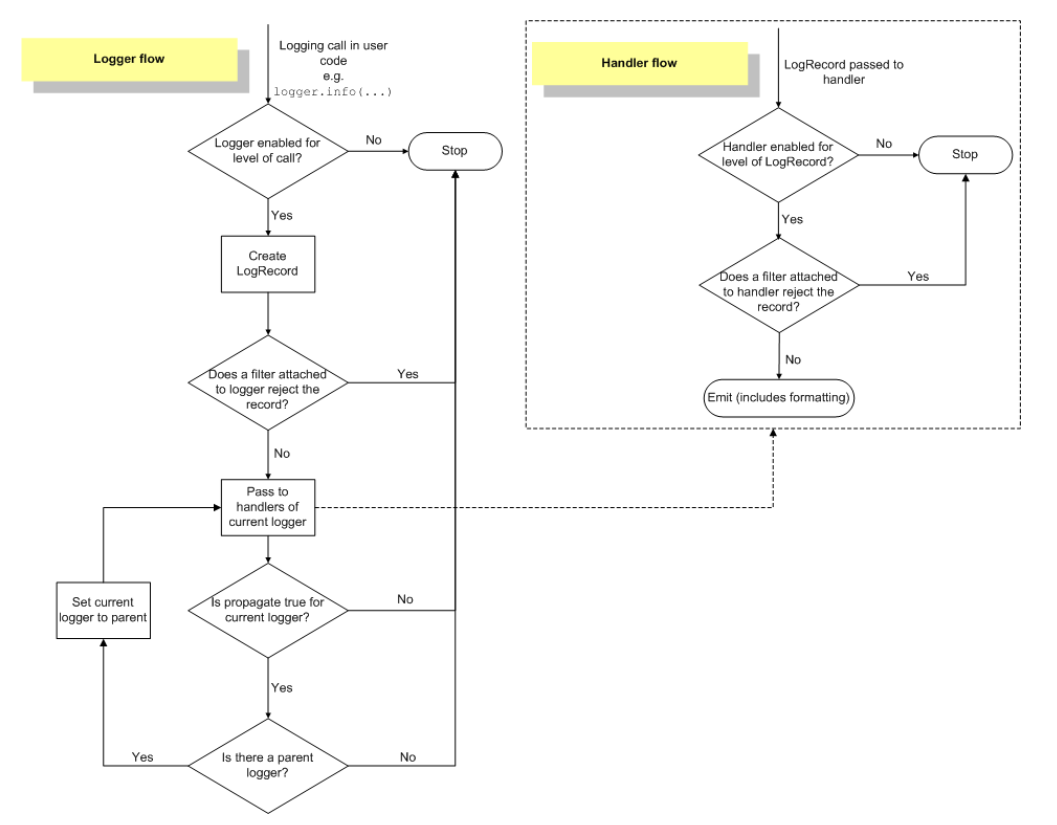
Logging là một yêu cầu không thể thiếu của mọi chương trình phần mềm khi sử dụng trong môi trường production. Với logging, có nghĩa là theo dõi những sự kiện xảy ra khi phần mềm chạy. Những lập trình viên thêm lời gọi logging vào code để hiển thị những sự kiện đã xảy ra, có thể theo dõi logging để giám sát hoạt động của phần mềm theo thời gian thực (real-time). Những sự kiện được mô tả bởi thông điệp có thể bao gồm dữ liệu (dữ liệu khác nhau cho mỗi lần xảy ra khác nhau). Sự kiện cũng có độ nghiêm trọng tùy theo nhà phát triển và có thể đưa ra các hành động cần thiết khi xảy ra lỗi.

Logging thường được ưu tiên để ghi vào các lỗi nhưng có những hệ thống đặc biệt, ngoài log lỗi thì log file còn có thể chứa các thông tin liên quan đến các action của người dùng vào hệ thống. Ví dụ: Hệ thống fintech liên quan đến thanh toán giao dịch tiền, toàn bộ các action của người dùng trên phần mềm (hoặc web-app) đều cần phải ghi vào log:

* Khi người dùng đăng nhập vào --> Ghi log lại để có thể theo dõi thói quen của người dùng. Từ đó có thể trích xuất ra thông tin trong thời gian từ mấy giờ đến mấy giờ người dùng sẽ sử dụng ứng dụng nhiều.
* Khi thực hiện đặt lệnh thanh toán tiền --> Ghi log lại với thông tin của khác hàng + thông tin đích đến + thông tin số tiền để thực hiện đối soát dữ liệu mỗi ngày

## Logging trong hệ thống

Luồng sự kiện logging trong quá trình ghi và xử lý được mịnh họa trong hình dưới đây:



*Hình 1. 1 Module Logging*

## Các module trong logging

### Module logging

#### Khái niệm

Module này xác định chức năng và lớp thực hiện một hệ thống ghi lại sự kiện logging cho ứng dụng và thư viện. Lợi ích của việc có API Logging được cung cấp thử viện module tiêu chuẩn là tất các module Python có thể tham gia được vào việc ghi logging, vì vậy ứng dụng log có thể bao gồm các thông báo riêng được tích hợp với các thông báo từ các module từ bên thứ ba.

Ví dụ các lớp cơ bản được định nghĩa bởi module, cùng với các chứ năng bên dưới:

* Trình xử lý gửi các bản ghi log (được tạo bởi người ghi log) đến đích thích hợp
* Bộ lọc cung cấp một cơ sở chi tiết để xác định bản ghi log nào xuất ra
* Bộ định dạng chỉ định cách bố trí các bản ghi nhật ký trong kết quả cuối cùng

#### Đối tượng logger

Bộ ghi logger không bao giờ dược khởi tạo trực tiếp, mà luôn thông qua các hàm cấp module logging.getLogger(\_name\_). Gọi đến getLogger() nhiều lần có cùng tên sẽ luôn tham chiếu đến cùng một đối tượng logger và \_name\_ có khả năng là 1 giá trị phân cấp được tách bằng dấu châm như foo.bar.baz

Một số Class của logging.Logger:

* *Propagate*: Nếu thuộc tính này có giá trị là true, các sự kiện được ghi vào bộ ghi này sẽ được chuyển đến bộ xử lý của bộ ghi cấp cao hơn, ngoài bất kỳ bộ xử lý nào được gắn vào bộ ghi này. Thông báo được chuyển trực tiếp đến trình xử lý của trình ghi logger cao hơn. Nếu là false, thì thông báo ghi logging sẽ không được chuyển đến bộ xử lý của bộ ghi logger cao hơn.
* *setLevel*: Đặt độ ưu tiên logger. Thông báo logging có cấp độ thấp hơn sẽ bị bỏ qua, thông báo logging có có mức độ nghiêm trọng hoặc cao hơn sẽ được phát ra bởi bất kỳ trình xử lý hoặc trình xử lý nào phục vụ trình ghi nhật ký này, trừ khi mức độ của trình xử lý đã được đặt thành mức độ nghiêm trọng cao hơn cấp độ.
* *debug*(*msg*, *\*args*, *\*\*kwargs*): Ghi một thông báo với mức debug trên trình ghi này. Msg là chuỗi định dạng thông báo và args là các đối số được hợp nhất thành msg bằng cách sử dụng toán tử định dạng chuỗi.
* *infor*(*msg*, *\*args*, *\*\*kwargs*)Ghi một thông báo với mức INFO trên trình ghi này. Các đối số được hiểu là đối với debug ().
* *warning*(*msg*, *\*args*, *\*\*kwargs*): Ghi một thông báo với mức WARNING trên trình ghi này. Các đối số được hiểu là đối với debug ().
* *error*(*msg*, *\*args*, *\*kwargs*): Ghi một thông báo với mức ERROR trên trình ghi này. Các đối số được hiểu là đối với debug ().
* *critical*(*msg*, *\*args*, *\*kwargs*): Ghi một thông báo với mức CRITICAL trên trình ghi này. Các đối số được hiểu là đối với debug ().
* *Exception*(*msg*, *\*args*, *\*kwargs*): Ghi lại một thông báo với mức ERROR trên trình ghi này. Các đối số được hiểu là đối với debug (). Thông tin ngoại lệ được thêm vào thông báo ghi logging. Phương thức này chỉ nên được gọi từ một trình xử lý ngoại lệ.

#### Đối tượng Handler

Handler có các thuộc tính và phương thức sau. Lưu ý rằng Handler không bao giờ được khởi tạo trực tiếp; lớp này hoạt động như một cơ sở cho các lớp con. Tuy nhiên, phương thức \_init \_() trong các lớp con cần phải gọi Trình xử lý. \_init \_().

* + Ví dụ một số class của longging.hanlder
* *\_init\_(level=NOTSET*): Khởi tạo thể hiện Hanlder bằng cách đặt mức của nó, đặt danh sách bộ lọc thành danh sách trống và tạo khóa (sử dụng createLock ()) để tuần tự hóa quyền truy cập vào cơ chế I / O.
* *createLock*(): Khởi tạo khóa luồng có thể được sử dụng để tuần tự hóa quyền truy cập vào chức năng I / O bên dưới có thể không an toàn luồng.
* *acquire*(): Nhận khóa luồng được tạo bằng createLock ().
* *Release*(): Giải phóng khóa luồng có được với acquire().
* *Format*(record): Thực hiện định dạng cho một bản ghi - nếu một bộ định dạng được đặt, sử dụng nó. Nếu không, sử dụng trình định dạng mặc định cho module.
* *Emit*(record): Làm bất cứ điều gì cần thiết để thực sự ghi lại log được chỉ định.

#### Đối tượng Formatter

Formatter có các thuộc tính và phương thức sau. Nó chịu trách nhiệm chuyển đổi LogRecord thành (thường) là một chuỗi ký tự có thể được giải thích bởi con người hoặc hệ thống bên ngoài. Formatter cơ sở cho phép chỉ định một chuỗi định dạng. Nếu không có giá trị nào được cung cấp, giá trị mặc định của '% (message) s' sẽ được sử dụng, giá trị này chỉ bao gồm tin nhắn trong cuộc gọi ghi nhật ký. Để có các mục thông tin bổ sung trong đầu ra được định dạng (chẳng hạn như dấu thời gian), hãy tiếp tục đọc.

*class logging.Formatter*(*fmt=None*, *datefmt=None*, *style='%'*) là Trả về một bản mới của lớp Formatter. Bản này được khởi tạo bằng một chuỗi định dạng cho toàn bộ thông báo, cũng như một chuỗi định dạng cho phần ngày / giờ của thông báo. Nếu không có fmt nào được chỉ định, thì '% (message) s' sẽ được sử dụng. Nếu không có datefmt nào được chỉ định, một định dạng được sử dụng được mô tả trong tài liệu formatTime (). Tham số kiểu có thể là một trong số ‘%’, ‘{‘ hoặc ‘$’ và xác định cách chuỗi định dạng sẽ được hợp nhất với dữ liệu của nó: sử dụng một trong các% -formatting, str.format () hoặc string.Template. Xem Sử dụng các kiểu định dạng cụ thể trong toàn bộ ứng dụng của bạn để biết thêm thông tin về cách sử dụng {- và $ -formatting cho thông báo log.

#### Đối tượng filter

Có thể được sử dụng bởi Người xử lý và Người ghi log để lọc phức tạp hơn mức được cung cấp bởi các cấp. Lớp bộ lọc cơ sở chỉ cho phép các sự kiện nằm dưới một điểm nhất định trong phân cấp trình ghi log. Ví dụ: một bộ lọc được khởi tạo bằng 'A.B' sẽ cho phép các sự kiện được ghi lại bởi các trình ghi 'A.B', 'A.B.C', 'A.B.C.D', 'A.B.D', v.v. nhưng không cho phép 'A.BB', 'B.A. B 'vv Nếu được khởi tạo bằng chuỗi trống, tất cả các sự kiện đều được chuyển.

*Class logging.filter(name=’’)* là Trả về một thể hiện của lớp Bộ lọc. Nếu tên được chỉ định, nó đặt tên cho một trình ghi nhật ký, cùng với các con của nó, sẽ cho phép các sự kiện của nó thông qua bộ lọc. Nếu tên là chuỗi trống, cho phép mọi sự kiện.

*filter(*record*)* là log được chỉ định có được ghi lại không? Trả về 0 cho không, khác 0 cho có. Nếu thấy phù hợp, hồ sơ có thể được sửa đổi tại chỗ bằng phương pháp này.

#### Đối tượng logrecord

Các thể hiện LogRecord được Logger tạo tự động mỗi khi có nội dung nào đó được ghi nhật ký và có thể được tạo thủ công thông qua makeLogRecord () (ví dụ: từ một sự kiện đã nhận qua dây).

*Class* logging*.LongRecord(name, level, pathname, msg,lineno , msg, arg,exc\_info, func=None, sinfo=None)* là Thông tin chính được chuyển dưới dạng msg và args, được kết hợp bằng cách sử dụng msg% args để tạo trường thông báo của bản ghi.

*getMessage*() là Trả về thông báo cho phiên bản LogRecord này sau khi hợp nhất bất kỳ đối số nào do người dùng cung cấp với thông báo. Nếu đối số thông báo do người dùng cung cấp cho cuộc gọi logging không phải là một chuỗi, str () được gọi trên đó để chuyển đổi nó thành một chuỗi. Điều này cho phép sử dụng các lớp do người dùng định nghĩa làm thông báo, có phương thức \_\_str\_\_ có thể trả về chuỗi định dạng thực tế sẽ được sử dụng.

#### Đối tượng LoggerAdapter

Các thể hiện LoggerAdapter được sử dụng để chuyển thông tin ngữ cảnh vào các cuộc gọi ghi nhật ký một cách thuận tiện. Để biết ví dụ sử dụng, hãy xem phần thêm thông tin theo ngữ cảnh vào kết quả ghi nhật ký của bạn.

class logging.LoggerAdapter(logger, extra) là trả về một phiên bản LoggerAdapter được khởi tạo với một phiên bản Logger bên dưới

*process*(*msg*, *kwargs*) là Sửa đổi thông báo và / hoặc đối số từ khóa được chuyển đến lệnh gọi logging để chèn thông tin theo ngữ cảnh. Việc triển khai này đưa đối tượng được truyền dưới dạng bổ sung cho hàm tạo và thêm nó vào kwargs bằng cách sử dụng khóa ‘extra’. Giá trị trả về là một bộ giá trị (msg, kwargs) có các phiên bản (có thể được sửa đổi) của các đối số được truyền vào.

### Module logging.config

Để mô tả API để định cấu hình module logging.

#### Các chức năng cấu hình

Các các chức năng sau đây định cấu hình module logging. Chúng nằm trong module logging.config. Việc sử dụng chúng là tùy chọn – người dùng có thể định cấu hình module logging bằng chức năng này hoặc bằng cách thực hiện các lệnh gọi tới API chính (được định nghĩa trong chính việc logging) và xác định các trình xử lý được khai báo trong logging hoặc logging.handlers.

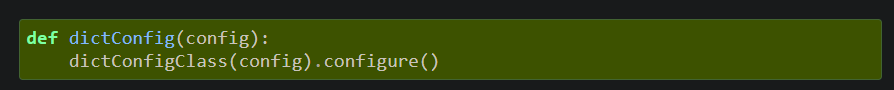
logging.config.dictConfig(config) là lấy logging configuration từ điển. Nội dung của từ điển này được mô tả trong lược đồ từ điển cấu hình bên dưới.

Nếu gặp lỗi trong khi cấu hình, chức năng này sẽ tạo ra một ValueError, TypeError, AttributeError hoặc ImportError với một thông báo mô tả phù hợp. Sau đây là danh sách (có thể không đầy đủ) các điều kiện sẽ gây ra lỗi:

Một mức không phải là một chuỗi hoặc một chuỗi không tương ứng với một mức logging thực tế.

* Giá trị truyền không phải là boolean.
* Một id không có đích tương ứng.
* Đã tìm thấy một id trình xử lý không tồn tại khi gọi tăng dần.
* Tên người ghi không hợp lệ.
* Không có khả năng phân giải đối tượng bên trong hoặc bên ngoài.

Việc phân tích cú pháp được thực hiện bởi lớp DictConfigurator, có hàm tạo được chuyển vào từ điển được sử dụng để cấu hình và có phương thức config (). Mô-đun logging.config có thuộc tính có thể gọi là dictConfigClass ban đầu được đặt thành DictConfigurator. Người dùng có thể thay thế giá trị của dictConfigClass bằng cách triển khai phù hợp của riêng.



DictConfig () gọi dictConfigClass truyền từ điển được chỉ định, sau đó gọi phương thức config () trên đối tượng trả về để đặt cấu hình có hiệu lực.

logging.config.fileConfig (fname, defaults = none, disable\_existing\_ \_logger = True) Đọc cấu hình logging từ tệp định dạng configparser. Định dạng của tệp phải được mô tả trong định dạng tệp cấu hình. Chức năng này có thể được gọi nhiều lần từ một ứng dụng, cho phép người dùng cuối chọn từ các cấu hình được soạn sẵn khác nhau (nếu nhà phát triển cung cấp cơ chế trình bày các lựa chọn và tải cấu hình đã chọn).

logging.config.listen (port = DEFAULT LOGGING\_CONFIG PORT, verify = none) là khởi động máy chủ socket trên cổng được chỉ định và lắng nghe các cấu hình mới. Nếu không có cổng nào được chỉ định thì DEFAULT\_LOGGING\_CONFIG\_PORT mặc định của module sẽ được sử dụng. Cấu hình logging sẽ được gửi dưới dạng tệp phù hợp để xử lý bởi dictConfig () hoặc tệp Config (). Trả về một cá thể Thread mà bạn có thể gọi start () để khởi động máy chủ và bạn có thể tham gia () khi thích hợp. Để dừng máy chủ, hãy gọi stop Listening ().

#### Cấu hình từ điển schema

Mô tả cấu hình logging yêu cầu liệt kê các đối tượng khác nhau để tạo và các kết nối giữa chúng; ví dụ: bạn có thể tạo một trình xử lý có tên là ‘console’ và sau đó nói rằng trình ghi nhật ký có tên là ‘startup’ sẽ gửi các thông điệp của nó tới trình xử lý ‘console’. Những đối tượng này không bị giới hạn đối với những đối tượng được cung cấp bởi module logging vì người dùng có thể viết lớp trình định dạng hoặc trình xử lý của riêng mình. Các tham số của các lớp này cũng có thể cần bao gồm các đối tượng bên ngoài như sys.stderr. Cú pháp để mô tả các đối tượng và kết nối này được định nghĩa trong Kết nối đối tượng bên dưới.

#### Định dạng cấu hình tệp

Định dạng tệp cấu hình được hiểu bởi fileConfig () dựa trên chức năng của configparser. Tệp phải chứa các phần được gọi là [loggers], [handlers] và [formatters] xác định theo tên các thực thể của từng loại được xác định trong tệp. Đối với mỗi thực thể như vậy, có một phần riêng biệt xác định cách thực thể đó được định cấu hình. Do đó, đối với một trình logger có tên là log01 trong phần [logger], các chi tiết cấu hình liên quan được lưu giữ trong một phần [logger\_log01]. Tương tự, một handler được gọi là hand01 trong phần [handler] sẽ có cấu hình của nó được giữ trong một phần được gọi là [handler\_hand01], trong khi một formatter có tên là form01 trong phần [formatters] sẽ có cấu hình được chỉ định trong một phần được gọi là [formatter\_form01]. Cấu hình bộ ghi nhật ký gốc phải được chỉ định trong một phần được gọi là [logger\_root].

### Module logging.handlers

#### Định nghĩa

Các handlers được cung cấp trong gói. Lưu ý rằng ba trong số các Handlers (StreamHandler, FileHandler và NullHandler) thực sự được xác định trong chính module.

#### StreamHandler

Lớp StreamHandler, nằm trong gói logging, gửi kết quả ghi logging đến các luồng như sys.stdout, sys.stderr hoặc bất kỳ đối tượng giống tệp nào (hoặc chính xác hơn là bất kỳ đối tượng nào hỗ trợ các phương thức write () và flush ()) .

*class logging.StreamHandler(stream=None)* là Trả về một phiên bản mới của lớp StreamHandler. Nếu luồng được chỉ định, cá thể sẽ sử dụng nó để ghi đầu ra; nếu không, sys.stderr sẽ được sử dụng.

*emit(record)* là Nếu một định dạng được chỉ định, nó được sử dụng để định dạng bản ghi. Bản ghi sau đó được ghi vào luồng với một dấu chấm dứt. Nếu có thông tin ngoại lệ, nó được định dạng bằng traceback.print\_exception () và được nối vào luồng.

*flush()* là Xả luồng bằng cách gọi phương thức flush () của nó. Lưu ý rằng phương thức close () được kế thừa từ Handler và do đó không có đầu ra, vì vậy đôi khi có thể cần một lệnh gọi flush () rõ ràng.

#### FileHandler

Lớp FileHandler, nằm trong gói ghi nhật ký cốt lõi, gửi kết quả ghi nhật ký đến tệp đĩa. Nó kế thừa chức năng đầu ra từ StreamHandler.

class logging.FileHandler(filename, mode='a', encoding=None, delay=False) là Trả về một phiên bản mới của lớp FileHandler. Tệp được chỉ định được mở và được sử dụng làm luồng để ghi nhật ký. Nếu chế độ không được chỉ định, 'a' được sử dụng. Nếu mã hóa không phải là Không, nó được sử dụng để mở tệp có mã hóa đó. Nếu sự chậm trễ là đúng, thì việc mở tệp sẽ bị hoãn lại cho đến khi có lệnh gọi emit() đầu tiên. Theo mặc định, tệp phát triển vô thời hạn.

close() là đóng file.

emit(record) xuất bản ghi vào tệp.

#### NullHandler

Lớp NullHandler, nằm trong gói logging, không thực hiện bất kỳ định dạng hoặc đầu ra nào. Về cơ bản, nó là một trình xử lý 'no-op' để các nhà phát triển thư viện sử dụng.

#### WatchedHandler

Lớp WatchedFileHandler, nằm trong module logging.handlers, là một FileHandler xem tệp mà nó đang đăng nhập. Nếu tệp thay đổi, nó sẽ được đóng và mở lại bằng tên tệp.

class logging.handlers.WatchedFileHandler (filename, mode='a', encoding=None, delay=False) là trả về một phiên bản mới của lớp WishedFileHandler. Tệp được chỉ định được mở và được sử dụng làm luồng để ghi nhật ký. Nếu chế độ không được chỉ định, 'a' được sử dụng. Nếu mã hóa không phải là Không, nó được sử dụng để mở tệp có mã hóa đó. Nếu sự chậm trễ là đúng, thì việc mở tệp sẽ bị hoãn lại cho đến khi có lệnh gọi đầu tiên để phát ra (). Theo mặc định, tệp phát triển vô thời hạn.

reopenIfNeeded() là Kiểm tra xem tệp đã thay đổi chưa. Nếu có, luồng hiện có sẽ được xả và đóng lại và tệp được mở lại, thường là tiền thân để xuất bản ghi ra tệp.

emit(record) là Xuất bản ghi vào tệp, nhưng trước tiên gọi reopenIfNeeded () để mở lại tệp nếu nó đã thay đổi.

#### BaseRotatingHandler

Lớp BaseRotatingHandler, nằm trong module logging.handlers, là lớp cơ sở cho các trình xử lý tệp routing, RotatingFileHandler và TimedRotatingFileHandler. Người dùng không cần phải khởi tạo lớp này, nhưng nó có các thuộc tính và phương thức mà bạn có thể cần ghi đè.

* Class logging.handlers.BaseRotatingHandler (filename, mode, encoding=None, delay=False) Các tham số giống như đối với FileHandler. Các thuộc tính là:
* Namer: Nếu phương thức thuộc tính này được đặt thành có thể gọi, phương thức Rotation\_filename () sẽ ủy quyền cho có thể gọi này. Các tham số được truyền cho có thể gọi là những tham số được truyền cho Rot\_filename ().
* Rotator: Nếu thuộc tính này được đặt thành có thể gọi, phương thức xoay () ủy nhiệm cho có thể gọi này. Các tham số được truyền cho có thể gọi là những tham số được truyền cho rotate ().
* rotation\_filename(default\_name): Sửa đổi tên tệp của tệp nhật ký khi rotating.
* rotate(source, dest) Khi rotating, route log hiện hiện tại

#### SysLogHandler

Lớp SysLogHandler, nằm trong mô-đun logging.handlers, hỗ trợ gửi thông báo ghi nhật ký đến một nhật ký hệ thống Unix cục bộ hoặc từ xa.

Class logging.handlers.SysLogHandler(localhost', SYSLOG UDP PORT), facility=LOG\_USER, socktype=socket.SOCK\_DGRAM) là Trả về một phiên bản mới của lớp SysLogHandler nhằm giao tiếp với một máy Unix từ xa có địa chỉ được cung cấp theo địa chỉ dưới dạng một bộ (máy chủ, cổng). Nếu địa chỉ không được chỉ định, ('localhost', 514) được sử dụng. Địa chỉ được sử dụng để mở một ổ cắm. Một giải pháp thay thế cho việc cung cấp tuple (máy chủ, cổng) là cung cấp địa chỉ dưới dạng chuỗi, ví dụ: ‘/ dev / log’. Trong trường hợp này, một ổ cắm miền Unix được sử dụng để gửi thông báo đến syslog. Nếu cơ sở không được chỉ định, LOG\_USER được sử dụng. Loại ổ cắm được mở phụ thuộc vào đối số socktype, đối số này được mặc định là socket.SOCK\_DGRAM và do đó sẽ mở ra một ổ cắm UDP. Để mở một ổ cắm TCP (để sử dụng với các daemon nhật ký hệ thống mới hơn như rsyslog), hãy chỉ định một giá trị của socket.SOCK\_STREAM.

close() Đóng ổ cắm vào máy chủ từ xa.

emit(record) Bản ghi được định dạng, và sau đó được gửi đến syslog server. Nếu có thông tin ngoại lệ, nó sẽ không được gửi đến máy chủ.

encodePriority(facility, priority) Mã hóa cơ sở và mức độ ưu tiên thành một số nguyên. Người dùng có thể chuyển chuỗi hoặc số nguyên - nếu chuỗi được chuyển, từ điển ánh xạ nội bộ sẽ được sử dụng để chuyển đổi chúng thành số nguyên.

|  |  |
| --- | --- |
| Name(String) | Symbolic Value |
| Alert | LOG\_ALERT |
| Crit or Critical | LOG\_CRIT |
| Debug | LOG\_DEBUG |
| Emerg or panic | LOG\_EMERG |
| Err or error | LOG\_ERR |
| Info | LOG\_INFO |
| Notice | LOG\_NOTICE |
| Warn or warning | LOG\_WARNING |

*Bảng 1.1 Bảng ưu tiên trong syslogHandler*

|  |  |
| --- | --- |
| Name (String) | Symbolic value |
| Auth | LOG\_AUTH |
| Authpriv | LOG\_AUTHPRIV |
| Cron | LOG\_CRON |
| Daemon | LOG\_DAEMON |
| Ftp | LOG\_FTP |
| Kern | LOG\_KERN |
| Lpr | LOG\_LPR |
| Mail | LOG\_MAIL |
| News | LOG\_NEWS |
| Syslog | LOG\_SYSLOG |
| User | LOG\_USER |
| Uucp | LOG\_UUCP |
| Local0 | LOG\_LOCAL0 |
| Local1 | LOG\_LOCAL1 |
| Local2 | LOG\_LOCAL2 |
| Local3 | LOG\_LOCAL3 |
| Local4 | LOG\_LOCAL4 |
| Local5 | LOG\_LOCAL5 |
| Local6 | LOG\_LOCAL6 |
| Local7 | LOG\_LOCAL7 |

*Bảng 1.2 Bảng cơ sở trong syslogHandler*

# CÁC KIẾN THỨC VỀ LOGGING

## Các kiến thức cơ bản về logging

### Các cấp độ của python logging

Logging cung cấp một loạt những hàm thuận tiện cho việc sử dụng logging đơn giản. Đó là debug(), info(), warning(), error(), và critical() được sắp xếp theo mức độ nghiêm trọng lớn dần

|  |  |
| --- | --- |
| **Mức** | **Nội dung** |
| Debug | Thông tin chi tiết, thường là thông tin để tìm lỗi.  Ví dụ: Hệ thống dữ liệu trên PRODUCTION thường lớn hơn nhiều so với DEVELOPING nên số lượng trường hợp xảy ra khi áp dữ liệu vào phần mềm cũng nhiều hơn. Các test-case có thể chưa bao quát được hết các lỗi --> Việc đặt debug có thể giúp developer/tester có thể giả lập lại được chính xác toàn bộ thao tác, workflow của cụm dữ liệu khi đi vào/đi ra khỏi phần mềm. |
| Info | Thông báo thông thường, các thông tin in ra khi chương trình chạy theo đúng kịch bản.  Ví dụ: Hành vi đăng nhâp vào hệ thống bởi tài khoản nào, version của phần mềm, ... |
| Warning | Thông báo khi nghi vấn bất thường hoặc lỗi có thể xảy ra, tuy nhiên chương trình vẫn có thể hoạt động.  Ví dụ: Khách hàng cố tình nhấn nhiều lần vào button thanh toán trong khi người dùng không đủ điều kiện để thanh toán, các warning sẽ được log lại khi khách hàng "cố gắng" làm sai với hướng dẫn. |
| Error | Lỗi, chương trình có thể không hoạt động được một số chức năng hoặc nhiệm vụ nào đó, thường thì nên dùng ghi bắt được Exception.  Ví dụ: Khách hàng thanh toán tiền cho dịch vụ A nhưng không thể trừ tiền của khách hàng do hệ thống thanh toán đang quá tải. Mỗi ERROR sẽ được ghi vào log-files để sau này có thể đối soát được lỗi quá tải đã gây ảnh hưởng đến bao nhiêu khách hàng. |
| Critical | Lỗi, chương trình gặp lỗi nghiêm trọng không thể giải quyết được và bắt buộc phải dừng lại.  Ví dụ: Phần mềm sử dụng nhiều, chiếm dụng RAM nhiều, gây tràn memory, phần mềm trước khi bị tự tắt đi sẽ đưa ra các thông báo. |

*Bảng 2. 1 Bảng các cấp độ trong logging*

Mức độ mặc định là Warning, nó có nghĩa là những sự kiện trong mữa độ này và lớn hơn sẽ được theo dõi, trừ khi gói logging được dùng để cấu hình để làm việc khác

Sự kiện được theo dõi có thể được xử theo nhiều cách khác nhau. Cách đơn giản nhất là để xử lý là in chúng ra màn hình. Một cách khác thông thường khác là viết chúng vào file

Một ví dụ đơn giản về in log ra màn hình:

import logging

logging.warning ('Warning!!!') #hiện trên console

logging.info ('info….') #không hiện trên console

Kết quả của đoạn code trên sẽ in mỗi dòng thông báo Warning còn dòng thông báo INFO không xuất hiện vì cấp độ thông báo mặc định là Warning. Dòng thông báo được in ra bao gồm mức độ và mô tả về sự kiện trong lời gọi logging, ví dụ như trên là ’Warning!!!’.

### Logging vào trong file

Ngoài việc hiện thông báo trên console thì còn có ghi lại log đó vào file:

import logging

logging.basicConfig(filename='example.log',level=logging.DEBUG)

logging.debug('Message go to log file')

logging.warning('and this warning!!!')

logging.info('and this info….')

(Câu lệnh ‘*filename=’* dùng để ghi lại log vào file có tên example.log và mức độ của log này là DEBUG được thể hiện ở lệnh ‘*level*=’)

Và khi chúng ta thực thi câu lệnh đó nó sẽ ghi lại những log vào trong file ‘example.log’:

DEBUG:root:Message go to log file

WARNING:root:and this Warning!!!

INFO:root:and this info....

Ví dụ trên cũng chỉ ra cách mà bạn đặt cấp độ logging đóng vai trò là ngưỡng để theo dõi. Trong trường hợp trên, chúng ta đặt cấp độ là DEBUG, tất cả các dòng thông báo sẽ được in ra

Ngoài ra nếu bạn chạy dòng lệnh trên nhiều lần, những thông báo mới sẽ được ghi tiếp vào sau các thông báo cũ vào trong file example.log. Nếu bạn muốn ghi đè lên những log cũ thì thêm biến ‘*filemode=w*’, câu lệnh này giúp thay đổi cách ghi file trong python.

logging.basicConfig(file='example.log',filemode='w',level=logging.DEBUG)

Đầu ra cũng sẽ như trước đó, nhưng file log sẽ không ghi tiếp mà thay vào đó sẽ ghi đè lên thông báo cũ.

### Logging từ nhiều module khác

Nếu chương trình của bạn bao gồm nhiều module, đây là ví dụ để có thể tổ chức logging:

Test.py:

import logging

import test2

def main():

logging.basicConfig(filename='app.log',level=logging.INFO)

logging.info('Started')

test2.do\_someting()

logging.info('Finished')

if \_\_name\_\_ == '\_\_main\_\_':

main()

Test2.py:

import logging

def do\_someting():

logging.info('Doing abc')

Sau khi thực hiện chạy file test.py thì ta thu được kết quả ở file app.log như sau :

INFO:root:Started

INFO:root:Doing abc

INFO:root:Finished

### Biến dữ liệu trong logging

Logging ghi biến dữ liệu, sử dụng chuỗi định dạng cho thông báo mô tả sự kiện và ghi tiếp dữ liệu biến vào dưới dạng đối số. Ví dụ:

Import logging

Logging.warning(‘%s behind %s’, ‘look’,’you!’)

Sau khi thực thi ta sẽ nhận được dòng thông báo sau trên console:

WARNING : root : look behind you!

### Thay đổi định dạng hiển thị thông báo

Để thay đổi định dạng hiển thị thông báo, ta thêm lệnh *‘format = ’* ở *logging.basicConfig*

import logging

logging.basicConfig(format='%(levelname)s:%(message)s', level=logging.DEBUG)

logging.debug('1 duck')

logging.info('spread')

logging.warning('two wing')

Thực thi câu lệnh ta thu được kết quả như sau:

DEBUG: 1 duck

INFO: spread

WARNING: two wing

Ở ví dụ trước, chữ root có xuất hiện ở dòng thống báo trên console còn ở ví dụ này đã biến mất do sử dụng ‘*format=*’ nên định dạng thông báo log thay đổi theo các thuộc tính ta sử dụng.

|  |  |  |
| --- | --- | --- |
| Tên thuộc tính | Format | Giải thích |
| asctime | %(asctime)s | Hiện thời gian |
| created | %(created)f | Thời gian khi mà bản log được tạo |
| filename | %(filename)s | Tên của file |
| message | %(message)s | Thông báo |
| levelname | %(levelname)s | Mức độ của logging ('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL'). |
| lineno | %(lineno)d | Dòng thứ bao nhiêu ở lời gọi hàm |

*Bảng 2. 2 Bảng thuộc tính xuất hiện trong chuỗi format*

### Hiển thị thời gian của thông báo

Như ở phần 2.1.5 để hiển thị thời gian của thông báo ta thêm *%(asctime)s* vào chuỗi ‘*format=’:*

import logging

logging.basicConfig(format='%(asctime)s %(message)s' , level=logging.DEBUG)

logging.warning('end class')

Sau khi thực thi câu lệnh thì trên console sẽ hiện kết quả sau:

2023-02-27 13:12:13,101 end class

Ta cũng có thể thay đổi thứ tự của ngày giờ hoặc thay đổi dấu gạch giữa thứ ngày tháng thành dấu ‘/’ bằng cách sử dụng *‘datefmt=’* trong *‘format=’*

logging.basicConfig(format='%(asctime)s %(message)s' , datefmt='%m/%d/%Y % H:%M:%S %p' ,level=logging.DEBUG)

Và đây là kết quả

02/27/2023 01:14:14 PM end calss

## Kiến thức nâng cao về logging

Ở phần trước chúng ta có thể thấy logger mặc định có tên là root (Ví dụ ở 2.1.1) nó được sử dụng ở logging module khi được gọi trực tiếp ví dụ như *logging.debug().* Ta có thể (và nên) định dạng logger của ta bằng cách tạo một đối tượng trong lớp *Logger*, đặc biệt nếu ứng dụng của bạn có nhiều module. Hãy xem một vài lớp và hàm trong module:

* Logger: Đây là lớp có các đối tượng sẽ được sử dụng trực tiếp trong code để gọi hàm.
* LogRecord: Logger tự động tạo đối tượng LogRecord có đầy đủ thông tin về sự kiện đã được ghi, như là tên của logger, hàm, dòng số bao nhiều, thông báo, và nhiều hơn nữa.
* Handler: Handler gửi LogRecord tới output, như là màn hình console hay tập tin. Handler dựa trên các lớp con như là StreamHandler, FileHandler, SMTPHandler, HTTPHandler, … Các lớp con này gửi kết quả ghi nhật ký đến các đích tương ứng.
* Formatter: Đây là nơi bạn có thể thay đổi định dạng của output bằng cách đưa vào các thuộc tính mà output chứa.

### Tạo trình ghi nhật ký (logger)

Chúng ta sẽ chủ yếu xử lý với đối tượng của lớp Logger, được khởi tạo bằng cách sử dụng hàm *logging.getLogger(name).*

import logging

logger = logging.getLogger('example')

logger.warning('Warning!!!!!!!')

Output:

Warning!!!!!!

### Tạo trình ghi nhật ký với trình xử lý (handler) và trình định dạng (formatter)

Trình xử lý(handler) được dùng khi mà ta muốn cấu hình logger của ta và gửi logs tới nhiều nơi khi chúng được tạo.   
 Trình định dạng(formatter) được dùng để thay đổi đầu ra hiển thị trên console hoặc file.

Dưới đây là ví dụ về logger hiển thị trên console và ghi file(file.log):

import logging

logger = logging.getLogger(\_\_name\_\_)

C\_handler = logging.StreamHandler()

F\_handler = logging.FileHandler(filename='file.log')

C\_handler.setLevel(logging.WARNING)

F\_handler.setLevel(logging.ERROR)

C\_format=logging.Formatter('%(name)s-%(levelname)s-%(message)s')

F\_format=logging.Formatter('%(asctime)s-%(name)s-%(levelname)s-%(message)s')

C\_handler.setFormatter(C\_format)

F\_handler.setFormatter(F\_format)

logger.addHandler(C\_handler)

logger.addHandler(F\_handler)

logger.warning('Warning!!!!!!')

logger.error('Error!!!!!')

Đầu tiên, logger.warning() tạo LogRecord nó lưu thông tin của sự kiện và đưa tới tất cả Handlers là: c\_handler và f\_handler

c\_handler là StreamHandler với level WARNING lấy thông tin từ LogRecord sử dụng định dạng c\_format và in ra console:

\_\_main\_\_ - WARNING - Warning!!!!!!

\_\_main\_\_ - ERROR - Error!!!!!!

Còn f\_handler là FileHandler có level ERROR và nó bỏ qua LogRecord vì level ERROR cao hơn WARNING

Và khi logger.error() được gọi, c\_handler vẫn hoạt động như trước và f\_handler nhận LogRecord ở mức ERROR, vì vậy nó tạo ra đầu ra như c\_handlers, nhưng thay vì hiện ra console nó ghi vào file.log:

2023-03-02 11:31:10,574-\_\_main\_\_-ERROR-Error!!!!!

### Tải và lưu cấu hình logging

Ta có thể cấu hình logging như trên với module và hàm hoặc tạo một file config. Nó sẽ hữu ích trong trường hợp ta muốn thay đổi cấu hình trong một ứng dụng khác với các ứng dụng khác.

Ví dụ về file config

[loggers]

keys=root,sampleLogger

[handlers]

keys=consoleHandler

[formatters]

keys=sampleFormatter

[logger\_root]

level=DEBUG

handlers=consoleHandler

[logger\_sampleLogger]

level=DEBUG

handlers=consoleHandler

qualname=sampleLogger

propagate=0

[handler\_consoleHandler]

class=StreamHandler

level=DEBUG

formatters=sampleFormatter

arg=(sys.stdout,)

[formatter\_sampleFormatter]

format=%(message)s

Nó có ba phần chính là [loggers], [handlers] và [formatters], mỗi phần đều chứa các keys khác nhau tương ứng với các loại logger, handler và formatter.

Phần [loggers] chứa khóa "keys" cho biết danh sách các logger được sử dụng trong tệp cấu hình, trong trường hợp này là root và sampleLogger.

Phần [handlers] chứa khóa "keys" cho biết danh sách các handlers được sử dụng trong tệp cấu hình, trong trường hợp này chỉ có consoleHandler.

Phần [formatters] chứa khóa "keys" cho biết danh sách các formatters được sử dụng trong tệp cấu hình, trong trường hợp này chỉ có sampleFormatter.

Các logger được cấu hình bao gồm hai logger là "root" và "sampleLogger". Logger "root" được cấu hình để đưa các tin nhắn log vào consoleHandler với mức độ DEBUG. Logger "sampleLogger" cũng được cấu hình để đưa các tin nhắn log vào consoleHandler, tuy nhiên nó có mức độ DEBUG và một tên gọi "sampleLogger". Nó được thiết lập không truyền dẫn log đến các logger cha.

Chúng ta có một handler consoleHandler, là một StreamHandler, được thiết lập với mức độ DEBUG và định dạng sampleFormatter. Cuối cùng, chúng ta có một formatter duy nhất là sampleFormatter, với định dạng được chỉ định là "%(message)s".

Ở file config trên có 2 logger, 1 handler và 1 formatter. Để sử dụng file config, ta sử dụng hàm fileConfig()

import logging

import logging.config

logging.config.fileConfig(fname='test.conf')

logger = logging.getLogger(\_\_name\_\_)

logger.debug('test config')

# THỰC NGHIỆM

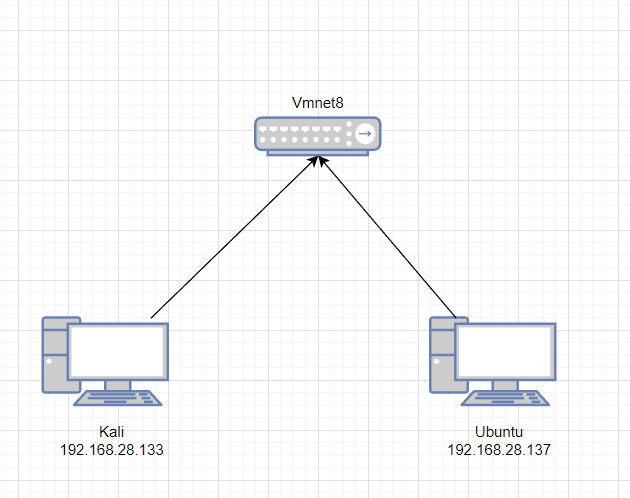
## Giới thiệu

### Mô tả

* Yêu cầu
* Ubuntu 64-bit
* Kali Linux 64-bit
* Nanopot
* Công việc:
* Máy Kali gửi những gói tin qua cổng có sẵn của nanopot tại máy Ubuntu qua giao thức ssh
* Máy Ubuntu nhận được gói tin, đồng thời đóng cổng có sẵn lại

### Yêu cầu

Hai máy Ubuntu và Kali Linux phải đc kết nối mạng với nhau.



*Hình 3. 1 Mô hình mạng kểt nối của hai máy Kali và Ubuntu*

### Mục tiêu

Máy Ubuntu đã khởi động logging và ghi log thành công về các gói tin được gửi từ máy Kali Linux thông qua ssh.

## Triển khai, thử nghiệm

### Cài đặt

* Sử dụng phần mềm VMware để cài đặt 2 máy Ubuntu và Kali Linux
* Máy Ubuntu:
* Cài đặt Openssh-server
* Cài đặt nanopot
* Cấu hình Network Adapter trên cả 2 máy để có thể kết nối được với nhau

### Xây dựng

* **Tạo \_init\_:**

Đây là một hàm khởi tạo cho một lớp đối tượng trong Python. Hàm này có ba tham số đầu vào là "bind\_ip" để chỉ định địa chỉ IP muốn ràng buộc, "ports" để chỉ định danh sách các cổng mà lớp đối tượng sẽ lắng nghe kết nối và "log\_filepath" để chỉ định đường dẫn đến tệp nhật ký.

Nếu danh sách các cổng không được cung cấp hoặc rỗng, hàm sẽ ném ra một ngoại lệ Exception.

Lớp đối tượng này cũng có một số thuộc tính, bao gồm "listener\_threads" để lưu trữ các luồng lắng nghe kết nối, "logger" để ghi lại các hoạt động của lớp đối tượng và "ports\_error" để lưu trữ các cổng không thể kết nối được.

Trong phần đầu tiên của hàm khởi tạo, hàm sẽ thiết lập giá trị ban đầu cho các thuộc tính và khởi tạo một logger mới bằng cách gọi phương thức "prepare\_logger".

Sau đó, hàm ghi lại thông tin về các cổng và đường dẫn đến tệp nhật ký trong logger và in một dòng kẻ ngang vào bản ghi nhật ký để phân biệt giữa các thông tin khác nhau.

def \_\_init\_\_(self, bind\_ip, ports, log\_filepath):

if len(ports) < 1:

raise Exception("No ports provided.")

self.bind\_ip = bind\_ip

self.ports = ports

self.log\_filepath = log\_filepath

self.listener\_threads = {}

self.logger = self.prepare\_logger()

self.ports\_error = list()

self.logger.info('{:-^20}'.format(' Logging test '))

self.logger.info("Ports: %s" % self.ports)

self.logger.info("Log filepath: %s" % self.log\_filepath)

* **port và log filepath sẽ được config ở file main:**

ports = config.get('default', 'ports', raw=True, fallback="22,443,8080,8888,9999,53")

host = config.get('default', 'host', raw=True, fallback="0.0.0.0")

log\_filepath = config.get('default', 'logfile', raw=True, fallback="/var/log/nanopot.yaml")

Đoạn mã trên sử dụng phương thức get của đối tượng config để lấy giá trị các thông số từ một tệp cấu hình. config được giả định là một đối tượng được khởi tạo từ một thư viện cấu hình như configparser.

Cụ thể, đoạn mã trên đang lấy giá trị của ba thông số khác nhau:

* ports: đây là một chuỗi chứa danh sách các cổng được ngăn cách bằng dấu phẩy. Nếu giá trị này không tồn tại trong tệp cấu hình, giá trị mặc định fallback sẽ được sử dụng là "22,443,8080,8888,9999,53".
* host: đây là địa chỉ IP mà ứng dụng sẽ ràng buộc để lắng nghe kết nối. Nếu giá trị này không tồn tại trong tệp cấu hình, giá trị mặc định fallback sẽ được sử dụng là "0.0.0.0".

log\_filepath: đây là đường dẫn đến tệp nhật ký mà ứng dụng sẽ sử dụng để ghi lại các hoạt động của mình. Nếu giá trị này không tồn tại trong tệp cấu hình, giá trị mặc định fallback sẽ được sử dụng là "/var/log/nanopot.yaml".

* **Sử dụng socket để tạo kết nối handler:**

def handle\_connection(self, client\_socket, port, ip, remote\_port):

self.logger.info("Connection received: %s: %s:%d" % (port, ip, remote\_port))

client\_socket.settimeout(4)

try:

data = client\_socket.recv(64)

self.logger.info("Data received: %s: %s:%d: %s" % (port, ip, remote\_port, data))

client\_socket.send("Access Denied .\n".encode('utf8'))

except timeout:

pass

client\_socket.close()

Hàm handle\_connection được định nghĩa để xử lý một kết nối đến máy chủ từ một khách hàng. Đối số client\_socket là một đối tượng socket được tạo để xử lý kết nối. port, ip, và remote\_port lần lượt là số cổng và địa chỉ IP của máy chủ, và số cổng của khách hàng.

Trong hàm này, đầu tiên ghi nhật ký về việc kết nối được nhận và các thông tin liên quan đến nó, bao gồm số cổng, địa chỉ IP của máy khách và số cổng của nó.

Tiếp theo, đối tượng socket được thiết lập với thời gian chờ là 4 giây bằng cách gọi settimeout với đối số là 4.

Sau đó, hàm sẽ cố gắng nhận dữ liệu từ khách hàng với kích thước tối đa 64 byte. Dữ liệu này được ghi nhật ký kèm theo thông tin liên quan đến kết nối.

Cuối cùng, hàm trả về cho khách hàng thông báo "Access Denied" và đóng đối tượng socket của khách hàng bằng cách gọi close.

Nếu hàm recv hoặc send gây ra lỗi, hàm sẽ không thông báo lỗi và đơn giản là tiếp tục chạy mà không làm gì cả.

Khi máy ubuntu nhận được kết nối thì dữ liệu sẽ tự động được ghi lại thông qua hàm *‘self.logger.info’*

* **Tạo listener mới để có thể nghe được các gói tin từ máy khác:**

def start\_new\_listener\_thread(self, port):

try:

listener = socket() # Defaults (socket.AF\_INET, socket.SOCK\_STREAM)

listener.bind((self.bind\_ip, int(port)))

listener.listen(5)

while True:

client, addr = listener.accept()

client\_handler = threading.Thread(target=self.handle\_connection, args=(client, port, addr[0], addr[1]))

client\_handler.start()

except :

self.logger.error("Already in use this port: %s" % (port))

pass

Hàm start\_new\_listener\_thread được định nghĩa để tạo một luồng lắng nghe mới cho một cổng được chỉ định. Đối số port là số cổng cần lắng nghe.

Trong hàm này, đầu tiên tạo một đối tượng socket để lắng nghe kết nối từ khách hàng. Đối tượng socket này được gán cho biến listener.

Sau đó, gọi phương thức bind của đối tượng socket để ràng buộc nó với địa chỉ IP và số cổng được chỉ định.

Hàm tiếp tục gọi phương thức listen của đối tượng socket để bắt đầu lắng nghe kết nối đến cổng được chỉ định. Tham số 5 được đưa vào để chỉ định số lượng kết nối tối đa được chấp nhận trước khi các kết nối mới bị từ chối.

Sau đó, vòng lặp vô hạn được bắt đầu để chấp nhận các kết nối đến từ khách hàng. Mỗi khi một kết nối mới được chấp nhận, một đối tượng socket mới và địa chỉ của khách hàng sẽ được trả về bởi phương thức accept.

Đối tượng socket mới này sẽ được chuyển đến hàm handle\_connection bằng cách tạo một đối tượng Thread để xử lý kết nối này. Thông tin về kết nối, bao gồm số cổng và địa chỉ IP của máy khách, cũng được truyền vào hàm handle\_connection dưới dạng các đối số.

Cuối cùng, hàm bắt đầu luồng mới bằng cách gọi phương thức start của đối tượng Thread, và tiếp tục lắng nghe các kết nối mới.

Nếu cổng đã được sử dụng bởi một chương trình khác, hàm sẽ ghi nhật ký một thông báo lỗi và tiếp tục thực thi mà không làm gì cả.

* **Bắt đầu cho listener hoạt động:**

def start\_listening(self):

for port in self.ports:

self.listener\_threads[port] = threading.Thread(target=self.start\_new\_listener\_thread, args=(port,))

self.listener\_threads[port].start()

def run(self):

self.start\_listening()

Hàm start\_listening được định nghĩa để bắt đầu lắng nghe kết nối đến các cổng được chỉ định trong danh sách self.ports.

Trong hàm này, vòng lặp được sử dụng để tạo một luồng lắng nghe mới cho mỗi cổng được liệt kê trong danh sách self.ports.

Đối với mỗi cổng, hàm tạo một đối tượng Thread mới và gán nó cho biến self.listener\_threads để theo dõi. Sau đó, hàm gọi phương thức start của đối tượng Thread để bắt đầu lắng nghe kết nối đến cổng này.

Hàm run được định nghĩa để bắt đầu chạy server. Trong hàm này, hàm start\_listening được gọi để bắt đầu lắng nghe kết nối đến các cổng được chỉ định.

* **Config file log:**

def prepare\_logger(self):

logging.basicConfig(level=logging.DEBUG,

format='%(asctime)s %(levelname)-8s %(message)s',

datefmt='%Y/%m/%d %H:%M:%S',

filename=self.log\_filepath,

filemode='a')

logger = logging.getLogger(\_\_name\_\_)

Hàm prepare\_logger được định nghĩa để cấu hình logger cho server.

Trong hàm này, logging được cấu hình với cấp độ logging được đặt thành DEBUG, định dạng logging được thiết lập bao gồm thời gian, cấp độ, và nội dung, định dạng thời gian được đặt thành %Y/%m/%d %H:%M:%S, filename được thiết lập đến đường dẫn file log được chỉ định trong biến self.log\_filepath, filemode được thiết lập thành 'a' để ghi log vào cuối file, mở file log trong chế độ ghi.

Cuối cùng, logger được khởi tạo bằng cách gọi logging.getLogger(\_\_name\_\_) với tên của module được sử dụng để khởi tạo logger. Logger này sẽ được sử dụng để ghi lại các thông tin về kết nối và lỗi của server.

* **Tạo hàm \_main\_ và load config:**

config\_filepath = sys.argv[1]

config = configparser.ConfigParser()

config.read(config\_filepath)

Đầu tiên, biến config\_filepath được định nghĩa là tham số thứ nhất được truyền vào chương trình bằng sys.argv[1], tức là đường dẫn đến file cấu hình.

Sau đó, một đối tượng ConfigParser được tạo và gọi phương thức read() để đọc file cấu hình từ đường dẫn được chỉ định trong biến config\_filepath. Kết quả là các thông tin trong file cấu hình được lưu trữ trong biến config dưới dạng một đối tượng ConfigParser.

* **Kiểm tra ngoại lệ:**

ports\_list = []

try:

ports\_list = ports.split(',')

except Exception as e:

sys.exit(1)

Đầu tiên, một danh sách trống được khởi tạo bằng cách gán ports\_list bằng một danh sách rỗng.

Tiếp theo, một khối try-except được sử dụng để chuyển đổi chuỗi ports thành danh sách ports\_list. Trong khối try, hàm split() được gọi trên chuỗi ports để tách các cổng thành một danh sách các chuỗi, được phân tách bởi dấu phẩy. Nếu thành công, danh sách mới được gán cho biến ports\_list.

Nếu lỗi xảy ra trong quá trình chuyển đổi, chương trình sẽ thoát với mã lỗi 1 bằng câu lệnh sys.exit(1). Lỗi có thể là do ports không phải là một chuỗi có thể chuyển đổi thành danh sách các cổng hoặc có lỗi xảy ra trong quá trình chuyển đổi.

### Thực nghiệm

* Khởi động nanopot trên máy Ubuntu

Text

Description automatically generated

* Text

  Description automatically generatedSử dụng ssh bên máy kali để trao đổi khóa với máy Ubuntu, nanopot sẽ bắt được sự kiệnText

  Description automatically generated
* Text

  Description automatically generatedTrở về máy Ubuntu và kiểm tra file log theo đường dẫn trên sẽ thấy kết quả được log lại

# KẾT LUẬN

Logging rất quan trọng nếu như chúng ta muốn chẩn đoán sự cố hoặc biết điều gì đang xảy ra với hệ thống của mình. Trong báo cáo này, chúng tôi hy vọng có thể thuyết phục bạn rằng việc sử dụng thư viện Logging của Python tốt hơn là chỉ đặt các câu lệnh ‘print’ ở mọi nơi trong code. Không những tiết kiệm thời gian mà việc sử dụng các mức độ thông báo của logging còn giúp cho việc ưu tiên xử lý vấn đề nào trước khi có lỗi xảy ra khi chạy phần mềm. Ngoài ra còn có thể giúp người dùng giám sát được hệ thống máy tính khi chạy một giao thức và muốn biết dữ liệu trao đổi trong quá trình chạy giao thức đó.

Thư viện ghi nhật ký Python cho phép:

* Kiểm soát những gì được phát ra.
* Xác định loại thông tin bạn muốn đưa vào nhật ký của mình.
* Cấu hình giao diện của file log khi nó được tạo ra.
* Quan trọng nhất, đặt điểm đến cho bản log của bạn.

Bằng cách ghi lại dữ liệu hữu ích từ nguồn hợp lí, ta không chỉ có thể debug dễ dàng mà còn có thể sử dụng dữ liệu để phân tích hiệu suất của ứng dụng nhằm lập kế hoạch mở rộng cho sau này.

Thư viện Logging Python ngoài những gì được trình bày ở trên còn có các tài liệu giúp bạn hiểu thêm những kiến thức cơ bản và nâng cao hơn cũng như cung cấp các ví dụ về code để có thể hiểu rõ về Logging và cách thức hoạt động của nó.

**TÀI LIỆU THAM KHẢO**

1. <https://docs.python.org/3.6/howto/logging.html>
2. <https://cuccode.com/python_logging.html>
3. <https://realpython.com/python-logging>
4. <https://www.javatpoint.com/logging-in-python>
5. <https://www.youtube.com/watch?v=77qj9o9pmek>
6. https://github.com/amilstead/python-logging-examples

**BẢNG PHÂN CÔNG VIỆC TRONG NHÓM**

|  |  |
| --- | --- |
| **Nguyễn Đăng Long** | Làm demo về logging trong hệ thống mạng |
| **Hoàng Dương Tùng** | Tìm tài liệu và lý thuyết liên quan đến logging và logging trong hệ thống mạng |
| **Dương Minh Nghĩa** | Tìm tài liệu và lý thuyết liên quan đến logging và logging trong hệ thống mạng |
| **Nguyễn Thị Thùy Trang** | Làm word + Powerpoint thuyết trình |