logging.config—配置文件参数详解

这一节描述了用于配置 logging 模块的 API。

python 版本为：3.8.4

翻译的不好，请移步：
转载：logging.config — Logging configuration。配置文件详解
官网：logging.config — Logging configuration

配置函数

下列函数可配置 logging 模块。 它们位于 logging.config 模块中。 它们的使用是可选的 — 要配置 logging 模块你可以使用这些函数，也可以通过调用主 API (在 logging 本身定义) 并定义在 logging 或 logging.handlers 中声明的处理程序。

- logging.config.``dictConfig(*config*)

  从一个字典获取 日志记录 配置。 字典的内容描述见下文的 配置字典架构。如果在配置期间遇到错误，此函数将引发 ValueError，TypeError，AttributeError 或 ImportError 并附带适当的描述性消息。 下面是将会引发错误的（可能不完整的）条件列表: level 不是字符串或者不是对应于实际日志记录级别的字符串。 propagate 值不是布尔类型。id 没有对应的目标。在增量调用期间发现不存在的处理程序 id。无效的日志记录器名称。无法解析为内部或外部对象。解析由 DictConfigurator 类执行，该类的构造器可传入用于配置的字典，并且具有 configure() 方法。 logging.config 模块具有可调用属性 dictConfigClass，其初始值设为 DictConfigurator。 你可以使用你自己的适当实现来替换 dictConfigClass 的值。 dictConfig() 会调用 dictConfigClass 并传入指定的字典，然后在所返回的对象上调用 configure() 方法以使配置生效: def dictConfig(config): dictConfigClass(config).configure() 例如， DictConfigurator 的子类可以在它自己的 __init__() 中调用 DictConfigurator.__init__()，然后设置可以在后续 configure() 调用中使用的自定义前缀。 dictConfigClass 将被绑定到这个新的子类，然后就可以与在默认的未定制状态下完全相同的方式调用 dictConfig()。*3.2* 新版功能.

- logging.config.``fileConfig(*fname*, *defaults=None*, *disable_existing_loggers=True*)

  从一个 configparser 格式文件中读取日志记录配置。 文件格式应当与 Configuration file format 中的描述一致。 此函数可在应用程序中被多次调用，以允许最终用户在多个预设配置中进行选择（如果开发者提供了展示选项并加载选定配置的机制）。参数**fname** – 一个文件名，或一个文件类对象，或是一个派生自 RawConfigParser 的实例。 如果传入了一个派生自 RawConfigParser 的实例，它会被原样使用。 否则，将会实例化一个 Configparser，并且它会从作为 fname 传入的对象中读取配置。 如果存在 readline() 方法，则它会被当作一个文件类对象并使用 read_file() 来读取；在其它情况下，它会被当作一个文件名并传递给 read()。**defaults** – 要传递给 ConfigParser 的默认值可在此参数中指定。**disable_existing_loggers** – 如果指定为 False，则当执行此调用时已存在的日志记录器会保持启用。 默认值为 True 因为这将以向下兼容方式启用旧行为。 此行为是禁用任何现有的非根日志记录器除非它们或它们的上级在日志记录配置中被显式地命名。在 *3.4* 版更改: 现在接受 RawConfigParser 子类的实例作为 fname 的值。 这有助于:使用一个配置文件，其中日志记录配置只是全部应用程序配置的一部分。使用从一个文件读取的配置，它随后会在被传给 fileConfig 之前由使用配置的应用程序来修改（例如基于命令行参数或运行时环境的其他部分）。

- logging.config.``listen(*port=DEFAULT_LOGGING_CONFIG_PORT*, *verify=None*)

  在指定的端口上启动套接字服务器，并监听新的配置。 如果未指定端口，则会使用模块默认的 DEFAULT_LOGGING_CONFIG_PORT。 日志记录配置将作为适合由 dictConfig() 或 fileConfig() 进行处理的文件来发送。 返回一个 Thread 实例，你可以在该实例上调用 start() 来启动服务器，对该服务器你可以在适当的时候执行 join()。 要停止该服务器，请调用 stopListening()。如果指定 verify 参数，则它应当是一个可调用对象，该对象应当验证通过套接字接收的字节数据是否有效且应被处理。 这可以通过对通过套接字发送的内容进行加密和/或签名来完成，这样 verify 可调用对象就能执行签名验证和/或解密。 verify 可调用对象的调用会附带一个参数 —— 通过套接字接收的字节数据 —— 并应当返回要处理的字节数据，或者返回 None 来指明这些字节数据应当被丢弃。 返回的字节数据可以与传入的字节数据相同（例如在只执行验证的时候），或者也可以完全不同（例如在可能执行了解密的时候）。要将配置发送到套接字，请 读取配置文件 并将其作为字节序列发送到套接字，字节序列要以使用 struct.pack('>L', n) 打包为二进制格式的四字节长度的字符串打头。注解 因为配置的各部分是通过 eval() 传递的，使用此函数可能让用户面临安全风险。 虽然此函数仅绑定到 localhost 上的套接字，因此并不接受来自远端机器的连接，但在某些场景中不受信任的代码可以在调用 listen() 的进程的账户下运行。 具体来说，如果如果调用 listen() 的进程在用户无法彼此信任的多用户机器上运行，则恶意用户就能简单地通过连接到受害者的 listen() 套接字并发送运行攻击者想在受害者的进程上执行的任何代码的配置的方式，安排运行几乎任意的代码。 如果是使用默认端口这会特别容易做到，即便使用了不同端口也不难做到。 要避免发生这种情况的风险，请在 listen() 中使用 verify 参数来防止未经认可的配置被应用。在 *3.4* 版更改: 添加了 verify 参数。注解 如果你希望将配置发送给未禁用现有日志记录器的监听器，你将需要使用 JSON 格式的配置，该格式将使用 dictConfig() 进行配置。 此方法允许你在你发送的配置中将 disable_existing_loggers 指定为 False。

- logging.config.``stopListening()

  停止通过对 listen() 的调用所创建的监听服务器。 此函数的调用通常会先于在 listen() 的返回值上调用 join()。

配置字典架构

描述日志记录配置需要列出要创建的不同对象及它们之间的连接；例如，你可以创建一个名为 'console' 的处理程序，然后名为 'startup' 的日志记录器将可以把它的消息发送给 'console' 处理程序。 这些对象并不仅限于 logging 模块所提供的对象，因为你还可以编写你自己的格式化或处理程序类。 这些类的形参可能还需要包括 sys.stderr 这样的外部对象。 描述这些对象和连接的语法会在下面的 对象连接 中定义。

字典架构细节

传给 dictConfig() 的字典必须包含以下的键:

- *version* - 应设为代表架构版本的整数值。 目前唯一有效的值是 1，使用此键可允许架构在继续演化的同时保持向下兼容性。

所有其他键都是可选项，但如存在它们将根据下面的描述来解读。 在下面提到 'configuring dict' 的所有情况下，都将检查它的特殊键 `'()'` 以确定是否需要自定义实例化。 如果需要，则会使用下面 用户定义对象 所描述的机制来创建一个实例；否则，会使用上下文来确定要实例化的对象。

- *formatters* - 对应的值将是一个字典，其中每个键是一个格式器 ID 而每个值则是一个描述如何配置相应 `Formatter` 实例的字典。

  将在配置字典中搜索键 `format` 和 `datefmt` (默认值均为 `None`) 并且这些键会被用于构造 `Formatter` 实例。

  在 *3.8* 版更改: 一个 `validate` 键 (默认值为 `True`) 可被添加到配置字典的 `formatters` 部分，这会被用来验证格式的有效性。

- *filters* - 对应的值将是一个字典，其中每个键是一个过滤器 ID 而每个值则是一个描述如何配置相应 Filter 实例的字典。

  将在配置字典中搜索键 `name` (默认值为空字符串) 并且该键会被用于构造 `logging.Filter` 实例。

- *handlers* - 对应的值将是一个字典，其中每个键是一个处理程序 ID 而每个值则是一个描述如何配置相应 Handler 实例的字典。

  将在配置字典中搜索下列键:

  - `class` (强制)。 这是处理程序类的完整限定名称。
  - `level` (可选)。 处理程序的级别。
  - `formatter` (可选)。 处理程序所对应格式化器的 ID。
  - `filters` (可选)。 由处理程序所对应过滤器的 ID 组成的列表。

  所有 其他 键会被作为关键字参数传递给处理程序类的构造器。 例如，给定如下代码段:

  ```
  handlers:
    console:
      class : logging.StreamHandler
      formatter: brief
      level   : INFO
      filters: [allow_foo]
      stream  : ext://sys.stdout
    file:
      class : logging.handlers.RotatingFileHandler
      formatter: precise
      filename: logconfig.log
      maxBytes: 1024
      backupCount: 3
  ```

  ID 为 `console` 的处理程序会被实例化为 `logging.StreamHandler`，并使用 `sys.stdout` 作为下层流。 ID 为 `file` 的处理程序会被实例化为 `logging.handlers.RotatingFileHandler`，并附带关键字参数 `filename='logconfig.log', maxBytes=1024, backupCount=3`。

- *loggers* - 对应的值将是一个字典，其中每个键是一个日志记录器名称而每个值则是一个描述如何配置相应 Logger 实例的字典。

  将在配置字典中搜索下列键:

  - `level` (可选)。 日志记录器的级别。
  - `propagate` (可选)。 日志记录器的传播设置。
  - `filters` (可选)。 由日志记录器对应过滤器的 ID 组成的列表。
  - `handlers` (可选)。 由日志记录器对应处理程序的 ID 组成的列表。

  指定的日志记录器将根据指定的级别、传播、过滤器和处理程序来配置。

- *root* - 这将成为根日志记录器对应的配置。 配置的处理方式将与所有日志记录器一致，除了 `propagate` 设置将不可用之外。

- *incremental* - 配置是否要被解读为在现有配置上新增。 该值默认为 `False`，这意味着指定的配置将以与当前 `fileConfig()` API 所使用的相同语义来替代现有的配置。

  如果指定的值为 `True`，配置会按照 增量配置 部分所描述的方式来处理。

- *disable_existing_loggers* - 是否要禁用任何现有的非根日志记录器。 该设置对应于 `fileConfig()` 中的同名形参。 如果省略，则此形参默认为 `True`。 如果 *incremental* 为 `True` 则该省会被忽略。

增量配置

为增量配置提供完全的灵活性是很困难的。 例如，由于过滤器和格式化器这样的对象是匿名的，一旦完成配置，在增加配置时就不可能引用这些匿名对象。

此外，一旦完成了配置，在运行时任意改变日志记录器、处理程序、过滤器、格式化器的对象图就不是很有必要；日志记录器和处理程序的详细程度只需通过设置级别即可实现控制（对于日志记录器则可设置传播旗标）。 在多线程环境中以安全的方式任意改变对象图也许会导致问题；虽然并非不可能，但这样做的好处不足以抵销其所增加的实现复杂度。

这样，当配置字典的 `incremental` 键存在且为 `True` 时，系统将完全忽略任何 `formatters` 和 `filters` 条目，并仅会处理 `handlers` 条目中的 `level` 设置，以及 `loggers` 和 `root` 条目中的 `level` 和 `propagate` 设置。

使用配置字典中的值可让配置以封存字典对象的形式通过线路传送给套接字监听器。 这样，长时间运行的应用程序的日志记录的详细程度可随时间改变而无须停止并重新启动应用程序。

对象连接

该架构描述了一组日志记录对象 —— 日志记录器、处理程序、格式化器、过滤器 —— 它们在对象图中彼此连接。 因此，该架构需要能表示对象之间的连接。 例如，在配置完成后，一个特定的日志记录器关联到了一个特定的处理程序。 出于讨论的目的，我们可以说该日志记录器代表两者间连接的源头，而处理程序则代表对应的目标。 当然在已配置对象中这是由包含对处理程序的引用的日志记录器来代表的。 在配置字典中，这是通过给每个目标对象一个 ID 来无歧义地标识它，然后在源头对象中使用该 ID 来实现的。

因此，举例来说，考虑以下 YAML 代码段:

```
formatters:
  brief:
    # configuration for formatter with id 'brief' goes here
  precise:
    # configuration for formatter with id 'precise' goes here
handlers:
  h1: #This is an id
   # configuration of handler with id 'h1' goes here
   formatter: brief
  h2: #This is another id
   # configuration of handler with id 'h2' goes here
   formatter: precise
loggers:
  foo.bar.baz:
    # other configuration for logger 'foo.bar.baz'
    handlers: [h1, h2]
```

（注：这里使用 YAML 是因为它的可读性比表示字典的等价 Python 源码形式更好。）

日志记录器 ID 就是日志记录器的名称，它会在程序中被用来获取对日志记录器的引用，例如 `foo.bar.baz` 。 格式化器和过滤器的 ID 可以是任意字符串值 (例如上面的 `brief` , `precise`) 并且它们是瞬态的，因为它们仅对处理配置字典有意义并会被用来确定对象之间的连接，而当配置调用完成时不会在任何地方保留。

上面的代码片段指明名为 `foo.bar.baz` 的日志记录器应当关联到两个处理程序，它们的 ID 是 `h1` 和 `h2` 。 `h1` 的格式化器的 ID 是 `brief` ，而 `h2` 的格式化器的 ID 是 `precise` 。

用户定义对象

此架构支持用户定义对象作为处理程序、过滤器和格式化器。 （日志记录器的不同实例不需要具有不同类型，因此这个配置架构并不支持用户定义日志记录器类。）

字典描述了要配置的对象，并详细说明了它们的配置。在某些地方，日志记录系统将能够从上下文中推断出如何实例化对象，但是当要实例化用户定义的对象时，系统将不知道如何执行此操作。为了为用户定义的对象实例化提供完全的灵活性，用户需要提供一个“工厂”-可调用对象，该对象可通过配置字典进行调用并返回实例化的对象。这可以通过特殊键下的工厂绝对导入路径来表示 `'()'` 。这是一个具体的例子:

```
formatters:
  brief:
    format: '%(message)s'
  default:
    format: '%(asctime)s %(levelname)-8s %(name)-15s %(message)s'
    datefmt: '%Y-%m-%d %H:%M:%S'
  custom:
      (): my.package.customFormatterFactory
      bar: baz
      spam: 99.9
      answer: 42
```

上面的YAML代码段定义了三个格式化程序。第一个ID `brief` 为id 是 `logging.Formatter` 具有指定格式字符串的标准实例。第二个ID为id `default` ，具有更长的格式，还显式定义了时间格式，并将导致 `logging.Formatter` 使用这两个格式字符串进行初始化。 `brief` 和 `default` 格式化程序以Python源代码形式显示，具有配置子词典:

```
{
  'format' : '%(message)s'
}
```

和:

```
{
  'format' : '%(asctime)s %(levelname)-8s %(name)-15s %(message)s',
  'datefmt' : '%Y-%m-%d %H:%M:%S'
}
```

并且，由于这些词典不包含特殊键 `'()'` ，因此从上下文中推断出实例化：因此， `logging.Formatter` 创建了标准实例。id为的第三个格式化程序的配置子字典为 `custom` :

```
{
  '()' : 'my.package.customFormatterFactory',
  'bar' : 'baz',
  'spam' : 99.9,
  'answer' : 42
}
```

并且其中如果包含特殊键 `'()'` ，这意味着需要用户定义的实例化。在这种情况下，将使用指定的工厂可调用对象。如果它是一个实际的可调用对象，则将直接使用它-否则，如果您指定一个字符串（如示例中所示），则将使用常规的导入机制来定位该实际的可调用对象。将使用配置子词典中的**其余**项作为关键字参数来调用可调用对象。在上面的示例中，具有ID的格式化程序 `custom` 将被假定为通过调用返回：

```
my.package.customFormatterFactory(bar='baz', spam=99.9, answer=42)
```

该键 `'()'` 已用作特殊键，因为它不是有效的关键字参数名称，因此不会与调用中使用的关键字参数名称冲突。在 `'()'` 还用作助记符相应值是一个可调用。

访问外部对象

例如，有时配置需要引用配置外部的对象 `sys.stderr` 。如果配置命令是使用Python代码构造的，这很简单，但是当通过文本文件（例如JSON，YAML）提供配置时会出现问题。在文本文件中，没有标准的方法来区别 `sys.stderr` 文字字符串 `'sys.stderr'` 。为了促进这种区分，配置系统会在字符串值中查找某些特殊前缀，并对其进行特殊处理。例如，如果 `'ext://sys.stderr'` 在配置中将文字字符串作为值提供，则将 `ext://` 剥离字符串 ，并使用常规导入机制处理值的其余部分。

此类前缀的处理方式类似于协议处理：存在一种通用机制来查找与正则表达式匹配的前缀， `^(?P<prefix>[a-z]+)://(?P<suffix>.*)$` 从而，如果 `prefix` 识别出， `suffix` 则以与前缀相关的方式处理，并得到结果。处理将替换字符串值。如果未识别前缀，则字符串值将保持不变。

访问内部对象

除外部对象外，有时还需要引用配置中的对象。配置系统将针对它所了解的内容隐式完成此操作。例如，字符串值 `'DEBUG'` 用于 `level` 在一个记录器或处理程序将自动转换为值 `logging.DEBUG` ，并且 `handlers` ， `filters` 和 `formatter` 条目将一个对象ID和决心到适当的目的地的对象。

但是，对于 `logging` 模块不知道的用户定义对象，需要一种更通用的机制。例如，考虑 `logging.handlers.MemoryHandler` ，它接受一个 `target` 参数，该参数是另一个要委托给的处理程序。由于系统已经知道此类，因此在配置中，给定的 `target` 只是相关目标处理程序的对象ID，系统将从该ID解析为处理程序。但是，如果用户定义 `my.package.MyHandler` 具有 `alternate` 处理程序的，则配置系统将不知道所 `alternate` 引用的处理程序。为此，通用解析系统允许用户指定：

```
handlers:
  file:
    # configuration of file handler goes here

  custom:
    (): my.package.MyHandler
    alternate: cfg://handlers.file
```

文字字符串 `'cfg://handlers.file'` 将以类似于带有 `ext://` 前缀的字符串的方式解析，但查找的是配置本身而不是导入名称空间。该机制允许以点或索引的方式访问，类似于所提供的 `str.format` 。因此，给出以下代码段：

```
handlers:
  email:
    class: logging.handlers.SMTPHandler
    mailhost: localhost
    fromaddr: my_app@domain.tld
    toaddrs:
      - support_team@domain.tld
      - dev_team@domain.tld
    subject: Houston, we have a problem.
```

在配置方面，该字符串 `'cfg://handlers'` 将解析为与关键的字典 `handlers` ，字符串 `'cfg://handlers.email` 将解析为与关键的字典 `email` 中的 `handlers` 字典，等等。字符串 `'cfg://handlers.email.toaddrs[1]` 将解析为 `'dev_team.domain.tld'` ，字符串 `'cfg://handlers.email.toaddrs[0]'` 将解析为value `'support_team@domain.tld'` 。该 `subject` 值可以使用任何访问 `'cfg://handlers.email.subject'` 或者等价地， `'cfg://handlers.email[subject]'` 。仅当键包含空格或非字母数字字符时，才需要使用后一种形式。如果索引值仅由十进制数字组成，则将尝试使用相应的整数值进行访问，如果需要，则返回到字符串值。

给定一个字符串 `cfg://handlers.myhandler.mykey.123` ，它将解决 `config_dict['handlers']['myhandler']['mykey']['123']` 。如果将字符串指定为 `cfg://handlers.myhandler.mykey[123]` ，则系统将尝试从中检索值 `config_dict['handlers']['myhandler']['mykey'][123]` ， `config_dict['handlers']['myhandler']['mykey']['123']` 如果失败则退回到该值 。

导入分辨率和自定义导入器

默认情况下，导入解析使用内置 `__import__()` 函数进行导入。您可能希望用自己的导入机制替换它：如果是这样，则可以替换或其超类（即 类）的 `importer` 属性 。但是，由于需要通过描述符从类访问函数的方式，因此需要小心。如果您使用可调用的Python进行导入，并且想在类级别而不是实例级别定义它，则需要使用进行包装。例如： `DictConfigurator``BaseConfigurator staticmethod()`

```
from importlib import import_module
from logging.config import BaseConfigurator

BaseConfigurator.importer = staticmethod(import_module)
```

`staticmethod()` 如果您要在配置程序实例上设置import可调用，则无需包装。

配置文件格式

理解的配置文件格式 `fileConfig()` 基于 `configparser` 功能。该文件必须包含称为的部分 `[loggers]`， `[handlers]` 并 `[formatters]` 通过名称标识文件中定义的每种类型的实体。对于每个这样的实体，都有单独的部分标识该实体的配置方式。因此，对于 `log01` 在该 `[loggers]` 部分中命名的记录器，相关的配置详细信息保留在一节中 `[logger_log01]`。同样， `hand01` 在 `[handlers]` 区段中调用的处理程序将在名为的区段中保存其配置 `[handler_hand01]`，而在区段中调用的格式化程序将 `form01` 在称为的 `[formatters]` 节中指定其配置。 `[formatter_form01]`。根记录器配置必须在一节中指定 `[logger_root]`。

> 注解: 该 `fileConfig()` API早于 `dictConfig()` API，并且不提供覆盖日志记录某些方面的功能。例如，您不能使用来配置 `Filter` 对象，该对象提供了对超出简单整数级别的消息的过滤 `fileConfig()`。如果需要 `Filter` 在日志记录配置中包含的实例，则需要使用 `dictConfig()`。请注意，将来会在中添加对配置功能的增强功能 `dictConfig()`，因此值得在方便时考虑过渡到此较新的API。

文件中这些部分的示例如下。

```
[loggers]
keys=root,log02,log03,log04,log05,log06,log07

[handlers]
keys=hand01,hand02,hand03,hand04,hand05,hand06,hand07,hand08,hand09

[formatters]
keys=form01,form02,form03,form04,form05,form06,form07,form08,form09
```

根记录器必须指定级别和处理程序列表。下面给出了根记录器部分的示例。

```
[logger_root]
level=NOTSET
handlers=hand01
```

该 `level` 条目可以是一个或 。仅对于root记录器，意味着将记录所有消息。级别值在 包的命名空间的上下文中使用。 `DEBUG, INFO, WARNING, ERROR, CRITICAL``NOTSET``NOTSET eval() logging`

该 `handlers` 条目是处理程序名称的逗号分隔列表，该列表必须出现在此 `[handlers]` 部分中。这些名称必须出现在该 `[handlers]` 部分中，并且在配置文件中具有相应的部分。

对于除根记录器之外的其他记录器，还需要一些其他信息。下面的示例对此进行了说明。

```
[logger_parser]
level=DEBUG
handlers=hand01
propagate=1
qualname=compiler.parser
```

的 `level` 和 `handlers` 如果非根记录的电平被指定为条目解释为根记录器，不同之处在于 `NOTSET`，越往上层次结构中的系统参考记录器，以确定记录器的有效电平。该 `propagate` 条目设置为1表示消息必须从此记录器传播到更高级别的记录程序，或者设置为0表示消息**不**传播到该层次结构的处理器。该 `qualname` 条目是记录器的分层通道名称，即应用程序用来获取记录器的名称。

指定处理程序配置的部分如下所示。

```
[handler_hand01]
class=StreamHandler
level=NOTSET
formatter=form01
args=(sys.stdout,)
```

该 `class` 条目指示处理程序的类（如通过 `eval()` 在 `logging` 包的命名空间）。将 `level` 被解释为记录仪，并 `NOTSET` 认为是指“日志一切”。

该 `formatter` 条目指示此处理程序的格式化程序的键名。如果为空白， `logging._defaultFormatter` 则使用默认格式器 （）。如果指定了名称，则该名称必须出现在该 `[formatters]` 部分中，并且在配置文件中具有相应的部分。

在 包的名称空间的上下文中使用 `args` 时，该条目是处理程序类的构造函数的参数列表。请参阅相关处理程序的构造函数，或参考以下示例，以了解如何构造典型条目。如果未提供，则默认为。 `eval() logging``()`

在包的名称空间的上下文中使用 `kwargs` 时，可选条目是处理程序类的构造函数的关键字参数dict。如果未提供，则默认为。 `eval() logging``{}`

```
[handler_hand02]
class=FileHandler
level=DEBUG
formatter=form02
args=('python.log', 'w')

[handler_hand03]
```

```
class=handlers.SocketHandler
level=INFO
formatter=form03
args=('localhost', handlers.DEFAULT_TCP_LOGGING_PORT)

[handler_hand04]
class=handlers.DatagramHandler
level=WARN
formatter=form04
args=('localhost', handlers.DEFAULT_UDP_LOGGING_PORT)

[handler_hand05]
class=handlers.SysLogHandler
level=ERROR
formatter=form05
args=(('localhost', handlers.SYSLOG_UDP_PORT), handlers.SysLogHandler.LOG_USER)

[handler_hand06]
class=handlers.NTEventLogHandler
level=CRITICAL
formatter=form06
args=('Python Application', '', 'Application')

[handler_hand07]
class=handlers.SMTPHandler
level=WARN
formatter=form07
args=('localhost', 'from@abc', ['user1@abc', 'user2@xyz'], 'Logger Subject')
kwargs={'timeout': 10.0}

[handler_hand08]
class=handlers.MemoryHandler
level=NOTSET
formatter=form08
target=
args=(10, ERROR)

[handler_hand09]
class=handlers.HTTPHandler
level=NOTSET
formatter=form09
args=('localhost:9022', '/log', 'GET')
kwargs={'secure': True}
```

指定格式化程序配置的部分由以下内容代表。

```
[formatter_form01]
format=F1 %(asctime)s %(levelname)s %(message)s
datefmt=
class=logging.Formatter
```

该 `format` 条目是整体格式字符串，而该 `datefmt` 条目是 `strftime()` -compatible日期/时间格式字符串。如果为空，则包将替换几乎等同于指定日期格式字符串的内容。此格式还指定毫秒，并使用逗号分隔符将其附加到使用上述格式字符串的结果中。这种格式的示例时间为。 `'%Y-%m-%d %H:%M:%S'``2003-01-23 00:29:50,411`

该 `class` 条目是可选的。它指示格式化程序的类的名称（作为点缀的模块和类名称。）此选项对于实例化子 `Formatter` 类很有用 。的子类 `Formatter` 可以扩展或压缩格式显示异常回溯。

> 注意: 由于 `eval()` 如上所述的使用，使用 `listen()` 通过套接字发送和接收配置会导致潜在的安全风险。风险仅限于没有相互信任的多个用户在同一台计算机上运行代码的情况； `listen()` 有关更多信息，请参见 文档。

简单使用帮助

```
import logging

# NOTSET (0) 、DEBUG (10) 、INFO (20) 、WARNING (30) 、ERROR (40) 、CRITICAL (50
logging.basicConfig(level=logging.DEBUG, format="%(asctime)s %(name)s:%(levelname)10s:%(message)s", datefmt="%d-%M-%Y %H:%M:%S")

```

你也可以看看其他帮助：

- 模块 `logging`

  日志记录模块的 API 参考。

- `logging.handlers` 模块

日志记录模块附带的有用处理器。