1. 特性

1.1 开箱即用

Loguru的主要概念**有且仅有**一个 `logger` 。

为了方便起见，它是预先配置的，并开始输出到 `stderr` （但这是完全可配置的）。

```
from loguru import logger

logger.debug("That's it, beautiful and simple logging!")
```

`logger` 只是一个将日志消息分派给已配置的处理器的接口。就是如此简单。

1.2 没有处理器，没有格式化器，没有过滤器：仅使用一个函数即可

如何添加处理器？如何设置日志格式？如何过滤消息？如何设置级别？

答案只有一个，即 `add()` 函数。

```
logger.add(sys.stderr, format="{time} {level} {message}", filter="my_module",
level="INFO")
```

此函数应该用于注册 接收器，这些接收器负责管理与记录字典关联的日志消息。 接收器可以采用多种形式：简单函数，字符串路径，类文件对象，协程函数或内置处理器。

请注意，还可以使用添加时返回的标识符来 `remove()` 先前添加的处理器。 如果要取代默认的 `stderr` 处理器，这将特别有用：只需调用 `logger.remove()` 即可重新开始。

1.3 使得文件记录 轮换/保留/压缩 更容易

如果要将日志消息发送到文件，则只需使用字符串路径作为接收器。 为了方便起见，它也可以自动计时：

```
logger.add("file_{time}.log")
```

如果需要轮换记录器、删除较旧的日志或希望在关闭时压缩文件，也可以轻松配置。

```
logger.add("file_1.log", rotation="500 MB")    # 自动轮换过大的文件
logger.add("file_2.log", rotation="12:00")     # 每天中午创建新文件
logger.add("file_3.log", rotation="1 week")    # 一旦文件太旧，则轮换

logger.add("file_X.log", retention="10 days")  # 某个时间段后自动清理

logger.add("file_Y.log", compression="zip")    # 压缩以节省空间
```

1.4 使用大括号样式的现代字符串格式化

Loguru倾向于使用更优雅，更强大的 `{}` 格式，而不是 `%` ，日志记录函数实际上等效于 `str.format()` 。

```
logger.info("If you're using Python {}, prefer {feature} of course!", 3.6,
feature="f-strings")
```

1.5 在线程或主线程中捕获异常

您是否曾经看到程序意外崩溃而没有在日志文件中看到任何内容？ 您是否注意到没有记录线程中发生的异常？ 可以使用 `catch()` 装饰器/上下文管理器解决此问题，它可以确保将任何错误正确传播到 `logger` 。

```
@logger.catch
def my_function(x, y, z):
    # An error? It's caught anyway!
    return 1 / (x + y + z)
```

1.6 带有颜色的漂亮的日志记录

如果您的终端兼容，Loguru会自动为日志添加颜色。 您可以使用接收器格式中的标记标签来定义自己喜欢的样式。

```
logger.add(sys.stdout, colorize=True, format="<green>{time}</green> <level>{message}
</level>")
```

1.7 异步，线程安全，多进程安全

所有添加到 `logger` 的接收器默认都是线程安全的，但不是多进程安全的。可以将消息放入队列来确保日志的完整性。如果要异步记录，也可以使用相同的参数。

```
logger.add("somefile.log", enqueue=True)
```

还支持将协程函数用作接收器，并且应该使用 `complete()` 等待。

1.8 完整的描述性异常

记录代码中发生的异常对于跟踪错误很重要，但是如果您不知道失败原因，则记录日志将毫无用处。 Loguru通过允许显示整个堆栈跟踪信息（包括变量的值）来帮助您发现问题（多亏了 `better_exceptions` ！）。

代码：

```
logger.add("out.log", backtrace=True, diagnose=True)  # 请注意，在生产环境，可能会泄漏敏感数据

def func(a, b):
    return a / b

def nested(c):
    try:
        func(5, c)
    except ZeroDivisionError:
        logger.exception("What?!")

nested(0)
```

的输出如下：

```
2018-07-17 01:38:43.975 | ERROR    | __main__:nested:10 - What?!
Traceback (most recent call last):

  File "test.py", line 12, in <module>
    nested(0)
    └ <function nested at 0x7f5c755322f0>

> File "test.py", line 8, in nested
    func(5, c)
    │       └ 0
```

```
    └ <function func at 0x7f5c79fc2e18>

  File "test.py", line 4, in func
    return a / b
           |   └ 0
           └ 5

ZeroDivisionError: division by zero
```

1.9 根据需要进行结构化日志记录

希望对日志进行序列化以便于解析或传递日志？ 使用 `serialize` 参数，每条日志消息在发送到已配置的接收器之前都将转换为JSON字符串。

```
logger.add(custom_sink_function, serialize=True)
```

使用 `bind()` 可以通过修改 *extra* 记录属性来使记录器消息上下文化。

可以使用 `contextualize()` 临时修改上下文本地状态：

```
with logger.contextualize(task=task_id):
    do_something()
    logger.info("End of task")
```

通过结合 `bind()` 和 `filter`，还可以对日志进行更细粒度的控制：

```
logger.add("special.log", filter=lambda record: "special" in record["extra"])
logger.debug("This message is not logged to the file")
logger.bind(special=True).info("This message, though, is logged to the file!")
```

最后， `patch()` 方法允许将动态值附加到每个新消息的记录字典上：

```
logger.add(sys.stderr, format="{extra[utc]} {message}")
logger = logger.patch(lambda record: record["extra"].update(utc=datetime.utcnow()))
```

1.10 延迟计算昂贵的函数

有时您希望记录详细信息而不会影响生产性能，可以使用 `opt()` 方法来实现。

```
logger.opt(lazy=True).debug("If sink level <= DEBUG: {x}", x=lambda:
expensive_function(2**64))

# By the way, "opt()" serves many usages
logger.opt(exception=True).info("Error stacktrace added to the log message (tuple
accepted too)")
logger.opt(colors=True).info("Per message <blue>colors</blue>")
logger.opt(record=True).info("Display values from the record (eg.
{record[thread]})")
logger.opt(raw=True).info("Bypass sink formatting\n")
logger.opt(depth=1).info("Use parent stack context (useful within wrapped
functions)")
logger.opt(capture=False).info("Keyword arguments not added to {dest} dict",
dest="extra")
```

1.11 可定制的日志级别

Loguru附带了所有标准日志记录级别，并向其中添加了 `trace()` 和 `success()` 。如果需要更多级别，只需使用 `level()` 函数进行创建。

```
new_level = logger.level("SNAKY", no=38, color="<yellow>", icon="🐍")

logger.log("SNAKY", "Here we go!")
```

1.12 更好的日期时间处理

标准日志记录模块中用于处理日期时间的参数比较繁杂、格式不够直观，并且使用的是不带时区信息的本地日期时间。Loguru对其进行了修复：

```
logger.add("file.log", format="{time:YYYY-MM-DD at HH:mm:ss} | {level} | {message}")
```

1.13 适合脚本和库

在脚本中使用记录器很容易，您可以在开始时对其进行 `configure()` 。要从库内部使用Loguru，请记住不要调用 `add()` ，而要使用 `disable()` ，这样日志函数就变成了无操作。如果开发人员希望查看库的日志，则可以再次使用 `enable()` 来启用。

```
# For scripts
config = {
    "handlers": [
        {"sink": sys.stdout, "format": "{time} - {message}"},
        {"sink": "file.log", "serialize": True},
    ],
    "extra": {"user": "someone"}
}
logger.configure(**config)

# For libraries
logger.disable("my_library")
logger.info("No matter added sinks, this message is not displayed")
logger.enable("my_library")
logger.info("This message however is propagated to the sinks")
```

1.14 与标准日志模块兼容

如果要使用内置的日志记录 `Handler` 作为 Loguru 的接收器，可以下面这样：

```
handler = logging.handlers.SysLogHandler(address=('localhost', 514))
logger.add(handler)
```

要将 Loguru 消息传播到标准日志记录：

```
class PropagateHandler(logging.Handler):
    def emit(self, record):
        logging.getLogger(record.name).handle(record)

logger.add(PropagateHandler(), format="{message}")

```

想要拦截到Loguru接收器的标准日志消息?

```python
class InterceptHandler(logging.Handler):
    def emit(self, record):
        # Get corresponding Loguru level if it exists
        try:
            level = logger.level(record.levelname).name
        except ValueError:
            level = record.levelno

        # Find caller from where originated the logged message
        frame, depth = logging.currentframe(), 2
        while frame.f_code.co_filename == logging.__file__:
            frame = frame.f_back
            depth += 1

        logger.opt(depth=depth, exception=record.exc_info).log(level,
record.getMessage())

logging.basicConfig(handlers=[InterceptHandler()], level=0)
```

1.15 通过环境变量进行个性化的默认设置

不喜欢默认的记录器格式吗？ 想要其他 `DEBUG` 颜色吗？ 没问题：

```
# Linux / OSX
export LOGURU_FORMAT="{time} | <lvl>{message}</lvl>"

# Windows
setx LOGURU_DEBUG_COLOR "<green>"
```

1.16 方便的解析器

从生成的日志中提取特定信息通常很有用，这就是为什么Loguru提供了 `parse()` 方法来帮助处理日志和正则表达式的原因。

```
pattern = r"(?P<time>.*) - (?P<level>[0-9]+) - (?P<message>.*)"  # Regex with named groups
caster_dict = dict(time=dateutil.parser.parse, level=int)        # Transform matching groups

for groups in logger.parse("file.log", pattern, cast=caster_dict):
    print("Parsed:", groups)
    # {"level": 30, "message": "Log example", "time": datetime(2018, 12, 09, 11, 23, 55)}
```

1.17 详尽的通知

Loguru可以轻松地与 notifiers（必须单独安装）结合使用，以在程序意外失败时接收电子邮件，或发送许多其他类型的通知。

```
import notifiers

params = {
    "username": "you@gmail.com",
    "password": "abc123",
    "to": "dest@gmail.com"
}

# Send a single notification
notifier = notifiers.get_notifier("gmail")
notifier.notify(message="The application is running!", **params)

# Be alerted on each error message
from notifiers.logging import NotificationHandler

handler = NotificationHandler("gmail", defaults=params)
logger.add(handler, level="ERROR")
```

1.18 比内置的日志模块块 10 倍

尽管在大多数情况下，日志记录对性能的影响可以忽略不计，但零成本的日志记录器将允许在任何地方使用它而无需过多担心。 在即将发布的版本中，Loguru的关键功能将以C语言实现，以实现最大速度。

2. 从logging迁移到loguru

2.1 logging 和 loguru 的基本区别

尽管loguru是"从头开始"编写的，并且在内部不依赖于标准库的 `logging` ，但是两个库都具有相同的目的：提供实现灵活的事件日志记录系统的功能。主要区别在于标准库的 `logging` 要求用户显式实例化命名的 `Logger` 并使用 `Handler` 、 `Formatter` 和 `Filter` 对其进行配置，而loguru尝试缩小配置步骤的数量。

除此之外，用法在全局上是相同的，一旦创建或导入了 `logger` 对象，就可以开始使用它来记录具有适当严重性的消息（ `logger.debug("Dev message")` ， `logger.warning("Danger")` 等等），然后将消息发送到已配置的处理程序。

对于标准库的 `logging` ，默认日志发送到 `sys.stderr` 而不是 `sys.stdout` 。 POSIX标准指定 `stderr` 是 "诊断输出" 的正确流。支持将日志记录到 `stderr` 的主要原因是，它避免将应用程序的实际输出与调试信息混合在一起。例如，考虑像 `python my_app.py | other_app` 这样的管道重定向，如果将日志发送到 `stdout` ，这将是不可能的。另一个主要优点是，Python在 `sys.stdout` 上引发 `UnicodeEncodeError` （"strict"策略）时，通过转义错误字符（"backslashreplace"策略）来解决 `sys.stderr` 上的编码问题。

2.2 替换 `getLogger()` 函数

通常在每个文件的开头调用 `getLogger()` 来检索和使用整个模块中的记录器，如下所示： `logger = logging.getLogger(__name__)` 。

使用Loguru，无需从loguru导入记录器就可以明确获取和命名记录器。 每次使用此导入的记录器时，都会创建一条记录，并将自动包含上下文 `__name__` 值。 对于标准库的 `logging` ，然后可以使用 `name` 属性来格式化和过滤日志。

2.3 替换 `Logger` 对象

Loguru通过适当的接收器定义替换了标准库的 `Logger` 配置。应该 `add()` 并参数化处理器，而不是配置日志记录器。 `setLevel()` 和 `addFilter()` 被配置的接收器 `level` 和 `filter` 参数抑制。 `propagate` 和 `disable()` 方法也可以由 `filter` 选项代替。 可以使用 `record ["extra"]` 字典替换 `makeRecord()` 方法。

有时，需要对特定记录器进行更细粒度的控制。 在这种情况下，Loguru提供了 `bind()` 方法，该方法尤其可以用于生成专门命名的记录器。 例如，通过调用 `other_logger = logger.bind(name="other")` ，使用 `other_logger` 记录的每条消息都将使用 `name` 值填充 `record ["extra"]` 字典，而使用 `logger` 则不会。 这样可以从接收器或过滤器函数中将日志与 `logger` 或 `other_logger` 区别开来。

假设您希望接收器仅记录一些非常具体的消息：

```python
def specific_only(record):
    return "specific" in record["extra"]

logger.add("specific.log", filter=specific_only)

specific_logger = logger.bind(specific=True)

logger.info("General message")          # This is filtered-out by the specific sink
specific_logger.info("Module message")  # This is accepted by the specific sink (and others)
```

另一个例子，如果你想把一个接收器连接到一个命名的日志记录器：

```python
# Only write messages from "a" logger
logger.add("a.log", filter=lambda record: record["extra"].get("name") == "a")
# Only write messages from "b" logger
logger.add("b.log", filter=lambda record: record["extra"].get("name") == "b")

logger_a = logger.bind(name="a")
logger_b = logger.bind(name="b")

logger_a.info("Message A")
logger_b.info("Message B")
```

2.4 替换 `Handler`、`Filter` 和 `Formatter` 对象

标准库的 `logging` 需要您创建一个 `Handler` 对象，然后调用 `addHandler()` 方法。使用Loguru，将使用 `add()` 启动处理程序。 接收器定义处理程序应如何处理传入的日志记录消息，就像 `handle()` 或 `emit()` 一样。 要从多个模块进行日志记录，只需导入记录器，所有消息都将分派到添加的处理程序中。

在调用 `add()` 时， `level` 参数将替换 `setLevel()` ， `format` 参数将替换 `setFormatter()` ， `filter` 参数将替换 `addFilter()` 。线程安全由Loguru自动管理，因此不需要 `createLock()` ， `acquire()` 或 `release()` 。 `removeHandler()` 的等效方法是 `remove()` ，该方法应与 `add()` 返回的标识符一起使用。

请注意，不必替换 `Handler` 对象，因为 `add()` 接受它们作为有效接收器。

简单来说，可以替换：

```python
logger.setLevel(logging.DEBUG)

```

```
fh = logging.FileHandler("spam.log")
fh.setLevel(logging.DEBUG)

ch = logging.StreamHandler()
ch.setLevel(logging.ERROR)

formatter = logging.Formatter("%(asctime)s - %(name)s - %(levelname)s - %(message)s")
fh.setFormatter(formatter)
ch.setFormatter(formatter)

logger.addHandler(fh)
logger.addHandler(ch)
```

为：

```
fmt = "{time} - {name} - {level} - {message}"
logger.add("spam.log", level="DEBUG", format=fmt)
logger.add(sys.stderr, level="ERROR", format=fmt)
```

2.5 替换 `%` 样式的消息格式化

Loguru 只支持 `{}` 样式的格式化。

必须使用 `logger.debug("Some variable: {}", var)` 替换 `logger.debug("Some variable: %s", var)`。传递给 `logging` 函数的所有的 `*args` 和 `**kwargs` 均用于调用 `message.format(*args, **kwargs)`。没有出现在消息字符串中的参数将被忽略。请注意，将参数传递给这样的日志记录函数可能对（稍微）提高性能很有用：如果级别太低而无法传递任何已配置的处理程序，则可以避免格式化消息。

要转换 `Formatter` 使用的通用格式，请参阅 可用记录标记的列表。

要转换 `datefmt` 使用的日期格式，请参阅 可用日期标记列表。

2.6 替换 `exc_info` 参数

在调用标准库的日志记录函数时，您可以传递 `exc_info` 作为参数，以将栈跟踪信息添加到消息中。取而代之的是，您应该使用带有 `exception` 参数的 `opt()` 方法，使用 `logger.opt(exception=True).debug("Debug error:")` 替换 `logger.debug("Debug error:", exc_info=True)`。

格式化的异常将包括整个栈跟踪信息和变量。为了防止这种情况，请确保在添加接收器时使用 `backtrace=False` 和 `diagnostic=False`。

2.7 替换 `extra` 参数和 `LoggerAdapter` 对象

要将上下文信息传递到日志消息，请通过内联 `bind()` 方法替换 `extra` 内容：

```
context = {"clientip": "192.168.0.1", "user": "fbloggs"}

logger.info("Protocol problem", extra=context)   # Standard logging
logger.bind(**context).info("Protocol problem")  # Loguru
```

这会将上下文信息添加到已记录消息的 `record["extra"]` 字典中，因此请确保适当配置处理程序格式：

```
fmt = "%(asctime)s %(clientip)s %(user)s %(message)s"     # Standard logging
fmt = "{time} {extra[clientip]} {extra[user]} {message}"  # Loguru
```

您也可以在使用 `LoggerAdapter` 之前通过调用 `logger = logger.bind(clientip="192.168.0.1")` 或将绑定的日志记录器分配给类实例来替换 `LoggerAdapter` ：

```
class MyClass:

    def __init__(self, clientip):
        self.logger = logger.bind(clientip=clientip)

    def func(self):
        self.logger.debug("Running func")
```

2.8 替换 `isenabledFor` 方法

如果您希望为调试日志记录有用的信息，但又不想在没有配置调试处理程序的情况下以release模式付出性能损失，则标准日志记录提供了 `isEnabledFor()` 方法：

```
if logger.isEnabledFor(logging.DEBUG):
    logger.debug("Message data: %s", expensive_func())
```

您可以将其替换为 `opt()` 方法和 `lazy` 选项：

```
# Arguments should be functions which will be called if needed
logger.opt(lazy=True).debug("Message data: {}", expensive_func)
```

2.9 替换 `addLevelName()` 和 `getLevelName()` 函数

要添加自定义级别，可以使用 `level()` 替换 `addLevelName()` 。

```
logging.addLevelName(33, "CUSTOM")                      # Standard logging
logger.level("CUSTOM", no=45, color="<red>", icon="🚨")  # Loguru
```

也可以使用 `level()` 替换 `getLevelName()` 。

```
logger.getLevelName(33)  # => "CUSTOM"
logger.level("CUSTOM")   # => (name='CUSTOM', no=33, color="<red>", icon="🚨")
```

请注意，与标准库的 `logging` 相反，Loguru不会将严重性编号与任何级别相关联，级别只能通过其名称来识别。

2.10 替换 `basicConfig()` 和 `dictConfig()` 函数

`basicConfig()` 和 `dictConfig()` 函数由 `configure()` 方法代替。

但是，它不接受 `config.ini` 文件，因此您必须使用自己喜欢的格式自行处理。

2.11 将 loguru与 Pytest 和 caplog一起使用

pytest 是一个常用的测试框架。 `caplog` 夹具捕获日志记录输出，以便可以对其进行测试。 例如：

```
# `some_func` adds two numbers, and logs a warning if the first is < 1
def test_some_func_logs_warning(caplog):
    assert some_func(-1, 3) == 2
    assert "Oh no!" in caplog.text
```

如果您到目前为止已遵循所有迁移准则，则会注意到该测试将失败。 这是因为pytest链接到标准库的 `logging` 模块。

因此，要解决问题，我们需要添加一个接收器，以将Loguru传播到 `logging` 。 这是通过对 `caplog` 使用猴子补丁在夹具本身上完成的。 在您的 `conftest.py` 文件中，添加以下内容：

```python
import logging
import pytest
from _pytest.logging import caplog as _caplog
from loguru import logger

@pytest.fixture
def caplog(_caplog):
    class PropogateHandler(logging.Handler):
        def emit(self, record):
            logging.getLogger(record.name).handle(record)

    handler_id = logger.add(PropogateHandler(), format="{message} {extra}")
    yield _caplog
    logger.remove(handler_id)
```

运行测试，一切都将按预期工作。 附加信息可以在 GH#59 中找到。

3. 代码片段和示例

3.1 更改现存的处理器的级别

添加处理程序后，实际上是不可能对其进行更新的。 为了尽量减少Loguru的API，这是一个有意的选择。 如果您需要更改处理程序的配置级别，则可能有几种解决方案，很难解决。 选择最适合您的用例的那个。

最直接的解决方法是 `remove()` 您的处理程序，然后使用更新的 `level` 参数将其重新 `add()` 。 为此，您必须保留对添加处理程序时返回的标识符编号的引用：

```python
handler_id = logger.add(sys.stderr, level="WARNING")

logger.info("Logging 'WARNING' or higher messages only")

...

logger.remove(handler_id)
logger.add(sys.stderr, level="DEBUG")

logger.debug("Logging 'DEBUG' messages too")
```

或者，您可以将 `bind()` 方法与 `filter` 参数结合起来，提供一个基于级别来动态过滤日志的函数：

```python
def my_filter(record):
    if record["extra"].get("warn_only"):  # "warn_only" is bound to the logger and set to 'True'
        return record["level"].no >= logger.level("WARNING").no
    return True  # Fallback to default 'level' configured while adding the handler


logger.add(sys.stderr, filter=my_filter, level="DEBUG")

# Use this logger first, debug messages are filtered out
logger = logger.bind(warn_only=True)
logger.warn("Initialization in progress")

# Then you can use this one to log all messages
logger = logger.bind(warn_only=False)
logger.debug("Back to debug messages")
```

最后，通过使用可调用对象作为过滤器，以实现对处理器级别的更高级控制：

```python
class MyFilter:

    def __init__(self, level):
        self.level = level

    def __call__(self, record):
        levelno = logger.level(self.level).no
        return record["level"].no >= levelno

my_filter = MyFilter("WARNING")
logger.add(sys.stderr, filter=my_filter, level=0)

logger.warning("OK")
logger.debug("NOK")

my_filter.level = "DEBUG"
logger.debug("OK")
```

3.2 在处理器或网络之间发送和接受日志消息

可以在不同进程之间，甚至在不同计算机之间传输日志(如果需要的话)。一旦在两个Python程序之间建立了连接，这就需要一边序列化日志记录，一边重新构造消息。

这可以使用客户端的自定义接收器和服务器的 `patch()` 来实现。

```python
# client.py
import sys
import socket
import struct
import time
import pickle

from loguru import logger


class SocketHandler:

    def __init__(self, host, port):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.connect((host, port))

    def write(self, message):
        record = message.record
        data = pickle.dumps(record)
        slen = struct.pack(">L", len(data))
        self.sock.send(slen + data)

logger.configure(handlers=[{"sink": SocketHandler('localhost', 9999)}])

while 1:
    time.sleep(1)
    logger.info("Sending message from the client")
```

```python
# server.py
import socketserver
import pickle
import struct

from loguru import logger


class LoggingStreamHandler(socketserver.StreamRequestHandler):
```

```
10
11      def handle(self):
12          while True:
13              chunk = self.connection.recv(4)
14              if len(chunk) < 4:
15                  break
16              slen = struct.unpack('>L', chunk)[0]
17              chunk = self.connection.recv(slen)
18              while len(chunk) < slen:
19                  chunk = chunk + self.connection.recv(slen - len(chunk))
20              record = pickle.loads(chunk)
21              level, message = record["level"], record["message"]
22              logger.patch(lambda record: record.update(record)).log(level, message)
23
24  server = socketserver.TCPServer(('localhost', 9999), LoggingStreamHandler)
25  server.serve_forever()
```

请记住，pickling是不安全的，使用时必须小心。

3.3 解决 `UnicodeEncodeError` 和其他编码问题

当您编写日志消息时，处理程序可能需要将接收到的unicode字符串编码为特定的字节序列。 用于执行此操作的编码取决于接收器类型和您的环境。 如果尝试编写处理程序编码不支持的字符，则可能会出现问题。 在这种情况下，Python可能会引发 `UnicodeEncodeError` 。

例如，这可能在打印到终端时发生：

```
1  print("天")
2  # UnicodeEncodeError: 'charmap' codec can't encode character '\u5929' in position 0:
   character maps to <undefined>
```

写入尚未使用适当编码打开的文件时，可能会发生类似的错误。 在将日志记录到标准输出或Windows上的文件时，会发生最常见的问题。 那么，如何避免这种错误呢？ 只需正确配置您的处理程序，使其可以处理任何类型的unicode字符串即可。

如果将日志记录到 `stdout` 时遇到此错误，则有以下几种选择：

- 使用 `sys.stderr` 代替 `sys.stdout` （前者会转义错误字符而不是引发异常）
- 将 `PYTHONIOENCODING` 环境变量设置为 `utf-8` 。
- 使用 `encoding='utf-8'` 和/或 `errors=backslashreplace` 调用 `sys.stdout.reconfigure()` 。

如果使用的是文件接收器，则可以在添加处理器时配置 `errors` 和 `encoding` 参数，例如 `logger.add("file.log", encoding="utf-8")` 。所有其他 `**kwargs*` 参数将会传递到内置的 `open()` 函数。

对于其他类型的处理器，必须检查是否有方法对编码或回退策略进行参数化。

3.4 用装饰器记录函数的进入和退出

在某些情况下，记录函数的进入和退出值可能很有用。 尽管Loguru并未提供现成的功能，但可以使用Python装饰器轻松实现：

```
import functools
from loguru import logger


def logger_wraps(*, entry=True, exit=True, level="DEBUG"):

    def wrapper(func):
        name = func.__name__

        @functools.wraps(func)
        def wrapped(*args, **kwargs):
            logger_ = logger.opt(depth=1)
            if entry:
                logger_.log(level, "Entering '{}' (args={}, kwargs={})", name, args,
kwargs)
            result = func(*args, **kwargs)
            if exit:
                logger_.log(level, "Exiting '{}' (result={})", name, result)
            return result

        return wrapped

    return wrapper
```

可以像下面这样使用上面定义的额装饰器：

```
@logger_wraps()
def foo(a, b, c):
    logger.info("Inside the function")
    return a * b * c

def bar():
    foo(2, 4, c=8)

bar()
```

输出如下：

```
2019-04-07 11:08:44.198 | DEBUG    | __main__:bar:30 - Entering 'foo' (args=(2, 4), kwargs={'c': 8})
2019-04-07 11:08:44.198 | INFO     | __main__:foo:26 - Inside the function
2019-04-07 11:08:44.198 | DEBUG    | __main__:bar:30 - Exiting 'foo' (result=64)
```

下面这个示例演示了使用上面定义的装饰器来记录一个函数的运行时间：

```
def timeit(func):

    def wrapped(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        logger.debug("Function '{}' executed in {:f} s", func.__name__, end - start)
        return result

    return wrapped
```

3.5 使用基于自定义添加级别的日志记录函数

添加新级别后，它通常与 `log()` 函数一起使用：

```
logger.level("foobar", no=33, icon="😀", color="<blue>")

logger.log("foobar", "A message")
```

为了方便起见，可以分配一个新的日志记录函数，该函数会自动使用添加的自定义级别：

```
from functools import partialmethod

logger.__class__.foobar = partialmethod(logger.__class__.log, "foobar")

logger.foobar("A message")
```

新方法只需添加一次，即可在导入 `logger` 的所有文件中使用。 将方法分配给 `logger.__class__` 而不是直接将其分配给 `logger`，以确保即使在调用 `logger.bind()`、`logger.patch()` 和 `logger.opt()` 之后该方法仍然可用（因为这些函数返回新的 `logger` 实例）。

3.6 为整个模块保留 `opt()` 参数

假设您希望为每个日志消息着色，而不必每次都调用 `logger.opt(colors=True)`，则可以在模块的最开始添加此代码：

```
logger = logger.opt(colors=True)

logger.info("It <green>works</>!")
```

但是，应注意，无法链式调用 `opt()`，再次使用此方法会将 `colors` 选项重置为其默认值（即 `False`）。 因此，还必须修补 `opt()` 方法，以便所有后续调用继续使用所需的值：

```
from functools import partial

logger = logger.opt(colors=True)
logger.opt = partial(logger.opt, colors=True)

logger.opt(raw=True).info("It <green>still</> works!\n")
```

3.7 使用自定义函数序列化日志消息

添加了 `serialize=True` 的每个处理程序将通过将日志记录转换为有效的JSON字符串来创建消息。 根据消息所指向的接收器，对生成的字符串进行更改可能会很有用。 您可以实现自己的序列化函数并直接在接收器中使用，而不必使用 `serialize` 参数：

```
def serialize(record):
    subset = {"timestamp": record["time"].timestamp(), "message": record["message"]}
    return json.dumps(subset)

def sink(message):
    serialized = serialize(message.record)
    print(serialized)

logger.add(sink)
```

如果需要将结构化日志发送到文件接收器（或一般而言，是任何类型的接收器），则可以通过使用自定义 `format` 函数来获得类似的结果：

```
def formatter(record):
    # Note this function returns the string to be formatted, not the actual message to be logged
    record["extra"]["serialized"] = serialize(record)
    return "{extra[serialized]}\n"

logger.add("file.log", format=formatter)
```

也可以使用 `patch()` 来实现此功能，因此，如果要在多个接收器中使用它，则只会调用一次序列化函数：

```
def patching(record):
    record["extra"]["serialized"] = serialize(record)

logger = logger.patch(patching)

# Note that if "format" is not a function, possible exception will be appended to the message
logger.add(sys.stderr, format="{extra[serialized]}")
logger.add("file.log", format="{extra[serialized]}")
```

3.8 基于大小和时间轮换日志文件

文件接收器的 `rotate` 参数接受大小或时间限制，但为简化起见，两者均不接受。 但是，可以创建自定义函数来支持更高级的方案：

```
import datetime
```

```
2
3   class Rotator:
4
5       def __init__(self, *, size, at):
6           now = datetime.datetime.now()
7
8           self._size_limit = size
9           self._time_limit = now.replace(hour=at.hour, minute=at.minute,
    second=at.second)
10
11          if now >= self._time_limit:
12              # The current time is already past the target time so it would rotate
    already.
13              # Add one day to prevent an immediate rotation.
14              self._time_limit += datetime.timedelta(days=1)
15
16      def should_rotate(self, message, file):
17          file.seek(0, 2)
18          if file.tell() + len(message) > self._size_limit:
19              return True
20          if message.record["time"].timestamp() > self._time_limit.timestamp():
21              self._time_limit += datetime.timedelta(days=1)
22              return True
23          return False
24
25  # Rotate file if over 500 MB or at midnight every day
26  rotator = Rotator(size=5e+8, at=datetime.time(0, 0, 0))
27  logger.add("file.log", rotation=rotator.should_rotate)
```

3.9 动态格式化消息以使值与填充正确对齐

默认格式化程序无法垂直对齐日志消息，因为 `{name}` 、 `{function}` 和 `{line}` 的长度不固定。 一种解决方法是使用具有大部分时间都足够的最大值的填充，例如：

```
1   fmt = "{time} | {level: <8} | {name: ^15} | {function: ^15} | {line: >3} |
    {message}"
2   logger.add(sys.stderr, format=fmt)
```

通过使用格式化函数或类，其他解决方案也是可能的。 例如，可以根据先前遇到的值动态调整填充长度：

```
1   class Formatter:
2
```

```
    def __init__(self):
        self.padding = 0
        self.fmt = "{time} | {level: <8} | {name}:{function}:{line}{extra[padding]}
| {message}\n{exception}"

    def format(self, record):
        length = len("{name}:{function}:{line}".format(**record))
        self.padding = max(self.padding, length)
        record["extra"]["padding"] = " " * (self.padding - length)
        return self.fmt

formatter = Formatter()

logger.remove()
logger.add(sys.stderr, format=formatter.format)
```

3.10 自定义异常格式

Loguru将在使用 `logger.exception()` 或 `logger.opt(exception=True)` 时自动添加发生异常的回溯：

```
def inverse(x):
    try:
        1 / x
    except ZeroDivisionError:
        logger.exception("Oups...")

if __name__ == "__main__":
    inverse(0)
```

```
2019-11-15 10:01:13.703 | ERROR    | __main__:inverse:8 - Oups...
Traceback (most recent call last):
File "foo.py", line 6, in inverse
    1 / x
ZeroDivisionError: division by zero
```

如果添加处理器时指定了 `backtrace=True`，则会扩展回溯以查看异常的来源：

```
2019-11-15 10:11:32.829 | ERROR    | __main__:inverse:8 - Oups...
Traceback (most recent call last):
  File "foo.py", line 16, in <module>
    inverse(0)
> File "foo.py", line 6, in inverse
    1 / x
ZeroDivisionError: division by zero
```

通过添加带有自定义 `format` 函数的处理程序，可以进一步个性化异常的格式。 例如，假设您要使用 stackprinter 库格式化错误：

```
import stackprinter

def format(record):
    format_ = "{time} {message}\n"

    if record["exception"] is not None:
        record["extra"]["stack"] = stackprinter.format(record["exception"])
        format_ += "{extra[stack]}\n"

    return format_

logger.add(sys.stderr, format=format)
```

```
2019-11-15T10:46:18.059964+0100 Oups...
File foo.py, line 17, in inverse
    15   def inverse(x):
    16       try:
--> 17           1 / x
    18       except ZeroDivisionError:
    ................................................
    x = 0
    ................................................

ZeroDivisionError: division by zero
```

3.11 在不使用错误上下文的情况下显示堆栈跟踪

有时，在未引发异常时的情况下记录消息并显示回溯可能会很有用。 尽管此功能不是Loguru内置的（因为与记录日志相比，其与调试更加相关），但是可以 `patch()` 记录器，然后根据需要显示堆栈跟踪（使用 traceback 模块）：

```
import traceback

def add_traceback(record):
    extra = record["extra"]
    if extra.get("with_traceback", False):
        extra["traceback"] = "\n" + "".join(traceback.format_stack())
    else:
        extra["traceback"] = ""

logger = logger.patch(add_traceback)
logger.add(sys.stderr, format="{time} - {message}{extra[traceback]}")

logger.info("No traceback")
logger.bind(with_traceback=True).info("With traceback")
```

这是另一个示例，演示如何在已记录的消息之前添加完整的调用堆栈：

```
import traceback
from itertools import takewhile

def tracing_formatter(record):
    # Filter out frames coming from Loguru internals
    frames = takewhile(lambda f: "/loguru/" not in f.filename,
traceback.extract_stack())
    stack = " > ".join("{}:{}:{}".format(f.filename, f.name, f.lineno) for f in
frames)
    record["extra"]["stack"] = stack
    return "{level} | {extra[stack]} - {message}\n{exception}"

def foo():
    logger.info("Deep call")

def bar():
    foo()

logger.remove()
logger.add(sys.stderr, format=tracing_formatter)

```

```
bar()
# Output: "INFO | script.py:<module>:23 > script.py:bar:18 > script.py:foo:15 - Deep call"
```

3.12 操作换行符以在同一行上写入多个日志

通过结合使用 `bind()` 、 `opt()` 和自定义 `format` 函数，可以将消息临时记录在连续的行上。 如果要说明进行中的分步过程，此功能将特别有用，例如：

```
def formatter(record):
    end = record["extra"].get("end", "\n")
    return "[{time}] {message}" + end + "{exception}"

logger.add(sys.stderr, format=formatter)
logger.add("foo.log", mode="w")

logger.bind(end="").debug("Progress: ")

for _ in range(5):
    logger.opt(raw=True).debug(".")

logger.opt(raw=True).debug("\n")

logger.info("Done")
```

```
[2020-03-26T22:47:01.708016+0100] Progress: .....
[2020-03-26T22:47:01.709031+0100] Done
```

但是请注意，根据使用的接收器，您可能会遇到不同的困难。日志记录并不总是适合于这种类型的最终用户消息。

3.13 捕获标准 `stdout` 、 `stderr` 和 `warnings`

日志记录的使用应优先于 `print()` ，但是，有时可能无法完全控制应用程序中执行的代码。 如果希望捕获标准输出，则可以通过 `contextlib.redirect_stdout()` 来使用自定义流对象禁止 `sys.stdout` （和 `sys.stderr` ）。 必须先删除默认的处理程序，并且不能在重定向后立即添加新的 `stdout` 接收器，否则将导致死锁（可以使用 `sys.__stdout__` ）：

```
import contextlib
import sys
```

```
from loguru import logger

class StreamToLogger:

    def __init__(self, level="INFO"):
        self._level = level

    def write(self, buffer):
        for line in buffer.rstrip().splitlines():
            logger.opt(depth=1).log(self._level, line.rstrip())

    def flush(self):
        pass

logger.remove()
logger.add(sys.__stdout__)

stream = StreamToLogger()
with contextlib.redirect_stdout(stream):
    print("Standard output is sent to added handlers.")
```

您还可以通过替换 `warnings.showwarning()` 来捕获应用程序发出的警告：

```
import warnings
from loguru import logger

showwarning_ = warnings.showwarning

def showwarning(message, *args, **kwargs):
    logger.warning(message)
    showwarning_(message, *args, **kwargs)

warnings.showwarning = showwarning
```

3.14 规避 `__name__` 值不存在的模块

Loguru使用全局变量 `__name__` 来确定所记录消息的来源。 但是，在非常特殊的情况下（例如某些Dask分布式环境），可能会发生此值未设置的情况。 在这种情况下，Loguru将使用 `None` 来代替。 这意味着，如果您要 `disable()` 来自此类特殊模块的消息，则必须显式调用 `logger.disable(None)` 。

处理 `filter` 属性时，也应该考虑这种情况。 由于缺少 `__name __` ，Loguru会将 `None` 值分配给 `record["name"]` 条目。 这也意味着，一旦在日志消息中进行了格式化， `{name}` 字段将等价于 `"None"` 。 这可以通过使用 `patch()` 从有故障的模块的内部手动覆盖 `record["name"]` 值来解决：

```
# If Loguru fails to retrieve the proper "name" value, assign it manually
logger = logger.patch(lambda record: record.update(name="my_module"))
```

您可能不需要担心所有这些，除非您注意到您的代码受制于这种行为。

3.15 与tqdm迭代的互操作性

在由 `tqdm` 库包装的迭代过程中尝试使用Loguru的记录器可能会干扰显示的进度条。 作为一种解决方法，可以使用 `tqdm.write()` 函数，而不是将日志直接写入 `sys.stderr`：

```
import time

from loguru import logger
from tqdm import tqdm

logger.remove()
logger.add(lambda msg: tqdm.write(msg, end=""))

logger.info("Initializing")

for x in tqdm(range(100)):
    logger.info("Iterating #{}", x)
    time.sleep(0.1)
```

在Windows上使用Spyder导入 `tqdm` 后，您可能会遇到日志着色问题。 GH#132 中讨论了此问题。 您可以在导入后立即调用 `colorama.deinit()` 轻松解决该问题。

3.16 在 Cython 模块中使用 Loguru 的 `logger`

Loguru和Cython无法很好地互操作。 这是因为Loguru（和 `logging`）在很大程度上依赖于Python堆栈帧，而Cython（作为另一种Python实现）则出于优化原因而尝试摆脱这些帧。

从用Cython编译的代码中调用 `logger` 可能会引发这种异常：

```
ValueError: call stack is not deep enough
```

当Loguru尝试访问被Cython禁止的堆栈帧时，会发生此错误。 Loguru无法检索已记录消息的上下文信息，但是存在一种解决方法，至少可以防止您的应用程序崩溃：

```
# Add this at the start of your file
logger = logger.opt(depth=-1)
```

请注意，记录的消息应正确显示，但是函数名称和其他信息将不正确。 GH#88 中讨论了此问题。

3.17 使用单独的处理程序集创建独立的记录器

Loguru从根本上设计为可与仅一个全局 `logger` 对象一起使用，该对象将记录消息分发到已配置的处理程序。 在某些情况下，将特定消息记录到特定处理程序可能会很有用。 例如，假设您想根据任意标识符将日志分为两个文件，则可以通过结合 `bind()` 和 `filter` 来实现：

```
from loguru import logger

def task_A():
    logger_a = logger.bind(task="A")
    logger_a.info("Starting task A")
    do_something()
    logger_a.success("End of task A")

def task_B():
    logger_b = logger.bind(task="B")
    logger_b.info("Starting task B")
    do_something_else()
    logger_b.success("End of task B")

logger.add("file_A.log", filter=lambda record: record["extra"]["task"] == "A")
logger.add("file_B.log", filter=lambda record: record["extra"]["task"] == "B")

task_A()
task_B()
```

通过这种方式， `"file_A.log"` 和 `file_B.log` 将仅包含分别来自 `task_A()` 和 `task_B()` 函数的日志。

现在，假设您有很多这些任务。 像这样配置每个处理程序可能有点麻烦。 最重要的是，它可能会不必要地减慢您的应用程序的速度，因为每个处理程序的过滤器函数都需要检查每个日志。 在这种情况下，建议依赖于 `copy.deepcopy()` 内置方法，该方法将创建一个独立的记录器对象。 如果将一个处理程序添加到深复制的记录器，该处理器将不会与使用原始日志记录器的其他函数共享:

```
import copy
from loguru import logger
```

```
3
4   def task(task_id, logger):
5       logger.info("Starting task {}", task_id)
6       do_something(task_id)
7       logger.success("End of task {}", task_id)
8
9   logger.remove()
10
11  for task_id in ["A", "B", "C", "D", "E"]:
12      logger_ = copy.deepcopy(logger)
13      logger_.add("file_%s.log" % task_id)
14      task(task_id, logger_)
```

请注意，如果您尝试复制添加了non-picklable 处理程序的记录器，则可能会遇到错误。 因此，通常建议在调用 `copy.deepcopy(logger)` 之前删除所有处理程序。

3.18 使用 `enqueue` 参数以与 `multiprocessing` 兼容

在Linux上，由于使用 `os.fork()` ，在由 `multiprocessing` 模块启动的另一个进程内使用记录器时，没有陷阱。 子进程将自动继承添加的处理程序， `enqueue=True` 参数是可选的，但建议使用，因为这样可以避免并发访问接收器：

```
1   # Linux implementation
2   import multiprocessing
3   from loguru import logger
4
5   def my_process():
6       logger.info("Executing function in child process")
7       logger.complete()
8
9   if __name__ == "__main__":
10      logger.add("file.log", enqueue=True)
11
12      process = multiprocessing.Process(target=my_process)
13      process.start()
14      process.join()
15
16      logger.info("Done")
```

在Windows上，事情变得更加复杂。 实际上，该操作系统不支持分叉，因此Python必须使用另一种方法来创建称为 "spawning" 的子进程。 此过程要求从头开始重新加载创建子进程的整个文件。 这与Loguru不能很好地互操作，导致在没有任何同步机制的情况下添加了两次处理程序，或者根本没有添加处理程序（取决于在 `__main__` 分支内部或外部调用的 `add()` 和 `remove()` ）。 因此，需要将 `logger` 对象作为子进程的初

始化参数显式传递：

```
# Windows implementation
import multiprocessing
from loguru import logger

def my_process(logger_):
    logger_.info("Executing function in child process")
    logger_.complete()

if __name__ == "__main__":
    logger.remove()  # Default "sys.stderr" sink is not picklable
    logger.add("file.log", enqueue=True)

    process = multiprocessing.Process(target=my_process, args=(logger, ))
    process.start()
    process.join()

    logger.info("Done")
```

Windows要求添加的接收器是 "picklable"，否则在创建子进程时会引发错误。许多流对象（例如，标准输出和文件描述符）不是"picklable"。在这种情况下，需要 `enqueue=True` 参数，因为它将允许子进程仅继承发送日志的队列对象。

`multiprocessing` 库通常还用于通过使用 `map()` 或 `apply()` 启动进程池。同样，它可以在Linux上完美运行，但需要在Windows上进行一些修改。您可能无法将 `logger` 作为worker函数的参数传递，因为worker 函数的参数需要是 "picklable"，但是使用 `enqueue=True` 添加的处理程序完全是 "可继承的"，而不是 "picklable"。相反，您在创建 `Pool` 对象时需要使用 `initializer` 和 `initargs` 参数，以允许worker访问共享 `logger` 。您可以将其分配给类属性，也可以覆盖子进程的全局记录器：

```
# workers_a.py
class Worker:

    _logger = None

    @staticmethod
    def set_logger(logger_):
        Worker._logger = logger_

    def work(self, x):
        self._logger.info("Square rooting {}", x)
        return x**0.5
```

```python
# main.py
from multiprocessing import Pool
from loguru import logger
import workers_a
import workers_b

if __name__ == "__main__":
    logger.remove()
    logger.add("file.log", enqueue=True)

    worker = workers_a.Worker()
    with Pool(4, initializer=worker.set_logger, initargs=(logger, )) as pool:
        resuts = pool.map(worker.work, [1, 10, 100])

    with Pool(4, initializer=workers_b.set_logger, initargs=(logger, )) as pool:
        results = pool.map(workers_b.work, [1, 10, 100])

    logger.info("Done")
```

不管在 Linux 下还是在 Windows 下，请注意，使用 `enqueue=True` 添加处理程序的进程负责内部使用的队列。 这意味着您应该避免从父进程中 `.move()` 所有子进程都可能继续使用的这些处理程序。 更重要的是，请注意，内部会启动一个线程来消耗队列。 因此，建议在离开 `Process` 之前调用 `complete()`，以确保队列保持稳定状态。

4. API 参考

Loguru 库提供了一个预先实例化的日志记录器，以方便在Python中处理日志。

只需要 `from loguru import logger` 即可。

4.1 Logger 类

`Logger` 类是 `loguru` 的核心对象，上面导入的 `logger` 即是该类的一个预先配置的实例，用于将日志消息分发到已配置的处理器。每个日志记录的配置和使用都通过对 `Logger` 类的某一个方法的调用来传递。只有一个记录器，因此无需在使用前重新创建。

`logger` 一旦导入，就可以用来编写有关代码中发生的事件的消息。通过阅读应用程序的输出日志，您可以更好地了解程序流程，并且可以更轻松地跟踪和调试意外行为。

使用 `add()` 方法向 `logger` 添加接收其所发送的日志消息的处理程序。请注意，您可以在导入后立即使用 `Logger`，因为它已预先配置（默认情况下，日志会发送到 `sys.stderr`）。可以使用不同的严重性级别和大括号属性（如 `str.format()` 方法）记录消息。

当消息被记录时，其将与 "记录" 相关联。记录是一个字典，其中包含有关日志记录上下文的信息：时间，函数，文件，行，线程，级别等等，它还包含模块的 `__name__`，这就是为什么不需要命名记录器的原因。

不应该自己实例化 `Logger` 类，使用 `from loguru import logger` 即可。

4.1.1 add()方法

```
def add(self,
        sink,
        *,
        level='DEBUG',
        format='<green>{time:YYYY-MM-DD HH:mm:ss.SSS}</green> | <level>{level: <8}
</level> | <cyan>{name}</cyan>:<cyan>{function}</cyan>:<cyan>{line}</cyan> - <level>
{message}</level>',
        filter=None,
        colorize=None,
        serialize=False, backtrace=True,
        diagnose=True,
        enqueue=False,
        catch=True,
        **kwargs
) -> int
```

该方法用于添加一个将日志消息发送到适当配置的接收器的处理程序。返回值为一个整数，表示与添加的接收器相关联的标识符，用于通过 `remove()` 方法来删除对应的处理程序。

该方法包含如下参数：

- `sink`

  负责接收格式化日志消息并将其传播到适当端点的对象。

  type：类文件对象、`str`、`pathlib.Path`、`callable`、协程函数或 `logging.Hander` 类型。
- `level='DEBUG'`

  将日志记录的消息发送到接收器的最低严重性级别。

  type：int 或 str` 类型。
- `format='<green>{time:YYYY-MM-DD HH:mm:ss.SSS}</green> | <level>{level: <8}
</level> | <cyan>{name}</cyan>:<cyan>{function}</cyan>:<cyan>{line}</cyan> -
<level>{message}</level>'`

  在将日志消息发送到接收器之前用于格式化日志消息的模板。

  type：`str` 或 返回值值类型为 `str` 的可调用对象。
- `filter=None`

  可选的指令，用于确定每个记录的消息是否应被发送到接收器。

type: `callable` 、 `str` 或 `dict` 类型。

- `colorize=None`

  格式化消息中包含的颜色标记是否应转换为用于终端着色的ANSI代码，如果不转换，则剥离（stripped）。如果为 `None` ，则根据接收器是否为tty自动进行选择。

  type: `bool` 。

- `serialize=False`

  在将日志消息及其记录发送到接收器之前，是否应首先将其转换为JSON字符串。

  type: `bool` 。

- `backtrace=True`

  格式化的异常跟踪是否应该向上扩展，超出捕获点，以显示生成错误的完整堆栈跟踪信息。

  type: `bool` 。

- `diagnose=True`

  异常跟踪是否应该显示变量值以简化调试。在生产中应该将其设置为 `False` ，以避免泄漏敏感数据。

  type: `bool` 。

- `enqueue=False`

  要记录的消息在到达接收器之前是否应该首先通过一个多进程安全队列。这在使多个进程将日志记录到一个文件时非常有用。这还可以使日志调用是非阻塞的。

  type: `bool` 。

- `catch=True`

  应该自动捕获接收器处理日志消息时发生的错误。如果为 `True` ，异常消息将显示在 `sys.stderr` 上，但该异常不会传播到调用者，从而防止应用程序崩溃。

  type: `bool` 。

- `**kwargs`

  仅对配置协程函数或文件类型的接收器有效的其他参数（见下文）。

当且仅当 `sink` 参数值是一个协程函数时，可以提供如下关键字参数：

- `loop`

  调度和执行异步日志记录任务的事件循环。如果为 `None` ，则使用 `asyncio.get_event_loop()` 返回的事件循环。

  type: `AbstractEventLoop` 。

当且仅当 `sink` 参数值是一个文件路径时，可以提供如下关键字参数：

- `rotation`

  指示何时关闭当前日志文件并启动新日志文件的条件。

  type: `str` 、 `int` 、 `datetime.time` 、 `datetime.timedelta` 或 `callable` 。

- `retention`

用来过滤在轮换或程序结束时应该删除的旧文件的指令。

type: `str`、`int`、`datetime.timedelta` 或 `callable`。

- `compression`

  一种压缩或归档格式，在关闭时日志文件应转换为这种格式。

  type: `str` 或 `callable`。

- `delay`

  是在配置接收器后立即创建文件，还是延迟到第一个记录消息。默认为 `False`。

  type: `bool`。

- `mode`

  内置的 `open()` 函数的打开模式。默认为 `"a"`（以附加模式打开文件）。

  type: `str`。

- `buffering`

  内置的 `open()` 函数的缓冲策略。默认值为 `1`（行缓冲文件）。

  type: `int`。

- `encoding`

  内置的 `open()` 函数的文件编码。如果没有，则默认为 `locale.getpreferredencoding()`。

  type: `str`。

- `**kwargs`

  传递给内置的 `open()` 函数的其他参数。

4.1.1.1 sink参数详解

`add()` 方法的 `sink` 参数所指定的接收器用于处理传入的日志消息，并在某个地方以某种方式继续将其写入。接收器可以有多种形式：

- 像 `sys.stderr` 或 `open("somefile.log", 'w')` 这样的类文件对象。任何具有 `.write()` 方法的对象都被认为是类文件对象。自定义处理器也可以实现 `flush()`（在每次记录消息后调用）、`stop()`（在完成接收时调用）和 `complete()`（通过同名方法进行异步等待）。
- 表示文件路径的 `str` 或 `pathlib.Path`。可以使用一些其他参数对其进行参数化，请参见下文。
- 可调用对象（例如，像 `lambda msg: print(msg)` 这样的函数）。这允许完全由用户首选项和需求定义的日志记录过程。
- 使用 `async def` 语句定义的异步协程函数。协程函数返回的协程对象将会通过 `loop.create_task()` 方法添加到事件循环中。在通过使用 `complete()` 结束事件循环结束之前，任务应该被异步等待。
- 类似 `logging.StreamHandler` 这样的内置 `logging.Handler` 对象。在这种情况下，Loguru 记录将会被自动转换为 `logging` 模块所期待的结构。

注意，应该避免在任何接收器中使用 `logger`，因为如果没有显式禁用模块的接收器，这会导致无限递归或死锁。

4.1.1.2 记录的消息

传递到所有添加的接收器的记录的消息只不过是表示格式化的日志的字符串，有一个与之关联的特殊的属性：`.recode`，该属性是一个字典，包含所有可能需要的上下文信息（参见下面的说明）。

记录的消息根据添加的接收器的 `format` 进行格式化。这种格式通常是包含大括号字段的字符串，用于显示来自记录字典的属性。

如果需要细粒度的控制，`format` 也可以是一个函数，它将记录作为参数并返回格式模板字符串。但是，请注意，在这种情况下，您应该注意将换行符和 `exception` 字段附加到返回的格式中，而如果格式是字符串，则为了方便起见，将自动添加 `"\n{exception}"`。

`filter` 属性可用于控制将哪些消息有效地传递到接收器，以及忽略哪些消息。可以使用一个将记录作为参数的函数，如果应该记录消息，则返回 `True`，否则返回 `False`。如果使用字符串，则只允许具有相同 `name` 的模块及其子模块的记录。还可以传递一个将模块名称映射到最低要求级别的字典。在这种情况下，每个日志记录将在该字典中搜索其最接近的父模块，并使用关联的级别作为过滤器。字典中的键值可以是 `int` 类型的严重级别、`str` 类型的级别名称或用来无条件地批准（authorize）和丢弃（discard）所有模块的日志的布尔值 `True` 和 `False`。为了设置默认级别，应该使用 `""` 作为模块名，因为它是所有模块的父模块（但是它不会抑制全局 `level` 阈值）。

请注意，在调用日志记录方法时，关键字参数（如果有的话）会自动添加到 `extra` 字典中，以方便上下文化（除了用于格式化之外）。

4.1.1.3 严重级别

每个记录的消息都与严重性级别相关联。 这些级别可以对消息进行优先级排序，并根据用途选择日志的详细程度。 例如，它允许向开发人员显示一些调试信息，但是对运行该应用程序的最终用户隐藏这些调试信息。

每个添加的接收器的 `level` 属性控制允许从其发出日志消息的最小阈值。使用 `logger` 时，您负责配置适当的日志粒度。 通过使用 `level()` 方法，可以添加更多自定义级别。 以下是具有默认严重性值的标准级别，每个级别均与同名的日志记录方法相关联：

级别名称	严重性值	Logger 方法
`TRACE`	5	`logger.trace()`
`DEBUG`	10	`logger.debug()`
`INFO`	20	`logger.info()`
`SUCCESS`	25	`logger.success()`
`WARNING`	30	`logger.warning()`
`ERROR`	40	`logger.error()`
`CRITICAL`	50	`logger.critical()`

4.1.1.4 记录字典

记录只是一个Python字典，可以通过 `message.record` 从接收器访问。 它包含日志记录调用的所有上下文信息（时间，函数，文件，行，级别等）。

每个键都可以在处理程序的 `format` 中使用，因此相应的值会正确显示在记录的消息中（例如 `"{level}"` -> `"INFO"`）。 某些记录的值是具有两个或多个属性的对象，这些属性可以使用 `"{key.attr}"` 进行格式化（默认情况下， `"{key}"` 会显示一个属性）。 格式指令（例如 `"{key: > 3}"`）也可以使用，并且对于时间特别有用（请参见下文）。

键	描述	属性
elapsed	从程序开始运行到现在所经过的时间	参见 `datetime.timedelta`
exception	格式化的异常（如果有的话），否则为 `None`	`type`，`value`，`traceback`
extra	用户绑定的属性的字典（参见 `bind()`）	`None`
file	发起日志记录调用的文件	`name`（默认值），`path`
function	发起日志记录调用的函数	`None`
level	用于记录日志的严重级别	`name`（默认值），`no`，`icon`
line	源代码中的行号	`None`
message	记录的消息（未格式化）	`None`
module	发起日志记录调用的模块	`None`
name	发起日志记录调用的模块的 `__name__` 属性	`None`
process	发起日志记录调用的进程	`name`，`id`（默认值）
thread	发起日志记录调用的线程	`name`，`id`（默认值）
time	发出日志记录调用时的带时区信息的本地时间（aware local time）	参见 `datetime.datetime`

4.1.1.5 时间格式

要使用您喜欢的时间表示形式，可以直接在处理程序格式的时间格式化程序说明符中进行设置，例如 `format="{time:HH:mm:ss} {message}"`。请注意，此日期时间表示您的本地时间，并且还可以识别时区，因此您可以显示UTC偏移量以避免歧义。

可以使用更人性化的标记来格式化时间字段。这些标记构成 @sdispater 的 Pendulum 库使用的标记的子集。要转义标记，只需在其周围加上方括号即可，例如，`"[YY]"` 将显式字面量 `"YY"`。

如果您希望显示UTC时间而非本地时间，则可以在时间格式的末尾添加 `"!UTC"`，例如 `{time:HH:mm:ss!UTC}`。这样做会将日期时间转换为UTC格式。

如果未使用时间格式说明符，例如，如果 `format="{time} {message}"`，则默认格式将使用 ISO 8601。

Token	Output

Year	YYYY	2000, 2001, 2002 … 2012, 2013
	YY	00, 01, 02 … 12, 13
Quarter	Q	1 2 3 4
Month	MMMM	January, February, March …
	MMM	Jan, Feb, Mar …
	MM	01, 02, 03 … 11, 12
	M	1, 2, 3 … 11, 12
Day of Year	DDDD	001, 002, 003 … 364, 365
	DDD	1, 2, 3 … 364, 365
Day of Month	DD	01, 02, 03 … 30, 31
	D	1, 2, 3 … 30, 31
Day of Week	dddd	Monday, Tuesday, Wednesday …
	ddd	Mon, Tue, Wed …
	d	0, 1, 2 … 6
Days of ISO Week	E	1, 2, 3 … 7
Hour	HH	00, 01, 02 … 23, 24
	H	0, 1, 2 … 23, 24
	hh	01, 02, 03 … 11, 12
	h	1, 2, 3 … 11, 12
Minute	mm	00, 01, 02 … 58, 59
	m	0, 1, 2 … 58, 59
Second	ss	00, 01, 02 … 58, 59
	s	0, 1, 2 … 58, 59

Fractional Second	S	0 1 … 8 9
	SS	00, 01, 02 … 98, 99
	SSS	000 001 … 998 999
	SSSS…	000[0..] 001[0..] … 998[0..] 999[0..]
	SSSSSS	000000 000001 … 999998 999999
AM / PM	A	AM, PM
Timezone	Z	-07:00, -06:00 … +06:00, +07:00
	ZZ	-0700, -0600 … +0600, +0700
	zz	EST CST … MST PST
Seconds timestamp	X	1381685817, 1234567890.123
Microseconds timestamp	x	1234567890123

4.1.1.6 文件接收器

如果接收器是 `str` 或 `pathlib.Path` 类型，则将打开相应的文件以写入日志。 该路径还可以包含一个特殊的 `"{time}"` 字段，该字段将在文件创建时以当前日期进行格式化。

在记录每个消息之前进行 `rotate` 检查。 如果已经存在与要创建的文件同名的文件，则通过在文件名的基础名称后附加日期来重命名现有文件，以防止文件被覆盖。 此参数接受：

- `int` 类型的值，对应于最大文件大小（以字节为单位），然后关闭当前记录的文件并重新开始一个新文件。
- `datetime.timedelta` 类型的值，用于指示每次新轮换的频率。
- `datetime.time` 类型的值，指定每一天在何时发生轮换。
- `str` 类型的值，用于对前面枚举的类型之一进行人类友好的参数化。例如 `"100 MB"`， `"0.5 GB"`， `"1 month 2 weeks"`， `"4 days"`， `"10h"`， `"monthly"`， `"18:00"`， `"sunday"`， `"w0"`， `"monday at 12:00"` 等等。
- `callable` 类型，将在记录日志之前被调用。它应该接受两个参数：要记录的日志消息和文件对象，如果现在应该进行轮换，它应该返回 `True`，否则返回 `False`。

`retention` 发生在轮换时，或当 `rotate` 为 `None` 时，发生在接收器停止时。 如果文件基于接收器文件与模式 `"basename(.\*).ext(.\*)"` 相匹配（则可能的时间字段预先用 `.*` 替换），则选择该文件。 此参数接受：

- `int` 类型值， 指示要保留的日志文件数，而较旧的文件将被删除。

- `datetime.timedelta` 类型的值，指定要保留的文件的最长期限。
- `str` 类型的值，用于对保留文件的最大期限进行人类友好的参数化。例如 `"1 week"`，`"2 days"`，`"2 month2"` 等等。
- `callable` 类型，将在保留过程之前被调用。 它应该接受日志文件列表作为参数，并对其进行任何想要的处理（移动文件，删除文件等）。

`compression` 发生在轮换时，或当 `rotate` 为 `None` 时，发生在接收器停止时。 此参数接受：

- `str` 类型的值，与压缩或存档文件扩展名相对应。其值可以为是 `"gz"`、`"bz2"`、`"xz"`、`"lzma"`、`"tar"`、`"tar.gz"`、`"tar.bz2"`、`"tar.xz"`、`"zip"` 之一。
- `callable` 类型，将在文件终止之前调用。 它应该接受日志文件的路径作为参数，然后处理所需的内容（自定义压缩，网络发送，删除等）。

无论哪种方式，如果您使用根据自己的喜好设计的自定义函数，则必须非常小心，不要在函数内使用 `logger`。 否则，您的程序可能会由于死锁而挂起。

4.1.1.7 颜色标记

要为日志添加颜色，您只需要用适当的标签（例如 `<red> some message </red>`）将格式字符串括起来即可。如果接收器不支持ANSI代码，则会自动删除这些标签。为了方便起见，您可以使用 `</>` 关闭最后一个开始标签而不重复其名称（例如 `<red>another message</>`）。

特殊标记 `<level>`（缩写为 `<lvl>`）根据记录的消息级别的配置颜色进行转换。

无法识别的标签将在解析过程中引发异常，以告知可能的错误使用。如果您希望按字面意义显示标记标签，则可以通过在前面加上 `\` 来将其转义，例如 `\<blue>`。如果由于某种原因需要以编程方式转义字符串，请注意内部用于解析标记标签的正则表达式为 `r"\\?</?((?:[fb]g\s)?[^<>\s]\*)>"`。

请注意，使用 `opt(colors=True)` 记录消息时，格式化参数（`args` 和 `kwargs`）中存在的颜色标记将被完全忽略。如果您需要记录包含可能会干扰颜色标签的标记的字符串（在这种情况下，请不要使用 `f-string`），则这一点很重要。以下是可用的标签（请注意兼容性可能因终端而异）：

Color (abbr)	Styles (abbr)
Black (k)	Bold (b)
Blue (e)	Dim (d)
Cyan (c)	Normal (n)
Green (g)	Italic (i)
Magenta (m)	Underline (u)
Red (r)	Strike (s)
White (w)	Reverse (v)
Yellow (y)	Blink (l)
	Hide (h)

用法：

	示例	
描述	前景	背景
Basic colors	`<red>`, `<r>`	`<GREEN>`, `<G>`
Light colors	`<light-blue>`, `<le>`	`<LIGHT-CYAN>`, `<LC>`
8-bit colors	`<fg 86>`, `<fg 255>`	`<bg 42>`, `<bg 9>`
Hex colors	`<fg #00005f>`, `<fg #EE1>`	`<bg #AF5FD7>`, `<bg #fff>`
RGB colors	`<fg 0,95,0>`	`<bg 72,119,65>`
Stylizing	`<bold>`, `<b>`, `<underline>`, `<u>`	

4.1.1.8 环境变量

接收器参数的默认值可以完全自定义。 如果您不喜欢预先配置的接收器的日志格式，此功能特别有用。

每个 `add()` 方法的默认参数都可以通过设置 `LOGURU_[PARAM]` 环境变量来修改。 例如，在Linux上，可以使用 `export LOGURU_FORMAT="{time} - {message}"` 或 `export LOGURU_DIAGNOSE=NO` 。

默认级别的属性也可以通过设置 `LOGURU_[LEVEL]_[ATTR]` 环境变量来修改。例如，在 Windows 上，可以使用 `setx LOGURU_DEBUG_COLOR "<blue>"` 或 `setx LOGURU_TRACE_ICON "🚀"` 。如果您使用 `set` 命令，请不要包含引号，而是根据需要转义特殊符号，例如 `set LOGURU_DEBUG_COLOR=^<blue^>` 。

如果想要禁用预先配置的接收器，可以将 `LOGURU_AUTOINIT` 环境变量设置为 `False` 。

在 Linux 上，您可能需要编辑 `~/.profile` 文件以使其持久化。 在Windows上，不要忘记重启终端以使更改生效。

4.1.1.9 示例

```
>>> logger.add(sys.stdout, format="{time} - {level} - {message}",
filter="sub.module")
```

```
>>> logger.add("file_{time}.log", level="TRACE", rotation="100 MB")
```

```
>>> def debug_only(record):
...     return record["level"].name == "DEBUG"
...
>>> logger.add("debug.log", filter=debug_only)  # Other levels are filtered out
```

```
>>> def my_sink(message):
...     record = message.record
...     update_db(message, time=record["time"], level=record["level"])
...
>>> logger.add(my_sink)
```

```
>>> level_per_module = {
...     "": "DEBUG",
...     "third.lib": "WARNING",
...     "anotherlib": False
... }
>>> logger.add(lambda m: print(m, end=""), filter=level_per_module, level=0)
```

```
>>> async def publish(message):
...     await api.post(message)
...
>>> logger.add(publish, serialize=True)
```

```
>>> from logging import StreamHandler
>>> logger.add(StreamHandler(sys.stderr), format="{message}")
```

```
>>> class RandomStream:
...     def __init__(self, seed, threshold):
...         self.threshold = threshold
...         random.seed(seed)
...     def write(self, message):
...         if random.random() > self.threshold:
...             print(message)
...
>>> stream_object = RandomStream(seed=12345, threshold=0.25)
>>> logger.add(stream_object, level="INFO")
```

4.1.2 remove() 方法

```
def remove(self, handler_id: Optional[int] = None) -> None
```

删除先前添加的处理程序并停止向其接收器发送日志。

参数：

- **handler_id** （`int` 或 `None`）

要删除的接收器的ID，它是由 `add()` 方法返回的。 如果为 `None`，则将删除所有处理程序。 预先配置的处理程序保证索引为 `0`。

引发的异常：

- `ValueError`

  如果 `handler_id` 不是 `None`，但没有活动的处理程序具有该ID。

```
>>> i = logger.add(sys.stderr, format="{message}")
>>> logger.info("Logging")
Logging
>>> logger.remove(i)
>>> logger.info("No longer logging")
```

4.1.3 complete() 方法

```
async def complete(self) -> Awaitable
```

等待处理程序安排的入队消息和异步任务结束。

此方法分两个步骤进行：首先等待所有使用 `enqueue=True` 添加到处理程序中的日志消息被处理，然后该方法返回一个对象，该对象可以被异步等待，以完成由协程接收器添加到事件循环中的所有日志记录任务。

可以从非异步代码中调用该方法。 当 `logger` 与 `multiprocessing` 一起使用以确保放置在内部队列中的消息在离开子进程之前已正确传输时，特别推荐这样做。

应该在 `asyncio.run()` 或 `loop.run_until_complete()` 执行的协程结束之前异步等待返回的对象，以确保处理了所有异步日志记录消息。函数 `asyncio.get_event_loop()` 是预先调用的，该方法仅等待与当前循环相同的循环中调度的任务。

该方法返回一个 `awaitable` 对象，确保所有异步日志调用在异步等待时完成。

```
>>> async def sink(message):
...     await asyncio.sleep(0.1)  # IO processing...
...     print(message, end="")
...
>>> async def work():
...     logger.info("Start")
...     logger.info("End")
...     await logger.complete()
...
```

```
>>> logger.add(sink)
1
>>> asyncio.run(work())
Start
End
```

```
>>> def process():
...     logger.info("Message sent from the child")
...     logger.complete()
...
>>> logger.add(sys.stderr, enqueue=True)
1
>>> process = multiprocessing.Process(target=process)
>>> process.start()
>>> process.join()
Message sent from the child
```

4.1.4 catch() 方法

该方法签名为：

```
def catch(
        self,
        exception=Exception,
        *,
        level="ERROR",
        reraise=False,
        onerror=None,
        exclude=None,
        default=None,
        message="An error has been caught in function '{record[function]}', "
                "process '{record[process].name}' ({record[process].id}), "
                "thread '{record[thread].name}' ({record[thread].id}):"
)
```

该方法返回一个装饰器以自动记录包装函数中可能捕获的错误。 这对于确保记录意外的异常非常有用，此方法可以包装整个程序。 在使用线程将错误传播到主记录程序线程时，这对于装饰 `Thread.run()` 方法也非常有用。

注意，变量值（使用 @Qix- 的 better_exceptions库）的可见性取决于每个配置的接收器的 `diagnose` 选项。

返回的对象还可以用作上下文管理器。

参数：

- **exception** （`Exception`）

  要拦截的异常类型。如果需要捕获几种类型，也可以使用异常元组。

- **level** （`str` 或 `int`）

  应该记录的消息的级别名称或严重性。

- **raises** （`bool`）

  是否应该再次引发异常并将其传播给调用者。

- **onerror** （`callable`）

  在记录消息后，如果发生错误将调用该可调用对象。它应该接受异常实例作为唯一的参数。

- **exclude** （`Exception`）

  将被故意忽略的一种异常类型（或类型的元组），因此其将在不进行日志记录的情况下传播给调用者。

- **default**

  当发生错误而没有重新引发该异常时，被装饰的函数的返回值。

- **message** （`str`）

  如果发生异常，将自动记录的消息。注意，它将使用 `record` 属性进行格式化。

该方法返回一个装饰器或上下文管理器，可用于装饰一个函数或者作为一个用于记录可能捕获的异常的上下文管理器。

```
>>> @logger.catch
... def f(x):
...     100 / x
...
>>> def g():
...     f(10)
...     f(0)
...
>>> g()
ERROR - An error has been caught in function 'g', process 'Main' (367), thread 'ch1' (1398):
Traceback (most recent call last):
  File "program.py", line 12, in <module>
    g()
    └ <function g at 0x7f225fe2bc80>
> File "program.py", line 10, in g
    f(0)
    └ <function f at 0x7f225fe2b9d8>
  File "program.py", line 6, in f
    100 / x
          └ 0
```

```
ZeroDivisionError: division by zero
```

```
>>> with logger.catch(message="Because we never know..."):
...     main()  # No exception, no logs
```

```
>>> # Use 'onerror' to prevent the program exit code to be 0 (if 'reraise=False')
while
>>> # also avoiding the stacktrace to be duplicated on stderr (if 'reraise=True').
>>> @logger.catch(onerror=lambda _: sys.exit(1))
... def main():
...     1 / 0
```

4.1.5 opt() 方法

该方法签名为：

```
def opt(
        self,
        *,
        exception=None,
        record=False,
        lazy=False,
        colors=False,
        raw=False,
        capture=True,
        depth=0,
        ansi=False
) -> Logger
```

对日志记录调用进行参数设置以略微更改生成的日志消息。 请注意，无法链式调用 `opt()` ，最后一此调用优先于其他调用，因为它将"重置"选项为其默认值。

参数：

- **exception** （`bool`, `tuple` 或 `Exception`）

  如果该参数的计算结果不为 `False`，则对传递的异常进行格式化并将其添加到日志消息中。 也可以是 `Exception` 对象或 `(type, value, traceback)` 元组。否则，将从 `sys.exc_info()` 中检索异常信息。

- **record** （`bool`）

如果为 `True`，通过在日志消息中使用 `{record[key]}`，可以使用将记录调用上下文化的记录字典来格式化消息。

- **lazy** （`bool`）

  如果为 `True`，用于格式化消息的日志记录调用属性应该是仅在级别足够高时才被调用的函数。 如果不需要，可以使用它来避免昂贵的函数。

- **colors** （`bool`）

  如果为 `True`，记录的消息将根据其可能包含的标记进行着色。

- **raw** （`bool`）

  如果为 `True`，则将忽略每个接收器的格式，并且将按原样发送消息。

- **capture** （`bool`）

  如果为 `False`，则记录的消息中的 `**kwargs` 将不会自动填充 `extra` 字典（尽管它们仍用于格式化）。

- **depth** （`int`）

  指定应使用哪个堆栈跟踪将记录的消息上下文化。 当从包装函数内部使用记录器来检索有价值的信息时，这很有用。

- **ansi** （`bool`）

  从0.4.1版本开始不推荐使用： `ansi` 参数将在Loguru 1.0.0中删除，它由更合适的名称 `colors` 取代。

该方法返回一个包装了核心 `logger` 的 `Logger` 对象，但是在发送之前已充分转换了记录的消息。

```
>>> try:
...     1 / 0
... except ZeroDivisionError:
...     logger.opt(exception=True).debug("Exception logged with debug level:")
...
[18:10:02] DEBUG in '<module>' - Exception logged with debug level:
Traceback (most recent call last, catch point marked):
> File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero
```

```
>>> logger.opt(record=True).info("Current line is: {record[line]}")
[18:10:33] INFO in '<module>' - Current line is: 1
```

```
>>> logger.opt(lazy=True).debug("If sink <= DEBUG: {x}", x=lambda:
math.factorial(2**5))
[18:11:19] DEBUG in '<module>' - If sink <= DEBUG:
263130836933693530167218012160000000
```

```
>>> logger.opt(colors=True).warning("We got a <red>BIG</red> problem")
[18:11:30] WARNING in '<module>' - We got a BIG problem
```

```
>>> logger.opt(raw=True).debug("No formatting\n")
No formatting
```

```
>>> logger.opt(capture=False).info("Displayed but not captured: {value}", value=123)
[18:11:41] Displayed but not captured: 123
```

```
>>> def wrapped():
...     logger.opt(depth=1).info("Get parent context")
...
>>> def func():
...     wrapped()
...
>>> func()
[18:11:54] DEBUG in 'func' - Get parent context
```

4.1.6 bind() 方法

该方法签名为：

```
def bind(self, **kwargs) -> Logger
```

将属性绑定到每个记录的消息记录的 `extra` 字典。

该方法用于向每个日志调用添加自定义上下文。

参数：

- ****kwargs**

  将添加到 `extra` 字典的键和值之间的映射。

该方法返回一个包装了核心 `logger` 的 `Logger` 对象，但是它用定制的 `extra` 字典发送记录。

```
>>> logger.add(sys.stderr, format="{extra[ip]} - {message}")
```

```
>>> class Server:
...     def __init__(self, ip):
...         self.ip = ip
...         self.logger = logger.bind(ip=ip)
...     def call(self, message):
...         self.logger.info(message)
...
>>> instance_1 = Server("192.168.0.200")
>>> instance_2 = Server("127.0.0.1")
>>> instance_1.call("First instance")
192.168.0.200 - First instance
>>> instance_2.call("Second instance")
127.0.0.1 - Second instance
```

4.1.7 contextualize() 方法

该方法签名为：

```
def contextualize(self, **kwargs)
```

在 `with` 块中将属性绑定到上下文本地的（context-local） `extra` 字典。

与 `bind()` 方法不同，该方法不返回 `logger`， `extra` 字典就地修改并全局更新。 最重要的是，它使用 `contextvars`，这意味着上下文化的值对于每个线程和异步任务都是唯一的。 退出上下文管理器后， `extra` 字典将恢复其初始状态。

参数：

- ****kwargs**

  将添加到上下文本地的 `extra` 字典的键和值之间的映射。

该方法返回的上下文管理器对象（也可以作为装饰器使用）将在进入时绑定属性，然后在退出时恢复 `extra` z 字典的初始状态。

```
>>> logger.add(sys.stderr, format="{message} | {extra}")
1
>>> def task():
...     logger.info("Processing!")
...
>>> with logger.contextualize(task_id=123):
...     task()
...
Processing! | {'task_id': 123}
>>> logger.info("Done.")
Done. | {}
```

4.1.8 patch() 方法

该方法签名为：

```
def patch(self, patcher: Callable[[Dict[str, Any]], None]) -> Logger
```

附加一个函数，以修改每个日志记录调用创建的记录字典。

`patcher` 可用于在记录传播到处理程序之前实时更新记录。 这允许用动态值填充 `extra` 字典，还允许在记录消息时对发出的记录进行高级修改。 在将日志消息发送到不同的处理程序之前，该函数被调用一次。

建议对 `record["extra"]` 字典进行修改，而不是对记录字典本身进行修改，因为Loguru在内部使用某些值，对其进行修改可能会产生意外的结果。

参数：

- **patcher** （`callable`）

  将记录字典将作为唯一参数的函数。 该函数负责就地更新记录，该函数不需要返回任何值，修改后的记录对象将被重用。

该方法返回一个包装核心 `logger` 的 `Logger` 对象，但是在将记录发送到添加的处理程序之前，这些记录通过了 `patcher` 函数。

```
>>> logger.add(sys.stderr, format="{extra[utc]} {message}")
>>> logger = logger.patch(lambda record:
record["extra"].update(utc=datetime.utcnow())
>>> logger.info("That's way, you can log messages with time displayed in UTC")
```

```
>>> def wrapper(func):
...     @functools.wraps(func)
...     def wrapped(*args, **kwargs):
...         logger.patch(lambda r:
r.update(function=func.__name__)).info("Wrapped!")
...         return func(*args, **kwargs)
...     return wrapped
```

```
>>> def recv_record_from_network(pipe):
...     record = pickle.loads(pipe.read())
...     level, message = record["level"], record["message"]
...     logger.patch(lambda r: r.update(record)).log(level, message)
```

4.1.9 level() 方法

该方法签名为：

```
def level(name, no=None, color=None, icon=None) -> Level
```

该方法用于添加、更新或获取日志记录级别。

日志记录级别由其 `name` 定义，并具有与之关联的严重性编号 `no` 、ANSI颜色标签 `color` 和图标 `icon` ，并且可以在运行时进行修改。 要将 `log()` 转到自定义级别，您必须使用其名称，严重性编号未链接回级别名称（这意味着多个级别可以共享相同的严重性）。

要添加新级别，需要其 `name` 和 `no` 。 `color` 和 `icon` 也可以指定，默认情况下为空。

要更新现有级别，请将其 `name` 与要更改的参数一起传递。 一旦添加了级别，就无法修改它的 `no` 。 要检索级别信息，该名称就足够了。

参数：

- **name** （ `str` ）

  日志记录级别的名称。
- **no** （ `int` ）

  要添加或更新的级别的严重性。
- **color** （ `str` ）

  要添加或更新的级别的颜色标记。
- **icon** （ `str` ）

要添加或更新的级别的图标。

引发的异常：

- `ValueError` - 如果没有用这样的 `name` 注册的级别。

该方法返回一个 `Level` 类型的命名元组，其中包含级别相关的信息。

```
>>> level = logger.level("ERROR")
>>> print(level)
Level(name='ERROR', no=40, color='<red><bold>', icon='❌')
>>> logger.add(sys.stderr, format="{level.no} {level.icon} {message}")
1
>>> logger.level("CUSTOM", no=15, color="<blue>", icon="@")
Level(name='CUSTOM', no=15, color='<blue>', icon='@')
>>> logger.log("CUSTOM", "Logging...")
15 @ Logging...
>>> logger.level("WARNING", icon=r"/!\")
Level(name='WARNING', no=30, color='<yellow><bold>', icon='/!\\')
>>> logger.warning("Updated!")
30 /!\ Updated!
```

4.1.10 disable() 方法

该方法的签名为：

```
def disable(name: Optional[str]) -> None
```

禁用来自 `name` 模块及其及子模块的消息记录。

使用Loguru的库的开发人员应绝对禁用它，以避免无关的日志消息干扰用户。

请注意，在极少数情况下，Loguru无法确定模块的 `__name__` 值。 在这种情况下， `record["name"]` 将等于 `None` ，这就是为什么 `None` 也是有效的参数的原因。

参数：

- **name** （ `str` 或 `None` ）

  要禁用日志记录的父模块的名称。

```
>>> logger.info("Allowed message by default")
[22:21:55] Allowed message by default
>>> logger.disable("my_library")
>>> logger.info("While publishing a library, don't forget to disable logging")
```

4.1.11 enable() 方法

该方法的签名为：

```
def enable(name: Optional[str]) -> None
```

启用来自 `name` 模块及其子模块的消息记录。

通常，使用Loguru的导入库会禁用日志记录，因此该函数允许用户始终接收这些消息。

要启用所有日志而不管它们来自哪个模块，可以传递一个空字符串 `""`。

参数：

- **name**（`str` 或 `None`）

  要重新启用日志记录的父模块的名称。

```
>>> logger.disable("__main__")
>>> logger.info("Disabled, so nothing is logged.")
>>> logger.enable("__main__")
>>> logger.info("Re-enabled, messages are logged.")
[22:46:12] Re-enabled, messages are logged.
```

4.1.12 configure() 方法

该方法签名为：

```python
def configure(self,
              *,
              handlers: Optional[List[Dict[str, Any]]] = None,
              levels: Optional[List[Dict[str, Union[int, str]]]] = None,
              extra: Optional[Dict[str, Any]] = None,
              patcher: Optional[Callable[[Dict[str, Any]], None]] = None,
              activation: Optional[List[Tuple[str, bool]]] = None
) -> List[int]
```

该方法用于配置核心 `logger`。

应该注意的是，使用此方法设置的 `extra` 值在所有模块中都可用，因此这是设置整体默认值的最佳方法。

参数：

- **handlers**

  要添加的每个处理程序的列表。 该列表应包含作为关键字参数传递给 `add()` 方法的参数字典。 如果不是 `None`，则首先删除所有以前添加的处理程序。

- **levels**

  要添加或更新的每个级别的列表。该列表应该包含作为关键字参数传递给 `level()` 方法的参数字典。永远不会删除之前创建的级别。

- **extra**

  包含绑定到核心 `logger` 的附加参数的字典，如果您在几个文件模块中调用 `bind()`，它对于共享公共属性非常有用。如果该参数值不为 `None`，这将删除以前配置的 `extra` 字典。

- **patcher**

  使用 `logger` 应用于所有模块中每个已记录消息的记录字典的函数。它应该在不返回任何内容的情况下就地修改字典。该函数在可能由 `patch()` 方法添加的函数之前执行。如果该参数值为 `None`，这将取代以前配置的 `patcher` 函数。

- **activation**

  `(name, state)` 元组的列表，指示应启用（如果 `state` 为 `True`）或禁用（如果 `state` 为 `False`）的记录器。 对 `enable()` 和 `disable()` 的调用是根据列表顺序进行的。 这不会修改以前激活的记录器，因此，如果需要全新的开始，请在列表前添加 `("", False)` 或 `("", True)`。

该方法返回一个整数列表，其中包含已添加接收器的标识符的列表（如果有的话）。

```
>>> logger.configure(
...     handlers=[
...         dict(sink=sys.stderr, format="[{time}] {message}"),
...         dict(sink="file.log", enqueue=True, serialize=True),
...     ],
...     levels=[dict(name="NEW", no=13, icon="¤", color="")],
...     extra={"common_to_all": "default"},
...     patcher=lambda record: record["extra"].update(some_value=42),
...     activation=[("my_module.secret", False), ("another_library.module", True)],
... )
[1, 2]
```

```
>>> # Set a default "extra" dict to logger across all modules, without "bind()"
>>> extra = {"context": "foo"}
>>> logger.configure(extra=extra)
>>> logger.add(sys.stderr, format="{extra[context]} - {message}")
>>> logger.info("Context without bind")
>>> # => "foo - Context without bind"
>>> logger.bind(context="bar").info("Suppress global context")
>>> # => "bar - Suppress global context"
```

4.1.13 parse() 方法

该方法的签名为：

```
@staticmethod
def parse(file, pattern, *, cast={}, chunk=2 ** 16) -> Generator[Dict, None, None]
```

解析原始日志并将每个条目提取为字典。

必须将日志记录格式指定为正则表达式 `pattern`，然后将其用于解析 `file` 并根据正则表达式中存在的命名组检索每个条目。

参数：

- **file**（`str`、`pathlib.Path` 或 类文件对象）

  要解析的日志文件的路径，或者一个一打开的文件对象。

- **pattern**（`str` 或 `re.Pattern`）

  用于日志解析的正则表达式，应包含命名组，这些命名组将包含在返回的字典中。

- **cast**（`callable` 或 `dict`）

  该函数应将已解析的正则表达式分组（字符串值的字典）就地转换为更合适的类型。 如果传递了一个字典，则该字典应该是已解析的日志字典的键到用于转换该键的关联值的函数的映射。

- **chunk**（`int`）

  遍历日志时读取的字节数，这避免了将整个文件加载到内存中。

该方法是一个生成器函数，产出值为字典，该字典是正则表达式的命名组到匹配值的映射，就如 `re.Match.groupdict()` 的返回值，但匹配值可以根据 `cast` 参数进行转换。

```
>>> reg = r"(?P<lvl>[0-9]+): (?P<msg>.*)"    # If log format is "{level.no} - {message}"
>>> for e in logger.parse("file.log", reg):  # A file line could be "10 - A debug message"
...     print(e)                             # => {'lvl': '10', 'msg': 'A debug message'}
```

```
>>> caster = dict(lvl=int)                  # Parse 'lvl' key as an integer
>>> for e in logger.parse("file.log", reg, cast=caster):
...     print(e)
```

```
>>> def cast(groups):
...     if "date" in groups:
...         groups["date"] = datetime.strptime(groups["date"], "%Y-%m-%d %H:%M:%S")
...
>>> with open("file.log") as file:
...     for log in logger.parse(file, reg, cast=cast):
...         print(log["date"], log["something_else"])
```

4.1.14 trace() 方法

该方法签名为：

```
def trace(self, message, *args, **kwargs) -> None
```

记录严重性为 "TRACE" 的 `message.format(*args, **kwargs)`。

4.1.15 debug() 方法

该方法签名为：

```
def debug(self, message, *args, **kwargs) -> None
```

记录严重性为 "DEBUG" 的 `message.format(*args, **kwargs)` 。

4.1.16 info() 方法

该方法签名为：

```
def info(self, message, *args, **kwargs) -> None
```

记录严重性为 "INFO" 的 `message.format(*args, **kwargs)` 。

4.1.17 success() 方法

该方法签名为：

```
def success(self, message, *args, **kwargs) -> None
```

记录严重性为 "SUCCESS" 的 `message.format(*args, **kwargs)` 。

4.1.18 warning() 方法

该方法签名为：

```
def warning(self, message, *args, **kwargs) -> None
```

记录严重性为 "WARNING" 的 `message.format(*args, **kwargs)` 。

4.1.19 error() 方法

该方法签名为：

```
def error(self, message, *args, **kwargs) -> None
```

记录严重性为 "ERROR" 的 `message.format(*args, **kwargs)`。

4.1.20 critical() 方法

该方法签名为：

```
1   def critical(self, message, *args, **kwargs) -> None
```

记录严重性为 "CRITICAL" 的 `message.format(*args, **kwargs)`。

4.1.21 log() 方法

该方法签名为：

```
1   def log(self, level, message, *args, **kwargs) -> None
```

记录严重性为 `level` 的 `message.format(*args, **kwargs)`。

4.1.22 exception() 方法

该方法签名为：

```
1   def exception(self, message, *args, **kwargs) -> None
```

记录带有异常信息的 "ERROR" 的便捷方法。