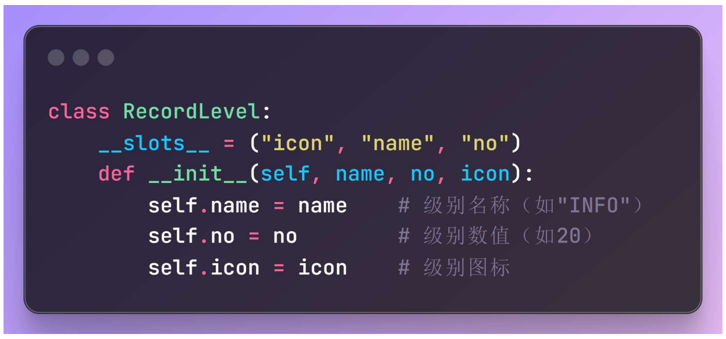
项目代码介绍：

- 项目地址：Delgan/loguru: Python logging made (stupidly) simple
- 项目 star 数：23k
- 最近更新时间：13hours 之前

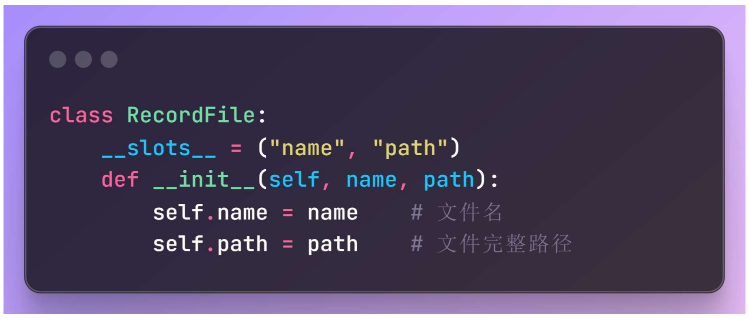
--------------------------以上数据记录于 2025 年 11 月 9 日 17 点 02 分-丁涛

核心数据结构：

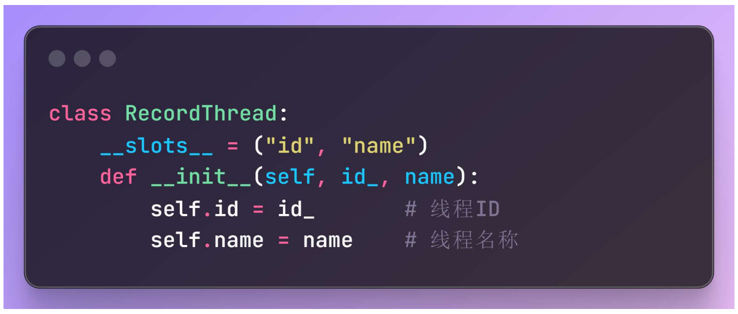
1. 属性记录



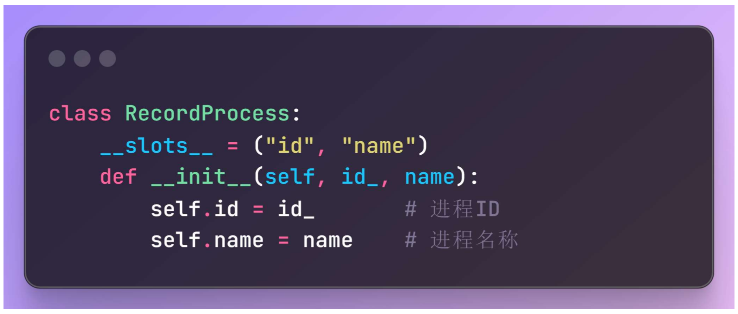
```python
class RecordLevel:
    __slots__ = ("icon", "name", "no")
    def __init__(self, name, no, icon):
        self.name = name     # 级别名称（如"INFO"）
        self.no = no         # 级别数值（如20）
        self.icon = icon     # 级别图标
```



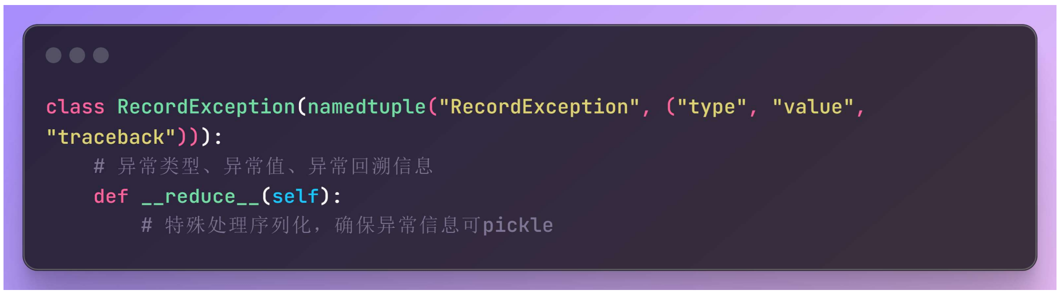
```python
class RecordFile:
    __slots__ = ("name", "path")
    def __init__(self, name, path):
        self.name = name     # 文件名
        self.path = path     # 文件完整路径
```



```python
class RecordThread:
    __slots__ = ("id", "name")
    def __init__(self, id_, name):
        self.id = id_        # 线程ID
        self.name = name     # 线程名称
```

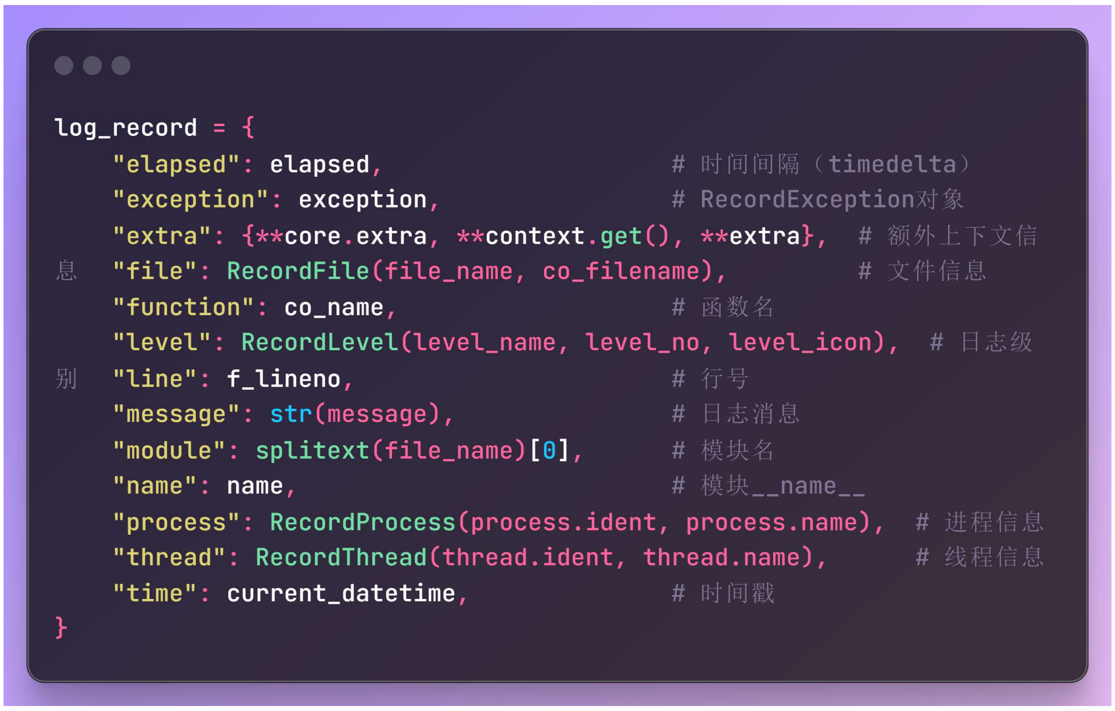


```python
class RecordProcess:
    __slots__ = ("id", "name")
    def __init__(self, id_, name):
        self.id = id_        # 进程ID
        self.name = name     # 进程名称
```



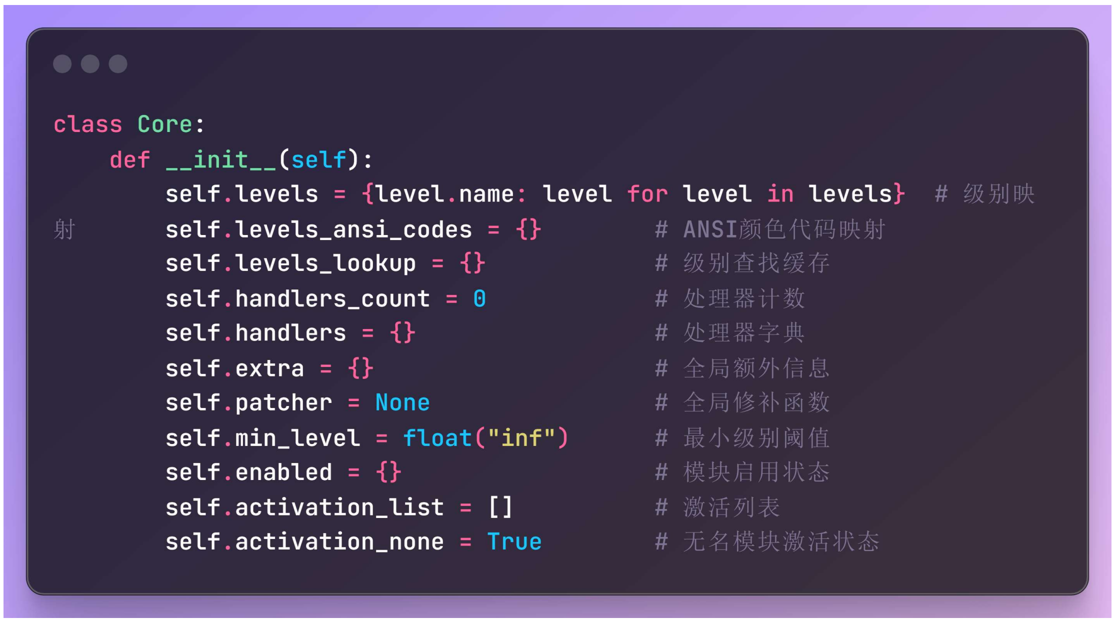
```python
class RecordException(namedtuple("RecordException", ("type", "value",
"traceback"))):
    # 异常类型、异常值、异常回溯信息
    def __reduce__(self):
        # 特殊处理序列化，确保异常信息可pickle
```

2. 日志记录



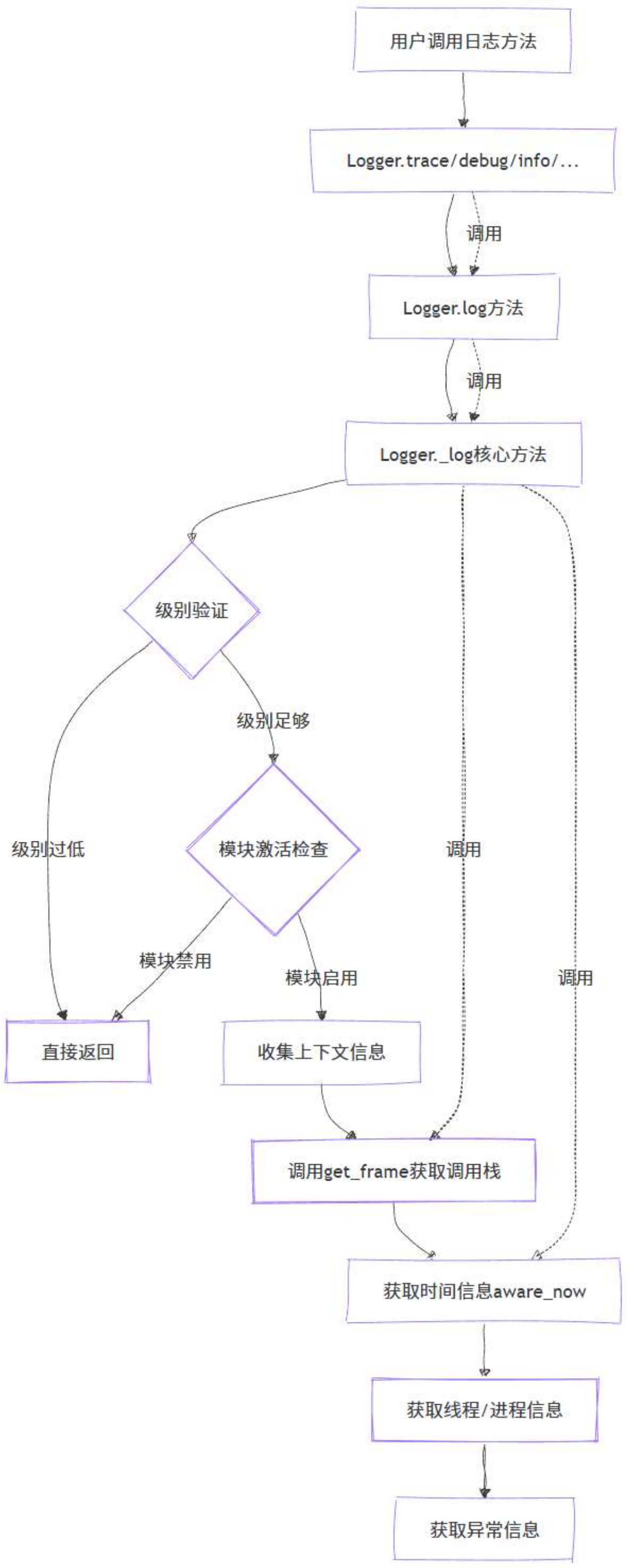
```python
log_record = {
    "elapsed": elapsed,                      # 时间间隔（timedelta）
    "exception": exception,                  # RecordException对象
    "extra": {**core.extra, **context.get(), **extra},  # 额外上下文信
息   "file": RecordFile(file_name, co_filename),         # 文件信息
    "function": co_name,                     # 函数名
    "level": RecordLevel(level_name, level_no, level_icon),  # 日志级
别   "line": f_lineno,                        # 行号
    "message": str(message),                 # 日志消息
    "module": splitext(file_name)[0],        # 模块名
    "name": name,                            # 模块__name__
    "process": RecordProcess(process.ident, process.name),  # 进程信息
    "thread": RecordThread(thread.ident, thread.name),      # 线程信息
    "time": current_datetime,                # 时间戳
}
```

3. 核心配置

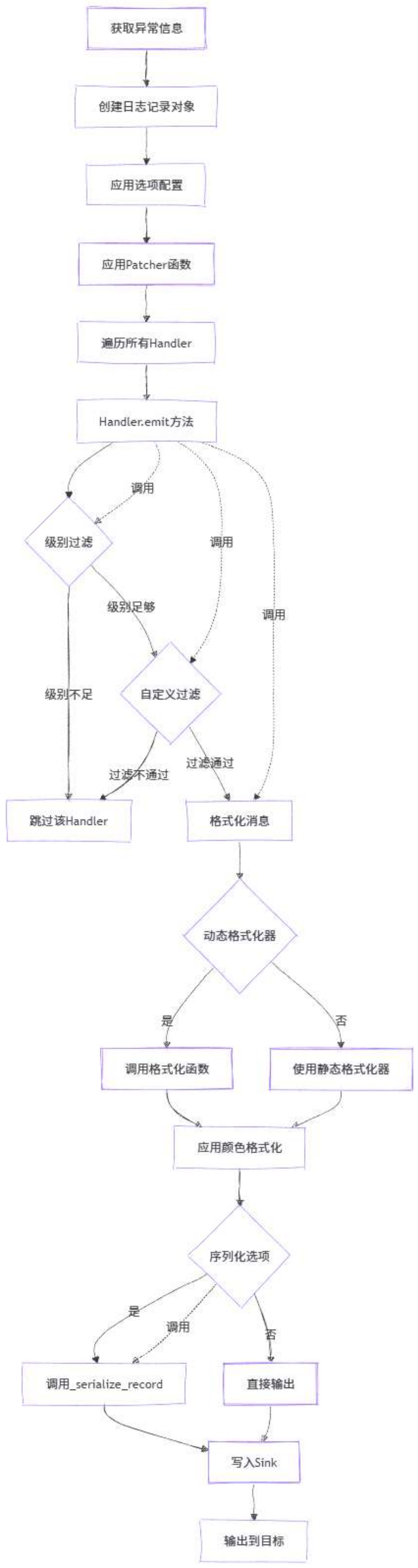


```python
class Core:
    def __init__(self):
        self.levels = {level.name: level for level in levels}  # 级别映
射       self.levels_ansi_codes = {}          # ANSI颜色代码映射
        self.levels_lookup = {}              # 级别查找缓存
        self.handlers_count = 0              # 处理器计数
        self.handlers = {}                   # 处理器字典
        self.extra = {}                      # 全局额外信息
        self.patcher = None                  # 全局修补函数
        self.min_level = float("inf")        # 最小级别阈值
        self.enabled = {}                    # 模块启用状态
        self.activation_list = []            # 激活列表
        self.activation_none = True          # 无名模块激活状态
```

核心功能流程图：



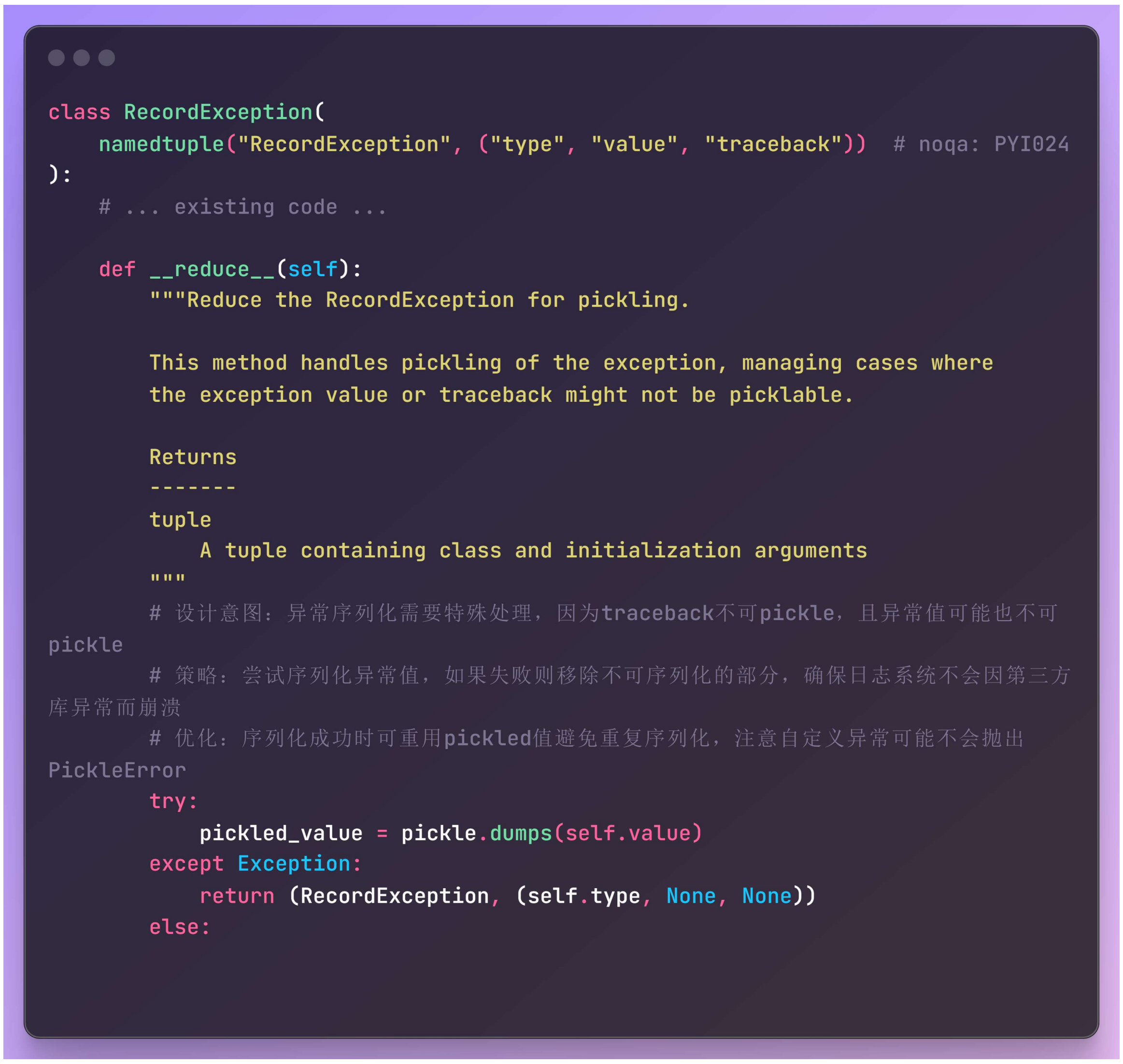
流程图 1


获取异常信息
创建日志记录对象
应用选项配置
应用Patcher函数
遍历所有Handler
Handler.emit方法
调用
调用
调用
级别过滤
级别足够
级别不足
自定义过滤
过滤不通过
过滤通过
跳过该Handler
格式化消息
动态格式化器
是
否
调用格式化函数
使用静态格式化器
应用颜色格式化
序列化选项
是
调用
否
调用_serialize_record
直接输出
写入Sink
输出到目标

流程图 2

设计意图型注释:

1. RecordException 类的序列化处理逻辑



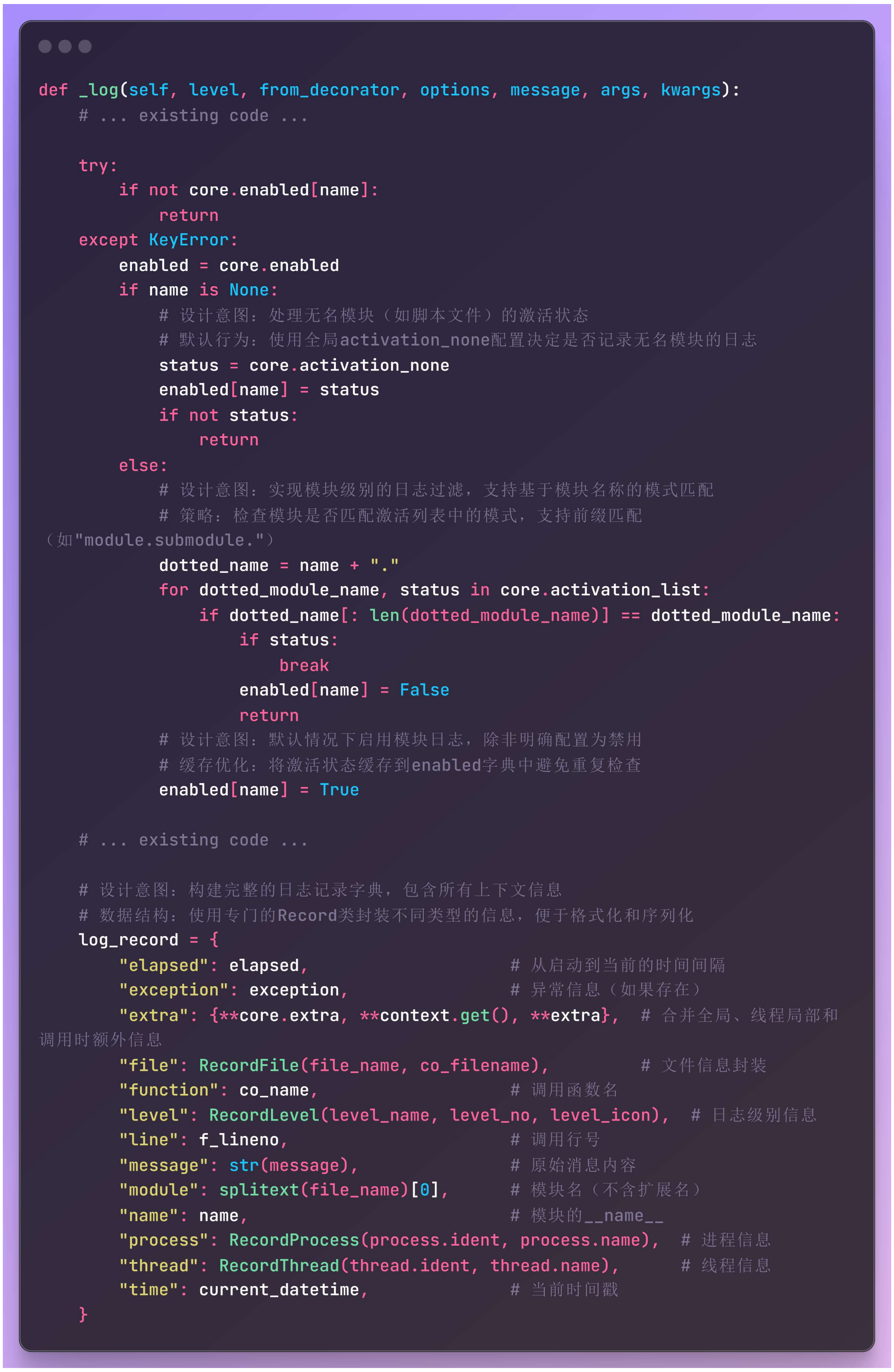
```python
class RecordException(
    namedtuple("RecordException", ("type", "value", "traceback"))  # noqa: PYI024
):
    # ... existing code ...

    def __reduce__(self):
        """Reduce the RecordException for pickling.

        This method handles pickling of the exception, managing cases where
        the exception value or traceback might not be picklable.

        Returns
        -------
        tuple
            A tuple containing class and initialization arguments
        """
        # 设计意图：异常序列化需要特殊处理，因为traceback不可pickle，且异常值可能也不可pickle
        # 策略：尝试序列化异常值，如果失败则移除不可序列化的部分，确保日志系统不会因第三方库异常而崩溃
        # 优化：序列化成功时可重用pickled值避免重复序列化，注意自定义异常可能不会抛出PickleError
        try:
            pickled_value = pickle.dumps(self.value)
        except Exception:
            return (RecordException, (self.type, None, None))
        else:
```

2. Logger 类的_log 方法中的模块激活检查逻辑



```python
def _log(self, level, from_decorator, options, message, args, kwargs):
    # ... existing code ...

    try:
        if not core.enabled[name]:
            return
    except KeyError:
        enabled = core.enabled
        if name is None:
            # 设计意图：处理无名模块（如脚本文件）的激活状态
            # 默认行为：使用全局activation_none配置决定是否记录无名模块的日志
            status = core.activation_none
            enabled[name] = status
            if not status:
                return
        else:
            # 设计意图：实现模块级别的日志过滤，支持基于模块名称的模式匹配
            # 策略：检查模块是否匹配激活列表中的模式，支持前缀匹配（如"module.submodule."）
            dotted_name = name + "."
            for dotted_module_name, status in core.activation_list:
                if dotted_name[: len(dotted_module_name)] == dotted_module_name:
                    if status:
                        break
                    enabled[name] = False
                    return
            # 设计意图：默认情况下启用模块日志，除非明确配置为禁用
            # 缓存优化：将激活状态缓存到enabled字典中避免重复检查
            enabled[name] = True

    # ... existing code ...

    # 设计意图：构建完整的日志记录字典，包含所有上下文信息
    # 数据结构：使用专门的Record类封装不同类型的信息，便于格式化和序列化
    log_record = {
        "elapsed": elapsed,                        # 从启动到当前的时间间隔
        "exception": exception,                    # 异常信息（如果存在）
        "extra": {**core.extra, **context.get(), **extra},  # 合并全局、线程局部和调用时额外信息
        "file": RecordFile(file_name, co_filename),         # 文件信息封装
        "function": co_name,                       # 调用函数名
        "level": RecordLevel(level_name, level_no, level_icon),  # 日志级别信息
        "line": f_lineno,                          # 调用行号
        "message": str(message),                   # 原始消息内容
        "module": splitext(file_name)[0],          # 模块名（不含扩展名）
        "name": name,                              # 模块的__name__
        "process": RecordProcess(process.ident, process.name),  # 进程信息
        "thread": RecordThread(thread.ident, thread.name),      # 线程信息
        "time": current_datetime,                  # 当前时间戳
    }
```

拓展功能(日志结构化输出增强)

1. 功能介绍
   基于Loguru现有的结构化日志记录能力，添加对JSON Lines格式的自动支持，同时保持与现有接口的完全兼容。命名为JSONLinesSink类，它可以:
   1.基于现有Sink接口：继承自Loguru的Handler机制
   2.保持架构兼容：不修改核心Logger/Core类
   3.利用现有配置：使用现有的format、filter等选项
   4.提供便捷接口：通过新的add_jsonl方法简化使用
2. 实现代码

```
"""
JSON Lines sink implementation for Loguru.

This module provides a JSON Lines format sink that can be used
with Loguru's
existing add() method or through the convenience method
add_jsonl().
"""


import json
from typing import Any, Dict, Optional, TextIO, Union

from ._handler import Handler
from ._logger import Logger



class JSONLinesSink:
    """
    A sink that writes log records in JSON Lines format.

    This sink automatically serializes log records to JSON format,
one record per line,
    making it ideal for log aggregation systems and data
processing pipelines.
    """


    def __init__(self, sink: Union[str, TextIO], **kwargs):
        """
        Initialize JSON Lines sink.

        Args:
            sink: File path or file-like object to write JSON
Lines to
            **kwargs: Additional arguments passed to the
```

```
underlying handler
        """
        self.sink = sink
        self.kwargs = kwargs

    def write(self, message: Dict[str, Any]) -> None:
        """
        Write a log record as JSON Line.

        Args:
            message: The log record dictionary to serialize
        """
        # Remove the 'message' key and use the actual message
content
        record = message.copy()
        formatted_message = record.pop('formatted_message',
record.get('message', ''))

        # Create JSON structure with message as the main content
        json_record = {
            'timestamp': record.get('time', {}).get('timestamp',
None),
            'level': record.get('level', {}).get('name', 'INFO'),
            'message': formatted_message,
            'module': record.get('file', {}).get('name', ''),
            'function': record.get('function', ''),
            'line': record.get('line', 0),
            'process': record.get('process', {}).get('name', ''),
            'thread': record.get('thread', {}).get('name', ''),
            'extra': record.get('extra', {})
        }

        # Add exception information if present
        if record.get('exception'):
            json_record['exception'] = {
                'type': record['exception'].get('type', ''),
                'message': record['exception'].get('message', ''),
                'traceback': record['exception'].get('traceback',
'')
            }

        # Write as JSON Line
        if hasattr(self.sink, 'write'):
            self.sink.write(json.dumps(json_record) + '\n')
            self.sink.flush()
        else:
            with open(self.sink, 'a', encoding='utf-8') as f:
                f.write(json.dumps(json_record) + '\n')


def add_jsonl(
    self: Logger,
    sink: Union[str, TextIO],
    *,
    level: Optional[Union[str, int]] = None,
    format: Optional[str] = None,
    filter: Optional[Union[str, dict]] = None,
    colorize: Optional[bool] = None,
    serialize: bool = False,
    backtrace: bool = True,
    diagnose: bool = True,
    enqueue: bool = False,
    catch: bool = True,
    **kwargs
) -> int:
    """
    Add a sink that outputs logs in JSON Lines format.

    This is a convenience method that creates a JSON Lines sink 
with appropriate
    formatting for JSON output.

    Args:
        sink: File path or file-like object for JSON Lines output
        level: Minimum logging level
        format: Log format string (optional, uses JSON-friendly 
format by default)
        filter: Log filter configuration
        colorize: Whether to colorize output (False for JSON 
Lines)
        serialize: Whether to serialize the message (True for JSON 
Lines)
        backtrace: Whether to format exception backtraces
        diagnose: Whether to display diagnostic information
        enqueue: Whether to enqueue messages for thread safety
        catch: Whether to catch errors in the sink
        **kwargs: Additional arguments for the sink
    Returns:
        The handler ID that can be used with remove()

    Example:
        >>> logger.add_jsonl("logs.jsonl")
        >>> logger.info("User login", user_id=123, action="login")
        # Output in logs.jsonl: {"timestamp": "...", "level": 
"INFO", "message": "User login", "user_id": 123, "action": 
"login"}
    """
    if format is None:
        format = "{time:YYYY-MM-DD HH:mm:ss.SSS} | {level: <8} | 
{name}:{function}:{line} - {message}"

    # Create JSON Lines sink handler
    jsonl_sink = JSONLinesSink(sink, **kwargs)

    # Add to logger with JSON-friendly configuration
    return self.add(
        jsonl_sink.write,
        level=level,
        format=format,
        filter=filter,
        colorize=False,  # JSON should not be colorized
        serialize=True,  # Ensure proper serialization for JSON
        backtrace=backtrace,
        diagnose=diagnose,
        enqueue=enqueue,
        catch=catch
    )


# Add the method to Logger class
Logger.add_jsonl = add_jsonl
```

在初始化函数中添加对 JSONLinssSink 类的引入（代码省略）

3. **使用实例**

```
from loguru import logger

# 基本用法 - 添加到 JSON Lines 文件
logger.add_jsonl("application.log.jsonl")

# 带级别过滤的用法
logger.add_jsonl("error.log.jsonl", level="ERROR")
```

```
# 使用示例
logger.info("User login successful", user_id=123,
ip="192.168.1.1")
logger.error("Database connection failed", db_host="localhost",
error="Connection timeout")

# 输出的JSON Lines格式:
# {"timestamp": "2024-01-15 10:30:00.123", "level": "INFO",
"message": "User login successful", "user_id": 123, "ip":
"192.168.1.1"}
# {"timestamp": "2024-01-15 10:31:00.456", "level": "ERROR",
"message": "Database connection failed", "db_host": "localhost",
"error": "Connection timeout"}
```