操作系统异常信息检测工具 -
项目设计文档

演示视频：

链接: https://pan.baidu.com/s/1QeAtJyTJ-2CZ2lF6zhaw9A?pwd=xcni

提取码: xcni

项目名称	**操作系统异常信息检测工具**
参赛队伍	自强不息独树一帜 (T202510730997619)
队伍成员	李尚泽、阮野、潘胜圆
指导老师	刘刚 (andyliu@lzu.edu.cn)
比赛	2025 年全国大学生计算机系统能力大赛 - 操作系统设计赛 - 西北区域赛
赛题导师	宋凯 (songkai01@ieisystem.com)
支持单位	浪潮电子信息产业股份有限公司、龙蜥社区

目录

1. 目标描述 2
2. 比赛题目分析和相关资料调研 3
3. 系统框架设计 7
4. 开发计划 23
5. 比赛过程中的重要进展 24
6. 系统测试情况 27
7. 遇到的主要问题和解决方法 40
8. 分工和协作 45
9. 提交仓库目录和文件描述 46
10. 比赛收获 51

1. 目标描述

1.1 项目背景

在现代生产环境中，Linux 操作系统作为服务器和云计算平台的核心组件，其稳定性和可靠性至关重要。系统运行过程中可能发生各种异常事件，包括内存不足导致的进程终止（OOM）、内核错误（Oops）、系统崩溃（Panic）、进程死锁、非正常重启以及文件系统异常等。这些异常事件往往会导致服务中断、数据丢失甚至系统瘫痪，给企业带来巨大的经济损失。

然而，当前 Linux 系统下缺乏一个统一、高效的工具来自动检测和监控这些异常事件。运维人员通常需要手动查看系统日志（如/var/log/kern.log、/var/log/syslog 等），通过关键字搜索来定位问题，这种方式不仅效率低下，而且容易遗漏关键信息。特别是在大规模集群环境中，日志量巨大，人工分析几乎不可能及时发现所有异常。

1.2 项目目标

本项目旨在开发一个功能完善的 Linux 操作系统异常信息检测工具（SuSG DetectTool），实现以下核心目标：

基础功能目标：

（1）支持检测六种主要的系统异常状态：OOM（内存不足）、Oops（内核错误）、Panic（系统崩溃）、Deadlock（进程死锁）、Reboot（非正常重启）、FS_Exception（文件系统异常）。

（2）提供灵活的规则配置机制，支持关键词匹配和正则表达式匹配两种检测方式。

（3）自动提取异常事件的关键信息，如进程 ID、进程名、阻塞时间等。

进阶功能目标：

（1）实现异常状态的实时监控功能，支持类似 tail -f 的日志跟踪模式。

（2）支持将检测工具部署为 systemd 守护进程，实现后台持续运行。

（3）提供异常事件的统计和分类功能，支持按类型、严重级别、频率等维度进行统计分析。

（4）实现多行日志聚合功能，能够完整捕获 Oops 和 Panic 的堆栈信息。

1.3 项目意义

本项目的开发具有重要的理论意义和实践价值：

理论意义：通过对 Linux 内核日志格式和异常事件特征的深入研究，加深了对操作系统内核机制的理解，特别是内存管理、进程调度、文件系统等核心子系统的工作原理。

实践价值：开发的检测工具可直接应用于生产环境，帮助运维人员及时发现系统异常，缩短故障定位时间，提高系统可用性。工具采用模块化设计，易于扩展和维护，可根据实际需求添加新的检测规则。

教育意义：通过完成本项目，团队成员系统学习了 Python 编程、软件工程实践、Linux 系统管理等知识，提升了分析问题和解决问题的能力。

2. 比赛题目分析和相关资料调研

2.1 赛题需求分析

根据比赛题目要求，本项目需要实现一个操作系统异常信息检测工具，具体需求如下：

基础功能需求：

异常类型	描述	检测要点
OOM	系统因内存不足导致进程被强制终止	检测"Out of memory"、"Killed process"等关键词
Oops	内核发生错误但系统仍可继续运行	检测"Oops:"、"BUG:"等关键词
Panic	内核发生无法恢复的致命错误	检测"Kernel panic"关键词
Deadlock	进程因资源竞争陷入死锁状态	检测 hung task 相关信息
Reboot	系统发生非正常重启	检测重启相关日志
FS_Exception	文件系统发生错误	检测 EXT4、XFS、BTRFS 等文件系统错误

进阶功能需求：

（1）实时监控：通过守护进程或定时任务实现对日志文件的持续监控。
（2）统计分类：提供按异常类型、频率等维度的统计功能。

技术特征要求：

（1）工具需支持 Linux 系统环境。
（2）提供命令行界面（CLI）和配置文件支持（YAML/JSON 格式）。
（3）异常检测需同时支持关键词匹配和正则表达式匹配。

2.2 Linux 系统日志机制调研

2.2.1 日志文件位置与格式

Linux 系统的内核日志主要存储在以下位置：

日志文件	内容描述

/var/log/kern.log	内核产生的日志消息
/var/log/syslog	系统综合日志
/var/log/messages	通用系统消息（部分发行版）
/var/log/dmesg	内核环缓冲区日志

Linux 内核日志通常采用 syslog 格式，典型格式如下：

```
Dec 24 17:40:10 hostname kernel: [12345.678901] Log message content
```

格式组成部分：

- 时间戳（月 日 时:分:秒）
- 主机名
- 日志来源（kernel 表示内核）
- 内核时间戳（可选，方括号内的秒数）
- 日志消息内容

2.2.2 OOM（Out of Memory）机制

当 Linux 系统内存不足时，内核的 OOM Killer 机制会选择并终止占用内存较多的进程以释放内存。OOM 事件的典型日志格式：

```
Dec 24 17:40:10 kernel: Out of memory: Killed process 1234 (python3) total-vm:123456kB, anon-rss:7890kB
```

OOM 日志包含的关键信息：

- 被终止的进程 ID（PID）
- 进程名称（comm）
- 虚拟内存大小（total-vm）
- 匿名内存 RSS（anon-rss）

2.2.3 Oops 内核错误

Oops 是 Linux 内核检测到错误时产生的错误报告，通常包含寄存器状态、调用堆栈等调试信息。Oops 不一定导致系统崩溃，但表明内核存在问题。典型 Oops 日志：

```
Dec 24 17:40:11 kernel: Oops: 0002 [#1] SMP PTI

Dec 24 17:40:11 kernel: CPU: 4 PID: 12890 Comm: mysqld Not tainted 4.18.0 #1
```

```
Dec 24 17:40:11 kernel: RIP: 0010:function_name+0x42/0x2e0

Dec 24 17:40:11 kernel: Call Trace:

Dec 24 17:40:11 kernel:    caller_function+0x2d/0xa0
```

Oops 日志的特征：

- 以"Oops:"或"BUG:"开头
- 包含 CPU、PID、进程名等信息
- 包含寄存器状态（RIP、RSP 等）
- 包含完整的调用堆栈（Call Trace）

2.2.4 Kernel Panic

Kernel Panic 是最严重的内核错误，表示内核无法继续安全运行，系统将停止工作。典型 Panic 日志：

```
Dec 24 17:40:13 kernel: Kernel panic - not syncing: Fatal exception

Dec 24 17:40:13 kernel: Kernel Offset: disabled

Dec 24 17:40:13 kernel: ---[ end Kernel panic - not syncing: Fatal exception ]---
```

Panic 日志特征：

- 包含"Kernel panic - not syncing"关键短语
- 通常伴随 Oops 信息
- 以结束标记"---[end"结尾

2.2.5 进程死锁（Hung Task）

当进程长时间处于不可中断睡眠状态（D 状态）时，内核会检测并报告 hung task。典型日志：

```
Dec 24 18:05:00 kernel: INFO: task kworker/0:1:4321 blocked for more than 120 seconds.

Dec 24 18:05:00 kernel:          Tainted: G          W          6.8.0 #1

Dec 24 18:05:00 kernel: task:kworker/0:1 state:D stack:0 pid:4321 ppid:2

Dec 24 18:05:00 kernel: Call Trace:

Dec 24 18:05:00 kernel:    schedule+0x2f/0x90
```

Hung Task 日志特征：

- 包含"blocked for more than X seconds"
- 显示进程名、PID、阻塞时间
- 包含进程状态和调用堆栈

2.2.6 文件系统异常

Linux 支持多种文件系统，各文件系统的错误日志格式不同：

EXT4 文件系统错误：

```
Dec 24 17:40:20 kernel: EXT4-fs error (device sda1): ext4_find_entry:1455: inode #2: reading directory lblock 0
```

XFS 文件系统错误：

```
Dec 24 12:05:30 kernel: XFS (sdb1): Metadata I/O Error: block 0x123456

Dec 24 12:05:30 kernel: XFS (sdb1): Log I/O Error Detected. Shutting down filesystem
```

BTRFS 文件系统错误：

```
Dec 24 12:10:45 kernel: BTRFS error (device sdc1): unable to read tree root
```

2.3 相关技术调研

2.3.1 Python CLI 框架

经过调研比较，我们选择了以下技术栈：

技术组件	选择方案	选择理由
CLI 框架	Typer	基于类型注解的现代 CLI 框架，语法简洁
输出美化	Rich	支持表格、颜色、进度条等丰富输出
配置解析	PyYAML	YAML 格式易读易写，适合规则配置

2.3.2 文件跟踪技术

实现类似 tail -f 的文件跟踪功能需要考虑：

- 轮询机制：定期检查文件是否有新内容

- 日志轮转处理：检测文件 inode 变化或文件大小减小
- 性能优化：避免频繁读取造成 CPU 占用过高

2.3.3 Systemd 服务管理

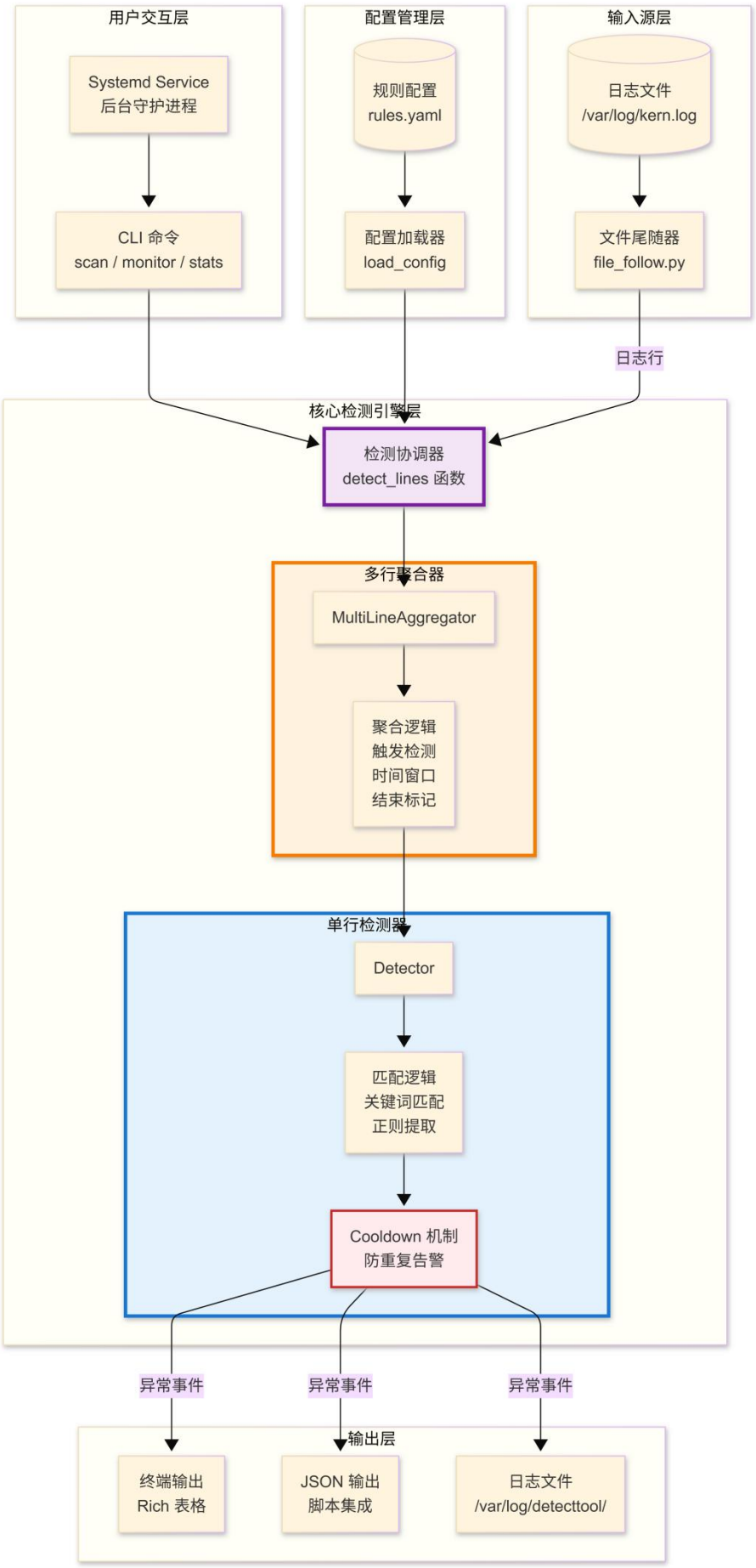
将工具部署为后台服务需要编写 systemd unit 文件，主要配置项：

- Type=simple：简单类型服务
- Restart=on-failure：失败时自动重启
- StandardOutput/StandardError：日志输出配置

3. 系统框架设计

3.1 整体架构设计

本项目采用经典的分层架构模式，将系统划分为四个逻辑层次：输入源层、核心引擎层、配置层和交互层。各层之间通过明确的接口进行通信，实现了高内聚、低耦合的设计目标。


用户交互层
Systemd Service
后台守护进程
CLI 命令
scan / monitor / stats
配置管理层
规则配置
rules.yaml
配置加载器
load_config
输入源层
日志文件
/var/log/kern.log
文件尾随器
file_follow.py
日志行
核心检测引擎层
检测协调器
detect_lines 函数
多行聚合器
MultiLineAggregator
聚合逻辑
触发检测
时间窗口
结束标记
单行检测器
Detector
匹配逻辑
关键词匹配
正则提取
Cooldown 机制
防重复告警
异常事件
异常事件
异常事件
输出层
终端输出
Rich 表格
JSON 输出
脚本集成
日志文件
/var/log/detecttool/

3.2 核心模块设计

3.2.1 配置模块（config.py）

配置模块负责加载和解析 YAML 格式的规则配置文件，将配置转换为程序内部使用的数据结构。

Rule 数据类设计：

```
@dataclass
class Rule:
    id: str
    type: str
    severity: str = "medium"
    keywords_any: List[str] = field(default_factory=list)
    keywords_all: List[str] = field(default_factory=list)

    regex_any: List[Pattern[str]] = field(default_factory=list)
    regex_all: List[Pattern[str]] = field(default_factory=list)

    cooldown_seconds: int = 0
```

Rule 类的字段说明：

- id：规则的唯一标识符，用于在检测结果中标识触发的规则
- type：异常类型，如 OOM、OOPS、PANIC 等
- severity：严重级别，支持 low、medium、high、critical 四个级别
- keywords_any：任意关键词列表，日志包含其中任一关键词即匹配
- keywords_all：全部关键词列表，日志需包含所有关键词才匹配
- regex_any：任意正则列表，编译后的正则表达式对象
- regex_all：全部正则列表，所有正则都需匹配
- cooldown_seconds：冷却时间，同一特征的事件在此时间内只报告一次

配置加载函数：

```
def load_config(path: str) -> Config:
    try:
        with open(path, "r", encoding="utf-8") as f:
            data = yaml.safe_load(f) or {}
    except FileNotFoundError:
        raise FileNotFoundError(
            f"Configuration file not found: {path}\n"
            f"Please ensure the file exists or specify a different config with --config"
```

```
            )
        except yaml.YAMLError as e:
            raise ValueError(f"Invalid YAML in configuration file {path}: {e}")

        rules: List[Rule] = []
        for idx, r in enumerate(data.get("rules", []) or [], start=1):
            # Validate required fields
            if "id" not in r:
                raise ValueError(f"Rule #{idx} is missing required field 'id'")
            if "type" not in r:
                raise ValueError(f"Rule '{r.get('id', idx)}' is missing required field 'type'")

            reg_any_str = r.get("regex_any", []) or []
            reg_all_str = r.get("regex_all", []) or []

            # Compile regexes with error handling
            try:
                regex_any_compiled = [re.compile(p) for p in reg_any_str]
                regex_all_compiled = [re.compile(p) for p in reg_all_str]
            except re.error as e:
                raise ValueError(
                    f"Invalid regex in rule '{r['id']}': {e}\n"
                    f"Please check the regex patterns in your configuration"
                )

            rules.append(
                Rule(
                    id=r["id"],
                    type=r["type"],
                    severity=r.get("severity", "medium"),
                    keywords_any=r.get("keywords_any", []) or [],
                    keywords_all=r.get("keywords_all", []) or [],
                    regex_any=regex_any_compiled,
                    regex_all=regex_all_compiled,
                    cooldown_seconds=int(r.get("cooldown_seconds", 0) or 0),
                )
            )

        return Config(version=int(data.get("version", 1)), rules=rules)
```

配置加载函数的设计要点：

- 使用 yaml.safe_load 安全加载 YAML，防止代码注入
- 对必需字段（id、type）进行验证
- 对正则表达式进行预编译，并捕获编译错误

- 提供清晰的错误提示信息

3.2.2 检测引擎模块（engine.py）

检测引擎是系统的核心模块，负责日志行的匹配检测和事件生成。

Incident 事件数据类：

```
@dataclass
class Incident:
    rule_id: str
    type: str
    severity: str
    message: str
    line_no: int
    extracted: Dict[str, str]
    context: List[str] = field(default_factory=list)

    def to_dict(self) -> Dict:
        return asdict(self)
```

Incident 类表示检测到的一个异常事件，包含：

- rule_id：触发的规则 ID
- type：异常类型
- severity：严重级别
- message：原始日志消息
- line_no：日志行号
- extracted：从日志中提取的字段（如 pid、comm 等）
- context：多行上下文（用于 Oops/Panic 的堆栈信息）

关键词匹配算法：

```
def _keywords_match(rule: Rule, text: str) -> bool:
    if rule.keywords_all and not all(k in text for k in rule.keywords_all):
        return False
    if rule.keywords_any and not any(k in text for k in rule.keywords_any):
        return False
    return True
```

关键词匹配采用短路求值：

- 对于 keywords_all，使用 all()函数检查所有关键词是否存在
- 对于 keywords_any，使用`any()`函数检查是否存在任一关键词
- 两个条件都满足时返回 True

正则表达式匹配与字段提取：

```
def _regex_match(rule: Rule, text: str) -> Tuple[bool, Dict[str, str]]:
    extracted: Dict[str, str] = {}

    def match_any(patterns) -> Optional[object]:
        for p in patterns:
            m = p.search(text)
            if m:
                return m
        return None

    if rule.regex_all:
        for p in rule.regex_all:
            if not p.search(text):
                return False, {}
        m0 = rule.regex_all[0].search(text)
        if m0:
            extracted.update({k: str(v) for k, v in m0.groupdict().items() if v is not None})

    if rule.regex_any:
        m = match_any(rule.regex_any)
        if not m:
            return False, {}
        extracted.update({k: str(v) for k, v in m.groupdict().items() if v is not None})

    return True, extracted
```

正则匹配的设计要点：

- 支持命名捕获组，自动提取匹配的字段到 extracted 字典
- regex_all 要求所有正则都匹配
- regex_any 只需要任一正则匹配
- 返回元组包含匹配结果和提取的字段

冷却机制：

```
class Cooldown:
    def __init__(self) -> None:
        self._last: Dict[str, float] = {}

    def allow(self, fingerprint: str, cooldown_seconds: int) -> bool:
        if cooldown_seconds <= 0:
            return True
```

```
        now = time.time()
        last = self._last.get(fingerprint)
        if last is not None and (now - last) < cooldown_seconds:
            return False
        self._last[fingerprint] = now
        return True
```

冷却机制用于防止短时间内重复报告相同事件：

- 使用事件指纹（fingerprint）作为去重依据
- 指纹由规则 ID、进程 ID、进程名和消息前 80 字符组成
- 在冷却时间内的重复事件会被过滤

Detector 检测器类：

```
class Detector:
    """
    Stateful detector for streaming logs.
    Keeps cooldown state across lines.
    """
    def __init__(self, rules: List[Rule]) -> None:
        self.rules = rules
        self.cooldown = Cooldown()

    def process_line(self, line_no: int, line: str) -> List[Incident]:
        text = line.rstrip("\n")
        hits: List[Incident] = []

        for rule in self.rules:
            if not _keywords_match(rule, text):
                continue
            ok, extracted = _regex_match(rule, text)
            if not ok:
                continue

            fp = f"{rule.id}|{extracted.get('pid','')}|{extracted.get('comm','')}|{text[:80]}"
            if not self.cooldown.allow(fp, rule.cooldown_seconds):
                continue

            hits.append(
                Incident(
                    rule_id=rule.id,
                    type=rule.type,
                    severity=rule.severity,
                    message=text,
```

```
                        line_no=line_no,
                        extracted=extracted,
                    )
                )
            return hits
```

Detector 类的处理流程：

1. 遍历所有规则
2. 先进行关键词匹配（快速过滤）
3. 再进行正则匹配（精确匹配并提取字段）
4. 计算事件指纹并检查冷却状态
5. 生成 Incident 事件对象

3.2.3 多行聚合器（MultiLineAggregator）

多行聚合器用于处理 Oops、Panic、Deadlock 等多行日志事件，将相关的后续行作为上下文聚合到一起。

触发类型识别：

```
def _trigger_type(text: str) -> Optional[str]:
    if "Kernel panic - not syncing" in text:
        return "PANIC"
    if ("Oops:" in text) or ("BUG:" in text) or ("Unable to handle kernel" in text):
        return "OOPS"
    if ("blocked for more than" in text) and ("task" in text):
        return "DEADLOCK"
    return None
```

结束标记定义：

```
_END_MARKERS = (
    "end trace",
    "End trace",
    "end Kernel panic",
    "End Kernel panic",
    "---[ end",
)
```

Syslog 时间戳解析：

```
_SYSLOG_TS =
re.compile(r"^(?P<mon>[A-Z][a-z]{2})\s+(?P<day>\d{1,2})\s+(?P<h>\d{2}):(?P<m>\d{2}):(?P<s>\d
```

```
{2})\b")
_MONTH = {"Jan": 1, "Feb": 2, "Mar": 3, "Apr": 4, "May": 5, "Jun": 6,
          "Jul": 7, "Aug": 8, "Sep": 9, "Oct": 10, "Nov": 11, "Dec": 12}

def _parse_syslog_ts(line: str) -> Optional[float]:
    m = _SYSLOG_TS.search(line)
    if not m:
        return None
    mon = _MONTH.get(m.group("mon"))
    if not mon:
        return None
    day = int(m.group("day"))
    h = int(m.group("h"))
    mi = int(m.group("m"))
    s = int(m.group("s"))
    year = datetime.now().year
    try:
        dt = datetime(year, mon, day, h, mi, s)
        return dt.timestamp()
    except Exception:
        return None
```

MultiLineAggregator 类核心逻辑：

```
class MultiLineAggregator:
    def __init__(
        self,
        detector: Detector,
        *,
        window_seconds: float = 5.0,
        max_lines: int = 200,
        idle_flush_seconds: float = 0.8,
    ) -> None:
        self.detector = detector
        self.window_seconds = window_seconds
        self.max_lines = max_lines
        self.idle_flush_seconds = idle_flush_seconds

        self.active_type: Optional[str] = None
        self.start_line_no: int = 0
        self.start_line: str = ""
        self.start_ts: Optional[float] = None
        self.context: List[str] = []
        self._last_activity_wall: float = 0.0
```

聚合器的关键参数：

- window_seconds：时间窗口，超过此时间间隔的日志不再聚合
- max_lines：最大上下文行数，防止无限聚合
- idle_flush_seconds：空闲超时，用于实时监控模式

聚合处理逻辑：

```
def process(self, line_no: int, line: str) -> List[Incident]:
    # heartbeat: (0, "") 用于 idle flush
    if line_no == 0 and line == "":
        if self.active_type and (time.time() - self._last_activity_wall) >= self.idle_flush_seconds:
            return self._emit()
        return []

    text = line.rstrip("\n")
    t = _trigger_type(text)

    # 若当前在聚合中，先处理"切断条件"
    if self.active_type:
        # 新触发：先 flush 老的，再 start 新的
        if t is not None:
            out = self._emit()
            self._start(t, line_no, text)
            return out

        # 时间窗口切断
        if self.start_ts is not None:
            cur_ts = _parse_syslog_ts(text)
            if cur_ts is not None and (cur_ts - self.start_ts) > self.window_seconds:
                out = self._emit()
                return out + self.process(line_no, line)

        # 追加到 context
        if len(self.context) < self.max_lines:
            self.context.append(text)
        self._last_activity_wall = time.time()

        # 结束标记：立刻 flush
        if any(m in text for m in _END_MARKERS) or len(self.context) >= self.max_lines:
            return self._emit()

        return []
```

```
        # 不在聚合中：如果触发多行类型 -> start block
        if t is not None:
            self._start(t, line_no, text)
            return []

        # 普通行：直接走规则检测
        return self.detector.process_line(line_no, line)
```

聚合器的状态机设计：

1.**空闲状态**：收到触发行时进入聚合状态

2.**聚合状态**：持续收集后续行作为上下文

3.**结束条件**：遇到新触发行、时间窗口超时、结束标记或达到最大行数时输出事件

3.2.4 文件跟踪模块（file_follow.py）

文件跟踪模块实现类似 tail -f 的功能，支持持续监控日志文件的新增内容。

```
def follow_file(
        path: str,
        *,
        start_at_end: bool = True,
        poll_interval: float = 0.2,
        yield_heartbeat: bool = False,
) -> Iterator[Tuple[int, str]]:
        line_no = 0

        def _open():
            return open(path, "r", encoding="utf-8", errors="replace")

        f = _open()
        try:
            st = os.stat(path)
            inode = st.st_ino

            if start_at_end:
                f.seek(0, os.SEEK_END)
                line_no = 0
            else:
                f.seek(0, os.SEEK_SET)
                line_no = 0

            while True:
                line = f.readline()
```

```
                if line:
                    line_no += 1
                    yield line_no, line
                    continue

                # 没有新行：检查是否被截断/轮转
                time.sleep(poll_interval)
                if yield_heartbeat:
                    yield 0, ""
                try:
                    st2 = os.stat(path)
                except FileNotFoundError:
                    continue

                if st2.st_ino != inode or st2.st_size < f.tell():
                    # 轮转/截断：重开文件
                    try:
                        f.close()
                    except Exception:
                        pass
                    f = _open()
                    inode = st2.st_ino
                    if start_at_end:
                        f.seek(0, os.SEEK_END)
                        line_no = 0
                    else:
                        f.seek(0, os.SEEK_SET)
                        line_no = 0

    finally:
        try:
            f.close()
        except Exception:
            pass
```

文件跟踪的关键特性：

- **轮询机制**：使用 time.sleep(poll_interval)避免 CPU 空转
- **日志轮转检测**：通过比较 inode 和文件大小检测日志轮转
- **心跳机制**：yield_heartbeat 选项用于触发多行聚合器的空闲刷新
- **位置选择**：start_at_end 选项控制从文件开头还是末尾开始

3.2.5 命令行接口模块（cli.py）

命令行接口模块使用 Typer 框架实现，提供四个主要命令。

scan 命令：

```
@app.command()
def scan(
    file: str = typer.Option(..., "--file", "-f", help="Path to a log file to scan"),
    config: str = typer.Option("configs/rules.yaml", "--config", "-c", help="Path to rules YAML"),
    json_out: bool = typer.Option(False, "--json", help="Output JSON instead of table"),
):
    try:
        cfg = load_config(config)
        incidents = detect_lines(_iter_file_lines(file), cfg.rules)
    except FileNotFoundError as e:
        console.print(f"[bold red]Error:[/bold red] {e}", style="red")
        raise typer.Exit(1)
```

monitor 命令：

```
@app.command()
def monitor(
    file: str = typer.Option(..., "--file", "-f", help="Path to a log file to follow"),
    config: str = typer.Option("configs/rules.yaml", "--config", "-c", help="Path to rules YAML"),
    json_out: bool = typer.Option(False, "--json", help="Output JSON lines"),
    from_start: bool = typer.Option(False, "--from-start", help="Read from beginning"),
    poll_interval: float = typer.Option(0.2, "--poll", help="Polling interval seconds"),
):
    detector = Detector(cfg.rules)
    agg = MultiLineAggregator(detector)

    for line_no, line in follow_file(
        file,
        start_at_end=(not from_start),
        poll_interval=poll_interval,
        yield_heartbeat=True,
    ):
        hits = agg.process(line_no, line)
        for inc in hits:
            # 输出处理...
```

stats 统计命令：

```
def _generate_statistics(incidents: List[Incident], total_lines: int, top_n: int = 10) -> Dict[str, Any]:
    total = len(incidents)
    type_counts = Counter(inc.type for inc in incidents)
    severity_counts = Counter(inc.severity for inc in incidents)
    rule_counts = Counter(inc.rule_id for inc in incidents)

    comm_list = [inc.extracted.get('comm') for inc in incidents if inc.extracted.get('comm')]
    comm_counts = Counter(comm_list)

    pid_list = [inc.extracted.get('pid') for inc in incidents if inc.extracted.get('pid')]
    pid_counts = Counter(pid_list)

    return {
        'total_lines_scanned': total_lines,
        'total_incidents': total,
        'unique_types': len(type_counts),
        'by_type': dict(type_counts),
        'by_severity': dict(severity_counts),
        'by_rule': dict(rule_counts),
        'top_processes': comm_counts.most_common(top_n),
        'top_pids': pid_counts.most_common(top_n),
    }
```

install-service 命令：

```
SYSTEMD_SERVICE_TEMPLATE = """\
[Unit]
Description=SuSG DetectTool - Linux Abnormal Log Detection Daemon
Documentation=https://github.com/e-wanerer/SuSG2025-DetectTool
After=syslog.target network.target

[Service]
Type=simple
Environment=PYTHONUNBUFFERED=1
ExecStart={detecttool_path} monitor -f {log_file} -c {config_path} --json
Restart=on-failure
RestartSec=5
User=root

StandardOutput=append:{output_dir}/incidents.log
StandardError=append:{output_dir}/error.log

[Install]
```

```
WantedBy=multi-user.target
"""
```

3.3 规则配置设计

规则配置采用 YAML 格式，结构清晰，易于维护。

配置文件结构：

```
version: 1
rules:
  - id: oom_basic
    type: OOM
    severity: high
    keywords_any: ["Out of memory", "Killed process", "oom-kill"]
    regex_any:
      - 'Killed process (?P<pid>\d+)\s+\((?P<comm>[^)]+)\)'
    cooldown_seconds: 30

  - id: oops_basic
    type: OOPS
    severity: high
    keywords_any: ["Oops:", "BUG:", "Unable to handle kernel"]

  - id: panic_basic
    type: PANIC
    severity: critical
    keywords_any: ["Kernel panic - not syncing"]

  - id: reboot_basic
    type: REBOOT
    severity: medium
    keywords_any: ["reboot:", "Restarting system"]

  - id: fs_exception_basic
    type: FS_EXCEPTION
    severity: high
    keywords_any: ["EXT4-fs error", "XFS (", "BTRFS error", "Buffer I/O error", "I/O error"]

  - id: deadlock_hung_task
    type: DEADLOCK
    severity: high
    keywords_any: ["blocked for more than", "hung task", "hung_task"]
    regex_any:
```

```
        - 'INFO:\s+task\s+(?P<comm>.+):(?P<pid>\d+)\s+blocked for more
than\s+(?P<secs>\d+)\s+seconds'
        - 'task\s+(?P<comm>.+):(?P<pid>\d+)\s+blocked for more
than\s+(?P<secs>\d+)\s+seconds'
    cooldown_seconds: 60
```

规则字段说明：

字段	类型	必需	说明
id	string	是	规则唯一标识符
type	string	是	异常类型名称
severity	string	否	严重级别，默认 medium
keywords_any	list	否	任一关键词匹配
keywords_all	list	否	所有关键词匹配
regex_any	list	否	任一正则匹配
regex_all	list	否	所有正则匹配
cooldown_seconds	int	否	冷却时间，默认 0

3.4 数据流转过程

一次完整的异常检测流程如下：

1. 用户通过 CLI 执行命令，指定要分析的日志文件路径和配置文件路径
2. config.py 加载 YAML 规则文件，编译其中的正则表达式，生成 Rule 对象列表
3. file_follow.py（monitor 模式）或直接文件迭代器（scan 模式）逐行读取日志内容
4. 每一行日志首先经过 `Detector` 的关键词和正则匹配判断
5. 如果匹配到需要多行聚合的异常类型（Oops/Panic/Deadlock），MultiLineAggregator 开始收集后续行
6. 当满足聚合结束条件时，生成完整的 `Incident` 对象
7. Cooldown 组件检查该事件的指纹，决定是否输出
8. 最终通过 Rich 表格或 JSON 格式将结果呈现给用户

4. 开发计划

4.1 项目开发阶段划分

本项目的开发周期分为五个阶段：

第一阶段：需求分析与技术调研

主要工作内容：

（1）深入分析比赛题目要求，明确功能边界
（2）调研 Linux 内核日志格式和各类异常事件特征
（3）调研 Python CLI 框架、配置解析、文件监控等技术方案
（4）确定技术选型和整体架构

第二阶段：核心引擎开发

主要工作内容：

（1）设计并实现 Rule 和 Config 数据结构
（2）实现配置文件加载和验证功能
（3）实现关键词匹配和正则表达式匹配算法
（4）实现 Detector 检测器和 Incident 事件模型

第三阶段：功能扩展与完善

主要工作内容：

（1）实现多行日志聚合功能
（2）实现冷却机制防止重复告警
（3）实现文件跟踪和实时监控功能
（4）实现统计分析功能

第四阶段：CLI 与服务化

主要工作内容：

（1）使用 Typer 框架实现命令行接口
（2）使用 Rich 库实现美化输出
（3）实现 systemd 服务安装和管理功能
（4）编写配置规则文件

第五阶段：测试与文档

主要工作内容：

（1）编写单元测试和集成测试
（2）创建示例日志文件用于演示

（3）编写用户文档和 API 文档
（4）进行系统测试和性能优化

4.2 里程碑节点

阶段	里程碑	交付物
第一阶段	完成技术方案设计	技术调研报告、架构设计文档
第二阶段	核心引擎可用	配置模块、检测引擎、基础测试
第三阶段	进阶功能完成	多行聚合、实时监控、统计功能
第四阶段	完整 CLI 可用	四个命令、systemd 服务支持
第五阶段	项目交付	完整测试、文档、示例

5. 比赛过程中的重要进展

5.1 第一阶段：基础框架搭建

在项目初期，我们完成了以下关键工作：

项目结构设计：

采用标准的 Python 项目结构，使用 pyproject.toml 进行现代化的包管理。将源代码放在 src/detecttool 目录下，测试代码放在 tests 目录，配置文件放在 configs 目录。

```
SuSG2025-DetectTool/
├─── src/
│    └─── detecttool/
│         ├─── __init__.py
│         ├─── cli.py
│         ├─── config.py
│         ├─── engine.py
│         └─── sources/
│              ├─── __init__.py
│              └─── file_follow.py
├─── configs/
│    └─── rules.yaml
```

```
├──── tests/
├──── examples/
├──── docs/
└──── pyproject.toml
```

依赖管理配置：

```
[project]
name = "susg-detecttool"
version = "0.1.0"
description = "Linux OS abnormal log detection tool (OOM/Oops/Panic/Reboot/FS...)"
readme = "README.md"
requires-python = ">=3.10"
dependencies = [
    "typer>=0.12",
    "PyYAML>=6.0",
    "rich>=13.0",
]

[project.optional-dependencies]
dev = [
    "pytest>=7.0",
    "pytest-cov>=4.0",
]

[project.scripts]
detecttool = "detecttool.cli:app"
```

5.2 第二阶段：核心检测能力实现

六种异常类型检测规则：

成功实现了比赛要求的六种异常类型检测：

异常类型	规则 ID	检测方式	提取字段
OOM	oom_basic	关键词 + 正则	pid, comm
Oops	oops_basic	关键词匹配	-
Panic	panic_basic	关键词匹配	-

Deadlock	deadlock_hung_task	关键词 + 正则	pid, comm, secs
Reboot	reboot_basic	关键词匹配	-
FS_Exception	fs_exception_basic	关键词匹配	-

字段提取正则表达式设计：

OOM 事件字段提取：

```
Killed process (?P<pid>\d+)\s+\((?P<comm>[^)]+)\)
```

Deadlock 事件字段提取：

```
INFO:\s+task\s+(?P<comm>.+):(?P<pid>\d+)\s+blocked for more
than\s+(?P<secs>\d+)\s+seconds
```

5.3 第三阶段：多行聚合与实时监控

多行聚合器实现：

这是本项目的技术难点之一。内核 Oops 和 Panic 事件通常包含多行堆栈信息，需要将这些信息作为上下文一起捕获。

聚合器的设计挑战：

（1）如何识别多行事件的开始和结束

（2）如何处理时间戳跨度较大的日志

（3）如何在实时监控模式下及时刷新

解决方案：

（1）使用特征关键词识别触发行（Kernel panic、Oops:等）

（2）使用时间窗口（默认 5 秒）限制上下文范围

（3）使用心跳机制（heartbeat）在空闲时触发刷新

实时监控功能实现：

实现了类似 tail -f 的文件跟踪功能，支持：

（1）从文件末尾开始监控（默认）或从开头开始

（2）可配置的轮询间隔

（3）自动检测和处理日志轮转

5.4 第四阶段：统计分析与服务化

统计分析功能：

实现了多维度的统计分析：

（1）按异常类型统计
（2）按严重级别统计
（3）按检测规则统计
（4）Top N 进程排名
（5）Top N PID 排名

Systemd 服务支持：
实现了完整的服务生命周期管理：
（1）install-service：生成并安装 systemd 服务
（2）uninstall-service：停止并卸载服务
（3）service-status：查看服务运行状态

5.5 第五阶段：测试与文档完善

测试覆盖：
编写了完整的测试套件，包括 47 个测试用例：
（1）核心引擎测试（test_engine.py）：20 个测试
（2）统计功能测试（test_stats.py）：16 个测试
（3）CLI 命令测试（test_cli.py）：11 个测试

示例日志文件：

创建了 6 个不同场景的示例日志文件：
（1）sample.log：基础示例，包含 6 种异常
（2）oom_storm.log：OOM 风暴场景
（3）kernel_panic_full.log：完整 Panic 堆栈
（4）deadlock_scenario.log：死锁场景
（5）mixed_production.log：生产环境混合日志
（6）filesystem_errors.log：文件系统错误集合

6. 系统测试情况

6.1 测试环境准备

本节描述在全新 Ubuntu 虚拟机上从零开始部署和测试系统的完整步骤。

6.1.1 虚拟机环境要求

项目	要求

操作系统	Ubuntu 20.04 LTS 或更高版本
Python 版本	Python 3.10 或更高版本
网络	需要访问 GitHub 和 PyPI

6.1.2 基础环境安装

步骤 1：更新系统包

```
sudo apt update && sudo apt upgrade -y
```

步骤 2：安装 Python 3.10+和 pip

```
sudo apt install -y python3 python3-pip python3-venv git
```

步骤 3：验证 Python 版本

```
python3 --version
```

预期输出：Python 3.10.x 或更高版本

6.1.3 项目部署

步骤 1：克隆项目仓库

```
cd ~

git clone https://github.com/MoGuW666/SuSG2025-DetectTool.git

cd SuSG2025-DetectTool
```

步骤 2：创建虚拟环境

使用虚拟环境可以隔离项目依赖，避免与系统 Python 包冲突：

```
python3 -m venv .venv

source .venv/bin/activate
```

激活成功后，命令提示符前会出现 (.venv) 前缀。

步骤 3：安装项目及依赖

```
pip install -e ".[dev]"
```

步骤 4：验证安装

```
detecttool --help
```

预期输出应包含 scan、monitor、stats 等子命令的帮助信息。

6.2 运行自动化测试

6.2.1 测试套件概览

项目的测试套件位于 tests/ 目录，包含以下测试文件：

文件	测试内容	测试用例数
test_engine.py	核心检测引擎（6 种异常检测、字段提取、多行聚合、冷却机制）	17
test_cli.py	CLI 命令（scan、stats 命令的输入输出、错误处理）	13
test_stats.py	统计功能（数据生成、计数准确性、Top N 排名）	17

总计 **47** 个测试用例。

6.2.2 运行全部测试

在项目根目录执行以下命令运行所有测试：

```
pytest
```

示例输出：

```
============================= test session starts ==============================
platform linux -- Python 3.12.3, pytest-9.0.2, pluggy-1.6.0 --
/home/pan/SuSG2025-DetectTool/.venv/bin/python3
cachedir: .pytest_cache
rootdir: /home/pan/SuSG2025-DetectTool
configfile: pytest.ini
testpaths: tests
```

```
plugins: cov-7.0.0
collected 47 items

tests/test_cli.py::TestScanCommand::test_scan_basic PASSED                    [  2%]
tests/test_cli.py::TestScanCommand::test_scan_json_output PASSED              [  4%]
tests/test_cli.py::TestScanCommand::test_scan_detects_all_types PASSED        [  6%]
tests/test_cli.py::TestScanCommand::test_scan_file_not_found PASSED           [  8%]
tests/test_cli.py::TestStatsCommand::test_stats_basic PASSED                  [ 10%]
tests/test_cli.py::TestStatsCommand::test_stats_json_output PASSED            [ 12%]
tests/test_cli.py::TestStatsCommand::test_stats_counts_accuracy PASSED        [ 14%]
tests/test_cli.py::TestStatsCommand::test_stats_top_n_parameter PASSED        [ 17%]
tests/test_cli.py::TestStatsCommand::test_stats_displays_tables PASSED        [ 19%]
tests/test_cli.py::TestStatsCommand::test_stats_file_not_found PASSED         [ 21%]
tests/test_cli.py::TestEdgeCases::test_stats_empty_log PASSED                 [ 23%]
tests/test_cli.py::TestEdgeCases::test_stats_empty_log_json PASSED            [ 25%]
tests/test_engine.py::TestDetection::test_total_incidents_count PASSED        [ 27%]
tests/test_engine.py::TestDetection::test_oom_detection PASSED                [ 29%]
tests/test_engine.py::TestDetection::test_oops_detection PASSED               [ 31%]
tests/test_engine.py::TestDetection::test_panic_detection PASSED              [ 34%]
tests/test_engine.py::TestDetection::test_deadlock_detection PASSED           [ 36%]
tests/test_engine.py::TestDetection::test_reboot_detection PASSED             [ 38%]
tests/test_engine.py::TestDetection::test_fs_exception_detection PASSED       [ 40%]
tests/test_engine.py::TestFieldExtraction::test_oom_field_extraction PASSED   [ 42%]
tests/test_engine.py::TestFieldExtraction::test_deadlock_field_extraction PASSED [ 44%]
tests/test_engine.py::TestMultiLineAggregation::test_panic_has_context PASSED [ 46%]
tests/test_engine.py::TestMultiLineAggregation::test_deadlock_has_context PASSED [ 48%]
tests/test_engine.py::TestMultiLineAggregation::test_oops_has_context PASSED [ 51%]
tests/test_engine.py::TestCooldown::test_cooldown_prevents_duplicates PASSED [ 53%]
tests/test_engine.py::TestCooldown::test_different_process_not_cooldown PASSED [ 55%]
tests/test_engine.py::TestEdgeCases::test_empty_log PASSED                    [ 57%]
tests/test_engine.py::TestEdgeCases::test_no_matches PASSED                   [ 59%]
tests/test_engine.py::TestEdgeCases::test_incident_types_are_unique PASSED    [ 61%]
tests/test_stats.py::TestStatisticsGeneration::test_total_incidents PASSED    [ 63%]
tests/test_stats.py::TestStatisticsGeneration::test_total_lines_scanned PASSED [ 65%]
tests/test_stats.py::TestStatisticsGeneration::test_unique_types_count PASSED [ 68%]
tests/test_stats.py::TestStatisticsGeneration::test_by_type_structure PASSED  [ 70%]
tests/test_stats.py::TestStatisticsGeneration::test_by_severity_structure PASSED [ 72%]
tests/test_stats.py::TestStatisticsGeneration::test_by_rule_structure PASSED  [ 74%]
tests/test_stats.py::TestStatisticsGeneration::test_top_processes_structure PASSED [ 76%]
tests/test_stats.py::TestStatisticsGeneration::test_top_pids_structure PASSED [ 78%]
tests/test_stats.py::TestStatisticsCounts::test_type_counts PASSED            [ 80%]
tests/test_stats.py::TestStatisticsCounts::test_severity_counts PASSED        [ 82%]
tests/test_stats.py::TestStatisticsCounts::test_total_equals_sum PASSED       [ 85%]
tests/test_stats.py::TestStatisticsCounts::test_rule_counts PASSED            [ 87%]
```

```
tests/test_stats.py::TestTopNRankings::test_top_processes_content PASSED [ 89%]
tests/test_stats.py::TestTopNRankings::test_top_pids_content PASSED [ 91%]
tests/test_stats.py::TestTopNRankings::test_top_n_limit PASSED [ 93%]
tests/test_stats.py::TestEdgeCases::test_empty_incidents PASSED [ 95%]
tests/test_stats.py::TestEdgeCases::test_incidents_without_extracted_fields PASSED [ 97%]
tests/test_stats.py::TestEdgeCases::test_duplicate_process_aggregation PASSED [100%]

============================== 47 passed in 0.65s ==============================
```

6.2.3 运行特定类别的测试

可以通过指定文件或标记来运行特定类别的测试：

```
# 只运行引擎测试

pytest tests/test_engine.py

# 只运行 CLI 测试

pytest tests/test_cli.py

# 只运行统计功能测试

pytest tests/test_stats.py
```

6.2.4 生成测试覆盖率报告

```
pytest --cov=detecttool --cov-report=term-missing
```

该命令会输出每个源文件的测试覆盖率百分比，以及未覆盖的代码行号。

6.3 功能验证测试

除了自动化测试，还可以通过手动执行命令来验证各项功能。

6.3.1 验证 scan 命令（日志扫描）

测试目标：验证 `scan` 命令能够正确检测测试日志中的所有 6 种异常类型。

执行命令：

```
detecttool scan -f tests/fixtures/test.log -c configs/rules.yaml
```

示例输出：

Incidents (6)

Line	Type	Severity	Rule	Extracted	Ctx	Message
2	OOM	high	oom_basic	{"pid": "1234", "comm": "python3"}	0	Dec 24 17:40:10 kernel: Out of memory: Killed process 1234 (python3) total-vm:123456kB, anon-rss:7890kB
3	OOPS	high	oops_basic	{}	1	Dec 24 17:40:11 kernel: Oops: 0002 SMP PTI
5	PANIC	critical	panic_basic	{}	2	Dec 24 17:40:13 kernel: Kernel panic - not syncing: Fatal exception
8	FS_EXCEPTION	high	fs_exception_basic	{}	0	Dec 24 17:40:20 kernel: EXT4-fs error (device sda1): ext4_find_entry:1455: inode #2: comm bash: reading directory lblock 0
9	REBOOT	medium	reboot_basic	{}	0	Dec 24 17:40:30 kernel: reboot: Restarting system
10	DEADLOCK	high	deadlock_hung_task	{"comm": "kworker/0:1", "pid": "4321", "secs": "120"}	6	Dec 24 18:05:00 kernel: INFO: task kworker/0:1:4321 blocked for more than 120 seconds.

验证要点：

- 应检测到 6 个异常事件（OOM、OOPS、PANIC、DEADLOCK、REBOOT、FS_EXCEPTION 各 1 个）
- OOM 事件应包含提取的 `pid` 和 `comm` 字段
- DEADLOCK 事件应包含提取的 `pid`、`comm` 和 `secs` 字段
- OOPS、PANIC、DEADLOCK 事件应包含多行上下文（Ctx 列不为 0）

6.3.2 验证 scan 命令 JSON 输出

测试目标：验证 JSON 输出格式的正确性和字段完整性。

执行命令：

```
detecttool scan -f tests/fixtures/test.log -c configs/rules.yaml --json
```

示例输出：

```
[
  {
    "rule_id": "oom_basic",
    "type": "OOM",
    "severity": "high",
    "message": "Dec 24 17:40:10 kernel: Out of memory: Killed process 1234 (python3) total-vm:123456kB, anon-rss:7890kB",
    "line_no": 2,
    "extracted": {
      "pid": "1234",
```

```
        "comm": "python3"
      },
      "context": []
    },
    {
      "rule_id": "oops_basic",
      "type": "OOPS",
      "severity": "high",
      "message": "Dec 24 17:40:11 kernel: Oops: 0002 [#1] SMP PTI",
      "line_no": 3,
      "extracted": {},
      "context": [
        "Dec 24 17:40:12 kernel: Call Trace:"
      ]
    },
    {
      "rule_id": "panic_basic",
      "type": "PANIC",
      "severity": "critical",
      "message": "Dec 24 17:40:13 kernel: Kernel panic - not syncing: Fatal exception",
      "line_no": 5,
      "extracted": {},
      "context": [
        "Dec 24 17:40:14 kernel: panic stack trace line 1",
        "Dec 24 17:40:15 kernel: panic stack trace line 2"
      ]
    },
    {
      "rule_id": "fs_exception_basic",
      "type": "FS_EXCEPTION",
      "severity": "high",
      "message": "Dec 24 17:40:20 kernel: EXT4-fs error (device sda1): ext4_find_entry:1455: inode #2: comm bash: reading directory lblock 0",
      "line_no": 8,
      "extracted": {},
      "context": []
    },
    {
      "rule_id": "reboot_basic",
      "type": "REBOOT",
      "severity": "medium",
      "message": "Dec 24 17:40:30 kernel: reboot: Restarting system",
      "line_no": 9,
      "extracted": {},
```

```
    "context": []
  },
  {
    "rule_id": "deadlock_hung_task",
    "type": "DEADLOCK",
    "severity": "high",
    "message": "Dec 24 18:05:00 kernel: INFO: task kworker/0:1:4321 blocked for more than 120 seconds.",
    "line_no": 10,
    "extracted": {
      "comm": "kworker/0:1",
      "pid": "4321",
      "secs": "120"
    },
    "context": [
      "Dec 24 18:05:00 kernel:          Tainted: G          W          6.8.0 #1",
      "Dec 24 18:05:00 kernel: \"echo 0 > /proc/sys/kernel/hung_task_timeout_secs\" disables this message.",
      "Dec 24 18:05:00 kernel: task:kworker/0:1 state:D stack:0 pid:4321 ppid:2 flags:0x00000008",
      "Dec 24 18:05:00 kernel: Call Trace:",
      "Dec 24 18:05:00 kernel:  schedule+0x2f/0x90",
      "Dec 24 18:05:00 kernel: ---[ end trace 1234567890abcdef ]---"
    ]
  }
]
```


验证要点：

- 输出应为合法的 JSON 数组
- 每个事件对象应包含 rule_id、type、severity、message、line_no、extracted、context 字段

6.3.3 验证 stats 命令（统计分析）

测试目标：验证 stats 命令能够生成正确的统计报表。

执行命令：

```
detecttool stats -f tests/fixtures/test.log -c configs/rules.yaml
```

示例输出：

```
==========================================
    Log Analysis Statistics Report
==========================================

Total lines scanned: 16
Total incidents detected: 6
Unique incident types: 6

              Incidents by Type
┌──────────────────────┬─────────┬──────────────┐
│ Type                 │ Count   │ Percentage   │
├──────────────────────┼─────────┼──────────────┤
│ OOM                  │       1 │        16.7% │
│ OOPS                 │       1 │        16.7% │
│ PANIC                │       1 │        16.7% │
│ FS_EXCEPTION         │       1 │        16.7% │
│ REBOOT               │       1 │        16.7% │
│ DEADLOCK             │       1 │        16.7% │
└──────────────────────┴─────────┴──────────────┘

            Incidents by Severity
┌──────────────────────┬─────────┬──────────────┐
│ Severity             │ Count   │ Percentage   │
├──────────────────────┼─────────┼──────────────┤
│ critical             │       1 │        16.7% │
│ high                 │       4 │        66.7% │
│ medium               │       1 │        16.7% │
└──────────────────────┴─────────┴──────────────┘

      Top 2 Affected Processes
┌────────┬──────────────┬─────────────┐
│   Rank │ Process Name │ Incidents   │
├────────┼──────────────┼─────────────┤
│      1 │ python3      │           1 │
│      2 │ kworker/0:1  │           1 │
└────────┴──────────────┴─────────────┘

     Top 2 Affected PIDs
┌────────┬───────┬─────────────┐
│   Rank │ PID   │ Incidents   │
├────────┼───────┼─────────────┤
│      1 │ 1234  │           1 │
│      2 │ 4321  │           1 │
└────────┴───────┴─────────────┘

 Incidents by Detection Rule
┌──────────────────────┬─────────┐
│ Rule ID              │ Count   │
├──────────────────────┼─────────┤
│ oom_basic            │       1 │
│ oops_basic           │       1 │
│ panic_basic          │       1 │
│ fs_exception_basic   │       1 │
│ reboot_basic         │       1 │
│ deadlock_hung_task   │       1 │
└──────────────────────┴─────────┘

==========================================
```

验证要点：

- "Total incidents detected" 应为 6
- "Unique incident types" 应为 6

- "Incidents by Type" 表格应包含 6 种类型，每种各 1 次
- "Incidents by Severity" 应显示 critical:1, high:4, medium:1
- "Affected Processes" 应包含 python3 和 kworker/0:1

6.3.4 验证 stats 命令 JSON 输出

执行命令：

```
detecttool stats -f tests/fixtures/test.log -c configs/rules.yaml --json
```

示例输出：


```
{
  "total_lines_scanned": 16,
  "total_incidents": 6,
  "unique_types": 6,
  "by_type": {
    "OOM": 1,
    "OOPS": 1,
    "PANIC": 1,
    "FS_EXCEPTION": 1,
    "REBOOT": 1,
    "DEADLOCK": 1
  },
  "by_severity": {
    "high": 4,
    "critical": 1,
    "medium": 1
  },
  "by_rule": {
    "oom_basic": 1,
    "oops_basic": 1,
    "panic_basic": 1,
    "fs_exception_basic": 1,
    "reboot_basic": 1,
    "deadlock_hung_task": 1
  },
  "top_processes": [
    [
      "python3",
      1
    ],
    [
```

```
        "kworker/0:1",
        1
      ]
    ],
    "top_pids": [
      [
        "1234",
        1
      ],
      [
        "4321",
        1
      ]
    ]
}
```

6.3.5 验证 monitor 命令（实时监控）

测试目标：验证实时监控功能能够检测动态写入的日志。

测试步骤：

1. 在终端 1 中启动监控：

```
# 创建临时测试文件

touch /tmp/test_monitor.log

# 启动监控（从头开始读取）

detecttool monitor -f /tmp/test_monitor.log -c configs/rules.yaml --from-start
```

2. 在终端 2 中写入测试日志：

```
# 写入 OOM 测试日志

echo "Dec 25 10:00:00 kernel: Out of memory: Killed process 9999 (test_proc)" >> /tmp/test_monitor.log

# 写入 Deadlock 测试日志

echo "Dec 25 10:00:01 kernel: INFO: task blocked_task:1234 blocked for more than 120 seconds." >> /tmp/test_monitor.log
```

3. 观察终端 1 的输出

示例输出：

```
Monitoring /tmp/test_monitor.log    (Ctrl+C to stop)
Config: configs/rules.yaml | from_start=True | poll=0.2s
OOM (rule=oom_basic, severity=high, line=1)
Dec 25 10:00:00 kernel: Out of memory: Killed process 9999 (test_proc)
extracted={"pid": "9999", "comm": "test_proc"}

DEADLOCK (rule=deadlock_hung_task, severity=high, line=2)
Dec 25 10:00:01 kernel: INFO: task blocked_task:1234 blocked for more than 120 seconds.
extracted={"comm": "blocked_task", "pid": "1234", "secs": "120"}

^CStopped.
```

验证要点：

- 写入 OOM 日志后，终端 1 应立即输出检测到的 OOM 事件
- 写入 Deadlock 日志后，应在约 0.8 秒后输出（等待 idle flush）
- 按 Ctrl+C 停止监控时应显示 "Stopped."

4. 测试完成后清理：

```
rm /tmp/test_monitor.log
```

6.4 异常类型检测验证

6.4.1 验证六种异常类型检测

使用测试日志验证所有六种异常类型都能被正确检测：

```
detecttool scan -f tests/fixtures/test.log --json | python3 -c "
import sys, json
incidents = json.load(sys.stdin)
types = set(i['type'] for i in incidents)
expected = {'OOM', 'OOPS', 'PANIC', 'DEADLOCK', 'REBOOT', 'FS_EXCEPTION'}
print(f'检测到的类型: {types}')
print(f'期望的类型: {expected}')
print(f'是否完全匹配: {types == expected}')
"
```

示例输出：

```
检测到的类型: {'REBOOT', 'OOM', 'DEADLOCK', 'PANIC', 'FS_EXCEPTION', 'OOPS'}
期望的类型: {'PANIC', 'REBOOT', 'OOM', 'DEADLOCK', 'FS_EXCEPTION', 'OOPS'}
是否完全匹配: True
```

6.4.2 验证字段提取功能

```
detecttool scan -f tests/fixtures/test.log --json | python3 -c "
import sys, json
incidents = json.load(sys.stdin)
for i in incidents:
    if i['extracted']:
        print(f\"{i['type']}: {i['extracted']}\")
"
```

示例输出：

```
OOM: {'pid': '1234', 'comm': 'python3'}
DEADLOCK: {'comm': 'kworker/0:1', 'pid': '4321', 'secs': '120'}
```

6.4.3 验证多行上下文聚合

```
detecttool scan -f tests/fixtures/test.log --json | python3 -c "
import sys, json
incidents = json.load(sys.stdin)
for i in incidents:
    if i['context']:
        print(f\"{i['type']}: {len(i['context'])} 行上下文\")
"
```

示例输出：

```
OOPS: 1 行上下文
PANIC: 2 行上下文
DEADLOCK: 6 行上下文
```

6.5 测试数据说明

测试套件使用 `tests/fixtures/test.log` 作为测试数据，该文件包含了所有 6 种异常类型的样本：

```
Dec 24 17:40:01 kernel: Booting Linux on physical CPU 0x0
Dec 24 17:40:10 kernel: Out of memory: Killed process 1234 (python3) total-vm:123456kB,
```

```
anon-rss:7890kB
Dec 24 17:40:11 kernel: Oops: 0002 [#1] SMP PTI
Dec 24 17:40:12 kernel: Call Trace:
Dec 24 17:40:13 kernel: Kernel panic - not syncing: Fatal exception
Dec 24 17:40:14 kernel: panic stack trace line 1
Dec 24 17:40:15 kernel: panic stack trace line 2
Dec 24 17:40:20 kernel: EXT4-fs error (device sda1): ext4_find_entry:1455: inode #2: comm bash: reading directory lblock 0
Dec 24 17:40:30 kernel: reboot: Restarting system
Dec 24 18:05:00 kernel: INFO: task kworker/0:1:4321 blocked for more than 120 seconds.
Dec 24 18:05:00 kernel:       Tainted: G        W        6.8.0 #1
Dec 24 18:05:00 kernel: "echo 0 > /proc/sys/kernel/hung_task_timeout_secs" disables this message.
Dec 24 18:05:00 kernel: task:kworker/0:1 state:D stack:0 pid:4321 ppid:2 flags:0x00000008
Dec 24 18:05:00 kernel: Call Trace:
Dec 24 18:05:00 kernel:  schedule+0x2f/0x90
Dec 24 18:05:00 kernel: ---[ end trace 1234567890abcdef ]---
```

行号	异常类型	说明
2	OOM	包含 pid=1234, comm=python3
3-4	OOPS	触发行 + Call Trace 上下文
5-7	PANIC	触发行 + 两行堆栈上下文
8	FS_EXCEPTION	EXT4 文件系统错误
9	REBOOT	系统重启事件
10-16	DEADLOCK	完整的 Hung Task 报告，包含多行上下文

7. 遇到的主要问题和解决方法

7.1 多行日志聚合的边界处理问题

问题描述：

在实现多行日志聚合功能时，遇到了如何准确判断多行事件结束边界的问题。内核 Oops 和 Panic 事件的日志行数不固定，需要找到可靠的结束标记。

问题分析：

（1）不同内核版本的日志格式可能略有差异
（2）同一时间可能有多个事件交织输出
（3）日志可能被截断或不完整

解决方案：

采用多重结束条件判断机制：

```
_END_MARKERS = (
        "end trace",
        "End trace",
        "end Kernel panic",
        "End Kernel panic",
        "---[ end",
)

#  结束条件判断
if any(m in text for m in _END_MARKERS) or len(self.context) >= self.max_lines:
        return self._emit()
```

（1）检测特定的结束标记字符串
（2）设置最大上下文行数限制（默认 200 行）
（3）使用时间窗口检测（超过 5 秒的时间间隔认为事件结束）
（4）检测新事件触发行时自动结束当前事件

7.2 实时监控的空闲刷新问题

问题描述：

在实时监控模式下，如果日志文件长时间没有新内容，多行聚合器中缓存的事件无法及时输出。

问题分析：

多行聚合器需要等待结束条件才能输出事件，但在实时监控中，结束条件可能迟迟不到来。

解决方案：

引入心跳机制和空闲刷新：

```
def follow_file(..., yield_heartbeat: bool = False):
```

```
    while True:
        line = f.readline()
        if line:
            yield line_no, line
            continue

        time.sleep(poll_interval)
        if yield_heartbeat:
            yield 0, ""   # 心跳信号
```

```
def process(self, line_no: int, line: str) -> List[Incident]:
    # 心跳处理：空行触发空闲刷新
    if line_no == 0 and line == "":
        if self.active_type and (time.time() - self._last_activity_wall) >= self.idle_flush_seconds:
            return self._emit()
        return []
```

当文件没有新内容时，定期发送心跳信号，触发空闲刷新检查。

7.3 日志轮转处理问题

问题描述：

Linux 系统通常使用 logrotate 进行日志轮转，轮转后文件 inode 会改变或文件会被截断，导致监控程序读取失败。

问题分析：

日志轮转有两种常见方式：
（1）重命名旧文件，创建新文件（inode 改变）
（2）截断当前文件（文件大小变小）

解决方案：

通过比较 inode 和文件大小检测轮转：

```
if st2.st_ino != inode or st2.st_size < f.tell():
    # 轮转/截断：重开文件
    try:
        f.close()
    except Exception:
        pass
    f = _open()
    inode = st2.st_ino
```

检测到轮转后自动重新打开文件，保证监控的连续性。

7.4 冷却机制的指纹设计问题

问题描述：

冷却机制用于防止短时间内重复报告相同事件，但如何定义"相同事件"需要仔细考虑。

问题分析：

（1）完全相同的日志行可能表示同一事件的持续
（2）同一进程的不同 OOM 事件应该分别报告
（3）不同进程的相同类型事件应该分别报告

解决方案：

设计了基于多字段的指纹算法：

```
fp = f"{rule.id}|{extracted.get('pid','')}|{extracted.get('comm','')}|{text[:80]}"
if not self.cooldown.allow(fp, rule.cooldown_seconds):
    continue
```

指纹由以下部分组成：
（1）规则 ID：区分不同类型的事件
（2）进程 ID：区分不同进程
（3）进程名：作为进程标识的补充
（4）消息前 80 字符：区分相同进程的不同事件

7.5 Systemd 服务路径问题

问题描述：

安装 systemd 服务时，需要确定 detecttool 可执行文件的正确路径，以及配置文件的绝对路径。

问题分析：

（1）用户可能在虚拟环境中安装 detecttool
（2）相对路径在服务运行时可能失效
（3）不同安装方式导致路径不同

解决方案：

自动检测可执行文件路径：

```
def _find_detecttool_path() -> str:
    # Try to find in PATH
    which_result = shutil.which("detecttool")
    if which_result:
        return which_result
    # Fallback: use python -m detecttool.cli
    return f"{sys.executable} -m detecttool.cli"
```

自动转换配置文件为绝对路径：

```
def _get_absolute_config_path(config: str) -> str:
    config_path = Path(config)
    if config_path.is_absolute():
        return str(config_path)
    return str(Path.cwd() / config_path)
```

7.6 Deadlock 误触发问题

问题描述：

在某些日志中，包含 hung_task_timeout_secs 的提示信息会误触发死锁检测。

问题分析：

内核 hung task 检测信息中包含这样的提示：

```
"echo 0 > /proc/sys/kernel/hung_task_timeout_secs" disables this message.
```

这行日志包含了 hung_task 关键词，但它只是一个提示信息，不是真正的死锁事件。

解决方案：

优化触发条件，只有同时满足多个条件才触发：

```
def _trigger_type(text: str) -> Optional[str]:
    # ...
    # DEADLOCK  只用真正的"起始行"触发
    if ("blocked for more than" in text) and ("task" in text):
        return "DEADLOCK"
    return None
```

同时要求包含 blocked for more than 和 task 两个关键词，排除了单独出现 hung_task 的提示信息。

8. 分工和协作

8.1 团队成员介绍

姓名	角色	主要职责
李尚泽	团队负责人	项目管理、核心算法设计、代码审查
阮野	核心开发	检测引擎开发、多行聚合实现、测试编写
潘胜圆	核心开发	CLI 开发、服务化功能、文档编写

8.2 详细分工

8.2.1 李尚泽的工作内容

（1）**项目架构设计：**负责整体技术架构的设计和评审，确定模块划分和接口定义。

（2）**核心算法设计：**设计规则匹配算法、冷却机制、多行聚合状态机等核心算法。

（3）**技术调研：**调研 Linux 内核日志格式、各类异常事件特征，编写技术调研报告。

（4）**代码审查：**对团队成员提交的代码进行审查，确保代码质量和一致性。

（5）**问题解决：**协助解决开发过程中遇到的技术难题。

8.2.2 阮野的工作内容

（1）**检测引擎开发：**实现 config.py 配置模块和 engine.py 核心引擎模块。

（2）**多行聚合器：**设计并实现 MultiLineAggregator 类，处理 Oops/Panic/Deadlock 的多行日志。

（3）**文件跟踪模块：**实现 file_follow.py，支持日志轮转检测和心跳机制。

（4）**测试用例编写：**编写 test_engine.py 和 test_stats.py 测试用例。

（5）**示例日志制作：**创建 kernel_panic_full.log、deadlock_scenario.log 等示例日志。

8.2.3 潘胜圆的工作内容

（1）**CLI 开发：**使用 Typer 框架实现命令行接口，包括 scan、monitor、stats 命令。

（2）**服务化功能：**实现 install-service、uninstall-service、service-status 命令，支持 systemd 服务管理。

（3）**输出美化：**使用 Rich 库实现表格输出、颜色高亮等美化功能。

（4）**统计分析功能：**实现_generate_statistics 函数和 stats 命令的完整功能。

（5）**文档编写：**编写 README.md 用户文档、tests/README.md 测试文档、本设计文档。

（6）CLI 测试：编写 test_cli.py 测试用例。

（7）**示例日志制作：**创建 sample.log、oom_storm.log、mixed_production.log、filesystem_errors.log 等日志。

8.3 协作方式

代码管理：使用 Git 进行版本控制，通过 GitHub 进行代码托管和协作。

分支策略：采用主干开发模式，所有成员在 main 分支上进行开发，通过频繁提交保持代码同步。

沟通方式：通过即时通讯工具进行日常沟通，定期进行线上或线下会议讨论进度和问题。

文档协作：使用 Markdown 格式编写文档，便于版本控制和协作编辑。

9. 提交仓库目录和文件描述

9.1 仓库整体结构

```
SuSG2025-DetectTool/
├─── .gitattributes                  # Git 属性配置（换行符规范）
├─── .gitignore                      # Git 忽略规则
```

```
├──── LICENSE                          # GPL-3.0 开源许可证
├──── LICENSE.docs                     # 文档许可证（CC BY-SA 4.0）
├──── README.md                        # 项目说明文档
├──── pyproject.toml                   # Python 项目配置文件
├──── pytest.ini                       # Pytest 测试配置
├──── configs/                         # 配置文件目录
│     └──── rules.yaml                 # 检测规则配置
├──── docs/                            # 文档目录
│     └──── Design.pdf                 # 项目设计文档（本文档）
├──── examples/                        # 示例文件目录
│     └──── logs/                      # 示例日志文件
│           ├──── README.md            # 示例日志说明
│           ├──── sample.log           # 基础示例日志
│           ├──── oom_storm.log        # OOM 风暴场景
│           ├──── kernel_panic_full.log  # 完整 Panic 堆栈
│           ├──── deadlock_scenario.log  # 死锁场景
│           ├──── mixed_production.log   # 生产环境混合日志
│           └──── filesystem_errors.log  # 文件系统错误集合
├──── src/                             # 源代码目录
│     └──── detecttool/                # 主程序包
│           ├──── __init__.py          # 包初始化
│           ├──── cli.py               # 命令行接口
│           ├──── config.py            # 配置加载模块
│           ├──── engine.py            # 核心检测引擎
│           └──── sources/             # 数据源模块
│                 ├──── __init__.py    # 包初始化
│                 └──── file_follow.py # 文件跟踪模块
└──── tests/                           # 测试目录
      ├──── __init__.py                # 包初始化
      ├──── README.md                  # 测试说明文档
      ├──── conftest.py                # Pytest 配置和 fixtures
      ├──── test_cli.py                # CLI 命令测试
      ├──── test_engine.py             # 核心引擎测试
      ├──── test_stats.py              # 统计功能测试
      └──── fixtures/                  # 测试数据
            └──── test.log             # 测试用日志文件
```

9.2 核心文件详细说明

9.2.1 源代码文件

src/detecttool/cli.py

命令行接口模块，实现以下功能：

- scan 命令：扫描日志文件，检测异常事件
- monitor 命令：实时监控日志文件
- stats 命令：统计分析检测结果
- install-service 命令：安装 systemd 守护服务
- uninstall-service 命令：卸载守护服务
- service-status 命令：查看服务状态

src/detecttool/config.py

配置加载模块，实现以下功能：

- Rule 数据类定义
- Config 数据类定义
- YAML 配置文件加载和解析
- 正则表达式预编译
- 配置验证和错误处理

src/detecttool/engine.py

核心检测引擎模块，实现以下功能：

- Incident 事件数据类
- 关键词匹配算法
- 正则表达式匹配和字段提取
- Cooldown 冷却控制器
- Detector 检测器类
- MultiLineAggregator 多行聚合器
- detect_lines 便捷函数

src/detecttool/sources/file_follow.py

文件跟踪模块，实现以下功能：

- 类似 tail -f 的文件跟踪
- 日志轮转自动检测
- 心跳信号支持
- 可配置的轮询间隔

9.2.2 配置文件

configs/rules.yaml

检测规则配置文件，定义了 6 条基础规则：

- oom_basic：OOM 内存不足检测
- oops_basic：内核 Oops 错误检测

- panic_basic：Kernel Panic 检测
- reboot_basic：系统重启检测
- fs_exception_basic：文件系统异常检测
- deadlock_hung_task：进程死锁检测

pyproject.tom

Python 项目配置文件，定义：
- 项目元数据（名称、版本、描述）
- 依赖项（typer、PyYAML、rich）
- 可选依赖（pytest、pytest-cov）
- 入口点（detecttool 命令）
- 构建系统配置

pytest.ini

Pytest 测试配置文件，定义：
- 测试文件发现规则
- 测试目录配置
- 命令行默认选项
- 测试标记定义

9.2.3 测试文件

tests/test_engine.py

核心引擎测试，包含：
- TestDetection 类：6 种异常类型检测测试
- TestFieldExtraction 类：字段提取测试
- TestMultiLineAggregation 类：多行聚合测试
- TestCooldown 类：冷却机制测试
- TestEdgeCases 类：边界情况测试

tests/test_cli.py

CLI 命令测试，包含：
- TestScanCommand 类：scan 命令测试
- TestStatsCommand 类：stats 命令测试
- TestEdgeCases 类：边界情况测试

tests/test_stats.py

统计功能测试，包含：

- TestStatisticsGeneration 类：统计生成测试
- TestStatisticsCounts 类：计数准确性测试
- TestTopNRankings 类：Top N 排名测试
- TestEdgeCases 类：边界情况测试

tests/fixtures/test.log

测试用日志文件，包含 6 种异常类型各一个实例，用于验证检测功能的完整性。

9.2.4 示例文件

examples/logs/sample.log

基础示例日志，包含所有 6 种异常类型，适合快速演示和功能验证。

examples/logs/oom_storm.log

OOM 风暴场景，模拟短时间内多个进程被 OOM Killer 终止的情况，用于展示统计功能和 Top N 排名。

examples/logs/kernel_panic_full.log

完整 Kernel Panic 场景，包含：
- 完整的 Oops 信息（NULL pointer dereference）
- 详细的寄存器状态
- 完整的 Call Trace 堆栈（40+行）
- Kernel Panic 结束标记

examples/logs/deadlock_scenario.log

死锁场景，包含多个 hung task 事件，展示不同阻塞时间和进程的检测。

examples/logs/mixed_production.log

生产环境混合日志，包含大量正常日志和少量异常事件，展示从噪音中提取信号的能力。

examples/logs/filesystem_errors.log

文件系统错误集合，覆盖 EXT4、XFS、BTRFS 三种文件系统的各类错误。

9.2.5 文档文件

README.md

项目主文档，包含：

- 项目介绍和功能特性
- 快速开始指南
- 详细使用说明
- 规则配置说明
- 使用场景示例
- 常见问题解答

tests/README.md

测试文档，包含：

- 测试结构说明
- 测试覆盖范围
- 运行测试的方法
- 故障排查指南

examples/logs/README.md

示例日志说明文档，包含：

- 各日志文件的用途说明
- 使用场景建议
- 演示命令示例

docs/Design.pdf

项目设计文档，包含完整的设计说明、开发过程和测试情况。

10. 比赛收获

10.1 技术能力提升

通过本次比赛，团队成员在以下技术领域获得了显著提升：

操作系统内核知识

深入学习了 Linux 内核的多个子系统：
（1）内存管理：理解了 OOM Killer 的工作原理，包括进程选择算法和内存压力检测机制。
（2）进程调度：了解了进程状态（特别是 D 状态不可中断睡眠）和 hung task 检测机制。
（3）文件系统：学习了 EXT4、XFS、BTRFS 等文件系统的错误处理和日志格式。
（4）内核调试：理解了 Oops 和 Panic 的产生原因、日志格式和调试信息含义。

Python 工程实践

掌握了现代 Python 项目的标准开发流程：
（1）使用 pyproject.toml 进行项目配置和依赖管理
（2）使用 Typer 框架开发命令行应用
（3）使用 Rich 库实现终端美化输出
（4）使用 pytest 框架编写单元测试和集成测试
（5）使用 dataclass 简化数据类定义

软件设计能力

提升了软件架构设计和模块化开发能力：
（1）学会了如何将复杂系统分解为独立的模块
（2）理解了接口设计和模块解耦的重要性
（3）掌握了状态机设计方法（多行聚合器）
（4）学会了设计可扩展的规则配置系统

10.2 工程经验积累

问题分析与解决

在开发过程中遇到了多个技术难题，通过分析问题本质、查阅资料、反复调试，最终找到了合理的解决方案。这个过程培养了我们分析问题和解决问题的能力。

代码质量意识

通过编写测试用例和进行代码审查，我们深刻认识到代码质量的重要性。良好的测试覆盖可以及时发现问题，清晰的代码结构便于维护和扩展。

文档编写能力

编写用户文档和设计文档的过程中，我们学会了如何将技术内容以清晰易懂的方式呈现，提高了技术写作能力。

10.3 团队协作经验

分工与协作

通过合理的任务分工和有效的沟通协作，团队成员各司其职，高效完成了项目开发。我们学会了如何在团队中发挥各自的优势，形成合力。

版本控制实践

使用 Git 进行团队协作开发，掌握了分支管理、代码合并、冲突解决等实用技能。

项目管理意识

通过制定开发计划和里程碑节点，我们学会了如何合理安排时间，确保项目按期完成。

10.4 对操作系统的深入理解

本次比赛让我们有机会深入研究 Linux 内核的日志系统和异常处理机制。通过分析真实的内核日志，我们对操作系统的运行原理有了更直观的认识：

（1）理解了内核如何记录和报告各类事件
（2）了解了系统管理员如何通过日志分析定位问题
（3）认识到系统监控和异常检测在生产环境中的重要性

10.5 对未来发展的启示

通过本次比赛，我们对系统软件开发产生了更大的兴趣。在未来的学习和工作中，我们计划：

（1）继续深入学习操作系统内核知识
（2）探索更多系统监控和运维工具的开发
（3）学习和实践更多软件工程方法论
（4）持续关注开源社区，参与开源项目贡献