자동매매 로그 분석 실전

제1장: 트레이딩 로그의 중요성과 설계 원칙

1.1 왜 트레이딩 로그인가

자동매매 시스템에서 로그는 단순한 디버깅 도구가 아닙니다. 로그는 시스템의 블랙박스이자, 전략 개선의 원천이며, 장애 대응의 첫 번째 단서입니다. 로그 없이 운영되는 자동매매 시스템은 계기판 없이 비행하는 것과 같습니다.

트레이딩 로그가 특별한 이유는 다음과 같습니다:

- **재현성**: 시장은 다시 돌아오지 않습니다. 그 순간의 데이터와 결정을 로그로 남겨야 나중에 분석할 수 있습니다
- **감사 추적(Audit Trail)**: 모든 주문과 포지션 변경의 근거를 추적할 수 있어야 합니다
- **장애 분석**: 새벽 3시에 발생한 비정상 청산의 원인을 다음 날 로그로 파악합니다
- **전략 개선**: 로그 데이터에서 패턴을 발견하여 전략을 고도화합니다
- **규정 준수**: 기관 트레이딩에서는 로그 보관이 법적 의무입니다

1.2 트레이딩 로그 분류 체계

효과적인 로그 시스템은 용도별로 명확히 분류됩니다.

```
import logging
import json
import time
from enum import Enum
from dataclasses import dataclass, field, asdict
from typing import Dict, Any, Optional, List
from datetime import datetime

class LogCategory(Enum):
    """트레이딩 로그 카테고리"""
    MARKET_DATA = "market_data"      # 시장 데이터 수신
    SIGNAL = "signal"                # 전략 신호 생성
    ORDER = "order"                  # 주문 생명주기
    FILL = "fill"                    # 체결 정보
    POSITION = "position"            # 포지션 변경
```

```
    RISK = "risk"                    # 리스크 이벤트
    SYSTEM = "system"                # 시스템 상태
    ERROR = "error"                  # 오류 및 예외
    PERFORMANCE = "performance"      # 성능 메트릭
    AUDIT = "audit"                  # 감사 추적


@dataclass
class TradingLogEntry:
    """구조화된 트레이딩 로그 엔트리"""
    timestamp: str
    category: str
    level: str
    strategy: str
    message: str
    data: Dict[str, Any] = field(default_factory=dict)
    correlation_id: Optional[str] = None  # 관련 이벤트 추적용
    sequence_num: int = 0                 # 순서 보장용

    def to_json(self) -> str:
        return json.dumps(asdict(self), ensure_ascii=False)

    @classmethod
    def create(
        cls,
        category: LogCategory,
        level: str,
        strategy: str,
        message: str,
        **kwargs
    ) -> 'TradingLogEntry':
        return cls(
            timestamp=datetime.utcnow().isoformat() + 'Z',
            category=category.value,
            level=level,
            strategy=strategy,
            message=message,
            data=kwargs
        )
```

1.3 로그 레벨 전략

트레이딩 시스템에 최적화된 로그 레벨 전략입니다.

```
class TradingLogLevels:
    """트레이딩 전용 로그 레벨 가이드"""

    GUIDELINES = {
        'CRITICAL': {
            'description': '시스템 중단 수준의 치명적 오류',
```

```
            'examples': [
                '거래소 API 인증 실패',
                '데이터베이스 연결 불가',
                '비상 정지(Emergency Stop) 발동',
                '자본금 보호 한도 도달',
            ],
            'action': '즉시 알림 + 자동 조치',
        },
        'ERROR': {
            'description': '정상 운영에 영향을 주는 오류',
            'examples': [
                '주문 제출 실패 (API 오류)',
                '시장 데이터 수신 중단',
                '포지션 불일치 감지',
                '리스크 한도 위반',
            ],
            'action': '알림 + 로그 기록',
        },
        'WARNING': {
            'description': '잠재적 문제 또는 비정상 상황',
            'examples': [
                '체결률 저하 감지',
                '레이턴시 스파이크',
                '부분 체결 발생',
                '드로다운 경고 수준 접근',
            ],
            'action': '모니터링 강화',
        },
        'INFO': {
            'description': '정상 운영 중 핵심 이벤트',
            'examples': [
                '주문 제출/체결/취소',
                '포지션 진입/청산',
                '전략 신호 발생',
                '리밸런싱 실행',
            ],
            'action': '일반 기록',
        },
        'DEBUG': {
            'description': '상세 디버깅 정보',
            'examples': [
                '지표 계산 결과',
                '호가창 스냅샷',
                '리스크 검증 상세',
                '주문 파라미터 상세',
            ],
            'action': '개발/디버깅 시에만 활성화',
        },
    }
```

1.4 상관 ID(Correlation ID) 기반 이벤트 추적

하나의 매매 결정이 여러 로그 이벤트를 생성합니다. 상관 ID로 연결합니다.

```
import uuid
from contextlib import contextmanager
from contextvars import ContextVar

# 컨텍스트 변수로 상관 ID 관리
_correlation_id: ContextVar[Optional[str]] = ContextVar('correlation_id', default=None)
_sequence_counter: ContextVar[int] = ContextVar('sequence_counter', default=0)


@contextmanager
def trading_context(strategy: str, trigger: str = ''):
    """트레이딩 컨텍스트 — 관련 이벤트를 하나의 ID로 묶음"""
    corr_id = f"{strategy}_{uuid.uuid4().hex[:12]}"
    token = _correlation_id.set(corr_id)
    seq_token = _sequence_counter.set(0)

    try:
        yield corr_id
    finally:
        _correlation_id.reset(token)
        _sequence_counter.reset(seq_token)


def get_correlation_id() -> Optional[str]:
    return _correlation_id.get()


def next_sequence() -> int:
    current = _sequence_counter.get()
    _sequence_counter.set(current + 1)
    return current


class CorrelationTracker:
    """상관 ID 기반 이벤트 체인 추적"""

    def __init__(self):
        self._chains: Dict[str, List[TradingLogEntry]] = {}

    def add_event(self, entry: TradingLogEntry):
        corr_id = entry.correlation_id
        if corr_id:
            if corr_id not in self._chains:
                self._chains[corr_id] = []
            self._chains[corr_id].append(entry)

    def get_chain(self, correlation_id: str) -> List[TradingLogEntry]:
        """상관 ID로 전체 이벤트 체인 조회"""
```

```
        return self._chains.get(correlation_id, [])

    def reconstruct_trade_lifecycle(self, correlation_id: str) -> dict:
        """매매 생명주기 재구성"""
        chain = self.get_chain(correlation_id)
        if not chain:
            return {}

        lifecycle = {
            'correlation_id': correlation_id,
            'start_time': chain[0].timestamp,
            'end_time': chain[-1].timestamp,
            'event_count': len(chain),
            'categories': [e.category for e in chain],
            'errors': [e for e in chain if e.level == 'ERROR'],
            'timeline': []
        }

        for event in chain:
            lifecycle['timeline'].append({
                'time': event.timestamp,
                'category': event.category,
                'message': event.message,
                'level': event.level
            })

        return lifecycle
```

제2장: 구조화 로깅 시스템 구축

2.1 구조화 로거(Structured Logger) 구현

JSON 기반 구조화 로그는 검색, 필터링, 분석이 용이합니다.

```
import sys
import logging
import json
import traceback
from typing import Any, Dict, Optional
from datetime import datetime
from pathlib import Path
from logging.handlers import RotatingFileHandler, TimedRotatingFileHandler

class StructuredFormatter(logging.Formatter):
    """구조화 JSON 로그 포매터"""

    def __init__(self, service_name: str = "trading-bot"):
        super().__init__()
```

```
        self.service_name = service_name

    def format(self, record: logging.LogRecord) -> str:
        log_data = {
            'timestamp': datetime.utcfromtimestamp(record.created).isoformat() + 'Z',
            'level': record.levelname,
            'logger': record.name,
            'message': record.getMessage(),
            'service': self.service_name,
        }

        # 추가 필드 (extra 딕셔너리에서)
        extra_fields = {}
        standard_attrs = {
            'name', 'msg', 'args', 'created', 'relativeCreated',
            'exc_info', 'exc_text', 'stack_info', 'lineno', 'funcName',
            'filename', 'module', 'levelno', 'levelname', 'pathname',
            'thread', 'threadName', 'process', 'processName', 'msecs',
            'taskName', 'message'
        }

        for key, value in record.__dict__.items():
            if key not in standard_attrs and not key.startswith('_'):
                extra_fields[key] = value

        if extra_fields:
            log_data['data'] = extra_fields

        # 예외 정보
        if record.exc_info and record.exc_info[0]:
            log_data['exception'] = {
                'type': record.exc_info[0].__name__,
                'message': str(record.exc_info[1]),
                'traceback': traceback.format_exception(*record.exc_info)
            }

        # 상관 ID
        corr_id = _correlation_id.get()
        if corr_id:
            log_data['correlation_id'] = corr_id
            log_data['sequence'] = next_sequence()

        return json.dumps(log_data, ensure_ascii=False, default=str)


class TradingLogger:
    """트레이딩 전용 구조화 로거"""

    def __init__(
        self,
        name: str,
        log_dir: str = "logs",
        max_bytes: int = 100 * 1024 * 1024,  # 100MB
```

```
        backup_count: int = 30,
        console_level: int = logging.INFO,
        file_level: int = logging.DEBUG
    ):
        self.logger = logging.getLogger(name)
        self.logger.setLevel(logging.DEBUG)
        self.logger.handlers.clear()

        log_path = Path(log_dir)
        log_path.mkdir(parents=True, exist_ok=True)

        formatter = StructuredFormatter(service_name=name)

        # 콘솔 핸들러
        console = logging.StreamHandler(sys.stdout)
        console.setLevel(console_level)
        console.setFormatter(formatter)
        self.logger.addHandler(console)

        # 전체 로그 파일 (로테이션)
        all_handler = RotatingFileHandler(
            log_path / "trading.log",
            maxBytes=max_bytes,
            backupCount=backup_count,
            encoding='utf-8'
        )
        all_handler.setLevel(file_level)
        all_handler.setFormatter(formatter)
        self.logger.addHandler(all_handler)

        # 카테고리별 분리 로그
        self._category_handlers = {}
        for category in LogCategory:
            handler = RotatingFileHandler(
                log_path / f"{category.value}.log",
                maxBytes=max_bytes // 5,
                backupCount=10,
                encoding='utf-8'
            )
            handler.setLevel(logging.DEBUG)
            handler.setFormatter(formatter)
            handler.addFilter(CategoryFilter(category.value))
            self.logger.addHandler(handler)
            self._category_handlers[category.value] = handler

        # 에러 전용 파일
        error_handler = RotatingFileHandler(
            log_path / "errors.log",
            maxBytes=max_bytes // 2,
            backupCount=30,
            encoding='utf-8'
        )
        error_handler.setLevel(logging.ERROR)
```

```
        error_handler.setFormatter(formatter)
        self.logger.addHandler(error_handler)

    def order(self, message: str, **kwargs):
        """주문 관련 로그"""
        self.logger.info(message, extra={'category': 'order', **kwargs})

    def fill(self, message: str, **kwargs):
        """체결 관련 로그"""
        self.logger.info(message, extra={'category': 'fill', **kwargs})

    def signal(self, message: str, **kwargs):
        """전략 신호 로그"""
        self.logger.info(message, extra={'category': 'signal', **kwargs})

    def position(self, message: str, **kwargs):
        """포지션 변경 로그"""
        self.logger.info(message, extra={'category': 'position', **kwargs})

    def risk(self, message: str, level: str = 'WARNING', **kwargs):
        """리스크 이벤트 로그"""
        log_func = getattr(self.logger, level.lower(), self.logger.warning)
        log_func(message, extra={'category': 'risk', **kwargs})

    def market_data(self, message: str, **kwargs):
        """시장 데이터 로그"""
        self.logger.debug(message, extra={'category': 'market_data', **kwargs})

    def perf(self, message: str, **kwargs):
        """성능 메트릭 로그"""
        self.logger.info(message, extra={'category': 'performance', **kwargs})

    def audit(self, message: str, **kwargs):
        """감사 추적 로그"""
        self.logger.info(message, extra={'category': 'audit', **kwargs})


class CategoryFilter(logging.Filter):
    """카테고리별 필터"""

    def __init__(self, category: str):
        super().__init__()
        self.category = category

    def filter(self, record: logging.LogRecord) -> bool:
        return getattr(record, 'category', '') == self.category
```

2.2 주문 생명주기 로깅

주문의 전체 생명주기를 빠짐없이 추적합니다.

```
from enum import Enum
from typing import Optional
from dataclasses import dataclass
import time

class OrderStatus(Enum):
    CREATED = "created"
    SUBMITTED = "submitted"
    ACKNOWLEDGED = "acknowledged"
    PARTIALLY_FILLED = "partially_filled"
    FILLED = "filled"
    CANCELLED = "cancelled"
    REJECTED = "rejected"
    EXPIRED = "expired"
    FAILED = "failed"


@dataclass
class OrderEvent:
    """주문 이벤트 기록"""
    order_id: str
    status: OrderStatus
    timestamp_ns: int
    details: Dict[str, Any]


class OrderLifecycleLogger:
    """주문 생명주기 전체 추적 로거"""

    def __init__(self, logger: TradingLogger):
        self.logger = logger
        self._active_orders: Dict[str, List[OrderEvent]] = {}

    def on_order_created(
        self,
        order_id: str,
        symbol: str,
        side: str,
        order_type: str,
        quantity: float,
        price: Optional[float] = None,
        strategy: str = "",
        reason: str = ""
    ):
        """주문 생성 시점"""
        event = OrderEvent(
            order_id=order_id,
            status=OrderStatus.CREATED,
            timestamp_ns=time.time_ns(),
            details={
                'symbol': symbol, 'side': side, 'type': order_type,
                'quantity': quantity, 'price': price,
                'strategy': strategy, 'reason': reason
```

```
            }
        )

        self._active_orders[order_id] = [event]

        self.logger.order(
            f"주문 생성: {side} {quantity} {symbol} @ {price or 'MARKET'}",
            order_id=order_id,
            symbol=symbol,
            side=side,
            order_type=order_type,
            quantity=quantity,
            price=price,
            strategy=strategy,
            reason=reason
        )

    def on_order_submitted(self, order_id: str, exchange_order_id: str = ""):
        """주문 제출 완료"""
        event = OrderEvent(
            order_id=order_id,
            status=OrderStatus.SUBMITTED,
            timestamp_ns=time.time_ns(),
            details={'exchange_order_id': exchange_order_id}
        )

        if order_id in self._active_orders:
            self._active_orders[order_id].append(event)

            # 생성 → 제출 지연 시간 계산
            created_ns = self._active_orders[order_id][0].timestamp_ns
            submit_latency_us = (event.timestamp_ns - created_ns) / 1000

            self.logger.order(
                f"주문 제출 완료 (지연: {submit_latency_us:.0f}μs)",
                order_id=order_id,
                exchange_order_id=exchange_order_id,
                submit_latency_us=submit_latency_us
            )

    def on_order_filled(
        self,
        order_id: str,
        fill_price: float,
        fill_quantity: float,
        commission: float = 0,
        is_partial: bool = False
    ):
        """주문 체결"""
        status = OrderStatus.PARTIALLY_FILLED if is_partial else OrderStatus.FILLED
        event = OrderEvent(
            order_id=order_id,
            status=status,
```

```
            timestamp_ns=time.time_ns(),
            details={
                'fill_price': fill_price,
                'fill_quantity': fill_quantity,
                'commission': commission,
                'is_partial': is_partial
            }
        )

        if order_id in self._active_orders:
            self._active_orders[order_id].append(event)

            # 전체 생명주기 시간
            created_ns = self._active_orders[order_id][0].timestamp_ns
            lifecycle_ms = (event.timestamp_ns - created_ns) / 1_000_000

            # 슬리피지 계산
            original_price = self._active_orders[order_id][0].details.get('price')
            slippage = 0
            if original_price and original_price > 0:
                slippage = (fill_price - original_price) / original_price * 10000  # bps

            self.logger.fill(
                f"체결: {fill_quantity} @ {fill_price} (수수료: {commission})",
                order_id=order_id,
                fill_price=fill_price,
                fill_quantity=fill_quantity,
                commission=commission,
                slippage_bps=round(slippage, 2),
                lifecycle_ms=round(lifecycle_ms, 2),
                is_partial=is_partial
            )

            if not is_partial:
                self._finalize_order(order_id)

    def on_order_rejected(self, order_id: str, reason: str):
        """주문 거부"""
        self.logger.order(
            f"주문 거부: {reason}",
            order_id=order_id,
            rejection_reason=reason
        )
        self._finalize_order(order_id)

    def on_order_cancelled(self, order_id: str, reason: str = ""):
        """주문 취소"""
        self.logger.order(
            f"주문 취소: {reason or '사용자 요청'}",
            order_id=order_id,
            cancel_reason=reason
        )
        self._finalize_order(order_id)
```

```
    def _finalize_order(self, order_id: str):
        """주문 완료 — 전체 생명주기 요약 기록"""
        events = self._active_orders.pop(order_id, [])
        if not events:
            return

        lifecycle_summary = {
            'order_id': order_id,
            'total_events': len(events),
            'statuses': [e.status.value for e in events],
            'duration_ms': (events[-1].timestamp_ns - events[0].timestamp_ns) / 1e6,
            'final_status': events[-1].status.value,
        }

        self.logger.audit(
            f"주문 생명주기 완료: {order_id}",
            **lifecycle_summary
        )

    def get_active_orders_summary(self) -> List[dict]:
        """현재 활성 주문 요약"""
        summaries = []
        for order_id, events in self._active_orders.items():
            summaries.append({
                'order_id': order_id,
                'current_status': events[-1].status.value,
                'age_ms': (time.time_ns() - events[0].timestamp_ns) / 1e6,
                'event_count': len(events)
            })
        return summaries
```

2.3 비동기 로그 버퍼링

로그 기록이 매매 로직의 성능에 영향을 주지 않도록 비동기 버퍼를 사용합니다.

```
import asyncio
from asyncio import Queue
from typing import Callable

class AsyncLogBuffer:
    """비동기 로그 버퍼 — 매매 루프 성능 보호"""

    def __init__(
        self,
        flush_interval: float = 1.0,
        max_buffer_size: int = 10000,
        writers: Optional[List[Callable]] = None
    ):
        self._queue: Queue = Queue(maxsize=max_buffer_size)
```

```
        self._flush_interval = flush_interval
        self._writers = writers or []
        self._running = False
        self._dropped_count = 0
        self._total_count = 0

    def emit(self, log_entry: dict):
        """로그 기록 — 논블로킹, 큐 포화 시 드롭"""
        self._total_count += 1
        try:
            self._queue.put_nowait(log_entry)
        except asyncio.QueueFull:
            self._dropped_count += 1

    async def start(self):
        """백그라운드 플러시 루프 시작"""
        self._running = True
        while self._running:
            await self._flush()
            await asyncio.sleep(self._flush_interval)

        # 종료 시 잔여 로그 플러시
        await self._flush()

    async def _flush(self):
        """버퍼의 로그를 일괄 기록"""
        batch = []
        while not self._queue.empty() and len(batch) < 1000:
            try:
                entry = self._queue.get_nowait()
                batch.append(entry)
            except asyncio.QueueEmpty:
                break

        if not batch:
            return

        for writer in self._writers:
            try:
                if asyncio.iscoroutinefunction(writer):
                    await writer(batch)
                else:
                    writer(batch)
            except Exception as e:
                print(f"로그 기록 실패: {e}")

    def stop(self):
        self._running = False

    @property
    def stats(self) -> dict:
        return {
            'total': self._total_count,
```

```
            'dropped': self._dropped_count,
            'drop_rate': self._dropped_count / max(self._total_count, 1),
            'queue_size': self._queue.qsize()
        }
```

제3장: 로그 수집과 저장 파이프라인

3.1 파일 기반 로그 저장

파일 시스템은 가장 기본적이면서도 신뢰성 높은 로그 저장소입니다.

```
import gzip
import shutil
from pathlib import Path
from datetime import datetime, timedelta

class LogFileManager:
    """트레이딩 로그 파일 관리자"""

    def __init__(self, base_dir: str = "logs"):
        self.base_dir = Path(base_dir)
        self.base_dir.mkdir(parents=True, exist_ok=True)

    def get_log_path(self, category: str, date: Optional[datetime] = None) -> Path:
        """날짜별 로그 파일 경로"""
        date = date or datetime.utcnow()
        date_dir = self.base_dir / date.strftime('%Y-%m-%d')
        date_dir.mkdir(parents=True, exist_ok=True)
        return date_dir / f"{category}.jsonl"

    def write_batch(self, category: str, entries: List[dict]):
        """배치 기록 — 원자적 쓰기"""
        path = self.get_log_path(category)

        # 원자적 쓰기: 임시 파일 → 이름 변경
        lines = '\n'.join(json.dumps(e, ensure_ascii=False) for e in entries)

        with open(path, 'a', encoding='utf-8') as f:
            f.write(lines + '\n')

    def compress_old_logs(self, days_threshold: int = 7):
        """오래된 로그 파일을 gzip으로 압축"""
        cutoff = datetime.utcnow() - timedelta(days=days_threshold)

        for date_dir in sorted(self.base_dir.iterdir()):
            if not date_dir.is_dir():
                continue
```

```
            try:
                dir_date = datetime.strptime(date_dir.name, '%Y-%m-%d')
            except ValueError:
                continue

            if dir_date >= cutoff:
                continue

            for log_file in date_dir.glob('*.jsonl'):
                gz_path = log_file.with_suffix('.jsonl.gz')
                if gz_path.exists():
                    continue

                with open(log_file, 'rb') as f_in:
                    with gzip.open(gz_path, 'wb', compresslevel=6) as f_out:
                        shutil.copyfileobj(f_in, f_out)

                log_file.unlink()
                print(f"압축: {log_file} → {gz_path}")

    def delete_old_logs(self, days_threshold: int = 90):
        """오래된 로그 삭제"""
        cutoff = datetime.utcnow() - timedelta(days=days_threshold)

        for date_dir in sorted(self.base_dir.iterdir()):
            if not date_dir.is_dir():
                continue

            try:
                dir_date = datetime.strptime(date_dir.name, '%Y-%m-%d')
            except ValueError:
                continue

            if dir_date < cutoff:
                shutil.rmtree(date_dir)
                print(f"삭제: {date_dir}")

    def get_disk_usage(self) -> dict:
        """로그 디스크 사용량 조회"""
        total_size = 0
        file_count = 0
        category_sizes = {}

        for path in self.base_dir.rglob('*'):
            if path.is_file():
                size = path.stat().st_size
                total_size += size
                file_count += 1

                category = path.stem.split('.')[0]
                category_sizes[category] = category_sizes.get(category, 0) + size

        return {
```

```
            'total_bytes': total_size,
            'total_mb': round(total_size / 1024 / 1024, 2),
            'file_count': file_count,
            'by_category': {
                k: round(v / 1024 / 1024, 2)
                for k, v in sorted(category_sizes.items(), key=lambda x: -x[1])
            }
        }
```

3.2 SQLite 기반 로그 저장소

빠른 쿼리가 필요한 핵심 로그는 SQLite에 저장합니다.

```
import sqlite3
from contextlib import contextmanager
from typing import Generator

class SQLiteLogStore:
    """SQLite 기반 트레이딩 로그 저장소"""

    def __init__(self, db_path: str = "logs/trading_logs.db"):
        self.db_path = db_path
        self._init_db()

    def _init_db(self):
        """데이터베이스 초기화"""
        with self._connect() as conn:
            conn.executescript("""
                -- 주문 로그 테이블
                CREATE TABLE IF NOT EXISTS order_logs (
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    timestamp TEXT NOT NULL,
                    order_id TEXT NOT NULL,
                    correlation_id TEXT,
                    strategy TEXT NOT NULL,
                    symbol TEXT NOT NULL,
                    side TEXT NOT NULL,
                    order_type TEXT NOT NULL,
                    quantity REAL NOT NULL,
                    price REAL,
                    status TEXT NOT NULL,
                    fill_price REAL,
                    fill_quantity REAL,
                    commission REAL DEFAULT 0,
                    slippage_bps REAL DEFAULT 0,
                    latency_us REAL,
                    error_message TEXT,
                    raw_data TEXT
                );
```

```
                -- 포지션 변경 로그
                CREATE TABLE IF NOT EXISTS position_logs (
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    timestamp TEXT NOT NULL,
                    strategy TEXT NOT NULL,
                    symbol TEXT NOT NULL,
                    action TEXT NOT NULL,
                    quantity_before REAL,
                    quantity_after REAL,
                    avg_price REAL,
                    realized_pnl REAL,
                    unrealized_pnl REAL,
                    trigger_order_id TEXT
                );

                -- 리스크 이벤트 로그
                CREATE TABLE IF NOT EXISTS risk_events (
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    timestamp TEXT NOT NULL,
                    strategy TEXT,
                    event_type TEXT NOT NULL,
                    severity TEXT NOT NULL,
                    description TEXT,
                    metrics TEXT,
                    action_taken TEXT
                );

                -- 성능 메트릭 로그
                CREATE TABLE IF NOT EXISTS performance_metrics (
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    timestamp TEXT NOT NULL,
                    metric_name TEXT NOT NULL,
                    value REAL NOT NULL,
                    tags TEXT
                );

                -- 인덱스
                CREATE INDEX IF NOT EXISTS idx_order_timestamp ON order_logs(timestamp);
                CREATE INDEX IF NOT EXISTS idx_order_strategy ON order_logs(strategy);
                CREATE INDEX IF NOT EXISTS idx_order_symbol ON order_logs(symbol);
                CREATE INDEX IF NOT EXISTS idx_order_correlation ON order_logs(correlation_i
                CREATE INDEX IF NOT EXISTS idx_position_timestamp ON position_logs(timestamp
                CREATE INDEX IF NOT EXISTS idx_risk_timestamp ON risk_events(timestamp);
                CREATE INDEX IF NOT EXISTS idx_perf_timestamp ON performance_metrics(timesta
                CREATE INDEX IF NOT EXISTS idx_perf_name ON performance_metrics(metric_name)
            """)

    @contextmanager
    def _connect(self) -> Generator[sqlite3.Connection, None, None]:
        conn = sqlite3.connect(self.db_path)
        conn.row_factory = sqlite3.Row
        conn.execute("PRAGMA journal_mode=WAL")  # WAL 모드 — 읽기/쓰기 병렬
        conn.execute("PRAGMA synchronous=NORMAL")
```

```
        try:
            yield conn
            conn.commit()
        except Exception:
            conn.rollback()
            raise
        finally:
            conn.close()

    def insert_order_log(self, **kwargs):
        """주문 로그 삽입"""
        with self._connect() as conn:
            conn.execute("""
                INSERT INTO order_logs
                (timestamp, order_id, correlation_id, strategy, symbol,
                 side, order_type, quantity, price, status,
                 fill_price, fill_quantity, commission, slippage_bps,
                 latency_us, error_message, raw_data)
                VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
            """, (
                kwargs.get('timestamp', datetime.utcnow().isoformat()),
                kwargs.get('order_id', ''),
                kwargs.get('correlation_id'),
                kwargs.get('strategy', ''),
                kwargs.get('symbol', ''),
                kwargs.get('side', ''),
                kwargs.get('order_type', ''),
                kwargs.get('quantity', 0),
                kwargs.get('price'),
                kwargs.get('status', ''),
                kwargs.get('fill_price'),
                kwargs.get('fill_quantity'),
                kwargs.get('commission', 0),
                kwargs.get('slippage_bps', 0),
                kwargs.get('latency_us'),
                kwargs.get('error_message'),
                kwargs.get('raw_data')
            ))

    def query_orders(
        self,
        strategy: Optional[str] = None,
        symbol: Optional[str] = None,
        start_time: Optional[str] = None,
        end_time: Optional[str] = None,
        status: Optional[str] = None,
        limit: int = 100
    ) -> List[dict]:
        """주문 로그 조건 검색"""
        conditions = []
        params = []

        if strategy:
```

```
            conditions.append("strategy = ?")
            params.append(strategy)
        if symbol:
            conditions.append("symbol = ?")
            params.append(symbol)
        if start_time:
            conditions.append("timestamp >= ?")
            params.append(start_time)
        if end_time:
            conditions.append("timestamp <= ?")
            params.append(end_time)
        if status:
            conditions.append("status = ?")
            params.append(status)

        where = f"WHERE {' AND '.join(conditions)}" if conditions else ""

        with self._connect() as conn:
            rows = conn.execute(
                f"SELECT * FROM order_logs {where} ORDER BY timestamp DESC LIMIT ?",
                params + [limit]
            ).fetchall()

        return [dict(row) for row in rows]

    def get_daily_summary(self, date: str) -> dict:
        """일일 거래 요약"""
        with self._connect() as conn:
            stats = conn.execute("""
                SELECT
                    COUNT(*) as total_orders,
                    SUM(CASE WHEN status = 'filled' THEN 1 ELSE 0 END) as filled,
                    SUM(CASE WHEN status = 'rejected' THEN 1 ELSE 0 END) as rejected,
                    SUM(CASE WHEN status = 'cancelled' THEN 1 ELSE 0 END) as cancelled,
                    AVG(slippage_bps) as avg_slippage_bps,
                    AVG(latency_us) as avg_latency_us,
                    SUM(commission) as total_commission,
                    AVG(CASE WHEN status = 'filled' THEN fill_price END) as avg_fill_price
                FROM order_logs
                WHERE timestamp LIKE ?
            """, (f"{date}%",)).fetchone()

        return dict(stats) if stats else {}
```

3.3 ELK 스택 연동

대규모 로그 분석을 위한 Elasticsearch 연동입니다.

```
from elasticsearch import Elasticsearch, helpers
from typing import List, Dict
```

```
class ElasticsearchLogShipper:
    """Elasticsearch로 로그 전송"""

    def __init__(
        self,
        hosts: List[str] = ["http://localhost:9200"],
        index_prefix: str = "trading-logs"
    ):
        self.es = Elasticsearch(hosts)
        self.index_prefix = index_prefix

    def _get_index_name(self, date: Optional[datetime] = None) -> str:
        date = date or datetime.utcnow()
        return f"{self.index_prefix}-{date.strftime('%Y.%m.%d')}"

    def create_index_template(self):
        """인덱스 템플릿 생성"""
        template = {
            "index_patterns": [f"{self.index_prefix}-*"],
            "mappings": {
                "properties": {
                    "timestamp": {"type": "date"},
                    "level": {"type": "keyword"},
                    "category": {"type": "keyword"},
                    "strategy": {"type": "keyword"},
                    "symbol": {"type": "keyword"},
                    "order_id": {"type": "keyword"},
                    "correlation_id": {"type": "keyword"},
                    "side": {"type": "keyword"},
                    "status": {"type": "keyword"},
                    "quantity": {"type": "float"},
                    "price": {"type": "float"},
                    "fill_price": {"type": "float"},
                    "slippage_bps": {"type": "float"},
                    "latency_us": {"type": "float"},
                    "commission": {"type": "float"},
                    "pnl": {"type": "float"},
                    "message": {"type": "text"},
                    "data": {"type": "object", "enabled": False}
                }
            },
            "settings": {
                "number_of_shards": 1,
                "number_of_replicas": 0,
                "refresh_interval": "5s"
            }
        }
        self.es.indices.put_template(name=self.index_prefix, body=template)

    def ship_batch(self, entries: List[dict]):
        """배치 전송"""
        actions = []
```

```
        for entry in entries:
            actions.append({
                "_index": self._get_index_name(),
                "_source": entry
            })

        if actions:
            success, errors = helpers.bulk(self.es, actions, raise_on_error=False)
            if errors:
                print(f"ES 전송 오류: {len(errors)}건")
            return success

    def search_orders(
        self,
        strategy: Optional[str] = None,
        time_range: str = "24h",
        status: Optional[str] = None,
        size: int = 100
    ) -> List[dict]:
        """주문 로그 검색"""
        must = [{"range": {"timestamp": {"gte": f"now-{time_range}"}}}]

        if strategy:
            must.append({"term": {"strategy": strategy}})
        if status:
            must.append({"term": {"status": status}})

        result = self.es.search(
            index=f"{self.index_prefix}-*",
            body={
                "query": {"bool": {"must": must}},
                "sort": [{"timestamp": "desc"}],
                "size": size
            }
        )

        return [hit["_source"] for hit in result["hits"]["hits"]]
```

제4장: 로그 분석 도구와 패턴

4.1 실시간 로그 분석기

```
import re
from collections import Counter, defaultdict
from typing import Callable

class RealTimeLogAnalyzer:
    """실시간 로그 스트림 분석기"""
```

```
    def __init__(self):
        self._error_counter = Counter()
        self._latency_buffer = defaultdict(list)
        self._order_stats = defaultdict(lambda: {'total': 0, 'filled': 0, 'rejected': 0})
        self._alert_rules: List[Callable] = []
        self._window_seconds = 300  # 5분 윈도우

    def process_entry(self, entry: dict):
        """로그 엔트리 실시간 처리"""
        category = entry.get('category', '')
        level = entry.get('level', '')

        # 에러 카운팅
        if level in ('ERROR', 'CRITICAL'):
            error_key = f"{category}:{entry.get('message', '')[:50]}"
            self._error_counter[error_key] += 1

        # 카테고리별 처리
        if category == 'order':
            self._process_order_log(entry)
        elif category == 'performance':
            self._process_performance_log(entry)

        # 알림 규칙 평가
        for rule in self._alert_rules:
            alert = rule(entry, self.get_stats())
            if alert:
                self._handle_alert(alert)

    def _process_order_log(self, entry: dict):
        """주문 로그 통계 수집"""
        data = entry.get('data', {})
        strategy = data.get('strategy', 'unknown')

        self._order_stats[strategy]['total'] += 1

        status = data.get('status', '')
        if status == 'filled':
            self._order_stats[strategy]['filled'] += 1
        elif status == 'rejected':
            self._order_stats[strategy]['rejected'] += 1

        # 레이턴시 수집
        latency = data.get('latency_us') or data.get('submit_latency_us')
        if latency:
            self._latency_buffer[strategy].append(latency)
            # 윈도우 크기 제한
            if len(self._latency_buffer[strategy]) > 10000:
                self._latency_buffer[strategy] = self._latency_buffer[strategy][-5000:]

    def _process_performance_log(self, entry: dict):
        """성능 로그 처리"""
        data = entry.get('data', {})
```

```
        metric = data.get('metric_name', '')
        value = data.get('value', 0)

        if metric == 'latency':
            segment = data.get('segment', 'unknown')
            self._latency_buffer[segment].append(value)

    def get_stats(self) -> dict:
        """현재 통계 조회"""
        latency_stats = {}
        for key, samples in self._latency_buffer.items():
            if samples:
                arr = np.array(samples[-1000:])
                latency_stats[key] = {
                    'mean': float(np.mean(arr)),
                    'p50': float(np.median(arr)),
                    'p99': float(np.percentile(arr, 99)),
                    'max': float(np.max(arr)),
                    'samples': len(samples)
                }

        return {
            'error_counts': dict(self._error_counter.most_common(10)),
            'order_stats': dict(self._order_stats),
            'latency': latency_stats
        }

    def add_alert_rule(self, rule: Callable):
        """알림 규칙 추가"""
        self._alert_rules.append(rule)

    def _handle_alert(self, alert: dict):
        """알림 처리"""
        print(f"🚨 ALERT: {alert}")


def error_spike_rule(entry: dict, stats: dict) -> Optional[dict]:
    """에러 급증 감지 규칙"""
    total_errors = sum(stats.get('error_counts', {}).values())
    if total_errors > 50:  # 5분 내 50건 이상
        return {
            'type': 'error_spike',
            'count': total_errors,
            'top_errors': list(stats['error_counts'].items())[:3]
        }
    return None


def rejection_rate_rule(entry: dict, stats: dict) -> Optional[dict]:
    """주문 거부율 이상 감지"""
    for strategy, s in stats.get('order_stats', {}).items():
        if s['total'] > 10:
            rejection_rate = s['rejected'] / s['total']
```

```
            if rejection_rate > 0.1:  # 10% 이상
                return {
                    'type': 'high_rejection_rate',
                    'strategy': strategy,
                    'rate': rejection_rate,
                    'total': s['total']
                }
    return None
```

4.2 로그 기반 트레이딩 리포트 생성

```
import numpy as np
from datetime import datetime, timedelta

class TradingReportGenerator:
    """로그 데이터 기반 트레이딩 리포트 생성"""

    def __init__(self, log_store: SQLiteLogStore):
        self.store = log_store

    def daily_report(self, date: str) -> dict:
        """일일 트레이딩 리포트"""
        orders = self.store.query_orders(
            start_time=f"{date}T00:00:00",
            end_time=f"{date}T23:59:59",
            limit=10000
        )

        if not orders:
            return {'date': date, 'message': '거래 없음'}

        # 전략별 분석
        strategy_stats = defaultdict(lambda: {
            'orders': 0, 'fills': 0, 'rejects': 0,
            'total_volume': 0, 'total_commission': 0,
            'slippages': [], 'latencies': []
        })

        for order in orders:
            s = strategy_stats[order['strategy']]
            s['orders'] += 1

            if order['status'] == 'filled':
                s['fills'] += 1
                if order['fill_quantity']:
                    s['total_volume'] += order['fill_quantity'] * (order['fill_price'] or 0)
                if order['commission']:
                    s['total_commission'] += order['commission']
                if order['slippage_bps']:
                    s['slippages'].append(order['slippage_bps'])
            elif order['status'] == 'rejected':
```

```
                s['rejects'] += 1

            if order['latency_us']:
                s['latencies'].append(order['latency_us'])

        # 리포트 구성
        report = {
            'date': date,
            'total_orders': len(orders),
            'strategies': {}
        }

        for strategy, stats in strategy_stats.items():
            slippages = np.array(stats['slippages']) if stats['slippages'] else np.array([0]
            latencies = np.array(stats['latencies']) if stats['latencies'] else np.array([0]

            report['strategies'][strategy] = {
                'orders': stats['orders'],
                'fill_rate': stats['fills'] / max(stats['orders'], 1),
                'rejection_rate': stats['rejects'] / max(stats['orders'], 1),
                'total_volume': round(stats['total_volume'], 2),
                'total_commission': round(stats['total_commission'], 2),
                'slippage': {
                    'mean_bps': round(float(np.mean(slippages)), 2),
                    'max_bps': round(float(np.max(slippages)), 2),
                },
                'latency': {
                    'mean_us': round(float(np.mean(latencies)), 0),
                    'p99_us': round(float(np.percentile(latencies, 99)), 0),
                },
            }

        return report

    def error_report(self, hours: int = 24) -> dict:
        """에러 분석 리포트"""
        with self.store._connect() as conn:
            rows = conn.execute("""
                SELECT
                    event_type,
                    severity,
                    strategy,
                    COUNT(*) as count,
                    MIN(timestamp) as first_seen,
                    MAX(timestamp) as last_seen
                FROM risk_events
                WHERE timestamp >= datetime('now', ?)
                GROUP BY event_type, severity, strategy
                ORDER BY count DESC
            """, (f"-{hours} hours",)).fetchall()

        return {
            'period_hours': hours,
```

```
            'events': [dict(row) for row in rows]
        }

    def format_report_text(self, report: dict) -> str:
        """리포트를 읽기 쉬운 텍스트로 포맷"""
        lines = [
            f"═══ 일일 트레이딩 리포트: {report['date']} ═══",
            f"총 주문 수: {report['total_orders']}",
            ""
        ]

        for strategy, stats in report.get('strategies', {}).items():
            lines.extend([
                f"▶ {strategy}",
                f"  주문: {stats['orders']} | 체결률: {stats['fill_rate']:.1%} | 거부률: {stats
                f"  거래대금: {stats['total_volume']:,.0f} | 수수료: {stats['total_commission'
                f"  슬리피지: {stats['slippage']['mean_bps']:.1f}bps (max: {stats['slippage']
                f"  레이턴시: {stats['latency']['mean_us']:.0f}μs (P99: {stats['latency']['p9
                ""
            ])

        return '\n'.join(lines)
```

4.3 이상 패턴 자동 감지

```
class LogPatternDetector:
    """로그에서 이상 패턴을 자동 감지"""

    def __init__(self):
        self._patterns = []
        self._register_default_patterns()

    def _register_default_patterns(self):
        """기본 이상 패턴 등록"""
        self._patterns = [
            self._detect_rapid_orders,
            self._detect_repeated_rejections,
            self._detect_position_discrepancy,
            self._detect_unusual_slippage,
            self._detect_latency_degradation,
        ]

    def analyze(self, orders: List[dict]) -> List[dict]:
        """모든 패턴에 대해 분석 실행"""
        findings = []
        for pattern_func in self._patterns:
            result = pattern_func(orders)
            if result:
                findings.extend(result if isinstance(result, list) else [result])
        return findings
```

```
    def _detect_rapid_orders(self, orders: List[dict]) -> List[dict]:
        """비정상적으로 빠른 연속 주문 감지"""
        findings = []
        strategy_orders = defaultdict(list)

        for order in orders:
            strategy_orders[order.get('strategy', '')].append(order)

        for strategy, strat_orders in strategy_orders.items():
            sorted_orders = sorted(strat_orders, key=lambda x: x.get('timestamp', ''))

            for i in range(1, len(sorted_orders)):
                t1 = datetime.fromisoformat(sorted_orders[i-1].get('timestamp', '').rstrip('
                t2 = datetime.fromisoformat(sorted_orders[i].get('timestamp', '').rstrip('Z'
                gap_ms = (t2 - t1).total_seconds() * 1000

                if gap_ms < 10:  # 10ms 미만 간격
                    findings.append({
                        'pattern': 'rapid_orders',
                        'severity': 'warning',
                        'strategy': strategy,
                        'gap_ms': gap_ms,
                        'message': f'{strategy}: 연속 주문 간격 {gap_ms:.1f}ms — 루프 버그 가능성'
                    })

        return findings

    def _detect_repeated_rejections(self, orders: List[dict]) -> List[dict]:
        """같은 원인의 반복 거부 감지"""
        findings = []
        rejection_reasons = Counter()

        for order in orders:
            if order.get('status') == 'rejected':
                reason = order.get('error_message', 'unknown')
                rejection_reasons[reason] += 1

        for reason, count in rejection_reasons.items():
            if count >= 5:
                findings.append({
                    'pattern': 'repeated_rejections',
                    'severity': 'error',
                    'reason': reason,
                    'count': count,
                    'message': f'반복 거부 {count}회: {reason}'
                })

        return findings

    def _detect_position_discrepancy(self, orders: List[dict]) -> List[dict]:
        """포지션 불일치 감지 (로그 기반)"""
        findings = []
        positions = defaultdict(float)
```

```
        for order in sorted(orders, key=lambda x: x.get('timestamp', '')):
            if order.get('status') != 'filled':
                continue

            symbol = order.get('symbol', '')
            side = order.get('side', '')
            qty = order.get('fill_quantity', 0)

            if side == 'BUY':
                positions[symbol] += qty
            else:
                positions[symbol] -= qty

        for symbol, net_pos in positions.items():
            if abs(net_pos) > 0 and abs(net_pos) < 0.0001:
                findings.append({
                    'pattern': 'position_dust',
                    'severity': 'info',
                    'symbol': symbol,
                    'residual': net_pos,
                    'message': f'{symbol}: 잔여 포지션 {net_pos} — 소수점 정밀도 이슈'
                })

        return findings

    def _detect_unusual_slippage(self, orders: List[dict]) -> List[dict]:
        """비정상 슬리피지 감지"""
        findings = []
        slippages = [
            (o.get('strategy', ''), o.get('symbol', ''), o.get('slippage_bps', 0))
            for o in orders
            if o.get('slippage_bps') and o.get('status') == 'filled'
        ]

        if len(slippages) < 10:
            return findings

        values = [s[2] for s in slippages]
        mean = np.mean(values)
        std = np.std(values)

        for strategy, symbol, slip in slippages:
            if std > 0 and abs(slip - mean) > 3 * std:
                findings.append({
                    'pattern': 'unusual_slippage',
                    'severity': 'warning',
                    'strategy': strategy,
                    'symbol': symbol,
                    'slippage_bps': slip,
                    'mean_bps': round(mean, 2),
                    'z_score': round((slip - mean) / std, 2),
                    'message': f'{strategy}/{symbol}: 비정상 슬리피지 {slip:.1f}bps (평균: {mean
```

```
                })

        return findings

    def _detect_latency_degradation(self, orders: List[dict]) -> List[dict]:
        """점진적 레이턴시 악화 감지"""
        findings = []
        latencies = [
            o.get('latency_us', 0)
            for o in sorted(orders, key=lambda x: x.get('timestamp', ''))
            if o.get('latency_us')
        ]

        if len(latencies) < 50:
            return findings

        first_half = np.mean(latencies[:len(latencies)//2])
        second_half = np.mean(latencies[len(latencies)//2:])

        if second_half > first_half * 1.5:
            findings.append({
                'pattern': 'latency_degradation',
                'severity': 'warning',
                'first_half_us': round(first_half, 0),
                'second_half_us': round(second_half, 0),
                'increase_pct': round((second_half - first_half) / first_half * 100, 1),
                'message': f'레이턴시 악화: {first_half:.0f}μs → {second_half:.0f}μs (+{(secon
            })

        return findings
```

제5장: 로그 기반 전략 디버깅

5.1 트레이드 리플레이 시스템

과거 로그를 기반으로 특정 시점의 매매 결정을 재현합니다.

```
from typing import Generator

class TradeReplayer:
    """로그 기반 트레이드 리플레이 시스템"""

    def __init__(self, log_store: SQLiteLogStore, file_manager: LogFileManager):
        self.store = log_store
        self.file_manager = file_manager

    def replay_order(self, order_id: str) -> dict:
        """특정 주문의 전체 컨텍스트 재현"""
```

```
        orders = self.store.query_orders(limit=10000)
        target = None
        for o in orders:
            if o.get('order_id') == order_id:
                target = o
                break

        if not target:
            return {'error': f'주문 {order_id} 찾을 수 없음'}

        # 상관 ID로 관련 이벤트 추적
        corr_id = target.get('correlation_id', '')

        # 관련 이벤트 조회
        related_orders = [
            o for o in orders
            if o.get('correlation_id') == corr_id and corr_id
        ]

        # 시점 전후 시장 데이터 로그 조회
        timestamp = target.get('timestamp', '')
        market_context = self._get_market_context(timestamp, target.get('symbol', ''))

        return {
            'order': target,
            'related_events': related_orders,
            'market_context': market_context,
            'timeline': self._build_timeline(related_orders),
            'analysis': self._analyze_decision(target, market_context)
        }

    def _get_market_context(self, timestamp: str, symbol: str) -> dict:
        """주문 시점의 시장 상황 재현"""
        date = timestamp[:10]
        market_log_path = self.file_manager.get_log_path('market_data', datetime.fromisoform

        context = {
            'symbol': symbol,
            'timestamp': timestamp,
            'nearby_ticks': []
        }

        if not market_log_path.exists():
            return context

        # 주문 시점 전후 5초 데이터 수집
        target_time = datetime.fromisoformat(timestamp.rstrip('Z'))

        with open(market_log_path, 'r') as f:
            for line in f:
                try:
                    entry = json.loads(line)
                    entry_time = datetime.fromisoformat(entry.get('timestamp', '').rstrip('Z
```

```
                    diff = abs((entry_time - target_time).total_seconds())

                    if diff <= 5 and entry.get('data', {}).get('symbol') == symbol:
                        context['nearby_ticks'].append(entry)
                except (json.JSONDecodeError, ValueError):
                    continue

        return context

    def _build_timeline(self, events: List[dict]) -> List[dict]:
        """이벤트 타임라인 구성"""
        timeline = []
        for event in sorted(events, key=lambda x: x.get('timestamp', '')):
            timeline.append({
                'time': event.get('timestamp', ''),
                'status': event.get('status', ''),
                'details': f"{event.get('side', '')} {event.get('quantity', '')} @ {event.ge
                'latency_us': event.get('latency_us')
            })
        return timeline

    def _analyze_decision(self, order: dict, market_context: dict) -> dict:
        """매매 결정 분석"""
        analysis = {
            'order_type': order.get('order_type', ''),
            'execution_quality': 'unknown'
        }

        # 체결 품질 평가
        if order.get('status') == 'filled' and market_context.get('nearby_ticks'):
            prices = [
                t['data'].get('price', 0)
                for t in market_context['nearby_ticks']
                if t.get('data', {}).get('price')
            ]

            if prices:
                mid_price = np.mean(prices)
                fill_price = order.get('fill_price', 0)

                if fill_price and mid_price:
                    improvement = (mid_price - fill_price) / mid_price * 10000
                    if order.get('side') == 'SELL':
                        improvement = -improvement

                    analysis['mid_price_at_time'] = round(mid_price, 2)
                    analysis['fill_price'] = fill_price
                    analysis['price_improvement_bps'] = round(improvement, 2)
                    analysis['execution_quality'] = (
                        'good' if improvement > 0
                        else 'fair' if improvement > -5
                        else 'poor'
                    )
```

```
        return analysis


class StrategyDebugger:
    """전략 로직 디버깅 도구"""

    def __init__(self, log_store: SQLiteLogStore):
        self.store = log_store

    def find_missed_opportunities(
        self,
        strategy: str,
        date: str,
        signal_threshold: float = 0.02
    ) -> List[dict]:
        """놓친 매매 기회 분석"""
        # 신호가 발생했지만 주문이 없었던 경우
        # signal 로그와 order 로그를 대조

        missed = []
        # 실제 구현은 signal 로그 파일에서 분석
        return missed

    def analyze_losing_trades(
        self,
        strategy: str,
        date: str
    ) -> dict:
        """손실 거래 패턴 분석"""
        orders = self.store.query_orders(
            strategy=strategy,
            start_time=f"{date}T00:00:00",
            end_time=f"{date}T23:59:59",
            status='filled',
            limit=10000
        )

        # 매수-매도 쌍으로 그룹화하여 PnL 계산
        trades = self._pair_trades(orders)

        losers = [t for t in trades if t['pnl'] < 0]
        winners = [t for t in trades if t['pnl'] >= 0]

        analysis = {
            'total_trades': len(trades),
            'winners': len(winners),
            'losers': len(losers),
            'win_rate': len(winners) / max(len(trades), 1),
        }

        if losers:
            loss_amounts = [t['pnl'] for t in losers]
```

```
            analysis['avg_loss'] = round(np.mean(loss_amounts), 2)
            analysis['max_loss'] = round(min(loss_amounts), 2)
            analysis['loss_patterns'] = self._identify_loss_patterns(losers)

        return analysis

    def _pair_trades(self, orders: List[dict]) -> List[dict]:
        """매수-매도 주문을 쌍으로 매칭"""
        trades = []
        open_positions = defaultdict(list)

        for order in sorted(orders, key=lambda x: x.get('timestamp', '')):
            symbol = order.get('symbol', '')
            side = order.get('side', '')
            price = order.get('fill_price', 0)
            qty = order.get('fill_quantity', 0)

            if side == 'BUY':
                open_positions[symbol].append({'price': price, 'quantity': qty, 'time': orde
            elif side == 'SELL' and open_positions[symbol]:
                entry = open_positions[symbol].pop(0)
                pnl = (price - entry['price']) * qty
                trades.append({
                    'symbol': symbol,
                    'entry_price': entry['price'],
                    'exit_price': price,
                    'quantity': qty,
                    'pnl': round(pnl, 2),
                    'entry_time': entry['time'],
                    'exit_time': order['timestamp'],
                    'hold_duration': order['timestamp']  # 간략화
                })

        return trades

    def _identify_loss_patterns(self, losers: List[dict]) -> List[str]:
        """손실 거래의 공통 패턴 식별"""
        patterns = []

        # 패턴 1: 빠른 손절 (보유 시간 < 1분)
        quick_stops = [t for t in losers if t.get('hold_duration_sec', float('inf')) < 60]
        if len(quick_stops) > len(losers) * 0.3:
            patterns.append("빠른 손절 빈번 — 진입 타이밍 재검토 필요")

        # 패턴 2: 큰 단일 손실
        if losers:
            loss_values = [abs(t['pnl']) for t in losers]
            if max(loss_values) > np.mean(loss_values) * 3:
                patterns.append("단일 대형 손실 — 손절 한도 점검 필요")

        # 패턴 3: 연속 손실
        consecutive = 0
        max_consecutive = 0
```

```
        for t in losers:
            consecutive += 1
            max_consecutive = max(max_consecutive, consecutive)

        if max_consecutive >= 5:
            patterns.append(f"연속 {max_consecutive}회 손실 — 시장 레짐 변화 가능성")

        return patterns
```

제6장: 로그 보안과 규정 준수

6.1 민감 정보 마스킹

로그에 포함될 수 있는 민감 정보를 자동으로 마스킹합니다.

```
import re
import hashlib

class LogSanitizer:
    """로그 민감 정보 마스킹"""

    PATTERNS = [
        # API 키
        (r'(?i)(api[_-]?key|apikey)["\s:=]+["\']?([a-zA-Z0-9]{16,})', r'\1=***MASKED***'),
        # 시크릿
        (r'(?i)(secret|password|token)["\s:=]+["\']?([a-zA-Z0-9+/=]{8,})', r'\1=***MASKED***
        # IP 주소 (선택적)
        (r'\b(\d{1,3}\.\d{1,3}\.\d{1,3})\.\d{1,3}\b', r'\1.***'),
        # 이메일
        (r'[\w.-]+@[\w.-]+\.\w+', '***@***.***'),
    ]

    def __init__(self, additional_patterns: Optional[List[tuple]] = None):
        self.patterns = self.PATTERNS.copy()
        if additional_patterns:
            self.patterns.extend(additional_patterns)

    def sanitize(self, text: str) -> str:
        """민감 정보 마스킹 적용"""
        result = text
        for pattern, replacement in self.patterns:
            result = re.sub(pattern, replacement, result)
        return result

    def sanitize_dict(self, data: dict) -> dict:
        """딕셔너리의 모든 문자열 값에 마스킹 적용"""
        sanitized = {}
        for key, value in data.items():
```

```
            if isinstance(value, str):
                # 키 이름이 민감한 경우
                if any(s in key.lower() for s in ['key', 'secret', 'password', 'token']):
                    sanitized[key] = '***MASKED***'
                else:
                    sanitized[key] = self.sanitize(value)
            elif isinstance(value, dict):
                sanitized[key] = self.sanitize_dict(value)
            else:
                sanitized[key] = value
        return sanitized


class AuditLogger:
    """규정 준수를 위한 감사 로거"""

    def __init__(self, log_store: SQLiteLogStore, sanitizer: LogSanitizer):
        self.store = log_store
        self.sanitizer = sanitizer

    def log_trade_decision(
        self,
        strategy: str,
        symbol: str,
        decision: str,
        rationale: dict,
        market_state: dict,
        timestamp: Optional[str] = None
    ):
        """매매 결정에 대한 감사 로그"""
        entry = {
            'timestamp': timestamp or datetime.utcnow().isoformat() + 'Z',
            'type': 'trade_decision',
            'strategy': strategy,
            'symbol': symbol,
            'decision': decision,
            'rationale': self.sanitizer.sanitize_dict(rationale),
            'market_snapshot': {
                'bid': market_state.get('bid'),
                'ask': market_state.get('ask'),
                'volume_24h': market_state.get('volume_24h'),
            },
            'hash': ''  # 무결성 해시
        }

        # 무결성 해시 생성
        content = json.dumps(entry, sort_keys=True)
        entry['hash'] = hashlib.sha256(content.encode()).hexdigest()

        self.store.insert_order_log(
            timestamp=entry['timestamp'],
            order_id='',
            strategy=strategy,
```

```
            symbol=symbol,
            side=decision,
            order_type='audit',
            quantity=0,
            status='audit',
            raw_data=json.dumps(entry, ensure_ascii=False)
        )

    def verify_log_integrity(self, date: str) -> dict:
        """로그 무결성 검증"""
        orders = self.store.query_orders(
            start_time=f"{date}T00:00:00",
            end_time=f"{date}T23:59:59",
            limit=100000
        )

        total = len(orders)
        verified = 0
        tampered = 0

        for order in orders:
            raw = order.get('raw_data')
            if not raw:
                continue

            try:
                data = json.loads(raw)
                stored_hash = data.pop('hash', '')
                computed = hashlib.sha256(
                    json.dumps(data, sort_keys=True).encode()
                ).hexdigest()

                if computed == stored_hash:
                    verified += 1
                else:
                    tampered += 1
            except (json.JSONDecodeError, KeyError):
                continue

        return {
            'date': date,
            'total_records': total,
            'verified': verified,
            'tampered': tampered,
            'integrity': 'OK' if tampered == 0 else 'COMPROMISED'
        }
```

6.2 로그 보존 정책

```
class LogRetentionPolicy:
    """로그 보존 정책 관리"""
```

```
    DEFAULT_POLICIES = {
        'market_data': {'hot_days': 3, 'warm_days': 30, 'cold_days': 90},
        'order': {'hot_days': 30, 'warm_days': 180, 'cold_days': 365 * 3},
        'fill': {'hot_days': 30, 'warm_days': 180, 'cold_days': 365 * 3},
        'signal': {'hot_days': 7, 'warm_days': 60, 'cold_days': 180},
        'risk': {'hot_days': 90, 'warm_days': 365, 'cold_days': 365 * 5},
        'audit': {'hot_days': 365, 'warm_days': 365 * 3, 'cold_days': 365 * 7},
        'performance': {'hot_days': 30, 'warm_days': 90, 'cold_days': 365},
        'system': {'hot_days': 7, 'warm_days': 30, 'cold_days': 90},
        'error': {'hot_days': 30, 'warm_days': 90, 'cold_days': 365},
    }

    def __init__(self, file_manager: LogFileManager):
        self.file_manager = file_manager
        self.policies = self.DEFAULT_POLICIES.copy()

    def enforce(self):
        """보존 정책 강제 적용"""
        for category, policy in self.policies.items():
            # Hot → Warm: 압축
            self.file_manager.compress_old_logs(
                days_threshold=policy['hot_days']
            )

            # Cold 이후: 삭제 (또는 아카이브)
            self.file_manager.delete_old_logs(
                days_threshold=policy['cold_days']
            )

        print("로그 보존 정책 적용 완료")
        print(f"디스크 사용량: {self.file_manager.get_disk_usage()}")
```

제7장: 실전 로그 시스템 구축과 운영

7.1 통합 로그 시스템 구현

지금까지의 모든 컴포넌트를 통합한 실전 시스템입니다.

```
import asyncio
import yaml
from pathlib import Path

class TradingLogSystem:
    """통합 트레이딩 로그 시스템"""

    def __init__(self, config_path: str = "config/logging.yaml"):
        self.config = self._load_config(config_path)
```

```
        # 핵심 컴포넌트 초기화
        self.logger = TradingLogger(
            name=self.config.get('service_name', 'trading-bot'),
            log_dir=self.config.get('log_dir', 'logs'),
            console_level=getattr(
                logging, self.config.get('console_level', 'INFO')
            )
        )
        self.file_manager = LogFileManager(self.config.get('log_dir', 'logs'))
        self.db_store = SQLiteLogStore(
            self.config.get('db_path', 'logs/trading_logs.db')
        )
        self.sanitizer = LogSanitizer()
        self.order_logger = OrderLifecycleLogger(self.logger)
        self.analyzer = RealTimeLogAnalyzer()
        self.pattern_detector = LogPatternDetector()
        self.report_generator = TradingReportGenerator(self.db_store)
        self.retention = LogRetentionPolicy(self.file_manager)
        self.correlation_tracker = CorrelationTracker()

        # 비동기 버퍼
        self.async_buffer = AsyncLogBuffer(
            flush_interval=self.config.get('flush_interval', 1.0),
            writers=[self._write_to_db, self.file_manager.write_batch]
        )

        # 알림 규칙 등록
        self.analyzer.add_alert_rule(error_spike_rule)
        self.analyzer.add_alert_rule(rejection_rate_rule)

    def _load_config(self, path: str) -> dict:
        try:
            with open(path) as f:
                return yaml.safe_load(f)
        except FileNotFoundError:
            return {}

    async def start(self):
        """로그 시스템 시작"""
        asyncio.create_task(self.async_buffer.start())
        print("트레이딩 로그 시스템 시작")

    async def stop(self):
        """로그 시스템 안전 종료"""
        self.async_buffer.stop()
        await asyncio.sleep(2)  # 잔여 로그 플러시 대기
        print(f"로그 시스템 종료 — 통계: {self.async_buffer.stats}")

    def _write_to_db(self, batch: List[dict]):
        """DB에 배치 기록"""
        for entry in batch:
            category = entry.get('category', '')
```

```
            if category in ('order', 'fill'):
                self.db_store.insert_order_log(**entry.get('data', {}))

    async def daily_maintenance(self):
        """일일 유지보수 루틴"""
        # 1. 로그 보존 정책 적용
        self.retention.enforce()

        # 2. 일일 리포트 생성
        yesterday = (datetime.utcnow() - timedelta(days=1)).strftime('%Y-%m-%d')
        report = self.report_generator.daily_report(yesterday)
        report_text = self.report_generator.format_report_text(report)

        # 3. 이상 패턴 분석
        orders = self.db_store.query_orders(
            start_time=f"{yesterday}T00:00:00",
            end_time=f"{yesterday}T23:59:59",
            limit=100000
        )
        findings = self.pattern_detector.analyze(orders)

        # 4. 디스크 사용량 확인
        disk_usage = self.file_manager.get_disk_usage()

        maintenance_report = {
            'date': yesterday,
            'trading_report': report,
            'anomaly_findings': findings,
            'disk_usage': disk_usage,
            'buffer_stats': self.async_buffer.stats
        }

        # 리포트 저장
        report_path = Path('logs') / 'reports' / f"{yesterday}_daily.json"
        report_path.parent.mkdir(parents=True, exist_ok=True)
        with open(report_path, 'w', encoding='utf-8') as f:
            json.dump(maintenance_report, f, ensure_ascii=False, indent=2, default=str)

        return maintenance_report
```

7.2 설정 파일 예시

```
# config/logging.yaml

service_name: "trading-bot"
log_dir: "logs"
db_path: "logs/trading_logs.db"
console_level: "INFO"
file_level: "DEBUG"
flush_interval: 1.0
max_buffer_size: 10000
```

```yaml
# 파일 로테이션
rotation:
  max_bytes: 104857600  # 100MB
  backup_count: 30

# 카테고리별 설정
categories:
  market_data:
    level: DEBUG
    retention_days: 3
  order:
    level: INFO
    retention_days: 365
  fill:
    level: INFO
    retention_days: 365
  signal:
    level: DEBUG
    retention_days: 30
  risk:
    level: WARNING
    retention_days: 365
  audit:
    level: INFO
    retention_days: 2555  # 7년

# 알림 설정
alerts:
  error_spike_threshold: 50
  rejection_rate_threshold: 0.10
  latency_spike_us: 100000

# Elasticsearch (선택)
elasticsearch:
  enabled: false
  hosts: ["http://localhost:9200"]
  index_prefix: "trading-logs"
```

7.3 로그 시스템 운영 체크리스트

일일 점검: - 에러 로그 파일 크기 확인 — 급증 시 조사 - 디스크 사용량 확인 (전체 용량의 80% 미만 유지) - 버퍼 드롭률 확인 (0.1% 미만 유지) - 일일 트레이딩 리포트 검토 - 이상 패턴 감지 결과 확인

주간 점검: - 로그 압축 및 정리 상태 확인 - SQLite DB 크기 및 쿼리 성능 확인 - 로그 레벨 적절성 검토 (너무 많거나 적은 로그) - 알림 규칙 유효성 검토

월간 점검: - 로그 보존 정책 준수 확인 - 백업 상태 확인 - 로그 분석 도구 성능 검토 - 새로운 이상 패턴 규칙 추가 검토

7.4 자주 발생하는 문제와 해결 방법

문제 1: 로그로 인한 디스크 부족 - 원인: market_data 로그가 과도하게 쌓임 - 해결: 틱 데이터는 샘플링하여 로그, 보존 기간 축소

문제 2: 로그 기록이 매매 성능에 영향 - 원인: 동기식 파일 쓰기가 이벤트 루프 블로킹 - 해결: AsyncLogBuffer 적용, 비동기 I/O 사용

문제 3: 로그에서 원하는 정보를 찾기 어려움 - 원인: 비구조화 텍스트 로그 사용 - 해결: JSON 구조화 로그 + SQLite 인덱싱

문제 4: 상관 이벤트 추적 불가 - 원인: 주문 ID만으로 추적, 전략 → 주문 → 체결 연결 불가 - 해결: Correlation ID 도입으로 이벤트 체인 구성

문제 5: 로그 유실 - 원인: 프로세스 강제 종료 시 버퍼 미플러시 - 해결: 시그널 핸들러에서 flush 호출, WAL 모드 DB 사용

이 가이드의 로그 시스템을 구축하면 자동매매 시스템의 모든 동작을 투명하게 추적하고, 장애 원인을 신속히 파악하며, 전략 개선을 위한 데이터 기반 인사이트를 얻을 수 있습니다. 로그는 비용이 아닌 투자입니다. 체계적인 로그 시스템이야말로 안정적인 자동매매 운영의 기반입니다.