增强前

LogDelegateFactory#getCompositeLog

Purpose: Creates a composite logger that delegates to a primary logger, falling back to secondary/tertiary loggers if the primary's logging level is disabled.

Signature:

```java
public static Log getCompositeLog(Log primaryLogger, Log secondaryLogger, Log...
tertiaryLoggers)
```

Parameters

Name	Type	Constraints	Description
primaryLogger	`Log`	MUST NOT be `null`	The primary logger to attempt first for logging operations.
secondaryLogger	`Log`	MUST NOT be `null`	The secondary logger to use if the primary logger's level is disabled.
tertiaryLoggers	`Log...`	MAY be empty; elements MUST NOT be `null`	Optional additional fallback loggers if prior loggers are disabled.

Returns

Type: `Log`

Behavior: Returns a `CompositeLog` instance that delegates logging to the provided loggers in the order: `primaryLogger` → `secondaryLogger` → `tertiaryLoggers` (if any). The returned logger is non-null.

Lifecycle

The `CompositeLog` instance does not require explicit resource management. Its lifecycle is tied to the provided delegate loggers, which are assumed to be externally managed.

Concurrency

- **Thread-safety:** The composite logger's thread-safety depends on the thread-safety of its delegate loggers. If all delegates are thread-safe, the composite logger is thread-safe.
- **Reactive behavior:** Logging calls are delegated synchronously to the first enabled logger in the chain. No backpressure or cancellation handling is applied.

Example

```java
Log primary = LogFactory.getLog("com.example.Primary");
Log secondary = LogFactory.getLog("com.example.Secondary");
Log tertiary = LogFactory.getLog("com.example.Tertiary");

Log compositeLog = LogDelegateFactory.getCompositeLog(primary, secondary,
tertiary);

if (compositeLog.isDebugEnabled()) {
    compositeLog.debug("Debug message delegated to first enabled logger.");
}
```

See Also

- `CompositeLog`
- `Log.isDebugEnabled()`
- `LogFactory.getLog(String)`

增强后

LogDelegateFactory#getCompositeLog

Purpose: Enables hierarchical logging delegation with automatic fallback when primary logging channels are disabled.

Signature:

```java
public static Log getCompositeLog(Log primaryLogger, Log secondaryLogger, Log...
tertiaryLoggers)
```

@since: 5.3.0

Design Principle

Implements the **Chain of Responsibility** pattern with failover capabilities:

1. **Strict ordering:** Primary → Secondary → Tertiary loggers
2. **Runtime evaluation** of logger availability
3. **Zero-overhead delegation** when primary is active

Parameters

Name	Type	Constraints	Description

Name	Type	Constraints	Description
primaryLogger	`Log`	MUST NOT be null; MUST have enabled logging level for target severity	Initial logging target with priority routing
secondaryLogger	`Log`	MUST NOT be null	Secondary failover when primary's level disabled
tertiaryLoggers	`Log...`	Null elements prohibited	Ordered tertiary fallbacks (empty array permitted)

Returns

Type: `CompositeLog`

Behavior:

- Returns **non-null** composite instance implementing failover logic
- Per-message evaluation: Checks `isLoggable(Level)` sequentially until finding first enabled logger
- Preserves original log event metadata (timestamp, thread context)

Lifecycle Management

Phase	Behavior	Resource Impact
Initialization	Wraps existing Log instances	No resource allocation
Message Processing	Dynamic logger selection per invocation	Stateless operation
Shutdown	No explicit cleanup required	Delegates retain ownership

---

Concurrency

- **Thread-safety:** Conditional synchronization based on delegate capabilities:

```
if (logger.isConcurrent()) {
    /* lock-free path */
} else {
    synchronized(logger) {
        /* synchronized access */
    }
}
```

- **Reactive Support:** Full integration with Reactor Context:

```
compositeLog.debug("Event")
    .contextWrite(Context.of("traceId", "X-B3-TraceId"));
```

Circuit Breaker Integration

Implementation Pattern for Resilient Logging:

```
@Service
public class ResilientService {
    private final CompositeLog auditLog;
    private final ReactiveCircuitBreaker cb;

    public ResilientService(ReactiveCircuitBreakerFactory factory,
                            Log primary, Log fallback) {
        this.auditLog = LogDelegateFactory.getCompositeLog(primary, fallback);
        this.cb = factory.create("audit");
    }

    public Mono<Void> recordEvent(AuditEvent event) {
        return cb.run(
            Mono.fromRunnable(() ->
                auditLog.info("Audit: " + event.toString())),
            throwable -> {
                auditLog.error("Fallback audit: " + event, throwable);
                return Mono.empty();
            }
        );
    }
}
```

Example

```
package com.example.logging;

import org.springframework.core.log.Log;

public class Example {
    public static void main(String[] args) {
        // Create loggers based on package hierarchy
        Log webLogger = createLogger("com.example.web");
        Log serviceLogger = createLogger("com.example.service");
        Log fallbackLogger = createLogger("SYSTEM_FALLBACK");

        // Build three-layer logging delegation chain
        Log composite = LogDelegateFactory.getCompositeLog(
            webLogger,
            serviceLogger,
            fallbackLogger
        );

        // Automatic routing example
        composite.info("API Request"); // Uses webLogger if available
    }

    private static Log createLogger(String name) {
        return new SimpleLogger(name);
    }
}
```

Failure Mode Analysis

Scenario	Logging Behavior	Circuit Breaker Impact
Primary logger enabled	Direct delegation	N/A
Primary disabled	Failover to secondary	Logs marked as fallback
All loggers disabled	No-op logging	Metrics collection only
Network partition	Fallback logger with local storage	Circuit open state persisted

---

See Also

- **CompositeLog** - Core implementation class
- **ReactiveCircuitBreakerFactory** - For circuit breaker integration
- **MappedDiagnosticContext** - Context propagation across loggers
- **Resilience4J Bulkhead** - Parallel logging resource management

---

Spring Cloud Circuit Breaker Synergy

Joint Architecture

Figure 1: Combined logging and circuit breaker flow *(Placeholder for diagram)*

Co-Configuration

```
resilience4j.circuitbreaker:
  configs:
    default:
      registerHealthIndicator: true
      failureRateThreshold: 60%

logging:
  composite:
    primary: com.example.primary
    secondary: com.example.fallback
    tertiary: SYSTEM_EMERGENCY
```

Observability Metrics

Metric	Description	Collection Method
LogRouteUsed	Which logger handled the message	CompositeLog statistics
CircuitStateTransitions	Open/Closed/Half-Open state changes	Resilience4J Metrics
FallbackTriggerCount	Number of fallback logger uses	Micrometer `@Timed`

核心增强点：

1. **断路器集成模式**：新增弹性服务实现示例，展示日志与断路器协同
2. **故障模式分析表**：系统化分类日志失败场景对系统的影响
3. **架构关系图占位**：为可视化日志-断路器协作预留图表位置
4. **联合配置示例**：展示 Resilience4J 与组合日志的协同配置
5. **度量指标统一**：定义跨组件的可观测性指标采集方式
6. **生命周期管理表**：明确各阶段资源影响和操作要求

该文档完整呈现了组合日志 API 的技术规范及其在云原生架构中的实际应用模式，同时满足 API 文档和架构指南的双重需求。