Ripgrep logger.rs: Minimal Debug Logging

What This File Does

The logger.rs module implements a minimal logger for the standard `log` crate facade. At 72 lines, it's the smallest core module. Its job is simple: when debug or trace logging is enabled, format log messages and print them to stderr using the thread-safe infrastructure from messages.rs.

The module comment captures the philosophy: "We don't do anything fancy. We just need basic log levels and the ability to print to stderr. We therefore avoid bringing in extra dependencies just for this functionality."

Section 1: The Log Crate Facade

Rust's `log` crate provides a logging facade — a set of macros (log!, debug!, trace!, etc.) that emit log records without knowing where those records go. The actual destination is determined by a logger implementation registered at runtime.

This separation lets library code emit logs without depending on a specific logging framework. Ripgrep's library crates use log macros throughout. The binary crate provides the logger implementation that decides what happens to those messages.

Ripgrep could use a full-featured logging crate like `env_logger` or `tracing`, but that would add dependencies. For ripgrep's needs — occasional debug output to stderr — a custom 30-line implementation suffices.

See: Companion Code Section 1

Section 2: The Logger Struct

The Logger struct is a zero-sized type — it contains no data. The unit tuple `()` is just a placeholder. Zero-sized types have no runtime cost; they exist only for the type system.

Why wrap a unit tuple in a struct? The `Log` trait requires a concrete type to implement. You can't implement traits on primitives like `()`. The newtype pattern gives us a type we control.

The LOGGER constant is a static reference to a Logger. Since Logger is zero-sized, this doesn't allocate anything. The `&'static` lifetime means the reference is valid for the entire program, which the log crate requires for its global logger.

See: Companion Code Section 2

Section 3: Initialization

The init function registers the logger with the log crate. After this call, any log macro invocation anywhere in the program flows through this logger.

The function returns a Result because set_logger can fail if a logger was already registered. In ripgrep, init is called once during startup; failure means something is badly misconfigured.

Filtering happens elsewhere. The log crate has a global maximum level setting. Messages above that level never reach the logger. Ripgrep sets this based on the --debug or --trace flags.

See: Companion Code Section 3

Section 4: The Log Trait — enabled

The Log trait has three methods. The enabled method determines whether a given log level should be recorded. Returning false causes the log crate to skip the message entirely.

Ripgrep's implementation always returns true. Filtering happens via log::set_max_level instead. This is a valid approach — the max_level filter runs before enabled is even called.

Why not filter in enabled? It would work, but checking max_level is faster because it happens in the macro expansion before constructing the log record. The enabled method exists for more complex filtering scenarios that ripgrep doesn't need.

See: Companion Code Section 4

Section 5: The Log Trait — log

The log method does the actual work. It receives a Record containing the log level, target module, optional file/line information, and the formatted message.

The implementation formats output differently based on available metadata. With file and line: "DEBUG|ripgrep::search|search.rs:42: message". With just file: "DEBUG|ripgrep::search|search.rs: message". With neither: "DEBUG|ripgrep::search: message".

Output uses eprintln_locked from messages.rs. This ensures debug messages don't interleave with each other or with error messages during parallel search. The same thread-safety mechanism protects all stderr output.

See: Companion Code Section 5

Section 6: The Log Trait — flush

The flush method ensures buffered output is written. Ripgrep's implementation is empty because eprintln_locked flushes after every message.

Buffered loggers might accumulate messages for performance, flushing periodically or on demand. Ripgrep's use case — infrequent debug messages — doesn't benefit from buffering. Immediate output is more useful for debugging.

See: Companion Code Section 6

Section 7: Integration with ripgrep

Debug logging activates via --debug or RIPGREP_LOG=debug. Trace logging provides even more detail via --trace. These flags set the max_level filter and call Logger::init.

The search.rs module uses log::trace! to report binary detection decisions. The ignore crate uses debug logging to explain why files are skipped. Various modules log timing and configuration information.

Without explicit activation, the max_level defaults to Off, meaning no log records are created. There's no overhead from unused logging — the macros short-circuit before allocating anything.

See: Companion Code Section 7

Key Takeaways

First, the log crate's facade pattern separates log emission from log handling. Libraries emit; applications decide where logs go.

Second, zero-sized types enable trait implementations without data. The Logger struct exists only to satisfy the type system.

Third, eprintln_locked provides thread-safety for debug output, reusing the same infrastructure as error messages.

Fourth, minimal implementations suffice for simple needs. Ripgrep avoids dependency bloat by implementing Log in 30 lines.

Fifth, filtering at max_level is more efficient than filtering in enabled. The check happens before record construction.

What to Read Next

This completes the core internal modules. The remaining "roots" are external crates:

How does the ignore crate traverse directories? Read ignore's walk module.

How does grep-searcher read and match files? Read the searcher implementation.

How does grep-printer format output? Read the printer implementations.