Ripgrep search.rs: The Search Coordinator

What This File Does

The search.rs file defines the SearchWorker, which coordinates the three core components of a search operation: the matcher (regex engine), the searcher (file I/O strategy), and the printer (output formatting). It's surprisingly compact at around 450 lines because it delegates the actual work to library crates. Its job is routing and coordination.

The module doc comment captures this perfectly: "A search worker manages the high level interaction points between the matcher, the searcher, and the printer." This is the glue that connects configuration to execution.

Section 1: The Config Struct

The Config struct holds five fields that control search behavior at a high level. The preprocessor field optionally holds a path to an external command that transforms file contents before searching. The preprocessor_globs field determines which files should run through the preprocessor. The search_zip boolean enables automatic decompression of compressed files.

The two binary detection fields are particularly interesting. Binary_implicit controls detection for files discovered during directory traversal. Binary_explicit controls detection for files explicitly named by the user. This distinction matters because explicitly requested files should never be silently skipped, even if they appear binary.

Default values disable all these features. No preprocessor, no decompression, no binary detection. Each must be explicitly enabled through the builder.

See: Companion Code Section 1

Section 2: The Builder Pattern

SearchWorkerBuilder follows Rust's builder pattern. It starts with default configuration and provides methods to modify each setting before final construction.

The build method takes the three core components as arguments: a PatternMatcher, a Searcher, and a Printer. These come from HiArgs, which already configured them based on command-line flags. The builder adds the coordination layer.

One subtle detail: the DecompressionReaderBuilder is only constructed when search_zip is enabled. This lazy initialization avoids unnecessary work, particularly on Windows where resolving decompression binary paths can be expensive.

The async_stderr(true) calls on command builders are worth noting. When running external commands like preprocessors or decompressors, their stderr output gets forwarded asynchronously. This prevents a hung child process from blocking the search.

See: Companion Code Section 2

Section 3: The Builder Methods

Each builder method configures one aspect of search behavior.

The preprocessor method validates the binary path before storing it. The resolve_binary function handles platform differences in executable lookup. This validation happens at configuration time, not search time, so errors surface early.

The preprocessor_globs method sets which files should be preprocessed. When empty (the default), all files go through the preprocessor. When populated, only matching files get preprocessed. The Override type comes from the ignore crate and supports gitignore-style patterns.

Binary detection methods demonstrate the explicit versus implicit distinction. Implicit detection typically uses "quit" mode to skip binary files entirely. Explicit detection uses "convert" mode to replace NUL bytes with something displayable, or "none" to show raw bytes.

See: Companion Code Section 3

Section 4: SearchResult

The SearchResult struct captures the outcome of a single search operation. It's minimal: just whether matches were found and optional statistics.

The has_match field drives exit code determination. A search with matches returns exit code zero. A search without matches returns exit code one.

Statistics are optional because they're expensive to compute. Only when explicitly requested through flags like --stats does the printer actually collect them. The stats field contains aggregate information like match counts, line counts, and timing data.

Both fields are accessed through methods rather than direct field access. This encapsulation lets the implementation change without affecting callers.

See: Companion Code Section 4

Section 5: PatternMatcher Enum

The PatternMatcher enum abstracts over regex engine implementations. Currently two variants exist: RustRegex wrapping Rust's regex crate, and PCRE2 wrapping the PCRE2 library.

PCRE2 is conditionally compiled via the pcre2 feature flag. When disabled, only RustRegex is available. This keeps binary size small for users who don't need PCRE2's advanced features.

The enum enables runtime polymorphism without trait objects. Each search method matches on the enum and calls the appropriate implementation. This approach has two advantages over dynamic dispatch: the compiler can inline through the match, and there's no vtable indirection at each call.

The Clone and Debug derives work because both inner types implement those traits. Cloning a PatternMatcher is cheap because the compiled regex is internally reference-counted via Arc.

See: Companion Code Section 5

Section 6: Printer Enum

The Printer enum abstracts over output formats. Three variants cover the major use cases: Standard for grep-like line output, Summary for aggregate counts and file lists, and JSON for structured machine-readable output.

The type parameter W represents the underlying writer. In practice, this is either stdout or a BufferWriter for parallel output. The WriteColor bound ensures color escape sequences can be written.

The get_mut method provides mutable access to the underlying writer. This is essential for parallel search, where each thread writes to a private buffer that gets cleared between files.

Each printer variant wraps a type from the grep-printer crate. The Standard printer handles line-by-line output with context. The Summary printer handles counts and file-only modes. The JSON printer emits structured data conforming to a documented schema.

See: Companion Code Section 6

Section 7: The SearchWorker Struct

SearchWorker brings together everything needed to execute searches. It holds the configuration, command builders, the optional decompression builder, and the three core components: matcher, searcher, and printer.

The struct is generic over W, the writer type. This allows the same SearchWorker to write to stdout, files, or memory buffers.

The Clone derive is significant. In parallel search, each thread needs its own SearchWorker. Cloning creates independent instances that can operate concurrently. The expensive parts (compiled regex) are internally shared via Arc, so cloning is cheap.

The Debug derive aids troubleshooting. When something goes wrong, being able to print the SearchWorker's state helps identify misconfiguration.

See: Companion Code Section 7

Section 8: The Search Method

The search method is the entry point for searching a single haystack. It implements a decision tree that routes to the appropriate search strategy.

First, it selects binary detection based on whether the haystack is explicit or implicit. An explicit file was directly named by the user. An implicit file was discovered during directory traversal. This distinction ensures user-requested files are never silently skipped.

Then comes the routing logic. If the haystack is stdin, search_reader handles it directly. If the file should be preprocessed (based on preprocessor configuration and glob matching),

search_preprocessor transforms it first. If the file should be decompressed, search_decompress handles that. Otherwise, search_path reads the file directly.

The log::trace call provides debugging visibility. With the appropriate log level, you can see exactly which binary detection mode applies to each file.

See: Companion Code Section 8

Section 9: Helper Predicates

Two helper methods determine whether special processing applies to a given path.

The should_decompress method checks whether decompression is enabled and whether the path matches a known compressed format. The DecompressionReaderBuilder maintains a mapping from file extensions to decompression commands. If no command matches, the file gets searched directly.

The should_preprocess method has three-way logic. If no preprocessor is configured, return false. If preprocessor globs are empty, preprocess everything. If globs are specified, only preprocess files that match. The is_ignore() check handles negated globs properly.

These predicates are checked in order during search routing. Preprocessing takes priority over decompression because the user explicitly requested it.

See: Companion Code Section 9

Section 10: Preprocessor Search

The search_preprocessor method runs an external command and searches its output. This enables searching files that ripgrep can't read directly, like PDFs or proprietary formats.

The method constructs a Command with the file path as an argument. The file itself becomes stdin for the command. This two-path approach lets preprocessors either read the file directly or process stdin.

Error handling wraps the underlying errors with context about which command failed. This makes debugging much easier when a preprocessor misbehaves.

The close() call after searching is critical. It waits for the child process to exit and returns any errors. Without this, zombie processes could accumulate.

See: Companion Code Section 10

Section 11: Decompression Search

The search_decompress method handles compressed files transparently. It recognizes formats like gzip, bzip2, xz, and others based on file extension.

The implementation mirrors search_preprocessor but uses the DecompressionReaderBuilder instead of a user-specified command. The builder knows which decompression command to use for each format.

If the decomp_builder is None (meaning search_zip is disabled), this method falls back to search_path. This shouldn't happen in practice because should_decompress would have returned false, but the defensive coding ensures correctness.

See: Companion Code Section 11

Section 12: Direct File Search

The search_path method searches a file by path, allowing the searcher to use optimizations like memory mapping. This is the fast path for most searches.

The method matches on the PatternMatcher enum and delegates to a free function. This pattern avoids code duplication between RustRegex and PCRE2 variants while letting each use its own optimized implementation.

The conditional compilation attribute on PCRE2 ensures it only appears when that feature is enabled. Without this, the code wouldn't compile when PCRE2 is disabled.

See: Companion Code Section 12

Section 13: Reader-Based Search

The search_reader method searches arbitrary Read implementations. It's used for stdin, preprocessor output, and decompression output.

The doc comment explains why this is less preferred than search_path: "Searching via search_path provides more opportunities for optimizations (such as memory maps)." Memory-mapped search avoids copying file contents into userspace, which can significantly speed up large file searches.

The method has the same enum matching structure as search_path. Both delegate to free functions that handle the actual search mechanics.

See: Companion Code Section 13

Section 14: The Free Functions

Two free functions, search_path and search_reader, contain the actual search logic. They're generic over the Matcher trait, allowing any regex engine implementation.

The functions match on the Printer enum to create an appropriate sink. A sink is an object that receives match events and handles them appropriately. Each printer type produces its own sink type.

The sink_with_path method creates a sink that knows which file is being searched. This path appears in output and statistics.

After searching completes, the function extracts results from the sink: whether any matches occurred and optional statistics. The clone on stats is necessary because the sink owns the stats and we need to return them.

See: Companion Code Section 14

Key Takeaways

First, SearchWorker is pure coordination. It holds components created elsewhere and routes searches to the right handler.

Second, the explicit versus implicit distinction pervades the design. User-requested files get different treatment than discovered files.

Third, external commands integrate cleanly. Preprocessors and decompressors are just readers that wrap child processes.

Fourth, enum-based polymorphism avoids dynamic dispatch costs. Matching on variants can be inlined and optimized.

Fifth, the sink pattern decouples searching from printing. The searcher produces match events; the sink decides what to do with them.

What to Read Next

Understanding search.rs raises questions about the components it coordinates:

How does the grep-searcher crate actually find matches in files? Read the searcher implementation.

How does the Sink trait work? Read grep-printer's sink documentation.

How does memory mapping interact with the searcher? Read grep-searcher's mmap handling.

What does the DecompressionReaderBuilder support? Read grep-cli's decompression module.