Ripgrep haystack.rs: The Search Target Abstraction

What This File Does

The haystack.rs file defines what ripgrep actually searches. A "haystack" is the thing you're looking for a needle in — it could be a file on disk, stdin, or something discovered during directory traversal.

This is a thin abstraction layer, but an important one. It wraps the directory entry from the ignore crate and adds application-level logic: should this thing be searched? How should its path be displayed? The abstraction normalizes different input sources so the search code doesn't need to care whether it's searching a file the user named explicitly or one discovered recursively.

Section 1: The Builder Pattern

Haystack uses the builder pattern, though with minimal configuration. Currently there's only one setting: whether to strip the "./" prefix from paths.

This might seem like overkill for a single boolean, but the pattern leaves room for future configuration without changing the API. More importantly, it separates construction concerns from usage concerns — the builder knows how to create haystacks, and the haystack knows how to be searched.

The builder is cloneable and stored in HiArgs, then passed to every search function. This ensures consistent haystack construction throughout a search operation.

See: Companion Code Section 1

Section 2: Building from Results

The build_from_result method is the primary entry point. Directory traversal yields Results because listing a directory can fail — permission denied, symlink loops, and other filesystem errors.

When traversal yields an error, the builder logs it and returns None. The search continues with other files. This is the "non-fatal error" philosophy from main.rs in action — one bad file shouldn't stop the entire search.

When traversal yields a valid directory entry, the builder applies filtering logic to decide if it's searchable. Not everything the walker yields should be searched.

See: Companion Code Section 2

Section 3: The Filtering Decision

The build method contains the core filtering logic. It asks a series of questions about each directory entry.

First, was this entry explicitly provided by the user? If someone types "rg pattern somefile", they want somefile searched regardless of what it is. Explicit entries always pass the filter.

Second, is this a regular file? During directory traversal, the walker yields all kinds of things: files, directories, symlinks. Ripgrep only wants to search files. Directories are obviously not searchable. Symlinks are trickier — by the time we see them, they've either been followed (if --follow was set) or they haven't. If they haven't been followed, we skip them.

If the entry passes neither check, it gets logged at debug level and filtered out. The debug logging helps users understand why certain files aren't being searched.

See: Companion Code Section 3

Section 4: Explicit vs Implicit

The distinction between explicit and implicit entries is subtle but important. It affects both filtering and output formatting.

An explicit entry is something the user directly asked to search. This includes stdin and any path argument with depth zero — meaning it wasn't discovered by recursing into a directory, it was provided directly.

An implicit entry is something discovered during directory traversal. When you run "rg pattern ./src", the ./src directory is explicit, but ./src/main.rs is implicit — ripgrep found it by traversing.

Why does this matter? Explicit files bypass certain filters. If you explicitly name a binary file, ripgrep searches it (with appropriate binary handling). If ripgrep discovers a binary file during traversal, it might skip it entirely. The user's explicit intent overrides automatic filtering.

See: Companion Code Section 4

Section 5: Path Display

The path method handles how file paths appear in output. This seems trivial but involves a subtle UX decision.

When you run "rg pattern" with no arguments, ripgrep searches the current directory. Internally, this becomes "./". Without special handling, every match would show paths like "./src/main.rs".

The strip_dot_prefix option removes this leading "./". So matches appear as "src/main.rs" instead. This matches user expectations — they didn't type "./", so they don't want to see it.

But when the user explicitly provides "./" as an argument, the prefix stays. Their explicit choice is respected.

See: Companion Code Section 5

Section 6: File Type Detection

The is_dir and is_file methods determine what kind of filesystem object we're dealing with. This sounds simple but has edge cases around symlinks.

For directories, the check follows symlinks. If a symlink points to a directory, is_dir returns true. This matters because we never want to "search" a directory — that doesn't make sense.

For files, the check is stricter. Only actual files return true, not symlinks to files. This is because symlink handling has already happened during traversal. If symlinks should be followed, the walker followed them. What we see here is what we should search.

The file_type method from the directory entry can return None if metadata couldn't be read. Both methods handle this gracefully by returning false — if we can't determine what something is, we don't search it.

See: Companion Code Section 6

Section 7: The Wrapper Structure

The Haystack struct itself is minimal: a directory entry plus the strip_dot_prefix flag. It's a wrapper that adds behavior without adding much state.

This wrapper pattern is common in Rust. The underlying ignore::DirEntry has its own API, but we want to present a different API to the rest of ripgrep. The wrapper lets us do that while keeping the original entry accessible for its metadata.

The struct derives Clone and Debug. Clone is needed because haystacks might be sent across threads in parallel search. Debug is useful for logging and debugging.

See: Companion Code Section 7

Key Takeaways

First, abstraction enables uniform handling. The search code doesn't care if it's searching stdin or a file — both are haystacks with the same interface.

Second, explicit user intent matters. Files the user names directly are treated differently than files discovered automatically.

Third, small abstractions have value. This file is under 100 lines, but it cleanly separates "what to search" from "how to search it."

Fourth, error handling is graceful. Filesystem errors during traversal become log messages, not crashes.

Fifth, the builder pattern scales. Even with one configuration option, the pattern makes the API clean and extensible.

What to Read Next

Understanding haystack raises questions about its context:

How does the ignore crate's DirEntry work? Read the ignore crate documentation.

How does HaystackBuilder get configured? Review hiargs.rs, specifically the haystack_builder method.

What does SearchWorker do with a Haystack? Read search.rs.

How do haystacks flow through parallel search? Review the search_parallel function in main.rs.