Ignore Crate lib.rs: Foundation Types

What This File Does

The lib.rs file is the ignore crate's entry point. Beyond the standard module declarations and re-exports, it defines two fundamental types that pervade the entire crate: the `Error` enum for error handling and the `Match<T>` enum for representing filtering decisions.

These types establish patterns that every other module follows. Understanding them first makes the rest of the crate much clearer.

Section 1: Crate Overview

The module documentation captures the crate's purpose succinctly: "a fast recursive directory iterator that respects various filters such as globs, file types and `.gitignore` files."

Two usage patterns dominate. The simple `Walk::new("./")` provides an iterator that applies all filtering automatically. The advanced `WalkBuilder` enables fine-grained control over every filtering behavior.

The crate exposes three public modules beyond the walker: `gitignore` for parsing and matching gitignore patterns, `types` for file type definitions, and `overrides` for command-line glob handling. Internal modules (`dir`, `pathutil`, `default_types`) remain private implementation details.

See: Companion Code Section 1

Section 2: The Error Enum — Variant Overview

The Error enum handles every failure mode in the crate. Eight variants cover the cases:

`Partial` wraps a vector of errors. This handles situations where some operations succeed while others fail — like parsing a gitignore file where some lines are valid and others aren't.

`WithLineNumber`, `WithPath`, and `WithDepth` wrap inner errors with additional context. The wrapping is recursive — you can have an error with a path containing an error with a line number.

`Loop` occurs when following symlinks creates a cycle. The ancestor and child paths identify where the loop was detected.

`Io` wraps standard I/O errors. `Glob` wraps glob parsing failures. `UnrecognizedFileType` and `InvalidDefinition` handle file type configuration errors.

See: Companion Code Section 2

Section 3: Error Context Wrapping

The `With*` variants demonstrate a pattern for adding context to errors without losing the original. Rather than converting errors to strings early, you wrap them in increasingly specific context.

A glob parse error in a gitignore file becomes: `WithPath { path: ".gitignore", err: WithLineNumber { line: 5, err: Glob { ... } } }`. When displayed, this produces: ".gitignore: line 5: error parsing glob...".

The wrapping uses `Box<Error>` for the inner error. This prevents infinite size from recursive types. Without boxing, `Error` would contain `Error` which contains `Error` — the compiler can't determine a finite size.

See: Companion Code Section 3

Section 4: Partial Errors

The Partial variant handles a subtle situation: what do you do when parsing a file partially succeeds?

Consider a gitignore file with ten lines. Nine parse correctly; one has invalid syntax. Failing entirely discards nine valid rules. Silently ignoring the bad line hides a real problem. The Partial variant reports all errors while letting valid rules work.

The `is_partial` method recursively checks whether any error in the chain is partial. This lets callers decide how to handle partial success — some might treat it as failure, others as warning-worthy success.

See: Companion Code Section 4

Section 5: I/O Error Handling

I/O errors receive special treatment because they're so common and because callers often need to inspect them.

The `is_io` method checks whether an error is purely an I/O error. Like `is_partial`, it recurses through wrappers. A WithPath containing an Io is still "an I/O error" for this purpose.

The `io_error` method provides borrowed access to the underlying `std::io::Error` if one exists. This enables checking error kinds without ownership transfer. The companion `into_io_error` consumes self for cases where ownership is needed.

The Clone implementation for Error specially handles the Io case. std::io::Error isn't Clone, but you can reconstruct one from the raw OS error code or by copying the kind and message.

See: Companion Code Section 5

Section 6: The PartialErrorBuilder

The PartialErrorBuilder is a private helper for accumulating errors during operations that can partially fail.

Its methods (`push`, `push_ignore_io`, `maybe_push`) provide convenient ways to accumulate errors. The `push_ignore_io` variant silently drops I/O errors — useful when I/O failures should be warnings rather than errors.

The `into_error_option` method collapses the accumulated errors. No errors returns None. One error returns that error directly. Multiple errors wraps them in Partial.

This builder appears throughout the crate. Any operation that processes multiple items and can fail per-item uses it.

See: Companion Code Section 6

Section 7: The Match Enum — The Filtering Decision

The Match enum is how the ignore crate represents filtering decisions. Every pattern match, every type check, every override evaluation returns a Match.

Three variants cover all cases: `None` means no rule matched. `Ignore` means a rule says to skip this file. `Whitelist` means a rule explicitly includes this file.

The type parameter T typically carries metadata about which rule matched. For gitignore, it identifies the specific file and glob. This enables debugging: you can trace exactly why a file was ignored.

See: Companion Code Section 7

Section 8: Match Semantics — Ignore vs Whitelist

The Ignore/Whitelist distinction handles negation patterns. In gitignore, lines starting with `!` negate previous patterns:

```
*.log          # Ignore all .log files
!important.log # But not this one
```

The first pattern produces `Ignore`. The second produces `Whitelist`. When evaluating a path, later matches take precedence. A Whitelist after an Ignore means "include this file."

The `is_ignore` and `is_whitelist` methods check the variant. Note that `None` is neither — it means no rule had an opinion about this file.

See: Companion Code Section 8

Section 9: Match Combinators

Match provides several combinator methods for working with filtering results.

The `invert` method flips Ignore to Whitelist and vice versa. This handles the `-T` (type negation) flag — selecting all files that don't match a type.

The `inner` method provides access to the wrapped value, returning `Option<&T>`. This enables inspecting match metadata without consuming the Match.

The `map` method transforms the inner value, enabling type conversions while preserving the match variant.

The `or` method provides fallback logic. If self is None, return other. Otherwise return self. This implements "first match wins" semantics when combining multiple matchers.

See: Companion Code Section 9

Section 10: How Match Flows Through the Crate

Understanding Match prepares you for the rest of the crate. Every filtering layer returns Match:

- Gitignore matching: does this path match a gitignore pattern?
- Type matching: does this path match selected file types?
- Override matching: does this path match a command-line glob?

These get combined with precedence rules. Overrides beat gitignore. Explicit includes beat excludes. The final Match determines whether the walker yields the entry.

The flow roughly looks like: 1. Check overrides → Match 2. Check file types → Match 3. Check gitignore rules → Match 4. Combine with precedence → final decision

See: Companion Code Section 10

Key Takeaways

First, errors wrap with context rather than converting to strings. This preserves information for callers who need it.

Second, partial errors handle the common case where some operations succeed and others fail. This is especially important for configuration files.

Third, Match encodes filtering decisions with three states: ignore, whitelist, and no-opinion. The distinction enables negation patterns.

Fourth, Match combinators enable composing multiple filtering layers with clear precedence semantics.

Fifth, the generic parameter T in Match carries provenance information. You can trace exactly which rule caused which decision.

What to Read Next

With Error and Match understood, proceed to:

gitignore.rs — See how Match gets produced from gitignore pattern matching.

types.rs — See how Match gets produced from file type filtering.

dir.rs — See how multiple Match results get combined with precedence.