ripgrep crates/globset/src/lib.rs: Code Companion

Reference code for the Glob Pattern Matching lecture. Sections correspond to the lecture document.

Section 1: The Builder Pattern in Context

```rust
/// GlobSetBuilder builds a group of patterns that can be used to
/// simultaneously match a file path.
#[derive(Clone, Debug)]
pub struct GlobSetBuilder {
    pats: Vec<Glob>,
}

impl GlobSetBuilder {
    /// Create a new `GlobSetBuilder`. A `GlobSetBuilder` can be used to add new
    /// patterns. Once all patterns have been added, `build` should be called
    /// to produce a [`GlobSet`], which can then be used for matching.
    pub fn new() -> GlobSetBuilder {
        GlobSetBuilder { pats: vec![] }
    }

    /// Builds a new matcher from all of the glob patterns added so far.
    ///
    /// Once a matcher is built, no new patterns can be added to it.
    pub fn build(&self) -> Result<GlobSet, Error> {
        GlobSet::new(self.pats.iter())
    }

    /// Add a new pattern to this set.
    /// Returns &mut Self for method chaining
    pub fn add(&mut self, pat: Glob) -> &mut GlobSetBuilder {
        self.pats.push(pat);
        self
    }
}

impl GlobSet {
    /// Alternative constructor that returns a GlobSetBuilder
    /// Provides flexibility at call sites
    #[inline]
    pub fn builder() -> GlobSetBuilder {
        GlobSetBuilder::new()
    }

    /// Create an empty `GlobSet`. An empty set matches nothing.
    /// Note: const fn allows use in static/const contexts
    #[inline]
    pub const fn empty() -> GlobSet {
        GlobSet { len: 0, strats: vec![] }
    }
}
```

The `add` method returns `&mut GlobSetBuilder` to enable method chaining (e.g., `builder.add(glob1).add(glob2)`). The `build` method borrows `&self`, allowing the builder to be reused if needed.

Section 2: Error Design Philosophy

```rust
/// Represents an error that can occur when parsing a glob pattern.
#[derive(Clone, Debug, Eq, PartialEq)]
pub struct Error {
    /// The original glob provided by the caller.
    glob: Option<String>,
    /// The kind of error.
    kind: ErrorKind,
}

/// The kind of error that can occur when parsing a glob pattern.
#[derive(Clone, Debug, Eq, PartialEq)]
#[non_exhaustive]  // Allows adding new variants in future versions
pub enum ErrorKind {
    /// **DEPRECATED** - kept for backward compatibility
    InvalidRecursive,
    /// Occurs when a character class (e.g., `[abc]`) is not closed.
    UnclosedClass,
    /// Occurs when a range in a character (e.g., `[a-z]`) is invalid.
    InvalidRange(char, char),
    /// Occurs when a `}` is found without a matching `{`.
    UnopenedAlternates,
    /// Occurs when a `{` is found without a matching `}`.
    UnclosedAlternates,
    /// **DEPRECATED** - nested alternates now supported
    NestedAlternates,
    /// Occurs when an unescaped '\' is found at the end of a glob.
    DanglingEscape,
    /// An error associated with parsing or compiling a regex.
    Regex(String),
}

impl ErrorKind {
    fn description(&self) -> &str {
        match *self {
            // Note the helpful suggestion in error messages
            ErrorKind::UnopenedAlternates => {
                "unopened alternate group; missing '{' \
                (maybe escape '}' with '[}]'?)"
            }
            ErrorKind::UnclosedAlternates => {
                "unclosed alternate group; missing '}' \
                (maybe escape '{' with '[{]'?)"
            }
            // Other variants...
            ErrorKind::InvalidRange(_, _) => "invalid character range",
            ErrorKind::UnclosedClass => {
                "unclosed character class; missing ']'"
            }
            // ...
        }
    }
}

impl std::fmt::Display for Error {
```

```
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
match self.glob {            // Include the problematic pattern in the error
message            None => self.kind.fmt(f),            Some(ref glob) => {
write!(f, "error parsing glob '{}': {}", glob, self.kind)            }        }
}}
```

The `#[non_exhaustive]` attribute forces downstream code to handle unknown variants with a wildcard arm, enabling library evolution without breaking changes.

Section 3: The Candidate Abstraction

```rust
/// A candidate path for matching.
///
/// All glob matching in this crate operates on `Candidate` values.
/// Constructing candidates has a very small cost associated with it, so
/// callers may find it beneficial to amortize that cost when matching a single
/// path against multiple globs or sets of globs.
#[derive(Clone)]
pub struct Candidate<'a> {
    path: Cow<'a, [u8]>,       // Full normalized path
    basename: Cow<'a, [u8]>,   // Filename without directory
    ext: Cow<'a, [u8]>,        // File extension (without dot)
}

impl<'a> Candidate<'a> {
    /// Create a new candidate for matching from the given path.
    pub fn new<P: AsRef<Path> + ?Sized>(path: &'a P) -> Candidate<'a> {
        Self::from_cow(Vec::from_path_lossy(path.as_ref()))
    }

    fn from_cow(path: Cow<'a, [u8]>) -> Candidate<'a> {
        // Normalize once, then extract derived values
        let path = normalize_path(path);
        let basename = file_name(&path).unwrap_or(Cow::Borrowed(B("")));
        let ext = file_name_ext(&basename).unwrap_or(Cow::Borrowed(B("")));
        Candidate { path, basename, ext }
    }

    // Efficient prefix/suffix extraction for Aho-Corasick matching
    fn path_prefix(&self, max: usize) -> &[u8] {
        if self.path.len() <= max { &*self.path } else { &self.path[..max] }
    }

    fn path_suffix(&self, max: usize) -> &[u8] {
        if self.path.len() <= max {
            &*self.path
        } else {
            &self.path[self.path.len() - max..]
        }
    }
}
```

`Cow<'a, [u8]>` (Copy-on-Write) allows the candidate to either borrow from the input path or own normalized data, avoiding allocation when the path is already in the correct form.

Section 4: Strategy-Based Matching Architecture

```rust
#[derive(Clone, Debug)]
enum GlobSetMatchStrategy {
    Literal(LiteralStrategy),           // Exact path match: "src/main.rs"
    BasenameLiteral(BasenameLiteralStrategy), // Filename match: "Makefile"
    Extension(ExtensionStrategy),       // Extension match: "*.rs"
    Prefix(PrefixStrategy),             // Prefix match: "src/**"
    Suffix(SuffixStrategy),             // Suffix match: "**/*.rs"
    RequiredExtension(RequiredExtensionStrategy), // Hybrid: ext + regex
    Regex(RegexSetStrategy),            // Full regex fallback
}

// Pattern categorization during GlobSet construction
pub fn new<I, G>(globs: I) -> Result<GlobSet, Error>
where
    I: IntoIterator<Item = G>,
    G: AsRef<Glob>,
{
    // Initialize collectors for each strategy
    let mut lits = LiteralStrategy::new();
    let mut base_lits = BasenameLiteralStrategy::new();
    let mut exts = ExtensionStrategy::new();
    let mut prefixes = MultiStrategyBuilder::new();
    let mut suffixes = MultiStrategyBuilder::new();
    let mut required_exts = RequiredExtensionStrategyBuilder::new();
    let mut regexes = MultiStrategyBuilder::new();

    for (i, p) in it.enumerate() {
        let p = p.as_ref();
        // Route each pattern to its optimal strategy
        match MatchStrategy::new(p) {
            MatchStrategy::Literal(lit) => {
                lits.add(i, lit);
            }
            MatchStrategy::BasenameLiteral(lit) => {
                base_lits.add(i, lit);
            }
            MatchStrategy::Extension(ext) => {
                exts.add(i, ext);
            }
            MatchStrategy::Prefix(prefix) => {
                prefixes.add(i, prefix);
            }
            MatchStrategy::Suffix { suffix, component } => {
                if component {
                    // Also add to literals for component matching
                    lits.add(i, suffix[1..].to_string());
                }
                suffixes.add(i, suffix);
            }
            MatchStrategy::RequiredExtension(ext) => {
                required_exts.add(i, ext, p.regex().to_owned());
            }
            MatchStrategy::Regex => {
                regexes.add(i, p.regex().to_owned());
```

```
            }        }    }    // ... build strategies and return GlobSet}
```

The `MatchStrategy::new(p)` call analyzes each glob pattern and determines which optimization strategy can handle it most efficiently.

Section 5: The Multi-Pattern Matching Engine

```rust
impl GlobSet {
    /// Returns true if any glob in this set matches the path given.
    /// Short-circuits on first match.
    pub fn is_match_candidate(&self, path: &Candidate<'_>) -> bool {
        if self.is_empty() {
            return false;
        }
        for strat in &self.strats {
            if strat.is_match(path) {
                return true;  // Early exit on first match
            }
        }
        false
    }

    /// Returns true if all globs in this set match the path given.
    /// Empty set returns true (vacuous truth).
    pub fn matches_all_candidate(&self, path: &Candidate<'_>) -> bool {
        for strat in &self.strats {
            if !strat.is_match(path) {
                return false;  // Early exit on first non-match
            }
        }
        true
    }

    /// Adds the sequence number of every glob pattern that matches.
    /// Results are sorted and deduplicated.
    pub fn matches_candidate_into(
        &self,
        path: &Candidate<'_>,
        into: &mut Vec<usize>,
    ) {
        into.clear();  // Always start fresh
        if self.is_empty() {
            return;
        }
        // Collect matches from all strategies
        for strat in &self.strats {
            strat.matches_into(path, into);
        }
        // Remove duplicates (pattern might match multiple strategies)
        into.sort();
        into.dedup();
    }
}
```

The `matches_into` variant lets callers provide a pre-allocated vector, enabling efficient batch processing by reusing allocations across many path matches.

Section 6: Hash-Based Strategies

```rust
#[derive(Clone, Debug)]
struct LiteralStrategy(fnv::HashMap<Vec<u8>, Vec<usize>>);

impl LiteralStrategy {
    fn new() -> LiteralStrategy {
        LiteralStrategy(fnv::HashMap::default())
    }

    fn add(&mut self, global_index: usize, lit: String) {
        // Multiple patterns can map to the same literal
        self.0.entry(lit.into_bytes()).or_insert(vec![]).push(global_index);
    }

    fn is_match(&self, candidate: &Candidate<'_>) -> bool {
        // O(1) hash lookup regardless of pattern count
        self.0.contains_key(candidate.path.as_bytes())
    }

    #[inline(never)]  // Hint to avoid inlining (code size optimization)
    fn matches_into(
        &self,
        candidate: &Candidate<'_>,
        matches: &mut Vec<usize>,
    ) {
        if let Some(hits) = self.0.get(candidate.path.as_bytes()) {
            matches.extend(hits);
        }
    }
}

#[derive(Clone, Debug)]
struct ExtensionStrategy(fnv::HashMap<Vec<u8>, Vec<usize>>);

impl ExtensionStrategy {
    fn is_match(&self, candidate: &Candidate<'_>) -> bool {
        if candidate.ext.is_empty() {
            return false;  // No extension means no match
        }
        self.0.contains_key(candidate.ext.as_bytes())
    }
    // ... similar structure to LiteralStrategy
}

#[derive(Clone, Debug)]
struct BasenameLiteralStrategy(fnv::HashMap<Vec<u8>, Vec<usize>>);
// ... same pattern, keyed by basename instead of full path
```

All three strategies use FNV (Fowler-Noll-Vo) hashing, optimized for small keys like file extensions and basenames.

Section 7: Prefix and Suffix Strategies

```rust
#[derive(Clone, Debug)]
struct PrefixStrategy {
    matcher: AhoCorasick,   // Multi-pattern string matching
    map: Vec<usize>,        // Pattern index to global index
    longest: usize,         // Maximum prefix length to check
}

impl PrefixStrategy {
    fn is_match(&self, candidate: &Candidate<'_>) -> bool {
        // Only check the relevant prefix of the path
        let path = candidate.path_prefix(self.longest);
        for m in self.matcher.find_overlapping_iter(path) {
            if m.start() == 0 {  // Must match from the beginning
                return true;
            }
        }
        false
    }

    fn matches_into(
        &self,
        candidate: &Candidate<'_>,
        matches: &mut Vec<usize>,
    ) {
        let path = candidate.path_prefix(self.longest);
        for m in self.matcher.find_overlapping_iter(path) {
            if m.start() == 0 {
                matches.push(self.map[m.pattern()]);
            }
        }
    }
}

#[derive(Clone, Debug)]
struct SuffixStrategy {
    matcher: AhoCorasick,
    map: Vec<usize>,
    longest: usize,
}

impl SuffixStrategy {
    fn is_match(&self, candidate: &Candidate<'_>) -> bool {
        let path = candidate.path_suffix(self.longest);
        for m in self.matcher.find_overlapping_iter(path) {
            if m.end() == path.len() {  // Must match at the end
                return true;
            }
        }
        false
    }
}
```

The `longest` field enables an optimization: only examine the portion of the path that could possibly match, reducing work for long paths.

Section 8: RequiredExtension and Regex Strategies

```rust
#[derive(Clone, Debug)]
struct RequiredExtensionStrategy(fnv::HashMap<Vec<u8>, Vec<(usize, Regex)>>);

impl RequiredExtensionStrategy {
    fn is_match(&self, candidate: &Candidate<'_>) -> bool {
        if candidate.ext.is_empty() {
            return false;
        }
        // First: O(1) extension lookup narrows candidates
        match self.0.get(candidate.ext.as_bytes()) {
            None => false,
            Some(regexes) => {
                // Then: run regex only on filtered candidates
                for &(_, ref re) in regexes {
                    if re.is_match(candidate.path.as_bytes()) {
                        return true;
                    }
                }
                false
            }
        }
    }
}

#[derive(Clone, Debug)]
struct RegexSetStrategy {
    matcher: Regex,
    map: Vec<usize>,
    // Pool of PatternSets to avoid allocation per match
    patset: Arc<Pool<PatternSet, PatternSetPoolFn>>,
}

type PatternSetPoolFn =
    Box<dyn Fn() -> PatternSet + Send + Sync + UnwindSafe + RefUnwindSafe>;

impl RegexSetStrategy {
    fn matches_into(
        &self,
        candidate: &Candidate<'_>,
        matches: &mut Vec<usize>,
    ) {
        let input = regex_automata::Input::new(candidate.path.as_bytes());
        let mut patset = self.patset.get();  // Get from pool
        patset.clear();
        self.matcher.which_overlapping_matches(&input, &mut patset);
        for i in patset.iter() {
            matches.push(self.map[i]);
        }
        PoolGuard::put(patset);  // Return to pool
    }
}
```

The `RequiredExtensionStrategy` demonstrates a two-phase filter: cheap hash lookup first, expensive regex only if needed. The `RegexSetStrategy` uses object pooling to amortize allocation costs.

Quick Reference

Strategy Selection Summary

Pattern Type	Example	Strategy	Lookup Complexity
Exact path	`src/main.rs`	Literal	O(1) hash
Basename only	`Makefile`	BasenameLiteral	O(1) hash
Extension	`*.rs`	Extension	O(1) hash
Prefix	`src/**`	Prefix (Aho-Corasick)	O(n) where n = prefix length
Suffix	`**/*.rs`	Suffix (Aho-Corasick)	O(n) where n = suffix length
Ext + complex	`src/**/*.rs`	RequiredExtension	O(1) + regex
Complex	`{foo,bar}/**`	Regex	Full regex cost

Key Types

```rust
// Primary API types
GlobSet           // Immutable, optimized matcher for multiple patterns
GlobSetBuilder    // Accumulates patterns before building
Candidate<'a>     // Pre-processed path for efficient matching

// Strategy enum (internal)
enum GlobSetMatchStrategy {
    Literal(LiteralStrategy),
    BasenameLiteral(BasenameLiteralStrategy),
    Extension(ExtensionStrategy),
    Prefix(PrefixStrategy),
    Suffix(SuffixStrategy),
    RequiredExtension(RequiredExtensionStrategy),
    Regex(RegexSetStrategy),
}
```

Matching Method Variants

Method	Returns	Use Case
`is_match(path)`	`bool`	Quick existence check
`is_match_candidate(candidate)`	`bool`	Amortized existence check
`matches(path)`	`Vec<usize>`	Get all matching pattern indices
`matches_candidate(candidate)`	`Vec<usize>`	Amortized match collection
`matches_into(path, vec)`	`()`	Reuse allocation
`matches_candidate_into(candidate, vec)`	`()`	Full amortization
`matches_all(path)`	`bool`	All patterns must match