ripgrep

https://github.com/BurntSushi/ripgrep

2024-02-27

Andrew Gallant (aka BurntSushi)

What is `grep`?

```
$ cat haystack
foo
quux
baz
$ grep 'q.*x' haystack
quux
```

Globally search a **RE**gular expression and **P**rint

What is ripgrep?

```
$ rg 'fn line_buffer'
crates/searcher/src/searcher/mod.rs
216:    fn line_buffer(&self) -> LineBuffer {
```

- Recursively search a directory.
- By default, it ignores:
  - Files that match your `.gitignore`, `.ignore` and `.rgignore`.
  - Hidden files and directories.
  - Binary files.
- `rg -uuu` disables all automatic filtering.
- Fast.

Also: the output format is different

Instead of:

```
$ grep -r from_unix ./
./src/time.rs:    pub fn from_unix(
./src/civil/date.rs:            Date::from(Time::from_unix(-86_400, 0).unwrap()),
```

We get:

```
./src/civil/date.rs
276:            Date::from(Time::from_unix(-86_400, 0).unwrap()),

./src/time.rs
25:    pub fn from_unix(
```

Brief History

- In the beginning, Ken Thompson gave us `grep`. First released in 1974.
  - POSIX, BSD grep and GNU grep
- `ack`, by Andy Lester, first released 2005.
- The Silver Searcher (`ag`), by Geoff Greer, first released 2011. (Originally named `bta` for "better than ack.")

Why?

- More CPUs.
- Code repositories are growing.
  - `.git`
  - npm's `node_modules`
  - Rust's `target`
- Beyond grep is indexing. grep is a sweetspot.

Recursive `grep` in under 20 lines of Rust

```rust
let Some(pattern) = env::args().nth(1) else { return };
let re = Regex::new(&pattern)?;
for result in WalkDir::new("./") {
    let entry = result?;
    if !entry.file_type().is_file() {
        continue;
    }

    let mut file = File::open(entry.path())?;
    let mut contents = String::new();
    if file.read_to_string(&mut contents).is_ok() {
        for line in contents.lines() {
            if re.is_match(&line) {
                let path = entry.path().display();
                println!("{}:{}", path, line);
            }
        }
    }
}
```

What makes ripgrep fast?

- Parallelism
- Regex engine
- SIMD (Single Instruction, Multiple Data)

Parallelism: simplest approach

- Create a pool of search worker threads.
- Use a library like `walkdir` to visit each file in a directory tree.
- Send each file to the pool.
- Avoid inter-leaving output.

  We don't want this:

```
$ rg from_unix
src/civil/date.rs:            Date::from(Time::from_unix(-86_400, 0).unwrap()),
src/time.rs:    pub fn from_unix(
src/civil/date.rs:            Date::from(Time::from_unix(-86_401, 0).unwrap()),
```

Parallelism: directory traversal

- Directory traversal itself takes time.
- Exacerbated in ripgrep because of glob matching.
- Worker threads accept files *or* directories as input.
- Worker threads become consumers and producers.
- Termination is tricky.

Regex engine

- Says whether `q.*x` matches `quux`.
- By default only supports "true" regular expressions.
- Uses finite automata internally. (No risk of catastrophic backtracking.)
- Maintained as part of the Rust project. Available as the crate `regex`.

Regex engine: NFAs versus DFAs

- Equivalent in what they can express.
- Epsilon transitions!
- But from an engineering perspective, very different trade-offs!
- NFAs are usually "simulated" with a virtual machine.
  - Easy to hack things into it, such as look-around and capturing groups.
  - But, each character of input might visit all of the states in the NFA.
  - Can be constructed in time proportional to the size of the pattern.
- DFAs are usually implemented with a table of transitions.
  - Hard to hack things into it.
  - Transition function is computed in a constant number of instructions.
  - Worst case construction time is exponential in the size of the pattern.

Regex engine: lazy DFAs strike a balance

- Initial construction is equivalent to building an NFA.
- A table of transitions for a DFA is lazily constructed.
- At most one new transition (and state) are added for each character of input.
- Constrained to a fixed amount of memory. Table is cleared when we reach this point.

Regex engine: literals

- Some regexes can be represented as a finite set of strings:

```
$ regex-cli debug literal 'ab?[yz]ghi'
abyghi
abzghi
ayghi
azghi
```

- Others might have useful prefixes:

```
$ regex-cli debug literal '(foo|bar)\w+'
foo
bar
```

- Finding literals can be *orders* of magnitude faster than the regex engine.

SIMD (Single Instruction, Multiple Data)

- Load a chunk of data (e.g., 16 bytes of a string) into one register.
- Perform "bulk" operations using that register.
- Can increase throughput dramatically.

SIMD: sketch of `memchr`

```
let needle_vector = _mm_set1_epi8(b'z');
let start_ptr = haystack.as_ptr();
let end_ptr = start_ptr.add(haystack.len());
let mut ptr = start_ptr;
while ptr < end_ptr {
    let chunk = _mm_load_si128(ptr as *const __m128i);
    let chunk_eq = _mm_cmpeq_epi8(needle_vector, chunk);
    if _mm_movemask_epi8(chunk_eq) != 0 {
        let mut at = ptr - start_ptr;
        let matching_offsets = _mm_movemask_epi8(chunk_eq);
        return Some(at + matching_offsets.trailing_zeros());
    }
    ptr = ptr.add(16);
}
None
```

SIMD: frequency heuristic

- Maximize amount of time spent in vector routines.
- `z` is probably going to occur less frequently than `a`.
- Use heuristic ranking of bytes to influence byte to use with `memchr`.
- Given `aezio`, look for `z`.

SIMD: multiple literals (briefly)

- Uses port of Teddy algorithm designed by Hyperscan authors.
- Quickly identifies 1, 2, 3 or 4 byte prefixes.
- Can be used with case insensitive regexes:

```
$ regex-cli debug literal '(?i)abc'
ABC
ABc
AbC
Abc
aBC
aBc
abC
abc
```

Future?

- More SIMD for bigger sets of literals.
- Glushkov automata (no epsilon transitions!)
- Indexing?

Links

Repository: https://github.com/BurntSushi/ripgrep

Regex engine: https://github.com/rust-lang/regex

Regex engine benchmarks: https://github.com/BurntSushi/rebar