cargo-guppy

track and query Cargo dependencies

Rain <rain1@fb.com>

February 2021

The problem

- Analyze Cargo dependency graphs for:
  - license checks
  - dependency audits
  - TCB tracking
  - ...
  - *cool new features*

Existing solutions

`cargo`?

- CLI doesn't have all the features we want
- Rust API isn't for external consumption
  - Large and unstable
  - Many dependencies (e.g. libgit2)
  - Missing documentation

But...

`cargo metadata` has:

- package information
- dependency information
- everything we need

```
% cargo metadata --format-version=1 | jq -C | less -FRXK
{
  "packages": [
    {
      "name": "adler",
      "version": "0.2.3",
      "id": "adler 0.2.3 (registry+https://github.com/rust-lang/crates.io-index)",
      "license": "0BSD OR MIT OR Apache-2.0",
      "license_file": null,
      "description": "A simple clean-room implementation of the Adler-32 checksum",
      "source": "registry+https://github.com/rust-lang/crates.io-index",
      "dependencies": [
        {
          "name": "compiler_builtins",
          "source": "registry+https://github.com/rust-lang/crates.io-index",
          "req": "^0.1.2",
          "kind": null,
          "rename": null,
          "optional": true,
          "uses_default_features": true,
          "features": [],
          "target": null,
          "registry": null
        },
        {
          "name": "rustc-std-workspace-core",
          "source": "registry+https://github.com/rust-lang/crates.io-index",
          "req": "^1.0.0",
          "kind": null,
          "rename": "core",
          "optional": true,
          "uses_default_features": true,
          "features": [],
          "target": null,
          "registry": null
        },
        {
```

Enter guppy

- Read `cargo metadata` input
- Parse as graph structure
- Present nice APIs

The package graph

- Central structure is `PackageGraph`
- Nodes are packages, edges are dependencies
- Directed, may be cyclic (dev-dependencies)
- Most other types borrow from `PackageGraph`
  - Indicated with a `'g` lifetime
- Uses `petgraph` with integer indexes internally
- Maps integers to borrowed structures externally
- Immutable + `Send` + `Sync` means easy parallelization

The feature graph

- `FeatureGraph<'g>` is a second, auxiliary graph built from `PackageGraph`
- Nodes are `(package, feature)` pairs, edges are either:
  - Feature dependencies, e.g. `foo = ["bar", "baz"]`
  - Cross-links, e.g. `dep = { version = "1", features = ["foo"] }`
- Computed on-demand

Core types

Abstraction	Package type	Feature type
Main graph	`PackageGraph`	`FeatureGraph<'g>`
Identifier	`PackageId`	`FeatureId<'g>`
Extended information	`PackageMetadata<'g>`	`FeatureMetadata<'g>`
Dependency edge triple	`PackageLink<'g>`	`CrossLink<'g>` *
Dependency query	`PackageQuery<'g>`	`FeatureQuery<'g>`
Query result	`PackageSet<'g>`	`FeatureSet<'g>`

* currently only cross-links are exposed, eventually `FeatureLink<'g>`

Core methods

from	to	method
`Graph`	`Metadata`	`metadata`
`Metadata`	`Link` iterator	`direct_links_`
`Graph`	`Query`	`query_`
`Graph` or `Query`	`Set`	`resolve_`
`Set`	`Metadata`	`packages` or `features`
`Set`	`Link` iterator	`links`

`_` indicates that it's several methods, e.g. `query_forward`, `query_reverse` and `query_directed`

Switching between graphs

abstraction	p □ f	f □ p
`Graph`	`feature_graph`	`package_graph`
`Query`	`to_feature_query`	`to_package_query`
`Set`	`to_feature_set`	`to_package_set`

Package □ feature requires a `FeatureFilter`. Most people will use `StandardFeatures::None`, `Default` or `All`.

Filtering during traversals

- Get all transitive dependencies: `PackageQuery::resolve`
- But what if you don't want to follow all edges?
- `PackageQuery::resolve_with()` accepts a `PackageResolver<'g>`
  - Trait with `fn accept(query, link) -> bool`
- Also available as a callback: `PackageQuery::resolve_with_fn`
- Also available for `FeatureQuery`

Applications

Basic traversals

- Get all transitive dependencies: `query.resolve()`
- Ignore dev-only dependencies:

  `query.resolve_with_fn(|_, link| !link.dev_only())`
- Direct dependencies of workspace:

```
query.resolve_with_fn(|_, link| {
    let (from, to) = link.endpoints();
    from.in_workspace() && !to.in_workspace()
})
```

Cargo builds

- Which packages and features will a build command include?
- Start from a `FeatureSet` describing initials
- Traverse dependency graphs the same way Cargo would
- Customize behavior through `CargoOptions`
  - Platforms and more

```
[metadata]
version = 'v1-install'
include-dev = true
initials-platform = 'proc-macros-on-target'

[metadata.host-platform]
triple = 'thumbv7a-pc-windows-msvc'
target-features = 'all'
flags = ['cargo_web', 'test-flag']

[[metadata.features-only]]
name = 'guppy-benchmarks'
version = '0.1.0'
workspace-path = 'internal-tools/benchmarks'
features = []

[[metadata.features-only]]
name = 'guppy-summaries'
version = '0.2.0'
workspace-path = 'guppy-summaries'
features = []

[[metadata.features-only]]
name = 'proptest-ext'
version = '0.1.0'
workspace-path = 'internal-tools/proptest-ext'
features = []

[[target-package]]
name = 'cargo-compare'
version = '0.1.0'
workspace-path = 'internal-tools/cargo-compare'
status = 'initial'
features = []

[[target-package]]
name = 'fixture-manager'
```

Cargo builds: v1 and v2 resolvers

- v1 (classic) resolver
  - Packages may or may not be enabled depending on dev, features or platforms
  - Feature resolution is *independent of* which packages are enabled
  - Simulated through 1 feature query + 2 package queries
    - One package query for the target platform, one for the host
- v2 (new) resolver
  - Packages may or may not be enabled depending on dev, features or platforms
  - Feature resolution is *dependent on* which packages are enabled
  - Simulated through 2 feature queries + 2 package queries
    - One each for the target, one each for the hos

Cargo builds: property testing

- Comparison testing with Cargo
  - Generate random queries and compare against Cargo
- Consistency testing with previous versions of guppy
  - Generate random queries and simulate builds
  - Summaries with build results checked into the repo
  - These should only change if there's a good reason

Cool new features

Determinator

- Only run tests for packages that changed from upstream
- Given old metadata, new metadata and paths changed:
  - Map each path to a package
  - Simulate Cargo builds for each package and see which changed
- Support for custom rules
- Diem CI: p25 90% faster, p50 60+%
- docs.rs/determinator

```
# Standard ignore and other metadata files.
[[path-rule]]
globs = ["**/.gitignore", "**/.gitattributes", ".dockerignore", ".hgignore", ".svnignore", "**/.ignore"]
mark-changed = []

# Files that can affect the global build. Cargo.toml may contain updates to build flags or profile overrides,
# so rebuild everything if it changes. (We could do a more sophisticated analysis in the future.)
[[path-rule]]
globs = ["rust-toolchain", "Cargo.toml", "**/.cargo/config", "**/.cargo/config.toml"]
mark-changed = "all"

# Tool files that don't influence builds or tests.
[[path-rule]]
globs = ["clippy.toml", "rustfmt.toml", ".lintrules/**/*"]
mark-changed = []

# Cargo.lock is ignored, since the determinator does a deeper analysis to figure out which packages changed.
[[path-rule]]
globs = ["Cargo.lock"]
mark-changed = []

# README, LICENSE and other metadata files are ignored throughout the codebase.
[[path-rule]]
globs = ["**/README*", "**/LICENSE*", "**/CONTRIBUTING*", "**/CODE_OF_CONDUCT*", "**/SECURITY*"]
mark-changed = []
```

Hakari

- Manage packages for dependency feature unification
  - Workspace-hack packages used by many large projects (rustc, Firefox, Diem)
- Simulate Cargo builds and look for non-workspace packages built with more than one feature set
- Speeds up Diem builds by 15-30% or more
- docs.rs/hakari

```
### BEGIN HAKARI SECTION
# version = 'v1'
# verify-mode = true
# unify-target-host = 'none'
# unify-all = false
# platforms = []
#
# [[omitted-packages]]
# name = 'either'
# version = '1.6.1'
# crates-io = true
#
# [[omitted-packages]]
# name = 'petgraph'
# version = '0.5.1'
# crates-io = true
#
# [[omitted-packages]]
# name = 'url'
# version = '2.1.1'
# crates-io = true
#
# [[omitted-packages]]
# name = 'web-sys'
# version = '0.3.45'
# crates-io = true

[target.'cfg(all())'.dependencies]
bstr = { version = "0.2", features = ["default", "lazy_static", "regex-automata", "serde", "serde1", "serd
byteorder = { version = "1", default-features = false, features = ["std"] }
clap = { version = "2", features = ["ansi_term", "atty", "color", "default", "strsim", "suggestions", "vec
itoa = { version = "0.4", features = ["default", "std"] }
libc = { version = "0.2", features = ["default", "std"] }
log = { version = "0.4", default-features = false, features = ["std"] }
memchr = { version = "2", features = ["default", "std", "use_std"] }
num-traits = { version = "0.2", features = ["default", "std"] }
regex = { version = "1", features = ["aho-corasick", "default", "memchr", "perf", "perf-cache", "perf-dfa"
regex-syntax = { version = "0.6", features = ["default", "unicode", "unicode-age", "unicode-bool", "unicod
serde = { version = "1", features = ["default", "derive", "serde_derive", "std"] }
serde_json = { version = "1", features = ["default", "raw_value", "std"] }
winapi = { version = "0.3", default-features = false, features = ["basetsd", "consoleapi", "errhandlingapi

[target.'cfg(all())'.build-dependencies]
libc = { version = "0.2", features = ["default", "std"] }
log = { version = "0.4", default-features = false, features = [] }
proc-macro2 = { version = "1", features = ["default", "proc-macro"] }
quote = { version = "1", features = ["default", "proc-macro"] }
syn = { version = "1", features = ["clone-impls", "default", "derive", "full", "parsing", "printing", "pro

### END HAKARI SECTION
```

Questions?