How to do code coverage in Rust

Now that we have source based coverage in Rust, we're going to look at how to do the following:

- Run coverage locally
- Integrate with IDE for inline coverage indicators
- Run coverage in CI and upload results to a coverage service provider

But first, let's look at the history of code coverage in Rust, it might help you when you Google around for solutions, to understand what's current and what's not.

History of code coverage in Rust

Tarpaulin

Tarpaulin is an easy tool to run code coverage with Rust, but has limitations. An intuitive description for how Tarpaulin works: it will **instrument your code** and then use `ptrace` to eavesdrop on what's happening in order to count lines for coverage analysis.

This is why Tarpaulin **only supports Linux**, and also specifically **only supports x86_64 processors**.

For accuracy's sake — Tarpaulin provides line coverage and is fairly reliable but still include some inaccuracies in the results.

gcov

Before source based coverage, Rust relied on the `gcov` technique, which relied on **debug info** to map from LLVM IR (bottom, generated and hidden from you) to lines of source code (top, what you're used to seeing). Think about it like making a really good guess **for *names* into lines of code**, and keeping count of which line is ran and how many times.

As such, and since the **resolution of information is lost** from layer to layer as you move from source to compiled binary form, the `gcov` based techinque was not the most accurate.

Source based coverage

Source based coverage lets Rust have instrumentation done right at the compiler level while then generating IR from source, which moves down to all compilation stages, and then **lets the coverage analysis tools use all of that high-quality information** to correlate and pinpoint exactly a given low-level construct to its higher-level source code (that you wrote).

This results in a **much higher accuracy level** and better implementation of coverage instrumentation. To get a feel of what "accuracy is", then take a look at this short-circuit statement:

```
if a < b && c > d {
  // ..
}
```

If `a > b` (the **inverse** of the first condition), we know that the **next condition does not need to be run**. This is shortcircuiting. And so, source based coverage will be able to tell you **that** `c > d` **was never exercised**, so you could write a test for that condition as well, where the gcov method would list all of that line as exercised (!).

Today, **source based coverage** in Rust is the recommended way to do coverage. It's also the most usable, most accurate, and most portable way to do coverage.

Our goals

Source based coverage opens up the opportunity to have a **holy-trinity of coverage**:

- Run locally for **fast feedback** for an individual.
- Seamless **inline** IDE integration
- Run in CI, PRs, and centrally **for a team**

Running Locally

To do code coverage with the new **source based coverage** features, you need to perform a few moves. **We'll see the manual steps first, and automate it all later**.

Generally we want **tests to drive our coverage**, so we'll run `cargo test`, but we need to enable instrumentation, and also tell LLVM to output its raw profiling data.

```
$ CARGO_INCREMENTAL=0 RUSTFLAGS='-Cinstrument-coverage' LLVM_PROFILE_FILE='cargo-test-%p-%m.profraw' cargo test
```

Next, we need to take all of that data, and **generate a report** that can be ingested by tools and read by machines, or that we can browse around as humans.

As HTML:

```
$ grcov . --binary-path ./target/debug/deps/ -s . -t html --branch --ignore-not-existing --ignore '../*' --ignore "/*" -o target/coverage/html
```

LINES	FUNCTIONS	BRANCHES
56.78 %	36.13 %	100 %

Directory	Line Coverage		Functions		Branches	
backpack/src	65.94%	513 / 778	35.83%	182 / 508	100%	0 / 0
backpack/src/bin	3.57%	1 / 28	33.33%	1 / 3	100%	0 / 0
backpack/src/bin/commands	0%	0 / 177	0%	0 / 20	100%	0 / 0

As `lcov` , a popular output format:

```
$ grcov . --binary-path ./target/debug/deps/ -s . -t lcov --branch --ignore-not-existing --ignore '../*' --ignore "/*" -o target/coverage/tests.lcov
```

Then, it's up to us what to do with the data:

- Tell **our IDE** to find the `lcov` files for inline coverage highlights
- Upload the `lcov` files **to code coverage providers** such as Codecov
- Use the `lcov` data to create **custom reports** or to fail tests

When done, we'll need to clean up the `*.profraw` files that were littered alongside our code.

Automating with `xtask`

`cargo-xtask` is the current implementation of **cargo tasks**. Think of it as a Rust-based alternative to `make`, but that it is **pure Rust**, all around -- it's just another project in your workspace, and there are very little strict rules around it other than a few **best practices**. To read more about it check out the xtask repo

Here's how all of this process looks with my current flavor of `cargo xtask cover` (you can take a look at the full example in my rust-starter):

Now if you run:

```
$ cargo xtask coverage
```

You'll have `lcov` files waiting for you in `coverage/` which you can upload to a coverage provider or use in other tools.

And if you run:

```
$ cargo xtask coverage --dev
```

You'll be offered to **open a report** which you can load in your browser, no other coverage product or provider needed.

Inline coverage with VSCode

```rust
fn to_value<T: serde::ser::Serialize>(value: &T) -> Result<serde_json::Value, Error> {
    serde_json::to_value(value)
}

fn from_value<T: serde::ser::Serialize + serde::de::DeserializeOwned>(
    value: serde_json::Value,
) -> Result<T, Error> {
    serde_json::from_value(value)      You, 6 days ago • ii …
}

fn merge_value(a: &mut Value, b: &Value) {
    match (a, b) {
        (Value::Object(ref mut a: &mut Map<String, Value>), &Value::Object(ref b: &Map<String, Value>)) => {
            for (k: &String, v: &Value) in b {
                merge_value(a: a.entry(key: k).or_insert(default: Value::Null), b: v);
            }
        }
        (Value::Array(ref mut a: &mut Vec<Value>), &Value::Array(ref b: &Vec<Value>)) => {
            a.extend(iter: b.clone());
        }
        (Value::Array(ref mut a: &mut Vec<Value>), &Value::Object(ref b: &Map<String, Value>)) => {
            a.extend(iter: [Value::Object(b.clone())]);
        }
        (_, Value::Null) => {} // do nothing
        (a: &mut Value, b: &Value) => {
            *a = b.clone();
        }
    }
}

pub fn merge<T: serde::ser::Serialize + serde::de::DeserializeOwned>(
```

If you want to see coverage **inline** with your code, as you type, you can do that with the coverage gutters extension, which is recommended to use with any language that can produce `lcov` really, so not only useful for Rust.

After you've installed it, you need to let it know where to find `lcov` files generically in your project.

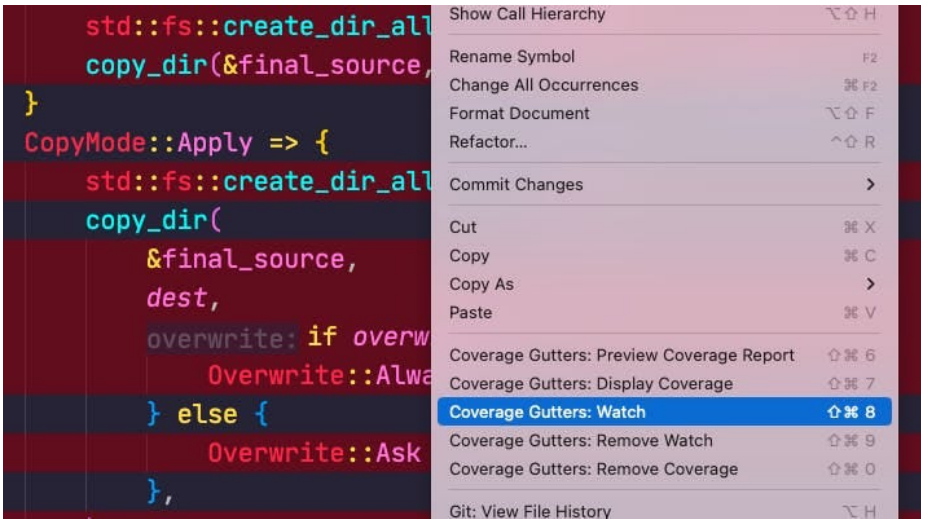
For us, if you use the `xtask` above for all Rust projects, it will always be in: `coverage/tests.lcov`.

So this is the configuration I have, as an example:

```
"coverage-gutters.coverageFileNames": [
    "coverage/tests.lcov",
    "lcov.info",
    "cov.xml",
```

```
    "coverage.xml",
    "jacoco.xml",
    "coverage.cobertura.xml"
  ],
```

You can then view your inline coverage with *watch* or just toggle it:



Running in your CI

Next, we want to use ***the same coverage task*** in our CI, which is another benefit we get by using `xtask`. Basically, running our coverage in CI will reduce to:

```
$ cargo xtask coverage
```

And finding a way to upload `coverage/*.lcov` to our coverage provider.

In this example, we'll use **Github Action** as our CI, and **Codecov** as our provider, but it shouldn't be too different for any other set of vendors.

Our **Github workflow** should also include set up for the various coverage tools we need, so first install `llvm-tools-preview`.

```
- name: Install toolchain
    id: toolchain
    uses: actions-rs/toolchain@v1
    with:
      toolchain: stable
      target: x86_64-unknown-linux-musl
      override: true
      components: llvm-tools-preview
```

Then, we download `grcov`, work through the rest of our task, and upload to Codecov:

```
- name: Download grcov
    run: |
      mkdir -p "${HOME}/.local/bin"
      curl -sL https://github.com/mozilla/grcov/releases/download/v0.8.10/grcov-x86_64-unknown-linux-gnu.tar.bz2 | tar jxf - -C "${HOME}/.local/bin"
      echo "$HOME/.local/bin" >> $GITHUB_PATH

- name: Run xtask coverage
    uses: actions-rs/cargo@v1
    with:
      command: xtask
      args: coverage

- name: Upload to codecov.io
    uses: codecov/codecov-action@v3
    with:
      files: coverage/*.lcov
```

That's it, at this point you should have:

1. **Local coverage** tools and task for fast feedback
2. **Smooth and seamless** IDE integration
3. **CI integration** with a nice code coverage provider for working in a team and automatically covering PRs

You can find the **full workflow** in the rust-starter project.

You can also use **the project itself** as a starting point to get all of those benefits **before writing a single line of code**.*Originally published at https://blog.rng0.io* .