THE DARK SECRETS LURKING INSIDE

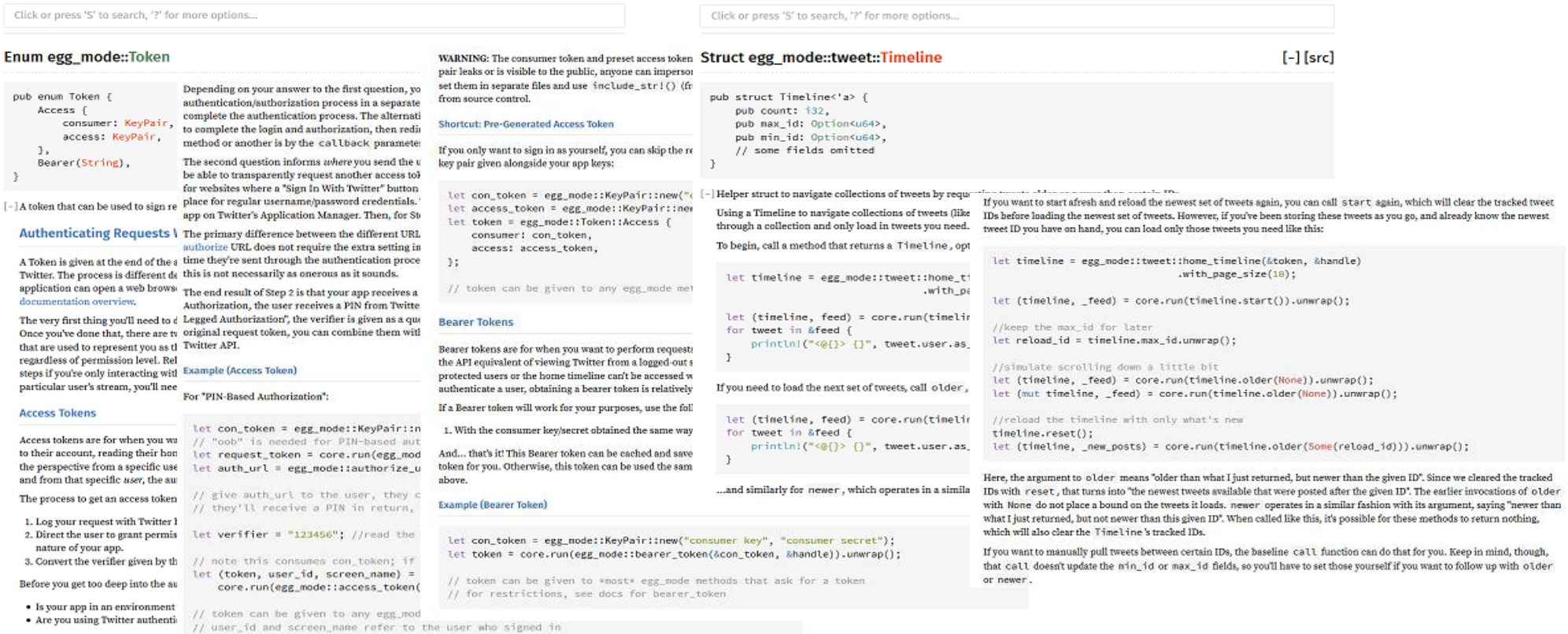
cargo doc

“hey, i just met you...”

- @QuietMisdreavus // “grey” // they/them
- Rustdoc Team (lead), Docs Team
- Sharer of music
- Knitter of hats
- Some code things, I guess

Lyric reference: Carly Rae Jepsen - “Call Me Maybe”

i like docs



@QuietMisdreavus 2018

i like docs… like, a lot

Guide to Rustc Development

The walking tour of rustdoc

Rustdoc actually uses the rustc internals directly. It lives in-tree with the compiler an
library. This chapter is about how it works. (A new implementation is also under way,

Rustdoc is implemented entirely within the crate `librustdoc`. It runs the compiler u
where we have an internal representation of a crate (HIR) and the ability to run some
the types of items. HIR and queries are discussed in the linked chapters.

`librustdoc` performs two major steps after that to render a set of documentation:

- "Clean" the AST into a form that's more suited to creating documentation (and
resistant to churn in the compiler).
- Use this cleaned AST to render a crate's documentation, one page at a time.

Naturally, there's more than just this, and those descriptions simplify out lots of deta
the high-level overview.

(Side note: `librustdoc` is a library crate! The `rustdoc` binary is created using the pr
`src/tools/rustdoc`. Note that literally all that does is call the `main()` that's in this c
though.)

Cheat sheet

- Use `x.py build --stage 1 src/libstd src/tools/rustdoc` to make a useable
can run on other projects.
  - Add `src/libtest` to be able to use `rustdoc --test`.
  - If you've used `rustup toolchain link local /path/to/build/$TARGET/s`
previously, then after the previous build command, `cargo +local doc` wi
- Use `x.py doc --stage 1 src/libstd` to use this rustdoc to generate the stand
docs.
  - The completed docs will be available in `build/$TARGET/doc/std`, though
meant to be used as though you would copy out the `doc` folder to a web
that's where the CSS/JS and landing page are.
- Most of the HTML printing code is in `html/format.rs` and `html/render.rs`. It'
`fmt::Display` implementations and supplementary functions.
- The types that get `Display` impls above are defined in `clean/mod.rs`, right ne
`Clean` trait used to process them out of the rustc HIR.
- The bits specific to using rustdoc as a test harness are in `test.rs`.
- The Markdown renderer is loaded up in `html/markdown.rs`, including function
doctests from a given block of Markdown.
- The tests on rustdoc output are located in `src/test/rustdoc`, where they're ha
test runner of rustbuild and the supplementary script `src/etc/htmldocck.py`.
- Tests on search index generation are located in `src/test/rustdoc-js`, as a ser
files that encode queries on the standard library search index and expected res

From crate to clean

In `core.rs` are two central items: the `DocContext` struct, and the `run_core` functio
where rustdoc calls out to rustc to compile a crate to the point where rustdoc can tal
former is a state container used when crawling through a crate to gather its docume

The main process of crate crawling is done in `clean/mod.rs` through several implem
`Clean` trait defined within. This is a conversion trait, which defines one method:

```
pub trait Clean<T> {
    fn clean(&self, cx: &DocContext) -> T;
}
```

`clean/mod.rs` also defines the types for the "cleaned" AST used later on to render documentation
pages. Each usually accompanies an implementation of `Clean` that takes some AST or HIR type from
rustc and converts it into the appropriate "cleaned" type. "Big" items like modules or associated
items may have some extra processing in its `Clean` implementation, but for the most part these
impls are straightforward conversions. The "entry point" to this module is the
`impl Clean<Crate> for visit_ast::RustdocVisitor`, which is called by

You see, I actually lied a little earlier: There's another AST transformation t
events in `clean/mod.rs`. In `visit_ast.rs` is the type `RustdocVisitor`, wh
`hir::Crate` to get the first intermediate representation, defined in `doctr
to get a few intermediate wrappers around the HIR types and to process v
where `#[doc(inline)]`, `#[doc(no_inline)]`, and `#[doc(hidden)]` are pr
logic for whether a `pub use` should get the full page or a "Reexport" line i

The other major thing that happens in `clean/mod.rs` is the collection of d
`#[doc=""]` attributes into a separate field of the Attributes struct, present
hand-written documentation. This makes it easier to collect this document

The primary output of this process is a `clean::Crate` with a tree of items
publicly-documentable items in the target crate.

Hot potato

Before moving on to the next major step, a few important "passes" occur
These do things like combine the separate "attributes" into a single string
whitespace to make the document easier on the markdown parser, or dro
or deliberately hidden with `#[doc(hidden)]`. These are all implemented in
one file per pass. By default, all of these passes are run on a crate, but the
private/hidden items can be bypassed by passing `--document-private-it`
unlike the previous set of AST transformations, the passes happen on the

(Strictly speaking, you can fine-tune the passes run and even add your own
deprecate that. If you need finer-grain control over these passes, please le

Here is current (as of this writing) list of passes:

- `propagate-doc-cfg` - propagates `#[doc(cfg(...))]` to child items.
- `collapse-docs` concatenates all document attributes into one docu
necessary because each line of a doc comment is given as a separate
combine them into a single string with line breaks between each attr
- `unindent-comments` removes excess indentation on comments in or
it. This is necessary because the convention for writing documentatio
between the `///` or `//!` marker and the text, and stripping that lea
text easier to parse by the Markdown parser. (In the past, the markd
Commonmark- compliant, which caused annoyances with extra whit
be less of an issue today.)
- `strip-priv-imports` strips all private import statements (`use`, `ext
This is necessary because rustdoc will handle public imports by either
documentation to the module or creating a "Reexports" section with
ensures that all of these imports are actually relevant to documentat
- `strip-hidden` and `strip-private` strip all `doc(hidden)` and private
`strip-private` implies `strip-priv-imports`. Basically, the goal is to remove items that are
not relevant for public documentation.

From clean to crate

This is where the "second phase" in rustdoc begins. This phase primarily li
and it all starts with `run()` in `html/render.rs`. This code is responsible fo
`SharedContext`, and `Cache` which are used during rendering, copying out
in every rendered set of documentation (things like the fonts, CSS, and Jav
`html/static/`), creating the search index, and printing out the source cod
beginning the process of rendering all the documentation for the crate.

Several functions implemented directly on `Context` take the `clean::Crat`
between rendering items or recursing on a module's child items. From her
begins, via an enormous `write!()` call in `html/layout.rs`. The parts that
from the items and documentation occurs within a series of `std::fmt::Di`
and functions that pass around a `&mut std::fmt::Formatter`. The top-lev
writes out the page body is the `impl<'a> fmt::Display for Item<'a>` in
switches out to one of several `item_*` functions based on the kind of `Ite`

Depending on what kind of rendering code you're looking for, you'll proba
`html/render.rs` for major items like "what sections should I print for a str
`html/format.rs` for smaller component pieces like "how should I print a w
some other item".

Whenever rustdoc comes across an item that should print hand-written do
calls out to `html/markdown.rs` which interfaces with the Markdown parse
series of types that wrap a string of Markdown, and implement `fmt::Disp`
takes special care to enable certain features like footnotes and tables and
Rust code blocks (via `html/highlight.rs`) before running the Markdown p
function in here (`find_testable_code`) that specifically scans for Rust cod
code can find all the doctests in the crate.

From soup to nuts

(alternate title: "An unbroken thread that stretches from those first `cell`s

It's important to note that the AST cleaning can ask the compiler for inform
`DocContext` contains a `TyCtxt`), but page rendering cannot. The `clean::`
`run_core` is passed outside the compiler context before being handed to
means that a lot of the "supplementary data" that isn't immediately availat
definition, like which trait is the `Deref` trait used by the language, needs t
cleaning, stored in the `DocContext`, and passed along to the `SharedConte`
rendering. This manifests as a bunch of shared state, context variables, an

Also of note is that some items that come from "asking the compiler" don't
`DocContext` - for example, when loading items from a foreign crate, rustdoc will ask about trait

Also of note is that some items that come from "asking the compiler" don't go directly into the
`DocContext` - for example, when loading items from a foreign crate, rustdoc will ask about trait
implementations and generate new `Item`s for the impls based on that information. This goes
directly into the returned `Crate` rather than roundabout through the `DocContext`. This way, these
implementations can be collected alongside the others, right before rendering the HTML.

Other tricks up its sleeve

All this describes the process for generating HTML documentation from a Rust crate, but there are
couple other major modes that rustdoc runs in. It can also be run on a standalone Markdown file, or
it can run doctests on Rust code or standalone Markdown files. For the former, it shortcuts straight
to `html/markdown.rs`, optionally including a mode which inserts a Table of Contents to the output
HTML.

For the latter, rustdoc runs a similar partial-compilation to get relevant documentation in `test.rs`,
but instead of going through the full clean and render process, it runs a much simpler crate walk to
grab just the hand-written documentation. Combined with the aforementioned "
`find_testable_code`" in `html/markdown.rs`, it builds up a collection of tests to run before handing
them off to the libtest test runner. One notable location in `test.rs` is the function `make_test`,
which is where hand-written doctests get transformed into something that can be executed.

Some extra reading about `make_test` can be found here.

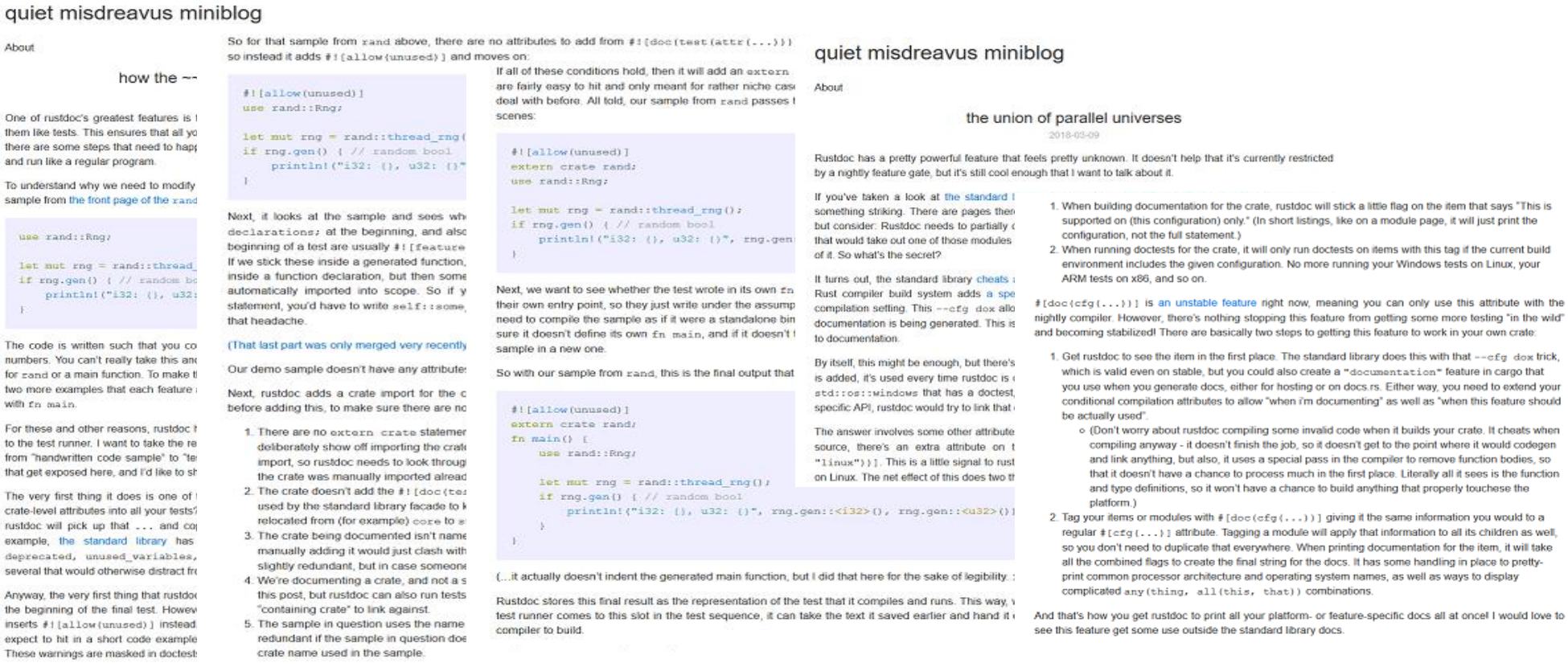
Dotting i's and crossing t's

So that's rustdoc's code in a nutshell, but there's more things in the repo that deal with it. Since we
have the full `compiletest` suite at hand, there's a set of tests in `src/test/rustdoc` that make sure
the final HTML is what we expect in various situations. These tests also use a supplementary script,
`src/etc/htmldocck.py`, that allows it to look through the final HTML using XPath notation to get a
precise look at the output. The full description of all the commands available to rustdoc tests is in
`htmldocck.py`.

In addition, there are separate tests for the search index and rustdoc's ability to query it. The files in
`src/test/rustdoc-js` each contain a different search query and the expected results, broken out
by search tab. These files are processed by a script in `src/tools/rustdoc-js` and the Node.js
runtime. These tests don't have as thorough of a writeup, but a broad example that features results
in all tabs can be found in `basic.js`. The basic idea is that you match a given `QUERY` with a set of
`EXPECTED` results, complete with the full item path of each item.

@QuietMisdreavus 2018

i like docs… like, maybe too much



quiet misdreavus miniblog

About

how the ~~

One of rustdoc's greatest features is
them like tests. This ensures that all yo
there are some steps that need to hap
and run like a regular program.

To understand why we need to modify
sample from the front page of the rand

```
use rand::Rng;

let mut rng = rand::thread_
if rng.gen() { // random bo
    println!("i32: {}, u32:
}
```

The code is written such that you co
numbers. You can't really take this an
for rand or a main function. To make t
two more examples that each feature
with fn main.

For these and other reasons, rustdoc
to the test runner. I want to take the re
from "handwritten code sample" to "te
that get exposed here, and I'd like to sh

The very first thing it does is one of
crate-level attributes into all your tests?
rustdoc will pick up that ... and co
example, the standard library has
deprecated, unused_variables,
several that would otherwise distract fr

Anyway, the very first thing that rustdo
the beginning of the final test. Howev
inserts #![allow(unused)] instead
expect to hit in a short code example
These warnings are masked in doctest

So for that sample from rand above, there are no attributes to add from #![doc(test(attr(...)))
so instead it adds #![allow(unused)] and moves on:

```
#![allow(unused)]
use rand::Rng;

let mut rng = rand::thread_rng(
if rng.gen() { // random bool
    println!("i32: {}, u32: {}"
}
```

Next, it looks at the sample and sees wh
declarations; at the beginning, and also
beginning of a test are usually #![feature
If we stick these inside a generated function,
inside a function declaration, but then some
automatically imported into scope. So if y
statement, you'd have to write self::some
that headache.

(That last part was only merged very recently

Our demo sample doesn't have any attributes

Next, rustdoc adds a crate import for the c
before adding this, to make sure there are no

1. There are no extern crate statemer
   deliberately show off importing the crat
   import, so rustdoc needs to look throug
   the crate was manually imported alread
2. The crate doesn't add the #![doc(tes
   used by the standard library facade to k
   relocated from (for example) core to s
3. The crate being documented isn't name
   manually adding it would just clash with
   slightly redundant, but in case someone
4. We're documenting a crate, and not a s
   this post, but rustdoc can also run tests
   "containing crate" to link against.
5. The sample in question uses the name
   redundant if the sample in question doe
   crate name used in the sample.

If all of these conditions hold, then it will add an extern
are fairly easy to hit and only meant for rather niche cas
deal with before. All told, our sample from rand passes t
scenes:

```
#![allow(unused)]
extern crate rand;
use rand::Rng;

let mut rng = rand::thread_rng();
if rng.gen() { // random bool
    println!("i32: {}, u32: {}", rng.gen
}
```

Next, we want to see whether the test wrote in its own fn
their own entry point, so they just write under the assump
need to compile the sample as if it were a standalone bin
sure it doesn't define its own fn main, and if it doesn't t
sample in a new one.

So with our sample from rand, this is the final output that

```
#![allow(unused)]
extern crate rand;
fn main() {
    use rand::Rng;

    let mut rng = rand::thread_rng();
    if rng.gen() { // random bool
        println!("i32: {}, u32: {}", rng.gen::<i32>(), rng.gen::<u32>()
    }
}
```

(…it actually doesn't indent the generated main function, but I did that here for the sake of legibility. :

Rustdoc stores this final result as the representation of the test that it compiles and runs. This way,
test runner comes to this slot in the test sequence, it can take the text it saved earlier and hand it
compiler to build.

quiet misdreavus miniblog

About

the union of parallel universes

2018-03-09

Rustdoc has a pretty powerful feature that feels pretty unknown. It doesn't help that it's currently restricted
by a nightly feature gate, but it's still cool enough that I want to talk about it.

If you've taken a look at the standard l
something striking. There are pages ther
but consider: Rustdoc needs to partially
that would take out one of those modules
of it. So what's the secret?

It turns out, the standard library cheats
Rust compiler build system adds a spe
compilation setting. This --cfg dox allo
documentation is being generated. This is
to documentation.

By itself, this might be enough, but there's
is added, it's used every time rustdoc is
std::os::windows that has a doctest,
specific API, rustdoc would try to link that

The answer involves some other attribute
source, there's an extra attribute on t
"linux")]. This is a little signal to rust
on Linux. The net effect of this does two th

1. When building documentation for the crate, rustdoc will stick a little flag on the item that says "This is supported on (this configuration) only." (In short listings, like on a module page, it will just print the configuration, not the full statement.)
2. When running doctests for the crate, it will only run doctests on items with this tag if the current build environment includes the given configuration. No more running your Windows tests on Linux, your ARM tests on x86, and so on.

#[doc(cfg(...))] is an unstable feature right now, meaning you can only use this attribute with the nightly compiler. However, there's nothing stopping this feature from getting some more testing "in the wild" and becoming stabilized! There are basically two steps to getting this feature to work in your own crate:

1. Get rustdoc to see the item in the first place. The standard library does this with that --cfg dox trick, which is valid even on stable, but you could also create a "documentation" feature in cargo that you use when you generate docs, either for hosting or on docs.rs. Either way, you need to extend your conditional compilation attributes to allow "when i'm documenting" as well as "when this feature should be actually used".
   - (Don't worry about rustdoc compiling some invalid code when it builds your crate. It cheats when compiling anyway - it doesn't finish the job, so it doesn't get to the point where it would codegen and link anything, but also, it uses a special pass in the compiler to remove function bodies, so that it doesn't have a chance to process much in the first place. Literally all it sees is the function and type definitions, so it won't have a chance to build anything that properly touchese the platform.)
2. Tag your items or modules with #[doc(cfg(...))] giving it the same information you would to a regular #[cfg(...)] attribute. Tagging a module will apply that information to all its children as well, so you don't need to duplicate that everywhere. When printing documentation for the item, it will take all the combined flags to create the final string for the docs. It has some handling in place to pretty-print common processor architecture and operating system names, as well as ways to display complicated any(thing, all(this, that)) combinations.

And that's how you get rustdoc to print all your platform- or feature-specific docs all at once! I would love to see this feature get some use outside the standard library docs.

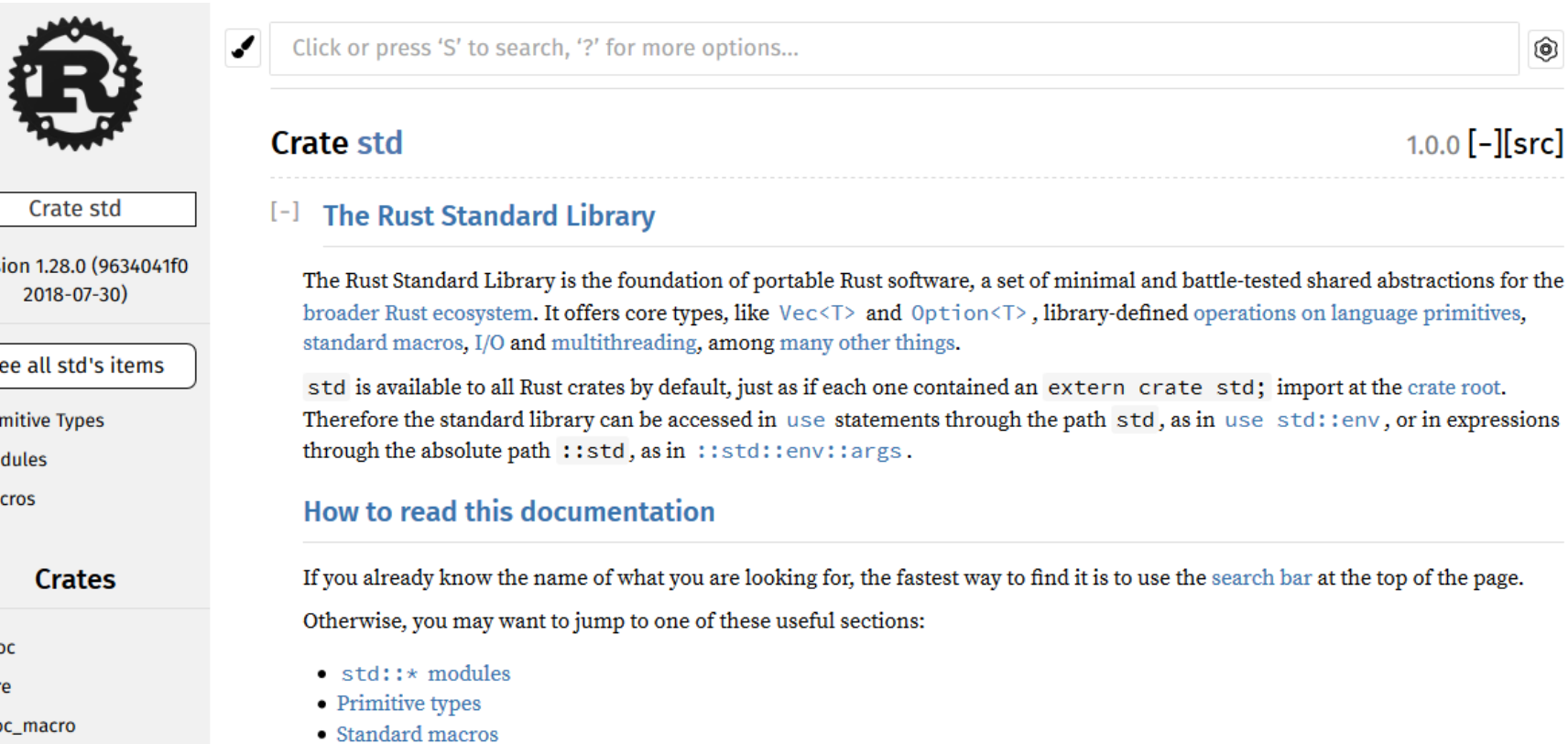
@QuietMisdreavus 2018

“...and this is crazy...”

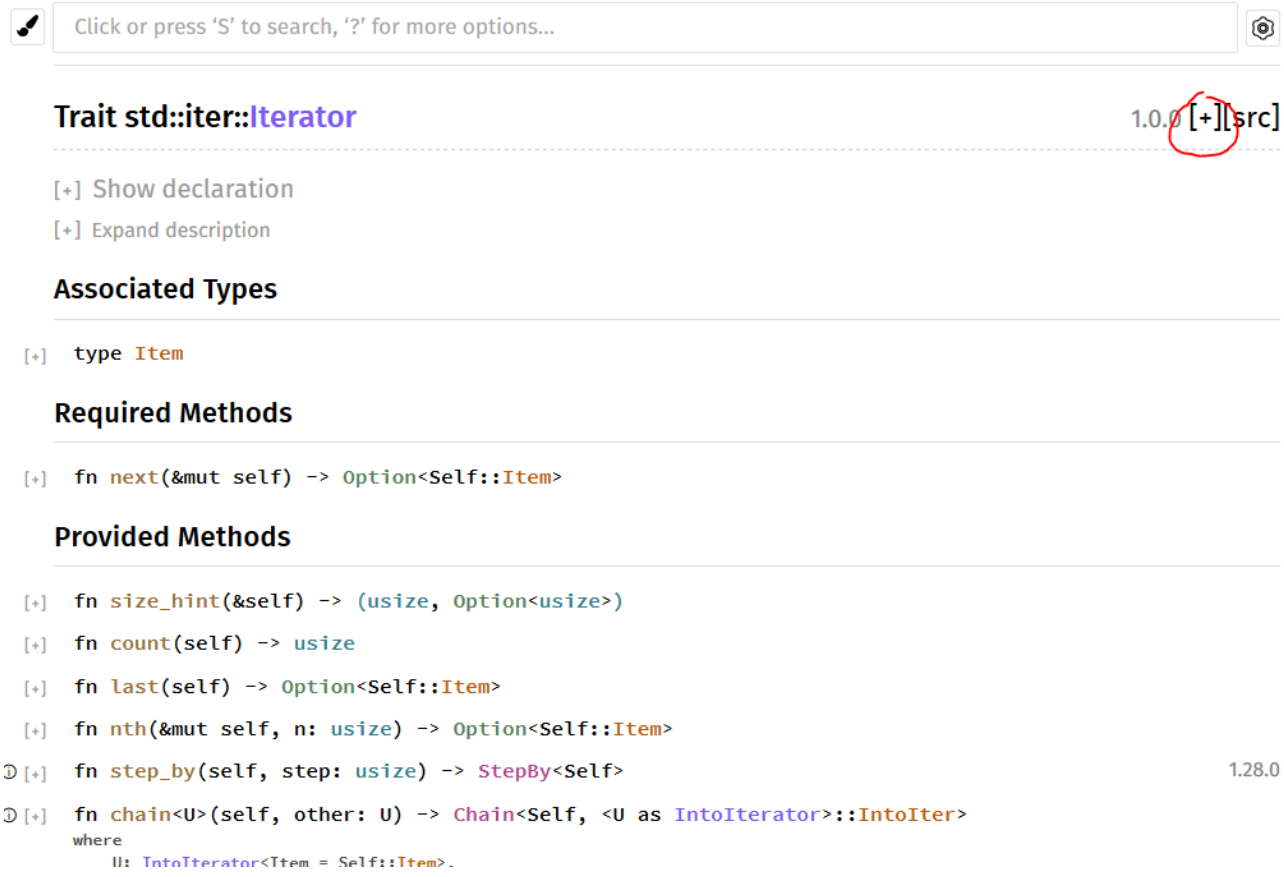
- Become a Docs Power User (tm)
- Deny lints on your doctests
- Document all your platforms at once
- Cover your docs in ponies
- Peek under the hood

Lyric reference: Carly Rae Jepsen - “Call Me Maybe”

Rustdoc output



Want to cut to the chase?



Click this link to fold (or unfold) everything on the page!

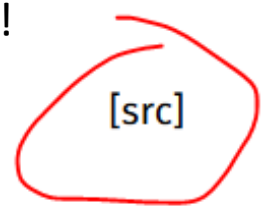
Want to see how it's done?

```
[-] impl<K, V, S> Extend<(K, V)> for HashMap<K, V, S>                    [src]
    where
        K: Eq + Hash,
        S: BuildHasher,

[-] fn extend<T: IntoIterator<Item = (K, V)>>(&mut self, iter: T)        [src]
```

Extends a collection with the contents of an iterator. Read more

```
[-] impl<'a, K, V, S> Extend<(&'a K, &'a V)> for HashMap<K, V, S>  1.4.0 [src]
    where
        K: Eq + Hash + Copy,
```

Check the source!



Check the source!

```
2523  #[stable(feature = "rust1", since = "1.0.0")]
2524  impl<K, V, S> Extend<(K, V)> for HashMap<K, V, S>
2525      where K: Eq + Hash,
2526            S: BuildHasher
2527  {
2528      fn extend<T: IntoIterator<Item = (K, V)>>(&mut self, iter: T) {
2529          // Keys may be already present or show multiple times in the iterator.
2530          // Reserve the entire hint lower bound if the map is empty.
2531          // Otherwise reserve half the hint (rounded up), so the map
2532          // will only resize twice in the worst case.
2533          let iter = iter.into_iter();
2534          let reserve = if self.is_empty() {
2535              iter.size_hint().0
2536          } else {
2537              (iter.size_hint().0 + 1) / 2
2538          };
2539          self.reserve(reserve);
2540          for (k, v) in iter {
2541              self.insert(k, v);
2542          }
2543      }
2544  }
```

The crate's source code is shipped and highlighted alongside all its docs!

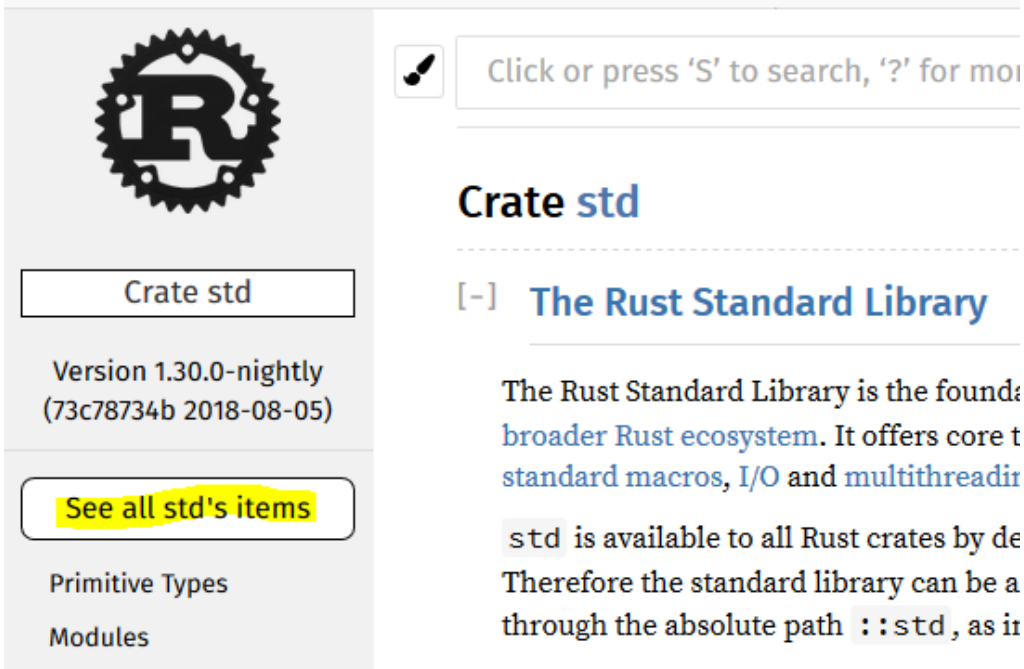
Color-coded links!

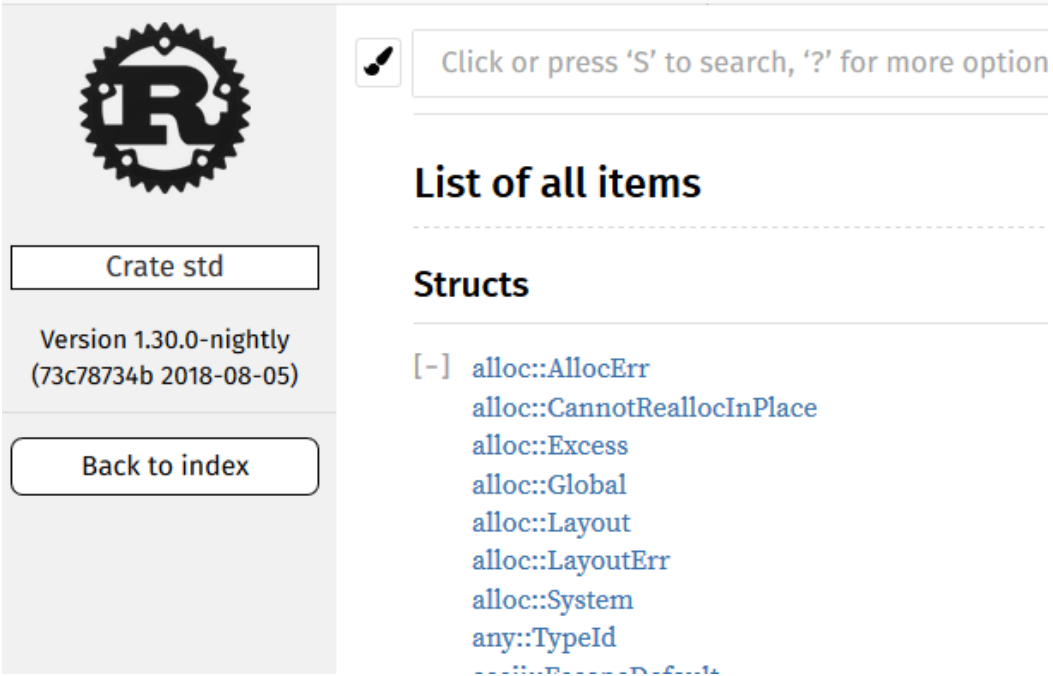
```
[-] impl<K, V, S> HashMap<K, V, S>                                          [src]
    where
        K: Eq + Hash,
        S: BuildHasher,

  [+] pub fn with_hasher(hash_builder: S) -> HashMap<K, V, S>          1.7.0 [src]

  [+] pub fn with_capacity_and_hasher(                                 1.7.0 [src]
          capacity: usize,
          hash_builder: S
      ) -> HashMap<K, V, S>

ⓘ [+] pub fn hasher(&self) -> &S                                       1.9.0 [src]

  [+] pub fn capacity(&self) -> usize                                        [src]

  [+] pub fn reserve(&mut self, additional: usize)                           [src]

  [+] pub fn try_reserve(                                                    [src]
          &mut self,
          additional: usize
      ) -> Result<(), CollectionAllocErr>

    ▸ This is a nightly-only experimental API. (try_reserve #48043)

  [+] pub fn shrink_to_fit(&mut self)                                        [src]
```

All types in rustdoc's generated signatures are links to their docs!

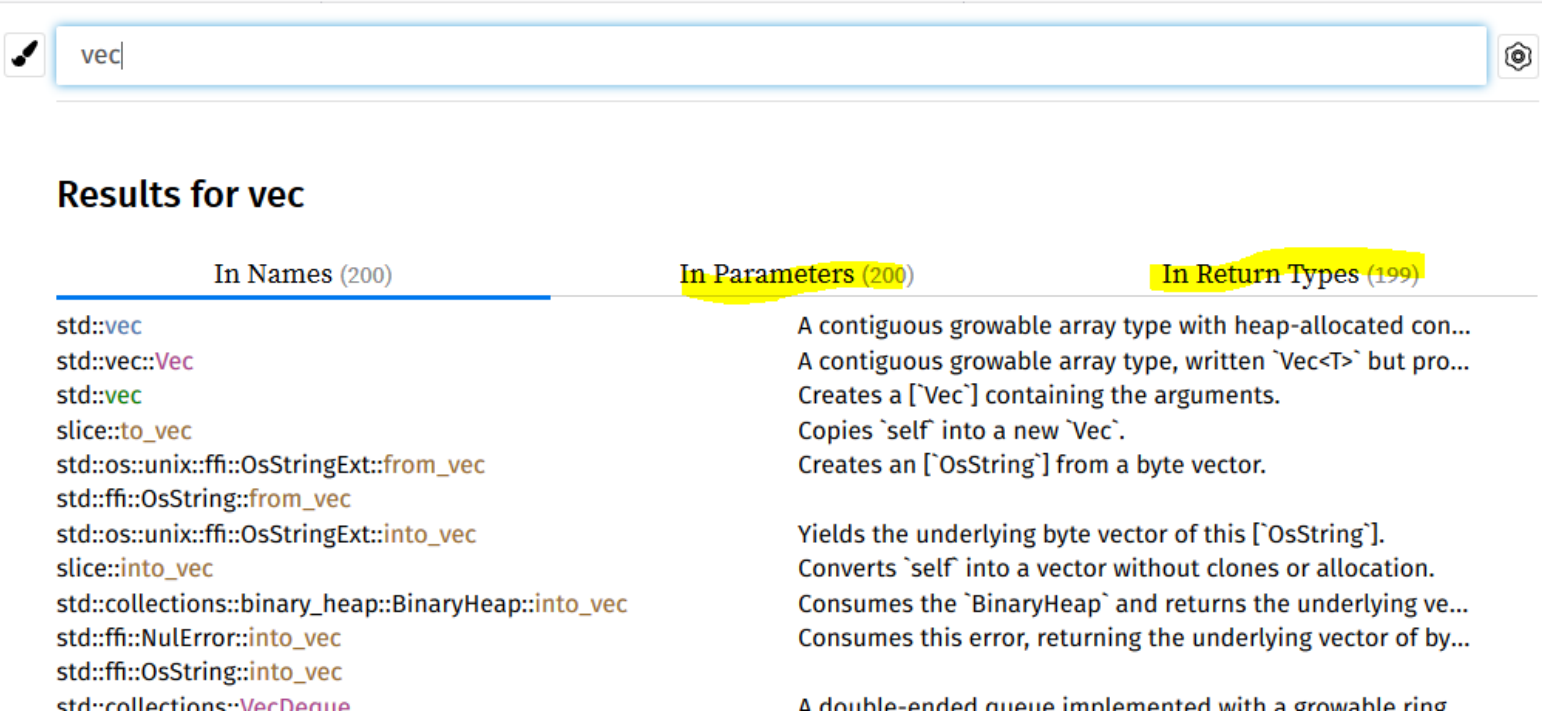
- Magenta: Structs
- Purple: Traits
- Green: Enums
- Blue: Primitives
- Tan: Functions

Want everything on one page?



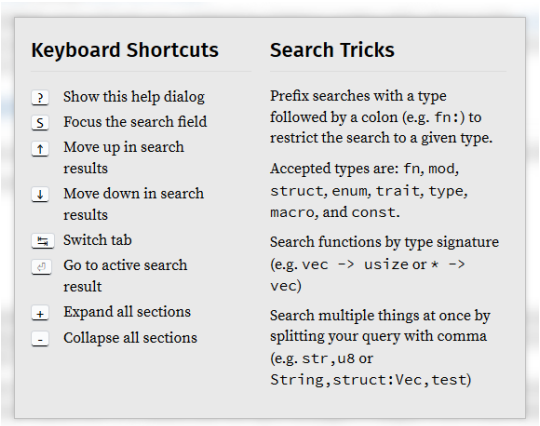


Curious how to use a type?

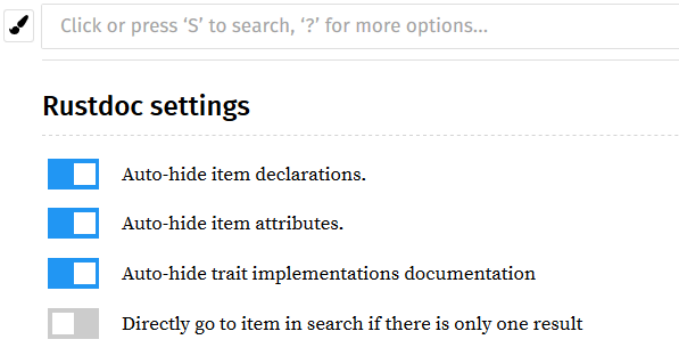


Tabs in the search results can show where a type is used by or returned from a function!

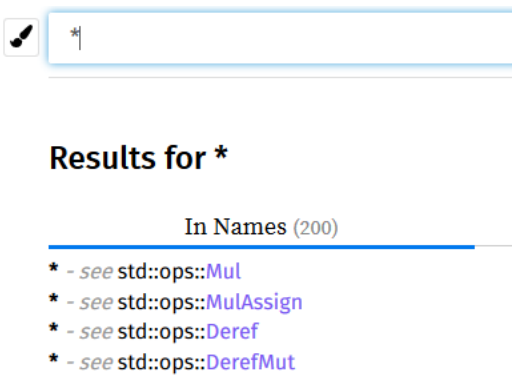
And more!



Press “?” for keyboard shortcuts and search hints!



Click the gear by the search box for doc settings!



Search operators in the standard library to see their traits!

introducing rustdoc

The tool behind `cargo doc`!

cargo build	rustc
cargo doc	rustdoc

Yo dawg, we heard you like code, so we put code in your docs, so you can read code while you read about code

```rust
/// This is some struct, here.
///
/// ```
/// let my_struct = my_project::SomeStruct;
/// println!("hey, here's some code in your code");
/// ```
pub struct SomeStruct;
```

Put some code samples into your docs...

...and rustdoc can run them with your tests!

```
[misdreavus@tonberry my_project]$ cargo test --doc
   Compiling my_project v0.1.0 (file:///home/misdreavus/git/my_project)
    Finished dev [unoptimized + debuginfo] target(s) in 2.47s
   Doc-tests my_project

running 1 test
test src/lib.rs - SomeStruct (line 3) ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Meme reference: “Xzibit Yo Dawg”

The journey of a doctest

```
let my_struct = my_project::SomeStruct;
println!("hey, here's some code in your code!");
```

Rustdoc wants to compile your doctest as an executable…

…so it wraps your code in a main function…

```
fn main() {
    let my_struct = my_project::SomeStruct;
    println!("hey, here's some code in your code!");
}
```

```
extern crate my_project;
fn main() {
    let my_struct = my_project::SomeStruct;
    println!("hey, here's some code in your code!");
}
```

…and adds in a reference to your crate!

Guiding doctests on their journey

```
#![feature(sick_rad)]

#[macro_use] extern crate my_project;

let my_struct = my_project::SomeStruct;
println!("hey, here's some code in your code!");
```

Not everything goes inside `fn main()`, though! Let's extend that test some…

Crate attributes and `extern crate` statements are preserved outside the generated `main`

```
#![feature(sick_rad)]

#[macro_use] extern crate my_project;

fn main() {
    let my_struct = my_project::SomeStruct;
    println!("hey, here's some code in your code!");
}
```

Doctests and Lints

```
#![allow(unused)]
extern crate my_project;
fn main() {
    let my_struct = my_project::SomeStruct;
    println!("hey, here's some code in your code!");
}
```

By default, doctests also get `#![allow(unused)]`

```
#![doc(test(attr(deny(warnings))))]

/// This is some struct, here.
///
/// ```
/// let my_struct = my_project::SomeStruct;
/// println!("hey, here's some code in your code");
/// ```
pub struct SomeStruct;
```

But you can change that! Add this attribute to your crate…

Doctests and Lints

```
#![deny(warnings)]
extern crate my_project;
fn main() {
    let my_struct = my_project::SomeStruct;
    println!("hey, here's some code in your code!");
}
```

…and change that attribute with whatever you want!

Doctests and Lints

```
[misdreavus@tonberry my_project]$ cargo test --doc
   Compiling my_project v0.1.0 (file:///home/misdreavus/git/my_project)
    Finished dev [unoptimized + debuginfo] target(s) in 1.91s
   Doc-tests my_project

running 1 test
test src/lib.rs - SomeStruct (line 5) ... FAILED

failures:

---- src/lib.rs - SomeStruct (line 5) stdout ----
        error: unused variable: `my_struct`
 --> src/lib.rs:6:5
  |
4 | let my_struct = my_project::SomeStruct;
  |     ^^^^^^^^^ help: consider using `_my_struct` instead
  |
note: lint level defined here
 --> src/lib.rs:3:9
  |
1 | #![deny(warnings)]
  |         ^^^^^^^^
  = note: #[deny(unused_variables)] implied by #[deny(warnings)]

thread 'src/lib.rs - SomeStruct (line 5)' panicked at 'couldn't compile the test', librustdoc/test.rs:321:13
note: Run with `RUST_BACKTRACE=1` for a backtrace.


failures:
    src/lib.rs - SomeStruct (line 5)

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out

error: test failed, to rerun pass '--doc'
```

hecking docs, how do they work

Doc comments are special!

```
//! Hey, here are some module docs!

/// (written on a spider's web) Some Struct
///
/// Wow, that must be some struct! Gotta take care of that one.
pub struct SomeStruct;
```

```
#![doc = " Hey, here are some module docs!"]

#[doc = " (written on a spider's web) Some Struct"]
#[doc = ""]
#[doc = " Wow, that must be some struct! Gotta take care of that one."]
pub struct SomeStruct;
```

Rustdoc compiles your crate to scrape out these attributes

Lyric reference: Insane Clown Posse - “Miracles”

The #[doc] attribute does a lot!

- `#![doc(html_root_url)]`
- `#![doc(test(attr))]`
- `#[doc(inline)]`, `#[doc(no_inline)]`
- `#[doc(hidden)]`
- `#[doc(include)]`
- `#[doc(cfg)]`

`#[doc(cfg)]`: All your platforms at once

Module std::os

[-] OS-specific functionality.

Modules

linux	[Linux] Linux-specific definitions
raw	Platform-specific types, as defined by C.
unix	[Unix] Experimental extensions to `std` for Unix platforms.
windows	[Windows] Platform-specific extensions to `std` for Windows.

Trait std::os::windows::ffi::OsStringExt

[-]

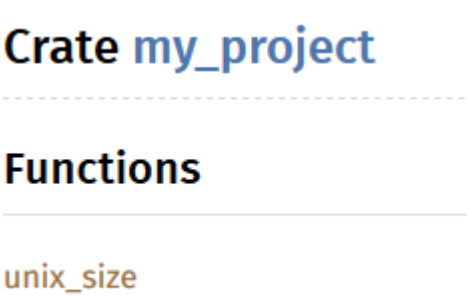
```
pub trait OsStringExt {
    fn from_wide(wide: &[u16]) -> Self;
}
```

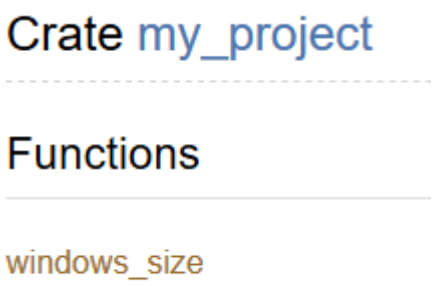
This is supported on **Windows** only.

[-] Windows-specific extensions to `OsString`.

Conditional compilation vs. your docs

```rust
use std::io;
use std::fs;

#[cfg(unix)]
use std::os::unix::fs::MetadataExt;

#[cfg(unix)]
pub fn unix_size() -> io::Result<u64> {
    let meta = fs::metadata("foo.txt")?;
    Ok(meta.size())
}

#[cfg(windows)]
use std::os::windows::fs::MetadataExt;

#[cfg(windows)]
pub fn windows_size() -> io::Result<u64> {
    let meta = fs::metadata("foo.txt")?;
    Ok(meta.file_size())
}
```





Rust handles conditional compilation before rustdoc can make your docs!

Forcing rustdoc to see the items

```rust
#![feature(doc_cfg)]

use std::io;
use std::fs;

#[cfg(unix)]
use std::os::unix::fs::MetadataExt;

#[cfg(any(unix, rustdoc))]
pub fn unix_size() -> io::Result<u64> {
    let meta = fs::metadata("foo.txt")?;
    Ok(meta.size())
}

#[cfg(windows)]
use std::os::windows::fs::MetadataExt;

#[cfg(any(windows, rustdoc))]
pub fn windows_size() -> io::Result<u64> {
    let meta = fs::metadata("foo.txt")?;
    Ok(meta.file_size())
}
```

Crate my_project

Functions

unix_size

windows_size

By compiling them in whenever rustdoc is running, we can show everything! But…

(Note: #[cfg(rustdoc)] is not available yet! There's an open PR for it!)

```
[misdreavus@tonberry my_project]$ cargo test --doc
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
   Doc-tests my_project

running 2 tests
test src/lib.rs - windows_size (line 23) ... FAILED
test src/lib.rs - unix_size (line 9) ... ok

failures:

---- src/lib.rs - windows_size (line 23) stdout ----
        error[E0425]: cannot find function `windows_size` in module `my_project`
 --> src/lib.rs:24:45
  |
4 | println!("it's {} bytes long.", my_project::windows_size().unwrap());
  |                                             ^^^^^^^^^^^^ not found in `my_project`

thread 'src/lib.rs - windows_size (line 23)' panicked at 'couldn't compile the test', librustdoc/test.rs:321:13
note: Run with `RUST_BACKTRACE=1` for a backtrace.


failures:
    src/lib.rs - windows_size (line 23)

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out

error: test failed, to rerun pass '--doc'
```

Just making rustdoc see the item means it will try to run its doctests on the wrong platforms!

Enter #[doc(cfg)]

```
/// Returns the size of the file `foo.txt`.
///
/// ```
/// println!("it's {} bytes long.", my_project::unix_size().unwrap());
/// ```
#[doc(cfg(unix))]
#[cfg(any(unix, rustdoc))]
pub fn unix_size() -> io::Result<u64> {
    let meta = fs::metadata("foo.txt")?;
    Ok(meta.size())
}
```

Telling rustdoc specifically about the platform…

…means it knows when to run (and to ignore) the doctests!

```
[misdreavus@tonberry my_project]$ cargo +nightly test --doc
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
   Doc-tests my_project

running 1 test
test src/lib.rs - unix_size (line 11) ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Bonus!

Now rustdoc can tell your users about the platform in the docs for you!

Crate my_project

Functions

unix_size	[Unix] Returns the size of the file `foo.txt`.
windows_size	[Windows] Returns the size of the file `foo.txt`.

Function my_project::windows_size

```
pub fn windows_size() -> Result<u64>
```

This is supported on **Windows** only.

[-] Returns the size of the file `foo.txt`.

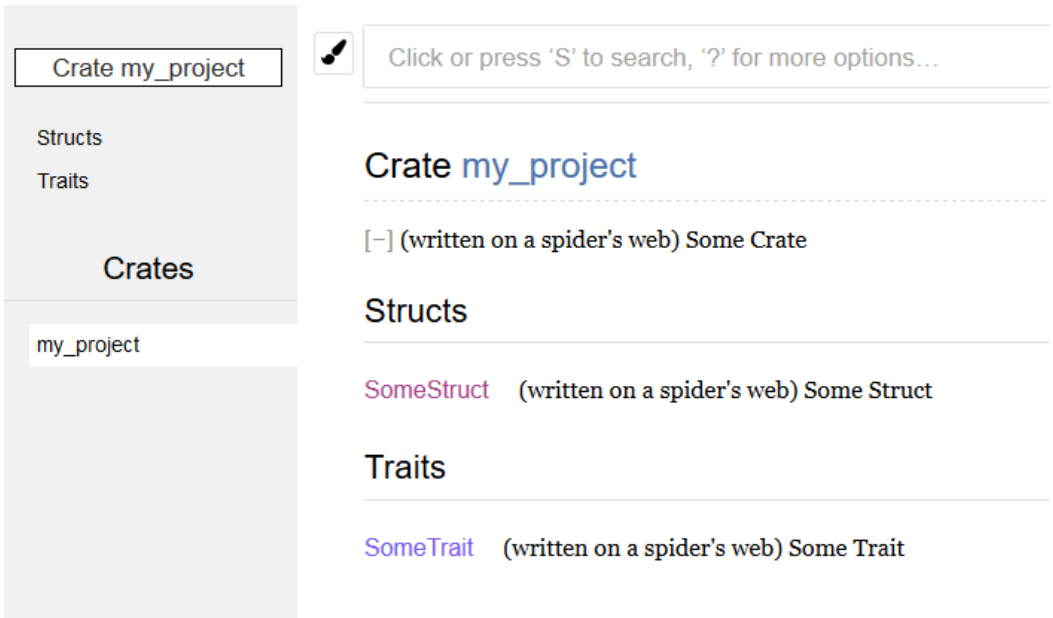
```
println!("it's {} bytes long.", my_project::windows_size().unwrap());
```

CLI Flags

- Compile flags
  - --cfg, --extern, -C, --target, --edition
- Content modification
  - --html-in-header, --html-before-content, --html-after-content
  - --document-private-items, --sort-modules-by-appearance
- Process modification
  - --passes, --no-defaults, --resource-suffix, --disable-minification

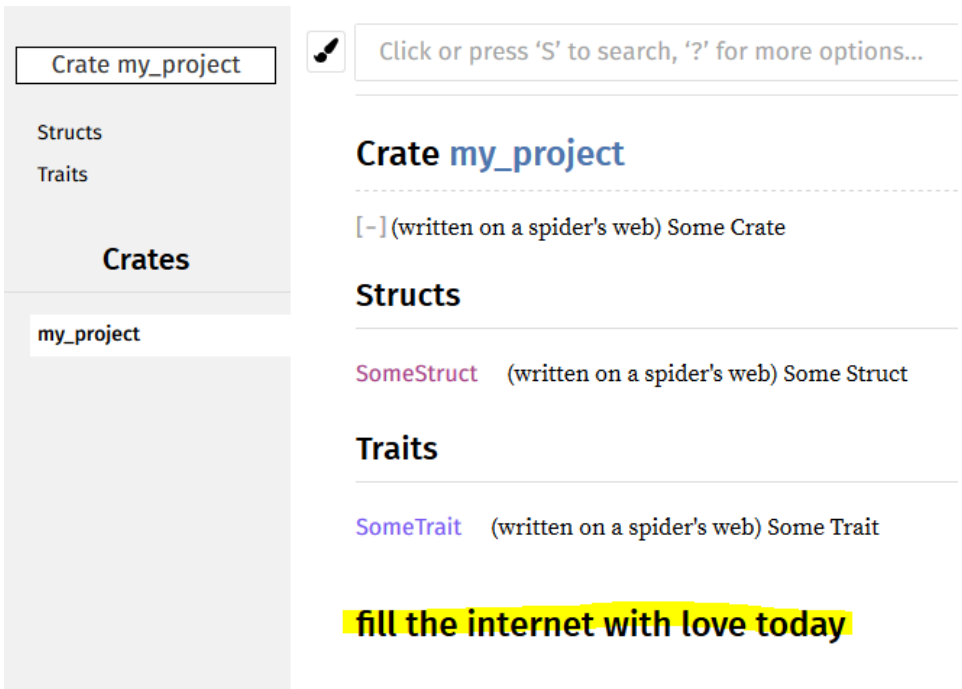
Splicing new content into your docs

```rust
//! (written on a spider's web) Some Crate

/// (written on a spider's web) Some Struct
pub struct SomeStruct;

/// (written on a spider's web) Some Trait
pub trait SomeTrait {}
```



…but add a flag to rustdoc…

```
[misdreavus@tonberry my_project]$ cargo rustdoc -- --html-after-content message.html
 Documenting my_project v0.1.0 (file:///home/misdreavus/git/my_project)
    Finished dev [unoptimized + debuginfo] target(s) in 2.91s
```




```
<section style="margin-left: 230px">
    <h1>fill the internet with love today</h1>
</section>
```

Want some KaTeX in your docs?

Module curve_models

Re-exports

Structs

curve25519_dalek

Modules

backend

constants

curve_models

edwards

field

Click or press 'S' to search, '?' for more options...

Module curve25519_dalek::curve_models [-][src]

[-] Internal curve representations which are not part of the public API.

Curve representations

Internally, we use several different models for the curve. Here is a sketch of the relationship between the models, following a post by Ben Smith on the `moderncrypto` mailing list. This is also briefly discussed in section 2.5 of *Montgomery curves and their arithmetic* by Costello and Smith.

Begin with the affine equation for the curve,

$$-x^2 + y^2 = 1 + dx^2y^2.$$

Next, pass to the projective closure $\mathbb{P}^1 \times \mathbb{P}^1$ by setting $x = X/Z$, $y = Y/T$. Clearing denominators gives the model

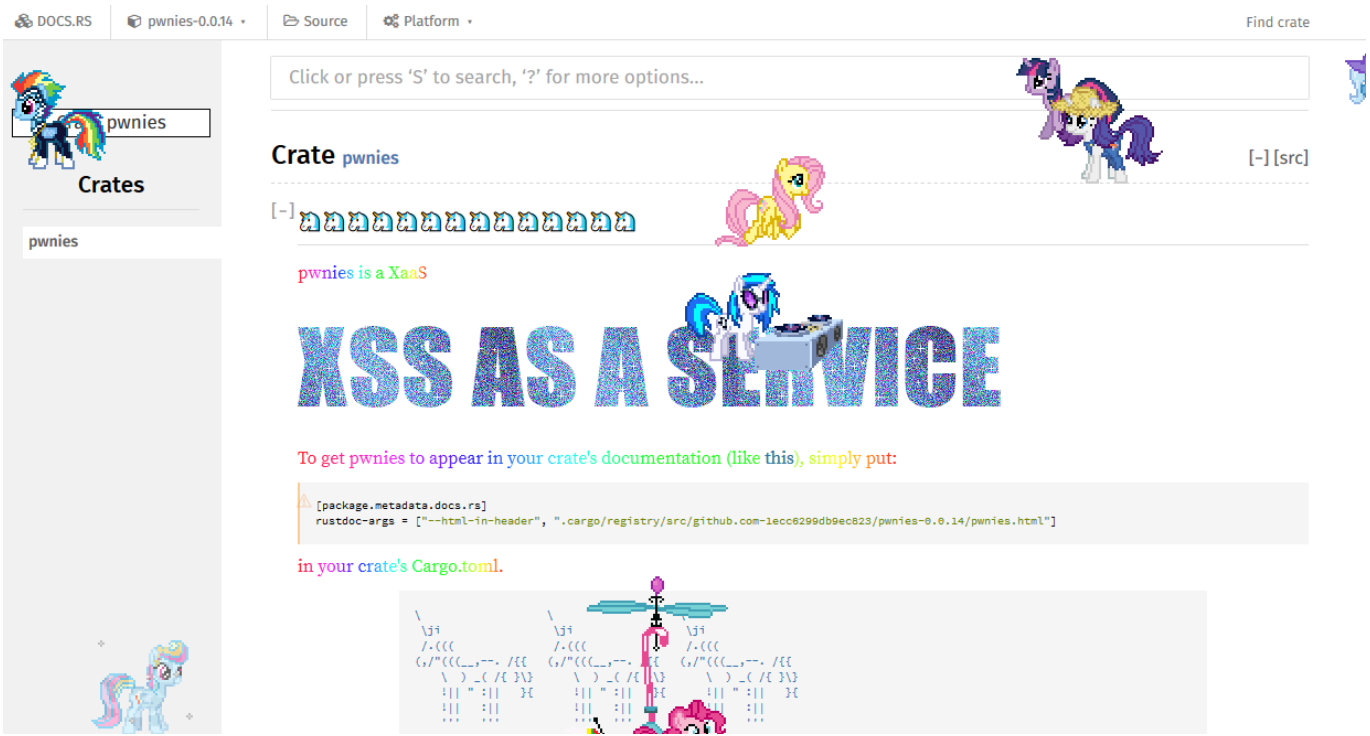
$$-X^2T^2 + Y^2Z^2 = Z^2T^2 + dX^2Y^2.$$

In `curve25519-dalek`, this is represented as the `CompletedPoint` struct. To map from $\mathbb{P}^1 \times \mathbb{P}^1$, a product of two lines, to $\mathbb{P}^3$, we use the Segre embedding

@QuietMisdreavus 2018

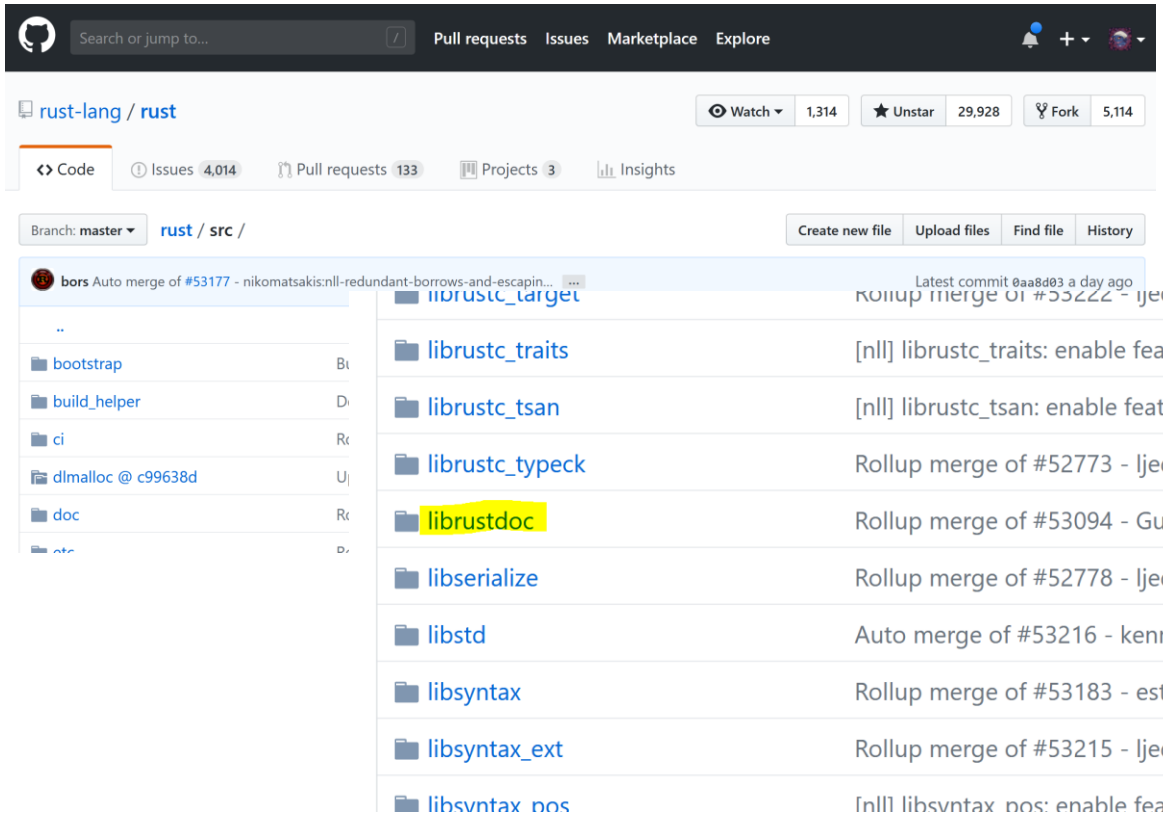
No? How about some ponies?

`https://docs.rs/pwnies`



@QuietMisdreavus 2018

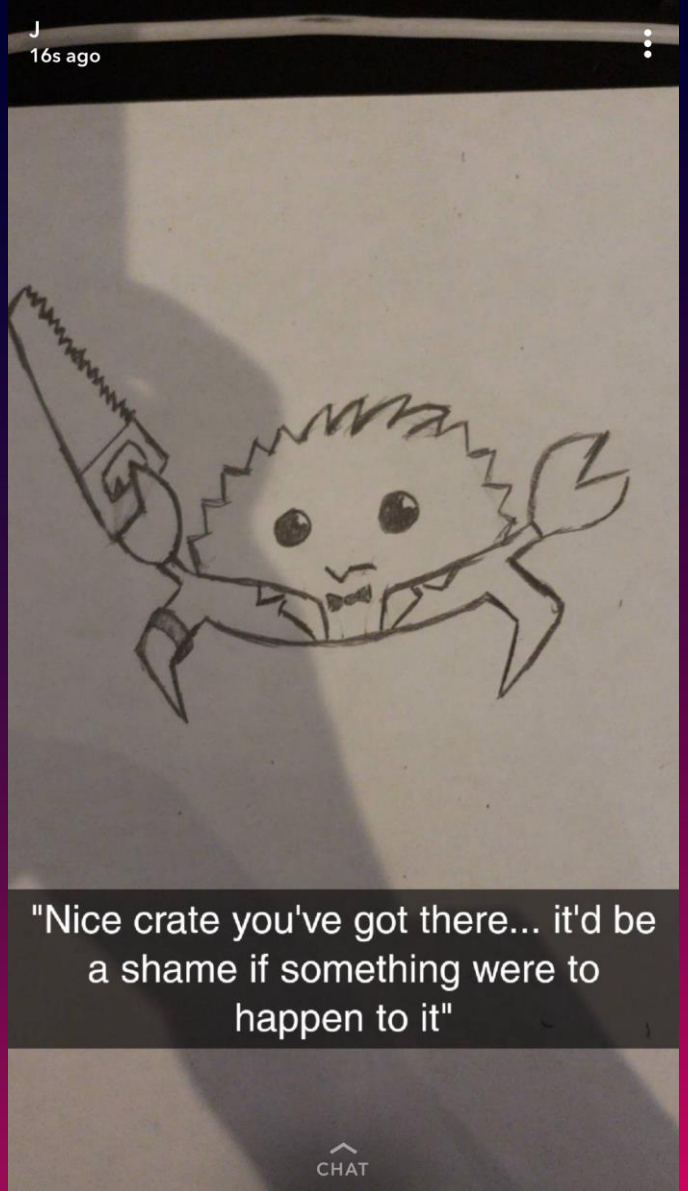
A peek behind the curtain



Rustdoc's code lives in the main "rust-lang/rust" repo, right next to the compiler and standard library!

Data gathering practices

- Asking the compiler nicely for what we want
- Going around the compiler's back to get what we want
- Breaking the compiler to get what we want
- Breaking the compiler's friends to get what we want



Thanks @DebugSteven!

Documentation flow (jargon-filled version)

- Source code is first handed directly to the compiler
- After macro expansion, the name resolver is saved to handle “intra-doc links” later
- After crate analysis, while the TyCtxt is still active, scan the HIR to collect all items in the crate
- “Clean” up all these items so we can have an AST more suited to rustdoc’s purposes

Documentation flow (jargon-filled version)

- Run the cleaned AST through several “passes” to strip out private items, massage doc comments, and otherwise process the crate for later doc generation
    - The TyCtxt is dropped here, leaving the compiler context
- Scan through the crate again to collect all the trait impls, gather/highlight source code, generate search index
- Run through the crate one last time to generate a file for each item and module

Highlights of rustdoc internals

These “Auto Trait Implementations” don’t come directly from the code, so rustdoc has to make them up on the spot

Auto Trait Implementations

```
impl<T> Send for Vec<T>
where
    T: Send,

impl<T> Sync for Vec<T>
where
    T: Sync,
```

Highlights of rustdoc internals

```
34  pub fn render<T: fmt::Display, S: fmt::Display>(
35      dst: &mut dyn io::Write, layout: &Layout, page: &Page, sidebar: &S, t: &T,
36      css_file_extension: bool, themes: &[PathBuf])
37      -> io::Result<()>
38  {
39      write!(dst,
40  "<!DOCTYPE html>\
41  <html lang=\"en\">\
42  <head>\
43      <meta charset=\"utf-8\">\
44      <meta name=\"viewport\" content=\"width=device-width, initial-scale=1.0\">\
45      <meta name=\"generator\" content=\"rustdoc\">\
46      <meta name=\"description\" content=\"{description}\">\
47      <meta name=\"keywords\" content=\"{keywords}\">\
48      <title>{title}</title>\
49      <link rel=\"stylesheet\" type=\"text/css\" href=\"{root_path}normalize{suffix}.css\">\
50      <link rel=\"stylesheet\" type=\"text/css\" href=\"{root_path}rustdoc{suffix}.css\" \
51            id=\"mainThemeStyle\">\
52      {themes}\
53      <link rel=\"stylesheet\" type=\"text/css\" href=\"{root_path}dark{suffix}.css\">\
54      <link rel=\"stylesheet\" type=\"text/css\" href=\"{root_path}light{suffix}.css\" \
55            id=\"themeStyle\">\
56      <script src=\"{root_path}storage{suffix}.js\"></script>\
57      {css_extension}\
58      {favicon}\
59      {in_header}\
60  </head>\
61  <body class=\"rustdoc {css_class}\">\
```

Rustdoc “templating engine” is a massive `write!()` call and a series of `Display` impls

Highlights of rustdoc internals

```
// @has structfields/Foo.t.html
// @has - struct.Foo.html
// @has structfields/struct.Foo.html
pub struct Foo {
    // @has - //pre "pub a: ()"
    pub a: (),
    // @has - //pre "// some fields omitted"
    // @!has - //pre "b: ()"
    b: (),
    // @!has - //pre "c: usize"
    #[doc(hidden)]
    c: usize,
    // @has - //pre "pub d: usize"
    pub d: usize,
}
```

Rustdoc has its own test suite, to make sure we output files and their content correctly

The Rustdoc Team

@GuillaumeGomez

@QuietMisdreavus

@steveklabnik

@ollie27

@onur

“...but here’s my number”

- @QuietMisdreavus
- quietmisdreavus.net
- “misdreavus” on Mozilla IRC
- Fill your world with love today

Lyric reference: Carly Rae Jepsen - “Call Me Maybe”