

*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

i like docs

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

Enum egg_mode::Token

```
pub enum Token {
    Access {
        consumer: KeyPair,
        access: KeyPair,
    },
    Bearer(String),
}
```

[~] A token that can be used to sign re

Authenticating Requests

A Token is given at the end of the s
Twitter. The process is different d
application can open a web brows
[documentation overview](#).

The very first thing you'll need to d
Once you've done that, there are tv
that are used to represent you as t
regardless of permission level. Rel
steps if you're only interacting wit
particular user's stream, you'll need

Access Tokens

Access tokens are for when you wa
to their account, reading their hon
the perspective from a specific use
and from that specific user, the au

The process to get an access token

1. Log your request with Twitter I
2. Direct the user to grant permis
nature of your app.
3. Convert the verifier given by th

Before you get too deep into the au

- Is your app in an environment
- Are you using Twitter authenti

Depending on your answer to the first question, you
authentication/authorization process in a separate
complete the authentication process. The alternati
to complete the login and authorization, then redi
method or another is by the `callback` paramete

The second question informs *where* you send the u
be able to transparently request another access toI
for websites where a "Sign In With Twitter" button
place for regular username/password credentials.
app on Twitter's Application Manager. Then, for St

The primary difference between the different URL
`authorize` URL does not require the extra setting in
time they're sent through the authentication proce
this is not necessarily as onerous as it sounds.

The end result of Step 2 is that your app receives a
Authorization, the user receives a PIN from Twitte
Legged Authorization", the verifier is given as a qu
original request token, you can combine them with
Twitter API.

Example (Access Token)

For "PIN-Based Authorization":

```
let con_token = egg_mode::KeyPair::n
// "oob" is needed for PIN-based aut
let request_token = core.run(egg_mod
let auth_url = egg_mode::authorize_u

// give auth_url to the user, they c
// they'll receive a PIN in return,

let verifier = "123456"; //read the

// note this consumes con_token; if
let (token, user_id, screen_name) =
    core.run(egg_mode::access_token(

// token can be given to any egg_mod
// user_id and screen_name refer to the user who signed in
```

WARNING: The consumer token and preset access token
pair leaks or is visible to the public, anyone can imperson
set them in separate files and use `include_str!()` (fr
from source control.

Shortcut: Pre-Generated Access Token

If you only want to sign in as yourself, you can skip the re
key pair given alongside your app keys:

```
let con_token = egg_mode::KeyPair::new(
let access_token = egg_mode::KeyPair::new(
let token = egg_mode::Token::Access {
    consumer: con_token,
    access: access_token,
};

// token can be given to any egg_mode met
```

Bearer Tokens

Bearer tokens are for when you want to perform request
the API equivalent of viewing Twitter from a logged-out s
protected users or the home timeline can't be accessed w
authenticate a user, obtaining a bearer token is relatively

If a Bearer token will work for your purposes, use the fol

1. With the consumer key/secret obtained the same way

And... that's it! This Bearer token can be cached and save
token for you. Otherwise, this token can be used the sam
above.

Example (Bearer Token)

```
let con_token = egg_mode::KeyPair::new("consumer key", "consumer secret");
let token = core.run(egg_mode::bearer_token(&con_token, &handle)).unwrap();

// token can be given to *most* egg_mode methods that ask for a token
// for restrictions, see docs for bearer_token
```

Struct egg_mode::tweet::Timeline

[~] [src]

```
pub struct Timeline<'a> {
    pub count: i32,
    pub max_id: Option<u64>,
    pub min_id: Option<u64>,
    // some fields omitted
}
```

[~] Helper struct to navigate collections of tweets by requ

Using a Timeline to navigate collections of tweets (like
through a collection and only load in tweets you need.

To begin, call a method that returns a Timeline, opt

```
let timeline = egg_mode::tweet::home_timeline(&token, &handle)
    .with_page_size(10);

let (timeline, feed) = core.run(timeline.start()).unwrap();
for tweet in &feed {
    println!("{}", tweet.user.as
```

If you need to load the next set of tweets, call `older`,

```
let (timeline, feed) = core.run(timeline.older(None)).unwrap();
for tweet in &feed {
    println!("{}", tweet.user.as
```

...and similarly for `newer`, which operates in a simila

If you want to start afresh and reload the newest set of tweets again, you can call `start` again, which will clear the tracked tweet
IDs before loading the newest set of tweets. However, if you've been storing these tweets as you go, and already know the newest
tweet ID you have on hand, you can load only those tweets you need like this:

```
let timeline = egg_mode::tweet::home_timeline(&token, &handle)
    .with_page_size(10);

let (timeline, _feed) = core.run(timeline.start()).unwrap();

//keep the max_id for later
let reload_id = timeline.max_id.unwrap();

//simulate scrolling down a little bit
let (timeline, _feed) = core.run(timeline.older(None)).unwrap();
let (mut timeline, _feed) = core.run(timeline.older(None)).unwrap();

//reload the timeline with only what's new
timeline.reset();
let (timeline, _new_posts) = core.run(timeline.older(Some(reload_id))).unwrap();
```

Here, the argument to `older` means "older than what I just returned, but newer than the given ID". Since we cleared the tracked
IDs with `reset`, that turns into "the newest tweets available that were posted after the given ID". The earlier invocations of `older`
with `None` do not place a bound on the tweets it loads. `newer` operates in a similar fashion with its argument, saying "newer than
what I just returned, but not newer than this given ID". When called like this, it's possible for these methods to return nothing,
which will also clear the Timeline's tracked IDs.

If you want to manually pull tweets between certain IDs, the `baseline` call function can do that for you. Keep in mind, though,
that call doesn't update the `min_id` or `max_id` fields, so you'll have to set those yourself if you want to follow up with `older`
or `newer`.

i like docs... like, a lot

Guide to Rustdoc Development

The walking tour of rustdoc

Rustdoc actually uses the rustc internals directly. It lives in-tree with the compiler and 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 where we have an internal representation of a crate (HIR) and the ability to run some of 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 details of the high-level overview.

(Side note: `librustdoc` is a library crate! The `rustdoc` binary is created using the `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 3 src/libtest 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/built/STAGE2/` previously, then after the previous build command, `cargo -local doc` will use `rustdoc` to generate the stand docs.
 - The completed docs will be available in `build/STAGE2/doc/site`, though I 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/renderer.rs`. It `fmt::Display` implementations and supplementary functions.
- The types that get `Display` impls above are defined in `clean/html.rs`, right in the `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 functions doctests from a given block of Markdown.
- The tests on rustdoc output are located in `src/test/rustdoc`, where they’re run by test runner of rustbuild and the supplementary script `src/etc/htmldock.py`.
- Tests on search index generation are located in `src/test/rustdoc-js`, as a set files that encode queries on the standard library search index and expected results.

From crate to clean

In `clean.rs` are two central items: the `DocContext` struct, and the `run_crate` function where rustdoc calls out to rustc to compile a crate to the point where rustdoc can take former is a state container used when crawling through a crate to gather its documents.

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

```
pub trait CleanTr {
    fn clean(&self, cx: &DocContext) -> Tr;
}
```

`clean/html.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 CleanCrate` for `rustc::ast::BustdocVisitor`, which is called by `run_crate`.

You see, I actually lied a little earlier: There’s another AST transformation it events in `clean/html.rs`. In `visit_ast.rs` is the type `RustdocVisitor`, which `visitCrate` to get the first intermediate representation, defined in `docx` to get a few intermediate wrappers around the HIR types and to process `v` where `docx::doc::in_line`, `docx::doc::in_line`, and `docx::doc::hidden` are `pr` logic for whether a `pub` use should get the full page or a “Reexport” line it.

The other major thing that happens in `clean/html.rs` is the collection of `docx::doc::in_line` 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 `CleanCrate` 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 `do` or deliberately hidden with `docx::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` to it. Unlike the previous set of AST transformations, the passes happen on the `CleanCrate`.

(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 let me know.)

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 doc 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.
- `remove-comments`: removes excess indentation on comments in `on` it. This is necessary because the convention for writing documentation between the `///` or `/**` marker and the text, and stripping that `le` text easier to parse by the Markdown parser. (In the past, the `markdown` `Commonmark` compliant, which caused annoyances with extra white less of an issue today.)
- `strip-private-imports`: strips all private `Import` statements `use, use` 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 document.
- `strip-hidden` and `strip-private`: strip all `doc(hidden)` and `private` `doc(hidden)`. `strip-private` implies `strip-private-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/renderer.rs`. This code is responsible for `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 code beginning the process of rendering all the documentation for the crate.

Several functions implemented directly on `Context` take the `CleanCrate` between rendering items or recurring on a module’s child items. From `run` 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::Display` and functions that pass around a `fmt::Formatter`. The top-level writes out the page body is the `impl::fmt::Display` for `Item::HTML`. It switches out to one of several `Item::HTML` functions based on the kind of `Item`.

Depending on what kind of rendering code you’re looking for, you’ll probably `html/renderer.rs` for major items like “what sections should I print for a `struct` `html/format.rs` for smaller component pieces like “how should I print a `var` some other item”.

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

From soup to nuts

(alternate title: “An unbroken thread that stretches from those first `cells`”)

It’s important to note that the AST cleaning can ask the compiler for inform `DocContext` contains a `ty::Text`, but page rendering cannot. The `CleanCrate` `run_crate` is passed outside the compiler context before being handed to means that a lot of the “supplementary data” that isn’t immediately available definition, like which trait is the `default` trait used by the language, needs to cleaning, stored in the `DocContext`, and passed along to the `SharedContext` rendering. This manifests as a bunch of shared state, context variables, and

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 `strip-hidden` and `strip-private` strip all `doc(hidden)` and `private` `strip-private` implies `strip-private-imports`. Basically, the goal is to remove items that are not relevant for public documentation.

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::HTML` 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 `fmt::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 `wake_test`, which is where hand-written doctests get transformed into something that can be executed.

Some extra reading about `wake_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/htmldock.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 `htmldock.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.

i like docs... like, maybe too much

quiet misdreavus miniblog

About

how the --

One of rustdoc's greatest features is that it runs like tests. This ensures that all you have to do to run the docs is run the code. There are some steps that need to happen 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_rng();
if rng.gen() { // random bool
    println!("i32: {}, u32: {}",
```

The code is written such that you can run it. You can't really take this and run it as a main function. To make it work with `fn main`.

For these and other reasons, rustdoc runs the code in a test runner. I want to take the code from "handwritten code sample" to "let it get exposed here, and I'd like to see

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

Anyway, the very first thing that rustdoc does is the beginning of the final test. However, it inserts `#![allow(unused)]` instead. I 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 what declarations are at the beginning, and also the 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 you have a 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 attribute:

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

1. There are no extern crate statements deliberately show off importing the crate import, so rustdoc needs to look through the crate was manually imported already.
2. The crate doesn't add the `#![doc(test(attr(...)))]` used by the standard library facade to be relocated from (for example) `core` to `std`.
3. The crate being documented isn't name manually adding it would just clash with slightly redundant, but in case someone else does.
4. We're documenting a crate, and not a `std` test, but rustdoc can also run tests "containing crate" to link against.
5. The sample in question uses the name `rand` redundant if the sample in question does crate name used in the sample.

If all of these conditions hold, then it will add an `extern crate` for the crate. These are fairly easy to hit and only meant for rather niche cases that deal with before. All told, our sample from `rand` passes 1 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` entry point, so they just write under the assumption 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 I 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, a test runner comes to this slot in the test sequence, it can take the text it saved earlier and hand it to the 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` library, you'll see something striking. There are pages there 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 doc` also documentation is being generated. This is to documentation.

By itself, this might be enough, but there's 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 `linux`. This is a little signal to rust on Linux. The net effect of this does two things

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 doc` 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".
 - o (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 touches 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.

“...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

Rustdoc output



Crate std

Version 1.28.0 (9634041f0
2018-07-30)

See all std's items

Primitive Types

Modules

Macros

Crates

alloc

core

proc_macro



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



Crate std

1.0.0 [-][src]

[\[-\]](#) The Rust Standard Library

The Rust Standard Library is the foundation of portable Rust software, a set of minimal and battle-tested shared abstractions for the [broader Rust ecosystem](#). It offers core types, like `Vec<T>` and `Option<T>`, library-defined [operations on language primitives](#), [standard macros](#), I/O and [multithreading](#), among [many other things](#).

`std` is available to all Rust crates by default, just as if each one contained an `extern crate std;` import at the [crate root](#). Therefore the standard library can be accessed in `use` statements through the path `std`, as in `use std::env`, or in expressions through the absolute path `::std`, as in `::std::env::args`.



How to read this documentation

If you already know the name of what you are looking for, the fastest way to find it is to use the [search bar](#) at the top of the page.

Otherwise, you may want to jump to one of these useful sections:

- [std::* modules](#)
- [Primitive types](#)
- [Standard macros](#)

Want to cut to the chase?

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

Trait std::iter::iterator 1.0.0 [\[+\]](#) [\[src\]](#)

[\[+\] Show declaration](#)
[\[+\] Expand description](#)

Associated Types

[\[+\] type Item](#)

Required Methods

[\[+\] fn next\(&mut self\) -> Option<Self::Item>](#)

Provided Methods

[\[+\] fn size_hint\(&self\) -> \(usize, Option<usize>\)](#)
[\[+\] fn count\(self\) -> usize](#)
[\[+\] fn last\(self\) -> Option<Self::Item>](#)
[\[+\] fn nth\(&mut self, n: usize\) -> Option<Self::Item>](#)
[\[+\] fn step_by\(self, step: usize\) -> StepBy<Self>](#) 1.28.0
[\[+\] fn chain<U>\(self, other: U\) -> Chain<Self, <U as IntoIterator>::IntoIter>](#)
where
 U: IntoIterator<Item = Self::Item>.

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>  
  where  
    K: Eq + Hash,  
    S: BuildHasher,  
  
[ - ] fn extend<T: IntoIterator<Item = (K, V)>>(&mut self, iter: T)
```

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>  
  where  
    K: Eq + Hash + Copy,
```

[src]

Check the source!

[src]

1.4.0 [src]

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>
  where
    K: Eq + Hash,
    S: BuildHasher,

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

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

① [+] pub fn hasher(&self) -> &S

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

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

[+] pub fn try_reserve(
    &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!

1.7.0 [src]

1.7.0 [src]

1.9.0 [src]

[src]

[src]

[src]

[src]

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

Want everything on one page?



Crate std

Version 1.30.0-nightly
(73c78734b 2018-08-05)

[See all std's items](#)

Primitive Types

Modules



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

Crate std

[\[-\] The Rust Standard Library](#)

The Rust Standard Library is the foundation of the [broader Rust ecosystem](#). It offers core types, [standard macros](#), [I/O](#) and [multithreading](#).

`std` is available to all Rust crates by default. Therefore the standard library can be accessed through the absolute path `::std`, as in



Crate std

Version 1.30.0-nightly
(73c78734b 2018-08-05)

[Back to index](#)



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

List of all items

Structs

[\[-\] alloc::AllocErr](#)
[alloc::CannotReallocInPlace](#)
[alloc::Excess](#)
[alloc::Global](#)
[alloc::Layout](#)
[alloc::LayoutErr](#)
[alloc::System](#)
[any::TypeId](#)
[any::TypeId::Default](#)

Curious how to use a type?

 vec| 

Results for vec

In Names (200)

In Parameters (200)

In Return Types (199)

std::vec
std::vec::Vec
std::vec
slice::to_vec
std::os::unix::ffi::OsStringExt::from_vec
std::ffi::OsString::from_vec
std::os::unix::ffi::OsStringExt::into_vec
slice::into_vec
std::collections::binary_heap::BinaryHeap::into_vec
std::ffi::NulError::into_vec
std::ffi::OsString::into_vec
std::collections::VecDeque

A contiguous growable array type with heap-allocated con...
A contiguous growable array type, written `Vec<T>` but pro...
Creates a [`Vec`] containing the arguments.
Copies `self` into a new `Vec`.
Creates an [`OsString`] from a byte vector.

Yields the underlying byte vector of this [`OsString`].
Converts `self` into a vector without clones or allocation.
Consumes the `BinaryHeap` and returns the underlying ve...
Consumes this error, returning the underlying vector of by...

A double-ended queue implemented with a growable ring

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

And more!

Keyboard Shortcuts	Search Tricks
Show this help dialog	Prefix searches with a type followed by a colon (e.g. <code>fn:</code>) to restrict the search to a given type.
Focus the search field	
Move up in search results	Accepted types are: <code>fn</code> , <code>mod</code> , <code>struct</code> , <code>enum</code> , <code>trait</code> , <code>type</code> , <code>macro</code> , and <code>const</code> .
Move down in search results	
Switch tab	Search functions by type signature (e.g. <code>vec -> usize</code> or <code>* -> vec</code>)
Go to active search result	
Expand all sections	Search multiple things at once by splitting your query with comma (e.g. <code>str,u8</code> or <code>String,struct:Vec,test</code>)
Collapse all sections	

Press “?” for
keyboard shortcuts
and search hints!

Click the gear by
the search box for
doc settings!

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

Rustdoc settings

- ☐ Auto-hide item declarations.
- ☐ Auto-hide item attributes.
- ☐ Auto-hide trait implementations documentation
- ☐ Directly go to item in search if there is only one result

Results for *

In Names (200)

- * - see `std::ops::Mul`
- * - see `std::ops::MulAssign`
- * - see `std::ops::Deref`
- * - see `std::ops::DerefMut`
- ...

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

```
1 /// This is some struct, here.  
2 ///  
3 /// ```  
4 /// let my_struct = my_project::SomeStruct;  
5 /// println!("hey, here's some code in your code");  
6 /// ```  
7 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
```


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!

```
1 /// Hey, here are some module docs!  
2  
3 /// (written on a spider's web) Some Struct  
4 ///  
5 /// Wow, that must be some struct! Gotta take care of that one.  
6 pub struct SomeStruct;
```

```
1 #![doc = " Hey, here are some module docs!"]  
2  
3 #[doc = " (written on a spider's web) Some Struct"]  
4 #[doc = ""]  
5 #[doc = " Wow, that must be some struct! Gotta take care of that one."]  
6 pub struct SomeStruct;
```

Rustdoc compiles your crate to scrape out these attributes

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

```
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())
}
```

Crate `my_project`

Functions

`unix_size`

Crate `my_project`

Functions

`windows_size`

Rust handles
conditional compilation
before rustdoc can
make your docs!

Forcing rustdoc to see the items

```
#![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!

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());
```

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`.

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

```
//! (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 {}
```


Crate my_project

Structs

Traits

Crates

my_project

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

Crate my_project

[-] (written on a spider's web) Some Crate

Structs

SomeStruct (written on a spider's web) Some Struct

Traits

SomeTrait (written on a spider's web) Some Trait

...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
```

Crate my_project

Structs

Traits

Crates

my_project



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

Crate my_project

[-] (written on a spider's web) Some Crate

Structs

SomeStruct (written on a spider's web) Some Struct

Traits

SomeTrait (written on a spider's web) Some Trait

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

fill the internet with love today

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 [Sage embedding](#)



No? How about some ponies?

<https://docs.rs/pwnies>

DOCS.RS pwnies-0.0.14 Source Platform Find crate

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

Crate **pwnies** [-] [src]

[-]  


pwnies is a XaaS

XSS AS A SERVICE

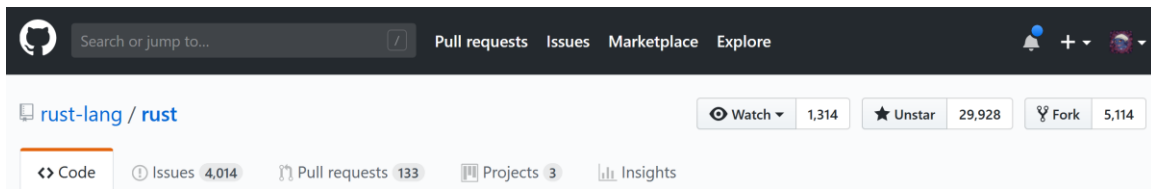
To get **pwnies** to appear in your crate's documentation (like this), simply put:

```
[package.metadata.docs.rs]
rustdoc-args = ["--html-in-header", ".cargo/registry/src/github.com-1ecc8299db9ec823/pwnies-0.0.14/pwnies.html"]
```

in your crate's Cargo.toml.



A peek behind the curtain



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

Branch: master ▾ rust / src /			Create new file	Upload files	Find file	History
bors Auto merge of #53177 - nikomatsakis:nll-redundant-borrows-and-escapin... Latest commit 0aa8d03 a day ago						
..	librustc_target	Rollup merge of #53222 - ljedr:cleanup_rustc_target, r=mark-simulacrum				2 days ago
bootstrap	librustc_traits	[nll] librustc_traits: enable feature(nll) for bootstrap				3 days ago
build_helper	librustc_tsan	[nll] librustc_tsan: enable feature(nll) for bootstrap				3 days ago
ci	librustc_typeck	Rollup merge of #52773 - ljedr:unnecessary_patterns, r=nikomatsakis				2 days ago
dllmalloc @ c99638d	librustdoc	Rollup merge of #53094 - GuillaumeGomez:automatic-expand, r=nrc				2 days ago
doc	libserialize	Rollup merge of #52778 - ljedr:readable_serialize, r=kennytm				11 days ago
etc	libstd	Auto merge of #53216 - kennytm:rollup, r=kennytm				2 days ago
	libsyntax	Rollup merge of #53183 - estebank:println-comma, r=oli-obk				2 days ago
	libsyntax_ext	Rollup merge of #53215 - ljedr:refactor_format, r=estebank				2 days ago
	libsvntax nos	[nll] libsvntax nos: enable feature(nll) for bootstrap				2 days ago

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

J
16s ago



"Nice crate you've got there... it'd be
a shame if something were to
happen to it"

CHAT

Thanks @DebugSteven!

@QuietMisdreavus 2018

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>\n"  
41         "<html lang=\"en\">\n"  
42         "<head>\n"  
43         "    <meta charset=\"utf-8\">\n"  
44         "    <meta name=\"viewport\" content=\"width=device-width, initial-scale=1.0\">\n"  
45         "    <meta name=\"generator\" content=\"rustdoc\">\n"  
46         "    <meta name=\"description\" content=\"{description}\">\n"  
47         "    <meta name=\"keywords\" content=\"{keywords}\">\n"  
48         "    <title>{title}</title>\n"  
49         "    <link rel=\"stylesheet\" type=\"text/css\" href=\"{root_path}normalize{suffix}.css\">\n"  
50         "    <link rel=\"stylesheet\" type=\"text/css\" href=\"{root_path}rustdoc{suffix}.css\" \n"  
51         "        id=\"mainThemeStyle\">\n"  
52         "{themes}\n"  
53         "    <link rel=\"stylesheet\" type=\"text/css\" href=\"{root_path}dark{suffix}.css\">\n"  
54         "    <link rel=\"stylesheet\" type=\"text/css\" href=\"{root_path}light{suffix}.css\" \n"  
55         "        id=\"themeStyle\">\n"  
56         "    <script src=\"{root_path}storage{suffix}.js\"></script>\n"  
57         "{css_extension}\n"  
58         "{favicon}\n"  
59         "{in_header}\n"  
60     "</head>\n"  
61     "<body class=\"rustdoc {css_class}\">\n"  
62     "    <!-- If lto is on -->
```

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