Unstable features

Rustdoc is under active development, and like the Rust compiler, some features are only available on nightly releases. Some of these features are new and need some more testing before they're able to be released to the world at large, and some of them are tied to features in the Rust compiler that are unstable. Several features here require a matching `#![feature(...)]` attribute to enable, and thus are more fully documented in the Unstable Book. Those sections will link over there as necessary.

Nightly-gated functionality

These features just require a nightly build to operate. Unlike the other features on this page, these don't need to be "turned on" with a command-line flag or a `#![feature(...)]` attribute in your crate. This can give them some subtle fallback modes when used on a stable release, so be careful!

Error numbers for `compile-fail` doctests

As detailed in the chapter on documentation tests, you can add a `compile_fail` attribute to a doctest to state that the test should fail to compile. However, on nightly, you can optionally add an error number to state that a doctest should emit a specific error number:

```
```compile_fail,E0044
extern { fn some_func<T>(x: T); }
```
```

This is used by the error index to ensure that the samples that correspond to a given error number properly emit that error code. However, these error codes aren't guaranteed to be the only thing that a piece of code emits from version to version, so this is unlikely to be stabilized in the future.

Attempting to use these error numbers on stable will result in the code sample being interpreted as plain text.

Extensions to the `#[doc]` attribute

These features operate by extending the `#[doc]` attribute, and thus can be caught by the compiler and enabled with a `#![feature(...)]` attribute in your crate.

`#[doc(cfg)]`: Recording what platforms or features are required for code to be present

- Tracking issue: #43781

You can use `#[doc(cfg(...))]` to tell Rustdoc exactly which platform items appear on. This has two effects:

1. doctests will only run on the appropriate platforms, and
2. When Rustdoc renders documentation for that item, it will be accompanied by a banner explaining that the item is only available on certain platforms.

`#[doc(cfg)]` is intended to be used alongside `#[cfg(doc)]`. For example, `#[cfg(any(windows, doc))]` will preserve the item either on Windows or during the documentation process. Then, adding a new attribute `#[doc(cfg(windows))]` will tell Rustdoc that the item is supposed to be used on Windows. For example:

```
#![feature(doc_cfg)]

/// Token struct that can only be used on Windows.
#[cfg(any(windows, doc))]
#[doc(cfg(windows))]
pub struct WindowsToken;

/// Token struct that can only be used on Unix.
#[cfg(any(unix, doc))]
#[doc(cfg(unix))]
pub struct UnixToken;

/// Token struct that is only available with the `serde` feature
#[cfg(feature = "serde")]
#[doc(cfg(feature = "serde"))]
#[derive(serde::Deserialize)]
pub struct SerdeToken;
```

In this sample, the tokens will only appear on their respective platforms, but they will both appear in documentation.

`#[doc(cfg(...))]` was introduced to be used by the standard library and currently requires the `#![feature(doc_cfg)]` feature gate. For more information, see its chapter in the Unstable Book and its tracking issue.

`doc_auto_cfg`: Automatically generate `#[doc(cfg)]`

- Tracking issue: #43781

`doc_auto_cfg` is an extension to the `#[doc(cfg)]` feature. With it, you don't need to add `#[doc(cfg(...)]` anymore unless you want to override the default behaviour. So if we take the previous source code:

```
#![feature(doc_auto_cfg)]

/// Token struct that can only be used on Windows.
#[cfg(any(windows, doc))]
pub struct WindowsToken;
```

```rust
/// Token struct that can only be used on Unix.
#[cfg(any(unix, doc))]
pub struct UnixToken;

/// Token struct that is only available with the `serde` feature
#[cfg(feature = "serde")]
#[derive(serde::Deserialize)]
pub struct SerdeToken;
```

It'll render almost the same, the difference being that `doc` will also be displayed. To fix this, you can use `doc_cfg_hide`:

```rust
#![feature(doc_cfg_hide)]
#![doc(cfg_hide(doc))]
```

And `doc` won't show up anymore!

Adding your trait to the "Notable traits" dialog

- Tracking issue: #45040

Rustdoc keeps a list of a few traits that are believed to be "fundamental" to types that implement them. These traits are intended to be the primary interface for their implementers, and are often most of the API available to be documented on their types. For this reason, Rustdoc will track when a given type implements one of these traits and call special attention to it when a function returns one of these types. This is the "Notable traits" dialog, accessible as a circled `i` button next to the function, which, when clicked, shows the dialog.

In the standard library, some of the traits that are part of this list are `Iterator`, `Future`, `io::Read`, and `io::Write`. However, rather than being implemented as a hard-coded list, these traits have a special marker attribute on them: `#[doc(notable_trait)]`. This means that you can apply this attribute to your own trait to include it in the "Notable traits" dialog in documentation.

The `#[doc(notable_trait)]` attribute currently requires the `#![feature(doc_notable_trait)]` feature gate. For more information, see its chapter in the Unstable Book and its tracking issue.

Exclude certain dependencies from documentation

- Tracking issue: #44027

The standard library uses several dependencies which, in turn, use several types and traits from the standard library. In addition, there are several compiler-internal crates that are not considered to be part of the official standard library, and thus would be a distraction to include in documentation. It's not enough to exclude their crate documentation, since information about trait implementations appears on the pages for both the type and the trait, which can be in different crates!

To prevent internal types from being included in documentation, the standard library adds an attribute to their `extern crate` declarations: `#[doc(masked)]`. This causes Rustdoc to "mask out" types from these crates when building lists of trait implementations.

The `#[doc(masked)]` attribute is intended to be used internally, and requires the `#![feature(doc_masked)]` feature gate. For more information, see its chapter in the Unstable Book and its tracking issue.

Document primitives

This is for Rust compiler internal use only.

Since primitive types are defined in the compiler, there's no place to attach documentation attributes. The `#[doc(primitive)]` attribute is used by the standard library to provide a way to generate documentation for primitive types, and requires `#![feature(rustdoc_internals)]` to enable.

Document keywords

This is for Rust compiler internal use only.

Rust keywords are documented in the standard library (look for `match` for example).

To do so, the `#[doc(keyword = "...")]` attribute is used. Example:

```
#![feature(rustdoc_internals)]

/// Some documentation about the keyword.
#[doc(keyword = "keyword")]
mod empty_mod {}
```

Unstable command-line arguments

These features are enabled by passing a command-line flag to Rustdoc, but the flags in question are themselves marked as unstable. To use any of these options, pass `-Z unstable-options` as well as the flag in question to Rustdoc on the command-line. To do this from Cargo, you can either use the `RUSTDOCFLAGS` environment variable or the `cargo rustdoc` command.

`--markdown-before-content`: include rendered Markdown before the content

- Tracking issue: #44027

Using this flag looks like this:

```
$ rustdoc src/lib.rs -Z unstable-options --markdown-before-content extra.md
$ rustdoc README.md -Z unstable-options --markdown-before-content extra.md
```

Just like `--html-before-content`, this allows you to insert extra content inside the `<body>` tag but before the other content `rustdoc` would normally produce in the rendered documentation. However, instead of directly inserting the file verbatim, `rustdoc` will pass the files through a Markdown renderer before inserting the result into the file.

`--markdown-after-content`: include rendered Markdown after the content

Using this flag looks like this:

```
$ rustdoc src/lib.rs -Z unstable-options --markdown-after-content extra.md
$ rustdoc README.md -Z unstable-options --markdown-after-content extra.md
```

Just like `--html-after-content`, this allows you to insert extra content before the `</body>` tag but after the other content `rustdoc` would normally produce in the rendered documentation. However, instead of directly inserting the file verbatim, `rustdoc` will pass the files through a Markdown renderer before inserting the result into the file.

`--playground-url`: control the location of the playground

Using this flag looks like this:

```
$ rustdoc src/lib.rs -Z unstable-options --playground-url https://play.rust-lang.org/
```

When rendering a crate's docs, this flag gives the base URL of the Rust Playground, to use for generating `Run` buttons. Unlike `--markdown-playground-url`, this argument works for standalone Markdown files *and* Rust crates. This works the same way as adding `#![doc(html_playground_url = "url")]` to your crate root, as mentioned in the chapter about the `#[doc]` attribute. Please be aware that the official Rust Playground at https://play.rust-lang.org does not have every crate available, so if your examples require your crate, make sure the playground you provide has your crate available.

If both `--playground-url` and `--markdown-playground-url` are present when rendering a standalone Markdown file, the URL given to `--markdown-playground-url` will take precedence. If both `--playground-url` and `#![doc(html_playground_url = "url")]` are present when rendering crate docs, the attribute will take precedence.

`--sort-modules-by-appearance`: control how items on module pages are sorted

Using this flag looks like this:

```
$ rustdoc src/lib.rs -Z unstable-options --sort-modules-by-appearance
```

Ordinarily, when `rustdoc` prints items in module pages, it will sort them alphabetically (taking some consideration for their stability, and names that end in a

number). Giving this flag to `rustdoc` will disable this sorting and instead make it print the items in the order they appear in the source.

`--show-type-layout`: add a section to each type's docs describing its memory layout

Using this flag looks like this:

```
$ rustdoc src/lib.rs -Z unstable-options --show-type-layout
```

When this flag is passed, rustdoc will add a "Layout" section at the bottom of each type's docs page that includes a summary of the type's memory layout as computed by rustc. For example, rustdoc will show the size in bytes that a value of that type will take in memory.

Note that most layout information is **completely unstable** and may even differ between compilations.

`--resource-suffix`: modifying the name of CSS/JavaScript in crate docs

- Tracking issue: #54765

Using this flag looks like this:

```
$ rustdoc src/lib.rs -Z unstable-options --resource-suffix suf
```

When rendering docs, `rustdoc` creates several CSS and JavaScript files as part of the output. Since all these files are linked from every page, changing where they are can be cumbersome if you need to specially cache them. This flag will rename all these files in the output to include the suffix in the filename. For example, `light.css` would become `light-suf.css` with the above command.

`--extern-html-root-url`: control how rustdoc links to non-local crates

Using this flag looks like this:

```
$ rustdoc src/lib.rs -Z unstable-options --extern-html-root-url some-crate=https://example.c
```

Ordinarily, when rustdoc wants to link to a type from a different crate, it looks in two places: docs that already exist in the output directory, or the `#![doc(doc_html_root)]` set in the other crate. However, if you want to link to docs that exist in neither of those places, you can use these flags to control that behavior. When the `--extern-html-root-url` flag is given with a name matching one of your dependencies, rustdoc use that URL for those docs. Keep in mind that if those docs exist in the output directory, those local docs will still override this flag.

`-Z force-unstable-if-unmarked`

Using this flag looks like this:

```
$ rustdoc src/lib.rs -Z force-unstable-if-unmarked
```

This is an internal flag intended for the standard library and compiler that applies an `#[unstable]` attribute to any dependent crate that doesn't have another stability attribute. This allows `rustdoc` to be able to generate documentation for the compiler crates and the standard library, as an equivalent command-line argument is provided to `rustc` when building those crates.

`--index-page`: provide a top-level landing page for docs

This feature allows you to generate an index-page with a given markdown file. A good example of it is the rust documentation index.

With this, you'll have a page which you can custom as much as you want at the top of your crates.

Using `index-page` option enables `enable-index-page` option as well.

`--enable-index-page`: generate a default index page for docs

This feature allows the generation of a default index-page which lists the generated crates.

`--nocapture`: disable output capture for test

When this flag is used with `--test`, the output (stdout and stderr) of your tests won't be captured by rustdoc. Instead, the output will be directed to your terminal, as if you had run the test executable manually. This is especially useful for debugging your tests!

`--check`: only checks the documentation

When this flag is supplied, rustdoc will type check and lint your code, but will not generate any documentation or run your doctests.

Using this flag looks like:

```
rustdoc -Z unstable-options --check src/lib.rs
```

`--static-root-path`: control how static files are loaded in HTML output

Using this flag looks like this:

```
$ rustdoc src/lib.rs -Z unstable-options --static-root-path '/cache/'
```

This flag controls how rustdoc links to its static files on HTML pages. If you're hosting a lot of crates' docs generated by the same version of rustdoc, you can use this flag to cache rustdoc's CSS, JavaScript, and font files in a single location, rather than duplicating it once per "doc root" (grouping of crate docs generated into the same output directory, like with `cargo doc`). Per-crate files like the

search index will still load from the documentation root, but anything that gets renamed with `--resource-suffix` will load from the given path.

`--persist-doctests`: persist doctest executables after running

- Tracking issue: #56925

Using this flag looks like this:

```
$ rustdoc src/lib.rs --test -Z unstable-options --persist-doctests target/rustdoctest
```

This flag allows you to keep doctest executables around after they're compiled or run. Usually, rustdoc will immediately discard a compiled doctest after it's been tested, but with this option, you can keep those binaries around for farther testing.

`--show-coverage`: calculate the percentage of items with documentation

- Tracking issue: #58154

Using this flag looks like this:

```
$ rustdoc src/lib.rs -Z unstable-options --show-coverage
```

It generates something like this:

```
+--------------------------------------+------------+------------+------------+------------+
| File                                 | Documented | Percentage |   Examples | Percentage |
+--------------------------------------+------------+------------+------------+------------+
| lib.rs                               |          4 |     100.0% |          1 |      25.0% |
+--------------------------------------+------------+------------+------------+------------+
| Total                                |          4 |     100.0% |          1 |      25.0% |
+--------------------------------------+------------+------------+------------+------------+
```

If you want to determine how many items in your crate are documented, pass this flag to rustdoc. When it receives this flag, it will count the public items in your crate that have documentation, and print out the counts and a percentage instead of generating docs.

Some methodology notes about what rustdoc counts in this metric:

- Rustdoc will only count items from your crate (i.e. items re-exported from other crates don't count).
- Docs written directly onto inherent impl blocks are not counted, even though their doc comments are displayed, because the common pattern in Rust code is to write all inherent methods into the same impl block.
- Items in a trait implementation are not counted, as those impls will inherit any docs from the trait itself.
- By default, only public items are counted. To count private items as well, pass `--document-private-items` at the same time.

Public items that are not documented can be seen with the built-in `missing_docs` lint. Private items that are not documented can be seen with Clippy's `missing_docs_in_private_items` lint.

Calculating code examples follows these rules:

1. These items aren't accounted by default:

- struct/union field
- enum variant
- constant
- static
- typedef

2. If one of the previously listed items has a code example, then it'll be counted.

JSON output When using `--output-format json` with this option, it will display the coverage information in JSON format. For example, here is the JSON for a file with one documented item and one undocumented item:

```
/// This item has documentation
pub fn foo() {}

pub fn no_documentation() {}
```

```
{"no_std.rs":{"total":3,"with_docs":1,"total_examples":3,"with_examples":0}}
```

Note that the third item is the crate root, which in this case is undocumented.

`-w/--output-format`: output format

`--output-format json` emits documentation in the experimental JSON format. `--output-format html` has no effect, and is also accepted on stable toolchains.

It can also be used with `--show-coverage`. Take a look at its documentation for more information.

`--enable-per-target-ignores`: allow `ignore-foo` style filters for doctests

- Tracking issue: #64245

Using this flag looks like this:

```
$ rustdoc src/lib.rs -Z unstable-options --enable-per-target-ignores
```

This flag allows you to tag doctests with compiletest style `ignore-foo` filters that prevent rustdoc from running that test if the target triple string contains foo. For example:

```
///```ignore-foo,ignore-bar
///assert!(2 == 2);
```

```
///```
struct Foo;
```

This will not be run when the build target is `super-awesome-foo` or `less-bar-awesome`. If the flag is not enabled, then rustdoc will consume the filter, but do nothing with it, and the above example will be run for all targets. If you want to preserve backwards compatibility for older versions of rustdoc, you can use

```
///```ignore,ignore-foo
///assert!(2 == 2);
///```
struct Foo;
```

In older versions, this will be ignored on all targets, but on newer versions `ignore-gnu` will override `ignore`.

`--runtool, --runtool-arg`: program to run tests with; args to pass to it

- Tracking issue: #64245

Using these options looks like this:

```
$ rustdoc src/lib.rs -Z unstable-options --runtool runner --runtool-arg --do-thing --runtool
```

These options can be used to run the doctest under a program, and also pass arguments to that program. For example, if you want to run your doctests under valgrind you might run

```
$ rustdoc src/lib.rs -Z unstable-options --runtool valgrind
```

Another use case would be to run a test inside an emulator, or through a Virtual Machine.

`--with-examples`: include examples of uses of items as documentation

- Tracking issue: #88791

This option, combined with `--scrape-examples-target-crate` and `--scrape-examples-output-path`, is used to implement the functionality in RFC #3123. Uses of an item (currently functions / call-sites) are found in a crate and its reverse-dependencies, and then the uses are included as documentation for that item. This feature is intended to be used via `cargo doc --scrape-examples`, but the rustdoc-only workflow looks like:

```
$ rustdoc examples/ex.rs -Z unstable-options \
    --extern foobar=target/deps/libfoobar.rmeta \
    --scrape-examples-target-crate foobar \
    --scrape-examples-output-path output.calls
$ rustdoc src/lib.rs -Z unstable-options --with-examples output.calls
```

First, the library must be checked to generate an `rmeta`. Then a reverse-dependency like `examples/ex.rs` is given to rustdoc with the target crate being documented (`foobar`) and a path to output the calls (`output.calls`). Then, the generated calls file can be passed via `--with-examples` to the subsequent documentation of `foobar`.

To scrape examples from test code, e.g. functions marked `#[test]`, then add the `--scrape-tests` flag.

`--check-cfg`: check configuration flags

- Tracking issue: #82450

This flag accepts the same values as `rustc --check-cfg`, and uses it to check configuration flags.

Using this flag looks like this:

```
$ rustdoc src/lib.rs -Z unstable-options \
    --check-cfg='names()' --check-cfg='values(feature, "foo", "bar")'
```

The example above check every well known names (`target_os`, `doc`, `test`, ... via `names()`) and check the values of `feature`: `foo` and `bar`.