rustbuild - Bootstrapping Rust

This is an in-progress README which is targeted at helping to explain how Rust is bootstrapped and in general some of the technical details of the build system.

Using rustbuild

The rustbuild build system has a primary entry point, a top level `x.py` script:

```
$ python ./x.py build
```

Note that if you're on Unix you should be able to execute the script directly:

```
$ ./x.py build
```

The script accepts commands, flags, and arguments to determine what to do:

- `build` - a general purpose command for compiling code. Alone `build` will bootstrap the entire compiler, and otherwise arguments passed indicate what to build. For example:

  ```
  # build the whole compiler
  ./x.py build --stage 2

  # build the stage1 compiler
  ./x.py build

  # build stage0 libstd
  ./x.py build --stage 0 library/std

  # build a particular crate in stage0
  ./x.py build --stage 0 library/test
  ```

  If files are dirty that would normally be rebuilt from stage 0, that can be overridden using `--keep-stage 0`. Using `--keep-stage n` will skip all steps that belong to stage n or earlier:

  ```
  # build stage 1, keeping old build products for stage 0
  ./x.py build --keep-stage 0
  ```

- `test` - a command for executing unit tests. Like the `build` command this will execute the entire test suite by default, and otherwise it can be used to select which test suite is run:

  ```
  # run all unit tests
  ./x.py test

  # execute tool tests
  ./x.py test tidy
  ```

```
# execute the UI test suite
./x.py test src/test/ui

# execute only some tests in the UI test suite
./x.py test src/test/ui --test-args substring-of-test-name

# execute tests in the standard library in stage0
./x.py test --stage 0 library/std

# execute tests in the core and standard library in stage0,
# without running doc tests (thus avoid depending on building the compiler)
./x.py test --stage 0 --no-doc library/core library/std

# execute all doc tests
./x.py test src/doc
```

- `doc` - a command for building documentation. Like above can take arguments for what to document.

Configuring rustbuild

There are currently two methods for configuring the rustbuild build system.

First, rustbuild offers a TOML-based configuration system with a `config.toml` file. An example of this configuration can be found at `config.toml.example`, and the configuration file can also be passed as `--config path/to/config.toml` if the build system is being invoked manually (via the python script).

Next, the `./configure` options serialized in `config.mk` will be parsed and read. That is, if any `./configure` options are passed, they'll be handled naturally. `./configure` should almost never be used for local installations, and is primarily useful for CI. Prefer to customize behavior using `config.toml`.

Finally, rustbuild makes use of the cc-rs crate which has its own method of configuring C compilers and C flags via environment variables.

Build stages

The rustbuild build system goes through a few phases to actually build the compiler. What actually happens when you invoke rustbuild is:

1. The entry point script, `x.py` is run. This script is responsible for downloading the stage0 compiler/Cargo binaries, and it then compiles the build system itself (this folder). Finally, it then invokes the actual `bootstrap` binary build system.
2. In Rust, `bootstrap` will slurp up all configuration, perform a number of sanity checks (compilers exist for example), and then start building the stage0 artifacts.

3. The stage0 `cargo` downloaded earlier is used to build the standard library and the compiler, and then these binaries are then copied to the `stage1` directory. That compiler is then used to generate the stage1 artifacts which are then copied to the stage2 directory, and then finally the stage2 artifacts are generated using that compiler.

The goal of each stage is to (a) leverage Cargo as much as possible and failing that (b) leverage Rust as much as possible!

Incremental builds

You can configure rustbuild to use incremental compilation with the `--incremental` flag:

```
$ ./x.py build --incremental
```

The `--incremental` flag will store incremental compilation artifacts in `build/<host>/stage0-incremental`. Note that we only use incremental compilation for the stage0 -> stage1 compilation – this is because the stage1 compiler is changing, and we don't try to cache and reuse incremental artifacts across different versions of the compiler.

You can always drop the `--incremental` to build as normal (but you will still be using the local nightly as your bootstrap).

Directory Layout

This build system houses all output under the `build` directory, which looks like this:

```
# Root folder of all output. Everything is scoped underneath here
build/

  # Location where the stage0 compiler downloads are all cached. This directory
  # only contains the tarballs themselves as they're extracted elsewhere.
  cache/
    2015-12-19/
    2016-01-15/
    2016-01-21/
    ...

  # Output directory for building this build system itself. The stage0
  # cargo/rustc are used to build the build system into this location.
  bootstrap/
    debug/
    release/

  # Output of the dist-related steps like dist-std, dist-rustc, and dist-docs
```

```
dist/

# Temporary directory used for various input/output as part of various stages
tmp/

# Each remaining directory is scoped by the "host" triple of compilation at
# hand.
x86_64-unknown-linux-gnu/

  # The build artifacts for the `compiler-rt` library for the target this
  # folder is under. The exact layout here will likely depend on the platform,
  # and this is also built with CMake so the build system is also likely
  # different.
  compiler-rt/
    build/

  # Output folder for LLVM if it is compiled for this target
  llvm/

    # build folder (e.g. the platform-specific build system). Like with
    # compiler-rt this is compiled with CMake
    build/

    # Installation of LLVM. Note that we run the equivalent of 'make install'
    # for LLVM to setup these folders.
    bin/
    lib/
    include/
    share/
    ...

  # Output folder for all documentation of this target. This is what's filled
  # in whenever the `doc` step is run.
  doc/

  # Output for all compiletest-based test suites
  test/
    ui/
    debuginfo/
    ...

  # Location where the stage0 Cargo and Rust compiler are unpacked. This
  # directory is purely an extracted and overlaid tarball of these two (done
  # by the bootstrapy python script). In theory the build system does not
  # modify anything under this directory afterwards.
  stage0/
```

```
# These to build directories are the cargo output directories for builds of
# the standard library and compiler, respectively. Internally these may also
# have other target directories, which represent artifacts being compiled
# from the host to the specified target.
#
# Essentially, each of these directories is filled in by one `cargo`
# invocation. The build system instruments calling Cargo in the right order
# with the right variables to ensure these are filled in correctly.
stageN-std/
stageN-test/
stageN-rustc/
stageN-tools/

# This is a special case of the above directories, **not** filled in via
# Cargo but rather the build system itself. The stage0 compiler already has
# a set of target libraries for its own host triple (in its own sysroot)
# inside of stage0/. When we run the stage0 compiler to bootstrap more
# things, however, we don't want to use any of these libraries (as those are
# the ones that we're building). So essentially, when the stage1 compiler is
# being compiled (e.g. after libstd has been built), *this* is used as the
# sysroot for the stage0 compiler being run.
#
# Basically this directory is just a temporary artifact use to configure the
# stage0 compiler to ensure that the libstd we just built is used to
# compile the stage1 compiler.
stage0-sysroot/lib/

# These output directories are intended to be standalone working
# implementations of the compiler (corresponding to each stage). The build
# system will link (using hard links) output from stageN-{std,rustc} into
# each of these directories.
#
# In theory there is no extra build output in these directories.
stage1/
stage2/
stage3/
```

Cargo projects

The current build is unfortunately not quite as simple as `cargo build` in a directory, but rather the compiler is split into three different Cargo projects:

- `library/std` - the standard library
- `library/test` - testing support, depends on libstd
- `compiler/rustc` - the actual compiler itself

Each "project" has a corresponding Cargo.lock file with all dependencies, and this means that building the compiler involves running Cargo three times. The structure here serves two goals:

1. Facilitating dependencies coming from crates.io. These dependencies don't depend on `std`, so libstd is a separate project compiled ahead of time before the actual compiler builds.
2. Splitting "host artifacts" from "target artifacts". That is, when building code for an arbitrary target you don't need the entire compiler, but you'll end up needing libraries like libtest that depend on std but also want to use crates.io dependencies. Hence, libtest is split out as its own project that is sequenced after `std` but before `rustc`. This project is built for all targets.

There is some loss in build parallelism here because libtest can be compiled in parallel with a number of rustc artifacts, but in theory the loss isn't too bad!

Build tools

We've actually got quite a few tools that we use in the compiler's build system and for testing. To organize these, each tool is a project in `src/tools` with a corresponding `Cargo.toml`. All tools are compiled with Cargo (currently having independent `Cargo.lock` files) and do not currently explicitly depend on the compiler or standard library. Compiling each tool is sequenced after the appropriate libstd/libtest/librustc compile above.

Extending rustbuild

So you'd like to add a feature to the rustbuild build system or just fix a bug. Great! One of the major motivational factors for moving away from `make` is that Rust is in theory much easier to read, modify, and write. If you find anything excessively confusing, please open an issue on this and we'll try to get it documented or simplified pronto.

First up, you'll probably want to read over the documentation above as that'll give you a high level overview of what rustbuild is doing. You also probably want to play around a bit yourself by just getting it up and running before you dive too much into the actual build system itself.

After that, each module in rustbuild should have enough documentation to keep you up and running. Some general areas that you may be interested in modifying are:

- Adding a new build tool? Take a look at `bootstrap/tool.rs` for examples of other tools.
- Adding a new compiler crate? Look no further! Adding crates can be done by adding a new directory with `Cargo.toml` followed by configuring all `Cargo.toml` files accordingly.

- Adding a new dependency from crates.io? This should just work inside the compiler artifacts stage (everything other than libtest and libstd).
- Adding a new configuration option? You'll want to modify `bootstrap/flags.rs` for command line flags and then `bootstrap/config.rs` to copy the flags to the `Config` struct.
- Adding a sanity check? Take a look at `bootstrap/sanity.rs`.

If you make a major change, please remember to:

- Update `VERSION` in `src/bootstrap/main.rs`.
- Update `changelog-seen = N` in `config.toml.example`.
- Add an entry in `src/bootstrap/CHANGELOG.md`.

A 'major change' includes

- A new option or
- A change in the default options.

Changes that do not affect contributors to the compiler or users building rustc from source don't need an update to `VERSION`.

If you have any questions feel free to reach out on the `#t-infra` channel in the Rust Zulip server or ask on internals.rust-lang.org. When you encounter bugs, please file issues on the rust-lang/rust issue tracker.