Docker images for CI

This folder contains a bunch of docker images used by the continuous integration (CI) of Rust. An script is accompanied (`run.sh`) with these images to actually execute them. To test out an image execute:

```
./src/ci/docker/run.sh $image_name
```

for example:

```
./src/ci/docker/run.sh x86_64-gnu
```

Images will output artifacts in an `obj` dir at the root of a repository.

To match conditions in rusts CI, also set the environment variable `DEPLOY=1` , e.g.:

```
DEPLOY=1 ./src/ci/docker/run.sh x86_64-gnu
```

NOTE: Re-using the same `obj` dir with different docker images with the same target triple (e.g. `dist-x86_64-linux` and `dist-various-1`) may result in strange linker errors, due shared library versions differing between platforms.

If you encounter any issues when using multiple Docker images, try deleting your `obj` directory before running your command.

Filesystem layout

- Each host architecture has its own `host-{arch}` directory, and those directories contain a subdirectory for each Docker image (plus the `disabled` subdirectory).
- `host-{arch}/disabled` contains images that are not built on CI.
- `scripts` contains files shared by multiple Docker images.

Docker Toolbox on Windows

For Windows before Windows 10, the docker images can be run on Windows via Docker Toolbox. There are several preparation needs to be made before running a Docker image.

1. Stop the virtual machine from the terminal with `docker-machine stop`
2. If your Rust source is placed outside of `C:\Users\**` , e.g. if you place the repository in the `E:\rust` folder, please add a shared folder from VirtualBox by:
    1. Select the "default" virtual machine inside VirtualBox, then click "Settings"
    2. Go to "Shared Folders", click "Add shared folder" (the folder icon with a plus sign), fill in the following information, then click "OK":
        - Folder path: `E:\rust`
        - Folder name: `e/rust`
        - Read-only: ☐ *unchecked*
        - Auto-mount: ☑ *checked*
        - Make Permanent: ☑ *checked*

3. VirtualBox might not support creating symbolic links inside a shared folder by default. You can enable it manually by running these from `cmd.exe`:

```
cd "C:\Program Files\Oracle\VirtualBox"
VBoxManage setextradata default
VBoxInternal2/SharedFoldersEnableSymlinksCreate/e/rust 1
::
^~~~~~
::
folder name
```

4. Restart the virtual machine from terminal with `docker-machine start`.

To run the image,

1. Launch the "Docker Quickstart Terminal".
2. Execute `./src/ci/docker/run.sh $image_name` as explained at the beginning.

Cross toolchains

A number of these images take quite a long time to compile as they're building whole gcc toolchains to do cross builds with. Much of this is relatively self-explanatory but some images use crosstool-ng which isn't quite as self explanatory. Below is a description of where these `*.config` files come form, how to generate them, and how the existing ones were generated.

Generating a `.config` file

NOTE: Existing Dockerfiles can also be a good guide for the process and order of script execution.

If you have a `linux-cross` image lying around you can use that and skip the next two steps.

- First we spin up a container and copy all scripts into it. All these steps are outside the container:

```
# Note: We use ubuntu:16.04 because that's the "base" of linux-cross Docker
# image, or simply run ./src/ci/docker/run.sh once, which will download the correct
# one and you can check it out with `docker images`
$ docker run -it ubuntu:16.04 bash
# in another terminal:
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES
cfbec05ed730        ubuntu:16.04        "bash"              16 seconds ago      Up 15
seconds                             drunk_murdock
$ docker cp src/ci/docker/scripts drunk_murdock:/tmp/
```

- Then inside the container we build crosstool-ng by simply calling the bash script we copied in the previous step:

```
$ cd /tmp/scripts
# Download packages necessary for building
$ bash ./cross-apt-packages.sh
```

```
# Download and build crosstool-ng
$ bash ./crosstool-ng.sh
```

- In case you want to adjust or start from an existing config, copy that to the container. `crosstool-ng` will automatically load `./.config` if present. Otherwise one can use the TUI to load any config-file.

```
$ docker cp arm-linux-gnueabi.config drunk_murdock:/tmp/.config
```

- Now, inside the container run the following command to configure the toolchain. To get a clue of which options need to be changed check the next section and come back.

```
$ cd /tmp/
$ ct-ng menuconfig
```

- Finally, we retrieve the `.config` file from the container and give it a meaningful name. This is done outside the container.

```
$ docker cp drunk_murdock:/tmp/.config arm-linux-gnueabi.config
```

- Now you can shutdown the container or repeat the two last steps to generate a new `.config` file.

Toolchain configuration

Changes on top of the default toolchain configuration used to generate the `.config` files in this directory. The changes are formatted as follows:

```
$category > $option = $value -- $comment
```

`arm-linux-gnueabi.config`

For targets: `arm-unknown-linux-gnueabi`

- Path and misc options > Prefix directory = /x-tools/${CT_TARGET}
- Path and misc options > Patches origin = Bundled only
- Target options > Target Architecture = arm
- Target options > Architecture level = armv6 -- (+)
- Target options > Floating point = software (no FPU) -- (*)
- Operating System > Target OS = linux
- Operating System > Linux kernel version = 3.2.101
- C-library > glibc version = 2.17.0
- C compiler > gcc version = 8.3.0
- C compiler > C++ = ENABLE -- to cross compile LLVM

`arm-linux-gnueabihf.config`

For targets: `arm-unknown-linux-gnueabihf`

- Path and misc options > Prefix directory = /x-tools/${CT_TARGET}
- Path and misc options > Patches origin = Bundled only
- Target options > Target Architecture = arm
- Target options > Architecture level = armv6 -- (+)
- Target options > Use specific FPU = vfp -- (+)

- Target options > Floating point = hardware (FPU) -- (*)
- Target options > Default instruction set mode = arm -- (+)
- Operating System > Target OS = linux
- Operating System > Linux kernel version = 3.2.101
- C-library > glibc version = 2.17.0
- C compiler > gcc version = 8.3.0
- C compiler > C++ = ENABLE -- to cross compile LLVM

`armv7-linux-gnueabihf.config`

For targets: `armv7-unknown-linux-gnueabihf`

- Path and misc options > Prefix directory = /x-tools/${CT_TARGET}
- Path and misc options > Patches origin = Bundled only
- Target options > Target Architecture = arm
- Target options > Suffix to the arch-part = v7
- Target options > Architecture level = armv7-a -- (+)
- Target options > Use specific FPU = vfpv3-d16 -- (*)
- Target options > Floating point = hardware (FPU) -- (*)
- Target options > Default instruction set mode = thumb -- (*)
- Operating System > Target OS = linux
- Operating System > Linux kernel version = 3.2.101
- C-library > glibc version = 2.17.0
- C compiler > gcc version = 8.3.0
- C compiler > C++ = ENABLE -- to cross compile LLVM

(*) These options have been selected to match the configuration of the arm toolchains shipped with Ubuntu 15.10
(+) These options have been selected to match the gcc flags we use to compile C libraries like jemalloc. See the mk/cfg/arm(v7)-uknown-linux-gnueabi{,hf}.mk file in Rust's source code.

`aarch64-linux-gnu.config`

For targets: `aarch64-unknown-linux-gnu`

- Path and misc options > Prefix directory = /x-tools/${CT_TARGET}
- Target options > Target Architecture = arm
- Target options > Bitness = 64-bit
- Operating System > Target OS = linux
- Operating System > Linux kernel version = 4.2.6
- C-library > glibc version = 2.17 -- aarch64 support was introduced in this version
- C compiler > gcc version = 5.2.0
- C compiler > C++ = ENABLE -- to cross compile LLVM

`powerpc-linux-gnu.config`

For targets: `powerpc-unknown-linux-gnu`

- Path and misc options > Prefix directory = /x-tools/${CT_TARGET}
- Path and misc options > Patches origin = Bundled, then local
- Path and misc options > Local patch directory = /tmp/patches
- Target options > Target Architecture = powerpc
- Target options > Emit assembly for CPU = powerpc -- pure 32-bit PowerPC
- Operating System > Target OS = linux
- Operating System > Linux kernel version = 2.6.32.68 -- ~RHEL6 kernel

- C-library > glibc version = 2.11.1 -- ~SLE11-SP4 glibc
- C compiler > gcc version = 5.2.0
- C compiler > C++ = ENABLE -- to cross compile LLVM

`powerpc64-linux-gnu.config`

For targets: `powerpc64-unknown-linux-gnu`

- Path and misc options > Prefix directory = /x-tools/${CT_TARGET}
- Path and misc options > Patches origin = Bundled, then local
- Path and misc options > Local patch directory = /tmp/patches
- Target options > Target Architecture = powerpc
- Target options > Bitness = 64-bit
- Target options > Emit assembly for CPU = power4 -- (+)
- Target options > Tune for CPU = power6 -- (+)
- Operating System > Target OS = linux
- Operating System > Linux kernel version = 2.6.32.68 -- ~RHEL6 kernel
- C-library > glibc version = 2.11.1 -- ~SLE11-SP4 glibc
- C compiler > gcc version = 5.2.0
- C compiler > C++ = ENABLE -- to cross compile LLVM

(+) These CPU options match the configuration of the toolchains in RHEL6.

`s390x-linux-gnu.config`

For targets: `s390x-unknown-linux-gnu`

- Path and misc options > Prefix directory = /x-tools/${CT_TARGET}
- Path and misc options > Patches origin = Bundled, then local
- Path and misc options > Local patch directory = /tmp/patches
- Target options > Target Architecture = s390
- Target options > Bitness = 64-bit
- Operating System > Target OS = linux
- Operating System > Linux kernel version = 2.6.32.71 -- ~RHEL6 kernel
- C-library > glibc version = 2.12.1 -- ~RHEL6 glibc
- C compiler > gcc version = 8.3.0
- C compiler > gcc extra config = --with-arch=z10 -- LLVM's minimum support
- C compiler > C++ = ENABLE -- to cross compile LLVM