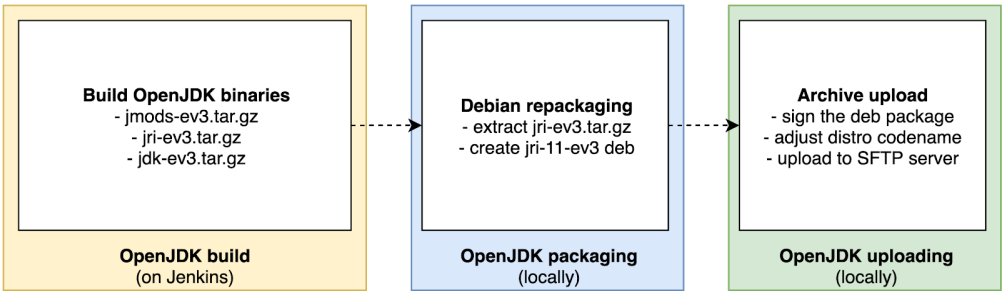
OpenJDK-EV3 Maintainer's Manual

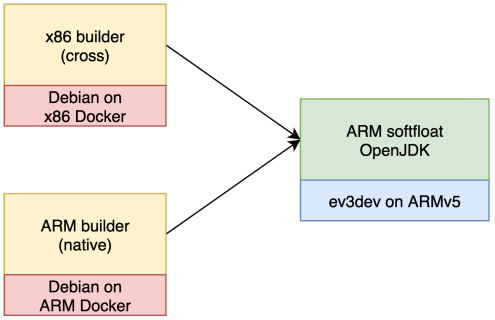
Overview

This manual goes through an overview of the build, packaging & maintenance process.



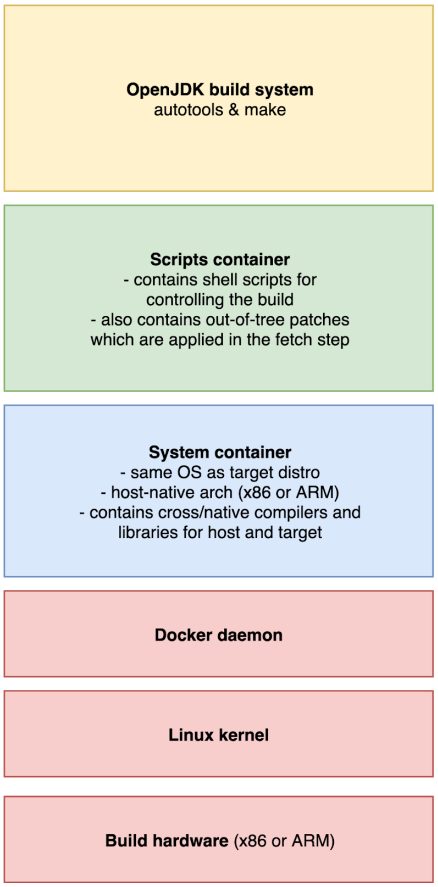
Builds

The goal of this project is to provide a small enough Java runtime that would run on ev3dev on EV3. Currently OpenJDK is used for this. The reason is simple - it contains an Oracle-developed and then opensourced fast ARM32 JIT from JDK9 onwards. The second reason is that leJOS EV3 had used a similar JDK (Embedded JDK 7/8), but it was a proprietary offering prepared and not later updated by Oracle.



To build the OpenJDK binaries in an environment suitable for cross-compilation for EV3, the repository is using a couple of Docker containers. They have two variants: native (ARM host -> ARM target) and cross (x86 host -> ARM target). The native builders have an advantage in that the JDK can be partially tested right on the build machine.

The layers of the build environment are shown on the picture on the right. The lower docker container/image contains the target OS (Debian/ev3dev) with its libraries and compilers. This is done to ensure that the API provided by the system and its libraries are compatible between the build and target machines.



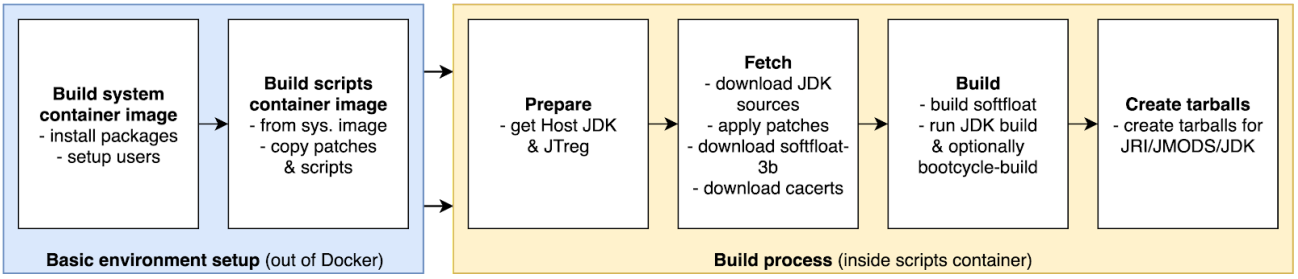
On top of that, there is a set of shell scripts which control the build. Their task is to download JDK and source-level dependencies and to build the JDK binaries with its build system.

These two containers have to be built before the build of OpenJDK itself. Afterwards, the scripts in the upper container take over the rest of the build process.

As the builds run in a container, it is necessary to mount a directory from the host to the container as a build workspace. This makes the access to the built files easier.

Let's return back to the scripts. There are six build scripts in total:

- autobuild.sh - This is the entrypoint for the build. It calls the last four scripts one after another.
- config.sh - This script is source'd from the last four scripts. It takes in the JDK version and other parameters from environment variables given to the container and exports different environment variables to be used by other build scripts (like the JDK source URL).
- prepare.sh - This script contains the ***Prepare*** phase. It downloads and unpacks the Host JDK (needed for OpenJDK build) to a directory in the build workspace. It is used to build the Java classes of the resulting JDK. It also downloads JTreg, which is used when building included test cases (not currently run on Jenkins; but they can be run manually).
- fetch.sh - This scripts contains the ***Fetch*** phase. It has to download the OpenJDK source code (latest GA release). It determines the SCM parameters (commit hash, real version). From them, it generates a metadata file used to identify the build. Then, it applies the patches stored in this repository. Lastly, it clones one extra repository - the AdoptOpenJDK openjdk-build repository used for the CA certificates.
- build.sh - This script is doing the ***Build*** phase. First, it configures and builds the JDK. If the build is running on native (ARM) hardware, it also runs a *bootcycle build*, which uses the newly built JDK to build itself once more. This provides at least a basic level of testing.
- zip.sh - This script finishes the build with the ***Archiving*** stage. It generates the JRI (Java Runtime Image) for the EV3, which is basically a reduced Java runtime. It then packs that and remaining build outputs (such as JMOD packages and the full JDK) into a tarball which is then uploaded as a build artifact.



The build scripts accept a predefined set of environment variables describing the build:

- JDKVER - sets the JDK version to build. One of *9*, *10*, *11*, *12*, *13*, *tip, loom*. Numeric versions will *usually*[1] build the latest tagged General-Availability releases; tip will build the latest commit.

[1] When a new JDK version is branched out for stabilization, then the latest non-GA tag must used, as there are no GA tags yet.

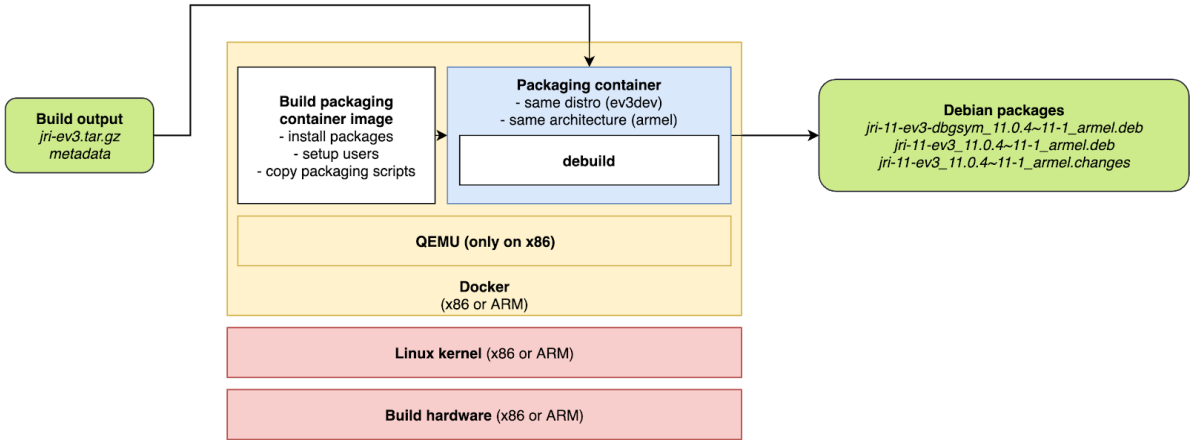
- JDKVM - sets the JVM JIT to use. One of:
    - *zero*: portable non-assembler JIT/interpreter. It's quite slow, but it should always work.
    - *client*: fast ARM assembler-assisted JIT.
    - *minimal*: similar to *client*, but should be a bit smaller. Unfortunately, this configuration does not build successfully (the linking process will fail).
- JDKPLATFORM - sets the platform for the build scripts. Only *ev3* is allowed.
- JDKDEBUG - sets the JVM debug level. Optional. One of (from JDK source):
    - *release*: no debug information, all optimizations, no asserts.
    - *optimized*: no debug information, all optim., no asserts, HS tgt is 'optimized'.
    - *fastdebug*: debug information (-g), all optimizations, all asserts
    - *slowdebug*: debug information (-g), no optimizations, all asserts
- AUTOBUILD - when set to 1 or yes, it runs the *autorun.sh* script directly instead of the shell.

The output of the build process are three different archives:

- jmods-ev3.tar.gz - contains Java JMODs. There are pieces of the JDK that can be assembled into a proper and useful JRE/JDK/JRI. They are intended to be processed with jlink tool provided with host-native Java JDK.
- jri-ev3.tar.gz - EV3 JRI. This is a smaller version of traditional JRE. It is intended to be used as a Java runtime for the brick. It contains only a few Java modules in order to reduce its size.
- jdk-ev3.tar.gz - Full EV3 JDK. This directory comes from the JDK build process; however, it is likely also built using jlink. This could be useful for someone who wants to do full Java development on the brick.

Packaging

Overview



While it is certainly possible, using the *jri-ev3.tar.gz* archive directly is a bit cumbersome. To make the default installation easy to handle, it is necessary to repackage Java to the Debian package format. This also enables easy preinstallation into the ev3dev EV3 image via APT.

The repackaging environment is once again provided by a Docker image. There is a small caveat though. AFAIK due to a limitation of the debian debuild utility, it is necessary to build the package on the same Debian version and architecture as the one the package needs to target. This is not a problem - thanks to the qemu-user-static package, it is possible to run EV3 armel binaries on x86.

The repackaging process inside the container starts with a small wrapper script (packaging.sh). It parses the metadata file generated by the JDK build and it fills templating placeholders in the Debian package definition. It also takes in a debian package revision number - this is a necessary for having the possibility of uploading a new package with the same java version as the previous one. Finally, it calls *debuild* package building utility to build the package.

The packaging/debian subdirectory in the repository contains the template package definition and build scripts. These are processed by debuild. The rules script controls the process; it copies the contents of the JRI archive to the appropriate places for a debian Java package. However most of the heavy lifting is done by *debhelper scripts* called automatically from the rules script.

The repackaging procedure produces five files overall, but only three of them are useful. Those are the debian package, its debug symbols package and an unsigned package upload request. This package upload request can be signed with GPG and after that it has to be uploaded with the packages to the ev3dev repository.

First-time setup

This step is common all following steps (packaging, signing, uploading).

1. Install and run preferably a Debian-based Linux distribution. If you are not running Linux on your hardware, then it should be possible to install Linux in a virtual machine. I recommend running Xubuntu 18.04.
2. Install Docker: https://docs.docker.com/install/linux/docker-ce/ubuntu/
3. Install gnupg, openssh-client, devscripts, qemu-user-static and binfmt-support packages using APT.
    - gnupg is the GNU Privacy Guard, a set of utilities for among other things signing and verifying digital signatures on files (e.g. Debian packages).
    - openssh-client is the Secure Shell client. Its sftp command can used for uploading the packages to the repository server.
    - devscripts contains among other things the debsign utility which is later used for signing the Debian package upload request.
    - qemu-user-static and binfmt-support together allow running ARM Linux binaries on x86 Linux system. This is necessary for the repackaging process as it involves locally running the Debian system that the EV3 uses.
4. To upload the package to the repository sever, you will need a SSH keypair. See GitHub help for how to generate a SSH key if you do not have one.
5. You will also need to have a GPG keypair so that you can sign the generated packages (more on this later). The process is covered on GitHub as well.

6. Now that you have the SSH & GPG keys, you need to submit them to ev3dev maintainers in order for them to become trusted for uploading. To do this, send the public halves of these keys to David Lechner <david@lechnology.com>. A good way to complement this could be an issue on ev3dev GitHub, as that way your GitHub identity will be linked to the origin of the keys. Include your email address (which should be the same as the one in the GPG key) in the request. The ev3dev maintainer will then (hopefully) add the keys to the repository server.
7. You will also need to ask the ev3dev maintainers for becoming a maintainer of the package you want to upload. This will give you the permission on the repository server to overwrite the existing package.
8. Build locally the Docker container for the repackaging process. You will need to have a separate container for each Debian release that you want to target. First, you will need to download the openjdk-ev3 repository and go into into it inside terminal. Then, in order to build packages for ev3dev-stretch, run the following command:

   ```
   sudo docker build \
       --tag ev3dev-lang-java:jdk-stretch-package \
       --build-arg DEBIAN_RELEASE=stretch \
       <repo>/packaging
   ```

   Alternatively, for ev3dev-buster, the following command will do it:

   ```
   sudo docker build \
       --tag ev3dev-lang-java:jdk-buster-package \
       --build-arg DEBIAN_RELEASE=buster \
       <repo>/packaging
   ```

Steps

This operation is not automated, it is necessary to follow it each time a new package version is to be released.

1. Download and move the jri-ev3.tar.gz and metadata from the AdoptOpenJDK Jenkins to an empty directory somewhere on the host. Then, run `chmod -R 777 <path to the directory>` to make the directory accessible to all users (including the UID 1001 in the container). This directory will be used as a working space of the container.
2. Start the repackaging process. The command again differs between the distro releases. You will also need to provide the `JAVA_PACKAGE_REVISION` environment variable. This variable sets the suffix placed after the JDK version number.
   - *Conceptual note.* This is about the version number that the Debian package will contain. Let's analyze the following version number: 11.0.6~10-2
     i. 11.0.6 is the "big" release number
     ii. 10 is the "revision" number (not revision hash) of the SCM tag
     iii. 2 is the packaging suffix intended for package maintainers.
   - The packaging suffix is useful for fixing various issues not directly caused by OpenJDK. For example, if you want to immediately add a new JMOD to the JRI image, you can release a new package with the same Java version but an incremented packaging version.

BEWARE: if you are building packages for Stretch and Buster together, use a different revision for each package. If you create both packages with the same revision, upload of the second package will fail because it will conflict with the first uploaded package.
Command for Stretch:

```
sudo docker run \
    --rm \
    --interactive \
    --tty \
    --volume <path to the directory with JRI>:/build \
    --env "JAVA_PACKAGE_REVISION=-1" \
    ev3dev-lang-java:jdk-stretch-package
```

Command for Buster:

```
sudo docker run \
    --rm \
    --interactive
    --t \
    --volume <path to the directory with JRI>:/build \
    --env "JAVA_PACKAGE_REVISION=-2" \
    ev3dev-lang-java:jdk-buster-package
```

3. Run `chmod -R 777 <path to the directory>` again on the workspace. This will override the permissions coming from the container, eliminating errors stemming from UID mismatch between on-host user and in-container user.
4. Enter the <path to the directory>/pkg subdirectory. This is where the debian package was generated. However, not all files are necesary. Delete all files that do not end with *.deb* or *.changes*.

Signing the packages

Overview

In the Debian world, trust in the package integrity is ensured through GPG signatures. The root of trust is the repository key with which a so called Release file is signed. This file then verifies packages lists (via hashes), which in turn verify the packages (again via hashes). The Release private key resides on the repository server and it is not usually manipulated manually. Instead it is managed automatically by a repository software. Ev3dev is using Reprepro for this.

In order to prevent unauthorized package uploads, the repository software needs to have a list of keys that it trusts. When a maintainer wants to upload a new package, they upload the packages itself (unsigned .deb files) and a package upload request (signed .changes file). The upload request contains hashes of the packages itself and this way it ensures their integrity and their trustworthiness.

Steps

Provided that you have the devscripts debian package installed and the GPG key set up, you can sign the package upload request by running:
`debsign jri-11-ev3_11.0.4~11-1_armel.changes`

Uploading

Overview

Ev3dev maintainers need to have a way to send new packages to the repository software. This is handled via a SFTP server running at *reprepro.ev3dev.org* coupled with a file watcher. After a maintainer uploads a .changes file to a special directory, the server will take the associated files. It will verify their signature and uploader permissions and if all checks pass, it will put the packages into the repository ifself.

First-time setup

See First-time setup in the *Packaging* section.

Steps

For this operation, you can either use the sftp command line client or any other SFTP client like FileZilla.

1. Change the extension of the *.changes* file to a *.tmp* file. Without this, the repository software will detect the upload attempt too soon the the process will fail.
    - `$ mv <name>.changes <name>.tmp`
2. Open a SFTP client and connect to *ev3dev-upload@reprepro.ev3dev.org*
    - `$ sftp ev3dev-upload@reprepro.ev3dev.org`
3. Go to the */debian/* directory on the SFTP server.
    - `sftp> cd debian`
4. Upload the JRI packages (2× .deb) and the upload request (*.tmp*) to the directory.
    - `sftp> put <pkg1>.deb`
    - `sftp> put <pkg2>.deb`
    - `sftp> put <name>.tmp`
5. Rename the *.tmp* file to *.changes* file ***remotely***. By renaming the file after complete upload, the repository software (reprepro) sees the complete upload request and the request can succeed.
    - `sftp> rename <name>.tmp <name>.changes`
6. Leave the SFTP session.
    - `sftp> exit`
7. Check your email inbox. After a few seconds, you should receive an email either confirming the successful upload or describing the error that occured.

The package should now be present in the repository. To verify this, download http://archive.ev3dev.org/debian/dists/stretch/main/binary-armel/Packages (stretch) or

http://archive.ev3dev.org/debian/dists/buster/main/binary-armel/Packages (buster) and look for the JRI package version.

Integration with AdoptOpenJDK Jenkins

To automate the build process, the project uses the AdoptOpenJDK Jenkins CI hosted at https://ci.adoptopenjdk.net/view/ev3dev/ for its builds. The *ev3dev* tab is dedicated to jobs of this project.

The integration is based on Jenkins Pipelines. In the root of the repository, there is a *Jenkinsfile* which describes the build steps & nodes where it can run. It handles the following actions:

- *Docker container image building* - This is a prerequisite of the OpenJDK build. Docker pipelines integration is used to build the container.
- *OpenJDK build* - This builds the Java binary archives. However, Jenkinsfile bypasses the autorun.sh script and runs the build phases directly. By doing that, it is possible to have a separate Jenkins pipeline stages for our phases.
- *Artifact upload* - This uploads the binary archives to the Jenkins master, where they are made available for download by the Internet.
- *Cleanup* - As the build workspace & container images take quite a lot of space, it is necessary to remove them after the build finishes (whether successfully or not).

The *Jenkinsfile* expects to be given some parameters. At the time of writing, there are these parameters:

- JDKVM_VALUE - sets the JVM JIT to be used. See JDKVM for config.sh.
- JDKVER_VALUE - sets the JDK version to be built. See JDKVER for config.sh.
- JDKPLATFORM_VALUE - sets the target platform. See JDKPLATFORM for config.sh.
- DOCKER_ARCH - target arch. It should match the platform. Currently only *armel*.
- DEBIAN - Debian release - can be set to *stretch* or *buster*.
- BUILD_TYPE - either *cross* or *native* - sets the build model (x86>ARM vs ARM>ARM)
- DISABLED - with this enabled, the build will always do nothing and it will always succeed. This is useful for cleanup of old jobs.

There parameters can be fixated for individual jobs by using single-choice “choice parameter”.
Currently all configured builds reside in the *eljbuild* folder: https://ci.adoptopenjdk.net/view/ev3dev/job/eljbuild/. Except for JDK9 & JDK10, they all represent the ARMv8->ARM compilation for the given Debian & JDK versions.
To create a new Jenkins job, create a copy of an existing job and then customize the parameters.

Manual build

It is also possible to run the build OpenJDK on your own computer. This script should do that:

```
# define parameters
TARGET_WORKSPACE="$(pwd)/build" # 10 GB of free space should be sufficient
TARGET_DEBIAN_VERSION="stretch" # stretch or buster
TARGET_OPENJDK_VERSION="11"     # 11, 12, 13, tip, loom are on Jenkins

# clone repository
git clone https://github.com/ev3dev-lang-java/openjdk-ev3.git
cd openjdk-ev3

# prepare working directory
mkdir -p "$TARGET_WORKSPACE"
chmod -R 777 "$TARGET_WORKSPACE" # docker may not share UID with the current user

# build base system container
docker build --build-arg DEBIAN_RELEASE="$TARGET_DEBIAN_VERSION" \
             --build-arg ARCH="armel" \
             --tag "ev3dev-lang-java:jdk-cross-$TARGET_DEBIAN_VERSION" \
             --file ./system/Dockerfile.cross \
             ./system

# on top of that, create a build scripts container
docker build --build-arg commit="$(git rev-parse HEAD)" \
             --build-arg extra="Manual build #1 by $(whoami)" \
             --build-arg DEBIAN_RELEASE="$TARGET_DEBIAN_VERSION" \
             --build-arg BUILD_TYPE="cross" \
             --tag "ev3dev-lang-java:jdk-cross-build" \
             ./scripts

# now run the build
docker run --rm \
           --interactive \
           --tty \
           --volume "$TARGET_WORKSPACE:/build" \
           --env JDKVER="$TARGET_OPENJDK_VERSION" \
           --env JDKVM="client" \
           --env JDKPLATFORM="ev3" \
           --env JDKDEBUG="release" \
           --env AUTOBUILD="1" \
           ev3dev-lang-java:jdk-cross-build

# finally, make workspace accessible for all users (i.e. current one too)
chmod -R 777 "$TARGET_WORKSPACE"
# and list the output directory (now it should contain three *-ev3.tar.gz files)
ls "$TARGET_WORKSPACE"
```

See https://github.com/ev3dev-lang-java/openjdk-ev3/issues/34.

Testing

There isn't much testing done. Although I (@JakubVanek) have pushed others to do it, I also don't have a clear vision of how to do this. There are some ideas:

- Run JTreg tests in QEMU. This hits the problem that QEMU is an emulator - to catch everything, it would have to emulate everything (and work fully with Hotspot JIT,

which I think wasn't fully the case back then). This was once implemented, but it hasn't been maintained.
- Run JTreg tests on ARMv8 capable of running 32bit binaries. This hasn't been done yet, but it could yield some results. However, we are depending on that the AArch32 ABI will produce the same effects as the ARMv5TEJ ABI of the ARM926EJ-S in the EV3.
    - **This has some potential.** *Adopt has ARMv8 32bit servers, but I'm not sure there's any that is targetted for testing workloads.*
- Run JTreg tests on ARMv7. This has been done and it has catched some regressions with OpenJDK's bundled test suite. However, it hasn't been automated. It depends also on the behaviour on ARMv5 not being different from ARMv7.
    - **This has some potential.** *Adopt has ARMv7 test build servers.*
- Run bootcycle-builds on ARMv8 or ARMv7. This is currently implemented. When the native build model is selected, the OpenJDK build is actually performed twice - first with the Host JDK and then with the newly built JDK. This makes sure that at least basic functionality is present; however once again, the behaviour might be different.
- Run JTreg tests on the EV3. This isn't possible due to the small amount of RAM on the EV3 (64 MB). The system will basically swap to death.
- Run JTreg tests split on the EV3 and on the PC. This alleviated a little of the memory pressure by building the tests on a powerful computer, only running them on the EV3. This also hasn't worked; it was still very slow and I think it eventually died due to swapping.
- Run JTreg tests on some ARMv5 board with more memory. This hasn't been implemented and it might fail too. There are some boards with 128 MB of RAM, which is however still quite little.
- Run Oracle TCK <somewhere>. This hasn't been implemented, as the TCK is not public. However this would provide a very wide test coverage.

Regarding out-of-tree patches

There are out-of-tree OpenJDK patches stored and used in this repository. This is due to some incompatibilities and bugs in the upstream codebase.

List of patches

JDK9:
- jdk9.patch - adds sflt workaround, fixes runtime errors & adds cflags optimizations.
- jdk9_lib.patch - adds Debian JNI library path to the build.

JDK10:
- jdk10.patch - adds sflt workaround, fixes runtime errors & adds cflags optimizations.
- jdk10_lib.patch - adds Debian JNI library path to the build.

JDK11:
- jdk11.patch - fixes runtime errors & adds cflags optimizations.
- jdk11_nosflt.patch - removes need for softfloat-3e.
- jdk11_lib.patch - adds Debian JNI library path to the build.
- jdk11_new.patch - fixes bugs found on JDK12 by JTreg incl. test on ARMv7.

- jdk11_bkpt.patch - disables ARM-specific breakpoint instruction on VM errors.
- jdk11_cds.patch - fixes crash (JTreg incl. test) caused by incorrect CDS data alignment.
- jdk11_jfr.patch - fixes crash (JTreg incl. test) in Java Flight Recorder file writer.

JDK12:

- jdk12_nosflt.patch - removes need for softfloat-3e.
- jdk12_new.patch - fixes bugs found on JDK12 by JTreg incl. test on ARMv7.
- jdk12_bkpt.patch - disables ARM-specific breakpoint instruction on VM errors.
- jdk12_cds.patch - fixes crash (JTreg incl. test) caused by incorrect CDS data alignment.
- jdk12_jfr.patch - fixes crash (JTreg incl. test) in Java Flight Recorder file writer.

JDK13:

- jdk13_nosflt.patch - removes need for softfloat-3e.
- jdk13_new.patch - fixes bugs found on JDK12 by JTreg incl. test on ARMv7.
- jdk13_bkpt.patch - disables ARM-specific breakpoint instruction on errors.
- jdk13_cds.patch - fixes crash (JTreg incl. test) caused by incorrect CDS data alignment.
- jdk13_jfr.patch - fixes crash (JTreg incl. test) in Java Flight Recorder file writer.

JDK14:

- jdk14_nosflt.patch - removes need for softfloat-3e.
- jdk14_new.patch - fixes bugs found on JDK12 by JTreg incl. test on ARMv7.
- jdk14_bkpt.patch - disables ARM-specific breakpoint instruction on errors.
- jdk14_cds.patch - fixes crash (JTreg incl. test) caused by incorrect CDS data alignment.
- jdk14_jfr.patch - fixes crash (JTreg incl. test) in Java Flight Recorder file writer.

JTreg included tests = tests bundled with OpenJDK source code.

Investigating regressions

Sometimes it is necessry to investigate a native crash. There is what would I probably do:

- Try to reproduce it on some more powerful machine (like with ARMv7 CPU). If it reproduces, continue testing on this machine.
- Try to reproduce it in QEMU (like in a Docker packaging container). If it reproduces, continue there.
- Somehow get the debug symbols for the Java runtime. The jri-11-ev3-dbgsym package from the repositories could do it. Alternatively, run a manual build to get a fastdebug or slowdebug build (value of JDKDEBUG environment variable) and try to reproduce the issue there. (Optionally, you can also build hsdis native disassembler for diagnosing JIT compiler issues. However, I have not tried this. It should give the ability to print JIT-generated ASM code; however, it is also possible to disassemble it from GDB.)
- Try to diagnose the issue with GDB - either by investigating the core dump generated by the crash (to enable them, run `ulimit -c unlimited`) or by triggering it in GDB. Another way is to start the GDB only once the crash happens (see https://neugens.wordpress.com/2015/02/26/debugging-the-jdk-with-gdb/, however I think the inner gdb command should be `gdb -p %p`).

- Try to discover the cause of the bug by walking through the stack trace and the source code.

Patch rebasing

As the development continues in the OpenJDK upstream, is is necessary to rebase the patches from time to time (or properly upstream them). This can be done this way:

1. Clone the JDK Git repositories on the correct tags:
   `git clone --branch jdk-11.0.4-ga --depth 1 https://github.com/openjdk/jdk11u.git && cd jdk11u`
2. Apply the current patches one by one as individual commits, fixing the conflicts that arise.
   `patch -p1 -i <patch1>`
   `rm <...>.orig`
   `git add <...>`
   `git commit -m "patch1"`
3. Print the commit log to get the hashes of the created commits.
   `git log # or git hist, but you have to add an alias`
4. Re-export the patches by running
   `git diff <previoushash> <thishash>  > <patch1>`
   between individual commits.

Upstreaming

To upstream a patch (see also OpenJDK guide):

- Sign the Oracle Contributor Agreement and send it to Oracle.
- Create an issue in the JDK Bug System. (might be optional)
- Work on the patch.
- Send it to an appropriate mailing list and iterate on enhancing it until it's OK.

I have been too lazy to do it.

Adding new OpenJDK versions

Upstream produces new OpenJDK releases in a half-year cadence. The *tip*/master build (masqueraded as latest version) on Jenkins should keep up with this, as master is always master.

To add a new released version, add an appropriate section in the config.sh shell script. It's probably best to start by copying an older release configuration and modifying it to match the new release configuration.

There is hopefully only one catch: the Host/Boot JDK provided should be of the same or previous release (e.g. for building JDK13, it should be JDK13 or JDK12). Usually AdoptOpenJDK builds are used for x86 builders and our builds are used for ARM builders.

To set up a Jenkins build, simply copy a job of the previous version and change the distro/openjdk version appropriately.