Yubo – Cmake, Docker and KAFKA

Compilation without Docker

Here is the file structure for YLibrary :

```
>> cd YLibrary; ll
- Test/src
- YLibrary/include/y/Book
  YLibrary/include/y/Config
  YLibrary/include/y/Event
  YLibrary/include/y/Hitter
- YLibrary/include/y/Oms
  YLibrary/src/Book
  YLibrary/src/Config
  YLibrary/src/Event
  YLibrary/src/Hitter
  YLibrary/src/Oms
- CMakeLists.txt
- README.MD
>> mkdir build;   cd build;
>> mkdir debug;   cd debug;   cmake -DCMAKE_BUILD_TYPE=Debug   ../..; make -j4; ./Test/y; cd ..    // compile and run test
>> mkdir release; cd release; cmake -DCMAKE_BUILD_TYPE=Release ../..; make -j4; ./Test/y          // compile and run test
```

Compilation with Docker

First of all, `git clone` the *RHEL Docker* repository, which has three folders :

- all following `.in` files are seeds (i.e. initial files) to be processed into runnable scripts or config files
- most scripts take two arguments, which are the versions for *RHEL* and `gcc` respectively
- in order to use `RHEL7.2` image downloaded from *RHEL*, we have to register a developer account in *RedHat*

```
>> git clone git@ygit.yubo.local:developer/rhel-docker.git
>> cd rhel-docker; ll
- base/build                    Python script for building RHEL7.2 + gcc10.2.0 image from RHEL7.2 image downloaded from RHEL
  base/Dockerfile.in            initial Docker file used by docker build
- local/build.sh                BASH script for building RHEL7.2 + gcc10.2.0 + YTL_user image from RHEL7.2 + gcc10.2.0 image
  local/Dockerfile              Docker file used by docker build
- project/configure             Python script for copying all utilities to YLibrary project folder
  project/bash.sh.in            utility to launch Docker container and run terminal
  project/build.sh.in           utility to launch Docker container and run both cmake/make
  project/make.sh               utility to run cmake followed by make
  project/build/debug.sh        final cmake command for debug (such as cmake -DCMAKE_BUILD_TYPE=Debug ../..)
  project/build/release.sh      final cmake command for release (such as cmake -DCMAKE_BUILD_TYPE=Release ../..)
```

Secondly, build `RHEL7.2 + gcc10.2.0` image :

```
>> cd rhel-docker/base;
>> ./build 7.2 10.2.0
prompt for RHEL developer account
prompt for RHEL developer password
```

Thirdly, build `RHEL7.2 + gcc10.2.0 + YTL_user` image specific for `YLibrary`

```
>> cd rhel-docker/local;
>> ./build.sh 7.2 10.2.0
```

After that, copy utilities to `YLibrary` project folder by `configure` :

```
>> cd rhel-docker/project;
>> ./configure
```

After that, you can find a new `scripts` folder generated inside `YLibrary` project (they are massaged and copied from the seeds in `rhel-docker/project`), we can then build `YLibrary` using *RHEL Docker* container (simulate the environment in production) :

```
>> cd YLibrary/scripts; ll
- YLibrary/scripts/bash.sh
- YLibrary/scripts/build.sh
- YLibrary/scripts/make.sh
- YLibrary/scripts/build/debug.sh
  YLibrary/scripts/build/release.sh
>> ./build debug                // cmake debug version and make it
>> ./build release              // cmake release version and make it
>> ./build release y Test       // cmake release version and make it, run Test exe
```

You can also run make in `nvim` by :

```
:set makeprg=scripts/build
:make debug
:make release
:make release y Test
```

Ouput will be shown in QuickFix.

Basic Docker

Docker concepts :

- *Docker* image - a file from which a container can be loaded
- *Docker* container - a box for running selected commands / tasks
- *Docker* registry - a *Github* for *Docker* image

```
// list all docker images, now we have 3 images
>> docker images
registry.redhat.io/rhel7:7.2      which is RHEL7.2 image
RHEL7.2:10.2.0                    which is RHEL7.2 + gcc10.2.0 image
YTL7.2:10.2.0                     which is RHEL7.2 + gcc10.2.0 + YTL_user image

// list all docker containers, now we have nothing, as we have not loaded any container yet
>> docker ps -a
nothing

// load docker container from image, then run desired command
>> docker run --rm --it registry.redhat.io/rhel7:7.2 bash
```

- option `--rm` means remove the container after completing the assigned task
- option `--it` means interactive mode, so that we can continue to use the terminal
- when *Docker* container is closed, all its files are gone, thus in order to save desired output ...
  we need to add mount points so that *Docker* containers can read / write files in local machine, using option `-v` and `-w`
- option `-v "$(pwd):$(pwd)"` means mounting current folder in local machine to current folder in container
- option `-w "$(pwd)"` means changing current directory to `$(pwd)`

```
>> docker run --rm -v "$(pwd):$(pwd)" -w "$(pwd)" ytl7.2:10.2.0 scripts/make.sh debug y Test
```

Basic scripting

`$(abc)`	run command `abc`
`$(pwd)`	run command `pwd` which returns current working directory
`$(nproc)`	run command `nproc` which returns the number of processors
`${@}`	all arguments of the `BASH` script
`${0}`	the zeroth argument of the `BASH` script (i.e. `BASH` script name)
`${1}`	the first argument of the `BASH` script
`${2}`	the second argument of the `BASH` script
`${#}`	number of arguments of the `BASH` script
`${?}`	return code of the latest command
`${abc}`	variable `abc`
`echo ${abc}`	print it
`${BASH_SOURCE[0]}`	same as `${0}` with minor differences, please check

Command `BASH` terminal command that are common in `BASH` scripting :

`>> shift`	pop away the first argument
`>> pushd path123`	equivalent to `cd path123` and push `pwd` into a directory stack
`>> popd`	equivalent to `cd directory_stack.front()` followed by pop
`>> dirname a/b/c`	return parent directory of input path, which is `a/b` in this case (`a/b/c` may not exist)

What's inside `YLibrary/scripts`?

There are 4/5 scripts in `scripts` folder. The first one `scripts/bash.sh` is an example of using *Docker* container. Here is its content :

```
#!/usr/bin/env bash
pushd "$(dirname "${BASH_SOURCE[0]}")/.." &> /dev/null
docker run --rm -it -v "$(pwd):$(pwd)" -w "$(pwd)" ytl7.2:10.2.0 bash
popd &> /dev/null
```

- line 1 specifies `bash` interpreter
- line 2 jumps up two layers above `bash.sh`, which is folder `YLibrary`
- line 3 loads *Docker* container, create mount point to folder `YLibrary` in local machine from exactly the same point in container
- line 4 goes back to `bash.sh`

The second one `scripts/build.sh` is similar to `scripts/bash.sh`, which is specific to `cmake` and `make` only :

```
#!/usr/bin/env bash
pushd "$(dirname "${BASH_SOURCE[0]}")/.." &> /dev/null
docker run --rm -v "$(pwd):$(pwd)" -w "$(pwd)" ytl7.2:10.2.0 scripts/make.sh ${@}
rc=${?}
popd &> /dev/null
exit ${rc}
```

- line 4 and 6 are extra (compared to `bash.sh`), they get the return code from `scripts/make.sh` and return
- line 3 includes `${@}`, which means forwarding all arguments of `build.sh` to `make.sh`
- in other words, `build.sh` is run in local machine, while `make.sh` is run in container

The third one `scripts/make.sh` is the one who invokes `(1)cmake` and `(2)make` :

```
#!/usr/bin/env bash
if [[ ${#} -lt 1 ]]; then                                    -lt means less-than
    exit 1                                                   red characters denote the IF-CONDITION
fi

BUILD_NAME="${1}"                                            BUILD_NAME =       debug,       release,       production
BUILD_DIR="build/${1}"                                       BUILD_DIR  = build/debug, build/release, build/production
shift                                                        pop away :         debug,       release,       production

pushd "$(dirname "${BASH_SOURCE[0]}")/.." &> /dev/null
mkdir -p "${BUILD_DIR}"
pushd "${BUILD_DIR}" &> /dev/null

# step 1
ulimit -c unlimited                                          set coredump limit for Test
CMAKE="../../scripts/build/${BUILD_NAME}.sh"
cat "${CMAKE}" | grep cmake                                  display the command to terminal or nvim's quickfix
"${CMAKE}"                                                   execute the command
rc=${?}                                                      return code of the command
if [[ ${rc} -ne 0 ]]; then                                   return code equals to zero for success, quits if cmake fails
    exit ${rc}
fi

# step 2
make -j ${nproc} ${@}
rc=${?}
if [[ ${rc} -ne 0 ]]; then
    exit ${rc}
fi

popd &> /dev/null
popd &> /dev/null
```

Finally, `scripts/build/*.sh` is just `cmake` invocation :

```
#!/usr/bin/env bash
cmake -DCMAKE_C_COMPILER=$(which gcc)
      -DCMAKE_CXX_COMPILER=$(which g++)
      -DCMAKE_BUILD_TYPE=Debug ../..
```

KAFKA

For topology of KAFKA, please read `http://cloudurable.com/blog/kafka-architecture/index.html`

How to install and run KAFKA?

1. visit `https://kafka.apache.org/quickstart` and download KAFKA
2. unzip and install as follows ...

```
>> cd ~/Downloads
>> tar -xzf kafka_2.13-2.8.0.tgz
```

Zookeeper is for :

- coordination, management of KAFKA servers (clusters)
- including load balancing
- including recovery

KAKFA is for :

- KAFKA server = cluster
- KAFKA client = producer / client
- producer = publisher
- consumer = subscriber

Lets view and modify config :

```
>> cd kafka_2.13-2.8.0
>> nvim config/zookeeper.properties (nothing changed)
>> nvim config/server.properties
```

Add two lines at the end of server.properties :

```
auto.create.topics.enable = false
delete.topic.enable = true
```

Steps in sequence in different terminal (you can use `tmux`) :

1. start zookeeper in terminal 0
2. start KAFKA server in terminal 1
3. create KAFKA topics in terminal 2 (terminal 2 can be re-used)
4. start client console – producer in terminal 2
5. start client console – consumer in terminal 3 (both producer and consumer are KAFAK clients)

All the executables (bash scripts) are available here :

```
>> ll bin
```

Step 1&2 : Run zookeeper and KAFKA server with appropriate config and options :

```
>> bin/zookeeper-server-start.sh config/zookeeper.properties
>> bin/kafka-server-start.sh     config/server.properties
```

Step 3 : Create multiple topics

```
>> bin/kafka-topics.sh --bootstrap-server localhouse:9092 \      (identifier of KAFKA server)
                       --create \
                       --topic my_topic_ABC \
                       --replication-factor 1 \
                       --partitions 1
>> bin/kafka-topics.sh --bootstrap-server localhouse:9092 \      (identifier of KAFKA server)
                       --list                                    (list all topics)
>> bin/kafka-topics.sh --bootstrap-server localhouse:9092 \      (identifier of KAFKA server)
                       --describe --topic my_topic_ABC           (list one topic)
```

Step 4&5 : Start two client instances, one for producer and one for client :

```
>> bin/kafka-console-producer.sh --bootstrap-server localhouse:9092 \
                                 --topic my_topic_ABC
>> bin/kafka-console-consumer.sh --bootstrap-server localhouse:9092 \
                                 --topic my_topic_ABC
>> bin/kafka-console-consumer.sh --bootstrap-server localhouse:9092 \
                                 --topic my_topic_ABC \
                                 --group group_name_DEF
```

where :

`--bootstrap-server`	specifies ip:port of KAFKA server (port 9092 can be found in config)
`--create`	option to create topic
`--list`	option to list topic

Stop clients by ctrl-c and stop servers by :

```
>> bin/kafka-server-stop.sh
>> bin/zookeeper-server-stop.sh
```

KAFKA vs database

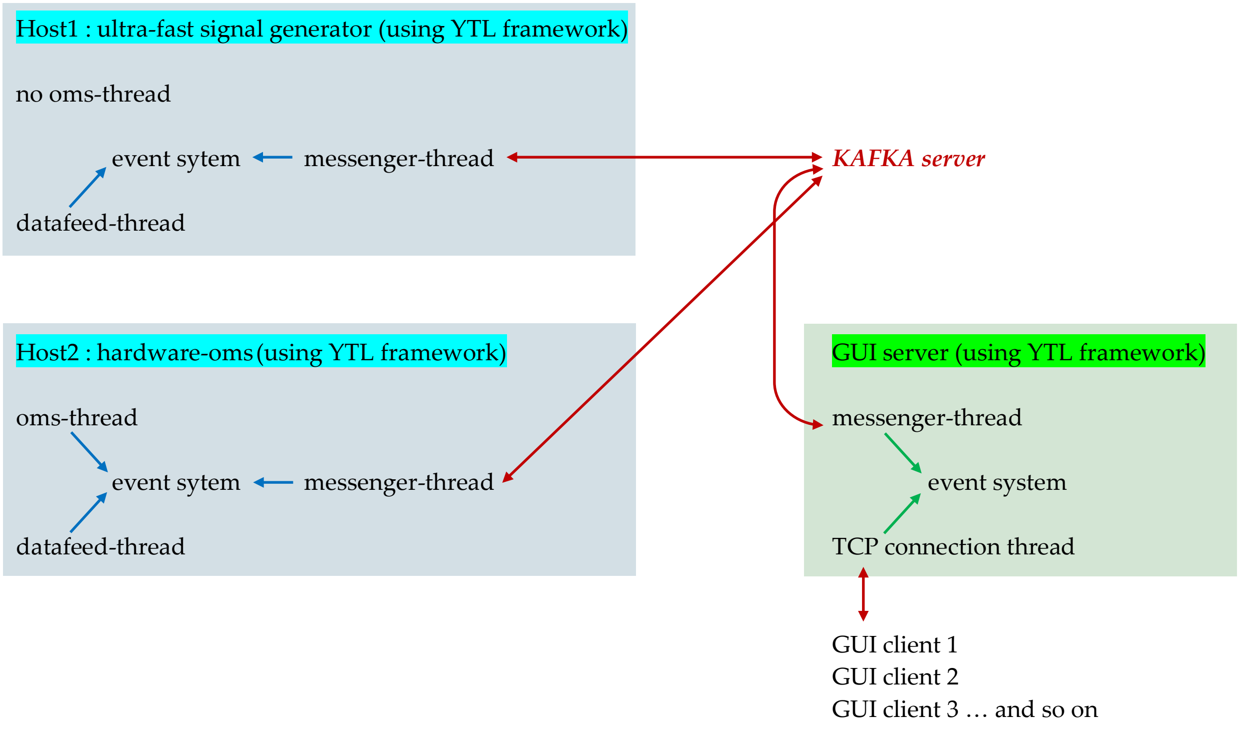
About database concepts :
ACID = xxx-ability : atomicity / consistency / isolation / durability
CRUD = xxx-operation : create / read / update / delete

```
CRUD     SQL      HTTP    STL
------------------------------------
create   insert   post    insert
read     select   get     operator[] const
update   update   put     operator[]
delete   delete   delete  erase
```

ELK stack =
Elastic search +
Log stash +
Kibana

Here is the diagram showing integration of :

- event system (inter-threads in same machine) with
- KAFKA pub-sub sytem (inter-machines)



In the above architecture, there are two hosts :

- signal generator can estimate theoretical-value of underlying in ultra-fast way making use of TCP checksum to price mapping
- hardware-oms can emit buy and sell order when market price is more favorable to theoretical-value from signal generator

Please trace the route :

- from GUI client to GUI server to KAFKA and finally to host
- from host to KAFKA to GUI server and finally back to GUI client
- on both directions, it involves several event systems