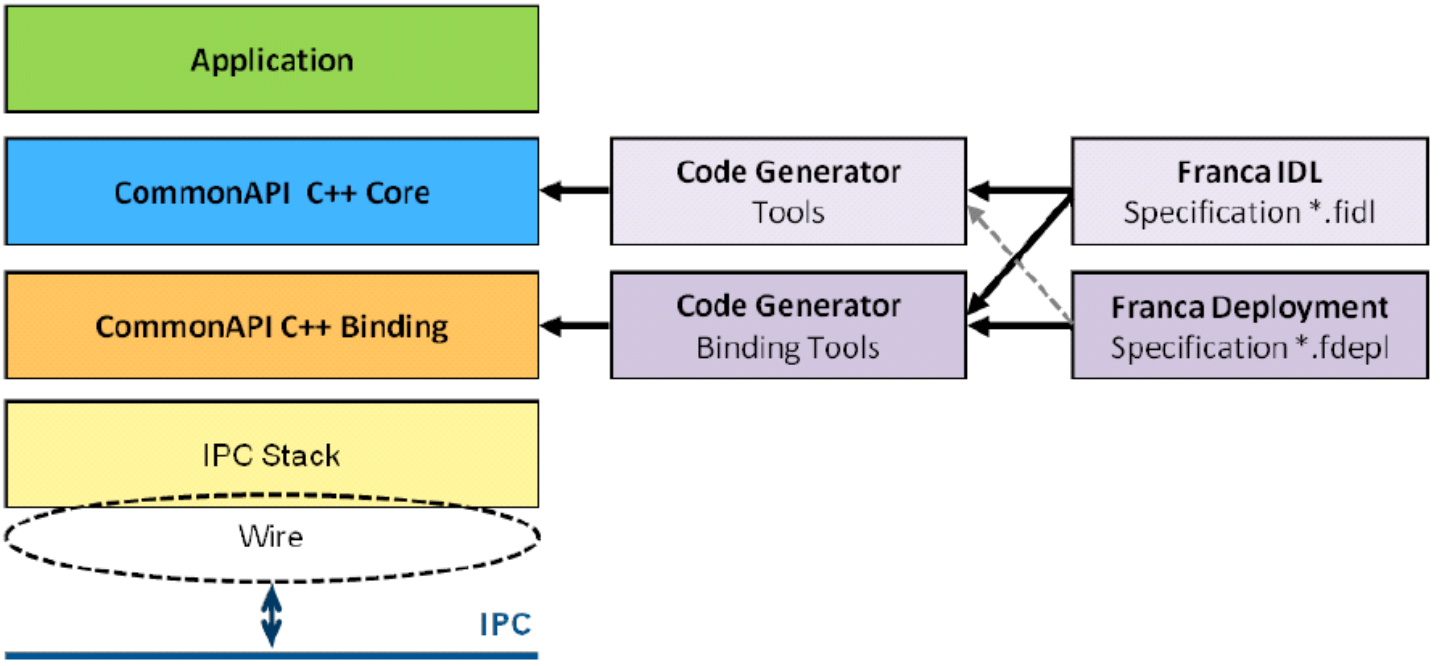
简介

Tuesday, February 18, 2020 9:14 AM

简介：

CommonAPI C++是一个用于开发分布式应用的标准的C++ API规范，该分布式应用通过一个中间件实现进程间通信。其基本原理如下：

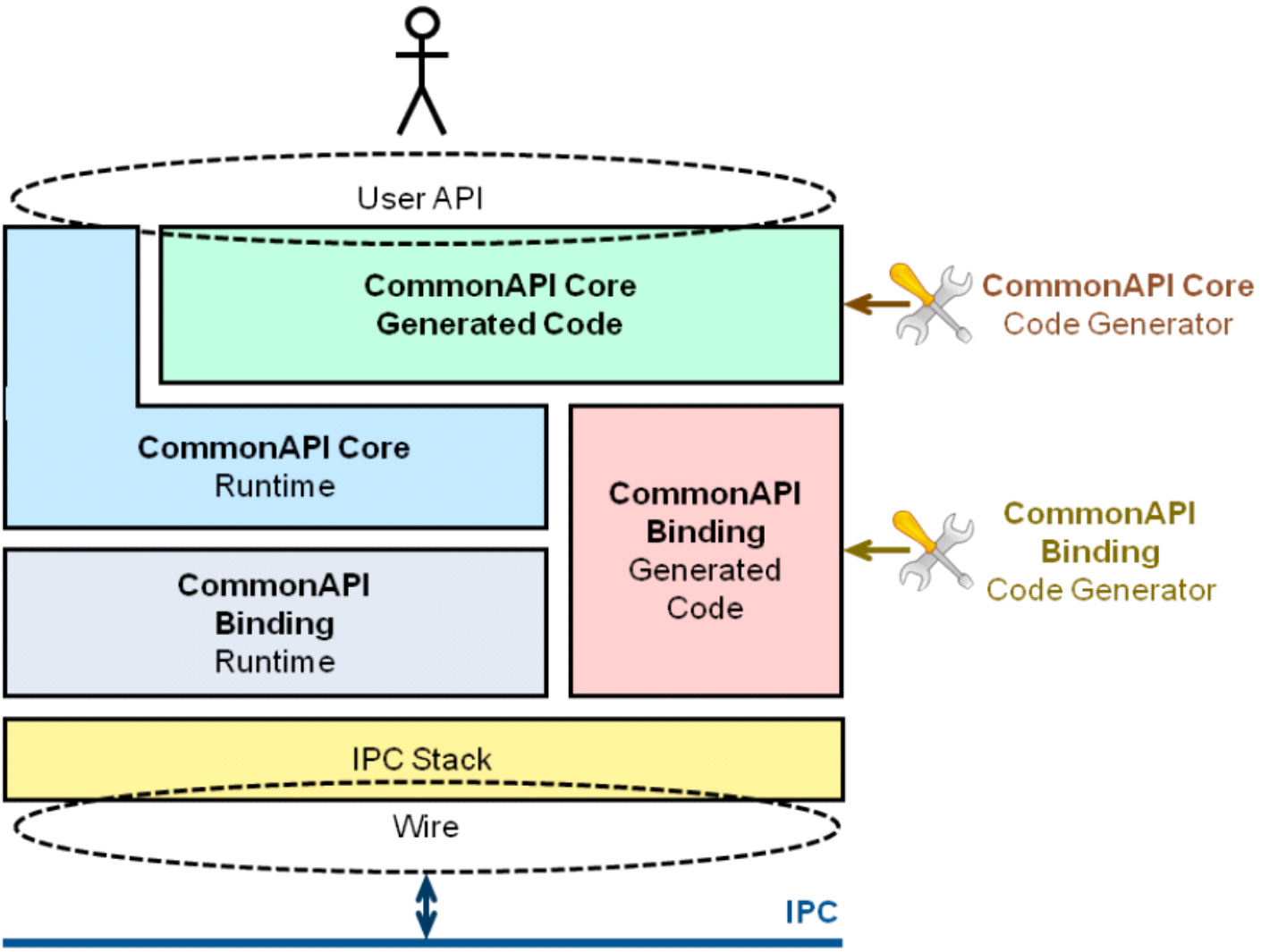


CommonAPI C++由两部分组成：中间件无关部分(CommonAPI Core)和中间件相关部分(CommonAPI Binding)；
CommonAPI使用接口描述语言FrancalIDL来指定接口。从FrancalIDL产生代码的代码生成器是CommonAPI的一部分；
CommonAPI C++ Bindings的代码生成器需要根据中间件的不同指定对应的参数(deployment parameters，部署参数)。这些参数定义在Franca部署文件中(*.fdepl)。
注意：CommonAPI C++核心没有强制的部署参数。但事实证明，向CommonAPI C++核心添加额外的部署参数是有意义的。

用户：

commonAPI的用户可分为两部分：

1. 基于FrancalIDL的部分，其中包含与FrancalIDL文件的类型、属性和方法相关的API函数；
2. 公共部分(即运行时API部分)，它包含用于加载运行时环境、创建代理等的API函数。



开发流程：

应用开发者的工作流程是：

1. 根据带有属性和方法的接口规范创建FrancalIDL文件；
2. 使用CommonAPI代码生成器为客户端和服务器生成代码；
3. 通过实现生成的框架中的方法来实现服务端；
4. 通过创建代理和调用这个代理中的方法来实现客户端。

集成指南

Tuesday, February 18, 2020 10:56 AM

需求：

1. 编译器需要支持C++11（如gcc4.8）；
2. 构建系统是CMake，需要高于2.8.2的Cmake；
3. 此指南仅描述CommonAPI公共部分的集成指南，具体Binding的集成请参考具体Binding的集成指南；
4. 代码生成器的构建工具链是Maven，Maven的版本需要高于3。

编译运行时：

1. 下载代码
   git clone https://github.com/GENIVI/capicxx-core-runtime.git
2. 命令行编译:

```
$ cd capicxx-core-runtime
$ mkdir build
$ cd build
$ cmake -D CMAKE_INSTALL_PREFIX=./out ..
$ make
$ make install
```

3. 编译完之后会在./out目录下生成对应的头文件和库文件。
4. 也可从http://genivi.github.io/capicxx-core-tools/下载编译好的版本。

额外的cmake参数；

-DBUILD_SHARED_LIBS	ON/OFF	OFF时生成静态库,默认ON
-DCMAKE_BUILD_TYPE	Release/Debug	默认Debug
-DCMAKE_INSTALL_PREFIX	目录名	安装目录名
-DUSE_INSTALLED_COMMONAPI	OFF/ON	
-DMAX_LOG_LEVEL	ERROR/WARNING/INFO/DEBUG/VERBOSE	低于对应等级的log将被忽略

make目标:

make all	和make一样，编译和链接CommonAPI
make clean	删除二进制文件，但不会删除由cmake产生的中间文件
make maintainer-clean	删除构建目录下的所有文件

make install	拷贝库文件和头文件到安装目录
make DESTDIR=<install_dir> install	

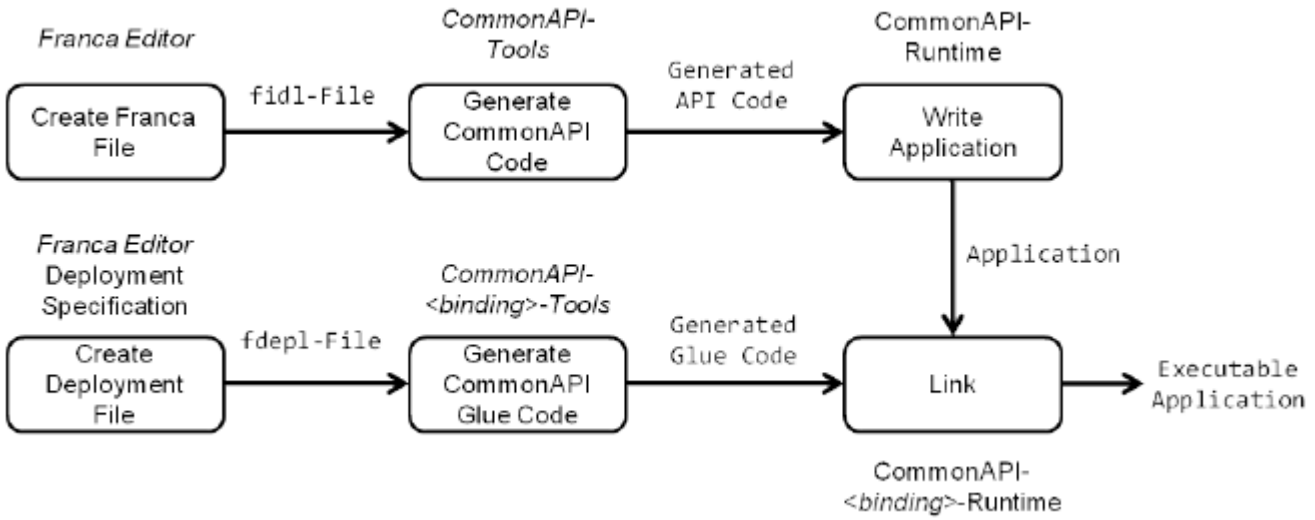
编译代码生成工具：

可以从http://genivi.github.io/capicxx-core-tools/网站上直接下载编译好的代码生成工具，也可以自己编译代码生成工具，编译方法如下：

1. 下载代码：
   git clone https://github.com/GENIVI/capicxx-core-tools.git
2. 命令行编译：
   cd capicxx-core-tools
   cd org.genivi.commonapi.core.releng
   mvn -Dtarget.id=org.genivi.commonapi.core.target clean verify
3. 编译完成后可以在org.genivi.commonapi.core.cli.product/target/products/下找到基于命令行的代码生成工具的压缩文件commonapi-generator.zip

编写应用程序：

开发基于CommonAPI的可执行程序的基本流程如下图：



1. 生成代码：
   commonapi_generator [options] file [file...]
   注意： 如果你的CommonAPI绑定需要部署文件，则还需要以对应部署文件(该文件import了fidl文件)作为输入，使用对应绑定的代码生成工具生成粘合代码。某些绑定(如D-Bus)不需要使用部署文件,在这种情况下，只需要用fidl文件作为输入来启动代码生成器。
2. 构建应用程序：
   应用那程序需要编译生成的CommonAPI代码，并链接CommonAPI的运行时库。
   可以参考capicxx-core-tools/CommonAPI-Examples目录下的示例快速上手。

交叉编译boost

Wednesday, February 19, 2020 10:48 AM

ubuntu下交叉编译boost步骤：

1. 下载boost发行版： 从www.boost.org下载boost发行版，目前最新的是1.72.0，下载文件是boost_1_72_0.tar.bz2；
2. 解压1中下载的文件: tar -jxvf boost_1_72_0.tar.bz2;
3. 进入解压后的文件夹,运行./bootstrap.sh;
4. 修改3步中生成的project-config.jam,在以下行中指定交叉编译的gcc:
   ```
   if ! gcc in [ feature.values <toolset> ]
   {
      using gcc ;
   }
   ```
   将其中的using gcc;行修改为如下类似的消息：
   ```
   using gcc : : /usr/local/linaro-aarch64-2018.08-gcc8.2/bin/aarch64-linux-gnu-gcc ;
   ```
   注意 “：” 后面及";"前面的空格
5. 使用如下指令编译：
   ```
   ./b2 install --prefix=./out/boost/
   ```
   若只编译静态库： ./b2 cxxflags=-fPIC cflags=-fPIC --prefix=./out/boost/ link=static runtime-link=static install
   编译完成后，会在./out/boost/下面看到生成的头文件目录和库文件目录

交叉编译vsomeip

Wednesday, February 19, 2020 2:56 PM

ubuntu下交叉编译vsomeip的步骤：

1. 为交叉编译工具编写toolchain.cmake，该文件中需要指定编译器及ROOT PATH，如ambarella的toolchain.cmake内容如下：

```
# this is required
set(CMAKE_SYSTEM_NAME "Linux")

# specify the cross compiler
set(CMAKE_C_COMPILER "/usr/local/linaro-aarch64-2018.08-gcc8.2/bin/aarch64-linux-gnu-gcc")
set(CMAKE_CXX_COMPILER "/usr/local/linaro-aarch64-2018.08-gcc8.2/bin/aarch64-linux-gnu-g++")

#where is the target environment
set(CMAKE_FIND_ROOT_PATH "/usr/local/linaro-aarch64-2018.08-gcc8.2/aarch64-linux-gnu/debug-root")

set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE ONLY)

add_definitions("-DAARCH64")
```

2. 下载vsomeip源码：
   git clone https://github.com/GENIVI/vsomeip.git
3. 进入vsomeip目录，修改CMakelist.txt中boost有关的部分：
   将以下内容注释掉：

```
#find_package( Boost 1.55 COMPONENTS system thread log REQUIRED )
#if(Boost_FOUND)
#  if(Boost_LIBRARY_DIR)
#    MESSAGE( STATUS "Boost_LIBRARY_DIR not empty using it: ${Boost_LIBRARY_DIR}" )
#  else()
#    if(BOOST_LIBRARYDIR)
#      MESSAGE( STATUS "Boost_LIBRARY_DIR empty but BOOST_LIBRARYDIR is set setting Boost_LIBRARY_DIR to: ${BOOST_LIBRARYDIR}" )
#      set(Boost_LIBRARY_DIR ${BOOST_LIBRARYDIR})
#    endif()
#  endif()
#else()
#  MESSAGE( STATUS "Boost was not found!")
#endif()

# cmake 3.15 introduced a new variable and a new format for the old one
#if (DEFINED Boost_VERSION_MACRO)
#  set(VSOMEIP_BOOST_VERSION ${Boost_VERSION_MACRO})
#else()
```

```
#  set(VSOMEIP_BOOST_VERSION ${Boost_VERSION})
#endif()
```

新增以下boost相关的内容：

```
add_definitions(-DBOOST_ALL_NO_LIB=1)
message ("Boost_INCLUDE_DIR:" ${Boost_INCLUDE_DIR})
message ("Boost_LIBRARY_DIR:" ${Boost_LIBRARY_DIR})
message ("Boost_VERSION:" ${Boost_VERSION})
include_directories( ${Boost_INCLUDE_DIR} )
link_directories(
   ${Boost_LIBRARY_DIR}
)
set(VSOMEIP_BOOST_VERSION ${Boost_VERSION})
set(Boost_LIBRARIES boost_log boost_system boost_thread boost_filesystem boost_date_time)
```

另外，还需要把以下的目录注释掉：

```
#add_subdirectory( examples/routingmanagerd )
```

4. 执行以下命令：

```
mkdir build
cd build
cmake -DCMAKE_TOOLCHAIN_FILE=../toolchain.cmake \
        -DCMAKE_BUILD_TYPE=Debug \
        -DCMAKE_CXX_FLAGS=-std=c++11 -frtti -fPIC -v -fexceptions \
        -DCMAKE_INSTALL_PREFIX=../out/vsomeip/ \
        -DBoost_INCLUDE_DIR=../out/boost/include/  \
        -DBoost_LIBRARY_DIR=../out/boost/lib/  \
        -DBoost_VERSION=107200                    \
        ..
make
make install
```

交叉编译capicxx-core-runtime

Thursday, February 20, 2020 10:33 AM

ubuntu下交叉编译capicxx-core-runtime的步骤：

1. 编写toolchain.cmake,参考《交叉编译vsomeip》的第一步；
2. 下载capicxx-core-runtime源码：
   git clone https://github.com/GENIVI/capicxx-core-runtime.git
3. 进入capicxx-core-runtime目录，执行以下命令编译：

```
mkdir build
cd build
cmake -DCMAKE_TOOLCHAIN_FILE=../toolchain.cmake \
      -DCMAKE_BUILD_TYPE=Debug \
      -DCMAKE_CXX_FLAGS=-std=c++11 -frtti -fPIC -v -fexceptions \
      -DCMAKE_INSTALL_PREFIX=../out/capicxx-core-runtime/ \
      ..
make
make install
```

交叉编译capicxx-someip-runtime

Thursday, February 20, 2020 10:50 AM

ubuntu下交叉编译capicxx-someip-runtime的步骤：

1. 编写toolchain.cmake,参考《交叉编译vsomeip》的第一步；
2. 下载capicxx-someip-runtime源码：
   git clone https://github.com/GENIVI/capicxx-someip-runtime.git
3. 进入capicxx-someip-runtime目录，修改CMakelist.txt：

   COMMONAPI相关：

   注释掉以下这行的内容：

   ```
   FIND_PACKAGE(CommonAPI 3.1.12 REQUIRED CONFIG NO_SYSTEM_ENVIRONMENT_PATH
   NO_CMAKE_SYSTEM_PATH)
   ```

   新增：

   ```
   link_directories(${COMMONAPI_LIBRARY_DIRS})
   ```

   BOOST相关：

   注释掉以下这行的内容：

   ```
   link_directories(${COMMONAPI_LIBRARY_DIRS})
   ```

   vsomip相关：

   注释掉以下这行的内容：

   ```
   link_directories(${COMMONAPI_LIBRARY_DIRS})
   ```

   新增：

   ```
   link_directories(${COMMONAPI_LIBRARY_DIRS})
   ```

   修改以下这行：

   ```
   target_link_libraries (CommonAPI-SomeIP CommonAPI vsomeip)
   ```

   为：

   ```
   target_link_libraries (CommonAPI-SomeIP CommonAPI vsomeip3)
   ```

4. 执行以下命令编译：

   ```
   mkdir build
   cd build
   cmake -DCMAKE_TOOLCHAIN_FILE=../toolchain.cmake  \
           -DUSE_INSTALLED_COMMONAPI=OFF   \
           -DCMAKE_BUILD_TYPE=Debug  \
           -DCMAKE_CXX_FLAGS=-std=c++11 -frtti -fPIC -v -fexceptions       \
           -DCMAKE_INSTALL_PREFIX=../out/capicxx-someip-runtime/       \
           -DCOMMONAPI_INCLUDE_DIRS=/home/walker/work/commonapi/out/capicxx-core-
           time/include/CommonAPI-3.1/  \
           -DCOMMONAPI_LIBRARY_DIRS=/home/walker/work/commonapi/out/capicxx-core-
           time/lib/          \
           -DBoost_INCLUDE_DIR=/home/walker/work/commonapi/out/boost/include/     \
           -DBoost_LIBRARY_DIR=/home/walker/work/commonapi/out/boost/lib/ \
           -DBoost_VERSION=107200                  \
           -DVSOMEIP_INCLUDE_DIRS=/home/walker/work/commonapi/out/vsomeip/include/compat
                   \
           -DVSOMEIP_LIBRARY_DIRS=/home/walker/work/commonapi/out/vsomeip/lib/  \
           ..
   make
   make install
   ```

编译capicxx-core-tools

Thursday, February 20, 2020 3:02 PM

可以从http://genivi.github.io/capicxx-core-tools/网站上直接下载编译好的代码生成工具，也可以自己编译代码生成工具，编译方法如下：

1. 下载代码：
   git clone https://github.com/GENIVI/capicxx-core-tools.git
2. 命令行编译：
   cd capicxx-core-tools
   cd org.genivi.commonapi.core.releng
   mvn -Dtarget.id=org.genivi.commonapi.core.target clean verify
3. 编译完成后可以在org.genivi.commonapi.core.cli.product/target/products/下找到基于命令行的代码生成工具的压缩文件commonapi-generator.zip

编译capicxx-someip-tools

Thursday, February 20, 2020 3:35 PM

可以从http://genivi.github.io/capicxx-someip-tools/网站上直接下载编译好的代码生成工具，也可以自己编译代码生成工具，编译方法如下：

1. 下载代码：
   git clone https://github.com/GENIVI/capicxx-someip-tools.git
2. 命令行编译：
   cd capicxx-someip-tools
   cd org.genivi.commonapi.someip.releng
   mvn -DCOREPATH=/home/walker/work/commonapi/ok/capicxx-core-tools/ -Dtarget.id=org.genivi.commonapi.someip.target clean verify
3. 编译完成后可以在org.genivi.commonapi.someip.cli.product/target/products/下找到基于命令行的代码生成工具的压缩文件commonapi_someip_generator.zip

基于CommonAPI-SomeIP的应用开发

Thursday, February 20, 2020 5:44 PM

参考：https://github.com/GENIVI/capicxx-someip-tools/wiki/CommonAPI-C---SomeIP-in-10-minutes

1. 构建CommonAPI运行时库，参考《交叉编译capicxx-core-runtime》；
2. 编译boost库，参考《交叉编译boost》；
3. 编译vsomeip库，参考《交叉编译vsomeip》；
4. 构建CommonAPI SOME/IP运行时库，参考《交叉编译capicxx-someip-runtime》；
5. 编写Franca文件并生成代码：

   编写HelloWorld.fidl文件，其内容如下：

```
package commonapi
interface HelloWorld {
  version {major 1 minor 0}
  method sayHello {
    in {
      String name
    }
    out {
      String message
    }
  }
}
```

   使用commonapi代码生成工具（参考《编译capicxx-core-tools》）生成代码：

   ../../commonapi-generator/commonapi-generator-linux-x86_64 ./HelloWorld.fidl

   生成的代码在./src-gen目录下。

6. 编写部署文件并生成代码：

   编写HelloWorld.fdepl文件，其内容如下：

```
import "platform:/plugin/org.genivi.commonapi.someip/deployment/CommonAPI-SOMEIP_deployment_spec.fdepl"
import "HelloWorld.fidl"

define org.genivi.commonapi.someip.deployment for interface commonapi.HelloWorld {
        SomeIpServiceID = 4660

        method sayHello {
                SomeIpMethodID = 123
        }
}

define org.genivi.commonapi.someip.deployment for provider MyService {
        instance commonapi.HelloWorld {
                InstanceId = "test"
                SomeIpInstanceID = 22136
        }
```

```
}
```

使用commonapi_someip代码生成工具生成部署代码：

```
../../commonapi_someip_generator/commonapi-someip-generator-linux-x86_
64 ./HelloWorld.fdepl
```

生成的代码在./src-gen目录下。

7. 开发客服端应用程序

编写HelloWorldClient.cpp文件，内容如下：

```
#include <iostream>
#include <string>
#include <unistd.h>
#include <CommonAPI/CommonAPI.hpp>
#include <v1/commonapi/HelloWorldProxy.hpp>

using namespace v1_0::commonapi;

int main() {
   std::shared_ptr < CommonAPI::Runtime > runtime = CommonAPI::Runtime::get();
   std::shared_ptr<HelloWorldProxy<>> myProxy = runtime->buildProxy<HelloWorldProxy>
("local", "test");

       std::cout << "Checking availability!" << std::endl;
   while (!myProxy->isAvailable())
      usleep(10);
       std::cout << "Available..." << std::endl;

       CommonAPI::CallStatus callStatus;
       std::string returnMessage;
       myProxy->sayHello("Bob", callStatus, returnMessage);
       std::cout << "Got message: '" << returnMessage << "'\n";
       return 0;
}
```

8. 开发服务端应用程序

编写HelloWorldService.cpp文件，内容如下：

```
#include <iostream>
#include <thread>
#include <CommonAPI/CommonAPI.hpp>
#include "HelloWorldStubImpl.hpp"
using namespace std;
int main() {
    std::shared_ptr<CommonAPI::Runtime> runtime = CommonAPI::Runtime::get();
    std::shared_ptr<HelloWorldStubImpl> myService =
       std::make_shared<HelloWorldStubImpl>();
    runtime->registerService("local", "test", myService);
    std::cout << "Successfully Registered Service!" << std::endl;
while (true) {
        std::cout << "Waiting for calls... (Abort with CTRL+C)" << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(30));
    }
    return 0;
 }
```

编写服务实现头文件HelloWorldStubImpl.hpp,内容如下：

```
#ifndef HELLOWORLDSTUBIMPL_H_
#define HELLOWORLDSTUBIMPL_H_
#include <CommonAPI/CommonAPI.hpp>
#include <v1/commonapi/HelloWorldStubDefault.hpp>
class HelloWorldStubImpl: public v1_0::commonapi::HelloWorldStubDefault {
public:
    HelloWorldStubImpl();
    virtual ~HelloWorldStubImpl();
    virtual void sayHello(const std::shared_ptr<CommonAPI::ClientId> _client,
       std::string _name, sayHelloReply_t _return);
};
#endif /* HELLOWORLDSTUBIMPL_H_ */
```

编写服务实现源文件HelloWorldStubImpl.cpp，内容如下：

```
#include "HelloWorldStubImpl.hpp"
HelloWorldStubImpl::HelloWorldStubImpl() { }
HelloWorldStubImpl::~HelloWorldStubImpl() { }
void HelloWorldStubImpl::sayHello(const std::shared_ptr<CommonAPI::ClientId> _client,
       std::string _name, sayHelloReply_t _reply) {
           std::stringstream messageStream;
           messageStream << "Hello " << _name << "!";
           std::cout << "sayHello('" << _name << "'): '" << messageStream.str()
<< "'\n";
_reply(messageStream.str());
};
```

9. 编译应用程序：

编写CMakeLists.txt文件，然后使用cmake编译：

```
cmake_minimum_required(VERSION 2.8)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -pthread -std=c++0x")
include_directories(
    src-gen
    $ENV{RUNTIME_PATH}/capicxx-core-runtime/include
    $ENV{RUNTIME_PATH}/capicxx-someip-runtime/include
    $ENV{RUNTIME_PATH}/vsomeip/interface
)
link_directories(
    $ENV{RUNTIME_PATH}/capicxx-core-runtime/build
    $ENV{RUNTIME_PATH}/capicxx-someip-runtime/build
    $ENV{RUNTIME_PATH}/vsomeip/build
)
add_executable(HelloWorldClient
       src/HelloWorldClient.cpp
       src-gen/v1/commonapi/HelloWorldSomeIPProxy.cpp
       src-gen/v1/commonapi/HelloWorldSomeIPDeployment.cpp
)
target_link_libraries(HelloWorldClient CommonAPI CommonAPI-SomeIP vsomeip)
add_executable(HelloWorldService
       src/HelloWorldService.cpp
       src/HelloWorldStubImpl.cpp
       src-gen/v1/commonapi/HelloWorldSomeIPStubAdapter.cpp
```

```
        src-gen/v1/commonapi/HelloWorldStubDefault.cpp
        src-gen/v1/commonapi/HelloWorldSomeIPDeployment.cpp
)
target_link_libraries(HelloWorldService CommonAPI CommonAPI-SomeIP vsomeip)
```

使用如下命令编译：

```
mkdir build
cd build
cmake -DCMAKE_TOOLCHAIN_FILE=../toolchain.cmake  \
        -DCMAKE_BUILD_TYPE=Debug  \
        -DCMAKE_CXX_FLAGS=-std=c++11 -frtti -fPIC -v -fexceptions       \
        -DCMAKE_INSTALL_PREFIX=../out/test/ \
        ..
make
make install
```