

**dynatrace**  
2021 DEVOPS REPORT

22% of DevOps leaders **sacrifice code quality** to innovate faster.

[Download report](#)

## Android HIDL learning-HelloWorld introduction (compilation 1)

### Android HIDL learning (finishing 1)

- Overview
- Get ready
- Example application
  - 1. HIDL interface file definition
  - 2. Generate HAL related files
  - 3. Implement the shared library of the HAL server
  - 4. Start the registration procedure on the Hal server
  - 5. HIDL Client test program
  - 6. Test case

### Overview

HAL interface definition language (HIDL for short, pronounced "hide-I") is an interface description language (IDL) used to specify the interface between HAL and its users. HIDL allows specifying types and method calls (which will be aggregated into interfaces and software packages). In a broader sense, HIDL is a system for communicating between code bases that can be compiled independently.

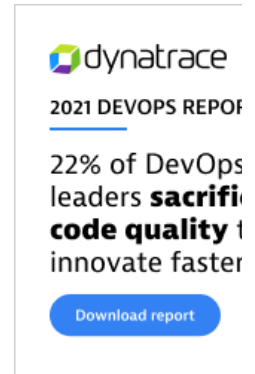
HIDL is intended for inter-process communication (IPC). The communication between processes uses the Binder mechanism. For code bases that must be associated with processes, you can also use pass-through mode (not supported in Java).

HIDL can specify data structures and method signatures. These contents will be sorted into interfaces (similar to classes), and interfaces will be assembled into software packages. Although HIDL has a series of different keywords, C++ and Java programmers are no strangers to the syntax of HIDL. In addition, HIDL also uses Java-style comments. You can learn more about the official reference link (<https://source.android.google.cn/devices/architecture/hidl>)

### Get ready

Programmers have a hobby. No matter what new knowledge they are learning, they like to start with HelloWorld as a simple example, and we are no exception. OK, we are all dry goods here, so let's not talk more nonsense, we still need some preparations and thresholds to learn HIDL.

Ready to work:



**dynatrace**  
2021 DEVOPS REPORT

22% of DevOps leaders **sacrifice code quality** to innovate faster.

[Download report](#)



**norton**

**Miljontals människor världen över litar på**

- Antivirus
- Integritet på nätet
- Dark Web Monitoring

[Mer information](#)



**norton**

**Miljontals människor världen över litar på**

- Antivirus
- Integritet på nätet
- Dark Web Monitoring

[Mer information](#)



**norton**

**Miljontals människor världen över litar på**

- Antivirus
- Integritet på nätet
- Dark Web Monitoring

[Mer information](#)

Android BSP compilation environment, need to be reorganized

BSP code for Android devices

Android device, used to run test code

I am using a device from MTK, based on the Android O system platform for development.

Of course, if you don't have a device on hand, you can also use the Android emulator for development. The image of the Android emulator can be compiled with the official AOSP code, but note that if you use the emulator, the kernel has to download the goldfish code, I won't go into details here, you can google to find out.

### Example application

Next we start the first HIDL application example, we need to give this simple example a great name, my name is **Naruto** here, don't ask me why, brother is a hardcore **Naruto** fan, haha, that's it. It's called Naruto, so let's tell a story:

We are going to write a HAL for Android. Don't confuse the original intention. Let's take a look at what HALs AOSP has:

- Camera
- Audio
- Sensor
- Wait, [VIEW IMAGE \(HTTPS://IMG-BLOG.CSDNIMG.CN/20200727153122592.PNG\)](https://img-blog.csdnimg.cn/20200727153122592.PNG)

these are all hardware on Android devices, because Google in theory only cares about the Android framework layer and upper layer software, but the upper layer software depends on the underlying hardware implementation, but each mobile phone manufacturer, or the bottom layer of the CPU manufacturer The hardware implementations are all different, so this HAL layer is basically implemented by mobile phone manufacturers or CPU manufacturers. Google is only used as a framework to guide and define the API interface of the Framework layer. The implementation of these interfaces must be implemented by HAL. carry out.

So our Naruto shoulders this important task. It controls the underlying hardware. The underlying hardware is controlled by the Linux kernel driver. You can simply control the driver by providing file reading and writing. Let's do the virtual driver here, and omit it. Implementation of kernel driver;

Android HIDL learning-HelloWorld introduction (compilation 1) - Karthi softeK  
 Wait, ours is HelloWorld. Well, Naruto, you can provide a HelloWorld interface. It's overkill. The purpose is to let us know the process of writing a HIDL is enough!!

## 1. HIDL interface file definition

Enter the code, we assume that Naruto is the HAL of the standard AOSP, we will rub the code into the standard HAL (hardware/interface) layer, enter the code directory to create the HIDL directory:

```
mkdir -p hardware/interfaces/naruto/1.0/default
```

Then create the interface description file INaruto.hal and place it in the 1.0/directory you just created:

```
//INaruto.hal

Package Penalty for Android . Hardware . Naruto @ 1.0;

interface INaruto {
    helloWorld( string name ) generates ( string result );
};
```

Yes, this is a language format defined by Google for specific data type conversion. Please check the corresponding relationship

(<https://source.android.google.cn/devices/architecture/hidl-cpp/types>) . The combination of C++ and Java. I believe that we are using Android BSP to send it. What language does not, right?

- Assembly: may be used in bootloader and kernel
- C language: don't you know how to play with Mao's Linux Kernel?
- C++: You can't do that, so don't engage in the underlying development of Android, HAL and intermediate libraries
- Java: So you won't kill yourself again, the code of framework and app are all Java
- Python: This is nothing, it's okay, compilation related
- Shell: This is impossible
- Makefile: Certainly, I will not jump off the building, right?

Here we define an INaruto interface file, simply add a helloWorld interface, the incoming is a string, and the return is a string. We will implement this interface later.

## 2. Generate HAL related files

As above, the hal interface file has been defined, and then we need to inherit and implement this interface, so how should we inherit and write it? Fortunately, Google still provides us with a set of tools hidl-gen to generate the HAL layer-related code framework And code examples, so we only need to care about the implementation part, instead of writing a bunch of useless code, wasting time on Makefile and some low-level errors.

- hidl-gen can automatically help us generate the hal layer instance framework and Android.mk/bp compilation control. Therefore, the system has provided us with an execution script hardware/interface/update-makefiles.sh. The following execution will automatically generate all the files for us. Documents needed:

```
./hardware/interface/update-makefiles.sh
```

Check out our latest code directory: hardware/interface/naruto:

```
├── 1.0
│   ├── Android.bp//automatically generated
│   ├── Android.mk//Auto generated
│   ├── default
│   │   ├── Android.bp//Need to write by yourself
│   │   └── android.hardware.naruto@1.0-service.rc//Used to
│   │       add boot service
│   ├── Naruto.cpp//automatically generated
│   ├── Naruto.h//Automatically generated
│   └── service.cpp//Hidl server registration entry program
└── INaruto.hal//Define interface
    └── Android.bp//Auto generated
```

Is it easy? We wrote INaruto.hal when we wrote the code, and the rest of the code is automatically generated, especially the two files Naruto.cpp and Naruto.h are the key files to implement the interface.

### 3. Implement the shared library of the HAL server

Open the Naruto.h file:

```
//Naruto.h

#ifndef ANDROID_HARDWARE_NARUTO_V1_0_NARUTO_H
#define ANDROID_HARDWARE_NARUTO_V1_0_NARUTO_H

#include <android/hardware/naruto/1.0/INaruto.h>

#include <hidl/MQDescriptor.h>
```

```

#include <hidl/Status.h>

namespace android {
namespace hardware {
namespace naruto {
namespace V1_0 {
namespace implementation {

using :: android :: hardware :: hidl_array ;
using :: android :: hardware :: hidl_memory ;
using :: android :: hardware :: hidl_string ;
using :: android :: hardware :: hidl_vec ;
using :: android :: hardware :: Return ;
using :: android :: hardware :: Void ;
using :: android :: sp ;

struct Naruto : public INaruto {

    //Methods from INaruto follow.

    Return <void> helloWorld(const hidl_string & name ,
helloWorld_cb _hidl_cb ) override ;

    //Methods from ::android::hidl::base::V1_0::IBase follow.

};

//FIXME: most likely delete, this is only for passthrough
implementations
//extern "C" INaruto* HIDL_FETCH_INaruto(const char*
name);

} //namespace implementation
} //namespace V1_0
} //namespace naruto
} //namespace hardware
} //namespace android

#endif //ANDROID_HARDWARE_NARUTO_V1_0_NARUTO_H

```

```
extern "C" __attribute__((weak)) void __hidl_inject_code__();
```

We know that there are two ways to implement HIDL, one is Binderized mode and the other is Passthrough mode. We see that there are two lines of commented out code above. It seems that this code is the key to choose whether to implement Binderized or Passthrough.

We use Passthrough mode to demonstrate here. In fact, after you try these two methods later, you will find that the two are actually the same. At present, most manufacturers use Passthrough to continue a lot of previous codes, but they will gradually be Changed, so let's open this comment-extern "C" INaruto\* HIDL\_FETCH\_INaruto(const char\* name);

Open and modify Naruto.cpp:

```
//Naruto.cpp

#include "Naruto.h"

namespace android {
namespace hardware {
namespace naruto {
namespace V1_0 {
namespace implementation {

//Methods from INaruto follow.

Return <void> Naruto ::helloWorld(const hidl_string & name ,
helloWorld_cb _hidl_cb ) {

    //TODO implement

    char buf [100];

    memset( buf , 0, 100);

    snprintf( buf , 100, "HelloWorld, %s", name .c_str());

    hidl_string result( buf );

    _hidl_cb( result );

    return Void();
}

//Methods from ::android::hidl::base::V1_0::IBase follow.

INaruto * HIDL_FETCH_INaruto(const char* /* name */) {
```

```

        return new Naruto();
    }

} //namespace implementation
} //namespace V1_0
} //namespace naruto
} //namespace hardware
} //namespace android

```

We made two changes to this file:

- Open the comment of `HIDL_FETCH_I`, let our HIDL use Passthrough method to achieve
- Add the implementation of the `helloWorld` function, simple string splicing (students of C/C++) should all understand

Next, we write the `default/Android.bp` file to generate the so library that we need to implement by impl.

```

default/Android.bp
cc_library_shared {
    name:"android.hardware.naruto@1.0-impl",
    relative_install_path: "hw",
    proprietary: true,
    srcs: [
        "Naruto.cpp",
    ],
    shared_libs: [
        "libhidlbase",
        "libhidltransport",
        "libutils",
        "android.hardware.naruto@1.0",
    ],
}

```

Eventually an `android.hardware.naruto@1.0-impl.so` will be generated, which will be generated under `/vendor/lib/hw/`, if it is a 64bits platform, it will be generated under `/vendor/lib64/hw/`, here I am using a 32bits platform, Will eventually be generated in the `/vendor/lib/hw/` directory; we can use `mmm` to compile and

```
mmm hardware/interface/naruto
```

This will generate two so files, one is the INaruto interface framework library file: android.hardware.naruto@1.0.so and the other is the INaruto implementation interface library file: android.hardware.naruto@1.0-impl.so their direct calling process Probably as follows:

[VIEW IMAGE \(HTTPS://IMG-BLOG.CSDNIMG.CN/2020072717031930.PNG\)](https://img-blog.csdnimg.cn/2020072717031930.PNG)

#### 4. Start the registration procedure on the Hal server

Now that we have the HIDL interface and the implementation instance of the interface, we need to register the server to HwServiceManager, so that the client can obtain the server interface and use it through the binder interface getService().

Remember the two files we created before, we haven't implemented it yet, let's take a look at the rc file first

```
# android.hardware.naruto@1.0-service.rc

service
    naruto_hal_service/vendor/bin/hw/android.hardware.naruto
    @1.0-service
    class hal
    user system
    group system
```

It's very simple. It is to execute the/vendor/bin/hw/android.hardware.naruto@1.0-service program when the device starts. This program is to register the service to the serviceManager management. The specific implementation is as follows:

service.cpp file:

```
//service.cpp

# define LOG_TAG "android.hardware.naruto@1.0-service"

# include <android/hardware/naruto/1.0/INaruto.h>
# include <hidl/LegacySupport.h>

using Android :: Hardware :: naruto :: V1_0 :: INaruto ;

using      Android      ::      Hardware      ::

defaultPassthroughServiceImplementation ;
```



```
int main() {
    return defaultPassthroughServiceImplementation <
    INaruto >();
}
```

This service is the interface registered in the INaruto interface file. As the binder server, it is very simple in one sentence. Because we use the passthrough mode, Android helps us to encapsulate this function. We don't need to addService by ourselves. We call defaultPassthroughServiceImplementation That's it. There is another registration method for binderies: the registration implementation of the server (<https://source.android.google.cn/devices/architecture/hidl-cpp/interfaces#server>)

```
:: Android :: SP < the IFoo > myFoo = new FooImpl();
:: Android :: SP < the IFoo > mySecondFoo = new
FooAnotherImpl();
status_t Status = myFoo ->registerAsService();
status_t anotherStatus = mySecondFoo -
>registerAsService("another_foo");
```

Next, add the corresponding compilation control logic in Android.bp:

```
cc_binary {
    name: "android.hardware.naruto@1.0-service",
    defaults: ["hidl_defaults"],
    proprietary: true,
    relative_install_path: "hw",
    srcs: ["service.cpp"],
    init_rc: ["android.hardware.naruto@1.0-service.rc"],
    shared_libs: [
        "libhidlbase",
        "libhidltransport",
        "libutils",
        "liblog",
        "android.hardware.naruto@1.0"
```

```

        android.hardware.naruto@1.0 ,

        "android.hardware.naruto@1.0-impl" ,

    ],

}

```

After compiling, you can find the corresponding file under vendor/bin/hw/.

OK, our server-side process and implementation-side shared library have been completed.

But if you burn the image at this time, you will find that the process will fail to start. The reason is that we did not configure sepolicy for this process, so the correct way is to **add selinux permissions to it, we will not do it here** . Because we can use root permissions (push directly to the device) to manually start this service.

Okay, let's take a look at how to write the client code.

## 5. HIDL Client test program

We have the server, and then we need to write a client to verify that the server interface is correct. I like to write code step by step, and make a test code for each step to test. One is to verify the function of each step, but to use it for later testing.

I have a colleague who writes code quickly. After finishing writing, I don't know how to debug. This method is not good. It's not good. Don't follow suit.

Writing code is not difficult, writing good code is not easy, good code is improved through a large number of tests, no one can write good code at once, so everyone must test the interface in the design phase. Save it, otherwise, if you rework and go to re-design later, it will be very shameless and there is no way. If you are in our business, you are rushing to the progress every day. You TMD tells the boss to rework and do re-design, the boss will not chop you No.

```

//naruto_test.cpp

#include <android/hardware/naruto/1.0/INaruto.h>

#include <hidl/Status.h>

#include <hidl/LegacySupport.h>

#include <utils/misc.h>

#include <hidl/HidlSupport.h>

#include <stdio.h>

```

```

using Android :: Hardware :: Naruto :: V1_0 :: INaruto ;

using Android :: SP ;

using Android :: Hardware :: hidl_string ;

int main()
{
    int ret ;

    android :: sp < INaruto > service = INaruto ::getService();

    if( service == nullptr ) {
        printf("Failed to get service\n");
        return -1;
    }

    -Service ->helloWorld("SvenCheng", [&]( hidl_string Result )
    {
        printf("%s\n", Result .c_str());
    });

    return 0;
}

```

The code is quite simple, it doesn't look like it, instantiate the binder service, through `INaruto::getService()`, get the binder server termination interface proxy class, and then you can call his method, we call the `helloWorld` interface here, Then get the result through callback.

Let's paste Makefile for those ignorant programmers

```

LOCAL_PATH: = $(call my-dir)

include $(CLEAR_VARS)

LOCAL_PROPRIETARY_MODULE: = true

LOCAL_MODULE: = naruto_test

LOCAL_SRC_FILES: =/
    naruto_client.cpp/

```

```
LOCAL_SHARED_LIBRARIES: =/

liblog/
libhidlbase/
libutils/
android.hardware.naruto@1.0/

include $(BUILD_EXECUTABLE)
```

Above, after compiling with mmm vendor/mediatek/proprietary/external/naruto, it will be generated in /system/bin/naruto\_test.

*Important: Remember to add the definition of the vendor interface in the manifest file, otherwise the service will not be available on the client side. Add in the corresponding manifest.xml: Important:*

*Remember to add the definition of the vendor interface in the manifest file, otherwise in the client The end is unable to get the service, add in the corresponding manifest.xml:*

```
# device/mediatek/mt8167/manifest.xml

< hal format ="hidl">

    < name > android.hardware.naruto </name >

    < transport > hwbinder </transport >

    < version > 1.0 </version >

    < interface >

        < name > INaruto </name >

        < instance > default </instance >

    </interface >

</hal>
```

The manifest.xml is placed in different locations according to each manufacturer, so you need to identify where it is, usually in device///manifest.xml

After modifying the manifest.xml file, you need to recompile the entire aosp and burn the image to the device, otherwise it will report when you start the program: Cannot find the interface getService()

## 6. Test case

Through the above, we have produced the following documents

in total:

- android.hardware.naruto@1.0.so: Naruto module hal framework interface, exposed to the client.
- android.hardware.naruto@1.0-impl.so: Naruto module server code implementation, the real behavior of the binder server after the interface is called.
- android.hardware.naruto@1.0-service: Naruto server starts the registration entry program
- android.hardware.naruto@1.0-service.rc: Android native process entry
- naruto\_test: client calling program

We need to push the above files to the corresponding directories:

```
adb push android.hardware.naruto@1.0.so/system/lib/  
adb          push          android.hardware.naruto@1.0-  
impl.so/vendor/lib/hw/  
adb push android.hardware.naruto@1.0-service/system/bin  
adb push naruto_test/vendor/bin/hw      #This directory is  
necessary, otherwise dlsym failure will occur
```

Start the service:

```
aiv8167sm3_bsp:/# android.hardware.naruto@1.0-service
```

Start the client:

```
aiv8167sm3_bsp:/# naruto_test  
HelloWorld, SvenCheng
```

Seeing that, our test code passes in the "SvenCheng" string, and the result is "HelloWorld, SvenCheng", which is in line with our expected results.

So this article is over. If the people who eat melon don't code the code quickly, it's not as good as bad writing. How can you remember if you don't write it again.

This knowledge is as simple as the use of this HIDL, don't worry, there will be other knowledge points later, after all, this is just a simple HelloWorld.

Author: Kiba Aeolus herein by reference link:  
<https://www.jianshu.com/p/ca6823b897b5>

© 2022 - **Karthi Softek**

[POLICIES \(/POLICIES\)](#)

[CONTACT \(/CONTACT\)](#)

[ABOUT \(/ABOUT\)](#)