

在WIN从零开始在QMUE上添加一块自己的开发板（一） _qemu添加自定义开发板-CSDN博客

 blog.csdn.net/DreamTrue520/article/details/135703895

一、前言

笔者这篇博客作为平时学习时的笔记记录，如有不对还望指正，本博客大量借鉴资料，笔者只是拾人牙慧的小屁孩。

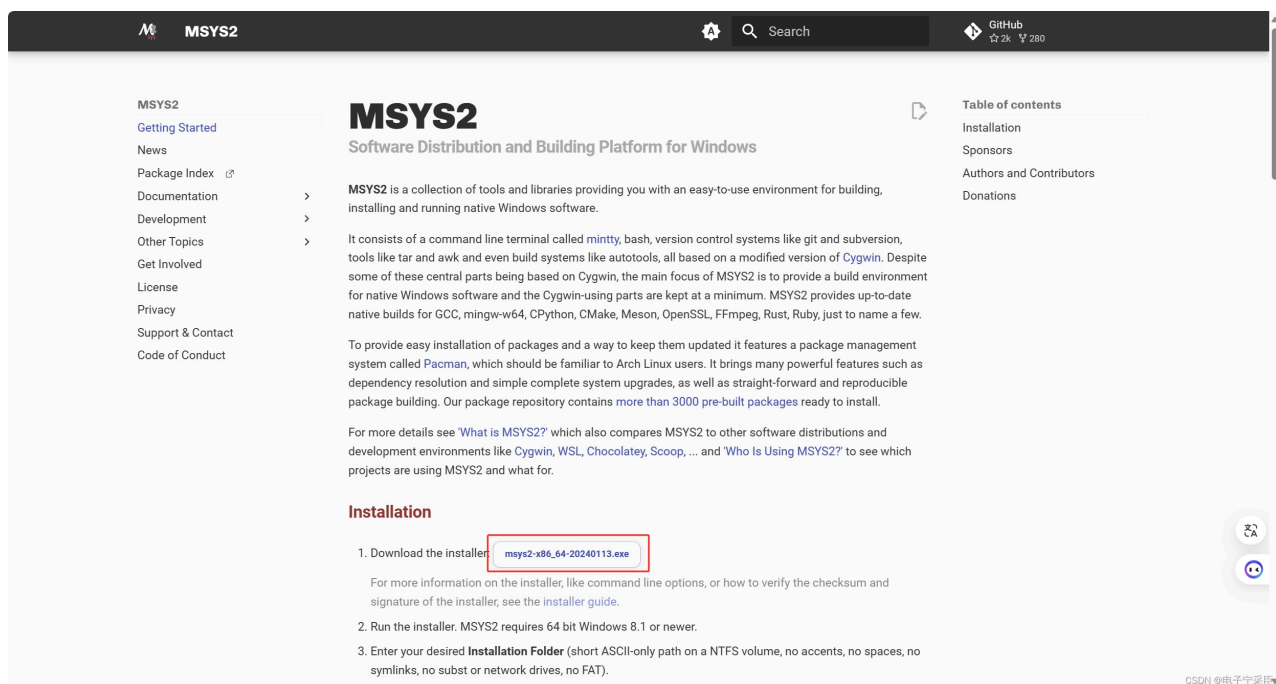
QEMU是一种通用的开源计算机仿真器和虚拟机。而QUME内置支持了一些开发板，我们可以基于这些内置的板子来做操作系统等软件的配置，但是实际市面上很多板子QUME中是没有提供支持的，这需要我们根据QUME的源码自定义一些开发板，然后再重新编译。

二、源码编译

笔者是在Win系统上利用Msys2进行的QUME源码编译。

（一）安装Msys2

打开 <https://www.msys2.org/>，下载最新Msys2的安装包并安装。



完成安装后，我们先进行更新源。

（笔者的安装路径为：`C:\msys64`）

进入目录`C:\msys64\etc\pacman.d`，

- 在文件`mirrorlist.msys`的前面插入
`Server = http://mirrors.ustc.edu.cn/msys2/msys/$arch`
- 在文件`mirrorlist.mingw32`的前面插入
`Server = http://mirrors.ustc.edu.cn/msys2/mingw/i686`

- 在文件[mirrorlist.mingw64](http://mirrors.ustc.edu.cn/msys2/mingw/x86_64/mirrorlist.mingw64)的前面插入
Server = http://mirrors.ustc.edu.cn/msys2/mingw/x86_64

然后我们启动 MSYS2 终端(MSYS2 MINGW64), 进行更新:

```
pacman -Syu
pacman -Su
• 1
• 2
```

(二) 配置GCC工具链

```
pacman -Sy mingw-w64-x86_64-toolchain
1
```

(三) 安装QEMU构建依赖

```
pacman -Sy mingw-w64-x86_64-meson mingw-w64-x86_64-ninja \  
mingw-w64-x86_64-python \  
mingw-w64-x86_64-python-sphinx \  
mingw-w64-x86_64-python-sphinx_rtd_theme \  
mingw-w64-x86_64-autotools \  
mingw-w64-x86_64-tools-git \  
mingw-w64-x86_64-cc \  
mingw-w64-x86_64-angleproject \  
mingw-w64-x86_64-capstone \  
mingw-w64-x86_64-curl \  
mingw-w64-x86_64-cyrus-sasl \  
mingw-w64-x86_64-expat \  
mingw-w64-x86_64-fontconfig \  
mingw-w64-x86_64-freetype \  
mingw-w64-x86_64-fribidi \  
mingw-w64-x86_64-gcc-libs \  
mingw-w64-x86_64-gdk-pixbuf2 \  
mingw-w64-x86_64-gettext \  
mingw-w64-x86_64-glib2 \  
mingw-w64-x86_64-gmp \  
mingw-w64-x86_64-gnutls \  
mingw-w64-x86_64-graphite2 \  
mingw-w64-x86_64-gst-plugins-base \  
mingw-w64-x86_64-gstreamer \  
mingw-w64-x86_64-gtk3 \  
mingw-w64-x86_64-harfbuzz \  
mingw-w64-x86_64-jbigkit \  
mingw-w64-x86_64-lerc \  
mingw-w64-x86_64-libc++ \  
mingw-w64-x86_64-libdatrie \  
mingw-w64-x86_64-libdeflate \  
mingw-w64-x86_64-libepoxy \  
mingw-w64-x86_64-libffi \  
mingw-w64-x86_64-libiconv \  
mingw-w64-x86_64-libidn2 \  
mingw-w64-x86_64-libjpeg-turbo \  
mingw-w64-x86_64-libnfs \  
mingw-w64-x86_64-libpng \  
mingw-w64-x86_64-libpsl \  
mingw-w64-x86_64-libslirp \  
mingw-w64-x86_64-libssh \  
mingw-w64-x86_64-libssh2 \  
mingw-w64-x86_64-libtasn1 \  
mingw-w64-x86_64-libthai \  
mingw-w64-x86_64-libtiff \  
mingw-w64-x86_64-libunistring \  
mingw-w64-x86_64-libunwind \  
mingw-w64-x86_64-libusb \  
mingw-w64-x86_64-libwebp \  
mingw-w64-x86_64-libwinpthread-git \  
mingw-w64-x86_64-lz4 \  
mingw-w64-x86_64-lzo2 \  
mingw-w64-x86_64-nettle \  
mingw-w64-x86_64-openssl \  
mingw-w64-x86_64-opus \  
mingw-w64-x86_64-orc \  
mingw-w64-x86_64-p11-kit \  

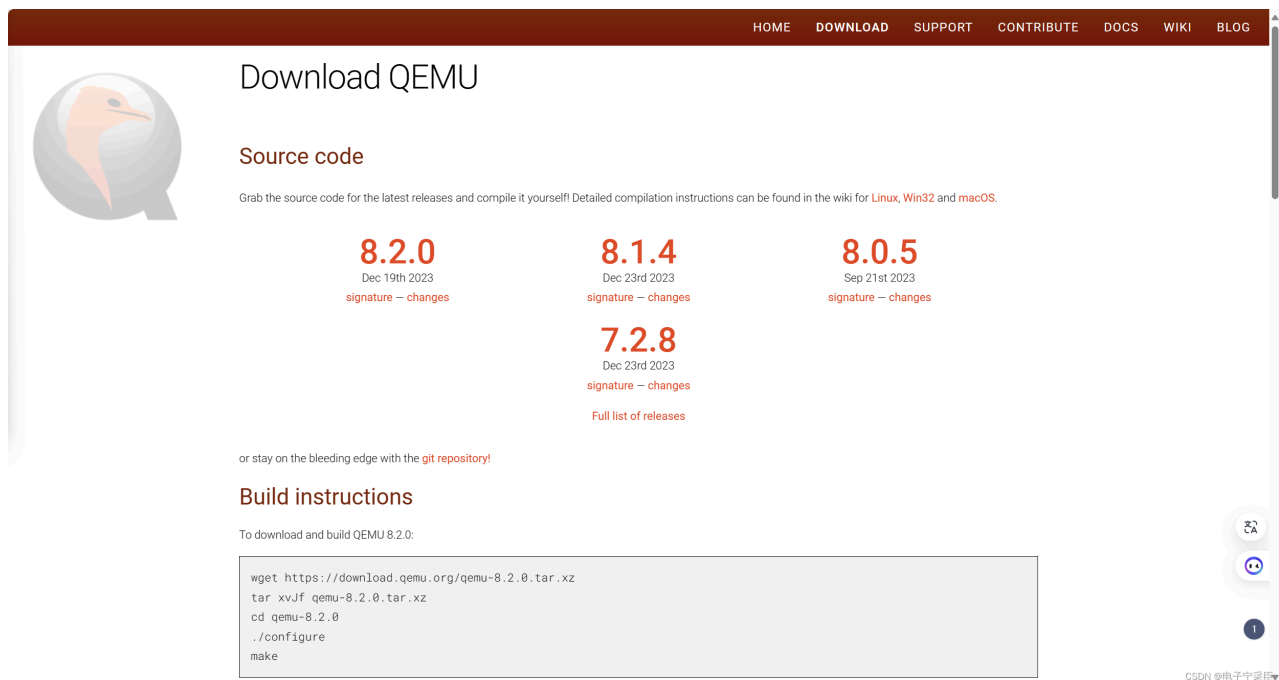
```

```
mingw-w64-x86_64-pango \  
mingw-w64-x86_64-pixman \  
mingw-w64-x86_64-SDL2 \  
mingw-w64-x86_64-SDL2_image \  
mingw-w64-x86_64-snappy \  
mingw-w64-x86_64-spice \  
mingw-w64-x86_64-usbredir \  
mingw-w64-x86_64-xz \  
mingw-w64-x86_64-zlib \  
mingw-w64-x86_64-zstd
```

(四) 下载编译QEMU源码

```
mkdir qemu  
cd qemu/
```

下载QUME的版本为8.2.0:

The screenshot shows the QEMU download page. At the top is a navigation bar with links: HOME, DOWNLOAD, SUPPORT, CONTRIBUTE, DOCS, WIKI, BLOG. The main heading is "Download QEMU" next to the QEMU logo. Below this is a "Source code" section with a note: "Grab the source code for the latest releases and compile it yourself! Detailed compilation instructions can be found in the wiki for Linux, Win32 and macOS." There are four version cards: 8.2.0 (Dec 19th 2023), 8.1.4 (Dec 23rd 2023), 8.0.5 (Sep 21st 2023), and 7.2.8 (Dec 23rd 2023). Each card has links for "signature" and "changes". Below these is a link for "Full list of releases". A note says "or stay on the bleeding edge with the git repository!". The "Build instructions" section says "To download and build QEMU 8.2.0:" and includes a code block with the following commands:

```
wget https://download.qemu.org/qemu-8.2.0.tar.xz  
tar xvJf qemu-8.2.0.tar.xz  
cd qemu-8.2.0/  
./configure  
make
```

源码下载与编译:

(这里需要管理员权限打开Msys2)

```
wget https://download.qemu.org/qemu-8.2.0.tar.xz  
tar xvJf qemu-8.2.0.tar.xz  
cd qemu-8.2.0/  
./configure  
make -j8
```

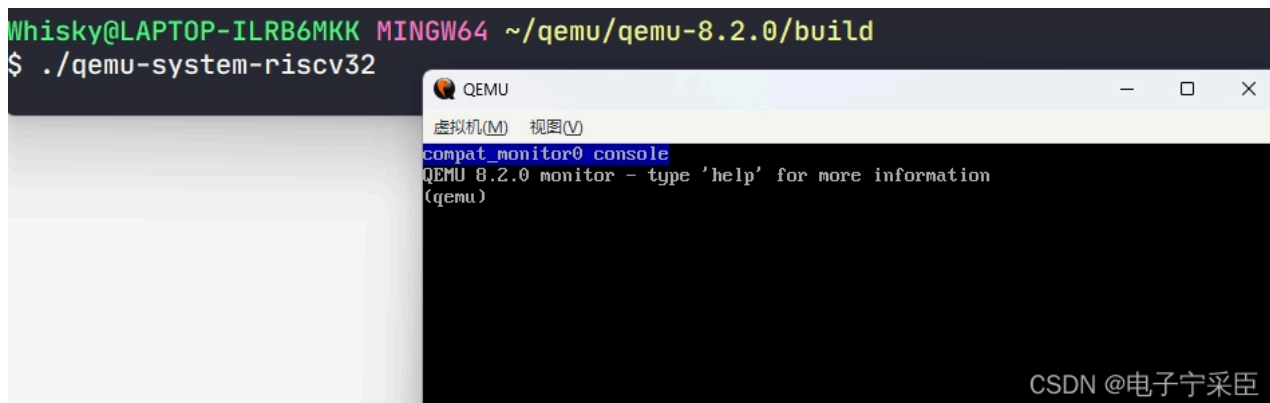
编译完成后会生成一个./build目录

```
cd build/  
make install
```

之后我们测试一下——查看QEMU的版本号:

```
Whisky@LAPTOP-ILRB6MKK MINGW64 ~/qemu/qemu-8.2.0/build
$ ./qemu-img -V
qemu-img version 8.2.0
Copyright (c) 2003-2023 Fabrice Bellard and the QEMU Project developers
```

启动QEMU：
这里以riscv32为例



至此我们已经编译完了QUME的源码了。

二、QUME编程基础

QEMU是一款开源的模拟器及虚拟机监管器(Virtual Machine Monitor, VMM)，通过动态二进制翻译来模拟CPU，并提供一系列的硬件模型，使guest os认为自己和硬件直接打交道，其实是同QEMU模拟出来的硬件打交道，QEMU再将这些指令翻译给真正硬件进行操作。

(一) QOM机制

QOM——The QEMU Object Model

QEMU提供了一套面向对象编程的模型——QOM，即QEMU Object Module，几乎所有的设备如CPU、内存、总线等都是利用这一面向对象的模型来实现的。

QEMU对象模型提供了一个**注册用户可创建类型**并从这些类型**实例化对象**的框架。

其实也就是一种**OOP IN C**(C上实现面对对象)。

一段面对对象的程序代码（C++语言）

```
class MyClass {
public:
    int a;
    void set_A(int a);
}
```

切换为C语言也就是：

```
struct MyClass {
    int a;
    void (*set_A)(MyClass *this, int a);
}
```

当然，这只是一个例子。

在QOME中，我们通常一个对象的初始化分为四步：

1. 将 `TypeInfo` 注册 `TypeImpl`
2. 实例化 `ObjectClass`
3. 实例化 `Object`
4. 添加 `Property`

QOM模型的实现代码位于`qom/`文件夹下的文件中，这涉及了几个结构`TypeImpl`，`ObjectClass`，`Object`和`TypeInfo`。看了下它们的定义都在`/include/qom/object.h`可以找到，只有`TypeImpl`的具体结构是在`/qom/object.c`中。

ObjectClass: 是所有类对象的基类，第一个成员变量为类型`typedef struct TypeImpl *`的`type`。

Object: 是所有对象的基类`Base Object`，第一个成员变量为指向`ObjectClass`类型的指针。

TypeInfo: 是用户用来定义一个`Type`的工具型的数据结构。

TypeImpl: 对数据类型的抽象数据结构，`TypeInfo`的属性与`TypeImpl`的属性对应。

(二) 将 `TypeInfo` 注册 `TypeImpl`

```
struct TypeInfo
{
    const char *name;
    const char *parent;

    size_t instance_size;
    void (*instance_init)(Object *obj);
    void (*instance_post_init)(Object *obj);
    void (*instance_finalize)(Object *obj);

    bool abstract;
    size_t class_size;

    void (*class_init)(ObjectClass *klass, void *data);
    void (*class_base_init)(ObjectClass *klass, void *data);
    void *class_data;

    InterfaceInfo *interfaces;
};
```

其中的重点有：

1. Name：包含了自己的名字`name`和parent的名字的`parent`。
2. Class（针对`ObjectClass`）：`ObjectClass`的信息包括，`class_size`，`class_data`，`class`相关函数：`class_base_init`，`class_init`，`class_finalize`等。
这些函数都是为了初始化，释放结构体`ObjectClass`。
3. Instance（针对的是`Object`）：对象`Object`信息包括：`instance_size`，`instance`相关函数：`instance_post_init`，`instance_init`，`instance_finalize`。
这些函数都是为了初始化，释放结构体`Object`。
4. 其他信息：`abstract`是否为抽象。`interface`数组。

一般是定义一个TypeInfo，然后调用 `type_register(TypeInfo)` 或者 `type_register_static(TypeInfo)` 函数（使用`type_register_static`比较多），就会生成相应的TypeImpl实例，将这个TypeInfo注册到全局的TypeImpl的hash表中。我们来看一个例程：

```
#define TYPE_MY_DEVICE "my-device"

static void my_device_class_init(ObjectClass *oc, void *data)
{
}

static void my_device_init(Object *obj)
{
}

typedef struct MyDeviceClass
{
    DeviceClass parent;
    void (*init) (MyDevice *obj);
} MyDeviceClass;

typedef struct MyDevice
{
    DeviceState parent;
    int reg0, reg1, reg2;
}MyDevice;

static const TypeInfo my_device_info = {
    .name = TYPE_MY_DEVICE,
    .parent = TYPE_DEVICE,
    .instance_size = sizeof(MyDevice),
    .instance_init = my_device_init,
    .class_size = sizeof(MyDeviceClass),
    .class_init = my_device_class_init,
};

static void my_device_register_types(void)
{
    type_register_static(&my_device_info);
}

type_init(my_device_register_types)
```

当然，其中的代码

```
static void my_device_register_types(void)
{
    type_register_static(&my_device_info);
}

type_init(my_device_register_types)
```

也可以简化为

```
DEFINE_TYPES(my_device_infos)
```

举个实际的例子

1. 定义设备

```

/* SOC state定义 */
#define TYPE_NUCLEI_HBIRD_SOC "riscv.nuclei.hbird.soc"
#define RISCVM_NUCLEI_HBIRD_SOC(obj) \
OBJECT_CHECK(NucleiHBSocState, (obj), TYPE_NUCLEI_HBIRD_SOC)
typedef struct NucleiHBSocState
{
    /*< private >*/
    SysBusDevice parent_obj;
    /*< public >*/
} NucleiHBSocState;

/* Machine state定义 */
#define TYPE_HBIRD_FPGA_MACHINE MACHINE_TYPE_NAME("hbird_fpga")
#define HBIRD_FPGA_MACHINE(obj) \
OBJECT_CHECK(NucleiHBState, (obj), TYPE_HBIRD_FPGA_MACHINE)
typedef struct
{
    /*< private >*/
    SysBusDevice parent_obj;
    /*< public >*/
    NucleiHBSocState soc;
} NucleiHBState;

```

2. SOC设备注册

```

static void nuclei_soc_init(Object *obj)
{
    qemu_log(">>nuclei_soc_init \n");
}
static void nuclei_soc_realize(DeviceState *dev, Error **errp)
{
    qemu_log(">>nuclei_soc_realize \n");
}
static void nuclei_soc_class_init(ObjectClass *oc, void *data)
{
    qemu_log(">>nuclei_soc_class_init \n");
    DeviceClass *dc = DEVICE_CLASS(oc);
    dc->realize = nuclei_soc_realize;
    dc->user_creatable = false;
}

static const TypeInfo nuclei_soc_type_info = {
    .name = TYPE_NUCLEI_HBIRD_SOC,
    .parent = TYPE_DEVICE,
    .instance_size = sizeof(NucleiHBSocState),
    .instance_init = nuclei_soc_init,
    .class_init = nuclei_soc_class_init,
};
static void nuclei_soc_register_types(void)
{
    type_register_static(&nuclei_soc_type_info);
}
type_init(nuclei_soc_register_types)

```

3. Machine设备注册


```

static void nuclei_board_init(MachineState *machine)
{
    NucleiHBState *s = HBIRD_FPGA_MACHINE(machine);
    qemu_log(">>nuclei_board_init \n");
    /* Initialize SOC */
    object_initialize_child(OBJECT(machine), "soc", &s->soc, TYPE_NUCLEI_HBIRD_SOC);
    qdev_realize(DEVICE(&s->soc), NULL, &error_abort);
}
static void nuclei_machine_instance_init(Object *obj)
{
    qemu_log(">>nuclei_machine_instance_init \n");
}
static void nuclei_machine_class_init(ObjectClass *oc, void *data)
{
    qemu_log(">>nuclei_machine_class_init \n");
    MachineClass *mc = MACHINE_CLASS(oc);
    mc->desc = "Nuclei HummingBird Evaluation Kit";
    mc->init = nuclei_board_init;
}

static const TypeInfo nuclei_machine_typeinfo = {
    .name = MACHINE_TYPE_NAME("hbird_fpga"),
    .parent = TYPE_MACHINE,
    .class_init = nuclei_machine_class_init,
    .instance_init = nuclei_machine_instance_init,
    .instance_size = sizeof(NucleiHBState),
};
static void nuclei_machine_init_register_types(void)
{
    type_register_static(&nuclei_machine_typeinfo);
}
type_init(nuclei_machine_init_register_types)

```

4. 修改编译文件

hw/riscv/Kconfig:

```

config NUCLEI_N
bool
select MSI_NONBROKEN
select UNIMP

```

hw/riscv/meson.build:

```

riscv_ss = ss.source_set()
riscv_ss.add(files('boot.c'), fdt)
riscv_ss.add(files('numa.c'))
riscv_ss.add(files('riscv_hart.c'))
...
riscv_ss.add(when: 'CONFIG_NUCLEI_N', if_true: files('nuclei_n.c'))

hw_arch += {'riscv': riscv_ss}

```

configs\devices\riscv32-softmmu\default.mak:

```

...
CONFIG_NUCLEI_N=y

```

```
./configure --target-list=riscv32-softmmu  
make -j16
```

(三) 测试

编译完成后，我们进行安装（Msys2在管理员权限下运行）

```
make install
```

当然，为了方便我们测试，也可以编写脚本，然后不混用build文件夹，保证我们自己平时也能使用qume纯净版：

build.sh:

```
# 获取当前脚本文件所在的目录  
SHELL_FOLDER=$(cd "$(dirname "$0")";pwd)  
  
if [ ! -d "$SHELL_FOLDER/output/qemu" ]; then  
./configure --prefix=$SHELL_FOLDER/output/qemu --target-list=riscv32-softmmu  
fi  
make -j8  
make install  
cd ..
```

run.sh:

```
SHELL_FOLDER=$(cd "$(dirname "$0")";pwd)  
$SHELL_FOLDER/output/qemu/qemu-system-riscv32.exe \  
-M hbird_fpga
```

安装完成后

我们开始测试。

先看看板子的列表：

```
./qemu-system-riscv32.exe -M ?
```

得到的板子列表中有我们刚刚编写的板子：

```
Supported machines are:  
hbird_fpga      Nuclei HummingBird Evaluation Kit  
none           empty machine  
opentitan       RISC-V Board compatible with OpenTitan  
sifive_e        RISC-V Board compatible with SiFive E SDK  
sifive_u        RISC-V Board compatible with SiFive U SDK  
spike           RISC-V Spike board (default)  
virt            RISC-V VirtIO board
```

我们直接运行这块板子：

```
./qemu-system-riscv32.exe -M hbird_fpga
```

```
>>nuclei_soc_class_init
>>nuclei_machine_class_init
>>nuclei_machine_instance_init
>>nuclei_board_init
>>nuclei_soc_init
>>nuclei_soc_realize
```

```
Whisky@LAPTOP-ILRB6MKK MINGW64 ~/qemu/qemu-8.2.0/build
# ./qemu-system-riscv32.exe -M hbird_fpga
>>nuclei_soc_class_init
>>nuclei_machine_class_init
>>nuclei_machine_instance_init
>>nuclei_board_init
>>nuclei_soc_init
>>nuclei_soc_realize
```



(四) 从结果中的反思

ObjectClass的初始化

在测试结果中，我们还可以回味整个QUME的运行流程。

首先在我们注册TypeInfo时，其类的构造函数会在其创建其类的时候执行，也就是在TypeImpl的hash表已经有了之后，下一步要初始化每个type的时候。（这一步可以看成是class的初始化，可以理解成每一个type对应了一个class，接下来会初始化class）main函数中的module_call_init(MODULE_INIT_QOM);调用了MODULE_INIT_QOM类型的ModuleTypeList中的所有ModuleEntry中的init()函数，也就是第一步type_init的第一个参数XXX_register_types函数指针。（__attribute__((constructor))的修饰让type_init在main之前执行，type_init的参数是XXX_register_types函数指针，将函数指针传递到ModuleEntry的init函数指针，最后就是将这个ModuleEntry插入到ModuleTypeList）那接下来就是XXX_register_types函数的操作了，就是一个一个创建完TypeImpl的哈希表。

如果这里有看不懂，可以深究QEMU的一些基础知识及QOM(Qemu Object Model)的部分相关源码阅读。

之后main函数会调用machine_class = select_machine();在里面的调用链中将会有ti->class_init初始化的实现。

所以，会首先看见

```
>>nuclei_soc_class_init
>>nuclei_machine_class_init
```

实例化 Instance(Object)

其次，我们发现main函数接下来调用了qemu_opts_foreach，循环查找参数（options）：

```
qemu_opts_foreach(qemu_find_opts("device"),
                  default_driver_check, NULL, NULL);
qemu_opts_foreach(qemu_find_opts("device"),
                  device_help_func, NULL, NULL)
...
qemu_opts_foreach(qemu_find_opts("device"),
                  device_init_func, NULL, &error_fatal);
```

前二者default_driver_check和device_help_func参数的qemu_opts_foreach输出driver的help信息，还有那些option什么的。

重点在device_init_func参数的qemu_opts_foreach，在其中调用了qdev_device_add。而在

`qdev_device_add`里面，重要的一行是调用了`dev = DEVICE(object_new(driver));`，而且上一行有个注释——`/* create device */`：

`DEVICE`是一个宏，实际是`OBJECT_CHECK`，主要是看看`obj`是否是`TYPE_DEVICE`的一个实例：

```
#define DEVICE(obj) OBJECT_CHECK(DeviceState, (obj), TYPE_DEVICE)
#define OBJECT_CHECK(type, obj, name) \
    ((type *)object_dynamic_cast_assert(OBJECT(obj), (name), \
                                         __FILE__, __LINE__, __func__))
```

更重要的是`object_new(driver)`，它利用`object_new_with_type`进行实例：

它调用`type_initialize`，在其中调用`parent`的`class_base_init`进行初始化，最后调用自己`class_init`进行初始化。

其次调用`object_init_with_type`函数首先判断`ti`是否有`parent`（即`type->parent != NULL`），有`parent`就会递归调用`object_init_with_type`，最终就是调用`ti->instance_init`函数。

所以，再接着是

```
>>nuclei_machine_instance_init
```

之后又因为我们在`nuclei_machine_class_init`中赋值`mc->init = nuclei_board_init;`，所以执行`ti->instance_init`：

```
>>nuclei_board_init
```

当然我们知道，在`nuclei_board_init`里面，我们进行了SOC的实例化：

```
object_initialize_child(OBJECT(machine), "soc", &s->soc, TYPE_NUCLEI_HBIRD_SOC);
qdev_realize(DEVICE(&s->soc), NULL, &error_abort);
```

所以最后：

```
>>nuclei_soc_init
>>nuclei_soc_realize
```

参考资料

1. 如何在 Windows 10/11 上构建 QEMU
2. 在Windows上编译QEMU
3. 从源码构建Qemu
4. [完结]从零开始的RISC-V模拟器开发·第一季·2021春季
5. QEMU 的一些基础知识及QOM(Qemu Object Model)的部分相关源码阅读