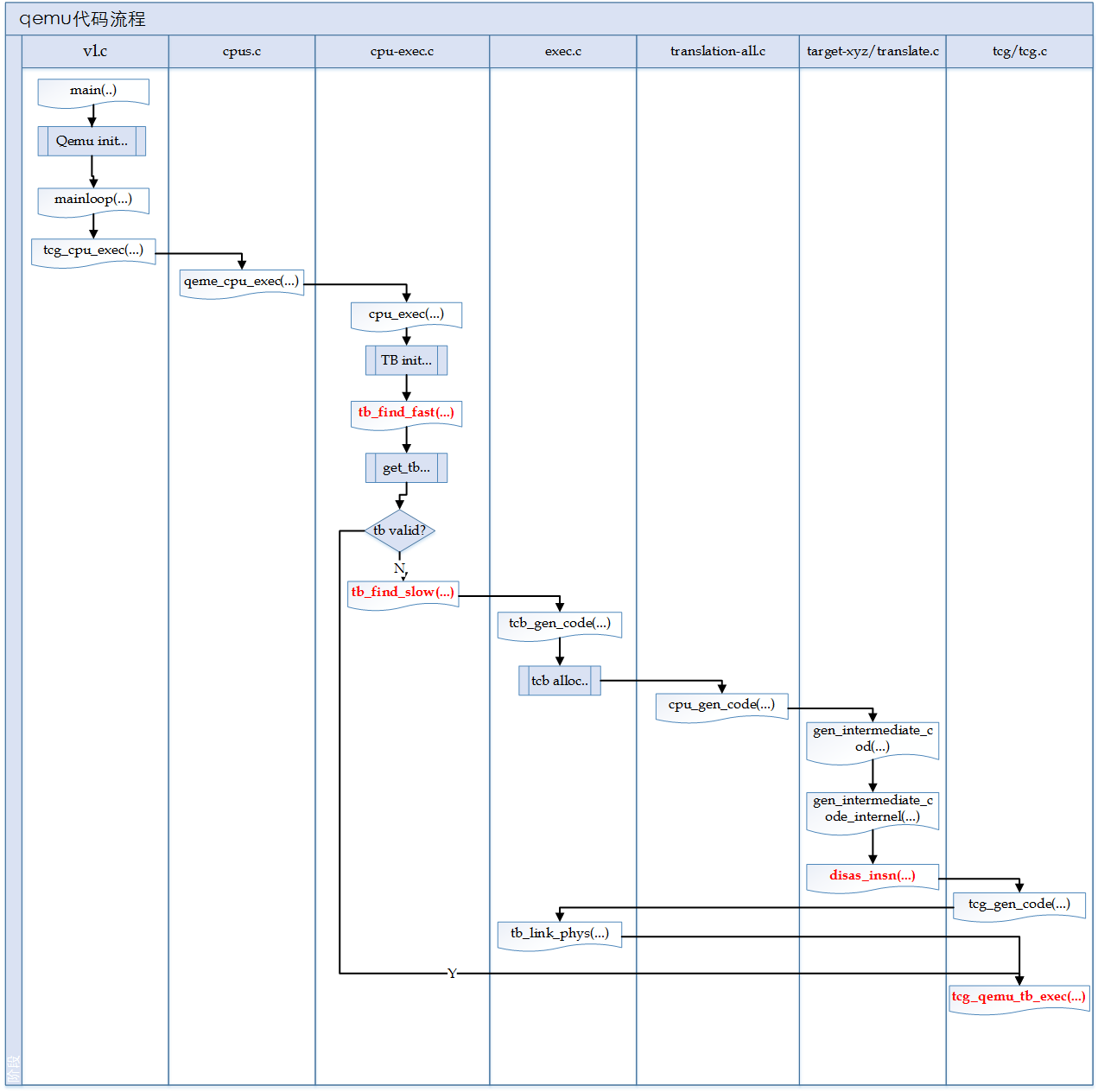
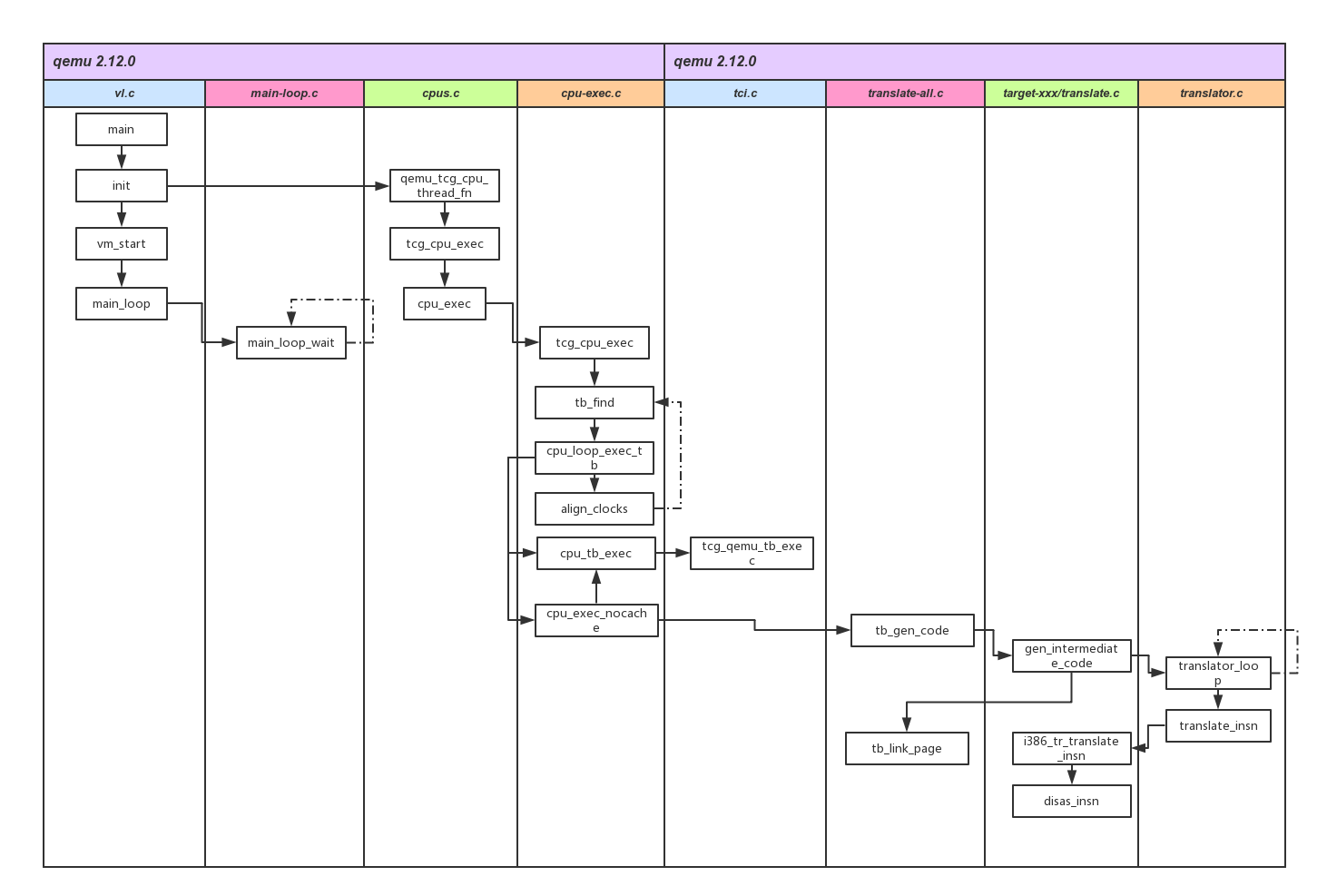
# qemu代码流程

QEMU通过把target instruction ->micro-op->tcg->host instruction的方式实现对不同架构的模拟。下图是网上找到的，印象中之前的流程是这样的，但是在查看qemu 2.12.0源码时发现现在的qemu架构已经有些不同。

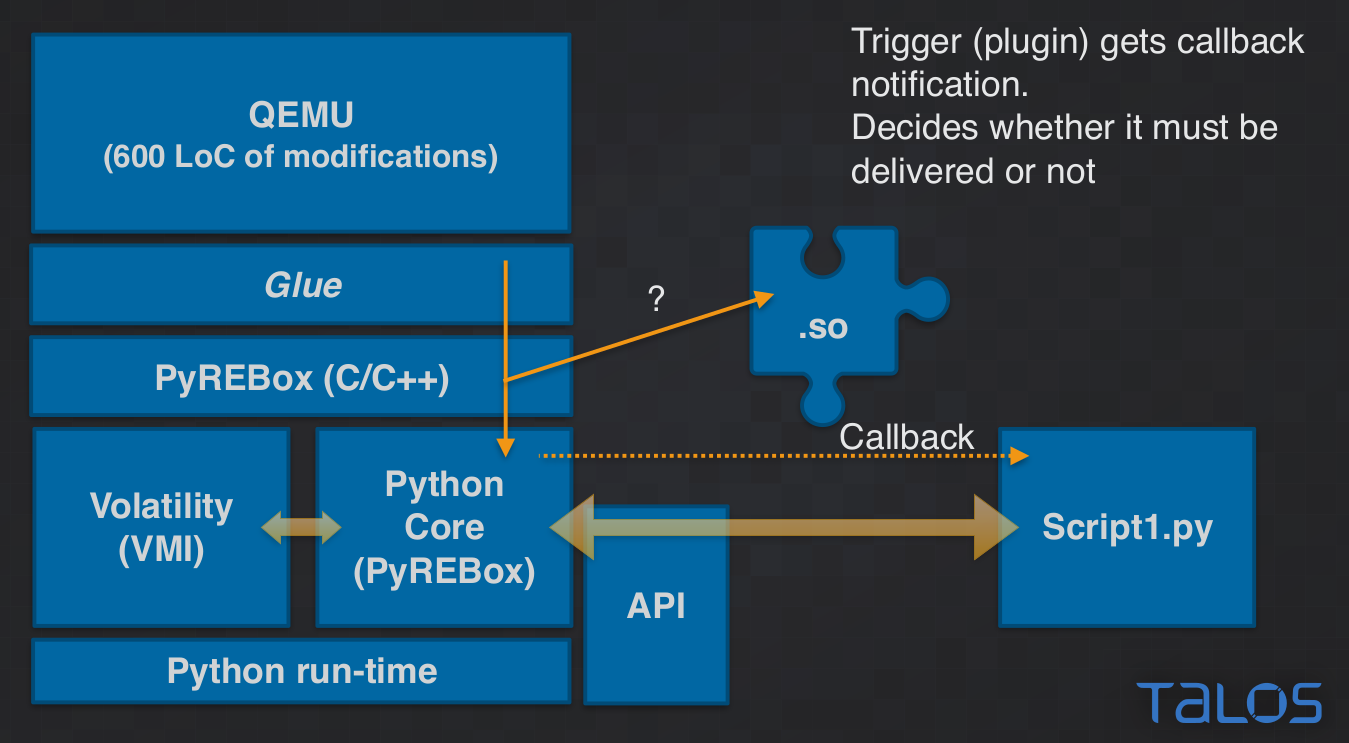


# qemu2.12.0的架构

在看流程之前首先要了解qemu的整体架构，QEMU实际上使用的是将事件驱动机制和线程相结合的一种混合架构，为每一个vCPU分配一个QEMU线程，以及一个专用的事件处理循环线程，这个模型称为iothread。详见https://www.ibm.com/developerworks/community/blogs/8b3f8d48-b83c-4bcb-a34e-2b9c8c13ab1f/entry/20161206?lang=en

# Pyrebox的架构

结合上面的qemu源码分析，可以找到合适的位置插装代码、提取信息。



具体来讲，

## qemu/vl.c：

添加入了头文件pyrebox/pyrebox.h，解析pyrebox的设置，初始化pyrebox模块，主要是设置python环境。

## qemu/accel/tcg/cpu-exec.c：

添加了头文件pyrebox/qemu\_glue\_callbacks.h，更新记录的cpu执行状态，notify\_cpu\_executing()

## qemu/target/i386/translate.c：

主要功能的实现，大量改动，添加了头文件pyrebox/qemu\_glue\_callbacks\_needed.h，

1. 在DisasContext结构中添加了4个变量，

> target\_ulong pgd;

//Pyrebox: Save the PGD in the DisasContext to allow conditional instrumentation based on the PGD

target\_ulong pgd;

//Pyrebox: Save the pc between each pair of instructions, because when the insn\_end / block\_end is triggered, the enviroment CPU is already pointing to the next block

target\_ulong saved\_pc;

//Pyrebox: Save the opcode while doing the dissasembly, in order to call the corresponding opcode range callback.

uint32\_t saved\_opcode;

//Pyrebox: the cpu.

CPUState\* cs;

1. 在gen\_goto\_tb()、do\_gen\_eob\_worker()等函数，包括TB\_begin、insn\_begin中插装代码实现分析

//Pyrebox: insn end

gen\_helper\_qemu\_insn\_end\_callback()

//Pyrebox: block\_end

gen\_helper\_qemu\_block\_end\_callback()

//Pyrebox: opcode range

gen\_helper\_qemu\_opcode\_range\_callback()

1. 在gen\_illegal\_opcode()中插装代码实现qemu运行暂停
2. 在disas\_insn()中插装代码实现对指令的分析，特别是int3、intN的处理

## /target/i386/helper.h

添加了qemu helper函数，实现对qemu的状态监控和信息提取。

//void qemu\_block\_begin\_callback(CPUState\* cpu,TranslationBlock\* tb);

DEF\_HELPER\_2(qemu\_block\_begin\_callback, void, ptr, ptr)

//void qemu\_op\_block\_begin\_callback(CPUState\* cpu,TranslationBlock\* tb);

//DEF\_HELPER\_2(qemu\_op\_block\_begin\_callback, void, ptr, ptr)

//void qemu\_op\_insn\_begin\_callback(CPUState\* cpu);

//DEF\_HELPER\_1(qemu\_op\_insn\_begin\_callback, void, ptr)

//void qemu\_block\_end\_callback(CPUState\* cpu,TranslationBlock\* next\_tb, target\_ulong from);

DEF\_HELPER\_4(qemu\_block\_end\_callback, void, ptr,ptr,tl,tl)

//void qemu\_insn\_begin\_callback(CPUState\* cpu);

DEF\_HELPER\_1(qemu\_insn\_begin\_callback, void, ptr)

//void qemu\_insn\_end\_callback(CPUState\* cpu);

DEF\_HELPER\_1(qemu\_insn\_end\_callback, void, ptr)

//void qemu\_opcode\_range\_callback(CPUState\* cpu, target\_ulong from, target\_ulong to, uint32\_t opcode);

DEF\_HELPER\_5(qemu\_opcode\_range\_callback, void, ptr,tl,tl,i32,tl)

DEF\_HELPER\_1(qemu\_trigger\_cpu\_loop\_exit\_if\_needed, void, ptr)

## qemu\_glue\_callbacks.h

实现qemu helper函数，最基础的glue函数。qemu的状态信息被当做raw CPU，提取信息的时机包括qemu\_block\_begin/end，qemu\_insn\_begin/end，qemu\_opcode\_range，qemu\_trigger\_cpu\_loop\_exit\_if\_needed，这些helper函数会在translation的时候插装进TB，即在translate.c中。

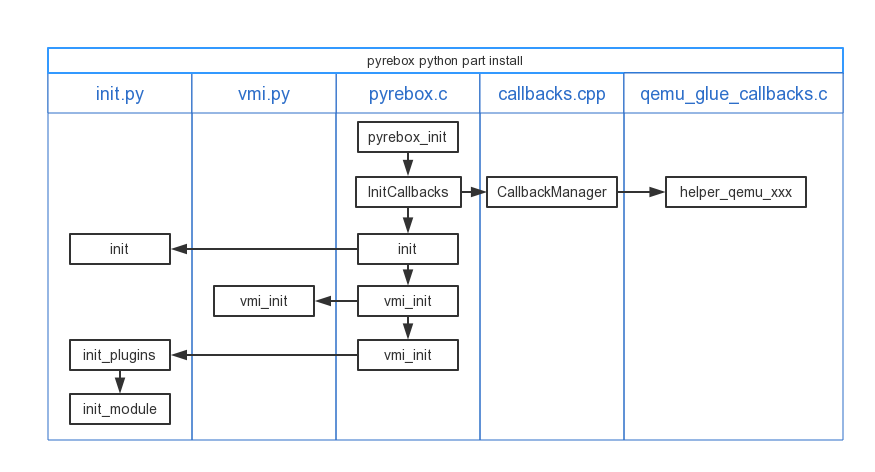
还有2个helper函数在emulation时调用，分别是notify\_cpu\_executing (位于cpu-exec.c的cpu\_exec())，qemu\_tlb\_exec\_callback (位于/qemu/accel/tcg/cputlb.c tlb\_set\_page\_with\_attrs())。

## vmi.cpp/volatility

实现VMI来恢复高层语义，包括basic vmi和volatility插件。

# python接口实现

另外，为了支持ipython，pyrebox做了对qemu做了额外的修改：



## qemu/chardev/char-fd.c：

添加了头文件pyrebox/pyrebox.h和qemu/main-loop.h，在这里处理了ipython和qemu原有的控制字符之间冲突的问题，利用pyrebox\_mutex实现

## callbacks.h/.cpp

实现c callback和python callback的链接。向下通过qemu\_glue\_callbacks.c和qemu通讯，向上提供python接口。定义了callback\_params，class Callback，class CallbackManager。

## api.py/.h/.cpp

实现了python shell cmd。class CallbackManager的python定义。

PyMethodDef api\_methods[] = {

{"register\_callback", register\_callback, METH\_VARARGS, "register\_callback"},

{"unregister\_callback", unregister\_callback, METH\_VARARGS, "unregister\_callback"},

{"r\_pa",r\_pa, METH\_VARARGS, "r\_pa"},

{"r\_va",r\_va, METH\_VARARGS, "r\_va"},

{"r\_cpu",r\_cpu, METH\_VARARGS, "r\_cpu"},

{"w\_pa",w\_pa, METH\_VARARGS, "w\_pa"},

{"w\_va",w\_va, METH\_VARARGS, "w\_va"},

{"w\_r",w\_r, METH\_VARARGS, "w\_r"},

{"w\_sr",w\_sr, METH\_VARARGS, "w\_sr"},

{"r\_ioport",r\_ioport,METH\_VARARGS,"r\_ioport"},

{"w\_ioport",w\_ioport,METH\_VARARGS,"w\_ioport"},

{"va\_to\_pa",va\_to\_pa, METH\_VARARGS, "va\_to\_pa"},

{"start\_monitoring\_process",start\_monitoring\_process, METH\_VARARGS, "start\_monitoring\_process"},

{"is\_monitored\_process",py\_is\_monitored\_process, METH\_VARARGS, "is\_monitored\_process"},

{"stop\_monitoring\_process",stop\_monitoring\_process, METH\_VARARGS, "stop\_monitoring\_process"},

{"get\_running\_process",py\_get\_running\_process, METH\_VARARGS, "get\_running\_process"},

{"is\_kernel\_running",is\_kernel\_running, METH\_VARARGS, "is\_kernel\_running"},

{"save\_vm",save\_vm, METH\_VARARGS, "save\_vm"},

{"load\_vm",load\_vm, METH\_VARARGS, "load\_vm"},

{"add\_trigger",py\_add\_trigger, METH\_VARARGS, "add\_trigger"},

{"remove\_trigger",py\_remove\_trigger, METH\_VARARGS, "remove\_trigger"},

{"set\_trigger\_uint32",set\_trigger\_uint32, METH\_VARARGS, "set\_trigger\_uint32"},

{"set\_trigger\_uint64",set\_trigger\_uint64, METH\_VARARGS, "set\_trigger\_uint64"},

{"set\_trigger\_str",set\_trigger\_str, METH\_VARARGS, "set\_trigger\_str"},

{"get\_trigger\_var",py\_get\_trigger\_var, METH\_VARARGS, "get\_trigger\_var"},

{"call\_trigger\_function",py\_call\_trigger\_function, METH\_VARARGS, "call\_trigger\_function"},

{"vol\_get\_memory\_size",py\_vol\_get\_memory\_size, METH\_VARARGS, "vol\_get\_memory\_size"},

{"vol\_read\_memory",py\_vol\_read\_memory, METH\_VARARGS, "vol\_read\_memory"},

{"vol\_write\_memory",py\_vol\_write\_memory, METH\_VARARGS, "vol\_write\_memory"},

{"get\_process\_list",get\_process\_list, METH\_VARARGS, "get\_process\_list"},

{"get\_num\_cpus",py\_get\_num\_cpus, METH\_VARARGS, "get\_num\_cpus"},

{"plugin\_print\_internal",py\_print\_plugin, METH\_VARARGS, "plugin\_print\_internal"},

{"get\_os\_bits",py\_get\_os\_bits,METH\_VARARGS,"get\_os\_bits"},

{"import\_module",py\_import\_module,METH\_VARARGS,"import\_module"},

{"unload\_module",py\_unload\_module,METH\_VARARGS,"unload\_module"},

{"reload\_module",py\_reload\_module,METH\_VARARGS,"reload\_module"},

{"get\_loaded\_modules",py\_get\_loaded\_modules,METH\_VARARGS,"get\_loaded\_modules"},

{ NULL, NULL, 0, NULL }

};

# Script编写

以hook api为例，原script库中包含对NtAllocateVirtualMemory的hook，尝试添加对RtlAllocateheap的hook。

利用requirements = ["plugins.guest\_agent"]可以指定scripts之间的依赖，先import related scripts