QEMU的最大亮点就是动态翻译技术，正是由于这个强劲的引擎，使QEMU可以在不使用任何加速技术的情况下也能达到良好的速度，并能够横跨多种平台运行，借助于特定版本的GCC编译器，还能够仿真多种架构的处理器。这里我说的动态翻译指的是QEMU早期版本使用的“dynamic translation”，因为从0.10版本开始使用的是“TCG”，摆脱了对GCC版本的依赖，并且不再需要编译中间工具。

简单来说，动态翻译的基本思想就是把每一条x86指令切分成为若干条微指令，每条微指令由一段简单的C代码来实现（见'target-i386/op.c'），然后通过中间工具（'dyngen'）提取相应的目标文件（'op.o'）来生成一个动态代码生成器，最后把这些微指令组合成一个函数（见'op.h:dyngen\_code()'）。

在一个真实的CPU里，执行流程由取指、译指、执行指令三部分组成。在QEMU仿真的处理器中同样如此，取指和执行指令不需多说，关键的是译指这道工序，由反汇编器、dyngen程序、动态代码生成器三部分来共同完成。我的实验环境是X86平台＋0.7.2版本源码，这里我以BIOS启动代码的第一条指令jmp f000:e05b来详细说明，该指令的汇编代码是：EA 5B E0 00 F0，反汇编器首先分析EA，知道这是一条16位的跳转指令，因此接着取出后面的EIP和CS。具体过程在 translate.c:disas\_insn() 可见，它被分解为如下几条微指令：

gen\_op\_movl\_T0\_im(selector); // 把0xf000放到T0中

gen\_op\_movl\_T1\_imu(offset); // 把0xe05b放到T1中

gen\_op\_movl\_seg\_T0\_vm(offsetof(CPUX86State,segs[R\_CS])); // 把T0的值放到env结构的CS段寄存器变量中

gen\_op\_movl\_T0\_T1(); // T1 -> T0

gen\_op\_jmp\_T0(); // 跳转到T0

gen\_op\_movl\_T0\_0(); // 0 -> T0

gen\_op\_exit\_tb(); // 返回

它们的实现函数分别如下：

static inline void gen\_op\_movl\_T0\_im(long param1)

{

\*gen\_opparam\_ptr++ = param1;

\*gen\_opc\_ptr++ = INDEX\_op\_movl\_T0\_im;

}

static inline void gen\_op\_movl\_T1\_imu(long param1)

{

\*gen\_opparam\_ptr++ = param1;

\*gen\_opc\_ptr++ = INDEX\_op\_movl\_T1\_imu;

}

static inline void gen\_op\_movl\_seg\_T0\_vm(long param1)

{

\*gen\_opparam\_ptr++ = param1;

\*gen\_opc\_ptr++ = INDEX\_op\_movl\_seg\_T0\_vm;

}

static inline void gen\_op\_movl\_T0\_T1(void)

{

\*gen\_opc\_ptr++ = INDEX\_op\_movl\_T0\_T1;

}

static inline void gen\_op\_jmp\_T0(void)

{

\*gen\_opc\_ptr++ = INDEX\_op\_jmp\_T0;

}

static inline void gen\_op\_movl\_T0\_0(void)

{

\*gen\_opc\_ptr++ = INDEX\_op\_movl\_T0\_0;

}

static inline void gen\_op\_exit\_tb(void)

{

\*gen\_opc\_ptr++ = INDEX\_op\_exit\_tb;

}

可以看出，以上函数都非常简单，其实就是在操作码缓冲区中放一个索引号。真正调用的函数在op.c中，如下：

void OPPROTO op\_movl\_T0\_im(void)

{

T0 = (int32\_t)PARAM1;

}

void OPPROTO op\_movl\_T1\_imu(void)

{

T1 = (uint32\_t)PARAM1;

}

void OPPROTO op\_movl\_seg\_T0\_vm(void)

{

int selector;

SegmentCache \*sc;

selector = T0 & 0xffff;

/\* env->segs[] access \*/

sc = (SegmentCache \*)((char \*)env + PARAM1);

sc->selector = selector;

sc->base = (selector << 4);

}

void OPPROTO op\_movl\_T0\_T1(void)

{

T0 = T1;

}

void OPPROTO op\_jmp\_T0(void)

{

EIP = T0;

}

void OPPROTO op\_movl\_T0\_0(void)

{

T0 = 0;

}

#define EXIT\_TB() asm volatile ("ret")

void OPPROTO op\_exit\_tb(void)

{

EXIT\_TB();

}

在我的实验环境中，T0和T1的定义如下：

#define T0 (env->t0)

#define T1 (env->t1)

t0和t1都是长整型，分别是env结构的第1和第2个成员变量。上述函数被编译在目标文件op.o，在执行时经过 op.h:dyngen\_code 动态翻译后，以上微指令运行在Host上的实际代码如下：

mov eax,dword ptr [env (1FD1F14h)] // -> gen\_op\_movl\_T0\_im(selector)

mov dword ptr [eax],0F000h

mov eax,dword ptr [env (1FD1F14h)] // -> gen\_op\_movl\_T1\_imu(offset)

mov dword ptr [eax+4],0E05Bh

mov edx,dword ptr [env (1FD1F14h)] // -> gen\_op\_movl\_seg\_T0\_vm(offsetof(CPUX86State,segs[R\_CS]))

mov eax,dword ptr [edx]

and eax,0FFFFh

mov dword ptr [edx+58h],eax

shl eax,4

mov dword ptr [edx+5Ch],eax

mov edx,dword ptr [env (1FD1F14h)] // -> gen\_op\_movl\_T0\_T1()

mov eax,dword ptr [edx+4]

mov dword ptr [edx],eax

mov edx,dword ptr [env (1FD1F14h)] // -> gen\_op\_jmp\_T0()

mov eax,dword ptr [edx]

mov dword ptr [edx+2Ch],eax

mov eax,dword ptr [env (1FD1F14h)] // -> gen\_op\_movl\_T0\_0()

mov dword ptr [eax],0

ret // -> gen\_op\_exit\_tb()

现在可以清楚看到了，这就是Target上一条JMP指令在Host上的对应代码实现。

本来还应该再讲讲 rep、call 之类的指令，因为这也是QEMU比其它仿真器（如Bochs之类）快的原因之一，包括翻译后指令的重用、一次性执行多条Target指令、直接使用常量等特性，但是发现打字实在是很累，代码多了大家也看的眼花，所以就先说到这里吧。论坛上高手很多，希望有感兴趣的来一起讨论下。