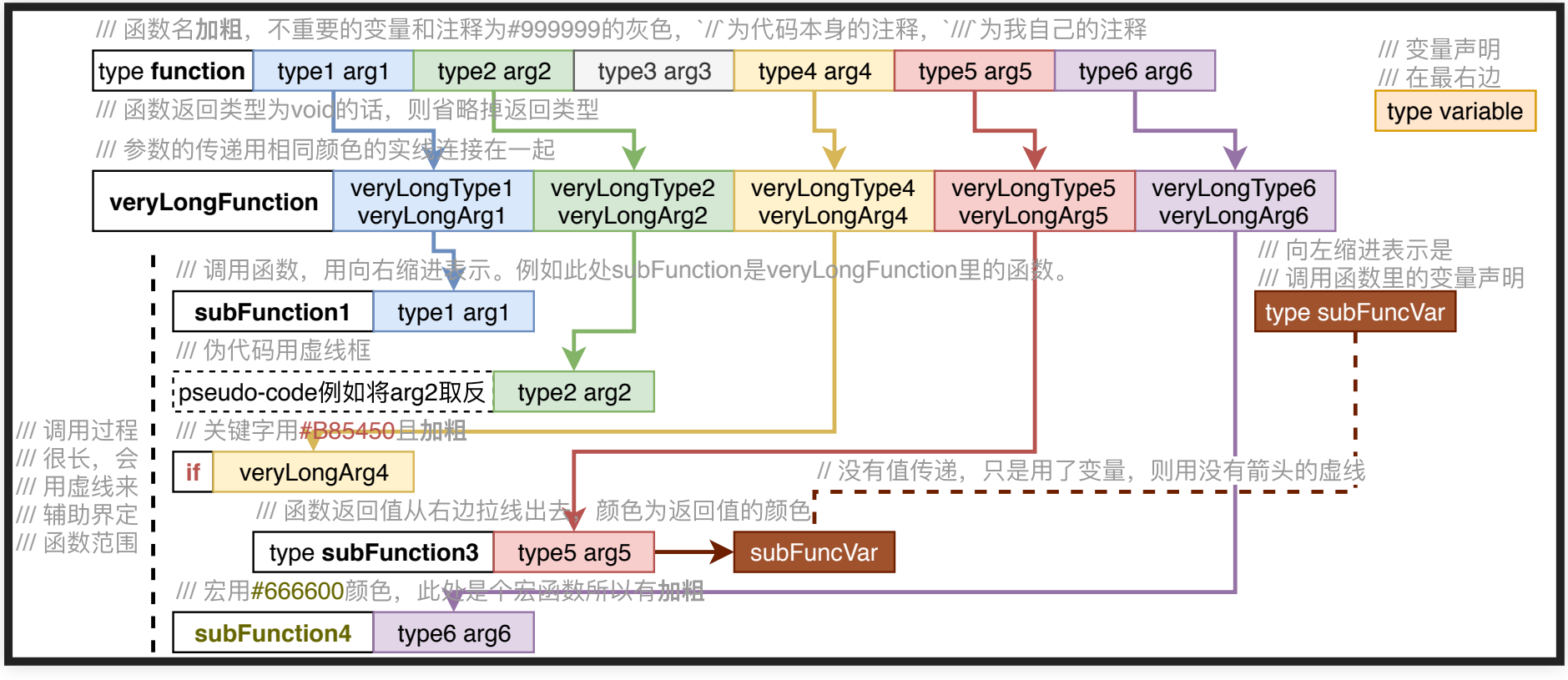
QEMU系统调用框架

THE OUTLINE OF QEMU SYSTEM CALL

谢本壹 2019.11.28

前言

报告中会用到代码框架图，框架图的约定如下：



采用draw.io绘制

OUTLINE

OUTLINE

1. 问题出发点

OUTLINE

1. 问题出发点
2. 动态翻译器翻译过程的特点

OUTLINE

1. 问题出发点
2. 动态翻译器翻译过程的特点
3. X86toMips处理系统调用的方法

OUTLINE

1. 问题出发点
2. 动态翻译器翻译过程的特点
3. X86toMips处理系统调用的方法
4. QEMU处理系统调用的方法

OUTLINE

1. 问题出发点
2. 动态翻译器翻译过程的特点
3. X86toMips处理系统调用的方法
4. QEMU处理系统调用的方法
5. 总结

问题出发点

用QEMU系统调用还是修补X86toMips系统调用?

动态翻译器翻译过程的特点

分成两个部分

1. 指令翻译成本地码
2. 执行本地码

X86TOMIPS处理系统调用的方法

1. X86toMips指令翻译成本地码
2. X86toMips执行本地码

1. X86TOMIPS指令翻译成本地码

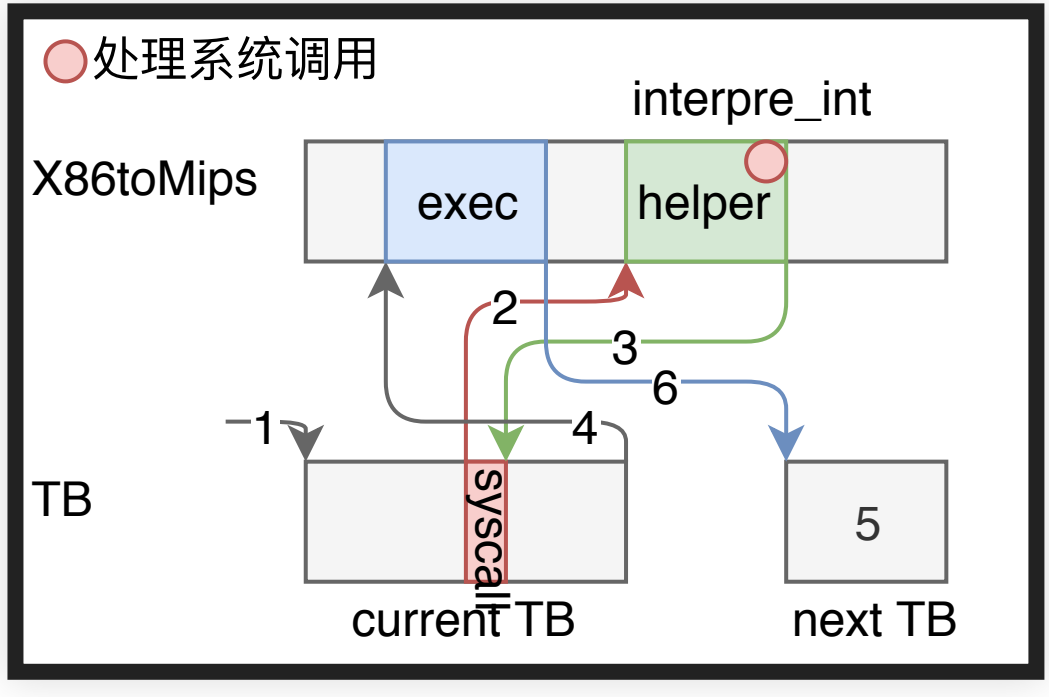
```
// translator/tr_misc.cpp
translate_int(IR1_INST *pir1)
```

将如下的指令放入TB中，

- 保存上下文
- 解释系统调用`interpret_int`
- 还原上下文

2. X86TOMIPS执行本地码

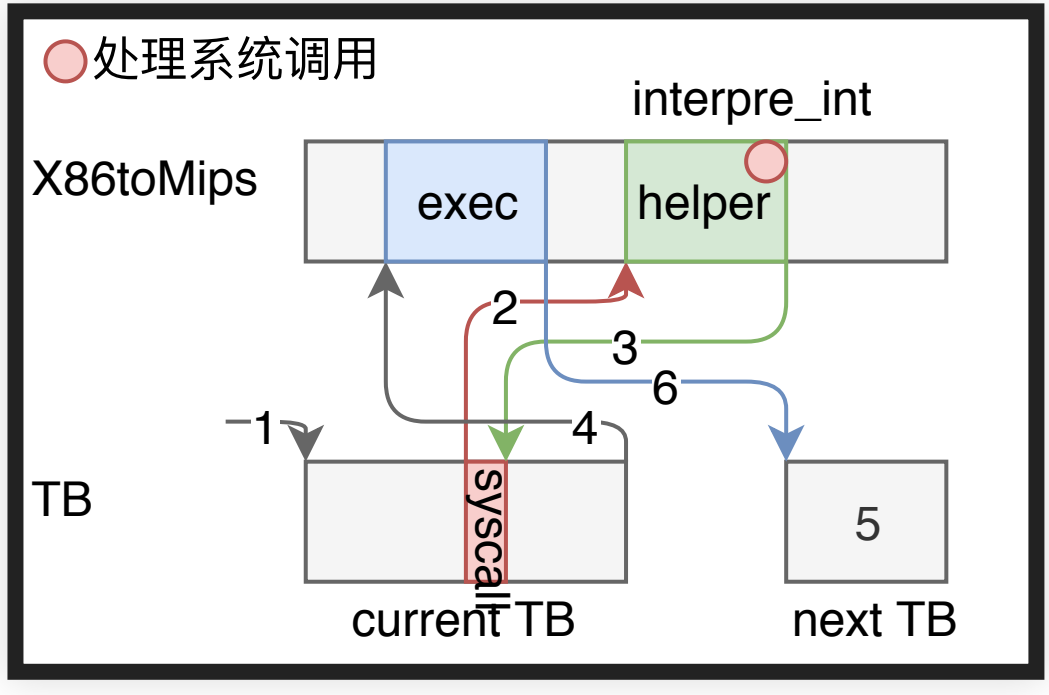
控制流图（首次执行）



X86toMips是翻译一个TB然后执行一个TB

2. X86TOMIPS执行本地码

控制流图（首次执行）



1. 开始执行当前TB
2. 保存当前TB上下文然后转到系统调用的helper函数
3. 从helper函数返回当前TB且恢复上下文
4. 当前TB执行完成，回到exec函数，寻找下一个TB
5. 发现没有TB了，生成下一个TB（没有进行反汇编，也没有进行翻译）
6. 开始执行下一个TB

X86toMips是翻译一个TB然后执行一个TB

interpret_int的不足

只支持x86-32且支持的系统调用不全。

用QEMU系统调用还是修补X86toMips系统调用?

所以有必要研究清楚QEMU处理系统调用的方法。

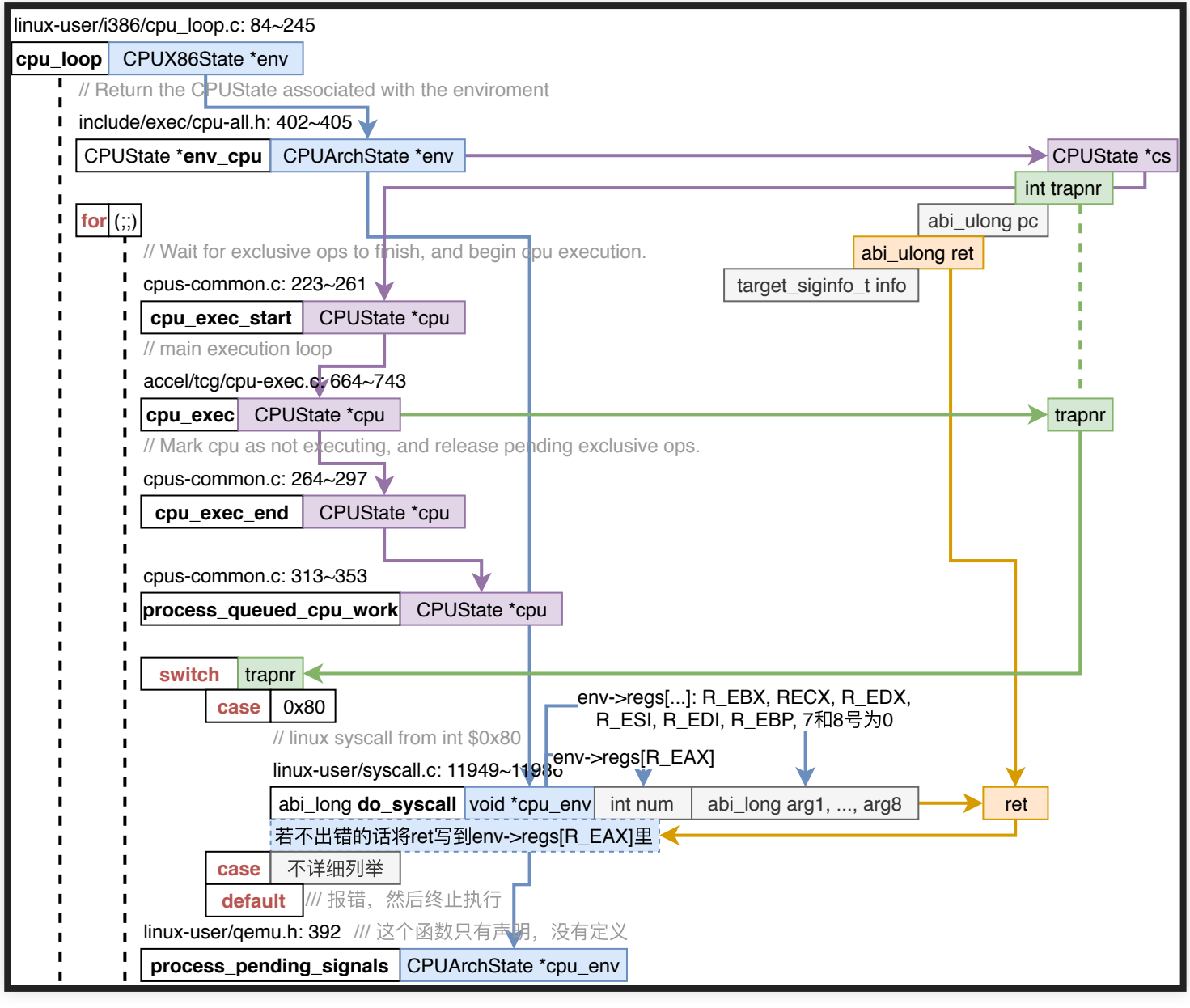
QEMU处理系统调用的方法

1. QEMU执行本地码
2. QEMU指令翻译成本地码

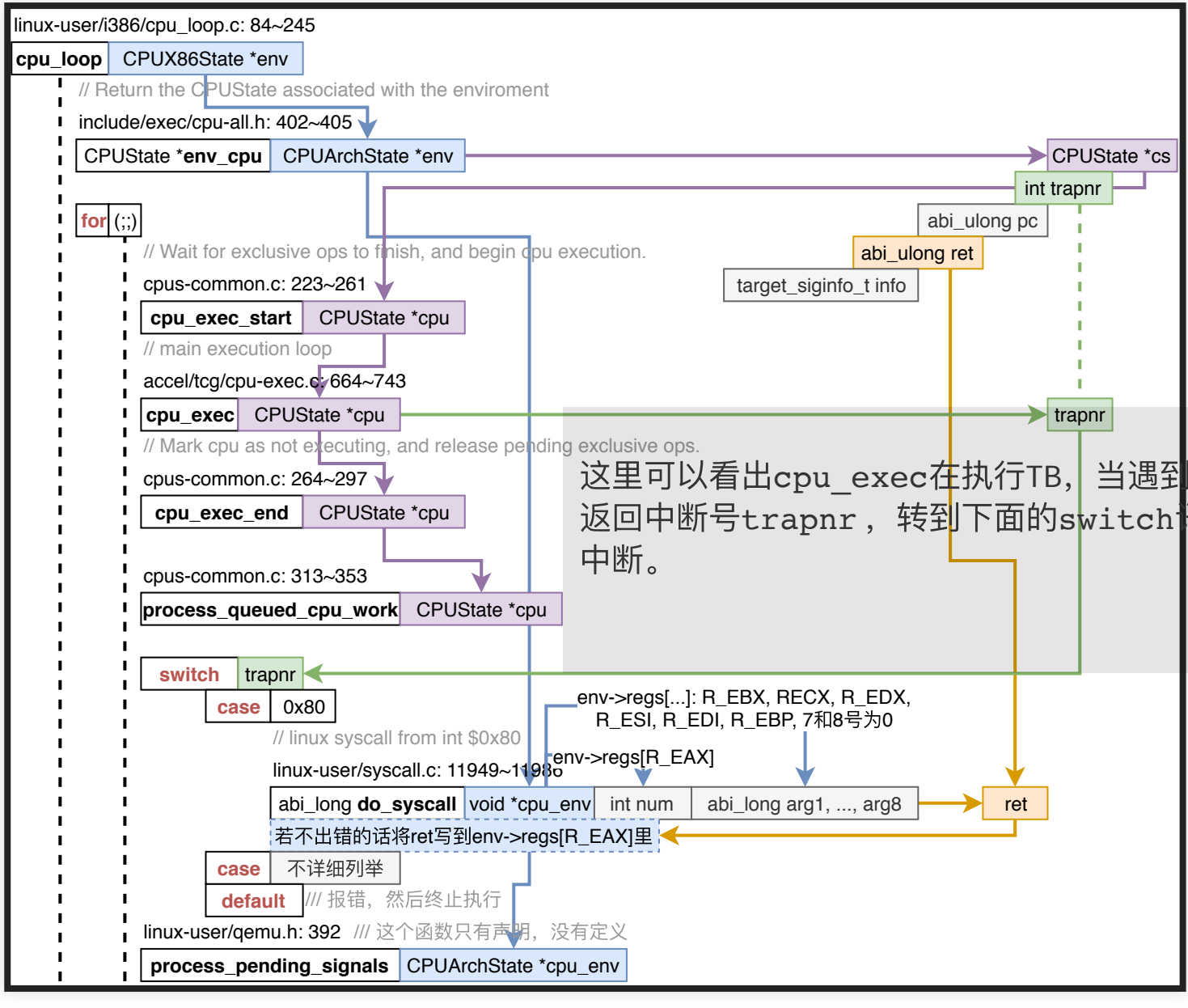
1. QEMU执行本地码

- QEMU用户级模拟框架图
- do_syscall框架图
- cpu_exec框架图

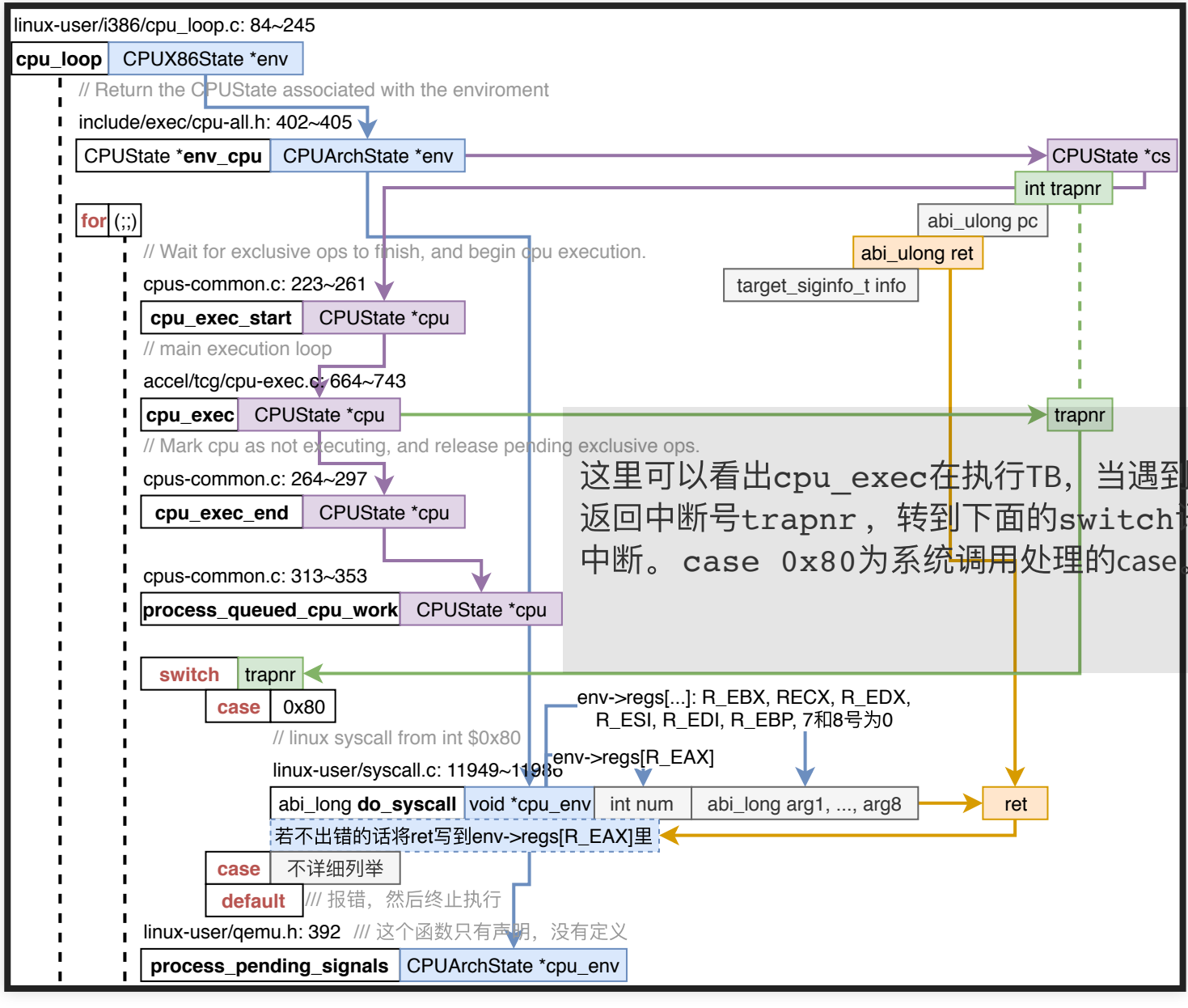
QEMU用户级模拟框架图



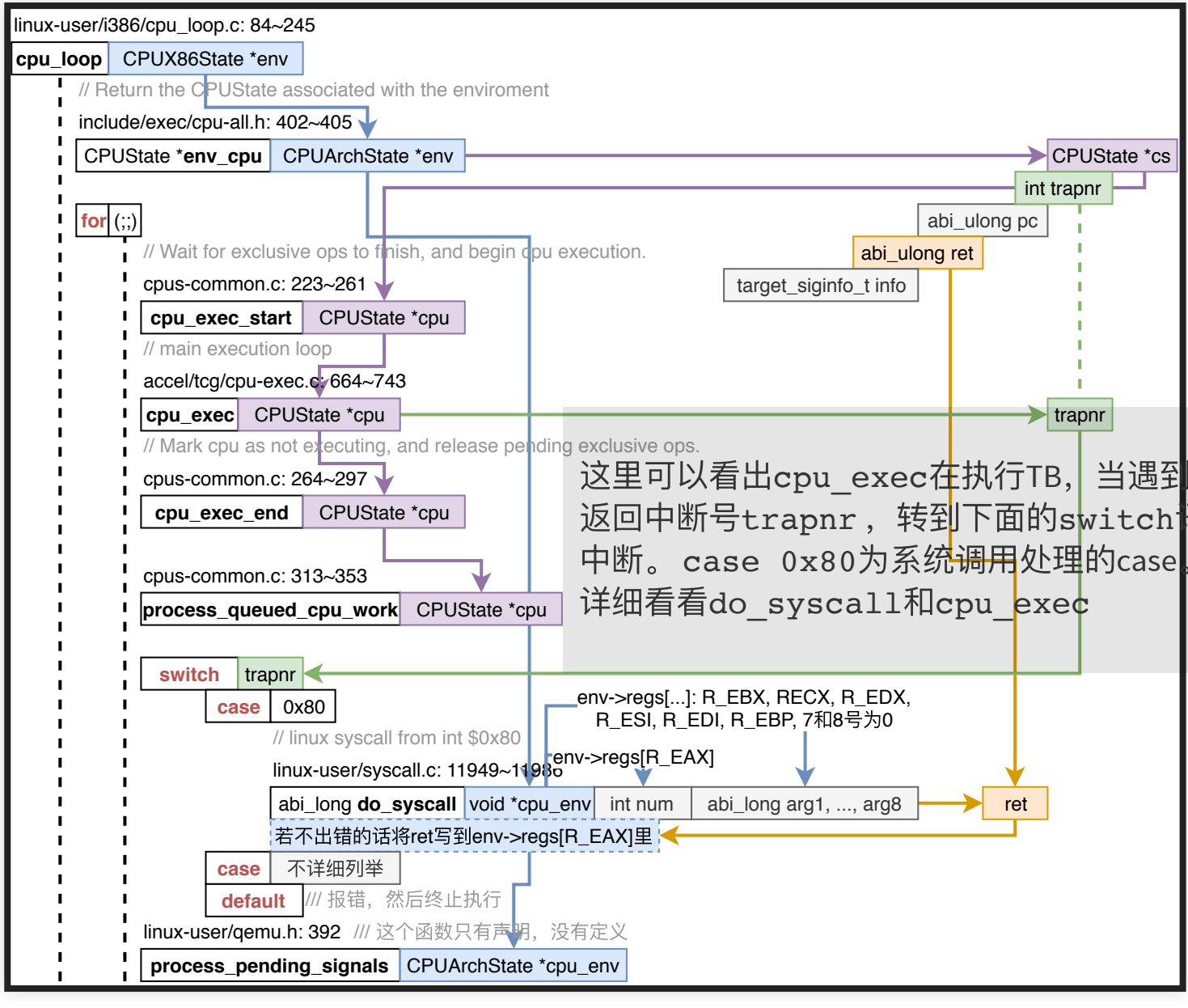
QEMU用户级模拟框架图



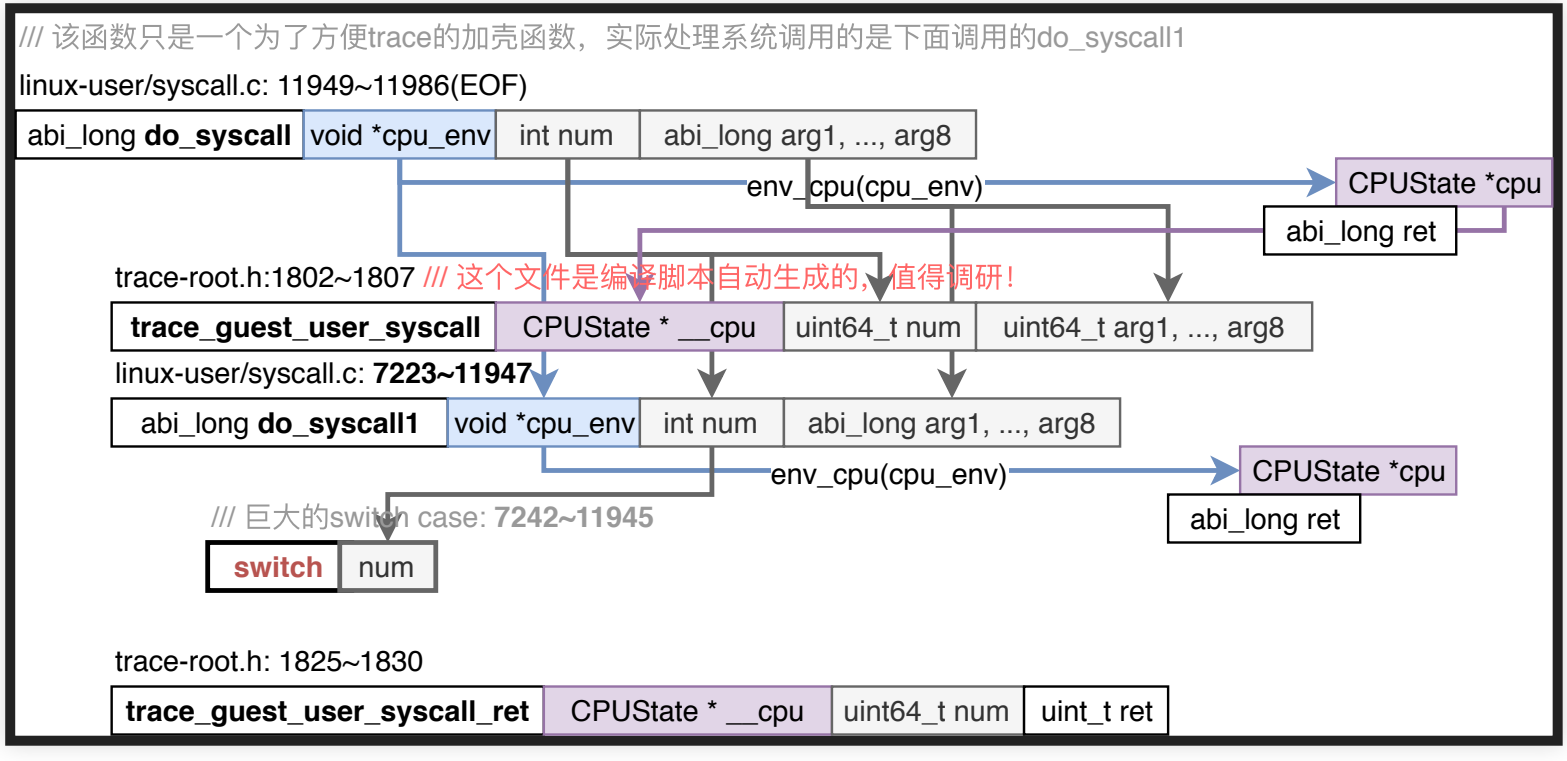
QEMU用户级模拟框架图



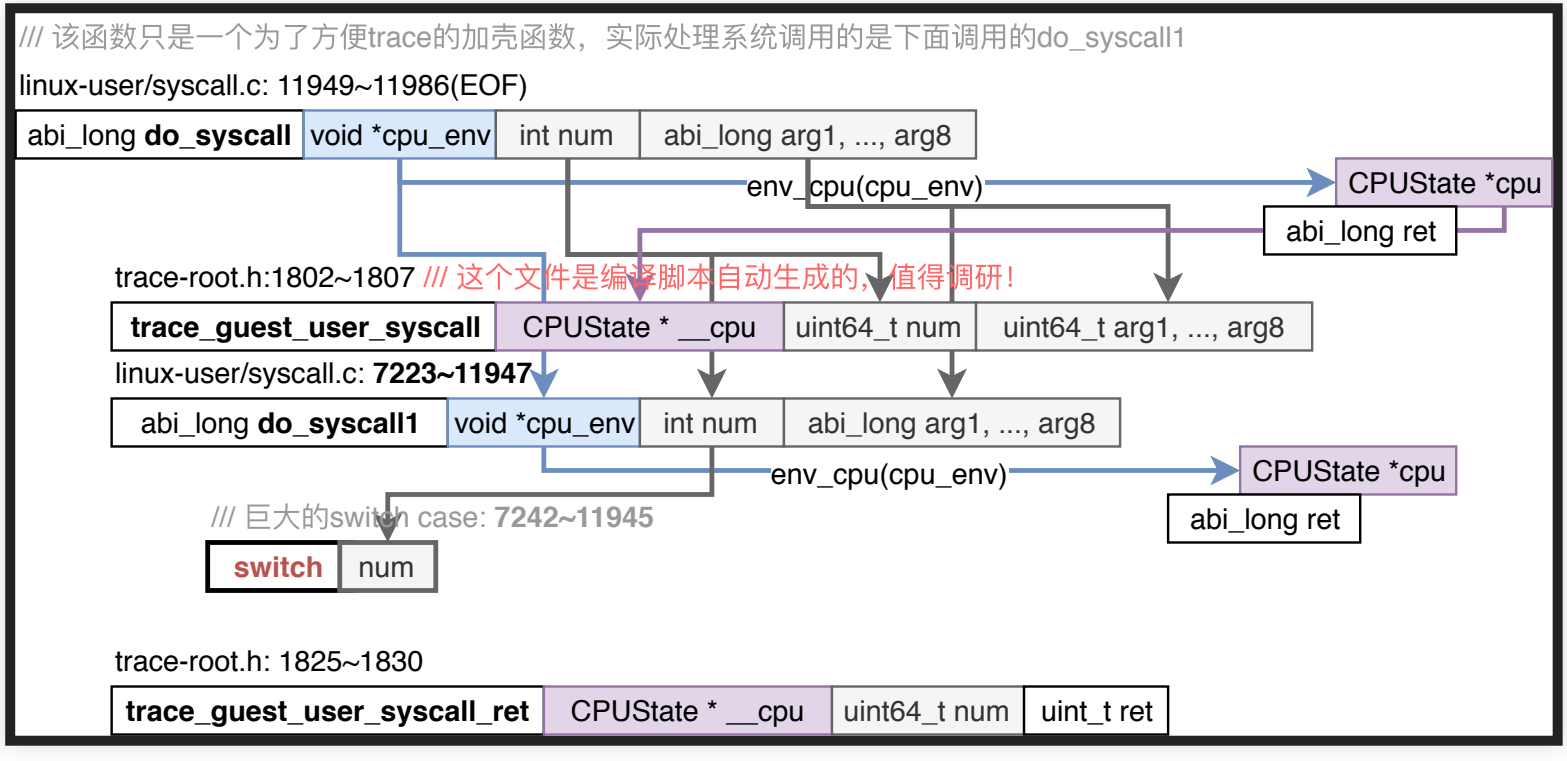
QEMU用户级模拟框架图



do_syscall框架图

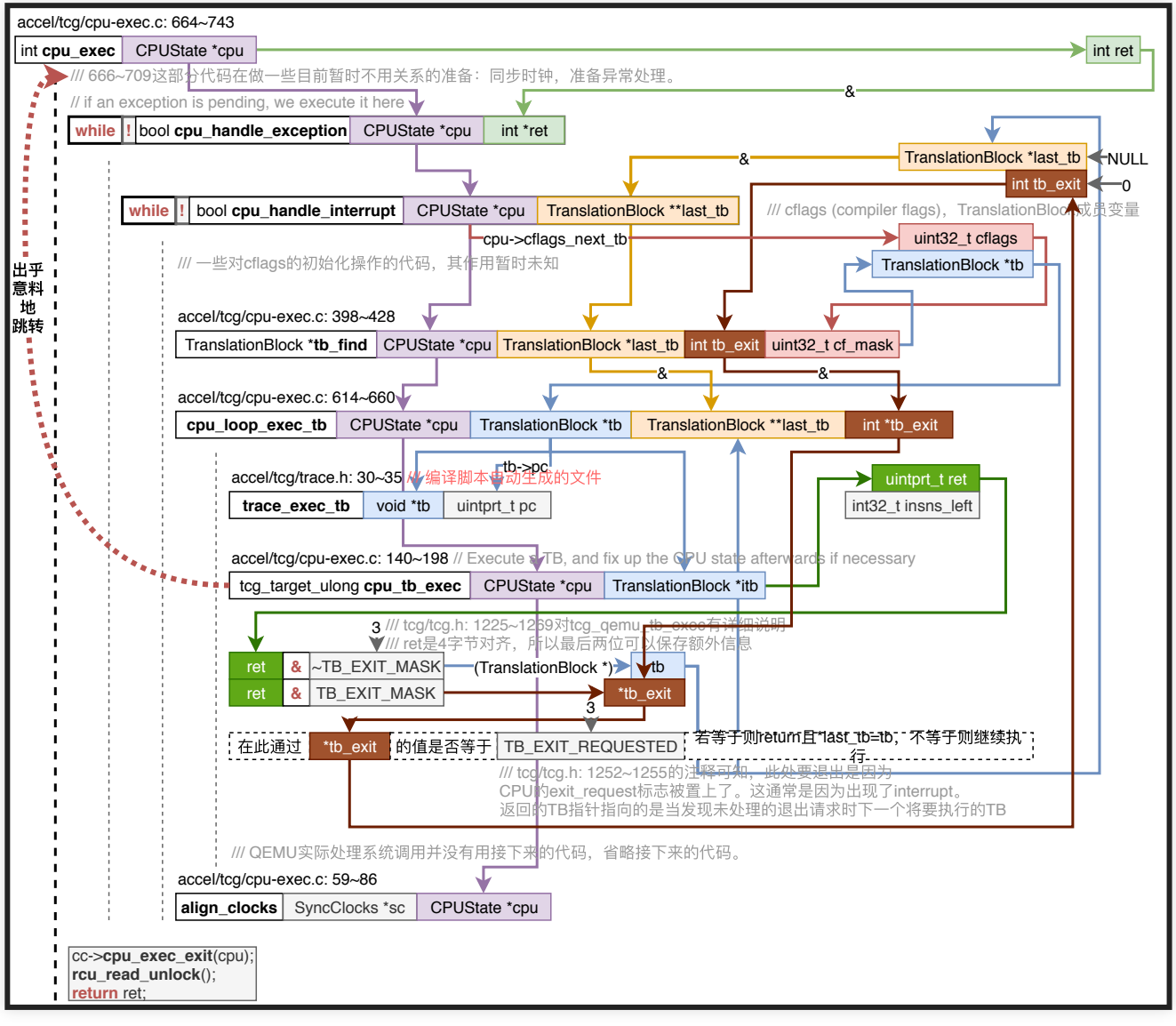


do_syscall框架图

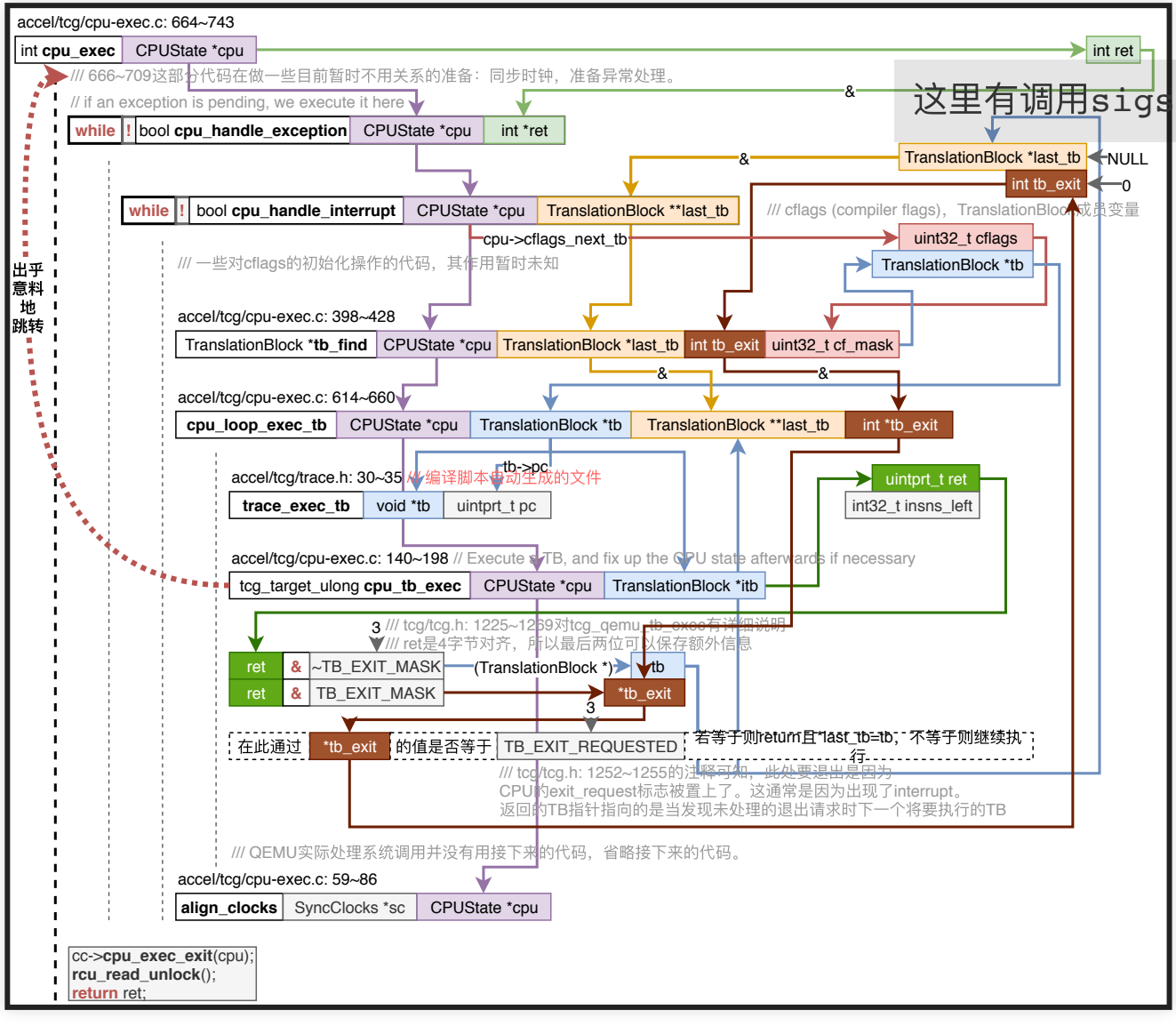


QEMU在这里完成了对x86系统调用的模拟，包括了32/64ABI的转换问题。

cpu_exec框架图

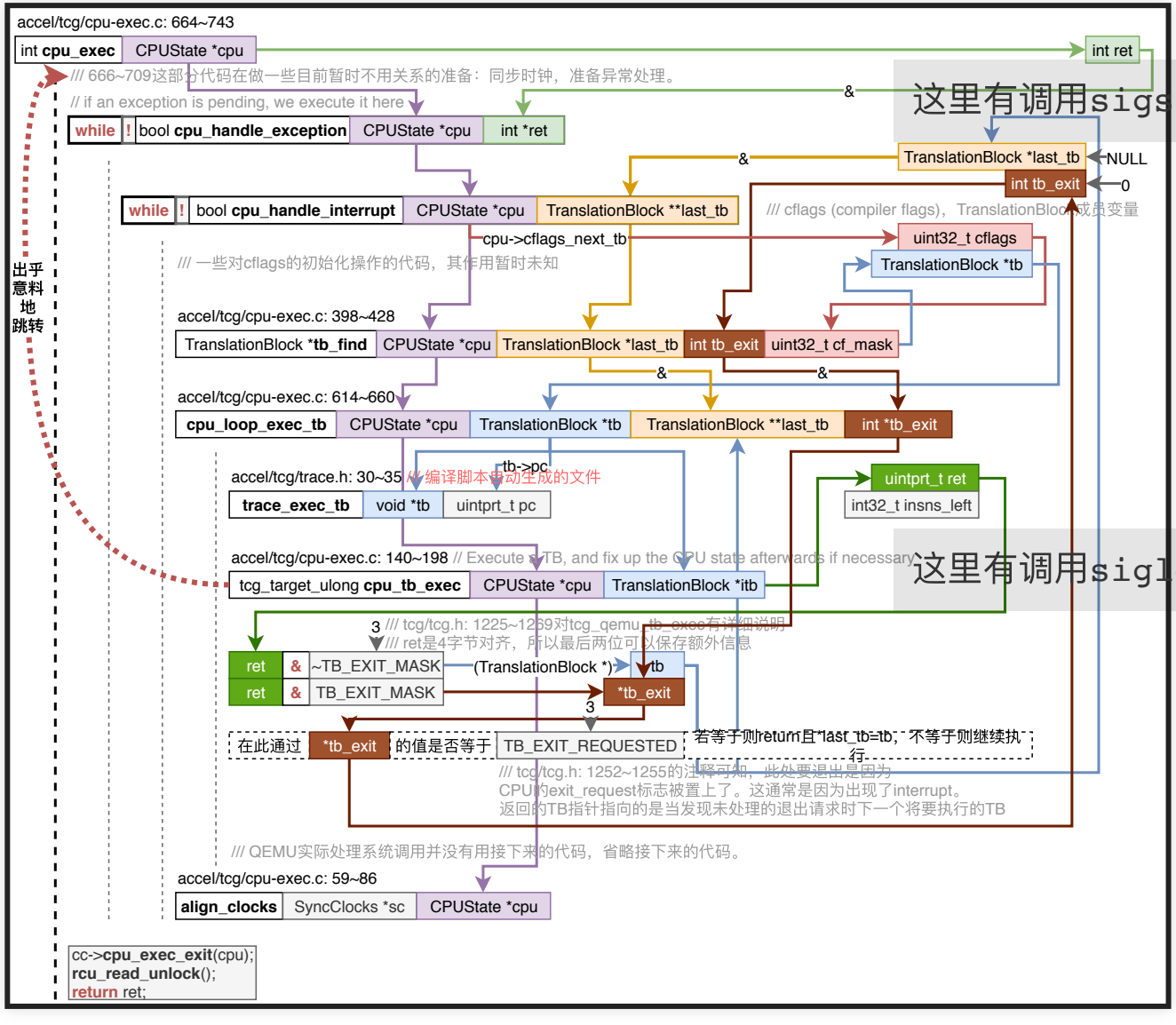

accel/tcg/cpu-exec.c: 664~743
int cpu_exec
CPUState *cpu
int ret
/// 666~709这部分代码在做一些目前暂时不用关系的准备：同步时钟，准备异常处理。
// if an exception is pending, we execute it here
while
!
bool cpu_handle_exception
CPUState *cpu
int *ret
&
TranslationBlock *last_tb
NULL
int tb_exit
0
while
!
bool cpu_handle_interrupt
CPUState *cpu
TranslationBlock **last_tb
/// cflags (compiler flags)，TranslationBlock成员变量
cpu->cflags_next_tb
uint32_t cflags
TranslationBlock *tb
/// 一些对cflags的初始化操作的代码，其作用暂时未知
出乎意料地跳转
accel/tcg/cpu-exec.c: 398~428
TranslationBlock *tb_find
CPUState *cpu
TranslationBlock *last_tb
int tb_exit
uint32_t cf_mask
&
&
accel/tcg/cpu-exec.c: 614~660
cpu_loop_exec_tb
CPUState *cpu
TranslationBlock *tb
TranslationBlock **last_tb
int *tb_exit
tb->pc
accel/tcg/trace.h: 30~35 //编译脚本自动生成的文件
trace_exec_tb
void *tb
uintprt_t pc
uintprt_t ret
int32_t insns_left
accel/tcg/cpu-exec.c: 140~198 // Execute a TB, and fix up the CPU state afterwards if necessary
tcg_target_ulong cpu_tb_exec
CPUState *cpu
TranslationBlock *itb
3
/// tcg/tcg.h: 1225~1259对tcg_qemu_tb_exec有详细说明
/// ret是4字节对齐，所以最后两位可以保存额外信息
ret
&
~TB_EXIT_MASK
(TranslationBlock *)
tb
ret
&
TB_EXIT_MASK
*tb_exit
3
在此通过
*tb_exit
的值是否等于
TB_EXIT_REQUESTED
若等于则return且*last_tb=tb；不等于则继续执行
/// tcg/tcg.h: 1252~1255的注释可知，此处要退出是因为
CPU的exit_request标志被置上了。这通常是因为出现了interrupt。
返回的TB指针指向的是当发现未处理的退出请求时下一个将要执行的TB
/// QEMU实际处理系统调用并没有用接下来的代码，省略接下来的代码。
accel/tcg/cpu-exec.c: 59~86
align_clocks
SyncClocks *sc
CPUState *cpu
cc->cpu_exec_exit(cpu);
rcu_read_unlock();
return ret;

cpu_exec框架图



这里有调用sigsetjmp函数。

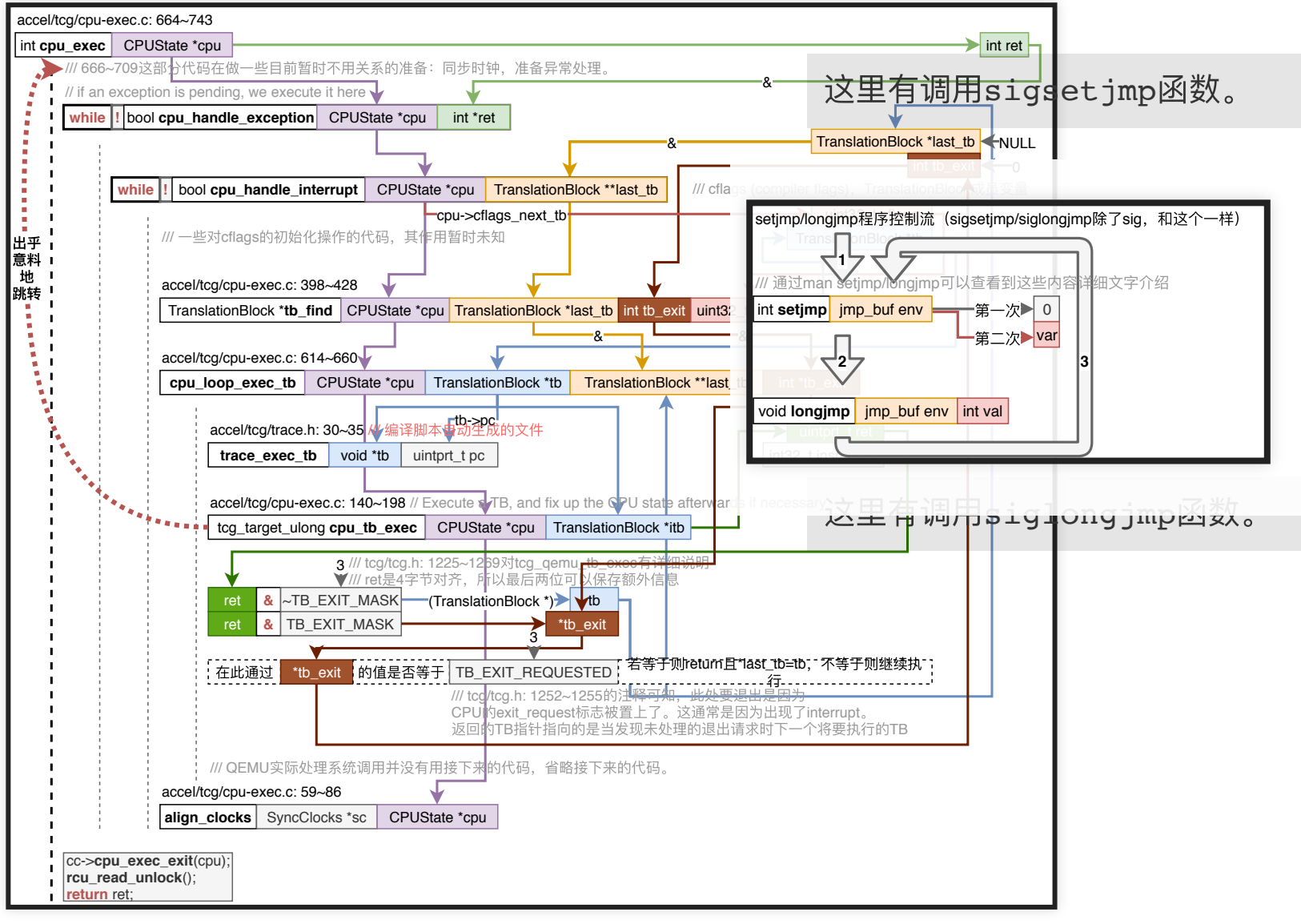
cpu_exec框架图



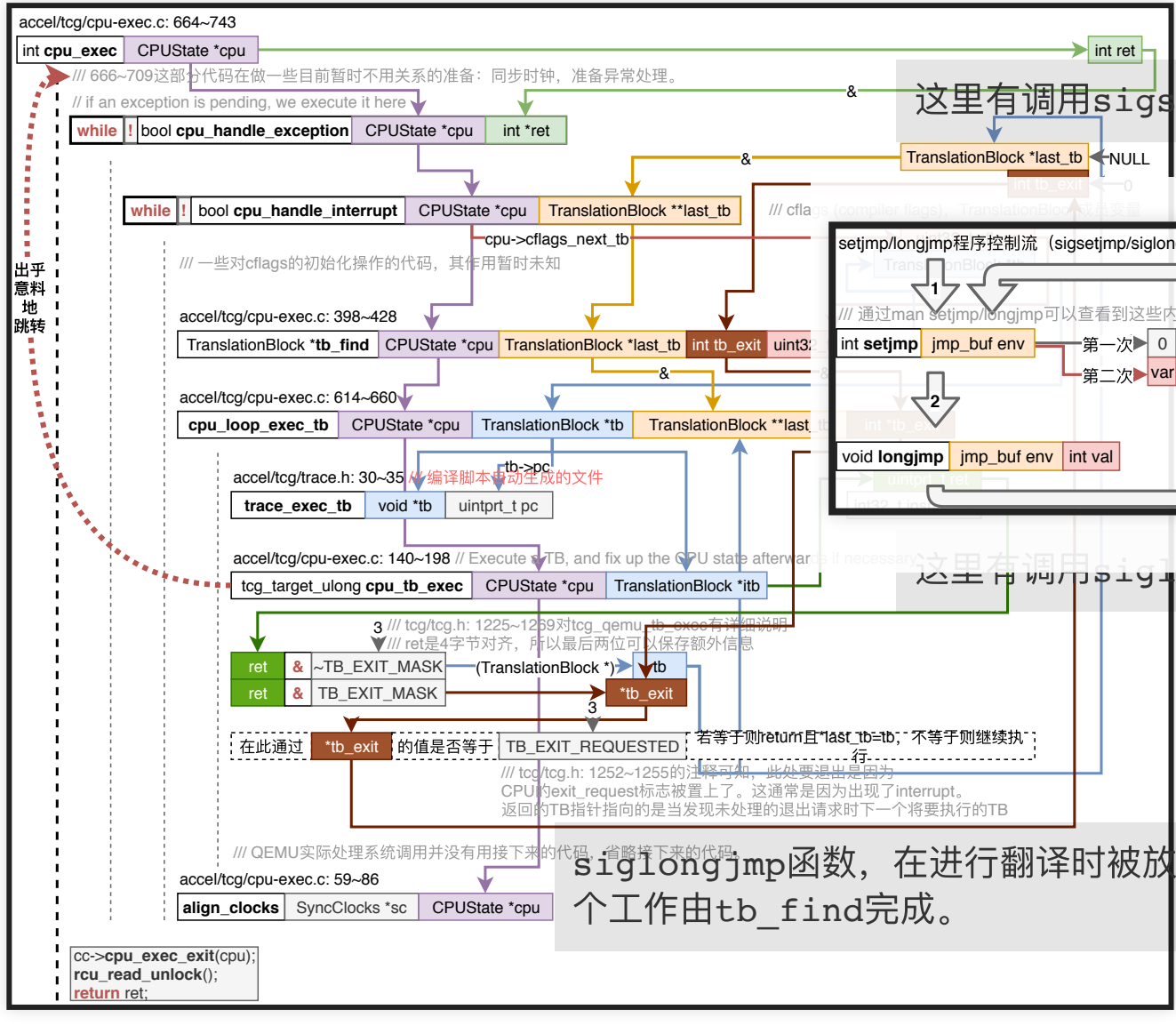
这里有调用sigsetjmp函数。

这里有调用siglongjmp函数。

cpu_exec框架图


accel/tcg/cpu-exec.c: 664~743
int cpu_exec CPUState *cpu
int ret
/// 666~709这部分代码在做一些目前暂时不用关系的准备：同步时钟，准备异常处理。
// if an exception is pending, we execute it here
while ! bool cpu_handle_exception CPUState *cpu int *ret
&
TranslationBlock *last_tb NULL
这里有调用sigsetjmp函数。
while ! bool cpu_handle_interrupt CPUState *cpu TranslationBlock **last_tb
cpu->cflags_next_tb
/// 一些对cflags的初始化操作的代码，其作用暂时未知
出乎意料地跳转
setjmp/longjmp程序控制流（sigsetjmp/siglongjmp除了sig，和这个一样）
1
/// 通过man setjmp/longjmp可以查看到这些内容详细文字介绍
int setjmp jmp_buf env
第一次 0
第二次 var
2
3
void longjmp jmp_buf env int val
accel/tcg/cpu-exec.c: 398~428
TranslationBlock *tb_find CPUState *cpu TranslationBlock *last_tb int tb_exit
&
accel/tcg/cpu-exec.c: 614~660
cpu_loop_exec_tb CPUState *cpu TranslationBlock *tb TranslationBlock **last
tb->pc
accel/tcg/trace.h: 30~35 /// 编译脚本自动生成的文件
trace_exec_tb void *tb uintprt_t pc
accel/tcg/cpu-exec.c: 140~198 // Execute a TB, and fix up the CPU state afterwar
tcg_target_ulong cpu_tb_exec CPUState *cpu TranslationBlock *itb
这里有调用siglongjmp函数。
3 /// tcg/tcg.h: 1225~1259对tcg_qemu_tb_exec有详细说明
/// ret是4字节对齐，所以最后两位可以保存额外信息
ret & ~TB_EXIT_MASK
ret & TB_EXIT_MASK
(TranslationBlock *)
tb
*tb_exit
3
在此通过 *tb_exit 的值是否等于 TB_EXIT_REQUESTED 若等于则return且*last_tb=tb；不等于则继续执行。
/// tcg/tcg.h: 1252~1255的注释可知，此处要退出是因为
CPU的exit_request标志被置上了。这通常是因为出现了interrupt。
返回的TB指针指向的是当发现未处理的退出请求时下一个将要执行的TB
/// QEMU实际处理系统调用并没有用接下来的代码，省略接下来的代码。
accel/tcg/cpu-exec.c: 59~86
align_clocks SyncClocks *sc CPUState *cpu
cc->cpu_exec_exit(cpu);
rcu_read_unlock();
return ret;

cpu_exec框架图



这里有调用sigsetjmp函数。

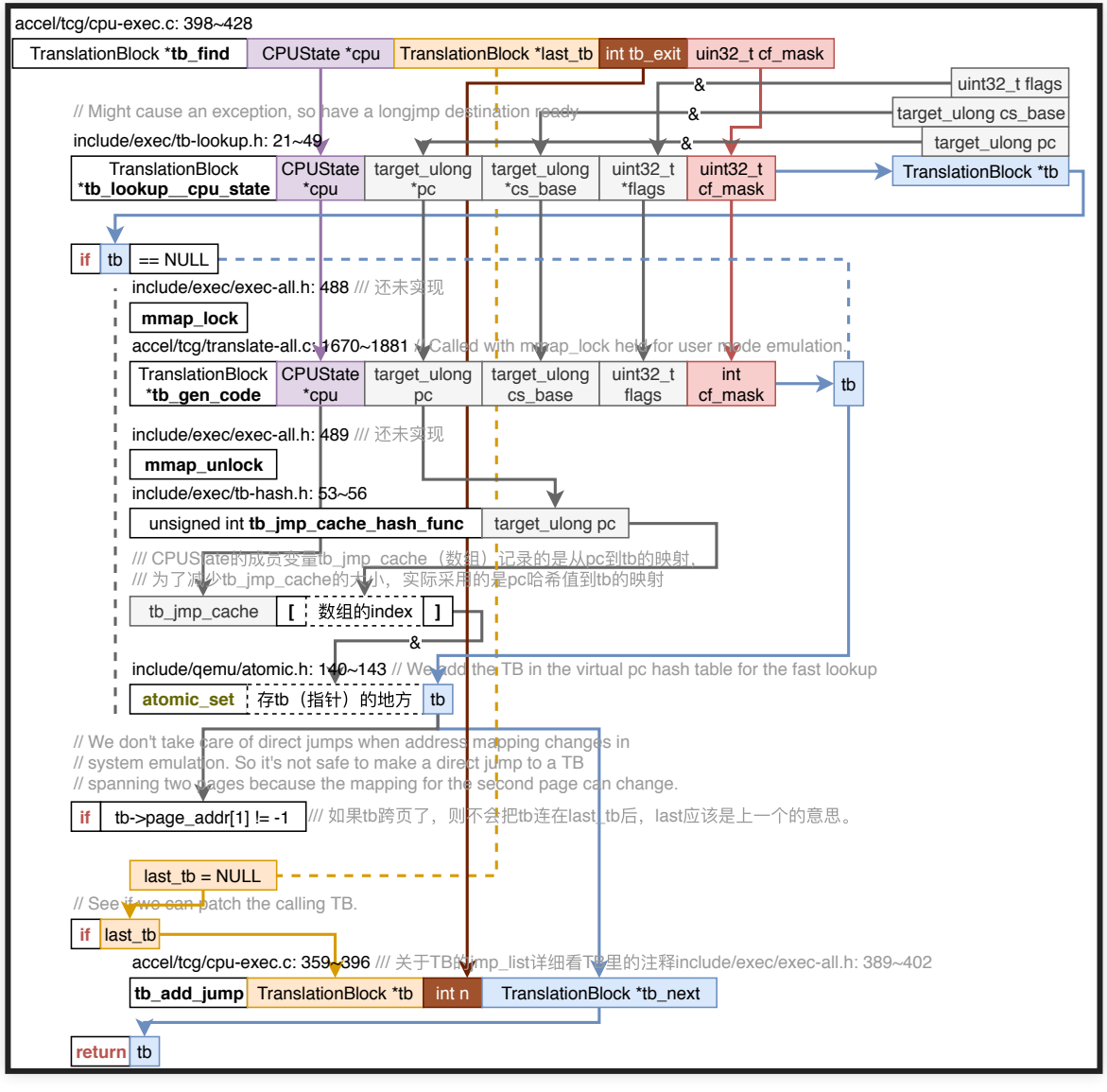
这里有调用siglongjmp函数。

siglongjmp函数，在进行翻译时被放进了TB里。这个工作由tb_find完成。

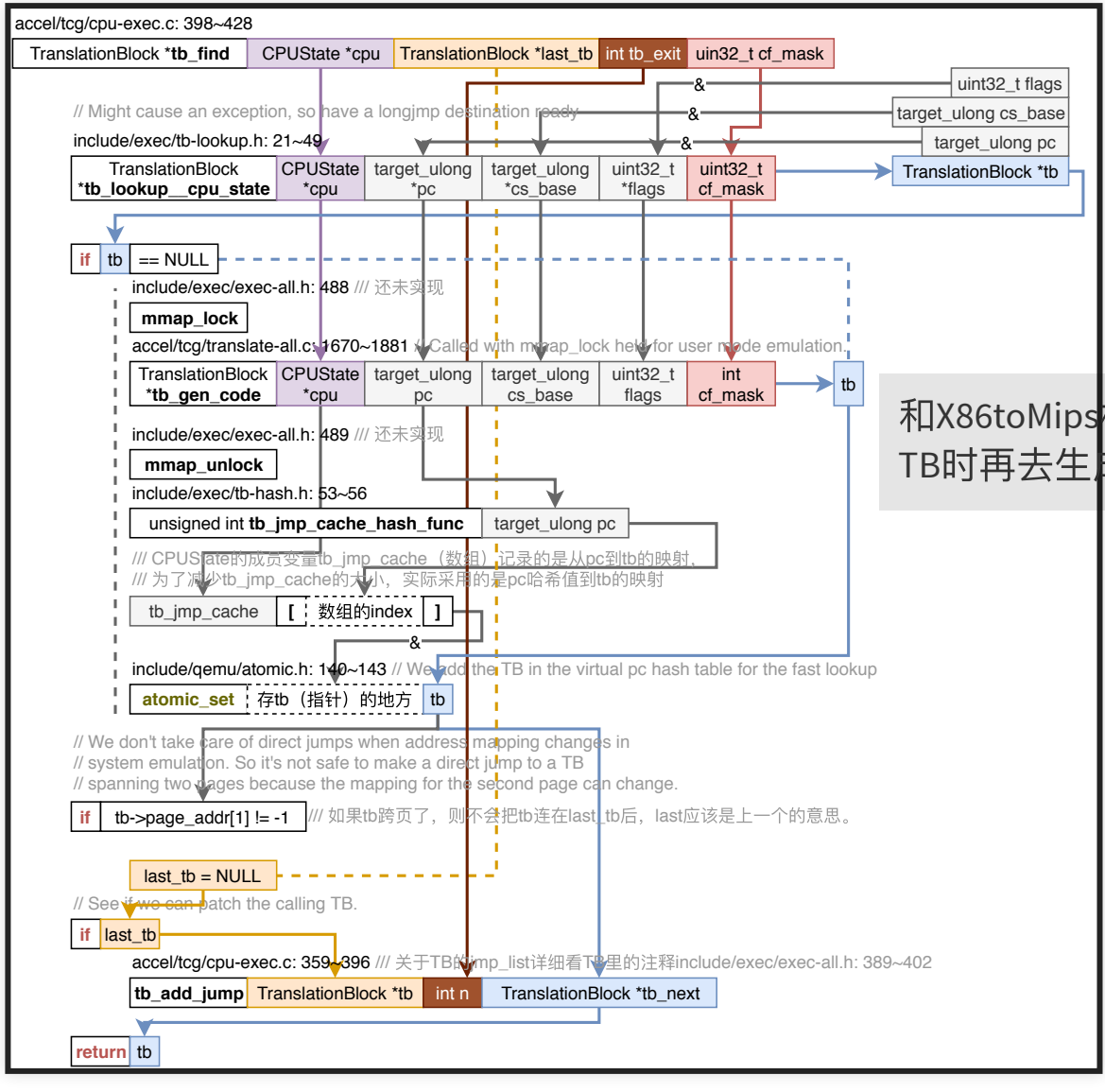
2. QEMU指令翻译成本地码

1. tb_find框架图
2. tb_gen_code框架图
3. QEMU翻译过程控制流图

tb_find框架图

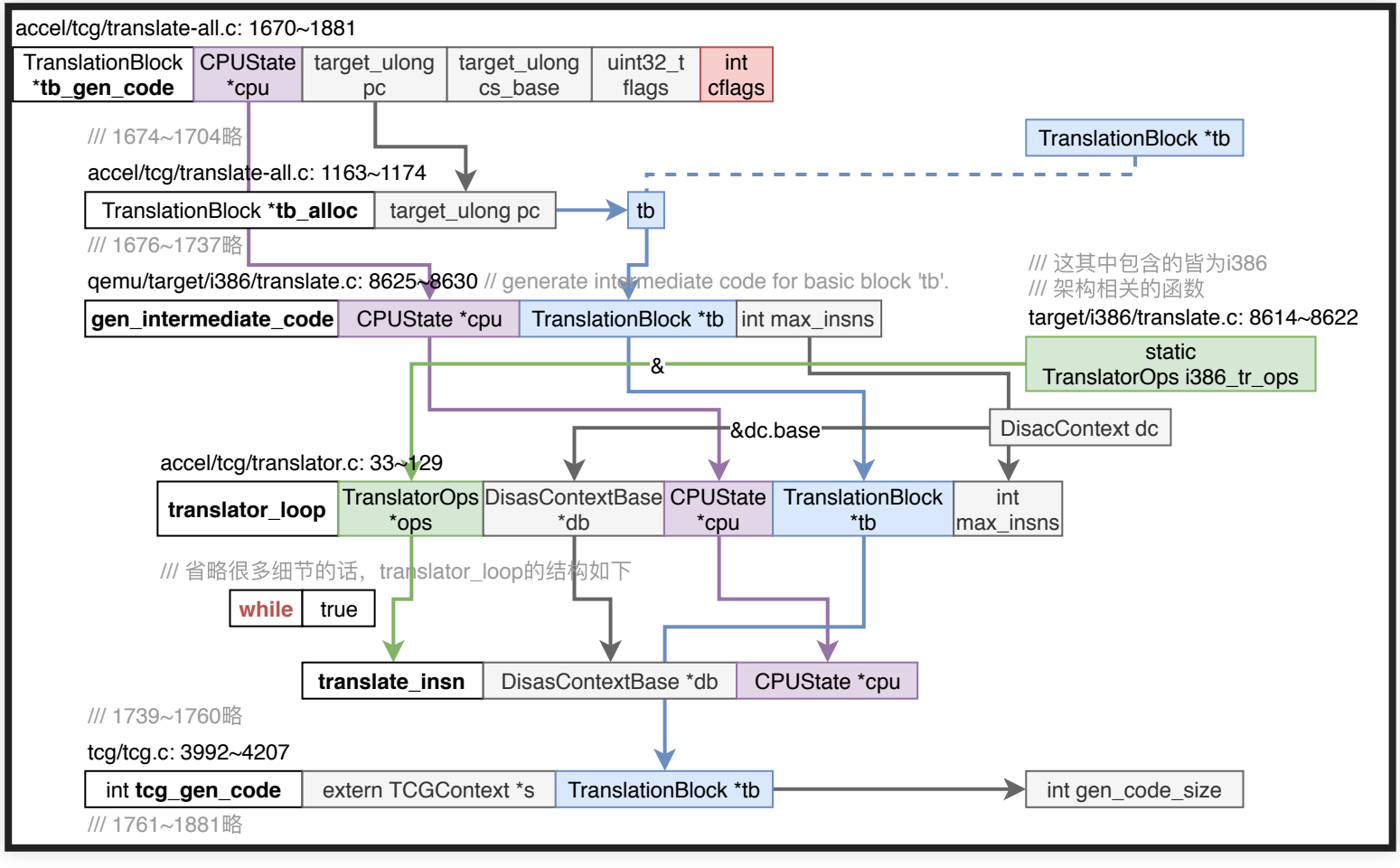

accel/tcg/cpu-exec.c: 398~428
TranslationBlock *tb_find
CPUState *cpu
TranslationBlock *last_tb
int tb_exit
uin32_t cf_mask
uint32_t flags
target_ulong cs_base
target_ulong pc
// Might cause an exception, so have a longjmp destination ready
include/exec/tb-lookup.h: 21~49
TranslationBlock *tb_lookup__cpu_state
CPUState *cpu
target_ulong *pc
target_ulong *cs_base
uint32_t *flags
uint32_t cf_mask
TranslationBlock *tb
if
tb
== NULL
include/exec/exec-all.h: 488 /// 还未实现
mmap_lock
accel/tcg/translate-all.c: 1670~1881 // Called with mmap_lock held for user mode emulation.
TranslationBlock *tb_gen_code
CPUState *cpu
target_ulong pc
target_ulong cs_base
uint32_t flags
int cf_mask
tb
include/exec/exec-all.h: 489 /// 还未实现
mmap_unlock
include/exec/tb-hash.h: 53~56
unsigned int tb_jmp_cache_hash_func
target_ulong pc
/// CPUState的成员变量tb_jmp_cache（数组）记录的是从pc到tb的映射，
/// 为了减少tb_jmp_cache的大小，实际采用的是pc哈希值到tb的映射
tb_jmp_cache
[
数组的index
]
&
include/qemu/atomic.h: 140~143 // We add the TB in the virtual pc hash table for the fast lookup
atomic_set
存tb（指针）的地方
tb
// We don't take care of direct jumps when address mapping changes in
// system emulation. So it's not safe to make a direct jump to a TB
// spanning two pages because the mapping for the second page can change.
if
tb->page_addr[1] != -1
/// 如果tb跨页了，则不会把tb连在last_tb后，last应该是上一个的意思。
last_tb = NULL
// See if we can patch the calling TB.
if
last_tb
accel/tcg/cpu-exec.c: 359~396 /// 关于TB的jmp_list详细看TB里的注释include/exec/exec-all.h: 389~402
tb_add_jump
TranslationBlock *tb
int n
TranslationBlock *tb_next
return
tb

tb_find框架图

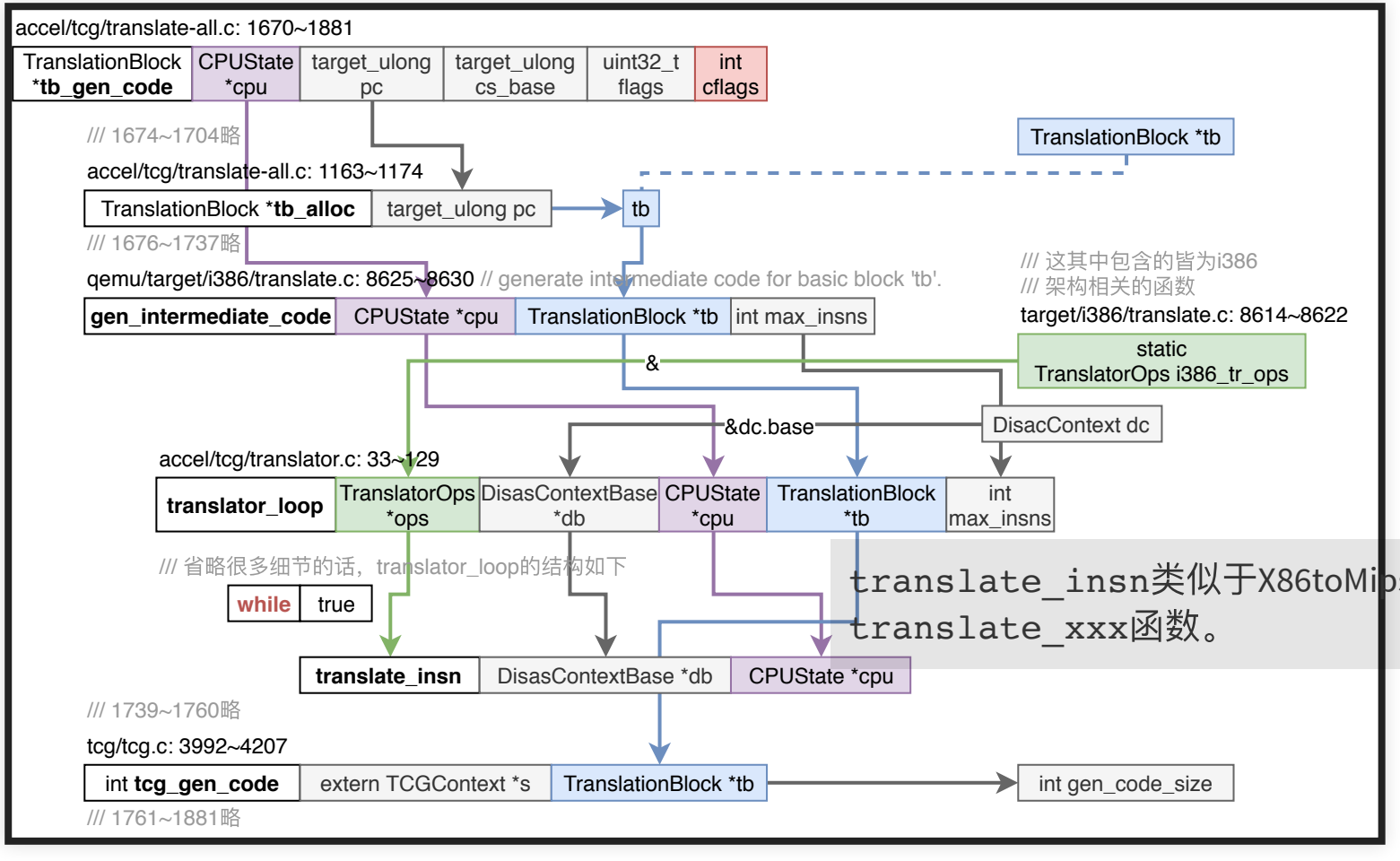


和X86toMips相同，也是发现没有TB时再去生成新的TB。

tb_gen_code框架图

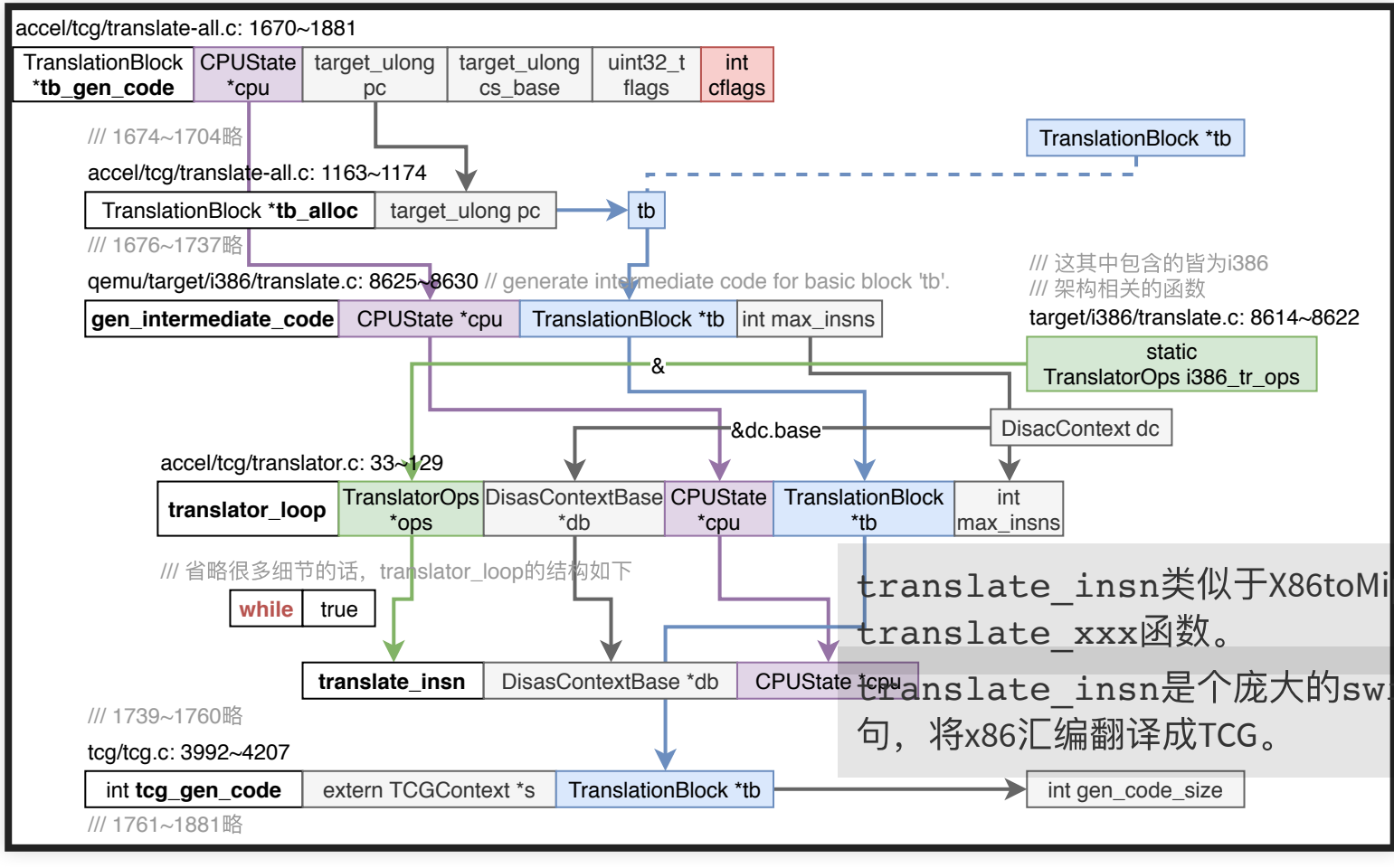


tb_gen_code框架图



translate_insn类似于X86toMips里的translate_xxx函数。

tb_gen_code框架图



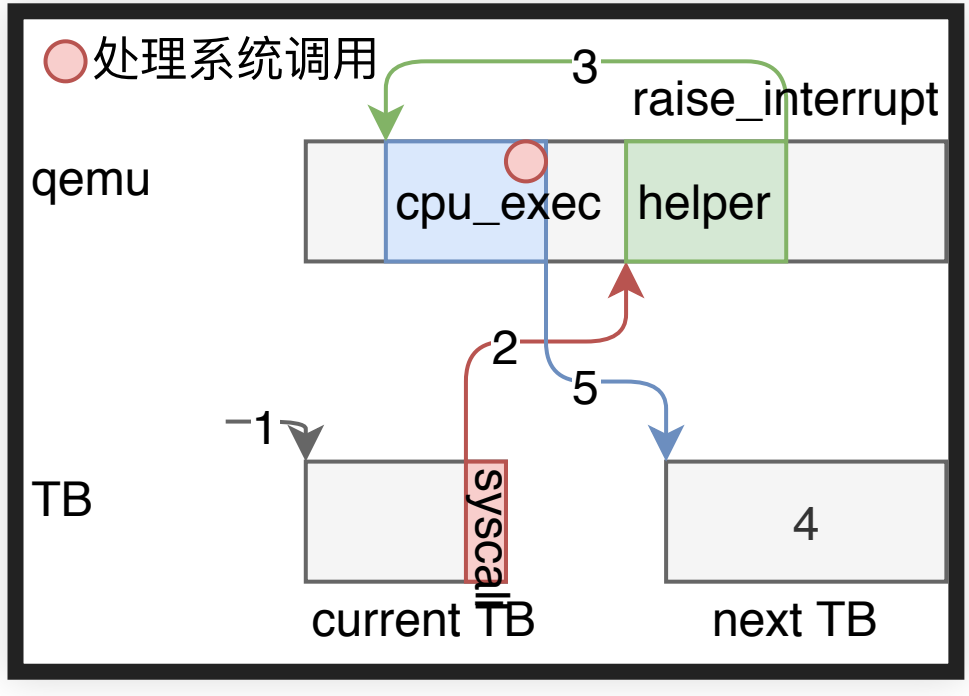
translate_insn类似于X86toMips里的translate_xxx函数。

translate_insn是个庞大的switch语句，将x86汇编翻译成TCG。

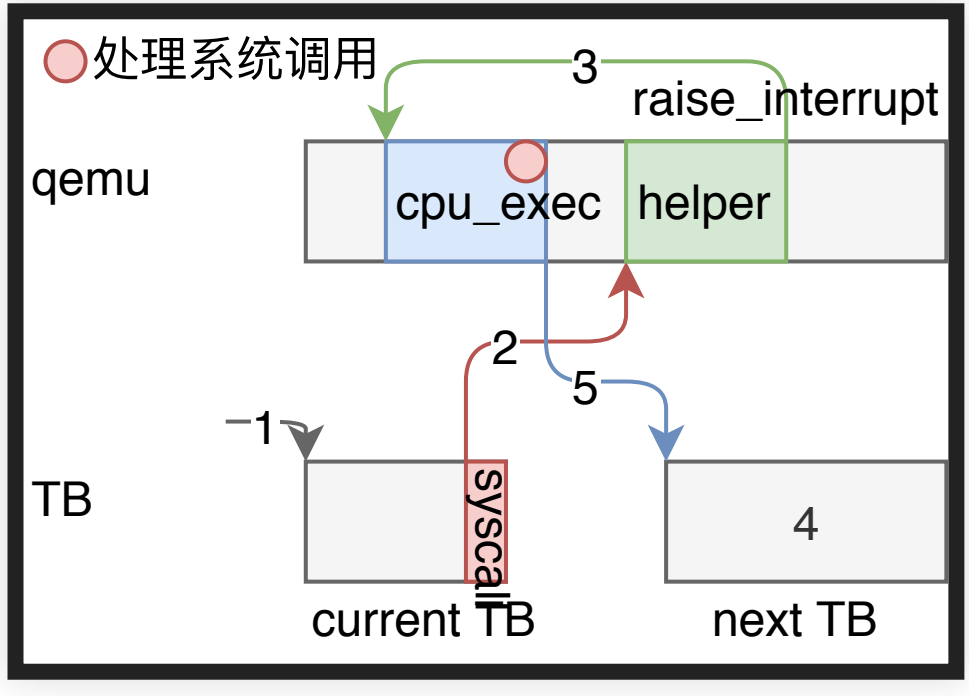
举个translate_insn里case的例子

```
// disas_insn (target/i386/translate.c: 4486~8382)
...
switch(b) // b为x86机器指令的opcode
...
// 7055行开始如下，为处理x86的中断指令
case 0xcd: /* int N */
    val = x86_ldub_code(env, s);
    if (s->vm86 && s->iopl != 3) {
        gen_exception(s, EXCP0D_GPF, pc_start - s->cs_base);
    } else {
        gen_interrupt(s, val, pc_start - s->cs_base, s->pc - s
    }
break;
...
```

QEMU翻译过程总结成如下的控制流图（首次执行）



QEMU翻译过程总结成如下的控制流图（首次执行）



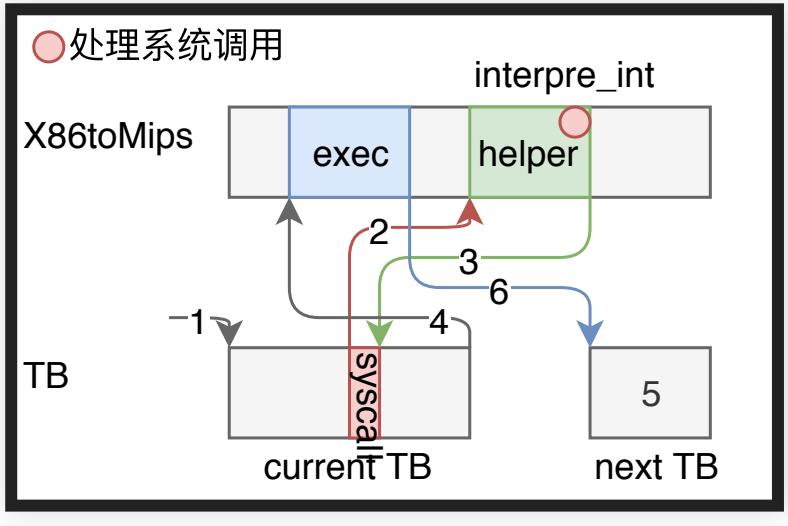
1. 开始执行当前TB
2. 转到系统调用的helper函数
3. helper函数通过siglongjmp跳到cpu_exec处理系统调用，然后寻找下一个TB
4. 发现没有TB了，生成下一个TB（进行反汇编且进行翻译）
5. 开始执行下一个TB

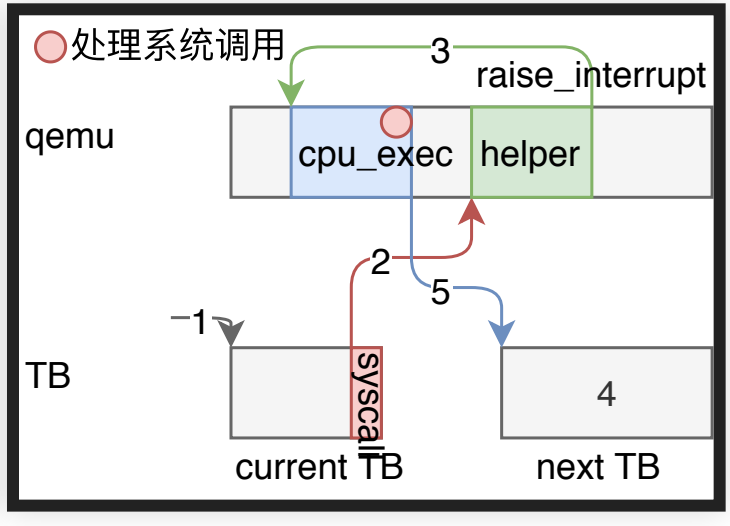
总结

比较X86toMips和QEMU对系统调用的区别

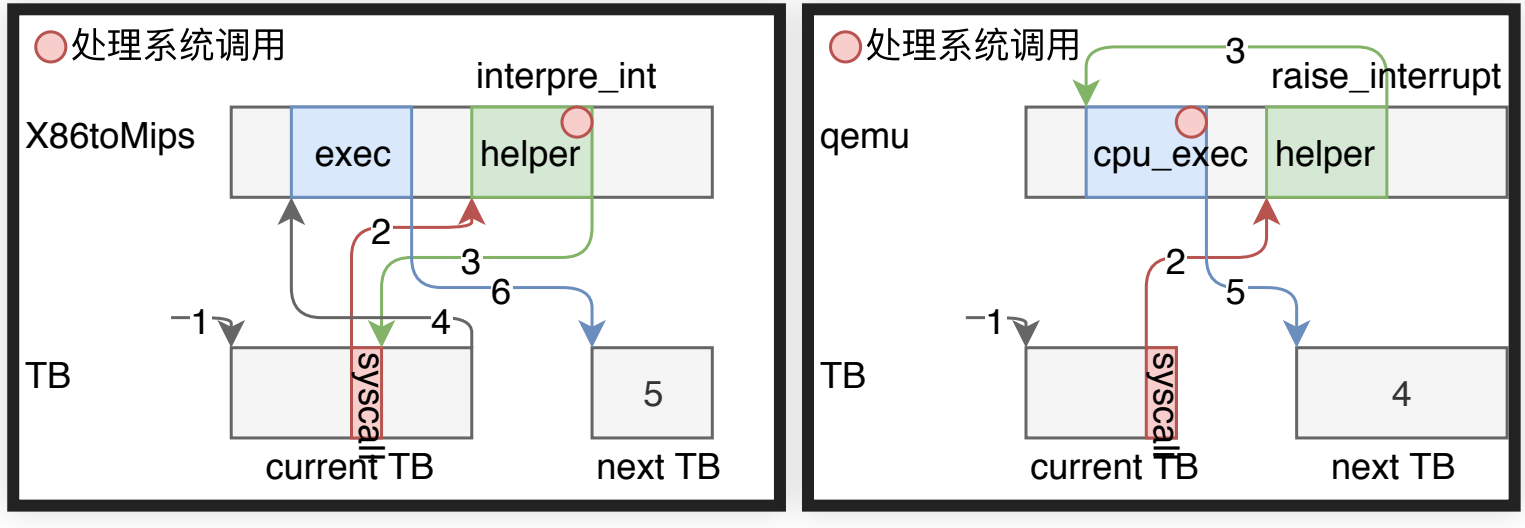
1. 首次执行
2. 非首次执行

X86TOMIPS和QEMU对系统调用的区别（首次执行）



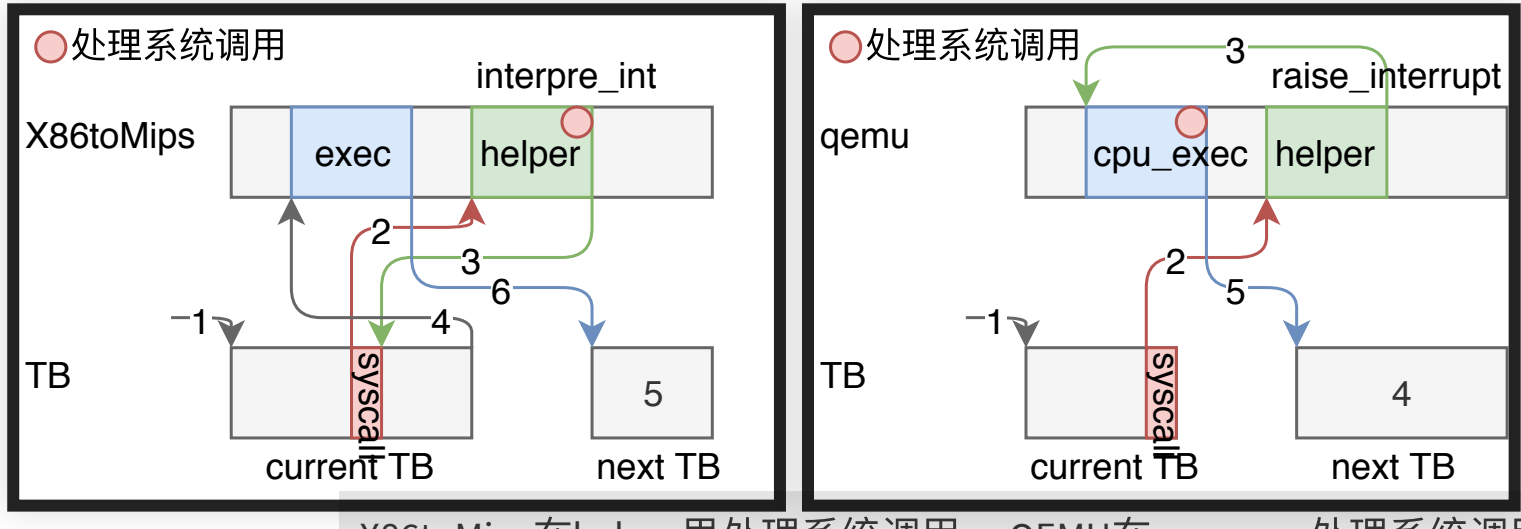


X86TOMIPS和QEMU对系统调用的区别（首次执行）



1. 步骤3: X86toMips回TB，QEMU回cpu_exec

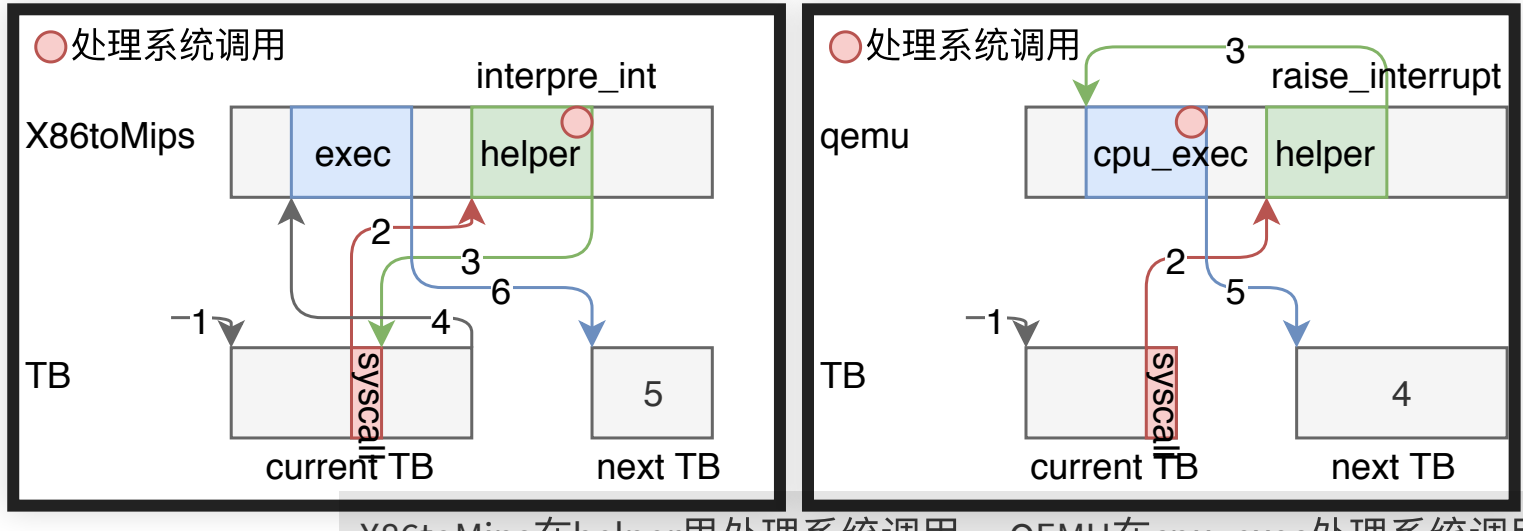
X86TOMIPS和QEMU对系统调用的区别（首次执行）



X86toMips在helper里处理系统调用。 QEMU在cpu_exec处理系统调用。

1. 步骤3: X86toMips回TB，QEMU回cpu_exec

X86TOMIPS和QEMU对系统调用的区别（首次执行）



X86toMips在helper里处理系统调用。 QEMU在cpu_exec处理系统调用。

1. 步骤3: X86toMips回TB，QEMU回cpu_exec
2. syscall是否是TB结束的判定标志

X86TOMIPS和QEMU对系统调用的区别（非首次执行）

TB已经都翻译好了。

X86TOMIPS和QEMU对系统调用的区别（非首次执行）

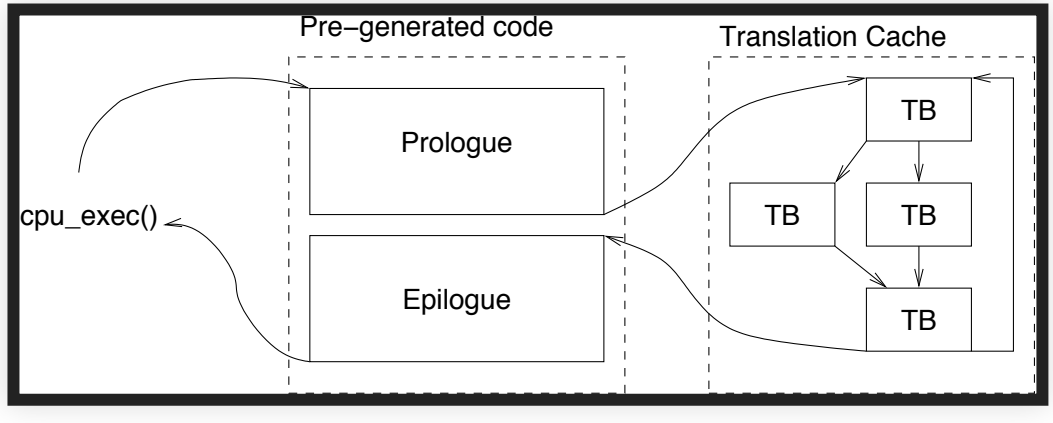
TB已经都翻译好了。

- X86toMips: 每次都只执行一个TB
- QEMU: 每次执行多个TB（直到遇到异常情况）

X86TOMIPS和QEMU对系统调用的区别（非首次执行）

TB已经都翻译好了。

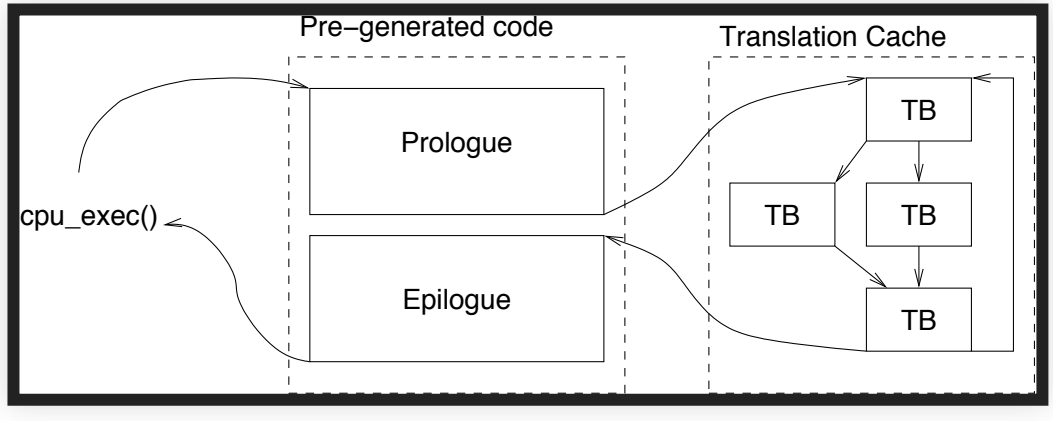
- X86toMips: 每次都只执行一个TB
- QEMU: 每次执行多个TB（直到遇到异常情况）



X86TOMIPS和QEMU对系统调用的区别（非首次执行）

TB已经都翻译好了。

- X86toMips: 每次都只执行一个TB
- QEMU: 每次执行多个TB（直到遇到异常情况）



切换上下文的开销！所以在这个方面QEMU更好。

在系统调用是否有必要在cpu_exec里执行?

在系统调用是否有必要在cpu_exec里执行?

在系统调用是否有必要在cpu_exec里执行?

自修改代码，修改之后的TB块对应的内存代码。

相比X86toMips处理系统调用的方法， QEMU更好。

相比X86toMips处理系统调用的方法,QEMU更好。

QEMU系统调用框架

谢谢

2019.11.28