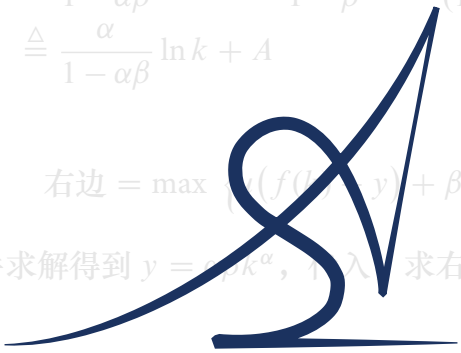
QEMU Source Code Note

QEMU 源代码 笔记



Victory won't come to us unless we go to it.

作者：张琼

编写时间：January 25, 2018

Email: zhagnqiongxingzhe@gmail.com

Version: 1.00

目　　录

第 1 章
QEMU 概述

QEMU 是一个通用的机器模拟器和虚拟机。作为机器模拟器，利用动态二进制翻译，性能很好；作为虚拟机，配合 kvm 或者 xen 使用。

1.1 原理

QEMU 的核心是动态翻译。

1.2 目录结构

QEMU 的代码目录及功能如下所示：

- accel 模拟器加速模块，包括 KVM 模块和 TCG 动态翻译引擎
- audio 声卡仿真
- backends 一些服务后端的代码
- block 块设备仿真
- capstone 第三方反汇编工具
- chardev 字符设备仿真
- crypto 加密模块
- disas 反编译模块
- dtc 设备树编译器工具链 & 与硬件描述文件有关
- fpu 浮点运算仿真器
- fsdev 基于 Plan9 协议的文件系统，沟通 Guest 和 Host 的文件读写
- hw 各种硬件仿真
- io IO 通道
- libdecnumber 十进制计算模块
- migration 虚拟机实时迁移
- nbd 网络块设备仿真

- net 网络
- pc-bios bios 源代码
- qapi QEMU API ，monitor 和虚拟机交互的 json 协议
- qga QEMU Guest Agent 在虚拟机内部作为代理，提供信息
- qobject & qom QEMU 设备模型
- replay record&replay，供逆向分析和 debug 之用
- roms 二进制 BIOS
- target 将 target 体系结构的二进制代码翻译为 TCG 中间表示的模块
- tcg TCG 中间表示的关键代码，并负责中间表示翻译为 host 代码
- trace trace 调试
- ui 虚拟机图形化模块
- util 各种基本函数
- vl.c 主函数入口，参数解析

1.3 多线程架构

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。我们知道，处理事件的主循环只在一个线程上执行，它并不能够发挥多核 CPU 的优势，所以混合架构有其必然的合理性。另外，相比于把所有工作的分发处理都集成到事件驱动架构中，有时候用专用的线程去执行专门的任务会更简单。当然，QEMU 的大多数代码运行还是受事件驱动的，QEMU 核心程序基于事件驱动的模型。事实上，在 QEMU 中除了在配合事件循环机制的多线程之外，还存在多个事件循环，如 VNC server 和 QMP 会在同一个事件循环中处理。默认的事件循环是 main-loop.c 中的主循环，还可以通过-object iothread,id=my-thread 创建额外的事件循环。主循环和 IOThread 都是事件循环，但是他们并不是完全一样的，上文分析过，IOThread 使用 glib 实现的事件循环，而主循环则是自己实现的。IOThread 的作用：IOThread 允许用户控制任务的放置位置。主循环和 QEMU 全局锁紧密相关，因此在多 CPU 的主机上是扩展性的瓶颈。任务可以在多个 IOThread 之间传播，而不是一个主循环。可以提高 IO 延迟，并降低客户机的抖动。vCPU 线程和主循环使用全局锁串行执行 QEMUdiamante。这个锁很必要，因为很多历史代码都是非线程安全的。

事实上所有的 IO 处理都在主循环中完成，并且 QEMU 全局锁被所有的 vCPU 线程和主循环竞争，这解释了为什么把任务放到 IOThread 中是可取的。很多子系统都可以跑在 AioContext 的上下文，这也是现在的趋势。对于 AioContext 如何提高性能还需要具体的理解 todo。

我们先来探索 QEMU 中的线程。QEMU 的主要线程包括，

1. 主循环线程：主线程执行 main_loop，处理 host 设备的 IO 事件以及各种 QEMU 事件。
2. vCPU 线程：动态翻译核心线程
3. io task 有关的线程：在 qio channel 相关代码中，用于处理 dns 解析、socket 等。io channel 类似 Gsource，可作为事件源
4. iothread 有关的线程：处理任何放入 IOThread 的任务。
5. thread-pool 有关的线程：用于处理主机设备（host device）的块设备 driver 发出的 aio 请求。
6. rcu 有关的线程：处理 rcu （todo）
7. 内存 touch page 的线程：预分配内存，遍历所有的内存，保证之后不会发生缺页。
8. 信号处理线程：
9. 实时迁移有关的线程：
10. 其他模拟硬件创建的线程：

QEMU 实现用户态的 rcu，用于提供显式的内存屏障。

第 2 章

数据结构

2.1 基本数据结构

QEMU 定义的基础数据结构，如 queue 包括单向链表、双向链表、单向队列、双向队列。链表和队列的区别就是队列头部有一对指针，指向第一个和最后一个元素，而链表只有一个指向第一个元素的指针。

```
#define QSLIST_HEAD(name, type)
struct name {
        struct type *slh_first; /* first element */
}

#define QSLIST_ENTRY(type)
struct {
        struct type *sle_next;  /* next element */
}

#define QLIST_HEAD(name, type)
 struct name {
         struct type *lh_first;  /* first element */
 }
#deine QLIST_ENTRY(type)
struct {
        struct type *le_next;   /* next element */
        struct type **le_prev;  /* address of previous next element */
}

#define QSIMPLEQ_HEAD(name, type)
struct name {
    struct type *sqh_first;    /* first element */
    struct type **sqh_last;    /* addr of last next element */
}
#define QSIMPLEQ_ENTRY(type)
struct {
    struct type *sqe_next;    /* next element */
```

```
}
#define Q_TAILQ_HEAD(name, type, qual)
struct name {
        qual type *tqh_first;           /* first element */
        qual type *qual *tqh_last;      /* addr of last next element */
}

#define Q_TAILQ_ENTRY(type, qual)
struct {
        qual type *tqe_next;            /* next element */
        qual type *qual *tqe_prev;      /* address of previous next
           element */
}
```

2.2 选项

选项参数解析设计：

```
static QemuOptsList *vm_config_groups[48];
static QemuOptsList *drive_config_groups[5];

struct QemuOptsList {
    const char *name;
    const char *implied_opt_name;
    bool merge_lists;  /* Merge multiple uses of option into a single
       list? */
    QTAILQ_HEAD(, QemuOpts) head;
    QemuOptDesc desc[];
};

typedef struct QemuOptDesc {
    const char *name;
    enum QemuOptType type;
    const char *help;
    const char *def_value_str;
} QemuOptDesc;

struct QemuOpts {
    char *id;
    QemuOptsList *list;
    Location loc;
    QTAILQ_HEAD(QemuOptHead, QemuOpt) head;
```

```
    QTAILQ_ENTRY(QemuOpts) next;
};
struct QemuOpt {
    char *name;
    char *str;

    const QemuOptDesc *desc;
    union {
        bool boolean;
        uint64_t uint;
    } value;

    QemuOpts     *opts;
    QTAILQ_ENTRY(QemuOpt) next;
};
```

QEMU 一共设计了 48 个选项 group，5 个磁盘块设备相关的选项 group。每个 group 的关键信息是 name 、QemuOpts 链表头、以及 QemuOptDesc 描述数组。QemuOpts 链表节点的成员是 QemuOpt 链表。因此，就有两级链表：QemuOpt 链表和 QemuOpts 链表。这些数据结构都是为选项解析而设计的。每个 group 代表一个大的选项，如 -smp，每个 QemuOpt 代表大选项中的小选项，如-smp cpus=2.j

```
static const QEMUOption *lookup_opt(int argc, char **argv, const char **
   poptarg, int *poptind);
typedef struct QEMUOption {
    const char *name;
    int flags;
    int index;
    uint32_t arch_mask;
} QEMUOption;
```

此函数对选项合法性检查，并解析具体的参数，选项类型是返回值 QEMUOption，参数存储在 poptarg 中。和选项参数解析还有 qemu-options.hx，定义了所有的选项。

2.3 对象管理

QEMU 的对象管理机制实现在 qom 和 qobject 两个文件夹中。QEMU 为了管理设备，设计了一套面向对象的机制，将所有的设备、总线、机器等都抽象为对象。(采

用的应该是 Java 的 OO 思想，单继承，有接口类。）首先介绍第一个概念是模块(module)，

```
typedef struct ModuleEntry
{
    void (*init)(void);
    QTAILQ_ENTRY(ModuleEntry) node;
    module_init_type type;
} ModuleEntry;
```

在系统中存在 4 种 module，

```
 typedef enum {
       MODULE_INIT_BLOCK,
       MODULE_INIT_OPTS,
       MODULE_INIT_QOM,
       MODULE_INIT_TRACE,
       MODULE_INIT_MAX
   } module_init_type;
```

块设备、OPTS（选项有关，代码中涉及到的不多，应该还没有广泛应用）、QOM（QEMU 设备模型）、Trace 有关（调试跟踪），针对这四种 module 定义了四种初始化函数，本质上都是 module_init，

```
  #define block_init(function) module_init(function, MODULE_INIT_BLOCK)
  #define opts_init(function) module_init(function, MODULE_INIT_OPTS)
  #define type_init(function) module_init(function, MODULE_INIT_QOM)
  #define trace_init(function) module_init(function, MODULE_INIT_TRACE)
```

这四种模块分别对应了四个模块链表，还有一个特别的链表 dso_init_list（应该跟动态加载有关）。

```
  static ModuleTypeList init_type_list[MODULE_INIT_MAX];
```

module_init 初始化时将初始化函数赋给 ModuleEntry 成员 init，在 main 函数中进行初始化时会调用四个链表上所有的模块的 init 成员函数，而这个 init 函数又通常用于初始化（模块代表的类的）重要的成员函数。也就是说 module 和 class 有着类似的含义（还需要再进一步理解），或者说他们指代的对象是一致的。

与类有关的数据结构就是 Object、ObjectClass、xxxClass、TypeImpl、TypeInfo。下面以 x86_cpu_type_info 为例，理解这些结构的关系：

```
type_init(x86_cpu_register_types)
```

此函数注册了一个 x86_cpu 的模块，其中 x86_cpu_register_types 用于注册 TypeImpl 类。

```
static const TypeInfo x86_cpu_type_info = {
    .name = TYPE_X86_CPU,
    .parent = TYPE_CPU,
    .instance_size = sizeof(X86CPU),
    .instance_init = x86_cpu_initfn,
    .abstract = true,
    .class_size = sizeof(X86CPUClass),
    .class_init = x86_cpu_common_class_init,//此函数基本用于初始化 x86cpu 的各个特性的处理函数。
};
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_finalize)(ObjectClass *klass, void *data);
    void *class_data;

    InterfaceInfo *interfaces;
};
```

可以看到 TypeInfo 中有 instance_init 方法和 class_init 方法，分别是实例和类的初始化函数，参数也分别是 Object 和 ObjectClass。

```
struct TypeImpl
{
    const char *name;
```

```
4
5       size_t class_size;
6
7       size_t instance_size;
8
9       void (*class_init)(ObjectClass *klass, void *data);
10      void (*class_base_init)(ObjectClass *klass, void *data);
11      void (*class_finalize)(ObjectClass *klass, void *data);
12
13      void *class_data;
14
15      void (*instance_init)(Object *obj);
16      void (*instance_post_init)(Object *obj);
17      void (*instance_finalize)(Object *obj);
18
19      bool abstract;
20
21      const char *parent;
22      TypeImpl *parent_type;
23
24      ObjectClass *class;
25
26      int num_interfaces;
27      InterfaceImpl interfaces[MAX_INTERFACES];
28  };
29
30  static TypeImpl *type_register_internal(const TypeInfo *info)
31  {
32      TypeImpl *ti;
33      ti = type_new(info);
34
35      type_table_add(ti);
36      return ti;
37  }
```

对比 TypeImpl 和 TypeInfo 可以发现两者很相似，而上述函数根据 TypeInfo 的信息，生成 TypeImpl 类，并将该类加入到 hash table 中。可以看到，TypeImpl 有 ObjectClass 成员，跟类有关。

static void type_initialize(TypeImpl *ti) 函数以 TypeImpl * 类型为参数，初始化 ti 的 class 成员以及和类信息有关的成员，接口等。在 type_initialize 中会先对 ti 的父类初始化，最后调用本类 ti->class_init，类似 C++ 的构造函数。至此，我们看到了类的生成。在生成过程中，我们发现 Type

和 Class 的关系非常紧密（接口相关没有分析，后续还要加深理解 todo）。

那么对象又是怎么生成的呢?

```
struct Object
{
    /*< private >*/
    ObjectClass *class;
    ObjectFree *free;
    GHashTable *properties;
    uint32_t ref;
    Object *parent;
};

Object *object_new(const char *typename)
{
    TypeImpl *ti = type_get_by_name(typename);

    return object_new_with_type(ti);
}
static void object_initialize_with_type(void *data, size_t size, TypeImpl
    *type)
{
    Object *obj = data;

    g_assert(type != NULL);
    type_initialize(type);

    g_assert_cmpint(type->instance_size, >=, sizeof(Object));
    g_assert(type->abstract == false);
    g_assert_cmpint(size, >=, type->instance_size);

    memset(obj, 0, type->instance_size);
    obj->class = type->class;
    object_ref(obj);
    obj->properties = g_hash_table_new_full(g_str_hash, g_str_equal,
                                            NULL, object_property_free);
    object_init_with_type(obj, type);
    object_post_init_with_type(obj, type);
}

static void object_init_with_type(Object *obj, TypeImpl *ti)
{
    if (type_has_parent(ti)) {
        object_init_with_type(obj, type_get_parent(ti));
    }
```

```
42
43      if (ti->instance_init) {
44          ti->instance_init(obj);
45      }
46  }
```

在初始化对象的时候，先从 hash table 中找到 TypeImpl 类型数据，然后传递给 object_new_with_type 函数，返回一个对象。对象初始化主要是对它所指向的类、引用数、properties（成员）赋值，并调用对象实例方法。从关系上讲，ObjectClass 是所有类的父类，xxxClass 是 ObjectClass 的实例，也就是具体的类，Object 是 xxxClass 类的实例，即对象。在源代码中检索会发现，很多概念都使用对象管理，比如各种模拟外设、总线、cpu、machine 抽象、IOthread 等。（外设管理一般采用总线-设备抽象）

2.4 事件循环

QEMU 还根据需要（块设备 IO）定制了自己的事件循环机制 Aio。事件循环的具体实现在 util/aio-posix.c（util/aio-win32.c）和 util/async.c 两个文件中。后者提供共同抽象，前者提供跟系统耦合的部分，以及接口相关的代码。关键数据结构是 struct AioContext：

```
1   struct AioContext {
2       GSource source;
3
4       /*多线程访问 */
5       QemuRecMutex lock;
6
7       /* 所有注册的 AIO handler， list_lock 可保证多线程访问*/
8       QLIST_HEAD(, AioHandler) aio_handlers;
9
10      /* Used to avoid unnecessary event_notifier_set calls in aio_notify;
11       * accessed with atomic primitives.  If this field is 0, everything
12       * (file descriptors, bottom halves, timers) will be re-evaluated
13       * before the next blocking poll(), thus the event_notifier_set call
14       * can be skipped.  If it is non-zero, you may need to wake up a
15       * concurrent aio_poll or the glib main event loop, making
16       * event_notifier_set necessary.
17       *
18       * Bit 0 is reserved for GSource usage of the AioContext, and is 1
19       * between a call to aio_ctx_prepare and the next call to
             aio_ctx_check.
```

```
20      * Bits 1-31 simply count the number of active calls to aio_poll
21      * that are in the prepare or poll phase.
22      *
23      * The GSource and aio_poll must use a different mechanism because
24      * there is no certainty that a call to GSource's prepare callback
25      * (via g_main_context_prepare) is indeed followed by check and
26      * dispatch.  It's not clear whether this would be a bug, but let's
27      * play safe and allow it---it will just cause extra calls to
28      * event_notifier_set until the next call to dispatch.
29      *
30      * Instead, the aio_poll calls include both the prepare and the
31      * dispatch phase, hence a simple counter is enough for them.
32      */
33     uint32_t notify_me;
34
35     /* A lock to protect between QEMUBH and AioHandler adders and deleter
           ,
36      * and to ensure that no callbacks are removed while we're walking
            and
37      * dispatching them.
38      QEMUBH 和 AioHandler 的添加和删除时要使用 list_lock
39      */
40     QemuLockCnt list_lock;
41
42     /* Anchor of the list of Bottom Halves belonging to the context */
43     struct QEMUBH *first_bh;
44
45     /* Used by aio_notify.
46      *
47      * "notified" is used to avoid expensive
            event_notifier_test_and_clear
48      * calls.  When it is clear, the EventNotifier is clear, or one
            thread
49      * is going to clear "notified" before processing more events.  False
50      * positives are possible, i.e. "notified" could be set even though
            the
51      * EventNotifier is clear.
52      *
53      * Note that event_notifier_set *cannot* be optimized the same way.
            For
54      * more information on the problem that would result, see "#ifdef
            BUG2"
55      * in the docs/aio_notify_accept.promela formal model.
56      */
57     bool notified;
```

```
58      EventNotifier notifier;
59
60      QSLIST_HEAD(, Coroutine) scheduled_coroutines;
61      QEMUBH *co_schedule_bh;//和 coroutine，待研究。
62
63      /* Thread pool for performing work and receiving completion callbacks
            .
64       * Has its own locking.
65       */
66      struct ThreadPool *thread_pool;//线程池用于执行任务（work）和接受完成
            回调函数（实现异步）。
67
68  #ifdef CONFIG_LINUX_AIO
69      /* State for native Linux AIO.  Uses aio_context_acquire/release for
70       * locking.
71       */
72      struct LinuxAioState *linux_aio;
73  #endif
74
75      /* TimerLists for calling timers - one per clock type.  Has its own
76       * locking.自带锁
77       */
78      QEMUTimerListGroup tlg;
79
80      int external_disable_cnt;
81
82      /* 禁止 poll 的 AIOHandler 的个数 */
83      int poll_disable_cnt;
84
85      /* Polling mode parameters */
86      int64_t poll_ns;        /* current polling time in nanoseconds */
87      int64_t poll_max_ns;    /* maximum polling time in nanoseconds */
88      int64_t poll_grow;      /* polling time growth factor */
89      int64_t poll_shrink;    /* polling time shrink factor */
90
91      /*处于poll 模式还是文件描述符监控模式*/
92      bool poll_started;
93
94      int epollfd;
95      bool epoll_enabled;
96      bool epoll_available;
97  };
```

QEMU 利用 glibc 提供的 main event loop 机制，实现异步操作。在 glib 的 main event loop 机制中，AioContext 类型的第一个元素就是 GSource，符合 glib 的规定,也就是说 AioContext 是 QEMU 自定义的事件源。AioContext 拓展了 glib 中 source 的功能，不但支持 fd、超时的轮询，还模拟内核中的下半部机制实现了事件的异步通知功能，其中的通知功能是基于 eventfd 实现的。AioContext 还自定义了四个函数：

```
static GSourceFuncs aio_source_funcs = {

aio_ctx_prepare,

aio_ctx_check,

aio_ctx_dispatch,

aio_ctx_finalize

};
```

这几个函数分别在 g_main_context_prepare()，g_main_context_check() 和 g_main_context_dispatch() 中被调用。下面分别介绍一下这几个函数的主要功能：

1. aio_ctx_prepare 会调用 aio_compute_timeout 来计算需要的超时时间，这个超时时间是在轮询过程中使用的，它是由 AioContext 中注册的 bh（下半部）的属性决定的，当 AioContext 中注册的所有的 bh 都是空闲的时，则返回一个有效的超时时间；当至少有一个 bh（下半部）不是空闲的时，则返回 0，从而保证 bh 会被尽快执行。

2. aio_ctx_check 用来检查如果 bh、aiohandler(fd，包括信号、事件通知等）或 timer 存在就绪,则返回 TRUE,从而调用 g_main_context_dispatch()

3. aio_ctx_dispatch 调用 aio_dispatch,依次执行就绪的 bh、aiohandler(fd，包括信号、事件通知等）和 timer，完成依次主循环。

QEMU 会在初始化的过程中，通过 g_source_new 函数（glib 的库函数）把 aio_source_funcs 注册到 AioContext。在 QEMU 主循环中有 4 个实例，qemu_aio_context，iohandler->ctx，iothread 中的 AioContext，描述磁盘镜像的 BlockDriverState 中的 AioContext。他们负责处理的事件分别是：

qemu_aio_context：是默认的 AioContext，可以用于 VNC,QMP 命令，后半部 bh，定时器等；iohandler->ctx：负责监控信号，中断，事件通知，socket 等；iothread->ctx：主要负责 io 方面的监控；bs->ctx：负责 blockjob 等相关任务的监控；

但是在 QEMU 中，主循环并没有使用默认的 g_main_loop_run，而是自己实现了 loop run，前两个 AioContext 事件源是在主循环中的。与 iothread 有关的 AioContext 使用 glib 默认的 g_main_loop_run。(原因不知道 todo)

//待补充留坑，以后用到再具体分析。todo

Note: Glib 中的事件处理 Glib 中是由一个主事件循环（main event loop）来负责处理所有的事件源（source），事件源包括文件描述符（文件、管道或者 socket）和超时。新的事件源可以通过 g_source_attach() 来添加。为了实现在不同的线程中处理多个、独立的事件源，每一个事件源都关联一个主上下文 GMainContext 的数据结构。一个 GMainContex 只能在一个线程中运行，但不同线程中的事件源可以互相添加或删除。GMainContext 中的事件源会在 GMainContext 关联的主事件循环中进行检查和发送（dispatch）。新的事件源类型可以通过包含 GSource 结构体来创建。新的事件源类型中 GSource 结构体必须是第一个成员，其他成员放在其后。要创建一个新事件源类型实例，可以调用 g_source_new() 函数，传入新的事件源类型大小和一个 GSourceFuncs 类型的变量，这个变量决定了新的事件源类型的控制方式。新的事件源通过两种方式跟主上下文交互。第一种方式是 GSourceFuncs 中的 prepare 函数可以设置一个超时时间，来决定主事件循环中轮询的超时时间；第二种方式是通过 g_source_add_poll() 函数来添加文件描述符。

主上下文的一次循环包含四个步骤,分别由四个函数实现:g_main_context_prepare(), g_main_context_query(), g_main_context_check() 和 g_main_context_dispatch

下面分别简单介绍一下这四个函数的作用：

1. g_main_context_prepare()：对于没有设置 G_SOURCE_READY 标志的 source,调用 source->source_funcs->prepare 函数,如果返回 TRUE，则设置 source 的 G_SOURCE_READY；调用 prepare 函数时会通过参数返回一个超时时间，选取最小的一个超时时间赋值给 context->timeout。

2. g_main_context_query()：从 context->poll_records 中返回指定个数的 fd，返回 context->timeout，context->poll_changed 设为 FALSE.

3. g_main_context_check()：如果 context->poll_changed 为 TRUE，则返回 FALSE；否则复制传入的 fds 的 revents 到相应的 context->poll_records->fd->revents 中;遍历所有的 source,对未设置 G_SOURCE_READY

标志的 source 调用其 check 函数；将已经设置 G_SOURCE_READY 标志的 source 添加到 context->pending_dispatches 中；

4. g_main_context_dispatch()：清除 context->pending_dispatches 中 source 的 G_SOURCE_READY 标志，然后调用其 dispatch 函数；

```
AioContext *aio_context_new(Error **errp)
{
    int ret;
    AioContext *ctx;

    ctx = (AioContext *) g_source_new(&aio_source_funcs, sizeof(
        AioContext));
    //aio_source_funcs 即上述的四个函数赋予 ctx

    aio_context_setup(ctx);
    //利用 epoll 机制创建epoll 描述符，作为事件循环的核心。
    ret = event_notifier_init(&ctx->notifier, false);
    //初始化 notifier，类似于信号处理，有两种方法：一种是 eventfd，另一种
        是 pipe 管道。
    //下面会对 notifier 的事件进行设置。
    if (ret < 0) {
        error_setg_errno(errp, -ret, "Failed to initialize event notifier
            ");
        goto fail;
    }
    g_source_set_can_recurse(&ctx->source, true);
    qemu_lockcnt_init(&ctx->list_lock);

    ctx->co_schedule_bh = aio_bh_new(ctx, co_schedule_bh_cb, ctx);
    QSLIST_INIT(&ctx->scheduled_coroutines);

    aio_set_event_notifier(ctx, &ctx->notifier,
                           false,
                           (EventNotifierHandler *)
                           event_notifier_dummy_cb,
                           event_notifier_poll);
    //类似于将信号处理为 AIO 监控的对象，设置事件为 AIO 监控对象，并为其
        配置 AIOhandler。

#ifdef CONFIG_LINUX_AIO
    ctx->linux_aio = NULL;
#endif
    ctx->thread_pool = NULL;
```

```
35      qemu_rec_mutex_init(&ctx->lock);
36      timerlistgroup_init(&ctx->tlg, aio_timerlist_notify, ctx);
37      //初始化 ctx 的计时器列表，并挂载到响应的时钟上。
38      ctx->poll_ns = 0;
39      ctx->poll_max_ns = 0;
40      ctx->poll_grow = 0;
41      ctx->poll_shrink = 0;
42
43      return ctx;
44  }
45  struct EventNotifier {
46  #ifdef _WIN32
47      HANDLE event;
48  #else
49      int rfd;
50      int wfd;
51  #endif
52  };
```

此函数是 AioContext 的初始化函数，

上面提到 bh，其数据结构如下所示：

```
1  struct QEMUBH {
2    AioContext *ctx;
3    QEMUBHFunc *cb;
4    void *opaque;
5    QEMUBH *next;
6    bool scheduled;
7    bool idle;
8    bool deleted;
9  };
```

下半部在 Linux 中用于延迟处理耗时较长、而又不太紧急的任务，例如软中断的典型应用就是下半部。在 QEMU 应该类似。每个下半部和一个 AioContext 绑定，功能类似一个立即到期的定时器，用于避免调用栈的冲入和溢出。(抄的，还需要验证 todo) qemu_bh_schedule 函数用于通知 BH 到期，并利用 event 机制，通知相应的部件。下半部还可能与协程有关，待研究 todo。

```
1  QEMUBH *aio_bh_new(AioContext *ctx, QEMUBHFunc *cb, void *opaque)
2  {
3      QEMUBH *bh;
4      bh = g_new(QEMUBH, 1);
```

```
5       *bh = (QEMUBH){
6           .ctx = ctx,
7           .cb = cb,
8           .opaque = opaque,
9       };
10      qemu_lockcnt_lock(&ctx->list_lock);
11      bh->next = ctx->first_bh;
12      /* Make sure that the members are ready before putting bh into list
            */
13      smp_wmb();
14      ctx->first_bh = bh;//向 ctx 的 bh 列表中添加一个新元素，回调函数是 cb
            。
15      qemu_lockcnt_unlock(&ctx->list_lock);
16      return bh;
17  }
18
19  void qemu_bh_schedule(QEMUBH *bh)
20  {
21      AioContext *ctx;
22
23      ctx = bh->ctx;
24      bh->idle = 0;
25      /* The memory barrier implicit in atomic_xchg makes sure that:
26       * 1. idle & any writes needed by the callback are done before the
27       *    locations are read in the aio_bh_poll.
28       * 2. ctx is loaded before scheduled is set and the callback has a
             chance
29       *    to execute.
30       */
31      if (atomic_xchg(&bh->scheduled, 1) == 0) {
32          aio_notify(ctx); // 先把 shceduled 置1，再调用 aio_notify=>
                event_notifier_set，后者会向 notifier 的文件描述符写入1，通知
                主循环.
33      }
34  }
```

2.5 线程池

qemu 模仿 glib 实现了线程池的功能，目前 qemu 中线程池主要应用在 raw 文件的支持上，当 linux-aio 不可用时，就像 glibc，通过线程实现 aio 机制。我们也可看到，代表线程池中的线程成员的数据结构 ThreadPoolElement 就包

含了用来描述 aio 的 BlockAIOCB 结构。相关数据结构如下：

```
    typedef struct AIOCBInfo {

    void (*cancel_async)(BlockAIOCB *acb);

    AioContext *(*get_aio_context)(BlockAIOCB *acb);

    size_t aiocb_size;

} AIOCBInfo;

struct BlockAIOCB {

    const AIOCBInfo *aiocb_info;

    BlockDriverState *bs;

    BlockCompletionFunc *cb;

    void *opaque;

    int refcnt;

};

struct ThreadPoolElement {

    BlockAIOCB common;

    ThreadPool *pool;

    ThreadPoolFunc *func;

    void *arg;

    /* Moving state out of THREAD_QUEUED is protected by lock.  After

     * that, only the worker thread can write to it.  Reads and writes

     * of state and ret are ordered with memory barriers.

     */

    enum ThreadState state;
```

```
44
45      int ret;
46
47      /* Access to this list is protected by lock.  */
48
49      QTAILQ_ENTRY(ThreadPoolElement) reqs;
50
51      /* Access to this list is protected by the global mutex.  */
52
53      QLIST_ENTRY(ThreadPoolElement) all;
54
55  };
56
57  struct ThreadPool {
58
59      AioContext *ctx;
60
61      QEMUBH *completion_bh;
62
63      QemuMutex lock;
64
65      QemuCond worker_stopped;
66
67      QemuSemaphore sem;
68
69      int max_threads;
70
71      QEMUBH *new_thread_bh;
72
73      /* The following variables are only accessed from one AioContext. */
74
75      QLIST_HEAD(, ThreadPoolElement) head;
76
77      /* The following variables are protected by lock.  */
78
79      QTAILQ_HEAD(, ThreadPoolElement) request_list;
80
81      int cur_threads;
82
83      int idle_threads;
84
85      int new_threads;     /* backlog of threads we need to create */
86
87      int pending_threads; /* threads created but not running yet */
88
```

```
89      bool stopping;
90
91  };
```

ThreadPool 中有 5 个负责维护不同状态下的线程成员的计数器：

1. max_threads 负责统计线程池中允许创建的线程的最大值；
2. new_threads 负责统计需要创建的线程数；
3. pending_threads 负责统计已创建但还没有运行的线程数；
4. idle_threads 负责统计空闲的线程数；
5. cur_threads 负责统计当前线程池中线程的个数；注意 cur_threads 包含 new_threads 中尚未创建的线程。

线程池的生命周期：

1. 线程池创建首先通过 thread_pool_new 函数为特定的 AioContext 实例创建一个新的线程池。在这个函数中初始化 ThreadPool 数据结构的各个成员，包括负责创建新线程的 new_thread_bh 和线程执行完毕后用来调度执行任务完成回调函数的 completion_bh。
2. 线程生成和任务提交 ThreadPool 数据结构负责维护线程池里面的线程成员,线程的创建是通过下半部来实现的。当块设备执行 thread_pool_submit_aio 提交任务时，对线程池的下半部 new_thread_bh 进行调度，此时创建具体的线程。
3. 线程执行创建的线程执行 worker_thread 函数,这个函数从 pool->request_list 链表中取下第一个 ThreadPoolElement 节点，执行其任务函数。
4. 线程结束线程执行完一个任务后，也就是一个 ThreadPoolElement 实例被执行后，调度执行完成下半部的回调函数 completion_bh，这个 bh 回调函数会遍历 pool->head 链表，根据其 ThreadPoolElement 成员的状态来决定是否执行该任务注册的任务完成回调函数。然后，线程进入 idle 状态，等待下一个任务提交动作。
5. 线程调度线程执行完一个任务后，也就是一个 ThreadPoolElement 实例被执行后，线程就出在 idle 状态，等待下一个任务提交动作。任务提交与线程执行之间的同步是通过 pool->sem 来实现的。thread_pool_submit_aio 中任务提交后会调用 qemu_sem_post(&pool->sem) 来增加 pool->sem

的计数,worker_thread 在 pool->sem 上醒来后从 pool->request_list 链表上获取下一个要执行的 ThreadPoolElement 节点。

为了处理 IO，QEMU 提供了主循环外的 IO 事件循环，每个 IO 设备都可以创建自己的事件循环或者加入某个事件循环，防止出现阻塞主线程的情况。

2.6 IOThread

```
typedef struct {
    Object parent_obj;

    QemuThread thread;
    AioContext *ctx;
    GMainContext *worker_context;
    GMainLoop *main_loop;
    GOnce once;
    QemuMutex init_done_lock;
    QemuCond init_done_cond;    /* is thread initialization done? */
    bool stopping;
    int thread_id;

    /* AioContext poll parameters */
    int64_t poll_max_ns;
    int64_t poll_grow;
    int64_t poll_shrink;
} IOThread;
```

可以看到 IOThread 的成员包括了 AioContext 和 GMainContext，它既支持 glib 的事件循环，也支持自己实现的 aio 循环。添加 glib 事件循环主要是为了兼容其他 IO 设备，使其可以使用 IOThread。

2.7 协程 coroutine

https://steemit.com/coroutine/@waterflier/7wnfah （完整描述 IO、中断、进程、线程、协程之间的关系）

协程是一种用户态的轻量级线程，相对于系统独立，有自己的上下文，协程的切换也由自己控制，所以相对于进程和线程来说其运行的开销要小得多。

多线程乃至多进程本质上是由编译器和操作系统配合完成的抽象，我们如果将所有的任务交给编译器和运行时，就可以在用户态实现一个多线程调度。我们先来看看进程和线程是如何在不同任务间进行切换的。这种切换被定义为上下文切换

(context switch)，进程和线程做上下文切换的时候总是要进到内核中，然后内核帮我们把当前进程/线程的寄存器保存起来，最后加载下一个进程/线程的寄存器，开始运行。既然只是状态的保存，为何一定要进到内核态，能否把保存的代码挪到用户态，从而节省切换到内核态的开销嘛？这种思想的产物就是协程，因此协程也可以认为是用户态线程。

但是协程的思想很早就出现，例如在抢占式调度出现之前，进程之间的调度要靠进程主动放弃 CPU，就是协同式调度，默认所有的进程都是“善”的。协程在思想上的亮点是让协同式调度重出江湖。进程和线程都是抢占式调度，定时器中断会执行周期抢占，一旦时间片耗尽，即使你还有一行代码就执行完了，不好意思，轮到下一个进程/线程执行了，于是当前进行/线程只能含恨而退。而对于协程来说，不存在抢占的说法，一个协程一旦得到执行，那么它能够一直执行下去，直到其通过 yield 主动释放执行权为止。因此协程的调度依赖于协程之间的相互协作。

为了解决高并发的问题，前些年流行的编程模式是异步 IO + 事件循环，但是这种模式依赖回调函数。在这种情况下，应用开发者需要设定 callback 函数的调用时机，同时保存大量的执行状态，这导致逻辑代码支离破碎，复杂并难以理解。

目前来看，最好的解决方式是采用协程的方式将同异步代码同步化。给使用者提供同步的 IO 操作，但是后台却是异步的。这种能力非常类似内核，因为内核是在异步硬件（利用异步中断，在内核态和用户态之间切换）的基础上抽象出了同步 IO 操作，而协程是在操作系统的异步 IO 的基础上抽象出同步 IO 操作。所以协程也需要实现类似内核的调度器。

QEMU 作为虚拟化软件其主体架构采用的是事件驱动模式，在 main-loop 中监控各种、大量的文件，事件，消息和状态的变化并进行各种操作，当大量的阻塞操作发生时，为不影响 VM 环境的执行效率，一般都采用异步的方式。为了解决异步造成的代码逻辑混乱，和线程模型本身的切换开销的问题，QEMU 中把所有的 block I/O 函数都做成了协程。开发者可以很方便地在执行耗时操作时 yield 出去，然后执行完操作后从该位置继续往下执行，保证了逻辑上的连续性。

如链接的文章作者所说，协程实现通常有三层：1. 协程本身的接口，定义协程的“执行体”和执行体的切换 2. 协程调度器，即后台运行的将异步转为同步的关键（类似内核）3. 同步 IO 接口，供上层应用使用。下面我们从这三个方面，分析 QEMU 协程的实现。

```
1 struct Coroutine {
2     CoroutineEntry *entry;
3     void *entry_arg;
4     Coroutine *caller;
5
6     /* Only used when the coroutine has terminated.  */
7     QSLIST_ENTRY(Coroutine) pool_next;
8
```

```
    size_t locks_held;

    /* Only used when the coroutine has yielded.  */
    AioContext *ctx;

    /* Used to catch and abort on illegal co-routine entry.
     * Will contain the name of the function that had first
     * scheduled the coroutine. */
    const char *scheduled;

    QSIMPLEQ_ENTRY(Coroutine) co_queue_next;

    /* Coroutines that should be woken up when we yield or terminate.
     * Only used when the coroutine is running.
     */
    QSIMPLEQ_HEAD(, Coroutine) co_queue_wakeup;

    QSLIST_ENTRY(Coroutine) co_scheduled_next;
}

Coroutine *qemu_coroutine_create(CoroutineEntry *entry)
void qemu_coroutine_enter(Coroutine *coroutine, void *opaque);
void coroutine_fn qemu_coroutine_yield(void);
```

coroutine 有 win32、ucontext、sigaltstack 三种模式。在 util/qemu-coroutine.c 文件中定义了共有部分的代码和对外接口。创建新的协程，并使用 qemu_coroutine_enter() 进入协程执行环境，转移执行控制权到协程的调用者处需要调用 qemu_coroutine_yield。随后，此函数并不会返回，除非再次调用 qemu_coroutine_enter() 重新进入协程。这就是协程的逻辑。yielding 会切换回 qemu_coroutine_enter 的调用者，在 qemu 中则在发起一个异步的 I/O 请求后切回主线程的 event loop。这些函数也就是上述的第一个层次的函数。

这里主要分析 ucontext 模式。coroutine 的基础是 setjmp/longjmp，这两个 C 库函数的主要作用就是 save、restore 当前程序的运行上下文，包括寄存器、堆栈信息等到 jmp_buf 中。

ucontext 函数组是 setjmp/longjmp 的升级版:int getcontext(ucontext_t *); //初始化 ucontext_t 结构体，将当前的上下文保存到 ucontext_t 中 int setcontext(const ucontext_t *); //设置当前的上下文为 ucontext_t，并跳转至其中 void makecontext(ucontext_t *, (void *)(), int, ...); //制造一个上下文,并设置入口函数 int swapcontext(ucontext_t *, const ucontext_t *); //保存当前上下文到第一个参数中，然后切换到第二个参数代表

的上下文。ucontext 中除了 Coroutine 之外，还有 stack、stack_size 负责保存每个实体的栈，sigjmp_buf 是 sigjmp 提供的上下文保存。从代码上看，只在 qemu_coroutine_new 即创建新的 coroutine 时，使用了 ucontext 相关的函数，其他的都是使用 sigsetjmp。(理由：没懂。todo) 在创建 coroutine 时，需要为新协程创建栈和切换栈，因此使用 ucontext。

```
typedef struct {
    Coroutine base;
    void *stack;
    size_t stack_size;
    sigjmp_buf env;

#ifdef CONFIG_VALGRIND_H
    unsigned int valgrind_stack_id;
#endif

} CoroutineUContext;
```

CoroutineUContext 是对 Coroutine 的封装，多了栈的相关信息。

协程创建结束后，要切换到该协程中，入口函数即 coroutine_trampoline

2.8 虚拟 CPU

虚拟 CPU 的抽象是 CPUState，在这个数据结构里，包含了对 CPU 的描述。

2.9 内存虚拟化

众所周知，内存是计算机系统的一个关键组成部分。使用 Qemu 创建虚拟机时，guest 物理内存是由几个不同层面共同管理的。guest 内部的物理地址到虚拟地址的管理则依靠软件实现的 softmmu 或者 kvm 提供的硬件虚拟化。

2.9.1 物理内存虚拟化

Qemu 命令行选项“-m [size=]megs[,slots=n,maxmem=size]”分别定义了 guest 物理内存的初始化值，内存条(如 DIMM)的可用插槽数以及可支持的 guest 物理内存最大值。这样，在 Qemu 模拟 DIMM 热插拔的过程中，guest 操作系统像 host 一样，能够监测到内存块的添加或移除。比如向 guest 中热插拔一块 DIMM 内存条，就像是在真实物理机上进行的。再者，guest 内存热插拔的操作单位并不是字节，而是插入的 DIMM 内存条数目。

Qemu 代码中的“pc-dimm”设备（hw/mem/pc-dimm.c 文件中 Typeinfo pc_dimm_info 的定义）用来模拟 DIMM 内存条。代码中“pc-dimm”设备的创建相当于逻辑上热插拔了一块内存（参考代码 type_register_static(&pc_dimm_info)）。尽管该设备名字包含的“pc”容易让人误解，但 ppc 和 s390 机器仍沿用了该名字。在 QOM（Qemu Object Model）模型中，“pc-dimm”对象（参考代码 include/hw/mem/pc-dimm.h 文件中的 PCDIMMDevice 结构体）中并没有定义表示其对应物理内存的变量，但定义了一个指针用来指向其对应的“memory-backend”对象。

```
typedef struct PCDIMMDevice {
    /* private */
    DeviceState parent_obj;

    /* public */
    uint64_t addr;
    uint32_t node;
    int32_t slot;
    HostMemoryBackend *hostmem;
} PCDIMMDevice;
```

设备“memory-backend”（参考 Qemu 源码 backends/hostmem.c 文件中 TypeInfo host_memory_backend_info 的定义）描述的是支撑 guest 物理内存的 host 上真实的内存资源。这些内存既可以是匿名映射的内存（参考 backends/hostmem-ram.c 中 TypeInfo ram_backend_info 的定义），也可以是文件映射的内存（参考 backends/hostmem-file.c 中 TypeInfo file_backend_info 的定义）。文件映射这种方式允许 Linux 在 host 宿主机上使用大页分配的内存能够映射到 guest 物理内存，同时实现了共享内存，允许 host 的其他应用程序访问 guest 物理内存。“pc-dimm”对象和“memory-backend”对象（参考 include/sysemu/hostmem.h 文件中的 HostMemoryBackend 结构体）作为 Qemu 中用户可见的 guest 物理内存，可以通过 Qemu 命令行或者 QMP（Qemu Monitor Protocol）对其进行管理。然而这只是冰山一角，下面继续介绍用户不可见的 guest 物理内存的管理。

内存数据结构示意图 memory

```
struct RAMBlock {
    struct rcu_head rcu;
    struct MemoryRegion *mr;
    uint8_t *host;
    ram_addr_t offset;
    ram_addr_t used_length;
    ram_addr_t max_length;
```

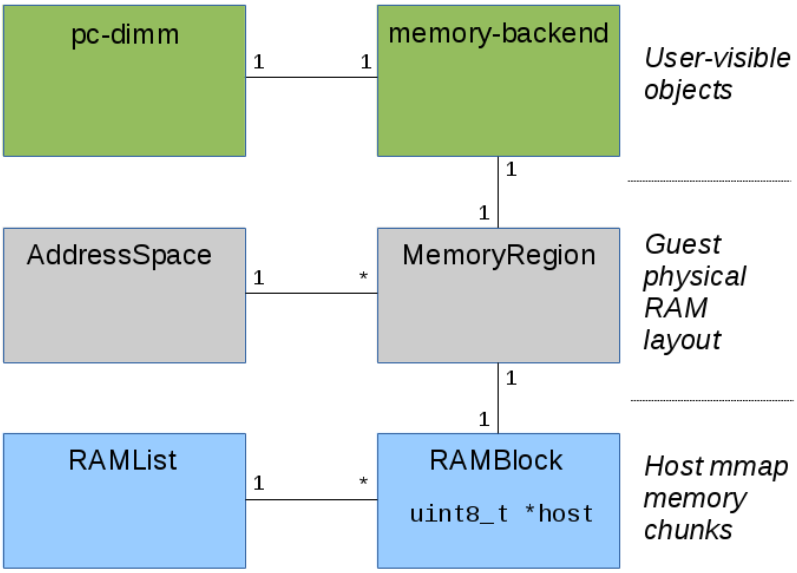


图 2.1: 抄的

```
8     void (*resized)(const char*, uint64_t length, void *host);
9     uint32_t flags;
10    /* Protected by iothread lock.  */
11    char idstr[256];
12    /* RCU-enabled, writes protected by the ramlist lock */
13    QLIST_ENTRY(RAMBlock) next;
14    QLIST_HEAD(, RAMBlockNotifier) ramblock_notifiers;
15    int fd;
16    size_t page_size;
17    /* dirty bitmap used during migration */
18    unsigned long *bmap;
19    /* bitmap of pages that haven't been sent even once
20     * only maintained and used in postcopy at the moment
21     * where it's used to send the dirtymap at the start
22     * of the postcopy phase
23     */
24    unsigned long *unsentmap;
25    /* bitmap of already received pages in postcopy */
26    unsigned long *receivedmap;
27};
28
29typedef struct RAMList {
30    QemuMutex mutex;
31    RAMBlock *mru_block;
```

```
32      /* RCU-enabled, writes protected by the ramlist lock. */
33      QLIST_HEAD(, RAMBlock) blocks;
34      DirtyMemoryBlocks *dirty_memory[DIRTY_MEMORY_NUM];
35      uint32_t version;
36      QLIST_HEAD(, RAMBlockNotifier) ramblock_notifiers;
37  } RAMList;
```

RAMBlock 是 QEMU 管理宿主机虚拟内存的单位，负责存储宿主机某段虚拟地址。“memory-backend” 表示的 host 上真实内存资源由 RAMBlock 调用 exec.c 文件中的 qemu_ram_alloc() 函数，最终通过 mmap 映射到 host 上的一块虚拟内存。RAMBlock 结构体中的 uint8_t *host 指向 host 上虚拟内存的起始值，ram_addr_t offset 表示当前 RAMBlock 相对于 RAMList（描述 host 虚拟内存的全局链表，存储的是宿主机虚拟内存和客户机物理内存的关系）的偏移量。也就是说 ram_addr_t offset 位于一个全局命名空间中，可以通过此 offset 偏移量定位某个 RAMBlock。然而 ram_addr_t 命名空间并不等同于 guest 物理内存空间，它仅表示所有 RAMBlock 集合构成的一个地址空间。举个例子，guest 物理地址 0x100001000 可能并不对应于 ram_addr_t 命名空间中的 0x100001000 地址，因为 ram_addr_t 命名空间不包含 guest 物理内存区域中用于预留和 I/O 内存映射的这些部分。（也就是说 RAMBlock 中不包含此种内存？todo）此外 ram_addr_t offset 的值取决于 RAMBlock 被创建的顺序，而在 guest 物理内存空间中每块内存都有其固定的地址。所有的 RAMBlock 以链表节点的形式存放在全局链表 RAMList（参考 include/exec/ram_addr.h 文件中的 struct RAMList）中，即 ram_list。结构体 ram_list 中记录了所有 RAMBlock 以及“脏”内存 bitmap 的信息。

对于脏页的跟踪有以下用途：

1. 当 guest CPU 或通过设备 DMA 向 guest RAM 中存储数据时，需要通知以下特性：动态迁移特性会一直跟踪“脏”内存页，在迁移过程中一旦发现这些内存页发生变化就会重新向目的主机发送这些“脏”页面。

2. Qemu 中的动态翻译器 TCG（Tiny Code Generator）会一直追踪自调整的代码，当上游指令发生变化时对其重新编译。

3. 显卡仿真会一直追踪视频相关的“脏”内存，并重新绘制扫描线。

以上每种特性在结构体 ram_list 的成员变量 DirtyMemoryBlocks *dirty_memory[D (DIRTY_MEMORY_NUM 等于 3）中都有其对应的 bitmap，为以上三种特性独立的开启或关闭“脏”页面跟踪机制。

```
1
```

```
struct MemoryRegion {
    Object parent_obj;

    /* All fields are private - violators will be prosecuted */

    /* The following fields should fit in a cache line */
    bool romd_mode;
    bool ram;
    bool subpage;
    bool readonly; /* For RAM regions */
    bool rom_device;
    bool flush_coalesced_mmio;
    bool global_locking;
    uint8_t dirty_log_mask;
    bool is_iommu;
    RAMBlock *ram_block;
    Object *owner;

    const MemoryRegionOps *ops;
    void *opaque;
    MemoryRegion *container;
    Int128 size;
    hwaddr addr;
    void (*destructor)(MemoryRegion *mr);
    uint64_t align;
    bool terminates;
    bool ram_device;
    bool enabled;
    bool warning_printed; /* For reservations */
    uint8_t vga_logging_count;
    MemoryRegion *alias;
    hwaddr alias_offset;
    int32_t priority;
    QTAILQ_HEAD(subregions, MemoryRegion) subregions;
    QTAILQ_ENTRY(MemoryRegion) subregions_link;
    QTAILQ_HEAD(coalesced_ranges, CoalescedMemoryRange) coalesced;
    const char *name;
    unsigned ioeventfd_nb;
    MemoryRegionIoeventfd *ioeventfds;
};
struct AddressSpace {
    /* All fields are private. */
    struct rcu_head rcu;
    char *name;
    MemoryRegion *root;
```

```
47
48      /* Accessed via RCU.  */
49      struct FlatView *current_map;
50
51      int ioeventfd_nb;
52      struct MemoryRegionIoeventfd *ioeventfds;
53      QTAILQ_HEAD(memory_listeners_as, MemoryListener) listeners;
54      QTAILQ_ENTRY(AddressSpace) address_spaces_link;
55  };
```

所有 CPU 体系架构都有其内存空间，有一些架构还有单独的 I/O 地址空间。I/O 地址空间由 AddressSpace 结构体描述，一个 AddressSpace 对应一棵 MemoryRegion 树(其对应关系在 include/exec/memory.h 文件的 AddressSpace 结构体中定义：mr = as->root)。结构体 MemoryRegion 是联系 guest 物理地址空间和描述真实内存的 RAMBlocks 之间的桥梁。每个 MemoryRegion 结构体中定义了 RAMBlock *ram_block 成员指向其对应的 RAMBlock，而在 RAMBlock 结构体中则定义了 struct MemoryRegion *mr 指向对应的 MemoryRegion。

要注意 MemoryRegion 不仅用来描述 RAM，还可以用来描述 I/O 内存，在访问 I/O 内存时调用 read/write 回调函数。如 guest CPU 在访问硬件设备寄存器时通过查找其对应 MemoryRegion 结构体中的信息去访问指定的模拟设备。

函数 address_space_rw() 访问 MemoryRegion 并对该 MR 描述的内存执行 load/store 操作。RAM 类型的 MemoryRegion 描述的内存可通过访问 RAMBlock 中的 guest 物理内存来获取。Guest 物理内存空间定义了一个全局变量 AddressSpace address_space_memory，用来表示跟 RAM 相关的内存。

MemoryRegin 的类型有以下几种：- RAM: a RAM region is simply a range of host memory that can be made available to the guest. You typically initialize these with memory_region_init_ram(). Some special purposes require the variants memory_region_init_resizeable_ram() memory_region_init_ram_from_file(), or memory_region_init_ram_ptr().

- MMIO: a range of guest memory that is implemented by host callbacks; each read or write causes a callback to be called on the host. You initialize these with memory_region_init_io(), passing it a MemoryRegionOps structure describing the callbacks.

- ROM: a ROM memory region works like RAM for reads (directly accessing a region of host memory), and forbids writes. You initialize these with memory_region_init_rom().

- ROM device: a ROM device memory region works like RAM for reads (directly accessing a region of host memory), but like MMIO

for writes (invoking a callback). You initialize these with memory_regior

- IOMMU region: an IOMMU region translates addresses of accesses made to it and forwards them to some other target memory region. As the name suggests, these are only needed for modelling an IOMMU, not for simple devices. You initialize these with memory_region_init_iomm

- container: a container simply includes other memory regions, each at a different offset. Containers are useful for grouping several regions into one unit. For example, a PCI BAR may be composed of a RAM region and an MMIO region.

A container's subregions are usually non-overlapping. In some cases it is useful to have overlapping regions; for example a memory controller that can overlay a subregion of RAM with MMIO or ROM, or a PCI controller that does not prevent card from claiming overlapping BARs.

You initialize a pure container with memory_region_init().

- alias: a subsection of another region. Aliases allow a region to be split apart into discontiguous regions. Examples of uses are memory banks used when the guest address space is smaller than the amount of RAM addressed, or a memory controller that splits main memory to expose a "PCI hole". Aliases may point to any type of region, including other aliases, but an alias may not point back to itself, directly or indirectly. You initialize these with memory_region_init_alias().

- reservation region: a reservation region is primarily for debugging. It claims I/O space that is not supposed to be handled by QEMU itself. The typical use is to track parts of the address space which will be handled by the host kernel when KVM is enabled. You initialize these with memory_region_init_reservation(), or by passing a NULL callback parameter to memory_region_init_io().

关于物理内存的诸多讲解如物理内存映射方式、优先级、子区域、生命周期、，参看 docs/devel/memory.txt。

MemoryRegionOps:操作 Memory Region 内存的回调函数,MemoryListener:更新客户机物理内存映射的回调函数 MemoryRegionSection: Memory Region 的片段 FlatView: 内存平坦展开在 QEMU 的内存管理中的 FlatView 描述了 QEMU 虚拟机内存平坦展开的情况。

```
1    struct FlatRange {
2    MemoryRegion *mr;
```

```
3       hwaddr offset_in_region;
4       AddrRange addr;
5       uint8_t dirty_log_mask;
6       bool romd_mode;
7       bool readonly;
8   };
9
10  /*
11  * FlatView是将树状AS的平行展开，可以想象为一个GUEST内存条，所以FlatView里
        面都是GPA相关的内容
12  * FlatView里含有多个FlatRange，每个FlatRange代表了一段内存，
13  * 所有的FlatRange共同构成了FlatView的Guest内存条
14  * 每个FlatRange通过AddrRange标记该段GUEST内存的大小和长度
15  */
16  struct FlatView {
17      unsigned ref;                /*引用计数*/
18      FlatRange *ranges;           /*指向FlatRange数组*/
19      unsigned nr;                /*已经使用了多少个FlatRange*/
20      unsigned nr_allocated;      /*总共分配了多少FlatRange*/
21  };
```

FlatView 相关数据结构关系图

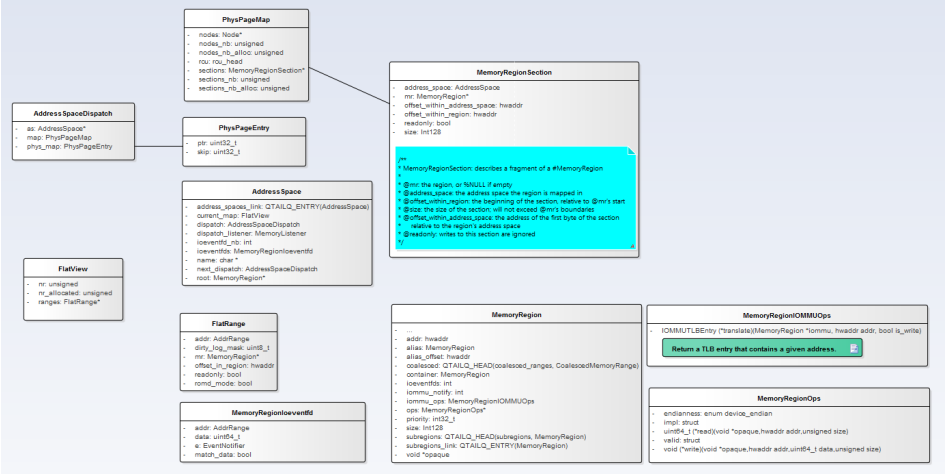


图 2.2: 抄的

FlatView 相关数据结构关系图

FlatView 的原理:1. 首先 FlatView 模型是通过 FlatView 和 FlatRange 两个对象组成。2. FlatView 是该段内存的整体视图的管理结构，一个 FlatView 由一组 FlatRange 组成。3. 每个 FlatRange 代表了虚拟机上的一段内存，多个 FlagRange 就组成了一个内存视图，这些 FlatRange 在物理地址空间上不

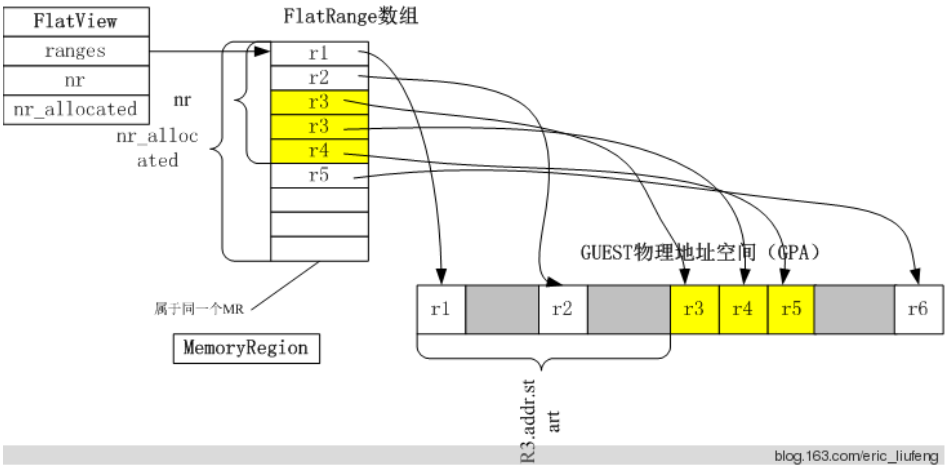


图 2.3：抄的

一定是相邻的。3. 每个 FlatView 代表了某一类内存的组合，用作特殊的用途(如系统内存空间，MMIO 内存地址空间)，通常一个 FlatView 同一个特定用途的 Address_Space 进行关联 4. 每个 FlatRange 所界定的 GUEST 物理地址空间范围通过 AddrRange 所界定,。5. FlatRange 数组在 FlatView 初始化的时候为 0 个，也就是没有分配数组。当进行 flatview_insert() 的操作的时候，才会动态分配出来 6. 为了简化 FlatView，通常将地址空间上连续的 FlatRange 进行合并，合并为 1 个 FlatRange。

2.9.2 客户机虚拟内存

TARGET_PAGE_BITS(12)确定页的大小 V_L1_BITS 最小是 4,最大是 V_L2_BITS+3 V_L2_BITS 10 (可以接着拓展出 L3, L4) v_l1_shift 根据目标机的物理地址空间和宿主机的字长确定 P_L1_BITS 和 V_L1_BITS 剩余部分可以被 L2_BITS 均分

PageDesc: 主要保存 guest page 中的第一個 tb (TranslationBlock *)，是关于 TB 和物理页面的关联。QEMU 为 PageDesc 维护了一个二级页表 l1_map,page_find 函数根据输入的地址查找 l1_map，返回对应的 PageDesc。这跟 QEMU 内部运作机制有关，在某些情况下 guest page (guest binary) 可能被替换或者被写。这时候，QEMU 会以 guest page 为单位，清空与它相关的 TB，需要用到这个数据结构。在 TranslationBlock 中有两个成员，page_addr[2]，存放的就是客户机二进制所在的 guest page，有可能跨页，因此长度是 2;page_next[2],存放下一个 tb。当通过 PageDesc->first_tb 找到該 guest page 的第一个 tb，tb->page_next 就被用来找寻 guest page 的下一個 tb。

```
1    typedef struct CPUTLBEntry {
2    /* bit TARGET_LONG_BITS to TARGET_PAGE_BITS : virtual address
```

```
       bit TARGET_PAGE_BITS-1..4  : Nonzero for accesses that should not
                                    go directly to ram.
       bit 3                      : indicates that the entry is invalid
       bit 2..0                   : zero
    */
    union {
        struct {
            target_ulong addr_read;
            target_ulong addr_write;
            target_ulong addr_code;
            /* Addend to virtual address to get host address.  IO
               accesses
               use the corresponding iotlb value.  */
            uintptr_t addend;
        };
        /* padding to get a power of two size */
        uint8_t dummy[1 << CPU_TLB_ENTRY_BITS];
    };
} CPUTLBEntry;
```

我们跟踪一下 atomic_mmu_lookup 函数，该函数是返回客户机虚拟地址所对应的客户机物理地址，虚拟 tlb 中记录了客户机虚拟地址和宿主机虚拟地址之间的偏移，而没有客户机虚拟和物理地址之间偏移。如果 tlb 命中就直接返回，否则调用 tlb_fill 对 tlb 填充，调用 helper 函数 x86_cpu_handle_mmu_fault ，该函数可以从 c 代码调用也可以从生成的二进制代码中调用（因为是 helper 函数）。x86_cpu_handle_mmu_fault 函数会检查此缺页是由缺页引发的，还是已经在页表中存在。如果已经存在，那么直接填充 tlb。填充的过程涉及到查找页表，因此需要客户机物理地址到宿主机虚拟地址翻译（x86_ldq_phys 函数利用 memory region），然后从该虚拟地址拿到数据，经过逐级页表翻译，最后得到客户机想要访问的虚拟地址对应的客户机物理地址，在 tlb 表中添加新的 tlbentry。也就是说这个函数完成了 mmu 的基本功能。除了 atomic_mmu_lookup 函数之外，还有很多函数调用 tlb_fill，如 aget_page_addr_code、helper_le_ld_name 等。

梳理完思路，我们可以确定 QEMU 定义的 softmmu 函数有以下几个：

```
static void * const qemu_ld_helpers[16] = {
    [MO_UB]   = helper_ret_ldub_mmu,
    [MO_LEUW] = helper_le_lduw_mmu,
    [MO_LEUL] = helper_le_ldul_mmu,
    [MO_LEQ]  = helper_le_ldq_mmu,
    [MO_BEUW] = helper_be_lduw_mmu,
```

```
    [MO_BEUL] = helper_be_ldul_mmu,
    [MO_BEQ]  = helper_be_ldq_mmu,
};
static void * const qemu_st_helpers[16] = {
    [MO_UB]   = helper_ret_stb_mmu,
    [MO_LEUW] = helper_le_stw_mmu,
    [MO_LEUL] = helper_le_stl_mmu,
    [MO_LEQ]  = helper_le_stq_mmu,
    [MO_BEUW] = helper_be_stw_mmu,
    [MO_BEUL] = helper_be_stl_mmu,
    [MO_BEQ]  = helper_be_stq_mmu,
};
```

这几个函数是在 accel/tcg/softmmu_template.h 中实际定义的，

```
# define USUFFIX
# define MMUSUFFIX mmu
# define helper\_le\_ld\_name  glue(glue(helper\_le\_ld, USUFFIX),
   MMUSUFFIX)
```

所以上文提到的 helper_le_ld_name 就是一个 softmmu 的函数，其输入参数为客户机虚拟地址，经过 softmmu 转换为客户机虚拟地址，然后对该地址进行读写。

根据我的观察，QEMU 对 softmmu 的访问分为两种，一种是 QEMU 本身对 softmmu 的访问，如在翻译客户机代码时，QEMU 必须将 pc 指针转换为客户机虚拟地址才能正确访问，因此要使用到 softmmu；另一种是翻译客户机代码的访存指令时，这些指令肯定要访问 softmmu 才能正确访存，所以生成代码过程中需要考虑 softmmu。针对这两种情况，QEMU 的处理方式不同。前者是通过进一步包装上述函数，给上层程序提供更好用的接口：cpu_ldub_code(CPUArchState *env, target_ulong ptr) cpu_ldub_data(CPUArchState *env, target_ulong ptr) 后者则是使用 qemu_ld_helpers 调用这些函数:例如 tcg_out_call(s, qemu_ld_helpers[opc & (MO_BSWAP | MO_SIZE)]); tcg_out_call 函数会生成一条 qemu 的调用指令，调用 qemu_ld_helpers 中注册的函数。这样当生成的 host 二进制代码访问内存时就会通过这些接口。

除了在 C 程序里调用 softmmu 相关的函数，在客户机代码执行期间也会访存，那在生成的宿主机二进制代码是怎么访问客户机的虚拟地址？首先在生成 IR 过程中，所有访存操作都被翻译成 qemu_ld 和 qemu_st 指令。生成宿主机二进制时，分为 tlb 命中和不命中两种，如果不命中，在 tb 的结尾处调用 qemu_ld_helpers[]，执行 tlb 填充。（todo）

2.10 TCG

TCG 是动态二进制翻译的核心。QEMU 里有两次代码翻译，一是从客户机二进制代码到 TCG IR，一是从 TCG IR 到宿主机二进制代码。

2.10.1 TCG IR

TCG 定义了一种中间表示 IR，即 intermediate representation，IR 大致分成如下几类：

1. Move 操作：mov_i32,movi_i32 等
2. Load/store 操作：ld8u_i32,st8_i32 等
3. Arithmetic 操作：add_i32,sub_i32 等
4. Logic 操作：shr_i32,rotl_i32 等
5. Branch 操作：br，brcond
6. Call 操作：call
7. QEMU 特定操作：insn_start,exit_tb,goto_ptr,qemu_ld_i32 等。

TCG 在将客户机的二进制代码翻译为 IR 时，是以 TB 为单位的，其结尾是分支指令。

```
 typedef enum TCGOpcode {
#define DEF(name, oargs, iargs, cargs, flags) INDEX_op_ ## name,
#include "tcg-opc.h"
#undef DEF
    NB_OPS,
} TCGOpcode;
DEF(mul_i32, 1, 2, 0, 0)
```

上面的代码就是 IR 的操作码定义，最后一句话定义了一个 mul_i32 指令，这个指令的输出参数有 1 个，输入参数两个，常量操作数为 0，符号位为 0。符合常规乘法的定义。

```
    typedef struct TCGOp {
    TCGOpcode opc   : 8;        /*  8 */

    /* The number of out and in parameter for a call.  */
    unsigned calli  : 4;        /* 12 */
```

```
    unsigned callo  : 2;        /* 14 */
    unsigned        : 2;        /* 16 */

    /* Index of the prev/next op, or 0 for the end of the list.  */
    unsigned prev   : 16;       /* 32 */
    unsigned next   : 16;       /* 48 */

    /* Lifetime data of the operands.  */
    unsigned life   : 16;       /* 64 */

    /* Arguments for the opcode.  */
    TCGArg args[MAX_OPC_PARAM];
} TCGOp;
```

一条 IR 操作的定义如上所示：包括一个操作码，参数数目，上一个/下一个操作指针，操作数生命周期，操作的参数数组。最后一个数组中存储的是，具体的操作参数。在 QEMU 运行期间，将客户机二进制指令翻译为上述的 IR 形式。

在 TCG 中变量就有三种，临时变量、局部临时变量、全局变量，生命周期分别是，基本块、函数、全局。变量在 TCG 的实体表示就是 TCGArg，即上述的操作参数。变量的类型就两种，32 位和 64 位整型，除此之外就是 host 寄存器的长度，这与具体的体系结构有关。

```
    typedef enum TCGTempVal {
    TEMP_VAL_DEAD,
    TEMP_VAL_REG,
    TEMP_VAL_MEM,
    TEMP_VAL_CONST,
} TCGTempVal;

typedef struct TCGTemp {
    TCGReg reg:8;
    TCGTempVal val_type:8;
    //存储了变量类型，包括dead，寄存器，内存，常量等。
    TCGType base_type:8;
    //存储数据类型，32位 or 64位，or host 寄存器类型。
    TCGType type:8;//作用未知，可能涉及到复杂的设计


    unsigned int fixed_reg:1;
    unsigned int indirect_reg:1;
    unsigned int indirect_base:1;
    unsigned int mem_coherent:1;
    unsigned int mem_allocated:1;
```

```
    /* If true, the temp is saved across both basic blocks and
       translation blocks.  */
    unsigned int temp_global:1;
    /* If true, the temp is saved across basic blocks but dead
       at the end of translation blocks.  If false, the temp is
       dead at the end of basic blocks.  */
    unsigned int temp_local:1;
    unsigned int temp_allocated:1;
    //以上1bit 的标志位分别表示是否是host固定的寄存器，间接寄存器等。
    //还有标志变量类型的 bit 位，确定是局部临时还是临时。

    tcg_target_long val;
    //保存值得地方
    struct TCGTemp *mem_base;
    intptr_t mem_offset;
    const char *name;

    /* Pass-specific information that can be stored for a temporary.
       One word worth of integer data, and one pointer to data
       allocated separately.  */
    uintptr_t state;
    void *state_ptr;
} TCGTemp;
```

每个 vCPU 有自己的 TCGContext，在生成 TCG IR 时要用到该数据结构。

```
   struct TCGContext {
    uint8_t *pool_cur, *pool_end;
    TCGPool *pool_first, *pool_current, *pool_first_large;
    int nb_labels;
    int nb_globals;
    int nb_temps;
    int nb_indirects;

    /* goto_tb support */
    tcg_insn_unit *code_buf; //tb 翻译得到的host指令
    uint16_t *tb_jmp_reset_offset; /* tb->jmp_reset_offset */
    uintptr_t *tb_jmp_insn_offset; /* tb->jmp_target_arg if direct_jump
        */
    uintptr_t *tb_jmp_target_addr; /* tb->jmp_target_arg if !direct_jump
        */

    TCGRegSet reserved_regs;
    uint32_t tb_cflags; /* cflags of the current TB */
```

```
17      intptr_t current_frame_offset;
18      intptr_t frame_start;
19      intptr_t frame_end;
20      TCGTemp *frame_temp;
21
22      tcg_insn_unit *code_ptr;//指向当前翻译到的指令的指针
23
24  #ifdef CONFIG_PROFILER
25      TCGProfile prof;
26  #endif
27
28  #ifdef CONFIG_DEBUG_TCG
29      int temps_in_use;
30      int goto_tb_issue_mask;
31  #endif
32
33      int gen_next_op_idx;
34
35      /* Code generation.  Note that we specifically do not use
           tcg_insn_unit
36         here, because there's too much arithmetic throughout that relies
37         on addition and subtraction working on bytes.  Rely on the GCC
38         extension that allows arithmetic on void*.  */
39      void *code_gen_prologue;
40      void *code_gen_epilogue;
41      void *code_gen_buffer;//存储翻译得到的指令
42      size_t code_gen_buffer_size;
43      void *code_gen_ptr; // 指向翻译得到的指令
44      void *data_gen_ptr;
45
46      /* Threshold to flush the translated code buffer.  */
47      void *code_gen_highwater;
48
49      /* Track which vCPU triggers events */
50      CPUState *cpu;                       /* *_trans */
51
52      /* These structures are private to tcg-target.inc.c.  */
53  #ifdef TCG_TARGET_NEED_LDST_LABELS
54      struct TCGLabelQemuLdst *ldst_labels;
55  #endif
56  #ifdef TCG_TARGET_NEED_POOL_LABELS
57      struct TCGLabelPoolData *pool_labels;
58  #endif
59
60      TCGLabel *exitreq_label;
```

```
61
62      TCGTempSet free_temps[TCG_TYPE_COUNT * 2];
63      TCGTemp temps[TCG_MAX_TEMPS]; /* globals first, temps after */ //存储
            IR 中所有的变量
64
65      /* Tells which temporary holds a given register.
66         It does not take into account fixed registers */
67      TCGTemp *reg_to_temp[TCG_TARGET_NB_REGS];
68
69      TCGOp gen_op_buf[OPC_BUF_SIZE]; //存储翻译得到的IR
70
71      //每条客户机指令对应多条 IR 指令，因此此处用于记录insn_start 时，
72      uint16_t gen_insn_end_off[TCG_MAX_INSNS];
73       //存储某条客户机指令的位置在生成的 host 指令缓存 code_buf 中的
             offset
74      target_ulong gen_insn_data[TCG_MAX_INSNS][TARGET_INSN_START_WORDS];
75      //记录某条客户机指令的 insn_start 指令的参数（不知道指的是什么todo）
76      //此处参数一般包括两个:一个是条件码操作 一个是 tb 中待翻译的客户机的
            PC指针
77
78    //由于在 x86等体系架构中，条件码是在执行完一条指令后，自动计算条件码，
        在 ARM 中则是只在请求的情况下才计算条件码。对于模拟来说，每条指令都
        计算会造成巨大开销。因此只记录下来，等到需要条件码时，根据 这些信息
        重新计算条件码。同时 PC 指针也是，x86每次执行一条指令就更新 pc。此处
        记录的信息与这一目的有关）
```

TCGv_i32 等系列类型是变量地址相对于 TCGContext 基址的偏移。gen_helper_x_y 函数用于处理复杂指令。

DisasContext 与反汇编有关，当客户机二进制反汇编时需要用到此数据结构。在客户机二进制代码翻译为 TCG IR 之前，我们必须能够把二进制翻译成汇编，当然所谓翻译成汇编并不是真正翻译，而是根据具体体系结构的机器码，对二进制代码进行解析。所有的体系结构都采用硬编码的方法。

```
1
2   typedef struct DisasContext {
3       DisasContextBase base;
4
5       /* current insn context */
6       int override; /* -1 if no override */
7       int prefix;
8       TCGMemOp aflag;
9       TCGMemOp dflag;
10      target_ulong pc_start;
```

```
    target_ulong pc; /* pc = eip + cs_base */
    /* current block context */
    target_ulong cs_base; /* base of CS segment */
    int pe;     /* protected mode */
    int code32; /* 32 bit code segment */
#ifdef TARGET_X86_64
    int lma;    /* long mode active */
    int code64; /* 64 bit code segment */
    int rex_x, rex_b;
#endif
    int vex_l;  /* vex vector length */
    int vex_v;  /* vex vvvv register, without 1's compliment.  */
    int ss32;   /* 32 bit stack segment */
    CCOp cc_op;  /* current CC operation */
    bool cc_op_dirty;
    int addseg; /* non zero if either DS/ES/SS have a non zero base */
    int f_st;   /* currently unused */
    int vm86;   /* vm86 mode */
    int cpl;
    int iopl;
    int tf;     /* TF cpu flag */
    int jmp_opt; /* use direct block chaining for direct jumps */
    int repz_opt; /* optimize jumps within repz instructions */
    int mem_index; /* select memory access functions */
    uint64_t flags; /* all execution flags */
    int popl_esp_hack; /* for correct popl with esp base handling */
    int rip_offset; /* only used in x86_64, but left for simplicity */
    int cpuid_features;
    int cpuid_ext_features;
    int cpuid_ext2_features;
    int cpuid_ext3_features;
    int cpuid_7_0_ebx_features;
    int cpuid_xsave_features;
    sigjmp_buf jmpbuf;
} DisasContext;
```

2.10.2 Translation Block

QEMU 翻译代码是以 TB 为单位的，每个 TB 对应一个基本块。基本块结束于分支指令，goto_bt and exit_tb 指令。基本块开始于前一个基本块的结束或者在 set_label 之处。

```
struct TranslationBlock {
```

```
    target_ulong pc;   /* simulated PC corresponding to this block (EIP + CS base) */
    target_ulong cs_base; /* CS base for this block */
    uint32_t flags; /* flags defining in which context the code was generated */
    uint16_t size;      /* size of target code for this block (1 <=
                           size <= TARGET_PAGE_SIZE) */
    uint16_t icount;
    uint32_t cflags;    /* compile flags */
#define CF_COUNT_MASK  0x00007fff
#define CF_LAST_IO     0x00008000 /* Last insn may be an IO access.  */
#define CF_NOCACHE     0x00010000 /* To be freed after execution */
#define CF_USE_ICOUNT  0x00020000
#define CF_INVALID     0x00040000 /* TB is stale. Setters need tb_lock */
#define CF_PARALLEL    0x00080000 /* Generate code for a parallel context */
/* cflags' mask for hashing/comparison */
#define CF_HASH_MASK   \
    (CF_COUNT_MASK | CF_LAST_IO | CF_USE_ICOUNT | CF_PARALLEL)

    /* Per-vCPU dynamic tracing state used to generate this TB */
    uint32_t trace_vcpu_dstate;

    struct tb_tc tc;

    /* original tb when cflags has CF_NOCACHE */
    struct TranslationBlock *orig_tb;
    /* first and second physical page containing code. The lower bit
       of the pointer tells the index in page_next[] */
    struct TranslationBlock *page_next[2];
    tb_page_addr_t page_addr[2];

    /* The following data are used to directly call another TB from
     * the code of this one. This can be done either by emitting direct or
     * indirect native jump instructions. These jumps are reset so that the TB
     * just continues its execution. The TB can be linked to another one by
     * setting one of the jump targets (or patching the jump instruction). Only
     * two of such jumps are supported.
     */
    uint16_t jmp_reset_offset[2]; /* offset of original jump target */
#define TB_JMP_RESET_OFFSET_INVALID 0xffff /* indicates no jump generated
```

```
     */
40   uintptr_t jmp_target_arg[2];  /* target address or offset */
41
42   /* Each TB has an associated circular list of TBs jumping to this one
         .
43    * jmp_list_first points to the first TB jumping to this one.
44    * jmp_list_next is used to point to the next TB in a list.
45    * Since each TB can have two jumps, it can participate in two lists.
46    * jmp_list_first and jmp_list_next are 4-byte aligned pointers to a
47    * TranslationBlock structure, but the two least significant bits of
48    * them are used to encode which data field of the pointed TB should
49    * be used to traverse the list further from that TB:
50    * 0 => jmp_list_next[0], 1 => jmp_list_next[1], 2 => jmp_list_first.
51    * In other words, 0/1 tells which jump is used in the pointed TB,
52    * and 2 means that this is a pointer back to the target TB of this
          list.
53    */
54   uintptr_t jmp_list_next[2];
55   uintptr_t jmp_list_first;
56 };
```

2.10.3　TCG IR 具体分析

TCG 的相关约定:辅助生成 TCG 操作数的函数命名格式是:tcg_gen_<op>[i]_<reg_size>，其中 op 是为参数生成 TCG 操作；[i] 后缀用于表示 TCG 操作使用立即数而不是正常的寄存器；<reg_size> 是指正在使用的寄存器的尺寸。在大多数情况下，该值和被模拟的 target 的原生尺寸一致，因此不会强制使用 i32 和 i64，像 tl 也可用于 helper 函数在 32bit 的机器上执行 32bit 寄存器 move 指令，形如：tcg_gen_mov_tl 而不是 tcg_gen_mov_i32。tcg_gen_xxx 参数的排列顺序是，先返回值，后操作数。以 a=b+c 为例，tcg_gen_xxx(a,b,c)。

TCG 的寄存器：大部分被模拟的处理器的核心寄存器应该有一个等价的 TCG 寄存器。对于处理器状态标志位（相对诡异，如 x86）来说，移植需要使用一些技巧来加速。这些由移植维护者来决定。声明一个命名 TCG 寄存器。TCGv reg = tcg_global_mem_new(TCG_AREG0, offsetof(CPUState, reg), "reg");

TCG temporaries：通常，target 的指令不能分离成一个或者两个简单的 RISC 指令，这就意味着 temporary 寄存器可能需要存储中间结果。不同于每个前端维护自己的临时 scratch 寄存器的静态集合，temporary 提供 helper 来动态管理这些寄存器。

TCG Labels：用于生成条件代码。

TCG 操作：上面说了一般都是 tcg_gen_<op>[i]_<reg_size>，这里特别介绍一下内存访问有关的。因为这块对理解 QEMU 动态翻译很重要。QEMU 生成的代码应该有两大类访存，一是访问宿主机内存，形如 tcg_gen_ld8s_tl(reg, cpu_env, offsetof(CPUState, reg)) 即是指将 CPUState 中的一个寄存器的内容加载到 TCG 的寄存器中，而 CPUState 的寄存器就是 host 内存的内容。另一类是访问客户机内存,如 tcg_gen_qemu_ld8s(ret, addr, mem_idx), 将客户机的内存加载到 TCG 寄存器中，这里会涉及到 TLB 是否命中和 softmmu 的问题。

在这个地方，可以总结一下 CPUState 的状态怎么模拟的问题，我们知道 CPUState 里存储了关于 CPU 的所有状态，比如各个寄存器的值，CPU 运行模式等等。这些值在运行过程中会不断更新，那么 QEMU 是怎么更新这些值得呢？首先由于 TCG IR 的指令属于 RISC，而且支持的指令也很少，也就是说很多复杂指令（特别地涉及到很多有强大功能的寄存器）只能依靠宿主机的 helper 函数模拟，如果涉及到 CPUState 中寄存器的修改就直接修改之（参看下面的代码）。

```
1  void helper_write_crN(CPUX86State *env, int reg, target_ulong t0)
2  {
3  //对 CPU 的控制寄存器 CRn 进行写操作。
4      cpu_svm_check_intercept_param(env, SVM_EXIT_WRITE_CR0 + reg, 0, GETPC
           ());
5      switch (reg) {
6      case 0:
7          cpu_x86_update_cr0(env, t0);
8          break;
9      case 3:
10         cpu_x86_update_cr3(env, t0);
11         break;
12     case 4:
13         cpu_x86_update_cr4(env, t0);
14         break;
15     case 8:
16         if (!(env->hflags2 & HF2_VINTR_MASK)) {
17             qemu_mutex_lock_iothread();
18             cpu_set_apic_tpr(x86_env_get_cpu(env)->apic_state, t0);
19             qemu_mutex_unlock_iothread();
20         }
21         env->v_tpr = t0 & 0x0f;
22         break;
23     default:
24         env->cr[reg] = t0;
25         //直接修改 env 的 cr 寄存器内容
26         break;
27     }
```

```
28
29      void cpu_x86_update_cr3(CPUX86State *env, target_ulong new_cr3)
30  {
31      X86CPU *cpu = x86_env_get_cpu(env);
32
33      env->cr[3] = new_cr3;//直接修改 env 的 cr 寄存器内容
34      if (env->cr[0] & CR0_PG_MASK) {
35          qemu_log_mask(CPU_LOG_MMU,
36                        "CR3 update: CR3=" TARGET_FMT_lx "\n", new_cr3);
37          tlb_flush(CPU(cpu));
38      }
39  }
```

其他的通用寄存器在客户机运行过程中需要修改是依赖二进制翻译的，QEMU 保证 CPUState 结构的指针在翻译过程中始终处于 R14（x64）或者 EBP（x86）寄存器中。在客户机代码涉及到对寄存器操作的时候，QEMU 首先翻译出一条 TCG 指令，读取该“客户机寄存器”（如 tcg_gen_ld8s_tl(reg, cpu_env, offsetof(CPUState, reg)），然后再对该寄存器进行操作如加减乘除等。可以看到这条生成中间代码的指令参数是正是 CPUState 以及要操作的寄存器的偏移，在操作结束再将”客户机寄存器“数据写回到 CPUState 中，这样就达成了”客户机寄存器“的动态变化。上面我们介绍了 TCG 对于寄存器和变量的规定，也就是说 TCG 生成 IR 时，会自动生成这些微操作达成目的。

2.11　TCG README 翻译

2.11.1　TCG 相关的定义

在 TCG 中，guest 是指被模拟的体系结构，target 是指 host 的体系结构。TCG “function” 对应 QEMU 的 TB TCG “temporary” 对应基本块中的变量 TCG “local temporary” 对应只存在于函数中的变量 TCG ” global “是指在所有函数中都存在的变量（类似 C 语言全局变量。既可以是内存位置（如 QEMU CPU 寄存器），还可以是固定的 host 寄存器（如 QEMU CPU 状态指针）。(?? 没懂）TCG “basic block” 对应于以分支指令结束的一系列指令。未定义行为的操作会导致 crash。未指定行为的操作不会 crash，它的结果是很多种可能性中的一个，因此被认为是未定义的结果。

2.11.2 中间表示

介绍：TCG 指令操作临时、本地临时、全局三种变量。TCG 指令和变量都是强类型的。支持两种类型：32 位整数和 64 位整数。指针被定义为 32 和 64 位整型的别名，长度取决于 TCG target 的字长。

每个指令都有固定的数量的输出变量操作数、输入变量操作数和常量操作数。

值得注意的是 call 指令的操作数是不定的。

在文本模式之下，输出操作数通常先出现，然后是输入操作数，最后是常量操作数。指令名称包括了输出的类型。常量以 $ 为前缀。如 add_i32 t0,t1,t2 (t0 = t1 + t2)

假设：

1. Basic blocks 基本块结束于分支指令，goto_bt and exit_tb 指令。基本块开始于前一个基本块的结束或者在 set_label 之处。（基本块结束时，临时变量被销毁，local 临时变量和全局变量仍然存在。）

2. 浮点类型不支持

3. 指针：取决于 TCG target，指针长度是 32 或者 64 位。

4. helpers：使用 tcg_gen_helper_x_y 可以调用任何函数，无论是 i32，i64 还是指针类型。在默认情况下，调用 helper 之前，所有的全局变量都存储在自己的 canonical 位置，并且假设函数可以修改他们。默认情况下，helper 可以修改 CPU 状态或者触发异常。

   可以使用下面的函数修饰符:1.TCG_CALL_NO_READ_GLOBALS 2.TCG_CALL_NO_WRITE_G 3.TCG_CALL_NO_SIDE_EFFECTS

5. 分支使用指令 br 跳转到一个标签

代码优化：当生成指令时，考虑至少以下优化：

2.11.3 后端

tcg-target.h 包含了 target 特定的定义。tcg-target.inc.c 包含了 target 特定的代码，这个文件由 tcg/tcg.c 包含，而不是作为单独的 C 文件。

假设：target 字长是 32 位或 63 位。指针长度和字长相同。在 32 位 target 上，所有 64 位操作都被转化为 32 位。很少的特定操作允许。在 64 位 target 上，值在 32 位和 64 位寄存器之间传送时，为了能正确截断或者扩展，需要使用以下操作：trunc_shr_i64_i32 ext_i32_i64 extu_i32_i64

约束：GCC

第 3 章
主流程

QEMU 的 main 函数主要是负责初始化。初始化 CPU 列表，初始化 CPU 循环，设置执行退出的清理函数，初始化 QEMU 模块，设置 QEMU 选项参数相关信息，运行状态初始化，加密初始化，参数解析，启动 IO 线程、信号处理线程等。在根据选项设置完成初始化之后，QEMU 开始运行，进入 main_loop 函数。退出 main_loop 就意味着虚拟机运行结束，此时，结束所有的 IO 线程、停止 vcpu 运行、关闭块设备、释放各种资源、清除可信部件、网络、声卡、监控器、字符设备等。

在 main_loop 函数中,对外设进行模拟和 IO 处理。在 machine_run_board_init(curr 函数中，调用 machine_class->init(machine) 对虚拟机进行初始化，创建 vcpu，开始运行机器。

3.1 重要函数分析

3.1.1 QEMU 初始化函数

此处重点是根据对初始化函数跟踪，理解 AIO 机制。

```
1  qemu_init_main_loop(Error **errp)
2  int qemu_init_main_loop(Error **errp)
3  {
4      int ret;
5      GSource *src;
6      Error *local_error = NULL;
7
8      init_clocks(qemu_timer_notify_cb);
9      /*初始化时钟，每个时钟都有自己定时器队列，以及定时器对应的通知器队
           列，初始化的回调函数是qemu_timer_notify_cb，回调函数在定时器到期时
           会执行。
10     这个默认的回调函数根据时钟类型和模式处理，时钟有两种模式：一种 icount
           模式会考虑虚拟机与宿主机的时间同步，一种不启用 icount 模式。在
           icount 模式下或者时钟类型是虚拟时钟，这两种情况都需要同步时钟时间
           （虚拟时钟在虚拟机停止的情况下不再计时），如果 qemu 的 vcpu 正在运
           行，需要打断它，再实现同步的目的；其他情况直接对 AIO 进行置位，通
           知事件到来。
```

```
11      */
12      ret = qemu_signal_init();
13      /*信号初始化，屏蔽主线程不需要处理的信号：SIG_IPI、SIGIO、SIGALRM、
            SIGBUS。
14      信号处理有两种方式：
15      一种是调用 signalfd 系统调用，该系统调用可以将信号转换到文件描述符，
            以便于用 epoll 等对信号监控。
16      二种是 传统方式，创建一个信号处理线程，调用 sigwaitinfo将异步信号转换
            成同步信号，排队处理。同时建立 pipe，每当信号到来时，就向 pipe 的
            一端写入信号 signo，另一端由 epoll 监控，达到和以第一种同样的效
            果。
17
18      接下来，为该信号的文件描述符设置处理函数，这个地方再次与 AIO 发生关
            系。前文讲过，在 QEMU 中有两个 AioContext 类型的变量，
            qemu_aio_context 和 iohandler_ctx。此处需要把信号的文件描述符加入
            到 iohandler_ctx 变量中。
19      在此函数中iohandler_init 对 iohandler_ctx 进行初始化， 上面我们已经讲
            述过qemu_aio_context 的初始化。
20      */
21
22      if (ret) {
23          return ret;
24      }
25
26      qemu_aio_context = aio_context_new(&local_error);
27      //此处是 qemu_aio_context 的初始化。
28
29      if (!qemu_aio_context) {
30          error_propagate(errp, local_error);
31          return -EMFILE;
32      }
33
34      qemu_notify_bh = qemu_bh_new(notify_event_cb, NULL);
35      //初始化 QEMUBU* qemu_notify_bh。 qemu_bh 是下半部的概念。
36
37      gpollfds = g_array_new(FALSE, FALSE, sizeof(GPollFD));
38      src = aio_get_g_source(qemu_aio_context);
39      g_source_set_name(src, "aio-context");
40      g_source_attach(src, NULL);
41      g_source_unref(src);
42      src = iohandler_get_g_source();
43      g_source_set_name(src, "io-handler");
44      g_source_attach(src, NULL);
45      g_source_unref(src);
46      //最后把 qemu_aio_context 和 iohandler_ctx attach 到默认的主循环中。
```

```
47      //QEMU在初始化的过程中用 g_source_attach 函数把 qemu_aio_context和
            iohandler->ctx 添加到主循环。
48      return 0;
49  }
```

3.1.2 main_loop

现在进入 main_loop 函数。main_loop 函数非常简单：

```
1   static void main_loop(void)
2   {
3       do {
4           main_loop_wait(false);
5       } while (!main_loop_should_exit());
6   }
7
8   static bool main_loop_should_exit(void)
9   {
10      RunState r;
11      ShutdownCause request;
12
13      if (qemu_debug_requested()) {
14          vm_stop(RUN_STATE_DEBUG);
15      }
16      if (qemu_suspend_requested()) {
17          qemu_system_suspend();
18      }
19      request = qemu_shutdown_requested();
20      if (request) {
21          qemu_kill_report();
22          qapi_event_send_shutdown(shutdown_caused_by_guest(request),
23                                   &error_abort);
24          if (no_shutdown) {
25              vm_stop(RUN_STATE_SHUTDOWN);
26          } else {
27              return true;
28          }
29      }
30      .....
```

判断条件是 main_loop_should_exit 的返回值。每次虚拟机从 main_loop_wait 函数退出时，都要在 main_loop_should_exit 函数里判断退出的原因，包括

debug 请求、休眠、关机、重置、唤醒、虚拟机停止等请求。如果确实有关机请求，那么就退出此循环。

```
void main_loop_wait(int nonblocking)
{
    int ret;
    uint32_t timeout = UINT32_MAX;
    int64_t timeout_ns;

    if (nonblocking) {
        timeout = 0;
    }

    /* poll any events */
    g_array_set_size(gpollfds, 0); /* reset for new iteration */
    /* XXX: separate device handlers from system ones */
    slirp_pollfds_fill(gpollfds, &timeout);

    if (timeout == UINT32_MAX) {
        timeout_ns = -1;
    } else {
        timeout_ns = (uint64_t)timeout * (int64_t)(SCALE_MS);
    }

    timeout_ns = qemu_soonest_timeout(timeout_ns,
                                      timerlistgroup_deadline_ns(
                                          &main_loop_tlg));

    ret = os_host_main_loop_wait(timeout_ns);
    slirp_pollfds_poll(gpollfds, (ret < 0));

    /* CPU thread can infinitely wait for event after
       missing the warp */
    qemu_start_warp_timer();
    qemu_clock_run_all_timers();

    //处理全局 timer
}


static int os_host_main_loop_wait(int64_t timeout)
{
    GMainContext *context = g_main_context_default();
    int ret;
    static int spin_counter;
```

```
43
44      g_main_context_acquire(context);
45
46      glib_pollfds_fill(&timeout);
47
48      /* If the I/O thread is very busy or we are incorrectly busy waiting
            in
49       * the I/O thread, this can lead to starvation of the BQL such that
             the
50       * VCPU threads never run.  To make sure we can detect the later case
             ,
51       * print a message to the screen.  If we run into this condition,
             create
52       * a fake timeout in order to give the VCPU threads a chance to run.
53       */
54      if (!timeout && (spin_counter > MAX_MAIN_LOOP_SPIN)) {
55          static bool notified;
56
57          if (!notified && !qtest_enabled() && !qtest_driver()) {
58              warn_report("I/O thread spun for %d iterations",
59                          MAX_MAIN_LOOP_SPIN);
60              notified = true;
61          }
62
63          timeout = SCALE_MS;
64      }
65
66      if (timeout) {
67          spin_counter = 0;
68          qemu_mutex_unlock_iothread();
69      } else {
70          spin_counter++;
71      }
72
73      ret = qemu_poll_ns((GPollFD *)gpollfds->data, gpollfds->len, timeout)
            ;
74
75      if (timeout) {
76          qemu_mutex_lock_iothread();
77      }
78
79      glib_pollfds_poll();
80      //执行 poll 轮询循环。
81      g_main_context_release(context);
82
```

```
83      return ret;
84  }
```

main_loop_wait 是用来执行外围设备模拟代码，当然这些并不是完全绝对的，比如 cpu 在对 io 设备的地址空间进行 load/store 操作时，就会调用 io 设备读写 handler。在 main_loop_wait 函数中，可以响应外围输入，以及处理 timer 事件。在执行过程中不断地去监听对应的 host 设备是否有输入输出，若有，则调用相应的回调函数去处理。

Qemu 中常用的 IO 描述符有下面几类：文件描述符 fd(block io)：虚拟磁盘相关的 io，为了保证高性能，主要使用 aio；signalfd：qemu 的时钟模拟利用了 linux kernel 的 signalfd，定期产生 SIGALRM 信号；eventfd：主要用于 qemu 和 kvm 之间的 notifier，比如 qemu 的模拟设备可以通过 notifier 向 kvm 发送一个模拟中断，kvm 也可以通过 notifier 向 qemu 报告 guest 的各种状态；socket：用于虚拟机迁移，qmp 管理等 timer ：该函数同时还负责轮询系统中所有的定时器，并调用定时器的回调函数；

3.1.3 x86_cpu_realizefn

前面研究了 QEMU 的整体框架设计，下面我们以 x86 的实现为例，进入真正的动态翻译的模块。跟踪 main 函数中的 machine_class->init(machine)，可以发现初始化函数 init 通过层层调用，最终到达了 x86_cpu_realizefn 函数。QEMU 在 x86 平台上可以不使用动态翻译，而是利用 x86 提供的虚拟化加速组件 vmx 和 vt-d 等对虚拟机执行加速，因此 QEMU 支持 kvm、hax、xen 等利用虚拟化硬件的项目。这只有在 x86 平台上虚拟化 x86 平台才可以使用。除此之外，必须利用 tcg 进行动态翻译，在最新的 QEMU 中开始支持 multitcg，充分利用多核 CPU。在最早的 QEMU 中，所有的 vcpu 都在一个线程中分时执行，由于 tcg 代码不具有线程安全性，以及实现上不利用重构为多线程，因此一直是单线程。只有开启 kvm 时，才能实现每个 vcpu 一个线程。不过，最新的代码中已经支持 multitcg（不知道怎么实现的，待看 todo）。我们下面先分析 TCG 的执行过程。当不启用虚拟化硬件加速时，使用 QEMU 的 tcg 进行二进制翻译执行。

在 cpu_exec_realizefn 函数中对 CPU 执行进行初始化，由于我们关注的是 tcg 动态翻译，因此函数跳到 tcg_initialize，在 x86 下就跳到了 tcg_x86_init，此函数位于 target/i386/translate.c 文件中。这个文件是包含特定 CPU 的相关数据结构，如 CPU 通用寄存器名称、长度、个数，访存模式等 CPU 特性，以及从 tcg 中间表示翻译到特定体系结构 CPU 的相关函数。

接下去，执行 qemu_init_vcpu=>qemu_tcg_init_vcpu，在这个函数里，初始化 region partition，创建 vcpu 线程。如果启动 multitcg 会为每个

vcpu 创建一个线程，我们先关注单线程的情况。单线程情况下，每个 vcpu 对线程进行分时抢占。启动一个 kick 定时器，该定时器每过 0.1s 抢占 CPU，保证所有 CPU 分时共享。最后退出 x86_cpu_realizefn 函数，继续执行主循环 main_loop.

3.1.4 tcg_cpu_exec

新创建的 vcpu 线程开始执行客户机代码。

```
    static int tcg_cpu_exec(CPUState *cpu)
{
    int ret;
#ifdef CONFIG_PROFILER
    int64_t ti;
#endif

#ifdef CONFIG_PROFILER
    ti = profile_getclock();
#endif
    qemu_mutex_unlock_iothread();
    cpu_exec_start(cpu);
    ret = cpu_exec(cpu);
    cpu_exec_end(cpu);
    qemu_mutex_lock_iothread();
#ifdef CONFIG_PROFILER
    tcg_time += profile_getclock() - ti;
#endif
    return ret;
}
```

真正执行客户机代码在 cpu_exec 里,该函数位于 accel/tcg/cpu-exec.c。

```
/* if an exception is pending, we execute it here */
while (!cpu_handle_exception(cpu, &ret)) {
    TranslationBlock *last_tb = NULL;
    int tb_exit = 0;

    while (!cpu_handle_interrupt(cpu, &last_tb)) {
        uint32_t cflags = cpu->cflags_next_tb;
        TranslationBlock *tb;
/*如果有请求，为下次执行的 cflags 使用精确设置。
这被用于 icount，精确 smc和访问后停止的 watchpoints。
 Since this request should never have CF_INVALID set, -1 is a convenient
    invalid value that does not require tcg headers for cpu_common_reset.
```

```
        */

          if (cflags == -1) {
              cflags = curr_cflags();
          } else {
              cpu->cflags_next_tb = -1;
          }

          tb = tb_find(cpu, last_tb, tb_exit, cflags);
          cpu_loop_exec_tb(cpu, tb, &last_tb, &tb_exit);
          /* Try to align the host and virtual clocks
             if the guest is in advance */
          align_clocks(&sc, cpu);
      }
}
```

cpu_exec 有两层循环。内层循环处理中断，外层处理异常。当退出执行客户机代码时，判断退出原因，如果有中断或者异常时要在这里处理。

tb_find 负责查找第一个可以执行的 translation block。cpu_loop_exec_tb 负责执行 tb。tb_find 查找 tb 时，首先根据 pc（客户机虚拟地址）值，确定是否已经翻译过该地址，并存放在 code cache 中。

```
    cpu_get_tb_cpu_state(env, pc, cs_base, flags);
    hash = tb_jmp_cache_hash_func(*pc);
    tb = atomic_rcu_read(&cpu->tb_jmp_cache[hash]);
    if (likely(tb &&
               tb->pc == *pc &&
               tb->cs_base == *cs_base &&
               tb->flags == *flags &&
               tb->trace_vcpu_dstate == *cpu->trace_dstate &&
               (tb_cflags(tb) & (CF_HASH_MASK | CF_INVALID)) == cf_mask))
                   {
        return tb;
    }
    tb = tb_htable_lookup(cpu, *pc, *cs_base, *flags, cf_mask);
    if (tb == NULL) {
        return NULL;
    }
    atomic_set(&cpu->tb_jmp_cache[hash], tb);
```

CPUArchState 的 tb_jmp_cache 就保存着已经翻译的 code cache。找到的话直接返回，找不到就重新进入慢速查找，即通过 tb_htable_lookup 查找。

```
    desc.env = (CPUArchState *)cpu->env_ptr;
    desc.cs_base = cs_base;
    desc.flags = flags;
    desc.cf_mask = cf_mask;
    desc.trace_vcpu_dstate = *cpu->trace_dstate;
    desc.pc = pc;
    phys_pc = get_page_addr_code(desc.env, pc);
    desc.phys_page1 = phys_pc & TARGET_PAGE_MASK;
    h = tb_hash_func(phys_pc, pc, flags, cf_mask, *cpu->trace_dstate);
    return qht_lookup(&tb_ctx.htable, tb_cmp, &desc, h);
```

tb_htable_lookup 对 TBContext 的 qht 哈希表进行查找，此查找是根据客户机虚拟地址、物理地址、cs 段选择子、标志寄存器以及其他的特征计算哈希值，从哈希表中找到已经翻译过的 tb。其中 qht_lookup 函数的参数，tb_cmp 是比较找到的 tb 是否和 desc 的各个成员一致，以便确定该 tb 是我们要找的 tb。

```
static bool tb_cmp(const void *p, const void *d)
{
    const TranslationBlock *tb = p;
    const struct tb_desc *desc = d;

    if (tb->pc == desc->pc &&
        tb->page_addr[0] == desc->phys_page1 &&
        tb->cs_base == desc->cs_base &&
        tb->flags == desc->flags &&
        tb->trace_vcpu_dstate == desc->trace_vcpu_dstate &&
        (tb_cflags(tb) & (CF_HASH_MASK | CF_INVALID)) == desc->cf_mask) {
        /* check next page if needed */
        if (tb->page_addr[1] == -1) {
            return true;
        } else {
            tb_page_addr_t phys_page2;
            target_ulong virt_page2;

            virt_page2 = (desc->pc & TARGET_PAGE_MASK) + TARGET_PAGE_SIZE
                ;
            phys_page2 = get_page_addr_code(desc->env, virt_page2);
            if (tb->page_addr[1] == phys_page2) {
                return true;
            }
        }
    }
    return false;
```

```
}
```

这个函数比较了 tb 和 desc 的各个属性，保证所有属性相等。特别地，如果 tb 的第二个页不存在，即 page_addr[1] == -1，可以直接返回；如果第二个页存在，要保证第二个与 PC 相邻的页所对应的物理地址和 page_addr[1] 相等。即，要保证 tb 的两个物理页对应从 PC 开始连续的两页。如果此处找到 tb，就会把这个 tb 加入到 tb_jmp_cache 中，加速下次的查找。

如果实在查找不到，就会调用 tb_gen_code 重新生成 tb。

```
    tb = tb_alloc(pc);
    tcg_func_start(tcg_ctx);//初始化 tcg 翻译相关的数据结构
    gen_intermediate_code(cpu, tb); //生成中间表示
    gen_code_size = tcg_gen_code(tcg_ctx, tb);//生成目标二进制代码
    tb_link_page(tb, phys_pc, phys_page2);//将 tb 加入qht 哈希表中，tb 如
        果跨越两页的话就会分配两个物理页。
```

```
    void gen_intermediate_code(CPUState *cpu, TranslationBlock *tb)
{
    DisasContext dc;

    translator_loop(&i386_tr_ops, &dc.base, cpu, tb);
}
```

gen_intermediate_code=>translator_loop，利用 i386 体系结构操作函数，i386_tr_ops 在循环中对指令进行翻译。translator_loop 的主要逻辑就是根据客户机操作系统的 PC 指针对它所指向的指令进行翻译，直到指令数目达到上限，缓冲区填满或所有指令翻译完毕，退出翻译。真正负责翻译工作的是 disas_insn 函数。该函数形如：byte=get_byte_from_mmu() .... switch(byte): case 0xf1: case 0xf2: ..... 通过对 x86 指令解析生成 TCG IR。由于 X86 是复杂指令集，其指令解析及其复杂，有 1 字节、2 字节、3 字节操作符等数百条指令。而 ARM 是 RISC 指令集，其指令都是定长的，处理起来很方便。此处不再具体分析指令解析过程。

我们还是来看看生成 IR 的具体过程：如 mov 指令，mov cpu_T0,0

```
tcg_gen_movi_tl(cpu_T0, 0);

#define tcg_gen_mov_tl tcg_gen_mov_i32
static inline void tcg_gen_mov_i32(TCGv_i32 ret, TCGv_i32 arg)
{
```

```
6       if (ret != arg) {
7           tcg_gen_op2_i32(INDEX_op_mov_i32, ret, arg);
8       }
9   }
10  static inline void tcg_gen_op2_i32(TCGOpcode opc, TCGv_i32 a1, TCGv_i32
        a2)
11  {
12      tcg_gen_op2(opc, tcgv_i32_arg(a1), tcgv_i32_arg(a2));
13  }
14
15  void tcg_gen_op2(TCGOpcode opc, TCGArg a1, TCGArg a2)
16  {
17      TCGOp *op = tcg_emit_op(opc);
18      //发射操作
19      op->args[0] = a1;
20      op->args[1] = a2;
21      //将两个参数分别记入 op 的 args 成员中。
22  }
23  static inline TCGOp *tcg_emit_op(TCGOpcode opc)
24  {
25      TCGContext *ctx = tcg_ctx;
26      int oi = ctx->gen_next_op_idx;
27      int ni = oi + 1;
28      int pi = oi - 1;
29      TCGOp *op = &ctx->gen_op_buf[oi];
30      tcg_debug_assert(oi < OPC_BUF_SIZE);
31      ctx->gen_op_buf[0].prev = oi;
32      ctx->gen_next_op_idx = ni;
33
34      memset(op, 0, offsetof(TCGOp, args));
35      op->opc = opc;
36      //发射操作就是把该操作记入到 TCGContext 中生成的中间代码gen_op_buf。
37      op->prev = pi;
38      op->next = ni;
39
40      return op;
41  }
```

通过一系列函数调用，mov 操作被记录到了中间代码之中。这就是中间代码生成的结果。

对于复杂指令，无法直接用 IR 表示的，需要利用 tcg_helper_x_y 进行模拟。如果该指令是当前 tb 的最后一条指令，允许进行 IO（没有动懂 todo）。中间代码生成的最后步骤，还生成一些 exit_tb 等指令（QEMU 特有的指令，用法

待定 todo)。在客户机二进制转 TCG IR 时存在 helper，表示无法用 IR 表示这些指令，那么在 IR 转宿主机二进制时是否也存在某些指令需要 helper 呢?

下面我们研究一下 helper 函数是怎么工作的:

```
1  define DEF_HELPER_FLAGS_0(name, flags, ret) \
2  static inline void glue(gen_helper_, name)(dh_retvar_decl0(ret)) \
3  {  \
4    tcg_gen_callN(HELPER(name), dh_retvar(ret), 0, NULL); \
5  }
6
7  void tcg_gen_callN(void *func, TCGTemp *ret, int nargs, TCGTemp **args)
8  {
9      ......
10     i = s->gen_next_op_idx;
11     tcg_debug_assert(i < OPC_BUF_SIZE);
12     s->gen_op_buf[0].prev = i;
13     s->gen_next_op_idx = i + 1;
14     op = &s->gen_op_buf[i];
15     /* Set links for sequential allocation during translation.  */
16     memset(op, 0, offsetof(TCGOp, args));
17     .....
18     op->opc = INDEX_op_call;
19     op->prev = i - 1;
20     op->next = i + 1;
21     ....
22     op->args[pi++] = temp_arg(ret);
23     op->args[pi++] = temp_arg(ret + 1);
24     ....
25     op->args[pi++] = (uintptr_t)func;
26     op->args[pi++] = flags;
27     op->calli = real_args;
28 }
```

由 DEF_HELPER_FLAGS_0(name,flags,ret) 宏创建出形如 gen_helper_name 的函数,如 gen_helper_load_seg 函数。在这个宏里,调用 tcg_gen_callN,参数是 HELPER (name) , 即 helper_name，这些函数是由 QEMU 自定义的辅助函数。在 tcg_gen_callN 中，生成 call 指令，参数分别是指定的 helper_name 函数和返回值。也就是说 call 指令要调用外部的 C 函数，而不是生成的 TCG IR，这就导致 IR 的不完整性。

TCG 代码生成的最后一步，把 TCG IR 翻译为 host 二进制代码。在生成宿主机二进制代码时，会先进行 IR 优化，寄存器分配 (todo)。QEMU 注释提到对于一些 common 的参数模式来说，针对特定的指令采用不同的寄存器分配函

数会更加快速，QEMU 目前针对 mov,insn_start,discard,set_label,call 指令提供了特定函数，而其他的都采用通用处理方式。

下面分析一下寄存器分配,在 tcg_gen_code 中首先调用 tcg_reg_alloc_start，设置全局变量和局部变量的相关参数,如变量类型是 TEMP_VAL_REG 还是 TEMP_VAL_MEM，还是 TEMP_VAL_DEAD，是否是固定的寄存器等。

分配寄存器之后直接进行指令翻译，我们看一个简单的，mov 指令的生成。

```
1  static void tcg_reg_alloc_mov(TCGContext *s, const TCGOp *op){
2      ots = arg_temp(op->args[0]);
3      ts = arg_temp(op->args[1]);
4
5      /* Note that otype != itype for no-op truncation.  */
6      otype = ots->type;
7      itype = ts->type;
8      if (ts->val_type == TEMP_VAL_CONST) {
9          /*如果输入参数是常量就直接生成 mov 指令 ，如果输入参数是变量就复
               杂了*/
10         tcg_target_ulong val = ts->val;
11         if (IS_DEAD_ARG(1)) {
12             temp_dead(s, ts);
13         }
14         tcg_reg_alloc_do_movi(s, ots, val, arg_life);
15         return;
16     }
17 }
18 static void tcg_reg_alloc_do_movi(TCGContext *s, TCGTemp *ots,
19                                   tcg_target_ulong val, TCGLifeData
                                      arg_life)
20 {
21     if (ots->fixed_reg) {
22         /* For fixed registers, we do not do any constant
23         propagation.  */
24         //如果输出参数的寄存器是固定的就不需要分配，直接翻译即可。
25         tcg_out_movi(s, ots->type, ots->reg, val);
26         return;
27     }
28
29     /* The movi is not explicitly generated here.  */
30     if (ots->val_type == TEMP_VAL_REG) {
31         s->reg_to_temp[ots->reg] = NULL;
32     }
33     ots->val_type = TEMP_VAL_CONST;
34     ots->val = val;
35     ots->mem_coherent = 0;
```

```
36      if (NEED_SYNC_ARG(0)) {
37          temp_sync(s, ots, s->reserved_regs, IS_DEAD_ARG(0));
38      } else if (IS_DEAD_ARG(0)) {
39          temp_dead(s, ots);
40      }
41  }
```

至此，guest binary=>TCG IR = > host binary 工作完成，还要把生成的 TB 加入到 QEMU 的管理结构之中，这就是 tb_link_page 的工作。

```
1   static void tb_link_page(TranslationBlock *tb, tb_page_addr_t phys_pc
        ,
2                            tb_page_addr_t phys_page2)
3   {
4       uint32_t h;
5
6       assert_memory_lock();
7
8       /* add in the page list */
9       tb_alloc_page(tb, 0, phys_pc & TARGET_PAGE_MASK);
10      if (phys_page2 != -1) {
11          tb_alloc_page(tb, 1, phys_page2);
12      } else {
13          tb->page_addr[1] = -1;
14      }
15
16      /* add in the hash table */
17      h = tb_hash_func(phys_pc, tb->pc, tb->flags, tb->cflags &
            CF_HASH_MASK,
18                       tb->trace_vcpu_dstate);
19      qht_insert(&tb_ctx.htable, tb, h);
20  }
```

tb_alloc_page 将 tb 加入页表中，并设置页表权限。同时把页表加入到 qht 哈希表中，我们上面分析过，当查找 tb 时如果 cache 中没有会到哈希表中查询。

代码生成结束之后就要接着执行,cpu_loop_exec_tb=>cpu_tb_exec=>tcg_qemu_tb_ex
此时我们发现 tcg_qemu_tb_exec 有两个定义，一个是 tci 的定义（qemu 解释器），一个是宏定义。我们的目标明显是第二个：

```
1   # define tcg_qemu_tb_exec(env, tb_ptr) \
2   ((uintptr_t (*)(void *, void *))tcg_ctx->code_gen_prologue)(env,
        tb_ptr)
```

```
#endif
```

调用的竟燃是一个函数指针，我们只能回头寻找该指针的内容。在 tcg 初始化过程中(tcg_init) 我们找到了 tcg_prologue_init。buf0 = s->code_gen_buffer; s->code_gen_prologue = buf0; code_gen_prologue 指向了生成代码的缓冲区。

接着调用了 tcg_target_qemu_prologue，

```
static void tcg_target_qemu_prologue(TCGContext *s)
{
    tcg_set_frame(s, TCG_REG_CALL_STACK, TCG_STATIC_CALL_ARGS_SIZE,
    CPU_TEMP_BUF_NLONGS * sizeof(long));
                    /* Save all callee saved registers.  */
    for (i = 0; i < ARRAY_SIZE(tcg_target_callee_save_regs); i++) {
        tcg_out_push(s, tcg_target_callee_save_regs[i]);
    }

    s->code_gen_epilogue = s->code_ptr;
    for (i = ARRAY_SIZE(tcg_target_callee_save_regs) - 1; i >= 0; i--) {
        tcg_out_pop(s, tcg_target_callee_save_regs[i]);
    }
   }
```

tcg_set_frame 指定了栈帧的大小，接着就生成了保存寄存器的指令，输出到 code_buf 中，自然也就是 code_gen_prologue 所指向的位置。随后，又在 code_gen_epilogue 指向的位置生成了 pop 寄存器的指令。所以 prologue 和 epilogue 所做的事情类似于操作系统所做的保存上下文、传参和恢复上下文的工作。

当执行 prologue 时，就意味着客户机代码开始执行。所以整个的执行流程是 QEMU=>prologue=>code cache=>epilogue=>QEMU。

我们上面分析了整个 QEMU 的执行流程。在这个流程中，QEMU 开始执行客户机代码是在 tcg_qemu_tb_exec。它从一个给定的 tb 开始执行，在有 tb 的链接的情况下，会接着执行后续的 tb。只有当高层循环的特定事件发生时，控制流才会最终退出：或者是由于控制需要传递给一个还没有直接链接进来的 tb，或者是一个异步事件如中断或异常。

```
    cpu->can_do_io = !use_icount;
    ret = tcg_qemu_tb_exec(env, tb_ptr);
    cpu->can_do_io = 1;
    last_tb = (TranslationBlock *)(ret & ~TB_EXIT_MASK);
```

```
    tb_exit = ret & TB_EXIT_MASK;
    trace_exec_tb_exit(last_tb, tb_exit);
    if (tb_exit > TB_EXIT_IDX1) {
        /* We didn't start executing this TB (eg because the instruction
         * counter hit zero); we must restore the guest PC to the address
         * of the start of the TB.
         */
        CPUClass *cc = CPU_GET_CLASS(cpu);
        qemu_log_mask_and_addr(CPU_LOG_EXEC, last_tb->pc,
                               "Stopped execution of TB chain before %p [
                                  "
                               TARGET_FMT_lx "] %s\n",
                               last_tb->tc.ptr, last_tb->pc,
                               lookup_symbol(last_tb->pc));
        if (cc->synchronize_from_tb) {
            cc->synchronize_from_tb(cpu, last_tb);
        } else {
            assert(cc->set_pc);
            cc->set_pc(cpu, last_tb->pc);
        }
    }
    return ret;
```

tcg_qemu_tb_exec 的返回值是 0 或者 4 字节对齐的指向最后一个尝试执行的 tb 的指针。指针的最低两个有效位保存额外信息，

1. 0,1: 该 TB 和下一个 TB 之间通过特定的 tb 索引（0 或 1）链接。也就是说，我们通过"goto_tb index" 离开当前 tb。主循环使用这个值决定怎么把当前 tb 和下一个 tb 链接起来。

2. 2: 我们使用指令对代码生成计数，并且我们不开始执行这个 tb，因为指令计数器可能会中途置为 0. 在这种情况下，返回的指针是我们将要执行的 tb，并且调用者必须安排剩余指令的执行。

3. 3: 由于 CPU 的 exit_request 标志被置位，这通常意味着有中断需要处理。返回的指针就是我们注意到 exit request 时，将要执行的 tb。

如果最后两位表明 exit-via-index,CPU 状态正确同步；否则我们放弃执行 TB，并且调用者必须通过调用 CPU 的 synchronize_from_tb 方法，利用返回的 tb 指针，修复 CPU 状态。

我们上面知道 tcg_qemu_tb_exec 函数是指向生成的宿主机二进制代码，它是怎么从一个 tb 跳到另一个 tb 以及怎么知道返回值是多少呢? 其实 tb 跳

转有一个专有名词 tb block chaining，而返回值是在生成代码的过程中由 gen_tb_end 产生的。

3.1.5 Block Chaining

上面构建了 QEMU 翻译和执行客户机代码的整体流程。我们知道 QEMU 是以 Translation Block 为单位进行翻译和执行的，也就是说，每当 Cache 执行完一个 tb 之后，控制权交还到 QEMU ，这是做二进制翻译最基本的问题，就是翻译块之间的连接。QEMU 的方法是把 Cache 中的 tb 串联起来，在 tb 执行结束后，如果跳转目标正好在 Cache 中，那就把这两个 tb 串联起来。也叫做 block chaining。

block chaining 有两种做法，一是直接跳转，即修改 code cache 中分支指令的跳转目标，依据 host 不同有不同的 patch 方式；二是通过修改 TranslationBlock 的 tb_next 成员。如此我们重新审视一下 tb_find 函数，QEMU 在找到待执行的 tb 之后到底做了什么?

```
1  static inline TranslationBlock *tb_find(CPUState *cpu,
2                                          TranslationBlock *last_tb,
3                                          int tb_exit, uint32_t cf_mask)
4  {
5      if (tb->page_addr[1] != -1) {
6      //last_tb 是指上一个执行的 tb
7      //如果地址映射改变了，那直接跳转就失效了。因此如果 TB 跨越两个页面，
           第二个页面很可能失效，那直接跳转也就不安全了。所以不再跳转。
8          last_tb = NULL;
9      }
10
11     if (last_tb && !qemu_loglevel_mask(CPU_LOG_TB_NOCHAIN)) {
12         if (!acquired_tb_lock) {
13             tb_lock();
14             acquired_tb_lock = true;
15         }
16     //如果 last_tb 有值，意味着可以直接跳转。
17         if (!(tb->cflags & CF_INVALID)) {
18         //tb_exit 是由 tb 地址的最后两位获得的，这两位地址存储的是 tb 退
               出的原因，
19         //前一节最后描述过这两个 bit 位的意义。在此处是指示 block
               chaining 的方向 (0或1) 。
20             tb_add_jump(last_tb, tb_exit, tb);
21
22         }
23     }
24     if (acquired_tb_lock) {
```

```
25          tb_unlock();
26      }
27      return tb;
28
29  }
30
31  static inline void tb_add_jump(TranslationBlock *tb, int n,
32                                 TranslationBlock *tb_next)
33  {
34      assert(n < ARRAY_SIZE(tb->jmp_list_next));
35      if (tb->jmp_list_next[n]) {
36      //说明在我们获取锁之前，已经有另一个线程完成了这件事，直接返回。
37          return;
38      }
39
40      /* patch the native jump address */
41      tb_set_jmp_target(tb, n, (uintptr_t)tb_next->tc.ptr);
42      //建立跳转关系
43
44      /* add in TB jmp circular list */
45      tb->jmp_list_next[n] = tb_next->jmp_list_first;
46      tb_next->jmp_list_first = (uintptr_t)tb | n;
47      //将待执行的 tb 加入到循环链表中，n代表着有 tb 的哪一个分支跳转到
            tb_next
48
49  }
50
51  void tb_set_jmp_target(TranslationBlock *tb, int n, uintptr_t addr)
52  {
53      if (TCG_TARGET_HAS_direct_jump) {
54          uintptr_t offset = tb->jmp_target_arg[n];
55          uintptr_t tc_ptr = (uintptr_t)tb->tc.ptr;
56          tb_target_set_jmp_target(tc_ptr, tc_ptr + offset, addr);
57
58          //第一个参数是上一个 tb 对应的 host 地址，第二个参数是需要 patch
                的地址（当前 tb  开始的 host 地址加上当前地址相对于tb开始地址
                的偏移,jmp_target_arg虽然没有在之前赋值，但是后面我们发现，
                jmp_target_arg 被赋值给了tb_jmp_insn_offset，而后者在指令生成
                期间会在需要 patch 时会记录一下偏移地址)， 第三个参数是下一个
                tb 对应的 host 地址。
59      } else {
60
61          //间接跳转
62          tb->jmp_target_arg[n] = addr;
63      }
```

```
}
static inline void tb_target_set_jmp_target(uintptr_t tc_ptr,
                                            uintptr_t jmp_addr, uintptr_t
                                                addr)
{
    /* patch the branch destination */
    atomic_set((int32_t *)jmp_addr, addr - (jmp_addr + 4));
    //生成跳转目的地址和当前地址之间的偏移
    /* no need to flush icache explicitly */
}
```

可以从上述代码看到，如果体系结构不支持直接跳转，目的地址会被直接放到 jmp_target_arg[n] 之中，然后再间接跳转到这个地址。

事实上，到此处我们并没有解答上一节的问题，tcg_qemu_tb_exec 怎么知道返回值是多少呢？也就是上面分析的 tb_exit 是怎么得到的？

在 guest binary 到 IR 的过程中，gen_goto_tb 生成的 goto_tb 指令为 block chaining 做准备。

```
static inline void gen_jcc(DisasContext *s, int b,
                           target_ulong val, target_ulong next_eip)
{
    TCGLabel *l1, *l2;

    if (s->jmp_opt) {
        l1 = gen_new_label();
        gen_jcc1(s, b, l1);

        gen_goto_tb(s, 0, next_eip);
        //顺序执行下一条指令，第二个参数明显是指示返回值地址的最后两位为0
        gen_set_label(l1);
        gen_goto_tb(s, 1, val);
        //跳转执行另一条路径
    } else {
        l1 = gen_new_label();
        l2 = gen_new_label();
        gen_jcc1(s, b, l1);

        gen_jmp_im(next_eip);
        tcg_gen_br(l2);

        gen_set_label(l1);
        gen_jmp_im(val);
        gen_set_label(l2);
```

```
26          gen_eob(s);
27      }
28  }
29
30  static inline void gen_goto_tb(DisasContext *s, int tb_num, target_ulong
        eip)
31  {
32      target_ulong pc = s->cs_base + eip;
33      //此处PC指向客户机跳转的目的虚拟地址
34
35
36
37      if (use_goto_tb(s, pc))  {
38      //如果客户机跳转指令和跳转地址同属一个页，则做 direct block linking
39          /* jump to same page: we can use a direct jump */
40          tcg_gen_goto_tb(tb_num);
41          //生成做 block linking 的 IR。
42
43          gen_jmp_im(eip);
44          //更新 env 的 eip，使其指向此 tb 之后执行的指令的地址
45
46          tcg_gen_exit_tb((uintptr_t)s->base.tb + tb_num);
47          //生成回到 tcg_qemu_tb_exec 的 IR， tb_num 则是返回值的最后两位。
48          s->base.is_jmp = DISAS_NORETURN;
49      } else {
50      //
51          /* jump to another page */
52          gen_jmp_im(eip);
53          gen_jr(s, cpu_tmp0);
54      }
55  }
```

以上就追踪了生成 IR 时所做的操作，进一步跟踪宿主机代码生成就可以搞明白这个问题了。

```
1       case INDEX_op_goto_tb:
2           if (s->tb_jmp_insn_offset) {
3               /* direct jump method */
4               int gap;
5               /* jump displacement must be aligned for atomic patching;
6                * see if we need to add extra nops before jump
7                */
8               gap = tcg_pcrel_diff(s, QEMU_ALIGN_PTR_UP(s->code_ptr + 1, 4)
                    );
```

```
            if (gap != 1) {
                tcg_out_nopn(s, gap - 1);
            }
            //对齐操作

            tcg_out8(s, OPC_JMP_long); /* jmp im */
            //需要 patch 的地方

            s->tb_jmp_insn_offset[a0] = tcg_current_code_size(s);//记录下
                需要 patch 的地址，在生成 patch 时会用到。
            tcg_out32(s, 0);
            //jmp 的参数是0，所以接着执行 gen_jmp_im(eip)和
                tcg_gen_exit_tb生成的指令，退出执行，回到 qemu。


        } else {
            /* indirect jump method */
            tcg_out_modrm_offset(s, OPC_GRP5, EXT5_JMPN_Ev, -1,
                                 (intptr_t)(s->tb_jmp_target_addr + a0));
        }
        s->tb_jmp_reset_offset[a0] = tcg_current_code_size(s);
        break;
```

如果第一次执行该 tb,还没有发生 block chaining ,那么执行 exit_tb, 退出指令执行,并返回当前 tb 的地址,地址最后两位是标记退出原因。tb_add_jump(last_tb, tb_exit, tb) 负责对 tb 进行 block chaining。

3.1.6 Block Unchaining

某些情况需要 QEMU 从 code cache 中跳出来，如：host sigalarm，DMA，IO Thread，Single step，这时就需要 unlink tb。

3.1.7 中断和异常

我们在前面分析 cpu 主循环执行指令时看到，整体的运行结构如下：

```
cpu_thread{
    while(1){
        tcg_cpu_exec{
            cpu_exec{
                setsigjmp;//可作为异常返回点，如果跳到此处会直接进入异常
                    处理函数，cpu_handle_execption.
```

```
                while !cpu_handle_execption{
                    while !cpu_handle_interrupt{
                        cpu_loop_exec_tb{
                            cpu_tb_exec
                        }
                    }
                }
            }

        }
    }
}
}
```

cpu 退出执行的第一道判断是中断处理，如果中断能处理就直接处理，否则(如 reset，halt，init，debug，或者 exit_request) 返回 True，跳出一重循环，到 cpu_handle_execption 中进行异常处理。execption 的逻辑和 interrupt 一样，无法处理就跳出循环，发给上层循环处理。那么中断异常是怎么出现的? 1. 同步方式是通过生成指令时自动生成的。客户机指令翻译时，通过 gen_interrupt 生成 raise_interrupt 或者 raise_exception 即可跳转到 setsigjmp 的位置，进行异常处理，如各种软中断 int 指令，page fault 等; 2. 异步产生的中断。中断产生时，一般是由硬件主动调用 qemu_set_irq，这个函数负责调用 cpu_interrupt 对 cpu 的 interrput_request 进行置位。这是由中断源做的事情，CPU 也需要主动中断检查状态位 (毕竟是软件模拟的异步) 。qemu 的做法就是，在生成的每个 tb 之前插入一条指令，检查 icount_decr.u16.high 的值,无论 cpu_exit/cpu_interrupt/tcg_handle_interrupt 函数都会对该位进行置位，因此如果 high 被置位就表示需要退出 CPU 执行循环，响应异常事件。

(之前的版本的策略: 在 block chaining 机制之下，cpu 很少退出执行客户机代码，导致中断无法及时响应。因此需要定时器定时打断 CPU 执行，检查中断异常。) todo

精确异常:

当一条客户机指令发生异常时，此条指令之前的运算必须完成，之后的运算必须舍弃。当 qemu 在 code cache 中执行翻译过的指令发生异常时，qemu 必须维护异常产生前客户机的寄存器和内存内容.

对于寄存器来说，qemu 在每个可能发生异常的指令或者 helper 函数之前，会把 CPUState 中的大部分内容更新，少数未更新的内容会在客户机发生异常时重新计算。在 x86 体系结构中，pc 和条件码都是属于后者。在使用 tcg 的二进制

翻译时，一般都在基本块结束时更新客户机 pc。这就出现上面的问题。

对于内存来说，qemu 首先按照客户机的 memory store operation 的顺序翻译；其次，针对潜在发生异常的客户机指令，qemu 保留其相对于客户机 memory sotre operation 顺序。所以 qemu 不会对客户机指令进行乱序处理，这简化了精确异常的复杂度，但同时也丧失了潜在可能的优化。

前面说过，qemu 要找到发生异常的基本块对应开头的客户机 pc。从客户机 pc 重新开始翻译直到发生异常的地址，并把发生异常的客户机 pc 存入 CPUState，这是因为 qemu 不会执行完一条客户机指令后就更新客户机 pc。事实上，我们并不会重新翻译指令，而是在翻译指令的过程中记录下来相关信息。发生异常的时候利用这些信息迅速找到发生异常的客户机 pc 指针。在分析 TCG 指令的时候我们描述过，但是并没有理解它的作用。

```
    case INDEX_op_insn_start:
    if (num_insns >= 0) {
        s->gen_insn_end_off[num_insns] = tcg_current_code_size(s);
    }
    num_insns++;
    for (i = 0; i < TARGET_INSN_START_WORDS; ++i) {
        target_ulong a;

        a = op->args[i];
        s->gen_insn_data[num_insns][i] = a;
    }
    break;
```

每条客户机指令被翻译为 IR 时，都会被插入一条指令 insn_start，这条指令在生成宿主机二进制时，负责记录上述信息。首先，gen_insn_end_off 记录这条指令对应的 code cache 的偏移，gen_insn_data 记录 insn_start 指令的操作码和参数，而 insn_start 的参数正是当前指令的客户机 pc。这两个信息就能恢复出 CPU 的 pc 指针。

我们仍然以 page fault 为例，上面分析 page fault 时只是描述 softmmu 工作机理的，这里我们看看如果真的发生缺页中断该怎么办。

```
void tlb_fill(CPUState *cs, target_ulong addr, MMUAccessType access_type,
              int mmu_idx, uintptr_t retaddr)
{
    int ret;

    ret = x86_cpu_handle_mmu_fault(cs, addr, access_type, mmu_idx);
    if (ret) {
        X86CPU *cpu = X86_CPU(cs);
```

```
        CPUX86State *env = &cpu->env;

        raise_exception_err_ra(env, cs->exception_index, env->error_code,
            retaddr);
    }
}
```

如果确实发生缺页中断，x86_cpu_handle_mmu_fault 返回 1，此时进入缺页中断处理流程。调用 raise_exception_err_ra 函数，cpu 主动触发异常，进入 raise_interrupt2 函数。

```
static void QEMU_NORETURN raise_interrupt2(CPUX86State *env, int intno,
                                           int is_int, int error_code,
                                           int next_eip_addend,
                                           uintptr_t retaddr)
{
    CPUState *cs = CPU(x86_env_get_cpu(env));

    if (!is_int) {
        cpu_svm_check_intercept_param(env, SVM_EXIT_EXCP_BASE + intno,
                                      error_code, retaddr);
        intno = check_exception(env, intno, &error_code, retaddr);
    } else {
        cpu_svm_check_intercept_param(env, SVM_EXIT_SWINT, 0, retaddr);
    }

    cs->exception_index = intno;
    env->error_code = error_code;
    env->exception_is_int = is_int;
    env->exception_next_eip = env->eip + next_eip_addend;
    cpu_loop_exit_restore(cs, retaddr);
}

void cpu_loop_exit_restore(CPUState *cpu, uintptr_t pc)
{
    if (pc) {
        cpu_restore_state(cpu, pc);
    }
    siglongjmp(cpu->jmp_env, 1);
    //退出 CPU exec 循环，跳转到 cpu exec 之前，重新进入 cpu exec 循环，
        处理中断异常。
}
```

在这个函数里，检查异常嵌套，设置 CPU 异常状态，最后进入 CPU 恢复函数，因为我们现在只知道发生异常的宿主机虚拟地址，而不知道它所对应的客户机虚拟地址，现在要把 CPU 状态恢复到发生异常的那条指令的状态。

```
bool cpu_restore_state(CPUState *cpu, uintptr_t host_pc)
{
    TranslationBlock *tb;
    bool r = false;
    uintptr_t check_offset;

    check_offset = host_pc - (uintptr_t) tcg_init_ctx.code_gen_buffer;
    //计算发生异常的指令地址和当前翻译代码的开始地址之间偏移


    if (check_offset < tcg_init_ctx.code_gen_buffer_size) {
        tb_lock();
        tb = tb_find_pc(host_pc);
        if (tb) {
        //找到它所属的 tb 之后，进行 cpu 恢复
            cpu_restore_state_from_tb(cpu, tb, host_pc);
            if (tb->cflags & CF_NOCACHE) {
                /* one-shot translation, invalidate it immediately */
                //如果不允许 cache，那么将 tb 所有相关的页面都删除
                tb_phys_invalidate(tb, -1);
                tb_remove(tb);
            }
            r = true;
        }
        tb_unlock();
    }

    return r;
}
```

cpu_restore_state 参数是 CPUState 和 host_pc 两个，前者是 CPU 状态代表着 CPU，host_pc 是发生异常的宿主机虚拟地址。

```
static int cpu_restore_state_from_tb(CPUState *cpu, TranslationBlock *tb,
                                     uintptr_t searched_pc)
{
    target_ulong data[TARGET_INSN_START_WORDS] = { tb->pc };
    uintptr_t host_pc = (uintptr_t)tb->tc.ptr;
    CPUArchState *env = cpu->env_ptr;
    uint8_t *p = tb->tc.ptr + tb->tc.size;
```

```
    int i, j, num_insns = tb->icount;
    searched_pc -= GETPC_ADJ;

    if (searched_pc < host_pc) {
        return -1;
    }

    /* Reconstruct the stored insn data while looking for the point at
       which the end of the insn exceeds the searched_pc.  */
       //此处就是利用之前存储的信息找到发生异常的指令pc。
       //由于信息存储时进行了编码因此使用 decode_sleb128进行解码。
    for (i = 0; i < num_insns; ++i) {
        for (j = 0; j < TARGET_INSN_START_WORDS; ++j) {
            data[j] += decode_sleb128(&p);
        }
        host_pc += decode_sleb128(&p);
        if (host_pc > searched_pc) {
        //找到 pc，跳转到成功。
            goto found;
        }
    }
    return -1;

 found:
    if (tb->cflags & CF_USE_ICOUNT) {
        assert(use_icount);
        /* Reset the cycle counter to the start of the block.  */
        cpu->icount_decr.u16.low += num_insns;
        /* Clear the IO flag.  */
        cpu->can_do_io = 0;
    }
    cpu->icount_decr.u16.low -= i;
    restore_state_to_opc(env, tb, data);
    //data 中存储发生异常的客户机指令的 pc 值和条件码操作。
    return 0;
}
```

找到发生异常的客户机指令之后，进入 cpu_handle_excpetion 处理异常。

3.1.8 条件码生成

x86 CPU 在每条指令结尾都会更新条件码。为了提高性能，必须有良好的 CPU 条件码模拟 (eflags register on x86)。QEMU 使用惰性条件码估值：它只保存

某个指令中的源操作数（CC_SRC），目标操作数（CC_DST）和操作类型（CC_OP），而不是在每条 x86 指令执行之后计算条件码。对于 32 位的加法计算，例如 R=A+B，我们可以得到如下表达式：

CC_SRC=A

CC_DST=R

CC_OP=CC_OP_ADDL

由于可以通过存放在 CC_OP 中的常数知道有一个 32 位的加法，我们可以通过 CC_SRC 和 CC_DST 恢复 A，B 和 R。然后，如果下一条指令需要，所有相关的条件码，例如零结果（ZF），非负结果（SF），进位（CF）或者溢出（OF）都可以很容易获得。

翻译指令时，在每条可能会导致条件码改变的指令后面都追加一个更新 cc_src，cc_dst，cc_op 的指令，以保证当需要计算条件码时可以及时得到。

3.1.9 代码自修改和翻译代码失效

在大多数 CPU 上，自修改代码很容易处理。通过执行一条特殊的缓存废弃指令，可以发出信号指示出该代码已被修改。这足以废弃相应的翻译代码。然而，在一些 CPU 例如 x86 上，当代码被修改时，应用程序不能发出信号以废弃指令缓存。所以，自修改代码是一个特殊的挑战。

当生成了一个 TB 的翻译代码时，如果相应的宿主机页不是只读的，那么它将会被设置为写保护。如果有一个针对该页的写访问产生，QEMU 会废弃该页中所有的翻译代码，并使该页重置为可写。

通过维护一个包含给定页中所有 TB 的链表，可以有效完成正翻译代码的废弃任务。除此之外，还有其他链表用来取消直接 block 链。

当使用软件 MMU 时，代码废弃将更加高效：如果某个代码页由于写访问而频繁做废弃代码操作，将会创建一个展示该页内部代码的 bitmap。每次往该页的存储操作都将检查 bitmap，以知晓该页的代码是否需要废弃。这避免了该页仅作数据修改时就进行代码废弃操作。

QEMU 访存方式（tlb，softmmu，页表）QEMU 设备模拟 QEMU 原子指令和内存序问题