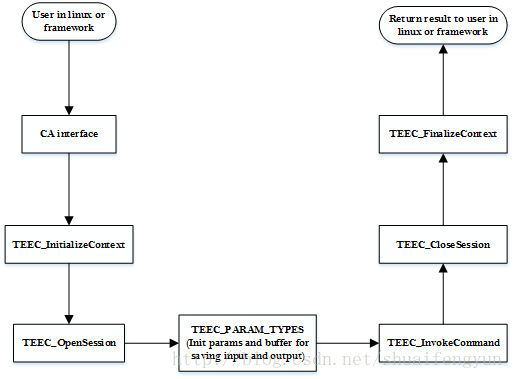
OP-TEE之REE切换进TEE的过程（自顶向下）

1. **CA执行流程（应用层）**

REE侧CA的完整接口调用过程如下图所示：



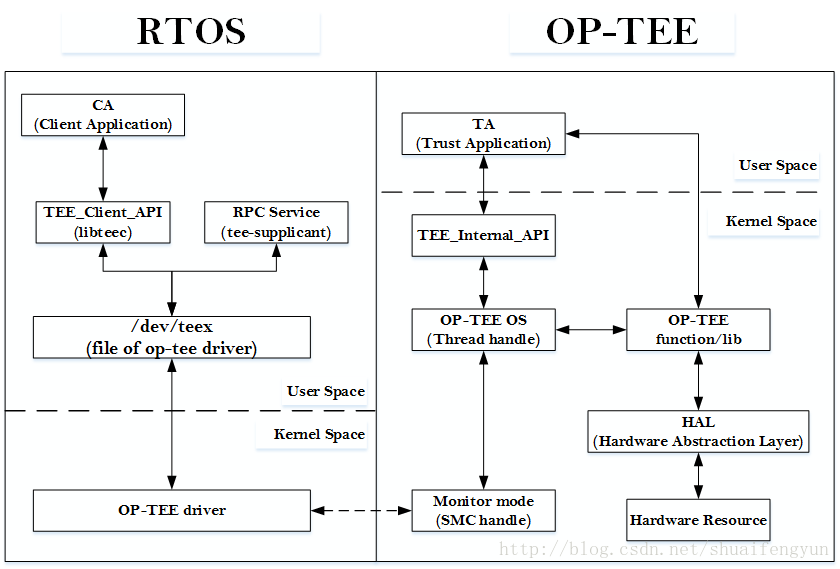
CA接口调用流程

该过程包括以下5个步骤：

1. 调用TEEC\_InitializeContext函数打开op-tee驱动文件，获取到操作句柄并存放到TEE\_Context类型的变量中。
2. 调用TEEC\_OpenSession函数，通过获取到的TEE\_Context类型的变量创建一个特定CA与特定TA之间进行通信的通道，如果TA image被存放在file system中，那个在创建session的时候，OP-TEE OS端还会将TA image从file system中加载到OP-TEE。
3. 初始化TEEC\_Operation类型的变量，并根据实际需要借助TEEC\_PARAM\_TYPES宏来设定TEEC\_Operation类型变量中paramTypes成员的值，该值规定传递到OP-TEE中的最多4个变量缓存或者是数据的作用（作为输入还是输出）。并且还要根据paramTypes的值设定对应的params[x]成员的值或者是指向的地址以及缓存的长度。
4. 使用已经创建好的session，TA与CA端规定的command ID以及配置好的TEEC\_Operation类型变量作为参数调用TEEC\_InvokeCommand函数来真正发起请求。 调用TEEC\_InvokeCommand成功之后，剩下的事情就有OP-TEE和TA进行处理并将结果和相关的数据通过TEEC\_Operation类型变量中的params成员返回给CA。
5. 注销session和释放掉context，这两个操作一次通过调用TEEC\_CloseSession函数和TEEC\_FinalizeContext函数来实现。

**二、CA与TA调用过程**

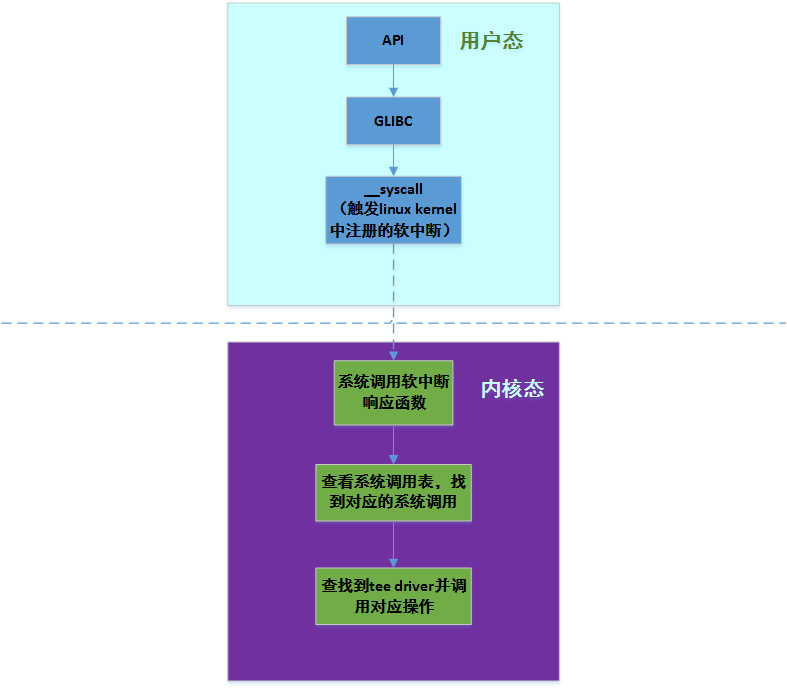
CA与TA调用过程如下图所示：



CA与TA调用过程

一次完整的功能调用一般都是起源于CA，TA做具体功能实现并返回数据到CA，而整个过程需要经过OP-TEE的client端接口，OP-TEE在Linux kernel端的驱动，Monitor模式下的SMC处理，OP-TEE OS的thread处理，OP-TEE中的TA程序运行，OP-TEE端底层库或者硬件资源支持等几个阶段。当TA执行完具体请求之后会按照原路径将得到的数据返回给CA。

其中章节一所示的CA接口即由libteec库提供。如上图所示，当应用层通过调用CA接口后，系统最终会陷入内核态，整个过程可以通过下图进行更清晰的展示。

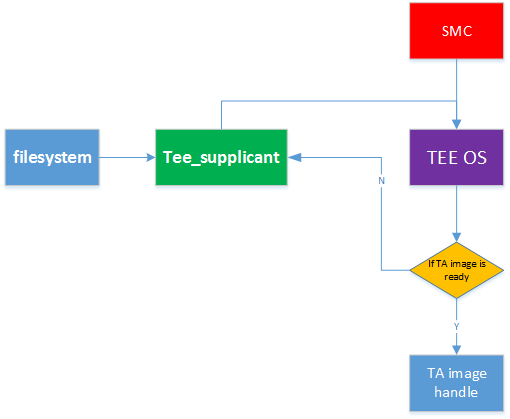


CA接口调用系统流程

当应用层调用CA接口之后会触发system call操作，系统调用会将linux陷入内核态，此时系统处于kernel space，然后根据传入的参数，找到对应的TEE driver并执行对应的操作。

在TEE的driver中，最终会调用SMC这条汇编指令来实现与OP-TEE的沟通，当调用SMC汇编指令之后，会触发SMC软中断，该软中断的处理在Cortex的Monitor态中进行，随着ARM提供了ATF（ARM trust firmware），SMC的软中断具体处理函数在ATF中被实现，该软中断会根据发送指令是属于安全侧请求还是非安全侧的请求来控制是将cortex切换到secure world态还是non-secure world态。该切换动作在ATF的SMC中断处理函数中被实现。

当SMC中断处理函数完成了将cortex的状态切换到secure world态以及相关参数的拷贝动作之后，TEE OS将接管剩下的操作。TEE OS首先将会获取从CA端传递过来的数据，然后解析出数据中写入的TA的UUID，然后查找对应的TA image是否被挂载到了 TEE OS中。如果没有被挂载到TEE OS中，TEE OS将会与常驻在linux中的tee\_supplicant服务进程通信，从文件系统中获取到TA image文件，并传递给TEE OS，然后加载该TA image。处理完整之后，TEE OS会切换到TEE userspace态，并将CA传递过来的其他参数传给具体的TA process，TA process获取到参数后，首先需要解除出参数中的commond ID值，根据具体的command ID值来做具体的操作。上述大致流程如下图所示：



tee\_supplicant加载TA image流程图

三、libteec介绍

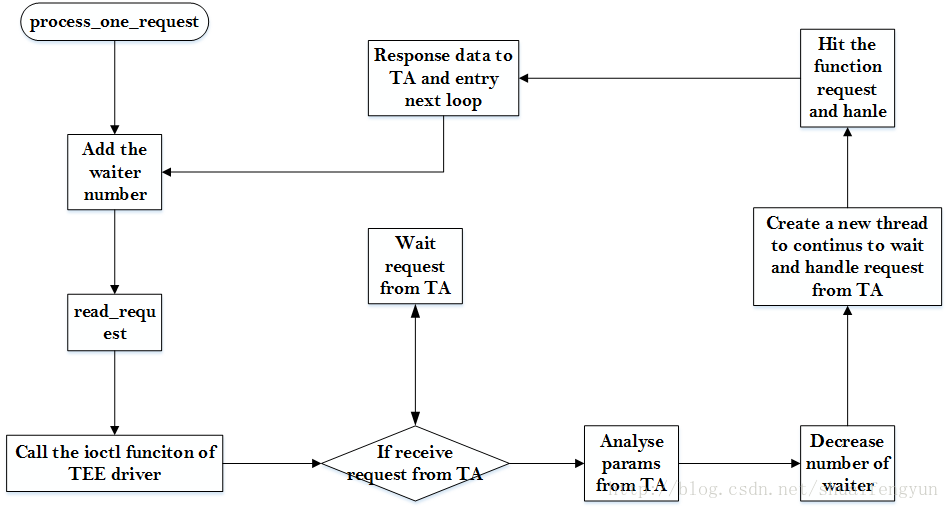
libteec库是OP-TEE提供给用户在linux userspace层面调用的接口实现。在OP-TEE中，libteec代码的具体实现存放在optee\_client/libteec目录下，OP-TEE提供给linux端使用的接口源代码的实现存放在optee\_client/libteec/src/tee\_client\_api。在OP-TEE中libteec提供给上层用户使用的API一共有10个，它们将在下面进行具体介绍。

1. TEEC\_Result TEEC\_InitializeContext，作用：初始化一个TEEC\_Context变量，该变量用于CA和TEE之间建立联系。其中参数name是用来定义TEE的身份，如果该参数为NULL，则CA将会选择默认的TEE方案来建立联系。
2. void TEEC\_FinalizeContex，作用：释放一个已经被初始化过的类型为TEEC\_Context变量，关闭CA与TEE之间的连接。在调用该函数之前必须确保打开的session已经被关闭了。
3. TEEC\_Result TEEC\_OpenSession，作用：打开一个CA与对应TA之间的一个session，该session用于该CA与对应TA之间的联系，该CA需要连接的TA是由UUID指定的。session具有不同的打开和连接方式，根据不同的打开和连接方式CA可以在执行打开session的时候传递数据给TA，以便TA对打开操作做出权限检查。各种打开方式说明如下：
4. TEEC\_CloseSession，作用：关闭已经被初始化的CA与对应TA之间的session，在调用该函数之前需要保证所有的command已经执行完毕。如果session为NULL,则不执行任何操作。
5. TEEC\_InvokeCommand，作用：通过cmd\_id和打开的session，来通知session对应的TA执行cmd\_id指定的操作。
6. TEEC\_RequestCancellation，作用：　取消某个CA与TA之间的操作，该接口只能由除执行TEEC\_OpenSession和TEEC\_InvokeCommand的thread之外的其他thread进行调用，而在TA端或者TEE OS可以选着并不响应该请求。只有当operation中的started域被设置成0之后，该操作方可有效。
7. TEEC\_RegisterShareMemory，作用：注册一块在CA端的内存作为CA与TA之间的共享内存。
8. TEEC\_RegisterShareMemoryFileDescriptor，作用：　注册一块在CA与TA之间的共享文件。
9. TEEC\_AllocateSharedMemory，作用：分配一块共享内存，共享内存是由OP-TEE中分配的。
10. TEEC\_ReleaseSharedMemory，作用：释放已经被分配或者是注册过的共享内存。根据代码可知，该函数可以释放（7）、（8）、（9）中申请的共享内存。

**四、tee-supplicant介绍**

tee-supplicant作为常驻在REE侧的服务进程，主要作用是接受来自TEE侧的请求，向TA提供特定的服务。具体而言，tee\_supplicant使OP-TEE能够通过tee\_supplicant来访问REE端文件系统中的资源，例如加载存放在文件系统中的TA镜像到TEE中，对REE端数据库的操作，对EMMC中RPMB分区的操作，提供socket通信等。其源代码optee\_client/tee-supplicant目录中。编译之后会生成一个名字为tee\_supplicant的可执行文件，该可执行文件在REE启动的时候会作为一个后台程序被自动启动，而且常驻于系统中。

tee\_supplicant启动之后最终会进入一个loop循环，调用process\_one\_request函数来监控，接收，处理，回复OP-TEE的请求。整个处理过程如下图所示：



tee\_supplicant处理流程图

其中，tee\_supplicant通过read\_request来接收来自TA端的请求。该函数会block在驱动中的ioctl函数上，直到接收来自TA的请求。tee\_supplicant使用find\_params函数来解析来自TA的请求，并解析出对应的请求func ID。，tee\_supplicant将会根据func ID来执行具体的请求操作。当TA请求的func ID为RPC\_CMD\_LOAD\_TA时，tee\_supplicant将会到文件系统中将TA镜像的内容读取到共享内存中；当func ID为RPC\_CMD\_FS时，tee\_supplicant将会根据TA请求调用tee\_supp\_fs\_process函数来完成对文件系统的具体操作，包括常规的文件和目录的打开，关闭，读取，写入，重命名，删除，等等；当func ID为RPC\_CMD\_RPMB时，tee\_supplicant将会根据TA请求调用process\_rpmb函数来完成对EMMC中rmpb分区的操作。在EMMC中的rpmb分区，在读写过程中会执行验签和加解密的操作。当func ID为RPC\_CMD\_SHM\_ALLOC时，tee\_supplicant将会根据TA请求调用process\_alloc函数来分配TA与tee\_supplicant之间的共享内存。　当func ID为RPC\_CMD\_SHM\_FREE时，tee\_supplicant将会根据TA请求调用process\_free函数来释放TA与tee\_supplicant之间的共享内存。当func ID为RPC\_CMD\_GPROF时，tee\_supplicant将会根据TA请求调用gprof\_process函数来将某个特定的TA执行效率信息记录到文件系统中。当func ID为OPTEE\_MSG\_RPC\_CMD\_SOCKET时，tee\_supplicant将会根据TA请求调用tee\_socket\_process函数来完成网络socket的相关操作，包括socket的建立，发送，接收和ioctl操作。

**五、libteec和tee-supplicant的驱动实现**

OP-TEE驱动主要作用是REE与TEE端进行数据交互的桥梁作用。tee\_supplicant和libteec调用接口之后几乎都会首先通过系统调用陷入到kernel space，然后kernel根据传递的参数找到OP-TEE驱动，并命中驱动的operation结构体中的具体处理函数来完成实际的操作，对于OP-TEE驱动，一般都会触发SMC调用，并带参数进入到ARM cortex的monitor模式，在monitor模式中对执行normal world和secure world的切换，待状态切换完成之后，会将驱动端带入的参数传递给OP-TEE中的thread进行进一步的处理。

从OP-TEE的驱动的挂载过程来看，OP-TEE驱动会分别针对libteec和tee\_supplicant建立不同的设备/dev/tee0和/dev/teepriv0。并且为两个设备中的des执行各自独有的operation，并建立类似消息队列来存放normal world与secure world之间的请求，这样libteec和tee\_supplicant使用OP-TEE驱动的时候就能做到相对的独立。secure world与OP-TEE驱动之间使用共享内存进行数据交互。用于作为共享内存的物理内存块在OP-TEE启动的时做MMU初始化时需要被reserve出来，在OP-TEE挂载过程中需要将该内存块映射到系统内存中。

驱动挂载完成之后，CA程序通过调用libteec中的接口调用到OP-TEE驱动来穿透到secure world中调用对应的TA程序。OP-TEE的驱动挂载后会在/dev目录下分别创建两个设备接待，分别为/dev/tee0和/dev/teepriv，libteec和tee\_supplicant分别对上述两个文件进行相关操作就能够穿透到secure world中。

如前文所述，libteec提供给上层使用的接口总共有十个，这十个接口通过系统调用最终会调用到驱动中。在接口libteec和tee\_supplicant中调用Open函数的时候，在驱动中就调用到file\_operations结构体变量tee\_fops中的Open成员。同理在libteec和tee\_supplicant接口中调用ioctl函数，则在驱动中最终会调用tee\_fops中的ioctl函数。

值得注意的是在libteec中获取OP-TEE版本信息，创建和关闭session，调用TA，分配和注册共享内存和fd以及释放共享内存，接收来自OP-TEE的请求以及回复数据给OP-TEE都是通过ioctl来完成的。在libteec和tee\_supplicant中通过带上对应的参数调用ioctl函数来实现对应的操作需求，最终会调用到OP-TEE驱动中file\_operation结构体变量tee\_fops变量中的tee\_ioctl函数。

另外从tee\_supplicant处理来自TEE侧的请求来看主要是有三个点。

第一是驱动在触发smc操作之后会进入到loop循环中，根据secure world中的返回值来判定该返回时来自TEE的RPC请求还是最终处理结果，如果是RPC请求，也就是需要驱动或者tee\_supplicant执行其他操作，驱动将RPC请求会保存到驱动的请求消息队列中，然后block住等待请求处理结果。

第二是在tee\_supplicant作为一个常驻进程存在于REE中，它会不停的尝试从驱动的请求消息队列中获取到来自TEE侧的请求。如果请求消息队列中并没有请求则会block住，直到拿到了请求才返回。拿到请求之后会对请求进行解析，然后根据func执行具体的操作。

第三是在tee\_supplicant处理完来自TEE的请求后，会调用send操作将处理结果存放到该消息队列的参数区域，并使用complete函数通知驱动该请求已经被处理完毕。驱动block住的地方可以继续往下执行，调用smc操作将结果返回给TEE侧。

**六、TEE侧的处理**

（1）open session的处理

OP-TEE支持动态TA和静态TA。即可以将TA镜像与OP-TEE的镜像编译在同一image中，只是静态TA镜像会存放在OP-TEE镜像的特定分区中。在OP-TEE启动的时候会被加载到属性为MEM\_AREA\_TA\_RAM的安全内存中。而动态TA的方式则是将TA镜像文件保存到文件系统中，在打开session的时候再通过RPC请求加载到OP-TEE的安全内存中。open session在OP-TEE中的操作都是根据UUID值找到对应的TA镜像，然后读取TA image head部分的数据，并将相关数据保存到tee\_ta\_ctx结构体变量中，然后将填充好的tee\_ta\_ctx结构体变量保存到tee\_ctxes链表中，以便后期CA执行invoke操作来调用TA中的command的时候可以通过查找tee\_ctxes链表来获取对应的session，然后根据session的内容进入到TA中根据command ID执行特定的command操作。

（2）invoke command的处理

在REE侧的CA执行open session成功之后，CA就可以使用获取到的session和command ID调用TEEC\_InvokeCommand接口来实现让TA指定特定的command的操作。在REE侧调用TEEC\_InvokeCommand接口之后，该函数会将调用时带入的session变量，command ID，以及需要传递给TA的参数信息通过ioctl的系统调用发送到OP-TEE的驱动中，驱动最终会调用optee\_invoke\_func函数会将需要传递给TA的参数信息保存在共享内存中，并触发smc操作，切换到monitor模式进行secure world端的处理，invoke command操作的smc请求最终会被作为标准smc（std smc）进行解析并建立一个专门的thread进入thread\_std\_smc\_entry函数执行，线程运行到tee\_entry\_std函数时会对smc请求做出判定进入invoke command分支。

线程会调用session中注册的ops成员中的enter\_invoke\_cmd指定的函数来处理invoke command操作，而enter\_invoke\_cmd指向user\_ta\_enter\_invoke\_cmd函数，该函数会调用user\_ta\_enter执行，在user\_ta\_enter函数中加载userspace的上下文然后调用\_\_thread\_enter\_user\_mode函数将该线程切换到userspace继续执行。在OP-TEE中定义了两个entry\_invoke\_command函数，一个是kernel space的函数，一个是user space的函数。当程序运行到user space后则会调用user space的entry\_invoke\_command函数执行invoke command的操作。调用到user space的entry\_invoke\_command函数的时候，其实线程已经进入到了TA image上下文中运行了，所以当调用TA\_InvokeCommandEntryPoint函数的时候就会去执行TA image中定义的TA\_InvokeCommandEntryPoint函数，而该函数具体会执行什么操作就由特定的TA决定了，一般做法是根据command id的值去执行特定的操作

**七、secure world和non-secure world切换的物理基础**

ARM支持的trustzone技术后提供了ARM cortex的虚拟化基础。支持trustzone技术的cortex具有八种模式和两个状态：secure world态和non-secure world态。而mon模式就起到进行secure world与non-secure world之间进行切换的桥梁作用。

在持trustzone技术的时候，ARM在AXI系统总线上增加了一个NS位（详细情况请查阅ARM给出的trustzone白皮书），而NS位就是用来标记当前的数据，指令时属于secure world态还是non-secure world态，NS位会被保存到scr寄存器的第0位。当NS=1时，处理器处于non-secure world态，当NS=0时，处理器处于secure world态。

当需要让处理器进入到monitor模式的时候，ARM要求执行smc指令来实现。如果该汇编指令执行成功，则处理器就切换到了monitor模式下并且更新monitor模式下的重要寄存器。该操作与ARM进入到IRQ, ABT等模式的操作一样，采取的是产生异常来进行模式的切换。当系处理器进入到monitor之后，处理器就回去找寻该模式下的异常处理向量表的位置，进而找到smc操作的异常处理函数。

当smc异常是在non-secure world态中触发时，则SCR寄存器中的NS位必定为1，在进入中断处理函数后，SCR寄存器的NS位会被置为0以完成从non-secure world态到secure world态的切换。non-secure world态中的smc请求被TEE处理完成之后，SCR寄存器的NS位会被重置为1以完成secure world态切换到non-secure world态。

参考文献：<https://blog.csdn.net/shuaifengyun/article/details/73716352>