linux 的 initcall 机制

linux 的 initcall 机制 (针对编译进内核的驱动).

initcall 机制的由来.

我们都知道，linux 对驱动程序提供静态编译进内核和动态加载两种方式，当我们试图将一个驱动程序编译进内核时，开发者通常提供一个 xxx_init() 函数接口以启动这个驱动程序同时提供某些服务。

那么，根据常识来说，这个 xxx_init() 函数肯定是要在系统启动的某个时候被调用，才能启动这个驱动程序。

最简单直观地做法就是：开发者试图添加一个驱动程序时，在内核启动 init 程序的某个地方直接添加调用自己驱动程序的 xxx_init() 函数，在内核启动时自然会调用到这个程序。

但是，回头一想，这种做法在单人开发的小系统中或许可以，但是在 linux 中，如果驱动程序是这么个添加法，那就是一场灾难，这个道理我想不用我多说。

不难想到另一种方式，就是集中提供一个地方，如果你要添加你的驱动程序，你就将你的初始化函数

在这个地方进行添加，在内核启动的时候统一扫描这个地方，再执行这一部分的所有被添加的驱动程序。

那到底怎么添加呢？直接在 C 文件中作一个列表，在里面添加初始化函数？我想随着驱动程序数量的增加，这个列表会让人头昏眼花。

当然，对于 linus 大神而言，这些都不是事，linux 的做法是：

底层实现上，在内核镜像文件中，自定义一个段，这个段里面专门用来存放这些初始化函数的地址，内核启动时，只需要在这个段地址处取出函数指针，一个个执行即可。

对上层而言，linux 内核提供 xxx_init(init_func) 宏定义接口, 驱动开发者只需要将驱动程序的 init_func 使用 xxx_init() 来修饰，这个函数就被自动添加到了上述的段中，开发者完全不需要关心实现细节。
对于各种各样的驱动而言，可能存在一定的依赖关系，需要遵循先后顺序来进行初始化，考虑到这个，linux 也对这一部分做了分级处理。

initcall 的源码.

在平台对应的 init.h 文件中，可以找到 xxx_initcall 的定义：

```
  /*Only for built-in code, not modules.*/
#define early_initcall(fn)              __define_initca

#define pure_initcall(fn)               __define_initca
#define core_initcall(fn)               __define_initca
```

```
#define core_initcall_sync(fn)          __define_initca
#define postcore_initcall(fn)           __define_initca
#define postcore_initcall_sync(fn)      __define_initca
#define arch_initcall(fn)               __define_initca
#define arch_initcall_sync(fn)          __define_initca
#define subsys_initcall(fn)             __define_initca
#define subsys_initcall_sync(fn)        __define_initca
#define fs_initcall(fn)                 __define_initca
#define fs_initcall_sync(fn)            __define_initca
#define rootfs_initcall(fn)             __define_initca
#define device_initcall(fn)             __define_initca
#define device_initcall_sync(fn)        __define_initca
#define late_initcall(fn)               __define_initca
#define late_initcall_sync(fn)          __define_initca
```

xxx_init_call(fn) 的原型其实是__define_initcall(fn, n)，n 是一个数字或者是数字 + s，这个数字代表这个 fn 执行的优先级，数字越小，优先级越高，带 s 的 fn 优先级低于不带 s 的 fn 优先级。
继续跟踪代码，看看__define_initcall(fn,n):

```
  #define __define_initcall(fn, id) \
static initcall_t __initcall_##fn##id __used \
__attribute__((__section__(".initcall" #id ".init")))
```

值得注意的是，_attribute_() 是 gnu C 中的扩展语法，它可以用来实现很多灵活的定义行为，这里不细究。

attribute((_section_(".initcall" #id ".init"))) 表示编译时将目标符号放置在括号指定的段中。

而 #在宏定义中的作用是将目标字符串化，## 在宏定义中的作用是符号连接，将多个符号连接成一个符号，并不将其字符串化。

__used 是一个宏定义，

```
#define  __used  __attribute__((__used__))
```

使用前提是在编译器编译过程中，如果定义的符号没有被引用，编译器就会对其进行优化，不保留这个符号，而__attribute__((*_used_*)) 的作用是告诉编译器这个静态符号在编译的时候即使没有使用到也要保留这个符号。

为了更方便地理解，我们拿举个例子来说明，开发者声明了这样一个函数：pure_initcall(test_init);

所以 pure_initcall(test_init) 的解读就是：

```
首先宏展开成：__define_initcall(test_init, 0)

然后接着展开：static initcall_t __initcall_test_init0

同时声明__initcall_test_init0这个变量即使没被引用也保留符
```

需要注意的是，根据官方注释可以看到 early_initcall(fn) 只针对内置的核心代码，不能描述模块。

xxx_initcall 修饰函数的调用.

既然我们知道了 xxx_initcall 是怎么定义而且目标函数的放置位置，那么使用 xxx_initcall() 修饰的函数是怎么被调用的呢？

我们就从内核 C 函数起始部分也就是 start_kernel 开始往下挖, 这里的调用顺序为：

```
start_kernel
      -> rest_init();
            -> kernel_thread(kernel_init, NULL, CLO
                  -> kernel_init()
                        -> kernel_init_free
                              -> do_basic
                                    ->
```

这个 do_initcalls() 就是我们需要寻找的函数了，在这个函数中执行所有使用 xxx_initcall() 声明的函数，接下来我们再来看看它是怎么执行的：

```
static initcall_t *initcall_levels[] __initdata =
      __initcall0_start,
      __initcall1_start,
      __initcall2_start,
      __initcall3_start,
      __initcall4_start,
      __initcall5_start,
      __initcall6_start,
      __initcall7_start,
      __initcall_end,
};

int __init_or_module do_one_initcall(initcall_t fn)
{
      ...
      if (initcall_debug)
            ret = do_one_initcall_debug(fn);
      else
            ret = fn();
      ...
      return ret;
}

static void __init do_initcall_level(int level)
{
      initcall_t *fn;
      ...
      for (fn = initcall_levels[level]; fn < init
            do_one_initcall(*fn);
}

static void __init do_initcalls(void)
{
```

```
        int level;
        for (level = 0; level < ARRAY_SIZE(initcall
                do_initcall_level(level);
}
```

在上述代码中，定义了一个静态的 initcall_levels 数组，这是一个指针数组，数组的每个元素都是一个指针.

do_initcalls() 循环调用 do_initcall_level(level)，level 就是 initcall 的优先级数字，由 for 循环的终止条件 ARRAY_SIZE(initcall_levels) - 1 可知，总共会调用 do_initcall_level(0)~do_initcall_level(7)，一共七次。

而 do_initcall_level(level) 中则会遍历 initcall_levels[level] 中的每个函数指针，initcall_levels[level] 实际上是对应的 __initcall##level##_start 指针变量，然后依次取出 __initcall##level##_start 指向地址存储的每个函数指针，并调用 do_one_initcall(*fn)，实际上就是执行当前函数。

可以猜到的是，这个__initcall##level##*start 所存储的函数指针就是开发者用 xxx_initcall() 宏添加的函数，对应 ".initcall*#*#level##.init" 段。

do_one_initcall(*fn) 的执行：判断 initcall_debug 的值，如果为真，则调用 do_one_initcall_debug(fn); 如果为假，则直接调用 fn。事实上，调用 do_one_initcall_debug(fn) 只是在调用 fn 的基础上

添加一些额外的打印信息，可以直接看成是调用 fn。

那么，在 initcall 源码部分有提到，在开发者添加 xxx_initcall(fn) 时，事实上是将 fn 放置到了 ".initcall##level##.init" 的段中，但是在 do_initcall() 的源码部分，却是从 initcall_levels level 取出，initcall_levels[level] 是怎么关联到 ".initcall##level##.init" 段的呢？

答案在 vmlinux.lds.h 中：

```
  #define INIT_CALLS_LEVEL(level)
        VMLINUX_SYMBOL(__initcall##level##_start) = .;
        KEEP(*(.initcall##level##.init))
        KEEP(*(.initcall##level##s.init))

#define INIT_CALLS
        VMLINUX_SYMBOL(__initcall_start) = .;
        KEEP(*(.initcallearly.init))
        INIT_CALLS_LEVEL(0)
        INIT_CALLS_LEVEL(1)
        INIT_CALLS_LEVEL(2)
        INIT_CALLS_LEVEL(3)
        INIT_CALLS_LEVEL(4)
        INIT_CALLS_LEVEL(5)
        INIT_CALLS_LEVEL(rootfs)
        INIT_CALLS_LEVEL(6)
        INIT_CALLS_LEVEL(7)
        VMLINUX_SYMBOL(__initcall_end) = .;
```

在这里首先定义了__initcall_start，将其关联到 ".initcallearly.init" 段。

然后对每个 level 定义了 INIT_CALLS_LEVEL(level)，将

INIT_CALLS_LEVEL(level) 展开之后的结果是定义 __initcall##level##_start，并将 __initcall##level##_start 关联到 ".initcall##level##.init" 段和 ".initcall##level##s.init" 段。

到这里，__initcall##level##_start 和 ".initcall##level##.init" 段的对应就比较清晰了，所以，从 initcall_levels[level] 部分一个个取出函数指针并执行函数就是执行 xxx_init_call() 定义的函数。

总结.

便于理解，我们需要一个示例来梳理整个流程，假设我是一个驱动开发者，开发一个名为 beagle 的驱动，在系统启动时需要调用 beagle_init() 函数来启动启动服务。

我需要先将其添加到系统中：

```
core_initcall(beagle_init)
```

core_initcall(beagle_init) 宏展开为 __define_initcall(beagle_init, 1)，所以 beagle_init() 这个函数被放置在 ".initcall1.init" 段处。

在内核启动时，系统会调用到 do_initcall() 函数。根据指针数组 initcall_levels[1] 找到__initcall1_start 指针，在 vmlinux.lds.h 可以查到：__initcall1_start 对应 ".initcall1.init" 段的起始地址，依次取出段中的每个函数指针，并执行函数。

添加的服务就实现了启动。

可能有些 C 语言基础不太好的朋友不太理解 do_initcall_level() 函数中依次取出地址并执行的函数执行逻辑：

```
for (fn = initcall_levels[level]; fn < initcall_
      do_one_initcall(*fn);
```

fn 为函数指针，fn++ 相当于函数指针 + 1，相当于：内存地址 + sizeof(fn)，sizeof(fn) 根据平台不同而不同，一般来说，32 位机上是 4 字节，64 位机则是 8 字节 (关于指针在操作系统中的大小可以参考另一篇博客： 不同平台下指针大小)。

而 initcall_levels[level] 指向当前 ".initcall##level##s.init" 段，initcall_levels[level+1] 指向 ".initcall##(level+1)##s.init" 段, 两个段之间的内存就是存放所有添加的函数指针。
也就是从 ".initcall##level##s.init" 段开始，每次取一个函数出来执行，并累加指针，直到取完。

好了，关于 linux 中 initcall 系统的讨论就到此为止啦，如果朋友们对于这个有什么疑问或者发现有文章中有什么错误，欢迎留言

原创博客，转载请注明出处！

祝各位早日实现项目丛中过，bug 不沾身.

全文完