Alkaid的makefile实现

1. 设计目标
   1. 分离编译和配置，构筑累积技术的基础

之前采用的buildsettings以及menuconfig使用的方法过於灵活，产生的方案数过多，而实际可行方案很少，做好的可行方案也没有实际标杆，派生方案也不会回溯到原方案。

Alkaid吸取上述教训，试图将成熟方案参数化，并将配置文件分类切分，使得以前的成熟方案可以重复使用，派生方案可以回溯，并观察其演变路径。

Config.mk

Configs

makefile

.o/bin

Dir

image

* 1. 使用参数规范化信息，阻止规则被破坏

规则破坏分两种情况：

* + 1. Reviewer和owner协商无法fix进而采取破坏规则的决定
    2. Owner暂时采取破坏规则的决定

从技术积累的角度，两种情况都涉及到完善规则，以及规则不完善的情况下事后完善的过程，但是实际上经常出现的情况并不是规则本身的问题，而是缺少遵守的意愿，所以这里试图从makefile本身来规范化。

Configs

Config.mk

rules

Action

* 1. 加入防呆设计，防患未然

工业界已经广泛使用了一些具有凹凸的卡槽作为防呆设计，程序内部做这种设计会比较难，但是makefile里面本身因为target是需要被匹配的，所以可以通过设计target做防呆设计，一旦出现无法匹配的target，就是因为配置出错引起的

Configs

Config.mk

rules

unmatch

match

Report error

action

1. 基本技术要点
   1. 映射表

在C语言里面会希望通过索引取得列表，比如定义一个数组

Char arr[3]，那么0, 1, 2是索引，通过索引可以获得保存在数组里面的数据， 例如char arr[3] = { ‘a’, ‘b’, ‘c’}，那麽我们建立了映射关系1 => ‘a’，2 => ‘b’， 3 => ‘c’，取值的时候，使用arr[index]，其中index = [0,1,2]

在makefile里面要做到这一点则是如下安排：

ARR\_1 = A

ARR\_2 = B

ARR\_3 = C

那映射关系1 => A，2 => B，3 => C就通过如下做法体现：

$(ARR\_$(IDX))，其中IDX=[0,1,2]

* 1. 自动补全规则

Makefile的基本规则是：

目标：依赖

命令

其中目标和依赖里面可以使用通配符%，因此妥善安排目标和依赖之间的模式，就可以对任意目标进行分类，比如说：

%\_CLASS: %\_CDEP\_1

$(MAKE) $@

1\_CLASS:

DO 1 CLASS

2\_CLASS:

DO 2 CLASS

Make 1\_CLASS

Make 2\_CLASS

因此makefile的补全规则使得makefile具备了自动识别分类的能力，这种能力类似於C++的虚函数，所以上述的例子类似於：

Class base\_class {

Virtual void action() = 0;

};

Class 1\_class : public base\_class {

Int dep1;

Void action() {//do action 1;}

};

Class 2\_class: public base\_class {

Int dep2;

Void action() {//do action 2;}

};

* 1. 过滤器

Makefile里面的函数是纯函数式的，因此操作起来会比较麻烦一些。

比如说，有一个变量foo，他只允许使用one two three这三个值，six不准用，虽然他也是数字，类似於switch的非法值走到default的意思

这个时候可以直接用$(filter one two three, $(foo))

接下来是进行复杂分类，比如说在一堆食物里面，分类蔬菜水果，颜色形状，这个如果在C语言里面是

Struct {

Int type;

Int color;

Int shape;

};

然后去搜索属性，在makefile里面只能通过临时构造类似struct的变量：

\_TYPE\_COLOR\_SHAPE\_，这样就可以利用filter和filter-out进行过滤了，比如我们有\_apple\_read\_circle\_，\_banana\_yellow\_bar\_，那使用$(filter \_apple\_%, $(foo))就可以把foo里面的apple拿出来

* 1. 变量属性
     1. 全局变量

MAKEFILE\_LIST这个全局变量保存了所有include的makefile名称，因此可以从这个变量里面把include执行时所在的文件夹找出来

* + 1. 赋值

FOO ?=表示如果没有人赋值，给他赋值，这可以用来做初始化

FOO :=表示即刻赋值，这意味着这个值随着时间的推移会变化

FOO =表示等到取值的时候再求值，这意味着这个值只取最后一次赋值

* + 1. 数据包

A = 10

B = 20

PAIR = \_A\_B\_

这个相当於创建了一个数据包，随后可以拆开：

$($(patsubst \_%\_B\_,%,$(PAIR)))返回10

$($(patsubst \_A\_%\_,%,$(PAIR)))返回20

* 1. 面向对象编程

Makefile里面的面向对象需要利用变量模式，下面是计算面积的程序：

WID = 10

HEI = 20

ROUND = 5

rect\_area:

Make SHAPE\_RECT\_WID\_HEI

Circle\_area:

Make SHAPE\_CIRCLE\_ROUND

SHAPE = SHAPE\_RECT\_WID\_HEI

area:

Make $(SHAPE)

SHAPE\_RECT\_%:

#echo area = $(patsubst SHAPE\_RECT\_%\_HEI,%,$@) \* $(patsubst SHAPE\_RECT\_WID\_%,%,$@)

SHAPE\_CIRCLE\_%:

#echo area = $(patsubst SHAPE\_RECT\_%,%,$@)2 \* TT

* 1. 并发

在makefile里面并发是通过-jN实现的，由make去根据当前的N值对编译进度进行调度，但是在makefile里面必须对调度进行正确暗示，否则会出现概率编译出错

并发的位置发生在2个地方，一个是条件，一个是执行，比如说：

Target1 target2 target3:dep1 dep2 dep3

$(MAKE) targetA targetB targetC

$(MAKE) targetD

$(MAKE) targetE

注意必须使用$(MAKE)而不是make，否则make调度会退回1，这是因为$(MAKE)是make自己，但是make是从PATH路径搜索，而PATH路径是可变的

Dep1 dep2 dep3是并发的

targetA targetB targetC是并发的

targetD和targetE是顺序的

* 1. 防呆设计

Makefile里面借助模式匹配做防呆设计，是最好的选择，一把钥匙配一把锁，是因为钥匙跟锁是匹配的，在makefile里面，模式就是那个锁，配置就是钥匙，比如说：

A = very

B = nice

C = good

D = work

我只能接受very nice和good work，very good虽然听起来合理，但是我们要拒绝他，那我们需要这样设计target：

\_very\_nice \_good\_work:

#accpt

%:

#reject

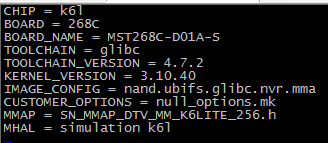
Test:

Make \_$(LEFT)\_$(RIGHT)\_

在设计makefile的时候过滤非法配置，可以节约很多时间，因为有些非法设置既不报错，也不立刻崩溃，可能在一个非常特定的场景冒出问题，查起来非常麻烦

1. 架构
   1. 整体做法是依赖一系列的配置文件，配置文件彼此完全隔离，达到最低的耦合度
   2. 配置文件拆分成2类，一类是固定命名的，也就是config.mk，专门用於处理编译配置，另外一类是不固定命名的，用於处理其余的配置
   3. Project
      1. Project的总控开关位於configs/current.configs，这个开关负责配置所有相关的目录和配置文件名称
      2. Configs目录存放所有的项目配置

这个目录里面的文件模式如下图所示：

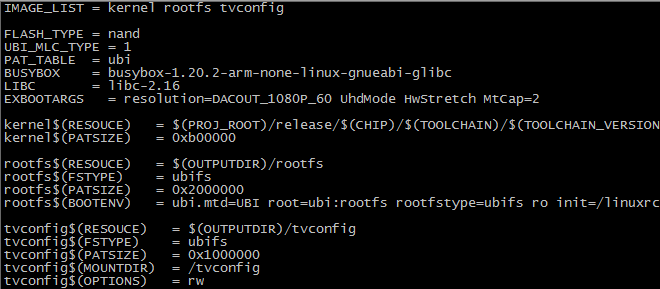


其中名字大多数都是文件夹的名字或配置文件名，注意到IMAGE\_CONFIG指向image/configs目录下的配置文件，CUSTOMER\_OPTIONS指向release/customer\_options目录下的配置文件

MHAL是一个特殊的变量，他可能具备多个值，他代表当前mhal的编译目录，如果有多个值，表示编译多个目录

* + 1. image/configs目录存放image的配置

image的配置分三段描述，典型的配置如下图所示：



这是按照过去的模式，将image配置抽象出来的结果，IMAGE\_LIST表示当前的分区名列表

FLASH\_TYPE表示flash类型，他会影响到烧录命令

UBI\_MLC\_TYPE是UBI专用的，其余的flash不需要这个变量

PAT\_TABLE表示分区表类型，他会影响到分区表

BUSYBOX/LIBC表示busybox和C库，适应不同的工具链以及flash要求

EXBOOTARGS表示设给kernel的额外选项

下面的是分区属性设置，分区属性有5个，如果不填写表示该选项不可用

RESOURCE表示源文件从哪里拷贝过来

PATSIZE表示这个分区要多大

FSTYPE表示使用什麽文件系统，如果没有文件系统则不指定

BOOTENV表示给kernel设置的参数

MOUNTDIR表示挂到板子上的哪个目录

OPTIONS表示挂载参数选项

* + 1. release/customer\_options目录存放编译选项的配置

这个里面保存一些编译选项，目前还没使用

* + 1. board

board里面放配置文件，沿用以前SN/MI的方案，删除了多余的

* + 1. kbuild

kbuild提供kernel编译环境，这个环境支持三种不同的目的：

* + - * + 快速切换kernel

这个可以支持很多场景，比如一个芯片多个kernel，比如一个生产kernel一个调试kernel，比如临时在两个芯片之间进行横向对比

* + - * + 提供可控稳定的编译环境

这个主要是为了控制编译选项，kernel的makefile和头文件是稳定的，经常改动的只有配置文件，所以可以控制kernel的makefile来对编译环境进行管控

* + - * + 避免与特定kernel形成依赖

如果直接放到Kernel的代码树，可能会依赖一些比较奇怪的特性，而且随着kernel捆绑在一起，单独拿出来的话，代码需要同时适应多个kernel

Kbuild同步了kernel的以下文件夹：

Arch/arm/include

Include/linux

Include/uapi

Include/asm-generic

Include/config

Include/generated

Scripts

Makefile

这些就是基础的kernel环境

Kernel的可变部分是：

.config

Makefile

Module.symvers

include/config/auto.conf

include/generated/\*

每个kernel只需要拿到这5个可变的部分就可以切换

对於编译管控的部分，定制了scripts/Makefile.build，目前只在里面加入了MSTAR\_REMOVE\_INCS用来控制头文件选择，这里利用了过滤器的机制，在如果MSTAR\_REMOVE\_INCS=all-exception，则会把所有的-I开头的参数都过滤掉，效果就是无法引用到kernel的头文件

后续可以根据需要，对宏和其他参数进行管控

* + 1. Image

这里结合了之前的经验，将产生image的过程拆分成了3个过程，每个过程单独使用一个.mk，这样防止3个过程在维护的过程中变得混乱

按顺序，分别是：

Rootfs.mk

Rootfs主要是展开文件包，拷贝配置文件，重点应该是配置etc目录，这个目录里面fstab和profile这两个文件是需要动态配置的，其中fstab的配置要根据image/configs里面的配置项来产生，主要是针对带有MOUNTDIR的分区设置挂载点

Image.mk

Image只负责针对已经准备好的文件夹产生image，他主要参考RESOURCE, PATSIZE, FSTYPE这三个变量

Scripts.mk

Script负责产生烧写脚本，烧写脚本与2个因素有关，一个是flash type，一个是文件系统类型，另外他还要负责处理内核的环境变量，以及可能的mboot环境变量，因此他主要关心的是BOOTENV, EXBOOTARGS, FLASH\_TYPE, PAT\_TABLE

通过这3个脚本处理好整个image的构建过程

* 1. Sdk

Sdk里面主要处理的是编译有关的配置文件，4个目录使用了完全不一样的编译策略，所以目前暂时还没有统一成一个makefile，只是分散在4个文件夹里面，除了misc以外，其余3个文件夹使用一项共同的策略，如下图所示：

a/config.mk

ARG:=1

include prepare.mk

include submit.mk

LIST+=a

a\_ARG := 1

Prepare-config

ARG := NULL

Makefile

Include \*/config.mk

Get LIST

PROC LIST

LIST+=b

b\_ARG := 2

b/config.mk

ARG:=2

include prepare.mk

include submit.mk

submit-config

Get DIR

LIST += $(DIR)

$(DIR)\_ARG=$(ARG)

c/config.mk

ARG:=3

include prepare.mk

include submit.mk

LIST+=c

c\_ARG := 3

图中兰色的三个文件作为框架，处理琐碎事务，config.mk作为配置文件，收集信息，左边这个是汇总文件，他得到的是一系列变量名，他根据变量名去获取参数，所以这个过程就是一个类似下面C代码显示的程序：

struct config {

const char \*name;

int arg;

};

struct config list [] = { {“a”, 1}, {“b”, 2}, {“c”, 3} };

for(int i = 0; i < sizeof(list)/sizeof(\*list); ++i){

//do sth with list[i];

}

Verify/interface/mhal的核心部分都是按照这种逻辑组织的，这样使得makefile系统可以动态适配新增和删除，以及细粒度到每个模块的修改，一个config.mk相当於创建了一个参数对象，prepare-config相当於分配内存函数，submit-config相当於赋值函数，prepare-config和submit-config共同创建对象

同时，因为所有的配置会汇总到一个makefile里面统一进行处理，这样可以有效对makefile系统进行管控，下面分别介绍4个文件夹的差异

* + 1. Mhal

Mhal是最复杂的，因为这个目录里面本身就包含了3种不同的布局，一开始设计为单层布局，后来又新增了2种双层布局。

* + - * + 单层布局

单层布局里面只有2个文件夹，pub/inc/src，因为考虑到要所有chip使用同样的头文件，因此单层布局的头文件是放在mhal/include里面的，不过因为后面引入了双层布局，因此单层布局的也可能需要把头文件收进目录自己的目录，避免头文件被删掉

* + - * + 双层布局

双层是后面引入的带有drv/hal的布局，他是双层的不是说他有2个文件夹，而是因为他本身包含依赖关系，drv是高层，hal是底层，这种双层布局是为了直接从其他地方拷贝代码而准备的，按照要求，又使用了2种有少许差别的细分结构：一种为hal的结构使用固定的子目录，pub/inc/src，而raw的结构使用内部任意安排的子目录

Utpa是一个独立的文件夹，因为他没有在文件夹里面放config.mk，而是直接放了makefile

所以mhal的makefile要同时兼容上述4种不同的目录

为了达到上述目的，prepare-config使用了3种不同的参数进行适配，最终每个config.mk获取到的参数为：

MACROS表示值为1的宏列表，所有的宏使用方法为：

#if XXX

XXX == 1

#else

XXX != 1

#endif

这样做开关可以避免出现不一致的情况导致混乱/预期之外的情形

DEPS表示当前依赖列表

STYLE表示单层，双层hal，双层raw的文件结构

INCS表示单层布局的依赖头文件文件夹列表

SRCS表示单层布局的代码文件夹列表

DRV\_INCS表示双层布局的高层依赖头文件文件夹列表

DRV\_SRCS表示双层布局的高层代码文件夹列表

HAL\_INCS表示双层布局的底层依赖头文件文件夹列表

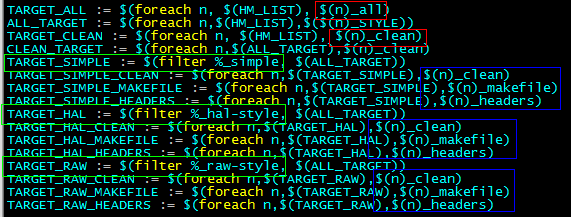
HAL\_SRCS表示双层布局的底层代码文件夹列表

DRV\_PUB\_PATH表示如果其他人依赖了本模块，本模块可以给他看到哪些文件夹

RELEASE\_INC表示对外release的头文件列表

HM\_LIST表示HAL MODULE LIST

在makefile里面，根据HM\_LIST构建target，根据target所具备的模式，选择不同的编译方法，模式列表如下图所示：



红色是根据模块名得出的target，比如i2/vpe\_all i2/vpe\_clean

绿色是中间使用的临时target，用来区分不同的目录布局

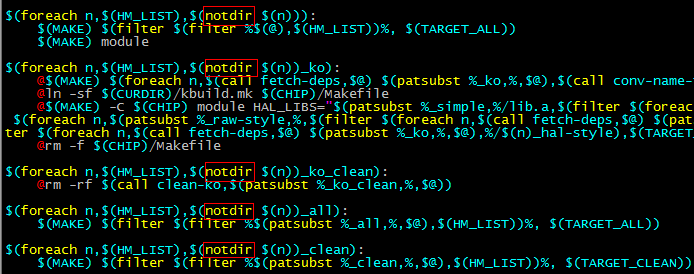
兰色是在临时target之上构建的子逻辑，这里只有3个，分别是产生clean，makefile和headers

处理单层和双层布局的头文件和clean时逻辑不一样，单层布局按原计划是不需要release头文件的，而双层的需要

清理的逻辑也不一样，因为双层布局他具备高层和底层的概念，因此要底层和高层分别编译，而单层布局只编译一次

根据上面的target分别编写makefile规则就得到了最终的makefile

Mhal还有一个复杂的地方在於，需要同时处理多个文件夹的代码，在config.mk给出的HM\_LIST就可以发现，HM\_LIST的结构是dir/dir，而owner在编译的时候其实是不关心模块位於哪个文件夹的，比如i2/vpe，owner只关心vpe，因此mhal里面专门新增了几个剔除文件夹的target：



红色的notdir就是去除前缀用的

* + 1. Interface

Interface里面做的是类似的，所不同的地方在於，interface里面会同时处理user mode和kernel mode的编译，并且会根据依赖关系安排insmod的次序。

这里重点讲一下interface里面的config.mk的参数：

DEP\_MODULE(S)表示依赖的interface模块，这个参数会用来当作依赖ko

DEP\_EXT\_INC(S)表示依赖interface模块但这个模块没有ko

DEP\_HAL(S)表示依赖mhal的模块，这个参数控制包含mhal的头文件

API\_FILE表示当作API的那个文件，这个文件默认会编译两次

UAPI\_DISABLE表示不在user mode编译API\_FILE，这样只有kernel能访问

KAPI\_DISABLE表示不在kernel mode编译API\_FILE，这样只有user能访问

UINTERNAL\_FILES表示在user mode编译的文件

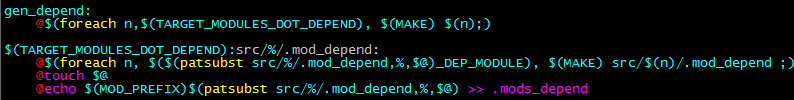
WRAPPER\_FILE表示在kernel mode编译的函数表文件，如果不提供这个文件将不产生ko

IMPL\_FILE(S)表示在kernel mode编译的模块文件

USER\_INC表示在开发so时需要借用第三方库时库头文件的路径

这些参数里面，DEP\_MODULE(S)被用於安排依赖，因为每个模块一个ko，所以依赖关系必须体现在insmod时的次序上，不能出现循环依赖，如果出现循环依赖将无法insmod。

因此在interface里面加入了如下的代码安排依赖问题：



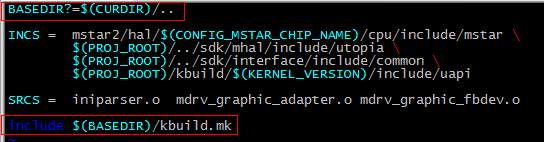
这里安排依赖很简单，就是把DEP\_MODULE的内容写入到.mods\_depend

* + 1. Verify

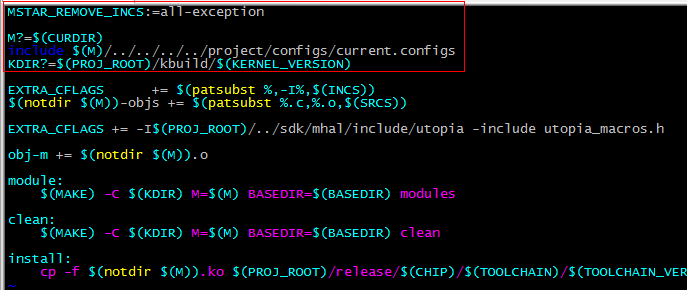
Verify里面的特别之处在於，他提供了两级搜索，除此之外也没有特别之处

* + 1. Misc

Misc里面存放的是config\_tool和fdev，这里user mode的代码没有特别管控，kernel mode的代码可能会需要借用一个特别的makefile，比如fbdev的是：



Kbuild.mk的特别之处在於，他是一个alkaid专用的编译模板，alkaid的kernel环境是受管控的，因此需要取消管控：



如果不用alkaid提供的当然不存在这个问题

1. 裁剪
   1. 按feature裁剪

Feature宏全部放在project/release/customer\_options，全局可见的，在代码里面通过这里的宏对某个特性进行开关

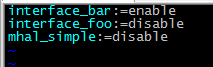
* 1. 按module裁剪

模块使用了3个前缀，mhal\_控制mhal文件夹下面的代码

Interface控制了verify/module interface impl三个文件夹下面的代码

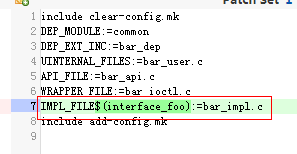
Verify控制了verify/feature下面的代码

* + 1. 整个模块开关



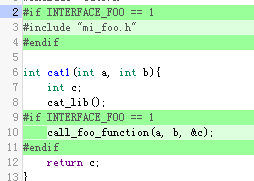
这里表示bar是启用的，foo和simple是关闭的

* + 1. A模块有几个文件依赖B模块



将依赖的文件保存在带$(interface\_foo)后缀的变量名之后就可以感知到模块开关

* + 1. A模块的某个文件有几行代码依赖B模块



使用同名的大写宏可以感知到