概观

========

Kconfiglib是一个Python 2/3库，用于从Kconfig（https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt）配置系统编写脚本并提取信息。

有关详细信息，请参阅https://github.com/ulfalizer/Kconfiglib上的主页。

在Linux内核上使用Kconfiglib和Makefile目标

================================================== ============

对于Linux内核，scripts / kconfig / Makefile补丁提供了一个方便的界面，可以使用'git am'或'patch'实用程序：

  $ wget -qO- https://raw.githubusercontent.com/ulfalizer/Kconfiglib/master/makefile.patch |我是

  $ wget -qO- https://raw.githubusercontent.com/ulfalizer/Kconfiglib/master/makefile.patch |补丁-p1

警告：不将-p1传递给补丁会导致错误的文件被修补。

如果补丁不适用，请告诉我。手动应用应该是微不足道的，因为它只是需要在scripts / kconfig / Makefile中的其他\* conf：targets附近插入的文本块。

进一步了解Makefile补丁的动机以及如何在没有它的情况下使用Kconfiglib的说明。

如果您不希望通过pip安装Kconfiglib，则会设置Makefile补丁，以便您也可以将Kconfiglib克隆到内核根目录中：

  $ git clone git：//github.com/ulfalizer/Kconfiglib.git

  $ git am Kconfiglib / makefile.patch（或'patch -p1 <Kconfiglib / makefile.patch'）

警告：在这种情况下，目录名称Kconfiglib /很重要，因为它被makefile.patch中的新目标添加到PYTHONPATH。

Makefile补丁添加的目标如下所述部分。

制作kmenuconfig

----------------

此目标使用Python 3运行curses menuconfig接口（menuconfig当前不支持Python 2）。

make [ARCH = <arch>] iscriptconfig

--------------------------------

此目标提供了一个交互式Python提示符，其中已预加载Kconfig实例并且可在“kconf”中使用。要更改使用的Python解释器，请传递PYTHONCMD = <executable>。默认为“python”。

要了解API，请尝试在kconf.defined\_syms中评估和打印符号，并通过跟随'next'和'list'指针来探索从kconf.top\_node开始的MenuNode菜单树。

菜单节点中包含的项目位于MenuNode.item中（请注意，这可以是常量kconfiglib.MENU和kconfiglib.COMMENT之一），并且所有符号和选项都具有包含其菜单节点的“nodes”属性（通常只有一）。打印菜单节点将以Kconfig格式打印其项目。

如果要按名称查找符号，请使用kconf.syms字典。

make scriptconfig SCRIPT = <script> [SCRIPT\_ARG = <arg>]

-------------------------------------------------- -

此目标运行由配置上的SCRIPT参数给出的Python脚本。 sys.argv [1]保存顶级Kconfig文件的名称（实际上当前总是“Kconfig”），sys.argv [2]保存SCRIPT\_ARG参数（如果给定）。

有关示例脚本，请参阅examples /子目录。

make dumpvarsconfig

-------------------

此目标打印从Kconfig文件引用的所有环境变量的列表及其值。请参阅Kconfiglib / examples / dumpvars.py脚本。

仅包括通过Kconfig预处理器$（FOO）语法引用的环境变量。预处理器是在Linux 4.18中添加的。

使用没有Makefile目标的Kconfiglib

=============================================

make目标仅用于获取从Kbuild makefile导出的环境变量，并通过例如Kconfig文件在Kconfig文件中引用。 'source“arch / $（SRCARCH）/ Kconfig”和命令通过'$（shell，...）'运行。

这些变量在写作时引用（Linux 4.18），以及示例值：

  srctree（。）

  ARCH（x86）

  SRCARCH（x86）

  KERNELVERSION（4.18.0）

  CC（gcc）

  HOSTCC（gcc）

  HOSTCXX（g ++）

  CC\_VERSION\_TEXT（gcc（Ubuntu 7.3.0-16ubuntu3）7.3.0）

较旧的内核仅引用ARCH，SRCARCH和KERNELVERSION。

如果您的内核足够新（4.18+），您可以通过'make dumpvarsconfig'获取引用环境变量的列表（参见上文）。请注意，此命令由Makefile补丁添加。

要在没有Makefile补丁的情况下运行Kconfiglib，请手动设置环境变量：

  $ srctree =。 ARCH = x86 SRCARCH = x86 KERNELVERSION =`make kernelversion` ... python（3）

  >>>导入kconfiglib

  >>> kconf = kconfiglib.Kconfig（）#filename默认为“Kconfig”

在顶级Makefile中搜索“其他ARCH设置”​​以查看ARCH和SRCARCH的其他可能性。

符号值简介

======================

Kconfiglib具有与C实现相同的赋值语义。

用户可以为任何符号分配值（通过Kconfig.load\_config（）或Symbol.set\_value（）），但只有符号可见时才会尊重此用户值，这与（当前）在menuconfig中可见相对应接口。

对于带提示的符号，符号的可见性由提示中的条件确定。没有提示的符号永远不可见，因此在它们上设置用户值毫无意义。如果在无提示符号上调用Symbol.set\_value（），则默认情况下将打印警告。在.config文件中对无提示符号的赋值是正常的，因此load\_config（）不会打印类似的警告。

来自父项的依赖关系和'if'/'取决于'会传播到属性，包括提示，因此这两个配置在逻辑上是等效的：

（1）

  菜单“菜单”

      取决于A.

  如果B

  配置FOO

      如果D，那就是“foo”

      默认的y

      取决于C.

  万一

  endmenu

（2）

  菜单“菜单”

      取决于A.

  配置FOO

      如果A && B && C && D，则为“foo”

      如果A && B && C，则默认为y

  endmenu

在此示例中，A && B && C && D（提示条件）需要为非n，以使FOO可见（可分配）。如果其值为m，则只能为符号赋值m：可见性设置用户可以分配的值的上限，并且任何更高的用户值将被截断。

'default'属性与可见性无关，但由于依赖性传播，'default'通常会得到与提示相同的条件。如果符号不可见或没有用户值，则使用“默认”属性。

没有用户值（或具有用户值但不可见）的符号和没有（活动）'default'的符号默认为bool / tristate符号的n，以及其他符号类型的空字符串。

'select'与符号可见性的工作方式类似，但设置符号值的下限。下限由select \* ing \*符号的值确定。 'select'不尊重可见性，因此也可以通过select将不可见符号强制为特定（最小）值。

对于非bool / tristate符号，只关注可见性是n还是非n：m可见性与y可见性相同。

“默认”和“选择”的条件以大多数直观的方式工作。如果条件为n，则禁用“默认”或“选择”。如果是m，则“默认”或“选择”值（选择符号的值）将被截断为m。

使用Kconfig.write\_config（）编写配置时，只有可见，具有（活动）默认值或被选中的符号才会被写出（请注意，这包括所有可接受用户值的符号）。 Kconfiglib将C实现生成的.config格式与字符匹配。这样可以简化测试。

对于值为n的可见bool / tristate符号FOO，此行将写入.config：

    #CONFIG\_FOO未设置

关键是要记住用户n的选择（可能与符号的默认值不同），同时坚持未定义的规则对应于n（.config使用Makefile格式，使得该行在注释之上） 。当读回.config文件时，该行将被视为与以下赋值相同：

    CONFIG\_FOO =正

在Kconfiglib中，bool / tristate符号的（当前）可分配值集显示在Symbol.assignable中。对于其他符号类型，只需检查sym.visibility是否为非0（非n）以查看用户值是否有效。

菜单树简介

======================

菜单结构，如例如menuconfig，由MenuNode对象树表示。配置的顶部节点对应于隐式顶级菜单，其标题显示在标准menuconfig界面的顶部。 （标题也可以在Kconfig.ib中使用Kconfig.mainmenu\_text。）

顶部节点位于Kconfig.top\_node中。从那里，您可以通过跟随'list'指针访问子菜单节点，并通过跟随'next'指针访问任何后续菜单节点。通常，非“无'列表”指针指示菜单或选项，但由于从依赖项隐式创建的子菜单，符号的菜单节点有时也可能具有非无'列表'指针。

MenuNode.item可以是Symbol或Choice对象，也可以是常量MENU和COMMENT之一。菜单节点的提示可以在MenuNode.prompt中找到，它还包含菜单和注释的标题。对于Symbol和Choice，MenuNode.help保存帮助文本（如果有的话，否则为None）。

大多数符号只有一个菜单节点。在多个位置定义的符号将为每个位置提供一个菜单节点。可以在Symbol / Choice.nodes属性中找到Symbol或Choice的菜单节点列表。

请注意，符号和选项的提示和帮助文本存储在其菜单节点中，而不是存储在Symbol或Choice对象本身中。这使得可以在多个位置定义符号，并在每个位置使用不同的提示或帮助文本。要获取带有单个菜单节点的符号的帮助文本或提示，请分别执行sym.nodes [0] .help和sym.nodes [0] .prompt。提示符是（文本，条件）元组，其中condition确定可见性（请参阅下面的'表达式简介'）。

该组织反映了C实现。调用MenuNode

'struct menu'在那里，但我认为“菜单”是一个令人困惑的名字。

可以为Choice提供一个名称并在多个位置定义它，因此Choice.nodes也是一个列表。

为方便起见，在特定的定义位置添加的属性可在MenuNode本身上获得，例如， MenuNode.defaults。这在生成文档时很有用，因此可以在每个位置使用正确的属性显示在多个位置定义的符号/选项。

表达式简介

====================

表达式可以使用expr\_value（）函数进行计算，并使用expr\_str（）函数打印（这些函数也在内部使用）。评估表达式总是产生三态值，其中n，m和y分别表示为0,1和2。

下表可帮助您了解表达式的表示方式。

A，B，C，...是符号（符号实例），NOT是kconfiglib.NOT

不变等

表达表示

---------- --------------

A

“A”A（常数符号）

！A（不，A）

A && B（AND，A，B）

A && B && C（AND，A，（AND，B，C））

A || B（OR，A，B）

A || （B && C && D）（OR，A，（AND，B，（AND，C，D）））

A = B（EQUAL，A，B）

A！=“foo”（UNEQUAL，A，foo（常数符号））

A && B = C && D（AND，A，（AND，（EQUAL，B，C），D））

n Kconfig.n（常数符号）

m Kconfig.m（常数符号）

y Kconfig.y（常数符号）

“y”Kconfig.y（常数符号）

'default“foo”'或'取决于SYM =“foo”'中的“foo”字符串表示为常量符号，因此表达式中出现的唯一值是符号\*\*\*。这反映了C实现。

\*\*\*对于选择符号，父选项也将出现在表达式中，但它通常是不可见的，因为Symbol和Choice的值接口是相同的。这反映了C实现，并使不同的选择模式“正常工作”。

手动评估示例：

   - A && B的值是min（A.tri\_value，B.tri\_value）

   - A ||的值B是max（A.tri\_value，B.tri\_value）

   - ！A的值是2 - A.tri\_value

   - 如果A.str\_value == B.str\_value，则A = B的值为2（y），否则为0（n）。请注意，此处使用str\_value而不是tri\_value。

    对于常量（以及未定义）符号，str\_value匹配符号的名称。这反映了C实现，并解释了为什么“取决于SYM =”foo“'以上是按预期工作的。

在解析期间，n / m / y自动转换为相应的常数符号“n”/“m”/“y”（Kconfig.n / m / y）。

Kconfig.const\_syms是一个类似Kconfig.syms的字典，但是用于常量符号。

如果缺少某个条件（例如，当'if <cond>'从'default a cond>'中移除'if <cond>'时<cond>），它实际上是Kconfig.y。标准\_\_str \_\_（）函数只是避免打印'if y'条件以提供更清晰的输出。

Kconfig扩展

==================

Kconfiglib包含几个Kconfig扩展：

'源'与相对路径

---------------------------

'rsource'语句使用相对于包含'rsource'语句的Kconfig文件目录的路径来源Kconfig文件，而不是相对于项目根目录。

考虑以下目录树：

  项目

  + - 的Kconfig

  |

  + - SRC

     + - 的Kconfig

     |

     + - SubSystem1

        + - 的Kconfig

        |

        + - ModuleA

           + - 的Kconfig

在此示例中，假设src / SubSystem1 / Kconfig想要获取src / SubSystem1 / ModuleA / Kconfig。

使用'source'，将使用以下语句：

  源“src / SubSystem1 / ModuleA / Kconfig”

随着'rsource'，这变成了

  rsource“ModuleA / Kconfig”

如果给'rsource'一个绝对路径，它的作用与'source'相同。

'rsource'可用于创建“位置无关”的Kconfig树，可以自由移动。

全球'来源'

-----------------

'source'和'rsource'接受glob模式，获取所有匹配的Kconfig文件。它们至少需要一个匹配的文件，否则抛出KconfigError。

例如，以下语句可能源为sub1 / foofoofoo和sub2 / foobarfoo：

  源“sub [12] / foo \* foo”

接受的glob模式与标准的glob.glob（）函数相同。

对于模式可以匹配没有文件的情况，提供了另外两个语句：'osource'和'orsource'（o代表“可选”）。

例如，如果“foo”和任何与“bar \*”匹配的文件都不存在，则以下语句将为no-ops：

  osource“foo”

  osource“bar \*”

'orsource'执行相对可选的来源。

'source'和'osource'类似于Make中的'include'和'-include'。

广义的def\_ \*关键字

--------------------------

除了def\_bool和def\_tristate之外，还可以使用def\_int，def\_hex和def\_string，允许int，hex和string符号同时被赋予类型和默认值。

额外的可选警告

-----------------------

可以通过环境变量控制一些可选警告：

   - KCONFIG\_WARN\_UNDEF：如果设置为“y”，将为Kconfig文件中对未定义符号的所有引用生成警告。唯一的问题是所有十六进制文字必须以“0x”或“0X”作为前缀，以便能够将它们与符号引用区分开来。

    某些项目（例如Linux内核）使用多个Kconfig树和许多共享Kconfig文件，从而导致一些安全的未定义符号引用。

    KCONFIG\_WARN\_UNDEF在仅具有单个Kconfig树的项目中很有用。

    KCONFIG\_STRICT是此环境变量的旧别名，支持向后兼容性。

   - KCONFIG\_WARN\_UNDEF\_ASSIGN：如果设置为“y”，将为.config文件中未定义符号的所有分配生成警告。默认情况下，不会生成此类警告。

    也可以通过Kconfig.enable / disable\_undef\_warnings（）启用/禁用此警告。

Python中定义的预处理器用户函数

---------------------------------------------

预处理器函数可以在Python中定义，这使得将现有Python工具中的信息集成到Kconfig中变得简单（例如，使Kconfig符号依赖于以某种其他格式存储的硬件信息）。

在sys.path中的任何位置放置名为kconfigfunctions（.py）的Python模块将导致它由Kconfiglib导入（在Kconfig .\_\_ init \_\_（）中）。请注意，可以通过PYTHONPATH自定义sys.path，包括默认运行的模块目录以及安装目录。

如果设置了KCONFIG\_FUNCTIONS环境变量，则它将使用不同的模块名称而不是“kconfigfunctions”。

导入的模块应该定义一个名为'functions'的全局字典，它将函数名映射到Python函数，如下所示：

  def my\_fn（kconf，name，arg\_1，arg\_2，...）：

      #kconf：

      #Kconfig实例

      ＃

      ＃ 名称：

      ＃用户自定义函数的名称（“my-fn”）。想想argv [0]。

      ＃

      ＃arg\_1，arg\_2，...：

      ＃从Kconfig传递给函数的参数（字符串）

      ＃

      ＃返回一个要替换的字符串作为调用的结果

      #function

      ...

  def my\_other\_fn（kconf，name，arg\_1，arg\_2，...）：

      ...

  functions = {

      “my-fn”:( my\_fn，<min.args>，<max.args> / None），

      “my-other-fn”:( my\_other\_fn，<min.args>，<max.args> / None），

      ...

  }

  ...

<min.args>和<max.args>是函数预期的最小和最大参数数（不包括隐式'name'参数）。如果<max.args>为None，则参数数量没有上限。传递无效数量的参数将生成KconfigError异常。

定义后，可以使用与其他预处理器功能相同的方式从Kconfig调用用户函数：

    配置FOO

        ...

        取决于$（my-fn，arg1，arg2）

如果my\_fn（）返回“n”，则会导致

    配置FOO

        ...

        取决于n

警告

\*\*\*\*\*\*\*

用户定义的预处理器函数在解析时，在处理完所有Kconfig文件之前以及在菜单树完成之前遇到它们。无法保证通过'kconf'参数访问Kconfig符号或菜单树将起作用，并且可能导致崩溃。 'kconf'参数是为将来的扩展而提供的（因为预定义的函数无论如何都会接受它）。

优选地，用户定义的函数应该是无状态的。

反馈

========

将错误报告，建议和问题发送到ulfalizer a.t Google的电子邮件服务，或在GitHub页面上打开票证。

“””