openwrt LUCI工作原理浅析

fog

2022-08-04 约 6742 字 预计阅读 14 分钟

LUCI 安装

```
$ ./scripts/feeds update packages luci
$ ./scripts/feeds install -a -p luci
$ make menuconfig
```

- LUCI
  - Collections
    - luci

LUCI 的工作原理

首先说明，这里分析的版本为 openwrt-21.02.3

LUCI 可以理解为 lua + UCI 。是用 lua 实现的读写 UCI 配置的一个框架。这个框架配合 uhttpd 可以实现简单快速的页面开发和访问配置。

在 openwrt 中，默认 uhttpd 的启动参数如下

```
/usr/sbin/uhttpd -f -h /www -r OpenWrt -x /cgi-bin -u /ubus -t 60 -T 30 -k 20 -A 1 -n 3 -N 100 -R -p 0.0.0.0:80 -p [::]:80
```

页面显示

以访问主页面为例。访问的页面为 http://192.168.1.1/ ，此时访问的是 uhttpd 提供的 80 端口开放的服务。

查看 uhttpd 的配置文件 `/etc/config/uhttpd` 可以知道

```
        list listen_http '0.0.0.0:80'
        list listen_http '[::]:80'
        list listen_https '0.0.0.0:443'
        list listen_https '[::]:443'
        option redirect_https '0'
        option home '/www'
        option rfc1918_filter '1'
        option max_requests '3'
        option max_connections '100'
        option cert '/etc/uhttpd.crt'
        option key '/etc/uhttpd.key'
        option cgi_prefix '/cgi-bin'
        list lua_prefix '/cgi-bin/luci=/usr/lib/lua/luci/sgi/uhttpd.lua'
        option script_timeout '60'
        option network_timeout '30'
        option http_keepalive '20'
        option tcp_keepalive '1'
        option ubus_prefix '/ubus'

config cert 'defaults'
        option days '730'
        option key_type 'ec'
        option bits '2048'
        option ec_curve 'P-256'
        option country 'ZZ'
        option state 'Somewhere'
        option location 'Unknown'
        option commonname 'OpenWrt'
```

默认读取的应该是 `/www` （home 目录）下的 index.html 文件。该文件包含以下语句

```
<meta http-equiv="refresh" content="0; URL=cgi-bin/luci/" />
```

所以会被自动导向 `/cgi-bin/luci/` 。而从 `/etc/config/uhttpd` 文件中的 `list lua_prefix` 设置可以知道当解析到 url 字串为 `/cgi-bin/luci` 的时候会转给 lua 脚本 `/usr/lib/lua/luci/sgi/uhttpd.lua` 来处理。

看看里面的主要内容

```
require "luci.dispatcher"

function handle_request(env)
    local x = coroutine.create(luci.dispatcher.httpdispatch)
```

可以看到在获得 http 的 request 之后又转给 `/usr/lib/lua/luci/dispatcher.lua` 里面的 httpdispatch 函数处理去了，最终执行的是函数 dispatch 。

```
    end, error500)

function dispatch(request)
    local menu = menu_json()
    local page = menu
    -- 分析 /usr/share/luci/menu.d/*.json 文件，获得页面的节点树

    if action.type == "view" then
    elseif action.type == "call" then
    elseif action.type == "firstchild" then
    -- ...
    -- 根据节点里面的action属性做相应的处理
```

dispatch 函数会先根据目录 `/usr/share/luci/menu.d/` 下的内容产生节点树，然后由带进来的 request 找到对应的节点，根据 `action.type` 再作相应的处理，将请求的页面返回给客户端浏览器。

由于 action.type 为 view 和 firstchild 的情形比较典型，所以这里只分析这两种的处理过程。理解了这两种，其他的也就很容易看懂了。

action.type == view

先来看看 `action.type == view` 的情况 。比如访问路径为 `http://192.168.1.111/cgi-bin/luci/admin/system`，在文件 `/usr/share/luci/menu.d/luci-mod-system.json` 中定义的路径如下:

```
"admin/system/system": {
        "title": "System",
        "order": 1,
        "action": {
                "type": "view",
                "path": "system/system"
        },
        "depends": {
                "acl": [ "luci-mod-system-config" ]
        }
},
```

当访问该页面时（菜单 System->System ），lua 脚本的调试输出如下（调试的方法见后面章节）：

```
[09:48:00]: init_template_engine enter, media = /luci-static/bootstrap
[09:48:00]: Template.__init__ enter, name = themes/bootstrap/header
[09:48:00]: sourcefile =/usr/lib/lua/luci/view/themes/bootstrap/header.htm
[09:48:00]: Template.__init__ exit
[09:48:00]: init_template_engine after Template themes/bootstrap/header
[09:48:00]: init_template_engine exit
[09:48:00]: dispatch action =
{
    type = 'view'
    path = 'system/system'
}
[09:48:00]: Template.__init__ enter, name = view
[09:48:00]: sourcefile =/usr/lib/lua/luci/view/view.htm
[09:48:00]: Template.__init__ exit
[09:48:00]: Template.render enter, scope =
{
    view = 'system/system'
}
[09:48:00]: Template.__init__ enter, name = header
[09:48:00]: sourcefile =/usr/lib/lua/luci/view/header.htm
[09:48:00]: Template.__init__ exit
[09:48:00]: Template.render enter, scope =
{
}
[09:48:00]: Template.__init__ enter, name = themes/bootstrap/header
[09:48:00]: Template.__init__ exit
[09:48:00]: Template.render enter, scope =
{
}
[09:48:00]: Template.render exit
[09:48:00]: Template.render exit
[09:48:00]: Template.__init__ enter, name = footer
[09:48:00]: sourcefile =/usr/lib/lua/luci/view/footer.htm
[09:48:00]: Template.__init__ exit
[09:48:00]: Template.render enter, scope =
{
}
[09:48:00]: Template.__init__ enter, name = themes/bootstrap/footer
[09:48:00]: sourcefile =/usr/lib/lua/luci/view/themes/bootstrap/footer.htm
[09:48:00]: Template.__init__ exit
[09:48:00]: Template.render enter, scope =
{
}
[09:48:00]: Template.render exit
[09:48:00]: Template.render exit
[09:48:00]: Template.render exit
[09:48:00]: dispatcher.lua  httpdispatch exit
[09:48:00]: dispatcher.lua  httpdispatch enter, pathinfo = /admin/translations/en
[09:48:00]: dispatch enter
[09:48:00]: init_template_engine enter, media = /luci-static/bootstrap
```

```
[09:48:00]: template.__init__ enter, name = themes/bootstrap/header
[09:48:00]: init_template_engine exit
[09:48:00]: dispatch action =
{
    module = 'luci.controller.admin.index'
    type = 'call'
    function = 'action_translations'
}
[09:48:00]: dispatcher.lua  httpdispatch exit
```

对应代码进行理解

```lua
function dispatch(request)
    local tpl = init_template_engine(ctx)
    -- ...
    if action.type == "view" then
        tpl.render("view", { view = action.path })
```

```lua
local function init_template_engine(ctx)
    local tpl = require "luci.template"
    -- ...
    return tpl
```

可以看出 init_template_engine 函数的返回值就是 `luci/template.lua` 模块，所以 tpl.render 也就对应了文件 template.lua 中的 render 函数。

```lua
function render(name, scope)
    return Template(name):render(scope or getfenv(2))
end
```

render 函数会先创建一个 `Template` 对象，然后调用其 `render` 方法。所以首先执行的就是 `Template.__init__` 函数。参数 `name` 的值为 `view` ， `scope` 的值为 `{view = 'system/system'}`

根据函数 `Template.__init__` 的代码

```lua
local tparser = require "luci.template.parser"

function Template.__init__(self, name, template)
    sourcefile = viewdir .. "/" .. name .. ".htm"
    self.template, _, err = tparser.parse(sourcefile)
```

此时会去读取文件 `/usr/lib/lua/luci/view/view.htm` 进行分析。看一下这个文件

```
    <div class="spinning"><%:Loading view…%></div>
    <script type="text/javascript">
        L.require('ui').then(function(ui) {
            ui.instantiateView('<%=view%>');
        });
    </script>
</div>

<%+footer%>
```

还是比较简单的。这里需要理解一下 luci 的 template 解析时的语法。这里只对该 htm 文件涉及的内容做注解，完整的语法可以参考 Templates.md on Github

标记	说明
`<%+header%>`	载入名为 header 的 template
`<%+footer%>`	载入名为 footer 的 template
`<%=view%>`	读取 lua 中变量 view 的值，即 system/system

由于 header 和 footer 都是 template，所以也会调用 Template 来解析。（所以在调试信息中看到 Template 构造函数被多次调用）

这里有兴趣还可以看一下 tparser.parse 的源码。其中就可以看到对应的语法解析。

在 `/usr/lib/lua/luci/template` 目录可以看到里面只有一个 parser.so，所以 tparser 其实就是这个动态库文件。那这个动态库是怎么来的呢？这就得看看 luci 的源码了。文件 `luci-base/src/Makefile` 定义了 parser.so 的源码文件

```
parser.so: template_parser.o template_utils.o template_lmo.o template_lualib.o plural_formula.o
```

找到代码中的 `template_L_parse` 函数，就是 parse 的入口了。

好了，回到刚才的 view.htm 文件。这个文件在经过 parse 之后其实就变成了如下的内容

```
<!-- 当前主题对应的header.js 的内容 -->

<div id="view">
    <div class="spinning"><%:Loading view…%></div>
    <script type="text/javascript">
        L.require('ui').then(function(ui) {
            ui.instantiateView('<system/system>');
        });
    </script>
</div>

<!-- 当前主题对应的footer.js 的内容 -->
```

这些内容会被服务器返回给浏览器客户端。然后浏览器客户端会调用其中嵌入的 js 脚本，执行脚本中的函数（来源于 luci.js）从服务器端获取需要的数据（通过 json 封装），完成最终的绘制。

生成菜单

页面上的菜单是怎么来的？如果我要添加一个新的 luci application，怎么加入现有的菜单？

这一节我们就来讨论这个问题。

在 dispatch 函数刚刚进入就有如下的语句

```
1 local menu = menu_json()
```

menu_json 的源码如下：

```
1 function menu_json(acl)
2     local tree = context.tree or createtree()
3     local json_tree = createtree_json()
4     local menu_tree = merge_trees(lua_tree, json_tree)
5 
6     return menu_tree
7 end
```

我这里把无关菜单生成的部分去掉了。所以可以很清楚的看到生成菜单分为 2 部分。

- 第一部分是函数 `createtree()` 干的活，主要是运行 controller 下面的各个 lua 文件的 index 函数，生成菜单
- 第二部分是函数 `createtree_json()` 做的事，主要是分析 menu.d 目录下各个 json 文件，生成菜单
- 最后再通过函数 `merge_trees()` 把两者整合起来，形成最终的菜单
- 使用 lua 文件添加菜单

  知道大体流程了，我们先来看 `createtree` ，可以看到函数调用关系如下

  ```
  1 - menu_json
  2   - createtree
  3     - createindex
  ```

  createindex 函数的主要内容如下

```
    local _, path

    for path in (fs.glob("%s*.lua" % base) or function() end) do
        controllers[#controllers+1] = path
    end

    for path in (fs.glob("%s*/*.lua" % base) or function() end) do
        controllers[#controllers+1] = path
    end

    index = {}

    for _, path in ipairs(controllers) do
        local modname = "luci.controller." .. path:sub(#base+1, #path-4):gsub("/", ".")
        local mod = require(modname)
        assert(mod ~= true,
               "Invalid controller file found\n" ..
               "The file '" .. path .. "' contains an invalid module line.\n" ..
               "Please verify whether the module name is set to '" .. modname ..
               "' - It must correspond to the file path!")

        local idx = mod.index
        if type(idx) == "function" then
            index[modname] = idx
        end
    end
```

可以看到，该函数会遍历 `/usr/lib/lua/luci/controller/` 目录和其第一层子目录中所有 lua 文件，然后读取其中的 `index()` 函数，放在全局变量 index table 中。也就是说该函数执行完后，全局变量 index 中存放了所有的 controller 的 index 函数列表。

再来看看 `createtree()` 函数

```
function createtree()
    -- ...
    for k, v in pairs(index) do
        scope._NAME = k
        setfenv(v, scope)
        v()
    end

    return tree
end
```

这个函数很简单，我们只看最关键的部分，就是遍历 index table，执行其中保存的各个 `index()` 函数。

而事实上，添加菜单的工作都是由这个 `index()` 函数来完成的。

举个例子，例子来源于 luci-app-lxc，源码的文件 `luasrc/controller/lxc.lua` 内容如下：

```
    end

    page = node("admin", "services", "lxc")
    page.target = cbi("lxc")
    page.title = _("LXC Containers")
    page.order = 70
    page.acl_depends = { "luci-app-lxc" }

    page = entry({"admin", "services", "lxc_create"}, call("lxc_create"), nil)
    page.acl_depends = { "luci-app-lxc" }
    page.leaf = true

    page = entry({"admin", "services", "lxc_action"}, call("lxc_action"), nil)
    page.acl_depends = { "luci-app-lxc" }
    page.leaf = true

    page = entry({"admin", "services", "lxc_get_downloadable"}, call("lxc_get_downloadable"), nil)
    page.acl_depends = { "luci-app-lxc" }
    page.leaf = true

    page = entry({"admin", "services", "lxc_configuration_get"}, call("lxc_configuration_get"), nil)
    page.acl_depends = { "luci-app-lxc" }
    page.leaf = true

    page = entry({"admin", "services", "lxc_configuration_set"}, call("lxc_configuration_set"), nil)
    page.acl_depends = { "luci-app-lxc" }
    page.leaf = true
end
```

其中函数 node 来自于 dispatcher.lua

```
    c.module = getfenv(2)._NAME
    c.auto = nil

    return c
end

function _create_node(path)
    if #path == 0 then
        return context.tree
    end
    -- name = admin.srevices.lxc
    local name = table.concat(path, ".")
    local c = context.treecache[name]

    if not c then
        local last = table.remove(path)
        local parent = _create_node(path)

        c = {nodes={}, auto=true, inreq=true}

        parent.nodes[last] = c
        context.treecache[name] = c
    end

    return c
end
```

函数 cbi 和 entry 也来自于 dispatcher.lua

```
function cbi(model, config)
    return {
        type = "cbi",
        post = { ["cbi.submit"] = true },
        config = config,
        model = model
    }
end

function entry(path, target, title, order)
    local c = node(unpack(path))

    c.target = target
    c.title  = title
    c.order  = order
    c.module = getfenv(2)._NAME

    return c
end
```

通过以上分析可以知道，entry 函数调用就可以添加一个 uri。

比如

```
1 entry({"admin", "services", "lxc_create"}, call("lxc_create"), nil)
```

就会创建 uri admin/services/lxc_create，对应的操作为 call lxc_create

而代码

```
1     page = node("admin", "services", "lxc")
2     page.target = cbi("lxc")
3     page.title = _("LXC Containers")
4     page.order = 70
5     page.acl_depends = { "luci-app-lxc" }
```

则是创建一个 uri admin/services/lxc ，访问此页面，则会调用 model/cbi/lxc.lua 脚本，绘制页面。

- 使用 json 文件添加菜单

  这个就比较简单了，只需要在 menu.d 目录下添加对应的 json 文件即可。拿 `luci-app-ddns` 举例

```
1 {
2     "admin/services/ddns": {
3         "title": "Dynamic DNS",
4         "order": 59,
5         "action": {
6             "type": "view",
7             "path": "ddns/overview"
8         },
9         "depends": {
10            "acl": [ "luci-app-ddns" ]
11        }
12    }
13 }
```

显示菜单

那么页面的菜单是如何显示的呢？答案是通过 header。看 一下 themes 里面的 `header.js` 。

以 `/usr/lib/lua/luci/view/themes//bootstrap/header.htm` 为例，在 `<body>` 中有如下语句：

```
1 <header>
2 <ul class="nav" id="topmenu" style="display:none"></ul>
3 </header>
```

上面的语句只是给 topmenu （也就是最上方横向的菜单）占位，而在
`/usr/lib/lua/luci/view/themes/bootstrap/footer.htm` 中载入了 menu 的具体内容

```
1 <script type="text/javascript">L.require('menu-bootstrap')</script>
```

- renderMainMenu 把内容写入 topmenu（一层菜单）
- renderTabMenu 把内容写入 tabmenu （二层菜单）

```
"admin/system/admin": {
        "title": "Administration",
        "order": 2,
        "action": {
                "type": "firstchild"
        },
        "depends": {
                "acl": [ "luci-mod-system-config", "luci-mod-system-ssh" ]
        }
},

"admin/system/admin/password": {
        "title": "Router Password",
        "order": 1,
        "action": {
                "type": "view",
                "path": "system/password"
        },
        "depends": {
                "acl": [ "luci-mod-system-config" ]
        }
},
```

说一层菜单，二层菜单可能不太好理解，对应到 `/usr/share/luci/menu.d/luci-mod-system.json` 里面的内容就会好理解一些。比如上面的 admin 就是一层菜单，显示在最上面的菜单的 system 的子菜单中；而 admin/password 就是二层菜单显示在主界面的最上方的 tab 页。

```

   +---------------------------------------+
   |  +---------------------------------+  |
   |  |Status System Service Network ...  |  |
   |  +---------------------------------+  |
   |  +---------------------------------+  |
   |  |Router Password ... |             |  |
   |  +--------------------+             |  |
   |  |                                  |  |
   |  |                                  |  |
   |  |                                  |  |
   |  |                                  |  |
   |  |                                  |  |
   |  |                                  |  |
   |  +---------------------------------+  |
   +---------------------------------------+

```

了解了原理，那么如果需要自己定制化界面，比如添加侧边的 menu，可以用同样的方法，在 header.ntm 中添加

```
1 <div id="sidemenu" style="display:none"></div>
```

然后在对应的 menu-xxx.js 文件中加入 render 的代码即可。

action.type == firstchild

再来看 firstchild 。就以上面的 System->Administrator 为例。可以看到如下的调试信息

```
[08:58:51]: init_template_engine enter, media = /luci-static/bootstrap
[08:58:51]: Template.__init__ enter, name = themes/bootstrap/header
[08:58:51]: sourcefile =/usr/lib/lua/luci/view/themes/bootstrap/header.htm
[08:58:51]: Template.__init__ exit
[08:58:51]: init_template_engine after Template themes/bootstrap/header
[08:58:51]: init_template_engine exit
[08:58:51]: dispatch action =
{
    type = 'firstchild'
}
[08:58:51]: dispatch enter
[08:58:51]: init_template_engine enter, media = /luci-static/bootstrap
[08:58:51]: Template.__init__ enter, name = themes/bootstrap/header
[08:58:51]: Template.__init__ exit
[08:58:51]: init_template_engine after Template themes/bootstrap/header
[08:58:51]: init_template_engine exit
[08:58:51]: dispatch action =
{
    type = 'view'
    path = 'system/password'
}
[08:58:51]: Template.__init__ enter, name = view
[08:58:51]: sourcefile =/usr/lib/lua/luci/view/view.htm
[08:58:51]: Template.__init__ exit
[08:58:51]: Template.render enter, scope =
{
    view = 'system/password'
}
[08:58:51]: Template.__init__ enter, name = header
[08:58:51]: sourcefile =/usr/lib/lua/luci/view/header.htm
[08:58:51]: Template.__init__ exit
[08:58:51]: Template.render enter, scope =
{
}
[08:58:51]: Template.__init__ enter, name = themes/bootstrap/header
[08:58:51]: Template.__init__ exit
[08:58:51]: Template.render enter, scope =
{
}
[08:58:51]: Template.render exit
[08:58:51]: Template.render exit
[08:58:51]: Template.__init__ enter, name = footer
[08:58:51]: sourcefile =/usr/lib/lua/luci/view/footer.htm
[08:58:51]: Template.__init__ exit
[08:58:51]: Template.render enter, scope =
{
}
[08:58:51]: Template.__init__ enter, name = themes/bootstrap/footer
[08:58:51]: sourcefile =/usr/lib/lua/luci/view/themes/bootstrap/footer.htm
[08:58:51]: Template.__init__ exit
```

```
[08:58:51]: Template.render exit
[08:58:51]: Template.render exit
[08:58:51]: dispatcher.lua  httpdispatch exit
[08:58:51]: dispatcher.lua  httpdispatch enter, pathinfo = /admin/translations/en
[08:58:51]: dispatch enter
[08:58:51]: init_template_engine enter, media = /luci-static/bootstrap
[08:58:51]: Template.__init__ enter, name = themes/bootstrap/header
[08:58:51]: sourcefile =/usr/lib/lua/luci/view/themes/bootstrap/header.htm
[08:58:51]: Template.__init__ exit
[08:58:51]: init_template_engine after Template themes/bootstrap/header
[08:58:51]: init_template_engine exit
[08:58:51]: dispatch action =
{
    module = 'luci.controller.admin.index'
    type = 'call'
    function = 'action_translations'
}
[08:58:51]: dispatcher.lua  httpdispatch exit
```

结合代码

```
    elseif action.type == "firstchild" then
        local sub_request = find_subnode(page, requested_path_full, action.recurse)
        if sub_request then
            dispatch(sub_request)
        else
            tpl.render("empty_node_placeholder", getfenv(1))
        end
```

可以看到其实就是找到第一个子节点，然后进行显示。

页面设置

显示的部分清楚了，那么设置呢？如果我们在页面上修改了一些参数，点击 Save 或者 Save & Apply 按钮，又发生了些什么?

一句话概括就是：luci 通过 ubus 调用 rpcd 的 uci commit，然后 rpcd 发出 ubus 的 config.change event。

页面在修改配置后，会把修改的内容(uci change)存放在 `/var/run/rpcd/uci-<session id>` 目录下。这个其实是通过配置 uci 的 savedir 来实现的。

当点击按钮 Apply unchecked 或者 Save & Apply 的时候，浏览器会发 uri `admin/uci/apply_rollback` 或 `admin/uci/apply_unchecked` 的 request 给服务器

这里需要说明一下 `apply_rollback` 和 `apply_unchecked` 的区别。前者在应用新的配置的时候，如果出错，则会回滚（也就是退回到旧的配置）；而后者则不会。

根据 `/usr/share/luci/menu.d/luci-base.json` 的配置，会调用 `/usr/lib/lua/luci/controller/admin/uci.lua` 中的函数 action_apply_unchecked 或者函数 action_apply_rollback ，之后会调用

中暂存的 uci 修改文件调用 rpc_uci_apply_config，来对每一个 config 进行处理。最终执行了函数 rpc_uci_trigger_event，为每个提交的配置发出对应的 ubus 的 config.change 事件。

为帮助理解，我们打印出了修改 system 页面的 timezone 后，调用 apply 函数时当 rollback 为 false 的时候 `-` 和 `-.changes` 的内容

```
[10:46:13]: _=
{
    changes = {
        system = {
            1 = {
                1 = 'set'
                2 = 'cfg01e48a'
                3 = 'timezone'
                4 = 'CAT-2'
            }
            2 = {
                1 = 'set'
                2 = 'cfg01e48a'
                3 = 'zonename'
                4 = 'Africa/Blantyre'
            }
        }
    }
}
{
    system = {
        1 = {
            1 = 'set'
            2 = 'cfg01e48a'
            3 = 'timezone'
            4 = 'CAT-2'
        }
        2 = {
            1 = 'set'
            2 = 'cfg01e48a'
            3 = 'zonename'
            4 = 'Africa/Blantyre'
        }
    }
}
```

以 firewall 为例。在 `/etc/init.d/firewall` 脚本中，在系统启动时，通过 `procd_add_reload_trigger firewall` 由 procd 向 ubus 注册 config.change event。一旦 ubus 收到 config.change 事件，将触发 `/etc/init.d/firewall reload`

```
    local file

    _procd_open_trigger
    for file in "$@"; do
        _procd_add_config_trigger "config.change" "$file" /etc/init.d/$name reload
    done
    _procd_close_trigger
}
```

LUCI 页面编写

E() 是什么?

我们经常看到 js 文件里有用 `E()` 函数包起来的内容。那么这个 E 到底是什么呢？查看代码可以发现 E 的定义在 `cbi.js` 中

```
function E(){return L.dom.create.apply(L.dom,arguments)}
```

所以这个 `E()` 其实只是 LuCI.dom.create() 的一个别名。

Templates

<% code %>	包含 lua 代码
<% write(value) %>	调用 lua 函数
<%=value%>	输出 lua 变量
<% include(templateName) %>	加载模板
<%+templateName%>	加载模板
<%= translate(“Text to translate”) %>	转换文本语言
<%:Text to translate%>	同上
<%# comment %>	注释

menu.d

depends

depends 用来配置这个节点对应的依赖条件，如果条件不满足，则节点不显示。（注意如果 uci 值改变并不会马上刷新 menu，必须要重新 login）

下面是一个例子，upnp 页面只有在 uci 的 option `wireless.mesh.role` 的值为 agent 的时候才显示。

```
    "action": {
        "type": "view",
        "path": "sercomm-admin/upnp"
    },
    "depends": {
        "acl": [ "luci-app-upnp" ],
        "uci": { "wireless": {"mesh": {"role": "agent"} } }
    }
}
```

对应的代码可以参考 check_uci_depends@dispatcher.lua

注意到 depends 后面的描述可以是一个 object (用 `{ }` 表示)，也可以是一个 array （用 `[ ]` 表示）。当为 object 的时候，里面的每一个选项之间是 and 的关系；当为 array 的时候，则为 or 的关系。

FAQ

什么是 CBI ?

CBI 是 Configuration Bind Interface 的缩写。在 老的源码 中有以下说明

```
Description:

Offers an interface for binding configuration values to certain data
types. Supports value and range validation and basic dependencies.
```

简单的理解，CBI 就是一个库，提供了一系列的接口用来在 UCI 的配置文件和相关的语言（lua 或者 js）的对象之间进行转换。

为什么 LUCI 提供了 2 套 API?

翻看 luci 的官方说明文档，会发现提供了 2 套 api。一套是 lua 的，一套是 js 的。

根据资料，最初 luci 的设计是通过 lua 脚本来生成界面元素（后端），后来考虑到效率问题，把页面的生成转移到到了客户端浏览器（前端），所以改成了 js 的方式。

有哪些相关的目录?

从 openwrt 开发板的角度来看，有以下的相关目录或文件

- /usr/share/luci/menu.d 存放菜单配置
    - /usr/share/luci/menu.d/luci-base.json
        - [admin/status, type: firstchild]
            - /usr/share/luci/menu.d/luci-mod-status.json
                - [admin/status/overview, type: template, path: admin_status/index]
                    - /usr/lib/lua/luci/view/admin_status/index.htm

- /www
  - luci-static/resources/view 存放页面对应的 js 文件
- /usr/lib/lua/luci
  - /usr/lib/lua/luci/model 动态获取数据的 lua 脚本
  - /usr/lib/lua/luci/view htm 格式的静态网页文件

LUCI 的 theme 是怎么应用的?

相关的目录有

- /usr/lib/lua/luci/view/themes/<themes-name> 存放 header.htm 和 footer.htm
- /www/luci-static/<themes-name> 存放相关的 css, js, image 等
- /etc/uci-defaults
- /etc/config/luci

举例，你新创建了一个主题，名字叫 vodacom，那么通过以下指令可以切过去

```
uci set luci.main.mediaurlbase=/luci-static/vodacom
uci commit luci
```

要恢复成老的界面，只需要

```
uci set luci.main.mediaurlbase=/luci-static/bootstrap
uci commit luci
```

修改后如何更新?

删除 cache

```
rm -rf /tmp/luci-*
```

然后通过以下命令禁止 cache，这样以后就不用删除 cache 了

```
uci set luci.ccache.enable=0; uci commit luci
```

如何实时的显示或隐藏菜单项?

这个问题稍微有点复杂，需要先分析一下菜单是如何显示的

首先，在客户端，会通过 `menu.js` 去获取菜单，大致的调用流程如下

```
- www/luci-static/resources/menu.js
  - ui.menu
    - UIMenu.load@ui.js
      - get url admin/menu
```

可以看到，最终是发送对 uri `admin/menu` 的请求，向服务器获取 menu 的内容

```
- action_menu
  - action_menu@usr/lib/lua/luci/controller/admin/index.lua
    - menu_json@/usr/lib/lua/dispatcher.lua
```

最终执行 `dispatcher.lua` 中的函数 `menu_json` 获得相应的内容，返回给客户端。

这里需要注意，luci 会把 menu 的内容存放在 cache （ `/tmp/luci-*` ）中。所以为了获取新的 menu，要将 cache 清空才行。

知道了原理，这个问题就有了思路。下面举例说明

假设 package mesh 包含了以下文件

- `/etc/config/mesh`
- `/www/luci-static/resources/view/system/mesh.js`
- `/usr/share/rpcd/acl.d/luci-base.json` （添加部分 acl 控制）

配置文件 `/etc/config/mesh` 内容如下：

```
config mesh 'mesh'
        option enabled '1'
```

其中 option enabled 控制了菜单 System->Startup 的显示与否。所以文件 `/usr/share/luci/menu.d/luci-mod-system.json` 中关于 Startup 页面的配置如下：

```
    "admin/system/startup": {
        "title": "Startup",
        "order": 45,
        "action": {
            "type": "view",
            "path": "system/startup"
        },
        "depends": {
            "uci": { "mesh": {"mesh": {"enabled": "1"} } }
        }
    },
```

depends 就说明了页面的显示依赖于 `mesh.mesh.enabled` 的取值，当其为 1 的时候才显示。

为了达到目的， `mesh.js` 应该按照如下方式编写：

```
'require fs';
'require uci';
'require rpc';
'require form';

return view.extend({
  render: function() {
        var m, s, o;
        m = new form.Map('mesh',('Easy Mesh'));
        s = m.section(form.NamedSection, 'mesh', 'mesh');
        o = s.option(form.Flag, 'enabled', _('Enable'));

        return m.render();
  },
  handleSaveApply: function(ev, mode) {
    return this.super('handleSaveApply', [ev, mode]).then(function() {
      console.log("handleSaveApply is called");
      fs.exec('/usr/bin/cl_luci_cache.sh').then(function() {
        if (L.ui.menu && L.ui.menu.flushCache)
            L.ui.menu.flushCache();
        console.log("ui.flushCache is called");
        })
    });
  }
});
```

其中 `/usr/bin/cl_luci_cache.sh` 内容如下：

```
#!/bin/sh
/bin/rm -f /tmp/luci-indexcache* > /dev/null 2>&1
```

可以看到，在 apply 的时候，先调用脚本 `/usr/bin/cl_luci_cache.sh` 清除服务器端的 cache，然后再调用 `L.ui.menu.flushCache` 清楚客户端的 cache，重新跟服务器端索取 menu 的内容。

当然为了能执行脚本 `/usr/bin/cl_luci_cache.sh` ，还需要在 `luci-base.json` 中添加 acl 规则，内容如下：

```
    "luci-app-mesh": {
        "description": "Grant access to mesh procedures",
        "read": {
            "uci": [ "mesh" ]
        },
        "write": {
            "file": {
                "/usr/bin/cl_luci_cache.sh": [ "exec" ]
            },
            "uci": [ "mesh" ]
        }
    }
```

这样，就可以通过 enable/disable 页面上的 mesh enable 来显示或隐藏 Startup 页面了。

一般的 debug 信息输出可以使用

```
1  luci.util.perror("blah blah")
```

然后用 logread 就能看到输出。

为了打印更多的信息，我们可以在目录 `/usr/lib/lua/luci` 中添加文件 `log.lua` ，内容如下

```lua
local M = {}
local tinsert = table.insert
local srep = string.rep

local function local_print(str)
    local dbg = io.open("/tmp/luci.output", "a+")
    local str = str or ""
    if dbg then
        dbg:write(str..'\n')
        dbg:close()
    end
end

function M.print(...)
    local dbg = io.open("/tmp/luci.output", "a+")
    if dbg then
        dbg:write(os.date("[%H:%M:%S]: "))
        for _, o in ipairs({...}) do
            dbg:write(tostring(o)..'  ')
        end
        dbg:write("\n")
        dbg:close()
    end
end

function M.print_r(data, depth)
    local depth = depth or 3
    local cstring = "";
    local top_flag = true

    local function table_len(t)
    local i = 0
    for k, v in pairs(t) do
        i = i + 1
    end
    return i
    end

    local function tableprint(data,cstring, local_depth)
        if data == nil then
            local_print("core.print data is nil");
        end

        local cs = cstring .. "    ";
        if top_flag then
            local_print(cstring .."{");
            top_flag = false
        end
        if(type(data)=="table") then
            for k, v in pairs(data) do
        if type(v) ~= "table" then
```

```
            if type(v) == "string" then
                local_print(cs..tostring(k).." = "..v)
            end
        elseif table_len(v) == 0 then
            local_print(cs..tostring(k).." = "..{})
        elseif local_depth < depth then
                    local_print(cs..tostring(k).." = {");
                      tableprint(v,cs,local_depth+1);
        else
            local_print(cs..tostring(k).." = "..{*})
        end
            end
        else
            local_print(cs..tostring(data));
        end
        local_print(cstring ..});
    end

    tableprint(data,cstring,0);
end

return M
```

然后用类似以下的方式使用。比如在 dispatcher.lua 的开头添加

```
local log = require "log.lua"
```

在函数 `dispatch` 中打印

```
log.print("action.type =".."action.type)
log.print_r(action)
```

这样在刷新页面时，用以下命令就可以看到调试输出了

```
tail -f /tmp/luci.output
```

注意如果看不到输出，那可能是你的 cache 没有禁用或清除。

如果调试的时候要获取当前文件名，可以用以下函数

```
function __FILE__()
    return debug.getinfo(2, 'S').source:match("^.+/(.+)$")
end
```

参考

- luci_tutorials
- Description of the JSON in menu.d and acl.d
- https://github.com/openwrt/luci/tree/master/docs
- https://openwrt.org/docs/guide-developer/luci
- http://openwrt.github.io/luci/jsapi/index.html

- LuCI入门 LuCI之CBI LuCI之UCI
- OpenWrt达人教程之开发人员入门指南
- luci example

更新于 2023-06-09

返回 | 主页

内核空间和用户空间通信 - proc 虚拟文件系统 **IPv6 FAQ**